JWT

1. Introducción a Spring Security

Link: [OpenWebinars]

La seguridad responde a dos preguntas esenciales:

- ¿Quién eres? → Autenticación
- ¿Para qué tienes permiso? → Autorización

👁 Antes de empezar, debemos poner en el pom.xml la siguiente dependencia:

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

También podemos agregarla al crear nuestro proyecto de Spring agregando la siguiente dependencia en Spring Initializr:

spring-boot-starter-security

Java Filter

Funcionalidad donde se asienta la seguridad en Java. Se encuentra entre el cliente y el servidor.

Nos permite dejar pasar peticiones, así como rechazarlas o añadirles una funcionalidad determinada. Por ello, es útil su implementación en seguridad.

Un servlet es algo similar a un controlador.

2. Autenticación y Autorización

Link: [Open Webinars]

2. 1. Autenticación

Spring Security proporciona una interfaz llamada ***AuthenticationManager,*** que implementa el patrón estrategia.

```
public interface AuthenticationManager {

  Authentication authenticate(Authentication authentication)
    throws AuthenticationException;

}
```

Un ***AuthenticationManager*** puede hacer tres cosas con su único método:

- Devolver un Authentication (normalmente con **authenticated=true**)
- Lanzar una excepción de tipo AuthenticationException
- Devolver *null*

https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/core/Authentication.html

2. 2. Autorización

Configuración de la autorización

Hay dos métodos:

- Extendiendo la clase WebSecurityConfigurerAdapter y el uso de AntMatchers.
- A través de anotaciones (**@PreAuthorize** y **@PostAuthorize**)

3. Clases e Interfaces de Spring Security

Link: Open Webinars

3. 1. WebSecurityConfigurerAdapter

Esta clase la crearemos en un paquete propio llamado “security” o “seguridad”, donde implementaremos todas las configuraciones de seguridad necesarias.

3. 2. @EnableWebSecurity

Apaga la configuración por defecto de Spring Boot para aplicarle la que nosotros le estemos pidiendo.

Se utiliza anotando una clase que extienda a ***WebSecurityConfigurerAdapter***.

Interfaz *Principal*

Esta interfaz representa la noción abstracta de un principal, que puede ser usado para representar cualquier entidad, como un individuo, una corporación o una máquina.

Authentication

Interfaz que extiende a Principal. Es fundamental para implementar la autenticación en una aplicación.

Interfaz *UserDetails*

Interfaz fundamental que representa la información nuclear de un usuario para Spring Security.

Si queremos crear una clase propia de gestión de usuarios deberá extender a esta interfaz.

GrantedAuthority

Representa un privilegio individual. Controla el acceso de forma más detallada e individualizada.

Además, podremos crear privilegios más generalizados añadiendo roles de usuario con ROLE_

UserDetailsService

Interfaz que funciona como un DAO de usuarios. Sirve para cargar la información de un usuario.

Se compone de un sólo método:

UserDetails loadUserByUsername (String username).

Si nuestra forma de definir un usuario fuese a raíz de su email, el método pasaría a ser ***UserDetails loadUserByEmail (String email)***.

4. Gestión de usuarios

Link: Open Webinars

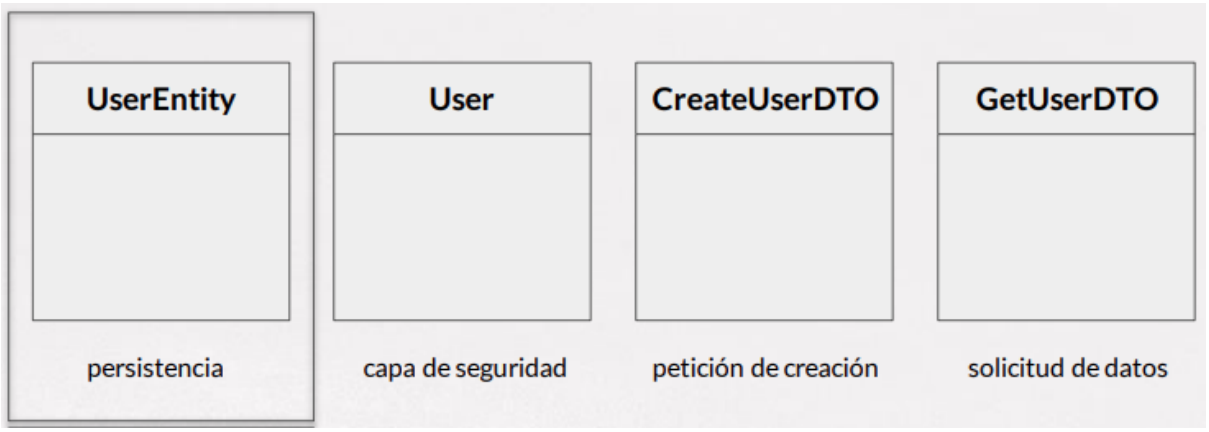
4.1 Modelo de usuario

Representará a una persona que utilice nuestro sistema. Será una entidad, lo que nos facilitará su gestión en la base de datos.

Tendrá la siguiente información básica: nombre de usuario, contraseña y avatar.

Nuestro modelo de datos será implementado con **UserDetails**, el cual nos provee de lo necesario para la gestión de usuarios. Esto conlleva la desventaja de que está **más acoplado** a Spring Security (y sus posibles cambios).

Necesitaremos múltiples clases para gestionar los usuarios: UserEntity, User, CreateUserDTO, GetUserDTO...



Link al proyecto del repositorio: Ejemplo 5 y Ejemplo 6

```java
@Override
public Collection<? extends GrantedAuthority> getAuthorities() {
    return List.of(new SimpleGrantedAuthority( role: "ROLE_" + role.name()));
    /*
    Este metodo simple recibe el nombre del usuario, el cual debe empezar por ROLE_ siempre
     */
}
```

El método name() sirve para extraer el nombre de la enumeración.

Tendremos una *clave primaria natural* (@NaturalId) en el email, que va a ser como nuestro username en este caso.

```java
@NaturalId
@Column(nullable = false,updatable = false)
private String email;
```

Si anotamos un atributo con @NaturalId, deberemos **implementar** los **métodos Equals** y **HashCode**, dado que no pueden existir dos emails iguales.

Guardamos la fecha de la última vez que el usuario cambió su contraseña.

```java
/*
Guarda en la base de datos la ultima vez que se cambió la contraseña, de forma que podamos gestionar ciertos
aspectos del usuario según esta fecha.
 */
@Builder.Default
private LocalDateTime lastPasswordChangedAt = LocalDateTime.now();
```

5. UserDetailsService

Link: OpenWebinars

Repositorio: Ejemplo 7

Es puramente un **DAO para datos de usuario** y no realiza otra función que no sea suministrar esos datos a otros componentes dentro del framework.

```java
/*
Anotamos como un servicio cuyo bean se llamará userDetails, tal y como se especifica en el paréntesis.
 */
@Service("userDetails")
public class CustomUserDetailsService {

}
```

Será utilizada por el **AuthenticationProvider** para obtener el usuario y sus datos. Tiene un solo método: *UserDetails* ***loadUserByUsername(String)***

Implementamos esta interfaz a través de la anotación **@Service("userDetailsService")** en el servicio del CustomUserDetails, que extiende al UserDetailsService.

```
private final UserEntityRepository repository;

/*
Este método nos devolverá una excepción que lanza un error 401 si el usuario no tiene los permisos necesarios.
 */
@Override
public UserDetails loadUserByUsername(String email) throws UsernameNotFoundException {
    return null;
}
```

```
@Override
public UserDetails loadUserByUsername(String email) throws UsernameNotFoundException {
    return repository.findFirstByEmail(email)
            .orElseThrow(() -> new UsernameNotFoundException("" + "no encontrado"));
    /*
    Este método devolverá la información del usuario y, si no se encuentra, devolverá una excepción de tipo
    UserNameNotFoundException que devolverá un String que dirá "El email tal no encontrado".
     */
}
```

Si no se encuentra el usuario, lanzamos una excepción de tipo

UsernameNotFoundException.

Comprobación de las contraseñas

```
public UserEntity save(CreateUserDto newUser) {
    if(newUser.getPassword().equals(newUser.getPassword2())){
        return repository.save(
          UserEntity.builder()
                  .email(newUser.getEmail())
                  .password(newUser.getPassword())
                  .role(UserRole.USER)
                  .build()
        );
    }else{
        /*
        Lo ideal será lanzar una excepción, lo cual veremos más adelante
         */
        return null;
    }
}
```

Tenemos que acordarnos de **cifrar la contraseña**. Para ello, instanciamos una variable private final **PasswordEncoder**.

```java
private final UserEntityRepository repository;
private final PasswordEncoder encoder;//Inyectamos la dependencia que nos permitirá encriptar contraseñas
```

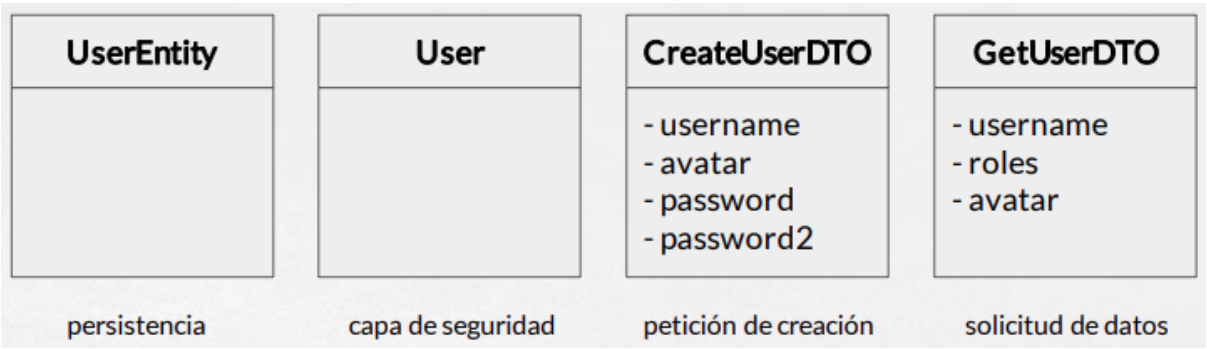
```java
public UserEntity save(CreateUserDto newUser) {
    if(newUser.getPassword().equals(newUser.getPassword2())){
        return repository.save(
          UserEntity.builder()
                  .email(newUser.getEmail())
                  .password(encoder.encode(newUser.getPassword()))//Se guardará encriptada en lugar de como texto plano
                  .role(UserRole.USER)
                  .build()
        );
```

A continuación, vamos a crear un método en el controlador para registrar a un usuario:

```java
private CustomUserDetailsService service;

@GetMapping("/")
/*
Sería necesario cambiar el tipo de devolución ya que UserEntity incluye datos sensibles como la contraseña,
por lo tanto, debemos crear otro DTO sin esa información.
 */
public ResponseEntity<UserEntity> doRegister(@RequestBody CreateUserDto newUser){
    UserEntity created = service.save(newUser);
    if(created == null){
        return ResponseEntity.badRequest().build();
    }else{
        return ResponseEntity.status(201).body(created);
    }
}
```

Es conveniente devolver una DTO, dado que nuestra clase entidad contiene datos como la contraseña del usuario.



6. Seguridad Básica

Link: OpenWebinars

Repositorio: Ejemplo 11 y Ejemplo 12

Método para que un cliente (o navegador web) pueda enviar las credenciales de un usuario (usuario y contraseña) al servidor.

6.1 Configuración de seguridad

Debemos configurar nuestro mecanismo de autenticación (UserDetailsService)

También debemos configurar el control de acceso.

Refactorizamos el código del Password Encoder para sacarlo fuera, a otro bean.

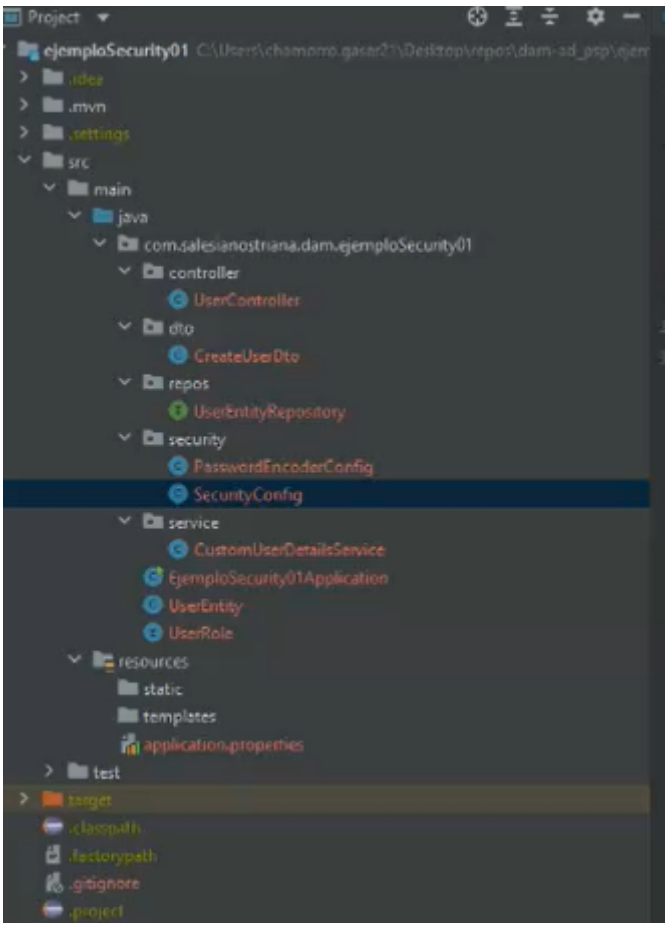
6.2 Control de acceso (autorización)

Lo implementamos sobrescribiendo el método **WebSecurityConfigurerAdapter.configure(HttpSecurity http).**

Identificamos rutas, métodos HTTP y roles que pueden acceder.

Indicamos que la autenticación será básica.

Establecemos un **AuthenticationEntryPoint** personalizado.

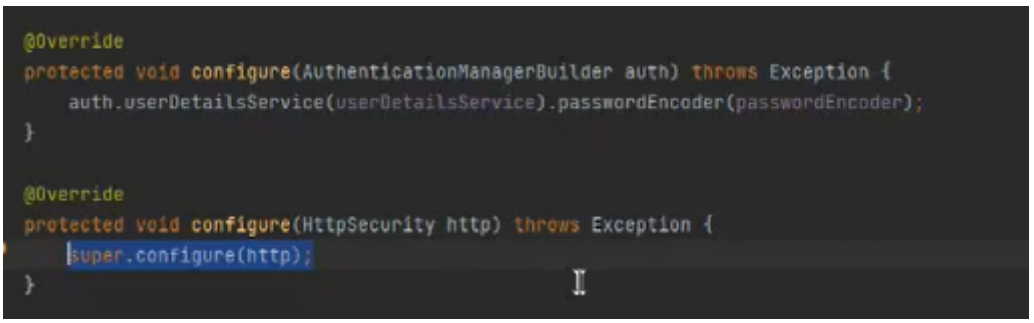



```java
@Configuration //Esta anotación sirve para configurar beans
public class PasswordEncoderConfig {

    @Bean
    public PasswordEncoder passwordEncoder(){
        return new BCryptPasswordEncoder();
    }
}
```

```java
@Configuration
@EnableWebSecurity
@RequiredArgsConstructor
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    private final UserDetailsService userDetailsService;
    private final PasswordEncoder passwordEncoder;
```


@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
auth.userDetailsService(userDetailsService).passwordEncoder(passwordEncoder);
}
@Override
protected void configure(HttpSecurity http) throws Exception {
super.configure(http);
}