Lab - Construir una aplicación web con Spring Boot y Angular

1. Overview

Spring Boot y Angular forman un potente tándem que funciona de maravilla para desarrollar aplicaciones web con una huella mínima.
En este laboratorio, **utilizaremos Spring Boot para implementar un backend RESTful, y Angular para crear un frontend basado en JavaScript.**

2. The Spring Boot Application

La funcionalidad de nuestra aplicación web de demostración será bastante simplista.
Se limitará a obtener y mostrar una lista de entidades JPA desde una base de datos H2 en memoria, y a persistir las nuevas a través de un simple formulario HTML.

2.1. The Maven Dependencies

Aquí están las dependencias de nuestro proyecto Spring Boot:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

Ten en cuenta que hemos incluido spring-boot-starter-web porque lo usaremos para crear el servicio REST, y spring-boot-starter-jpa para implementar la capa de persistencia.
La versión de la base de datos H2 también es gestionada por el padre de Spring Boot.

2.2. The JPA Entity Class

Para crear rápidamente un prototipo de la capa de dominio de nuestra aplicación, vamos a definir una simple clase de entidad JPA, que se encargará de modelar los usuarios:

```
@Entity
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long id;
    private final String name;
```

```
    private final String email;

    // standard constructors / setters / getters / toString
}
```

2.3. The *UserRepository* Interface

Dado que necesitaremos una funcionalidad CRUD básica en las entidades de Usuario, también debemos definir una interfaz UserRepository:

```
@Repository
public interface UserRepository extends CrudRepository<User,
Long>{}
```

2.4. The REST Controller

Ahora vamos a implementar la API REST. En este caso, se trata de un simple controlador REST:

```
@RestController
@CrossOrigin(origins = "http://localhost:4200")
public class UserController {

    // standard constructors

    private final UserRepository userRepository;

    @GetMapping("/users")
    public List<User> getUsers() {
        return (List<User>) userRepository.findAll();
    }

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

No hay nada inherentemente complejo en la definición de la clase UserController.
Por supuesto, el detalle de implementación que merece la pena destacar aquí es el uso de la anotación @CrossOrigin. Como su nombre indica, la anotación habilita el uso compartido de recursos entre orígenes (CORS) en el servidor.
Este paso no siempre es necesario, pero dado que estamos desplegando nuestro frontend de Angular en http://localhost:4200, y nuestro backend de Boot en http://localhost:8080, el navegador denegaría las peticiones de uno a otro.
En cuanto a los métodos del controlador, getUser() obtiene todas las entidades de usuario de la base de datos. Del mismo modo, el método addUser() persigue una nueva entidad en la base de datos, que se pasa en el cuerpo de la petición.

Para mantener las cosas simples, hemos omitido deliberadamente la implementación del controlador que activa la validación de Spring Boot antes de persistir una entidad. En producción, sin embargo, no podemos confiar únicamente en la entrada del usuario, por lo que la validación del lado del servidor debería ser una característica obligatoria.

2.5. Bootstrapping the Spring Boot Application

Por último, vamos a crear una clase de arranque estándar de Spring Boot, y a rellenar la base de datos con algunas entidades de usuario:

```
@SpringBootApplication
public class Application {

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

    @Bean
    CommandLineRunner init(UserRepository userRepository) {
        return args -> {
            Stream.of("John",   "Julie",   "Jennifer",   "Helen",
"Rachel").forEach(name -> {
                User user = new User(name, name.toLowerCase() +
"@domain.com");
                userRepository.save(user);
            });

userRepository.findAll().forEach(System.out::println);
        };
    }
}
```

Ahora vamos a ejecutar la aplicación. Como era de esperar, deberíamos ver una lista de entidades de usuario impresa en la consola al iniciarse:

```
User{id=1, name=John, email=john@domain.com}
User{id=2, name=Julie, email=julie@domain.com}
User{id=3, name=Jennifer, email=jennifer@domain.com}
User{id=4, name=Helen, email=helen@domain.com}
User{id=5, name=Rachel, email=rachel@domain.com}
```

3. The Angular Application

Con nuestra aplicación de demostración de Spring Boot en funcionamiento, ahora podemos crear una sencilla aplicación Angular capaz de consumir la API del controlador REST.

3.1. Angular CLI Installation

Utilizaremos Angular CLI, una potente utilidad de línea de comandos, para crear nuestra aplicación Angular.

Angular CLI es una herramienta muy valiosa ya que nos permite crear todo un proyecto Angular desde cero, generando componentes, servicios, clases e interfaces con sólo unos pocos comandos.
Una vez que hemos instalado npm (Node Package Manager), abriremos una consola de comandos y escribiremos el comando

```
npm install -g @angular/cli@1.7.4
```

Eso es todo. El comando anterior instalará la última versión de Angular CLI.

3.2. Project Scaffolding With Angular CLI
Podemos generar la estructura de nuestra aplicación Angular desde el principio, pero honestamente, esta es una tarea propensa a errores y que consume mucho tiempo que deberíamos evitar en todos los casos.
En su lugar, dejaremos que Angular CLI haga el trabajo duro por nosotros. Así que podemos abrir una consola de comandos, luego navegar a la carpeta donde queremos que se cree nuestra aplicación, y escribir el comando

```
ng new angularclient
```

El comando new generará toda la estructura de la aplicación dentro del directorio angularclient.

3.3. The Angular Application's Entry Point
Si miramos dentro de la carpeta angularclient, veremos que Angular CLI ha creado efectivamente un proyecto completo para nosotros.
Los archivos de aplicación de Angular utilizan TypeScript, un superconjunto tipado de JavaScript que compila a JavaScript plano. Sin embargo, el punto de entrada de cualquier aplicación de Angular es un viejo archivo index.html.
Vamos a editar este archivo:

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Spring Boot - Angular Application</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-
scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
  <link rel="stylesheet"

href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootst
rap.min.css"
    integrity="sha384-
Gn5384xqQ1aoWXA+058RXPxPg6fy4IWvTNh0E263XmFcJlSAwiGgFAW/dAiS6JXm
"
    crossorigin="anonymous">
```

```
</head>
<body>
  <app-root></app-root>
</body>
</html>
```

Como podemos ver arriba, hemos incluido Bootstrap 4 para poder dar a los componentes de la interfaz de usuario de nuestra aplicación un aspecto más elegante. Por supuesto, es posible elegir otro kit de interfaz de usuario del montón disponible por ahí.
Fíjate en las etiquetas personalizadas <app-root></app-root> dentro de la sección <body>. A primera vista, parecen bastante extrañas, ya que <app-root> no es un elemento estándar de HTML 5.
Las mantendremos ahí, ya que **<app-root> es el selector raíz que Angular utiliza para renderizar el componente raíz de la aplicación**.

3.4. The *app.component.ts* Root Component

Para entender mejor cómo Angular vincula una plantilla HTML a un componente, vamos a ir al directorio src/app y editar el archivo TypeScript app.component.ts, el componente raíz:

```
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';
  }
}
```

Por razones obvias, no nos sumergiremos en el aprendizaje de TypeScript. Aun así, observemos que el archivo define una clase AppComponent, que declara un campo título de tipo string (en minúsculas). En definitiva, se trata de JavaScript tipado.
Además, el constructor inicializa el campo con un valor de cadena, lo que es bastante similar a lo que hacemos en Java.
La parte más relevante es el marcador de metadatos o decorador @Component, que define tres elementos
1. selector - el selector HTML utilizado para vincular el componente al archivo de plantilla HTML
2. templateUrl - el archivo de plantilla HTML asociado al componente
3. styleUrls - uno o más archivos CSS asociados al componente

Como era de esperar, podemos utilizar los archivos app.component.html y app.component.css para definir la plantilla HTML y los estilos CSS del componente raíz. Por último, el elemento selector vincula todo el componente al selector <app-root> incluido en el archivo index.html.

3.5. The *app.component.html* File

Como el archivo app.component.html nos permite definir la plantilla HTML del componente raíz, la clase AppComponent, la utilizaremos para crear una barra de navegación básica con dos botones.
Si hacemos clic en el primer botón, Angular mostrará una tabla que contiene la lista de entidades de usuario almacenadas en la base de datos. Del mismo modo, si hacemos clic en el segundo, se mostrará un formulario HTML, que podemos utilizar para añadir nuevas entidades a la base de datos:

```
<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>
```

La mayor parte del archivo es HTML estándar, con algunas advertencias que vale la pena señalar.
La primera es la expresión {{ title }}. Las dobles llaves {{ nombre-variable }} es el marcador de posición que Angular utiliza para realizar la interpolación de variables.
Tengamos en cuenta que la clase AppComponent inicializó el campo title con el valor Spring Boot - Angular Application. Así, Angular mostrará el valor de este campo en la plantilla. Del mismo modo, el cambio del valor en el constructor se reflejará en la plantilla.
La segunda cosa a tener en cuenta es el atributo routerLink. Angular utiliza este atributo para el enrutamiento de las solicitudes a través de su módulo de

enrutamiento (más sobre esto más adelante). Por ahora, es suficiente saber que el módulo enviará una petición a la ruta /users a un componente específico y una petición a /adduser a otro componente.
En cada caso, la plantilla HTML asociada con el componente correspondiente se mostrará dentro del marcador de posición <router-outlet></router-outlet>.

3.6. The *User* Class

Dado que nuestra aplicación Angular obtendrá y persistirá entidades de usuario en la base de datos, vamos a implementar un modelo de dominio simple con TypeScript.
Abramos una consola de terminal y creemos un directorio de modelo:

```
ng generate class user
```

El CLI de Angular generará una clase User vacía, así que vamos a rellenarla con unos cuantos campos:

```
export class User {
    id?: string;
    name?: string;
    email?: string;
}
```

3.7. The *UserService* Service

Con nuestra clase de usuario de dominio del lado del cliente ya establecida, ahora podemos implementar una clase de servicio que realice peticiones GET y POST al punto final de http://localhost:8080/users.
Esto nos permitirá encapsular el acceso al controlador REST en una sola clase, que podemos reutilizar en toda la aplicación.
Abramos un terminal de consola, creemos un directorio de servicio, y dentro de ese directorio, emitamos el siguiente comando:

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

Ahora vamos a abrir el archivo user.service.ts que Angular CLI acaba de crear y a refactorizarlo::

```
import { Injectable } from '@angular/core';
import { HttpClient, HttpHeaders } from '@angular/common/http';
import { User } from '../model/User';
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class UserService {
  private usersUrl: string;

  constructor(private http: HttpClient) {
```

```
    this.usersUrl = 'http://localhost:8080/users';
   }

   public findAll(): Observable<User[]> {
    return this.http.get<User[]>(this.usersUrl);
  }

  public save(user: User) {
    return this.http.post<User>(this.usersUrl, user);
  }

}
```

No necesitamos una sólida formación en TypeScript para entender cómo funciona la clase UserService. En pocas palabras, encapsula dentro de un componente reutilizable toda la funcionalidad necesaria para consumir la API del controlador REST que implementamos antes en Spring Boot.
El método findAll() realiza una petición GET HTTP al endpoint http://localhost:8080/users a través del HttpClient de Angular. El método devuelve una instancia Observable que contiene un array de objetos User.
Asimismo, el método save() realiza una petición HTTP POST al endpoint http://localhost:8080/users.
Al especificar el tipo User en los métodos de petición del HttpClient, podemos consumir las respuestas del back-end de una manera más fácil y efectiva.
Por último, **observemos el uso del marcador de metadatos @Injectable(). Esto indica que el servicio debe ser creado e inyectado a través de los inyectores de dependencia de Angular.**

3.8. The *UserListComponent* Component

In this case, the *UserService* class is the thin middle-tier between the REST service and the application's presentation layer. Therefore, we need to define a component responsible for rendering the list of *User* entities persisted in the database.
Let's open a terminal console, then create a *user-list* directory, and generate a user list component:

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

Angular CLI will generate an empty component class that implements the *ngOnInit* interface. The interface declares a hook *ngOnInit()* method, which Angular calls after it has finished instantiating the implementing class, and also after calling its constructor.
Let's refactor the class so that it can take a *UserService* instance in the constructor:

```
import { Component, OnInit } from '@angular/core';
import { User } from 'src/app/model/User';
import { UserService } from 'src/app/service/user.service';

@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: UserService) { }

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

  }

}
```

La implementación de la clase UserListComponent se explica por sí misma. Simplemente utiliza el método findAll() del UserService para obtener todas las entidades persistentes en la base de datos y las almacena en el campo users.
Además, necesitamos editar el archivo HTML del componente, user-list.component.html, para crear la tabla que muestra la lista de entidades:

```
<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>
```

Cabe destacar el uso de la directiva *ngFor. La directiva se llama repetidor, y podemos utilizarla para iterar sobre el contenido de una variable y renderizar iterativamente los elementos HTML. En este caso, la utilizamos para renderizar dinámicamente las filas de la tabla.

Además, utilizamos la interpolación de variables para mostrar el id, el nombre y el correo electrónico de cada usuario.

3.9. The *UserFormComponent* Component

Del mismo modo, necesitamos crear un componente que nos permita persistir un nuevo objeto Usuario en la base de datos.
Creemos un directorio user-form y escribamos lo siguiente:

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

A continuación abrimos el archivo user-form.component.ts, y añadimos a la clase UserFormComponent un método para guardar un objeto User:

```
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute, Router } from '@angular/router';
import { User } from 'src/app/model/User';
import { UserService } from 'src/app/service/user.service';

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

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

  ngOnInit(): void {
  }
  onSubmit() {
    this.userService.save(this.user).subscribe(result => this.go
toUserList());
  }

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

}
```

En este caso, UserFormComponent también toma una instancia de UserService en el constructor, que el método onSubmit() utiliza para guardar el objeto User suministrado.

Como necesitamos volver a mostrar la lista actualizada de entidades una vez que hemos persistido una nueva, llamamos al método gotoUserList() después de la inserción, que redirige al usuario a la ruta /users.
Además, necesitamos editar el archivo user-form.component.html, y crear el formulario HTML para persistir un nuevo usuario en la base de datos:

```
<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>
```

A simple vista, el formulario parece bastante estándar, pero encapsula gran parte de la funcionalidad de Angular entre bastidores.
Observemos el uso de la directiva ngSubmit, que llama al método onSubmit() cuando se envía el formulario.
A continuación, definimos la variable de plantilla #userForm, por lo que Angular añade automáticamente una directiva NgForm, que nos permite seguir el formulario en su conjunto.
La directiva NgForm contiene los controles que hemos creado para los elementos del formulario con una directiva ngModel y un atributo name. También controla sus propiedades, incluyendo su estado.

La directiva ngModel nos proporciona una funcionalidad de enlace de datos bidireccional entre los controles del formulario y el modelo de dominio del lado del cliente, la clase User.
Esto significa que los datos introducidos en los campos de entrada del formulario fluirán hacia el modelo, y viceversa. Los cambios en ambos elementos se reflejarán inmediatamente a través de la manipulación del DOM.
Además, ngModel nos permite hacer un seguimiento del estado de cada control del formulario, y realizar la validación del lado del cliente añadiendo diferentes clases CSS y propiedades DOM a cada control.
En el archivo HTML anterior, utilizamos las propiedades aplicadas a los controles del formulario sólo para mostrar un cuadro de alerta cuando los valores del formulario han sido cambiados.

3.10. The *app-routing.module.ts* File

Aunque los componentes son funcionales de forma aislada, todavía necesitamos utilizar un mecanismo para llamarlos cuando el usuario hace clic en los botones de la barra de navegación.
Aquí es donde entra en juego el RouterModule. Abramos el archivo app-routing.module.ts y configuremos el módulo para que pueda enviar peticiones a los componentes correspondientes:

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

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

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

Como podemos ver arriba, la matriz de Rutas indica al enrutador qué componente debe mostrar cuando un usuario hace clic en un enlace o especifica una URL en la barra de direcciones del navegador.
Una ruta se compone de dos partes:
1. *Ruta* - una cadena que coincide con la URL en la barra de direcciones del navegador
2. *Componente* - el componente a crear cuando la ruta está activa (navegada)

Si el usuario hace clic en el botón List Users, que enlaza con la ruta /users, o introduce la URL en la barra de direcciones del navegador, el enrutador mostrará el archivo de

plantilla del componente UserListComponent en el marcador de posición <router-outlet>.
Del mismo modo, si hacen clic en el botón Add User, se mostrará el componente UserFormComponent.

3.11. The *app.module.ts* File

A continuación tenemos que editar el archivo app.module.ts, para que Angular pueda importar todos los módulos, componentes y servicios necesarios.
Además, tenemos que especificar qué proveedor usaremos para crear e inyectar la clase UserService. De lo contrario, Angular no podrá inyectarla en las clases de los componentes:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { UserListComponent } from './component/user-list/user-list.component';
import { UserFormComponent } from './component/user-form/user-form.component';
import { FormsModule } from '@angular/forms';
import { HttpClientModule } from '@angular/common/http';
import { UserService } from './service/user.service';

@NgModule({
  declarations: [
    AppComponent,
    UserListComponent,
    UserFormComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    HttpClientModule,
    FormsModule
  ],
  providers: [UserService],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

4. Running the Application

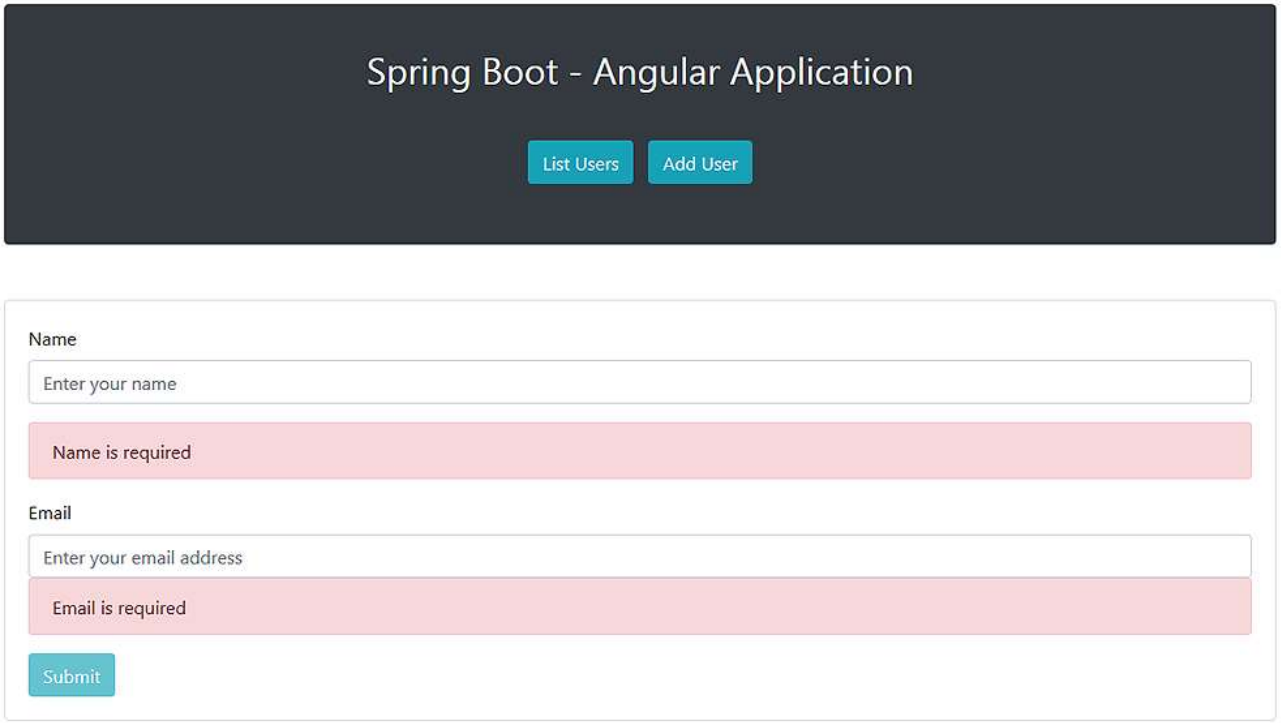
Finalmente, estamos listos para ejecutar nuestra aplicación.
Para ello, primero ejecutaremos la aplicación Spring Boot, para que el servicio REST esté vivo y a la escucha de peticiones.
Una vez iniciada la aplicación Spring Boot, abriremos una consola de comandos y escribiremos el siguiente comando

```
ng serve –open
```

Esto iniciará el servidor de desarrollo en vivo de Angular y también abrirá el navegador en http://localhost:4200.
Deberíamos ver la barra de navegación con los botones para listar las entidades existentes y para añadir otras nuevas. Si hacemos clic en el primer botón, deberíamos ver debajo de la barra de navegación una tabla con la lista de entidades persistidas en la base de datos:

#	Name	Email
1	John	john@domain.com
2	Julie	julie@domain.com
3	Jennifer	jennifer@domain.com
4	Helen	helen@domain.com
5	Rachel	rachel@domain.com

Similarly, clicking the second button will display the HTML form for persisting a new entity:


Spring Boot - Angular Application
List Users
Add User
Name
Enter your name
Name is required
Email
Enter your email address
Email is required
Submit