

# Spring Security con Spring Boot

DigitalHouse>



**Certified Tech  
Developer**

The Ultimate Degree

# Pasos



# 1- Agregar la librería, el Starter de Spring Security

Primero debemos asegurarnos de agregar la dependencia de **Spring Security** en nuestro archivo [pom.xml](#):

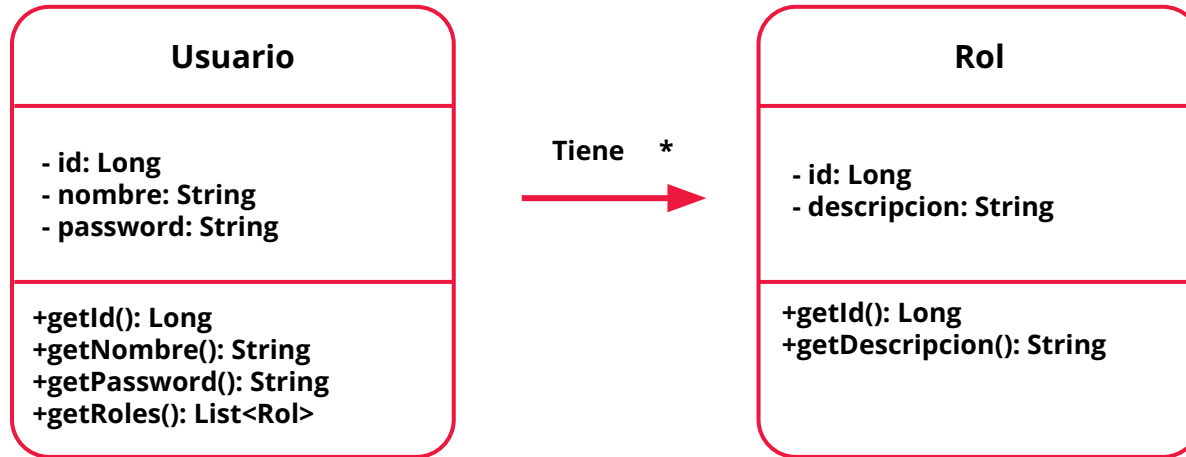
```
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-security</artifactId>  
</dependency>
```

## 2- Configurar la administración de los roles & user

Para poder tomar decisiones sobre el acceso a los recursos es necesario *identificar a los diferentes usuarios y que roles tienen* para validar o no si tienen autorización para acceder a los diferentes recurso de la aplicación.

Para ello debemos implementar la interfaz **UserDetailsService**. Esta interfaz describe un objeto que realiza un acceso a datos con un único método **loadUserByUsername** que devuelve la información de un usuario a partir de su nombre de usuario.

## 2- Configurar la administración de los roles & user



```
@Service
@Transactional
public class UserDetailsServiceImpl implements UserDetailsService {
    @Autowired
    UserRepository userRepository;

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

        Usuario appUser = userRepository.findByUsername(username);

        Set<GrantedAuthority> grantList = new HashSet<GrantedAuthority>();

        for (Rol rol: appUser.getRoles()) {
            GrantedAuthority grantedAuthority = new SimpleGrantedAuthority(rol.getDescription());
            grantList.add(grantedAuthority);
        }

        UserDetails user = null;
        user = (UserDetails) new User(username, "{noop}" + appUser.getPassword(), grantList);

        return user;
    }
}
```

Busca el usuario por  
nombre en nuestra  
base de datos

```

@Service
@Transactional
public class UserDetailsServiceImpl implements UserDetailsService {
    @Autowired
    UserRepository userRepository;

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

        Usuario appUser = userRepository.findByUsername(username);

        Set<GrantedAuthority> grantList = new HashSet<GrantedAuthority>();

        for (Rol rol: appUser.getRoles()) {
            GrantedAuthority grantedAuthority = new SimpleGrantedAuthority(rol.getDescription());
            grantList.add(grantedAuthority);
        }

        UserDetails user = null;
        user = (UserDetails) new User(username, "{noop}" + appUser.getPassword(), grantList);

        return user;
    }
}

```

Crea la lista de roles/accesos que tiene el usuario

```

@Service
@Transactional
public class UserDetailsServiceImpl implements UserDetailsService {
    @Autowired
    UserRepository userRepository;

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        Usuario appUser = userRepository.findByUsername(username);

        Set<GrantedAuthority> grantList = new HashSet<GrantedAuthority>();

        for (Rol rol: appUser.getRoles()) {
            GrantedAuthority grantedAuthority = new SimpleGrantedAuthority(rol.getDescription());
            grantList.add(grantedAuthority);
        }

        UserDetails user = null;
        user = (UserDetails) new User(username, "{noop}" + appUser.getPassword(), grantList);

        return user;
    }
}

```

En la base de datos  
cada rol tiene  
que tener  
antepuesto "ROLE\_"  
Ej: ROLE\_USER  
ROLE\_ADMIN



```

@Service
@Transactional
public class UserDetailsServiceImpl implements UserDetailsService {
    @Autowired
    UserRepository userRepository;

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

        Usuario appUser = userRepository.findByUsername(username);

        Set<GrantedAuthority> grantList = new HashSet<GrantedAuthority>();

        for (Rol rol: appUser.getRoles()) {
            GrantedAuthority grantedAuthority = new SimpleGrantedAuthority(rol.getDescripcion());
            grantList.add(grantedAuthority);
        }

        UserDetails user = null;
        user = (UserDetails) new User(username, "{noop}" + appUser.getPassword(), grantList);

        return user;
    }
}

```

Crea y retorna el  
objeto  
UserDetails  
soportado por  
Spring Security

anteponer "noop" si la password no esta encriptada en la base de datos.

### 3- Configurar la autenticación y seguridad de URL

Usando la anotación `@EnableWebSecurity` y extendiendo la clase `WebSecurityConfigurerAdapter` podemos rápidamente configurar y activar la seguridad para los diferentes usuarios que se loguean en nuestra aplicación.

A su vez, `@EnableWebSecurity` habilita el soporte de seguridad web de Spring Security y también proporciona la integración con Spring MVC y `WebSecurityConfigurerAdapter` proporciona un conjunto de métodos que se utilizan para habilitar la configuración de seguridad web específica.

Como vamos a usar nuestra propia configuración debemos crear una clase que herede de `WebSecurityConfigurerAdapter`, y en ella sobrescribir el método **configure** donde vamos a personalizar nuestra propia configuración de seguridad.

Indica que la clase es de configuración y necesita ser cargada durante el inicio del server.

```
@Configuration
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        // nuestra propia configuración de seguridad.
    }
}
```

Como vimos en nuestra clase de configuración de seguridad debemos sobrescribir el método `configure()` para habilitar la protección de las URL.

```
protected void configure(HttpSecurity http) throws Exception {  
    http.authorizeRequests()  
        .anyRequest()  
        .authenticated()  
        .and()  
        .httpBasic();  
}
```

Activa la protección HTTP básica, normalmente el navegador muestra un cuadro donde se nos pedirá introducir nombre de usuario y contraseña.

Veamos un ejemplo. Este genera un formulario de login automáticamente y es el que estaremos utilizando.

```
protected void configure(HttpSecurity http) throws Exception {  
    http.authorizeRequests()  
        .antMatchers("/home").hasRole("USER")  
        .antMatchers("/ventas").hasRole("ADMIN")  
        .and().formLogin()  
        .and().logout();  
}
```

```
protected void configure(HttpSecurity http) throws Exception {  
    http.authorizeRequests()  
        .antMatchers("/home").hasRole("USER")  
        .antMatchers("/ventas").hasRole("ADMIN")  
        .and().formLogin()  
        .and().logout();  
}
```

Indica que todas las peticiones estarán protegidas, es decir, requerimos autenticarnos para poder acceder a cualquier parte del sitio.

```
protected void configure(HttpSecurity http) throws Exception {  
    http.authorizeRequests()  
        .antMatchers("/home").hasRole("USER")  
        .antMatchers("/ventas").hasRole("ADMIN")  
        .and().formLogin()  
        .and().logout();  
}
```

Se utiliza para indicar que un usuario con el rol de USER puede ingresar a la home.

```
protected void configure(HttpSecurity http) throws Exception {  
    http.authorizeRequests()  
        .antMatchers("/home").hasRole("USER")  
        .antMatchers("/ventas").hasRole("ADMIN")  
        .and().formLogin()  
        .and().logout();  
}
```

Se utiliza para indicar que un usuario con el rol de ADMIN solo puede ingresar a la página de ventas.



## 4- Encriptar la contraseña

Las claves de usuario pueden ser codificadas a partir de un *algoritmo de encriptación*.

Una función de encriptación es una función unilateral que toma cualquier texto y lo transforma en un código encriptado que no se puede volver atrás.

Encriptar/codificar las contraseñas nos permite almacenarlas de forma segura.

**Spring Security** proporciona múltiples implementaciones de codificación de contraseña para elegir. Cada uno tiene sus ventajas y desventajas, y un desarrollador puede elegir cuál usar dependiendo del requisito de autenticación de su aplicación. Por fines práctico veremos **BCryptPasswordEncoder**.

## BCryptPasswordEncoder

Instanciando un objeto de la clases `BCryptPasswordEncoder`, podemos encriptar/hashear un pass, para ellos debemos:

- 1) Creamos encoder llamando al constructor de `BCryptPasswordEncoder` con un valor 12. Este valor puede ser entre 4 a 31 y cuanto más grande sea, más trabajo se necesita para calcular el hash.

```
BCryptPasswordEncoder encoder = new  
BCryptPasswordEncoder(12);
```

- 2) Invocamos al método `encode("pass_para_hashear")`, pasando la contraseña que deseamos encriptar, Así es como se ve una contraseña con hash, por ejemplo: **encodedPassword:**

`$2a$12$DIfnjD4YgCNbDEtgd/ITeOj.jmUZpuz1i4gt51YzetW/iKY203bqa`

```
String encodedPassword = encoder.encode("UserPassword");
```

## Uso de @PreAuthorize y @PostAuthorize

La anotación **@PreAuthorize** verifica la expresión dada antes de ingresar al método, para decidir si un usuario en sesión tiene acceso o no a utilizarlo, mientras que la anotación **@PostAuthorize** la verifica después de la ejecución del método y podría alterar el resultado.

```
@PreAuthorize("hasAnyRole('ROLE_ADMIN','ROLE_USER')")
public User updateUser(User formUser) throws Exception {
    ...
}
```

DigitalHouse>