**Ejercicio Roles y Permisos**

**Configurando Roles y Permisos**

En base a la arquitectura que vimos de manera teórica en clases anteriores y basados en el funcionamiento de cada una de sus partes, tomaremos como fuente el proyecto base de seguridad que llevamos a cabo la clase pasada y agregaremos las partes de la arquitectura que nos faltan.

**Paso 0: Security Filter Chain**

Como sabemos, en nuestro proyecto base ya configuramos la primera capa, que es el **SecurityFilterChain**, por el momento hicimos configuraciones sencillas que las seguiremos manteniendo. A futuro, estas configuraciones irán mudando, pero por el momento continuarán tal cual las dejamos.

**Paso 1: Configurar AuthenticationManager**

El segundo elemento a configurar es el AuthenticationManager. Como vimos anteriormente a nivel teórico, es el encargado de administrar todo el proceso de autenticación que tengamos en nuestra aplicación.

Para configurarlo, vamos a dirigirnos a nuestra clase **SecurityConfig** y debajo del método de la **SecurityFilterChain** crearemos uno nuevo para nuestro AuthenticationManager de la siguiente manera:

@Bean

public AuthenticationManager authenticationManager(AuthenticationConfiguration authenticationConfiguration) throws Exception {

        return authenticationConfiguration.getAuthenticationManager();

}

Veamos el código en detalle:

1. **@Bean**: Esta anotación se usa en Spring para indicar que un método produce un bean que debe ser administrado por el contenedor de Spring. En este caso, el método **authenticationManager** está siendo registrado como un bean en el contexto de la aplicación Spring.
2. **public AuthenticationManager authenticationManager(AuthenticationConfiguration authenticationConfiguration) throws Exception**:
   * Este método define un bean que es un **AuthenticationManager**.
   * Recibe un parámetro **AuthenticationConfiguration**, que es una clase proporcionada por Spring Security que contiene la configuración necesaria para gestionar la autenticación en la aplicación.
   * El método puede lanzar una excepción de tipo **Exception**.
3. **return authenticationConfiguration.getAuthenticationManager()**:
   * Este método llama a **getAuthenticationManager()** del objeto **authenticationConfiguration** y devuelve un **AuthenticationManager**.
   * **AuthenticationConfiguration** es una clase que facilita la configuración de **AuthenticationManager**. Al llamar a **getAuthenticationManager()**, se está delegando la responsabilidad de construir y configurar el **AuthenticationManager** a **AuthenticationConfiguration**, que ya tiene toda la información necesaria para hacerlo (por ejemplo, detalles sobre los proveedores de autenticación configurados).

⚠️ **¿Qué es un BEAN?:**un bean es un objeto que forma parte del contexto de la aplicación gestionado por el contenedor de Spring. En términos sencillos, es un componente que Spring Framework crea, configura y gestiona a lo largo del ciclo de vida de la aplicación.

**Paso 2: Configurar AuthenticationProvider**

Una vez configurado nuestro **AuthenticationManager**, llegó el momento de configurar nuestro **AuthenticationProvider**. Recordá que una aplicación que implementa Spring Security puede trabajar con diferentes proveedores de autenticación; sin embargo, para este ejemplo veremos la más común, que es mediante usuarios, contraseñas, roles y permisos provenientes de una base de datos.

Para implementarlo a nivel código, debemos llevar a cabo el siguiente código debajo del método de nuestro AuthenticationManager:

@Bean

    public AuthenticationProvider authenticationProvider(){

        DaoAuthenticationProvider provider = new DaoAuthenticationProvider();

        provider.setPasswordEncoder(null);

        provider.setUserDetailsService(null);

        return provider;

    }

1. **public AuthenticationProvider authenticationProvider(UserDetailServiceImpl userDetailService)**:
   * Este método es un factory method para crear un bean de tipo **AuthenticationProvider**.
2. **DaoAuthenticationProvider provider = new DaoAuthenticationProvider();**:
   * Se crea una instancia de **DaoAuthenticationProvider**. Este es un **AuthenticationProvider** que utiliza un **UserDetailsService** y un **PasswordEncoder** para autenticar usuarios.
3. **provider.setPasswordEncoder(passwordEncoder());**:
   * Aquí se configura el **PasswordEncoder** para el **DaoAuthenticationProvider**. Por defecto, cuando recién arranquemos vamos a ponerlo como null.
4. **provider.setUserDetailsService(userDetailService);**:
   * Se configura el **UserDetailsService** en el **DaoAuthenticationProvider**. Como todavía no tenemos configurado nuestro UserDetailsService, vamos a dejarlo por el momento como null.
5. **return provider;**:
   * El método devuelve el **DaoAuthenticationProvider** configurado, que luego es gestionado por el contenedor de Spring como un bean.

**Paso 3: Configurar PasswordEncoder**

En nuestro Authentication Provider dejamos como null el Password Encoder, por lo que llegó el momento de configurarlo. Para ello, vamos a llevar a cabo el siguiente código debajo de nuestro AuthenticationProvider:

@Bean

    public PasswordEncoder passwordEncoder(){

        //return new BCryptPasswordEncoder();

        //solo devuelve contraseña en texto plano, no usar en produccion, solo para probar!

        return NoOpPasswordEncoder.getInstance();

    }

Veamos el código en mayor detalle:

1. **public PasswordEncoder passwordEncoder()**:
   * Aquí se define un método público llamado **passwordEncoder** que retorna una instancia de **PasswordEncoder**.
   * **PasswordEncoder** es una interfaz en Spring Security que define un contrato para servicios que pueden codificar (encriptar) y verificar contraseñas.
2. **return NoOpPasswordEncoder.getInstance();**:
   * Aquí estamos creando y devolviendo una instancia de **NoOpPasswordEncoder**.
   * **NoOpPasswordEncoder** es una implementación de PasswordEncoder que no realiza ninguna codificación/encriptado en las contraseñas. En otras palabras, las contraseñas se almacenan y se comparan en texto plano).
   * **NoOpPasswordEncoder.getInstance()** devuelve una instancia de esta clase. Es un método estático que sigue el patrón singleton, asegurando que siempre se use la misma instancia.

⚠️ **ADVERTENCIA:**NoOpPasswordEncoder no se recomienda utilizar NUNCA en producción dado que las contraseñas no se encriptarían. En esta ocasión lo utilizaremos para fines meramente didácticos..

**Paso 4: Configurando UserDetailsService**

Lo ideal según vimos en la arquitectura básica que planteamos en clases anteriores, es que el **UserDetailsService** se encargue de traer los usuarios y roles desde la base de datos y los convierta en un objeto **UserDetails**. Sin embargo, primero vamos a hacer una serie de pruebas con usuarios almacenados en memoria de forma lógica, antes de probar hacerlos con una base de datos como tal. Para esto vamos a llevar el siguiente código dentro de nuestras configuraciones:

//simulo roles y permisos en forma logica para probar.

    //sino deberia obtenerlos de BD o consumir de alguna API

    @Bean

    public UserDetailsService userDetailsService () {

        List<UserDetails> userDetailsList = new ArrayList<>();

        userDetailsList.add(User.withUsername("centro8")

                .password("1234") // esto si no está codificado, sino, tiene que seguir el algoritmo de codificación

                .roles("ADMIN")

                .authorities("CREATE", "READ", "UPDATE", "DELETE")

                .build());

        userDetailsList.add(User.withUsername("seguidor")

                .password("1234") // esto si no está codificado, sino, tiene que seguir el algoritmo de codificación

                .roles("USER")

                .authorities("READ")

                .build());

        userDetailsList.add(User.withUsername("actualizador")

                .password("1234") // esto si no está codificado, sino, tiene que seguir el algoritmo de codificación

                .roles("USER")

                .authorities("UPDATE")

                .build());

        return new InMemoryUserDetailsManager(userDetailsList);

    }

Una vez configurados nuestros **UserDetailsService** y nuestro **PasswordEncoder**, procedemos a setear a ambos en el **Authentication Provider** que habíamos dejado como null.

@Bean

        public AuthenticationProvider authenticationProvider(){

            DaoAuthenticationProvider provider = new DaoAuthenticationProvider();

            provider.setPasswordEncoder(passwordEncoder());

            provider.setUserDetailsService(userDetailsService());

            return provider;

        }

**Paso 5: Actualizando nuestra Security Filter Chain**

Una vez llevadas a cabo todas estas configuraciones, tenemos que especificarlas en nuestro filtro de seguridad, que es quien se encarga de justamente manejar las operaciones de autenticación.

*Para ello, vamos a hacer las siguientes modificaciones al código:*

@Bean

    public SecurityFilterChain securityFilterChain(HttpSecurity httpSecurity) throws Exception {

        return httpSecurity

                .csrf(csrf -> csrf.disable())

                .httpBasic(Customizer.withDefaults())

.sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS))

                .authorizeHttpRequests(http -> {

            // Endpoints públicos

            http.requestMatchers(HttpMethod.GET, "/holanoseg").permitAll();

            http.requestMatchers(HttpMethod.GET, "/holaseg").hasAuthority("READ");

            http.anyRequest().denyAll();

        })

            .build();

    }

Probando en Postman el endpoint de holanoseg, veremos que accede sin problemas ya que tiene todos permitido.

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

El contenido generado por IA puede ser incorrecto.

Ahora, si probamos ejecutar nuestra aplicación con estas nuevas configuraciones, vamos a ver que tanto **centro8** como **seguidor** van a poder acceder al endpoint segurizado; sin embargo, **actualizador** no podría, porque no posee el permiso de lectura (que nombramos como READ).

Probamos con usuario centro8, accede al endpoint sin problema alguno ya que tiene permisos de READ, UPDATE, DELETE

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

El contenido generado por IA puede ser incorrecto.

Probamos con usuario seguidor, accede al endpoint sin problema alguno ya que tiene permisos de READ.

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

El contenido generado por IA puede ser incorrecto.

Probamos con usuario actualizador, no puede acceder, recibe status code 403 Forbidden

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

El contenido generado por IA puede ser incorrecto.