

➤ Control de Acceso mediante Spring Security

Plan formativo: Desarrollo de Aplicaciones Full Stack Java Trainee V2.0

HOJA DE RUTA

¿Cuáles **skill** conforman el programa?



REPASO CLASE ANTERIOR

En la clase anterior trabajamos :

- ✓ Administrar usuarios con Spring Security
- ✓ Configurar login y logout con Spring Security

LEARNING PATHWAY

¿Sobre qué temas trabajaremos?

6.3

Start! 🏁

**Control de Acceso
mediante Spring
Security**

Roles

ROLE_ADMIN

Manejo de roles

Añadiendo seguridad a los elementos de la capa de vista

Obteniendo el usuario autenticado en el controlador

OBJETIVOS DE APRENDIZAJE

¿Qué aprenderemos?



Aprender a manejar roles de usuarios autenticados



Comprender la configuración de seguridad de la vista con Spring Security



› Manejo de Roles



Manejo de Roles



La gestión de roles es un componente crucial en el diseño de sistemas de seguridad para aplicaciones web. Spring Security, un módulo de seguridad integral para aplicaciones basadas en Spring, proporciona un marco robusto para gestionar roles de manera efectiva.





Manejo de Roles

Roles en el Contexto de la Seguridad:

Los roles son un concepto central en el **control de acceso y la autorización**. Representan conjuntos de permisos asociados a usuarios específicos. La gestión de roles permite definir quién tiene acceso a determinados recursos o funcionalidades dentro de una aplicación.





Manejo de Roles

Implementación:

En Spring Security, la gestión de roles se realiza a través de configuraciones en el archivo de seguridad y mediante anotaciones en el código. Los roles son atribuidos a usuarios y se utilizan para restringir el acceso a partes específicas de la aplicación.

Por ejemplo, un usuario con el rol "**ADMIN**" podría tener acceso a funcionalidades administrativas, mientras que un usuario con el rol "**USER**" podría tener acceso a funciones básicas.





Manejo de Roles

Ventajas de la gestión de roles con Spring Security:

Granularidad en la Autorización: permite definir roles específicos para diferentes conjuntos de acciones, proporcionando un control preciso sobre quién puede hacer que dentro de la aplicación.

Estructura Escalable: al utilizar roles, se crea una estructura escalable que puede adaptarse a medida que la aplicación crece. Puedes añadir nuevos roles para gestionar diferentes niveles de acceso sin cambiar drásticamente la lógica de autorización existente.



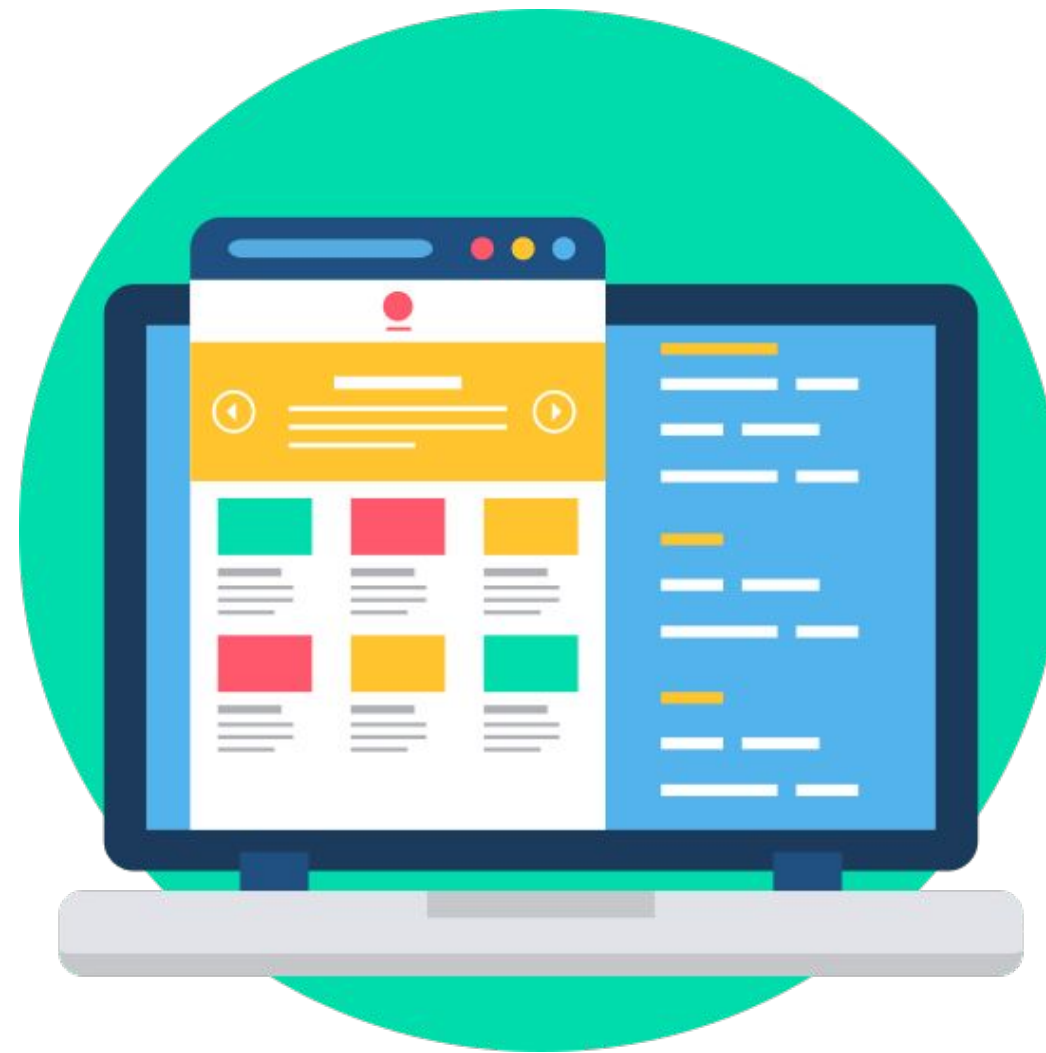


Manejo de Roles

Ventajas de la gestión de roles con Spring Security:

Seguridad Dinámica: Spring Security permite una asignación dinámica de roles en tiempo de ejecución, lo que significa que puedes ajustar la autorización de un usuario sin necesidad de reiniciar la aplicación. Esto es útil en entornos donde los roles pueden cambiar frecuentemente.

Integración con Anotaciones: Spring Security facilita la autorización basada en roles mediante el uso de anotaciones, como `@Secured`, `@PreAuthorize` y `@PostAuthorize`. Estas anotaciones permiten especificar roles directamente en el código, mejorando la legibilidad y el mantenimiento.





Manejo de Roles

Desventajas de la gestión de roles con Spring Security:



Complejidad en Aplicaciones Grandes: En aplicaciones grandes y complejas, la gestión de roles puede volverse complicada. La proliferación de roles específicos y la necesidad de gestionar quién tiene acceso a qué recursos pueden aumentar la complejidad y la posibilidad de errores.



Rigidez en la Estructura: La asignación de roles puede resultar rígida en algunos casos. Por ejemplo, si un usuario necesita un conjunto de permisos específicos que no encajan exactamente en un rol existente, puede ser necesario crear un nuevo rol o utilizar lógica adicional.

Dificultades en la Implementación Inicial: La implementación inicial de la gestión de roles puede requerir tiempo y esfuerzo, especialmente al definir roles y asignarlos correctamente a usuarios. Esto puede ser más evidente en proyectos pequeños donde la autorización basada en roles puede parecer una sobrecarga inicial.



Manejo de Roles

Para comenzar, vamos a definir cuáles roles vamos a manejar dentro de nuestra aplicación. Hay diversas maneras de crear los roles, pero en este caso como son roles muy puntuales los manejaremos desde la aplicación en un **Enum**.



```
public enum Role {  
    USER,  
    ADMIN;  
}
```





Manejo de Roles



En la clase entidad de usuario, en este ejemplo: **UserEntity**, contamos con los atributos **username** y **password**. Ahora también le agregaremos el atributo **rol**.



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

    private String username;

    private String password;

    @Enumerated(EnumType.STRING)
    private Role role;

    private String email;
}
```





Manejo de Roles


También es necesario adaptar el método **configure** de la clase de seguridad para indicar el comportamiento que debe tener la aplicación según el rol que tenga el usuario.

```
30  @Override
31  protected void configure(HttpSecurity http) throws Exception {
32      http
33          .authorizeHttpRequests()
34              .antMatchers("/admin/*").hasRole("ADMIN")
35              .antMatchers("/css/*", "/js/*", "/img/*", "/*")
36                  .permitAll()
37          .and().formLogin()
38              .loginPage("/login")
39              .loginProcessingUrl("/logincheck")
40              .usernameParameter("email")
41              .passwordParameter("password")
42              .defaultSuccessUrl("/start")
43              .permitAll()
44          .and().logout()
45              .logoutUrl("/logout")
46              .logoutSuccessUrl("/login")
47              .permitAll()
48          .and().csrf()
49              .disable();
50  }
```





Manejo de Roles

34- `.antMatchers("/admin/*").hasRole("ADMIN")`: indica que a todos los endpoints que nazcan desde /admin sólo podrán acceder aquellos usuarios que tengan el Rol ADMIN asignado. 

35- `.antMatchers("/css/*", "/js/*", "/img/*", "/*").permitAll()` : indica que todos aquellos archivos que provengan de las carpetas indicadas entre paréntesis, serán permitidos para todos los usuarios (estilos, animaciones, imágenes, etc)



```
30  @Override
31  protected void configure(HttpSecurity http) throws Exception {
32      http
33          .authorizeHttpRequests()
34              .antMatchers("/admin/*").hasRole("ADMIN")
35              .antMatchers("/css/*", "/js/*", "/img/*", "/*")
36                  .permitAll()
37          .and().formLogin()
38              .loginPage("/login")
39              .loginProcessingUrl("/logincheck")
40              .usernameParameter("email")
41              .passwordParameter("password")
42              .defaultSuccessUrl("/start")
43              .permitAll()
44          .and().logout()
45              .logoutUrl("/logout")
46              .logoutSuccessUrl("/login")
47              .permitAll()
48          .and().csrf()
49              .disable();
50  }
```





Manejo de Roles

A continuación tenemos las propiedades de **login** y **logout**, que también podrán ser accedidas por todos los usuarios.

En las líneas que indican los parámetros **.usernameParameter("email")** y **.passwordParameter("password")**, debemos recordar que el parámetro entre paréntesis deben tener el mismo nombre que el atributo a evaluar.

En este caso, el “nombre de usuario” o **username** está determinado por el atributo **email** del **User**.

```
30 @Override
31 protected void configure(HttpSecurity http) throws Exception {
32     http
33         .authorizeHttpRequests()
34             .antMatchers("/admin/*").hasRole("ADMIN")
35             .antMatchers("/css/*", "/js/*", "/img/*", "/*")
36                 .permitAll()
37         .and().formLogin()
38             .loginPage("/login")
39             .loginProcessingUrl("/logincheck")
40             .usernameParameter("email")
41             .passwordParameter("password")
42             .defaultSuccessUrl("/start")
43             .permitAll()
44         .and().logout()
45             .logoutUrl("/logout")
46             .logoutSuccessUrl("/login")
47             .permitAll()
48         .and().csrf()
49             .disable();
50 }
```

Evaluación Integradora ✨

¿Listos para un nuevo desafío? En esta clase comenzamos a construir nuestro...

Trabajo Integrador del Módulo 💪

Iremos completándolo progresivamente clase a clase.



LIVE CODING

Ejemplo en vivo

Login en la Wallet

- *Vamos a agregar las configuraciones de roles y permisos al proyecto.*

1- Implementar un **enum** con **roles: Admin y User**

2- Personalizar el método **configure** para definir las partes de la aplicación a la que podrán acceder cada tipo de usuario según su rol.

  **Tiempo: 20 minutos**



➤ Añadiendo seguridad a los elementos de la vista

Añadiendo seguridad a los elementos de la vista

Spring Security cuenta con **taglib** que nos provee acceso a la información de seguridad de los usuarios y posibilita el filtrado del contenido que se muestra al usuario en base a sus privilegios. Para utilizar **taglib** en una página **JSP**, debemos agregar la siguiente dependencia:

```
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-taglibs</artifactId>
  <version>${security.version}</version>
</dependency>
```

Añadiendo seguridad a los elementos de la vista

Y declaramos el taglib en el archivo JSP de la siguiente manera:

```
*index.jsp ×  
1 <%@ taglib prefix="sec" uri="http://www.springframework.org/security/tags" %>  
2
```

Añadiendo seguridad a los elementos de la vista

Si en lugar de utilizar una página JSP, decidimos utilizar Thymeleaf con HTML, debemos incluir la dependencia:

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

Añadiendo seguridad a los elementos de la vista

Y agregar la declaración de seguridad en el HTML, de ésta manera:

```
<!DOCTYPE html>  
<html xmlns:th="http://www.thymeleaf.org"  
      xmlns:sec="http://www.thymeleaf.org/extras/spring-security">
```


Añadiendo seguridad a los elementos de la vista

Una vez configurado el prefijo “**sec**”, podemos acceder a etiquetas que nos permitirán acceder a diversas personalizaciones.

La etiqueta **authorize** nos sirve para **filtrar el contenido que se mostrará en base a los privilegios del usuario**, por ejemplo, podemos definir un contenido que solo será mostrado a aquellos usuarios con privilegios de administrador, esta etiqueta la utilizaremos de la siguiente manera:

```
<sec:authorize access="hasRole('ROLE_ADMIN')">

    Este contenido se muestra si el usuario tiene el privilegio:
    <b>ROLE_ADMIN</b>.

</sec:authorize>
```

Añadiendo seguridad a los elementos de la vista

Estableciendo el atributo `access="isAuthenticated()"` **mostraremos contenido a todos los usuarios autenticados.**

```
<sec:authorize access= access="isAuthenticated() and principal.username == 'jesus'">

    Este contenido se muestra si:
    <b>Si el username del usuario autenticado es "jesus"</b>.

</sec:authorize>
```

Añadiendo seguridad a los elementos de la vista

Otra forma de usar esta etiqueta es indicando el atributo **url**, por medio del mismo, el contenido será generado si el usuario actual tiene permiso para acceder a la URL indicada.

```
<sec:authorize url="/user/delete">  
  
    Este contenido se muestra si:  
    <b>Si el usuario autenticado tiene permisos para acceder a  
    la url especificada</b>.  
  
</sec:authorize>
```

Añadiendo seguridad a los elementos de la vista

Si deseamos obtener el nombre del usuario y sus roles, usaremos las siguientes etiquetas:

```
Nombre de usuario:  
<sec:authentication property="principal.username" />  
  
Roles de usuario.  
<sec:authentication property="principal.authorities" />
```


➤ Obteniendo el usuario autenticado en el controlador



Obteniendo el usuario autenticado en el controlador



Para obtener el usuario autenticado en un controlador en una aplicación Spring Security, podemos hacer uso de la clase `SecurityContextHolder`. Esta clase proporciona acceso al contexto de seguridad de Spring, y puedes extraer información sobre el usuario autenticado. El usuario autenticado generalmente se almacena en un objeto `Authentication`.



Obteniendo el usuario autenticado en el controlador

Primero debemos tener los imports necesarios:

- `import org.springframework.security.core.Authentication;`
- `import org.springframework.security.core.context.SecurityContextHolder;`
- `import org.springframework.security.core.userdetails.UserDetails;`

Obteniendo el usuario autenticado en el controlador

Luego, en el método del controlador donde deseamos obtener el usuario actual, podemos hacer lo siguiente:

```
@GetMapping("/profile")
public String userProfile(Model model) {
    // Obtener el contexto de seguridad actual
    Authentication authentication = SecurityContextHolder.getContext().getAuthentication();

    // Obtener el nombre de usuario del usuario autenticado
    String username = authentication.getName();

    // Puedes obtener más información sobre el usuario, como roles, detalles, etc.
    // UserDetails userDetails = (UserDetails) authentication.getPrincipal();
    // List<GrantedAuthority> authorities = (List<GrantedAuthority>) authentication.getAuthorities();

    // Agregar el nombre de usuario al modelo para mostrarlo en la vista
    model.addAttribute("username", username);

    // Retornar la vista del perfil del usuario
    return "profile";
}
```


Obteniendo el usuario autenticado en el controlador

`SecurityContextHolder.getContext().getAuthentication()`: Obtiene el contexto de seguridad actual y, a partir de él, obtiene la información de autenticación.

```
@GetMapping("/profile")
public String userProfile(Model model) {
    // Obtener el contexto de seguridad actual
    Authentication authentication = SecurityContextHolder.getContext().getAuthentication();

    // Obtener el nombre de usuario del usuario autenticado
    String username = authentication.getName();

    // Puedes obtener más información sobre el usuario, como roles, detalles, etc.
    // UserDetails userDetails = (UserDetails) authentication.getPrincipal();
    // List<GrantedAuthority> authorities = (List<GrantedAuthority>) authentication.getAuthorities();

    // Agregar el nombre de usuario al modelo para mostrarlo en la vista
    model.addAttribute("username", username);

    // Retornar la vista del perfil del usuario
    return "profile";
}
```

Obteniendo el usuario autenticado en el controlador

`authentication.getName()`: Extrae el nombre de usuario del objeto `Authentication`. También puedes acceder a otras propiedades del usuario autenticado según tu configuración, como roles, detalles, etc.

```
@GetMapping("/profile")
public String userProfile(Model model) {
    // Obtener el contexto de seguridad actual
    Authentication authentication = SecurityContextHolder.getContext().getAuthentication();

    // Obtener el nombre de usuario del usuario autenticado
    String username = authentication.getName();

    // Puedes obtener más información sobre el usuario, como roles, detalles, etc.
    // UserDetails userDetails = (UserDetails) authentication.getPrincipal();
    // List<GrantedAuthority> authorities = (List<GrantedAuthority>) authentication.getAuthorities();

    // Agregar el nombre de usuario al modelo para mostrarlo en la vista
    model.addAttribute("username", username);

    // Retornar la vista del perfil del usuario
    return "profile";
}
```

Obteniendo el usuario autenticado en el controlador

`model.addAttribute("username", username)`: Agrega el nombre de usuario al modelo de Spring, lo que permitirá acceder a esta información en la vista asociada.

`return "profile"`: Retorna el nombre de la vista que mostrará el perfil del usuario.

```
@GetMapping("/profile")
public String userProfile(Model model) {
    // Obtener el contexto de seguridad actual
    Authentication authentication = SecurityContextHolder.getContext().getAuthentication();

    // Obtener el nombre de usuario del usuario autenticado
    String username = authentication.getName();

    // Puedes obtener más información sobre el usuario, como roles, detalles, etc.
    // UserDetails userDetails = (UserDetails) authentication.getPrincipal();
    // List<GrantedAuthority> authorities = (List<GrantedAuthority>) authentication.getAuthorities();

    // Agregar el nombre de usuario al modelo para mostrarlo en la vista
    model.addAttribute("username", username);

    // Retornar la vista del perfil del usuario
    return "profile";
}
```

Momento: ✚

Time-out!

🕒 5 min.





Ejercicio N° 1

ROLE_ADMIN



ROLE_ADMIN

Contexto: 🙌

Vamos a continuar con el ejercicio de la clase anterior. En este caso te toca a ti configurar los roles de usuarios de tu aplicación.



Consigna: ✍️

- 1- Implementar un enum que permita diferenciar un usuario ADMIN de un usuario común,
- 2- Agregar el atributo Rol a la entidad de Usuario.

Tiempo 🕒: 10 minutos (puedes continuar de manera asíncrona)

○

¿Alguna consulta?

+



RESUMEN

¿Qué logramos en esta clase?

- ✓ **Comprender la administración de roles de usuarios con Spring Security**
- ✓ **Aprender a configurar la seguridad según autenticación**



#WorkingTime

Continuemos ejercitando

¡Antes de cerrar la clase! Te invitamos a: 📌 📌 📌

1. Repasar nuevamente la grabación de esta clase
2. Revisar el material compartido en la plataforma de Moodle (lo que se vio en clase y algún ejercicio adicional)
 - a. *Lectura Modulo 6, Lección 4: páginas 9 - 13*
3. Traer al próximo encuentro, todas tus dudas y consultas para verlas antes de iniciar nuevo tema.

¡Muchas Gracias!

Nos vemos en la próxima clase 🙌

