COMPONENT PROJECT

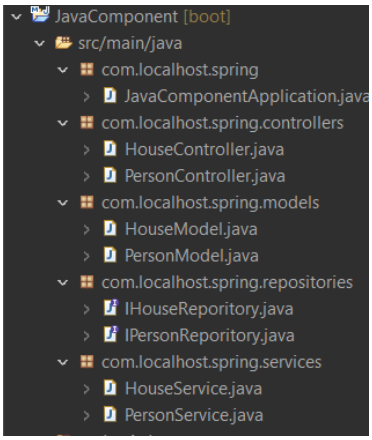
Manuel González | Carlos Murillo

Contenido

1.Spring Boot Programa

1.1 Funcionamiento del Programa

El proyecto del servidor Spring Tool cuenta con la siguiente estructura de paquetes:



Paquete **Controllers**: todos los puntos finales de comunicación que tendrá nuestra aplicación.

Paquete **Models**: representación de nuestro modelo de datos.

Paquete **Repositories**: para las clases que establezcan comunicación con la base de datos.

Paquete **Services**: clases que corresponden a la funcionalidad y la lógica.

1.1.1 Proyecto e Independencias

Deberemos de añadir las dependencias necesarias para que funcione correctamente la aplicación. Se añaden de forma automática al crear el proyecto seleccionando las dependencias que deseemos.

```xml
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <dependency>
        <groupId>com.mysql</groupId>
        <artifactId>mysql-connector-j</artifactId>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
```

1.1.2 Clases y Métodos

En la clase **HouseModel** definiremos las propiedades y los campos que tendrá la tabla house en la base de datos.

```java
package com.localhost.spring.models;

import jakarta.persistence.Column;

@Entity
@Table(name ="house")
public class HouseModel {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long idHouse;

    @Column
    private String direccion;

    @Column
    private String cp;
```

En la clase **PersonModel** añadiremos también los campos que veamos oportunos para posterior creación de la tabla en la base de datos. Para establecer la relación de que una persona puede tener varias casas añadiremos un HouseModel como propiedad de la clase además de incluir la etiqueta @ManyToOne y el @JoinColumn.

```java
package com.localhost.spring.models;

import jakarta.persistence.Column;

@Entity
@Table(name ="person")
public class PersonModel {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long idPerson;

    @Column
    private String nombre;

    @Column
    private String apellidos;

    @ManyToOne // entidad pertenece a una casa
    @JoinColumn(name = "id_house") // almacena la clave foránea en la tabla person
    private HouseModel house;
```

En la clase **PersonService** definiremos los métodos que a posteriori usaremos en la clase Person Controller

```java
package com.localhost.spring.services;

import java.util.ArrayList;

@Service
public class PersonService {
    @Autowired
    private IPersonReporitory personRepository; // Dependencia inyectada

    // Método que devuelve todas las personas
    public ArrayList<PersonModel> getPersons() {
        return (ArrayList<PersonModel>) personRepository.findAll();
    }

    // Método para guardar una nueva persona
    public PersonModel savePerson(PersonModel person) {
        return personRepository.save(person);
    }
```

En la clase **Person Controller** definiremos los métodos que usaremos en el futuro a la hora de usar la aplicación: como, obtener todas las personas, crear una nueva persona, actualizar una persona por su id, y eliminar una persona por id.

La etiqueta @RequestMapping es muy importante ya que dependiendo qué pongamos tendremos que poner lo mismo en la url a la hora de conectarnos mediante http.

```java
package com.localhost.spring.controllers;

import java.util.ArrayList;

@RestController
@RequestMapping("/person")
public class PersonController {

    @Autowired
    private PersonService personService;

    // Obtiene todas las personas
    @GetMapping
    public ArrayList<PersonModel> getPersons() {
        return personService.getPersons();
    }

    // Crea una nueva persona
    @PostMapping
    public PersonModel savePerson(@RequestBody PersonModel person) {
        return personService.savePerson(person);
    }
```

En la interfaz **IPersonRepository** definiremos la cabecera para usarla en la clase HouseService.

```
package com.localhost.spring.repositories;

import org.springframework.data.jpa.repository.JpaRepository;

public interface IPersonReporitory extends JpaRepository<PersonModel, Long> {

}
```

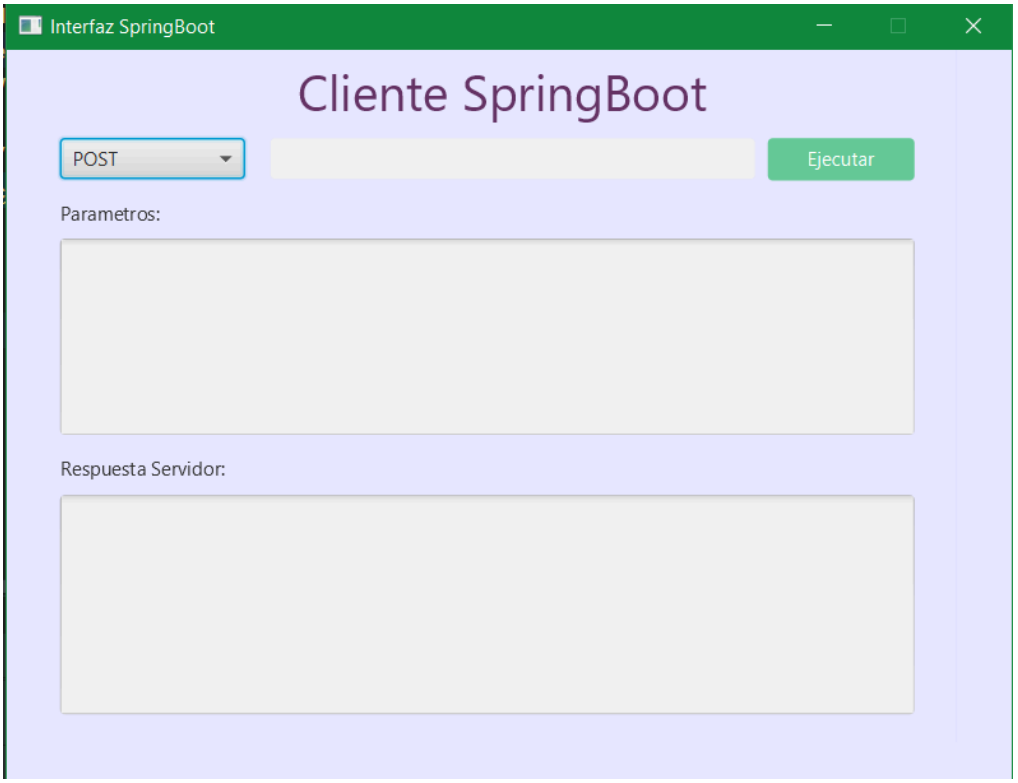
De igual forma sería para las clases de House como hemos visto con las clases de Person.

Para configurar la conexión a la base de datos tendremos que modificar el archivo **application.properties** añadiendo la cadena de conexión a la base de datos, el puerto del programa Spring Boot, el usuario para conectarnos a la base de datos y su respectiva contraseña si la tuviera.

```
server.port=8088
spring.datasource.url=jdbc:mysql://localhost:3306/db_users?useSSL=false&serverTimeZone=Europa/Madrid
spring.datasource.username=root
spring.jpa.hibernate.ddl-auto=update
```

2.Interfaz Gráfica

2.1 Diseño de la Interfaz



La interfaz ha sido diseñada en JavaFX (Java) basada en el diseño del Programa Postman , cuenta con una interfaz sencilla y útil, y una gama de colores de lo mas vistosa siendo agradable a la vista.

2.2 Funcionamiento del Programa

El programa cuenta con tres clases que se encargarán del funcionamiento:

-HelloAplication: Clase principal llamada al ejecutar el programa por primera vez

-HelloController: Clase que se encarga de los eventos del programa y del código en sí

-Operador: Clase que contiene los métodos para realizar las operaciones necesarias en el código.

2.2.1 Proyecto e Independencias

El Proyecto es un proyecto Maven, creado en el IDE IntelliJ.

El proyecto Maven tiene dos dependencias añadidas:

1.SpringFramework

```
<groupId>org.springframework</groupId>

<artifactId>spring-web</artifactId>

<version>5.3.14</version>
```

2.Apache HTTP Components

```
<groupId>org.apache.httpcomponents</groupId>

<artifactId>httpclient</artifactId>

<version>4.5.13</version>
```

2.2.2 Clases y Métodos

1.Hello Application

Clase que contiene un FXMLLoader que se encarga de lanzar la ventana principal , creamos una escena a la que se le añadirá un tamaño y unas propiedades determinadas, después de esto haremos un show() para mostrarla.

He de decir que la clase main(String[]args) contiene el launch() que lo que hace es llamar al método start() justo arriba

```
Carlos +1 *
public class HelloApplication extends Application {
    Carlos +1
    @Override
    public void start(Stage stage) throws IOException {
        FXMLLoader fxmlLoader = new FXMLLoader(HelloApplication.class.getResource( name: "hello-view.fxml"));
        Scene scene = new Scene(fxmlLoader.load(),  v: 630,  v1: 460);
        stage.setTitle("Interfaz SpringBoot");
        stage.setResizable(false);
        stage.setScene(scene);
        stage.show();
    }
    Carlos
    public static void main(String[] args) {
        launch();
    }
}
```

2.Hello Controller

La clase HelloController administra los eventos de la aplicación.

Tengo mis atributos públicos que corresponden con los componentes de la app(del diseño).

-**Initialize** = es el método que primero se ejecuta al iniciar el programa , es por ello que cuando iniciamos el programa por primera vez lo que necesitamos es ver la interfaz , pero además el comboBox con los parámetros para elegir(POST/PUT….) y el que viene por defecto el POST.

-**EjecutarConsulta** = es un método que se ejecuta al darle al botón de ejecutar en el que recogeremos la opción elegida por el comboBox y dependiendo de la opción llamaremos a la clase operador para ejecutar uno de sus métodos, después de ejecutar dicho método necesitaremos una respuesta del servidor , es por ello que todos los métodos devuelven una respuesta la cual es mostrada al cliente por pantalla

```
ManuelGonzalez709 +1 *
public class HelloController implements Initializable {
    @FXML TextArea txtParametros;
    @FXML TextArea txtRespuesta;
    @FXML Button btBEnviarConsulta;
    @FXML ComboBox listaOpciones;
    @FXML TextField Ruta;
    4 usages
    private Operador op = new Operador();
    ManuelGonzalez709 +1
    @Override
    public void initialize(URL url, ResourceBundle resourceBundle) {
        ObservableList<String> items = FXCollections.observableArrayList();
        items.addAll( ...es: "POST", "PUT", "GET", "DELETE");listaOpciones.setItems(items);
        listaOpciones.setValue("POST");
    }

    1 usage   ManuelGonzalez709 +1
    @FXML
    public void EjecutarConsulta(){
        switch (listaOpciones.getValue().toString()){
            case "POST":txtRespuesta.setText(op.post(Ruta.getText(),txtParametros.getText()));break;
            case "PUT":txtRespuesta.setText(op.put(Ruta.getText(),txtParametros.getText()));break;
            case "GET":txtRespuesta.setText(op.get(Ruta.getText()));break;
            case "DELETE":txtRespuesta.setText(op.delete(Ruta.getText()));break;
        }
    }
}
```

3.Operador

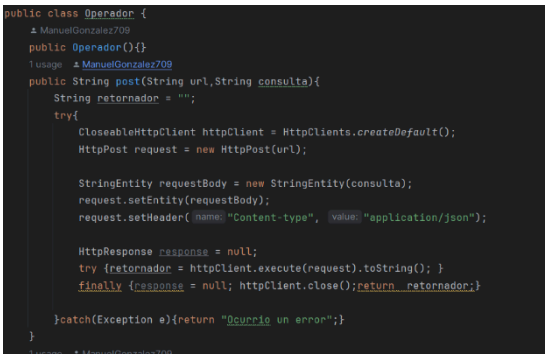
La clase Operador contiene 4 métodos :

-POST : Método que recibe por parámetro la url y la consulta desde el proyecto principal y al ejecutar la consulta con HttpPost para **insertar** datos, también lo que hace es mandar la contestación del servidor/springBoot server al cliente para mostrarlo por la pantalla.

-PUT : Método que recibe por parámetro la url y la consulta desde el proyecto principal y al ejecutar la consulta con HttpPut para **actualizar** datos, además lo que hace es mandar la contestación del servidor/springBoot server al cliente para mostrarlo por la pantalla.

-PUT : Método que recibe por parámetro la url y usa esta junto HttpResponse para **obtener** datos de la base de datos ,además lo que hace es mandar la contestación del servidor/springBoot server al cliente para mostrarlo por la pantalla.

-DELETE : Método que recibida una url (la cual contiene el id del que queremos borrar), realiza un borrado de dicho dato en la base de datos marcada por la url.

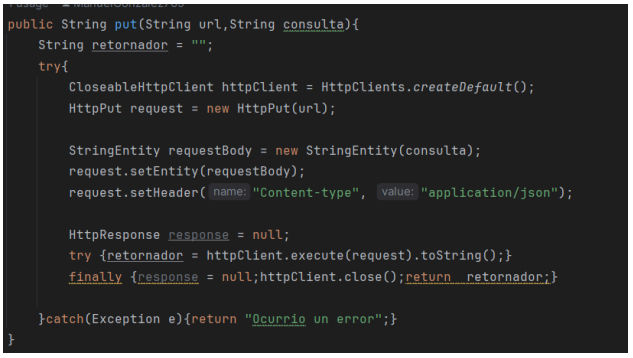


```
public class Operador {
    ManuelGonzalez709
    public Operador(){}
    1 usage   ManuelGonzalez709
    public String post(String url,String consulta){
        String retornador = "";
        try{
            CloseableHttpClient httpClient = HttpClients.createDefault();
            HttpPost request = new HttpPost(url);

            StringEntity requestBody = new StringEntity(consulta);
            request.setEntity(requestBody);
            request.setHeader( name: "Content-type",  value: "application/json");

            HttpResponse response = null;
            try {retornador = httpClient.execute(request).toString(); }
            finally {response = null; httpClient.close();return retornador;}

        }catch(Exception e){return "Ocurrio un error";}
    }
```



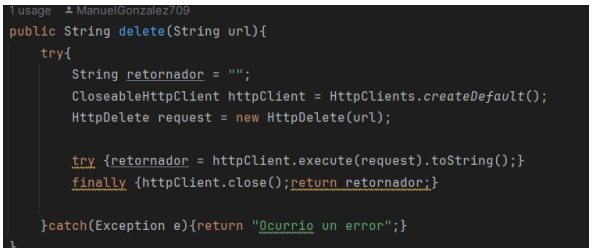
```
public String put(String url,String consulta){
    String retornador = "";
    try{
        CloseableHttpClient httpClient = HttpClients.createDefault();
        HttpPut request = new HttpPut(url);

        StringEntity requestBody = new StringEntity(consulta);
        request.setEntity(requestBody);
        request.setHeader( name: "Content-type",  value: "application/json");

        HttpResponse response = null;
        try {retornador = httpClient.execute(request).toString();}
        finally {response = null;httpClient.close();return retornador;}

    }catch(Exception e){return "Ocurrio un error";}
}
```

```
public String get(String url){
    try{
        String retornador = "";
        CloseableHttpClient httpClient = HttpClients.createDefault();
        HttpGet request = new HttpGet(url);
        CloseableHttpResponse response = httpClient.execute(request);
        try {
            BufferedReader reader = new BufferedReader(new InputStreamReader(response.getEntity().getContent()));
            String line;
            StringBuilder result = new StringBuilder();
            while ((line = reader.readLine()) != null) {
                result.append(line);
            }
            retornador = result.toString();
            retornador = retornador.replace( target: "},", replacement: "}\n");
        } finally {response.close();return retornador;}
    }catch(Exception e){return "Ocurrio un error";}
}
```



```
1 usage   ManuelGonzalez709
public String delete(String url){
    try{
        String retornador = "";
        CloseableHttpClient httpClient = HttpClients.createDefault();
        HttpDelete request = new HttpDelete(url);

        try {retornador = httpClient.execute(request).toString();}
        finally {httpClient.close();return retornador;}

    }catch(Exception e){return "Ocurrio un error";}
}
```