Storage: Implementación

IT BOARDING

BOOTCAMP

Índice

IT BOARDING

BOOTCAMP

STORAGE IMPLEMENTACIÓN

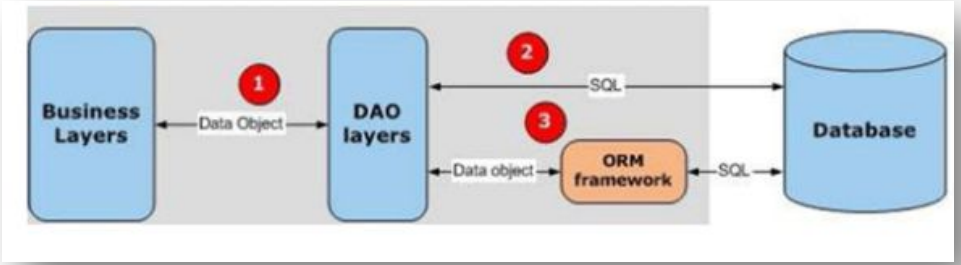
// Object Relational Mapping

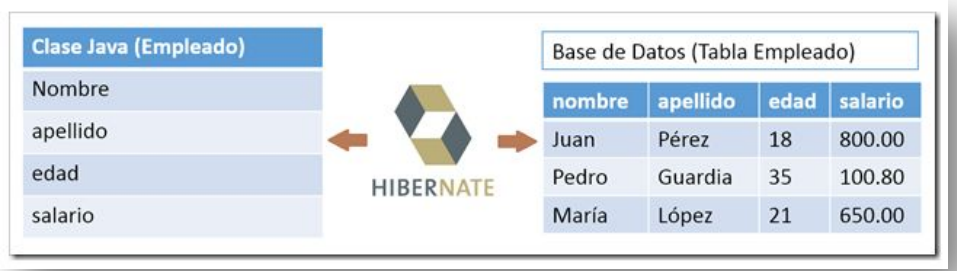
IT BOARDING

BOOTCAMP

// ¿Por qué ORM?

- Permite al código de negocios acceder a objetos en vez de tablas de BD.
- Oculta los detalles de bajo nivel de las consultas SQL.
- No es necesario preocuparse por la implementación de la BD.
- Las entidades pueden basarse en conceptos de negocios en vez de en la estructura de la BD.
- Manejo de transacciones y generación automática de claves.





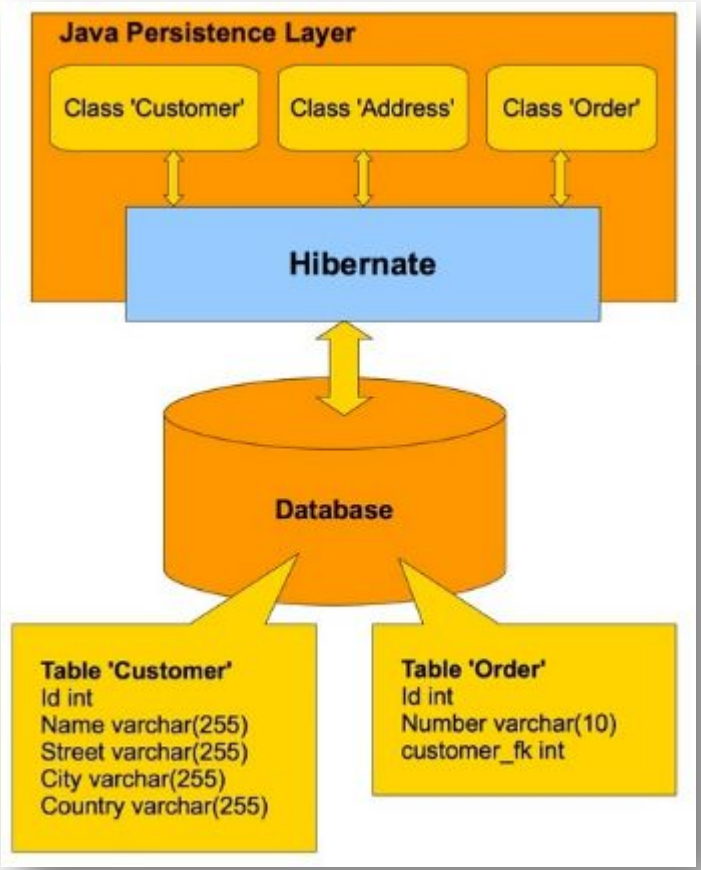
Object Relational Mapping (ORM)

Se trata de mapear o convertir datos del tipo Objeto a datos del tipo relacionales y viceversa.

La capa ORM tiene como función principal asociar un objeto a datos en la BD, posibilitando escribir las clases de persistencia utilizando OO e interactuar con las tablas y columnas de una base de datos relacional.

Hay muchos **ORM** para Java: EJB, JDO, **JPA**.

Mientras que estos son especificaciones, **Hibernate**, es una **implementación**.



STORAGE IMPLEMENTACIÓN

// Java Persistence API (JPA)

IT BOARDING

BOOTCAMP

Qué es JPA

Java Persistence API es una colección de clases y métodos que almacenan de forma persistente grandes cantidades de datos en una BD.

No es un framework, sino que define un conjunto de **conceptos** que **pueden ser implementados** por cualquier herramienta o **framework**.

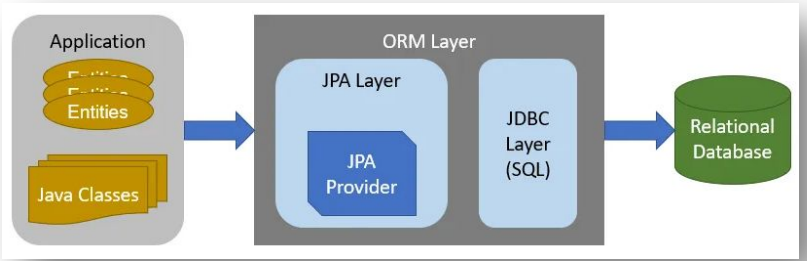
JPA busca solucionar la problemática planteada al intentar traducir un modelo orientado a objetos a un modelo relacional.

Permite almacenar entidades del negocio como entidades relacionales.

La especificación JPA posibilita al desarrollador definir qué objetos va a persistir, y cómo esos objetos deben ser persistidos en las aplicaciones Java

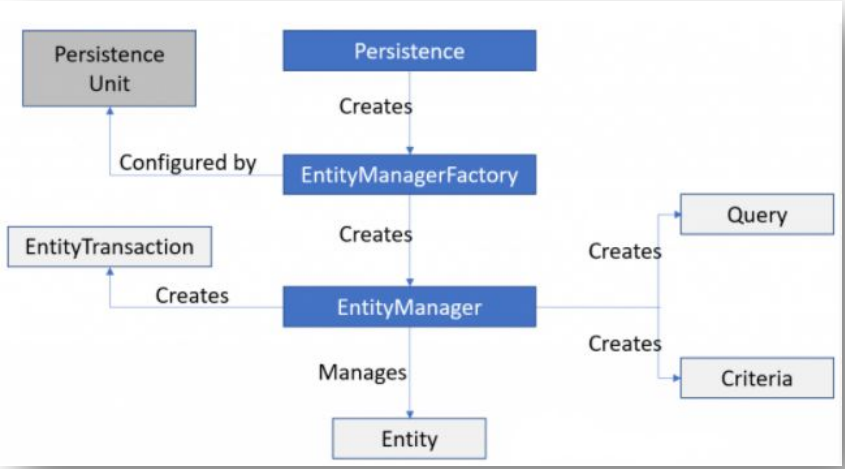
Al usar JPA se crea un **mapa** de la BD a los objetos del modelo de datos de la aplicación. La **conexión** entre la **BD relacional y la aplicación** es gestionada por **JDBC** (Java Database Connectivity API).

JPA es una API open source y **existen diferentes proveedores que lo implementan**, siendo utilizado en productos como por ejemplo **Hibernate**, Spring Data JPA, etc.



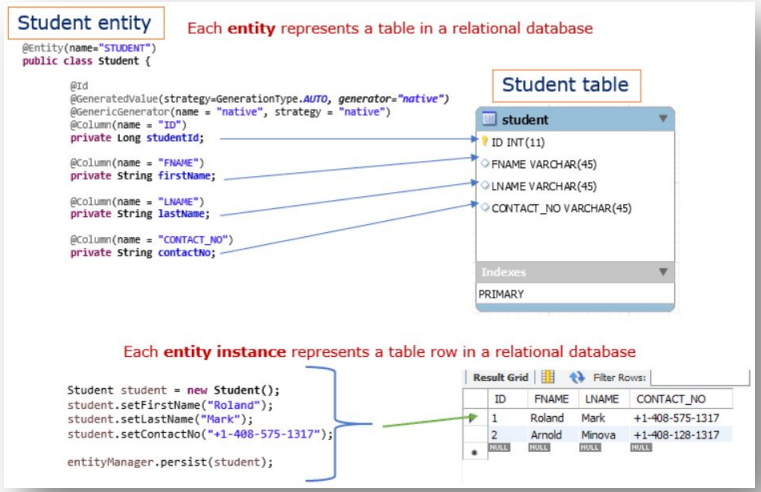
Arquitectura de JPA

El gráfico muestra las clases e interfaces que conforman el núcleo de JPA.



Paquete	Funcionalidad
EntityManagerFactory	Es una clase Factory que crea y gestiona múltiples instancias del Entity Manager. Suele haber uno por BD.
EntityManager	Es una interfaz que gestiona las operaciones de persistencia en los objetos. Crea y remueve instancias de persistencia. Encuentra entidades por su clave primaria. Permite que las queries sean ejecutadas sobre entidades.
EntityTransaction	Es un conjunto de operaciones SQL que son comitteadas o rolled backed como una unidad simple. Cualquier modificación iniciada por el EntityManager se coloca dentro de una transacción.

Entities, Queries, Criteria



Entidad: Es un objeto de persistencia. Cada Entidad representa una tabla en una base de datos relacional y cada instancia de una entidad corresponde a un registro en esa tabla. JPA utiliza anotaciones o xml para mapear entidades a una BD relacional.

```
CriteriaBuilder cb = entityManager.getCriteriaBuilder();
//create query object
CriteriaQuery<Student> query = cb.createQuery(Student.class);
//get object representing 'from' part
Root<Student> studentRoot = query.from(Student.class);
//combine 'select' and 'from' parts, equivalent to 'SELECT s FROM Student s;'
query.select(studentRoot);
TypedQuery<Student> typedQuery = entityManager.createQuery(query);
typedQuery.getResultList()
    .forEach(s->System.out.println(s.getFirstName()));
```

```
Query query = entityManager.createQuery("SELECT s FROM STUDENT s");
List<Student> s = query.getResultList();
s.forEach(student-> System.out.println(student.getFirstName()));
```

Query: Es una interfaz utilizada para controlar la ejecución de las consultas. Un EntityManager ayuda a crear un objeto Query, su implementación dependerá del proveedor de persistencia.

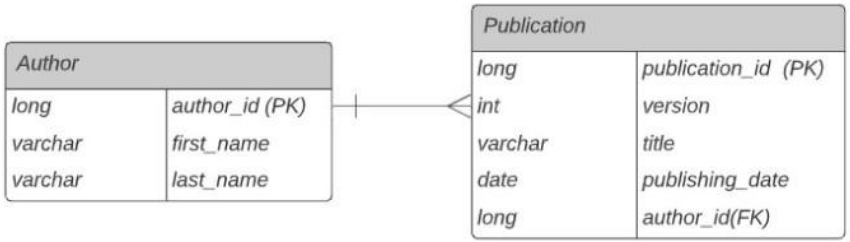
Criteria API: Permite construir queries SQL usando objetos java. Es posible realizar consultas tipadas seguras, que pueden ser chequeadas en tiempo de compilación.

Ejemplo: Mapeando del DER a clases Java

Al mapear debemos tener en consideración **el tipo de datos y sus relaciones.**

Por defecto el **nombre** del objeto persistido se convierte en el **nombre de la tabla** y los **atributos** se tornan en **columnas**.

Una vez que la tabla está creada cada registro corresponde a un objeto.




```java
@Entity
public abstract class Publication {

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

    private int version;

    private String title;

    @ManyToOne(fetch = FetchType.LAZY)
    private Author author;

     @Column(name="publishing_date")
    private LocalDate publishingDate;
}
```

```java
@Entity
public class Author {

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

    @Column(name="first_name")
    private String firstName;

    @Column(name="last_name")
    private String lastName;

    @OneToMany(mappedBy="author")
    private Set<Publication> publications = new HashSet<Publication>();
}
```

// HIBERNATE

IT BOARDING

BOOTCAMP

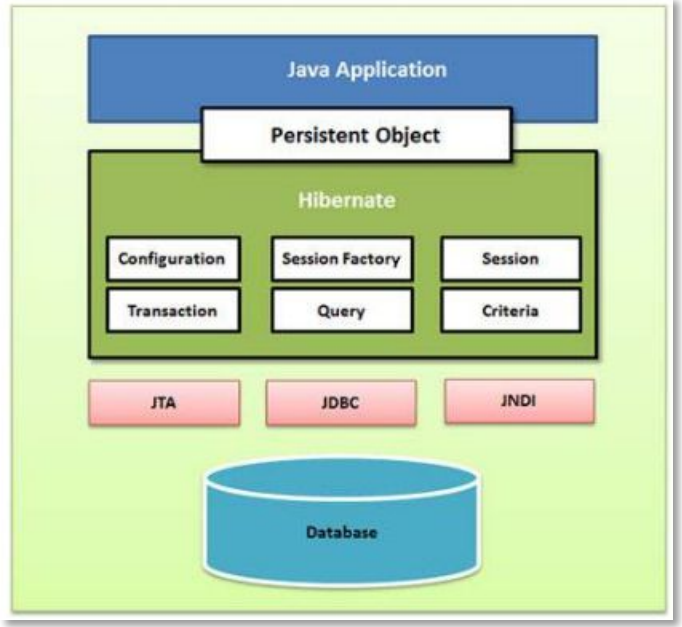
Hibernate

Hibernate es un **servicio ORM de persistencia** y **consultas** para Java. Se trata de una **implementación** de Java Persistence API **(JPA)**, pero no la única.

Mapea las clases Java en tablas de BD, y provee mecanismos para consultar datos.

El **mapeo** lo hace a través de una configuración **.xml** o de **anotaciones**.

Si es necesario un cambio en la BD, solo deberá cambiarse el archivo de configuración o las anotaciones.

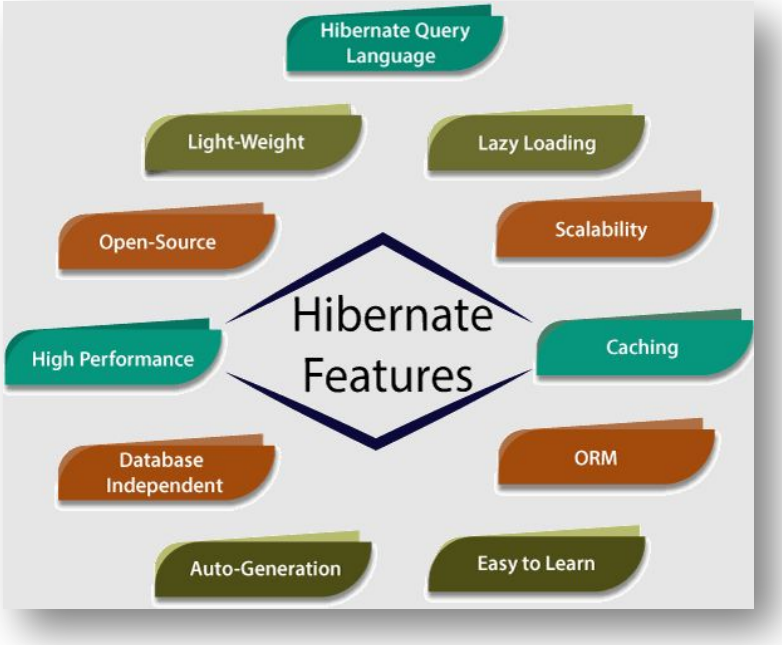


Arquitectura de Hibernate

(Implementa interfaces de JPA)

¿Por qué Hibernate?

- **Open Source**, utiliza pocos recursos.
- Alta **performance**. Es veloz porque la caché es internamente utilizada en el framework.
- **Queries independientes de la BD**. Con **HQL** (Hibernate Query Language) es posible generar queries independientes de la BD utilizada, antes de hibernate si la BD era cambiada era necesario cambiar todas las queries en la aplicación.
- **Creación automática de tablas**. No es necesario crear tablas manualmente.
- Simplifica **joins complejos**, es posible traer datos de múltiples tablas.
- Provee **estadísticas** a través del query cache, y sobre el status de la BD.



// Anotaciones

IT BOARDING

BOOTCAMP

Anotaciones

Las anotaciones que veremos a continuación pertenecen al estándar JPA y son utilizadas por Hibernate.

Son una forma potente de añadir metadata para el mapeo de Objetos y Tablas relacionales. Esta metadata se inyecta en la clase POJO de java a la par del código.

@Entity: Se etiqueta a la clase como un Bean del tipo entity que va a ser mapeado por el ORM con una tabla de la BD.

@Table: Especifica detalles de la tabla que va a ser usada para persistir la entidad en la BD.

- Con el atributo «**name**», podemos explicitar el nombre de la tabla a la que debe asociarse la clase. No es necesario usarlo si la tabla se llama exactamente igual que nuestra clase.

```java
@Entity
@Table(name="customers")
public class Customer {

   @Id
   @GeneratedValue(strategy=GenerationType.AUTO)
   @Column(name="cust_id")
   private int cid;

   @Column(name="cname")
   private String cname;

   // setters and getters
}
```

...más anotaciones

@Id: Cada Bean del tipo entity va a tener una Primary Key (que puede ser simple o compuesta). Especifica cuál es el índice, permite que la BD genere un nuevo valor con cada operación de inserción.

@GeneratedValue: Si no utilizamos esta anotación la aplicación es responsable de gestionar por si misma el campo @Id. El atributo **strategy** = **GenerationType** puede tener los siguientes valores.

- **AUTO**: Por defecto. El tipo de id generada puede ser numérica o UUID.
- **IDENTITY**: Asigna claves primarias para las entidades que utilizan una columna de identidad, son auto-incrementales.
- **SEQUENCE**: Asigna claves primarias para las entidades utilizando una secuencia que puede ser customizada.
- **TABLE**: Asigna claves primarias para las entidades utilizando una tabla de la BD, guardando en una tabla el último valor de clave primaria.

```java
@Entity
public class Course {

    @Id
    @GeneratedValue
    private UUID courseId;

    // ...
}
```

```java
@GeneratedValue(strategy=GenerationType.AUTO)

@GeneratedValue(strategy=GenerationType.IDENTITY)

@GeneratedValue(strategy=GenerationType.SEQUENCE)

@GeneratedValue(strategy=GenerationType.TABLE)
```

...y más anotaciones

@Column: Especifica los detalles de una columna, para indicar a qué atributo o campo será mapeada. Puede usarse con los siguientes atributos:

- **name** = explicita el nombre.
- **length** = especifica el tamaño de columna utilizado para mapear un valor especialmente en el caso de un String. Indica la característica del largo de la columna en la BD. Por ejemplo con length = 3, genera una columna del tipo VARCHAR(3) . Intentar insertar una String más larga daría un error de SQL.
- **nullable** = puede marcarse la columna con nullable = false cuando se genera el schema.
- **unique** = permite solo valores únicos en esa columna.

```java
@Entity
public class User {

    @Column(length = 3)
    private String firstName;

    // ...

}
```

```java
@Entity
public class User {

    // ...

    @Column(length = 5)
    @Size(min = 3, max = 5)
    private String city;

    // ...

}
```

// ¡Hagamos un ejemplo!

Ejemplo: Configurando Hibernate con SpringBoot

1) En el **pom.xml** de nuestro proyecto SpringBoot debemos agregar las siguientes **dependencias**:

- **spring-boot-starter-data-jpa:** Esta dependencia incluye la API JPA, la implementación de JPA, JDBC y otras librerías. Como la implementación por defecto de JPA es Hibernate, esta dependencia también lo trae incluído.

- **com.h2database:** Para hacer una prueba rápida, podemos agregar H2, que se trata de una base de datos en memoria muy liviana. En **application.properties** habilitamos la consola de la BD H2, para poder acceder a ella través de una UI.

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

<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>
```

Ejemplo: Configurando Hibernate con SpringBoot

2) Vamos a ir a nuestro PHPMyAdmin y vamos a crear una nueva Base de datos llamada “**prueba_jpa**”

3) Vamos a crear un usuario admin, vamos a asignarle una contraseña y permisos absolutos sobre la BD.

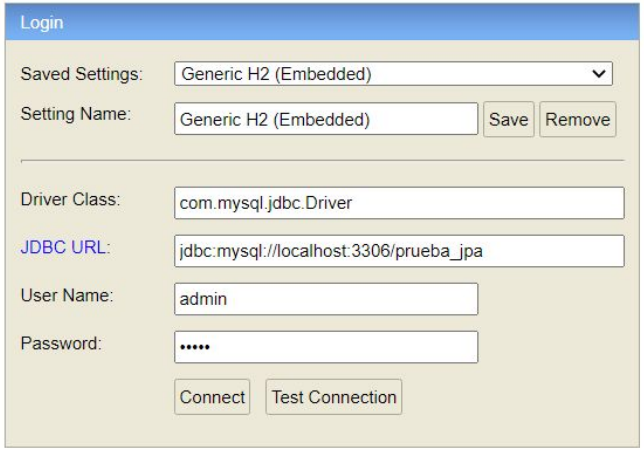
4) Vamos a nuestro archivo properties, y vamos a configurar los siguientes parámetros:

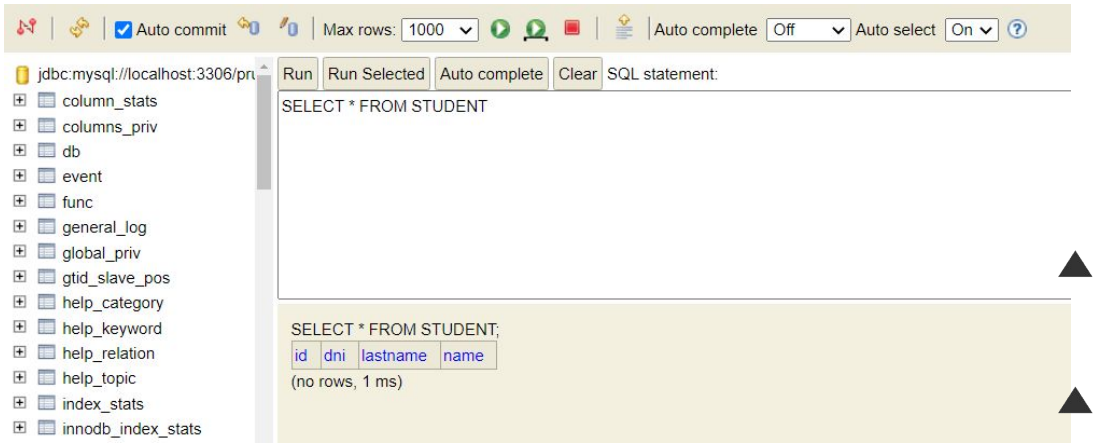
```
spring.jpa.hibernate.ddl-auto=update
spring.datasource.url = jdbc:mysql://localhost:3306/prueba_jpa?useSSL=false&serverTimezone=UTC
spring.datasource.username=admin
spring.datasource.password=admin
spring.jpa.database-platform=org.hibernate.dialect.MySQL5InnoDBDialect
```

5) Una vez que tenemos todo configurado, podemos probar si funciona ingresando a la url: *http://localhost:8080/h2-console/*

Ejemplo: Configurando Hibernate con SpringBoot

6) Ejecutamos nuestra aplicación y podemos probar **ingresar a la UI de H2** donde contamos con un panel donde a futuro vamos a poder realizar consultas SQL:





Ejemplo: Configurando Hibernate con SpringBoot

7) Creamos una **entidad Student**

```java
@Getter @Setter
@Entity
public class Student {

    @Id
    @GeneratedValue (strategy = GenerationType.SEQUENCE)
    private Long id;
    private String dni;
    private String name;
    private String lastname;

}
```

8) Creamos el **repositorio** y el **servicio**.

```java
@Repository
public interface StudentRepository extends JpaRepository <Student, Long> {

}
```

```java
@Service
public class StudentService {

        private final StudentRepository stuRepo;

    public StudentService (StudentRepository stuRepo) {

        this.stuRepo = stuRepo;

    }
}
```

Ejemplo: Configurando Hibernate con SpringBoot

9) Ejecutamos nuestra aplicación y si todo sale bien, vamos a ver lo siguiente en nuestra base de datos:

	Tabla	Acción						Filas	Tipo
☐	hibernate_sequence	☆	Examinar	Estructura	Buscar	Insertar	Vaciar	Eliminar	1	InnoDB
☐	student	☆	Examinar	Estructura	Buscar	Insertar	Vaciar	Eliminar	0	InnoDB
	2 tablas	Número de filas							1	InnoDB

- Una tabla para las secuencias
- Otra tabla para representar nuestra clase Student

Gracias.

IT BOARDING

BOOTCAMP