GUÍA DE INTEGRACIÓN DE SPRING BOOT Y ANGULAR CON MAVEN

ÍNDICE

1. ENTORNO DE DESARROLLO

Para crear este proyecto utilizaremos Visual Studio Code por ser una de las alternativas gratuitas de las que disponemos (https://code.visualstudio.com/).

Atajos de Visual Studio Code:

Algunos de los atajos que nos pueden resultar útiles a la hora de desarrollar en este entorno son los siguientes:

Shift+Alt+O: eliminar paquetes importados no usados.

Shift+Alt+cursor arriba o abajo: duplicar línea.

Shift+Alt+A: crear bloque de comentarios.

Shift+Alt+Click izquierdo: seleccionar varias líneas al mismo tiempo.

También podemos crear atajos personalizados, por ejemplo para transformar texto a mayúsculas o minúsculas. En este supuesto para ello presionamos pulsamos Ctrl+Shift+P y escribimos "Upper" o "Lower". Si hacemos doble click sobre cualquier opción podremos crear un atajo. En mi caso he utilizado Shift+Alt+y para convertir texto a mayúsculas y Shift+Alt+x para convertir texto a minúsculas.

2. PRIMEROS PASOS CON SPRING BOOT

Para crear nuestro primer proyecto utilizaremos Spring Initializr: https://start.spring.io/. Al importar dependencias es probable que aparezcan errores si no hemos configurado bien la aplicación. Esto puede ocurrir por ejemplo si importamos depedencias relacionadas con bases de datos y no especificamos la base de datos que vayamos a usar.

Deberemos tener instalado JDK 1.8 y la última versión de Maven. Para instalar Maven, después de descargar el zip, tendremos que extraer el contenido en archivos de programas. También deberemos configurar las variables de entorno, entre otros: JAVA_HOME (C:\Program Files\Java\jdk1.8.0_201), MAVEN_HOME (C:\Program Files\apache-maven-3.6.3) Y PATH (C:\Program Files\apache-maven-3.6.3\bin).

Para ejecutar la aplicación utilizar el comando siguiente:

```
mvnw spring-boot:run
```

Una vez la ejecutemos podremos acceder a la aplicación en localhost:8080 o utilizando el comando siguiente: curl localhost:8080 (ver más información en https://spring.io/guides/gs/spring-boot/ y https://spring.io/guides/gs/serving-web-content/)

Ahora para el renderizado de las plantillas necesitaremos importar Thymeleaf en nuestras dependencias (pom.xml):

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

También necesitaremos importar en dependencias Spring Dev-Tools para habilitar los cambios en caliente de las plantillas:

```
<dependency>
	<groupId>org.springframework.boot</groupId>
	<artifactId>spring-boot-devtools</artifactId>
	<optional>true</optional>
</dependency>
```

Así no necesitaremos reiniciar el servidor cuando hagamos cambios en nuestras vistas. Si vemos que los cambios en caliente de las plantillas no tienen efecto deberemos modificar el fichero pom.xml, en concreto el plugin de Maven (`spring-boot-maven-plugin`). Deberemos añadir lo resaltado:

```
<plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <configuration>
        <addResources>true</addResources>
    </configuration>
</plugin>
```

Para limpiar nuestro proyecto si tenemos algún problema utilizaremos el comando siguiente:

```
mvn clean package
```

-Si utilizamos Visual Code Studio, y queremos ejecutar el proyecto desde el mismo programa, podemos instalar las extensiones Debugger for Java, Language Support for Java y Java Extension Pack. Esto nos permitirá ejecutar y depurar nuestra aplicación de forma más cómoda desde el mismo IDE e identificar los errores en el código. Posteriormente deberemos acceder a preferencias y deshabilitar el compilado en tiempo real, esto es con el fin poder compilar nuestro proyecto multimodulos de Spring Boot y Angular.

3. PRIMEROS PASOS EN ANGULAR

Primero deberemos instalar Node.js y el gestor de paquetes npm. Para comprobar la versión que tenemos instalada podemos utilizar los siguientes comandos:

```
node -v
```

```
npm -v
```

Seguidamente instalaremos angular CLI que utilizaremos para crear nuestros proyectos de Angular. Para hacerlo ejecutar el siguiente comando:

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

Para crear un proyecto en el directorio que queramos haremos lo siguiente:

```
ng new nombreDelProyecto
```

A continuación para compilar y ejecutar el proyecto nos situaremos en el directorio del mismo y ejecutaremos el siguiente comando:

```
ng serve --open
```

El parámetro --open nos permite simplemente abrir en el navegador que tengamos por defecto la página de inicio de nuestro proyecto. Si queremos acceder manualmente deberemos hacerlo a través de localhost:4200. Si quisiéramos modificar el puerto de acceso deberemos modificar los ficheros schema.json, situados en la carpeta node_modules, concretamente en el módulo @angular-devkit (.\@angular-devkit\build-angular\src\dev-server\schema.json). Una vez modificado el puerto por defecto en ese fichero podremos acceder a través del mismo al ejecutar la aplicación nuevamente.

Para más información sobre la creación de proyectos, componentes y formularios ver la documentación oficial (https://angular.io/guide/setup-local).

4. CREACIÓN DE PROYECTOS MULTIMODULOS CON MAVEN

Una vez hayamos creado proyectos básicos con Spring Boot y Angular podremos pasar a integrarlos en una misma aplicación. De esta forma podremos utilizar Spring Boot para el backend y Angular para el frontend. Para hacer lo anterior primero deberemos crear un nuevo proyecto utilizando Maven, lo podemos hacer utilizando la misma herramienta que usamos para crear proyectos de Spring Boot (https://start.spring.io/) o simplemente utilizando Maven (mvn -B archetype:generate -DarchetypeGroupId=org.apache.maven.archetypes -DgroupId=com. nombreDeLaAplicacion.app -DartifactId=nombreDeLaAplicacion). Este proyecto sólo lo utilizaremos como contenedor para nuestros dos módulos, por lo que podremos eliminar la carpeta src.

A continuación crearemos dos carpetas dentro del proyecto, con los nombres de nuestros módulos: backend y frontend. En backend incluiremos la carpeta src que se encuentra en la raíz del proyecto que creamos. Este será el módulo que contendrá Spring Boot. Y en frontend incluiremos nuestro proyecto de Angular tal cual. Para que estos módulos se incluyen en nuestro proyecto de Maven deberemos editar el fichero pom.xml de la raíz del proyecto. Su contenido será algo parecido a lo siguiente:

```
<?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.a
pache.org/xsd/maven-4.0.0.xsd">
    <packaging>pom</packaging>
    <modelVersion>4.0.0</modelVersion>
    <groupId>com</groupId>
    <artifactId>demo</artifactId>
    <version>DEMO0.0.1</version>
    <name>DEMO</name>
    <description>Demo project for Spring Boot</description>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.2.6.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>

    <properties>
```

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

    <modules>
        <module>backend</module>
        <module>frontend</module>
    </modules>
</project>
```

La etiqueta packaging se establece como “pom” para únicamente instalar y desplegar los diferentes módulos que se incluyan. A su vez se incluye como padre el pom Spring Boot Starter Parent, el cual permite hacer uso de una serie de dependencias que usaremos en el módulo backend. Por último en la etiqueta modules se incluyen los módulos o subproyectos que contendrá nuestro proyecto. En nuestro caso estos módulos son backend y frontend.

Cada módulo deberá contener a su vez un pom.xml que establecerá las dependencias de su correspondiente subproyecto o módulo. Estos ficheros pom.xml los deberemos crear nosotros. En el módulo backend incluiremos el contenido de nuestro proyecto de Spring Boot y este archivo pom.xml. De la misma forma en el módulo frontend incluiremos un fichero pom.xml y todo el contenido de nuestro proyecto de Angular.

El contenido del fichero pom en el módulo backend incluirá en hará referencia en la etiqueta parent al proyecto Maven que agrupa los módulos. Deberemos incluir también una serie de dependencias entre las que se encuentran el módulo frontend. En la sección build incluiremos un directorio target donde se guardará el proyecto compilado con el jar, el cual ejecutaremos más tarde para hacer funcionar la aplicación. En este apartado determinamos el proceso de compilado. El contenido final del fichero será parecido al siguiente:

```
<?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.a
pache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>com</groupId>
        <artifactId>demo</artifactId>
        <version>DEMO0.0.1</version>
    </parent>
    <artifactId>backend</artifactId>
    <version>DEMO-0.0.1</version>
    <name>Backend</name>
    <description>Demo project for Spring Boot</description>

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

```
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
            <exclusions>
                <exclusion>
                    <groupId>org.junit.vintage</groupId>
                    <artifactId>junit-vintage-engine</artifactId>
                </exclusion>
            </exclusions>
        </dependency>
        <dependency>
            <groupId>com</groupId>
            <artifactId>frontend</artifactId>
            <version>${project.version}</version>
            <scope>runtime</scope>
        </dependency>
    </dependencies>
    <build>
        <directory>../target</directory>
        <plugins>
            <plugin>
                <artifactId>maven-resources-plugin</artifactId>
                <executions>
                    <execution>
                        <id>copy-resources</id>
                        <phase>validate</phase>
                        <goals><goal>copy-resources</goal></goals>
                        <configuration>
                            <outputDirectory>${project.build.directory}/c
lasses/resources/</outputDirectory>
                            <resources>
                                <resource>
                                    <directory>${project.parent.basedir}/
frontend/dist/demo/</directory>
                                </resource>
                            </resources>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>
```

Al contrario que en el módulo backend, en el fichero pom del módulo frontend no se incluirán dependencias. En este fichero deberemos por tanto borrar la etiqueta dependencies. Deberemos también modificar la etiqueta build para realizar la compilación de nuestro proyecto de Angular. Igualmente incluiremos el subdirectorio donde se localizarán las vistas tras la compilación, esto lo haremos con la etiqueta resources. El resultado será algo parecido a lo siguiente:

```
<?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.a
pache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>com</groupId>
        <artifactId>demo</artifactId>
        <version>DEMO0.0.1</version>
    </parent>
    <artifactId>frontend</artifactId>
    <version>DEMO-0.0.1</version>
    <name>Frontend</name>
    <description>Demo project for Spring Boot</description>
    <build>
        <plugins>
            <plugin>
                <groupId>com.github.eirslett</groupId>
                <artifactId>frontend-maven-plugin</artifactId>
                <version>1.9.1</version>
                <configuration>
                    <workingDirectory>src/main/angular</workingDirectory>
                    <nodeVersion>v10.16.0</nodeVersion>
                    <npmVersion>6.9.0</npmVersion>
                </configuration>
                <executions>
                    <execution>
                        <id>install node and npm</id>
                        <goals>
                            <goal>install-node-and-npm</goal>
                        </goals>
                    </execution>
                    <execution>
                        <id>npm install</id>
                        <goals>
                            <goal>npm</goal>
                        </goals>
                    </execution>
                    <execution>
```

```
                        <id>npm run build</id>
                        <goals>
                            <goal>npm</goal>
                        </goals>
                        <configuration>
                            <arguments>run build</arguments>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
        </plugins>
        <resources>
            <resource>
                <directory>./dist/qgen</directory>
                <targetPath>static</targetPath>
            </resource>
        </resources>
    </build>
</project>
```

En este punto sólo nos quedará ejecutar el comando mvn clean install para compilar el proyecto. Si lo hemos hecho todo correctamente el proceso terminará generando un jar en la carpeta target de la raíz del proyecto. Para ejecutar este jar utilizaremos el comando siguiente:

java –jar ./target/nombreDelJAR.jar

Hecho esto podremos acceder a la página principal realizada en Angular accediendo a localhost:8080. Si aparece algún error deberemos revisar los pasos realizados hasta ahora.

Resumiendo este apartado podríamos decir que la creación de nuestro proyecto multimódulos se resume en:

1. Crear un proyecto de Spring Boot en el cual tendremos dos carpetas: frontend y backend.

2. Modificaremos o añadiremos los tres ficheros pom.xml a sus correspondientes carpetas con el contenido que hemos visto en las páginas anteriores.

3. Situaremos el contenido del proyecto Angular dentro de la carpeta de frontend.

4. Situaremos el contenido de un nuevo proyecto Spring Boot dentro de la carpeta backend.

5. Ejecutamos el comando siguiente para compilar:

mvn clean install

6. Ejecutamos el comando siguiente para ejecutar la aplicación:

java –jar ./target/nombreDelJAR.jar

7. Accedemos a la aplicación en cualquier navegador a través de http://localhost:8080

5. HABILITANDO EL ACCESO MEDIANTE HTTPS CON SPRING BOOT

Para habilitar el acceso mediante el protocolo HTTPS deberemos primero crear un certificado. Podemos utilizar una herramienta que nos ofrece JDK llamada keytool. Nos deberemos dirigir por tanto al directorio de JDK:

```
cd C:\Program Files\Java\jdk1.8.0_201\bin
```

Hecho lo anterior utilizaremos el comando siguiente para crear el certificado, modificando si queremos el nombre del mismo:

```
keytool -genkeypair -alias qgen -keyalg RSA -keysize 2048 -storetype PKCS12 -keystore
qgen.p12 -validity 3650
```

Una vez tengamos nuestro certificado deberemos copiarlo y pegarlo en nuestro proyecto en la carpeta “resources” de nuestra aplicación. Podemos crear una carpeta llamada “keystore” para la ocasión. Seguidamente deberemos modificar el fichero application.properties de la aplicación para establecer una configuración como la siguiente:

```
http.port=8081
server.port=8080
# The format used for the keystore. It could be set to JKS in case
it is a JKS file
server.ssl.key-store-type=PKCS12
# The path to the keystore containing the certificate
server.ssl.key-store=classpath:keystore/qgen.p12
# The password used to generate the certificate
server.ssl.key-store-password=qgen1234
# The alias mapped to the certificate
server.ssl.key-alias=qgen

#trust store location
trust.store=classpath:keystore/qgen.p12
#trust store password
trust.store.password=qgen1234
```

Deberemos tener en cuenta que deberemos de cambiar en el texto anterior las referencias al nombre real de nuestro certificado y su contraseña, en este caso qgen y qgen1234. También si queremos deberemos modificar los puertos de acceso a nuestra aplicación (http.port y server.port).

Por último deberemos crear una nueva clase que extienda de WebSecurityConfigurerAdapter. Aquí configuraremos la aplicación para permitir el acceso a todas las direcciones, así como la utilización de los métodos POST, PUT y DELETE que veremos más adelante (apartado 7.2):

```
package com.example.demo;

import org.springframework.security.config.annotation.web.builders.
HttpSecurity;
import org.springframework.security.config.annotation.web.configura
tion.EnableWebSecurity;
```

```
import org.springframework.security.config.annotation.web.configura
tion.WebSecurityConfigurerAdapter;

@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {


    // Para habilitar el acceso a cualquier dirección.

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
                    .antMatchers("/**")
                    .permitAll();
    }

    // Para deshabilitar la protección contra ataques CSRF y
permitir los métodos POST, PUT y DELETE

    http.csrf().disable();
}
```

En este punto si hemos hecho todo correctamente ya podremos acceder a la aplicación a través de https://localhost:8080/. Por último, si queremos habilitar el acceso mediante http deberemos configurar el acceso programáticamente. Esto lo haremos creando una nueva clase que utilice ServletWebServerFactory:

```
package com.example.demo;

import org.apache.catalina.Context;
import org.apache.catalina.connector.Connector;
import org.apache.tomcat.util.descriptor.web.SecurityCollection;
import org.apache.tomcat.util.descriptor.web.SecurityConstraint;
import org.springframework.boot.web.embedded.tomcat.TomcatServletWe
bServerFactory;
import org.springframework.boot.web.servlet.server.ServletWebServer
Factory;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class TomcatHttpAndHttpsConfig {

  @Bean
  public ServletWebServerFactory servletContainer() {
    TomcatServletWebServerFactory tomcat =
        new TomcatServletWebServerFactory() {
          @Override
          protected void postProcessContext(Context context) {
```

```
                SecurityConstraint securityConstraint = new SecurityCon
straint();
                securityConstraint.setUserConstraint("CONFIDENTIAL");
                SecurityCollection collection = new SecurityCollection(
);
                collection.addPattern("/*");
                securityConstraint.addCollection(collection);
                context.addConstraint(securityConstraint);
              }
            };
        tomcat.addAdditionalTomcatConnectors(redirectConnector());
        return tomcat;
      }

      private Connector redirectConnector() {
        Connector connector = new Connector("org.apache.coyote.http11.H
ttp11NioProtocol");
        connector.setScheme("http");
        //System.out.println("check httpPort " + properties.http.port);
        connector.setPort(8081);
        connector.setSecure(false);
        connector.setRedirectPort(8080);
        return connector;
      }
    }
```

Hecho lo anterior cuando accedamos a http://localhost:8081 se nos redirigirá a https://localhost:8080. El puerto 8081 se corresponderá al puerto especificado en la propiedad http.port del application.properties. Asimismo el puerto 8080 corresponderá al especificado en la propiedad server.port.

6. PROPIEDADES CON SPRING BOOT

Con Spring Boot podemos utilizar y crear variables de entorno que pueden ser usadas en la aplicación en cualquier momento. Estás propiedades se definen en el fichero application.properties de nuestra aplicación. Podremos pues añadir una variable de ejemplo:

```
app.puerto=8082
```

Para obtener el valor de la propiedad que hemos definido deberemos incluir un nuevo modelo, app, que tendrá un campo llamado puerto. En la creación de este modelo deberemos utilizar una serie de anotaciones que nos permitirán acceder a los valores de los variables. La anotación @ConfigurationProperties incluirá como parámetros el nombre del prefijo de la propiedad (prefijo.nombreDelCampo):

```
import org.springframework.boot.context.properties.ConfigurationPropertie
s;
import org.springframework.context.annotation.Configuration;
```

```
@Configuration
@ConfigurationProperties("app")
public class AppProperties {
    public static String puerto;

    public void setPuerto(String puerto) {
        this.puerto = puerto;
    }
    public String getPuerto() {
        return puerto;
    }

    public static Objeto objeto = new Objeto();

    public void setObjeto(Objeto objeto) {
        this.objeto = objeto;
    }
    public Objeto getObjeto() {
        return objeto;
    }

    public static class Objeto{
        public String prueba;

        public void setPrueba(String prueba) {
            this.prueba = prueba;
        }
        public String getPrueba() {
            return prueba;
        }
    }
}
```

Como vemos el nombre de los campos deberá corresponderse con el que figura en el nombre de la propiedad. Como vemos en la zona marcada en amarillo, si nuestra propiedad (app.puerto) tiene un prefijo únicamente bastará con crear una variable. Sin embargo, como vemos en la zona marcada en verde, si tiene varios prefijos (app.objeto.prueba) deberemos crear objetos basados en los prefijos adicionales. Hay que tener en cuenta que a la etiqueta @ConfigurationProperties le pasaremos como parámetro el prefijo o prefijos del que arranque nuestro modelo. Si queremos crear un modelo que contenga todas las propiedades de un fichero podríamos omitir esta parte, la remarcada en azul celeste.

Finalmente, para obtener el valor de estas variables desde otra clase, tendremos que crear una instancia de nuestra entidad, en nuestro caso AppProperties. Para que funcione deberemos utilizar la anotación @EnableConfigurationProperties:

```
import org.springframework.stereotype.Controller;
import org.springframework.boot.context.properties.EnableConfigurationPro
perties;
```

(…)

```
@Controller
@EnableConfigurationProperties(AppProperties.class)
public class NombreDePruebaController {
```

(…)

Al utilizar la anotación @EnableConfigurationProperties utilizaremos como parámetro la clase con el modelo que creamos para las propiedades. Si quisiéramos utilizar varias entidades con propiedades diferentes haríamos lo siguiente:

```
@EnableConfigurationProperties({AppProperties.class, Properties.class})
```

Otra forma de acceder a los valores de estas propiedades es utilizando la anotación @Value. Para ello crearemos una clase de la siguiente forma:

(…)

```
import org.springframework.boot.context.properties.ConfigurationPropertie
s;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component
@ConfigurationProperties
public class Properties {
    @Value("${trust.store}")
    public String store;
}
```

Como vemos en el ejemplo anterior, el parámetro que utilizamos con la etiqueta Value es el nombre completo de la propiedad. El nombre del campo en este caso no es necesario que sea el mismo. En este caso para acceder al valor de estas variables desde otra clase utilizaremos la etiqueta @Autowired:

(…)

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

@Controller
@EnableConfigurationProperties({AppProperties.class, Properties.class})
public class CheckController {
    private AppProperties appProperties;
    @Autowired
    private Properties properties;
```

(…)

Esta etiqueta lo que hará será inyectar los valores en el objeto que instanciemos o declaremos. Por ello incluimos la anotación @Component al declarar nuestro modelo, ya que al utilizar Autowired se escanea la clase en busca de este componente así definido. Si no utilizáramos

esta etiqueta al hacer referencia al campo en cuestión obviamente el valor sería nulo. Esto siempre que inicializáramos el objeto, puesto que si no lo inicializáramos obviamente ocurriría una excepción (NullPointerException).

7. CONEXIÓN CON BASES DE DATOS

Existen diferentes formas de conectar nuestra aplicación a una base de datos. En este apartado veremos algunas de ellas.

7.1. JDBC e HIBERNATE

La primera opción que tenemos es utilizar JDBC e HIBERNATE. Utilizando estas dos herramientas podremos mapear entidades y conectarnos con Java a una base de datos MySQL. Para ello debemos declarar una serie de dependencias en el fichero pom.xml de nuestro proyecto de Spring Boot:

```
<dependency>
     <groupId>mysql</groupId>
     <artifactId>mysql-connector-java</artifactId>
     <version>8.0.19</version>
</dependency>
<dependency>
     <groupId>org.hibernate</groupId>
     <artifactId>hibernate-core</artifactId>
     <version>5.4.10.Final</version>
</dependency>
```

Lo siguiente que haremos será modificar la configuración de nuestra aplicación de acuerdo a la base de datos que vamos a usar. Tendremos que modificar el puerto de acceso, el nombre de la base de datos, el nombre de usuario y la contraseña. Para hacerlo debemos abrir el fichero application.properties y establecer las siguientes propiedades:

```
##### MySQL
#################### DataSource Configuration ######################
#####
spring.datasource.url=jdbc:mysql://localhost:3036/nombreBaseDeDatos
spring.datasource.username=root2
spring.datasource.password=fap
spring.datasource.driver.class=com.mysql.jdbc.Driver
#################### Hibernate Configuration ######################
####
spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto=update
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MyS
QL5Dialect
spring.jpa.generate-ddl=false
spring.jpa.properties.hibernate.hbm2ddl.auto=none
```

Igualmente tendremos que crear un fichero en resources llamado hibernate.cfg.xml. En este fichero modificaremos los mismos datos, pero además tendremos que declarar las clases que pretendemos mapear:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
    <session-factory>
        <!-- Database connection settings -->
        <property name="connection.driver_class">com.mysql.cj.jdbc.Driver
</property>
        <property name="connection.url">jdbc:mysql://localhost:3306/sprin
gjpa</property>
        <property name="connection.username">root2</property>
        <property name="connection.password">fap</property>

        <!-- SQL dialect -->
        <property name="dialect">org.hibernate.dialect.MySQL5Dialect</pro
perty>

        <!-- Echo the SQL to stdout -->
        <property name="show_sql">true</property>
        <!-- Drop and re-create the database schema on startup -->
        <property name="hbm2ddl.auto">validate</property>


        <!-- Names the annotated entity class -->
        <mapping class="com.example.demo.models.Persona"/>
    </session-factory>
</hibernate-configuration>
```

Las clases declaradas se deben corresponder con las tablas que debemos crear en nuestra base de datos. Más adelante veremos que no es necesario que creemos las tablas manualmente, pero en cualquier caso se deben corresponder con los modelos que establezcamos. Para crear los modelos tendremos que utilizar ciertas anotaciones, con una serie de parámetros, que deben corresponderse con la base de datos que crearemos. En nuestro ejemplo utilizaremos el siguiente ejemplo:

```
package com.example.demo.models;

import javax.persistence.*;
@Entity
@Table(name = "Persona")
public class Persona {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
    @Column(name = "id")
    Long id;
    @Column(name = "nombre")
    private String nombre;
    @Column(name = "apellidos")
    private String apellidos;

    public long getId() {
        return id;
    }
    public void setId(long id) {
        this.id = id;
    }
    public String getNombre() {
        return nombre;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    public String getApellidos() {
        return apellidos;
    }
    public void setApellidos(String apellidos) {
        this.apellidos = apellidos;
    }

    public Persona(String nombre, String apellidos) {
        this.nombre = nombre;
        this.apellidos = apellidos;
    }
    public Persona() {
        super();
    }
}
```

Del ejemplo anterior hay que destacar que en nuestra base de datos el campo id debe ser autoincremental, debemos habilitar la opción A_I. Eso sería al menos con el ejemplo que estamos viendo en este caso. Veremos más adelante que la creación de las tablas puede ser automática.

Seguidamente crearemos un paquete utils en nuestra aplicación, de la misma forma que hicimos con models y controllers. En este paquete crearemos una clase responsable de gestionar las conexiones a la base de datos. A esta clase la llamaremos HibernateUtil:

```
package com.example.demo.utils;

import org.hibernate.SessionFactory;
import org.hibernate.boot.Metadata;
import org.hibernate.boot.MetadataSources;
```

```
import org.hibernate.boot.registry.StandardServiceRegistry;
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;
public class HibernateUtil {
    private static StandardServiceRegistry registry;
    private static SessionFactory sessionFactory;
    public static SessionFactory getSessionFactory() {
        if (sessionFactory == null) {
            try {
                // Create registry
                registry = new StandardServiceRegistryBuilder().configure
().build();
                // Create MetadataSources
                MetadataSources sources = new MetadataSources(registry);
                // Create Metadata
                Metadata metadata = sources.getMetadataBuilder().build();
                // Create SessionFactory
                sessionFactory = metadata.getSessionFactoryBuilder().buil
d();
            } catch (Exception e) {
                e.printStackTrace();
                if (registry != null) {
                    StandardServiceRegistryBuilder.destroy(registry);
                }
            }
        }
        return sessionFactory;
    }
    public static void shutdown() {
        if (registry != null) {
            StandardServiceRegistryBuilder.destroy(registry);
        }
    }
}
```

En este punto, una vez hayamos creado la base de datos y si lo hemos hecho todo correctamente, ya tendremos configurado todo para acceder a nuestra base de datos. Para comprobar que funciona correctamente podemos realizar una simple consulta. El siguiente código lo podríamos utilizar por ejemplo en un controlador de alguna página que tengamos:

```
        Persona persona1 = new Persona("Miguel", "Santana");
        Persona persona2 = new Persona("Juanjo", "Cabrera");
        Transaction transaction = null;
        try (Session session = HibernateUtil.getSessionFactory().openSess
ion()) {
            // start a transaction
            transaction = session.beginTransaction();
            session.save(persona1);
```

```
            session.save(persona2);
            // commit transaction
            transaction.commit();
        } catch (Exception e) {
            if (transaction != null) {
                transaction.rollback();
            }
            e.printStackTrace();
        }
        try (Session session = HibernateUtil.getSessionFactory().openSess
ion()) {
            List < Persona > personas = session.createQuery("from Persona
", Persona.class).list();
            personas.forEach(s -> System.out.println(s.getNombre()));
        } catch (Exception e) {
            if (transaction != null) {
                transaction.rollback();
            }
            e.printStackTrace();
        }
```

Cada vez que se ejecute el código anterior veremos que se habrán creado dos nuevos registros en nuestra base de datos. Igualmente podremos comprobar el resultado de la consulta en la consola. Para borrar un registro en lugar del método save utilizaremos delete, pudiendo como vemos crear consultas personalizadas.

7.2. SPRING BOOT JPA

Otra forma un poco más complejo de conectarnos a una base de datos es mediante Spring Data JPA. Se trata de un módulo de Spring Boot que simplifica la forma de interactuar con una base de datos. De esta forma nos permite además evitar repetir código una y otra vez en nuestra aplicación, particularmente a la hora de realizar consultas.

En este caso, si seguimos con el ejemplo del apartado anterior, no necesitaremos algunos de los ficheros que sí que creamos para la conexión con JDBC. Podremos borrar los siguientes ficheros:

-JdbcConfiguration.java

-hibernate.cfg.xml

-HibernateUtil.java

Si hacemos lo anterior deberemos asegurarnos de borrar todas las referencias a dichos ficheros, ya que en caso contrario no podremos compilar la aplicación.

Seguidamente en nuestro archivo pom.xml quitaremos las dependencias que añadimos en el apartado anterior y utilizaremos las siguientes:

```
<dependency>
```

```
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.19</version>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
```

Como vemos si queremos seguir trabajando con mysql deberemos incluir mysql-connector-java. Notaremos que hemos eliminado la dependencia de Hibernate, pero esto es porque dicha dependencia viene incluida en spring-boot-starter-data-jpa. En este punto cabe señalar que la conexión a la base de datos la seguiremos configurando en el fichero de propiedades (application.properties). Utilizaremos la misma configuración en este apartado.

Lo siguiente que definiremos son las entidades que utilizaremos en nuestra aplicación de ejemplo. Seguiremos utilizando la entidad Persona que utilizamos en el ejemplo anterior. Sin embargo haremos algunos cambios:

```
package com.example.demo.models;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;

@Entity
public class Persona {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String nombre;
    private String apellidos;

    public Persona() {
        this(null);
    }

    public Persona(Long id) {
        this.setId(id);
    }

    public Persona(String nombre, String apellidos) {
        this.nombre=nombre;
        this.apellidos=apellidos;
    }

    public Persona(Long id, String nombre, String apellidos) {
```

```
        this.id=id;
        this.nombre=nombre;
        this.apellidos=apellidos;
    }

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public String getApellidos() {
        return apellidos;
    }

    public void setApellidos(String apellidos) {
        this.apellidos = apellidos;
    }
}
```

A continuación crearemos un repositorio a para cada entidad con la que vayamos a trabajar. Estos repositorios serán los que interactuarán directamente con la base de datos. Como veremos estos repositorios extenderán de JpaRepository. Sólo con extender esta clase ya podremos realizar las operaciones básicas de un CRUD, pero podremos añadir diferentes métodos para realizar consultas personalizas:

```
package com.example.demo.repositories;

import java.util.List;
import java.util.Optional;
import org.springframework.data.domain.Pageable;
import org.springframework.data.domain.Slice;
import org.springframework.data.domain.Sort;
import org.springframework.data.jpa.repository.Query;
import com.example.demo.models.*;
import org.springframework.stereotype.Repository;
import org.springframework.data.jpa.repository.JpaRepository;
```

```
@Repository
public interface PersonaRepository extends JpaRepository<Persona, L
ong> {
    Optional<Persona> findById(Optional<Long> id);

    List<Persona> findByApellidos(String apellidos);

    @Query("select p from Persona p where p.nombre = :nombre")
    List<Persona> findByNombre(String nombre);

    @Query("select p from Persona p where p.nombre = :name or p.ape
llidos = :name")
    List<Persona> findByNombreOApellidos(String name);

    Long removeByApellidos(String apellidos);

    Slice<Persona> findByApellidosOrderByIdAsc(String apellidos, Pa
geable page);

    List<Persona> findFirst2ByOrderByApellidosAsc();

    List<Persona> findTop2By(Sort sort);

    @Query("select p from Persona p where p.nombre = :#{#persona.no
mbre} or p.apellidos = :#{#persona.apellidos}")
    Iterable<Persona> findByNombreOApellidos(Persona persona);
}
```

Hay que tener en cuenta que la clase JpaRepository a su vez extiende de la clase PagingAndSortingRepository, que a su vez extiende de CrudRepository. Es esta clase la que implementa las operaciones básicas. Algunas de estas operaciones son: save para guardar y actualizar, delete para borrar, findAll para consutar todo, etc. Podemos consultar más información sobre estas clases y sus métodos en el siguiente enlace, poniendo especial atención en los métodos heredados: https://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/JpaRepository.html

Podemos utilizar en nuestras aplicaciones directamente los repositorios para realizar las consultas, sin embargo no es lo recomendable. Lo ideal es tener una capa intermedia para poder hacer diferentes operaciones antes realizar cambios en la base de datos. Esta capa intermedia son los servicios. Podremos crear un solo servicio para toda la aplicación, o incluso un servicio por cada entidad. En este ejemplo utilizaremos el siguiente:

```
package com.example.demo.services;

import java.util.List;
import java.util.Optional;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
```

```java
import com.example.demo.models.*;
import com.example.demo.repositories.*;

@Service
public class PersonaService {
    @Autowired
    private PersonaRepository personaRepository;

    //@Override
    public Optional<Persona> findById(Long id) {
        return personaRepository.findById(id);
    }

    //@Override
    public void save(Persona persona) {
        personaRepository.save(persona);
    }

    //@Override
    public List <Persona> findAll() {
        return personaRepository.findAll();
    }

    //@Override
    public void delete(long id) {
        personaRepository.deleteById(id);
    }
}
```

Estos servicios pueden ser llamados directamente o a través de una interfaz. Si quisiéramos utilizar una interfaz deberíamos renombrar esta clase como PersonaServiceImpl. También tendríamos que añadir “implements PersonaService” y descomentar las líneas comentadas. Lógicamente crearíamos una nueva clase llamada PersonaService que sería la interfaz que implementaríamos con PersonaServiceImpl:

```java
package com.example.demo.services;
import java.util.List;

public interface PersonaService {
    Optional<Persona> findById(Long id);
    void save(Persona persona);
    List<Persona> findAll()
    void delete(long id)
}
```

En este ejemplo no utilizaremos interfaz, en vez de ello utilizaremos el servicio o el repositorio directamente. Para ello, en el controlador donde queramos realizar las consultas, crearemos una instancia de alguna de estas dos clases utilizando Autowired:

```
@Autowired
private PersonaService service;
```

Podemos hacer lo mismo con el repositorio en vez de con el servicio, pero como ya dijimos no es lo más recomendado. Seguidamente en el método que sea podremos realizar las consultas que queramos simplemente haciendo referencia a dicha instancia:

```
(…)
service.save(new Persona("Luis", "Quesada"));//Create
Optional<Persona> op = service.findById(1L);//Read id 1
List<Persona> lp = service.findAll();//Read all
for(Persona p : lp){
    System.out.println("-"+p.getNombre());
}
service.save(new Persona(19L,"Luis", "Acosta"));//Update id 19
service.delete(new Persona(19L));//Delete id 19
(…)
```

Como vemos si utilizaramos PersonaRepository en lugar de PersonaService, sería exactamente igual salvo por el nombre de la instancia.

Para que todo lo anterior funcione debemos asegurarnos de que en la clase principal de la aplicación, o en los controladores, se declaran ciertas etiquetas. Etas etiquetas sirven para escanear el proyecto en busca de las entidades y sus repositorios. En este ejemplo lo podríamos hacer de la siguiente forma:

```
(…)
import org.springframework.boot.autoconfigure.SpringBootApplication
;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.data.jpa.repository.config.EnableJpaRepo
sitories;
import org.springframework.boot.autoconfigure.domain.EntityScan;

@SpringBootApplication
@ComponentScan(basePackages = {"com.example.demo"})
@EnableJpaRepositories(basePackages = "com.example.demo.repositorie
s")
@EntityScan(basePackages = "com.example.demo.models")
public class DemoApplication {
(…)
```

Concretamente nos referimos a las líneas resaltadas en amarillo. Obviamente deberemos importar los paquetes resaltados en verde, ya sea como hemos dicho en la clase principal o en el controlador que sea.

Otras anotaciones que utilizaremos serán las relacionadas con el acceso a las diferentes páginas y las consultas que realizaremos. Estas anotaciones serán las encargadas de mapear las consultas para acceder a ellas desde el front. Un ejemplo sencillo podría ser el siguiente:

(…)

```
import org.springframework.web.bind.annotation.*
```

(…)

```
@Controller
@Configuration
@EnableConfigurationProperties({AppProperties.class, Properties.class})
@RequestMapping(value="/people")
public class CheckController {
    private AppProperties appProperties;
    @Autowired
    private Properties properties = new Properties();

    @Autowired
    private PersonaRepository repository;

    @Autowired
    private PersonaService service;

    private Logger logger = LoggerFactory.getLogger(this.getClass());

    @RequestMapping("/add")
    public String index2() {
        return "forward:/index.html";
    }

    @RequestMapping("/update/{id}")
    public String index2(@PathVariable("id") Long id) {
        return "forward:/index.html";
    }

    @RequestMapping(value="/list", method = RequestMethod.GET)
    public String index() {
        return "forward:/index.html";
    }

    @ResponseBody
    @RequestMapping(value="/getlist", method = RequestMethod.GET)
    public List<Persona> checkFindAll(){
        return service.findAll();
    }

    @ResponseBody
    @RequestMapping(value="/get/{id}", method = RequestMethod.GET)
```

```
    public Optional<Persona> findById(@PathVariable("id") Long id){
        return service.findById(id);
    }

    @ResponseBody
    @RequestMapping(value = "/post", consumes = "application/json", produ
ces = "application/json", method = RequestMethod.POST)
    public boolean save(@RequestBody Persona persona){
        service.save(persona);
        return true;
    }

    @ResponseBody
    @RequestMapping(value = "/put/{id}", consumes = "application/json", p
roduces = "application/json", method = RequestMethod.PUT)
    public boolean save(@RequestBody Persona persona, @PathVariable Long
id){
        service.save(persona);
        return true;
    }

    @ResponseBody
    @RequestMapping(value = "/delete/{id}", method = RequestMethod.DELETE
)
    public boolean delete(@PathVariable("id") Long id){
        service.delete(id);
        return true;
    }

(…)
```

Como vemos el factor común de todas las llamadas que haremos será RequestMapping, que nos permite realizar peticiones de diferentes tipos (GET, POST, PUT y DELETE). Para que los métodos POST, PUT y DELETE funcionen apropiadamente previamente deberemos haber desahabilitado la protección contra ataques CSRF, en otro caso podemos utilizar sólo el método GET o adaptar la configuración a nuestras necesidades (para deshabilitar esta protección consultar el apartado 5).

Por otra parte, en la clase que acabamos de ver, se incluyen también algunos métodos index que se encargan de devolver al usuario la página a la que pretende acceder. Los otros métodos encargados de la consultas harán llamadas a la clase service. Por supuesto esta lógica puede variar, al igual que los parámetros que le pasamos a las diferentes funciones. Deberemos tener en cuenta especialmente el atributo value especificado en la anotación RequestMapping, ya que en Angular utilizaremos dichos valores para realizar las peticiones.

8. OTRAS ANOTACIONES.

Además de las anotaciones que hemos usado en los apartados anteriores, existen algunas más que pueden resultarnos de gran utilidad. Dos de estás anotaciones son PostConstruct y PreDestroy. Estas anotaciones nos permiten ejecutar código justo después de la inyección de dependencias y justo antes de finalizar la aplicación. Para usar estás dos anotaciones

deberemos utilizar también la anotación Component al inicio de la clase. A continuación veremos un ejemplo:

(…)

```
import javax.persistence.Entity;
import org.springframework.stereotype.Component;
import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;

@Entity
@Component
public class Persona {
(…)
    @PostConstruct
    private void init() {
        System.out.println("*Inicializando Persona");
    }

    @PreDestroy
    private void destroy() {
        System.out.println("*Finalizando Persona");
    }
(…)
```

Para ver en la consola el resultado del método destroy del ejemplo anterior sólo tendremos que finalizar la aplicación, ya sea manualmente o programáticamente. Para finalizar la aplicación de forma programática podemos hacer lo siguiente:

```
    (…)
    import org.springframework.beans.factory.annotation.Autowired;
    import org.springframework.boot.SpringApplication;
    import org.springframework.context.ApplicationContext;
    (…)

    @Autowired
    private ApplicationContext context;
    public void stop(){
        SpringApplication.exit(context);
    }
```

9. REALIZAR PETICIONES HTTP CON ANGULAR.

En este apartado veremos cómo podemos realizar peticiones a nuestro backend utilizando Angular. Para otros detalles sobre el manejo de este framework se recomienda consultar la documentación en el siguiente enlace: https://angular.io/docs. En este punto ya deberíamos

haber visto la creación de componentes y formularios. En cualquier caso veremos parte del código referente a esto mismo a continuación.

Al igual que hicimos con Spring Boot tendremos que crear un servicio que se encargará de realizar las peticiones HTTP. Para hacerlo utilizaremos el comando ng generate service <nombre del servicio>. Este servicio deberemos modificarlo, y adaptarlo a los métodos de los servicios que declaramos utilizando Spring (apartado 7.2):

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class PersonaService {

  private baseUrl = '/people';

  constructor(private http: HttpClient) { }

  getPersona(id: number): Observable<any> {
    return this.http.get(`${this.baseUrl}/get/${id}`);
  }

  createPersona(persona: Object): Observable<Object> {
    return this.http.post(`${this.baseUrl}/post`, persona);
  }

  updatePersona(id: number, persona: Object): Observable<Object> {
    return this.http.put(`${this.baseUrl}/put/${id}`, persona);
  }

  deletePersona(id: number): Observable<any> {
    return this.http.delete(`${this.baseUrl}/delete/${id}`);
  }

  getPersonasList(): Observable<any> {
    return this.http.get(`${this.baseUrl}/getlist`);
  }
}
```

Aunque se trata de un ejemplo sencillo, el código anterior nos sirve para ilustrar el funcionamiento básico de un servicio. Si repasamos el apartado 7.2, veremos que las direcciones relativas utilizadas deben corresponder a las que especificamos utilizando Spring Boot. De la misma forma los parámetros deben ser los mismos, y compatibles con los tipos que correspondan en las funciones que declaramos en Java. Por tanto si trabajamos con objetos deberemos crear modelos en Angular que se adapten a los que utilizamos con Spring Boot.

Estos modelos podemos guardarlos todo en una misma carpeta. Basándonos en nuestro ejemplo, y llamando al archivo persona.ts, crearemos el siguiente en la carpeta models:

```
export class Persona {
    id: number;
    nombre: string;
    apellidos: string;
}
```

Una vez tengamos nuestro servicio y el modelos definidos sólo tendremos que hacer llamadas a los métodos declarados cuando mejor nos convenga. Por ejemplo, si queremos mostrar un listado de registros crearemos un nuevo componente. El html de este componente será algo parecido a lo siguiente:

```
<div class="panel-body">
      <table class="table table-striped table-bordered">
          <thead>
              <tr>
                  <th>ID</th>
                  <th>Nombre</th>
                  <th>Apellidos</th>
              </tr>
          </thead>
          <tbody>
              <tr *ngFor="let persona of personas | async">
                  <td>{{persona.id}}</td>
                  <td>{{persona.nombre}}</td>
                  <td>{{persona.apellidos}}</td>
                  <td><button [routerLink]="['../../people/update',
persona.id]">Actualizar</button></td>
                  <td><button (click)="deletePersona(persona.id)">B
orrar</button></td>
              </tr>
          </tbody>
      </table>
  </div>
```

Como vemos en el texto resaltado en amarillo necesitaremos un array de objetos llamado personas. Este array lo declararemos en el fichero TypeScript de este componente. Y para poblar dicho array simplemente haremos una llamada al servicio al inicializar la página (texto resaltado en verde):

```
import { Observable } from "rxjs";
import { PersonaService } from "./../persona.service";
import { Persona } from "./../models/persona";
import { Component, OnInit } from "@angular/core";

@Component({
  selector: "app-persona-list",
```

```
  templateUrl: "./list.component.html",
  styleUrls: ["./list.component.css"]
})
export class PersonaListComponent implements OnInit {
  personas: Observable<Persona[]>;

  constructor(private personaService: PersonaService) {}

  ngOnInit() {
    this.reloadData();
  }

  reloadData() {
    this.personas = this.personaService.getPersonasList();
  }

  deletePersona(id: number) {
    this.personaService.deletePersona(id)
      .subscribe(
        data => {
          console.log(data);
          this.reloadData();
        },
        error => console.log(error));
  }
}
```

Si revisamos el código anterior vemos que la funcionalidad de borrar registros ya está implementada. Y como vemos simplemente se trata de utilizar el servicio que creamos pasándole los parámetros adecuados. Si seguimos con nuestro ejemplo, para crear registros, no será muy diferente. De esta forma crearemos otro componente que contendrá el siguiente html:

```
<h3>Crear nueva persona</h3>
<div [hidden]="submitted" style="width: 400px;">
    <form (ngSubmit)="onSubmit(persona)">
        <div class="form-group">
            <label for="name">Nombre</label>
            <input type="text" class="form-
control" id="nombre" (input)="change()" required [(ngModel)]="perso
na.nombre" name="nombre">
        </div>

        <div class="form-group">
            <label for="name">Apellidos</label>
            <input type="text" class="form-
control" id="apellidos" (input)="change()" required [(ngModel)]="pe
rsona.apellidos" name="apellidos">
```

```
        </div>

        <button type="submit" class="btn btn-
success">Crear</button>
    </form>
</div>

<div [hidden]="!done">
    <h4>Hecho</h4>
</div>
```

Como vemos en el texto resaltado en amarillo, tendremos que crear un método, llamado onSubmit en este caso, que se encargará de realizar la petición:

```
import { PersonaService } from './../persona.service';
import { Persona } from './../models/persona';
import { Component, OnInit } from '@angular/core';
import { FormBuilder } from '@angular/forms';

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

  persona: Persona = new Persona();
  submitted = false;
  checkoutForm;
  done = false;

  constructor(private personaService: PersonaService, private formB
uilder: FormBuilder) {
    this.checkoutForm = this.formBuilder.group({
      nombre: '',
      apellidos: ''
    });
  }

  ngOnInit() {
    this.done = false;
  }

  onSubmit(persona) {
    this.personaService.createPersona(persona)
      .subscribe(data => console.log(data), error => console.log(er
ror));
      this.done = true;
```

```
  }

  change() {
    this.done = false;
  }
}
```

Y será prácticamente igual para el caso de actualizar registros:

```
<h3>Actualizar</h3>
<div [hidden]="submitted" style="width: 400px;">
    <form [formGroup]="updateForm" (ngSubmit)="onSubmit(persona)">
        <div class="form-group">
            <label for="name">ID</label>
            <input type="text" class="form-
control" formControlName="id" id="id" required name="id" [readonly]
="true">
        </div>

        <div class="form-group">
            <label for="name">Nombre</label>
            <input type="text" class="form-
control" formControlName="nombre" id="nombre" (input)="change()" re
quired name="nombre" [readonly]="isDisabled">
        </div>

        <div class="form-group">
            <label for="name">Apellidos</label>
            <input type="text" class="form-
control" formControlName="apellidos" id="apellidos" (input)="change
()" required name="apellidos" [readonly]="isDisabled">
        </div>

        <button type="submit" class="btn btn-
success">Actualizar</button>
    </form>
</div>

<div [hidden]="!done">
    <h4>Hecho</h4>
</div>
```

En este último caso el fichero TypeScript será el siguiente:

```
import { PersonaService } from './../persona.service';
import { Persona } from './../models/persona';
import { Component, OnInit } from '@angular/core';
import { FormBuilder, FormControl } from '@angular/forms';
```

```
import { ActivatedRoute } from '@angular/router';
import { FormGroup } from '@angular/forms';


@Component({
  selector: 'app-update-persona',
  templateUrl: './update-persona.component.html',
  styleUrls: ['./update-persona.component.css']
})
export class UpdatePersonaComponent implements OnInit {
  ID = 0;
  persona: Persona = new Persona();
  submitted = false;
  checkoutForm;
  activatedRoute: ActivatedRoute;
  updateForm: FormGroup;
  id: FormControl;
  nombre: FormControl;
  apellidos: FormControl;
  isDisabled = false;
  done = false;

  constructor(private personaService: PersonaService, private formB
uilder: FormBuilder, private route: ActivatedRoute) {
    this.activatedRoute = route;
    this.checkoutForm = formBuilder;
    this.activatedRoute.params.subscribe(params => {
      this.ID = params['id'];
    });
    this.setForm();
  }

  ngOnInit(): void {
    this.done = false;
    this.personaService.getPersona(this.ID).subscribe(
      persona => {
        this.persona = persona;
        this.setForm();
      });
  }

  setForm() {
    this.updateForm = this.formBuilder.group({
      id: this.formBuilder.control({ value: this.persona?.id, disab
led: true}),
      nombre: this.formBuilder.control({ value: this.persona?.nombr
e, disabled: false}),
      apellidos: this.formBuilder.control({value: this.persona?.ape
llidos, disabled: false})
```

```
    });

    if(this.persona?.id==null) {
      setTimeout(() => {
        this.updateForm.controls.nombre.disable();
        this.updateForm.controls.apellidos.disable()}
      );
    }
    else {
      setTimeout(() => {
        this.updateForm.controls.nombre.enable();
        this.updateForm.controls.apellidos.enable()}
      );
    }
  }

  onSubmit(persona) {
    if(persona?.id!=null){
      persona.nombre=this.updateForm.controls.nombre.value;
      persona.apellidos=this.updateForm.controls.apellidos.value;
      this.personaService.updatePersona(this.ID, persona)
      .subscribe(data => console.log(data), error => console.log(er
ror));
      this.done = true;
    }
  }

  change() {
    this.done = false;
  }
}
```

Si revisamos el código anterior podremos ver también cómo deshabilitar y habilitar formularios, pero como dijimos al principio no nos detendremos en eso. Por supuesto para que este ejemplo funcione deberemos haber configurado correctamente nuestro fichero de rutas (app-routing.modules.ts). Estas rutas deben ajustarse a las que definimos en Java utilizando la anotación RequestMapping de Spring Boot:

```
import { CreatePersonaComponent } from './create-persona/create-
persona.component';
import { UpdatePersonaComponent } from './update-persona/update-
persona.component';
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { PersonaListComponent } from './list/list.component';

const routes: Routes = [
  { path: '', redirectTo: 'people/', pathMatch: 'full' },
```

```
  { path: 'people/list', component: PersonaListComponent },
  { path: 'people/add', component: CreatePersonaComponent },
  { path: 'people/update/:id', component: UpdatePersonaComponent }
];

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