

# SPRING FRAMEWORK

## Introducción

Spring Security es el principal marco de trabajo para gestionar la seguridad en aplicaciones basadas en **Spring** y **Spring Boot**. Su robustez y flexibilidad permiten implementar mecanismos avanzados de **autenticación y autorización**, protegiendo los recursos y datos sensibles de la aplicación sin afectar la lógica de negocio.

Entre sus principales características, se incluyen:

- Gestión de protocolos de seguridad.
- Definición de roles y control de acceso a recursos.
- Implementación de autenticación y autorización personalizadas

## 1. Autenticación y Autorización Básica

En **Spring Security**, la **autenticación** te permite verificar la identidad de un usuario, mientras que la **autorización** determina los permisos que este tiene dentro de la aplicación.

Al agregar **Spring Security** a tu proyecto, puedes gestionar estos procesos de manera sencilla y segura.

### Agregar la dependencia

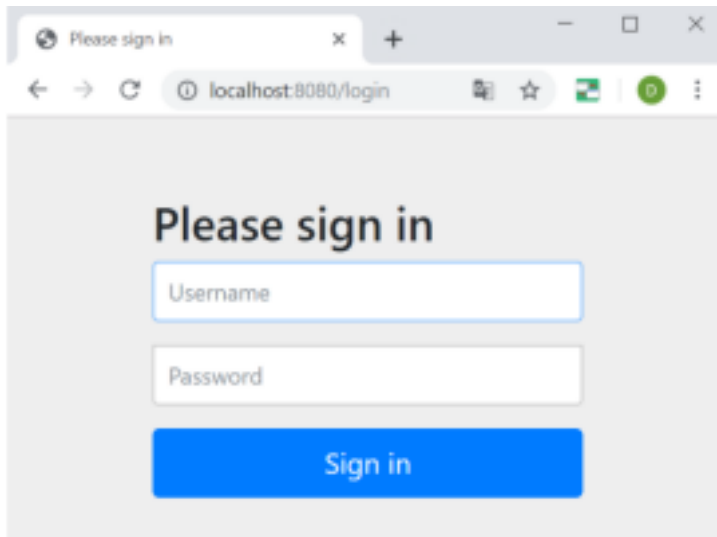
Para comenzar, agrega la siguiente dependencia en el archivo **pom.xml**:

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

### Comportamiento predeterminado

Una vez agregada la dependencia, **Spring Security** aplicará una configuración por defecto. Esto significa que:

- Al intentar acceder a cualquier **endpoint**, se mostrará un formulario de inicio de sesión predeterminado.
- En la consola, verás una contraseña generada automáticamente.
- El nombre de usuario por defecto será **"user"**.



Using generated security password: 30f20aec-fa73-4dee-a972-22f8ff77a1a5

### Personalización de la configuración

Para modificar la configuración de seguridad y definir tus propias reglas de autenticación y autorización, debes crear una clase de configuración personalizada.

## 2. Configuración Básica: Clase de Seguridad

Para configurar la seguridad en tu aplicación, debes crear una clase específica encargada de gestionar las reglas de autenticación y autorización. Esta clase debe ubicarse en el **paquete raíz** de tu proyecto para que Spring la detecte correctamente.

### Implementación en código

Aquí tienes la implementación de la clase **SeguridadWeb**:

```
package com.egg.biblioteca;
```

```

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.web.SecurityFilterChain;

@Configuration
@EnableWebSecurity
public class SeguridadWeb {

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

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http
            .authorizeHttpRequests((authorize) -> authorize
                .requestMatchers("/css/", "/js/", "/img/", "/*").permitAll()
            )
            .csrf(csrf -> csrf.disable());
        return http.build();
    }
}

```

## Desglose de la configuración

A continuación, se explican las principales anotaciones y métodos utilizados en la configuración de seguridad:

- **@Configuration**: Indica que la clase **SeguridadWeb** es una clase de configuración de Spring, encargada de definir **beans** que serán administrados por el contenedor de Spring.
- **@EnableWebSecurity**: Habilita la seguridad web en la aplicación, permitiendo que definas reglas personalizadas.
- **@Bean**: Marca el método **filterChain** como un **bean** administrado por Spring.
- **SecurityFilterChain**: Define una cadena de filtros de seguridad que procesan las solicitudes HTTP. Aquí es donde se establecen las reglas de acceso.
- **Método filterChain**: Configura las reglas de seguridad utilizando un objeto **HttpSecurity**.
  - **authorizeHttpRequests**: Permite definir qué solicitudes pueden acceder a la aplicación sin autenticación. En este caso, los archivos CSS, JS e imágenes son de acceso público.
  - **csrf.disable()**: Deshabilita la protección contra ataques CSRF (*Cross-Site Request Forgery*). Esto es útil si no estás manejando formularios de autenticación en esta configuración.

- **http.build()**: Finaliza la configuración de seguridad y devuelve un objeto **SecurityFilterChain** configurado.

Esta configuración inicial establece reglas de autorización para las solicitudes HTTP en tu aplicación. Permite el acceso sin autenticación a los archivos estáticos (CSS, JS, imágenes), y deshabilita la protección CSRF en esta fase inicial.

### 3. ROLES Y AUTORIZACIÓN

Para gestionar los permisos dentro de su aplicación, debe definir los **roles** de usuario. En este caso, utilizará un **Enum** para almacenar los roles, ya que son específicos y concretos. Si en el futuro necesita agregar más roles, podrá hacerlo fácilmente.

#### Definir los roles con un Enum

Cree una clase del tipo **Enum** que contenga los roles disponibles en la aplicación.

```
package com.egg.biblioteca.enumeraciones;
public enum Rol {
    USER,
    ADMIN;
}
```

#### Creación de entidad “Usuario”

Ahora, defina la clase **Usuario**, que representará a los usuarios en la base de datos.

```
@Entity
public class Usuario {
    @Id
    @GeneratedValue (generator = "uuid")
    @GenericGenerator (name ="uuid", strategy = "uuid2")
    private String id;

    private String nombre;
    private String email;
    private String password;

    @Enumerated(EnumType.STRING)
    private Rol rol;
    // Constructor Vacío
}
```

```
//Getter y Setters  
}
```

## Creación de repositorio “Usuario”

El repositorio de usuarios debe ser una **interfaz** que extienda **JpaRepository**.

```
package com.egg.biblioteca.repositorios;  
  
import com.egg.biblioteca.entidades.Usuario;  
import org.springframework.data.jpa.repository.JpaRepository;  
import org.springframework.data.jpa.repository.Query;  
import org.springframework.data.repository.query.Param;  
import org.springframework.stereotype.Repository;  
  
@Repository  
public interface UsuarioRepositorio extends JpaRepository<Usuario, String> {  
  
    @Query("SELECT u FROM Usuario u WHERE u.email = :email")  
    public Usuario buscarPorEmail(@Param("email")String email);  
}
```

Este método permite buscar usuarios por su correo electrónico, el cual, en la mayoría de los casos, se utiliza como nombre de usuario.

## Crear el servicio UsuarioServicio

Para gestionar los usuarios, debe crear un **servicio** que se conecte al repositorio. Este servicio debe estar anotado con **@Service** y **debe implementar la interfaz UserDetailsService**.

```
@Service  
  
public class UsuarioServicio implements UserDetailsService {  
  
    @Autowired  
    private UsuarioRepositorio usuarioRepositorio;  
    //Los métodos del servicio  
}
```

```
@Override  
    public UserDetails loadUserByUsername(String email) throws  
    UsernameNotFoundException {  
  
        Usuario usuario = usuarioRepositorio.buscarPorEmail(email);
```

```

        if (usuario != null) {
            List<GrantedAuthority> permisos = new ArrayList<>();
            GrantedAuthority p = new SimpleGrantedAuthority("ROLE_" +
usuario.getRol().toString());
            permisos.add(p);
            ServletRequestAttributes attr = (ServletRequestAttributes)
RequestContextHolder.currentRequestAttributes();
            HttpSession session = attr.getRequest().getSession(true);
            session.setAttribute("usuariosession", usuario);
            return new User(usuario.getEmail(), usuario.getPassword(),
permisos);
        } else {
            return null;
        }
    }
}

```

### Explicación del método `loadUserByUsername`

Este método forma parte de la implementación de `UserDetailsService` y su función es cargar los datos del usuario en base a su correo electrónico.

#### Funcionamiento del método

1. **Recibe el correo electrónico** del usuario como parámetro.
2. **Busca en la base de datos** un usuario con ese correo.
3. **Si encuentra un usuario:**
  - Crea una lista de permisos (`GrantedAuthority`).
  - Agrega un permiso basado en su rol (por ejemplo, `"ROLE_ADMIN"` si el usuario es administrador).
  - Devuelve un objeto `UserDetails` con el email, la contraseña y los permisos.
4. **Si no encuentra el usuario**, devuelve `null`.

## 4. Método de Registro en nuestro Controlador

Siguiendo la arquitectura **Modelo-Vista-Controlador (MVC)**, debes definir un método en el **controlador** que reciba los datos ingresados en el formulario de registro y llame al servicio correspondiente para guardar al usuario en la base de datos.

## 5. Encriptación de la contraseña

El uso de **BCryptPasswordEncoder** en Spring Security es una práctica recomendada para almacenar contraseñas de manera segura en la base de datos. Esto se debe a varias razones:

- **Seguridad de contraseñas:** BCrypt es un algoritmo de hash diseñado específicamente para almacenar contraseñas de forma segura. Utiliza un proceso de hash adaptativo que incorpora sal y múltiples rondas de iteración, lo que lo hace altamente resistente a ataques de fuerza bruta y de diccionario.
- **Prevención de ataques de ingeniería inversa:** Al emplear una función de hash irreversible como BCrypt, se dificulta enormemente la posibilidad de recuperar la contraseña original a partir del hash almacenado. Esto protege la información sensible de los usuarios incluso en caso de una brecha de seguridad.
- **Integración con Spring Security:** Spring Security proporciona soporte nativo para **BCrypt** a través de la clase **BCryptPasswordEncoder**, lo que facilita su implementación dentro de la aplicación. Con este mecanismo, puedes codificar las contraseñas antes de almacenarlas en la base de datos y verificarlas de manera segura durante el proceso de autenticación.

### Implementación

Para garantizar la seguridad en tu aplicación, debes configurar **BCryptPasswordEncoder** de la siguiente manera:


- **En el servicio de usuarios:** Dentro del método encargado de manejar el registro de usuarios, debes encriptar la contraseña antes de almacenarla en la base de datos.

```
usuario.setPassword(new BCryptPasswordEncoder().encode(password));
```

- **En la configuración de seguridad:** Debes declarar el método **passwordEncoder()** dentro de una clase de configuración de Spring Security. Al anotarlo con **@Bean**, Spring gestionará esta instancia de **BCryptPasswordEncoder**, asegurando que sea utilizada en toda la aplicación.

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

Esta configuración garantiza que las contraseñas de los usuarios se almacenen de forma segura y puedan ser verificadas correctamente durante el proceso de autenticación.

Puedes acceder a la documentación oficial de **Spring Security** en el siguiente enlace:  [Spring Security Reference Documentation](#). Ahí encontrarás información detallada sobre autenticación, autorización, configuración de seguridad y mejores prácticas.