UNIVERSIDAD DON BOSCO	**UNIVERSIDAD DON BOSCO** **FACULTA DE INGENIERIA** **ESCUELA DE COMPUTACION**
Ciclo I	**Desarrollo de aplicaciones con Web Frameworks** **Spring boot con JPA**

I. OBJETIVOS.

- Que el estudiante identifique las configuraciones necesarias para poder configurar JPA con Sprint Boot.
- Que el estudiante comprenda las anotaciones básicas y esenciales para poder trabajar con JPA.

II. INTRODUCTION

La mayoría de la información de las aplicaciones empresariales es almacenada en bases de datos relacionales. La persistencia de datos en Java, y en general en los sistemas de información, ha sido uno de los grandes temas a resolver en el mundo de la programación.
Al utilizar únicamente JDBC se tiene el problema de crear demasiado código para poder ejecutar una simple consulta. Por lo tanto, para simplificar el proceso de interacción con una base de datos (select, insert, update, delete), se ha utilizado desde hace ya varios años el concepto de frameworks ORM (Object Relational Mapping), tales como JPA e Hibernate.
Jakarta Persistence API, mejor conocido como JPA, es el estándar de persistencia en Java. JPA implementa conceptos de frameworks ORM (Object Relational Mapping).

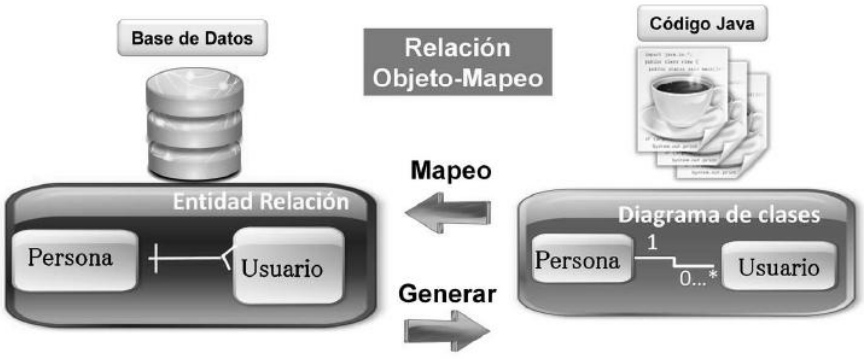


Figura 1. Object Relational Mapping en Java

Como se puede observar en la **figura 1**, un framework ORM nos permite "mapear" una clase Java con una tabla de Base de Datos. Por ejemplo, la clase **Persona** que representa una tabla de la base de datos del mismo nombre, al crear un objeto en memoria, podemos almacenarlo directamente en la tabla de Persona, simplemente ejecutando una línea de código: **em.persist(persona)**. Esto ha simplificado enormemente la cantidad de código a escribir en la capa de datos de una aplicación empresarial.

Componentes de la persistencia en Java

JEE define cuatro módulos principales para el control de la persistencia en Java:

1. El API de persistencia de Jakarta, que está basado en objetos persistentes denominados **entidades JPA**. Estos objetos están relacionados con los elementos de una base de datos de la siguiente manera:
    a. Una tabla de BD se asocia a la clase Entity de JPA.
    b. Un registro de una tabla de BD se identifica como una instancia de una entidad JPA.
    c. Una columna de una tabla de BD es un atributo de una clase Entity
    d. Una relación es una instancia de otra clase con la cual está relacionada

    Además, se definen anotaciones que sirven para establecer mapeos entre tablas y entidades JPA.
2. Un lenguaje propio para realizar consultas a bases de datos denominado **JPQL** (Jakarta Persistence Query Language).

3. **El API de criterios de persistencia.** Vinculados al interfaz EntityManager, que incluye las operaciones de persistencia que pueden realizarse sobre las entidades JPA:
    a. Define su ciclo de vida.
    b. Gestiona la comunicación con la base de datos.

4. Mapeo de metadatos de un modelo relacional de base de datos a un modelo de objetos (ORM-Object/ Relational Mapping) mediante anotaciones o ficheros XML.

La idea de JPA es trabajar con objetos Java y no con código SQL, de tal manera que podamos enfocarnos en el código Java. JPA permite abstraer la comunicación con las bases de datos y crea un estándar para ejecutar consultas y manipular la información de una base de datos.

ENTIDADES Y PERSISTENCIAS CON JPA

Los objetos Entity de JPA

Una clase de entidad es un POJO y puede configurarse por medio de anotaciones. Las clases que definen objetos tipo Entity, son marcadas por la anotación @Entity y su anatomía se caracteriza por las siguientes características:

- El POJO marcado por la anotación **@Entity** tiene estructura de JavaBean: un constructor vacío declarado como public o protected, propiedades declaradas como private/protected y métodos getX() y setX() para acceder a ellas.

- Todos los atributos de la entidad serán persistentes y se corresponderán con las columnas de la tabla que mapea. Solo los marcados por la anotación **@Transient** no lo serán.

- La clase no podrá ser declarada como final. Los atributos persistentes no podrán ser declarados como static o final.

A continuación, se presenta un ejemplo de una clase de entidad con anotaciones:

```java
@Entity
public class Persona implements Serializable {

    @Id
    @GeneratedValue
    private Long personaId;
    @Column(nullable = false)
    private String nombre;
    @Column(nullable = false)
    private String apePaterno;
    @Column(nullable = false)
    private String apeMaterno;
    @Column(nullable = false)
    private String email;
    @Column(nullable = false)
    private Integer telefono;

    //Constructores, getters, setters
}
```

Anotaciones para mapeo de atributos

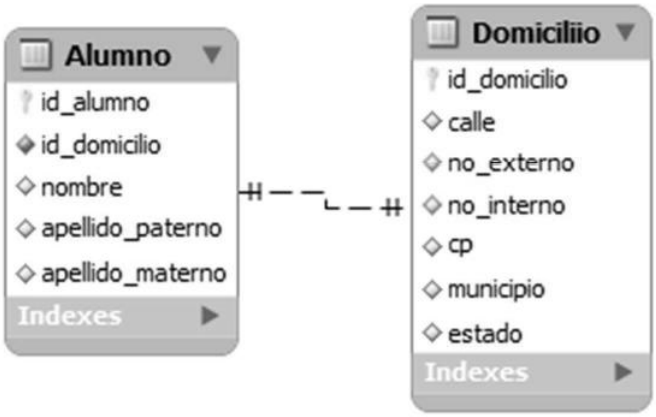
Se muestra a continuación una tabla que recoge las principales anotaciones que pueden utilizarse para configurar el mapeo de atributos entre tablas de bases de datos y propiedades del objeto Entity.

ANOTACIÓN	DESCRIPCIÓN
`@Column`	Especifica una columna de la tabla a mapear sobre un atributo de la clase `Entity`.
`@Transient`	Identifica a un atributo no persistente que no se mapeará en la base de datos.
`@Id`	Marca el atributo como identificador (llave primaria). Según JPA, es obligatorio que exista un atributo de este tipo en cada clase.
`@JoinColumn`	Indica un atributo que actúa como llave foránea en otra entidad. Algunos de sus atributos son los siguientes: `name:` Identifica a la columna que guarda la clave foránea. Es el único campo obligatorio. `referenced:` nombre de la tabla relacionada que contiene la columna.

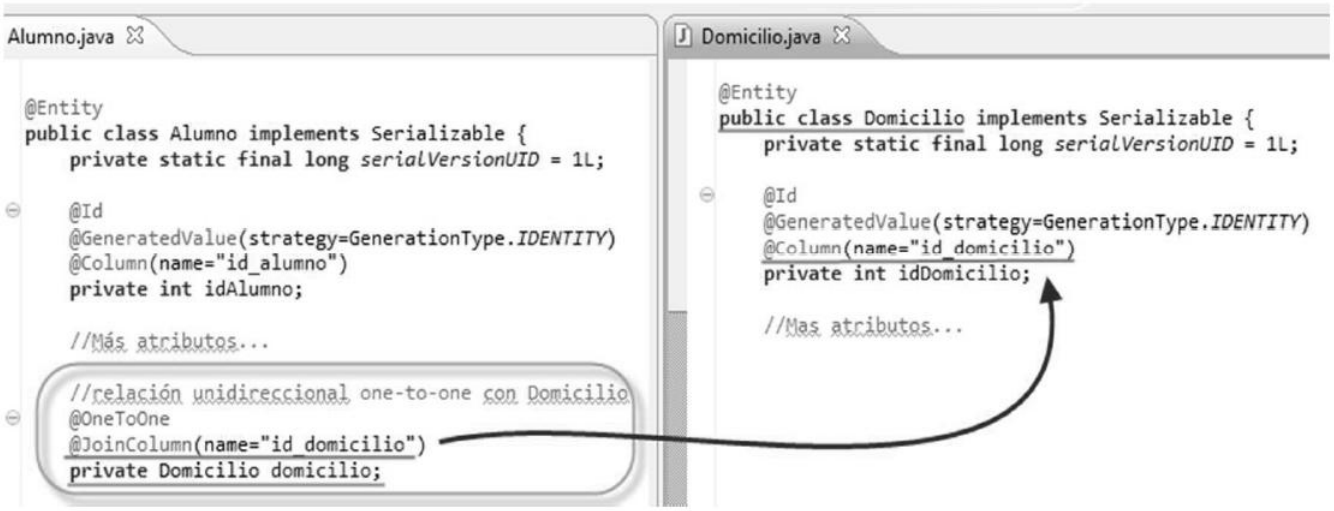
Relaciones entre entidades

Al igual que sucede entre las tablas de una base de datos relacional, es posible establecer relaciones entre diferentes objetos tipo Entity. Estas relaciones implican cardinalidad, es decir el número de referencias que tiene un objeto en relación con otro. Serán especificadas a través de las anotaciones:

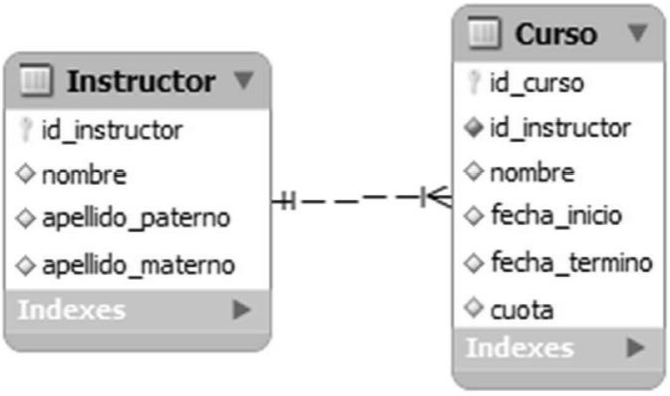
- **@OneToOne:** cada lado de la relación tiene como máximo una referencia al otro objeto.



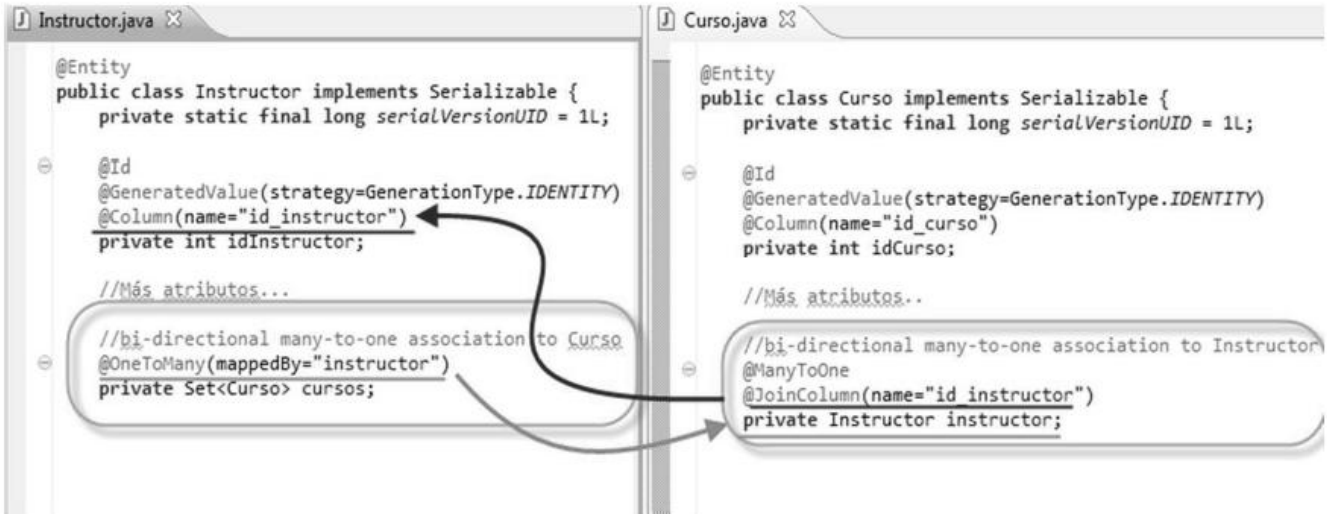
En la figura anterior podemos observar una relación de 1 a 1, en la cual una entidad Alumno tiene una relación con sólo un domicilio, y viceversa, esto es, la cardinalidad entre las entidades es de 1 a 1.
Podemos observar que la clase de Alumno es la que guarda la referencia de un objeto Domicilio, para mantener una navegabilidad unidireccional y que a partir de un objeto Alumno podamos recuperar el objeto Domicilio asociado.
Es importante destacar que el manejo de relaciones es por medio de objetos, y no atributos aislados, esto nos permitirá ejecutar queries con JPQL que recuperen objetos completos.



- **@OneToMany y @ManyToOne:** implican que una instancia de un objeto Entity puede estar relacionado con varias instancias de otro (y viceversa).



Cuando un objeto de Entidad está asociado con una colección de otros objetos de Entidad, es más común representar esta relación como una relación de Uno a Muchos. Por ejemplo, en la figura anterior podemos observar la relación de un Instructor el cual puede tener asignados varios Cursos (relación de 1 a muchos).
Si queremos saber desde la clase **Curso** qué instructor tiene asociado, deberemos agregar el mapeo bidireccional (ManyToOne) hacia Instructor. Y en la clase Instructor, se especifica una relación uno a muchos (OneToMany) hacia una colección de objetos de tipo Curso, el cual puede ser una estructura de datos Set o un List.



Métodos para gestionar el ciclo de vida de las entidades

- **public void persist (Object entity):** este método graba una entidad en la base de datos y la marca como gestionada. Bajo este método se configuran las sentencias INSERT que van a modificar los datos en la BD, pero esto no significa que sean ejecutados automáticamente. Lo normal es que no se ejecuten hasta que terminen de configurarse todas las sentencias que forman parte de una transacción.

- **public <T> T merge (T entity):** mezcla una entidad con el contexto de persistencia del EntityManager, devuelve el resultado de la mezcla y actualiza el estado de la entidad en la base de datos.

- **public void remove(Object entity):** elimina una entidad de la base de datos.

- **public <T> T find(Class<T> entityClass, Object primarykey):** localiza una instancia de una entidad a través de su clave primaria.

- **public void flush():** sincroniza el estado de las entidades en el contexto de persistencia del interfaz EntityManager con la base de datos.

- **public void refresh(Object entity):** refresca la entidad desde la base de datos.

- **public Query createQuery(String jpqlString):** crea una consulta query dinámica utilizando JPQL.

- **public Query createNativeQuery(String jqlString):** crea una query dinámica utilizando SQL nativo.

- **public Query createNamedQuery(String name):** crea una instancia predefinida para establecer consultas.

JAKARTA PERSISTENCE QUERY LANGUAGE (JPQL)

JPQL es un lenguaje de consulta similar a SQL. **La principal diferencia es que SQL funciona con esquemas, tablas y procedimientos almacenados del DBMS pero JPQL los manipula con objetos de Java y sus respectivos atributos.**
El origen de este lenguaje está en la necesidad de estandarizar un lenguaje de consultas aplicable al esquema de abstracción de datos y persistencia definido a través de objetos tipo Entity. Al ser un lenguaje de expresiones, éstas pueden compilarse para convertirse en lenguaje de consultas SQL o a cualquier otro.
Tipos de Queries:

- **Dynamic queries:** Consultas que reciben parámetros en tiempo de ejecución.
- **Named queries:** Consultas ya creadas previamente, y que se pueden ejecutar solo utilizando el nombre.
- **Native queries:** Consultas con SQL nativa, ya que hay casos de uso que lo requieren.

Sintaxis de JPQL
Al igual que SQL, JPQL basa su capacidad de acción en cuatro tipos de sentencias: SELECT, INSERT, UPDATE, DELETE. Las operaciones de inserción se pueden resolver de forma casi completa con el método persist() del Entity Manager. A continuación, se va a exponer la sintaxis del lenguaje en el uso de cada una de las sentencias restantes.

SENTENCIA SELECT

SELECT puede contener a su vez algunos tipos de modificadores o clausulas:

- **SELECT**, para especificar el tipo de objetos o valores que deben ser seleccionados.
- **FROM**, para especificar el dominio de datos al que se refiere la consulta.
- **WHERE**, modificador opcional, que puede ser utilizado para restringir los resultados devueltos por la consulta.
- **GROUP BY**, modificador opcional, que permite agrupar los resultados.
- **HAVING**, modificador opcional, que permite incluir condiciones de filtrado.
- **ORDER BY**, modificador opcional, que permite ordenar los resultados devueltos por la consulta.

A continuación, se incluyen algunos ejemplos sencillos de consultas:

- `SELECT m FROM MatriculasEntity m`

  Esta consulta devuelve todas las matrículas, identificadas de forma general por la variable m.

- `SELECT DISTINCT a FROM AlumnosEntity a`

  El modificador DISTINCT sirve para eliminar de la respuesta valores duplicados, en el hipotético caso de que existan.

- `SELECT DISTINCT a FROM AlumnosEntity WHERE a.nombre = :nombre AND a.apellido1 :=apellido1`

  En este caso se devolverán todos los elementos no duplicados de tipo entidad AlumnosEntity cuyo nombre y primer apellido coincida con el especificado como parámetro.

- `SELECT a FROM AlumnosEntity a WHERE a.nombre LIKE ‘Mari%’`

  En este caso se devolverán todos los elementos no duplicados de tipo entidad AlumnosEntity cuyo nombre comience por “Mari”. De esta forma la partícula LIKE sirve para introducir patrones de texto.

- `SELECT COUNT(p) FROM Pelicula p`

  COUNT() es una función agregada de JPQL, cuya misión es devolver el número de ocurrencias tras realizar una consulta. Por tanto, en el ejemplo anterior, el valor devuelto por la función agregada es el resultado de la sentencia al completo. Otras funciones agregadas son AVG para obtener la media aritmética, MAX para obtener el valor máximo, MIN para obtener el valor mínimo, y SUM para obtener la suma de todos los valores.

- `SELECT p FROM Pelicula p WHERE p.duracion < 120 AND NOT (p.genero = 'Terror')`

  Gracias a la sentencia anterior se obtendrían todas las instancias de Pelicula con una duración menor a 120 minutos que no (NOT) son del género Terror.

- `SELECT p FROM Pelicula p WHERE p.duracion BETWEEN 90 AND 150`

  La sentencia anterior obtiene todas las instancias de Pelicula con una duración entre (BETWEEN) 90 y (AND) 150 minutos. BETWEEN puede ser convertido en NOT BETWEEN, en cuyo caso se obtendrían todas las películas que una duración que no (NOT) se encuentren dentro del margen (BETWEEN) 90-150 minutos.
  `SELECT p FROM Pelicula p WHERE p.titulo LIKE 'El%'`
  La sentencia anterior obtiene todas las instancias de Pelicula cuyo título sea como (LIKE) *El%* (el símbolo de porcentaje es un comodín que indica que en su lugar puede haber entre cero y más caracteres). Resultados devueltos por esta consulta incluirían películas con un título como *El Renacido*, *El Pianista*, o si existe, *El*. El otro comodín aceptado por LIKE es el carácter "guion bajo" (_), el cual representa un único carácter indefinido (ni cero caracteres ni más de uno; uno y solo uno).

Existen otras cláusulas que pueden ser utilizadas en las consultas como son:

- **NULL:** condición de nulidad en consultas, por ejemplo, para devolver relaciones que no existen entre dos entidades dadas.
- **EMPTY:** es una expresión condicional para indicar que un elemento está vacío.
- **BETWEEN-AND:** sirve para imponer condiciones en las consultas en un rango de valores.
- **Operadores de comparación <, >, =:** sirven para establecer comparaciones con el modificador WHERE.

Además, es posible utilizar expresiones funcionales como las siguientes:

- **CONCAT(String, String):** devuelve un String que es la concatenación de los dos pasados como parámetro.
- **LENGTH(String):** devuelve el entero que indica la longitud del String pasado como parámetro.
- **LOCATE(String, String [, start]):** devuelve un entero con la posición de un determinado String dentro de otro String. Si no se localiza, se devuelve un 0.
- **SUBSTRING(String, start, length):** devuelve un String que es un subconjunto del pasado como parámetro comenzando en la posición marcada por start y de longitud length.
- **TRIM([[LEADING|TRAILING|BOTH] char) FROM] (String):** esta función elimina un determinado carácter desde el comienzo o final de un String. Si no ese especifica ningún carácter, se eliminan espacios en blanco.
- **LOWER(String):** convierte un String al equivalente en minúsculas.
- **UPPER(String):** convierte un String al equivalente en mayúsculas.

Y otras como:

- **Expresiones aritméticas:** `ABS(number)`, `MOD(int, int)`, `SQRT(double)` y `SIZE(Collection)`.

- **Expresiones horarias:** `CURRENT_DATE`, `CURRENT_TIME` y `CURRENT_TIMESTAMP`.

SENTENCIA UPDATE

Permite actualizar valores almacenados por la entidad. Puede incluir de forma opcional el modificador WHERE. Se ilustra con un ejemplo:

```
UPDATE CursosEntity c SET c.nota = 8.5 WHERE c.nombre = 'JavaEE'
OR c.nombre = 'JavaSE'
```

SENTENCIA DELETE

Permite borrar un registro completo correspondiente a una entidad. Al igual que UPDATE, puede incluir de forma opcional el modificador WHERE. Algunos ejemplos a continuación:

- `DELETE FROM CursosEntity c WHERE c.nota = 9`
- `DELETE FROM CursosEntity c WHERE c.nombre IS EMPTY`

Uso del JPQL

Las consultas expresadas en JPQL serán ejecutadas a través del interfaz EntityManager y los métodos createX como los siguientes:

- `Query createQuery(String qlString):` este método se utiliza para crear consultas dinámicas que toman valor en ejecución.
- `Query createNamedQuery(String name):` este método se utiliza para crear consultas estáticas que están codificadas previamente o predefinidas como metadatos a través del objeto `jakarta.persistence.NamedQuery` que las marca con la anotación `@NamedQuery.`

El interfaz proporciona además otro método: `Query createNativeQuery(String sqlString)` que permite ejecutar directamente sentencias SQL nativas.

Documentación recomendada para revisar:

- https://github.com/spring-projects/spring-framework
- https://docs.spring.io/spring-framework/reference/

(Spring Academy, 2024)

III. PROCEDIMIENTO

Para el desarrollo de esta guía únicamente vamos a utilizar nuestro IDE: IntelliJ idea en su versión Community, además, ahora le daremos un mayor significado a la base de datos en memoria H2.

Ejemplos

Manipulando entidades con JPA y SpringBoot;

Creación del proyecto.

Paso 1. Abrimos nuestro navegador de preferencia y nos dirigimos a la siguiente URL: https://start.spring.io/.

NOTA: Si usted posee la versión Ultimate del IDE, directamente proceda con la opción de Spring Boot, que se encuentra en el menú para crear proyectos nuevos y no olvide agregar la dependencia de Lombok.

Paso 2. Seleccionamos la siguiente configuración para crear nuestro proyecto con spring boot **(OJO: A este punto todavía, no presione el botón de generar proyecto)**.

- Project: Maven
- Language: Java
- Spring Boot: 3.3.8
- Group: sv.edu.udb
- Artifact: spring-database-jpa
- Name: spring-database-jpa
- Description: Ejercicio con Jpa
- Package name: sv.edu.udb
- Packaing: Jar
- Java: 17

Paso 3. Agregamos siempre la dependencia de Lombok, por lo cual haremos lo siguiente, presionamos el botón que dice **Add Dependencies** y vamos a seleccionar la dependencia que se llama **Lombok**.

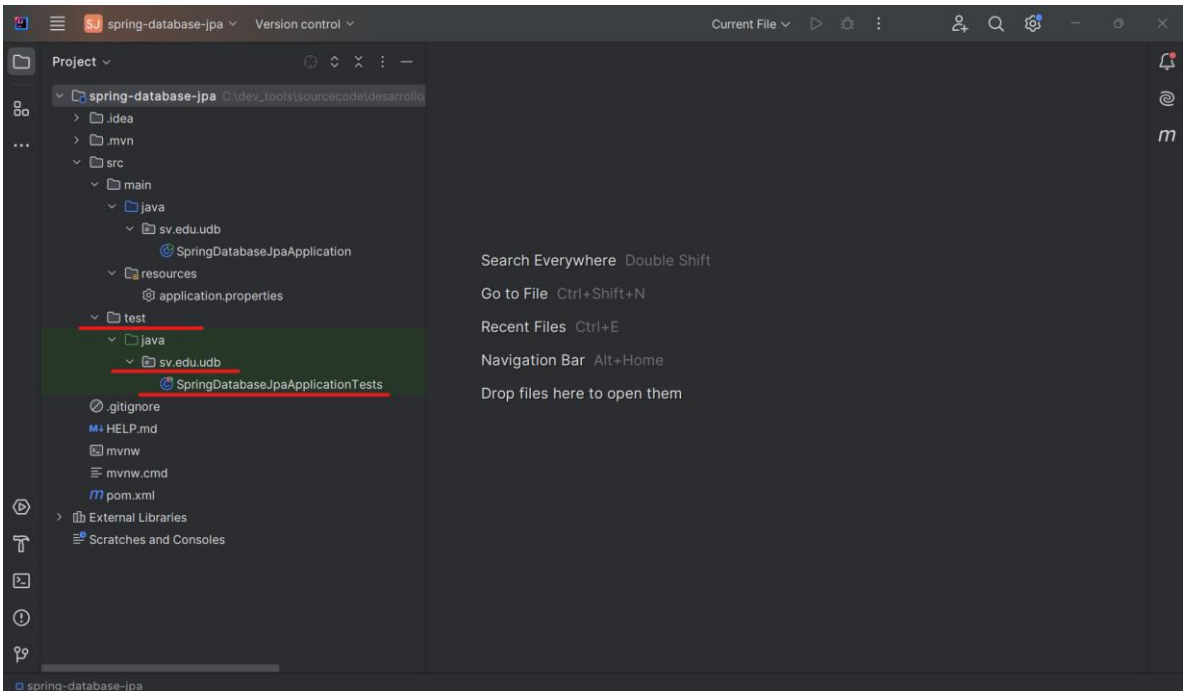
NOTA: Lombok es una librería que se utiliza para la generación de código, la cual nos ayuda a reducir la escritura de código boilerplate, es decir, la generación de setter y getter en nuestras clases, pueden revisarla en el siguiente enlace: https://projectlombok.org/

Paso 4. Una vez seleccionada, procedemos a crear el proyecto y descargarlo, dando click en el botón de **Generate**.

NOTA: Lo pueden guardar en el directorio de su preferencia.

Paso 5. Una vez descargado lo descomprimimos y lo importamos a nuestro IDE.

NOTA: **Borramos** la clase del paquete de testing, ya que no la estaremos utilizando.



Paso 6. Ya importado el proyecto al IDE procedemos a agregar las dependencias que estaremos utilizando a nuestro archivo pom.xml y además no olvide configurar la versión de Lombok para el pluguin de **maven-compiler-plugin**.

<version>${lombok.version}</version>

Quedando así:

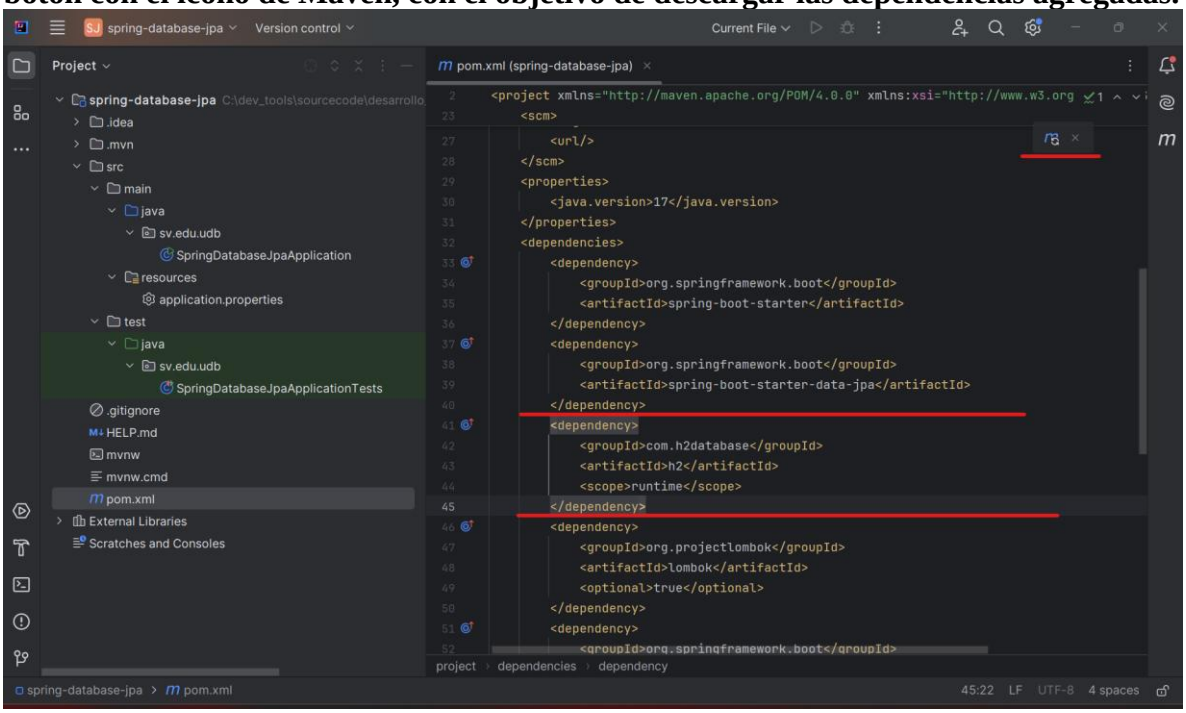
```
<plugin>
                    <groupId>org.apache.maven.plugins</groupId>
                    <artifactId>maven-compiler-plugin</artifactId>
                    <configuration>
                        <annotationProcessorPaths>
                            <path>

    <groupId>org.projectlombok</groupId>
                                        <artifactId>lombok</artifactId>

    <version>${lombok.version}</version>
                                    </path>
                                </annotationProcessorPaths>
                            </configuration>
                        </plugin>
```

Paso 6.1 Ya configurada la versión de Lombok en el pluguin procedemos a agregar las dependencias que estaremos utilizando a nuestro archivo pom.xml.

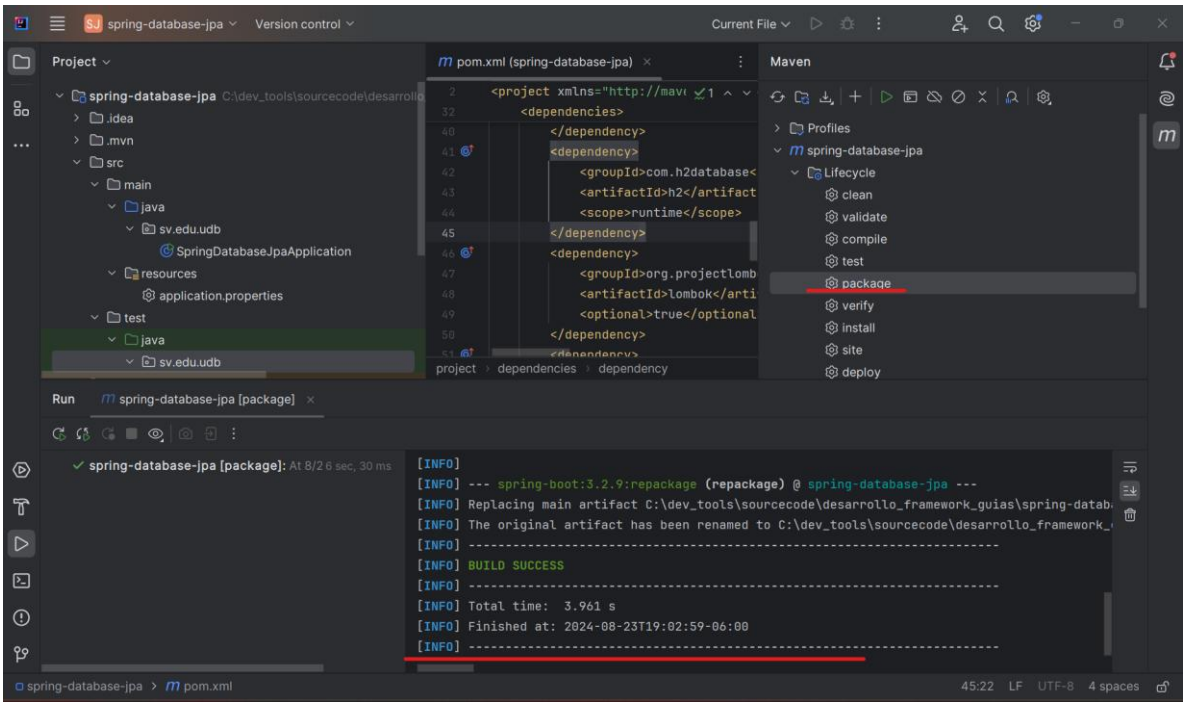
pom.xml

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

<dependency>
   <groupId>com.h2database</groupId>
   <artifactId>h2</artifactId>
   <scope>runtime</scope>
</dependency>
```

NOTA: Después de agregadas las dependencias al archivo pom.xml, presionar el botón con el icono de Maven, con el objetivo de descargar las dependencias agregadas.



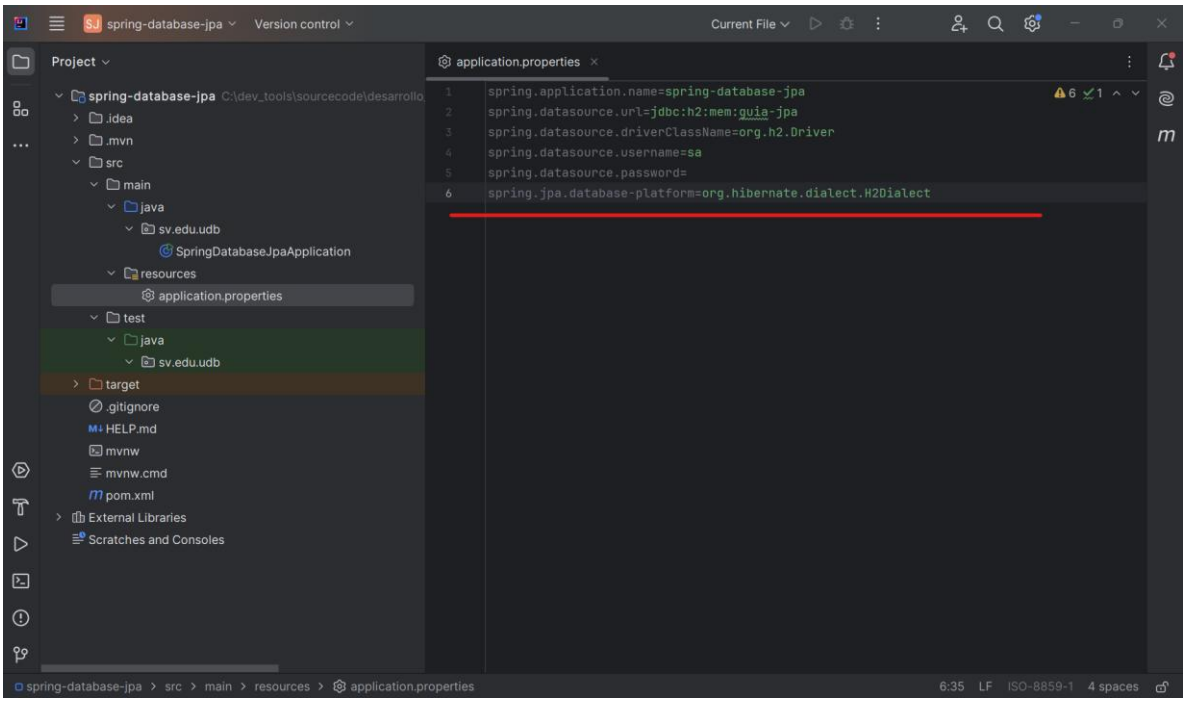
Paso 7. Procedemos a correr los comandos de **clean** y luego el de **package** de Maven, para verificar que todo está bien con el proyecto.



Paso 8. Ahora que estamos listos con el proyecto, empezamos configurando la conexión a la base de datos a través del archivo **application.properties** que nos provee el framework de spring boot.

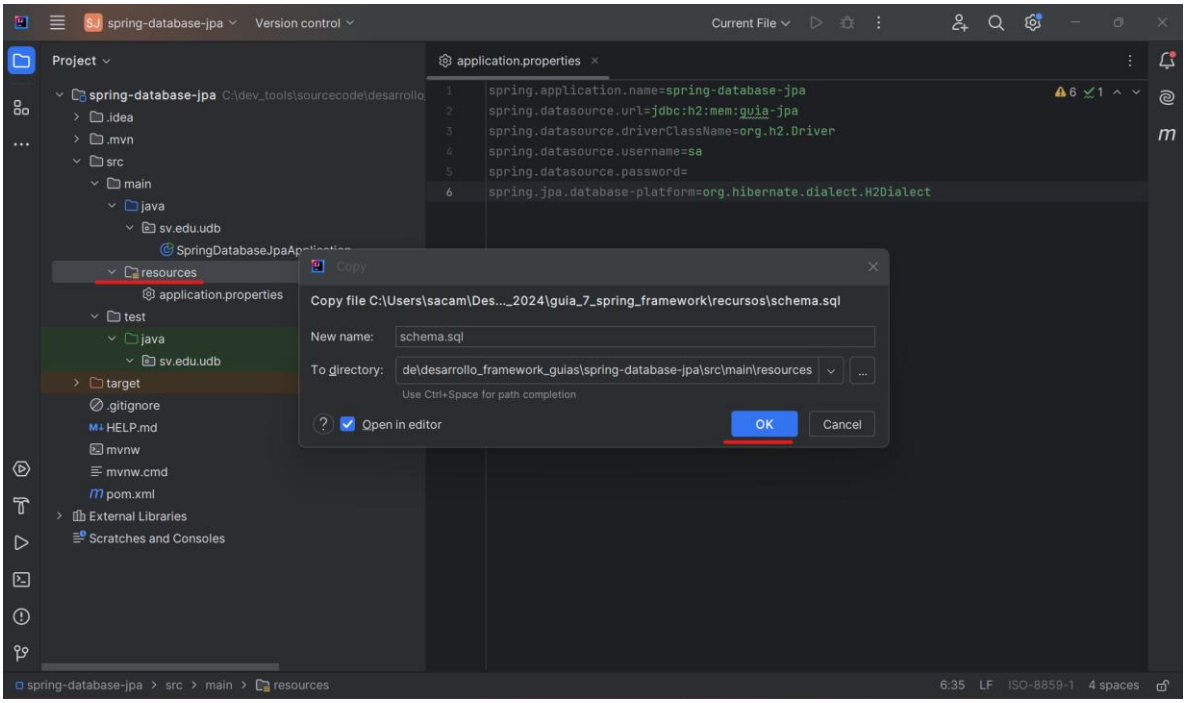
application.properties

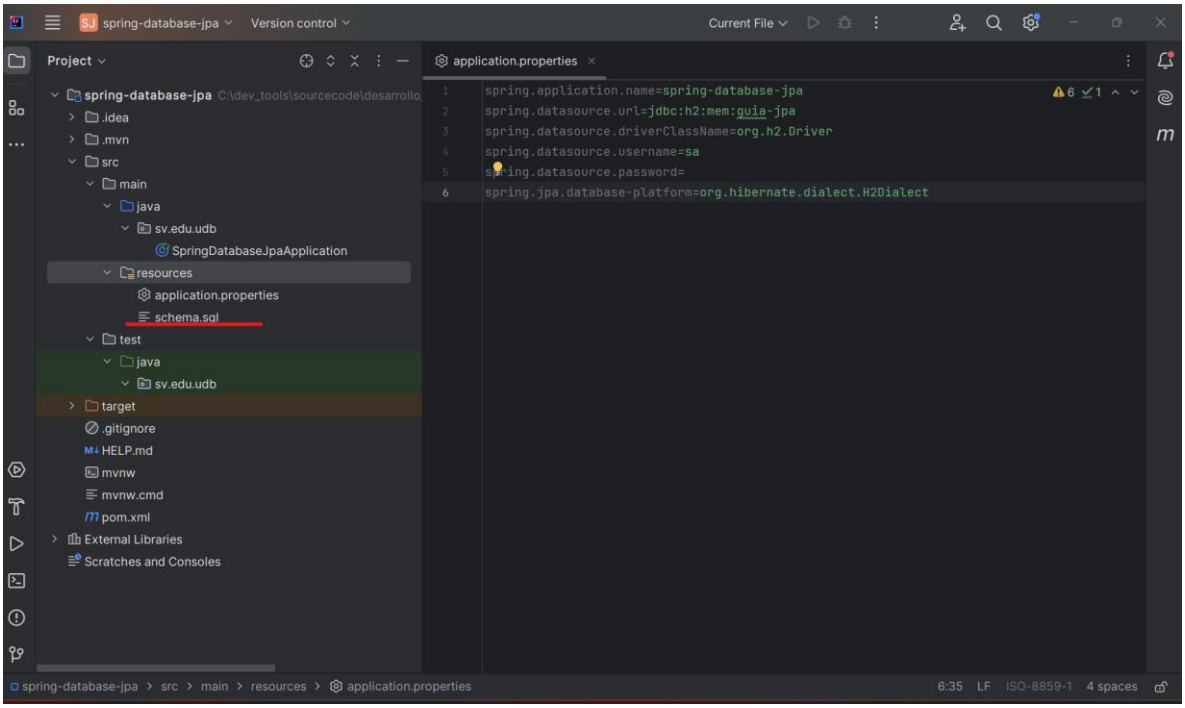
```
spring.datasource.url=jdbc:h2:mem:guia-jpa
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.jpa.hibernate.ddl-auto=none
```



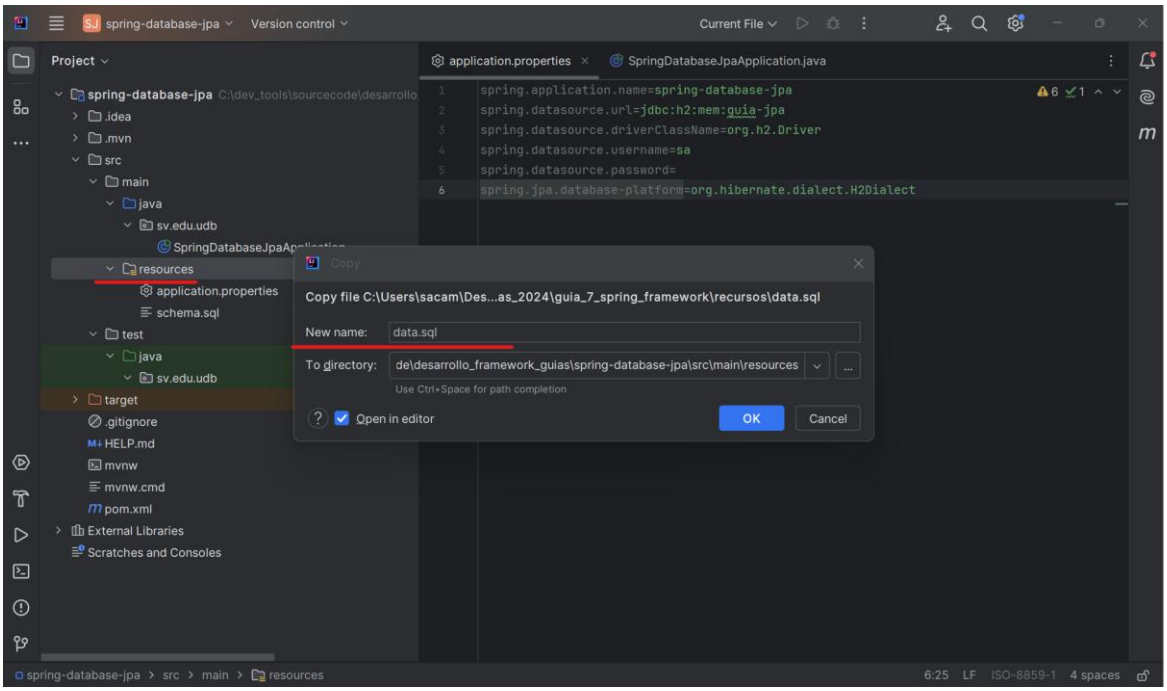
NOTA: Podemos verificar todas las configuraciones que estan disponibles para realizar desde el archivo application.properties en la documentacion oficial de spring boot: https://docs.spring.io/spring-boot/appendix/application-properties/index.html

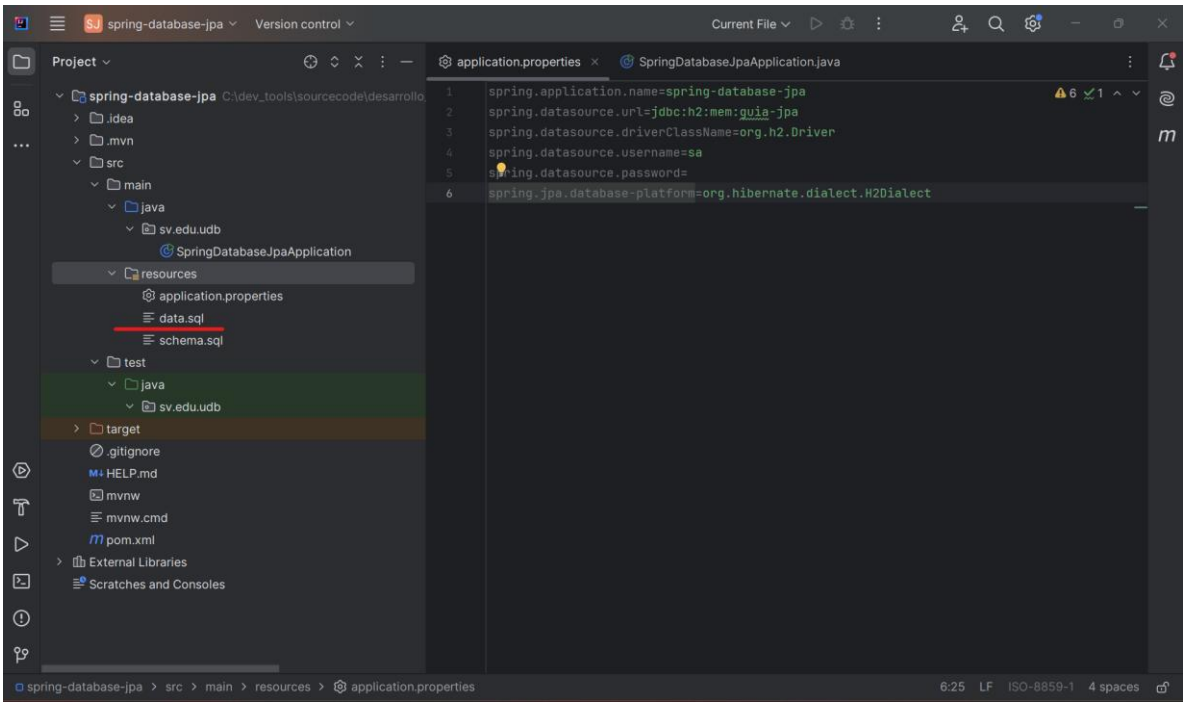
Paso 9. Una vez las configuraciones iniciales estén en el archivo application.properties, procedemos a copiar y pegar el archivo **schema.sql** proporcionado en los recursos de la guía a la siguiente carpeta dentro del proyecto: **src/main/resources.**





Paso 10. Hacemos lo mismo para el archivo **data.sql** proporcionado en los recursos de la guía, y siempre guardamos dentro de la carpeta del proyecto: **src/main/resources.**



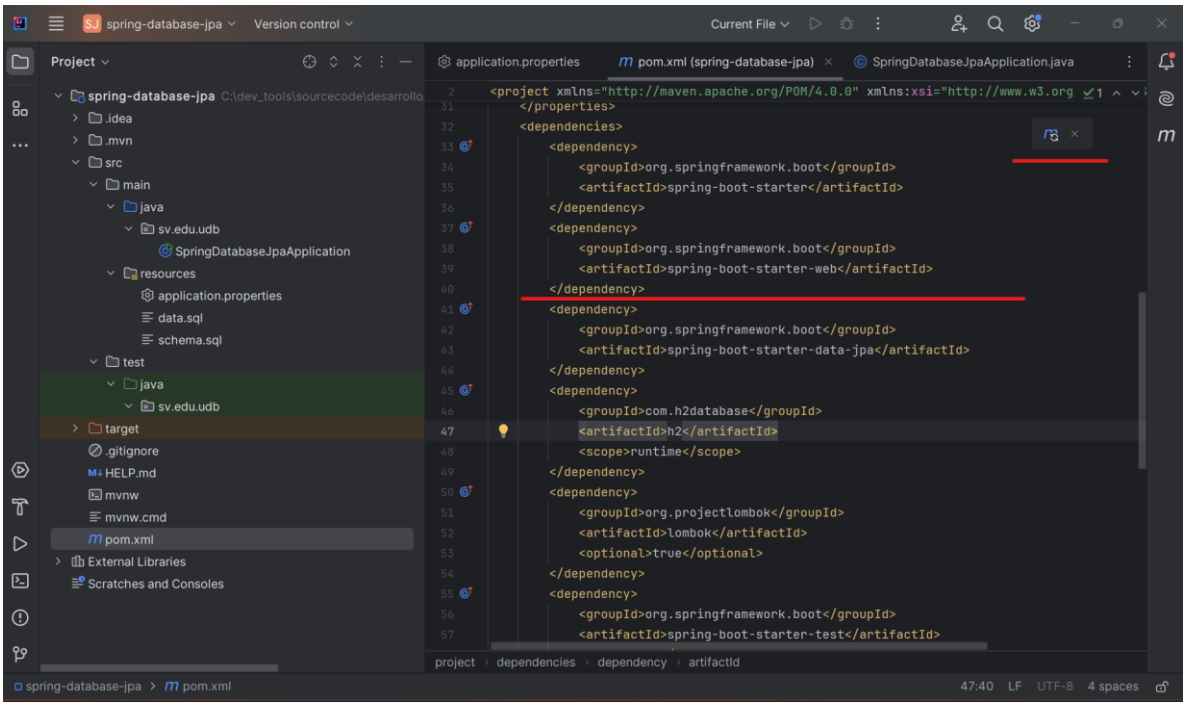


NOTA: Para la inicialización del esquema y la data en una base de datos, este método es bastante conveniente, debido a que el framework se encarga de correr ambos archivos directamente en la base de datos, esto es recomendable en entornos de desarrollo y pruebas.

Paso 11. Para poder visualizar la base de datos y las tablas necesitamos agregar una dependencia en el archivo **pom.xml** para poder habilitar el servidor web en nuestro proyecto.

pom.xml

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



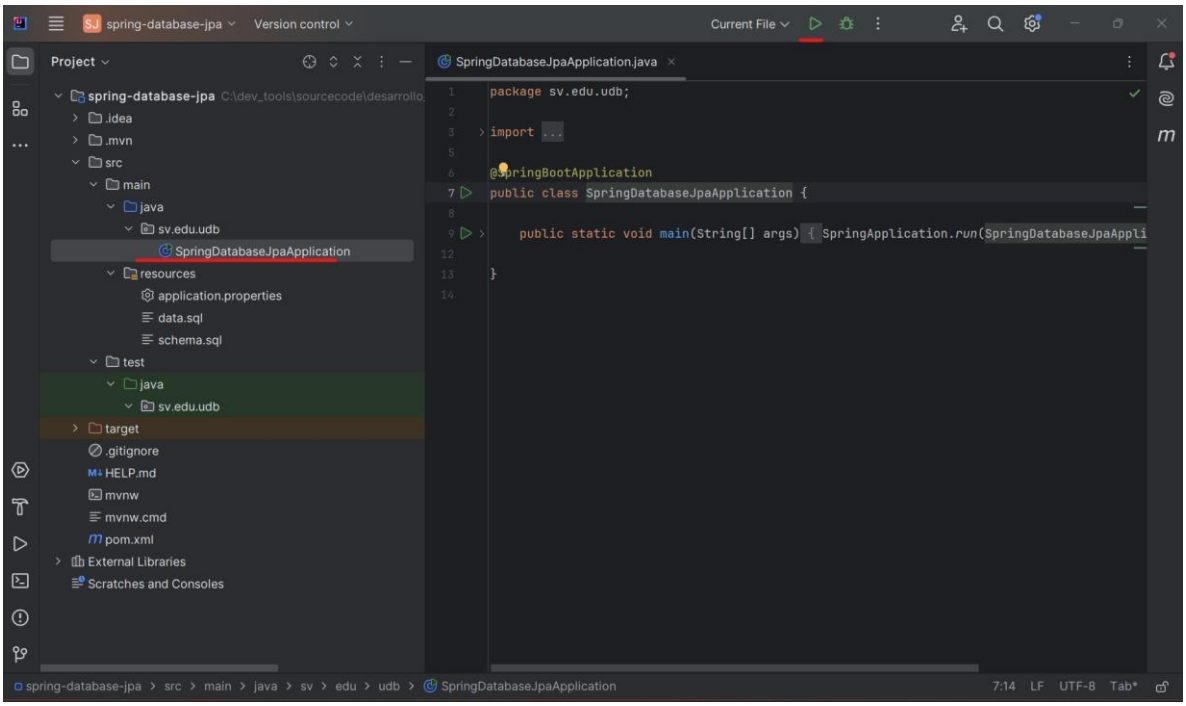
Paso 12. Una vez agregada la dependencia y configurados ambos archivos procedemos a activar la consola que H2 nos provee para poder visualizar las tablas y la información, esto se realiza a través de una configuración del framework**.** Volvemos nuevamente al archivo **application.properties** y al final agregamos la siguientes propiedades.

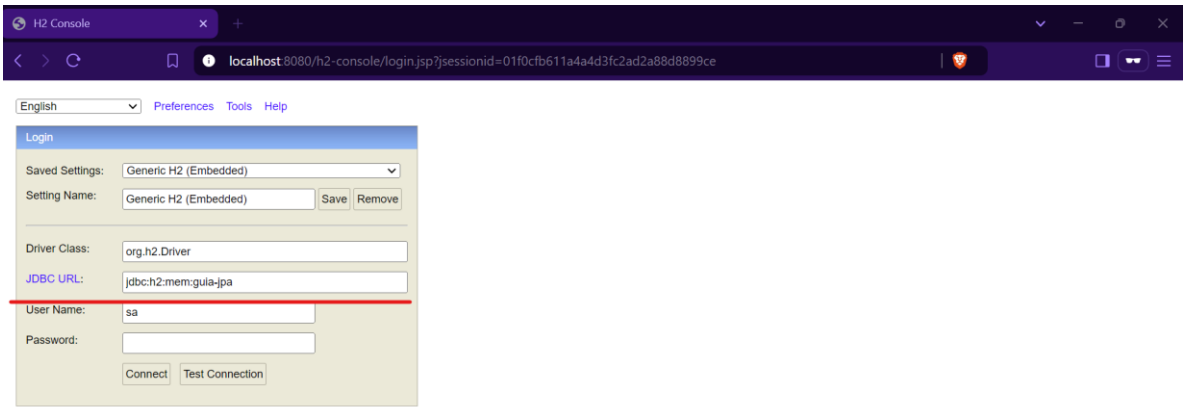
application.properties

```
spring.datasource.url=jdbc:h2:mem:guia-jpa
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.jpa.hibernate.ddl-auto=none
```

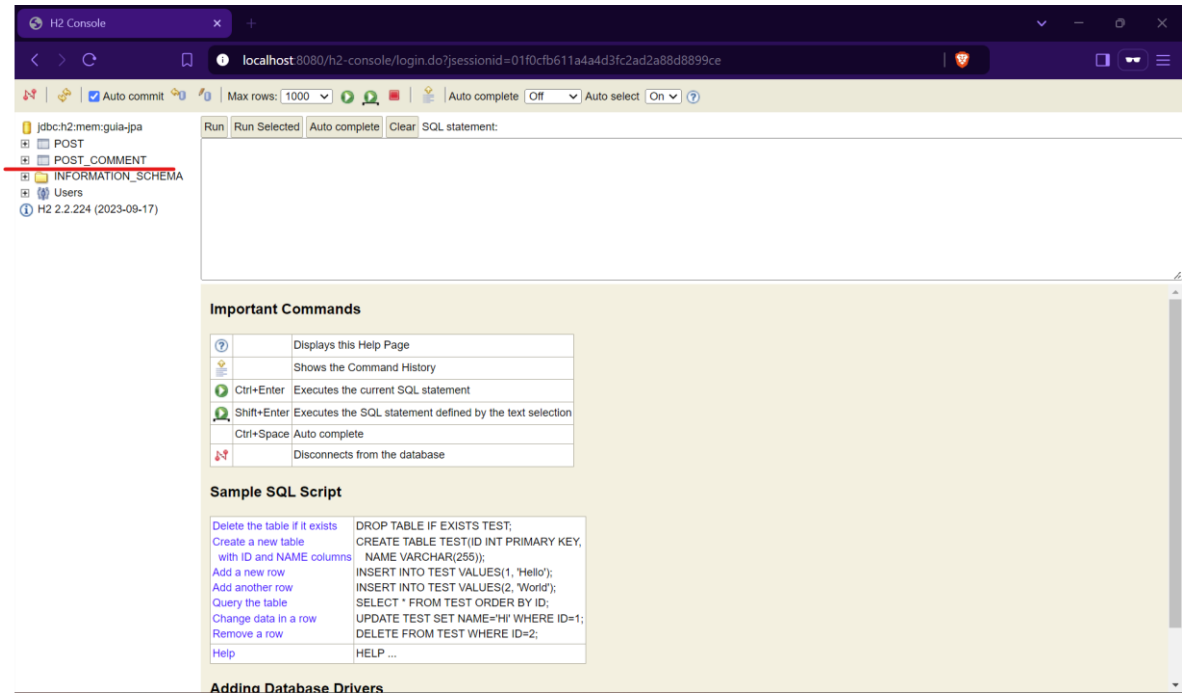
```
##H2 Configurations
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console
spring.h2.console.settings.trace=false
spring.h2.console.settings.web-allow-others=false
```

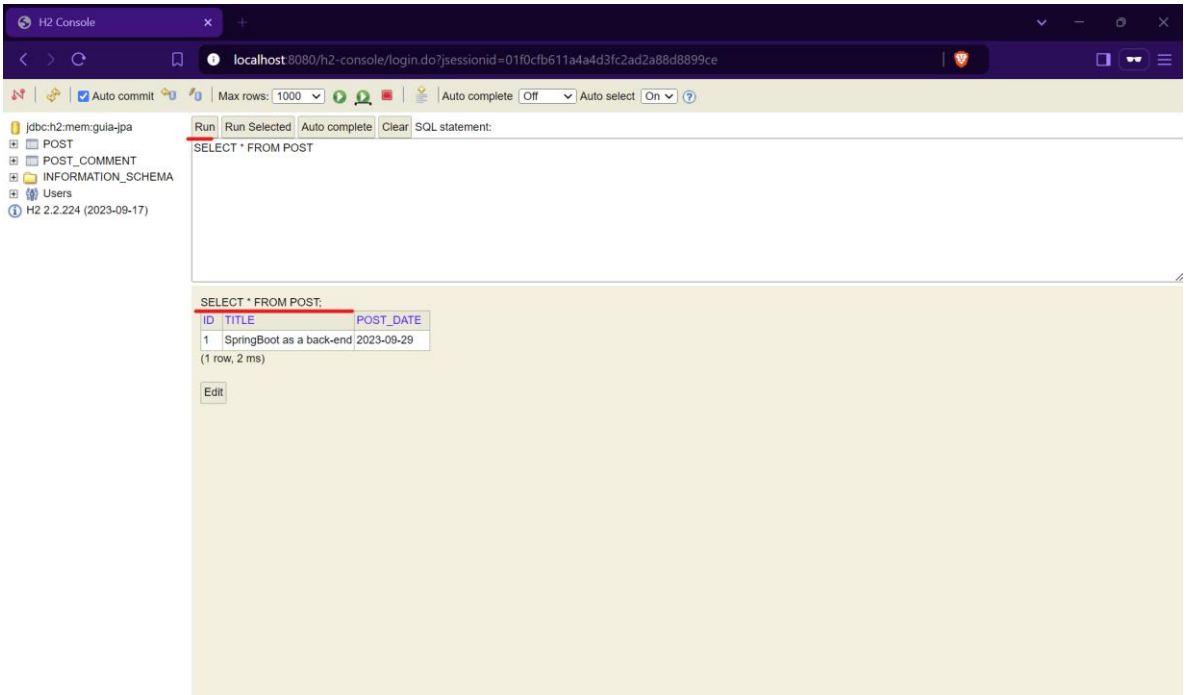
Paso 13. Corremos el proyecto y abrimos el navegador y accedemos a la siguiente dirección: **http://localhost:8080/h2-console**



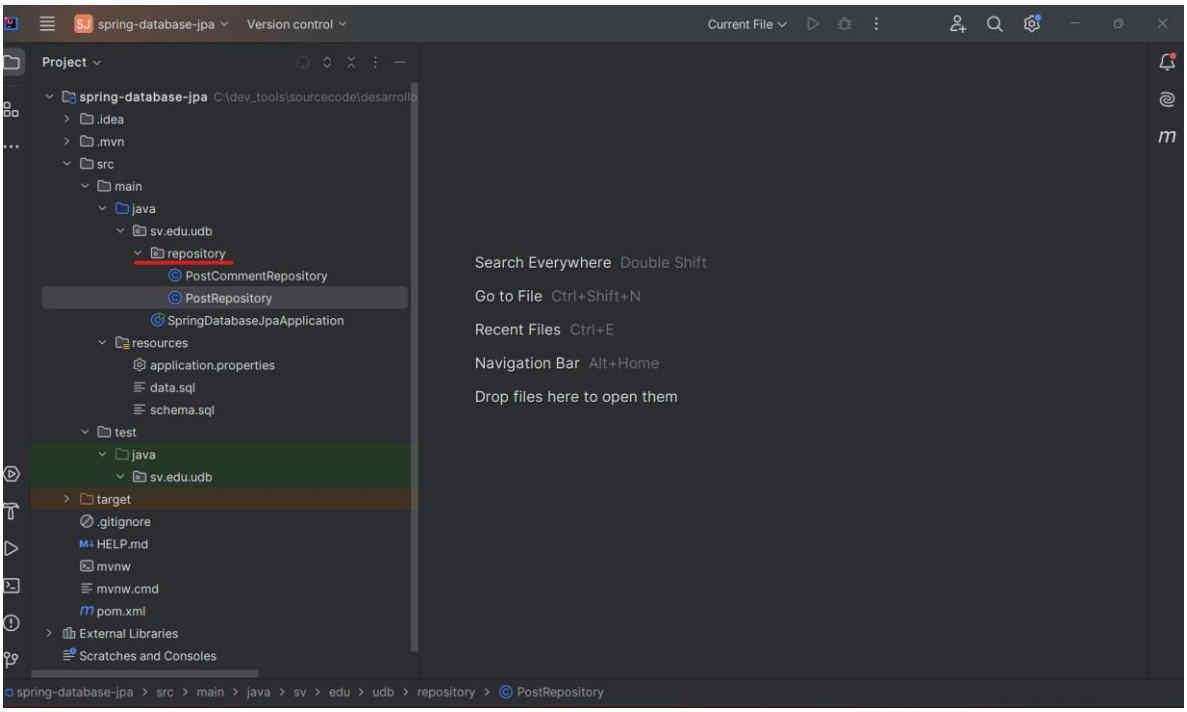


NOTA: Favor verificar el campo de JDBC URL, sino tiene la dirección que esta en el application.properties, favor modificarlo y proceder a darle click al botón de connect.

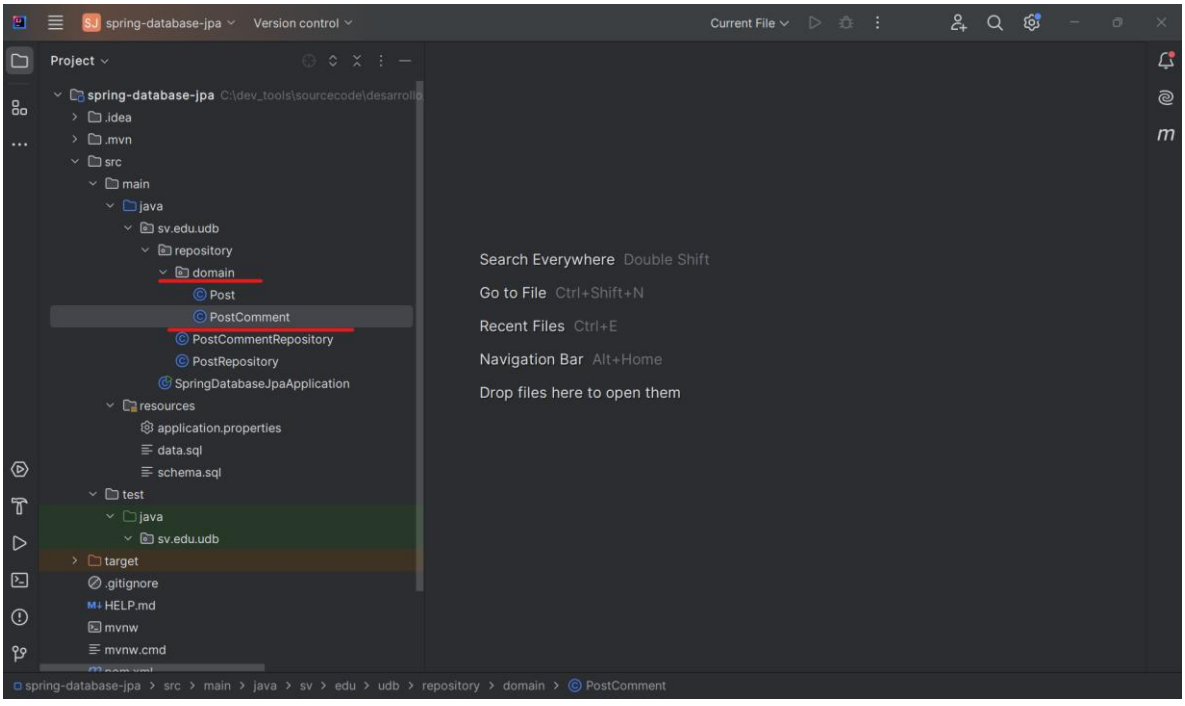




Paso 14. Detenemos el proyecto, y ahora pasamos a definir nuestras primeras clases, a las cuales se les conoce como **repositorios (Sinónimos de DAO [Data Access Object])**, para ello creamos la capa de **repository** y luego dentro de ella creamos dos clases, **PostRepository.java** y **PostCommentRepository.java, las cuales por el momento no contendrán código.**



Paso 15. Ahora creamos el paquete denominado **domain** dentro del paquete **repository**, el cual va a contener nuestras primeras clases correspondientes a las **entidades del negocio, las cuales son Post.java y PostComment.java**.



Paso 16. Procedemos a digitar las entidades de **Post.java y PostComment.java**.

Post.java

```
package sv.edu.udb.repository.domain;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

import java.time.LocalDate;

@Getter
@Setter
@Entity //Anotacion para marcar que es una entidad de negocio
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class Post {

    @Id //definicion del id (es una anotacion obligatoria si se usa entity)
    private Long id;

    @Column(nullable = false)
    private String title;

    @Column(nullable = false)
    private LocalDate postDate;
}
```

PostComment.java

```
package sv.edu.udb.repository.domain;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.FetchType;
import jakarta.persistence.Id;
import jakarta.persistence.JoinColumn;
import jakarta.persistence.ManyToOne;
import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;
```

```
import java.time.LocalDate;

@Getter
@Setter
@Entity
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class PostComment {

    @Id
    private Long id;

    @Column(nullable = false)
    private String review;

    @Column(nullable = false)
    private LocalDate commentDate;

    @ManyToOne(fetch = FetchType.LAZY) //Relacion de Muchos a uno
    @JoinColumn(name = "post_id") //Definimos el nombre de la llave foranea
    private Post post;
}
```

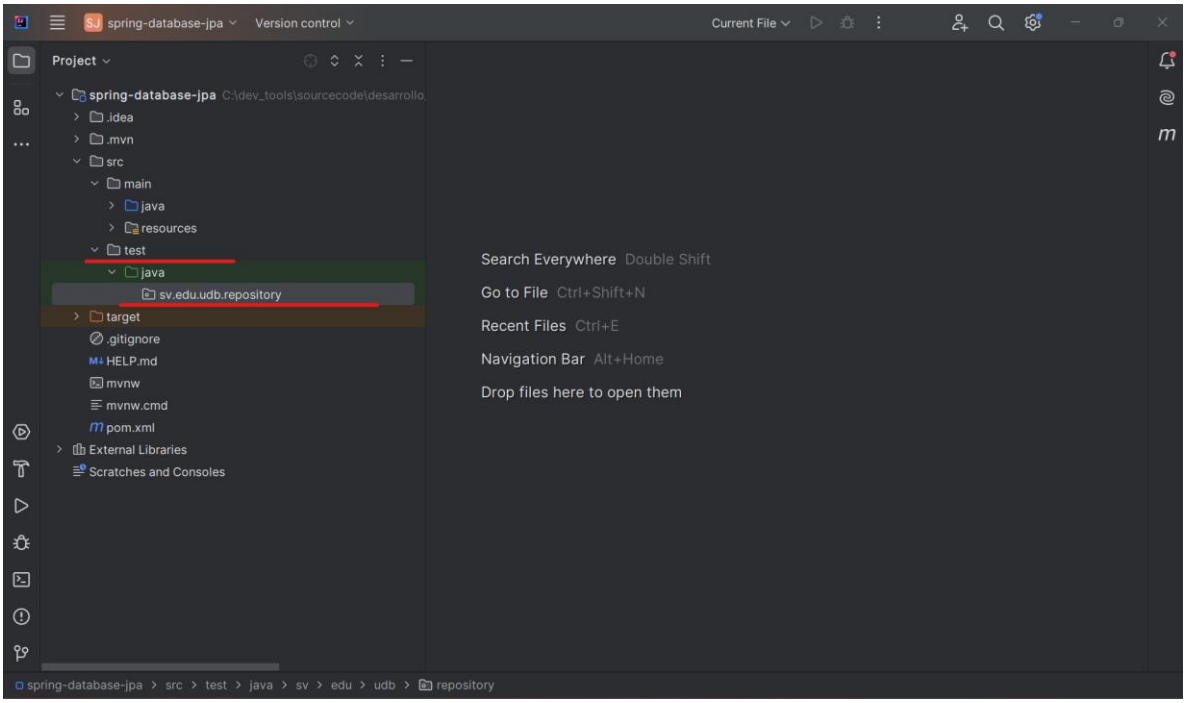
Paso 17. Procedemos ahora digitar las entidades de **PostRepository.java**.

PostRepository.java

```
package sv.edu.udb.repository;

import jakarta.persistence.EntityManager;
import jakarta.persistence.PersistenceContext;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;
import sv.edu.udb.repository.domain.Post;

import java.util.List;

@Repository
public class PostRepository {

    @PersistenceContext
    private EntityManager entityManager;

    public List<Post> findAll() {
        final String QUERY = "select p from Post p";
```

```
        return entityManager
                .createQuery(QUERY, Post.class)
                .getResultList();
    }

    public Post findById(final Long id) {
        return entityManager.find(Post.class, id);
    }

    @Transactional
    public void save(final Post post) {
        entityManager.persist(post);
    }

    public void delete(final Post post) {
        entityManager.remove(post);
    }
}
```

Paso 18. Una vez terminado de digitar, nos movemos al folder de **test** y agregamos el paquete de **repository**.



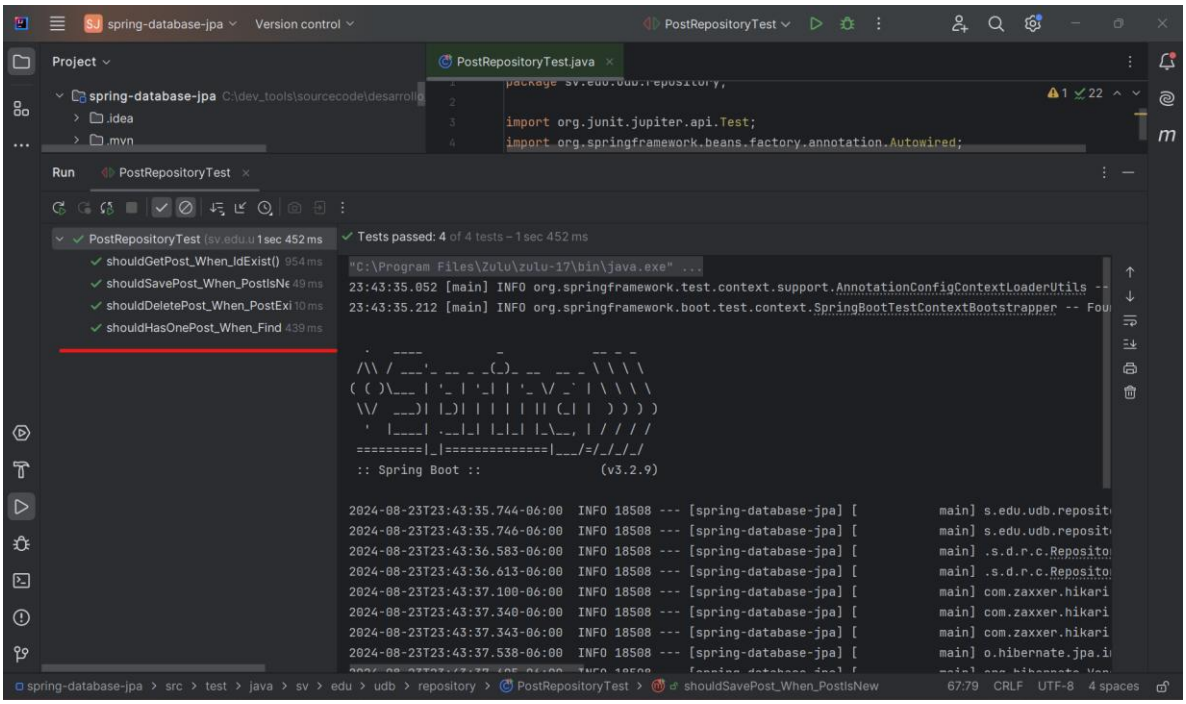
Paso 19. Una vez agregado el paquete, ahora vamos a crear la primera clase de **test**, con respecto al **PostRepository.java**.

PostRepositoryTest.java

```java
package sv.edu.udb.repository;

import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.transaction.annotation.Transactional;
import sv.edu.udb.repository.domain.Post;

import java.time.LocalDate;
import java.util.List;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertNotNull;
import static org.junit.jupiter.api.Assertions.assertNull;

@SpringBootTest
class PostRepositoryTest {

    @Autowired
    private PostRepository postRepository;

    @Test
    void shouldHasOnePost_When_FindAll() {
        int expectedPostNumber = 1;
        final List<Post> actualPostList = postRepository.findAll();
        assertNotNull(actualPostList);
        assertEquals(expectedPostNumber, actualPostList.size());
    }

    @Test
    void shouldGetPost_When_IdExist() {
        Long expectedId = 1L;
        final String expectedTitle = "SpringBoot as a back-end";
        final LocalDate expectedDate = LocalDate.of(2023, 9, 29);
        final Post actualPost = postRepository.findById(expectedId);
        assertNotNull(actualPost);
        assertEquals(expectedId, actualPost.getId());
        assertEquals(expectedTitle, actualPost.getTitle());
        assertEquals(expectedDate, actualPost.getPostDate());
    }

    @Test
    @Transactional
    void shouldSavePost_When_PostIsNew() {
        //Se utiliza @Transactional, porque estamos
        //realizando dos operaciones de modificado
```

```
    //de datos al mismo tiempo entonces es necesario
    //mantener la transaccion consistente con las operaciones
    final Long expectedId = 2L;
    final String expectedTitle = "Anything you want to write";
    final LocalDate expectedDate = LocalDate.of(2024, 8, 24);
    final Post newPost = Post
            .builder()
               .id(expectedId)
               .title(expectedTitle)
               .postDate(expectedDate)
            .build();
    postRepository.save(newPost);

    final Post actualPost = postRepository.findById(expectedId);
    assertNotNull(actualPost);
    assertEquals(expectedId, actualPost.getId());
    assertEquals(expectedTitle, actualPost.getTitle());
    assertEquals(expectedDate, actualPost.getPostDate());

    //Despues de comprobar que el post con Id 2 existe
    //lo borramos para mantener consistencia de los datos y los test cases
    postRepository.delete(newPost); //deleting
}

@Test
@Transactional
void shouldDeletePost_When_PostExist() {
    final Long expectedId = 3L;
    final String expectedTitle = "Deleted";
    final LocalDate expectedDate = LocalDate.of(2024, 8, 24);

    final Post newPost = Post
            .builder()
               .id(expectedId)
               .title(expectedTitle)
               .postDate(expectedDate)
            .build();

    postRepository.save(newPost); //saving

    final Post actualPost = postRepository.findById(expectedId);
    assertNotNull(actualPost);

    postRepository.delete(newPost); //deleting

    final Post deletedPost = postRepository.findById(expectedId);
    assertNull(deletedPost);
```

```
    }
}
```



NOTA: Como puede observar todas las prueban pasan, y recalcar que hacen operaciones directas a la base de datos en memoria.

IV. REFERENCIAS

Spring Framework. (2024). *JPA*. Obtenido de https://docs.spring.io/spring-framework/reference/data-access/orm/jpa.html

Baeldung. (2024). *Spring Boot With H2 Database*. Obtenido de https://www.baeldung.com/spring-boot-h2-database

Vlad Mihalcea. (2024). *How to map Date and Timestamp with JPA and Hibernate*. Obtenido de https://vladmihalcea.com/date-timestamp-jpa-hibernate/

Vlad Mihalcea. (2024). *ManyToOne JPA and Hibernate association best practices*. Obtenido de https://vladmihalcea.com/manytoone-jpa-hibernate/

ArquitecturaJava. (2024). *Spring Boot JPA y su configuración*. Obtenido de https://www.arquitecturajava.com/spring-boot-jpa-y-su-configuracion/

V. EJERCICIOS COMPLEMENTARIOS

- Codifique la clase de PostCommentRepository y realice las pruebas unitarias pertinentes.
- Investigue como cambie la conexión de una base de datos H2 a una de Mysql y verifique que el código sigue funcionando a pesar de que cambiamos la base de datos. **NOTA: Lo único que tendrá que trabajar es cambiar la sintaxis SQL del archivo schema.sql y crear la base de datos que es distinta la de Mysql sobre la H2.**