

Nº	Sección/Subsección	Página
1	Creación del proyecto y sus dependencias	11-14
2.1	Explicación de una Entity	15
2.2	Creación del Modelo o Entity	16
2.3	Configuración del Modelo o Entity	17-18
3.1	Explicación de @Repository	19
3.2	Creación del @Repository	20
3.3	Configuración del @Repository	21
3.4	Métodos del CrudRepository	22
4.1	Explicación de la Interface Service	23
4.2	Creación de la Interface Service	24
5.1	Método Listar del CRUD	25
5.2	Método Insertar del CRUD	26
5.3	Método Borrar por ID del CRUD	27
5.4	Método Consultar por ID del CRUD	28
5.5	Método Actualizar por ID del CRUD	29
6.1	Explicación de @Service	30
6.2	Creación de @Service	31
6.3.1	Implementación de la Interface Service	32
6.3.2	Instanciación del @Repository	33
6.3.3	Declaración como @Service	34
6.3.4	Método Listar del CRUD en @Service	35
6.3.5	Método Guardar del CRUD en @Service	36
6.3.6	Método Borrar del CRUD en @Service	37
6.3.7	Método Actualizar por ID del CRUD en @Service	38-40
7.1	Explicación de @Controller	41

7.2	Creación del @Controller	42
7.3	Declaración como @Controller y asignación de ruta base	43
7.4	Instanciación de la Interface Service en el @Controller	44
7.5	Método Listar en el @Controller	45
7.6	Método Insertar en el @Controller	46-48
7.6.1	Explicación detallada del método Insertar en el @Controller	47-49
7.7	Método Consultar por ID en el @Controller	50-53
7.7.1	Explicación del método Consultar por ID en el @Controller	51-53
7.8	Método Actualizar por ID en el @Controller	54-58
7.8.1	Explicación del método Actualizar por ID en el @Controller	55-58
7.9	Método Eliminar por ID en el @Controller	59-62
7.9.1	Explicación del método Eliminar por ID en el @Controller	59-62
8.1	Configuración de la base de datos (application.properties)	63
9	Pruebas con Postman	64-67
9.1	Prueba Insertar	64
9.2	Pruebas Listar	65
9.3	Prueba Actualizar por id	66
9.4	Prueba Eliminar por id	67
10	Conectar react con sprint boot Con Axios	68-71
10.1	<b>Instalar Axios en el frontend react</b>	68
10.2	Consumiendo API REST GET del Backend en React	69
10.3	Conectando Frontend con Backend	70-71
10.3.1	Crear metodo para cargar los Usuarios en el hook reducer	70
10.3.2	Modificar el metodo de cargar usuarios en el hook personalizado (userUsers)	71-72

10.3.3	Pasar la funcion al hook contentx (userprovider)	73
10.3.4	Buscar el componente que va cargar los usuarios y pasarla la funcion	74
10.3.5	Ejecutar la funcion	75
10.3.6	darle permiso al backen para que react acceda	76
10.4	Consumindo API RESTFul POST PUT y DELETE del Backend en React	77-79
10.4.1	Metodo save	77
10.4.2	Metodo actualizar	78
10.4.3	Metodo Eliminar	79
10.5	Conectando React con Backend Spring Boot CRUD	80-81
10.5.1	Modificar el metodo de cargar y actualizar usuarios en el hook personalisado (userUsers)	81
10.5.2	Modificar el metodo de eliminar usuarios en el hook personalisado (userUsers)	81
11	Validaciones desde el Backend Java bean	82-
11.1	Que es ?	82
11.2	Anotaciones comunes de validacion	83
11.3	Instalar dependencias	84
11.4	Validar en el Entity o modelo	85
11.5	Validar los metodos Post y Pust del controller	86-93
11.5.1	metodo post (para el pust es el mismo proceso)	86-93
12	Conectar validaciones del Backend en react	91-
12.1	Capturar el error en el service del fronted en los metodos save y update	91
12.2	Modificar los metodoss save y actualizar donde los estamos utilizando hook personalizado ((userUsers) para Capturar los errores	92
12.2.1	Envolver el metodo en un try cath	92
12.2.2	Crear objeto que va ser el estado inicial para el manejo de los errores y crear el hook useState	93
12.3	Eliminar las validaciones que tengamos en el frontend que no sean del backend (en caso de que tengamos)	95
12.4	Gestionar los errores	96
12.5	Mostrar los errores en un formulario POST y PUT	98-
12.5.1	Pasar el estado Inicial de los errores(errors) al contexto	98
12.5.2	Utilizar el hook contexto en el formulario para mostrar los errores	99
12.5.3	Mostrar los errores en el formulario POST Y PUT	100

12.5.4	Configurar para que los errores no se queden viendo al cerrar el formulario, y para que no se reinicie los campos	101
12.6	Configurar el actualizar en el backend para que no muestre el mensaje de error en el password	102
12.6.1	crear una nueva clase	103
12.6.1.1	Configuracion de la class UserRequest	105
12.6.1.2	Modificar el metodo actualizar de la interface UserInterfaceService	109
12.6.1.3	Modificarel metodo actualizar del controller	110
12.6.1.5	Modificarel metodo actualizar del service	111
12.7	Mostrar errores de campos unicos (nick o email ya existen)	112
12.7.1	Colocar nombre de clave a los campos unicos en la base de datos	112
12.7.2	Capturar el error para Mostrar los errores de campos unicos (nick o email ya existen) en el fomulario	114
12.7.2.1	Manejo de errores relacionados con restricciones de base de datos (500)	114
13	Backend Spring security JWT	115
13.1	Configurar Spring security	115
13.1.1	Instalar Dependencias	115
13.1.2	Crear la clase para Configuración de seguridad en Spring Boot con SecurityFilterChain	116
13.1.3	Configurar la class	117
13.1.3.1	Decirle a la class que va ser una configuracion	117
13.1.3.2	Método filterChain(HttpSecurity http)	118
13.1.3.3	Anotacion Bean para el Método filterChain(HttpSecurity http)	119
13.1.3.4	Configuracion de auotizacion del metodo filterChain(HttpSecurity http)	120
13.1.3.4.1	Permitir el acceso a las páginas sin autenticacion	120
13.1.3.4.2	Restringir el acceso a las páginas que requieran autenticacion	122
13.1.3.4.3	Desactivación de CSRF	123
13.1.3.4.4	Gestión de Sesiones	124
13.1.3.4.5	Construcción del SecurityFilterChain	125
13.2	Crear y Modificar el filtro por defecto de autenticacion para login de spring security para que funcione con Api res (JSON) por medio de Post en el cuerpo del request	127
13.2.1	Crear class para Filtro de Autenticación JWT	127
13.2.2	Configurar la class para Filtro de Autenticación JWT	128

13.2.2.1	Heredar la class UsernamePasswordAuthenticationFilter para poder Personanilzar el comportamiento del filtro de autenticacion	128
13.2.2.2	Generar metodo constructor y sobre escribir los metodos de la class heredada para poder Personanilzar el comportamiento del filtro de autenticacion	129
13.2.2.2.1	Metodo constructor	129
13.2.2.2.1.1	Pasar metodo constructor como atributo	131
13.2.2.2.2	Sobre escribir los métodos que se requieren del class heredada	133
13.2.2.2.2.1	Meotdo para realizar autenticacion	133
13.2.2.2.2.2	Meotdo si todo sale bien en la autenticacion	135
13.2.2.2.2.3	Meotdo si todo sale malen la autenticacion	137
13.3	Añadir Configuracion al metodo filterChain: Añadir filtro personalizado para manejar autenticacion utilizando JWT	139
13.3.1	Inyectar class AuthenticationConfiguration	139
13.3.2	Agregar filtro personalizado de autenticacion al metodo	141
13.4.1	Modificar e Implementar el metodo de atutenticacion para realizar autenticacion	143
13.4.1.1	Declarar variables locales	144
13.4.1.2	Deserialización del Usuario	145
13.4.1.3	Creardel Token de Autenticación	148
13.4.1.4	Autenticación del token a través del AuthenticationManager	149
13.5	Modificar e Implementar el metodo succesfulAuthentication	150
13.5.1	Obtener el usuario	151
13.5.2	Generación del TokenObtener el usuario	152
13.5.3	Agregar el token al encabezado de la respuesta	153
13.5.4	Creación del cuerpo de la respuesta	155
13.5.5	Escribir la respuesta en formato JSON	156
13.5.6	Diagrama visual dem metodo	157
13.5.6	Modificar e Implementar el metodo unsuccesfulAuthentication	158
13.6.1	Cuerpo de la respuesta JSON	159
13.7	Crear service que implemente la interface UserDetailsService	162
13.7.1	Crear que la class que va implementar la interface UserDetailsService	162
13.7.2	Decirle a la class que va ser un service e implemnetarla interface UserDetailsService	163

13.7.3	Implementar el Método loadUserByUsername	164
13.7.4	Validar del Usuario	165
13.7.5	Validar del Usuario	166
13.7.6	Creación del Objeto User	168
13.7.7	Diagrama visual	169
13.8	PassordEncoder y Passwor encriptado con Bcrypt. Pruebas con postman	170
13.9	Implementando Filtro para validar el token JwtValidationFilter	176
13.9.1	Crear class para crear filtro para la validacion del token	176
13.9.2	Heredar la class BasicAuthenticationFilter	177
13.9.3	Metodo constructor de la class heredada BasicAuthenticationFilter	178
13.9.4	Implementar filtrocon el Método doFilterInternal	179
13.9.5	Obtención del Header	181
13.9.6	Verificación del Prefijo Bearer	182
13.9.7	Decodificación del Token	184
13.9.8	Validación del Secret	186
13.9.9	Respuesta en Caso de Token	189
13.9.10	Flujo General y Consideraciones	191
13.10	Constantes y Registrando JwtValidationFilter en la configuración Spring Security	192
13.11	Encriptar password users db	196
13.11.1	Reiniciar tabla de datos	197
13.12	JWT	200
13.12.1	Generar y firmar token JWT modificar JwtAuthenticationFilter	206
14	Roles	209
14.1	Crear class role	209
14.1.1	Configurar la class role	210
14.1.2	Relacionar la class de los roles con la class de los usuarios	211
14.1.3	Configurando el mapeo de roles y esquema de tablas	212
14.1.4	Asignando el role al usuario al crear el usuario en POST	214
14.1.5	Obteniendo los roles desde la base de datos en UserDetailsService	219
14.1.6	gregando roles en los claims cuando se genera el token en filtro Authentication	220
14.1.7	Obteniendo los roles desde los claims en el filtro JwtValidationFilter	221
14.1.8	Implementando clase MixIn para que funcione el punto anterior	222

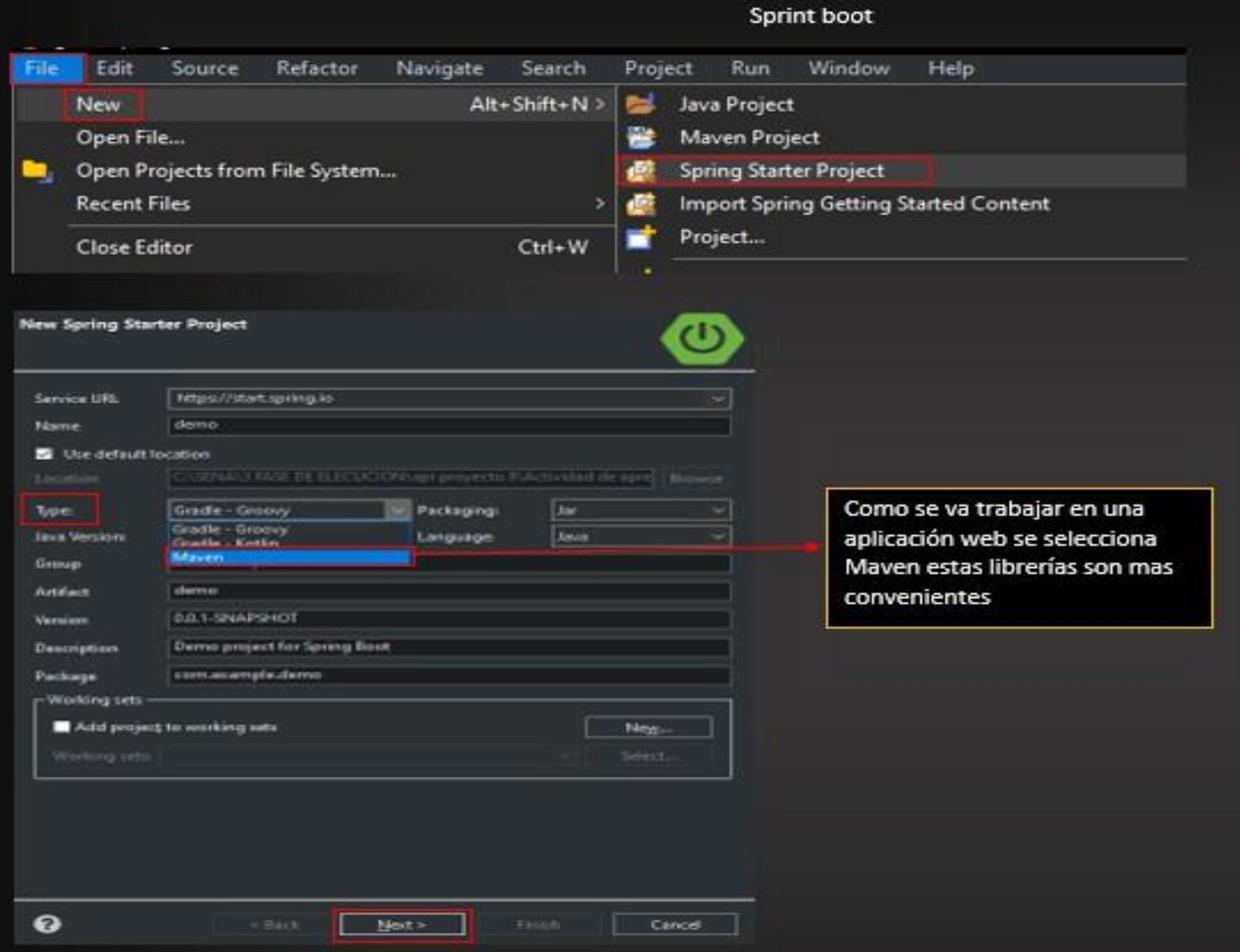
14.2	Añadiendo reglas de acceso y autorización para los roles	225
14.2.1	La clase DTO	226
14.2.2	La clase DtoMapperUser (DtoMapperBuilder)	227
14.2.3	Implementando el DTO en el Service y Controller	228
14.3	Agregar rol admin al backend	236
14.3.1	Crear atributo admin en los modelos que utilizan los metodos pots	236
14.3.1.1	Modelo UserRequest	236
14.3.1.2	Modelo user	237
14.3.1.3	Modelo UserDto	238
14.3.1.4	Modelo DtoMapper class helper	239
14.3.2	Agregar los rol admin al service del backend	241
14.3.2.1	agregar rol admin al Metodo save del service(userService)	241
14.3.2.2	agregar rol admin al Metodo update del service(userService)	243
14.4	Configurando Cors de Spring Security	246
14.2.1	Definición del método y anotación	246
14.4.2	Creación de la configuración básica de CORS	247
14.4.3	Configuración de los orígenes permitidos	248
14.4.4	Configuración de los métodos HTTP	249
14.4.5	Configuración de las cabeceras permitidas	250
14.4.6	Permitir credenciales (cookies y headers de autenticación)	251
14.4.7	Aplicar la configuración a todas las rutas	252
14.4.8	Retornar el CorsConfigurationSource	253
14.4.9	resumen breve	254
14.4.10	Agregar el metodo al Metodo filterchain	255
14.4.11	Crear filtro CORS en Spring Boot	257
14.5	Implementar login con axios	261
14.5.1	Modificar la funcion loginUser del authService. Para autenticar el usuario enviando credenciales al servidor	261
14.5.2	funcion handlerLogin del hook custom useAuth. Para el inicio de sesion	263
14.5.2.1	Proceso dentro de handlerLogin. Bloque try catch	264
14.5.2.2	Proceso dentro de handlerLogin. Llamada al Backend	265
14.5.2.3	Proceso dentro de handlerLogin.extraer Token y Usuario:	266
14.5.2.4	Proceso dentro de handlerLogin.Decodificar la parte de claims del token usando atob	267

14.5.2.5	Proceso dentro de handlerLogin. Actualización del Estado	268
14.5.2.6	Proceso dentro de handlerLogin. Almacenar el estado de inicio de sesión en sessionStorage	269
14.5.2.7	Proceso dentro de handlerLogin. Pasar el rol en false como estado inicial	270
14.5.2.8	Proceso dentro de handlerLogin. Guardar el token en sessionStorage y Redirecciónar al panel de usuarios	271
14.5.2.9	Proceso dentro de handlerLogin. Redirección:	272
14.5.2.10	Proceso dentro de handlerLogin. Manejo de Errores (HTTP 401)	273
14.5.2.11	Proceso dentro de handlerLogin. Manejo de Errores (HTTP 403)	274
14.5.2.12	Proceso dentro de handlerLogin. Manejo de Errores (otros errores)	275
14.5.3	Modificar el hook custom loginReducer. para manejar el estado relacionado con el inicio y cierre de sesión	276
14.5.4	maneja operaciones CRUD (Crear, Actualizar y Eliminar) en el cliente, utilizando axios para realizar solicitudes HTTP y una configuración de encabezados que incluye un token de autenticación	277
15.5.4.1	crear funcion para configurar los encabezados HTTP	277
15.5.4.2	Pasar la funcion config a los metodos del CRUD (Pots)	278
15.6	Ocultado botones si no es admin	279
15.6.1	Entodos los componentes donde tengamos acciones que sean post los escondemos para los usuarios que no sean admin	279
14.7	Ocultar página si no es admin	282
14.7.1	Escondemos las páginas que no se requieran si no es admin	282
14.8	Logout JWT invalido	283
14.8.1	corregir el estatus en el backend	283
14.8.2	Lanzar el error la funcion remove del service (userService)	284
14.8.3	Manejar error 401 en el metodo agregar usuario handlerAddUsers (useusers)	285
14.8.4	Manejar error 401 en el metodo remove handlerRemove (useusers)	286
14.9	Agregando role admin en el Frontend React	287
14.9.1	Agregando rol admin a los metodo save y update del service (UserSrvice)	287
14.9.2	Agregando rol admin a al estado inicial del formulario en el hook Custom (userUsers)	288
14.9.3	Agregar el nuevo campo admin en el formulario (userForm)	289
14.9.3.1	Crear checkbox para habilitar y desabilitar rol admin	291
14.9.3.2	crear funcion onCheckboxOnchange	293

14.9.4	Agregar rol admin en el componente UserList	295
14.9.5	Agregar rol admin en el componente UserRow	296

Luis  
Fernando  
Agudelo  
Gutiérrez

# 1. Creacion del proyecto y sus dependencias



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

Simplifica la configuración necesaria para conectar tu aplicación Spring Boot con una base de datos, manejar entidades y realizar operaciones CRUD (Crear, Leer, Actualizar, Borrar) de manera más fácil.

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

Este starter es particularmente útil en entornos de producción para asegurarte de que tu aplicación esté funcionando correctamente y para diagnosticar problemas rápidamente.

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

es esencial para cualquier proyecto que necesite exponer servicios RESTful o manejar solicitudes HTTP en una aplicación web.

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.27</version>
</dependency>
```

Esta dependencia agrega el conector JDBC para MySQL a tu proyecto

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

Recarga Automática: Permite la recarga automática de la aplicación cuando se detectan cambios en el código.

Configuración Simplicidad: Facilita la configuración para el desarrollo, por ejemplo, deshabilitando la caché de plantillas.

Consola Remota: Proporciona una consola remota para depuración.

```
<dependency>
    <groupId>com.microsoft.sqlserver</groupId>
    <artifactId>mssql-jdbc</artifactId>
    <scope>runtime</scope>
</dependency>
```

Este es el controlador JDBC proporcionado por Microsoft para conectarse a bases de datos SQL Server desde aplicaciones Java.

```
<dependency>
    <groupId>org.mariadb.jdbc</groupId>
    <artifactId>mariadb-java-client</artifactId>
    <scope>runtime</scope>
</dependency>
```

Este es el controlador JDBC proporcionado por MariaDB para conectarse a bases de datos MariaDB desde aplicaciones Java.

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

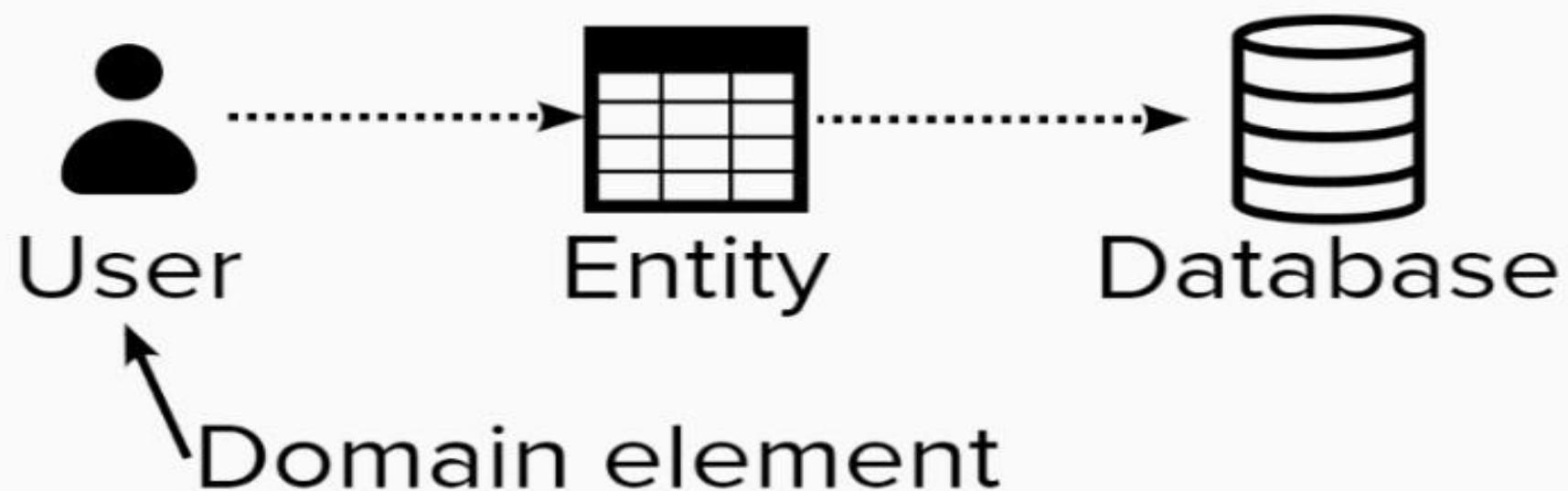
JUnit 5: Framework de pruebas unitarias.  
Spring Test & Spring Boot Test: Herramientas de prueba para aplicaciones Spring y Spring Boot.  
Mockito: Framework de mocking para pruebas unitarias.  
Hamcrest: Librería de matchers para hacer afirmaciones en pruebas.

## 2. Modelo o Entidad (Entity)

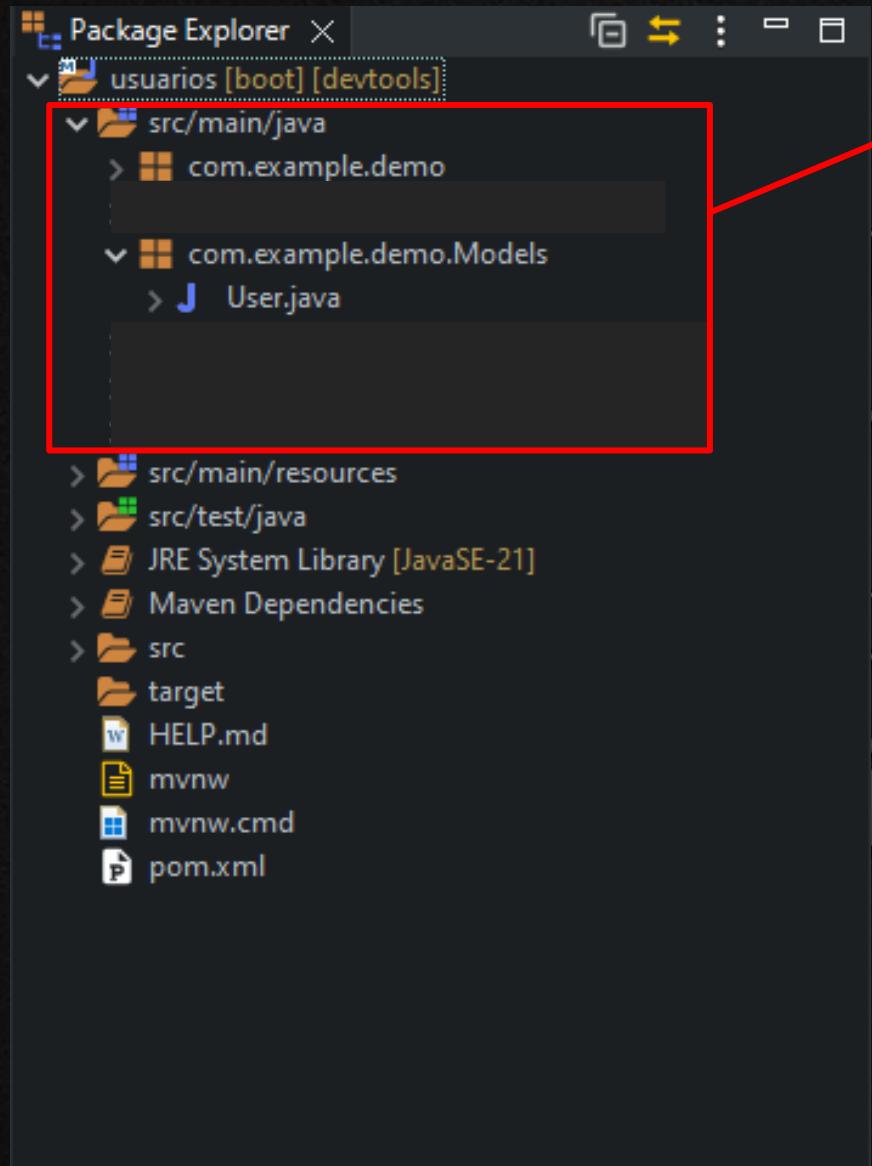
### 2.1 Explicacion de una Entity

#### Que es una entidad para Spring Boot

Una entidad es un objeto, elemento o ‘cosa’ con atributos particulares que lo distinguen. Por ejemplo, este podría ser un ‘user (usuario)’ sobre el que necesitamos conocer sus atributos como el nombre, edad, email, etc.



## 2.2 Creacion del modelo o Entity



1. **Crear un paquete:**
  - Dentro del paquete donde está la clase principal con el método `main`.
2. **Crear una clase:**
  - Dentro de ese paquete, crear una clase que será la Entidad (Entity).
  - Esta clase representará una tabla en la base de datos (BBDD).

## 2.3 Configuración de un Modelo o Entity

```
J User.java X
1 package com.example.demo.Models;
2
3
4 import jakarta.persistence.Entity;
5
6
7
8 import jakarta.persistence.Table;
9
10 @Entity
11 @Table(name="users")
12 public class User {
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51 }
52
```

Se importa la librería

Se importa la librería

Anotación `@Entity` para decirle que esta class va ser una entidad

Esta anotación especifica que la entidad User se mapeará a una tabla llamada "users" en la BBDD.

Al usar `@Table`, puedes personalizar varios aspectos del mapeo de la tabla, como el nombre de la tabla, el esquema, los índices, entre otros.

### User.java X

```
1 package com.example.demo.Models;
2 import jakarta.persistence.Column;
3 import jakarta.persistence.Entity;
4 import jakarta.persistence.GeneratedValue;
5 import jakarta.persistence.GenerationType;
6 import jakarta.persistence.Id;
7 import jakarta.persistence.Table;
8
9 @Entity
10 @Table(name="users")
11 public class User {
12
13     @Id
14     @GeneratedValue(strategy = GenerationType.IDENTITY)
15     private Long Id;
16
17     @Column(unique = true)
18     private String Username;
19
20     private String Password;
21
22     @Column(unique = true)
23     private String Email;
24
25     public Long getId() {}
26     public void setId(Long id) {}
27     public String getUsername() {}
28     public void setUsername(String username) {}
29     public String getPassword() {}
30     public void setPassword(String password) {}
31     public String getEmail() {}
32     public void setEmail(String email) {}
33
34 }
35
36 }
```

Se importa la librería

Esta anotación se utiliza para definir el campo id como la clave primaria de la entidad.

Esta anotación se utiliza para especificar la estrategia de generación de valores para el campo id. En este caso, GenerationType.IDENTITY indica que el valor del campo id se generará automáticamente por la base de datos utilizando una columna de incremento

Esta anotación asegura que los valores en la columna username sean únicos en la tabla users de la base de datos. Esto significa que no puede haber dos registros con el mismo valor de username.

Se crean los métodos getter y setter

## 3. Repository

### 3.1 Explicacion

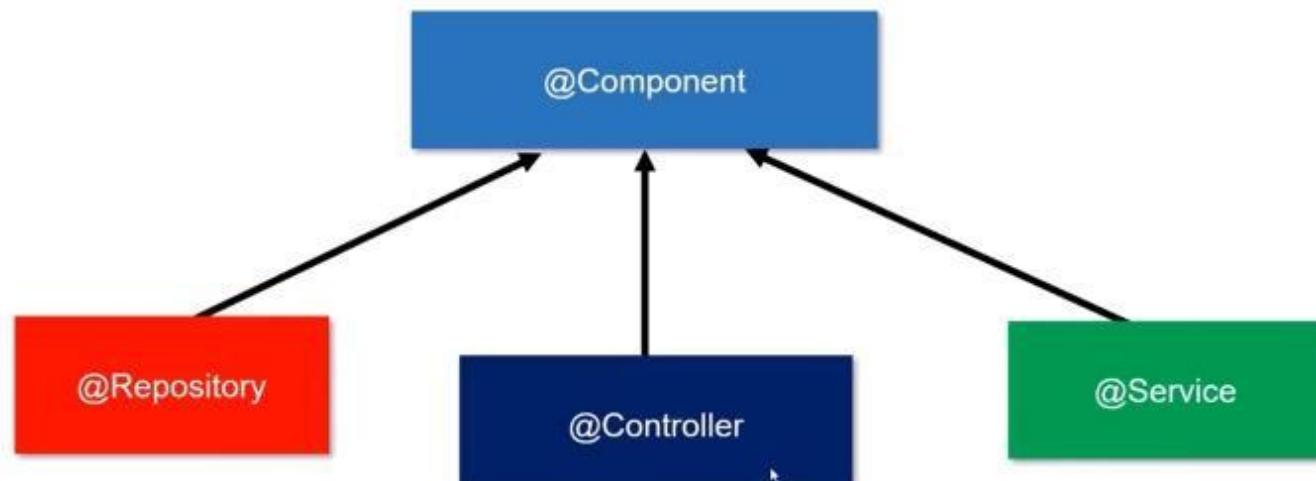
@Repository es una anotación(@) que se les llama estereotipos. Hija de la anotación

@Component

@Repository Es el que interactúa con la Base de datos

## Estereotipos

Spring define un conjunto de anotaciones core que categorizan cada uno de los componentes asociandoles una responsabilidad concreta .



## 3.2 Creación del @Repository

The screenshot shows the Eclipse IDE interface. On the left, the Package Explorer view displays the project structure under the package `com.example.demo.Repositories`. A red box highlights this package. On the right, the code editor shows the file `UserRepository.java` with the following content:

```
1 package com.example.demo.Repositories;
2
3 import org.springframework.data.repository.CrudRepository;
4
5
6 public interface UserRepository {
```

A red arrow points from the highlighted package in the Package Explorer to the code editor, indicating the step of creating the package. To the right of the code editor, a callout box contains the following steps:

1. **\*\*Crear un paquete: \*\***
  - Dentro del paquete donde está la clase principal con el método `main`.
2. **\*\*Crear una Interface: \*\***
  - Dentro de ese paquete, crear una Interface.

### 3.3 Configuración del @Repository

Creamos nuestro repository con las operaciones CRUD para nuestra entidad.

Las podemos utilizar desde nuestro Service

```
J Cliente_interface.java X
1 package com.aplicacion_inventario_Interfaces;
2
3 import org.springframework.data.repository.CrudRepository;
4
5 import com.aplicacion_inventario_Modelos.Cliente;
6 @Repository
7 public interface UserRepository extends CrudRepository<User, Long> {
8
9
10
11 }
12
```

Se le dice que va ser un repositorio

extendemos de CrudRepository

CrudRepository es una interfaz genérica que recibe dos tipos. El primero es la clase(modelo) que esta interfaz manejará y el segundo es el tipo de dato del ID de la entidad.

### 3.4 Métodos del CrudRepository

Los métodos que nos provee CrudRepository son:

- save: guarda una entidad
- saveAll: guarda las entidades de una lista iterable
- findById: busca por el identificador
- existsById: verifica si existe un identificador
- findAll: devuelve todos los elementos para la entidad
- findAllById: busca todos los elementos que tengan el identificador
- count: devuelve el total de registros de la entidad
- deleteById: elimina un registro para el identificador
- delete: elimina la entidad
- deleteAllById: elimina todos los elementos que correspondan con el id
- deleteAll(Iterable): elimina todos los elementos que se reciban en el parámetro
- deleteAll(): elimina todos los elementos

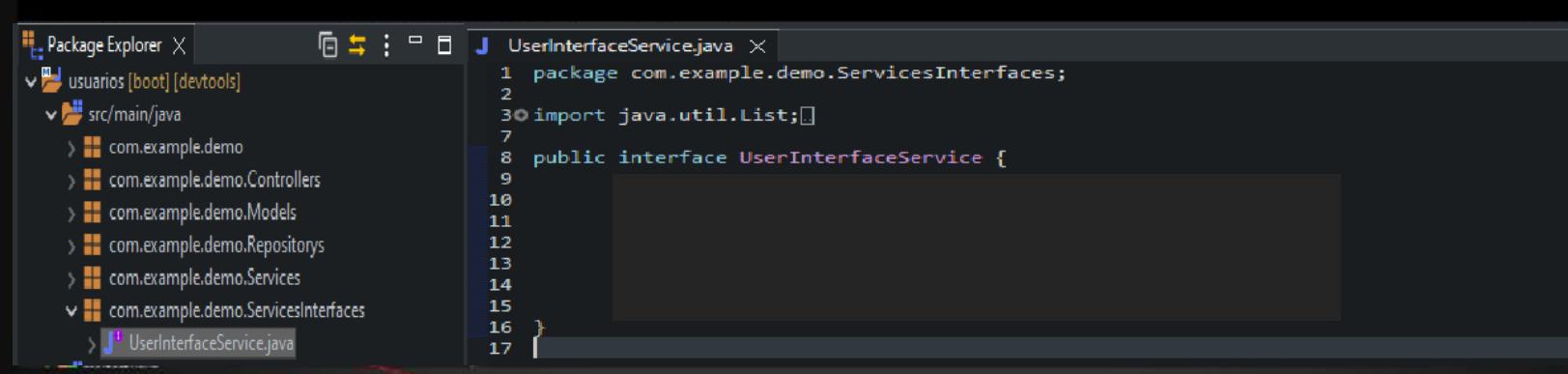
## 4 interface Service

### 4.1 Explicación Interface

Esta interface se crea con el fin de implementar los métodos del Crud. Esta interface Service es una manera de trabajar no se encuentra en libros. Es una opción para no trabajar los métodos directamente en el @Repository, aunque mas adelante se relaciona con el @Repository en el estereotipo @Service

## 4.2 Creación Interface

*Acaban todos los métodos para hacer el CRUD*



```
1 package com.example.demo.ServicesInterfaces;
2
3 import java.util.List;
4
5 public interface UserInterfaceService { }
```

Se crea una interface que va controlar los métodos para  
hacer el CRUD

Que es un Crud:

CRUD es un acrónimo que refiere a las cuatro funciones mínimas sobre una entidad -> Create, Read, Update, Delete.

Para Listar o crear se utiliza una List<>

## 5 métodos del CRUD

### 5.1 Método Listar

```
J *UserInterfaceService.java X
1 package com.example.demo.ServicesInterfaces;
2
3 import java.util.List;
4
5 public interface UserInterfaceService {
6
7     List<User>findAll(); //Listar
8
9
10
11
12
13
14
15
16
17
18
19 }
20
```

**Descripción:** Este método es responsable de recuperar todos los usuarios de la base de datos.

**Retorno:** Devuelve una lista (List<User>) que contiene todos los objetos User almacenados en la base de datos..

## 5.2 Método Insertar

```
J *UserInterfaceService.java X
1 package com.example.demo.ServicesInterfaces;
2
3 import java.util.List;
4
5
6 public interface UserInterfaceService {
7
8     List<User>findAll(); //Listar
9
10
11
12
13
14     User save(User user); // guardar
15
16
17
18
19 }
20
```

**Descripción:** Este método guarda un nuevo usuario en la base de datos.

**Parámetro:** User user - El objeto User que se desea guardar.

**Retorno:** Devuelve el objeto User que fue guardado, incluyendo cualquier valor de ID generado automáticamente.

## 5.3 Método Borrar por id

```
J *UserInterfaceService.java X
1 package com.example.demo.ServicesInterfaces;
2
3 import java.util.List;
4
5 public interface UserInterfaceService {
6
7     List<User>findAll(); //Listar
8
9
10    User save(User user); // guardar
11
12
13
14    void remove(Long id); // eliminar
15
16
17
18 }
19
20 }
```

**Descripción:** Este método elimina un usuario de la base de datos utilizando su ID.

**Parámetro:** Long id - El ID del usuario que se desea eliminar.

**Retorno:** No retorna ningún valor.

## 5.4 Método consultar por id

```
J *UserInterfaceService.java X
1 package com.example.demo.ServicesInterfaces;
2
3 import java.util.List;
4
5 public interface UserInterfaceService {
6
7     List<User>findAll(); //Listar
8
9
10    Optional<User> findById(Long id); //consultar
11
12    User save(User user); // guardar
13
14
15
16
17
18    void remove(Long id); // eliminar
19 }
20
```

**Descripción:** Este método busca un usuario específico en la base de datos utilizando su ID.

**Parámetro:** Long id - El ID del usuario que se desea consultar.

**Retorno:** Devuelve un objeto Optional<User>, que puede contener el usuario encontrado o estar vacío si no se encuentra.

## 5.5 Método actualizar por id

```
J *UserInterfaceService.java X
1 package com.example.demo.ServicesInterfaces;
2
3 import java.util.List;
4
5 public interface UserInterfaceService {
6
7     List<User>findAll(); //Listar
8
9     Optional<User> findById(Long id); //consultar
10
11    User save(User user); // guardar
12
13    Optional<User> update(User user, Long id); // actualizar
14
15    void remove(Long id); // eliminar
16
17
18
19 }
20
```

**Descripción:** Este método actualiza un usuario existente en la base de datos con la información proporcionada.

**Parámetros:**

User user - El objeto User que contiene la nueva información.

Long id - El ID del usuario que se desea actualizar.

**Retorno:** Devuelve un objeto Optional<User>, que puede contener el usuario actualizado o estar vacío si el usuario no se encuentra.

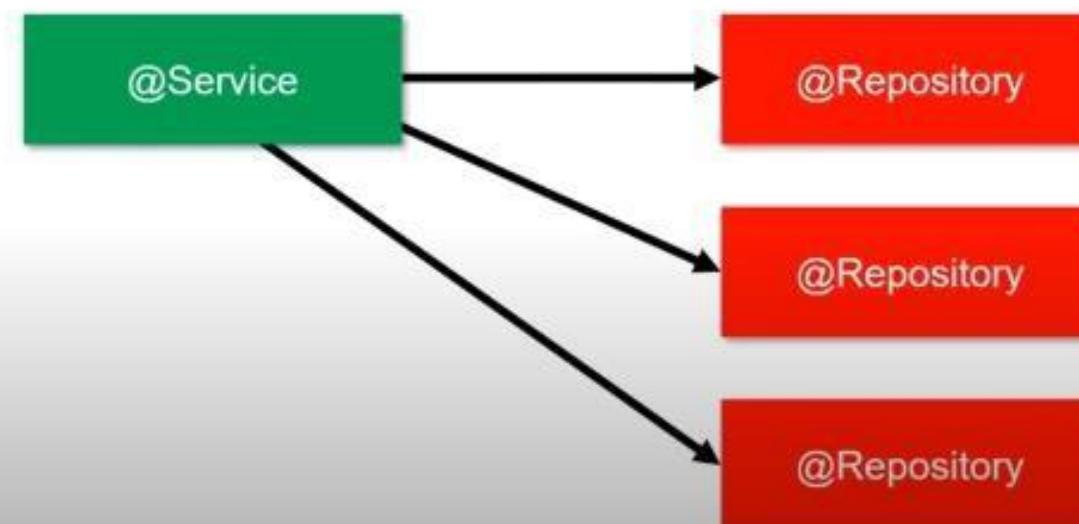
## 6. @Service

### 6.1 Explicación @Service

@Service hace las operaciones que necesitamos del negocio debe de llamar al @Repository que se encarga de guardar, rescatar la data de la base de datos

## @Service

Este estereotipo se encarga de gestionar las operaciones de negocio más importantes a nivel de la aplicación y aglutina llamadas a varios repositorios de forma simultánea. Su tarea fundamental es la de agregador implementa el patrón fachada y aglutina varios repositorios y llamadas a otros servicios.



## 6.2 Creación @Service

*SE crea el servicio es una class*

The screenshot shows the Eclipse IDE interface. On the left, the Package Explorer view displays a project structure under 'usuarios [boot] [devtools]'. The 'src/main/java' folder contains several packages: 'com.example.demo', 'com.example.demo.Controllers', 'com.example.demo.Models', 'com.example.demo.Repositories', 'com.example.demo.Services' (which is expanded to show 'UserService.java'), and 'com.example.demo.ServicesInterfaces'. The central workspace shows a code editor for 'UserService.java' with the following content:

```
1 package com.example.demo.Services;
2
3
14
15
16 public class UserService {
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67 }
```

## 6.3 Configuración @Service

### 6.3.1 Implementar la Interface Service

```
J UserService.java X
1 package com.example.demo.Services;
2
3 import java.util.List;
4
5
6 public class UserService implements UserInterfaceService {
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27 public List<User> findAll() {
28
29
30
31
32
33
34 public Optional<User>findById(Long id) {
35
36
37
38
39
40
41 public User save(User user) {
42
43
44
45
46
47
48 public void remove(Long id) {
49
50
51
52
53
54
55 public Optional<User> update(User user, Long id) {
56
57
58
59
60
61
62
63
64
65
66
67 }
68 }
```

Se implementa la interface donde tenemos todos los métodos del crud

Al implementar una interface por obligación debemos sobre escribir todos los métodos que contenga la interface

### 6.3.2 Instanciar el @Repository

```
J UserService.java X
1 package com.example.demo.Services;
2
3 import java.util.List;
4
5
6 public class UserService implements UserInterfaceService {
7
8     @Autowired
9     private UserRepository repository;
10
11
12     public List<User> findAll() {
13
14
15     }
16
17     public Optional<User> findById(Long id) {
18
19
20     }
21
22     public User save(User user) {
23
24
25     }
26
27     public void remove(Long id) {
28
29
30     }
31
32     public Optional<User> update(User user, Long id) {
33
34
35     }
36
37     }
38
39
40 }
```

La anotación `@Autowired` se utiliza en Spring para injectar automáticamente las dependencias requeridas en una clase.

Creo una variable objeto de la interface principal de la que hereda de CrudRepository

### 6.3.3 Decirle que va ser un @Service

```
J UserService.java X
1 package com.example.demo.Services;
2
3 import java.util.List;
4
5
6
7
8
9
10
11
12
13
14
15 @Service
16 public class UserService implements UserInterfaceService {
17
18     @Autowired
19     private UserRepository repository;
20
21
22
23
24
25
26
27     public List<User> findAll() {
28
29
30
31
32
33
34     public Optional<User>findById(Long id) {
35
36
37
38
39
40
41     public User save(User user) {
42
43
44
45
46
47
48     public void remove(Long id) {
49
50
51
52
53
54
55     public Optional<User> update(User user, Long id) {
56
57
58 }
```

Anotación para decirle a la class que va ser un Service

#### 6.3.4 Método Listar del Crud

\*UserService.java

```
1 package com.example.demo.Services;
2
3 import java.util.List;
4
5 @Service
6 public class UserService implements UserInterfaceService {
7
8     @Autowired
9     private UserRepository repository;
10
11     @Override
12     @Transactional (readOnly = true)
13
14     public List<User> findAll() {
15
16         return (List<User>) repository.findAll();
17     }
18
19 }
```

Se está sobre escribiendo el metodo

se utiliza para gestionar transacciones en la base de datos.  
El atributo readOnly = true indica que esta transacción es de solo lectura, lo que significa que no se realizarán modificaciones en la base de datos dentro de este método.

se utiliza para asegurarse de que el resultado sea tratado como una lista de objetos User

Nombre de la instancia de la interface UserRepository(repository), para poder utilizar los métodos del crud por sprint boot

Se esta llamando al metodo findAll() de CrudRepository que se heredó de UserRepository

### 6.3.5 Método guardar del Crud

```
J *UserService.java X
1 package com.example.demo.Services;
2
3 import java.util.List;
4
5 @Service
6 public class UserService implements UserInterfaceService {
7
8     @Autowired
9     private UserRepository repository;
10
11     public List<User> findAll() {
12
13
14     }
15
16     public User save(User user) {
17
18         return repository.save(user);
19     }
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37     @Override
38     @Transactional
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66 }
```

Se está sobre escribiendo el método

Una transacción es un conjunto de operaciones que se ejecutan como una unidad atómica: o se completan todas con éxito, o ninguna de ellas se aplica.  
En este caso, si ocurre algún error durante la ejecución del método, todos los cambios realizados en la base de datos dentro de esta transacción serán revertidos automáticamente.

Guarda un nuevo usuario en la base de datos y lo devuelve.

Es el parámetro que recibe el método. En este caso, recibe un objeto del tipo User, que contiene información que será almacenada en la base de datos.

## 6.3.6 Método Borrar del Crud

```
*UserService.java ×  
1 package com.example.demo.Services;  
2  
3 import java.util.List;  
4  
5 @Service  
6 public class UserService implements UserInterfaceService {  
7  
8     @Autowired  
9     private UserRepository repository;  
10  
11     public List<User> findAll() {}  
12  
13     public Optional<User>findById(Long id) {}  
14  
15     public User save(User user) {}  
16  
17     @Override  
18     @Transactional  
19     public void remove(Long id) {  
20         repository.deleteById(id);  
21     }  
22  
23 }
```

Se está sobre escribiendo el metodo

Una transacción es un conjunto de operaciones que se ejecutan como una unidad atómica: o se completan todas con éxito, o ninguna de ellas se aplica.

En este caso, si ocurre algún error durante la ejecución del método, todos los cambios realizados en la base de datos dentro de esta transacción serán revertidos automáticamente.

El parámetro id (de tipo Long) se utiliza para identificar el registro que se desea eliminar.

Elimina un usuario de la base de datos por su ID. Aquí se está llamando al método deleteById del objeto repository.

## 6.3.7 Método Actualizar del Crud por id

```
J *UserService.java X
1 package com.example.demo.Services;
2
3 import java.util.List;
4
5
6 @Service
7 public class UserService implements UserInterfaceService {
8
9     @Autowired
10    private UserRepository repository;
11
12
13    public List<User> findAll() {
14
15
16    }
17
18    public Optional<User> findById(Long id) {
19
20
21    }
22
23    public User save(User user) {
24
25
26    }
27
28    public void remove(Long id) {
29
30
31    }
32
33
34    @Override
35    @Transactional
36    public Optional<User> update(User user, Long id) {
37        // TODO Auto-generated method stub
38        Optional<User> o= this.findById(id);
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65 }
```

Se está sobre escribiendo el metodo

Una transacción es un conjunto de operaciones que se ejecutan como una unidad atómica: o se completan todas con éxito, o ninguna de ellas se aplica.

En este caso, si ocurre algún error durante la ejecución del método, todos los cambios realizados en la base de datos dentro de esta transacción serán revertidos automáticamente.

Recibe dos parámetros:

user: Un objeto de tipo User que contiene los nuevos datos que se desean actualizar.

id: El identificador único del usuario que se desea actualizar.

Aquí se llama a un método llamado findById de CrudRepository que se heredó de UserRepository, que busca un usuario en la base de datos utilizando su id. Este método devuelve un Optional<User>:  
Si encuentra al usuario, el Optional contendrá el objeto User.  
Si no lo encuentra, el Optional estará vacío.

```
J *UserService.java ×
1 package com.example.demo.Services;
2
3 import java.util.List;
4
5 @Service
6 public class UserService implements UserInterfaceService {
7
8     @Autowired
9     private UserRepository repository;
10
11     public List<User> finAll() {
12
13     }
14
15     public Optional<User>findById(Long id) {
16
17         return repository.findById(id);
18     }
19
20     public User save(User user) {
21
22         return repository.save(user);
23     }
24
25     public void remove(Long id) {
26
27         repository.deleteById(id);
28     }
29
30     @Override
31     @Transactional
32     public Optional<User> update(User user, Long id) {
33
34         // TODO Auto-generated method stub
35         Optional<User> o= this.findById(id);
36
37         if(o.isPresent()) {
38
39             User u = o.get();
40
41             u.setName(user.getName());
42             u.setAge(user.getAge());
43
44             repository.save(u);
45         }
46
47         return o;
48     }
49
50     public List<User> findByName(String name) {
51
52         return repository.findByName(name);
53     }
54
55     public List<User> findByAge(int age) {
56
57         return repository.findByAge(age);
58     }
59
60     public List<User> findAll() {
61
62         return repository.findAll();
63     }
64
65 }
```

if (o.isPresent()) {  
Se verifica si el Optional<User> contiene un valor, es decir, si el usuario con el ID proporcionado existe en la base de datos.

```

J *UserService.java ×
1 package com.example.demo.Services;
2
3 import java.util.List;
4
5 @Service
6 public class UserService implements UserInterfaceService {
7
8     @Autowired
9     private UserRepository repository;
10
11     public List<User> findAll() {
12
13     }
14
15     public Optional<User> findById(Long id) {
16
17         return Optional.ofNullable(repository.findById(id));
18     }
19
20     public User save(User user) {
21
22         return repository.save(user);
23     }
24
25     public void remove(Long id) {
26
27         repository.deleteById(id);
28     }
29
30     @Override
31     @Transactional
32     public Optional<User> update(User user, Long id) {
33
34         // TODO Auto-generated method stub
35         Optional<User> o= this.findById(id);
36
37         if(o.isPresent()) {
38
39             User UserDb= o.orElseThrow();
40
41             UserDb.setUsername(user.getUsername());
42
43             UserDb.setEmail(user.getEmail());
44
45             return Optional.of(this.save(UserDb));
46         }
47
48         return Optional.empty();
49     }
50
51 }
52
53
54
55
56
57
58
59
60
61
62
63
64
65 }

```

Si no se encuentra un usuario con el ID proporcionado (isPresent() es falso), el método devuelve un Optional.empty(), indicando que no se realizó ninguna actualización porque el usuario no existe.

se obtiene el objeto User contenido en el Optional utilizando el método orElseThrow(). Este método lanza una excepción si el Optional está vacío, pero en este caso no ocurrirá porque ya se verificó con isPresent().

Luego, se actualizan las propiedades del usuario encontrado (UserDb) con los datos del objeto user recibido como parámetro:  
setUsername(user.getUsername()): Actualiza el nombre de usuario.

Luego, se actualizan las propiedades del usuario encontrado (UserDb) con los datos del objeto user recibido como parámetro:  
setEmail(user.getEmail()): Actualiza el correo electrónico.

Después de actualizar los datos del usuario, se guarda el objeto actualizado en la base de datos utilizando un método llamado save.

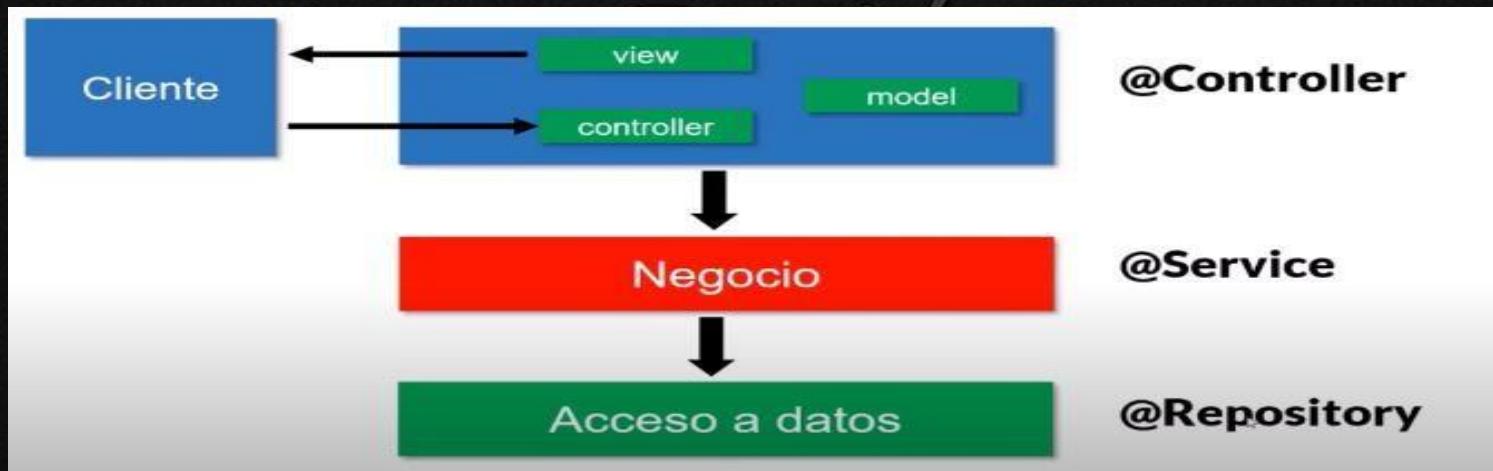
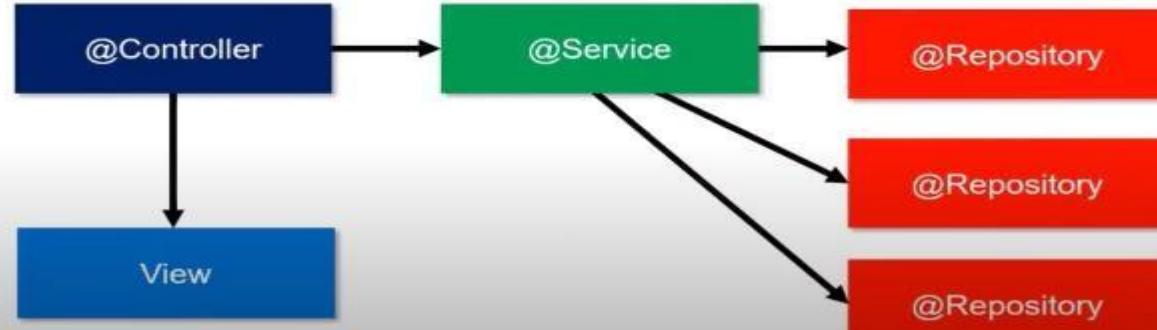
El resultado de la operación de guardado (el usuario actualizado) se envuelve en un Optional utilizando Optional.of() y se devuelve.

## 7. @Controller

### 7.1 Explicación @Controller

#### @Controller

El último de los estereotipos que es el que realiza las tareas de controlador y gestión de la comunicación entre el usuario y el aplicativo. Para ello se apoya habitualmente en algún motor de plantillas o librería de etiquetas que facilitan la creación de páginas.



## 7.2 Creación del @Controller

The screenshot shows a Java IDE interface with a dark theme. On the left, the Package Explorer view displays the project structure under the package `usuarios [boot] [devtools]`. A red dashed box highlights the `src/main/java/com.example.demo.Controllers` folder, which contains a file named `UserController.java`. An arrow points from this highlighted folder to the code editor on the right. The code editor shows the beginning of the `UserController.java` file:

```
1 package com.example.demo.Controllers;
2
3
19
20
21
22 public class UserController {
```

A callout box with a red border and a yellow arrow points to the `UserController.java` file in the code editor, containing the text:

Se crea una clase que va ser el controller. Se crea una por cada modelo

## 7.3 Decirle que va ser un @Controller y pasarle una ruta base

```
J UserController.java X
1 package com.example.demo.Controllers;
2
3 import java.util.List;
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19 @RestController
20 @RequestMapping("/users") //ruta la base
21 public class UserController {
22
23
24     @Autowired
25     private UserInterfaceService service;
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90 }
```

@RequestMapping("/users") define la ruta base para todos los métodos dentro de esta clase. Todas las rutas definidas dentro de esta clase comenzarán con /users.

## 7.4 Instanciar la Interface Service (donde están todos los métodos del crud)

```
J UserController.java X
1 package com.example.demo.Controllers;
2
3 import java.util.List;
19
20 @RestController
21 @RequestMapping("/users") //ruta la base
22 public class UserController {
23
24     @Autowired
25     private UserInterfaceService service;
26
28
32
34
39
40
42
52
54
60
63
66
68
75
77
89
90 }
```

## 7.5 Método Listar del @Controller

```
J UserController.java X
1 package com.example.demo.Controllers;
2
3 import java.util.List;
4
5
6 @RestController
7 @RequestMapping("/users") //ruta la base
8 public class UserController {
9
10
11     @Autowired
12     private UserInterfaceService service;
13
14     @GetMapping
15     public List<User> list(){
16         return service.findAll();
17     }
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34     • @GetMapping define que este método responderá a solicitudes HTTP GET.
35
36
37     • public List<User> list() es un método que devuelve una lista de objetos
38         User .
39
40
41
42     • return service.findAll(); llama al método findAll() del servicio
43         UserInterfaceService para obtener todos los usuarios y devolverlos.
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
79
80
81
82
83
84
85
86
87
88
89
89
90 }
```

## 7.6 Método Insertar del @Controller

```
J *UserController.java ×
30 import java.util.List;[]
19
20 @RestController
21 @RequestMapping("/users") //ruta la base
22 public class UserController []
23
24@  @Autowired
25     private UserInterfaceService service;
26
28@     public List<User> list(){}
32
34@     public ResponseEntity<User> guardar(@RequestBody User user) {
35         return ResponseEntity.status(HttpStatus.CREATED).body(service.save(user));
36     }
37
38@     public void eliminar(@PathVariable Long id) {
39         service.delete(id);
40     }
42@     public User editar(@PathVariable Long id, @RequestBody User user) {
43         return service.update(user);
44     }
45
46@     public User obtener(@PathVariable Long id) {
47         return service.findById(id);
48     }
49
50@     public void iniciarSesion(@RequestBody User user) {
51         service.iniciarSesion(user);
52     }
53
54@     public void cerrarSesion(@RequestBody User user) {
55         service.cerrarSesion(user);
56     }
57
58@     public void registrar(@RequestBody User user) {
59         service.register(user);
60     }
61@     @PostMapping // status por defecto es 200
62
63     public ResponseEntity<User> guardar(@RequestBody User user) {
64
65         return ResponseEntity.status(HttpStatus.CREATED).body(service.save(user));
66     }
67
68@     public void eliminar(@PathVariable Long id) {
69         service.delete(id);
70     }
71
72@     public User editar(@PathVariable Long id, @RequestBody User user) {
73         return service.update(user);
74     }
75
76@     public User obtener(@PathVariable Long id) {
77         return service.findById(id);
78     }
79
80@     public void iniciarSesion(@RequestBody User user) {
81         service.iniciarSesion(user);
82     }
83
84@     public void cerrarSesion(@RequestBody User user) {
85         service.cerrarSesion(user);
86     }
87
88@     public void registrar(@RequestBody User user) {
89         service.register(user);
90     }
91 }
```

## 7.6.1 Explicación de tallada del metodo insertar del controller

### Anotación `@PostMapping`:

```
@PostMapping // status por defecto es 200
```

- `@PostMapping`: Esta anotación indica que el método manejará solicitudes HTTP POST. Los métodos anotados con `@PostMapping` se utilizan típicamente para crear nuevos recursos en el servidor. Aunque el estado por defecto es 200 (OK), en este método cambiamos el estado de la respuesta a 201 (Created).

### método `guardar`:

```
public ResponseEntity<?> guardar(@RequestBody User user) {
```

- `public`: Este es un modificador de acceso que indica que el método es accesible desde cualquier otra clase.
- `ResponseEntity<?>`: `ResponseEntity` es una clase en Spring que representa una respuesta HTTP completa, incluyendo el cuerpo, los encabezados y el código de estado. El uso de `<?>` indica que el tipo de contenido del cuerpo de la respuesta puede ser cualquier cosa.
- `guardar`: Este es el nombre del método. La convención en Java para los nombres de métodos es utilizar verbos que describan la acción realizada.
- `@RequestBody User user`: Esta anotación indica que el objeto `User` se obtiene del cuerpo de la solicitud HTTP. Spring automáticamente convierte el JSON del cuerpo de la solicitud en un objeto `User`.

```
return ResponseEntity.status(HttpStatus.CREATED).body(service.save(user))
```

- `ResponseEntity.status(HttpStatus.CREATED)` : Esto crea un objeto `ResponseEntity` y establece el código de estado HTTP a 201 (Created), lo que indica que un nuevo recurso ha sido creado exitosamente en el servidor.
- `.body(service.save(user))` :
  - `service.save(user)` llama al método `save` del servicio `UserInterfaceService`. Este método guarda el objeto `User` en la base de datos o cualquier almacenamiento persistente que estés utilizando.
  - `.body(...)` establece el cuerpo de la respuesta HTTP con el objeto `User` que ha sido guardado. Esto permite que el cliente reciba los detalles del usuario que se acaba de crear.

Aguadero

En resumen, el método `guardar` recibe un objeto `User` desde el cuerpo de una solicitud HTTP POST, lo guarda utilizando el servicio `UserInterfaceService`, y devuelve una respuesta HTTP con un código de estado 201 (Created) junto con el usuario guardado. Este patrón es común en aplicaciones RESTful para operaciones de creación de recursos.

## 7.7 Método Consultar del @Controller por id

```
J *UserController.java X
30 import java.util.List;[]
19
20 @RestController
21 @RequestMapping("/users") //ruta la base
22 public class UserController {
23
24@  @Autowired
25     private UserInterfaceService service;
26
28@  public List<User> list(){}
32 |
34@  39
39
40
41@  @GetMapping("/{id}")
42     public ResponseEntity<?> show(@PathVariable Long id) {
43
44         Optional<User> userOptinal= service.findById(id);
45
46         if(userOptinal.isPresent()) {
47             return ResponseEntity.ok(userOptinal.orElseThrow());
48         }
49         return ResponseEntity.notFound().build(); // si no se encuentra muestra el error 404
50
51     }
52
54@  @PostMapping // status por defecto es 200 s[]
60
63@  public ResponseEntity<?> guardar(@RequestBody User user) {  []
67
69@  76
78@  90
91 \
```

## 7.1 Explicación del Método Consultar del @Controller por id

Anotación `@GetMapping("/{id}") :`

```
@GetMapping("/{id}")
```

`@GetMapping("/{id}")` : Esta anotación indica que el método manejará solicitudes HTTP GET. El `{id}` en la ruta es un parámetro de ruta que se utilizará para identificar el usuario específico que se quiere obtener. El patrón entre llaves `{}` permite capturar el valor de la URL y pasarlo al método como argumento.

método `show :`

```
public ResponseEntity<?> show(@PathVariable Long id) {
```

- `public` : Modificador de acceso que indica que el método es accesible desde cualquier otra clase.
- `ResponseEntity<?>` : `ResponseEntity` representa una respuesta HTTP completa, incluyendo el cuerpo, los encabezados y el código de estado. El uso de `<?>` indica que el tipo del cuerpo de la respuesta puede ser cualquier cosa.
- `show` : Nombre del método. Se usa un nombre descriptivo que indica que este método mostrará la información de un usuario específico.
- `@PathVariable Long id` : La anotación `@PathVariable` indica que el valor del parámetro `id` en la URL debe mapearse a este parámetro del método. `Long id` es el tipo y nombre del parámetro, que se espera sea un número entero largo.

### Cuerpo del método `show`:

```
Optional<User> userOptinal = service.findById(id);

if (userOptinal.isPresent()) {
    return ResponseEntity.ok(userOptinal.orElseThrow());
}
return ResponseEntity.notFound().build(); // si no se encuentra muestra un error
}
```

- `Optional<User> userOptinal = service.findById(id);`: Este código llama al método `findById` del servicio `UserInterfaceService`, que devuelve un `Optional<User>`. La clase `Optional` es una contenedora que puede contener un valor no nulo o estar vacía, ayudando a evitar `NullPointerExceptions`.
- `if (userOptinal.isPresent()) {`: Se verifica si el `Optional` contiene un valor.
  - `return ResponseEntity.ok(userOptinal.orElseThrow());`: Si el `Optional` contiene un valor, se retorna una respuesta HTTP 200 (OK) con el cuerpo de la respuesta siendo el usuario encontrado. El método `orElseThrow()` devuelve el valor contenido en el `Optional` o lanza una excepción si el `Optional` está vacío. En este caso, sabemos que no está vacío porque estamos dentro del bloque `if`.
- `return ResponseEntity.notFound().build();`: Si el `Optional` está vacío (es decir, el usuario no fue encontrado), se retorna una respuesta HTTP 404 (Not Found) utilizando el método `notFound()` de `ResponseEntity`, seguido de `build()` para construir la respuesta final.

En resumen, el método `show` busca un usuario por su `id` en la base de datos. Si el usuario es encontrado, retorna una respuesta HTTP 200 (OK) con el usuario. Si no es encontrado, retorna una respuesta HTTP 404 (Not Found). Este patrón es común en aplicaciones RESTful para manejar la recuperación de recursos individuales basados en su identificador único.

Luis  
Fernando  
Agudelo  
Gutiérrez

## 7.8 Método Actualizar del @Controller por id

```
J *UserController.java X
30 import java.util.List;[]
19
20 @RestController
21 @RequestMapping("/users") //ruta la base
22 public class UserController {
23
24@    @Autowired
25    private UserInterfaceService service;
26
28@    public List<User> list(){[]
32
34@    }
39
40
42@    public ResponseEntity<?> show(@PathVariable Long id) {[]
52
54@    @PostMapping // status por defecto es 200 s[]
60
63@    public ResponseEntity<?> guardar(@RequestBody User user) { []
67
68@        @PutMapping("/{id}")
69        public ResponseEntity<?> actualizar( @RequestBody User user, @PathVariable Long id){
70            Optional<User> o= service.update(user, id);
71            if(o.isPresent()) {
72                return ResponseEntity.status(HttpStatus.CREATED).body(o.orElseThrow());
73            }
74            return ResponseEntity.notFound().build() ;
75        }
76
78@    }
90
91    }
92 }
```

## 7.8.1 Explicación Método Actualizar del @Controller por id

Anotación `@PutMapping("/{id}")`:

```
@PutMapping("/{id}")
```

- `@PutMapping("/{id}")`: Esta anotación indica que el método manejará solicitudes HTTP PUT. El `{id}` en la ruta es un parámetro de ruta que se utilizará para identificar el recurso específico (usuario) que se quiere actualizar. El método HTTP PUT generalmente se utiliza para actualizar un recurso existente.

Firma del método `actualizar`:

```
public ResponseEntity<?> actualizar(@RequestBody User user, @PathVariable Long id) {
```

- `public`: Modificador de acceso que indica que el método es accesible desde cualquier otra clase.
- `ResponseEntity<?>`: `ResponseEntity` representa una respuesta HTTP completa, incluyendo el cuerpo, los encabezados y el código de estado. El uso de `<?>` indica que el tipo del cuerpo de la respuesta puede ser cualquier cosa.
- `actualizar`: Nombre del método. Se usa un nombre descriptivo que indica que este método actualizará la información de un usuario específico.
- `@RequestBody User user`: La anotación `@RequestBody` indica que el objeto `User` se obtiene del cuerpo de la solicitud HTTP. Spring automáticamente convierte el JSON del cuerpo de la solicitud en un objeto `User`.
- `@PathVariable Long id`: La anotación `@PathVariable` indica que el valor del parámetro `id` en la URL debe mapearse a este parámetro del método. `Long id` es el tipo y nombre del parámetro, que se espera sea un número entero largo.

## Cuerpo del método `actualizar`:

```
Optional<User> o = service.update(user, id);
if (o.isPresent()) {
    return ResponseEntity.status(HttpStatus.CREATED).body(o.orElseThrow());
}
return ResponseEntity.notFound().build();
}
```

- `Optional<User> o = service.update(user, id);`: Este código llama al método `update` del servicio `UserInterfaceService`, que devuelve un `Optional<User>`. La clase `Optional` es una contenedora que puede contener un valor no nulo o estar vacía. Este método está intentando actualizar un usuario existente basado en el `id` proporcionado.
- `if (o.isPresent()) {`: Se verifica si el `Optional` contiene un valor. Esta es una forma de manejar potenciales `NullPointerExceptions`.
  - `return ResponseEntity.status(HttpStatus.CREATED).body(o.orElseThrow());`: Si el `Optional` contiene un valor, se retorna una respuesta HTTP 201 (Created) con el cuerpo de la respuesta siendo el usuario actualizado. El método `orElseThrow()` devuelve el valor contenido en el `Optional` o lanza una excepción si el `Optional` está vacío. En este caso, sabemos que no está vacío porque estamos dentro del bloque `if`.
- `return ResponseEntity.notFound().build();`: Si el `Optional` está vacío (es decir, el usuario no fue encontrado o no se pudo actualizar), se retorna una respuesta HTTP 404 (Not Found) utilizando el método `notFound()` de `ResponseEntity`, seguido de `build()` para construir la respuesta final.

En resumen, el método `actualizar` intenta actualizar un usuario existente identificado por su `id`. Recibe un objeto `User` desde el cuerpo de una solicitud HTTP PUT, llama al método `update` del servicio para actualizar el usuario y devuelve una respuesta HTTP con un código de estado 201 (Created) junto con el usuario actualizado si la actualización es exitosa. Si no se encuentra el usuario o no se puede actualizar, devuelve una respuesta HTTP 404 (Not Found).

Fernando  
Agudelo  
Gutiérrez

## 7.9 Método Eliminar del @Controller por id

```
*UserController.java X
22  public class UserController {
23
24      @Autowired
25      private UserInterfaceService service;
26
28      public List<User> list(){}
29
30
31
32
33
34
35
36
37
38
39
40
41
42      public ResponseEntity<?> show(@PathVariable Long id) { }
43
44      @PostMapping // status por defecto es 200 s
45
46      public ResponseEntity<?> guardar(@RequestBody User user) { }
47
48
49      public ResponseEntity<?> actualizar( @RequestBody User user, @PathVariable Long id){}
50
51
52
53
54
55
56
57
58
59
59
60
61
62
63      public ResponseEntity<?> eliminar(@PathVariable Long id){
64
65
66
67
68
69
69
70
71
72
73
74
75
76
77      @DeleteMapping("/{id}")
78      public ResponseEntity<?> remove( @PathVariable Long id){
79
80
81          Optional<User> o=service.findById(id);
82          if(o.isPresent()) {
83              service.remove(id);
84
85              return ResponseEntity.noContent().build(); // noContent devuelve un 204
86          }
87          return ResponseEntity.notFound().build();
88
89      }
90
91 }
```

## 7.9.1 Explicación del Método Eliminar del @Controller por id

Anotación `@DeleteMapping("/{id}") :`

```
@DeleteMapping("/{id}")
```

- `@DeleteMapping("/{id}")` : Esta anotación indica que el método manejará solicitudes HTTP DELETE. El `{id}` en la ruta es un parámetro de ruta que se utilizará para identificar el recurso específico (usuario) que se quiere eliminar. El método HTTP DELETE generalmente se utiliza para eliminar un recurso existente.

Firma del método `remove :`

```
public ResponseEntity<?> remove(@PathVariable Long id) {
```

- `public` : Modificador de acceso que indica que el método es accesible desde cualquier otra clase.
- `ResponseEntity<?>` : `ResponseEntity` representa una respuesta HTTP completa, incluyendo el cuerpo, los encabezados y el código de estado. El uso de `<?>` indica que el tipo del cuerpo de la respuesta puede ser cualquier cosa.
- `remove` : Nombre del método. Se usa un nombre descriptivo que indica que este método eliminará un recurso específico.
- `@PathVariable Long id` : La anotación `@PathVariable` indica que el valor del parámetro `id` en la URL debe mapearse a este parámetro del método. `Long id` es el tipo y nombre del parámetro, que se espera sea un número entero largo.

### Cuerpo del método `remove`:

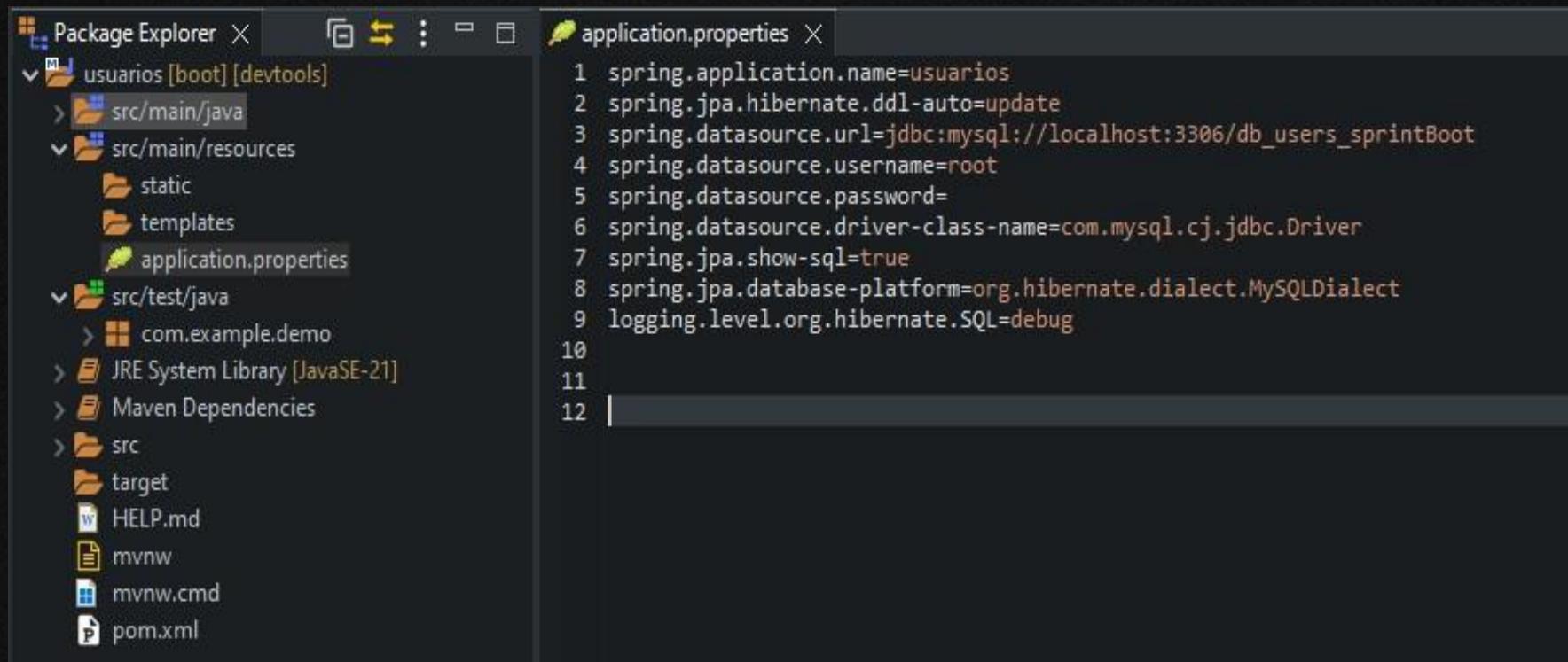
```
Optional<User> o = service.findById(id);
if (o.isPresent()) {
    service.remove(id);
    return ResponseEntity.noContent().build(); // noContent devuel
}
return ResponseEntity.notFound().build();
}
```

- `Optional<User> o = service.findById(id);`: Este código llama al método `findById` del servicio `UserInterfaceService`, que devuelve un `Optional<User>`. La clase `Optional` es una contenedora que puede contener un valor no nulo o estar vacía. Este método está intentando encontrar un usuario existente basado en el `id` proporcionado.
- `if (o.isPresent()) {`: Se verifica si el `Optional` contiene un valor. Esta es una forma de manejar potenciales `NullPointerExceptions`.
  - `service.remove(id);`: Si el `Optional` contiene un valor (es decir, el usuario fue encontrado), el método `remove` del servicio es llamado para eliminar el usuario basado en el `id`.
  - `return ResponseEntity.noContent().build();`: Se retorna una respuesta HTTP 204 (No Content) utilizando el método `noContent()` de `ResponseEntity`, seguido de `build()` para construir la respuesta final. El código de estado 204 indica que la solicitud fue exitosa y el servidor ha cumplido con la solicitud, pero no hay contenido que devolver.
- `return ResponseEntity.notFound().build();`: Si el `Optional` está vacío (es decir, el usuario no fue encontrado), se retorna una respuesta HTTP 404 (Not Found) utilizando el método `notFound()` de `ResponseEntity`, seguido de `build()` para construir la respuesta final. El código de estado 404 indica que el recurso solicitado no fue encontrado en el servidor.

En resumen, el método `remove` intenta encontrar y eliminar un usuario existente identificado por su `id`. Si el usuario es encontrado, se llama al método `remove` del servicio para eliminar el usuario y se devuelve una respuesta HTTP 204 (No Content). Si el usuario no es encontrado, se devuelve una respuesta HTTP 404 (Not Found).

Luis  
Fernando  
Agudelo  
Gutiérrez

## 8. Configuración BBDD (PROPERTIES)



The screenshot shows the Eclipse IDE interface. On the left, the Package Explorer view displays the project structure for 'usuarios [boot] [devtools]'. It includes 'src/main/java', 'src/main/resources' (containing 'static', 'templates', and 'application.properties'), 'src/test/java' (containing 'com.example.demo'), 'JRE System Library [JavaSE-11]', 'Maven Dependencies', 'src', 'target', 'HELP.md', 'mvnw', 'mvnw.cmd', and 'pom.xml'. On the right, the editor view shows the 'application.properties' file with the following content:

```
1 spring.application.name=usuarios
2 spring.jpa.hibernate.ddl-auto=update
3 spring.datasource.url=jdbc:mysql://localhost:3306/db_users_sprintBoot
4 spring.datasource.username=root
5 spring.datasource.password=
6 spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
7 spring.jpa.show-sql=true
8 spring.jpa.database-platform=org.hibernate.dialect.MySQLDialect
9 logging.level.org.hibernate.SQL=debug
10
11
12 |
```

Gutierrez

## 9. Pruebas con Postman

### 9.1 Prueba insertar

POST localhost:8080/users X +

localhost:8080/users

POST localhost:8080/users

Save Share

Params Authorization Headers (8) Body Scripts Settings Cookies

none form-data x-www-form-urlencoded raw binary GraphQL JSON Beautify

```
1 {  
2   "username": "Luis",  
3   "email": "toomy540@gmail.com",  
4   "password": "auc"  
5 }  
6
```

Send

Response History



## 9.2 Prueba Listar

The screenshot shows the Postman interface with the following details:

- Request Method:** GET (highlighted with a red dashed box)
- Request URL:** localhost:8080/users
- Headers:** (6 items listed)
- Body:** (highlighted with a red dashed box) - Note: This request does not have a body.
- Buttons:** Save, Share, Send (highlighted with a red dashed box)
- Params:** none, form-data, x-www-form-urlencoded, raw, binary, GraphQL
- Status:** 200 OK (highlighted with a green box)
- Time:** 1.27 s
- Size:** 242 B
- Visualize:** JSON (highlighted with a red dashed box) showing the response data:

```
[{"id": 1, "password": "12345", "email": "luisfer540@gmail.com", "username": "ffff"}]
```

### 9.3 Prueba Actualizar por id

The screenshot shows a POSTMAN interface with the following details:

- Method:** PUT
- URL:** `localhost:8080/users/1`
- Body:** Raw JSON (selected)
- Request Body:**

```
1 {  
2     "id": 1,  
3     "password": "12345",  
4     "email": "luisfer540@gmail.com",  
5     "username": "ffferchoff"  
6 }  
7
```

- Response:** 201 Created
- Preview:** Shows the updated user object with the same fields and values as the request body.

## 9.4 Prueba Eliminar por id

localhost:8080/users/1

Save Share

DELETE localhost:8080/users/1

Params Authorization Headers (6) Body Scripts Settings

none  form-data  x-www-form-urlencoded  raw  binary  GraphQL

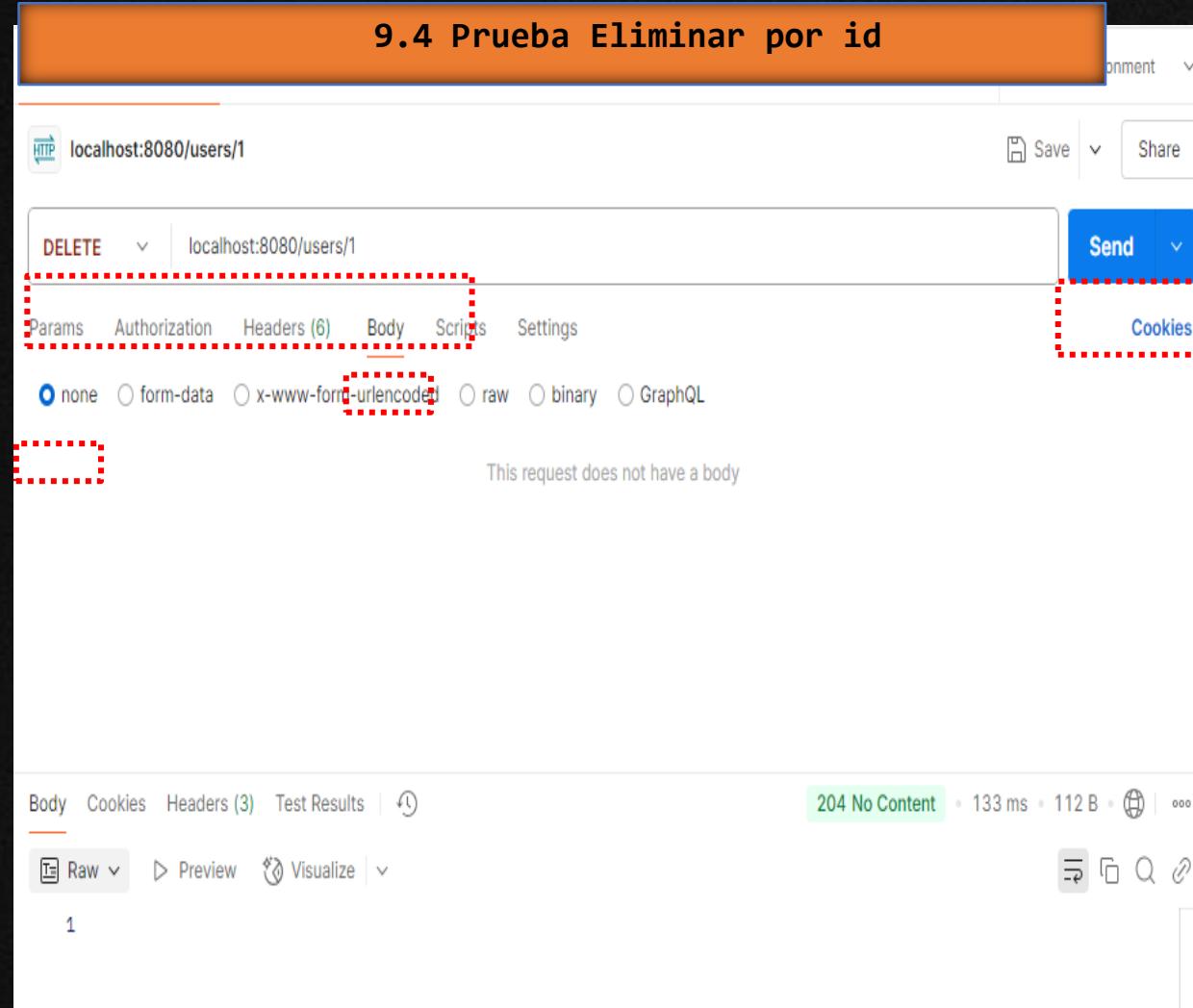
This request does not have a body

Body Cookies Headers (3) Test Results

204 No Content • 133 ms • 112 B

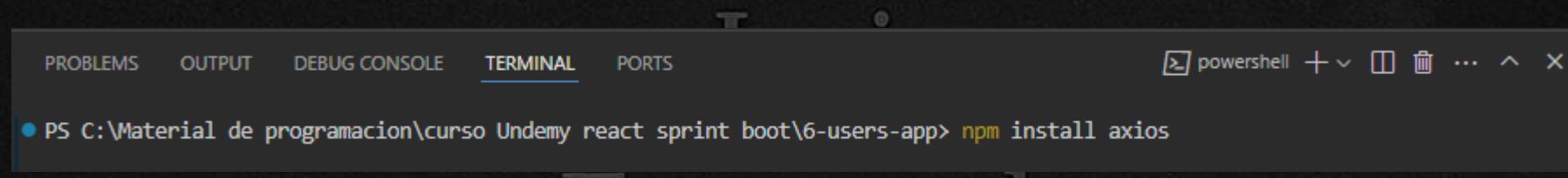
Raw Preview Visualize

1



## 10. Conectar react con sprint boot Con Axios

### 10.1 Instalar Axios en el frontend react

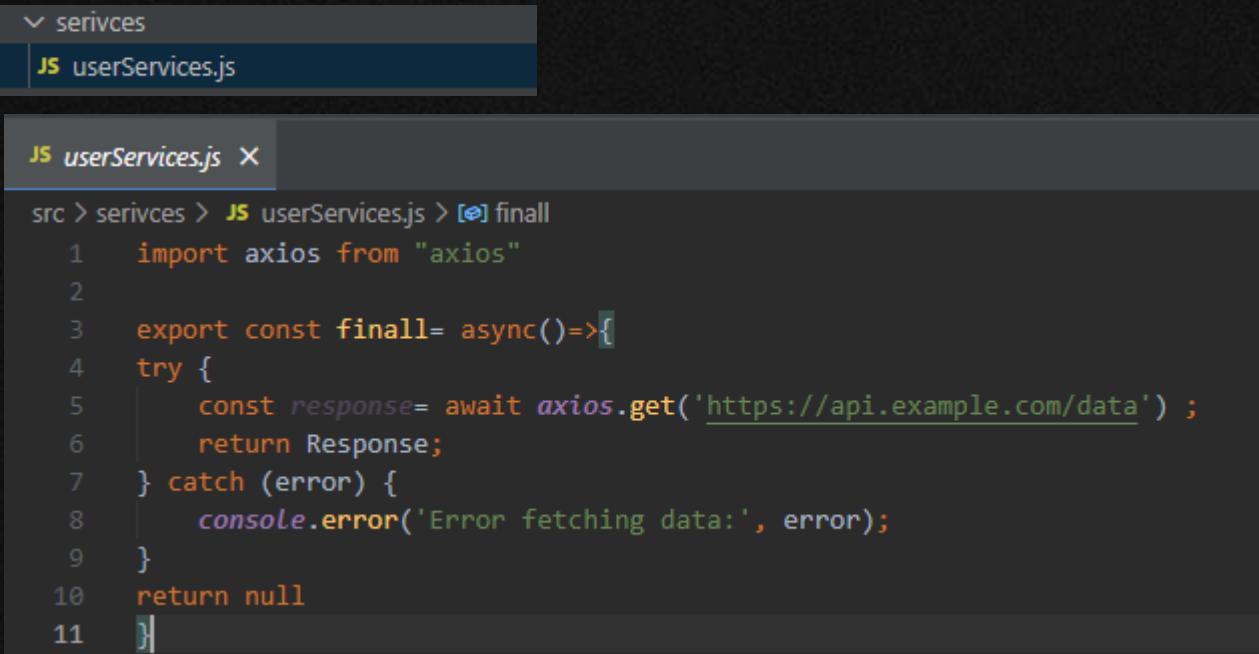


The screenshot shows a dark-themed terminal window from the VS Code interface. At the top, there are tabs for PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL (which is underlined), and PORTS. To the right of the tabs is a toolbar with icons for powershell, a dropdown menu, a trash can, and other options. The main area of the terminal contains a single line of text: "PS C:\Material de programacion\curso Undemy react sprint boot\6-users-app> npm install axios". The background of the terminal is dark, and the text is white.

```
PS C:\Material de programacion\curso Undemy react sprint boot\6-users-app> npm install axios
```

Fernando  
Agudelo  
Gutiérrez

## 10.2 Consumindo API REST GET del Backend en React



```
src > services > JS userServices.js > [e] finall
1 import axios from "axios"
2
3 export const finall= async()=>{
4 try {
5     const response= await axios.get('https://api.example.com/data') ;
6     return Response;
7 } catch (error) {
8     console.error('Error fetching data:', error);
9 }
10 return null
11 }
```

Gutiérrez

## 10.3 Conectando Frontend con Backend

### 10.3.1 Crear metodo para cargar los Usuarios en el hook reducer

JS usersReducer.js X

```
src > reducers > JS usersReducer.js > [o] usersReducer
 1
 2   export const usersReducer = (state = [], action) => {
 3
 4     switch (action.type) {
 5       case 'addUser': ...
 6       case 'removeUser': ...
 7       case 'updateUser': ...
 8       })
 9
10       case "loadingUsers":
11         return action.payload;
12
13       default:
14         return state;
15     }
16   }
```

Devuelve un arreglo vacío

### 10.3.2 Modificar el metodo de cargar usuarios en el hook personalizado (useUsers)

JS useUsers.js

```
src > hooks > JS useUsers.js > [e] useUsers
1  import { useReducer, useState } from "react";
2  import { useNavigate } from "react-router-dom";
3  import Swal from "sweetalert2";
4  import { usersReducer } from "../reducers/usersReducer";
5
6  const initialUsers = [];
7
8  > const initialUserForm = { ... }
13  }
14
15 > export const useUsers = () => { ...
94 }
```

Se deja un arreglo vacío

```
const getUsers= async ()=>{ // funcion para cargar los usuarios
    const result= await findAll(); // funcion que viene del service se guarda la respuesta
    dispatch({
        type:"loadingUsers",
        payload:result.data
    });
}
```

### JS useUsers.js ●

```
src > hooks > JS useUsers.js > [o] useUsers
16  export const useUsers = () => {
49
50  >      const handlerRemoveUser = (id) => { ...
76      }
77
78  >      const handlerUserSelectedForm = (user) => { ...
82      }
83
84  >      const handlerOpenForm = () => { ...
86      }
87
88  >      const handlerCloseForm = () => { ...
91      }
92      return {
93          users,
94          userSelected,
95          initialUserForm,
96          visibleForm,
97          handlerAddUser,
98          handlerRemoveUser,
99          handlerUserSelectedForm,
100         handlerOpenForm,
101         handlerCloseForm,
102         getUsers,
103     }
104 }
```

### 10.3.3 Pasar la función al hook context (userprovider)

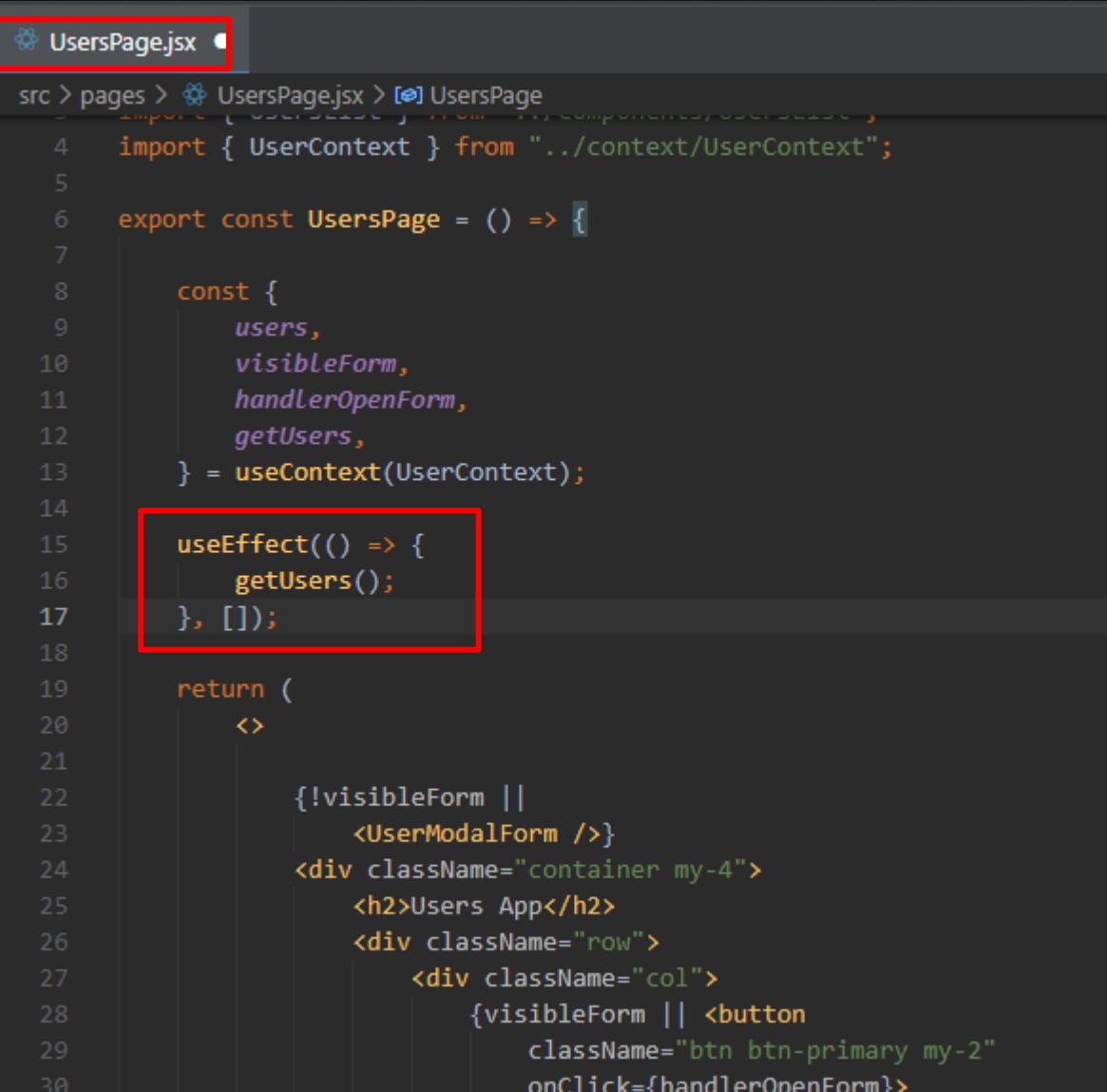
```
>UserProvider.jsx ●  
src > context > UserProvider.jsx > [o] UserProvider > ↗ getUsers  
  4  export const UserProvider = ({children}) => {  
  10    visibleForm,  
  11    handlerAddUser,  
  12    handlerRemoveUser,  
  13    handlerUserSelectedForm,  
  14    handlerOpenForm,  
  15    handlerCloseForm,  
  16    getUsers  
  17  } = useUsers();  
  18  
  19  return (  
  20    <UserContext.Provider value={  
  21      {  
  22        users,  
  23        userSelected,  
  24        initialUserForm,  
  25        visibleForm,  
  26        handlerAddUser,  
  27        handlerRemoveUser,  
  28        handlerUserSelectedForm,  
  29        handlerOpenForm,  
  30        handlerCloseForm,  
  31        getUsers|  
  32      }  
  33    }>  
  34    {children}  
  35  </UserContext.Provider>
```

#### 10.3.4 Buscar el componente que va cargar los usuarios y pasarla la función

UsersPage.jsx

```
src > pages > UsersPage.jsx > UsersPage
1  import { useContext } from "react";
2  import { UserModalForm } from "../components/UserModalForm";
3  import { UsersList } from "../components/UsersList";
4  import { UserContext } from "../context/UserContext";
5
6  export const UsersPage = () => {
7
8      const {
9          users,
10         visibleForm,
11         handlerOpenForm,
12         getUsers
13     } = useContext(UserContext);
14
15     return (
16         <>
17
18             {!visibleForm ||
19                 <UserModalForm />}
20             <div className="container my-4">
21                 <h2>Users App</h2>
22                 <div className="row">
23                     <div className="col">
24                         {visibleForm || <button
25                             className="btn btn-primary my-2"
26                             onClick={handlerOpenForm}>
27                             Nuevo Usuario
28                         </button>
29                     </div>
30                 </div>
31             </div>
32         </>
33     );
34 }
```

### 10.3.5 Ejecutar la función



```
src > pages > UsersPage.jsx > [UsersPage]
  import { useState } from "react";
  import { UserContext } from "../context/UserContext";
  ...
  export const UsersPage = () => {
    ...
    const [
      users,
      visibleForm,
      handlerOpenForm,
      getUsers,
    ] = useContext(UserContext);
    ...
    useEffect(() => {
      getUsers();
    }, []);
    ...
    return (
      ...
        {!visibleForm ||
          <UserModalForm />}
        <div className="container my-4">
          <h2>Users App</h2>
          <div className="row">
            <div className="col">
              {visibleForm || <button
                className="btn btn-primary my-2"
                onClick={handlerOpenForm}>
                ...
              </button>}
            </div>
          </div>
        </div>
      ...
    );
  }
}
```

### 10.3.6 darle permiso al backend para que react acceda

```
*UserController.java X
1 package com.example.demo.Controllers;
2
3+import java.util.List;[]
22
23 @RestController
24 @RequestMapping("/users") // ruta base
25 @CrossOrigin(originPatterns = "*")
26 public class UserController {
27
28+     @Autowired
29     private UserInterfaceService service;
30
32+     public List<User>list(){ //Listar[]
36
38+     public ResponseEntity<?> guardar(@RequestBody User user){ // metodo consultar por id[]
44
46+     public ResponseEntity<?> actualizar( @RequestBody User user,@PathVariable Long id){[]
56
58+     public ResponseEntity<?>show(@PathVariable Long id){[]
69
71+     public ResponseEntity<?>remove(@PathVariable Long id){[]
81
82 |
83 }
```

## 10.4 Consumiendo API RESTFul POST PUT y DELETE del Backend en React

### 10.4.1 Método save

JS userServices.js

```
src > services > JS userServices.js > ...
1 import axios from "axios"
2
3
4 const Base_Url='https://localhost:8080/users';
5
6 > export const findAll= async()=>{
14 }
15
16 export const save= async ({username,password,email}) =>{
17
18   try {
19     return await axios.post(Base_Url,{username,password,email}) ;
20   }
21   catch (error) {
22     console.error('Error fetching data:', error);
23   }
24   return undefined...
25 };
26
```

#### 10.4.2 Método actualizar

JS userServices.js

```
src > serivces > JS userServices.js > ...
1 import axios from "axios"
2
3 const Base_Url='https://localhost:8080/users';
4
5 > export const findAll= async()=>{ ...
13 }
14
15 > export const save= async ({username,password,email}) =>{ ... ...
24 };
25
26 export const update= async ({username,email,id}) =>{
27   try {
28     return await axios.put(` ${Base_Url}/${id}`,{username,email} );
29   } catch (error) {
30     console.error('Error fetching data:', error);
31   }
32   return undefined
33
34 };
```

### 10.4.3 Método Eliminar

JS userServices.js

```
src > services > JS userServices.js > ...
1 import axios from "axios"
2 import { Await } from "react-router-dom";
3
4 const Base_Url='https://localhost:8080/users';
5
6 > export const findAll= async()=>{ ...
14   }
15
16 > export const save= async ({username,password,email}) =>{
25   };
26
27 > export const update= async ({username,email,id}) =>{ ... ...
35   };
36
37 export const remove= async (id) =>{
38
39   try {
40     await axios.delete(` ${Base_Url}/${id}`);
41   } catch (error) {
42     console.error('Error fetching data:', error);
43   }
44
45 }
```

## 10.5 Conectando React con Backend Spring Boot CRUD

### 10.5.1 Modificar el metodo de cargar y actualizar usuarios en el hook personalizado (useUsers)

JS useUsers.js X

```
src > hooks > JS useUsers.js > [o] useUsers > [o] handlerAddUser
16  export const useUsers = () => {
29
30    const handlerAddUser = async(user) => {
31      // console.log(user);
32      let response;
33
34      if (user.id === 0) {
35        response=await save(user)
36      } else {
37        response=response=await update(user)
38      }
39
40      dispatch({
41        type: (user.id === 0) ? 'addUser' : 'updateUser',
42        payload: response.data,
43      });
44
45      Swal.fire(
46        (user.id === 0) ?
47          'Usuario Creado' :
48          'Usuario Actualizado',
49        (user.id === 0) ?
50          'El usuario ha sido creado con exito!' :
51          'El usuario ha sido actualizado con exito!',
52          'success'
53      );
54      handlerCloseForm();
55      navigate('/users');
56    }
  
```

## 10.5.2 Modificar el metodo de eliminar usuarios en el hook personalizado (useUsers)

JS useUsers.js

src > hooks > JS useUsers.js > useUsers > handlerUserSelectedForm

```
16  export const useUsers = () => {
57
58  const handlerRemoveUser = (id) => {
59      Swal.fire({
60          title: 'Esta seguro que desea eliminar?',
61          text: "Cuidado el usuario sera eliminado!",
62          icon: 'warning',
63          showCancelButton: true,
64          confirmButtonColor: '#3085d6',
65          cancelButtonColor: '#d33',
66          confirmButtonText: 'Si, eliminar!'
67      }).then((result) => {
68          if (result.isConfirmed) {
69              remove(id);
70              dispatch({
71                  type: 'removeUser',
72                  payload: id,
73              });
74              Swal.fire(
75                  'Usuario Eliminado!',
76                  'El usuario ha sido eliminado con exito!',
77                  'success'
78              )
79          }
80      })
81  }
```

# Luis

## 1. Introducción a la Validación de Beans

Java Bean Validation es una API que permite definir restricciones en las propiedades de los objetos y validar esas restricciones. Es parte del estándar Java EE (Enterprise Edition) y es útil en aplicaciones Java SE (Standard Edition) también. La API principal se encuentra en el paquete `javax.validation`.

## 11.2 Anotaciones comunes de validacion

`@NotNull` : El campo no debe ser nulo.

`@Null` : El campo debe ser nulo.

`@AssertTrue` : El campo debe ser `true`.

`@AssertFalse` : El campo debe ser `false`.

`@Min(value)` : El campo debe ser un número con un valor mínimo especificado.

`@Max(value)` : El campo debe ser un número con un valor máximo especificado.

`@DecimalMin(value)` : El campo debe ser un número decimal con un valor mínimo especificado.

`@DecimalMax(value)` : El campo debe ser un número decimal con un valor máximo especificado.

`@Size(min, max)` : El tamaño de la colección, mapa, cadena, matriz, etc., debe estar dentro del rango especificado.

`@Digits(integer, fraction)` : El número debe tener una cantidad específica de dígitos enteros y fraccionarios.

`@Past` : La fecha debe ser anterior al presente.

`@PastOrPresent` : La fecha debe ser anterior o igual al presente.

`@Future` : La fecha debe ser posterior al presente.

`@FutureOrPresent` : La fecha debe ser posterior o igual al presente.

`@Pattern(regexp)` : El campo debe coincidir con la expresión regular especificada.

`@Email` : El campo debe ser una dirección de correo electrónico válida.

`@NotEmpty` : El campo no debe estar vacío (utilizado en cadenas, colecciones, map etc.).

`@NotBlank` : El campo no debe estar en blanco (solo para cadenas).

`@Positive` : El número debe ser positivo.

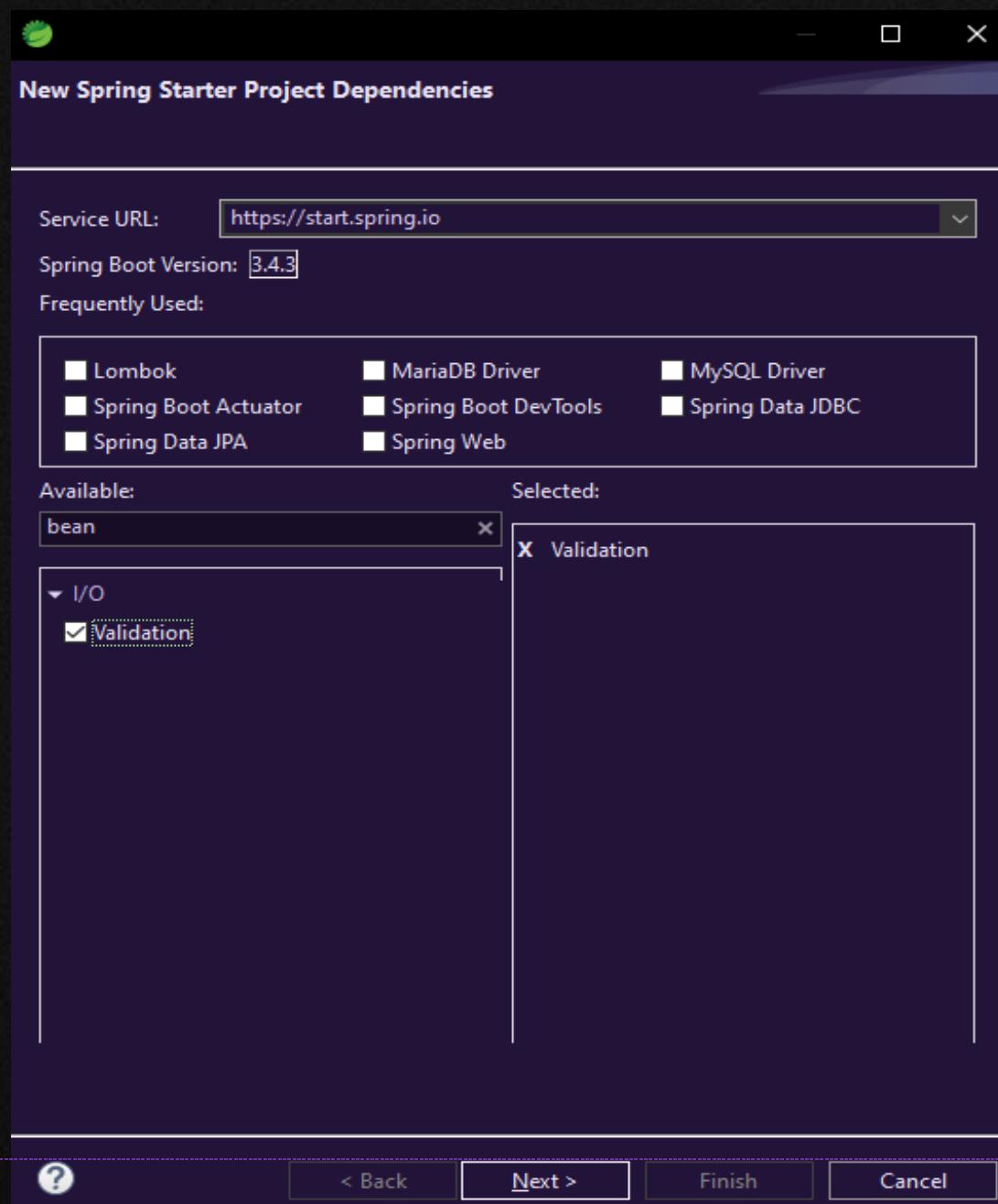
`@PositiveOrZero` : El número debe ser positivo o cero.

`@Negative` : El número debe ser negativo.

`@NegativeOrZero` : El número debe ser negativo o cero.

`@Range(min, max)` : El valor debe estar dentro de un rango específico.

## 11.3 Instalar dependencias



## 11.4 Validar en el Entity o modelo

```
*User.java X
13 import jakarta.validation.constraints.Size;
14 import jakarta.validation.constraints.Email;
15
16 @Entity
17 @Table(name = "users")
18 public class User {
19 @Id
20 @GeneratedValue(strategy = GenerationType.IDENTITY)
21 private long Id;
22
23 @NotBlank // solo para string
24 @Size(min = 4,max = 8)
25 @Column(unique = true)
26 private String Username;
27
28 @NotEmpty
29 @Column(unique = true)
30 private String Password;
31
32 @NotEmpty
33 @Email
34 @Column(unique = true)
35 private String Email;
36
37 public long getId() {
38     return Id;
39 }
```

## 11.5 Validar los métodos Post y Pust del controller

### 11.5.1 método post (para el pust es el mismo proceso)

```
J UserController.java X
1 package com.example.demo.Controllers;
2
3+import java.util.HashMap;...
27
28 @RestController
29 @RequestMapping("/users") // ruta base
30 @CrossOrigin(originPatterns = "*")
31 public class UserController {
32
34+    private UserInterfaceService service;...
35
37+    public List<User>list(){ //Listar...
38
39+        return service.list();
40
41+    }
42+    @PostMapping // status por defecto 200 metodo insertar
43+    public ResponseEntity<?> guardar(@Valid @RequestBody User user, BindingResult result )
44+    { // metodo consultar por id
45+        if (result.hasErrors()) {
46+            return new ResponseEntity<User>(result.getAllErrors(), HttpStatus.BAD_REQUEST);
47+        }
48+        return new ResponseEntity<User>(service.guardar(user), HttpStatus.CREATED);
49+    }
50
51
52 }
```

1. **@Valid:** Valida el objeto `User` según las restricciones definidas en su clase.

Valida el modelo segun se requiera, con las restricciones que se hicieron en con java bean en el punto 11.4

2. **@RequestBody:** Indica que el objeto `User` se deserializa del cuerpo de la solicitud HTTP.

Se desestructuraria en el cuerpo de la solicitu del modelo y se crea una instancia de la class `User` en este caso se llama `user`

3. **BindingResult**: Contiene los resultados de la validación. Si hay errores, puedes manejarlos adecuadamente.

Donde se guarda los resultados de la validacion y se guardan en una variable en este caso llamada result, por obligacion debe ir al lado del @RequestBody

UserController.java

```
1 package com.example.demo.Controllers;
2
3 import java.util.HashMap;
4
5
6 @RestController
7 @RequestMapping("/users") // ruta base
8 @CrossOrigin(originPatterns = "*")
9 public class UserController {
10
11     private UserInterfaceService service;
12
13     public List<User>list(){ //Listar
14
15     }
16
17     @PostMapping // status por defecto 200 metodo insertar
18     public ResponseEntity<?> guardar(@Valid @RequestBody User user, BindingResult result )
19     { // metodo consultar por id
20
21         if (result.hasErrors()) {
22             return validation(result);
23         }
24
25     }
26
27 }
```

**if (result.hasErrors()):** Verifica si hay errores de validación en el objeto `User`. Si hay errores, se ejecuta el bloque dentro del `if`.

**return validation(result);:** Llama a un método `validation`

Devolvemos los errores con un método en este caso se llama `validation` y se pasa como parametro al metodo el `result`. Hay que crear el metodo

### UserController.java

```
35  
37+     public List<User>list(){ //Listar..  
41  
43+     public ResponseEntity<?> guardar(@Valid @RequestBody User user, BindingResult result ){  
53  
54  
56+     public ResponseEntity<?> actualizar( @Valid @RequestBody User user, BindingResult result,..  
70  
71+     private ResponseEntity<?> validation(BindingResult result) {  
72  
73  
74  
75  
76  
77  
78  
79     }  
80  
82+     public ResponseEntity<?>show(@PathVariable Long id){..  
93  
95+     public ResponseEntity<?>remove(@PathVariable Long id){..  
105  
106  
107 }
```

\*\*private ResponseEntity<?> validation(BindingResult result)\*\*: Este es un método privado que toma un objeto `BindingResult` como parámetro y devuelve un `ResponseEntity<?>`. El `<?>` indica que el tipo de cuerpo de la respuesta puede ser cualquier tipo. Este método se utiliza para manejar los errores de validación.

### UserController.java

```
35
37+     public List<User>list(){ //Listar...
41
43+     public ResponseEntity<?> guardar(@Valid @RequestBody User user, BindingResult result ){}
53
54
56+     public ResponseEntity<?> actualizar( @Valid @RequestBody User user, BindingResult result,[])
70
71+         private ResponseEntity<?> validation(BindingResult result) {
72
73             Map<String, String> errors= new HashMap<>();
74
75
76
77
78
79         }
80
82+     public ResponseEntity<?>show(@PathVariable Long id){}
93
95+     public ResponseEntity<?>remove(@PathVariable Long id){}
105
106
107 }
```

**Map<String, String> errors = new HashMap<>();**: Se crea un `HashMap` para almacenar los errores de validación. La clave (`String`) será el nombre del campo que tiene el error, y el valor (`String`) será el mensaje de error asociado a ese campo.

### UserController.java

```
35
37+     public List<User>list(){ //Listar[]
41
43+     public ResponseEntity<?> guardar(@Valid @RequestBody User user, BindingResult result){}
53
54
56+     public ResponseEntity<?> actualizar( @Valid @RequestBody User user, BindingResult result,[])
70
71+     private ResponseEntity<?> validation(BindingResult result) {
72
73         Map<String, String> errors= new HashMap<>();
74         result.getFieldErrors().forEach(err->{
75
76             });
77
78         }
79
80
82+     public ResponseEntity<?>show(@PathVariable Long id){}
93
95+     public ResponseEntity<?>remove(@PathVariable Long id){}
105
106
107 }
```

**result.getFieldErrors().forEach(err->{** Se obtiene la lista de errores de campo ( `FieldError` ) del objeto `BindingResult` y se itera sobre cada error usando `forEach`. `err` es una variable que representa cada error de campo en la iteración.

### UserController.java

```
35
36+     public List<User>list(){ //Listar[]
41
42+     public ResponseEntity<?> guardar(@Valid @RequestBody User user, BindingResult result ){}
53
54
55+     public ResponseEntity<?> actualizar( @Valid @RequestBody User user, BindingResult result,[])
70
71+     private ResponseEntity<?> validation(BindingResult result) {
72
73         Map<String, String> errors= new HashMap<>();
74         result.getFieldErrors().forEach(err->{
75             errors.put(err.getField(), "El campo"+" " + err.getDefaultMessage());
76         });
77
78
79
80
82+     public ResponseEntity<?>show(@PathVariable Long id){}
93
95+     public ResponseEntity<?>remove(@PathVariable Long id){}
105
106
107 }
```

**errors.put(err.getField(), "El campo"+" " + err.getDefaultMessage());**: Para cada error de campo, se agrega una entrada al mapa `errors`. La clave es el nombre del campo (`err.getField()`), y el valor es un mensaje de error que combina la cadena "El campo" con el mensaje de error predeterminado (`err.getDefaultMessage()`).

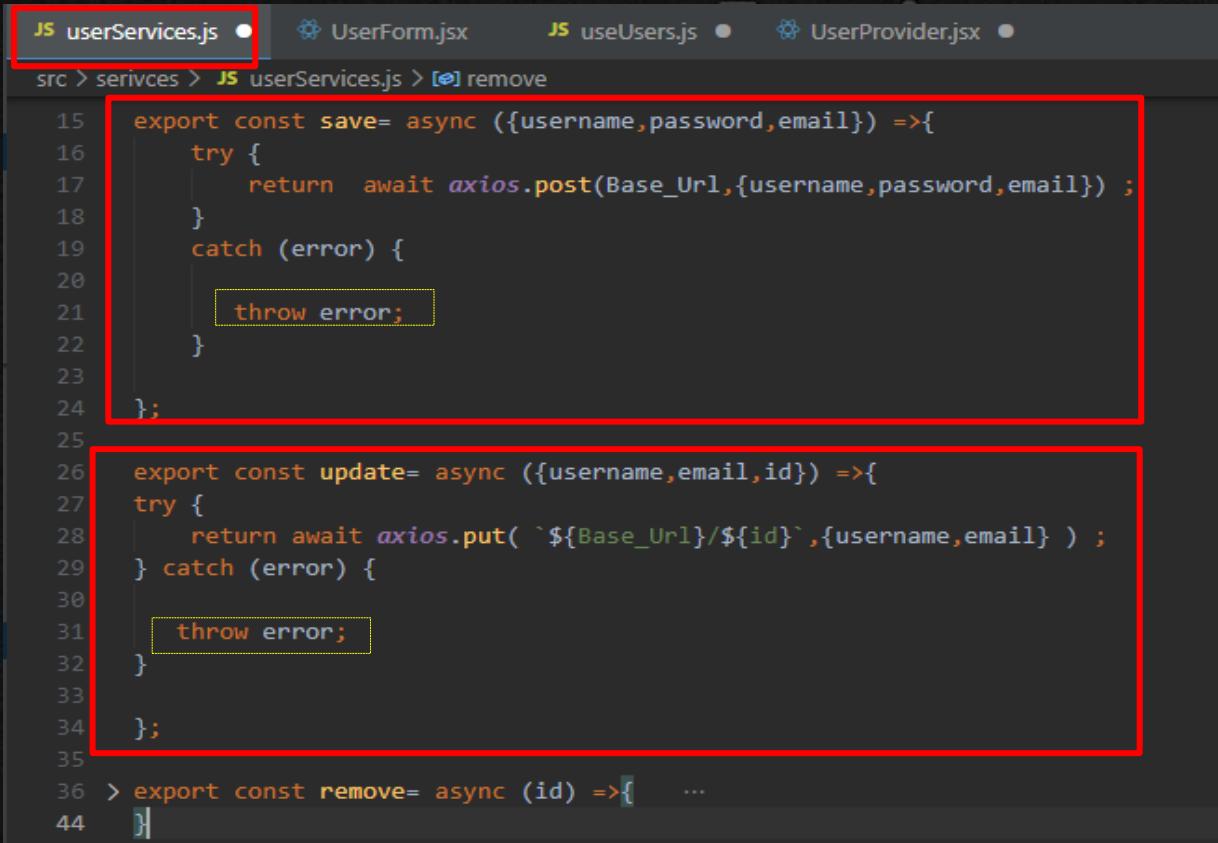
### UserController.java

```
35  
37+     public List<User>list(){ //Listar..  
41  
43+     public ResponseEntity<?> guardar(@Valid @RequestBody User user, BindingResult result )  
53  
54  
56+     public ResponseEntity<?> actualizar( @Valid @RequestBody User user, BindingResult result,..  
70  
71+         private ResponseEntity<?> validation(BindingResult result) {  
72  
73             Map<String, String> errors= new HashMap<>();  
74             result.getFieldErrors().forEach(err->{  
75                 errors.put(err.getField(), "El campo"+" " + err.getDefaultMessage());  
76             });  
77  
78             return ResponseEntity.badRequest().body(errors);  
79         }  
80  
82+     public ResponseEntity<?>show(@PathVariable Long id){}  
93  
95+     public ResponseEntity<?>remove(@PathVariable Long id){}  
105  
106  
107 }
```

**return ResponseEntity.badRequest().body(errors);**: Devuelve una respuesta HTTP con el código de estado 400 (BAD REQUEST) y el mapa `errors` como cuerpo de la respuesta. Esto indica que la solicitud no pudo ser procesada debido a errores de validación.

## 12. Conectar validaciones del Backend en react

### 12 .1 Capturar el error en el service del fronted en los metodos save y update



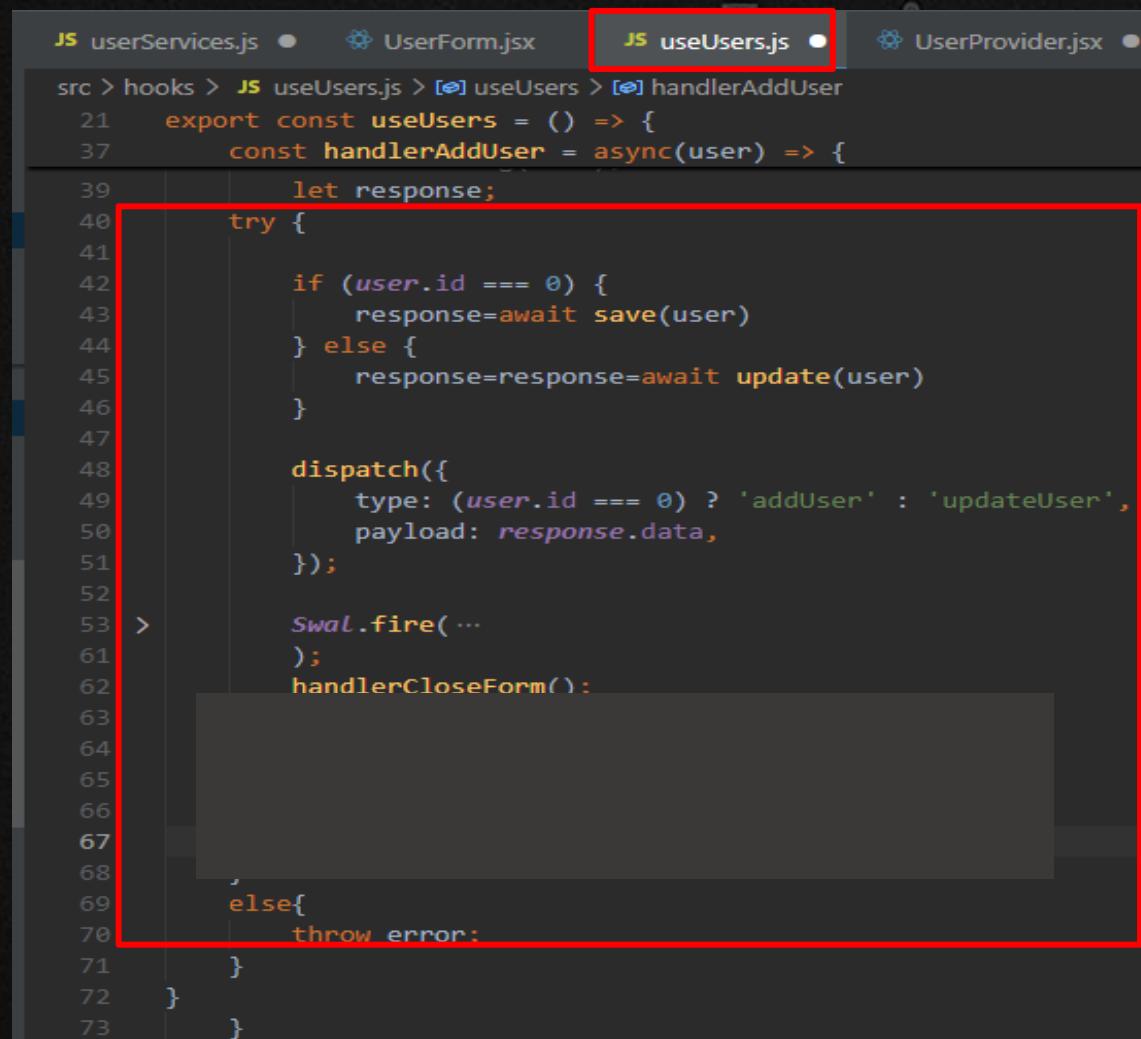
```
JS userServices.js • UserForm.jsx useUsers.js • UserProvider.jsx •
src > services > JS userServices.js > [e] remove

15 export const save= async ({username,password,email}) =>{
16   try {
17     return await axios.post(Base_Url,{username,password,email}) ;
18   }
19   catch (error) {
20
21     throw error;
22   }
23
24 };
25
26 export const update= async ({username,email,id}) =>{
27   try {
28     return await axios.put(` ${Base_Url}/${id}` ,{username,email} ) ;
29   } catch (error) {
30
31     throw error;
32   }
33
34 };
35
36 > export const remove= async (id) =>{ ...
44 }
```

`throw error;` vuelve a lanzar el error capturado, permitiendo que el código que llame a esta función gestione la excepción.

## 12.2 Modificar los metodos save y actualizar donde los estamos utilizando hook personalizado ((userUsers) para Capturar los errores

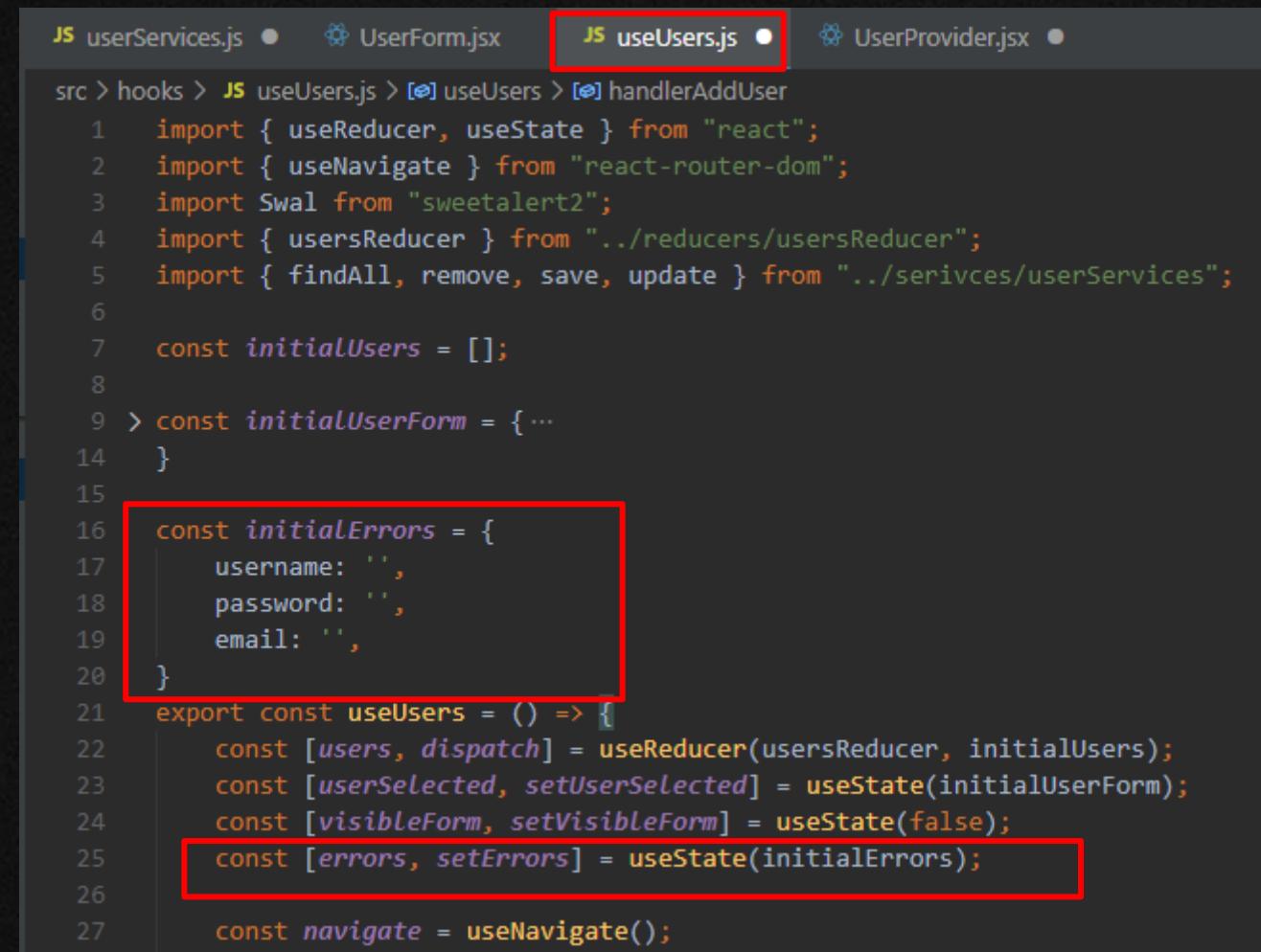
### 12.2.1 Envolver el metodo en un try catch



```
JS userServices.js ● UserForm.jsx JS useUsers.js ● UserProvider.jsx ●
src > hooks > JS useUsers.js > [e] useUsers > [e] handlerAddUser
21  export const useUsers = () => {
37    const handlerAddUser = async(user) => {
39      let response;
40      try {
41        if (user.id === 0) {
42          response=await save(user)
43        } else {
44          response=response=await update(user)
45        }
46      }
47
48      dispatch({
49        type: (user.id === 0) ? 'addUser' : 'updateUser',
50        payload: response.data,
51      });
52
53    >     Swal.fire(...
54      );
55      handlerCloseForm();
56
57
58    }
59    else{
60      throw error;
61    }
62  }
63
64
65
66
67
68
69
70
71  }
72 }
73 }
```

Se envuelve en el metodo en un try catch

## 12.2.2 Crear objeto que va ser el estado inicial para el manejo de los errores y crear el hook useState



```
JS userServices.js ● ⚡ UserForm.jsx ⚡ useUsers.js ● ⚡ UserProvider.jsx ●
src > hooks > JS useUsers.js > [e] useUsers > [e] handlerAddUser
1 import { useReducer, useState } from "react";
2 import { useNavigate } from "react-router-dom";
3 import Swal from "sweetalert2";
4 import { usersReducer } from "../reducers/usersReducer";
5 import { findAll, remove, save, update } from "../services/userServices";
6
7 const initialUsers = [];
8
9 > const initialUserForm = {
10   }
11
12 const initialErrors = {
13   username: '',
14   password: '',
15   email: ''
16 }
17
18 export const useUsers = () => {
19   const [users, dispatch] = useReducer(usersReducer, initialUsers);
20   const [userSelected, setUserSelected] = useState(initialUserForm);
21   const [visibleForm, setVisibleForm] = useState(false);
22   const [errors, setErrors] = useState(initialErrors);
23
24   const navigate = useNavigate();
25 }
```

JS userServices.js ● ❁ UserForm.jsx

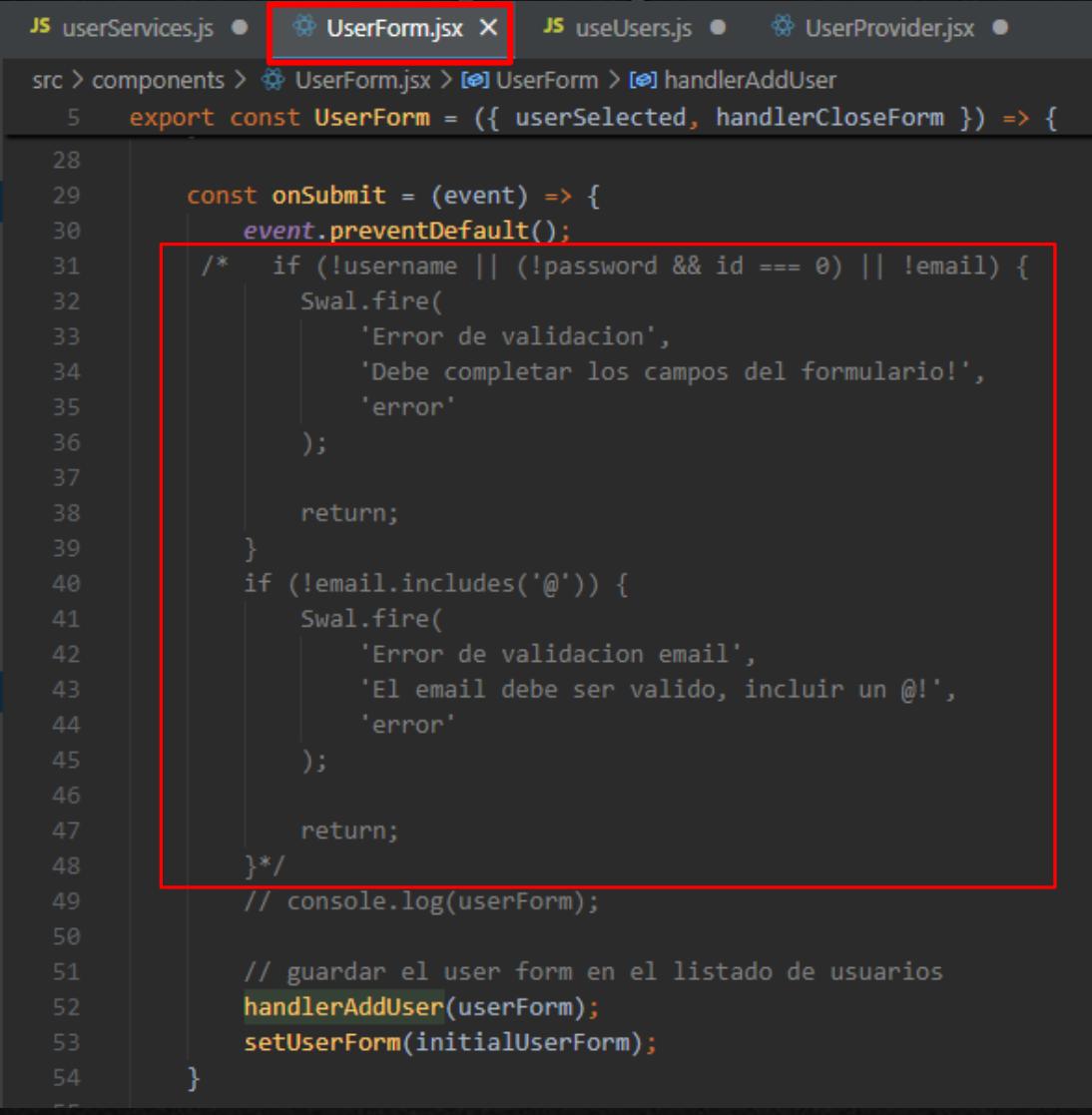
JS useUsers.js ●

❁ UserProvider.jsx ●

src > hooks > JS useUsers.js > [o] useUsers

```
21  export const useUsers = () => {
109
110    >   const handlerCloseForm = () => { ...
113    }
114    return {
115      users,
116      userSelected,
117      initialUserForm,
118      visibleForm,
119      errors, errors,
120      handlerAddUser,
121      handlerRemoveUser,
122      handlerUserSelectedForm,
123      handlerOpenForm,
124      handlerCloseForm,
125      getUsers,
126    }
127 }
```

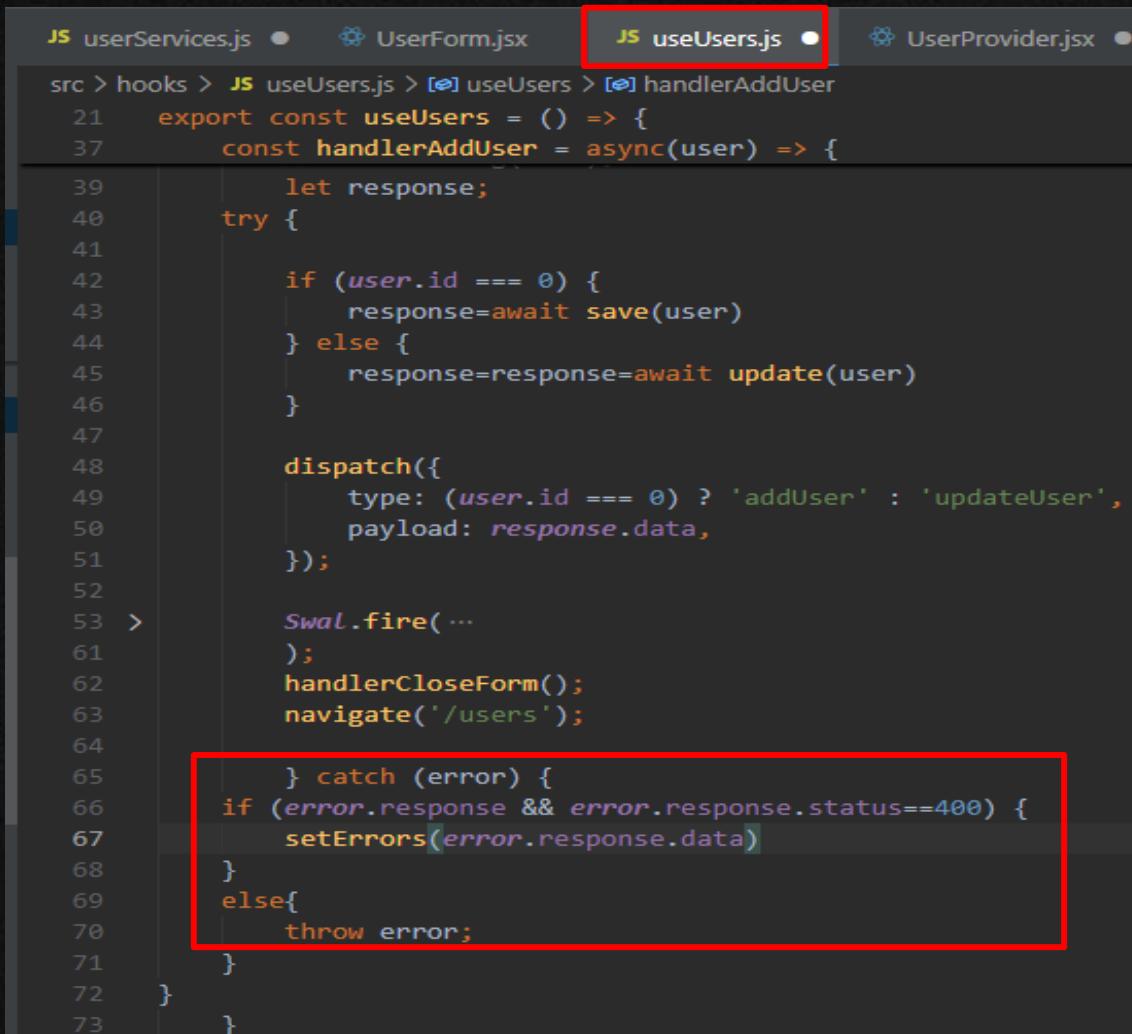
## 12.3 Eliminar las validaciones que tengamos en el frontend que no sean del backend (en caso de que tengaos)



The screenshot shows a code editor with the file `UserForm.jsx` open. The code is written in JavaScript and JSX. A red box highlights the validation logic starting at line 31.

```
JS userServices.js ●  UserForm.jsx X JS useUsers.js ●  UserProvider.jsx ●
src > components > UserForm.jsx > UserForm > handlerAddUser
5   export const UserForm = ({ userSelected, handlerCloseForm }) => {
28
29     const onSubmit = (event) => {
30       event.preventDefault();
31       /* if (!username || (!password && id === 0) || !email) {
32         Swal.fire(
33           'Error de validacion',
34           'Debe completar los campos del formulario!',
35           'error'
36         );
37
38         return;
39       }
40       if (!email.includes('@')) {
41         Swal.fire(
42           'Error de validacion email',
43           'El email debe ser valido, incluir un @!',
44           'error'
45         );
46
47         return;
48     }*/
49     // console.log(userForm);
50
51     // guardar el user form en el listado de usuarios
52     handlerAddUser(userForm);
53     setUserForm(initialUserForm);
54 }
```

## 12.4 Gestionar los errores



```
JS userServices.js ● ⚡ UserForm.jsx JS useUsers.js ● ⚡ UserProvider.jsx ●
src > hooks > JS useUsers.js > [x] useUsers > [x] handlerAddUser
21   export const useUsers = () => {
37     const handlerAddUser = async(user) => {
38       let response;
39       try {
40         if (user.id === 0) {
41           response=await save(user)
42         } else {
43           response=response=await update(user)
44         }
45
46         dispatch({
47           type: (user.id === 0) ? 'addUser' : 'updateUser',
48           payload: response.data,
49         });
50
51       } catch (error) {
52         if (error.response && error.response.status==400) {
53           setErrors(error.response.data)
54         } else{
55           throw error;
56         }
57       }
58     }
59   }
60 }
```

**error.response:** Verifica si el objeto `error` tiene una propiedad `response`. Esto es común en errores generados por bibliotecas como `axios`, donde `error.response` contiene la respuesta HTTP del servidor.

**error.response.status == 400:** Verifica si el código de estado de la respuesta es 400, que indica una "solicitud incorrecta" (Bad Request). Este código generalmente se usa cuando el servidor no puede procesar la solicitud debido a un error del cliente, como datos de entrada inválidos.

**setErrors(...):** Es una función (probablemente un "setter" de estado en React) que se utiliza para actualizar el estado de la aplicación con los errores recibidos del servidor. Esto permite mostrar los errores al usuario en la interfaz de la aplicación.

### 3. } else {

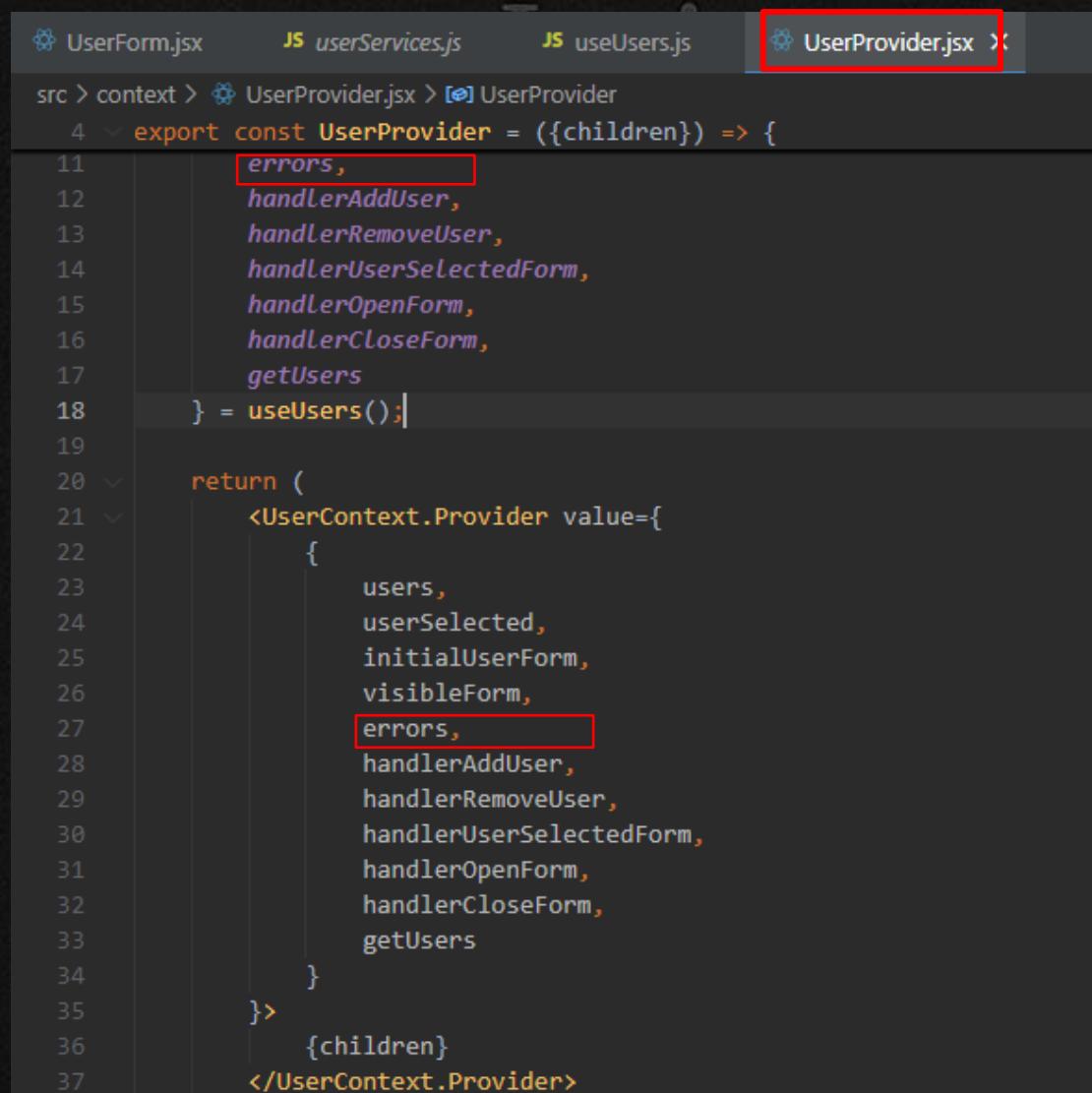
- Si la condición del `if` no se cumple (es decir, si no hay una respuesta o el código de estado no es 400), se ejecuta el bloque `else`.

### 4. `throw error;`

- **throw error;**: Lanza el error nuevamente para que pueda ser manejado en otro lugar de la aplicación. Esto es útil para errores que no son de tipo 400 y que podrían requerir un manejo diferente, como errores de red, errores del servidor (500), etc.

## 12.5 Mostrar los errores en un formulario POST y PUT

### 12.5.1 Pasar el estado Inicial de los errores(errors) al contexto



```
src > context > UserProvider.jsx > UserProvider
  4 export const UserProvider = ({children}) => {
  5   const [users, setUsers] = useState([]);
  6   const [userSelected, setUserSelected] = useState(null);
  7   const [initialUserForm, setInitialUserForm] = useState("add");
  8   const [visibleForm, setVisibleForm] = useState("none");
  9
 10   const errors = useState([]);
 11   const handlerAddUser = () => {
 12     setInitialUserForm("add");
 13     setVisibleForm("block");
 14   };
 15   const handlerRemoveUser = () => {
 16     setInitialUserForm("remove");
 17     setVisibleForm("block");
 18   };
 19   const handlerUserSelectedForm = () => {
 20     setVisibleForm("block");
 21   };
 22   const handlerOpenForm = () => {
 23     setVisibleForm("block");
 24   };
 25   const handlerCloseForm = () => {
 26     setVisibleForm("none");
 27   };
 28   const getUsers = () => {
 29     fetch("https://jsonplaceholder.typicode.com/users")
 30       .then((res) => res.json())
 31       .then((data) => setUsers(data));
 32   };
 33
 34   return (
 35     <UserContext.Provider value={{
 36       users,
 37       userSelected,
 38       initialUserForm,
 39       visibleForm,
 40       errors,
 41       handlerAddUser,
 42       handlerRemoveUser,
 43       handlerUserSelectedForm,
 44       handlerOpenForm,
 45       handlerCloseForm,
 46       getUsers
 47     }}
 48   >
 49     {children}
 50   </UserContext.Provider>
 51 
```

## 12.5.2 Utilizar el hook contexto en el formulario para mostrar los errores

```
❶ UserForm.jsx X JS userServices.js JS useUsers.js
src > components > ❷ UserForm.jsx > ❸ UserForm > ❹ onInputChange
1  import { useContext, useEffect, useState } from "react"
2  import Swal from "sweetalert2";
3  import { UserContext } from "../context/UserContext";
4
5  export const UserForm = ({ userSelected, handlerCloseForm }) => {
6
7      const { initialUserForm, handlerAddUser, errors } = useContext(UserContext);
8
9      const [userForm, setUserForm] = useState(initialUserForm);
10
```

Aguadejo  
Gutiérrez

### 12.5.3 Mostrar los errores en el formulario POST Y PUT

The screenshot shows a Java IDE interface with three files open:

- UserForm.jsx**: A React component file containing code for a form. It uses the `onInputChange` prop to handle changes in input fields and displays error messages using `<p className="text-danger">{ errors?.[field] }</p>` blocks.
- userServices.js**: A JavaScript file that imports `useUsers.js`.
- User.java**: A Java entity class with annotations for persistence. It defines three fields: `Id`, `Username`, and `Password`. Each field has a `@Column(unique = true)` annotation.

Dashed arrows from the `[field]` placeholder in the JSX code point to the `Username`, `Password`, and `Email` fields in the `User.java` class, indicating that the error messages should be displayed for these specific fields.

```
src > components > UserForm.jsx > UserForm > onInputChange
  5  export const UserForm = ({ userSelected, handlerCloseForm }) => {
  59 }
  60   return (
  61     <form onSubmit={ onSubmit }>
  62       <input
  63         className="form-control my-3 w-75"
  64         placeholder="Username"
  65         name="username"
  66         value={ username }
  67         onChange={onInputChange} />
  68       <p className="text-danger">{ errors?.Username}</p>
  69
  70     { id > 0 || <input
  71       className="form-control my-3 w-75"
  72       placeholder="Password"
  73       type="password"
  74       name="password"
  75       value={password}
  76       onChange={onInputChange} /> }
  77     <p className="text-danger">{ errors?.Password}</p>
  78
  79     <input
  80       className="form-control my-3 w-75"
  81       placeholder="Email"
  82       name="email"
  83       value={email}
  84       onChange={onInputChange} />
  85     <p className="text-danger">{ errors?.Email}</p>
  86   
```

```
4
 5+import jakarta.persistence.Column;
 15
 16 @Entity
 17 @Table(name = "users")
 18 public class User {
 19@Id
 20 @GeneratedValue(strategy = GenerationType.IDEN
 21 private long Id;
 22
 23@NotBlank // solo para string
 24 @Size(min = 4,max = 8)
 25 @Column(unique = true)
 26 private String Username;
 27
 28@NotBlank // solo para string
 29 @Column(unique = true)
 30 private String Password;
 31
 32@NotBlank // solo para string
 33 @Email
 34 @Column(unique = true)
 35 private String Email;
 36
 37public long getId() {
 38    return Id;
 39  }
```

Debe ser el mismo atributo del modelo

#### 12.5.4 Configurar para que los errores no se queden viendo al cerrar el formulario, y para que no se reinicie los campos

The screenshot shows a code editor with two files open: `UserForm.jsx` and `userServices.js`. The `UserForm.jsx` file is the active tab, indicated by a red border around its title bar.

```
src > components > UserForm.jsx > UserForm > onInputChange
  5  export const UserForm = ({ userSelected, handlerCloseForm }) => {
 29    const onSubmit = (event) => {
 50
 51      // guardar el user form en el listado de usuarios
 52      handlerAddUser(userForm);
 53      // setUserForm(initialUserForm);
 54    }
 55
 56    const onCloseForm = () => {
 57      handlerCloseForm();
 58      setUserForm(initialUserForm);
 59    }
 60
 61    return (
 62      <form onSubmit={ onSubmit }>...
 63      </form>
 64    )
 65  }
 66}
```

Two specific sections of the `UserForm.jsx` code are highlighted with dashed boxes:

- A yellow box surrounds the line `// setUserForm(initialUserForm);` in the `onSubmit` function. An arrow points from this box to the text "Para que no se reinicie los campos".
- A yellow box surrounds the line `setUserForm(initialUserForm);` in the `onCloseForm` function. An arrow points from this box to the text "Para que no se muestre los mensajes de error al cerrar el formulario y volverlo abrir".

The `userServices.js` file is visible in the background, showing some basic export statements.

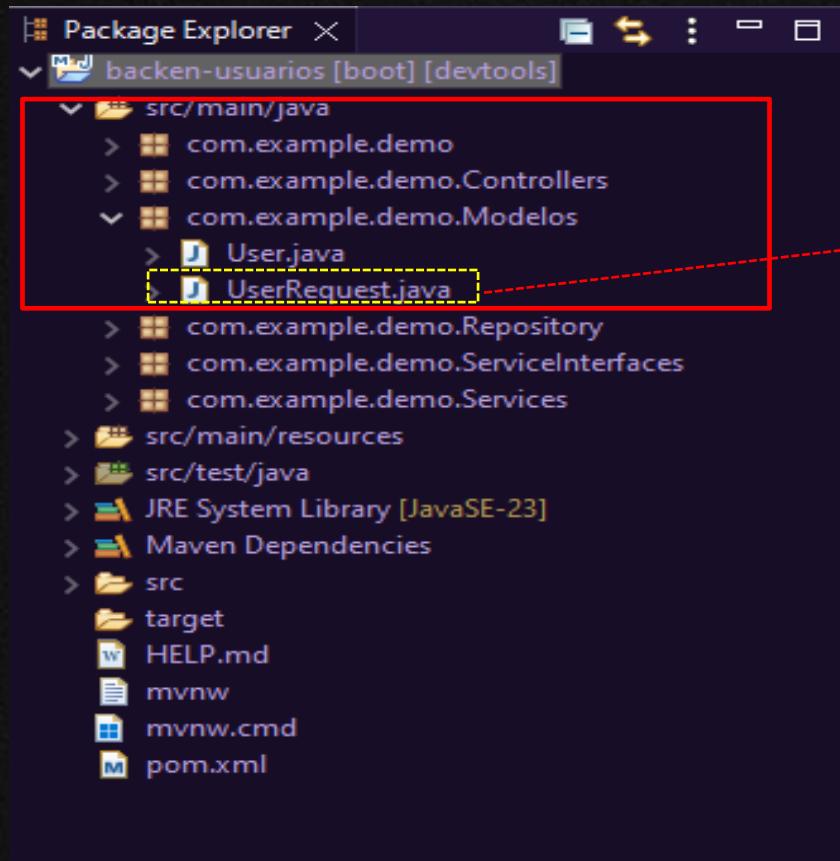
## 12.6 Configurar el actualizar en el backend para que no muestre el mensaje de error en el password



Solucionar para que no muestre este mensaje de error del password en editar, Solucionarlo desde el backend

Gutiérrez

### 12.6.1 crear una nueva clase



Se crea una class dentro del paquete  
modelos, no va ser un modelo

UserRequest.java X UserInterfaceService.java UserController.java

```
1 package com.example.demo.Modelos;
```

```
2
```

```
3
```

```
4
```

```
5
```

```
6
```

```
7
```

```
8 public class UserRequest {
```

```
9
```

```
10
```

```
11
```

```
12
```

```
13
```

```
14
```

```
15
```

```
16
```

```
17
```

```
18
```

```
19
```

```
20
```

```
21
```

```
22
```

```
23
```

```
24
```

```
25
```

```
26
```

```
27
```

```
28
```

```
29
```

```
30
```

```
31 }
```

```
32
```

### 12.6.1.1 Configuracion de la class UserRequest

```
UserRequest.java X UserInterfaceService.java UserController.java
1 package com.example.demo.Modelos;
2
3
4+import jakarta.validation.constraints.Email;[]
7
8 public class UserRequest {
9     @NotBlank // solo para string
10    @Size(min = 4,max = 8)
11    private String Username;
12
13    private String Email;[]
16+
17
18
19+    public String getUsername() {[]
22
23
24+    public void setUsername(String username) {[]
27
28
29+    public String getEmail() {[]
32
33
34+    public void setEmail(String email) {[]
37 }
38
```

## 1. Username

Java

 Copiar

```
@NotBlank  
@Size(min = 4, max = 8)  
private String Username;
```

- Este es un campo privado de tipo `String`, llamado `Username`.
- **Validaciones:**
  - `@NotBlank` : Se asegura de que `Username` no esté vacío ni tenga solo espacios.
  - `@Size(min = 4, max = 8)` : Restringe el número de caracteres que `Username` puede tener, entre 4 y 8.

## 2. Email

Java

 Copiar

```
@NotBlank  
@Email  
private String Email;
```

- Este es un campo privado de tipo `String`, llamado `Email`.
- **Validaciones:**
  - `@NotBlank` : Garantiza que el email no esté vacío.
  - `@Email` : Valida que el valor ingresado sea un correo electrónico con formato correcto.

## Métodos Getter y Setter

### 1. Para `Username`

- **Getter:**

Java

 Copiar

```
public String getUsername() {  
    return Username;  
}
```

Permite acceder al valor de `Username`.

- **Setter:**

Java

 Copiar

```
public void setUsername(String username) {  
    Username = username;  
}
```

Permite modificar el valor de `Username`.

## 2. Para Email

- Getter:

Java

 Copiar

```
public String getEmail() {  
    return Email;  
}
```

Devuelve el valor del campo Email.

- Setter:

Java

 Copiar

```
public void setEmail(String email) {  
    Email = email;  
}
```

## 12.6.1.2 Modificar el método actualizar de la interface UserInterfaceService

```
1 package com.example.demo.ServiceInterfaces;
2
3 import java.util.List;
4
5 public interface UserInterfaceService {
6
7     List<User>findAll(); // Listar
8     Optional<User>findById(Long id); // consultar por id
9     User save(User user); // Guardar
10    Optional<User>update(UserRequest user, Long id); // actualizar
11    void remove (Long id); // eliminar
12
13 }
14
```

Gutiérrez

### 12.6.1.3 Modificare el metodo actualizar del controller

```
UserRequest.java UserInterfaceService.java UserController.java X UserService.java
24 import com.example.demo.Modelos.UserRequest;
25 import com.example.demo.ServiceInterfaces.UserInterfaceService;
26
27 import jakarta.validation.Valid;
28
29 @RestController
30 @RequestMapping("/users") // ruta base
31 @CrossOrigin(originPatterns = "*")
32 public class UserController {
33
34+     @Autowired
35     private UserInterfaceService service;
36
38+     public List<User>list(){ //Listar
42
44+     public ResponseEntity<?> guardar(@Valid @RequestBody User user, BindingResult result ){}
54
55
57+     public ResponseEntity<?> actualizar( @Valid @RequestBody UserRequest user, BindingResult result,[])
71
72+     private ResponseEntity<?> validation(BindingResult result) {}
81
83+     public ResponseEntity<?>show(@PathVariable Long id){}
94
96+     public ResponseEntity<?>remove(@PathVariable Long id){}
106
107
108 }
109
```

### 12.6.1.5 Modificarel metodo actualizar del service

```
UserRequest.java UserInterfaceService.java UserController.java UserService.java
17 public class UserService implements UserInterfaceService {
18
19     @Autowired
20     private UserRepository repository;
21
24+     public List<User> findAll() {..}
28
30+     public Optional<User> findById(Long id) {..}
34
37+     public User save(User user) {..}
41
42     @Transactional
43     @Override
44     public Optional<User> update(UserRequest user, Long id) {
45         Optional<User> o= this.findById(id) ;
46         User userOptional=null;
47         if(o.isPresent()) {
48             User userDb=o.orElseThrow();
49             userDb.setUsername(user.getUsername());
50             userDb.setEmail(user.getEmail());
51             userOptional=(this.save(userDb));
52         }
53         return Optional.ofNullable(userOptional);
54
55     }
```

## 12.7 Mostrar errores de campos únicos (nick o email ya existen)

### 12.7.1 Colocar nombre de clave a los campos únicos en la base de datos

Servidor: 127.0.0.1 » Base de datos: db\_users\_sprintboot » Tabla: users

Examinar Estructura SQL Buscar Insertar Exportar Importar Privilegios Operaciones Seguimiento

Estructura de tabla Vista de relaciones

#	Nombre	Tipo	Cotejamiento	Atributos	Nulo	Predeterminado	Comentarios	Extra	Acción
1	id	bigint(20)			No	Ninguna		AUTO_INCREMENT	Cambiar Eliminar Más
2	email	varchar(255)	utf8mb4_general_ci		Sí	NULL			Cambiar Eliminar Más
3	password	varchar(255)	utf8mb4_general_ci		Sí	NULL			Cambiar Eliminar Más
4	username	varchar(8)	utf8mb4_general_ci		No	Ninguna			Cambiar Eliminar Más

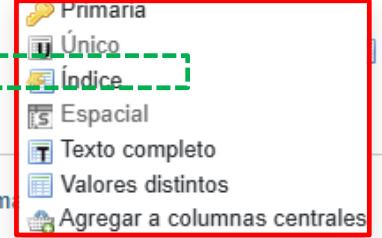
Seleccionar todo Para los elementos que están marcados: Examinar Cambiar Eliminar

Espacial Texto completo Agregar a columnas centrales Eliminar de las columnas centrales

Imprimir Planteamiento de la estructura de tabla Hacer seguimiento a la tabla Mover columnas Norma

Agregar 1 columna(s) después de username Continuar

Primaria Único Índice Espacial Texto completo Valores distintos Agregar a columnas centrales



Acción	Nombre de la clave	Tipo	Único	Empaquetado	Columna	Cardinalidad	Cotejamiento	Nulo	Comentario
Editar  Renombrar  Eliminar	PRIMARY	BTREE	Sí	No	id	0	A	No	
Editar  Renombrar  Eliminar	UK_username	BTREE	Sí	No	username	0	A	No	
Editar  Renombrar  Eliminar	UKr53o2ojjw4fikudfnusuuga336	BTREE	Sí	No	password	0	A	Sí	
Editar  Renombrar  Eliminar	UK_email	BTREE	Sí	No	email	0	A	Sí	
Editar  Renombrar  Eliminar	username	BTREE	No	No	username	0	A	No	

Se le une en editar y se pone el nombre segun el campo unico

Fernando  
Agudelo  
Gutiérrez

## 12.7.2 Capturar el error para Mostrar los errores de campos unicos (nick o email ya existen) en el formulario

### 12.7.2.1 Manejo de errores relacionados con restricciones de base de datos (500):

JS useUsers.js X

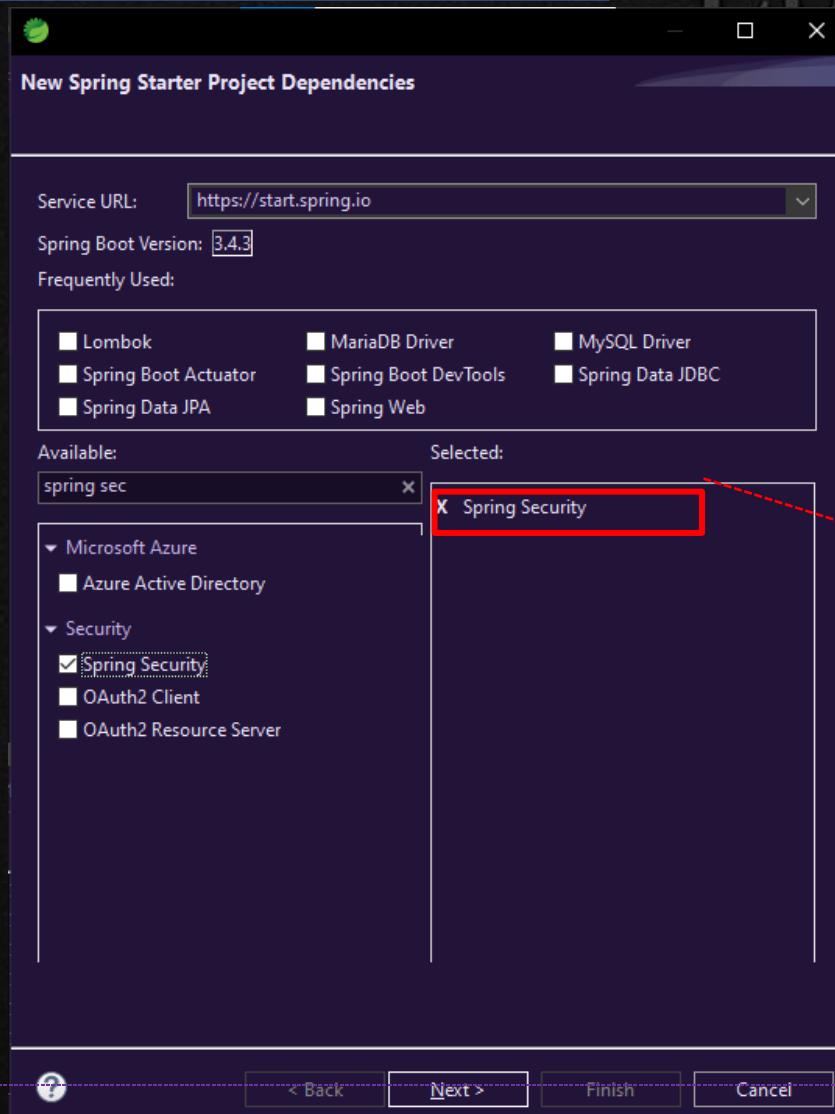
```
src > hooks > JS useUsers.js > [o] useUsers
21  export const useUsers = () => {
37    const handlerAddUser = async(user) =>
65      } catch (error) {
66
67      if (error.response && error.response.status==400) {
68          setErrors(error.response.data);
69      }
70
71      else if (error.response && error.response.status==500 &&
72          error.response.data?.message.includes('constraint')){
73
74
75
76
77
78
79
80
81      }
82
83      else{
84          throw error;
85      }
86
87  }
```

- `status == 500`: Verifica si el error tiene el código 500 (Internal Server Error). Este código puede ocurrir, por ejemplo, cuando hay una violación de restricciones en la base de datos.
- `error.response.data?.message.includes('constraint')`: Aquí se verifica si el mensaje de error contiene la palabra clave `constraint`, lo que sugiere que el error está relacionado con restricciones de base de datos (como claves únicas).

## 13. Backend Spring security JWT

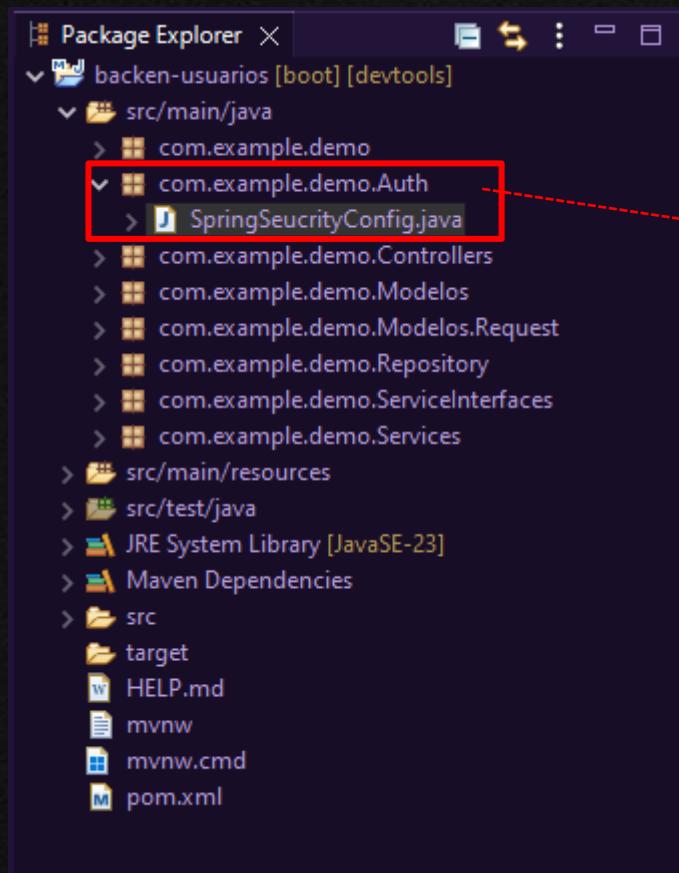
### 13.1 Configurar Spring security

#### 13.1.1 Instalar Dependencias



Instalar la depencia Spring Security

### 13.1.2 Crear la clase para Configuración de seguridad en Spring Boot con SecurityFilterChain



Se crea un paquete y dentro de este una class de java

### 13.1.3 Configurar la class

#### 13.1.3.1 Decirle a la class que va ser una configuracion

```
1 *SpringSeucrityConfig.java ×
2 package com.example.demo.Auth;
3
4 import org.springframework.context.annotation.Configuration;
5
6 @Configuration
7 public class SpringSeucrityConfig {
8 }
```

`@Configuration` es una anotación en Spring Framework que se utiliza para marcar una clase como una fuente de definiciones de beans. Esto significa que esa clase contendrá métodos que se encargan de registrar y configurar beans dentro del contexto de Spring. Esencialmente, le dice a Spring que la clase contiene configuración de la aplicación.

#### Propósito Principal:

- Se utiliza para definir beans de forma explícita en lugar de depender exclusivamente de la configuración basada en anotaciones como `@Component` o escaneos automáticos.

### 13.1.3.2 Método filterChain(HttpSecurity http)

```
1 package com.example.demo.Auth;
2
3 import org.springframework.context.annotation.Bean;
4 import org.springframework.context.annotation.Configuration;
5 import org.springframework.http.HttpMethod;
6 import org.springframework.security.config.annotation.web.builders.HttpSecurity;
7 import org.springframework.security.config.http.SessionCreationPolicy;
8 import org.springframework.security.web.SecurityFilterChain;
9
10 @Configuration
11 public class SpringSeucrityConfig {
12
14+ | SecurityFilterChain filterChain(HttpSecurity http) throws Exception {█
35
36     }
37 }
```

#### Método `filterChain(HttpSecurity http)`:

- Este método configura la seguridad de la aplicación utilizando el objeto `HttpSecurity`. Se devuelve un `SecurityFilterChain`, que define las reglas de autorización, autenticación y manejo de sesiones.

**SecurityFilterChain:** Es una interfaz que representa una cadena de filtros de seguridad en la aplicación.

**HttpSecurity:** Se utiliza para especificar la configuración de seguridad HTTP.

### 13.1.3.3 Anotacion Bean para el Método filterChain(HttpSecurity http)

```
1 *SpringSeucrityConfig.java X
2 import org.springframework.context.annotation.Bean;□
3
4
5 @Configuration
6 public class SpringSeucrityConfig {
7
8
9     @Bean
10    SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
11        return http
12            .cors(CorsConfiguration::enable)
13            .csrf(csrf ->
14                csrf().ignoringAntMatchers("http://localhost:8080/login")
15            )
16            .formLogin(form ->
17                form.loginPage("http://localhost:8080/login")
18                    .usernameParameter("username")
19                    .passwordParameter("password")
20            )
21            .httpBasic()
22            .build();
23    }
24}
```

**@Bean**: Indica que el método `filterChain` devuelve un bean que será administrado por el contenedor de Spring.

Fernando  
Agudelo  
Gutiérrez

#### 13.1.3.4 Configuracion de autorizacion del metodo filterChain(HttpSecurity http)

##### 13.1.3.4.1 Permitir el acceso a las páginas sin autenticacion

```
 9  
10 @Configuration  
11 public class SpringSeucrityConfig {  
12  
13     @Bean  
14     SecurityFilterChain filterChain(HttpSecurity http) throws Exception {  
15  
16         http.authorizeHttpRequests(auth -> auth  
17  
18  
19  
20  
21  
22  
23  
24     )|  
25  
26  
27         http.authorizeHttpRequests(auth -> auth  
28  
29  
30     1. http.authorizeHttpRequests : Aquí se está configurando la  
31         autorización de las solicitudes HTTP usando un método funcional  
32         (lambda). El objeto auth se utiliza para definir las reglas específicas  
33         de autorización.  
34  
35     }  
36  
37 }
```

1. `http.authorizeHttpRequests` : Aquí se está configurando la autorización de las solicitudes HTTP usando un método funcional (lambda). El objeto `auth` se utiliza para definir las reglas específicas de autorización.

```
 9  
10 @Configuration  
11 public class SpringSeucrityConfig {  
12  
13     @Bean  
14     SecurityFilterChain filterChain(HttpSecurity http) throws Exception {  
15  
16         http.authorizeHttpRequests(auth ->  
17             .requestMatchers(HttpMethod.GET, "/users").permitAll()  
18             // Permite acceso público al endpoint /users para solicitudes GET  
19         )  
20  
21         .requestMatchers(HttpMethod.GET, "/users").permitAll()  
22     }  
23  
24     .requestMatchers(HttpMethod.GET, "/users").permitAll()  
25  
26     .requestMatchers(HttpMethod.GET, "/users").permitAll()  
27  
28     .requestMatchers(HttpMethod.GET, "/users").permitAll()  
29  
30     .requestMatchers(HttpMethod.GET, "/users").permitAll()  
31  
32     .requestMatchers(HttpMethod.GET, "/users").permitAll()  
33  
34     .requestMatchers(HttpMethod.GET, "/users").permitAll()  
35  
36 }  
37
```

2. `.requestMatchers(HttpMethod.GET, "/users")` : Esta línea especifica que se está configurando una regla para las solicitudes HTTP **GET** dirigidas al endpoint `"/users"`. Es decir, cualquier solicitud que sea un **GET** y que tenga como ruta `"/users"` será evaluada.
3. `.permitAll()` : Esta parte indica que cualquier usuario (autenticado o no) tiene **permiso** para acceder a este endpoint. En otras palabras, no se requiere autenticación para realizar solicitudes GET al endpoint `"/users"`.

### 13.1.3.4.2 Restringir el acceso a las páginas que requieran autenticación

```
1 *SpringSecurityConfig.java X
2
3
4
5
6
7
8
9
10 @Configuration
11 public class SpringSecurityConfig {
12
13     @Bean
14     SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
15
16         http.authorizeHttpRequests(auth -> auth
17
18             .requestMatchers(HttpMethod.GET, "/users").permitAll()
19             // Permite acceso público al endpoint /users para solicitudes GET
20
21             .anyRequest().authenticated()
22             // Requiere autenticación para todas las demás solicitudes
23
24         )
25
26         .anyRequest().authenticated()
27
28
29
30
31
32
33
34
35
36     }
37 }
```

4. `.anyRequest()` : Esta línea define una regla genérica para **cualquier otra solicitud** que no cumpla con las reglas anteriores. Es decir, cualquier solicitud que no sea un **GET** a `"/users"` será evaluada aquí.

5. `.authenticated()` : Aquí se especifica que cualquier otra solicitud requiere que el usuario esté **autenticado**. Si el usuario no está autenticado, la solicitud será denegada.

Para cualquier otro tipo de solicitud (por ejemplo, POST, PUT, DELETE) o cualquier otro endpoint diferente a `"/users"`, se requiere que el usuario esté **autenticado**.

### 13.1.3.4.3 Desactivación de CSRF

```
*SpringSecurityConfig.java X
9
10 @Configuration
11 public class SpringSecurityConfig {
12
13     @Bean
14     SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
15
16         http.authorizeHttpRequests(auth -> auth
17
18             .requestMatchers(HttpMethod.GET, "/users").permitAll()
19             // Permite acceso público al endpoint /users para solicitudes GET
20
21             .anyRequest().authenticated()
22             // Requiere autenticación para todas las demás solicitudes
23
24     )
25
26     .csrf(csrf -> csrf.disable())
27     // Desactiva la protección CSRF porque no es necesaria en APIs REST
28
29     .csrf(...):
30
31         • La llamada al método .csrf() hace referencia a la configuración
32             de la protección contra ataques de tipo CSRF (Cross-Site Request
33             Forgery) en Spring Security.
34
35         • CSRF es un tipo de ataque en el que un atacante puede engañar a
36             un usuario autenticado para que realice acciones no deseadas en
37             una aplicación web donde ya está autenticado.
38
39     csrf -> csrf.disable():
40
41         • Este fragmento utiliza una expresión lambda (introducida en Java
42             8) para deshabilitar la protección CSRF.
43
44         • csrf.disable() indica que la protección contra ataques CSRF está
45             siendo desactivada explícitamente.
```

La protección CSRF (Cross-Site Request Forgery) se desactiva porque las APIs REST suelen manejarse desde clientes confiables y utilizan autenticación basada en tokens, por lo que no es necesario habilitar esta protección.

#### 13.1.3.4.4 Gestión de Sesiones

```
1 *SpringSecurityConfig.java X
2
3
4
5
6
7
8
9
10 @Configuration
11 public class SpringSecurityConfig {
12
13     @Bean
14     SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
15
16         http.authorizeHttpRequests(auth -> auth
17
18             .requestMatchers(HttpMethod.GET, "/users").permitAll()
19             // Permite acceso público al endpoint /users para solicitudes GET
20
21             .anyRequest().authenticated()
22             // Requiere autenticación para todas las demás solicitudes
23
24         )
25
26         .csrf(csrf -> csrf.disable())
27         // Desactiva la protección CSRF porque no es necesaria en APIs REST
28
29         .sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS));
30         // Configura la sesión como stateless, útil para APIs basadas en tokens
31
32
33         .sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
34
35
36
37 }
```

- **sessionManagement:** Permite configurar cómo se manejarán las sesiones en la aplicación.
- **SessionCreationPolicy.STATELESS:** Configura la gestión de sesiones como "sin estado". Esto significa que el servidor no almacenará información sobre la sesión del usuario, lo cual es adecuado para APIs que utilizan tokens (como JWT) para la autenticación.

#### 13.1.3.4.5 Construcción del SecurityFilterChain

```
*SpringSecurityConfig.java X
9
10 @Configuration
11 public class SpringSecurityConfig {
12
13     @Bean
14     SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
15
16         http.authorizeHttpRequests(auth -> auth
17
18             .requestMatchers(HttpMethod.GET, "/users").permitAll()
19             // Permite acceso público al endpoint /users para solicitudes GET
20
21             .anyRequest().authenticated()
22             // Requiere autenticación para todas las demás solicitudes
23
24         )|
25
26         .csrf(csrf -> csrf.disable())
27         // Desactiva la protección CSRF porque no es necesaria en APIs REST
28
29         .sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS));
29         // Configura la sesión como stateless, útil para APIs basadas en tokens
30
31         return http.build();
32         // Construye y devuelve el objeto SecurityFilterChain
33     }
34
35
36
37 }
```

return http.build();

- **build()**: Construye y devuelve el objeto `SecurityFilterChain` configurado, que se utilizará para aplicar las reglas de seguridad definidas.

### 1. `http` :

- Es una instancia de un objeto que generalmente se utiliza para configurar algo relacionado con HTTP. Por ejemplo, en **Spring Security**, `http` es una instancia de `HttpSecurity`, que se usa para definir las reglas de seguridad de las solicitudes HTTP.

### 2. `.build()` :

- El método `build()` suele ser utilizado en patrones de diseño como el **Builder Pattern**. Este método se encarga de construir o finalizar la configuración y devolver un objeto completamente configurado y listo para ser utilizado.
- En el caso de **Spring Security**, por ejemplo, `http.build()` devuelve un objeto tipo `SecurityFilterChain`, que representa la cadena de filtros configurados para manejar las solicitudes HTTP.

### 3. `return` :

- La palabra clave `return` indica que el método donde se encuentra este fragmento de código está devolviendo el resultado del método `http.build()`.
- Es decir, el objeto construido por `build()` será el valor de retorno del método.

## 13.2 Crear y Modificar el filtro por defecto de autenticacion para login de spring security para que funcione con Api res (JSON) por medio de Post en el cuerpo del request

En spring security ya existe este filtro de autenticacion pero por defecto es por formulario pero no es api res, y como estamos utilizando api res Necesitamos configurarlo para que el login venga un json en el cuerpo del request mediante Post

### 13.2.1 Crear class para Filtro de Autenticación JWT

The screenshot shows a Java project structure in a code editor. On the left, the project tree for 'backend-usuarios' shows a package named 'com.example.demo.Auth.Filters'. Inside this package, a file named 'JwtAuthenticationFilter.java' is being edited. The code is as follows:

```
1 package com.example.demo.Auth.Filters;
2
3 import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;
4
5 public class JwtAuthenticationFilter extends UsernamePasswordAuthenticationFilter {
6
7 }
```

A red box highlights the package declaration 'com.example.demo.Auth.Filters'. A red arrow points from the bottom of the screen towards the package name in the code editor.

Se crea un paquete dentro

## 13.2.2 Configurar la class para Filtro de Autenticación JWT

### 13.2.2.1 Heredar la class UsernamePasswordAuthenticationFilter para poder Personalizar el comportamiento del filtro de autenticacion



JwtAuthenticationFilter.java X

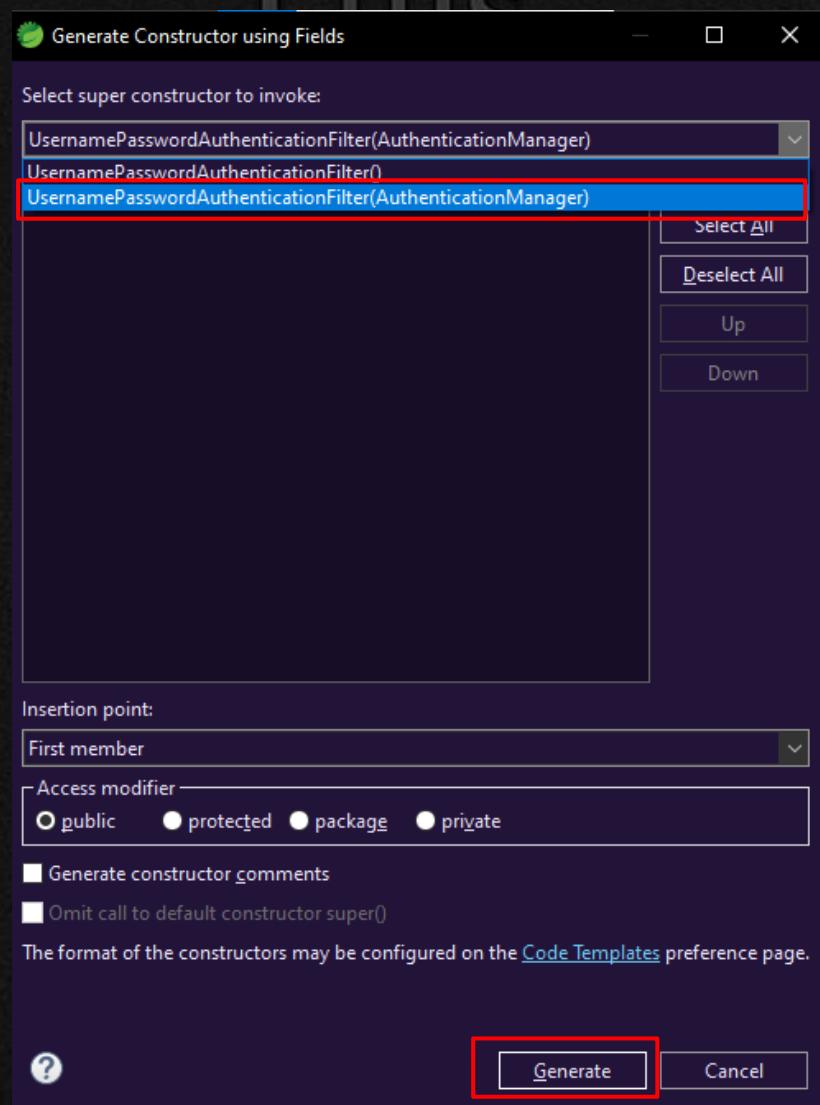
```
1 package com.example.demo.Auth.Filters;  
2  
3 import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;  
4  
5 public class JwtAuthenticationFilter extends UsernamePasswordAuthenticationFilter {  
6  
7 }  
8
```

#### ¿Qué es UsernamePasswordAuthenticationFilter ?

- Es una clase proporcionada por **Spring Security**. Esta clase se utiliza principalmente para manejar el proceso de autenticación basado en nombre de usuario y contraseña.
- Por defecto, esta clase intercepta las solicitudes HTTP (generalmente en la ruta `/login`) y verifica las credenciales del usuario.
- Si las credenciales son válidas, genera un token de autenticación y lo almacena en el contexto de seguridad de Spring.

13.2.2.2 Generar metodo constructor y sobre escribir los metodos de la class heredada para poder personalizar el comportamiento del filtro de autenticacion

### 13.2.2.2.1 Metodo constructor



```
JwtAuthenticationFilter.java ×  
1 package com.example.demo.Auth.Filthers;  
2  
3+import java.io.IOException;  
4  
5 public class JwtAuthenticationFilter extends UsernamePasswordAuthenticationFilter {  
6  
7  
8  
9+    public JwtAuthenticationFilter(AuthenticationManager authenticationManager) {  
10        super(authenticationManager);  
11    }  
12  
13+    @Override  
14        protected void successfulAuthentication(HttpServletRequest request,  
15            HttpServletResponse response, Authentication authentication)  
16            throws IOException {  
17                String jwt = generateToken(authentication);  
18                response.addHeader("Authorization", "Bearer " + jwt);  
19            }  
20  
21+    private String generateToken(Authentication authentication){  
22        String username = authentication.getName();  
23        String password = authentication.getCredentials().toString();  
24+        return null;  
25    }  
26  
27+    @Override  
28        protected void unsuccessfulAuthentication(HttpServletRequest request,  
29            HttpServletResponse response, AuthenticationException authenticationException)  
30            throws IOException {  
31                response.sendError(HttpServletResponse.SC_UNAUTHORIZED,  
32                    "Unauthorized");  
33            }  
34+    @Override  
35        protected void handleSuccess(HttpServletRequest request,  
36            HttpServletResponse response, Authentication authentication)  
37            throws IOException {  
38                String jwt = generateToken(authentication);  
39                response.addHeader("Authorization", "Bearer " + jwt);  
40            }  
41+    @Override  
42        protected void handleFailure(HttpServletRequest request,  
43            HttpServletResponse response, AuthenticationException authenticationException)  
44            throws IOException {  
45                response.sendError(HttpServletResponse.SC_UNAUTHORIZED,  
46                    "Unauthorized");  
47            }  
48    }  
49
```

Método constructor

### 13.2.2.2.1.1 Pasar metodo constructor como atributo

```
JwtAuthenticationFilter.java X
1 package com.example.demo.Auth.Filthers;
2
3 import java.io.IOException;
4
5 public class JwtAuthenticationFilter extends UsernamePasswordAuthenticationFilter {
6
7
8     private AuthenticationManager authenticationManager;
9     public JwtAuthenticationFilter(AuthenticationManager authenticationManager) {
10         this.authenticationManager=authenticationManager;
11     }
12 }
```

Se está pasando como atributo

#### Declaración de una variable privada:

java Copiar

```
private AuthenticationManager authenticationManager;
```

- Aquí se está declarando un atributo o variable de instancia llamado `authenticationManager` de tipo `AuthenticationManager`.
- La palabra clave `private` significa que esta variable solo puede ser accedida dentro de la misma clase. Es decir, no es visible ni accesible desde otras clases directamente.
- `AuthenticationManager` es probablemente una clase o interfaz que se utiliza para gestionar la autenticación de usuarios en la aplicación. Este tipo de clase es común en frameworks como Spring Security.

## 2. Constructor de la clase `JwtAuthenticationFilter`:

java

 Copiar

```
public JwtAuthenticationFilter(AuthenticationManager  
authenticationManager) {  
    this.authenticationManager = authenticationManager;  
}
```

- Aquí se define un **constructor** para la clase `JwtAuthenticationFilter`. Un constructor es un método especial que se llama automáticamente cuando se crea un objeto de esta clase.
- Este constructor recibe un parámetro llamado `authenticationManager` de tipo `AuthenticationManager`.
- Dentro del constructor, se asigna el valor del parámetro `authenticationManager` a la variable de instancia `this.authenticationManager`. El uso de `this` es necesario para diferenciar entre la variable de instancia y el parámetro del constructor, ya que ambos tienen el mismo nombre.

### Propósito del código:

- Este fragmento de código es típico en clases que necesitan recibir dependencias externas (en este caso, una instancia de `AuthenticationManager`) para funcionar correctamente. Este patrón es conocido como **inyección de dependencias**.
- En este caso, parece que la clase `JwtAuthenticationFilter` necesita un objeto del tipo `AuthenticationManager` para realizar tareas relacionadas con la autenticación, probablemente en el contexto de un sistema que utiliza JWT (JSON Web Tokens) para manejar sesiones y seguridad.

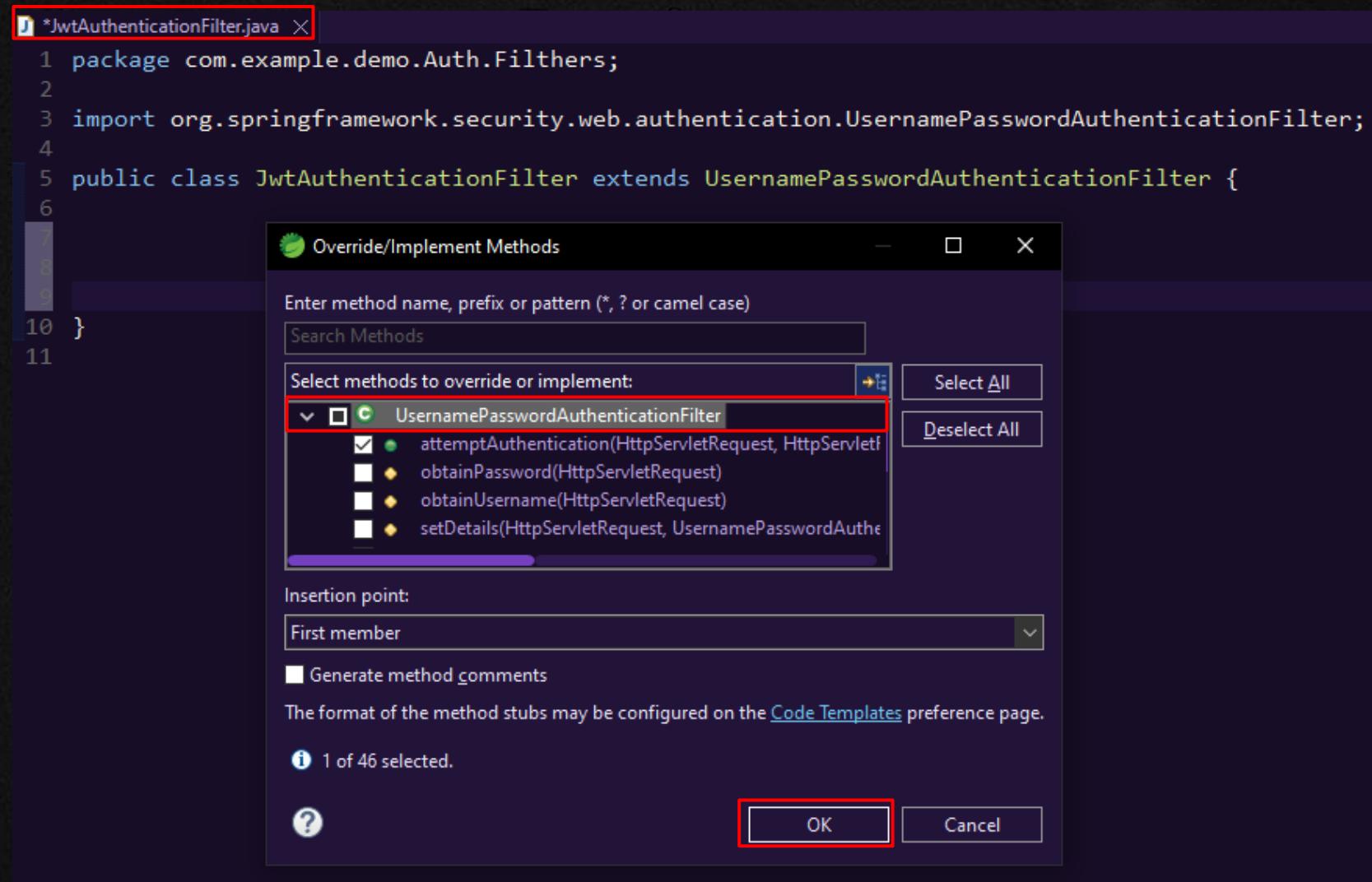
### Resumen:

El código define un campo privado para almacenar una instancia de `AuthenticationManager` y un constructor que permite inicializar esta instancia al crear un objeto de la clase `JwtAuthenticationFilter`. Esto es parte de un patrón común para manejar dependencias y garantizar que la clase tenga acceso a los recursos necesarios para cumplir su propósito (en este caso, autenticación).

## 13.2.2.2.2 Sobre escribir los métodos que se requieren del class heredada

### 13.2.2.2.1 Método para realizar autenticación

UsernamePasswordAuthenticationFilter



clic dererecho entro de la class seleccionamos source y luego override/implements metodos y seleccionamos el método

UsernamePasswordAuthenticationFilter

```
J *JwtAuthenticationFilter.java ×
1 package com.example.demo.Auth.Filthers;
2
3+import java.io.IOException;◻
14
15 public class JwtAuthenticationFilter extends UsernamePasswordAuthenticationFilter {
16
17
18     private AuthenticationManager authenticationManager;
19+    public JwtAuthenticationFilter(AuthenticationManager authenticationManager) {◻
20
21
22
23@Override
24    public Authentication attemptAuthentication(HttpServletRequest request, HttpServletResponse response)
25        throws AuthenticationException {
26        // TODO Auto-generated method stub
27        return super.attemptAuthentication(request, response);
28
29}
```

# Gutiérrez

## 13.2.2.2.2 Método si todo sale bien en la autenticación `successfulAuthentication`

The screenshot shows an IDE interface with a code editor and a 'Override/Implement Methods' dialog box.

**Code Editor:** The file `JwtAuthenticationFilter.java` is open. The class definition is as follows:

```
1 package com.example.demo.Auth.Filters;
2
3 import org.springframework.security.core.Authentication;
4 import org.springframework.security.core.AuthenticationException;
5 import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;
6
7 import jakarta.servlet.http.*;
8 import jakarta.servlet.http.*;
9
10 public class JwtAuthenticationFilter extends UsernamePasswordAuthenticationFilter {
11
12     @Override
13     public Authentication attemptAuthentication(
14         HttpServletRequest request,
15         HttpServletResponse response)
16         throws AuthenticationException {
17         // TODO Auto-generated method stub
18         return super.attemptAuthentication(request, response);
19     }
20
21 }
22 }
```

**Override/Implement Methods Dialog:** This dialog allows selecting methods to override or implement. The 'successfulAuthentication' method is selected (indicated by a checked checkbox).

Method	Description
<code>setSecurityContextHolderStrategy(SecurityContextHolder)</code>	
<code>setSecurityContextRepository(SecurityContextRepository)</code>	
<code>setSessionAuthenticationStrategy(SessionAuthenticationStrategy)</code>	
<code>successfulAuthentication(HttpServletRequest, HttpServletResponse)</code>	Selected
<code>unsuccessfulAuthentication(HttpServletRequest, HttpServletResponse)</code>	
<code>GenericFilterBean</code>	
<code>Object</code>	

Other options in the dialog include:

- Search Methods input field: `Enter method name, prefix or pattern (*, ? or camel case)`
- Select All and Deselect All buttons
- Insertion point dropdown: `After 'attemptAuthentication(HttpServletRequest, HttpServletResponse)'`
- Generate method comments checkbox
- A message at the bottom: `The format of the method stubs may be configured on the Code Templates preference page.`
- Information message: `i 1 of 45 selected.`
- OK and Cancel buttons

```
 1 package com.example.demo.Auth.Filters;
 2
 3+import java.io.IOException;
 4
 5 public class JwtAuthenticationFilter extends UsernamePasswordAuthenticationFilter {
 6
 7
 8     private AuthenticationManager authenticationManager;
 9+    public JwtAuthenticationFilter(AuthenticationManager authenticationManager) {
10
11
12+        super.setAuthenticationManager(authenticationManager);
13+        super.setFilterProcessesUrl("/api/authenticate");
14
15
16
17
18     private AuthenticationManager authenticationManager;
19+    public JwtAuthenticationFilter(AuthenticationManager authenticationManager) {
20
21
22+        super.setAuthenticationManager(authenticationManager);
23+        super.setFilterProcessesUrl("/api/authenticate");
24+        public Authentication attemptAuthentication(HttpServletRequest request, HttpServletResponse response) {
25
26
27+            @Override
28+            protected void successfulAuthentication(HttpServletRequest request, HttpServletResponse response,
29+                FilterChain chain, Authentication authResult) throws IOException, ServletException {
30+                    // TODO Auto-generated method stub
31+                    super.successfulAuthentication(request, response, chain, authResult);
32+                }
33
34
35
36
37 }
```

### 13.2.2.2.3 Meotdo si todo sale malen la autenticacion `unsuccessfulAuthentication`

The screenshot shows an IDE interface with a Java code editor and a 'Override/Implement Methods' dialog box.

**Code Editor:** The file `JwtAuthenticationFilter.java` is open. It contains the following code:

```
1 import org.springframework.security.core.AuthenticationException;
2 import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;
3
4 import jakarta.servlet.FilterChain;
5 import jakarta.servlet.ServletException;
6 import jakarta.servlet.http.HttpServletRequest;
7 import jakarta.servlet.http.HttpServletResponse;
8
9 public class JwtAuthenticationFilter extends UsernamePasswordAuthenticationFilter {
10
11     @Override
12     public Authentication attemptAuthentication(HttpServletRequest request,
13                                                 HttpServletResponse response)
14         throws AuthenticationException {
15         // TODO Auto-generated method stub
16         return super.attemptAuthentication(request, response);
17     }
18
19     @Override
20     protected void successfulAuthentication(
21         HttpServletRequest request,
22         HttpServletResponse response,
23         FilterChain chain,
24         Authentication authentication)
25         throws AuthenticationException {
26         // TODO Auto-generated method stub
27         super.successfulAuthentication(request, response, chain, authentication);
28     }
29
30
31
32
33 }
34 }
```

**Override/Implement Methods Dialog:** A modal dialog titled 'Override/Implement Methods' is displayed over the code editor. It contains the following fields:

- Search Methods input field (empty)
- Select methods to override or implement:
  - setRememberMeServices(RememberMeServices)
  - setSecurityContextHolderStrategy(SecurityContextHolderStrategy)
  - setSecurityContextRepository(SecurityContextRepository)
  - setSessionAuthenticationStrategy(SessionAuthenticationStrategy)
  - unsuccessfulAuthentication(HttpServletRequest, HttpServletResponse)
- Buttons: 'Select All' and 'Deselect All'
- Insertion point dropdown: 'After 'successfulAuthentication(HttpServletRequest, HttpServletResponse, FilterChain, Authentication)'
- Checkboxes: 'Generate method comments' (unchecked)
- Information message: 'The format of the method stubs may be configured on the [Code Templates](#) preference page.'
- Status message: '1 of 44 selected.'
- Buttons at the bottom: '?', 'OK' (highlighted with a red box), and 'Cancel'

```
*JwtAuthenticationFilter.java ×
1 package com.example.demo.Auth.Filthers;
2
3+import java.io.IOException;
4
5 public class JwtAuthenticationFilter extends UsernamePasswordAuthenticationFilter {
6
7
8     private AuthenticationManager authenticationManager;
9+    public JwtAuthenticationFilter(AuthenticationManager authenticationManager) {
10
11
12+        public Authentication attemptAuthentication(HttpServletRequest request, HttpServletResponse response)
13
14+        protected void successfulAuthentication(HttpServletRequest request, HttpServletResponse response, 
15
16
17+        @Override
18+        protected void unsuccessfulAuthentication(HttpServletRequest request, HttpServletResponse response,
19
20
21+            AuthenticationException failed) throws IOException, ServletException {
22
23
24+            // TODO Auto-generated method stub
25
26
27+            super.unsuccessfulAuthentication(request, response, failed);
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46 }
47 }
```

### 13.3 Añadir Configuracion al metodo filterChain: Añadir filtro personalizado para manejar autenticacion utilizando JWT

#### 13.3.1 Inyectar class AuthenticationConfiguration

```
1 *SpringSeucrityConfig.java X
2
3 public class SpringSeucrityConfig {
4
5     @Autowired
6     private AuthenticationConfiguration authenticationConfiguration;
7
8     @Bean
9     SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
10
11         http.authorizeHttpRequests( auth -> auth
12
13             // Permite acceso público al endpoint /users para solicitudes GET
14             .requestMatchers(HttpMethod.GET, "/users").permitAll()
15
16             // Requiere autenticación para todas las demás solicitudes
17             .anyRequest().authenticated()
18         )
19
20
21
22         // Desactiva la protección CSRF porque no es necesaria en APIs REST
23         .csrf(csrf -> csrf.disable())
24
25
26         // Configura la sesión como stateless, útil para APIs basadas en tokens
27         .sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS));
28
29
30
31         // Construye y devuelve el objeto SecurityFilterChain
32         return http.build();
33     }
34
35     @Autowired:
36
37
38
39
40 }
```

- Es una **anotación de Spring Framework** que se utiliza para realizar la **inyección de dependencias** de forma automática.

```
private AuthenticationConfiguration authenticationConfiguration; :
```

- Este es un campo privado de la clase que contiene este código.
- El tipo de este campo es `AuthenticationConfiguration`, que es una clase proporcionada por Spring Security y que se utiliza para trabajar con la configuración de autenticación en una aplicación Spring Security.
- `AuthenticationConfiguration` permite acceder a objetos relacionados con la autenticación, como el `AuthenticationManager`, que es responsable de gestionar los procesos de autenticación en{ "answer": "- \*\*" la aplicación.

## Aguadejo

En pocas palabras el filtro personalizado que se va crear para manejar la autenticacion por token JWT requiere un `AuthenticationManager`, por eso se inyecta y como ya lo tenemos en la class `JwtAuthenticationFilter` donde hereda de la class `UsernamePasswordAuthenticationFilter` y con el método constructor que se le paso como atributo podemos acceder a objetos de esta class

### 13.3.2 Agregar filtro personalizado de autenticacion al metodo

```
*SpringSecurityConfig.java ×
12 public class SpringSecurityConfig {
13
14     @Autowired
15     private AuthenticationConfiguration authenticationConfiguration;
16
17     @Bean
18     SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
19
20         http.authorizeHttpRequests( auth -> auth
21
22             // Permite acceso público al endpoint /users para solicitudes GET
23             .requestMatchers(HttpMethod.GET, "/users").permitAll()
24
25             // Requiere autenticación para todas las demás solicitudes
26             .anyRequest().authenticated()
27         )
28
29         //Añade un filtro personalizado para manejar la autenticación utilizando tokens JWT (JSON Web Tokens).
30         .addFilter(new JwtAuthenticationFilter(authenticationConfiguration.getAuthenticationManager()))
31
32         // Desactiva la protección CSRF porque no es necesaria en APIs REST
33         .csrf(csrf -> csrf.disable())
34
35         // Configura la sesión como stateless, útil para APIs basadas en tokens
36         .sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS));
37
38         // Construye y devuelve el objeto SecurityFilterChain
39         return http.build();
40     }
}
```

.addFilter(...):

- Este método se utiliza para añadir un filtro personalizado a la cadena de filtros de Spring Security. Los filtros son componentes que interceptan las solicitudes HTTP antes de que lleguen a los controladores, y permiten realizar tareas como autenticación, autorización o registro de actividad.

```
new JwtAuthenticationFilter(...):
```

- Aquí se está creando una nueva instancia del filtro `JwtAuthenticationFilter`. Este filtro es una clase personalizada (creada por el desarrollador) que probablemente extiende alguna clase base de filtros de Spring Security, como `OncePerRequestFilter` o `UsernamePasswordAuthenticationFilter`.
- El propósito principal de este filtro es interceptar las solicitudes entrantes y verificar si el cliente ha proporcionado un token JWT válido en los encabezados HTTP (generalmente en el encabezado `Authorization`).

```
authenticationConfiguration.getAuthenticationManager():
```

- Este método obtiene una instancia del `AuthenticationManager` configurado en Spring Security.
- El `AuthenticationManager` es un componente central en Spring Security que se encarga de manejar el proceso de autenticación. En este caso, se está pasando al filtro para que pueda delegar la validación del token JWT y la autenticación del usuario al `AuthenticationManager`.

## Flujo típico

- Cuando una solicitud HTTP llega al servidor, pasa por la cadena de filtros configurada en Spring Security.
- El `JwtAuthenticationFilter` intercepta la solicitud y verifica si contiene un token JWT en el encabezado `Authorization`.
- Si el token está presente y es válido:
  - El filtro extrae la información del usuario del token (como el nombre de usuario, roles, etc.).
  - Crea un objeto de autenticación (`Authentication`) y lo pasa al `AuthenticationManager` para que lo valide.
  - Si todo es correcto, la solicitud sigue su curso y llega al controlador correspondiente.
- Si el token no está presente o no es válido:
  - El filtro puede rechazar la solicitud devolviendo una respuesta con un código de error HTTP (como 401 Unauthorized).

### 13.4.1 Modificar e Implementar el metodo de autenticacion para realizar autenticacion `UsernamePasswordAuthenticationFilter`

#### 1. Propósito del Método:

El método `attemptAuthentication` es responsable de manejar el proceso inicial de autenticación. Es invocado automáticamente por Spring Security cuando el filtro detecta una solicitud de inicio de sesión (como un `POST` a una URL específica de autenticación).

Dentro del método:

- Se obtienen las credenciales del usuario desde la solicitud HTTP.
- Se crea un token de autenticación que representa estas credenciales.
- Este token se pasa al `AuthenticationManager`, que valida las credenciales y autentica al usuario.

### 13.4.1.1 Declarar variables locales

```
JwtAuthenticationFilter.java X
29
30    @Override
31    public Authentication attemptAuthentication(HttpServletRequest request, HttpServletResponse response)
32        throws AuthenticationException {
33        User user= null;
34        String username= null;
35        String password=null;
36
37
38
39
40
41
42
43    }
```

User es la class de nuestro modelo o Entity que vamos a utilizar

- **User user:** Se utiliza para almacenar la información del usuario que se deserializa de la solicitud.

En estas variables se van a almacenar el usuario y la contraseña del objeto user del entity

- **String username:** Almacena el nombre de usuario extraído del objeto User .
- **String password:** Almacena la contraseña extraída del objeto User .

**Inicialización previa:** En muchos lenguajes como Java, es una buena práctica inicializar las variables antes de usarlas, aunque sea con el valor null , para evitar errores de compilación o ejecución.

**Asignación futura:** Más adelante en el programa, estas variables probablemente recibirán valores concretos. Por ejemplo:

### 13.4.1.2 Deserialización del Usuario

```
JwtAuthenticationFilter.java X
29
30    @Override
31    public Authentication attemptAuthentication(HttpServletRequest request, HttpServletResponse response)
32            throws AuthenticationException {
33        User user= null;
34        String username= null;
35        String password=null;
36
37        try {
38            [user = new ObjectMapper().readValue(request.getInputStream(), User.class);]
39
40
41
42
43    }
```

User instancia del modelo o Entity que estamos utilizando

`new ObjectMapper()` : Crea una instancia de la clase `ObjectMapper`, que es parte de la biblioteca Jackson. Esta clase se utiliza para realizar conversiones entre JSON y objetos Java.

**readValue:** Este método lee el cuerpo de la solicitud HTTP (stream de entrada) y lo convierte en un objeto `User`.

Parámetro del método `readValue`

`request.getInputStream()` obtiene el flujo de datos del cuerpo de la solicitud HTTP.

Parámetro del método `readValue`

`User.class` indica que se espera que el JSON sea convertido a una instancia de la clase `User`.

Hay que deserializar el JSON recibido con los datos el nombre y el password y convertirlos en un objeto de tipo User en este caso es la class User el modelo o Entity

```
J JwtAuthenticationFilter.java X
29
30     @Override
31     public Authentication attemptAuthentication(HttpServletRequest request, HttpServletResponse response)
32             throws AuthenticationException {
33         User user= null;
34         String username= null;
35         String password=null;
36
37         try {
38             user = new ObjectMapper().readValue(request.getInputStream(), User.class);
39             username= user.getUsername();
40             password=user.getPassword();
41
42         }
43     }
```

Una vez que el JSON ha sido deserializado correctamente y convertido en un objeto `User`, se extraen los valores de las propiedades `username` y `password` utilizando los métodos `getUsername()` y `getPassword()` (que son típicamente métodos `getter` definidos en la clase `User`).

En pocas palabras después de haber deserializado el JSON en un objeto `User`, se extraen los valores y se guardan en las variables locales creadas anteriormente con los métodos `guetters`.

```
JwtAuthenticationFilter.java X
31     public Authentication attemptAuthentication(HttpServletRequest request, HttpServletResponse response)
32             throws AuthenticationException {
33         User user= null;
34         String username= null;
35         String password=null;
36
37         try {
38             user = new ObjectMapper().readValue(request.getInputStream(), User.class);
39             username= user.getUsername();
40             password=user.getPassword();
41             logger.info("Usernmae desde el request InputStream(raw)" + username);
42             logger.info("Password desde el request InputStream(raw)" + password);
43         }
44
45         catch (StreamReadException e) {
46
47             e.printStackTrace();
48         } catch (DatabindException e) {
49
50             e.printStackTrace();
51         } catch (IOException e) {
52
53             e.printStackTrace();
54         }
55     }
56
57
58 }
```



**Excepciones:** Se manejan varias excepciones que pueden ocurrir durante la deserialización:

- **StreamReadException:** Ocurre si hay un problema al leer el stream.
- **DatabindException:** Ocurre si hay un problema al vincular los datos JSON a la clase `User`.
- **IOException:** Ocurre si hay un problema de entrada/salida.

### 13.4.1.3 Crear del Token de Autenticación

```
JwtAuthenticationFilter.java X
31     public Authentication attemptAuthentication(HttpServletRequest request, HttpServletResponse response)
32         throws AuthenticationException {
33         User user= null;
34         String username= null;
35         String password=null;
36
37         try {
38             user = new ObjectMapper().readValue(request.getInputStream(), User.class);
39             username= user.getUsername();
40             password=user.getPassword();
41             logger.info("Usernmae desde el request InputStrea(raw)" + username);
42             logger.info("Password desde el request InputStrea(raw)" + password);
43         }
44
45         catch (StreamReadException e) {
46
47             e.printStackTrace();
48         } catch (DatabindException e) {
49
50             e.printStackTrace();
51         } catch (IOException e) {
52
53             e.printStackTrace();
54         }
55
56         UsernamePasswordAuthenticationToken authThoken= new UsernamePasswordAuthenticationToken(username, password);
57
58     }
```

#### 1. UsernamePasswordAuthenticationToken :

- Es una clase proporcionada por Spring Security que se utiliza para representar un token de autenticación basado en un **nombre de usuario (username)** y una **contraseña (password)**.
- Este token es una implementación de la interfaz `Authentication`, que es la base para las credenciales en Spring Security.

#### 2. new UsernamePasswordAuthenticationToken(username, password) :

- Aquí se está creando una nueva instancia del token de autenticación.
- `username` : Representa el nombre de usuario del usuario que intenta autenticarse.
- `password` : Representa la contraseña proporcionada por el usuario durante el inicio de sesión.

#### Propósito del token:

- Este token es un contenedor temporal que almacena las credenciales proporcionadas por el usuario (nombre de usuario y contraseña) antes de que sean verificadas por el sistema de autenticación.
- Por ejemplo, este objeto será enviado al gestor de autenticación (`AuthenticationManager`) para validar las credenciales.

#### 13.4.1.4 Autenticación del token a través del AuthenticationManager:

```
JwtAuthenticationFilter.java X
31     public Authentication attemptAuthentication(HttpServletRequest request, HttpServletResponse response)
32             throws AuthenticationException {
33         User user= null;
34         String username= null;
35         String password=null;
36
37         try {
38             user = new ObjectMapper().readValue(request.getInputStream(), User.class);
39             username= user.getUsername();
40             password=user.getPassword();
41             logger.info("Usernmae desde el request InputStream(raw)" + username);
42             logger.info("Password desde el request InputStream(raw)" + password);
43         }
44
45         catch (StreamReadException e) {
46
47             e.printStackTrace();
48         } catch (DatabindException e) {
49
50             e.printStackTrace();
51         } catch (IOException e) {
52
53             e.printStackTrace();
54         }
55
56         UsernamePasswordAuthenticationToken authThoken= new UsernamePasswordAuthenticationToken(username, password);
57         return authenticationManager.authenticate(authThoken);
58     }
```

El token se pasa al `AuthenticationManager`, que delega la validación al `AuthenticationProvider` configurado en el sistema (por ejemplo, una base de datos, LDAP, etc.).

Si las credenciales son válidas:

- o El método devuelve un objeto de tipo `Authentication`, que representa al usuario autenticado.
- o Este objeto contiene detalles como el nombre de usuario, roles y permisos asignados.

Si las credenciales son inválidas:

- o El `AuthenticationManager` lanza una excepción de tipo `AuthenticationException`.

## 13.5 Modificar e Implementar el metodo successfulAuthentication

### Método successfulAuthentication :

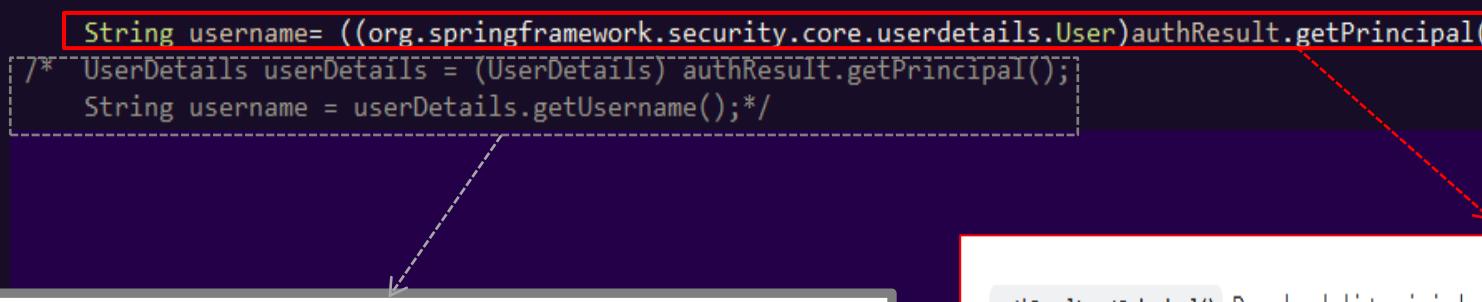
Este método se llama cuando la autenticación es exitosa. Recibe varios parámetros:

- `HttpServletRequest request` : La solicitud HTTP.
- `HttpServletResponse response` : La respuesta HTTP que se enviará al cliente.
- `FilterChain chain` : La cadena de filtros que se pueden aplicar.
- `Authentication authResult` : El resultado de la autenticación, que contiene información sobre el usuario autenticado.

### 13.5.1 Obtener el usuario

```
JwtAuthenticationFilter.java X
52
53
54  public Authentication attemptAuthentication(HttpServletRequest request, HttpServletResponse response) {
55
56  }
57
58  @Override
59
60  protected void successfulAuthentication(HttpServletRequest request, HttpServletResponse response,
61  FilterChain chain, Authentication authResult) throws IOException, ServletException {
62
63
64      String username= ((org.springframework.security.core.userdetails.User)authResult.getPrincipal()).getUsername();
65
66      /* UserDetails userDetails = (UserDetails) authResult.getPrincipal();
67      String username = userDetails.getUsername();*/
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82  }
```

Alternativa comentada: Usar `UserDetails` es más genérico y evita acoplamiento



- `authResult.getPrincipal()` : Devuelve el objeto principal que representa al usuario autenticado.
- Casting a `org.springframework.security.core.userdetails.User` : Se hace un casting para acceder a los métodos específicos de la clase `User` de Spring Security.
- `getUsername()` : Obtiene el nombre de usuario del objeto `User`.

### 13.5.2 Generación del Token Obtener el usuario

```
JwtAuthenticationFilter.java X
32
34+     public Authentication attemptAuthentication(HttpServletRequest request, HttpServletResponse response)
35
36     @Override
37
38     protected void successfulAuthentication(HttpServletRequest request, HttpServletResponse response,
39     FilterChain chain, Authentication authResult) throws IOException, ServletException {
40
41         String username= ((org.springframework.security.core.userdetails.User)authResult.getPrincipal()).getUsername();
42         /* UserDetails userDetails = (UserDetails) authResult.getPrincipal();
43             String username = userDetails.getUsername();*/
44         String originalInput="aca_va_el_token_JWT. " + username;
45         String token= Base64.getEncoder().encodeToString(originalInput.getBytes());
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
```

• `originalInput` : Es una cadena que combina un texto fijo con el nombre de usuario. En un caso real, aquí se generaría un token JWT válido.

• `Base64.getEncoder().encodeToString()` : Codifica la cadena `originalInput` en Base64, lo que simula la generación de un token.

Mas adelante se implementa con un token real de JWT.

- **Problema crítico:** Esto **no es un JWT real**, solo una cadena Base64
- **Estructura insegura:**
  - No tiene firma digital
  - Sin algoritmo de encriptación
  - No incluye claims estándar (expiración, roles, etc.)

### 13.5.3 Agregar el token al encabezado de la respuesta

```
JwtAuthenticationFilter.java X
32
34+     public Authentication attemptAuthentication(HttpServletRequest request, HttpServletResponse response)
35
36
37
38
39
40
41
42
43     protected void successfulAuthentication(HttpServletRequest request, HttpServletResponse response,
44         FilterChain chain, Authentication authResult) throws IOException, ServletException {
45
46         String username= ((org.springframework.security.core.userdetails.User)authResult.getPrincipal()).getUsername();
47         /* UserDetails userDetailsService = (UserDetails) authResult.getPrincipal();
48         String username = userDetailsService.getUsername();*/
49         String originalInput="aca_va_el_token_JWT. " + username;
50         String token= Base64.getEncoder().encodeToString(originalInput.getBytes());
51
52
53         response.addHeader("Authorization", "Bearer " + token);
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82 }
```

response.addHeader() : Agrega un encabezado HTTP a la respuesta. En este caso, el encabezado Authorization con el valor Bearer <token> .

#### Mas adelante se implementa se valida el token

```
response.addHeader("Authorization", "Bearer " + token);
```

- Formato estándar para tokens: Bearer <token>
- **Error común:** Usar este enfoque sin validar el token posteriormente

### 1. `response.addHeader(String name, String value)` :

- Este método pertenece a la clase `HttpServletResponse` en Java (comúnmente utilizada en aplicaciones web con Servlets o frameworks como Spring).
- Su propósito es agregar un encabezado HTTP a la respuesta. Los encabezados son partes de la comunicación HTTP que proporcionan información adicional sobre la solicitud o la respuesta.
- El método recibe dos parámetros:
  - `name` : El nombre del encabezado que deseas agregar. En este caso, el nombre del encabezado es `"Authorization"`.
  - `value` : El valor del encabezado. En este caso, el valor se forma concatenando la palabra `"Bearer "` con el contenido de la variable `token`.

### 3. `"Bearer " + token` :

- El valor del encabezado `Authorization` sigue el esquema de autenticación tipo *Bearer Token*. Esto significa que el cliente debe incluir un token en las solicitudes posteriores para demostrar su autenticación.
- `"Bearer "` es una palabra clave que indica el esquema de autenticación utilizado (*Bearer Token*).
- `token` es una variable que contiene el token de autenticación generado por el servidor. Este token suele ser una cadena alfanumérica única que identifica al usuario o a la sesión.

### 2. `"Authorization"` :

- Este es el nombre del encabezado HTTP que se está configurando. El encabezado `Authorization` se utiliza comúnmente para enviar credenciales de autenticación desde el cliente al servidor o viceversa.
- En este caso, parece que el servidor está enviando un token de autenticación al cliente en la respuesta.

#### 13.5.4 Creación del cuerpo de la respuesta

```
JwtAuthenticationFilter.java X
34+     public Authentication attemptAuthentication(HttpServletRequest request, HttpServletResponse response){}
62+     @Override
63     protected void successfulAuthentication(HttpServletRequest request, HttpServletResponse response,
64         FilterChain chain, Authentication authResult) throws IOException, ServletException {
65
66         String username= ((org.springframework.security.core.userdetails.User)authResult.getPrincipal()).getUsername();
67         /* UserDetails userDetailsService = (UserDetails) authResult.getPrincipal();
68         String username = userDetailsService.getUsername();*/
69         String originalInput="aca_va_el_token_JWT. " + username;
70         String token= Base64.getEncoder().encodeToString(originalInput.getBytes());
71
72         response.addHeader("Authorization", "Bearer " + token);
73
74         Map<String, Object> body= new HashMap<>();
75         body.put("token", token);
76         body.put("message", String.format("Hola %s, has iniciado sesión con éxito", username));
77         body.put("username", username);
78
79
80
81
82     }
```

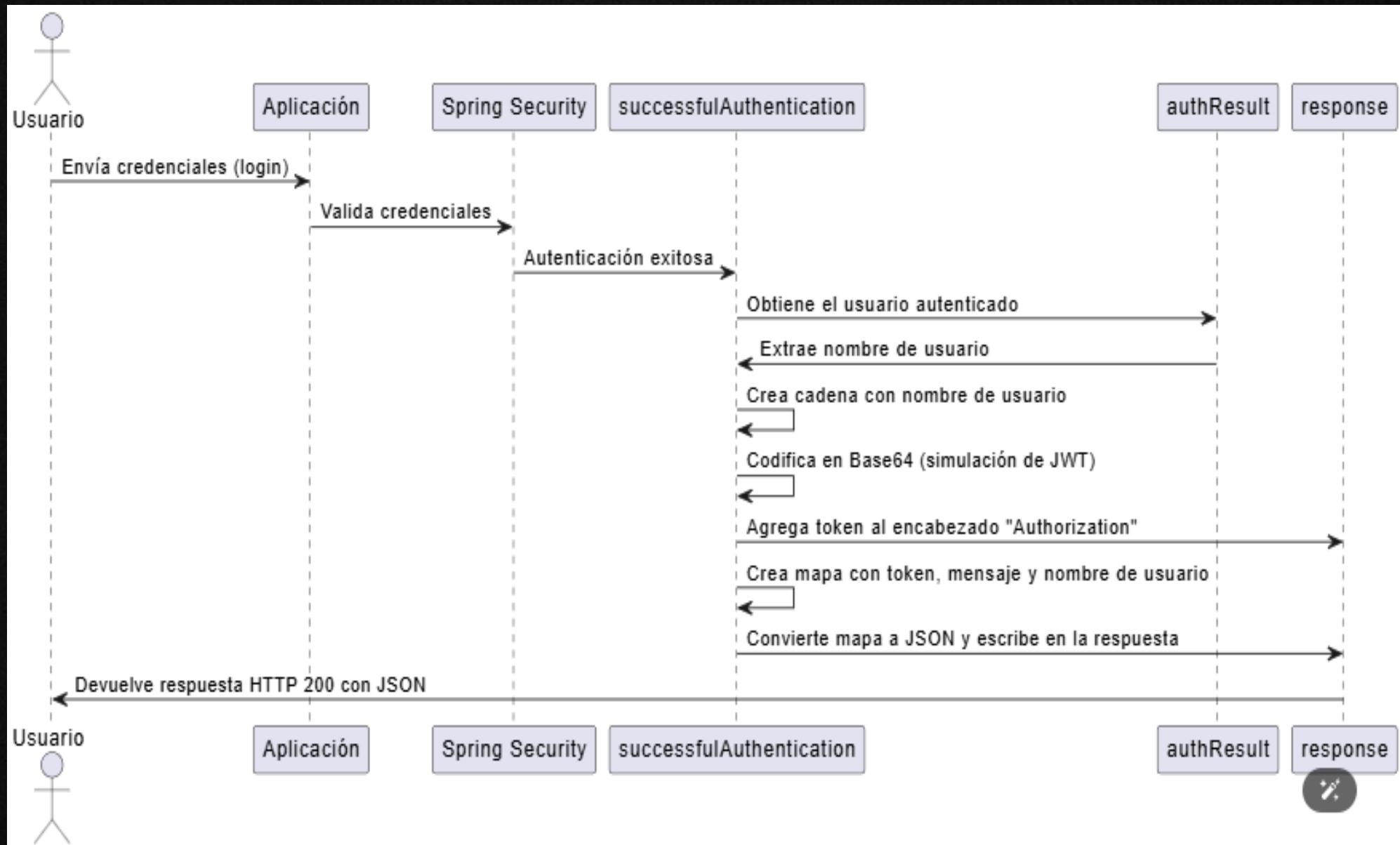
- `Map<String, Object> body` : Se crea un mapa para almacenar los datos que se enviarán en el cuerpo de la respuesta.
- `body.put()` : Agrega pares clave-valor al mapa, incluyendo el token, un mensaje de éxito y el nombre de usuario.

### 13.5.5 Escribir la respuesta en formato JSON

```
JwtAuthenticationFilter.java X
52
53     public Authentication attemptAuthentication(HttpServletRequest request, HttpServletResponse response) {
54
55         ...
56
57         @Override
58         protected void successfulAuthentication(HttpServletRequest request, HttpServletResponse response,
59             FilterChain chain, Authentication authResult) throws IOException, ServletException {
60
61             String username= ((org.springframework.security.core.userdetails.User)authResult.getPrincipal()).getUsername();
62             /* UserDetails userDetailsService = (UserDetails) authResult.getPrincipal();
63             String username = userDetailsService.getUsername();*/
64             String originalInput="aca_va_el_token_JWT. " + username;
65             String token= Base64.getEncoder().encodeToString(originalInput.getBytes());
66
67             response.addHeader("Authorization", "Bearer " + token);
68
69             Map<String, Object> body= new HashMap<>();
70             body.put("token", token);
71             body.put("message", String.format("Hola %s, has iniciado sesión con éxito", username));
72             body.put("username", username);
73
74             response.getWriter().write(new ObjectMapper().writeValueAsString(body));
75             response.setStatus(200);
76             response.setContentType("application/json");
77
78         }
79     }
```

- `response.getWriter().write()` : Escribe el cuerpo de la respuesta en formato JSON.
- `new ObjectMapper().writeValueAsString(body)` : Convierte el mapa `body` en una cadena JSON.
- `response.setStatus(200)` : Establece el código de estado HTTP a 200 (OK).
- `response.setContentType("application/json")` : Indica que el contenido de la respuesta es de tipo JSON.
- **Orden incorrecto:** Debe establecerse `setContentType()` antes de escribir el cuerpo
- **Mejor práctica:** Usar constantes como `HttpServletResponse.SC_OK` en lugar de 200

### 13.5.6 Diagrama visual del método



## 13.6 Modificar e Implementar el metodo

### unsuccessfulAuthentication

#### Método unsuccessfulAuthentication :

- Este método se ejecuta cuando la autenticación falla.
- Recibe tres parámetros:
  - `HttpServletRequest request` : La solicitud HTTP enviada por el cliente.
  - `HttpServletResponse response` : La respuesta HTTP que se enviará al cliente.
  - `AuthenticationException failed` : La excepción que se lanzó debido al fallo en la autenticación.

### 13.6.1 Cuerpo de la respuesta JSON

```
JwtAuthenticationFilter.java X
1 import java.io.IOException;
24
25 public class JwtAuthenticationFilter extends UsernamePasswordAuthenticationFilter {
26
27
28     private AuthenticationManager authenticationManager;
29+
30     public JwtAuthenticationFilter(AuthenticationManager authenticationManager) {
31
32
33+
34     public Authentication attemptAuthentication(HttpServletRequest request, HttpServletResponse response) {
35
36+
37     protected void successfulAuthentication(HttpServletRequest request, HttpServletResponse response, 
38
39     @Override
40     protected void unsuccessfulAuthentication(HttpServletRequest request, HttpServletResponse response,
41
42             AuthenticationException failed) throws IOException, ServletException {
43
44
45         Map<String, Object> body= new HashMap<>();
46
47
48         body.put("message", "error en la autenticacion username o password incorrecto");
49         body.put("error", failed.getMessage());
50
51
52
53
54
55
56
57
58
59
60
61
62
63+
64     protected void successfulAuthentication(HttpServletRequest request, HttpServletResponse response, 
65
66     @Override
67     protected void unsuccessfulAuthentication(HttpServletRequest request, HttpServletResponse response,
68
69             AuthenticationException failed) throws IOException, ServletException {
70
71
72         Map<String, Object> body= new HashMap<>();
73
74
75         body.put("message", "error en la autenticacion username o password incorrecto");
76         body.put("error", failed.getMessage());
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95 Se crea un HashMap para almacenar los datos que se enviarán en el cuerpo de la respuesta JSON.
96
97
98
99
100 }
```

Se agregan dos entradas al mapa:

- `"message"` : Un mensaje personalizado que indica que hubo un error en la autenticación debido a un nombre de usuario o contraseña incorrectos.
- `"error"` : El mensaje de la excepción ( `failed.getMessage()` ), que proporciona más detalles sobre el error específico.

```
JwtAuthenticationFilter.java X
1 import java.io.IOException;
2
3 public class JwtAuthenticationFilter extends UsernamePasswordAuthenticationFilter {
4
5
6     private AuthenticationManager authenticationManager;
7     public JwtAuthenticationFilter(AuthenticationManager authenticationManager) {
8
9
10    }
11
12    public Authentication attemptAuthentication(HttpServletRequest request, HttpServletResponse response) {
13
14        try {
15            return super.attemptAuthentication(request, response);
16        } catch (IOException e) {
17            throw new RuntimeException(e);
18        }
19    }
20
21    protected void successfulAuthentication(HttpServletRequest request, HttpServletResponse response, Authentication authResult) throws IOException {
22
23        Map<String, Object> body = new HashMap<>();
24
25        body.put("message", "error en la autenticacion username o password incorrecto");
26        body.put("error", authResult.getPrincipal());
27
28        response.getWriter().write(new ObjectMapper().writeValueAsString(body));
29        response.setStatus(401);
30        response.setContentType("application/json");
31
32    }
33
34    protected void unsuccessfulAuthentication(HttpServletRequest request, HttpServletResponse response, AuthenticationException failed) throws IOException, ServletException {
35
36        Map<String, Object> body = new HashMap<>();
37
38        body.put("message", "error en la autenticacion username o password incorrecto");
39        body.put("error", failed.getMessage());
40
41        response.getWriter().write(new ObjectMapper().writeValueAsString(body));
42        response.setStatus(401);
43        response.setContentType("application/json");
44
45    }
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100}
```

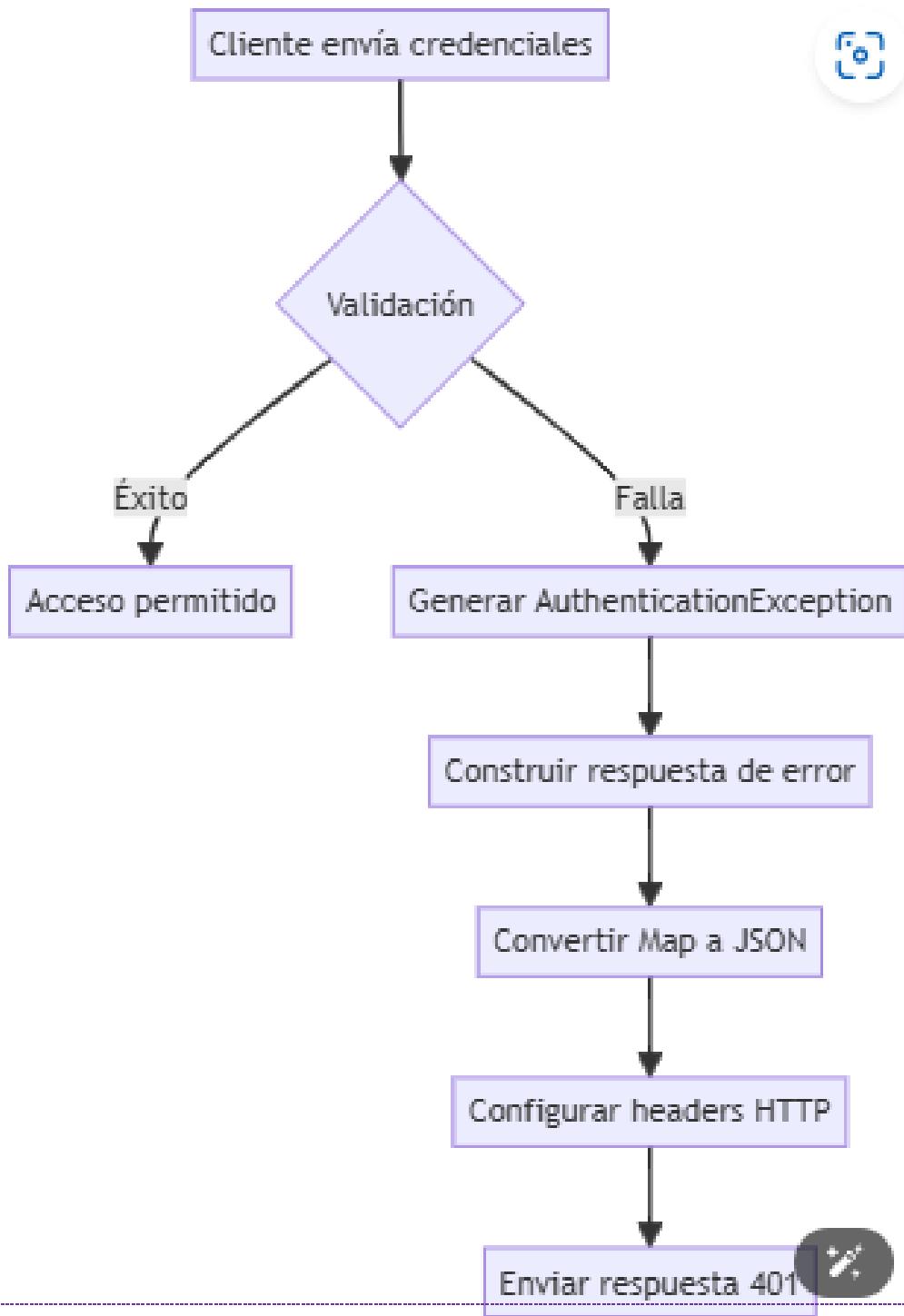
- `writeValueAsString` : Convierte el Map a JSON usando Jackson
- `setStatus(401)` : Establece código HTTP 401 (Unauthorized)
- `setContentType` : Define el tipo MIME como JSON

#### Orden de configuración:

- Es recomendable establecer primero el status y content-type:

```
java
```

```
response.setContentType("application/json");
response.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
response.getWriter().write(...);
```

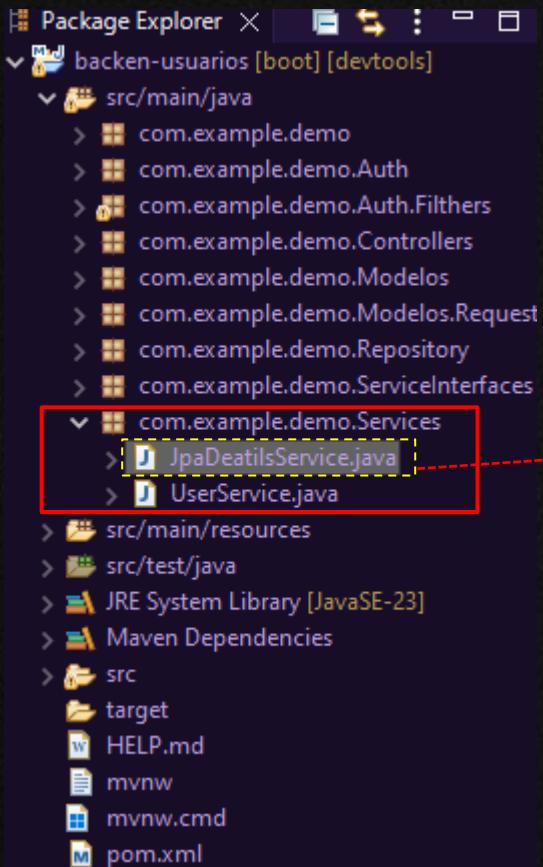


Paso	Acción
1. Inicio	Ejecución del método <code>unsuccessfulAuthentication</code> .
2. Crear Mapa	<code>body = new HashMap&lt;&gt;()</code> .
3. Agregar Mensajes	<code>body.put("message", "...")</code> y <code>body.put("error", failed.getMessage())</code> .
4. Convertir a JSON	<code>new ObjectMapper().writeValueAsString(body)</code> .
5. Escribir Respuesta	<code>response.getWriter().write(JSON)</code> .
6. Establecer Estado	<code>response.setStatus(401)</code> .
7. Establecer Tipo	<code>response.setContentType("application/json")</code> .
8. Fin	Finalización del método.

## 13.7 Crear service que implemente la interface UserDetailsService

es una implementación de un servicio en Spring Security que se encarga de cargar los detalles de un usuario para la autenticación.

### 13.7.1 Crear que la class que va implementar la interface UserDetailsService



Se crea una class en el paquete services

### 13.7.2 Decirle a la class que va ser un service e implemnetarla interface UserDetailsService

```
1 *JpaDeatilsService.java X
2 package com.example.demo.Services;
3 import java.util.ArrayList;
4
5 @Service
6 public class JpaDeatilsService implements UserDetailsService {
7
8
9 }
```

- `@Service` : Esta anotación indica que la clase es un componente de servicio en Spring, lo que significa que será gestionada por el contenedor de Spring y puede ser inyectada en otras partes de la aplicación.
- `implements UserDetailsService` : La clase implementa la interfaz `UserDetailsService`, que es una interfaz clave en Spring Security para cargar los detalles del usuario durante el proceso de autenticación.

### 13.7.3 Implementar el Método loadUserByUsername

```
1 *JpaDeatilsService.java X
2 package com.example.demo.Services;
3 import java.util.ArrayList;
4
5 @Service
6 public class JpaDeatilsService implements UserDetailsService {
7
8     @Override
9     public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25     }
26
27
28
29
30
31
32
33
34
35
36
37
38
39 }
```

Este método es el núcleo de la clase y es obligatorio implementarlo cuando se usa `UserDetailsService`.

Su propósito es cargar los detalles del usuario basándose en el nombre de usuario proporcionado.

**Parámetro** `username` : Es el nombre de usuario que se pasa al método para buscar los detalles del usuario.

**Excepción** `UsernameNotFoundException` : Si el usuario no se encuentra, se lanza esta excepción con un mensaje personalizado.

## 13.7.4 Validar del Usuario

```
*JpaDeatilsService.java X
1 package com.example.demo.Services;
2 import java.util.ArrayList;
3
4
5 @Service
6 public class JpaDeatilsService implements UserDetailsService {
7
8     @Override
9     public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
10
11         if (!username.equals("admin")) {
12
13             throw new UsernameNotFoundException( String.format("Username %s no existe en el sistema", username) );
14
15         }
16
17     }
18
19 }
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39 }
```

### 1. Condición `if`:

- `if (!username.equals("admin"))` verifica si la variable `username` **no es igual** a la cadena `"admin"`.
- El operador `!` es una negación lógica, por lo que el `if` se ejecutará **si `username` NO es igual a `"admin"`**.

### 2. Método `equals`:

- El método `.equals()` se utiliza para comparar el contenido de cadenas (Strings) en Java. En este caso, compara si el contenido de la variable `username` es igual al texto `"admin"`.

### 3. Lanzar una excepción (`throw`):

- Si la condición del `if` es verdadera (es decir, si el `username` no es `"admin"`), se lanza una excepción del tipo `UsernameNotFoundException`.
- La palabra clave `throw` se utiliza para lanzar una excepción en Java.

### 4. Mensaje personalizado en la excepción:

- El mensaje de la excepción se genera usando el método `String.format()`, que permite crear cadenas con formato.
- En este caso, el mensaje será: `"Username <valor_de_username> no existe en el sistema"`, donde `<valor_de_username>` se reemplaza dinámicamente por el valor actual de la variable `username`.

### 5. Excepción `UsernameNotFoundException`:

- Es una excepción personalizada o específica que probablemente esté definida en el sistema para manejar casos en los que un nombre de usuario no se encuentra.

### 13.7.5 Validar del Usuario

```
1 package com.example.demo.Services;
2 import java.util.ArrayList;
3
4
5 @Service
6 public class JpaDeatilsService implements UserDetailsService {
7
8     @Override
9     public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
10
11         if (!username.equals("admin")) {
12
13             throw new UsernameNotFoundException( String.format("Username %s no existe en el sistema", username) );
14
15         }
16
17         List<GrantedAuthority>authorities=new ArrayList<>();
18         authorities.add(new SimpleGrantedAuthority("ROLE_USER"));
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39 }
```

Se crea una lista de autoridades (roles) que se asignarán al usuario.

`GrantedAuthority` es una interfaz de Spring Security que

representa un permiso o rol que un usuario tiene en el sistema.

```
authorities.add(new SimpleGrantedAuthority("ROLE_USER"));
```

- Se agrega un nuevo objeto a la lista `authorities`.
- Este objeto es una instancia de `SimpleGrantedAuthority`, que es una clase que implementa la interfaz `GrantedAuthority`.
- El constructor de `SimpleGrantedAuthority` recibe como argumento una cadena (`String`) que representa un **rol** o **autoridad**. En este caso, el rol es `"ROLE_USER"`.
  - En Spring Security, los roles suelen estar prefijados con `"ROLE_"` para diferenciarlos de otras posibles cadenas de autoridad.
- `"ROLE_USER"` indica que el usuario asociado tiene el rol de "usuario" en el sistema.

#### Contexto:

Este código se utiliza generalmente cuando se está configurando la seguridad de una aplicación. Por ejemplo:

- Al autenticar a un usuario, se le asignan ciertos roles o permisos (como `"ROLE_USER"`, `"ROLE_ADMIN"`, etc.).
- Estos roles se almacenan en una lista de objetos `GrantedAuthority`.
- Más adelante, Spring Security utiliza esta lista para determinar qué recursos o acciones están permitidos para ese usuario.

Aguadelo  
Gutiérrez

### 13.7.6 Creación del Objeto User

```
*JpaDeatilsService.java X
1 package com.example.demo.Services;
2 import java.util.ArrayList;
3
4
5 @Service
6 public class JpaDeatilsService implements UserDetailsService {
7
8     @Override
9     public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
10
11         if (!username.equals("admin")) {
12
13             throw new UsernameNotFoundException( String.format("Username %s no existe en el sistema", username) );
14
15         }
16
17         List<GrantedAuthority>authorities=new ArrayList<>();
18         authorities.add(new SimpleGrantedAuthority("ROLE_USER"));
19
20
21         return new User(username,
22             // "12345" este funciona solo con NoOpPasswordEncoder.getInstance(); para pruebas
23             "$2a$10$DOMDxjYyfZ/e7RcBfUpzqeaCs8pLgcizuiQWXPKu35n0hZlFcE9MS" // password va encriptado solo funciona
24             ,true,
25             true,
26             true,
27             true,
28             authorities);
29
30     }
31 }
```

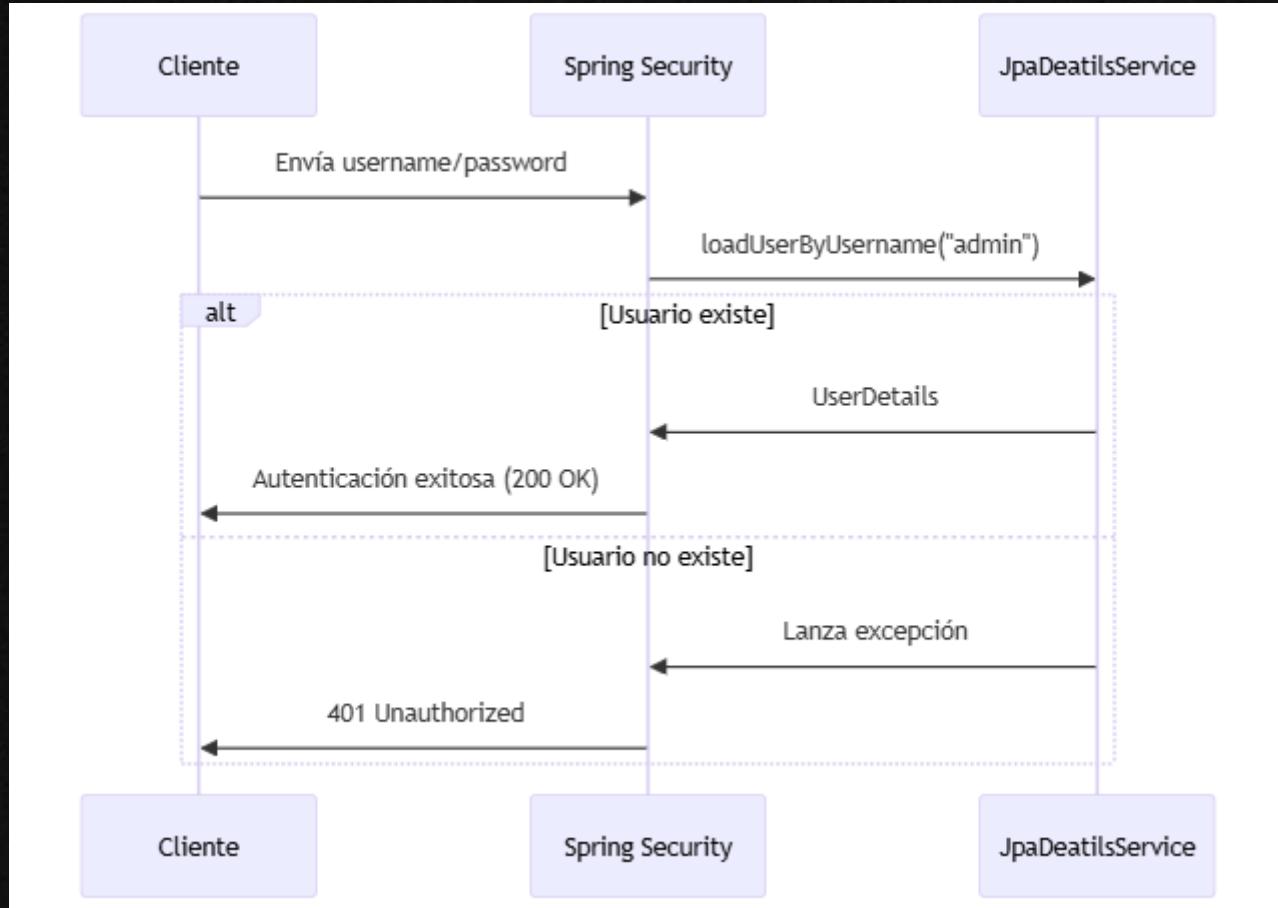
- **Parámetros del constructor** `User` :

- `username` : El nombre de usuario, en este caso "admin".
- `password` : La contraseña del usuario. Aquí se utiliza una contraseña encriptada con BCrypt (`$2a$10$DOMDxjYyfZ/e7RcBfUpzqeaCs8pLgcizuiQWXPKu35n0hZlFcE9MS`). Esta contraseña solo funcionará si se utiliza `BCryptPasswordEncoder` para la autenticación.
- `true, true, true, true` : Estos parámetros indican que la cuenta está activa, no está expirada, las credenciales no están expiradas y la cuenta no está bloqueada, respectivamente.
- `authorities` : La lista de autoridades (roles) que se creó anteriormente.

#### Hardcoding Limitations:

- Usuario y password están fijos en el código (solo para pruebas)
- En implementación real debería:
  - Consultar base de datos
  - Usar JPA/Hibernate para mapeo de entidades

### 13.7.7 Diagrama visual



## 13.8 PasswordEncoder y Password encriptado con Bcrypt. Pruebas con postman

```
*SpringSecurityConfig.java X
1 package com.example.demo.Auth;
2 import org.springframework.beans.factory.annotation.Autowired;□
3 @Configuration
4 public class SpringSecurityConfig {
5
6     @Autowired
7     private AuthenticationConfiguration authenticationConfiguration;
8
9     @Bean
10    PasswordEncoder passwordEncoder() {
11        return NoOpPasswordEncoder.getInstance(); // este solo es para pruebas
12    }
13
14    SecurityFilterChain filterChain(HttpSecurity http) throws Exception {□
15
16        }
17
18
19
20
21
22
23
24
25
26    }
```

## 1. La anotación `@Bean`

Java

 Copiar

`@Bean`

- **¿Qué hace?:** Esta anotación le indica a Spring que el método `passwordEncoder()` retorna un objeto que debe ser administrado por el contenedor de Spring (el `ApplicationContext`).
- En otras palabras, Spring creará una instancia de este objeto (en este caso, `PasswordEncoder`) y la almacenará como un "bean", que puede ser inyectado en otras partes de la aplicación cuando lo necesites.

## 2. El método `passwordEncoder`

Java

 Copiar

```
PasswordEncoder passwordEncoder() {  
    //return NoOpPasswordEncoder.getInstance();  
    return new BCryptPasswordEncoder();  
}
```

Este método crea y configura el tipo de `PasswordEncoder` que tu aplicación usará. Ahora veamos los dos codificadores que aparecen en el código:

### a) NoOpPasswordEncoder (comentado)

Java

Copiar

```
//return NoOpPasswordEncoder.getInstance();
```

- **¿Qué es?:** Este encoder simplemente no hace ningún tipo de cifrado, es decir, almacena las contraseñas en texto plano.
- **¿Por qué evitarlo?:** Usar el `NoOpPasswordEncoder` **no es seguro**, ya que las contraseñas no se protegen. Si un atacante accediera a la base de datos, tendría acceso directo a las contraseñas sin esfuerzo adicional.
- **Uso típico:** Solo se recomienda en **entornos de desarrollo** donde la seguridad no es una preocupación.

### b) BCryptPasswordEncoder (el que se usa)

Java

Copiar

```
return new BCryptPasswordEncoder();
```

- **¿Qué es?:** `BCryptPasswordEncoder` es un encoder robusto que utiliza el algoritmo BCrypt para cifrar contraseñas.
- **Ventajas del uso de BCrypt:**
  - Las contraseñas se cifran de forma **unidireccional**, lo que significa que no se pueden revertir a su forma original.
  - Incluye un **salt** (un dato aleatorio adicional) para evitar ataques de diccionario o de fuerza bruta.
  - Es resistente a ataques de tipo **rainbow table**, ya que utiliza sal aleatoria para cada contraseña.
  - Permite ajustar el "coste computacional", haciéndolo más lento si los recursos de cómputo aumentan, añadiendo protección futura.

```
JpaDeatilsService.java X
19     public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
20
21         if (!username.equals("admin")) {
22
23             throw new UsernameNotFoundException( String.format("Username %s no existe en el sistema", username) );
24
25         }
26
27         List<GrantedAuthority>authorities=new ArrayList<>();
28         authorities.add(new SimpleGrantedAuthority("ROLE_USER"));
29
30         return new User(username,
31             // "12345" este funciona solo con NoOpPasswordEncoder.getInstance(); para pruebas
32
33             // password va encriptado solo funciona con new BCryptPasswordEncoder();
34             "$2a$10$DOMDxjYyfZ/e7RcBfUpzqeaCs8pLgcizuiQWXPKU35n0hZlFcE9MS"
35             ,true,
36             true,
37             true,
38             true,
39             authorities);
40
41     }
42
43 }
```

Gutierrez

POST localhost:8080/login

No environment

localhost:8080/login

Save

Share

POST

localhost:8080/login

Send

La ruta por defecto que maneja  
springSecurity es login

Params

Authorization

Headers (8)

Body ●

Scripts

Settings

Cookies

none

form-data

x-www-form-urlencoded

raw

binary

GraphQL

JSON

Beautify

```
1 {  
2   "username": "admin",  
3   "password": "1234"  
4 }  
5  
6
```

La contraseña correcta es 12345 por  
agora estamos haciendo harcodeado.  
Se ingresa una contraseña incorrecta.  
Nos da el error 401

Body Cookies Headers (11) Test Results

401 Unauthorized

165 ms 451 B

{ } JSON ▾ Preview ⚡ Visualize

≡ ⌂ ⌂ ⌂

```
1 {  
2   "message": "error en la autenticacion username o password incorrecto",  
3   "error": "Bad credentials"  
4 }
```

POST localhost:8080/login +

No environment

localhost:8080/login

Save

Share

POST

localhost:8080/login

Send

La ruta por defecto que maneja  
springSecurity es login

Params Authorization Headers (8) Body Scripts Settings

Cookies

none  form-data  x-www-form-urlencoded  raw  binary  GraphQL

JSON

Beautify

```
1 {  
2   "username": "admin",  
3   "password": "12345"  
4 }  
5 }  
6 }
```

Se ingresa la contraseña correcta.  
Y nosda el status 200k. estamos adentro

Body Cookies Headers (12) Test Results

200 OK • 168 ms • 527 B

{ } JSON ▾ Preview Visualize

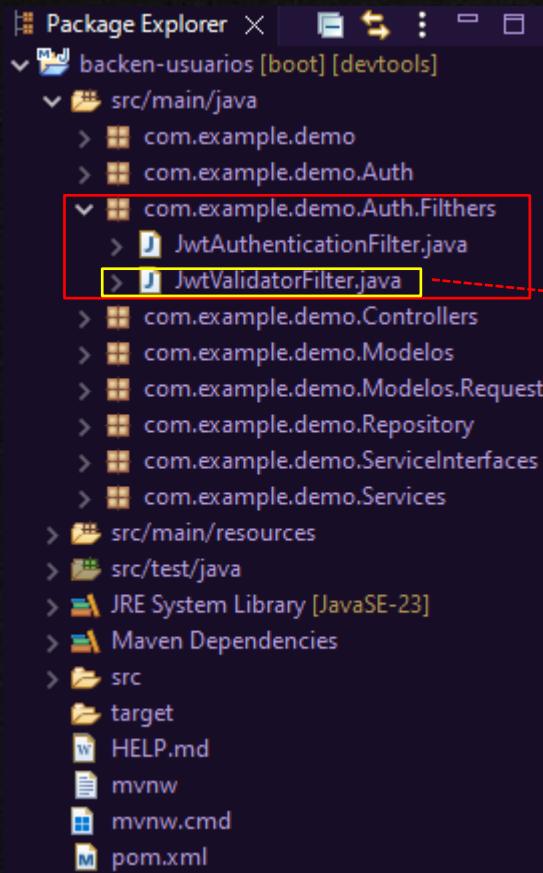
☰ 🔍 🔎

```
1 {  
2   "message": "Hola admin, has iniciado sesión con éxito",  
3   "token": "YWNhX3ZhX2VsX3Rva2VuX0pXVC4gYWRtaW4=",  
4   "username": "admin"  
5 }
```

Este token es de juguete.  
Mas adelante lo teneos que decodificar,  
y validarla con un metodo

## 13.9 Implementando Filtro para validar el token JwtValidationFilter

### 13.9.1 Crear class para crear filtro para la validacion del token



Dentro del paquete  
Auht.Filters se crea una  
class

### 13.9.2 Heredar la class BasicAuthenticationFilter

```
 JwtValidatorFilter.java X
1 package com.example.demo.Auth.Filters;
2
3+import java.io.IOException;
22
23 public class JwtValidatorFilter extends BasicAuthenticationFilter {
24
25+
29
30
31
33+
85
86 }
87
```

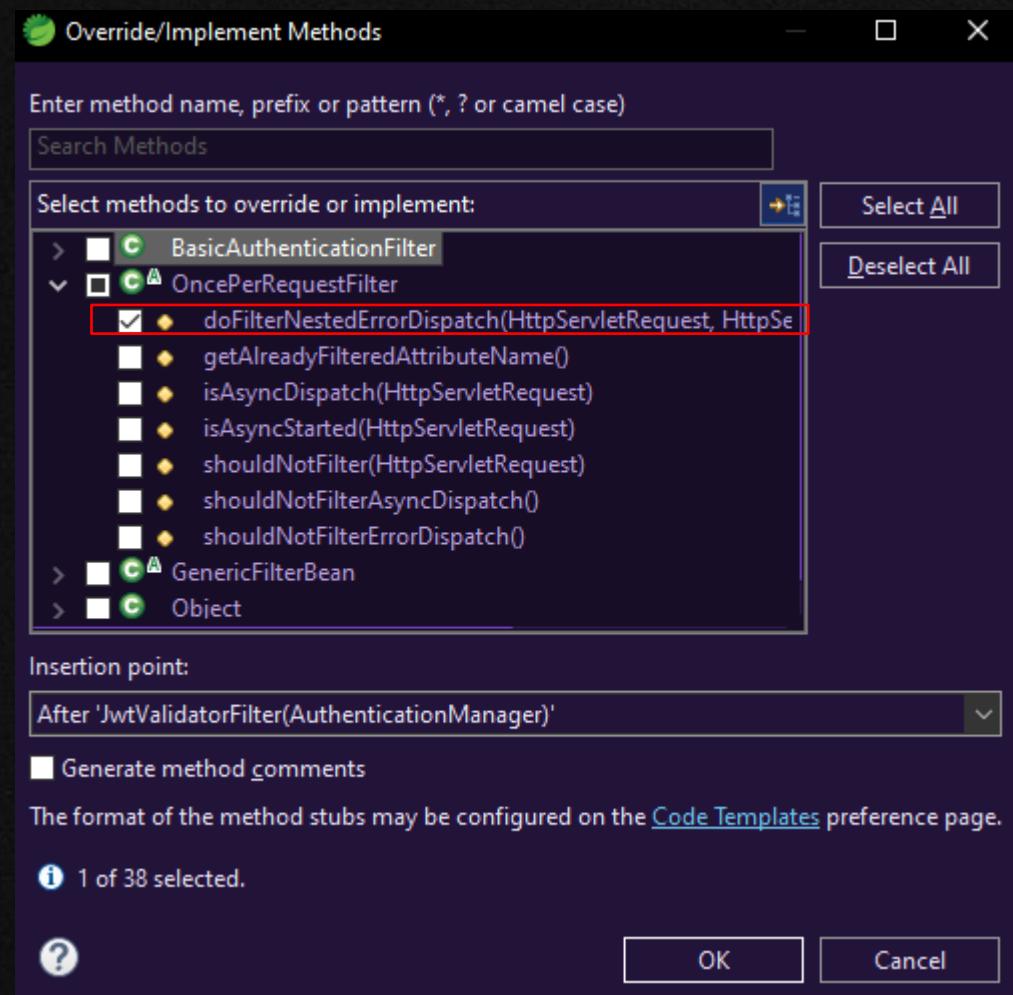
es una clase proporcionada por Spring Security para manejar la autenticación básica

### 13.9.3 Método constructor de la clase heredada BasicAuthenticationFilter

```
JwtValidatorFilter.java ×  
1 package com.example.demo.Auth.Filters;  
2  
3+import java.io.IOException;  
22  
23 public class JwtValidatorFilter extends BasicAuthenticationFilter {  
24  
25+    public JwtValidatorFilter(AuthenticationManager authenticationManager) {  
26        super(authenticationManager);  
27        // TODO Auto-generated constructor stub  
28    }  
29  
30  
31  
32+    protected void doFilterInternal()  
85  
86 }
```

**Constructor:** El constructor recibe un `AuthenticationManager` y lo pasa a la clase padre (`super(authenticationManager)`). Esto es necesario para que el filtro pueda manejar la autenticación.

#### 13.9.4 Implementar filtro con el Método doFilterInternal



JwtValidatorFilter.java X

```
24  
25+     public JwtValidatorFilter(AuthenticationManager authenticationManager) {}  
26  
27  
28  
29  
30  
31+     @Override  
32         protected void doFilterInternal(  
33             HttpServletRequest request,  
34             HttpServletResponse response,  
35             FilterChain chain)  
36             throws IOException, ServletException {  
37  
38  
39  
40  
41  
42  
43  
44  
45  
46  
47  
48  
49  
50  
51  
52  
53
```

Este método es el núcleo del filtro y se ejecuta cada vez que se realiza una solicitud HTTP. Aquí se realiza la validación del token JWT.

### 13.9.5 Obtención del Header

```
JwtValidatorFilter.java X
24
25+     public JwtValidatorFilter(AuthenticationManager authenticationManager) {□
26
27
28
29
30
31+     @Override
32     protected void doFilterInternal(
33         HttpServletRequest request,
34         HttpServletResponse response,
35         FilterChain chain)
36         throws IOException, ServletException {
37
38     // No todos los request viene el Authorization solo en los request privados(paginas privadas)
39     String header= request.getHeader(/*"Authorization"*/TokenJwtConfig.HEADER_AUTHORIZATION);
40
41
42
43
44
45
46
47
48
49
50
51
52
53
```

Se obtiene el valor del encabezado `Authorization` de la solicitud HTTP. Este encabezado debe contener el token JWT.

### 13.9.6 Verificación del Prefijo Bearer:

```
JwtValidatorFilter.java X
24
25+     public JwtValidatorFilter(AuthenticationManager authenticationManager) {□
26
27
28
29
30
31+     @Override
32     protected void doFilterInternal(
33         HttpServletRequest request,
34         HttpServletResponse response,
35         FilterChain chain)
36         throws IOException, ServletException {
37
38     // No todos los request viene el Authorization solo en los request privados(paginas privadas)
39     String header= request.getHeader(/*"Authorization"*/TokenJwtConfig.HEADER_AUTHORIZATION);
40
41     //si este token contiene la palabra Bearer. nos salimos y continuamos con chain
42     if (header==null || !header.startsWith(/*"Bearer "*/TokenJwtConfig.PREFIX_TOKEN)) {
43         chain.doFilter(request, response);
44         return;
45     }
46
47
48
49
50
51
52
53
```

Si el encabezado no existe o no comienza con el prefijo `Bearer`, el filtro simplemente pasa la solicitud al siguiente filtro en la cadena (`chain.doFilter(request, response)`).

```
1. if (header == null ||  
    !header.startsWith(TokentJwtConfig.PREFIX_TOKEN)) :
```

- Esta es una condición `if` que evalúa dos posibles escenarios:
  - a. `header == null`: Comprueba si la variable `header` es `null`, es decir, si no se ha proporcionado ningún valor o el encabezado está vacío.
  - b. `!header.startsWith(TokentJwtConfig.PREFIX_TOKEN)`: Comprueba si el contenido de `header` **no** empieza con un prefijo específico definido en `TokentJwtConfig.PREFIX_TOKEN`. El método `startsWith` verifica si una cadena comienza con un texto determinado, y el operador lógico `!` invierte el resultado (es decir, verifica que **no** comience con ese texto).
- Si **alguna de estas dos condiciones** es verdadera (es decir, si el encabezado es `null` o no tiene el prefijo esperado), entonces se ejecuta el bloque de código dentro de este `if`.

```
2. chain.doFilter(request, response); :
```

- Este método pertenece al flujo de filtros en una aplicación web basada en Java, como las que usan Servlets o Spring Framework.
- Lo que hace es pasar la solicitud (`request`) y la respuesta (`response`) al siguiente filtro en la cadena de filtros configurada. Básicamente, delega el control al siguiente componente sin realizar ninguna acción adicional.

```
3. return;
```

- Esta instrucción termina la ejecución del método actual inmediatamente. En este caso, si se cumple alguna de las condiciones del `if`, el método se detiene justo después de ejecutar `chain.doFilter(request, response)`.

### ¿Qué significa esto en términos funcionales?

Este fragmento de código se utiliza típicamente como parte de un filtro para manejar solicitudes HTTP en una aplicación web que utiliza autenticación basada en tokens JWT (JSON Web Token).

- Se está verificando si el encabezado (`header`) que contiene el token JWT es válido:
  - i. Si el encabezado no está presente (`header == null`) o
  - ii. Si el encabezado no comienza con un prefijo esperado (por ejemplo, "Bearer "),

Entonces:

- La solicitud se pasa al siguiente filtro de la cadena (`chain.doFilter(request, response)`), lo que significa que no se realiza ninguna validación adicional del token JWT.
- Y el método termina con `return`, evitando cualquier procesamiento adicional.

### 13.9.7 Decodificación del Token:

```
JwtValidatorFilter.java X
24
25+     public JwtValidatorFilter(AuthenticationManager authenticationManager) {} ...
26
27
28
29
30
31+     @Override
32     protected void doFilterInternal(
33         HttpServletRequest request,
34         HttpServletResponse response,
35         FilterChain chain)
36         throws IOException, ServletException {
37
38     // No todos los request viene el Authorization solo en los request privados(paginas privadas)
39     String header= request.getHeader(/*"Authorization"*/TokenJwtConfig.HEADER_AUTHORIZATION);
40
41     //si este token contiene la palabra Bearer. nos salimos y continuamos con chain
42     if (header==null || !header.startsWith(/*"Bearer "*/TokenJwtConfig.PREFIX_TOKEN)) {
43         chain.doFilter(request, response);
44         return;
45     }
46
47     // validar el token :
48     //eliminar la cabecera del token
49     String token= header.replace(/*"Bearer "*/TokenJwtConfig.PREFIX_TOKEN, "");
50     byte[] tokenDecodeBytes= Base64.getDecoder().decode(token);
51
52
53
```

El token se decodifica desde Base64 para obtener su contenido en texto plano.

1. `Base64.getDecoder().decode(token)` :

- Este fragmento utiliza la clase `Base64` de Java, específicamente el método `getDecoder()` para obtener un decodificador de Base64.
- Luego, el método `decode(token)` se encarga de decodificar una cadena que está codificada en formato Base64.
- **Base64** es un sistema de codificación que convierte datos binarios (como bytes) en una representación textual utilizando un conjunto limitado de caracteres (letras, números y algunos símbolos). Es comúnmente utilizado para transmitir datos binarios como texto (por ejemplo, en correos electrónicos o JSON).
- El resultado de este método es un arreglo de bytes (`byte[]`) que representa los datos originales antes de ser codificados en Base64.

#### Ejemplo:

Supongamos que `token` es una cadena codificada en Base64 como `"SG9sYSBNDW5kbw=="`. Al decodificarla, obtendremos los bytes originales que representan el texto "Hola Mundo".

2. `new String(tokenDecodeBytes)` :

- Este fragmento toma el arreglo de bytes (`tokenDecodeBytes`) obtenido en el paso anterior y lo convierte en un objeto de tipo `String`.
- La clase `String` tiene un constructor que permite crear una cadena a partir de un arreglo de bytes. Por defecto, utiliza la codificación de caracteres del sistema (generalmente UTF-8) para interpretar los bytes como texto.
- En este caso, el texto decodificado del Base64 se transforma nuevamente en una cadena legible.

#### Ejemplo:

Continuando con el ejemplo anterior, si `tokenDecodeBytes` contiene los bytes del texto "Hola Mundo", este paso convertirá esos bytes en la cadena `"Hola Mundo"`.

#### Resumen:

En conjunto, este código realiza lo siguiente:

1. Decodifica una cadena codificada en Base64 (`token`) y obtiene los datos originales en forma de un arreglo de bytes (`byte[]`).
2. Convierte esos bytes decodificados en una cadena legible (`String`).

#### Uso típico:

Este tipo de código es común cuando se trabaja con datos que han sido codificados en Base64, como tokens de autenticación, imágenes o archivos transmitidos como texto. Por ejemplo:

- Decodificar un token JWT (JSON Web Token) para extraer información.
- Recuperar datos binarios que han sido enviados como texto a través de una API.

### 13.9.8 Validación del Secret:

```
JwtValidatorFilter.java X
46
47     // validar el token :
48     //eliminar la cabecera del token
49     String token= header.replace(/*"Bearer */TokenJwtConfig.PREFIX_TOKEN, "");
50     byte[] tokenDecodeBytes= Base64.getDecoder().decode(token);
51
52
53     String tokenDecode= new String (tokenDecodeBytes);
54
55     String[] tokenArr= tokenDecode.split("\\.");
56     String secret= tokenArr[0];
57     String username=tokenArr[1];
58
59     if (/*"aca_va_el_token_JWT"*/TokenJwtConfig.SECRET_KEY.equals(secret)) {
60
61         List<GrantedAuthority>authorities=new ArrayList<>();
62         authorities.add(new SimpleGrantedAuthority("ROL_USER"));
63
64         UsernamePasswordAuthenticationToken authentication= new UsernamePasswordAuthenticationToken(
65             username,
66             null,
67             authorities
68         );
69
70         SecurityContextHolder.getContext().setAuthentication(authentication);
71
72         chain.doFilter(request, response);

```

Si el `secret` coincide con el valor configurado en `TokenJwtConfig.SECRET_KEY`, se crea un objeto `UsernamePasswordAuthenticationToken` con el `username` y una lista de autoridades (en este caso, solo se asigna el rol `ROL_USER`). Este objeto de autenticación se establece en el contexto de seguridad (`SecurityContextHolder`), lo que permite que Spring Security reconozca al usuario como autenticado.

## 1. Validación de la clave secreta

```
if (TokentJwtConfig.SECRET_KEY.equals(secret)) {
```

- Aquí se compara el valor de `secret` con `TokentJwtConfig.SECRET_KEY`.
- `TokentJwtConfig.SECRET_KEY` probablemente es una constante que almacena la clave secreta usada para validar tokens JWT (JSON Web Tokens).
- Si ambas claves coinciden, significa que el token o la solicitud es válida y se procede a autenticar al usuario.

## 3. Creación del token de autenticación

```
UsernamePasswordAuthenticationToken authentication = new  
UsernamePasswordAuthenticationToken(username, null, authorities);
```

- Aquí se crea un objeto de tipo `UsernamePasswordAuthenticationToken`.
- Este objeto es una implementación de `Authentication` (de Spring Security) y representa al usuario autenticado.
- El constructor recibe tres parámetros:
  - i. `username` : El nombre de usuario del usuario autenticado.
  - ii. `null` : La contraseña no es necesaria en este punto porque ya se validó el token JWT.
  - iii. `authorities` : La lista de roles o permisos asignados al usuario ("ROL\_USER" en este caso).

## 2. Creación de una lista de autoridades

```
List<GrantedAuthority> authorities = new ArrayList<>();  
authorities.add(new SimpleGrantedAuthority("ROL_USER"));
```

- En esta parte, se crea una lista de roles o **autoridades** para el usuario autenticado.
- `GrantedAuthority` es una interfaz en Spring Security que representa un permiso o rol asignado a un usuario.
- Se agrega un rol llamado "ROL\_USER" a la lista. Este rol indica que el usuario tiene permisos básicos o es un usuario estándar.

## 4. Establecer el contexto de seguridad

```
SecurityContextHolder.getContext().setAuthentication(authentication)
```

- Aquí se establece el contexto de seguridad para la sesión actual.
- `SecurityContextHolder` es una clase central en Spring Security que almacena información sobre la autenticación y los detalles de seguridad de la sesión actual.
- Mediante el método `setAuthentication(authentication)`, se indica que el usuario está autenticado y se asocian sus roles/permisos al contexto actual.

## 5. Continuar con la cadena de filtros

```
chain.doFilter(request, response);
```

- Finalmente, después de autenticar al usuario, se llama al siguiente filtro en la cadena utilizando `chain.doFilter(request, response)`.
- Esto permite que la solicitud continúe su curso normal dentro del pipeline de filtros y llegue al controlador o recurso solicitado.

### Resumen general

Este código realiza los siguientes pasos:

- Valida si la clave secreta proporcionada coincide con la clave esperada (`SECRET_KEY`).
- Si es válida, crea una lista de roles (en este caso, `"ROL_USER"`) para el usuario.
- Crea un objeto de autenticación (`UsernamePasswordAuthenticationToken`) con el nombre de usuario y los roles asignados.
- Establece este objeto como el contexto de seguridad actual.
- Continúa procesando la solicitud mediante la cadena de filtros.

Fernando  
Agudelo  
Gutiérrez

### 13.9.9 Respuesta en Caso de Token

```
JwtValidatorFilter.java X
60
61     List<GrantedAuthority>authorities=new ArrayList<>();
62     authorities.add(new SimpleGrantedAuthority("ROL_USER"));
63
64     UsernamePasswordAuthenticationToken authentication= new UsernamePasswordAuthenticationToken(
65         username,
66         null,
67         authorities
68     );
69
70     SecurityContextHolder.getContext().setAuthentication(authentication);
71
72     chain.doFilter(request, response);
73
74 }
75 else {
76
77     Map<String, String>body=new HashMap<>();
78     body.put("message", "el token jwt no es valido");
79     response.getWriter().write(new ObjectMapper().writeValueAsString(body));
80     response.setStatus(403);
81     response.setContentType("application/json");
82
83 }
84 }
85
86 }
```

```
1. else { ... }:
```

- Este bloque de código se ejecutará si no se cumple una condición previa (probablemente un `if` o `else if` anterior). Es decir, este código maneja un caso alternativo o una excepción.

```
2. Map<String, String> body = new HashMap<>(); :
```

- Se crea un objeto de tipo `HashMap` llamado `body`, que es un mapa (estructura de datos que asocia claves con valores). En este caso, tanto las claves como los valores son cadenas de texto (`String`).

```
3. body.put("message", "el token jwt no es valido"); :
```

- Se agrega un par clave-valor al mapa `body`. La clave es `"message"` y el valor asociado es `"el token jwt no es valido"`. Este mensaje probablemente indica que hubo un problema con la validación de un token JWT (JSON Web Token), el cual no es válido.

```
4. response.getWriter().write(new  
ObjectMapper().writeValueAsString(body)); :
```

- Aquí se utiliza `response.getWriter()` para obtener un escritor (`writer`) que permite enviar datos como respuesta al cliente HTTP.
- Se usa `ObjectMapper` (de la biblioteca Jackson) para convertir el objeto `body` (un mapa) en una cadena JSON. Esto se hace con el método `writeValueAsString(body)`, que transforma el mapa en un formato JSON.
- Finalmente, el JSON generado se escribe en la respuesta HTTP mediante el método `.write()`.

Ejemplo del JSON que se enviaría:

```
{  
  "message": "el token jwt no es valido"  
}
```

```
5. response.setStatus(403); :
```

- Se establece el código de estado HTTP de la respuesta en `403`. Este código significa "Prohibido" (Forbidden), lo que indica que el cliente no tiene permiso para acceder al recurso solicitado, probablemente debido a un token JWT inválido.

```
6. response.setContentType("application/json"); :
```

- Se establece el tipo de contenido (`Content-Type`) de la respuesta HTTP como `"application/json"`. Esto le indica al cliente (por ejemplo, un navegador o una aplicación) que los datos enviados están en formato JSON.

### Resumen:

Este bloque de código maneja un caso en el que un token JWT no es válido. Cuando esto ocurre:

- Se construye un objeto JSON con un mensaje de error.
- Se escribe este JSON como respuesta al cliente.
- Se establece un código de estado HTTP `403 Forbidden`.
- Se indica que la respuesta está en formato JSON.

## 13.9.10 Flujo General y Consideraciones

### Flujo General

**Paso 1:** El filtro verifica si la solicitud contiene un token JWT en el encabezado `Authorization`.

**Paso 2:** Si el token existe y es válido, se autentica al usuario y se permite que la solicitud continúe.

**Paso 3:** Si el token no es válido, se devuelve una respuesta de error.

### Consideraciones Adicionales

**Seguridad:** Este código asume que el token JWT está codificado en Base64 y que el `secret` es la primera parte del token. En una implementación real, es recomendable utilizar bibliotecas específicas para manejar JWT, como `jjwt`, que proporcionan métodos más seguros y robustos para la validación de tokens.

**Configuración:** La clase `TokentJwtConfig` parece ser una clase de configuración donde se definen constantes como `HEADER_AUTHORIZATION`, `PREFIX_TOKEN`, y `SECRET_KEY`. Estas constantes se utilizan en el código para mantener la coherencia y facilitar la configuración.

### 13.10 Constantes y Registrando JwtValidationFilter en la configuración Spring Security

```
1 package com.example.demo.Auth;  
2  
3 public class ToketnJwtConfig {  
4  
5     public final static String SECRET_KEY="aca_va_el_token_JWT";  
6     public final static String PREFIX_TOKEN="Bearer ";  
7     public final static String HEADER_AUTHORIZATION="Authorization";  
8  
9 }  
10
```

Gutiérrez

### JwtValidatorFilter.java X

```
31
32     @Override
33     protected void doFilterInternal(
34         HttpServletRequest request,
35         HttpServletResponse response,
36         FilterChain chain)
37         throws IOException, ServletException {
38
39     // No todos los request viene el Authorization solo en los request privados(paginas privadas)
40     String header= request.getHeader(/*"Authorization"*/ToketnJwtConfig.HEADER_AUTHORIZATION);
41
42     //si este token contiene la palabra Bearer. nos salimos y continuamos con chain
43     if (header==null || !header.startsWith(/*"Bearer "*/ToketnJwtConfig.PREFIX_TOKEN)) {
44         chain.doFilter(request, response);
45         return;
46     }
47
48     // validar el token :
49     String token= header.replace(/*"Bearer "*/ToketnJwtConfig.PREFIX_TOKEN, ""); //eliminar la cabecera de
50     byte[] tokenDecodeBytes= Base64.getDecoder().decode(token);
51
52
53     String tokenDecode= new String (tokenDecodeBytes);
54
55     String[] tokenArr= tokenDecode.split(".");
56     String secret= tokenArr[0];
57     String username=tokenArr[1].
```

```
JwtValidatorFilter.java *JwtAuthenticationFilter.java ×
63@Override
64protected void successfulAuthentication(HttpServletRequest request, HttpServletResponse response,
65FilterChain chain, Authentication authResult) throws IOException, ServletException {
66
67    String username= ((org.springframework.security.core.userdetails.User)authResult.getPrincipal()).ge
68    /* UserDetails userDetails = (UserDetails) authResult.getPrincipal();
69    String username = userDetailsService.getUsername();*/
70    String originalInput:/*"aca va el token JWT"*/ToketnJwtConfig.SECRET_KEY + username;
71    String token= Base64.getEncoder().encodeToString(originalInput.getBytes());
72
73    response.addHeader(
74        /*"Authorization"*/ToketnJwtConfig.HEADER_AUTHORIZATION,
75        /*"Bearer "*/ToketnJwtConfig.PREFIX_TOKEN + token
76    );
77
78    Map<String, Object> body= new HashMap<>();
79    body.put("token", token);
80    body.put("message", String.format("Hola %s, has iniciado sesión con éxito", username));
81    body.put("username", username);
82
83    response.getWriter().write(new ObjectMapper().writeValueAsString(body));
84    response.setStatus(200);
85    response.setContentType("application/json");
86}
87
88@Override
```

```
JwtValidatorFilter.java  *JwtAuthenticationFilter.java  SpringSecurityConfig.java X
34
35
36 @Bean
37 SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
38
39     http.authorizeHttpRequests( auth -> auth
40
41     // Permite acceso público al endpoint /users para solicitudes GET
42     .requestMatchers(HttpServletRequestMethod.GET, "/users").permitAll()
43
44     // Requiere autenticación para todas las demás solicitudes
45     .anyRequest().authenticated()
46     )
47     //Añade un filtro personalizado para manejar la autenticación utilizando tokens JWT (JSON Web Tokens).
48     .addFilter(new JwtAuthenticationFilter(authenticationConfiguration.getAuthenticationManager()))
49
50
51     .addFilter(new JwtValidatorFilter(authenticationConfiguration.getAuthenticationManager()))
52
53     // Desactiva la protección CSRF porque no es necesaria en APIs REST
54     .csrf(csrf -> csrf.disable())
55
56     // Configura la sesión como stateless, útil para APIs basadas en tokens
57     .sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS));
58
59     // Construye y devuelve el objeto SecurityFilterChain
60     return http.build();
}
```

Agregar el filtro a la class config

## 13.11 Encryptar password users db

UserService.java X

```
1 package com.example.demo.Services;
2
3+import java.util.List;[]
4
5
6
7
8
9
10
11
12
13
14
15
16
17 @Service
18 public class UserService implements UserInterfaceService {
19
20     @Autowired
21     private UserRepository repository;
22
23     @Autowired
24     private PasswordEncoder passwordEncoder;
25
26
27     public List<User> finAll() {[]
28
29
30
31
32
33
34     public Optional<User> findById(Long id) {[]
35
36
37
38
39     @Transactional
40     @Override
41     public User save(User user) {
42         String passworDbEncryp= passwordEncoder.encode(user.getPassword());
43         user.setPassword(passworDbEncryp);
44         return repository.save(user) ;
45     }
46
47
48
49     public Optional<User> update(UserRequest user, Long id) {[]
50
51
52
53
54
55
56
57
58
59
60
61
```

### 13.11.1 Reiniciar tabla de datos

```
UserService.java  *application.properties X
1 spring.application.name=backen-usuarios
2 spring.jpa.hibernate.ddl-auto=create
3 spring.datasource.url=jdbc:mysql://localhost:3306/db_users_sprintBoot
4 spring.datasource.username=root
5 spring.datasource.password=
6 spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
7 spring.jpa.show-sql=true
8
9 spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect
10
11 logging.level.org.hibernate.SQL=debug
```

The screenshot shows a MySQL database table named 'users'. The table has one row with the following data:

ID	email	password
1	luiser540@gmail.com	\$2a\$10\$HBz1qTS/fvgrkXY5k8S6g.P6Egh5wnM4Ht1M/FNRklg...

Below the table, there is a toolbar with various actions: Check all, With selected:, Edit, Copy, Delete, and Export.

```
JpaDeatilsService.java X UserRepository.java
22    @Autowired
23    private UserRepository repository ; // se inyecta el Repository
24
25    @Override
26    @Transactional(readOnly = true) // como es solo una consulta y no se guarda nada se utiliza (readOnly)
▲27    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
28
29        Optional<com.example.demo.Modelos.User> o = repository.findByUsername(username); //User del model
30
31        if (!o.isPresent()) { // sino esta presente lanzamos el error
32
33            throw new UsernameNotFoundException( String.format("Username %s no existe en el sistema", username)
34
35        }
36
37        com.example.demo.Modelos.User user= o.orElseThrow(); // aca se obtiene el usuario de la BBDD
38
39        List<GrantedAuthority>authorities=new ArrayList<>();
40        authorities.add(new SimpleGrantedAuthority("ROLE_USER"));
41
42        return new User(user.getUsername(), // nuevo
43                      user.getPassword() // aca accedemos a la contraseña de la BBDD y como ya la teneos encriptada
44                      ,true,
45                      true,
46                      true,
47                      true,
48                      authorities);
49
```

```
JpaDeatilsService.java  UserRepository.java X
1 package com.example.demo.Repository;
2
3 import java.util.Optional;
4
5
6 import org.springframework.data.repository.CrudRepository;
7 import org.springframework.stereotype.Repository;
8
9 import com.example.demo.Modelos.User;
10
11 @Repository
12 public interface UserRepository extends CrudRepository<User, Long> {
13
14     Optional<User> finfindByUsername(String username);
15
16
17
18 }
19
20
```

## Libraries for Token Signing/Verification

Filter by Java

- Groovy
- Harbour
- Haskell
- Haxe
- Java
- JavaScript
- kdb+/Q
- Kotlin

 auth0/java-jwt	
<input checked="" type="checkbox"/> Sign	<input checked="" type="checkbox"/> HS256
<input checked="" type="checkbox"/> Verify	<input checked="" type="checkbox"/> HS384
<input checked="" type="checkbox"/> iss check	<input checked="" type="checkbox"/> HS512
<input checked="" type="checkbox"/> sub check	<input checked="" type="checkbox"/> RS256
<input checked="" type="checkbox"/> aud check	<input checked="" type="checkbox"/> RS384
<input checked="" type="checkbox"/> exp check	<input checked="" type="checkbox"/> RS512
<input checked="" type="checkbox"/> nbf check	<input checked="" type="checkbox"/> ES256
<input checked="" type="checkbox"/> iat check	<input type="checkbox"/> ES256K

 b_c/jose4j	
<input checked="" type="checkbox"/> Sign	<input checked="" type="checkbox"/> HS256
<input checked="" type="checkbox"/> Verify	<input checked="" type="checkbox"/> HS384
<input checked="" type="checkbox"/> iss check	<input checked="" type="checkbox"/> HS512
<input checked="" type="checkbox"/> sub check	<input checked="" type="checkbox"/> RS256
<input checked="" type="checkbox"/> aud check	<input checked="" type="checkbox"/> RS384
<input checked="" type="checkbox"/> exp check	<input checked="" type="checkbox"/> RS512
<input checked="" type="checkbox"/> nbf check	<input checked="" type="checkbox"/> ES256
<input checked="" type="checkbox"/> iat check	<input checked="" type="checkbox"/> ES256K

 connect2id/nimbus-jose-jwt	
<input checked="" type="checkbox"/> Sign	<input checked="" type="checkbox"/> HS256
<input checked="" type="checkbox"/> Verify	<input checked="" type="checkbox"/> HS384
<input checked="" type="checkbox"/> iss check	<input checked="" type="checkbox"/> HS512
<input checked="" type="checkbox"/> sub check	<input checked="" type="checkbox"/> RS256
<input checked="" type="checkbox"/> aud check	<input checked="" type="checkbox"/> RS384
<input checked="" type="checkbox"/> exp check	<input checked="" type="checkbox"/> RS512
<input checked="" type="checkbox"/> nbf check	<input checked="" type="checkbox"/> ES256
<input checked="" type="checkbox"/> iat check	<input type="checkbox"/> ES256K



jwtk/jjwt

- |              |           |
|--------------|-----------|
| ✓ Sign       | ✓ HS256   |
| ✓ Verify     | ✓ HS384   |
| ✓ iss check  | ✓ HS512   |
| ✓ sub check  | ✓ RS256   |
| ✓ aud check  | ✓ RS384   |
| ✓ exp check  | ✓ RS512   |
| ✓ nbf check  | ✓ ES256   |
| ✓ iat check  | ?( ES256K |
| ✓ jti check  | ✓ ES384   |
| ?( typ check | ✓ ES512   |
|              | ✓ PS256   |
|              | ✓ PS384   |
|              | ✓ PS512   |
|              | ?( EdDSA  |

👤 Les Hazlewood

⭐ 9940    View repo

maven: io.jsonwebtoken / jjwt-root / 0.11.1

Luis  
ernando  
Agudelo  
utiérrez

## Maven

 Explicar 

```
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-api</artifactId>
    <version>0.12.6</version>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-impl</artifactId>
    <version>0.12.6</version>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-jackson</artifactId> <!-- or jjwt-gson if Gson is preferred -->
    <version>0.12.6</version>
    <scope>runtime</scope>
</dependency>
```

```
*JpaDeatilsService.java  JwtAuthenticationFilter.java  application.properties  backen-usuarios/pom.xml X
86
87          <scope>test</scope>
88      </dependency>
89      <dependency>
90          <groupId>io.jsonwebtoken</groupId>
91          <artifactId>jjwt-api</artifactId>
92          <version>0.12.6</version>
93      </dependency>
94      <dependency>
95          <groupId>io.jsonwebtoken</groupId>
96          <artifactId>jjwt-impl</artifactId>
97          <version>0.12.6</version>
98          <scope>runtime</scope>
99      </dependency>
100     <dependency>
101         <groupId>io.jsonwebtoken</groupId>
102         <artifactId>jjwt-jackson</artifactId> <!-- or jjwt-gson if Gson is preferred -->
103         <version>0.12.6</version>
104         <scope>runtime</scope>
105     </dependency>
106
107     </dependencies>
108
109    <build>
110        <plugins>
```

## Quickstart

Most complexity is hidden behind a convenient and readable builder-based [fluent interface](#), great for relying on IDE auto-completion to write code quickly. Here's an example:

```
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.security.Keys;
import java.security.Key;

// We need a signing key, so we'll create one just for this example. Usually
// the key would be read from your application configuration instead.
SecretKey key = Jwts.SIG.HS256.key().build();

String jws = Jwts.builder().subject("Joe").signWith(key).compact();
```



Gutiérrez

```
1 package com.example.demo.Auth;  
2  
3 import javax.crypto.SecretKey;  
4  
5 import io.jsonwebtoken.Jwts;  
6  
7 public class ToketnJwtConfig {  
8  
9     //public final static String SECRET_KEY="aca_va_el_token_JWT";  
10    public final static SecretKey SECRET_KEY=Jwts.SIG.HS256.key().build();  
11    public final static String PREFIX_TOKEN="Bearer ";  
12    public final static String HEADER_AUTHORIZATION="Authorization";  
13  
14}  
15}  
16
```

Gutiérrez

### 13.12.1 Generar y firmar token JWT modificar JwtAuthenticationFilter

```
JwtAuthenticationFilter.java X
▲ 66     protected void successfulAuthentication(HttpServletRequest request, HttpServletResponse response,
67         FilterChain chain, Authentication authResult) throws IOException, ServletException {
68
69         String username= ((org.springframework.security.core.userdetails.User)
70                         authResult.getPrincipal())
71                         .getUsername(); // se hace un casting
72         /* UserDetails userDetailsService = (UserDetails) authResult.getPrincipal();
73         String username = userDetailsService.getUsername();*/
74         // String originalInput=/*"aca_va_el_token_JWT"*/ToketnJwtConfig.SECRET_KEY + "." + username;
75         // String token= Base64.getEncoder().encodeToString(originalInput.getBytes());
76         String token=Jwts.builder()
77             .setSubject(username)
78             .signWith(ToketnJwtConfig.SECRET_KEY)
79             .setIssuedAt(new Date())
80             .setExpiration(new Date(System.currentTimeMillis() +3600000))
81             .compact();
82
83         response.addHeader(
84             /*"Authorization"*/ToketnJwtConfig.HEADER_AUTHORIZATION,
85             /*"Bearer "*/ToketnJwtConfig.PREFIX_TOKEN + token
86         );
87
88         Map<String, Object> body= new HashMap<>();
89         body.put("token", token);
90         body.put("message", String.format("Hola %s, has iniciado sesión con éxito", username));
91         body.put("username", username);
```

```
JwtValidatorFilter.java ×
```

```
51
52     String token= header.replace(/*"Bearer "*/ToketnJwtConfig.PREFIX_TOKEN, "");
53
54     byte[] tokenDecodeBytes= Base64.getDecoder().decode(token);
55     String tokenDecode= new String (tokenDecodeBytes);
56     //String[] tokenArr= tokenDecode.split("\\.");
57     String secret= tokenDecodeBytes[0];
58     String username=tokenDecodeBytes[1];
59     if /*"aca_va_el_token_JWT"*/ToketnJwtConfig.SECRET_KEY.equals(secret)) {
60
61         try {
62             List<GrantedAuthority>authorities=new ArrayList<>();
63             authorities.add(new SimpleGrantedAuthority("ROL_USER"));
64
65             UsernamePasswordAuthenticationToken authentication= new UsernamePasswordAuthenticationToken(
66                 username,
67                 null,
68                 authorities
69             );
70
71             SecurityContextHolder.getContext().setAuthentication(authentication);
72
73             chain.doFilter(request, response);
74
75         }
76         //else {
```

Se elimina

```
*JwtValidatorFilter.java X
52
53     try {
54
55         List<GrantedAuthority>authorities=new ArrayList<>();
56         authorities.add(new SimpleGrantedAuthority("ROL_USER"));
57
58         UsernamePasswordAuthenticationToken authentication= new UsernamePasswordAuthenticationToken(
59             username,
60             null,
61             authorities
62         );
63
64         SecurityContextHolder.getContext().setAuthentication(authentication);
65
66         chain.doFilter(request, response);
67
68     }
69     catch(JwtException e){
70
71         Map<String, String>body=new HashMap<>();
72         body.put("error", e.getMessage()); // ponemos el error
73         body.put("message", "el token jwt no es valido");
74         response.getWriter().write(new ObjectMapper().writeValueAsString(body));
75         response.setStatus(403);
76         response.setContentType("application/json");
77     }
}
```

## 14.1 Crear class role

The screenshot shows a Java development environment with a dark theme. On the left, the Package Explorer panel displays the project structure. A red box highlights the `Role.java` file under the `com.example.demo.Modelos` package. The main editor window on the right shows the code for the `Role.java` class:

```
1 package com.example.demo.Modelos;
2
3 public class Role {
4
5 }
6
```

#### 14.1.1 Configurar la class role

```
Role.java ×  
1 package com.example.demo.Modelos;  
2  
3+import jakarta.persistence.Entity;□  
8  
9 @Entity  
10 @Table(name="roles")  
11  
12 public class Role {  
13  
16+    private long id;□  
17  
18    private String name;  
19  
20+    public long getId() {□  
23  
24+    public void setId(long id) {□  
27  
28+    public String getName() {□  
31  
32+    public void setName(String name) {□  
35  
36  
37  
38 }|
```

#### 14.1.2 Relacionar la class de los roles con la class de los usuarios

```
Role.java *User.java X
1 package com.example.demo.Modelos;
2
3 import java.util.List;
4
5
6
7 @Entity
8 @Table(name = "users")
9
10 public class User {
11
12     private long id;
13
14     private String username;
15
16     private String password;
17
18     private String email;
19
20     @ManyToMany
21     private List<Role> roles;
22
23     public List<Role> getRoles() {
24
25         return roles;
26     }
27
28     public void setRoles(List<Role> roles) {
29
30         this.roles = roles;
31     }
32
33     public long getId() {
34
35         return id;
36     }
37
38     public void setId(long id) {
39
40         this.id = id;
41     }
42
43     public String getUsername() {
44
45         return username;
46     }
47
48     public void setUsername(String username) {
49
50         this.username = username;
51     }
52
53     public String getPassword() {
54
55         return password;
56     }
57
58     public void setPassword(String password) {
59
60         this.password = password;
61     }
62
63     public String getEmail() {
64
65         return email;
66     }
67
68     public void setEmail(String email) {
69
70         this.email = email;
71     }
72 }
```

### 14.1.3 Configurando el mapeo de roles y esquema de tablas

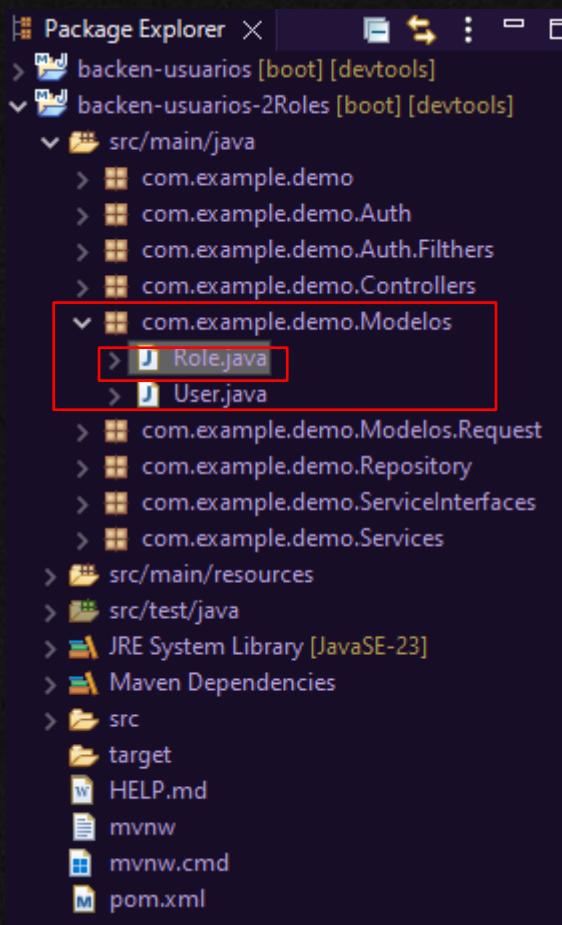
```
User.java X
22
23 public class User {
24
27+private long id;□
28
32+private String username;□
33
35+private String password;□
36
40+private String email;□
41
42+@ManyToMany
43 @JoinTable(
44     name="user_roles",
45     joinColumns = @JoinColumn(name="user_id"),
46     inverseJoinColumns = @JoinColumn(name="role_id"),
47     uniqueConstraints = {@UniqueConstraint(columnNames = {"user_id","role_id"})}
48 )
49
50 private List<Role> roles;
51
52+public List<Role> getRoles() {□
55
56+public void setRoles(List<Role> roles) {□
59
60+public long getId() {□
63
```

## Role.java X

```
1 package com.example.demo.Modelos;
2
3 import jakarta.persistence.Column;...
9
10 @Entity
11 @Table(name="roles")
12
13 public class Role {
14
15     @Id
16     @GeneratedValue(strategy = GenerationType.IDENTITY)
17     private long id;
18
19     @Column(unique = true)
20     private String name;
21
22     public long getId() {
23         return id;
24     }
25
26     public void setId(long id) {
27         this.id = id;
28     }
29
30     public String getName() {
31         return name;
32     }
33
```

#### 14.1.4 Asignando el role al usuario al crear el usuario en POST

```
UserService.java Role.java ✘ RolRepository.java
1 package com.example.demo.Modelos;
2
3 import jakarta.persistence.Column;
4
5
6 @Entity
7 @Table(name="roles")
8
9 public class Role {
10
11     // -----metodos constructores-----
12
13     public Role( ) { }; // se crea un contructor vacio para que jpa cree la instancia
14
15     public Role(long id, String name) { // se crea otro constructor para pasar el id y el rol
16         this.id = id;
17         this.name = name;
18     }
19
20     //-----
21     // -----Atributos-----
22
23     private long id;
24
25     private String name;
26
27     // -----
28     // -----Getters y setters-----
29
30     public long getId() {
31
32     }
33
34 }
```



UserService.java

Role.java

RolRepository.java X

```
1 package com.example.demo.Repository;
2
3 import java.util.Optional;
4
5
6 import org.springframework.data.repository.CrudRepository;
7 import org.springframework.stereotype.Repository;
8
9 import com.example.demo.Modelos.Role;
10
11
12 @Repository
13 public interface RolRepository extends CrudRepository<Role, Long> {
14
15     Optional<Role> findByName(String name);
16
17
18
19
20
21 }
22
```

```
UserService.java X Role.java RolRepository.java
1 package com.example.demo.Services;
2
3+import java.util.ArrayList;[]
18
19
20 @Service
21 public class UserService implements UserInterfaceService {
22
23+    @Autowired
24    private RolRepository rolRepository;
25
26
28+    private UserRepository repository;[]
29
31+    private PasswordEncoder passwordEncoder;[]
32
35+    public List<User> finAll() {[]
39
41+    public Optional<User> findById(Long id) {[]
45
48+    public User save(User user) {[]
65
66
67
70+    public Optional<User> update(UserRequest user, Long id) {[]
82
85+    public void remove(Long id) {[]
```

UserService.java

Role.java

RoleRepository.java

```
39
40+     public Optional<User> findById(Long id) {□
41
42
43     @Transactional
44     @Override
45     public User save(User user) {
46         String passwordEncryp= passwordEncoder.encode(user.getPassword());
47         user.setPassword(passwordEncryp);
48
49         Optional<Role>o= rolRepository.findByName("ROLE_USER");
50
51         List<Role>roles=new ArrayList<>();
52
53         if(o.isPresent()) {
54             roles.add(o.orElseThrow());
55         }
56
57         user.setRoles(roles);
58
59
60         return repository.save(user)  ;
61
62     }
63
64 }
65
66
67
```

```
JpaDeatilsService.java ×
1 public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
2
3     Optional<com.example.demo.Modelos.User> o = repository.findByUsername(username); //User del modelo
4
5     if (!o.isPresent()) { // sino esta presente lanzamos el error
6
7         throw new UsernameNotFoundException( String.format("Username %s no existe en el sistema", username)
8
9     }
10
11    com.example.demo.Modelos.User user= o.orElseThrow(); // aca se obtiene el usuario de la BBDD
12
13    List<GrantedAuthority>authorities=
14        user.getRoles().stream()
15            .map(r-> new SimpleGrantedAuthority(r.getName()))
16            .collect(Collectors.toList());
17
18
19    return new User(
20        user.getUsername() , // nuevo
21        user.getPassword() // aca accedemos a la contraseña de la BBDD y como ya la teneos encriptada
22        ,true,
23        true,
24        true,
25        true,
26        authorities);
27
28 }
```

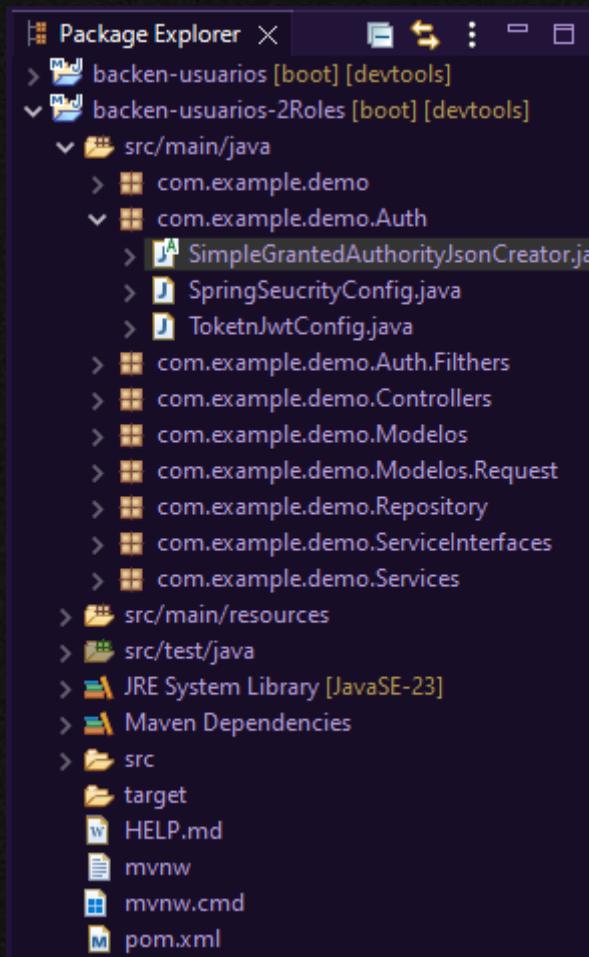
#### 14.1.6 Agregando roles en los claims cuando se genera el token en filtro Authentication

```
JwtAuthenticationFilter.java X
32
33     private AuthenticationManager authenticationManager;
34     public JwtAuthenticationFilter(AuthenticationManager authenticationManager) {
35
36
37     public Authentication attemptAuthentication(HttpServletRequest request, HttpServletResponse response) {
38
39     @Override
40     protected void successfulAuthentication(HttpServletRequest request, HttpServletResponse response,
41         FilterChain chain, Authentication authResult) throws IOException, ServletException {
42
43         String username= ((org.springframework.security.core.userdetails.User)
44             authResult.getPrincipal())
45             .getUsername(); // se hace un casting
46
47         Collection<? extends GrantedAuthority>roles= authResult.getAuthorities();
48
49         boolean isAdmin= roles.stream().anyMatch(r-> r.getAuthority().equals("ROL_ADMIN"));
50
51         Claims claims=Jwts.claims();
52         claims.put("authorities" , new ObjectMapper().writeValueAsString(roles) );
53         claims.put("isAdmin" , isAdmin);
54
55         String token=Jwts.builder()
56             .setClaims(claims)
57             .setSubject(username)
58             .signWith(TokentJwtConfig.SECRET_KEY)
59             .setIssuedAt(new Date())
60
61     }
62
63     return null;
64 }
```

#### 14.1.7 Obteniendo los roles desde los claims en el filtro JwtValidationFilter

```
JwtValidatorFilter.java ×
54     String token= header.replace(/*"Bearer "*/TokenJwtConfig.PREFIX_TOKEN, "");
55
56     try {
57         Claims claims= Jwts.parserBuilder()
58             .setSigningKey(TokenJwtConfig.SECRET_KEY)
59             .build()
60             .parseClaimsJws(token)
61             .getBody();
62
63         Object authoritiesClaims= claims.get("authorities");
64
65         String username= claims.getSubject();
66
67         Collection<? extends GrantedAuthority>authorities= Arrays
68             .asList(new ObjectMapper()
69                 .readValue(authoritiesClaims.toString().getBytes(),SimpleGrantedAuthority[].class ));
70
71
72         UsernamePasswordAuthenticationToken authentication= new UsernamePasswordAuthenticationToken(
73             username,
74             null,
75             authorities
76         );
77
78         SecurityContextHolder.getContext().setAuthentication(authentication);
79
80         chain.doFilter(request, response);
```

#### 14.1.8 Implementando clase MixIn para que funcione el punto anterior



```
1 package com.example.demo.Auth;
2
3 import com.fasterxml.jackson.annotation.JsonCreator;
4
5
6 public abstract class SimpleGrantedAuthorityJsonCreator {
7
8     @JsonCreator
9     public SimpleGrantedAuthorityJsonCreator(@JsonProperty("authprity") String role) {
10
11
12 }
13
14 }
```

# Gutiérrez

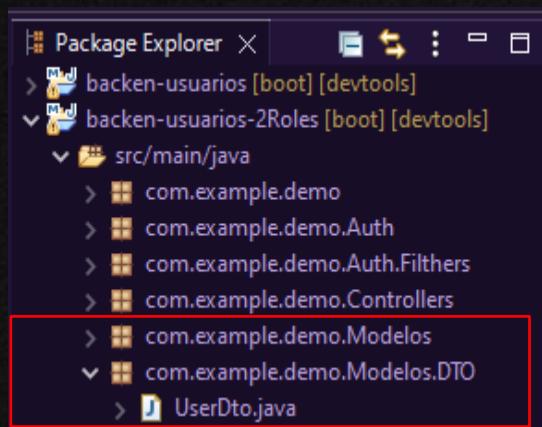
### JwtValidatorFilter.java ×

```
58     Claims claims= Jwts.parserBuilder()
59         .setSigningKey(TokentJwtConfig.SECRET_KEY)
60         .build()
61         .parseClaimsJws(token)
62         .getBody();
63
64     Object authoritiesClaims= claims.get("authorities");
65
66     String username= claims.getSubject();
67
68     Collection<? extends GrantedAuthority>authorities= Arrays
69         .asList(new ObjectMapper()
70             .addMixIn(SimpleGrantedAuthority.class, SimpleGrantedAuthorityJsonCreator.class)
71             .readValue(authoritiesClaims.toString().getBytes(),SimpleGrantedAuthority[].class ));
72
73
74     UsernamePasswordAuthenticationToken authentication= new UsernamePasswordAuthenticationToken(
75         username,
76         null,
77         authorities
78     ).
```

## 14.2 Añadiendo reglas de acceso y autorización para los roles

```
SpringSecurityConfig.java X
35
36 @Bean
37 SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
38
39     http.authorizeHttpRequests( auth -> auth
40
41     // Permite acceso público al endpoint /users para solicitudes GET
42     .requestMatchers(HttpMethod.GET, "/users").permitAll()
43
44     .requestMatchers(HttpMethod.GET,"/users/{id}").hasAnyRole("USERS","ADMIN")
45
46     .requestMatchers(HttpMethod.POST,"/users").hasAnyRole("ADMIN")
47
48     .requestMatchers(HttpMethod.DELETE,"/users/{id}").hasAnyRole("ADMIN")
49
50     .requestMatchers(HttpMethod.PUT,"/users/{id}").hasAnyRole("ADMIN")
51
52     // Requiere autenticación para todas las demás solicitudes
53     .anyRequest().authenticated()
54         )
55     //Añade un filtro personalizado para manejar la autenticación utilizando tokens JWT (JSON Web Tokens).
56     .addFilter(new JwtAuthenticationFilter(authenticationConfiguration.getAuthenticationManager()))
57
58 }
```

## 14.2.1 La clase DTO



```
*UserDto.java
```

```
1 package com.example.demo.Modelos.DTO;
2
3 public class UserDto {
4     private Long id;
5     private String username;
6     private String email;
7
8     public UserDto() {
9
10 }
11
12     public UserDto(Long id, String username, String email) {
13         this.id = id;
14         this.username = username;
15         this.email = email;
16     }
17
18
19     public Long getId() {
20         return id;
21     }
22     public void setId(Long id) {..}
23     public String getUsername() {..}
24     public void setUsername(String username) {..}
25     public String getEmail() {..}
26     public void setEmail(String email) {..}
27
28 }
```

## 14.2.2 La clase DtoMapperUser (DtoMapperBuilder)

The screenshot shows the Eclipse IDE interface with the following details:

- Package Explorer View:** Shows the project structure under "backen-usuarios". A red box highlights the package "com.example.demo.Modelos.DTO.mappe" and its sub-class "DtoMapperUser.java".
- Editor View:** Displays the code for "DtoMapperUser.java". The code implements a builder pattern to map a User entity to a UserDto object.
- Bottom View:** Shows the "Boot Dashboard" with various icons.

```
//va ser una class helper.  
//se va implementar un patron de diseño llamado builte  
public class DtoMapperUser {  
  
    private User user;  
  
    private DtoMapperUser() {}  
  
    public static DtoMapperUser bulider() {}  
  
    public DtoMapperUser setUser(User user) {  
        this.user = user;  
        return this;  
    }  
  
    public UserDto build() {  
  
        if(user== null) {  
  
            throw new RuntimeException("Debe pasar el Entity User!");  
        }  
  
        UserDto userDto= new UserDto( this.user.getId(),user.getUsername(),user.getEmail());  
  
        return userDto;  
    }  
}
```

### 14.2.3 Implementando el DTO en el Service y Controller

```
1 package com.example.demo.ServiceInterfaces;
2
3+import java.util.List;
4
5
6 public interface UserInterfaceService {
7
8     List<UserDto>findAll(); // Listar
9     Optional<UserDto>findById(Long id); // consultar por id
10    UserDto save(User user); // Guardar
11    Optional<UserDto>update(UserRequest user, Long id); // actualizar
12    void remove (Long id); // eliminar
13
14 }
```

Aguadelo  
Gutiérrez

### UserService.java X

```
31     private UserRepository repository;
32
33     @Autowired
34     private PasswordEncoder passwordEncoder;
35
36     @Override
37     @Transactional (readOnly=true)
38     public List<UserDto> findAll() {
39
40         List<User> users= (List<User> ) repository.findAll() ;
41
42         return users
43             .stream()
44             .map(u->DtoMapperUser.bulider()
45                 .setUser(u)
46                 .build())
47             .collect(Collectors.toList());
48     }
49
50
51     public Optional<UserDto> findById(Long id) {[]
52
53
54
55
56
57
58
59
59
60
61
62
63
64
65
66
67
68     public UserDto save(User user) {[]
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87     public Optional<UserDto> update(UserRequest user, Long id) {[]
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102     public void remove(Long id) {[]
103
104
105
106
107 }
```

\*UserService.java ×

```
24 public class UserService implements UserInterfaceService {  
25  
26     @Autowired  
27     private RolRepository rolRepository;  
28  
29  
30     @Autowired  
31     private UserRepository repository;  
32  
33     @Autowired  
34     private PasswordEncoder passwordEncoder;  
35  
36     public List<UserDto> finAll() {  
37  
38     @Override  
39     public Optional<UserDto> findById(Long id) {  
40  
41         return repository.findById(id).map(u-> DtoMapperUser  
42             .bulider()  
43             .setUser(u)  
44             .build());  
45  
46     }  
47  
48     public UserDto save(User user) {  
49  
50     public Optional<UserDto> update(UserRequest user, Long id) {  
51  
52  
53  
54  
55  
56  
57  
58  
59  
60  
61  
62  
63  
64  
65  
66  
67  
68  
69  
70  
71  
72  
73  
74  
75  
76  
77  
78  
79  
80
```

### UserService.java ×

```
38+     public List<UserDto> finAll() {..}
49
51+     public Optional<UserDto> findById(Long id) {..}
65
66+         @Transactional
67+         @Override
68     public UserDto save(User user) {
69         String passworDbEncryp= passwordEncoder.encode(user.getPassword());
70         user.setPassword(passworDbEncryp);
71
72         Optional<Role>o= rolRepository.findByName("ROLE_USER");
73
74         List<Role>roles=new ArrayList<>();
75
76         if(o.isPresent()) {
77             roles.add(o.orElseThrow());
78                     }
79         user.setRoles(roles);
80
81         return DtoMapperUser.bulider().setUser(repository.save(user)).build()    ;
82     }
83
84
87+     public Optional<UserDto> update(UserRequest user, Long id) {..}
99
102+    public void remove(Long id) {..}
106 |
107 }
```

### UserService.java X

```
33     @Autowired
34     private PasswordEncoder passwordEncoder;
35
36
37     public List<UserDto> finAll() {..}
38
39
40     public Optional<UserDto> findById(Long id) {..}
41
42
43     public UserDto save(User user) {..}
44
45
46
47     @Transactional
48     @Override
49     public Optional<UserDto> update(UserRequest user, Long id) {
50         Optional<User> o= repository.findById(id)  ;
51         User userOptional=null;
52         if(o.isPresent()) {
53             User userDb=o.orElseThrow();
54             userDb.setUsername(user.getUsername());
55             userDb.setEmail(user.getEmail());
56             userOptional=(repository.save(userDb));
57         }
58         return Optional.ofNullable(DtoMapperUser.bulider().setUser(userOptional).build());
59
60
61     }
62
63
64
65
66     public void remove(Long id) {..}
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
```

```
UserController.java X
1 package com.example.demo.Controllers;
2
3+import java.util.HashMap;□
29
30 @RestController
31 @RequestMapping("/users") // ruta base
32 @CrossOrigin(originPatterns = "*")
33 public class UserController {
34
35@  @Autowired
36     private UserInterfaceService service;
37
38@  @GetMapping
39     public List<UserDto>list(){ //Listar
40         return service.finAll();
41
42     }
43 }
```

lis  
ando  
delo  
,  
Gutiérrez

### UserController.java

```
56
57 @PutMapping("/{id}")
58 public ResponseEntity<?> actualizar( @Valid @RequestBody UserRequest user, BindingResult result,
59                                     @PathVariable Long id){
60
61     if (result.hasErrors()) {
62         return validation(result);
63     }
64     Optional<UserDto>o=service.update(user, id);
65     if(o.isPresent()) {
66         return ResponseEntity.status(HttpStatus.CREATED).body(o.orElseThrow());
67     }
68
69     return ResponseEntity.notFound().build(); // si no se encuentra muestra el error 404 ;
70
71 }
72
73 private ResponseEntity<?> validation(BindingResult result) {
74
75     Map<String, String> errors= new HashMap<>();
76     result.getFieldErrors().forEach(err->{
77         errors.put(err.getField(), "El campo"+" " + err.getDefaultMessage());
78     });
79
80     return ResponseEntity.badRequest().body(errors);
81 }
82
83 @GetMapping("/{id}")
84     public UserResponse obtenerUser(@PathVariable Long id) {
85         return service.findById(id);
```



UserController.java

```
80         return ResponseEntity.badRequest().body(errors);
81     }
82
83    @GetMapping("/{id}")
84    public ResponseEntity<?>show(@PathVariable Long id){
85        Optional<UserDto> userOptional=service.findById(id);
86        if(userOptional.isPresent()) {
87            return ResponseEntity.ok(userOptional.orElseThrow());
88        }
89
90
91        return ResponseEntity.notFound().build(); // si no se encuentra muestra el error 404 ;
92    }
93
94    }
95
96    @DeleteMapping("/{id}")
97    public ResponseEntity<?>remove(@PathVariable Long id){
98        Optional<UserDto>o=service.findById(id);
99        if(o.isPresent()) {
100            service.remove(id);
101            return ResponseEntity.noContent().build(); // nocontent devuelve 204
102        }
103
104        return ResponseEntity.notFound().build();
105
106    }
107
```

## 14.3 Agregar rol admin al backend

### 14.3.1 Crear atributo admin en los modelos que utilizan los métodos pots

#### 14.3.1.1 Modelo UserRequest

```
1 package com.example.demo.Modelos.Request;
2
3
4+ import jakarta.validation.constraints.Email;[]
5
6
7 public class UserRequest {
8     private String username;[]
9
10
11+    private String email;[]
12
13
14+    private boolean admin;[]
15
16
17     public boolean isAdmin() {
18         return admin;
19     }
20
21
22     public void setAdmin(boolean admin) {
23         this.admin = admin;
24     }
25
26
27     public String getUsername() {[]
28
29
30     public void setUsername(String username) {[
```

Se crea el atributo de tipo boolean con sus métodos get y set.

#### 14.3.2.1.2 Modelo user

```
1 UserRequest.java 2 User.java X 3 UserDto.java 4 *DtoMapperUser.java  
23  
24 public class User {  
25  
28+private long id;□  
29  
33+private String username;□  
34  
36+private String password;□  
37  
41+private String email;□  
42  
51+private List<Role> roles;□  
52  
53+@Transient  
54 private boolean admin;  
55  
56+public boolean isAdmin() {  
57     return admin;  
58 }  
59  
60+public void setAdmin(boolean admin) {  
61     this.admin = admin;  
62 }  
63  
64+public List<Role> getRoles() {□  
65  
66+public void setRoles(List<Role> roles) {□  
67  
72+public long getId() {□
```

uis  
lando  
idelo

Se crea el atributo de tipo boolean con sus métodos guetter y setter.

Se utiliza la anotación @Transient para que no se cree un

### 14.3.1.3 Modelo UserDto

```
UserRequest.java User.java UserDto.java X *DtoMapperUser.java
4     private Long id;
5     private String username;
6     private String email;
7     private boolean admin;
8
9     public boolean isAdmin() {
10
11    public void setAdmin(boolean admin) {
12
13    public UserDto() {
14
15
16
17    public UserDto(Long id, String username, String email,boolean admin) {
18        this.id = id;
19        this.username = username;
20        this.email = email;
21        this.admin=admin;
22    }
23
24
25
26
27
28
29    public Long getId() {
30        return id;
31    }
32    public void setId(Long id) {
33        this.id = id;
34    }
35    public String getUsername() {
36        return username;
```

Se crea el atributo de tipo boolean con sus métodos guetter y setter.  
Y se le pasa al método constructor

#### 14.3.1.4 Modelo DtoMapper class helper

```
1 UserRequest.java  2 User.java   3 UserDto.java  4 *DtoMapperUser.java X
15
16         return new DtoMapperUser();
17
18     }
19
20    public DtoMapperUser setUser(User user) {
21        this.user = user;
22        return this;
23    }
24
25    public UserDto build() {
26
27        if(user== null) {
28
29            throw new RuntimeException("Debe pasar el Entity User!");
30        }
31
32        boolean isAdmin=user.getRoles().stream().anyMatch(r->"ROLE_ADMIN".equals(r.getName()));
33
34        UserDto userDto= new UserDto(
35            this.user.getId(),
36            user.getUsername(),
37            user.getEmail(),
38            isAdmin
39        );
40
41        return userDto;
42    }
43 }
```

Se crea el atributo de tipo boolean con sus métodos guetter y setter.  
Y se le pasa al método constructor

Java

Copiar

```
boolean isAdmin = user.getRoles().stream().anyMatch(r -> "ROLE_ADMIN".equals(r.getName()));
```

1. `getRoles()`: Recupera la lista de roles asociados con el usuario.
2. `stream()`: Convierte la lista en un flujo para permitir operaciones funcionales.
3. `anyMatch()`: Evalúa si algún elemento del flujo cumple con la condición especificada.
  - Condición: `"ROLE_ADMIN".equals(r.getName())` verifica si el nombre del rol es exactamente `"ROLE_ADMIN"`.
4. **Resultado:** `isAdmin` será `true` si el usuario tiene el rol de administrador; de lo contrario, será `false`.

## Creación del DTO

Java

Copiar

```
UserDto userDto = new UserDto(  
    this.user.getId(),  
    user.getUsername(),  
    user.getEmail(),  
    isAdmin  
)
```

1. **Constructor de `UserDto`:** Inicializa un nuevo objeto con información básica del usuario y el estado administrativo (`isAdmin`).
2. **Parámetros utilizados:**
  - `user.getId()`: ID único del usuario.
  - `user.getUsername()`: Nombre de usuario.
  - `user.getEmail()`: Correo electrónico del usuario.
  - `isAdmin`: Indica si el usuario tiene privilegios de administrador.

## 14.3.2 Agregar los rol admin al service del backend

### 14.3.2.1 agregar rol admin al Metodo save del service(userService)

```
UserService.java X
60
61     @Override
62     public UserDto save(User user) {
63         String passworDbEncryp= passwordEncoder.encode(user.getPassword());
64         user.setPassword(passworDbEncryp);
65
66         Optional<Role>o= rolRepository.findByName("ROLE_USER");
67
68         List<Role>roles=new ArrayList<>();
69
70         if(o.isPresent()) {
71             roles.add(o.orElseThrow());
72         }
73         if(user.isAdmin()) {
74             Optional<Role>oa=rolRepository.findByName("ROLE_ADMIN");
75             if(oa.isPresent()) {
76                 roles.add(oa.orElseThrow());
77             }
78         }
79
80         user.setRoles(roles);
81
82         return DtoMapperUser.bulider().setUser(repository.save(user)).build() ;
83     }
84
85     @Transactional
86     @Override
87     public Optional<UserDto> update(UserRequest user, Long id) {
88         Optional<User> o= repository.findById(id) .
```

### 1. Condición:

```
Java
```

 Copiar

```
if(user.isAdmin()) {
```

Este condicional verifica si el usuario tiene privilegios de administrador. Si esta condición es verdadera, se procede al siguiente bloque.

### 2. Obtención del rol:

```
Java
```

 Copiar

```
Optional<Role> oa = rolRepository.findByName("ROLE_ADMIN");
```

Se utiliza un repositorio (`rolRepository`) para buscar el rol llamado `"ROLE_ADMIN"`. El resultado de esta búsqueda se almacena como un `Optional`, lo cual es útil para evitar errores en caso de que el rol no exista en la base de datos.

### 3. Validación de existencia del rol y asignación:

```
Java
```

 Copiar

```
if(oa.isPresent()) {  
    roles.add(oa.orElseThrow());  
}
```

- `oa.isPresent()`: Se verifica si el rol está disponible (no es vacío).
- `oa.orElseThrow()`: En caso de que el rol no esté disponible, lanza una excepción (puedes personalizar esta excepción). Si está disponible, el rol se añade a la lista de roles del usuario.

#### 14.3.2.2 agregar rol admin al Metodo update del service(userService)

```
*UserService.java ×  
85  @Transactional  
86  @Override  
87  public Optional<UserDto> update(UserRequest user, Long id) {  
88      Optional<User> o= repository.findById(id) ;  
89      User userOptional=null;  
90      if(o.isPresent()) {  
91  
92  
93          Optional<Role>ou= rolRepository.findByName("ROLE_USER");  
94  
95          List<Role>roles=new ArrayList<>();  
96  
97          if(o.isPresent()) {  
98              roles.add(ou.orElseThrow());  
99          }  
100         if(user.isAdmin()) {  
101             Optional<Role>oa=rolRepository.findByName("ROLE_ADMIN");  
102             if(oa.isPresent()) {  
103                 roles.add(oa.orElseThrow());  
104             }  
105         }  
106         User userDb=o.orElseThrow();  
107         userDb.setRoles(roles);  
108         userDb.setUsername(user.getUsername());  
109         userDb.setEmail(user.getEmail());  
110         userOptional=(repository.save(userDb));  
111     }  
112     return Optional.ofNullable(DtoMapperUser.bulider().setUser(userOptional).build());  
113 }
```

## Paso 1: Validación inicial

Java

 Copiar

```
if(o.isPresent()) {  
    Optional<Role> ou = rolRepository.findByName("ROLE_USER");  
}
```

- **Qué hace:** Verifica si el objeto `Optional` `o` contiene un valor (esto normalmente representa un usuario obtenido del repositorio de la base de datos).
- **Propósito:** Asegurarse de que el usuario existe antes de continuar con el proceso.
- **Acción:** Si el usuario existe, busca un rol llamado `"ROLE_USER"` en el repositorio de roles.

## Paso 2: Inicialización de la lista de roles

Java

 Copiar

```
List<Role> roles = new ArrayList<>();
```

- **Qué hace:** Crea una lista vacía para almacenar los roles que serán asignados al usuario.
- **Propósito:** Preparar una estructura para añadir roles según las condiciones.

### Paso 3: Asignación del rol de usuario

Java

 Copiar

```
if(ou.isPresent()) {  
    roles.add(ou.orElseThrow());  
}
```

- **Qué hace:** Si el rol `"ROLE_USER"` está disponible, lo agrega a la lista `roles`.
- **Propósito:** Garantizar que todos los usuarios tengan al menos el rol básico de usuario.
- **Excepción:** Si el rol no está presente, el programa lanza una excepción para manejar el error.

### Paso 4: Validación de privilegios de administrador

Java

 Copiar

```
if(user.isAdmin()) {  
    Optional<Role> oa = rolRepository.findByName("ROLE_ADMIN");  
    if(oa.isPresent()) {  
        roles.add(oa.orElseThrow());  
    }  
}
```

- **Qué hace:** Verifica si el usuario tiene privilegios de administrador (`user.isAdmin()`). Si es así, busca el rol `"ROLE_ADMIN"` y lo asigna al usuario.
- **Propósito:** Diferenciar entre usuarios regulares y administradores al asignar permisos.

### Paso 5: Actualización del usuario

Java

 Copiar

```
User userDb = o.orElseThrow();  
userDb.setRoles(roles);
```

- **Qué hace:** Obtiene el objeto `User` desde el `Optional` y asigna la lista de roles actualizada al usuario.
- **Propósito:** Modificar los datos del usuario en memoria antes de guardarlos en la base de datos.

## 14.4 Configurando Cors de Spring Security

### 14.4.1 Definición del método y anotación

```
SpringSecurityConfig.java X
...
72
73
74
75    // Construye y devuelve el objeto SecurityFilterChain
76    return http.build();
77
78}
79
80@Bean
81CorsConfigurationSource corsConfigurationSource() {
82
83
84
85
86
87
88
89
90
91        return source;
92    }
93
94
95
96}
97
```

#### 14.4.2 Creación de la configuración básica de CORS

```
SpringSecurityConfig.java X
72
73
74
75    // Construye y devuelve el objeto SecurityFilterChain
76    return http.build();
77
78 }
79
80 @Bean
81 CorsConfigurationSource corsConfigurationSource() {
82
83     CorsConfiguration config= new CorsConfiguration();
84
85
86
87
88
89
90
91     return config;
92 }
93
94
95
96 }
97
```

Se crea una instancia de CorsConfiguration que será usado para especificar las reglas de CORS.

#### 14.4.3 Configuración de los orígenes permitidos

```
J SpringSecurityConfig.java X
72
73
74
75    // Construye y devuelve el objeto SecurityFilterChain
76    return http.build();
77
78    }
79
80@Bean
81 CorsConfigurationSource corsConfigurationSource() {
82
83    CorsConfiguration config= new CorsConfiguration();
84    config.setAllowedOrigins(Arrays.asList("http://localhost:5173"));
85
86
87
88
89
90
91        return
92    }
93
94
95
96    }
97 }
```

setAllowedOrigins: Define qué orígenes (dominios) están permitidos para hacer solicitudes a este servidor.  
En este caso, solo se permite el dominio http://localhost:5173

#### 14.4.4 Configuración de los métodos HTTP

```
SpringSecurityConfig.java X
...
72
73
74
75    // Construye y devuelve el objeto SecurityFilterChain
76    return http.build();
77
78 }
79
80 @Bean
81 CorsConfigurationSource corsConfigurationSource() {
82
83     CorsConfiguration config= new CorsConfiguration();
84     config.setAllowedOrigins(Arrays.asList("http://localhost:5173"));
85     config.setAllowedMethods(Arrays.asList("GET","POST","PUT","DELETE"));
86
87
88
89
90
91     return config;
92 }
93
94
95
96 }
97 }
```

setAllowedMethods: Especifica qué métodos HTTP están permitidos para las solicitudes desde los orígenes permitidos.  
Aquí se permiten los métodos GET, POST, PUT y DELETE.

#### 14.4.5 Configuración de las cabeceras permitidas

```
SpringSecurityConfig.java X
...
72
73
74
75     // Construye y devuelve el objeto SecurityFilterChain
76     return http.build();
77
78 }
79
80 @Bean
81 CorsConfigurationSource corsConfigurationSource() {
82
83     CorsConfiguration config= new CorsConfiguration();
84     config.setAllowedOrigins((Arrays.asList("http://localhost:5173")));
85     config.setAllowedMethods((Arrays.asList("GET","POST","PUT","DELETE")));
86     config.setAllowedHeaders((Arrays.asList("Authorization","Content-Type")));
87
88
89
90
91     return null;
92 }
93
94
95 }
96
97 }
```

setAllowedHeaders: Define qué cabeceras HTTP pueden ser enviadas en las solicitudes desde los orígenes permitidos.

En este caso, se permiten las cabeceras:

Authorization: Usada comúnmente para enviar tokens de autenticación (por ejemplo, JWT).

Content-Type: Usada para indicar el tipo de contenido enviado, como application/json.

#### 14.5.6 Permitir credenciales (cookies y headers de autenticación)

```
SpringSecurityConfig.java X
71
72
73
74
75     // Construye y devuelve el objeto SecurityFilterChain
76     return http.build();
77
78 }
79
80 @Bean
81 CorsConfigurationSource corsConfigurationSource() {
82
83     CorsConfiguration config= new CorsConfiguration();
84     config.setAllowedOrigins(Arrays.asList("http://localhost:5173"));
85     config.setAllowedMethods(Arrays.asList("GET","POST","PUT","DELETE"));
86     config.setAllowedHeaders(Arrays.asList("Authorization","Content-Type"));
87     config.setAllowCredentials(true);
88
89
90
91     return
92 }
93
94
95
96 }
97 }
```

setAllowCredentials: Permite que las solicitudes incluyan credenciales, como cookies, cabeceras de autenticación o sesiones.

Si se establece en true, el navegador podrá enviar cookies o información de autenticación en las solicitudes CORS.

#### 14.4.7 Aplicar la configuración a todas las rutas

```
SpringSecurityConfig.java X
71
72
73
74
75    // Construye y devuelve el objeto SecurityFilterChain
76    return http.build();
77
78 }
79
80 @Bean
81 CorsConfigurationSource corsConfigurationSource() {
82
83     CorsConfiguration config= new CorsConfiguration();
84     config.setAllowedOrigins(Arrays.asList("http://localhost:5173"));
85     config.setAllowedMethods(Arrays.asList("GET","POST","PUT","DELETE"));
86     config.setAllowedHeaders(Arrays.asList("Authorization","Content-Type"));
87     config.setAllowCredentials(true);
88
89     UrlBasedCorsConfigurationSource source= new UrlBasedCorsConfigurationSource();
90     source.registerCorsConfiguration("/**", config);
91
92     return source;
93 }
94
95
96 }
97 }
```

UrlBasedCorsConfigurationSource: Es un objeto que asocia configuraciones de CORS a rutas específicas.

registerCorsConfiguration("/\*\*", config): Aplica la configuración definida a todas las rutas del servidor (\*\*, que es un patrón global).

#### 14.4.8 Retornar el CorsConfigurationSource

```
SpringSecurityConfig.java X
...
72
73
74
75 // Construye y devuelve el objeto SecurityFilterChain
76 return http.build();
77
78 }
79
80 @Bean
81 CorsConfigurationSource corsConfigurationSource() {
82
83     CorsConfiguration config= new CorsConfiguration();
84     config.setAllowedOrigins( Arrays.asList("http://localhost:5173"));
85     config.setAllowedMethods( Arrays.asList("GET","POST","PUT","DELETE"));
86     config.setAllowedHeaders( Arrays.asList("Authorization","Content-Type"));
87     config.setAllowCredentials(true);
88
89     UrlBasedCorsConfigurationSource source= new UrlBasedCorsConfigurationSource();
90     source.registerCorsConfiguration("/**", config);
91
92     return source;
93 }
94
95
96 }
97 }
```

Finalmente, el método retorna el objeto CorsConfigurationSource

#### 14.4.9 resumen breve

##### Resumen

Este código configura las reglas de CORS en una aplicación Spring Boot con los siguientes parámetros:

1. Permite solicitudes solo desde el origen `http://localhost:5173`.
2. Permite los métodos HTTP `GET`, `POST`, `PUT`, y `DELETE`.
3. Permite las cabeceras `Authorization` y `Content-Type`.
4. Permite el envío de credenciales (como cookies o tokens de autenticación).
5. Aplica estas reglas a todas las rutas del servidor (`/**`).

Esto es útil cuando tienes un front-end (por ejemplo, React, Angular o Vue) corriendo en un dominio diferente al back-end y necesitas permitir la comunicación entre ambos.

#### 14.9.10 Agregar el metodo al Metodo filterchain

```
SpringSecurityConfig.java X
71 .SESSIONMANAGEMENTSESSION ->SESSIONSESSIONCREATIONPOLICYSESSIONCREATIONPOLICY.S
72
73     .cors(cors-> cors.configurationSource(corsConfigurationSource()));
74
75     // Construye y devuelve el objeto SecurityFilterChain
76     return http.build();
77
78 }
79
80 @Bean
81 CorsConfigurationSource corsConfigurationSource() {
82
83     CorsConfiguration config= new CorsConfiguration();
84     config.setAllowedOrigins(Arrays.asList("http://localhost:5173"));
85     config.setAllowedMethods(Arrays.asList("GET","POST","PUT","DELETE"));
86     config.setAllowedHeaders(Arrays.asList("Authorization","Content-Type"));
87     config.setAllowCredentials(true);
88
89     UrlBasedCorsConfigurationSource source= new UrlBasedCorsConfigurationSource();
90     source.registerCorsConfiguration("/**", config);
91
92     return source;
93 }
94
95
96 }
97 }
```

## 1. Habilitar CORS en Spring Security

El método `.cors()` activa el soporte para solicitudes CORS en la cadena de filtros de seguridad.

## 2. Inyectar configuración personalizada

La expresión lambda `cors ->` provee acceso al objeto `CorsConfigurer`, donde se especifica la fuente de configuración mediante `.configurationSource()`.

## 3. Vincular con el Bean existente

`corsConfigurationSource()` referencia al bean previamente definido que contiene las políticas CORS (origenes, métodos, cabeceras, etc.).



#### 14.4.11 Crear filtro CORS en Spring Boot

```
*SpringSecurityConfig.java X
84     CorsConfigurationSource corsConfigurationSource() {
85
86         CorsConfiguration config= new CorsConfiguration();
87         config.setAllowedOrigins(Arrays.asList("http://localhost:5173"));
88         config.setAllowedMethods(Arrays.asList("GET","POST","PUT","DELETE"));
89         config.setAllowedHeaders(Arrays.asList("Authorization","Content-Type"));
90         config.setAllowCredentials(true);
91
92         UrlBasedCorsConfigurationSource source= new UrlBasedCorsConfigurationSource();
93         source.registerCorsConfiguration("/**", config);
94
95         return source;
96     }
97
98     @Bean
99     FilterRegistrationBean<CorsFilter>corsFilter(){
100
101         in;
102
103
104
105     }
106
107
108
109
110     }
111 }
```

```
@Bean
FilterRegistrationBean<CorsFilter> corsFilter() {
```

- `@Bean` : Es una anotación de Spring que indica que el método produce un bean que será gestionado por el contenedor de Spring. En este caso, el bean es de tipo `FilterRegistrationBean<CorsFilter>` .
- `FilterRegistrationBean<CorsFilter>` : Es una clase de Spring que se utiliza para registrar un filtro (en este caso, un filtro CORS) en la aplicación.
- `corsFilter()` : Es el nombre del método que define y configura el bean. Este nombre puede ser cualquier identificador válido.

### \*SpringSecurityConfig.java ×

```
84     CorsConfigurationSource corsConfigurationSource() {  
85  
86         CorsConfiguration config= new CorsConfiguration();  
87         config.setAllowedOrigins(Arrays.asList("http://localhost:5173"));  
88         config.setAllowedMethods(Arrays.asList("GET","POST","PUT","DELETE"));  
89         config.setAllowedHeaders(Arrays.asList("Authorization","Content-Type"));  
90         config.setAllowCredentials(true);  
91  
92         UrlBasedCorsConfigurationSource source= new UrlBasedCorsConfigurationSource();  
93         source.registerCorsConfiguration("/**", config);  
94  
95         return source;  
96     }  
97  
98     @Bean  
99     FilterRegistrationBean<CorsFilter>corsFilter(){  
100  
101         FilterRegistrationBean<CorsFilter>bean=  
102             new FilterRegistrationBean<>(new CorsFilter(corsConfigurationSource()));  
103  
104         return bean;  
105     }  
106  
107 }  
108  
109  
110 }  
111
```

```
// 1. Creación del filtro CORS con configuración personalizada  
FilterRegistrationBean<CorsFilter> bean =  
    new FilterRegistrationBean<>(new  
CorsFilter(corsConfigurationSource()));
```

- Aquí se crea un objeto de tipo `FilterRegistrationBean<CorsFilter>` llamado `bean`.
- El constructor recibe una instancia de `CorsFilter`, que es un filtro proporcionado por Spring para manejar las configuraciones de CORS.
- El `CorsFilter` se inicializa con un método llamado `corsConfigurationSource()`. Este método no está definido en el fragmento de código proporcionado, pero generalmente es otro método dentro de la misma clase que devuelve un objeto `CorsConfigurationSource`. Este objeto contiene las reglas y configuraciones específicas para las solicitudes CORS, como qué orígenes, métodos HTTP y encabezados están permitidos.

### \*SpringSecurityConfig.java

```
84     CorsConfigurationSource corsConfigurationSource() {  
85  
86         CorsConfiguration config= new CorsConfiguration();  
87         config.setAllowedOrigins(Arrays.asList("http://localhost:5173"));  
88         config.setAllowedMethods(Arrays.asList("GET","POST","PUT","DELETE"));  
89         config.setAllowedHeaders(Arrays.asList("Authorization","Content-Type"));  
90         config.setAllowCredentials(true);  
91  
92         UrlBasedCorsConfigurationSource source= new UrlBasedCorsConfigurationSource();  
93         source.registerCorsConfiguration("/**", config);  
94  
95         return source;  
96     }  
97  
98     @Bean  
99     FilterRegistrationBean<CorsFilter>corsFilter(){  
100  
101         FilterRegistrationBean<CorsFilter>bean=  
102             new FilterRegistrationBean<>(new CorsFilter(corsConfigurationSource()));  
103  
104         bean.setOrder(Ordered.HIGHEST_PRECEDENCE);  
105         return  
106     }  
107  
108  
109  
110     }  
111 }
```

```
// 2. Configuración de prioridad de ejecución  
bean.setOrder(Ordered.HIGHEST_PRECEDENCE);
```

- `bean.setOrder(...)` : Configura la prioridad del filtro en el orden de ejecución dentro de la aplicación. Los filtros con mayor prioridad se ejecutan antes que los demás.
- `Ordered.HIGHEST_PRECEDENCE` : Es una constante proporcionada por Spring que indica la prioridad más alta posible. Esto asegura que el filtro CORS se ejecute antes de otros filtros en la aplicación. Esto es importante porque las reglas CORS deben aplicarse antes de procesar cualquier otra lógica en las solicitudes HTTP.

ez

```
SpringSecurityConfig.java X
84     CorsConfigurationSource corsConfigurationSource() {
85
86         CorsConfiguration config= new CorsConfiguration();
87         config.setAllowedOrigins(Arrays.asList("http://localhost:5173"));
88         config.setAllowedMethods(Arrays.asList("GET","POST","PUT","DELETE"));
89         config.setAllowedHeaders(Arrays.asList("Authorization","Content-Type"));
90         config.setAllowCredentials(true);
91
92         UrlBasedCorsConfigurationSource source= new UrlBasedCorsConfigurationSource();
93         source.registerCorsConfiguration("/**", config);
94
95         return source;
96     }
97
98     @Bean
99     FilterRegistrationBean<CorsFilter>corsFilter(){
100
101         FilterRegistrationBean<CorsFilter>bean=
102             new FilterRegistrationBean<>(new CorsFilter(corsConfigurationSource()));
103         bean.setOrder(Ordered.HIGHEST_PRECEDENCE);
104         return bean;
105     }
106
107 }
108
109 }
110 }
111 }
```

- Finalmente, el método retorna el objeto `bean`, que representa el filtro CORS registrado y configurado.

## Resumen

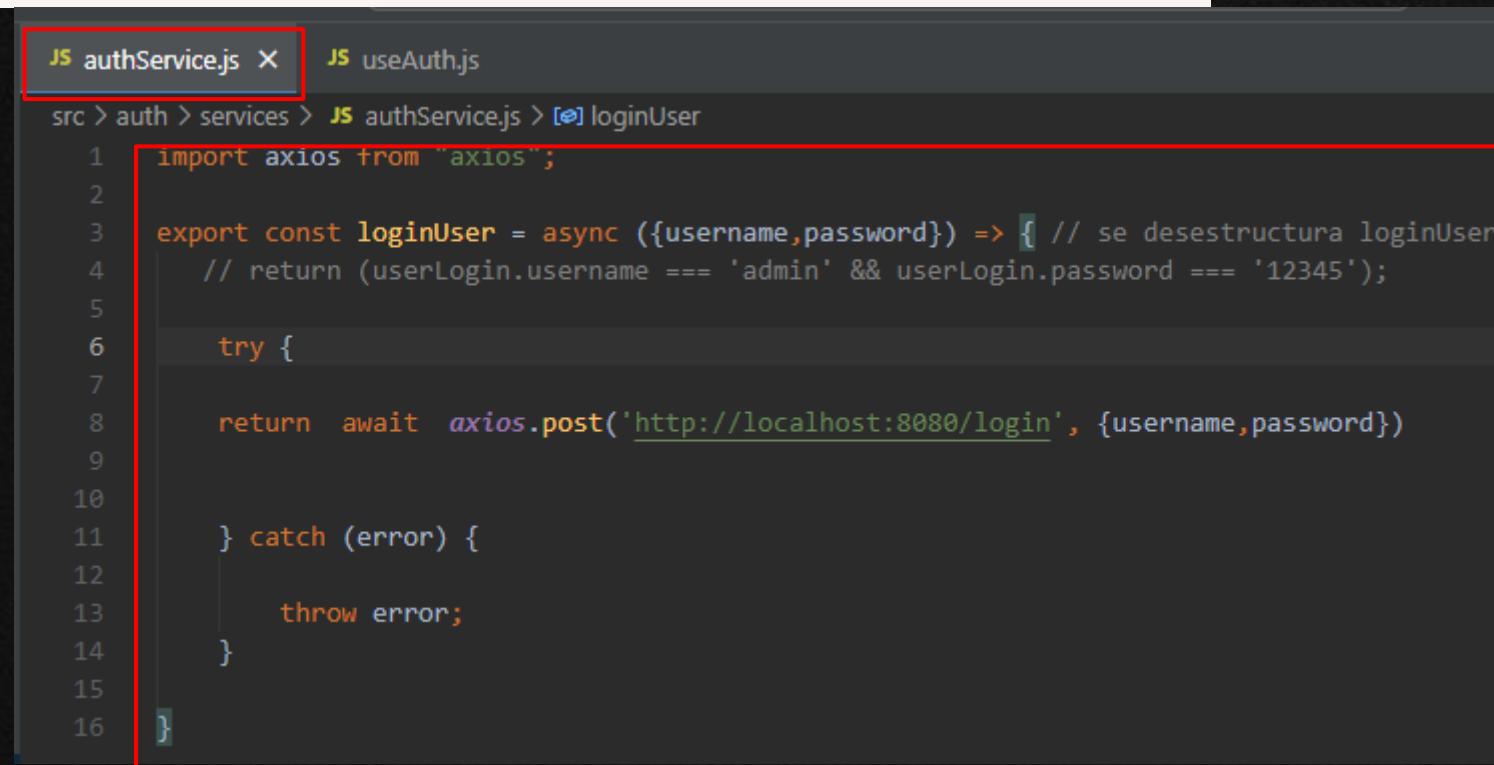
Este código configura un filtro CORS personalizado en una aplicación Spring Boot. El filtro se registra como un bean gestionado por Spring y se le da la máxima prioridad de ejecución. El propósito es permitir o restringir solicitudes HTTP entre diferentes orígenes según las reglas definidas en el método `corsConfigurationSource()` (que no está incluido aquí pero es esencial para definir estas reglas).

Este tipo de configuración es útil cuando tu aplicación necesita interactuar con clientes (como navegadores web o aplicaciones móviles) que están alojados en dominios diferentes al dominio del servidor backend, y necesitas controlar qué solicitudes están permitidas para garantizar la seguridad.

## 14.5 Implementar login con axios

### 14.5.1 Modificar la función loginUser del authService. Para autenticar el usuario enviando credenciales al servidor

La función tiene como objetivo autenticar a un usuario enviando sus credenciales (`username` y `password`) al servidor. Si la autenticación tiene éxito, se espera recibir una respuesta del servidor, posiblemente con un token de autenticación (como JWT) o algún indicador de éxito.



```
JS authService.js X JS useAuth.js
src > auth > services > JS authService.js > [o] loginUser
1 import axios from "axios";
2
3 export const loginUser = async ({username,password}) => { // se desestructura loginUser
4   // return (userLogin.username === 'admin' && userLogin.password === '12345');
5
6   try {
7
8     return await axios.post('http://localhost:8080/login', {username,password})
9
10    } catch (error) {
11
12      throw error;
13    }
14
15  }
16 }
```

#### 1. Importación de Axios:

Javascript

Copiar

```
import axios from "axios";
```

Se importa la biblioteca `axios`, que permite realizar solicitudes HTTP de manera sencilla y eficiente.

## 2. Definición de la función `loginUser`:

Javascript

Copiar

```
export const loginUser = async ({username, password}) => { ... }
```

- Es una función exportable y asíncrona.
- Recibe un objeto con las propiedades `username` y `password`, que se desestructuran para que puedan usarse directamente.

## 3. Uso de `try-catch`:

- Bloque `try`:

Javascript

Copiar

```
return await axios.post('http://localhost:8080/login', {username, pas
```

- Aquí se envía una solicitud HTTP de tipo POST al servidor.
- Los datos `username` y `password` se pasan como el cuerpo de la solicitud (payload).
- La palabra clave `await` asegura que el código espere a que la solicitud sea completada antes de continuar.
- Si la solicitud es exitosa, se devuelve la respuesta.

- Bloque `catch`:

Javascript

Copiar

```
throw error;
```

- Si ocurre un error durante la solicitud (por ejemplo, problemas de red o errores en el servidor), este se captura y se lanza nuevamente (`throw`) para que pueda ser manejado por quien llame a la función.

## 14.5.2 función handlerLogin del hook custom useAuth. Para el inicio de sesión

JS useAuth.js X

```
src > auth > hooks > JS useAuth.js > [o] useAuth > [o] handlerLogin
  ↴
11  export const useAuth = () => {
12
13    const [login, dispatch] = useReducer(loginReducer, initialLogin);
14    const navigate = useNavigate();
15
16    const handlerLogin = async ({ username, password }) => {
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34  }
```

Es una función asíncrona que se encarga de manejar el inicio de sesión del usuario.

## 14.5.2.1 Proceso dentro de handlerLogin. Bloque try catch

```
JS useAuth.js X
src > auth > hooks > JS useAuth.js > [o] useAuth > [o] handlerLogin
11  export const useAuth = () => {
12
13    const [Login, dispatch] = useReducer(loginReducer, initialLogin);
14    const navigate = useNavigate();
15
16    const handlerLogin = async ({ username, password }) => {
17      try {
18
19
20
21
22
23
24
25
26
27
28
29
30      } catch(error){
31
32
33
34    }
  }
```

Se crea bloque try catch

#### 14.5.2.2 Proceso dentro de handlerLogin. Llamada al Backend:

JS useAuth.js X

```
src > auth > hooks > JS useAuth.js > [o] useAuth > [o] handlerLogin
10
11  export const useAuth = () => {
12
13    const [login, dispatch] = useReducer(loginReducer, initialLogin);
14    const navigate = useNavigate();
15
16    const handlerLogin = async ({ username, password }) => {
17      try {
18        const response = await loginUser({ username, password });
19
20
21
22
23
24
25
26
27
28
29
30      }
31      catch(error){
32
33
34    }
```

Se crea una constante sera un nombre en este caso responde

Se le dice que es igual a la dunction loginUser que es la encargada de enviar las credenciales de autenticacion al backend y se le pasa el objeto como parametro y como es ayncronada lleva el await

#### 14.5.2.3 Proceso dentro de handlerLogin.extraer Token y Usuario:

JS useAuth.js X

```
src > auth > hooks > JS useAuth.js > [o] useAuth > [o] handlerLogin
11   export const useAuth = () => {
12
13     const [login, dispatch] = useReducer(loginReducer, initialLogin);
14     const navigate = useNavigate();
15
16     const handlerLogin = async ({username, password}) => {
17       try {
18         const response = await loginUser({username, password});
19         const token = response.data.token;
20         const user = {username: response.data.username}
21
22
23
24
25
26
27
28
29
30       }
31       catch(error){
32
33     }
34 }
```

Se extraen el `token` de autenticación y los datos del usuario desde la respuesta del backend.

#### 14.5.2.4 Proceso dentro de handlerLogin. Decodificar la parte de claims del token usando atob

```
JS useAuth.js ● JS authService.js JS loginReducer.js ●  
src > auth > hooks > JS useAuth.js > [o] useAuth > [o] handlerLogin  
13  export const useAuth = () => {  
14    const handlerLogin = async ({ username, password }) => {  
15      try {  
16        const response = await loginUser({ username, password });  
17        const token= response.data.token;  
18        const claims= JSON.parse(window.atob(token.split(",")[1])) ; // 1  
19        const user = { username: response.data.username }  
20      }  
21    }  
22  }  
23  
24  
25  const claims = JSON.parse(window.atob(token.split(",")[1]));  
26  
27  
28  • El token se divide en varias partes separadas por puntos (.), de acuerdo con la  
29    estructura estándar del JWT (cabecera, cuerpo/payload, firma). Sin embargo, aquí hay  
30    un pequeño error: debería usar " ." en lugar de ", " para dividir el token.  
31  
32  • La parte útil para extraer los datos del usuario generalmente es el payload, que es la  
33    segunda parte del token (índice 1).  
34  
35  • window.atob() convierte la información codificada en Base64 (como en los JWT) a  
36    texto plano.  
37  
38  • JSON.parse() convierte esa cadena de texto JSON en un objeto JavaScript.  
39  
40    }  
41    catch(error){  
42  }
```

#### 14.5.2.5 Proceso dentro de handlerLogin. Actualización del Estado:

JS useAuth.js ● JS authService.js JS loginReducer.js ●

src > auth > hooks > JS useAuth.js > [●] useAuth > [●] handlerLogin

```
13  export const useAuth = () => {  
14  
15    const handlerLogin = async ({ username, password }) => {  
16      try {  
17        const response = await loginUser({ username, password });  
18        const token= response.data.token;  
19        const claims= JSON.parse(window.atob(token.split(",")[1])) ; // 1  
20        const user = { username: response.data.username }  
21  
22        dispatch(  
23          {  
24            type: 'login',  
25            payload: {user, isAdmin:claims.isAdmin} //2  
26          }  
27        );  
28      } catch(error){  
29        console.log(error)  
30      }  
31    }  
32  }  
33  
34  return useAuth;  
35  
36  // 1 - token es el token JWT que se obtiene al loguearse  
37  // 2 - isAdmin es un booleano que indica si el usuario es administrador o no
```

- Dispara acción de login al reducer
- Envía datos del usuario y rol de admin

## 14.5.2.6 Proceso dentro de handlerLogin. Almacenar el estado de inicio de sesión en sessionStorage

JS useAuth.js ● JS authService.js JS loginReducer.js ●

```
src > auth > hooks > JS useAuth.js > [o] useAuth > [o] handlerLogin
13  export const useAuth = () => {
14    const handlerLogin = async ({ username, password }) => {
15      try {
16        const response = await loginUser({ username, password });
17        const token= response.data.token;
18        const claims= JSON.parse(window.atob(token.split(",")[1])) ; // 1
19        const user = { username: response.data.username }
20
21        dispatch({
22          type: 'login',
23          payload: {user,isAdmin:claims.isAdmin} //2
24        );
25
26        sessionStorage.setItem('login', JSON.stringify({
27          isAuth: true,
28          isAdmin: claims.isAdmin, //3
29          user,
30        }));
31
32      }
33    }
34  }
35
36
37
38  }
39
40  catch(error){
41
42 }
```

Los datos de inicio de sesión (estado de autenticación y usuario) se guardan en el almacenamiento de sesión del navegador.

En resumen, este código guarda en el `sessionStorage` un objeto con información sobre el estado de autenticación del usuario, si es administrador y sus datos personales.

## 14.5.2.7 Proceso dentro de handlerLogin. Pasar el rol en false como estado inicial

```
JS useAuth.js X JS authService.js JS loginReducer.js ●
src > auth > hooks > JS useAuth.js > [●] useAuth > [●] handlerLogin
1  ✓ import { useReducer } from "react";
2  import { useNavigate } from "react-router-dom";
3  import Swal from "sweetalert2";
● 4  import { loginReducer } from "../reducers/loginReducer";
5  import { loginUser } from "../services/authService";
6  import { transformWithEsbuild } from "vite";
7
8  ✓ const initialLogin = JSON.parse(sessionStorage.getItem('login')) || {
9    isAuthenticated: false,
10   isAdmin: false, // 4
11   user: undefined,
12 }
13 ✓ export const useAuth = () => {
14
15   const [login, dispatch] = useReducer(loginReducer, initialLogin);
16   const navigate = useNavigate();
17
18  ✓ const handlerLogin = async ({ username, password }) => {
19    try {
20      const response = await loginUser({ username, password });
21      const token = response.data.token;
22      const claims = JSON.parse(window.atob(token.split(".")[1])); // 1
23      const user = { username: response.data.username }
24
25    ✓ dispatch({
26      type: 'login',
27      payload: { user, isAdmin: claims.isAdmin } // 2
28    });
  
```

## 14.5.2.8 Proceso dentro de handlerLogin. Guardar el token en sessionStorage y Redirecciónar al panel de usuarios

```
JS useAuth.js ● JS authService.js      JS loginReducer.js ●  
src > auth > hooks > JS useAuth.js > [o] useAuth > [o] handlerLogin  
13   export const useAuth = () => {  
18     const handlerLogin = async ({ username, password }) => {  
19       try {  
20         const response = await loginUser({ username, password });  
21         const token= response.data.token;  
22         const claims= JSON.parse(window.atob(token.split(",")[1])) ; // 1  
23         const user = { username: response.data.username }  
24  
25         dispatch({  
26           type: 'login',  
27           payload: {user,isAdmin:claims.isAdmin} //2  
28         });  
29  
30         sessionStorage.setItem('login', JSON.stringify({  
31           isAuthenticated: true,  
32           isAdmin: claims.isAdmin, //3  
33           user,  
34         }));  
35  
36         sessionStorage.setItem('token', `Bearer ${token}`); //5  
37  
38       }  
39     }  
40     catch(error){  
41  
42   }
```

### Guardar el token en sessionStorage

Aquí se guarda el token de autenticación en el almacenamiento de la sesión, bajo la clave 'token'.

El token se precede con Bearer, que es un formato común para incluir tokens en encabezados de autorización (como en solicitudes HTTP).

#### 14.5.2.9 Proceso dentro de handlerLogin.

##### Redirección:

```
JS useAuth.js ● JS authService.js      JS loginReducer.js ●
src > auth > hooks > JS useAuth.js > [o] useAuth > [o] handlerLogin
13  export const useAuth = () => {
14
15    const handlerLogin = async ({ username, password }) => {
16      try {
17        const response = await loginUser({ username, password });
18        const token= response.data.token;
19        const claims= JSON.parse(window.atob(token.split(",")[1])) ; // 1
20        const user = { username: response.data.username }
21
22        dispatch({
23          type: 'login',
24          payload: {user,isAdmin:claims.isAdmin} //2
25        });
26
27        sessionStorage.setItem('login', JSON.stringify({
28          isAuth: true,
29          isAdmin: claims.isAdmin, //3
30          user,
31        }));
32
33        sessionStorage.setItem('token', `Bearer ${token}`); //5
34
35        navigate('/users');
36      }
37    }
38
39    catch(error){
40
41
42
```

Una vez autenticado, el usuario es redirigido a la ruta `/users`.

## 14.5.2.10 Proceso dentro de handlerLogin. Manejo de Errores (HTTP 401)

```
JS useAuth.js X JS authService.js JS loginReducer.js ●
src > auth > hooks > JS useAuth.js > [o] useAuth
13   export const useAuth = () => {
18     const handlerLogin = async ({ username, password }) => {
29
30       sessionStorage.setItem('login', JSON.stringify({
31         isAuthenticated: true,
32         isAdmin: claims.isAdmin, //3
33         user,
34       }));
35
36       sessionStorage.setItem('token', `Bearer ${token}`); //5
37
38       navigate('/users');
39     }
40     catch(error){
41
42       if (error.response?.status ==401) { //13
43         Swal.fire('Error Login', 'Username o password invalidos', 'error');
44       }
45
46
47
48       error.response?.status : Comprueba si existe un objeto response dentro del
49       error y evalúa el código de estado HTTP.
50
51       401 : Este estado indica que el usuario no está autenticado (credenciales inválidas).
52
53     }
54 }
```

error.response?.status : Comprueba si existe un objeto `response` dentro del error y evalúa el código de estado HTTP.

401 : Este estado indica que el usuario no está autenticado (credenciales inválidas).

Swal.fire() : Utiliza SweetAlert2 para mostrar un mensaje de error al usuario con los siguientes parámetros:

- 'Error Login' : Título de la alerta.
- 'Username o password invalidos' : Mensaje detallado.
- 'error' : Tipo de alerta, que generalmente muestra un icono de advertencia.

## 14.5.2.11 Proceso dentro de handlerLogin. Manejo de Errores (HTTP 403)

```
JS useAuth.js X JS authService.js JS loginReducer.js ●  
src > auth > hooks > JS useAuth.js > [o] useAuth  
13  export const useAuth = () => {  
18    const handlerLogin = async ({ username, password }) => {  
29  
● 30      sessionStorage.setItem('login', JSON.stringify({  
31        isAuthenticated: true,  
32        isAdmin: claims.isAdmin, //3  
33        user,  
34      }));  
35  
36      sessionStorage.setItem('token', `Bearer ${token}`); //5  
37  
38      navigate('/users');  
39    }  
40    catch(error){  
41  
42      if (error.response?.status ==401) { //13  
43        Swal.fire('Error Login', 'Username o password invalidos', 'error');  
44      }  
45      else if(error.response?.status ==403){ //14  
46        Swal.fire('Error Login', 'No tiene acceso al recurso o permisos', 'error')  
47      }  
48  
49  
50  
51  
52    }  
53  }
```

403 : Este estado indica que el usuario no tiene permisos para acceder al recurso solicitado (prohibido).

De forma similar al caso 401 , se muestra una alerta con SweetAlert2:

- o 'Error Login' : Título de la alerta.
- o 'No tiene acceso al recurso o permisos' : Mensaje detallado para indicar que el acceso está restringido.
- o 'error' : Tipo de alerta.

## 14.5.2.12 Proceso dentro de handlerLogin. Manejo de Errores (otros errores)

```
JS useAuth.js X JS authService.js JS loginReducer.js ●
src > auth > hooks > JS useAuth.js > [o] useAuth
  13  export const useAuth = () => {
  18    const handlerLogin = async ({ username, password }) => {
  29
●  30      sessionStorage.setItem('login', JSON.stringify({
  31        isAuthenticated: true,
  32        isAdmin: claims.isAdmin, //3
  33        user,
  34      }));
  35
  36      sessionStorage.setItem('token', `Bearer ${token}`); //5
  37
  38      navigate('/users');
  39    }
  40    catch(error){
  41
  42      if (error.response?.status ==401) { //13
  43        Swal.fire('Error Login', 'Username o password invalidos', 'error');
  44      }
  45      else if(error.response?.status ==403){ //14
  46        Swal.fire('Error Login', 'No tiene acceso al recurso o permisos', 'error');
  47      }
  48      else{ //15
  49        throw error;
  50      }
  51    }
  52  }
  53 }
```

Si el estado HTTP no es `401` ni `403`, se relanza el error (`throw error`) para que sea manejado por un nivel superior en la aplicación. Esto asegura que no se silencien errores inesperados y permite un diagnóstico más profundo.

### 14.5.3 Modificar el hook custom loginReducer. para manejar el estado relacionado con el inicio y cierre de sesión

```
JS useAuth.js      JS authService.js    JS loginReducer.js •  
src > auth > reducers > JS loginReducer.js > [o] loginReducer  
1  
2 export const loginReducer = (state = {}, action) => {  
3  
4     switch (action.type) {  
5         case 'login':  
6             return {  
7                 isAuthenticated: true,  
8                 isAdmin: action.payload.isAdmin, //6  
9                 user: action.payload.user, //7  
10            };  
11        case 'logout':  
12            return {  
13                isAuthenticated: false,  
14                isAdmin: false, // 8  
15                user: undefined, // 9  
16            };  
17        default:  
18            return state;  
19    }  
20}  
21}
```

`isAdmin: action.payload.isAdmin`: Se actualiza el estado con el rol de administrador, que proviene del `payload` de la acción.

`user: action.payload.user`: Incluye los datos del usuario autenticado.

`isAdmin: false`: Restablece el rol de administrador a `false`.

`user: undefined`: Borra cualquier dato del usuario (estableciéndolo como `undefined`).

## 14.5.4 maneja operaciones CRUD (Crear, Actualizar y Eliminar) en el cliente, utilizando axios para realizar solicitudes HTTP y una configuración de encabezados que incluye un token de autenticación

### 14.4.1 crear función para configurar los encabezados HTTP

JS userServices.js

```
src > services > JS userServices.js > ...
1 import axios from "axios"
2
3 const Base_Url='http://localhost:8080/users';
4
5 const config=()=>{
6   return{
7     headers: {
8       "authorization":sessionStorage.getItem('token'),
9       "Content-Type": "application/json",
10      }
11    }
12  }
13
14 > export const findAll= async()=>{
15
16
17
18
19
20
21
22
23
24 > export const save= async ({username,password,email}) =>{
25   ...
26   };
27
28
29 > export const update= async ({username,email,id}) =>{
30   ...
31   };
32
33
34 > export const remove= async (id) =>{
35   ...
36   };
37 }
```

#### Objeto devuelto:

- La función `config` devuelve un objeto que tiene una propiedad llamada `headers`. Este objeto contiene los encabezados HTTP que probablemente se usarán en una solicitud HTTP (por ejemplo, con `fetch` o `axios`).

#### Propiedades del objeto `headers`:

- `"authorization"`: Se define un encabezado llamado `"authorization"` (en minúsculas) cuyo valor se obtiene del almacenamiento de sesión (`sessionStorage`) mediante la clave `'token'`. Esto significa que la aplicación está intentando recuperar un token de autenticación almacenado previamente en el navegador.
- `"Content-Type"`: Este encabezado indica el tipo de contenido que se enviará en la solicitud HTTP. En este caso, se está especificando que el contenido será de tipo JSON (`application/json`).

## 14.5.2 Pasar la funcion config a los metodos del CRUD (Pots)

```
JS userServices.js X
src > services > JS userServices.js > config > headers > "Authorization"
● 24  ↘ export const save= async ({username,password,email}) =>{
25    ↘   try {
26      return await axios.post(Base_Url,{username,password,email},config()); //2
27    }
28    ↘   catch (error) {
29
30      throw error;
31    }
32
33  };
34
35  ↘ export const update= async ({username,email,id}) =>{
36    ↘   try {
37      return await axios.put(` ${Base_Url}/${id}`,{username,email},config()); //3
38    ↘ } catch (error) {
39
40      throw error;
41    }
42
43  };
44
45  ↘ export const remove= async (id) =>{
46
47    ↘   try {
48      await axios.delete(` ${Base_Url}/${id}`,config()); //4
49    ↘ } catch (error) {
```

## 14.6 Ocultado botones si no es admin

### 14.6.1 Entodos los componentes donde tengamos acciones que sean post los escondemos para los usuarios que no sean admin

```
UsersPage.jsx • Navbar.jsx • UsersList.jsx • UserRow.jsx
src > pages > UsersPage.jsx > [o] UsersPage
  8  export const UsersPage = () => {
 10 >    const { ...
 15     } = useContext(UserContext);
 16
 17     const {login}=useContext(AuthContext); //1. importamos el contexto
 18
 19 >    useEffect(() => { ...
 21     }, []);
 22
 23     return (
 24       <>
 25         {!visibleForm ||
 26           <UserModalForm />}
 27         <div className="container my-4">
 28           <h2>Users App</h2>
 29           <div className="row">
 30             <div className="col">
 31               {(visibleForm || !login.isAdmin) || <button //2. desestructuramos el login y comprobamos si es admin para mostrar el boton
 32                 className="btn btn-primary my-2"
 33                 onClick={handlerOpenForm}>
 34                   Nuevo Usuario
 35                 </button>}
 36
 37               {
 38                 users.length === 0
 39                   ? <div className="alert alert-warning">No hay usuarios en el sistema!</div>
 40                   : <UsersList />
 41               }
 42             </div>
 43           </div>
 44         </div>
 45       </>
 46     )
 47   )
 48 }
```

```
❶ UsersPage.jsx ● ❷ Navbar.jsx ● ❸ UsersList.jsx X ❹ UserRow.jsx
src > components > ❺ UsersList.jsx > [❻] UsersList
    import { UserContext } from "../auth/context/UserContext";
    import { AuthContext } from "../auth/context/AuthContext";
    ...
    export const UsersList = () => {
        ...
        const { users } = useContext(UserContext);
        const { Login } = useContext(AuthContext); // Importamos el contexto
        return (
            ...
            <table className="table table-hover table-striped">
                ...
                <thead>
                    <tr>
                        <th>#</th>
                        <th>username</th>
                        <th>email</th>
                        ...
                        { !Login.isAdmin || <>
                            <th>update</th>
                            <th>update route</th>
                            <th>remove</th>
                        </> }
                ...
            </thead>
            <tbody>
                ...
                <UserRow ...>
                    ...
                </UserRow>
            </tbody>
        )
    }

```

```
UsersPage.jsx ● Navbar.jsx ● UsersList.jsx UserRow.jsx ●
src > components > UserRow.jsx > [e] UserRow
  6  export const UserRow = ({id, username, email}) => {
 13    <td>{email}</td>
 14    {!Login.isAdmin || <>
 15      <td>
 16        <button
 17          type="button"
 18          className="btn btn-secondary btn-sm"
 19          onClick={() => handlerUserSelectedForm({
 20            id,
 21            username,
 22            email
 23            })}
 24          >
 25            update
 26          </button>
 27        </td>
 28        <td>
 29          <NavLink className={'btn btn-secondary btn-sm'}
 30            to={`/users/edit/${id}`}
 31            >
 32              update route
 33            </NavLink>
 34          </td>
 35          <td>
 36            <button
 37              type="button"
 38              className="btn btn-danger btn-sm"
 39              onClick={() => handlerRemoveUser(id)}
 40              >
 41                remove
 42              </button>
 43            </td>
 44          </>
 45        </td>
 46      </td>
 47    </tr>
 48  </tbody>
 49</table>
```

## 14.7 Ocultar página si no es admin

### 14.7.1 Escondemos las páginas que no se requieran si no es admin

UserRoutes.jsx

```
src > routes > UserRoutes.jsx > ...
8  export const UserRoutes = () => {
9    const {login}=useContext(AuthContext); // importamos el contexto de autenticacion
10
11
12    return (
13      <>
14        <UserProvider>
15          <Navbar />
16          <Routes>
17            <Route path="users" element={<UsersPage />} />
18
19            {!login.isAdmin || <>
20              <Route path="users/register" element={<RegisterPage />} />
21              <Route path="users/edit/:id" element={<RegisterPage />} />
22            </>
23          }
24
25            <Route path="/" element={<Navigate to="/users" />} />
26
27          </Routes>
28        </UserProvider>
29      </>
30    )
31 }
```

## 14.8.1 corregir el estatus en el backend

```
JWTvalidatorFilter.java X
70     .addMixin(SimpleGrantedAuthority.class, SimpleGrantedAuthorityJsonCreator.class)
71     .readValue(authoritiesClaims.toString().getBytes(),SimpleGrantedAuthority[].class )
72
73
74     UsernamePasswordAuthenticationToken authentication= new UsernamePasswordAuthenticationToker
75         username,
76         null,
77         authorities
78     );
79
80     SecurityContextHolder.getContext().setAuthentication(authentication);
81
82     chain.doFilter(request, response);
83
84 }
85 catch(JwtException e) {
86
87     Map<String, String>body=new HashMap<>();
88     body.put("error", e.getMessage());
89     body.put("message", "el token jwt no es valido");
90     response.getWriter().write(new ObjectMapper().writeValueAsString(body));
91     response.setStatus(401); //error de autenticacion
92     response.setContentType("application/json");
93
94 }
95
96
97 }
98 }
```

#### 14.8.2 Lanzar el error la función remove del service (userService)

JS useUsers.js ● JS userServices.js X

```
src > services > JS userServices.js > [x] remove
1   import axios from "axios"
2
3   const Base_Url='http://localhost:8080/users';
4
5   > const config=()=>{ //1...
6   }
7
8
9   > export const findAll= async()=>{ ...
10
11
12   }
13
14   > export const save= async ({username,password,email}) =>{
15
16   }
17
18   > export const update= async ({username,email,id}) =>{ ...
19
20   }
21
22
23
24   > export const remove= async (id) =>{
25
26
27   try {
28
29     await axios.delete(` ${Base_Url}/${id}` ,config()); //4
30
31   } catch (error) {
32     throw error;
33   }
34
35
36
37   }
```

#### 14.8.3 Manejar error 401 en el metodo agregar usuario handlerAddUsers (useusers)

```
JS useUsers.js ●
src > hooks > JS useUsers.js > [o] useUsers > [o] handlerRemoveUser
21  export const useUsers = () => {
39    const handlerAddUser = async(user) => {
67      } catch (error) {
68
69      if (error.response && error.response.status==400) {
70        setErrors(error.response.data);
71      }
72
73      else if (error.response && error.response.status==500 &&
74        error.response.data?.message.includes('constraint')){
75
76        if (error.response.data?.message.includes('UK_username')) {
77          setErrors({username: 'El username ya existe'});
78        }
79        if (error.response.data?.message.includes('UK_email')) {
80          setErrors({email: 'El email ya existe'});
81        }
82
83      }
84
85      else if (error.response?.status==401) {
86        handlerLogout();
87        navigate('/login');
88      }
89
90      else{
91        throw error;
92      }
93    }
94  }
```

#### 14.8.4 Manejar error 401 en el metodo remove handlerRemove (useusers)

```
JS useUsers.js ●
src > hooks > JS useUsers.js > [o] useUsers > [o] handlerAddUser
  21  export const useUsers = () => {
  22    const handlerRemoveUser = (id) => {
  23      const confirmDelete = () => {
  24        return new Promise((resolve, reject) => {
  25          Swal.fire({
  26            title: '¿Estás seguro de que deseas eliminar este usuario?',
  27            showCancelButton: true,
  28            cancelButtonColor: '#d33',
  29            confirmButtonText: 'Si, eliminar!'
  30          }).then(result => {
  31            if (result.isConfirmed) {
  32              try {
  33                await remove(id);
  34                dispatch({
  35                  type: 'removeUser',
  36                  payload: id,
  37                });
  38                Swal.fire(
  39                  'Usuario Eliminado!',
  40                  'El usuario ha sido eliminado con éxito!',
  41                  'success'
  42                )
  43              } catch (error) {
  44                if (error.response?.status==401) {
  45                  handlerLogout();
  46                  navigate('/login');
  47                }
  48              }
  49            }
  50          })
  51        );
  52      }
  53    };
  54  };
  55}

  56  
```

## 14.9 Agregando role admin en el Frontend React

### 14.9.1 Agregando rol admin a los metodo save y update del service (UserService)

JS userServices.js X

src > servives > JS userServices.js > ...

```
22 }
23
24 export const save= async ({username,password,email[admin]}) =>{
25   try {
26     return await axios.post(Base_Url,{username,password,email[admin],config()}); //2
27   }
28   catch (error) {
29     throw error;
30   }
31
32
33 };
34
35 export const update= async ({username,email,id[admin]}) =>{
36   try {
37     return await axios.put(` ${Base_Url}/${id}`,{username,email[admin],config()}); //3
38   } catch (error) {
39     throw error;
40   }
41
42
43 };
44
45 } export const remove= async (id) =>{ ...
46 }
```

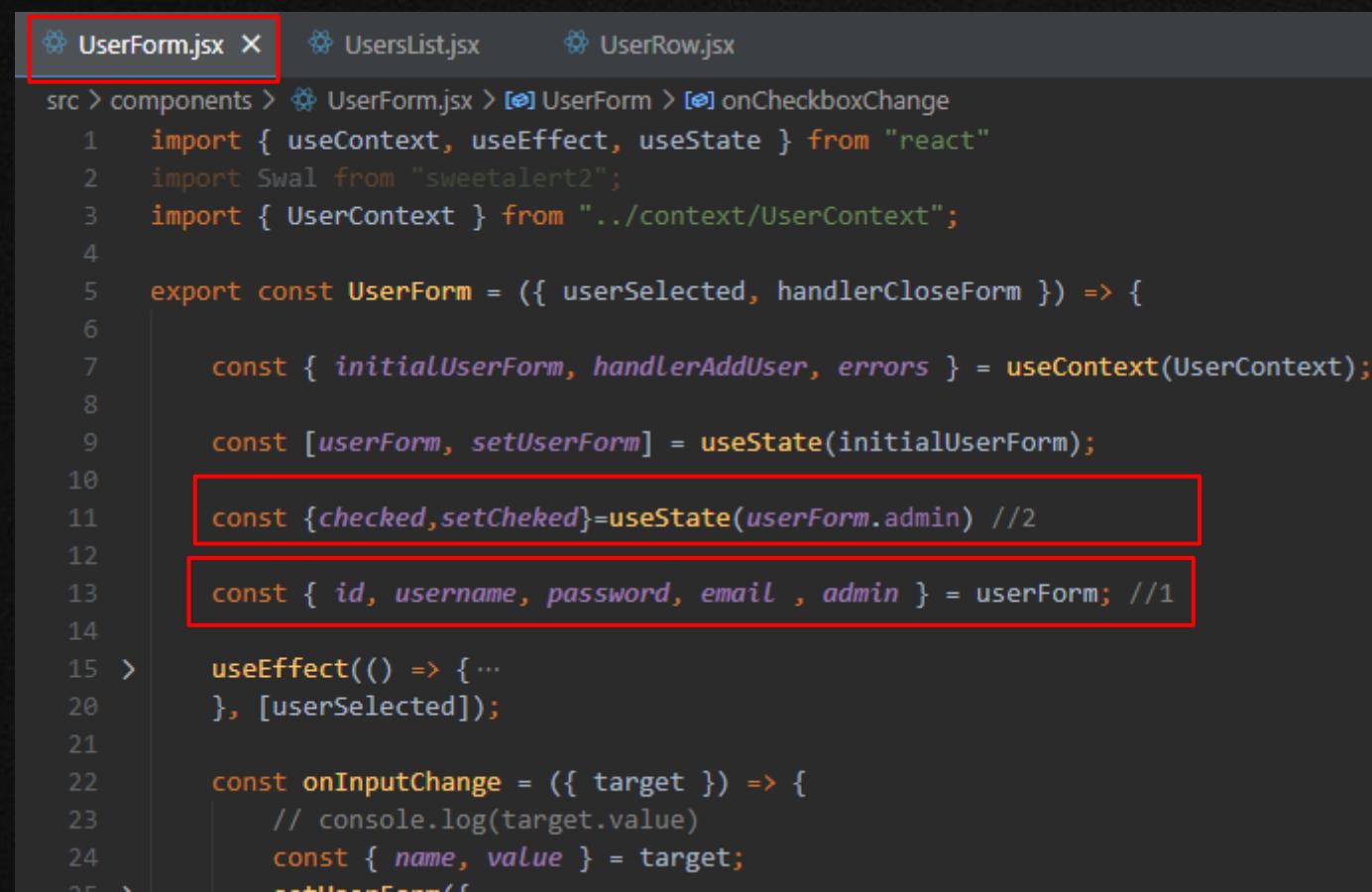
## 14.9.2 Agregando rol admin a al estado inicial del formulario en el hook Custom (useUsers)

JS useUsers.js X

```
src > hooks > JS useUsers.js > [e] initialUserForm
  1 import { useContext, useReducer, useState } from "react";
  2 import { useNavigate } from "react-router-dom";
  3 import Swal from "sweetalert2";
  4 import { usersReducer } from "../reducers/usersReducer";
  5 import { findAll, remove, save, update } from "../services/userServices";
  6 import { AuthContext } from "../auth/context/AuthContext";
  7 const initialUsers = [];
  8
  9 const initialUserForm = {
 10   id: 0,
 11   username: '',
 12   password: '',
 13   email: '',
 14   admin: false,
 15 }
 16
 17 > const initialErrors = { ...
 18   }
 19
 20 export const useUsers = () => {
 21   const [users, dispatch] = useReducer(usersReducer, initialUsers);
 22   const [userSelected, setUserSelected] = useState(initialUserForm);
 23   const [visibleForm, setVisibleForm] = useState(false);
 24   const [errors, setErrors] = useState(initialErrors);
 25
 26   const navigate = useNavigate();
 27
 28   const {handlerLogout} = useContext(AuthContext);
 29
 30   const getUsers= async ()=>{ // funcion para cargar los usuarios...
 31     }
 32   >
 33   const handlerAddUser = async(user) => { ...
 34 }
```

Se pasa al estado inicial  
y se va iniciar en false

### 14.9.3 Agregar el nuevo campo admin en el formulario (UserForm)



```
src > components > UserForm.jsx > UserForm > onCheckboxChange
1 import { useContext, useEffect, useState } from "react"
2 import Swal from "sweetalert2";
3 import { UserContext } from "../context/UserContext";
4
5 export const UserForm = ({ userSelected, handlerCloseForm }) => {
6
7     const { initialUserForm, handlerAddUser, errors } = useContext(UserContext);
8
9     const [userForm, setUserForm] = useState(initialUserForm);
10
11    const { checked, setChecked } = useState(userForm.admin) //2
12
13    const { id, username, password, email, admin } = userForm; //1
14
15    useEffect(() => {
16        [userSelected]);
17
18        const onInputChange = ({ target }) => {
19            // console.log(target.value)
20            const { name, value } = target;
21            setUserForm({
22                ...userForm,
23                [name]: value
24            });
25        };
26
27        target.addEventListener("change", onInputChange);
28
29        return () => {
30            target.removeEventListener("change", onInputChange);
31        };
32    }, [userSelected]);
33
34    const onFormSubmit = (e) => {
35        e.preventDefault();
36
37        if (!errors.length) {
38            handlerAddUser(userForm);
39            handlerCloseForm();
40        }
41    };
42
43    return (
44        <div>
45            <h3>Nuevo Usuario</h3>
46            <form onSubmit={onFormSubmit}>
47                <div>
48                    <label>Nombre de usuario</label>
49                    <input type="text" name="username" value={userForm.username} />
50                </div>
51                <div>
52                    <label>Contraseña</label>
53                    <input type="password" name="password" value={userForm.password} />
54                </div>
55                <div>
56                    <label>Email</label>
57                    <input type="email" name="email" value={userForm.email} />
58                </div>
59                <div>
60                    <label>Admin</label>
61                    <input checked={checked} type="checkbox" name="admin" value="true" />
62                </div>
63                <div>
64                    <button type="submit">Crear</button>
65                </div>
66            </form>
67        </div>
68    );
69}
```

## Línea 1: Desestructuración del objeto `userForm`

Javascript

 Copiar

```
const { id, username, password, email, admin } = userForm; //1
```

- **Qué hace:** Esta línea desestructura el objeto `userForm`, extrayendo las propiedades `id`, `username`, `password`, `email` y `admin` en variables individuales.
- **Propósito:** Facilita el acceso directo a estos valores sin tener que escribir constantemente `userForm.propiedad`.

## Línea 2: Uso del hook `useState`

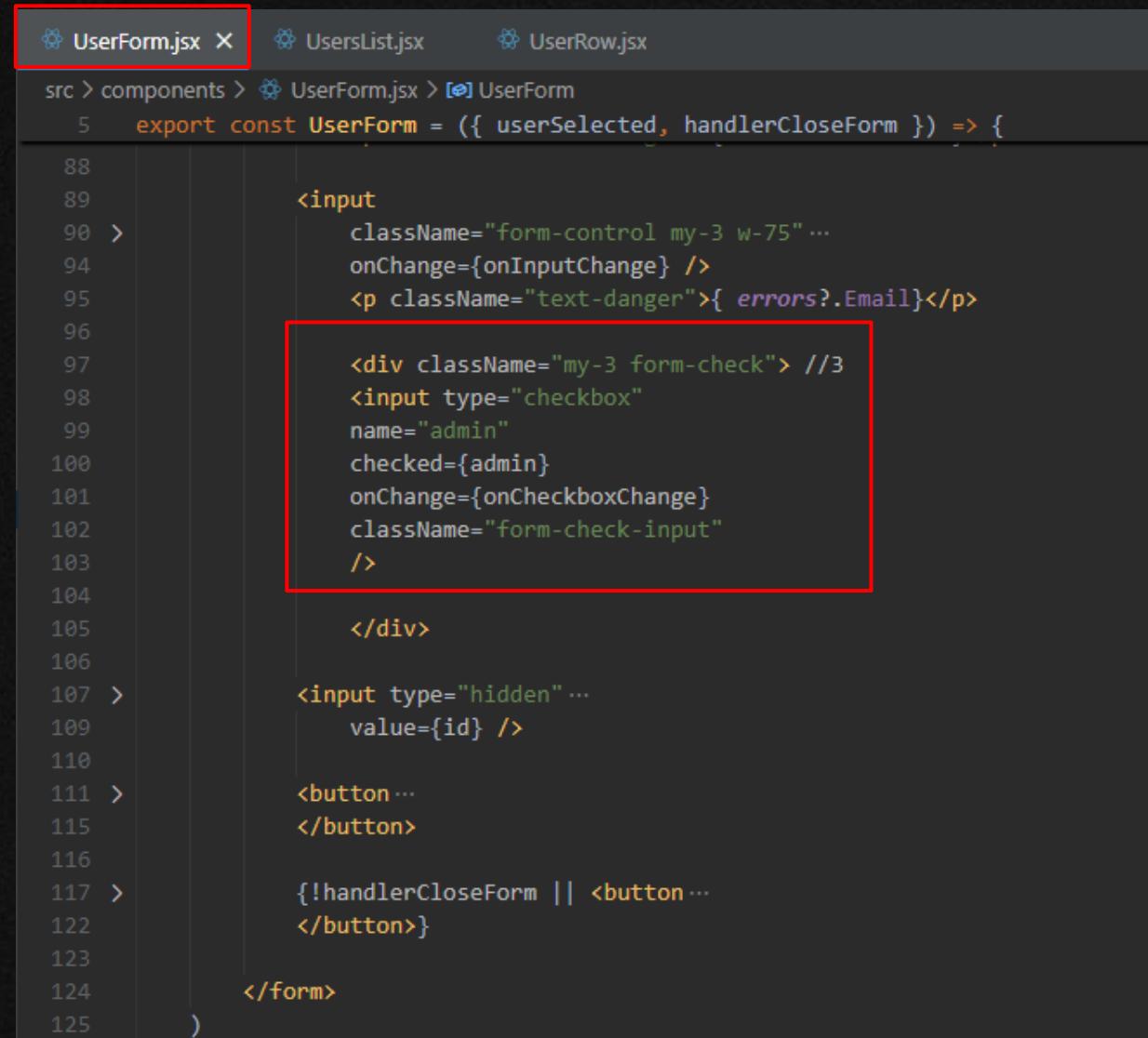
Javascript

 Copiar

```
const { checked, setChecked } = useState(userForm.admin); //2
```

- **Qué hace:** Inicializa un estado para manejar la propiedad `checked`.
  - `checked` será el valor actual del estado (basado en `userForm.admin`).
  - `setChecked` es la función para actualizar ese estado.
- **Propósito:** Permite controlar dinámicamente si el checkbox está marcado o no, basándose en el atributo `admin` de `userForm`.  
**Nota:** Parece que hay un pequeño error tipográfico: debería ser `setChecked` en lugar de `setCheked`. Aquí está corregido:

#### 14.9.3.1 Crear checkbox para habilitar y desabilitar rol admin



```
src > components > UserForm.jsx > UserForm
  5  export const UserForm = ({ userSelected, handlerCloseForm }) => {
  8
  9    <input
 10      className="form-control my-3 w-75" ...
 11      onChange={onInputChange} />
 12      <p className="text-danger">{ errors?.Email}</p>
 13
 14      <div className="my-3 form-check" //3
 15        <input type="checkbox"
 16          name="admin"
 17          checked={admin}
 18          onChange={onCheckboxChange}
 19          className="form-check-input"
 20        />
 21
 22        </div>
 23
 24      <input type="hidden" ...
 25        value={id} />
 26
 27      <button ...
 28        </button>
 29
 30      <button ...
 31        {!handlerCloseForm || <button ...
 32          </button>}
 33
 34      </form>
 35    )
 36  )
```

## 1. Contenedor del Checkbox:

```
Jsx
```

Copiar

```
<div className="my-3 form-check">
```

- Esto crea un contenedor `div` que utiliza las clases `my-3` y `form-check` para aplicar márgenes y estilos específicos.
- Estas clases suelen ser parte de un framework de CSS como Bootstrap, lo que ayuda a darle un diseño estandarizado al formulario.

## 2. Elemento `input` tipo checkbox:

```
Jsx
```

Copiar

```
<input  
  type="checkbox"  
  name="admin"  
  checked={admin}  
  onChange={onCheckboxChange}  
  className="form-check-input"  
/>
```

- `type="checkbox"` : Define que este es un checkbox.
- `name="admin"` : Asigna un nombre al input, útil para identificar este campo al manejar formularios.
- `checked={admin}` : Este valor está controlado mediante el estado `admin`. Si `admin` es `true`, el checkbox estará marcado.
- `onChange={onCheckboxChange}` : Vincula una función que maneja los cambios en el checkbox (como cuando el usuario selecciona o desmarca el campo).
- `className="form-check-input"` : Aplica estilos específicos al checkbox, probablemente también de Bootstrap.

#### 14.9.3.2 crear funcion onCheckboxOnchange

```
>UserForm.jsx X  UsersList.jsx  UserRow.jsx
src > components > UserForm.jsx > [o] UserForm
  1 import { useContext, useEffect, useState } from "react"
  2 import Swal from "sweetalert2";
  3 import { UserContext } from "../context/UserContext";
  4
  5 export const UserForm = ({ userSelected, handlerCloseForm }) => {
  6
  7   const { initialUserForm, handlerAddUser, errors } = useContext(UserContext);
  8
  9   const [userForm, setUserForm] = useState(initialUserForm);
10
11   const {checked,setChecked}=useState(userForm.admin) //2
12
13   const { id, username, password, email , admin } = userForm; //1
14
15 > useEffect(() => {
16   }, [userSelected]);
17
18 >   const onInputChange = ({ target }) => {
19     }
20
21
22 >   const onCheckboxChange=()=>{ //4
23     setChecked(!checked)
24     setUserForm({
25       ...userForm,
26       admin: !checked
27     })
28   }
29
30
31 >   const onSubmit = (event) => {
32     event.preventDefault();
33     if (!errors.length) {
34       handlerAddUser(userForm);
35     }
36   }
37
38
39 >   const onReset = () => {
40     setUserForm(initialUserForm);
41   }
42
43 >   const onLogout = () => {
44     handlerCloseForm();
45   }
46
47 >   return (
48     <div>
49       <h3>User Form</h3>
50       <form>
51         <input type="text" value={username} onChange={onInputChange} />
52         <input type="password" value={password} onChange={onInputChange} />
53         <input type="email" value={email} onChange={onInputChange} />
54         <input checked={checked} type="checkbox" onChange={onCheckboxChange} /> Admin
55         <button type="submit" onClick={onSubmit}>Submit</button>
56         <button type="button" onClick={onReset}>Reset</button>
57         <button type="button" onClick={onLogout}>Logout</button>
58       </form>
59     </div>
60   );
61
62 > }

63 > 
```

## 1. Declaración de la función:

Javascript

 Copiar

```
const onCheckboxChange = () => {
```

- Declara la función `onCheckboxChange` utilizando la sintaxis de funciones flecha.
- Esta función será llamada cuando el usuario interactúe con el checkbox (por ejemplo, al seleccionarlo o desmarcarlo).

## 2. Actualización del estado `checked`:

Javascript

 Copiar

```
setChecked(!checked);
```

- Cambia el valor del estado `checked` al opuesto de su valor actual.
- Si `checked` es `true`, lo establece en `false`, y viceversa.
- Este cambio refleja la acción del usuario sobre el checkbox.

## 3. Actualización del formulario (`userForm`):

Javascript

 Copiar

```
setUserForm({  
    ...userForm,  
    admin: checked  
});
```

- Actualiza el estado `userForm`, que parece ser un objeto que contiene los datos del formulario.
- Usa el operador *spread* (`...userForm`) para mantener intactos los valores actuales de las propiedades del formulario.
- Sobrescribe el valor de la propiedad `admin` con el valor actual de `checked`.

#### 14.9.4 Agregar rol admin en el componente UserList

```
UsersList.jsx X UserRow.jsx
src > components > UsersList.jsx > UsersList
6  export const UsersList = () => {
7
8      const { users } = useContext(UserContext);
9      const {login}=useContext(AuthContext); //. importamos el contexto
10     return (
11         <table className="table table-hover table-striped">
12
13         <thead>...
14         </thead>
15         <tbody>
16             {
17                 users.map(({id, username, email , [admin]})) => ( //5
18                     <UserRow
19                         key={id}
20                         id={id}
21                         username={username}
22                         email={email}
23                         admin={admin} //6
24                     />
25                 ))
26             }
27         </tbody>
28     </table>
29   )
30 }
```

#### 14.9.5 Agregar rol admin en el componente UserRow

UserRow.jsx X

```
src > components > UserRow.jsx > [o] UserRow > ⚡ <function>
  1 import { useContext } from "react"
  2 import { NavLink } from "react-router-dom"
  3 import { UserContext } from "../context/UserContext"
  4 import { AuthContext } from "../auth/context/AuthContext";
  5
  6 export const UserRow = ({id, username, email, admin}) => { //7
  7   const { handlerUserSelectedForm, handlerRemoveUser } = useContext(UserContext);
  8   const {login}=useContext(AuthContext); // importamos el contexto
  9   return (
10     <tr>
11       <td>{id}</td>
12       <td>{username}</td>
13       <td>{email}</td>
14       {!Login.isAdmin && ( >
15         <td>
16           <button
17             type="button"
18             className="btn btn-secondary btn-sm"
19             onClick={() => handlerUserSelectedForm({
20               id,
21               username,
22               email,
23               admin, //8
24             })}
25             >
26               update
27             </button>
28         </td>

```