

Índice

Índice

Introducción

Spring Data

- Preparando el entorno

- Creando el primer proyecto con Spring Data

Persistencia con JPA

Repositorios

- Usando el repositorio

- Obtener datos

- Actualizar datos

- Borrar datos

- Personalizar el repositorio

Repositorios REST

- Personalizar la url base

- Acepta peticiones de otros verbos

- Búsquedas personalizadas

CRUD Completo

- Vista previa

Anexo: Persistencia con JPA en base de datos relacionales

- Proyecto inicial

- Esquema Entidad-Relación

- Obteniendo información de otras tablas

 - @ManyToOne - de muchos a uno

 - @OneToMany - de uno a muchos

Introducción

Trabajar con una base de datos usando la forma estándar de JDBC en un proyecto web con Spring, se puede convertir en una ardua tarea, teniendo que escribir mucho código innecesario para tratar múltiples excepciones, abrir y cerrar conexiones, etc. Pero Spring Framework se encarga de realizar todos esos detalles a bajo nivel, tales como preparar y ejecutar instrucciones SQL, procesar las excepciones, manejar las transacciones y finalmente cerrar la conexión.

Spring proporciona muchas formas diferentes de realizar la interfaz con la base de datos. Con Spring-JDBC, se nos ofrece una manera mucho más fácil de acceder a una base de datos y realizar consultas de selección y manipulación de datos, todo a través de la clase `JdbcTemplate`. Pero con los repositorios de JPA se reduce todo a un par de líneas. Pronto lo veremos.

Spring Data

Spring Data es un proyecto de SpringSource cuyo propósito es unificar y facilitar el acceso a distintos tipos de tecnologías de persistencia, tanto a bases de datos relacionales como a las del tipo NoSQL.

Spring ya proporcionaba soporte para JDBC, Hibernate, JPA, JDO o Mybatis, simplificando la implementación de la capa de acceso a datos, unificando la configuración y creando una jerarquía de excepciones común para todas ellas.

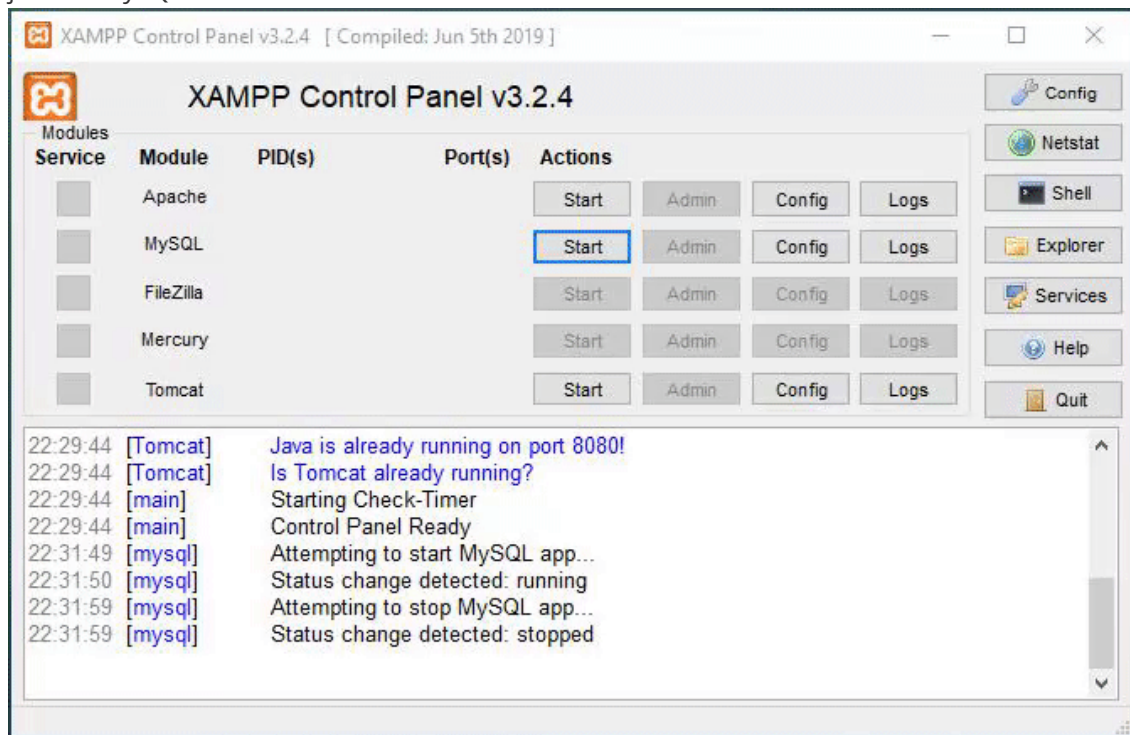
Y ahora, Spring Data viene a cubrir el soporte necesario para distintas tecnologías de bases de datos NoSQL y, además, integra las tecnologías de acceso a datos tradicionales, simplificando el trabajo a la hora de crear las implementaciones concretas.

Veremos cómo integrar la persistencia JPA en un proyecto Spring Boot para simplificar más todavía el acceso a datos a través de los nuevos repositorios.

Preparando el entorno

Antes de crear el proyecto con Spring Data, tendremos que tener preparada una base de datos en MySQL. Tan sólo deberemos abrir el servidor MySQL y crear una base de datos que será la que luego conectemos desde el proyecto Spring Boot.

1. Deberemos instalar un servidor MySQL. Podemos instalar [XAMPP](#) que en su página tiene versiones para Windows, Mac y Linux. Esto incluirá servidores Apache Tomcat, MariaDB (una versión OpenSource de MySQL, propiedad de Oracle), PHP y Perl. Sólo necesitaremos MySQL y Apache en el caso de que queremos usar PHPMyAdmin. En el tutorial usaremos XAMPP, pero se puede realizar todo con cualquier servidor MySQL. Si prefieres la versión gratis de MySQL Community de Oracle®, la podrás [encontrar aquí](#).
2. Una vez instalado, deberemos arrancar el servidor MySQL. En XAMPP es pulsar en **Start** junto a MySQL.



3. Si pulsamos en **Shell** se abrirá una ventana con una línea de comandos donde escribiremos lo siguiente para entrar (con la configuración predeterminada que viene en XAMPP). **-h** sirve para indicarle el host. **-u** el usuario y **-p** para que nos pida la contraseña. La contraseña está vacía por defecto, así que pulsaremos intro cuando nos pregunte.

```
1 | mysql -h localhost -u root -p
```

Nota: Si tenemos el path configurado correctamente, funcionará desde cualquier ubicación que estemos en la línea de comandos. Si no tendremos que entrar en la ubicación del archivo mysql.exe, que suele estar en `~/xampp/mysql/bin/mysql.exe`.

4. Una vez dentro del servidor, entraremos en OTRA consola de comandos, pero ahora la de MariaDB.

```

1 | Setting environment for using XAMPP for windows.
2 | expri@DESKTOP-ASGARD c:\users\expri\xampp
3 | # mysql -h localhost -u root -p
4 | Enter password:
5 | welcome to the MariaDB monitor.  Commands end with ; or \g.
6 | Your MariaDB connection id is 19
7 | Server version: 10.4.13-MariaDB mariadb.org binary distribution
8 |
9 | Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.
10 |
11 | Type 'help;' or '\h' for help. Type '\c' to clear the current input
    | statement.
12 |
13 | MariaDB [(none)]>

```

5. Ya que estamos dentro podemos crear una base de datos que llamaremos `dbejemplo`. Para ello escribiremos el siguiente comando en la consola de MariaDB:

```

1 | CREATE DATABASE dbejemplo;

```

6. Para comprobar que está creada correctamente, ejecutaremos el siguiente comando y debería tener la siguiente salida (aproximadamente):

```

1 | SHOW DATABASES;

```

```



1 | +-----+
2 | | Database      |
3 | +-----+
4 | | dbejemplo     |
5 | | information_schema |
6 | | mysql         |
7 | | performance_schema |
8 | | phpmyadmin    |
9 | | test          |
10 | +-----+
11 | 7 rows in set (0.001 sec)
12 | MariaDB [(none)]>

```

7. Pues ya estaría. Para salir escribimos el comando `exit` y ya podemos cerrar la ventana de la consola de comandos.

Creando el primer proyecto con Spring Data

1. Crearemos un proyecto como siempre, al cual le añadimos las siguientes dependencias:

-  Spring Data JPA
-  MySQL Driver
- Spring Web
- Thymeleaf
- Spring Boot DevTools

2. Creamos una vista principal llamada `index.html` en `/src/main/resources/templates`

```

1 <!DOCTYPE html>
2 <html lang="es" xmlns:th="http://www.thymeleaf.org">
3 <head>
4 <meta charset="utf-8">
5 <title>Spring Data</title>
6
7 <!-- Bootstrap 5 & Icons -->
8 <link
9
10 href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.mi
11 n.css"
12 rel="stylesheet"
13 data-integrity="sha384-
14 1BmE4kWBq78iYhF1dvKuhfTAU6auU8tT94WrHftjDbrCEXSU1oBoqyl2QvZ6jIW3"
15 data-crossorigin="anonymous" />
16 <link
17 href="https://cdn.jsdelivr.net/npm/bootstrap-
18 icons@1.7.2/font/bootstrap-icons.css"
19 rel="stylesheet" />
20 </head>
21 <body style="background-color: #ECECEC;">
22
23 <header class="bg-dark text-light text-center p-4">
24 <h1>Spring Data</h1>
25 <h3>Acceso a una base de datos MySQL usando Spring Boot</h3>
26 </header>
27
28 <div class="container">
29 <div class="row mt-5">
30 <div class="col"></div>
31 <div class="col-6">
32 <div class="card shadow">
33 <h3 class="card-title text-center">
34 <i class="bi bi-hdd"></i>
35 </h3>
36 <div class="card-body">
37
38 </div>
39 </div>
40 </div>
41 </div>
42 </div>
43 </body>
44 </html>

```

3. Creamos un controlador `controllers/HomeController.java` que nos lleve a la vista anterior.

```

1 @Controller
2 public class HomeController {
3     @GetMapping("/")
4     public String index() {
5         return "index";
6     }
7 }

```

4. Ya tendríamos todo lo necesario para poder arrancar la aplicación. Arriba con ella. Pues comprobarás que da un error al arrancar.

```

1 *****
2 APPLICATION FAILED TO START
3 *****
4
5 Description:
6
7 Failed to configure a DataSource: 'url' attribute is not specified and no
  embedded datasource could be configured.
8
9 Reason: Failed to determine a suitable driver class

```

5. Eso es debido a que hemos añadido las dependencias de JPA y MySQL, pero no hemos enlazado nuestro proyecto con ninguna base de datos. Así que tendremos que tocar por primera vez el archivo `application.properties` para indicarle lo siguiente:

```

1 # Base de datos MySQL
2 spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
3 spring.datasource.url=jdbc:mysql://localhost:3306/dbejemplo
4 spring.datasource.username=root
5 spring.datasource.password=

```

- En el `driver`, pondremos siempre el que viene ahí. Si usamos otro tipo de base de datos, usaremos el driver que indique el fabricante.
- En la `url` deberemos indicarle la url de acceso a nuestra BD (dominio+servidor+puerto+base de datos). En nuestro caso, la base de datos es `dbejemplo`, que es la que teníamos preparada.
- El `username` por defecto es `root`. La contraseña por defecto está vacía, por eso no indicamos nada. Si configuramos otro usuario y/o contraseña en la BD aquí es donde la estableceremos.

Nota1: Es posible que el archivo `application.properties` por defecto no esté en UTF-8, y si usamos tildes o ñ, nos da problema Maven. Debemos pulsar botón derecho sobre el archivo -> Properties -> Text file encoding -> Other -> UTF-8.

Nota2: En cualquier caso, si Maven nos da un error en el `pom.xml` que no tenga mucho sentido, se puede arreglar pulsando botón derecho sobre el proyecto -> Maven -> Update Project -> Ok. Esto puede pasar si alguna dependencia se ha quedado a medio descargar en la caché.

6. Y ya arrancará nuestra aplicación llevándonos a `index.html` como siempre. En siguientes apartados, añadiremos datos a la base de datos y los mostraremos en la vista.

🔔 **AVISO:** Un error muy MUY MUY común es tener el servidor MySQL apagado. Recuerda tenerlo arrancado todo el tiempo que esté el proyecto en ejecución.

Persistencia con JPA

Cuando desarrollamos una aplicación en Java, uno de los primeros requerimientos que debemos resolver es la integración con una base de datos para guardar, actualizar, borrar y recuperar la información que utiliza nuestra aplicación.

Se llama “persistencia” de los objetos a su capacidad para guardarse y recuperarse desde un medio de almacenamiento. La persistencia en Base de Datos relacionales se suele implementar mediante el desarrollo de funcionalidad específica utilizando la tecnología JDBC o mediante frameworks que automatizan el proceso a partir de mapeos (conocidos como *Object Relational Mapping*, ORM) como es el caso de Hibernate.

Spring Boot simplifica todo el proceso, eliminando archivos xml y clases Java de configuración.

Reutilicemos la clase `User` de anteriores prácticas, pero añadiéndole unos atributos nuevos. Lo añadiremos al constructor y le creamos sus `getters` y `setters` y nuevo `.toString()`. En el proyecto anterior creamos la clase en un paquete llamado `users` y tendrá el siguiente contenido:

```
1 //Corregir el paquete por el de tu proyecto
2 package com.ejemplo.testdb.controllers.users;
3
4 import java.util.Random;
5 public class User {
6     // Atributos
7     private int id;
8     private String userName;
9     private String email;
10    private String password;
11    private boolean admin;
12
13    // Constructores
14    public User() {
15    }
16
17    public User(int id, String userName, String email, String password,
18    boolean admin) {
19        super();
20        this.id = id;
21        this.userName = userName;
22        this.email = email;
23        this.password = password;
24        this.admin = admin;
25    }
26
27    public User(String email) {
28        Random r = new Random();
29        this.id = Math.abs(r.nextInt());
30        this.setUserName(email);
31        this.email = email;
32        this.password = Integer.toHexString(this.hashCode());
33        this.admin = false;
34    }
```

```

35 // Getters y Setters
36 public int getId() {
37     return id;
38 }
39
40 public void setId(int id) {
41     this.id = id;
42 }
43
44 public String getUsername() {
45     return userName;
46 }
47
48 public void setUsername(String userName) {
49     this.userName = userName;
50 }
51
52 public String getEmail() {
53     return email;
54 }
55
56 public void setEmail(String email) {
57     this.email = email;
58 }
59
60 public String getPassword() {
61     return password;
62 }
63
64 public void setPassword(String password) {
65     this.password = password;
66 }
67
68 public boolean isAdmin() {
69     return admin;
70 }
71
72 public void setAdmin(boolean admin) {
73     this.admin = admin;
74 }
75
76 @Override
77 public String toString() {
78     return "User [id=" + id +
79         ", userName=" + userName +
80         ", email=" + email +
81         ", password=" + password +
82         ", admin=" + admin + "]";
83 }
84 }

```

Ahora deberíamos de ir a nuestra base de datos, crear un script para definir la tabla Usuarios con los campos id, userName y password... etc, definir los tipos de los campos de MySQL compatibles con los tipos de Java. En resumidas cuentas, **debemos conocer el lenguaje SQL**, y no solo las consultas DML (*Data Manipulation Language*), si no también las DDL (*Data Definition Language*) para

la creación de las bases de datos y sus restricciones.

Si más adelante, decidimos añadir un atributo a nuestra clase (o modificar alguno existente), deberemos ir de nuevo a nuestra base de datos, hacer las modificaciones pertinentes al script y actualizar la tabla con los nuevos atributos que le hayamos hecho a la clase.

Con Spring Data, sólo tenemos que añadir una serie de anotaciones, y JPA se encargará de **CREAR** la tabla por nosotros realizando las conversiones entre los tipos de datos de Java y MySQL. Las anotaciones serán las siguientes:

```
1  import javax.persistence.Column;
2  import javax.persistence.Entity;
3  import javax.persistence.GeneratedValue;
4  import javax.persistence.Id;
5  import javax.persistence.Table;
6
7  @Entity
8  @Table(name="usuarios")
9  public class User {
10     // Atributos
11     @Id
12     @GeneratedValue
13     private int id;
14     @Column(unique = true)
15     private String userName;
16     @Column(unique = true)
17     private String email;
18     @Column(name="pass")
19     private String password;
20     private boolean admin;
21
22     //De aquí para abajo es todo igual
23 }
```

- Con `@Entity` especificamos que esa clase será una Entidad en la BD.
- Con `@Table` especificamos el nombre de la tabla en nuestra bd (por defecto, sin esta anotación, nos creará una tabla con el mismo nombre que la clase, `user`).
- `@Id` encima de un atributo, le indicamos que ese campo será el campo clave de la tabla, y con `@GeneratedValue` le indicamos que será un campo autogenerado.
 - `@GeneratedValue(strategy=GenerationType.AUTO)` Es la opción por defecto si no se indica nada (como en nuestro ejemplo). Permite al proveedor de la persistencia elegir la estrategia de generación de id's.
 - `@GeneratedValue(strategy=GenerationType.IDENTITY)` Usará una columna con incremento automático que permite que la BD genere un nuevo valor para cada operación de inserción.
- Si queremos que los nombres de los campos en MySQL sean distintos que los de la clase en Java, podemos usar anotaciones como `@Column(name="pass")`, así el atributo `password`, en la tabla se llamaría `pass`. Se recomienda usar los mismos nombres en Java y MySQL, en la medida de lo posible.

Ahora debemos añadirle unas nuevas líneas al `application.properties` para indicarle lo siguiente:


```

1 #Configuración JPA - Hibernate
2 spring.jpa.hibernate.ddl-auto = update
3 spring.jpa.show-sql = true

```

Con `spring.jpa.hibernate.ddl-auto = update`, le decimos a JPA que se encargue de actualizar el esquema de la base de datos si fuera necesario. (Se añadió algún atributo nuevo, modificó algún tipo de datos, etc.)

Con `spring.jpa.show-sql = true` nos mostrará por consola todas las sentencias sql que serán ejecutadas por JPA. Así veremos que es lo que hace bajo el capó y podremos detectar errores.

Si reiniciamos nuestra aplicación, ya tendremos nuestra tabla creada tal y como hemos definido en nuestra clase `User`.

```

1 MariaDB [(none)]> USE dbejemplo;
2 Database changed
3 MariaDB [dbejemplo]> SHOW TABLES;
4 +-----+
5 | Tables_in_dbejemplo |
6 +-----+
7 | hibernate_sequence |
8 | usuarios            |
9 +-----+
10 2 rows in set (0.000 sec)
11
12 MariaDB [dbejemplo]> DESCRIBE usuarios;
13 +-----+-----+-----+-----+-----+-----+
14 | Field      | Type          | Null | Key | Default | Extra |
15 +-----+-----+-----+-----+-----+-----+
16 | id         | int(11)       | NO   | PRI | NULL    |      |
17 | admin      | bit(1)        | NO   |     | NULL    |      |
18 | email      | varchar(255)  | YES  | UNI | NULL    |      |
19 | pass       | varchar(255)  | YES  |     | NULL    |      |
20 | user_name  | varchar(255)  | YES  | UNI | NULL    |      |
21 +-----+-----+-----+-----+-----+-----+
22 5 rows in set (0.027 sec)
23
24 MariaDB [dbejemplo]>

```

🔗 Podemos fijarnos que hasta hizo la conversión del nombre `userName` a `user_name`. De *camelCase* que se usa en Java a *snake_case* que se usa en MySQL.

🧠 **Nota:** Te habrás fijado que se creó una tabla llamada `hibernate_sequence`. La usará *hibernate* para generar las `id` de forma secuencial automáticamente, aunque dependerá de la estrategia elegida para generarlas con la anotación `@GeneratedValue(strategy = GenerationType.*)`

Repositorios

Para acceder a los datos, usamos servicios, que desarrollarían cada uno de los métodos para seleccionar, eliminar, actualizar e insertar objetos en la base de datos. Spring Data ofrece una nueva solución, agregando una nueva capa intermedia para el acceso a los datos, pudiendo ser usada en los servicios. **Son los repositorios.**

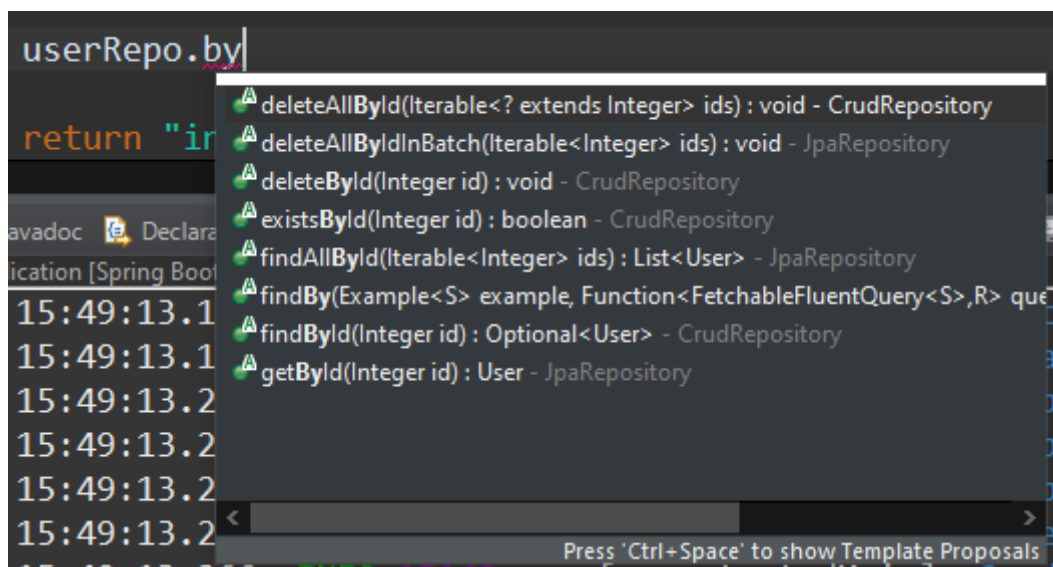
El repositorio es una especie de servicio que ya tiene desarrollado todos los métodos necesarios para interactuar con la base de datos, pudiendo prescindir de la típica capa DAO.

Tan sólo tenemos que definir la interfaz. Nuestra interfaz heredará de la interfaz `JpaRepository`. **Punto, no hay más.**

```
1 import org.springframework.data.jpa.repository.JpaRepository;
2 public interface UserRepo extends JpaRepository<User, Integer> {
3
4 }
```

La implementación de los métodos ya están desarrollados en la clase `JpaRepository`. Tenemos que parametrizarle la clase del objeto con la que va a tratar (en nuestro ejemplo, `User`), y la clase del tipo de dato de su clave primaria (en nuestro ejemplo, `Integer`).

Y ya tenemos disponible el repositorio, que una vez esté inyectado en el controlador que queramos, podremos crear, obtener, actualizar y borrar elementos de la tabla `usuarios`, entre otras muchas cosas.



Todo esto sin tener que cargar driver con `DriverManager`, crear una conexión con `getConnection()`, datasource, crear objetos `Statement` o `PreparedStatement` y manejar ninguna instrucción SQL, que nos devuelva objetos `ResultSet`, con lo entretenido que eran manejarlos. Y por supuesto capturar las mil excepciones que se podían producir en cada uno de los pasos.

Nota: En algunos manuales verás que también usan `CrudRepository` en lugar de `JpaRepository`. La principal diferencia es que `JpaRepository` hereda de `PagingAndSortingRepository`, que a su vez, hereda de `CrudRepository`.

- `CrudRepository` proporciona funciones de CRUD.
- `PagingAndSortingRepository` proporciona métodos para hacer paginación y ordenar registros.
- `JpaRepository` proporciona algunos métodos relacionados con la JPA, más todo lo anterior debido a la herencia.


Conclusión: Depende de las funciones que usemos, podemos usar unos u otros. Si vamos a hacer un CRUD simple, bastará con `CrudRepository`.

Usando el repositorio

Para usar nuestro repositorio, tendremos que inyectarlo en el controlador (o en el servicio) que queramos con la anotación `@Autowired`, de la misma forma que inyectamos los servicios u otros objetos.

Siguiendo nuestro ejemplo, lo vamos a usar en el `HomeController` para añadir un nuevo usuario a la tabla.

```
1  @Controller
2  public class HomeController {
3
4      @Autowired
5      private UserRepo userRepo;
6
7      ...
8
9      @GetMapping("/usuario/nuevo/{email}")
10     public String creaUsuario(
11         @PathVariable String email
12     ) {
13         User nuevo = new User(email);
14         nuevo.setId(0);
15         userRepo.save(nuevo);
16         return "index";
17     }
18 }
19
```

 **Nota:** Aquí inyectamos el repositorio directamente en el `HomeController` para simplificar el código, pero sería conveniente crear un servicio y usar el repositorio desde el servicio, tal y como hemos visto anteriormente.

Hemos creado una nueva url, `/usuario/nuevo/{email}` para poder crear un usuario fácilmente y guardarlo en la base de datos y probar que funciona.


- Aunque la tabla usuarios tenga varios campos, sólo necesitamos un email. `User` tiene un constructor que sólo recibe un email y asigna automáticamente unos valores a los demás atributos (en `id` pone un número aleatorio, en `userName` usa el email, en `password` usa el *hashcode*, y por defecto `admin` = false).
- Con `@PathVariable` extraemos de la url el email que necesitamos. Con eso crea el usuario y se lo pasa por parámetro al método `userRepo.save()`. El valor del atributo `id` lo ponemos a 0 para que entienda que es una nueva inserción (ningún usuario previo tendrá la id 0).

Si escribimos en la url del proyecto, `/usuario/nuevo/pepe@ejemplo.com`, nos deberá de crear ese registro en la base de datos. Y si nos vamos a la base de datos, podemos comprobar que se grabó correctamente.

```

1 MariaDB [dbejemplo]> SELECT * FROM usuarios;
2 +-----+-----+-----+-----+-----+-----+
3 | id | admin | email                | pass      | user_name      |
4 +-----+-----+-----+-----+-----+-----+
5 | 1 |      | pepe@ejemplo.com    | 3fbdfc0   | pepe@ejemplo.com |
6 +-----+-----+-----+-----+-----+-----+
7 1 row in set (0.000 sec)
8
9 MariaDB [dbejemplo]>

```

 **Nota:** Puedes observar por la consola del IDE, las consultas SQL que se enviaron a la base de datos.

El acceso a la capa de datos se ve simplificado usando los repositorios, además que su uso no depende de la estructura o tipo de base de datos que usemos, pudiendo cambiar la base de datos, sin que ello afecte a la implementación de nuestra capa DAO.

Obtener datos

Antes de hacer un CRUD completo, veamos de una manera rápida de recuperar la información de la base de datos y mostrarla en nuestra página principal sin tener que irnos manualmente a la consola MySQL para comprobar los datos.

Pues por hacerlo de la forma más simple posible, lo haremos en el mismo método `index()` que ya tenemos creado en el `HomeController`, de forma que visualicemos los datos tan pronto entremos a la aplicación.

```

1 @GetMapping("/")
2 public String index(Model model) {
3     List<User> lista = userRepo.findAll();
4     model.addAttribute("lista", lista);
5     return "index";
6 }

```

- Inyectamos el `model`, ya que tendremos que adjuntar la lista.
- Con el método `.findAll()` del repositorio, ya nos devuelve directamente un `List<User>`. Lo guardamos y lo adjuntamos al `model`.
- Y mostramos la vista `index`.

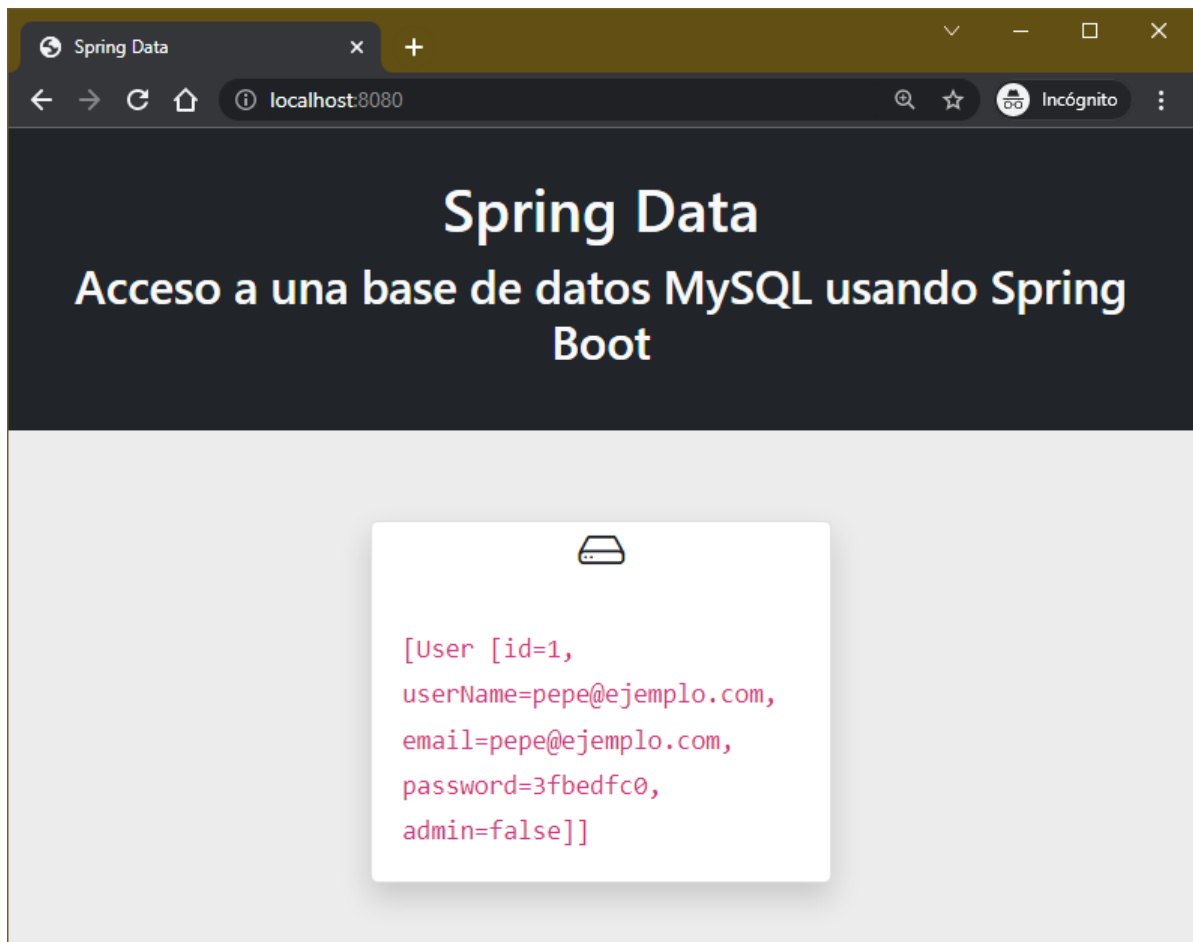
Ahora ya tenemos la lista con todos los usuarios, en el modelo y podemos representarla en la vista como ya hemos hecho otras veces. Imprimimos la lista directamente en el cuerpo de la tarjeta que ya teníamos antes. Hemos usado `<code>` pero se podría en cualquier elemento HTML.

```

1 ...
2 <div class="card-body">
3     <code>
4         [[${lista}]]
5     </code>
6 </div>
7 ...

```

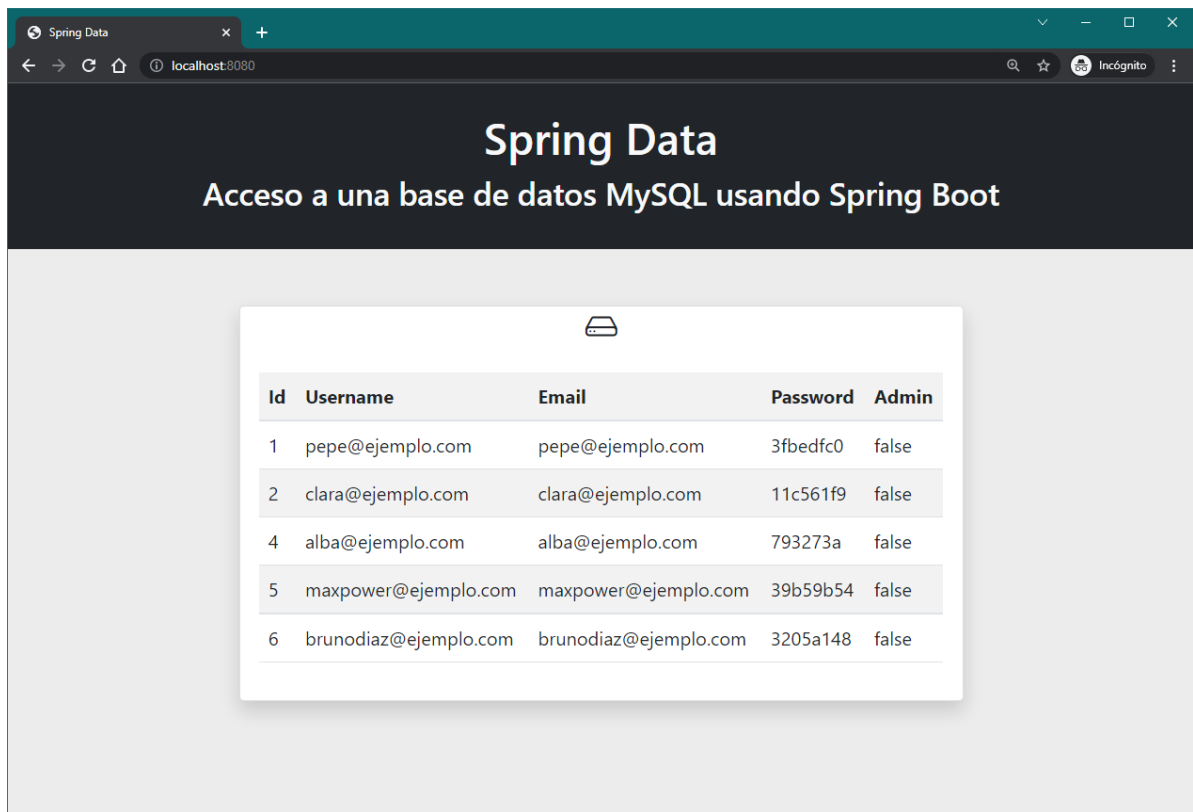
Al entrar la aplicación, debería mostrarse algo así:



Ahora que ya sabemos que los datos están llegando perfectamente, ya podemos mejorar la forma en la que mostramos los datos, visualizándolos por ejemplo, en una tabla.

```
1 <div class="card-body">
2   <table class="table table-striped">
3     <tr>
4       <th>Id</th>
5       <th>Username</th>
6       <th>Email</th>
7       <th>Password</th>
8       <th>Admin</th>
9     </tr>
10    <tr th:each="user : ${lista}">
11      <td>[[${user.id}]]</td>
12      <td>[[${user.userName}]]</td>
13      <td>[[${user.email}]]</td>
14      <td>[[${user.password}]]</td>
15      <td>[[${user.admin}]]</td>
16    </tr>
17  </table>
18 </div>
```

Y añadimos unos cuantos usuarios más, y así se vería finalmente.



⚠ **Atención:** Ten en cuenta que después de añadir usuarios, el controlador, nos lleva a `index.html`, pero SIN PASAR por el método que recupera los valores de la tabla y los coloca en el `model`. Por lo que la tabla saldrá vacía. Si en el método `creaUsuario()`, cambiamos el `return "index"` por `return "redirect:/"`, en lugar de llevarnos a la vista, nos REDIRECCIONARÁ a la url `/`, lo cual nos llevará al método `index()` que cargará los datos y nos llevará a `index.html`, pero ahora si, con los datos cargados en el model. Y si veremos que al crear un usuario nuevo, se refleja en la tabla.

Otro método útil para recuperar datos según una id:

- `.findById()` -> El cual recibirá una id y devolverá UN objeto (ambos del mismo tipo definido en el repositorio, respectivamente). En nuestro ejemplo, recibirá un `Integer` y devolverá un `User`.

Actualizar datos

Para actualizar datos usaremos el mismo método `.save()` con la única diferencia, que el objeto que reciba el método, deberá tener la id existente de un registro en la base de datos.

```
1 userActualizado.setId(6);
2 userActualizado.setNombre("Bruce Wayne");
3 userRepo.save(userActualizado);
```

Borrar datos

Para borrar algún registro de la base de datos, los repositorios tienen muchas opciones con métodos que empiezan por `delete`. Los más útiles son:

- `delete(User u)` -> Recibe un objeto del tipo definido como entidad en el repositorio y el método lo buscará y lo borrará.
- `deleteById(Integer id)` -> Recibe una id y borrará la entidad cuya id sea igual que la recibida.

```
1 User userBuscado = userRepo.findByEmail("enrico.palazzo@acp2.com");
2 userRepo.delete(userBuscado); //Borramos el usuario buscado
3
4 userRepo.deleteById(5); //Borramos el usuario cuya id es 5
```

Personalizar el repositorio

Hemos visto que los repositorios tienen muchos métodos para hacer prácticamente todo lo que necesitamos, pero, ¿y si necesito un método, por ejemplo, que me busque por `userName` y me devuelva el usuario que tenga un valor concreto en ese campo? **Pues es tan fácil como pedir un deseo.**

Definimos la firma del método que nos gustaría tener, y como por arte de magia, lo tendremos disponible en nuestro repositorio:

```
1 public interface UserRepo extends JpaRepository<User, Integer> {
2     public User findByUserName(String userName);
3 }
```

Para probarlo, hemos hecho el método siguiente:

```
1 @GetMapping("/usuario/username/{userName}")
2 public String findByUserName(
3     @PathVariable String userName
4 ) {
5
6     User userEncontrado = userRepo.findByUserName(userName);
7     System.out.println(userEncontrado);
8
9     return "index";
10 }
```

Y si hacemos una consulta a `/usuario/username/maxpower@ejemplo.com`, veremos por consola:

```
1 User [id=5, userName=maxpower@ejemplo.com, email=maxpower@ejemplo.com,
   password=39b59b54, admin=false]
```

🐞 Funcionó, sin escribir una consulta SQL, o tener que implementar el código del método `findByUserName()`.

Vamos más allá....se puede hacer cosas tan locas como lo siguiente:

```

1 public interface UserRepo extends JpaRepository<User, Integer> {
2     public User findByUserNameStartingWithIgnoreCase(String userName);
3     public User findByEmailAndPassword(String email, String pass);
4     public boolean existsByEmailAndPassword(String email, String pass);
5     public List<User> findByIdGreaterThan(int id);
6     public List<User> findByEmailContaining(String emailParcial);
7     public List<User> findByAdminTrue();
8 }

```

Nosotros decidimos si nos devuelve un `User`, o un `List<User>`. El método tendrá el prefijo `findBy` y contendrá el nombre de los atributos por los que se quiera filtrar unidos por los operadores `And`, `Or`, `Between`, `LessThan`, `GreaterThan`. Es la forma más simple de hacer una consulta aunque no nos servirá si tenemos muchos parámetros (el nombre del método será muy largo) o si la consulta es más compleja.

Puedes encontrar la lista completa de operadores para definir tus métodos [en el siguiente enlace de la documentación oficial](#).

⚠ Atención: El nombre de los atributos en los métodos deberán nombrarse de la misma forma que están definidos, usando la anotación camelCase. Si por ejemplo, el atributo es `userName`, el método deberá llamarse `findByUserName`. Sus getters y setters deberán estar bien definidos también, si no no funcionará.

Repositorios REST

El crear una api REST a nuestro proyecto es sumamente fácil. Debemos añadir la dependencia `Rest Repositories` al crear el proyecto o añadir el siguiente trozo al `pom.xml`. Después actualizar Maven y re-arrancar el proyecto.

```

1 <dependency>
2     <groupId>org.springframework.boot</groupId>
3     <artifactId>spring-boot-starter-data-rest</artifactId>
4 </dependency>

```

Después hay que añadir unas anotaciones a nuestros repositorios para tener disponible el acceso a nuestros objetos en formato JSON a través de REST.

```

1 @Repository
2 @RepositoryRestResource(path="usuarios")
3 public interface UserRepo extends JpaRepository<User, Integer> {
4     ...
5 }

```

Y ya automáticamente, al hacer una petición a `localhost:8080/usuarios` obtendremos la siguiente respuesta:

```

1 {
2     "_embedded": {
3         "users": [
4             {
5                 "userName": "pepe@ejemplo.com",
6                 "email": "pepe@ejemplo.com",

```



```
7     "password": "3fbedfc0",
8     "admin": false,
9     "_links": {
10         "self": {
11             "href": "http://localhost:8080/usuarios/1"
12         },
13         "user": {
14             "href": "http://localhost:8080/usuarios/1"
15         }
16     }
17 },
18 {
19     "userName": "clara@ejemplo.com",
20     "email": "clara@ejemplo.com",
21     "password": "11c561f9",
22     "admin": false,
23     "_links": {
24         "self": {
25             "href": "http://localhost:8080/usuarios/2"
26         },
27         "user": {
28             "href": "http://localhost:8080/usuarios/2"
29         }
30     }
31 },
32 {
33     "userName": "alba@ejemplo.com",
34     "email": "alba@ejemplo.com",
35     "password": "793273a",
36     "admin": true,
37     "_links": {
38         "self": {
39             "href": "http://localhost:8080/usuarios/4"
40         },
41         "user": {
42             "href": "http://localhost:8080/usuarios/4"
43         }
44     }
45 },
46 {
47     "userName": "maxpower@ejemplo.com",
48     "email": "maxpower@ejemplo.com",
49     "password": "39b59b54",
50     "admin": false,
51     "_links": {
52         "self": {
53             "href": "http://localhost:8080/usuarios/5"
54         },
55         "user": {
56             "href": "http://localhost:8080/usuarios/5"
57         }
58     }
59 },
60 {
61     "userName": "brunodiaz@ejemplo.com",
62     "email": "brunodiaz@ejemplo.com",
63     "password": "3205a148",
64     "admin": true,
```

```
65     "_links": {
66         "self": {
67             "href": "http://localhost:8080/usuarios/6"
68         },
69         "user": {
70             "href": "http://localhost:8080/usuarios/6"
71         }
72     },
73     {
74         "userName": "fernandosimon@ejemplo.com",
75         "email": "fernandosimon@ejemplo.com",
76         "password": "42c664c8",
77         "admin": false,
78         "_links": {
79             "self": {
80                 "href": "http://localhost:8080/usuarios/7"
81             },
82             "user": {
83                 "href": "http://localhost:8080/usuarios/7"
84             }
85         }
86     },
87     {
88         "userName": "enricopalazzo@otrodominio.com",
89         "email": "enricopalazzo@otrodominio.com",
90         "password": "7206eae8",
91         "admin": false,
92         "_links": {
93             "self": {
94                 "href": "http://localhost:8080/usuarios/8"
95             },
96             "user": {
97                 "href": "http://localhost:8080/usuarios/8"
98             }
99         }
100     }
101 ]
102 },
103 "_links": {
104     "self": {
105         "href": "http://localhost:8080/usuarios"
106     },
107     "profile": {
108         "href": "http://localhost:8080/profile/usuarios"
109     },
110     "search": {
111         "href": "http://localhost:8080/usuarios/search"
112     }
113 },
114 "page": {
115     "size": 20,
116     "totalElements": 7,
117     "totalPages": 1,
118     "number": 0
119 }
120 }
121 }
```

Y una petición a `localhost:8080/usuarios/8` obtendremos la siguiente respuesta:

```
1 {
2   "userName": "enricopalazzo@otrodominio.com",
3   "email": "enricopalazzo@otrodominio.com",
4   "password": "7206eaeb",
5   "admin": false,
6   "_links": {
7     "self": {
8       "href": "http://localhost:8080/usuarios/8"
9     },
10    "user": {
11      "href": "http://localhost:8080/usuarios/8"
12    }
13  }
14 }
```

Personalizar la url base

Podemos añadir una ruta url base para todo el rest, añadiendo la siguiente configuración en el `application.properties`:

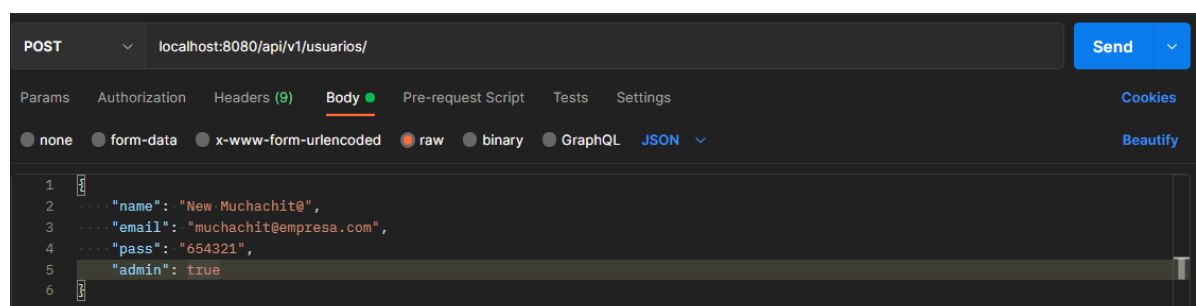
```
1 spring.data.rest.basePath=/api/v1
```

Así para usar las rutas anteriores, deberíamos ir a `localhost:8080/api/v1/usuarios`. Así tendríamos por un lado el acceso al api rest en una ruta, y nuestros controladores por otra.

Acepta peticiones de otros verbos

El API Rest que “hemos montado 😊😊”, no sólo nos devolverá en formato JSON los objetos que exista en la base de datos, también aceptará peticiones POST, PUT, DELETE, PATCH, etc.

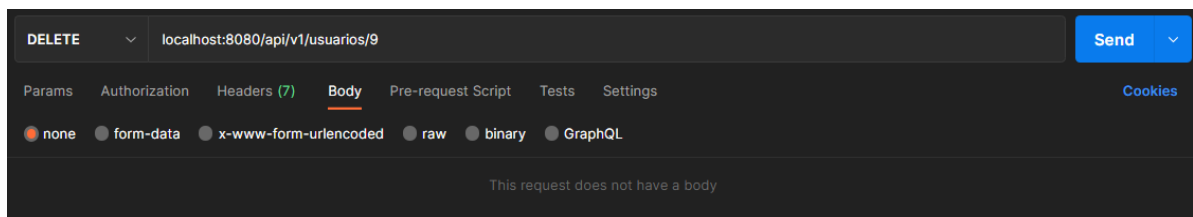
Veamos como desde postman acepta una petición `POST`, y lo inserta en la base de datos.



```
1 MariaDB [crud_usuarios]> select * from usuarios;
2 +----+-----+-----+-----+-----+
3 | id | name          | email                | pass    | admin |
4 +----+-----+-----+-----+-----+
5 | 1  | Administrador | admin@empresa.com    | 654321  | ☺     |
6 | 2  | Usuario       | user@empresa.com     | 123456  |       |
7 | 3  | Visor         | visor@empresa.com    | 123456  |       |
8 | 4  | Max Power     | max.power@empresa.com | 123456  |       |
9 | 5  | Bruno Díaz    | bruce.wayne@empresa.com | 123456  |       |
10 | 6  | Enrico Palazzo | enricco.palazo@empresa.com | 123456  |       |
11 | 7  | Ricardo Tapia | ri.cardo@empresa.com  | 123456  |       |
12 | 9  | New Muchachit@ | muchachit@empresa.com | 654321  | ☺     |
```

```
13 +-----+-----+-----+-----+-----+
14 8 rows in set (0.000 sec)
```

Y como acepta una petición `DELETE` y lo borra sin problemas.



Búsquedas personalizadas

Si vamos a `localhost:8080/api/v1/usuarios/search` nos mostrará todos los métodos personalizados que hayamos definido en nuestro repositorio, pudiendo usarlos también desde el API REST:

```
1 {
2   "_links" : {
3     "findByEmailAndPass" : {
4       "href" :
5       "http://localhost:8080/api/v1/usuarios/search/findByEmailAndPass{?
6       email,pass}",
7       "templated" : true
8     },
9     "self" : {
10      "href" : "http://localhost:8080/api/v1/usuarios/search/"
11    }
12  }
13 }
```

`localhost:8080/api/v1/usuarios/search/findByEmailAndPass?`
`email=ri.cardo@empresa.com&pass=123456`

```
1 {
2   "name" : "Ricardo Tapia",
3   "email" : "ri.cardo@empresa.com",
4   "pass" : "123456",
5   "admin" : false,
6   "_links" : {
7     "self" : {
8       "href" : "http://localhost:8080/api/v1/usuarios/7"
9     },
10    "usuario" : {
11      "href" : "http://localhost:8080/api/v1/usuarios/7"
12    }
13  }
14 }
```

CRUD Completo

Veamos una aplicación final, usando todo lo visto durante el curso.

1. Haremos un login, el cual tendremos que autenticarnos con un usuario y contraseña que deberá estar registrado en la base de datos.
2. Una vez hagamos login, guardará el usuario en la sesión y mientras no se cierre, iremos directamente a la página principal.
3. En la página principal habrá un listado de una tabla de una base de datos, en el cual podremos añadir nuevos registros, modificar su contenido y borrarlos.
4. Opcionalmente, podemos restringir las opciones de añadir, borrar y editar sólo para usuarios con rol de administrador.
5. Podremos cerrar sesión, de forma que nos llevará de nuevo al login.

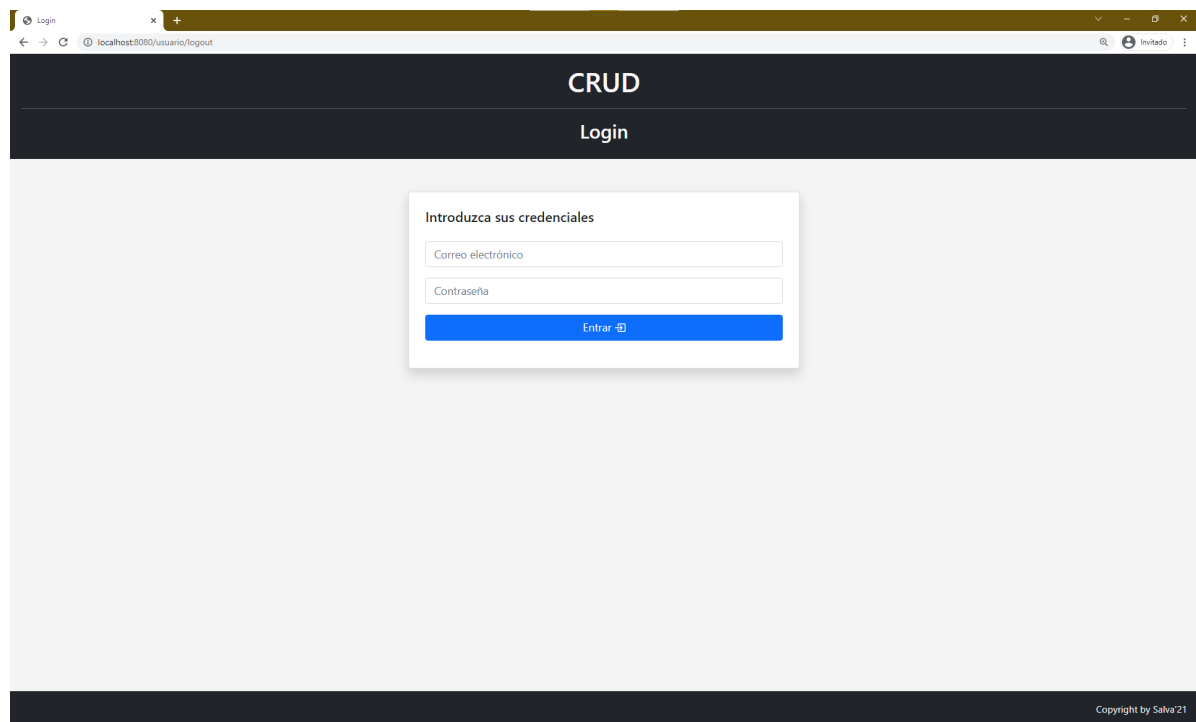
Encontrarás la vistas y las estructuras incompletas de algunas clases y servicios en el repositorio siguiente:

<https://github.com/borilio/curso-spring-boot/tree/master/assets/clases/practica-7>

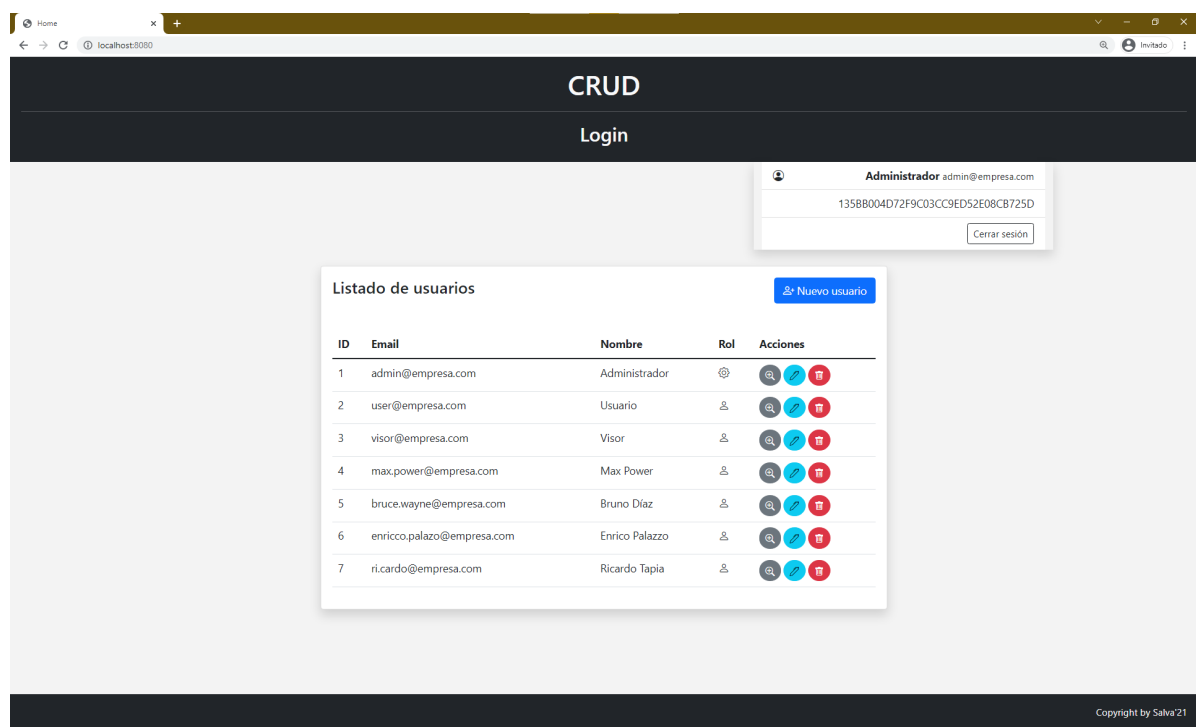
En la carpeta `/resources/sql` del repositorio de GIT encontrarás los scripts SQL necesarios para crear el esquema (`schema.sql`) y los datos (`data.sql`).

Vista previa

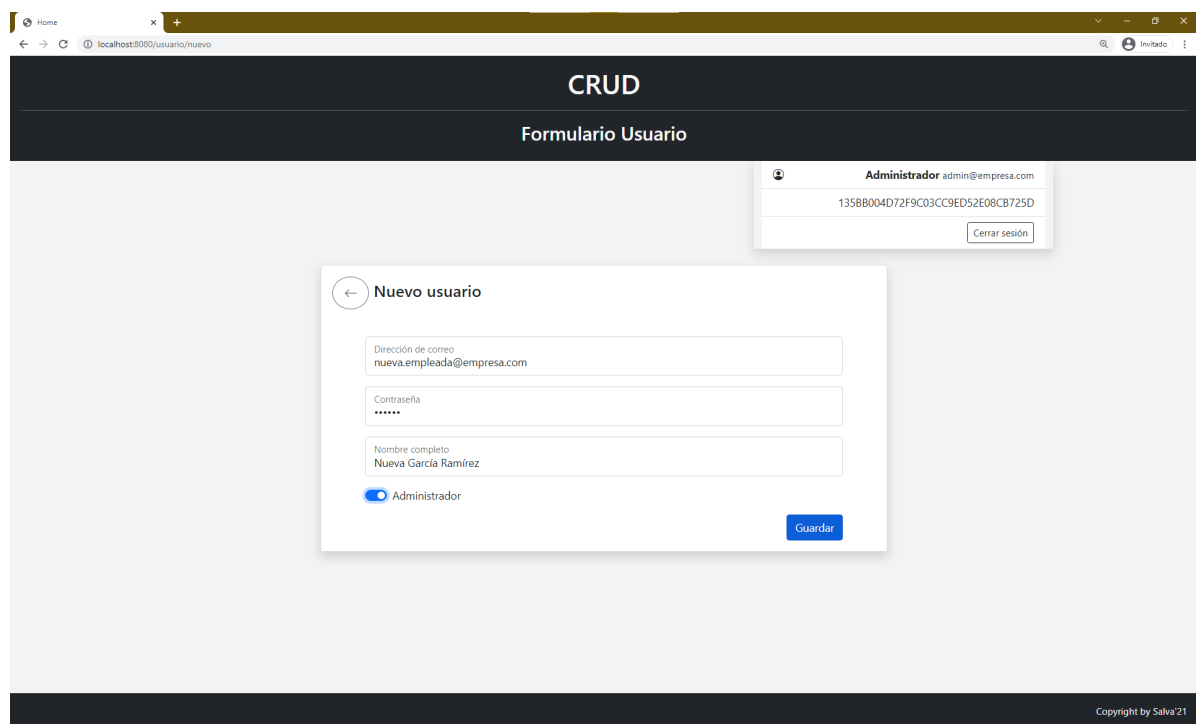
Login inicial:



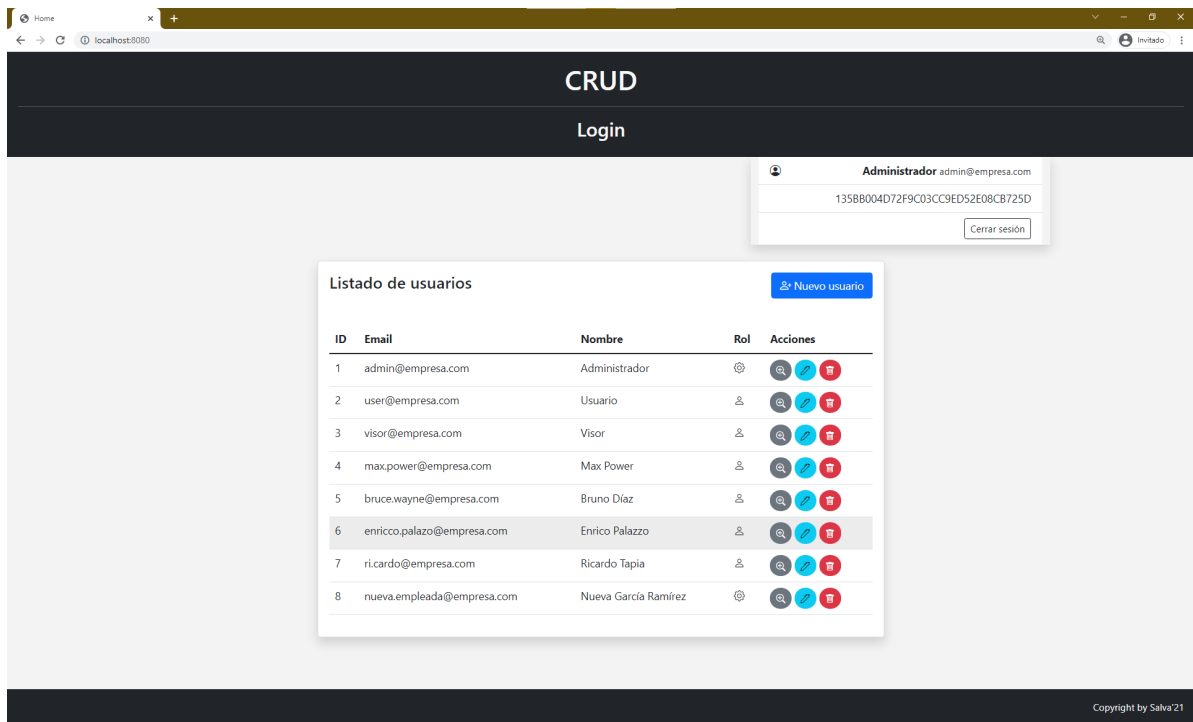
Menú principal:



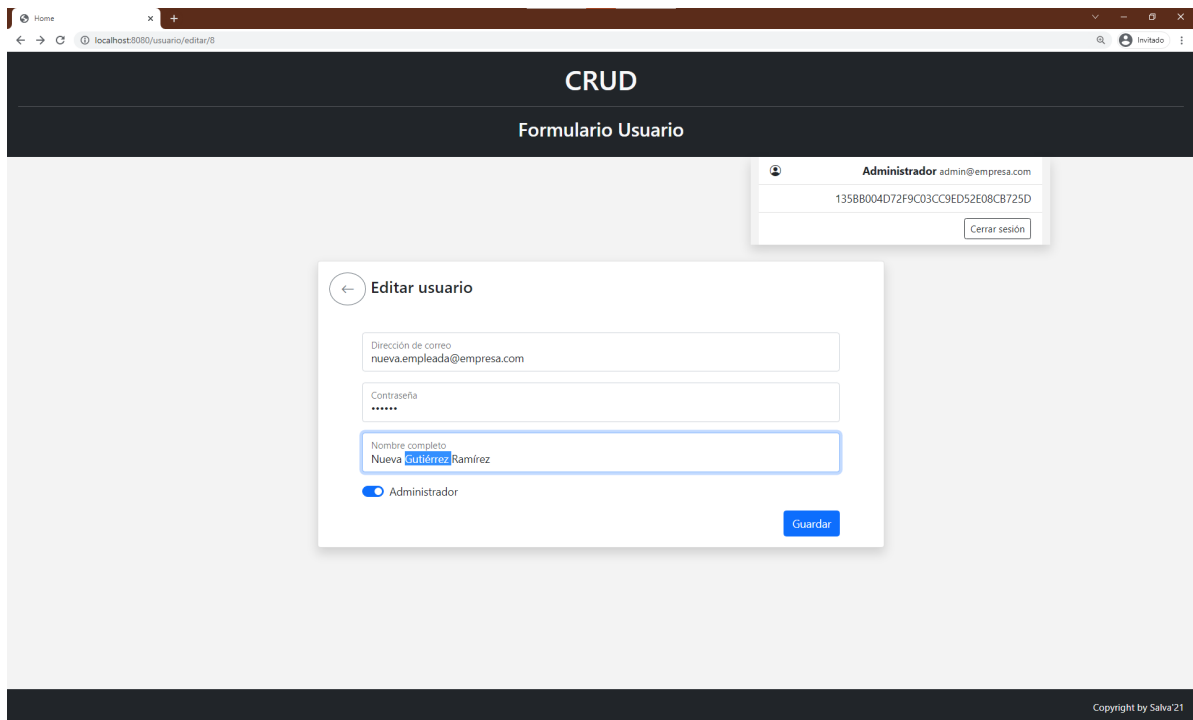
Nuevo usuario:



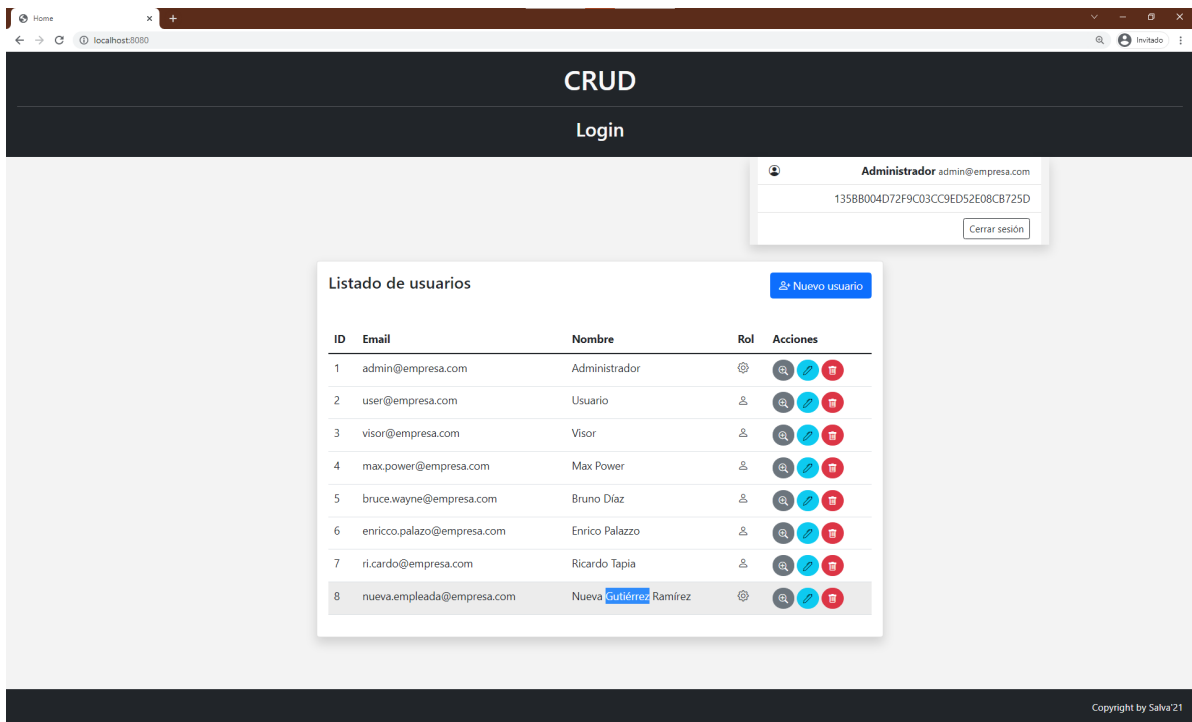
Comprobación de nuevo usuario creado:



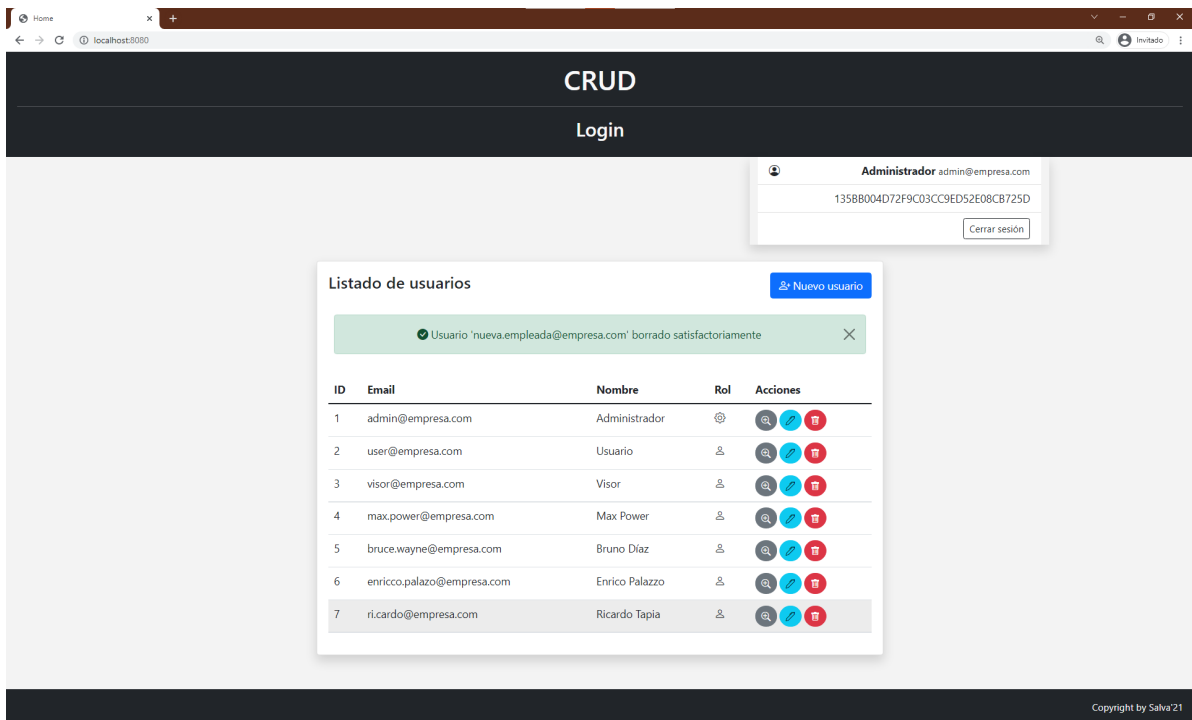
Editamos ese nuevo usuario (nos volcará la información en el formulario):



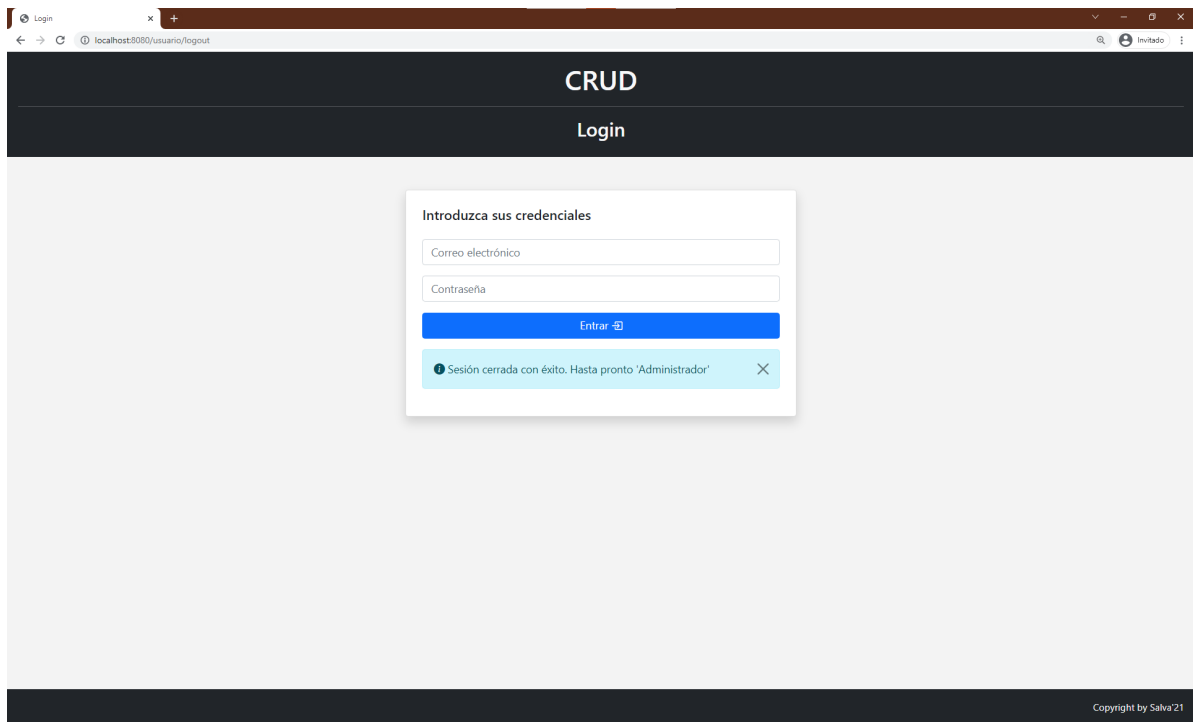
Comprobamos la actualización de sus datos:



Borramos el usuario nuevo:



Cerramos sesión:



Anexo: Persistencia con JPA en base de datos relacionales

 En construcción. Usar con precaución 

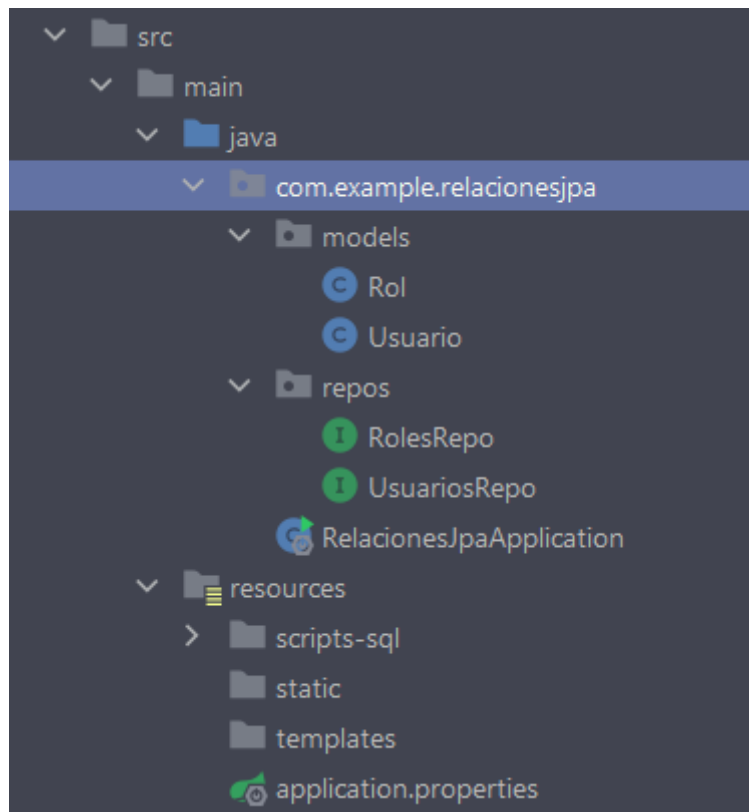
Este documento completo trata la persistencia de tablas independientes, que no están relacionadas con otras. En este apartado veremos como se usan en JPA una base de datos que posee relaciones entre tablas.

Proyecto inicial

En el siguiente repositorio, está el proyecto completo que nos servirá de ejemplo, con commits en cada punto importante para ver el proyecto desde el inicio y los cambios realizados durante la evolución del ejemplo.

 <https://github.com/borilio/curso-spring-boot-ejemplo-jpa>

Para simplificar, no tendremos vistas y solo tendremos una API REST creada con los repositorios, por lo que no tendremos `controllers`, si no únicamente los modelos (o beans) y sus correspondientes repositorios.



📁 El proyecto tiene momentos concretos señalados, identificados por un commit a git, para que se pueda ver el código fuente de proyecto en un momento especial dado. A continuación mostramos el primer momento, por si necesitas ver el código de los modelos o de los repositorios, o de cualquier otro archivo del proyecto.

🕒 **Momento 1:** Definidos Usuarios y Roles, y sus correspondientes repositorios.

<https://github.com/borilio/curso-spring-boot-ejemplo-jpa/tree/4bf2f8a767376108be33172ac8e4969ad0ee7d30>

Ahora mismo, si arrancamos el proyecto y vamos a `http://localhost:8080/api/usuarios` nos mostrará un JSON con los usuarios:

```
1 {
2   "_embedded": {
3     "usuarios": [
4       {
5         "idRol": 1,
6         "nombre": "Admin 1",
7         "correo": "admin1@empresa.com",
8         "clave": "654321",
9         "_links": {
10          "self": {
11            "href": "http://localhost:8080/api/usuarios/1"
12          },
13          "usuario": {
14            "href": "http://localhost:8080/api/usuarios/1"
15          }
16        }
17      },
18      {
19        "idRol": 1,
20        "nombre": "Admin 2",
21        "correo": "admin2@empresa.com",
22        "clave": "654321",
```

```

23         "_links": {
24             "self": {
25                 "href": "http://localhost:8080/api/usuarios/2"
26             },
27             "usuario": {
28                 "href": "http://localhost:8080/api/usuarios/2"
29             }
30         }
31     },
32     {
33         "idRol": 2,
34         "nombre": "Usuario",
35         "correo": "user@empresa.com",
36         "clave": "123456",
37         "_links": {
38             "self": {
39                 "href": "http://localhost:8080/api/usuarios/3"
40             },
41             "usuario": {
42                 "href": "http://localhost:8080/api/usuarios/3"
43             }
44         }
45     },
46     {
47         "idRol": 3,
48         "nombre": "Visor",
49         "correo": "visor@empresa.com",
50         "clave": "123456",
51         "_links": {
52             "self": {
53                 "href": "http://localhost:8080/api/usuarios/4"
54             },
55             "usuario": {
56                 "href": "http://localhost:8080/api/usuarios/4"
57             }
58         }
59     }
60 ]
61 },
62 "_links": {
63     "self": {
64         "href": "http://localhost:8080/api/usuarios"
65     },
66     "profile": {
67         "href": "http://localhost:8080/api/profile/usuarios"
68     },
69     "search": {
70         "href": "http://localhost:8080/api/usuarios/search"
71     }
72 },
73 "page": {
74     "size": 20,
75     "totalElements": 4,
76     "totalPages": 1,
77     "number": 0
78 }
79 }

```

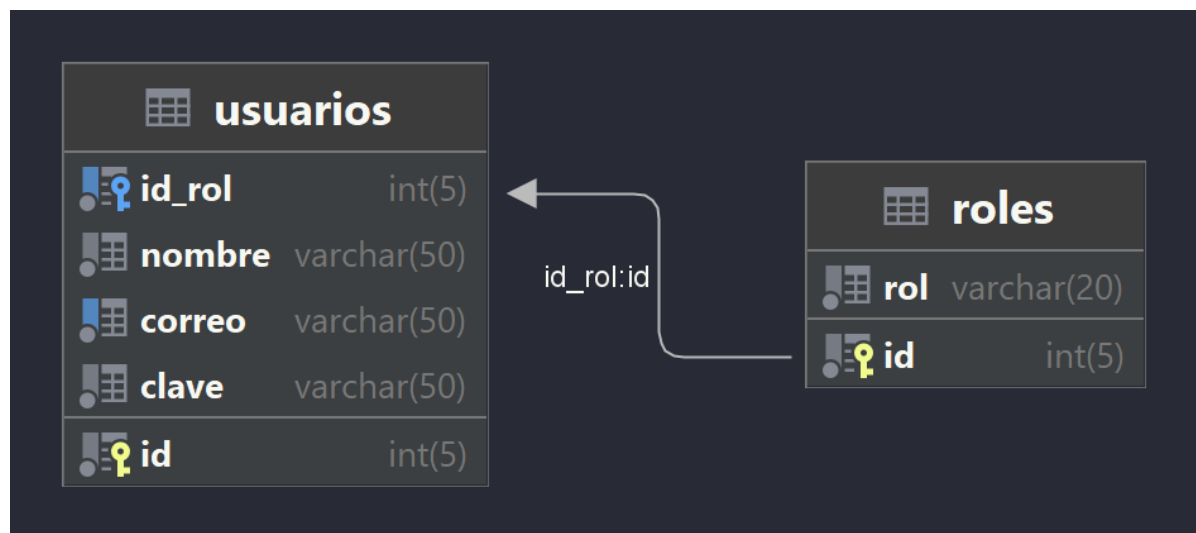
E igualmente con los roles en `http://localhost:8080/api/roles`:

```
1  {
2    "_embedded": {
3      "roles": [
4        {
5          "rol": "administrador",
6          "_links": {
7            "self": {
8              "href": "http://localhost:8080/api/roles/1"
9            },
10           "rol": {
11             "href": "http://localhost:8080/api/roles/1"
12           }
13         }
14       },
15       {
16         "rol": "usuario",
17         "_links": {
18           "self": {
19             "href": "http://localhost:8080/api/roles/2"
20           },
21           "rol": {
22             "href": "http://localhost:8080/api/roles/2"
23           }
24         }
25       },
26       {
27         "rol": "visor",
28         "_links": {
29           "self": {
30             "href": "http://localhost:8080/api/roles/3"
31           },
32           "rol": {
33             "href": "http://localhost:8080/api/roles/3"
34           }
35         }
36       }
37     ],
38     "_links": {
39       "self": {
40         "href": "http://localhost:8080/api/roles"
41       },
42       "profile": {
43         "href": "http://localhost:8080/api/profile/roles"
44       }
45     },
46     "page": {
47       "size": 20,
48       "totalElements": 3,
49       "totalPages": 1,
50       "number": 0
51     }
52   }
53 }
```

Por ahora no hay nada nuevo, vemos como cada repositorio nos da los elementos de cada entidad, por separado sin tener en cuenta las relaciones que están definidas en la base de datos.

Esquema Entidad-Relación

En el proyecto, tenemos una base de datos simple con varias relaciones, pero nos centraremos por ahora en los Usuarios y los Roles. Cada `usuario` tiene un `rol` definido en otra tabla.



Obteniendo información de otras tablas

Ahora mismo, con los repositorios obtenemos la información que hay en las tablas TAL CUAL. Es decir, que si obtengo la lista de usuarios, tendré un `List<Usuario>` donde cada `Usuario` tiene los atributos tal cual están definidos, por lo que podré saber la `id_rol` que tiene el usuario (1, 2, 3, etc.), pero no tendré información del nombre del `rol` ("administrador", "visor", "usuario", etc.).

¿Cómo podemos obtener esa información? Pues sustituyendo el atributo de la clave foránea por el objeto al que hace referencia en la otra tabla además de usar las anotaciones que indicamos a continuación.

En nuestro ejemplo sería cambiando el atributo `id_rol`, por un objeto de la clase `Rol`, y usando las anotaciones correctas.

@ManyToOne - de muchos a uno


Como la relación entre `usuarios` y `roles` es de muchos a uno (un rol está muchas veces en la tabla `usuarios`, y una única vez en la tabla `roles`), pues debemos hacer los siguientes cambios:

1. En la clase `Usuario`, eliminamos el atributo `id_rol` y lo sustituimos por un objeto de la clase `Rol`.
2. Le añadimos la anotación `@ManyToOne`, para indicar el tipo de relación que hay con ese atributo.
3. Le añadimos la anotación `@JoinColumn`, para indicar los campos que se relacionan en ambas entidades. Tiene los siguientes argumentos:
 - `name`: Le indicamos el campo de ESTA entidad que representa la clave foránea.
 - `referencedColumnName`: Le indicamos el campo clave de la tabla a la que hace referencia.
4. No es necesario hacer ningún cambio en la clase `Rol`, ni en ninguno de los dos repositorios.

El código de la clase `Usuario` quedaría así:

 Para simplificar el código, estamos usando Lombok

```
1  @Data
2  @NoArgsConstructor
3  @AllArgsConstructor
4  @Entity
5  @Table(name = "usuarios")
6  public class Usuario {
7      @Id
8      @GeneratedValue(strategy = GenerationType.IDENTITY)
9      private int id;
10     @ManyToOne
11     @JoinColumn(name = "id_rol", referencedColumnName = "id")
12     private Rol rol;
13     private String nombre;
14     private String correo;
15     private String clave;
16 }
```

 **Momento 2:** Modificamos la clase `Usuario` para cambiar la `id_rol` por un objeto de la clase `Rol`.

<https://github.com/borilio/curso-spring-boot-ejemplo-jpa/tree/13f5c9c4c6571eb0b40986c93f5c3e257d507913>

Usando los repositorios en cualquier parte de la aplicación (servicios, controlador, testing, etc), obtenemos el usuario con `id=1` usando la siguiente sentencia:

```
1  //Obtenemos el usuario con id=1 y lo imprimimos por consola...
2  System.out.println(usuariosRepo.findById(1).orElse(null));
```

Antes, sin usar las anotaciones y clases correspondientes, obtendríamos:

```
1  Usuario(id=1, id_rol=1, nombre=Admin 1, correo=admin1@empresa.com,
    clave=654321)
```

Ahora, usando las relaciones correctamente, obtenemos esto:

```
1  Usuario(id=1, rol=Rol(id=1, rol=administrador), nombre=Admin 1,
    correo=admin1@empresa.com, clave=654321)
```

Antes únicamente obtenemos `id_rol`, ahora tenemos un objeto de la clase `Rol`, relleno con todos los valores correctos para esa id concreta.

Por la parte del REST, veremos los siguientes cambios:

Si accedemos a `http://localhost:8080/usuarios/1`, obtendremos:

```
1  {
2      "nombre": "Admin 1",
3      "correo": "admin1@empresa.com",
4      "clave": "654321",
5      "_links": {
```

```

6         "self": {
7             "href": "http://localhost:8080/api/usuarios/1"
8         },
9         "usuario": {
10            "href": "http://localhost:8080/api/usuarios/1"
11        },
12        "rol": {
13            "href": "http://localhost:8080/api/usuarios/1/rol"
14        }
15    }
16 }

```

Y vemos que ahora podemos ir a `http://localhost:8080/usuarios/1/rol`, en la cual obtendremos:

```

1  {
2      "rol": "administrador",
3      "_links": {
4          "self": {
5              "href": "http://localhost:8080/api/roles/1"
6          },
7          "rol": {
8              "href": "http://localhost:8080/api/roles/1"
9          }
10     }
11 }

```

En la misma base de datos hay varias relaciones iguales, de muchos a uno, y que haciendo lo mismo que con los usuarios, podríamos:

- Obtener los tipos de centrales nucleares
- Obtener el nombre de la provincia de la central nuclear
- Obtener la central nuclear que originó una incidencia
- Obtener el usuario que originó la incidencia

@OneToMany - de uno a muchos

Si queremos bidireccionalidad en la relación, podemos usar anotaciones para indicar la relación contraria en la otra dirección. Es decir, si antes estamos obteniendo la información del rol para cada usuario, ahora podríamos obtener los usuarios que usen un determinado rol.


En la tabla contraria, podemos añadirle una lista de usuarios, y automáticamente, podremos obtener los usuarios que tengan una id de rol concreta.

Añadiendo la anotación `@OneToMany` decimos que “un mismo rol, lo pueden tener muchos usuarios”. Y con el argumento `mappedBy` le indicamos el atributo de la otra entidad el cual tiene que pertenecer a esta entidad (le decimos `mappedBy="rol"` porque queremos obtener una `List<Usuario>` que tengan como atributo un determinado `rol`).

```

1  @Data
2  @NoArgsConstructor
3  @AllArgsConstructor
4  @Entity
5  @Table(name = "roles")
6  public class Rol {
7      @Id
8      @GeneratedValue(strategy = GenerationType.IDENTITY)
9      private int id;
10     private String rol;
11     @OneToMany(mappedBy = "rol")
12     private List<Usuario> usuarios;
13 }

```

 **Momento 3:** Modificamos la clase Rol para añadir una bidireccionalidad la relación con `@OneToMany`.

<https://github.com/borilio/curso-spring-boot-ejemplo-jpa/tree/2b91a3a0f324681be3956d39639fc4f78523406d>

Ahora podemos hacer la petición `http://localhost:8080/api/roles/1/usuarios` y nos mostrará un JSON con los **usuarios que tengan el rol con id 1**:

```

1  {
2      "_embedded": {
3          "usuarios": [
4              {
5                  "nombre": "Admin 1",
6                  "correo": "admin1@empresa.com",
7                  "clave": "654321",
8                  "_links": {
9                      "self": {
10                         "href": "http://localhost:8080/api/usuarios/1"
11                     },
12                     "usuario": {
13                         "href": "http://localhost:8080/api/usuarios/1"
14                     },
15                     "rol": {
16                         "href": "http://localhost:8080/api/usuarios/1/rol"
17                     }
18                 }
19             },
20             {
21                 "nombre": "Admin 2",
22                 "correo": "admin2@empresa.com",
23                 "clave": "654321",
24                 "_links": {
25                     "self": {
26                         "href": "http://localhost:8080/api/usuarios/2"
27                     },
28                     "usuario": {
29                         "href": "http://localhost:8080/api/usuarios/2"
30                     },
31                     "rol": {
32                         "href": "http://localhost:8080/api/usuarios/2/rol"
33                     }
34                 }
35             }
36         ]
37     }
38 }

```



```
35         }
36     ]
37 },
38 "_links": {
39     "self": {
40         "href": "http://localhost:8080/api/roles/1/usuarios"
41     }
42 }
43 }
```

⚠ No podremos probar este tipo de relaciones en testing, ya que al no tener una sesión creada provocará una excepción del tipo `org.hibernate.LazyInitializationException`