**TP Nro. 5 Roles en Bases de Datos**

Desarrollar una Aplicación Web en Java con Spring que permita crear: usuarios (centro8 - seguidor), roles (admin - user) y permisos (read - create). Las entidades usuarios, roles y permisos deben persistirse en una base de datos gestionadas con MySQL.

**Paso 1: Dependencias**

Como vamos a empezar a trabajar con bases de datos es necesario que agreguemos las dependencias correspondientes de MySQL (SGBD con el que venimos trabajando) para poder manipular de igual manera a JPA como nuestro ORM.

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

El contenido generado por IA puede ser incorrecto.

[https://start.spring.io/#!type=gradle-project&language=java&platformVersion=3.4.3&packaging=jar&jvmVersion=21&groupId=ar.edu.centro8.ps&artifactId=rolesbd&name=rolesbd&description=Proyecto%20Spring%20Security%20Roles%20BD&packageName=ar.edu.centro8.ps.rolesbd&dependencies=devtools,lombok,web,security,mysql,data-jpa](https://start.spring.io/#!type=gradle-project&language=java&platformVersion=3.4.3&packaging=jar&jvmVersion=21&groupId=ar.edu.centro8.ps&artifactId=rolesbd&name=rolesbd&description=Proyecto%20Spring%20Security%20Roles%20BD&packageName=ar.edu.centro8.ps.rolesbd&dependencies=devtool)

**Paso 2: Paquete model**

Creamos en nuestro proyecto el paquete model (si no lo hicimos aún) y luego dentro crearemos 3 clases:

1. UserSec para representar a los usuarios (colocamos este nombre para que no se confunda con otras clases llamadas User de igual manera de alguna librería/biblioteca que podamos usar).
2. Role para representar a los roles
3. Permission para representar a los permisos.

Vamos a asignarle una serie de atributos a cada clase y a crear las relaciones entre sí que serán dos **ManyToMany** unidireccionales.

**Clase UserSec**

Definimos la clase UserSec con sus correspondientes atributos y la relacionamos con una clase Role mediante una relación **@ManyToMany**.

package ar.edu.centro8.ps.rolesbd.model;

import jakarta.persistence.\*;

import lombok.AllArgsConstructor;

import lombok.Getter;

import lombok.NoArgsConstructor;

import lombok.Setter;

import java.util.HashSet;

import java.util.Set;

@Entity

@Table(name="users")

@Getter @Setter

@AllArgsConstructor

@NoArgsConstructor

public class UserSec {

    @Id

    @GeneratedValue(strategy= GenerationType.IDENTITY)

    private Long id;

    @Column(unique = true)

    private String username;

private String password;

    private boolean enabled;

    private boolean accountNotExpired;

    private boolean accountNotLocked;

    private boolean credentialNotExpired;

    //Usamos Set porque no permite repetidos

    //List permite repetidos

    @ManyToMany(fetch = FetchType.EAGER, cascade = CascadeType.ALL) //el eager me va  a cargar todos los roles

    @JoinTable (name = "user\_roles", joinColumns = @JoinColumn(name = "user\_id"),

    inverseJoinColumns=@JoinColumn(name = "role\_id"))

    private Set<Role>rolesList = newHashSet<>();

}

**RECORDÁ QUE EN ESTE CÓDIGO UTILIZAMOS:**

* **@Column(unique = true):** Establece que la columna username debe ser única en la base de datos.
* **@ManyToMany(fetch = FetchType.EAGER, cascade = CascadeType.ALL):** Define una relación muchos a muchos con la entidad Role. La estrategia de carga EAGER indica que los roles se cargarán inmediatamente con el usuario. CascadeType.ALL significa que todas las operaciones de persistencia (persistir, eliminar, etc.) se realizarán en cascada a los roles asociados.
* **@JoinTable:** Especifica la tabla de unión para la relación muchos a muchos.
  + **name = «user\_roles»:** Nombre de la tabla de unión.
  + **joinColumns = @JoinColumn(name = «user\_id»):** Especifica la columna de unión correspondiente a UserSec en la tabla de unión.
  + **inverseJoinColumns = @JoinColumn(name = «role\_id»):** Especifica la columna de unión correspondiente a Role en la tabla de unión.
* **private Set<Role>rolesList = new HashSet<>();:** Define la colección de roles asociados a este usuario, inicializada como un HashSet para evitar duplicados.

**Clase Role**

Definimos la clase Role con sus correspondientes atributos y la relacionamos con una clase Permission mediante una relación **@ManyToMany**.

package ar.edu.centro8.ps.rolesbd.model;

import jakarta.persistence.\*;

import lombok.AllArgsConstructor;

import lombok.Getter;

import lombok.NoArgsConstructor;

import lombok.Setter;

import java.util.HashSet;

import java.util.Set;

@Entity

@Getter

@Setter

@AllArgsConstructor

@NoArgsConstructor

@Table(name="roles")

public class Role {

    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    private Long id;

    private String role;

    //Usamos Set porque no permite repetidos

    //List permite repetidos

    @ManyToMany(fetch = FetchType.EAGER, cascade = CascadeType.ALL)

    @JoinTable (name = "roles\_permissions", joinColumns = @JoinColumn(name = "role\_id"),

            inverseJoinColumns=@JoinColumn(name = "permission\_id"))

    privateSet<Permission>permissionsList = newHashSet<>();

}

**Clase Permission**

Definimos la clase Permission con sus correspondientes atributos.

package ar.edu.centro8.ps.rolesbd.model;

import jakarta.persistence.\*;

import lombok.AllArgsConstructor;

import lombok.Getter;

import lombok.NoArgsConstructor;

import lombok.Setter;

@Entity

@Getter

@Setter

@AllArgsConstructor

@NoArgsConstructor

@Table(name="permissions")

public class Permission {

    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

   private Long id;

    @Column(unique = true, nullable = false)

    private String permissionName;

}

**RECORDÁ QUE EN ESTE CÓDIGO UTILIZAMOS:**

* **@Column(unique = true , nullable = false):** Establece que la columna username debe ser única en la base de datos y que no puede ser nula.

**Paso 3: Configuración de BD**

Una vez creadas las 3 clases con sus respectivos atributos y relaciones, crearemos una base de datos en el gestor que estemos utilizando y le pondremos de nombre security (el nombre puede variar). Luego, procederemos a establecer las configuraciones de nuestra base de datos en nuestro archivo **application.properties**de la siguiente manera:

spring.application.name=rolesbd

# Configuraciones de BD

spring.jpa.hibernate.ddl-auto=update

spring.datasource.url=jdbc:mysql://localhost:3306/security?useSSL=false&serverTimezone=UTC

spring.datasource.username=root

spring.datasource.password=1234

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

# Configuraciones de seguridad

spring.security.user.name=${SS\_USER}

spring.security.user.password=${SS\_PASSWORD}

Con todo esto configurado estamos listos para ejecutar nuestro proyecto para que se cree la estructura en nuestra BD a partir de los mapeos que hayamos hecho. Si todo sale bien deberemos obtener algo similar a esto:

Texto

El contenido generado por IA puede ser incorrecto.

Y a nivel Diagrama Entidad Relación podríamos representarlo de la siguiente manera:

Diagrama

El contenido generado por IA puede ser incorrecto.

**Paso 4: Estructura del Proyecto**

Como ya tenemos nuestra capa model, llegó el momento de construir nuestras capas **Service** y **Repository** para cada una de nuestras **entidades**.

**Services – Permission**

package ar.edu.centro8.ps.rolesbd.service;

import ar.edu.centro8.ps.rolesbd.model.Permission;

import java.util.List;

import java.util.Optional;

public interface IPermissionService {

    List<Permission>findAll();

    Optional<Permission>findById(Long id);

    Permission save(Permission permission);

    Void deleteById(Long id);

    Permission update(Permission permission);

}

**Nuevo Concepto – OPTIONALS**

Los **Optionals** en **Java** se introdujeron en la versión 8 como una forma segura y elegante de manejar valores nulos. Antes de Optionals, era común usar el valor null para indicar la ausencia de un valor, lo que podía provocar **NullPointerExceptions** si no se manejaba adecuadamente.

Un **Optional** es un contenedor que puede contener un valor o no. Se puede pensar en él como una caja que puede estar vacía o contener un objeto. Si la caja está vacía, se dice que el **Optional** está «vacío». Si la caja contiene un objeto, se dice que el Optional está «presente» y el objeto se puede obtener utilizando el método get().

package ar.edu.centro8.ps.rolesbd.service;

import ar.edu.centro8.ps.rolesbd.model.Permission;

import ar.edu.centro8.ps.rolesbd.repository.IPermissionRepository;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import java.util.List;

import java.util.Optional;

@Service

public class PermissionService implements IPermissionService {

    @Autowired

    private IPermissionRepository permissionRepository;

    @Override

    public List<Permission>findAll() {

        return permissionRepository.findAll();

    }

    @Override

    public Optional<Permission>findById(Long id) {

        Return permissionRepository.findById(id);

    }

    @Override

    public Permission save(Permission permission) {

        Return permissionRepository.save(permission);

    }

    @Override

    public void deleteById(Long id) {

        permissionRepository.deleteById(id);

    }

    @Override

    public Permission update(Permission permission) {

        return permissionRepository.save(permission);

    }

}

**Services – Roles**

package ar.edu.centro8.ps.rolesbd.service;

import ar.edu.centro8.ps.rolesbd.model.Role;

import java.util.List;

import java.util.Optional;

public interface IRoleService {

    List<Role>findAll();

    Optional<Role>findById(Long id);

    Role save(Role role);

    Void deleteById(Long id);

    Role update(Role role);

}

package ar.edu.centro8.ps.rolesbd.service;

import ar.edu.centro8.ps.rolesbd.model.Role;

import ar.edu.centro8.ps.rolesbd.repository.IRoleRepository;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import java.util.List;

import java.util.Optional;

@Service

public class RoleService implements IRoleService {

    @Autowired

    private IRoleRepository roleRepository;

    @Override

    public List<Role>findAll() {

        return roleRepository.findAll();

    }

    @Override

    Public Optional<Role>findById(Long id) {

        Return roleRepository.findById(id);

    }

    @Override

    public Role save(Role role) {

        return roleRepository.save(role);

    }

    @Override

    public void deleteById(Long id) {

        roleRepository.deleteById(id);

    }

    @Override

    publicRole update(Role role) {

        return roleRepository.save(role);

    }

}

**Services – Users**

package ar.edu.centro8.ps.rolesbd.service;

import ar.edu.centro8.ps.rolesbd.model.UserSec;

import java.util.List;

import java.util.Optional;

public interface IUserService {

    public List<UserSec>findAll();

    public Optional<UserSec>findById(Long id);

    public UserSec save(UserSecuserSec);

    public void deleteById(Long id);

    public void update(UserSecuserSec);

}

package ar.edu.centro8.ps.rolesbd.service;

import ar.edu.centro8.ps.rolesbd.model.UserSec;

import ar.edu.centro8.ps.rolesbd.repository.IUserRepository;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import java.util.List;

import java.util.Optional;

@Service

public class UserServiceimplementsIUserService {

    @Autowired

    private IUserRepository userRepository;

    @Override

    public List<UserSec>findAll() {

        return userRepository.findAll();

    }

    @Override

    public Optional<UserSec>findById(Long id) {

        return userRepository.findById(id);

    }

    @Override

    public UserSec save(UserSec userSec) {

        return userRepository.save(userSec);

    }

    @Override

    public void deleteById(Long id) {

        userRepository.deleteById(id);

    }

    @Override

    public void update(UserSec userSec) {

        save(userSec);

    }

}

**Repositories**

package ar.edu.centro8.ps.rolesbd.repository;

import ar.edu.centro8.ps.rolesbd.model.Permission;

import org.springframework.data.jpa.repository.JpaRepository;

import org.springframework.stereotype.Repository;

@Repository

public interface IPermissionRepository extends JpaRepository<Permission, Long> {

}

package ar.edu.centro8.ps.rolesbd.repository;

import ar.edu.centro8.ps.rolesbd.model.Role;

import org.springframework.data.jpa.repository.JpaRepository;

import org.springframework.stereotype.Repository;

@Repository

public interface IRoleRepository extends JpaRepository<Role, Long> {

}

package ar.edu.centro8.ps.rolesbd.repository;

import ar.edu.centro8.ps.rolesbd.model.UserSec;

import org.springframework.data.jpa.repository.JpaRepository;

import org.springframework.stereotype.Repository;

import java.util.Optional;

@Repository

public interface IUserRepository extends JpaRepository<UserSec, Long> {

    //Crea la sentencia en base al nombre en inglés del método

    //Tmb se puede hacer mediante Query pero en este caso no es necesario

    Optional<UserSec>findUserEntityByUsername(String username);

}

**Nota:**en el método **findUserEntityByUsername(Stringusername)**de **IUserRepository** se define una consulta personalizada para buscar un usuario por su nombre de usuario. La anotación **@Query**no se utiliza en este caso, ya que **Spring Data JPA** puede generar automáticamente la consulta a partir del nombre del método.

El método devuelve un **Optional<UserSec>**, lo que significa que puede devolver un objeto **UserSec** si se encuentra un usuario con el nombre de usuario especificado, o un **Optional.empty()** si no se encuentra ningún usuario.

**Paso 5: End-points para crear usuarios**

Para poder manejar los CRUDS de nuestros usuarios, sus permisos y sus roles, vamos a proceder a llevar a cabo sus correspondientes Controllers.

⚠️**IMPORTANTE:** Recordá que el orden de alta (para poder asignar correctamente teniendo en cuenta las relaciones) debe ser:

1- Permisos  
2- Roles a los cuales se le asignan permisos que ya fueron creados  
3- Usuarios a los cuales se les asignan roles que ya fueron creados

**PermissionController**

package ar.edu.centro8.ps.rolesbd.controller;

import ar.edu.centro8.ps.rolesbd.model.Permission;

import ar.edu.centro8.ps.rolesbd.service.IPermissionService;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.\*;

import java.util.List;

import java.util.Optional;

@RestController

@RequestMapping("/api/permissions")

public class PermissionController {

    @Autowired

    Prívate IPermissionService permissionService;

    @GetMapping

    public ResponseEntity<List<Permission>>getAllPermissions() {

        List<Permission>permissions = permissionService.findAll();

        Return ResponseEntity.ok(permissions);

    }

    @GetMapping("/{id}")

    public ResponseEntity<Permission>getPermissionById(@PathVariableLong id) {

        Optional<Permission>permission = permissionService.findById(id);

        return permission.map(ResponseEntity::ok).orElseGet(() ->ResponseEntity.notFound().build());

    }

    @PostMapping

    public ResponseEntity<Permission>createPermission(@RequestBodyPermission permission) {

        Permission newPermission = permissionService.save(permission);

        return ResponseEntity.ok(newPermission);

    }

}

**EN ESTE CÓDIGO UTILIZAMOS:**

* **permission:** Esta variable representa el Optional<Permission> que se obtiene del resultado de la búsqueda por ID en el método getPermissionById.
* **Referencia a Métodos mediante**.map(ResponseEntity:**:**ok)**:**
  + El método map() de Optional se utiliza para aplicar una transformación al valor presente en el Optional, si lo hay. En este caso, la transformación se define mediante una referencia a método: ResponseEntity::ok.
  + ResponseEntity::ok es una referencia a un método estático de la clase ResponseEntity. Este método crea un nuevo objeto ResponseEntity con el código de estado HTTP 200 (OK) y el objeto Permission como cuerpo de la respuesta.
  + Al aplicar esta transformación, se crea un nuevo Optional<ResponseEntity<Permission>>.
* **Operaciones Lambda mediante .orElseGet(() ->ResponseEntity.notFound().build()):**
  + El método orElseGet() de Optional se utiliza para obtener un valor predeterminado si el Optional está vacío. En este caso, el valor predeterminado se define mediante una expresión lambda: () ->ResponseEntity.notFound().build().
  + La expresión lambda crea un nuevo objeto ResponseEntity con el código de estado HTTP 404 (NotFound).
  + Si el Optional<Permission> original está vacío, se devuelve este ResponseEntity con código de estado 404.

**Role Controller**

package ar.edu.centro8.ps.rolesbd.controller;

import ar.edu.centro8.ps.rolesbd.model.Role;

import ar.edu.centro8.ps.rolesbd.model.Permission;

import ar.edu.centro8.ps.rolesbd.service.IPermissionService;

import ar.edu.centro8.ps.rolesbd.service.IRoleService;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.\*;

import java.util.HashSet;

import java.util.List;

import java.util.Optional;

import java.util.Set;

@RestController

@RequestMapping("/api/roles")

public class RoleController {

    @Autowired

    private IRoleService roleService;

    @Autowired

    private IPermissionService permiService;

    @GetMapping

    public ResponseEntity<List<Role>>getAllRoles() {

        List<Role> roles = roleService.findAll();

        return ResponseEntity.ok(roles);

    }

    @GetMapping("/{id}")

    public ResponseEntity<Role>getRoleById(@PathVariableLong id) {

        Optional<Role> role = roleService.findById(id);

        return role.map(ResponseEntity::ok).orElseGet(() ->ResponseEntity.notFound().build());

    }

    @PostMapping

    public ResponseEntity<Role>createRole(@RequestBodyRole role) {

        Set<Permission>permiList = newHashSet<Permission>();

        Permission readPermission;

        // Recuperar la Permission/s por su ID

        for (Permission per :role.getPermissionsList()) {

            readPermission = permiService.findById(per.getId()).orElse(null);

            if (readPermission != null) {

                //si encuentro, guardo en la lista

                permiList.add(readPermission);

            }

        }

        role.setPermissionsList(permiList);

        Role newRole = roleService.save(role);

        return ResponseEntity.ok(newRole);

    }

}

**User Controller**

package ar.edu.centro8.ps.rolesbd.controller;

import ar.edu.centro8.ps.rolesbd.model.Role;

import ar.edu.centro8.ps.rolesbd.model.UserSec;

import ar.edu.centro8.ps.rolesbd.service.IRoleService;

import ar.edu.centro8.ps.rolesbd.service.IUserService;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.http.ResponseEntity;

import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.\*;

import java.util.HashSet;

import java.util.List;

import java.util.Optional;

import java.util.Set;

@RestController

@RequestMapping("/api/users")

public class UserController {

    @Autowired

    private IUserService userService;

    @Autowired

    private IRoleService roleService;

    @GetMapping

    public ResponseEntity<List<UserSec>>getAllUsers() {

        List<UserSec>users = userService.findAll();

        Return ResponseEntity.ok(users);

    }

    @GetMapping("/{id}")

    public ResponseEntity<UserSec>getUserById(@PathVariableLong id) {

        Optional<UserSec> user = userService.findById(id);

        Return user.map(ResponseEntity::ok).orElseGet(() ->ResponseEntity.notFound().build());

    }

    @PostMapping

    public ResponseEntity<UserSec>createUser(@RequestBodyUserSec userSec) {

        Set<Role>roleList = newHashSet<Role>();

        Role readRole;

        // Recuperar la Permission/s por su ID

        for (Role role:userSec.getRolesList()){

            readRole = roleService.findById(role.getId()).orElse(null);

            if (readRole != null) {

                //si encuentro, guardo en la lista

                roleList.add(readRole);

            }

        }

        if (!roleList.isEmpty()) {

            userSec.setRolesList(roleList);

            UserSec newUser = userService.save(userSec);

            return ResponseEntity.ok(newUser);

        }

        return null;

    }

}

⚠️**Nota:** Por ahora solo tendremos end-points de lectura (GET) y de creación (POST) y los dejaremos sin permisos

**Paso 6: UserDetailsService**

Cuando implementamos UserDetailsService por primera vez, lo hicimos en el archivo de configuración de seguridad y de forma manual especificando cada detalle en un método. Ahora llegó el momento de establecerlo como un servicio como tal y que obtenga los datos de los usuarios y sus detalles de la correspondiente base de datos.

Para ello, vamos a crear dentro de nuestro paquete service, una nueva clase llamada **UserDetailsServiceImp**que va a implementar la clase **UserDetailsService**que ofrece el propio **Spring Security** y vamos a agregar el siguiente código:

package ar.edu.centro8.ps.rolesbd.service;

import ar.edu.centro8.ps.rolesbd.model.Permission;

import ar.edu.centro8.ps.rolesbd.model.Role;

import ar.edu.centro8.ps.rolesbd.model.UserSec;

import ar.edu.centro8.ps.rolesbd.repository.IUserRepository;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.security.core.authority.SimpleGrantedAuthority;

import org.springframework.security.core.userdetails.User;

import org.springframework.security.core.userdetails.UserDetails;

import org.springframework.security.core.userdetails.UserDetailsService;

import org.springframework.security.core.userdetails.UsernameNotFoundException;

importorg.springframework.stereotype.Service;

import java.util.ArrayList;

import java.util.List;

@Service

Public class UserDetailsServiceImp implements UserDetailsService {

    @Autowired

    Prívate IUserRepository userRepo;

    @Override

    Public UserDetails loadUserByUsername (Stringusername) throws UsernameNotFoundException {

        //tenemos User sec y necesitamos devolver UserDetails

        //traemos el usuario de la bd

        UserSec userSec = userRepo.findUserEntityByUsername(username)

                .orElseThrow(()->new UsernameNotFoundException("El usuario " + username + "no fue encontrado"));

        //con GrantedAuthority Spring Security maneja permisos

        List<SimpleGrantedAuthority>authorityList = newArrayList<>();

        //Programación funcional

        //tomamos roles y los convertimos en SimpleGrantedAuthority para poder agregarlos a la authorityList

        userSec.getRolesList()

                .forEach(role ->authorityList.add(new SimpleGrantedAuthority("ROLE\_".concat(role.getRole()))));

        //ahora tenemos que agregar los permisos

        userSec.getRolesList().stream()

                .flatMap(role ->role.getPermissionsList().stream()) //acá recorro los permisos de los roles

                .forEach(permission->authorityList.add(new SimpleGrantedAuthority(permission.getPermissionName())));

        //retornamos el usuario en formato Spring Security con los datos de nuestro userSec

        return new User(userSec.getUsername(),

                userSec.getPassword(),

                userSec.isEnabled(),

                userSec.isAccountNotExpired(),

                userSec.isCredentialNotExpired(),

                userSec.isAccountNotLocked(),

                authorityList);

    }

}

Este código implementa la interfaz **UserDetailsService** de Spring Security. Su objetivo es proporcionar la información de usuario necesaria para la autenticación.

* @Autowired privateIUserRepositoryuserRepo: Inyecta una dependencia del repositorio IUserRepository que se encarga de acceder a los usuarios en la base de datos.
* **Método loadUserByUsername(Stringusername):**
  + Este método es el método principal de la interfaz UserDetailsService. Se invoca durante el proceso de autenticación de Spring Security para obtener la información de un usuario por su nombre de usuario.
  + El método recibe un argumento username que representa el nombre de usuario que intenta iniciar sesión.
  + Devuelve un objeto UserDetails que contiene la información de usuario necesaria para la autenticación.

**Pasos del método:**

* + **Obtener el usuario de la base de datos:**
    - Se utiliza el repositorio userRepo para buscar un usuario por su nombre de usuario usando el método findUserEntityByUsername.
    - El resultado se almacena en un Optional<UserSec>.
    - Se utiliza el método orElseThrow() para obtener el usuario del Optional o lanzar una excepción UsernameNotFoundException si el usuario no se encuentra.
  + **Convertir roles a GrantedAuthority:**
    - Se crea una lista vacía authorityList para almacenar los permisos del usuario.
    - Se utiliza programación funcional con forEach para iterar sobre la lista de roles del usuario (userSec.getRolesList())
    - Dentro del ciclo, se crea un objeto SimpleGrantedAuthority para cada rol y se agrega a la authorityList.
    - SimpleGrantedAuthority es una clase de Spring Security que representa un permiso otorgado a un usuario.
  + **Convertir permisos a GrantedAuthority:**
    - Se utiliza nuevamente programación funcional con stream y flatMap para iterar sobre los permisos de todos los roles del usuario.
    - flatMap permite «aplanar» dos streams en uno solo. En este caso, se recorren primero los roles y luego los permisos dentro de cada rol.
    - Dentro del ciclo, se crea un objeto SimpleGrantedAuthority para cada permiso y se agrega a la authorityList.
  + **Crear objeto UserDetails:**
    - Se crea un nuevo objeto User de Spring Security utilizando la información del usuario obtenido de la base de datos (userSec) y la lista de permisos (authorityList).
    - User es la implementación predeterminada de UserDetails en Spring Security.
    - El constructor de User recibe los siguientes parámetros:
      * username: Nombre de usuario.
      * password: Contraseña del usuario (hasheada).
      * enabled: Indica si el usuario está habilitado.
      * accountNonExpired: Indica si la cuenta del usuario no ha expirado.
      * credentialsNonExpired: Indica si las credenciales del usuario no han expirado.
      * accountNonLocked: Indica si la cuenta del usuario no está bloqueada.
      * authorities: Lista de permisos del usuario representados por objetos GrantedAuthority.
  + **Devolver el objeto UserDetails:**
    - Se devuelve el objeto User creado, que contiene la información de usuario necesaria para la autenticación.

Este código implementa un servicio que recupera la información de un usuario de la base de datos en base a su nombre de usuario y la transforma en el formato UserDetails requerido por Spring Security para la autenticación. El uso de **programación funcional** (lambdas, streams, etc) permite un código más conciso y expresivo.

**Paso 7: Asociar el nuevo UserDetailsService en nuestra configuración de seguridad**

Para quitar nuestro **UserDetailsService** que manejaba valores lógicos de usuarios y utilizar el nuevo que utiliza la base de datos, vamos a proceder a eliminar (o comentar, en caso de que lo quieras dejar para tu aprendizaje) el método **userDetailsService()** que teníamos en **SecurityConfig** .

En lugar detener un método con valores lógicos, vamos a ir a nuestro método **authenticationProvider**, agregar una instancia a nuestro **UserDetailsService**como parámetro y vamos a setearlo mediante **provider.serUserDetailsService()** de la siguiente manera:

//creamos authenticationmanager

    @Bean

    public AuthenticationManager authenticationManager(AuthenticationConfiguration authenticationConfiguration) throwsException {

        return authenticationConfiguration.getAuthenticationManager();

    }

    //creamos authenticationprovider

    //Agregamos el user DetailsService como parámetro

    @Bean

    public AuthenticationProvider authenticationProvider(UserDetailsService userDetailsService){

        DaoAuthenticationProvider provider = new DaoAuthenticationProvider();

        provider.setPasswordEncoder(passwordEncoder());

        //sacamos el anterior, el lógico y agregamos el nuevo

        //  provider.setUserDetailsService(userDetailsService());

        provider.setUserDetailsService(userDetailsService);

        return provider;

    }

**Paso 8: Agregar usuarios de prueba a la BD**

Una vez todo configurado, vamos a ejecutar nuestra aplicación y, mediante Postman, vamos a agregar una serie de permisos, roles y usuarios. Recordá que es MUY IMPORTANTE que las altas las hagas en dicho orden (caso contrario, quedarán vacíos ciertos valores dadas las relaciones que existen entre tablas y luego tendrás que modificarlos o, caso contrario, tendrás errores).

**JSONs para ALTAS**

**Alta de permisos POST localhost:8080/api/permissions**

{

"permissionName": "READ"

}

{

"permissionName": "CREATE"

}

**Alta de roles POST localhost:8080/api/roles**

{

"role": "ADMIN",

"permissionsList": [

{

"id": 1 // ID del permiso READ

},

{

"id": 2 // ID del permiso UPDATE

}

]

}

{

"role": "USER",

"permissionsList": [

{

"id": 1 // ID del permiso READ

}

]

}

**Alta de roles POST localhost:8080/api/users**

{

"username": "centro8",

"password": "1234",

"enabled": **true**,

"accountNotExpired": **true**,

"accountNotLocked": **true**,

"credentialNotExpired": **true**,

"rolesList": [

{

"id": 1// ID del rol ADMIN

}

]

}

{

"username": "seguidor",

"password": "1234",

"enabled": **true**,

"accountNotExpired": **true**,

"accountNotLocked": **true**,

"credentialNotExpired": **true**,

"rolesList": [

{

"id": 2// ID del rol USER

}

]

}

⚠️**Nota:** Recordá reemplazar los valores de ids, etc en base a lo que vos tengas en tu base de datos y a los valores que hayan quedado a medida que fuiste ingresando cada registro. Verificá siempre esto en tu base de datos antes de ejecutar el JSON en Postman.

**Paso 9: Probar nuestro end-point securizado**

Una vez cargados nuestros permisos, roles y usuarios, llegó el momento de probar la autenticación. Recordemos que en nuestro HelloWorldController establecimos que para acceder a **/holanoseg**no es necesario ningún tipo de seguridad porque establecimos @PreAuthorize(«permitAll()») pero, que para **/holaseg**el usuario debe tener permiso de lectura, por lo que, en este caso, ambos usuarios que creamos deberían de tener permisos si se loguean. ¡Probemos con Postman!

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

El contenido generado por IA puede ser incorrecto.

Interfaz de usuario gráfica, Texto, Correo electrónico

El contenido generado por IA puede ser incorrecto.

Si todo sale bien, la autenticación va a ser exitosa, nuestra app con Spring Security va a reconocer que centro8 es un usuario de la bd y 1234 es su contraseña y que tiene un rol que le permite hacer uso de lectura mediante READ, por lo que nos va a permitir acceder sin problema.

Para cualquier otro usuario, no tendríamos autorización.

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

El contenido generado por IA puede ser incorrecto.

Hay un punto de seguridad importante y es que la contraseña no está codificada ni encriptada, por lo cual, en el próximo proyecto resolveremos este problema.