Paso 1: Crear el proyecto con Spring Initializr

1. Abre **IntelliJ IDEA**.
2. Ve a **File > New > Project**.
3. Selecciona **Spring Initializr**.
    - **Group**: com.ejemplo.security
    - **Artifact**: spring-security-demo
    - **Packaging**: Jar
    - **Java Version**: 17 o superior (dependiendo de la configuración de tu JDK).
4. En **Dependencies**, selecciona las siguientes:
    - **Spring Web** (para construir APIs RESTful)
    - **Spring Security** (para agregar seguridad)
    - **Spring Data JPA** (para trabajar con bases de datos usando JPA)
    - **MySQL Driver** (para conectar con MySQL)
5. Haz clic en **Finish**. IntelliJ generará el proyecto con la estructura básica.

Paso 2: Configurar `pom.xml`

Tu archivo `pom.xml` debe contener las dependencias para **Spring Security**, **Spring Data JPA**, y **MySQL**. IntelliJ ya habrá agregado las dependencias seleccionadas, pero asegúrate de que se vean así:

```
Unset
<dependencies>
    <!-- Spring Boot Starter Web -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <!-- Spring Boot Starter Security -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-security</artifactId>
    </dependency>

    <!-- Spring Data JPA -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>

    <!-- MySQL Connector -->
    <dependency>
        <groupId>mysql</groupId>
```

```
        <artifactId>mysql-connector-java</artifactId>
    </dependency>

    <!-- Lombok (Opcional, para evitar boilerplate en las clases) -->
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <scope>provided</scope>
    </dependency>

    <!-- Spring Boot Test -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Paso 3: Configurar la base de datos MySQL

1. Crea una base de datos en MySQL:

```
Unset
CREATE DATABASE spring_security_demo;
```

1. Configura la conexión a la base de datos en el archivo `application.properties` o `application.yml` dentro de `src/main/resources`:

`application.properties`:

```
Unset
spring.datasource.url=jdbc:mysql://localhost:3306/spring_security_demo
spring.datasource.username=root
spring.datasource.password=tu_password
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
```

```
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

Paso 4: Crear las Entidades y Repositorios para Usuarios y Roles

Entidad `User`

Crea la clase `User` dentro del paquete `com.ejemplo.security.model`:

```java
package com.ejemplo.security.model;

import lombok.Data;
import javax.persistence.*;
import java.util.Set;

@Entity
@Data
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String username;
    private String password;

    @ManyToMany(fetch = FetchType.EAGER)
    @JoinTable(name = "user_roles",
            joinColumns = @JoinColumn(name = "user_id"),
            inverseJoinColumns = @JoinColumn(name = "role_id"))
    private Set<Role> roles;
}
```

Entidad `Role`

Crea la clase `Role` dentro del paquete `com.ejemplo.security.model`:

```java
package com.ejemplo.security.model;

import lombok.Data;
import javax.persistence.*;

@Entity
@Data
public class Role {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
}
```

Repositorio `UserRepository`

Crea la interfaz `UserRepository` en `com.ejemplo.security.repository`:

```java
package com.ejemplo.security.repository;

import com.ejemplo.security.model.User;
import org.springframework.data.jpa.repository.JpaRepository;

import java.util.Optional;

public interface UserRepository extends JpaRepository<User, Long> {
    Optional<User> findByUsername(String username);
}
```

Paso 5: Configurar Spring Security

Crea una clase de configuración llamada `SecurityConfig` en el paquete `com.ejemplo.security.config`:

Java

```java
package com.ejemplo.security.config;

import com.ejemplo.security.service.UserDetailsServiceImpl;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import
org.springframework.security.authentication.dao.DaoAuthenticationProvider;
import
org.springframework.security.config.annotation.authentication.builders.Authe
nticationManagerBuilder;
import
org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
org.springframework.security.config.annotation.web.configuration.EnableWebSe
curity;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.web.SecurityFilterChain;

@Configuration
@EnableWebSecurity
public class SecurityConfig {

    private final UserDetailsServiceImpl userDetailsService;

    public SecurityConfig(UserDetailsServiceImpl userDetailsService) {
        this.userDetailsService = userDetailsService;
    }

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
Exception {
        http
            .authorizeRequests()
                .antMatchers("/admin/**").hasRole("ADMIN")
                .antMatchers("/user/**").hasRole("USER")
                .anyRequest().authenticated()
            .and()
            .formLogin()
            .and()
            .httpBasic();

        return http.build();
    }

    @Bean
    public DaoAuthenticationProvider authenticationProvider() {
```

```
        DaoAuthenticationProvider authProvider = new
DaoAuthenticationProvider();
        authProvider.setUserDetailsService(userDetailsService);
        authProvider.setPasswordEncoder(passwordEncoder());
        return authProvider;
    }

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

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws
Exception {
        auth.authenticationProvider(authenticationProvider());
    }
}
```

Paso 6: Implementar `UserDetailsService`

Crea la clase `UserDetailsServiceImpl` en el paquete `com.ejemplo.security.service` para cargar usuarios desde la base de datos:

Java

```
package com.ejemplo.security.service;

import com.ejemplo.security.model.User;
import com.ejemplo.security.repository.UserRepository;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import
org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;

@Service
public class UserDetailsServiceImpl implements UserDetailsService {

    private final UserRepository userRepository;

    public UserDetailsServiceImpl(UserRepository userRepository) {
        this.userRepository = userRepository;
```

```
    }

    @Override
    public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
        User user = userRepository.findByUsername(username)
                .orElseThrow(() -> new UsernameNotFoundException("User not
found"));

        return org.springframework.security.core.userdetails.User
                .withUsername(user.getUsername())
                .password(user.getPassword())
                .roles(user.getRoles().stream().map(role ->
role.getName()).toArray(String[]::new))
                .build();
    }
}
```

Paso 7: Crear el Controlador y Rutas Protegidas

Crea un controlador básico para probar las rutas con roles en el paquete `com.ejemplo.security.controller`:

Java

```java
package com.ejemplo.security.controller;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class UserController {

    @GetMapping("/admin")
    public String admin() {
        return "Admin access";
    }

    @GetMapping("/user")
    public String user() {
        return "User access";
    }
}
```

Paso 8: Inicializar los datos en MySQL (Opcional)

Puedes crear un `CommandLineRunner` para insertar usuarios y roles en la base de datos al inicio:

```java
package com.ejemplo.security;

import com.ejemplo.security.model.Role;
import com.ejemplo.security.model.User;
import com.ejemplo.security.repository.RoleRepository;
import com.ejemplo.security.repository.UserRepository;
import org.springframework.boot.CommandLineRunner;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.stereotype.Component;

import java.util.Set;

@Component
public class DataInitializer implements CommandLineRunner {

    private final UserRepository userRepository;
    private final RoleRepository roleRepository;
    private final PasswordEncoder passwordEncoder;

    public DataInitializer(UserRepository userRepository, RoleRepository
roleRepository, PasswordEncoder passwordEncoder) {
        this.userRepository = userRepository;
        this.roleRepository = roleRepository;
        this.passwordEncoder = passwordEncoder;
    }

    @Override
    public void run(String... args) throws Exception {
        Role userRole = new Role();
        userRole.setName("USER");
        roleRepository.save(userRole);

        Role admin
```

Paso 1: Arrancar la Aplicación

Paso 2: Crear Usuarios y Roles en MySQL

Si no utilizaste el inicializador de datos en la aplicación (el `CommandLineRunner`), debes insertar manualmente los usuarios y roles en la base de datos.

Puedes usar una base de datos como **MySQL Workbench** o **DBeaver** para conectarte a la base de datos y ejecutar las siguientes consultas SQL:

Crear roles:

```
Java
INSERT INTO role (name) VALUES ('ROLE_USER');
INSERT INTO role (name) VALUES ('ROLE_ADMIN');
```

Crear usuarios con roles:

```
Java
INSERT INTO user (username, password) VALUES ('user',
'{bcrypt}$2a$10$uZSc5YyFbpYNoEum0Zd0GuAwoZk.gN/h2IF.A82deBkOb7fqEEmJy'); --
password: password
INSERT INTO user (username, password) VALUES ('admin',
'{bcrypt}$2a$10$uZSc5YyFbpYNoEum0Zd0GuAwoZk.gN/h2IF.A82deBkOb7fqEEmJy'); --
password: password

-- Asignar roles a los usuarios
INSERT INTO user_roles (user_id, role_id) VALUES (1, 1); -- Asignar USER a
user
INSERT INTO user_roles (user_id, role_id) VALUES (2, 2); -- Asignar ADMIN a
admin
```

Paso 3: Probar las Rutas Protegidas en Postman

1. **Abrir Postman**.
2. **Probar Ruta sin Autenticación**:
   - **URL**: `http://localhost:8080/admin`
   - **Método**: `GET`
3. Como esta ruta está protegida y requiere autenticación, recibirás un **error 401 Unauthorized** si intentas acceder sin credenciales.
4. **Autenticación con Credenciales (Admin)**:

    - En Postman, selecciona la pestaña **Authorization**.
    - En **Type**, selecciona **Basic Auth**.
    - Introduce las credenciales:
        - **Username**: `admin`
        - **Password**: `password`
    - Haz clic en **Send**.
5. Si la autenticación es correcta, deberías recibir la respuesta:

```
Java
"Admin access"
```

Autenticación con Credenciales (User):

- Cambia el **username** a `user` y la **password** a `password`.
- Prueba la ruta `/user`:
    - **URL**: `http://localhost:8080/user`
    - **Método**: `GET`
- Con las credenciales de usuario normal, recibirás la respuesta:

```
Java
"User access"
```

Acceso a Rutas Protegidas con Roles Incorrectos:

- Intenta acceder a la ruta `/admin` con las credenciales del usuario normal (`user`).
- Deberías recibir un **403 Forbidden**, ya que este usuario no tiene permisos de administrador.

Paso 4: Verificación de la Base de Datos

Puedes abrir **MySQL Workbench** o cualquier cliente de base de datos para verificar que los usuarios y roles estén correctamente insertados. Asegúrate de que las tablas `user`, `role`, y `user_roles` contengan los datos correctos.

Puedes ejecutar una consulta para listar los usuarios y roles asignados:

```
Java
SELECT u.username, r.name AS role
FROM user u
JOIN user_roles ur ON u.id = ur.user_id
JOIN role r ON ur.role_id = r.id;
```

Conclusión: Validaciones en Postman

- **Ruta `/admin`**: Solo accesible por usuarios con rol **ADMIN**.
- **Ruta `/user`**: Accesible por usuarios con rol **USER**.
- **Autenticación Básica**: Se debe usar **Basic Auth** en Postman para las pruebas de autenticación.

Con estos pasos, puedes comprobar que las configuraciones de **Spring Security** funcionan correctamente, y puedes validar el acceso mediante las rutas protegidas en Postman.

1. Ruta `/admin`

- **Propósito**: Solo accesible para los usuarios con rol **ADMIN**.

URL completa:
bash
Copiar código
`http://localhost:8080/admin`

-
- **Descripción**: Esta ruta está protegida por Spring Security, lo que significa que solo los usuarios con el rol **ADMIN** podrán acceder a ella. Si un usuario con otro rol intenta acceder, recibirá un **403 Forbidden**.

2. Ruta `/user`

- **Propósito**: Accesible para los usuarios con rol **USER**.

URL completa:
bash
Copiar código
`http://localhost:8080/user`

-

- **Descripción**: Esta ruta está protegida para que cualquier usuario autenticado con el rol **USER** pueda acceder. Si un usuario no autenticado o con otro rol intenta acceder, se le negará el acceso.

Otras posibles rutas que puedes configurar:

Si quisieras proteger rutas adicionales, aquí tienes algunos ejemplos:

- **Ruta pública `/public`** (sin autenticación):