Spring
Persistencia de Datos

Desarrollo Web en Entorno Servidor

Introducción

Hasta ahora hemos trabajado con datos que tan solo han existido en la memoria volátil del ordenador.

Esto provoca que los datos se pierdan cuando se reinicia la aplicación.

Para evitar esto, toda aplicación del lado servidor utiliza BBDD para hacer que la información con la que trabaja sea persistente.

Gracias a esto los datos perdurarán en el tiempo más allá del tiempo que la aplicación esté en ejecución.

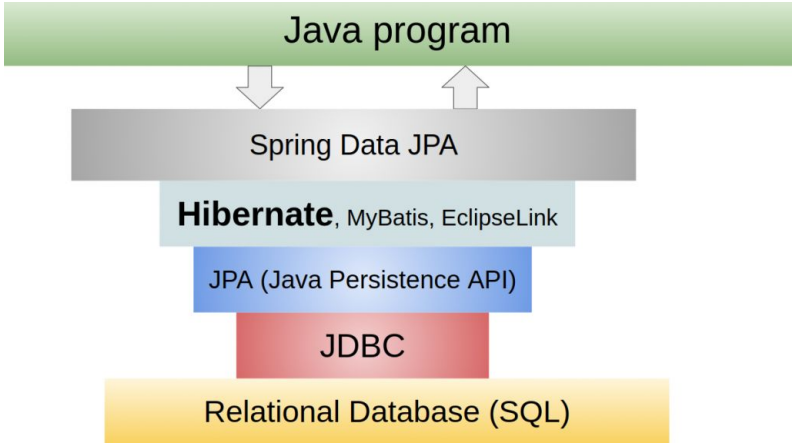
Las aplicaciones, una vez conectadas al SGBD, podrán realizar operaciones CRUD de los modelos con los que trabajan.

Elementos Para la Persistencia

La persistencia de los datos, en el contexto que nos encontramos (aplicación java con el framework Spring Boot), implica el uso (directo o indirecto) de ciertos elementos que tendremos que conocer, y que en nuestro caso, tomarán parte de este proceso:

- Base de datos Relacional o No Relacional
- JDBC
- JPA
- Hibernate
- Spring Data JPA

Veamos cada uno de ellos.



Elementos Para la Persistencia

Base de datos Relacional

En nuestro caso utilizaremos el sistema gestor de base de datos (SGBD) MySql.

Para facilitar las cosas, utilizaremos el paquete de herramientas XAMPP, que nos proporciona este SGBD.

Para utilizarlo tendremos que añadir como dependencia el driver de MySQL al proyecto.

Los datos de conexión con la BD se definen en el archivo “application.properties” del directorio “resources”. Ejemplo:

```
spring.jpa.hibernate.ddl-auto=update
spring.datasource.url=jdbc:mysql://${MYSQL_HOST:localhost}:<port>/<DBName>
spring.datasource.username=<DBUserName>
spring.datasource.password=<password>
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
#spring.jpa.show-sql: true
```

Los valores entre <> son particulares de cada implementación y se han de especificar

Elementos Para la Persistencia

JDBC

Acrónimo de *Java Database Connectivity* es la API de Java a más bajo nivel para poder ejecutar operaciones sobre bases de datos desde este lenguaje. Este ofrece operaciones independientes del SO o el tipo de BD utilizado.

Necesita de un driver específico del tipo de SGBD que va a manejar.

JPA

Acrónimo de Java/Jakarta Persitence API, es una interfaz de acceso y consulta a BD en Java. Ofrece una serie de métodos estandarizados para la gestión de la persistencia de datos. Existen varias implementaciones concretas de esta interfaz.

Hibernate

Es una implementación de JPA ampliamente utilizada. Podríamos decir que es prácticamente un estándar, al ser la más utilizada.

Esta herramienta facilita el trabajo con BBDD y se encarga del mapeo objeto-relacional.

Elementos Para la Persistencia

Spring Data JPA

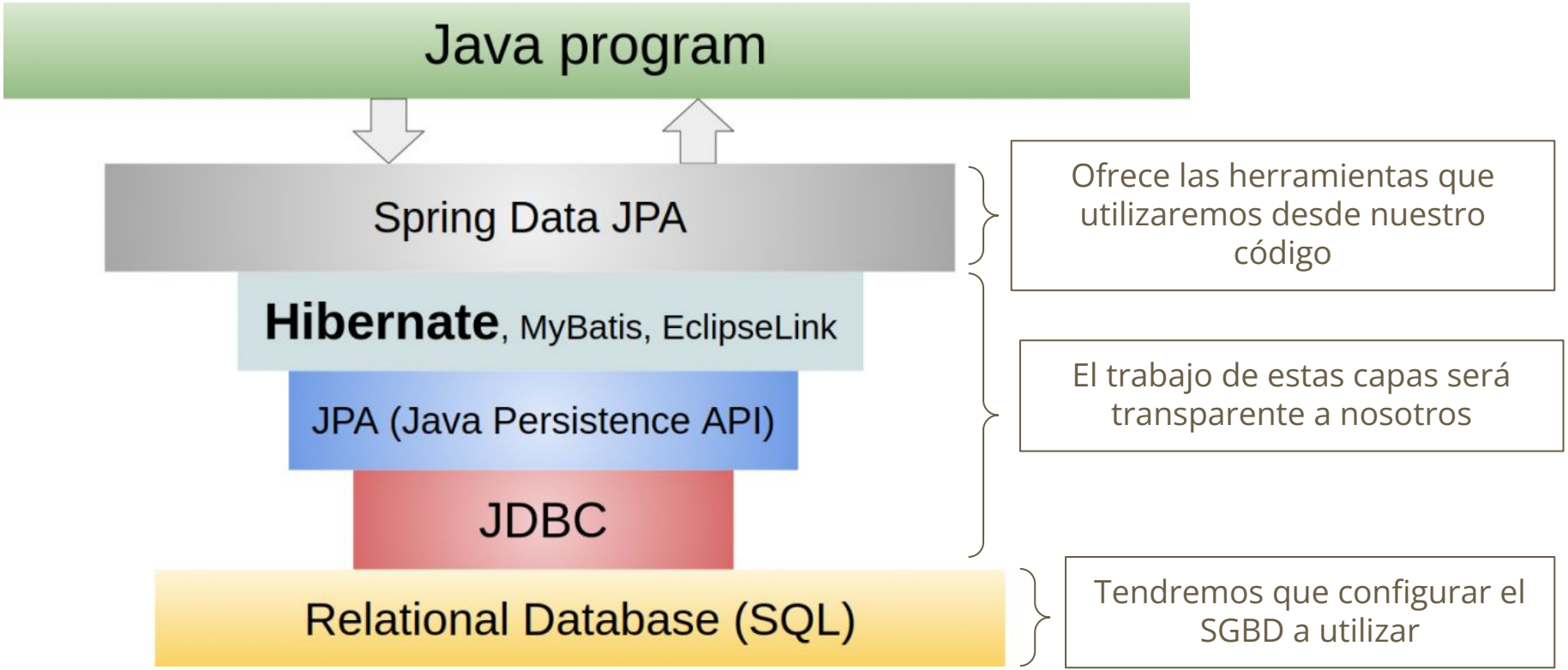
Es una dependencia de Spring para el manejo de persistencia de datos.

No es una implementación propia de JPA como Hibernate, y por lo tanto no compite con ella, sino que lo que hace es mejorar y completar Hibernate para facilitar el uso de la persistencia de datos.

Esta dependencia nos ofrecerá un elemento fundamental para utilizar de forma eficiente y sencilla una BD: **Los Repositorios**

Como se mostró antes, la relación entre estos elementos comentados se muestra en la siguiente figura.

Elementos Para la Persistencia



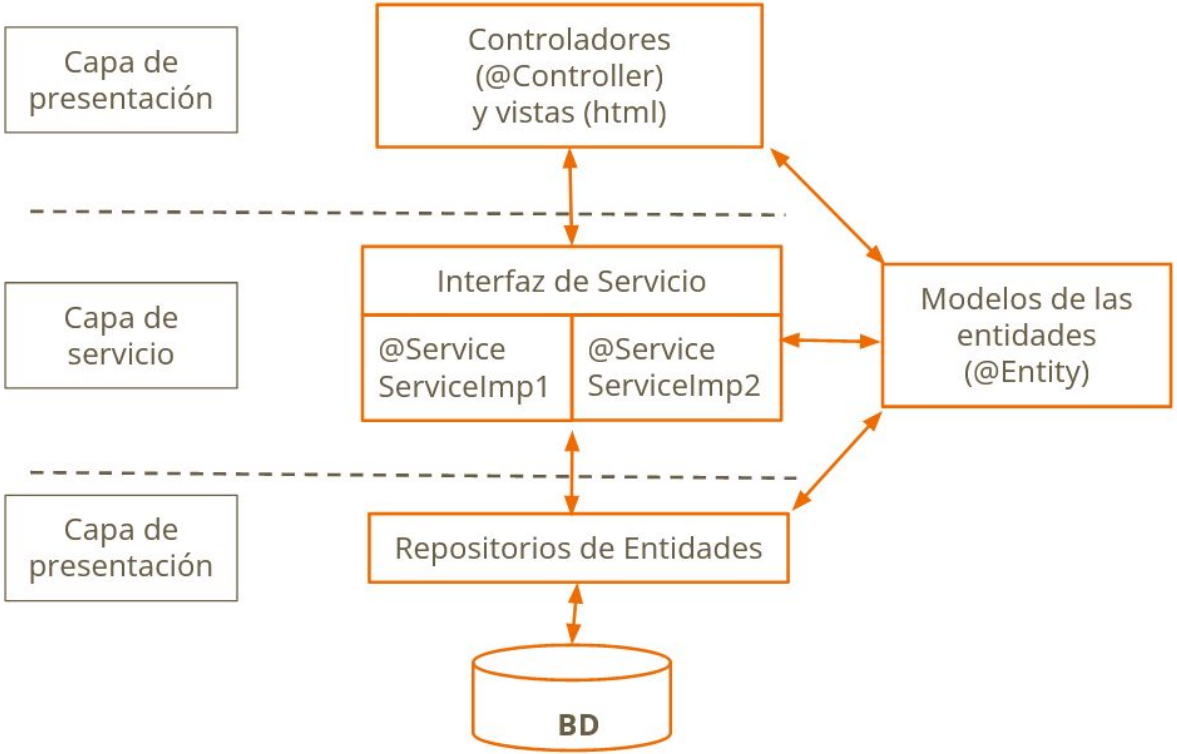
Arquitectura de la Persistencia

En cuanto al código de nuestro proyecto, estará formado por distintos elementos que conforman una determinada arquitectura representada por tres capas principales:

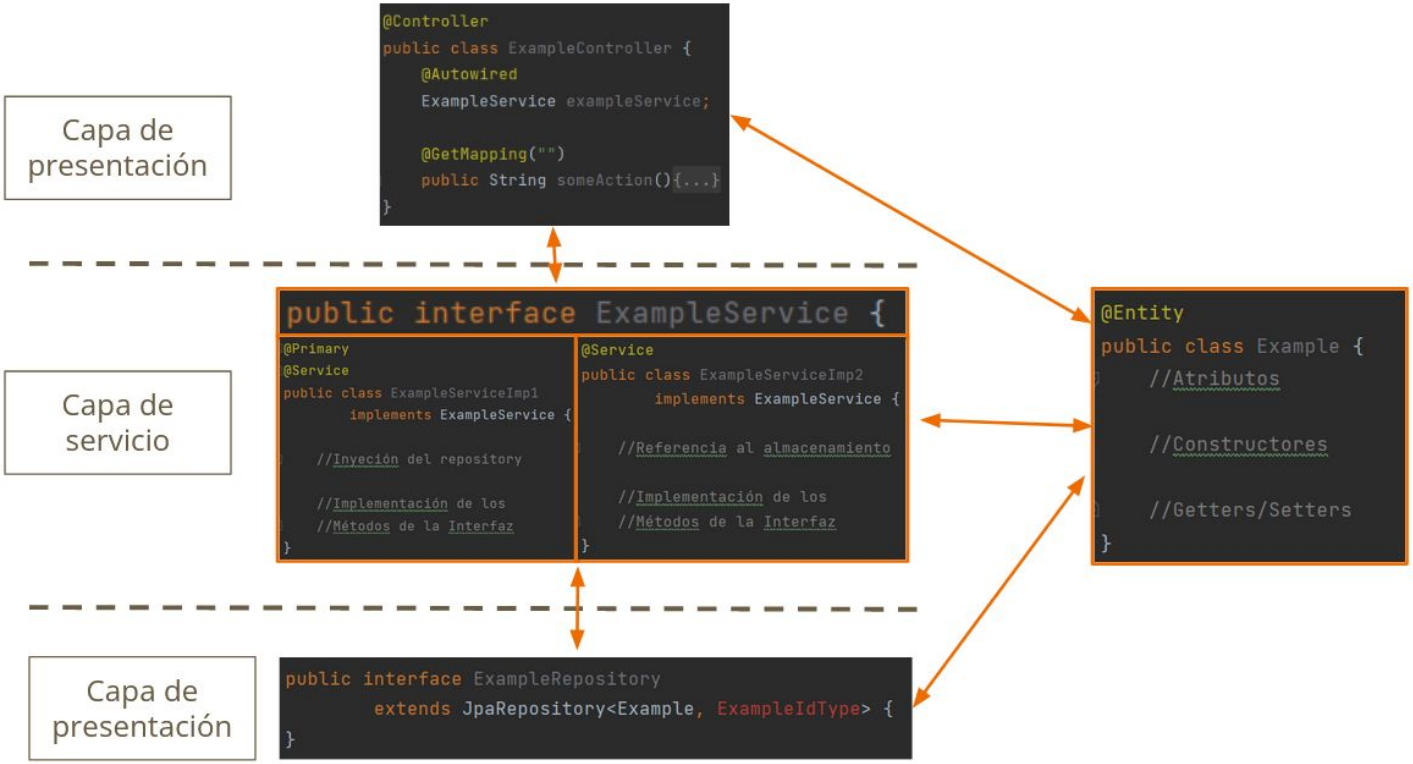
- Capa de presentación
  - La conforman los elementos que sirven de interfaz de comunicación con el usuario: **los controladores, las vistas, etc.**
- Capa de servicios
  - Es la capa encargada de asumir la lógica de negocio. A ella pertenecen nuestros ***Beans* de servicio**, así como otras clases que ofrecen servicios como *mailers*, *printers*, etc.
- Capa de persistencia
  - Formada por las interfaces de los ***repositories*** de nuestro proyecto.

Estas capas, y los elementos que las componen, conforman una estructura similar a la siguiente:

Arquitectura de la Persistencia



Arquitectura de la Persistencia



Arquitectura de la Persistencia

Modelos

Los modelos tendrán la anotación @Entity para que Spring pueda relacionarlos con las tablas de la BD.

Cuando la clase que representa un modelo es un *bean* de tipo @Entity, es necesario que al menos uno de sus atributos funcione como clave primaria.

El atributo que se usará como clave primaria tendrá dos anotaciones.

- @Id → para especificar que el atributo es la clave primaria
- @GeneratedValue → para indicar que la gestión de este campo la realizará el framework y no los desarrolladores.
- Para una generación automática e incremental de la clave primaria tendremos que poner esta anotación:

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

Arquitectura de la Persistencia

Modelos

Por defecto, Spring Data JPA utilizará los nombres de las clases y atributos de las entidades para crear/mapear las tablas y columnas de la BD.

Sin embargo, **es posible especificar distintos nombres de tablas y columnas** para mapearlos con las entidades y sus atributos.

Para ello se utilizarán las siguientes anotaciones encima de la definición de la clase y de cada atributo:

- **@Table(name="<table_name>")** → mapea la clase con la tabla especificada.
- **@Column(name="<column_name>")** → mapea el atributo con la columna especificada.

Arquitectura de la Persistencia

Servicios

Definen las operaciones CRUD de las entidades.

Será buena práctica definir una interfaz de los servicios donde se definan las cabeceras de los métodos que se han de implementar.

Se podrá tener 1 o más implementaciones de una interfaz, según necesidades.

Se podrá definir una implementación como principal con la anotación @Primary.

Los servicios se han de inyectar en los controladores con @Autowired

Arquitectura de la Persistencia

Repositorios

En ellos reside toda la “magia” de Spring Data JPA para facilitar las operaciones de recuperación y persistencia de los datos de los modelos.

Existirá un repositorio por cada entidad del proyecto.

Son interfaces que extienden de la clase “JpaRepository”, indicando:

- Nombre de la entidad con la que trabajan
- Tipo de dato de la clave primaria de la entidad

Estas interfaces no necesitan definir explícitamente las operaciones básicas, pero **pueden tener definiciones de operaciones complejas** con *query keywords* (Guía).

Los repositorios se han de inyectar en el servicio de la entidad correspondiente.

Arquitectura de la Persistencia

Repositorios: Operaciones Básicas

Algunos de los métodos que ofrecen por defecto estos elementos son:

- findAll() : recupera todos los elementos de una entidad
- save() : almacena, o actualiza, un elemento de una entidad
- findById().orElse() : Recupera un elemento por su Id, o sino devuelve el elemento especificado.
- deleteById() : Elimina entidades por su Id.

Estos métodos, y muchos más, están definidos en detalle en la documentación oficial.

Arquitectura de la Persistencia

Repositorios: Operaciones Complejas

Como ya se ha comentado, los repositorios pueden tener definiciones de operaciones complejas con *query keywords* (Guía).

Gracias a esto se podrán realizar consultas concretas sobre los datos de una entidad en función de las necesidades de nuestro proyecto.

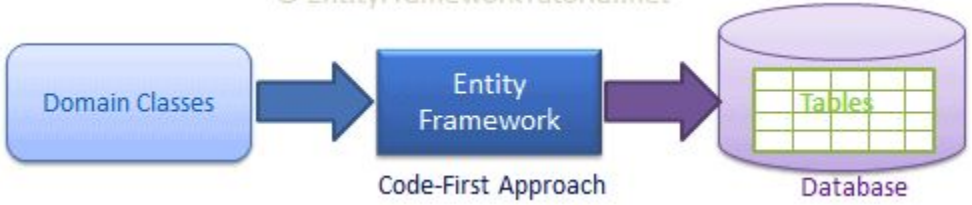
Un ejemplo de consulta *keyword*, para mostrar como se ha de incluir en el repositorio, sería la siguiente:

```
public interface ClaseRepository extends JpaRepository<Clase, Integer> {
    Clase findByName(String claseName);
}
```

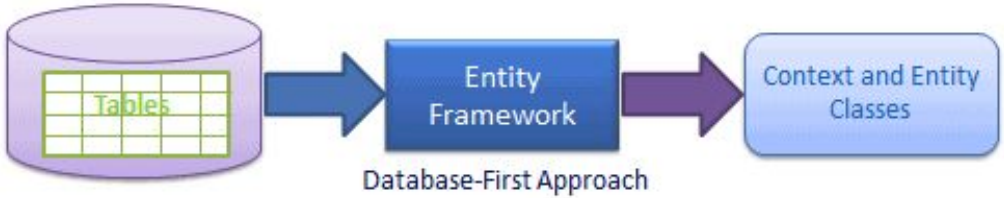
Enfoques de la Persistencia

La persistencia se puede trabajar de dos formas con la BD:

- **Code First**: Se definen las clases de los modelos, con sus relaciones, y se generan automáticamente las tablas en la base de datos



- **Database First**: Partiendo de una base de datos relacional, se utiliza el ORM Hibernate para generar las clases de los modelos de la aplicación.



Enfoques de la Persistencia

¿Code First ó Database First?

Hay que elegir una de las dos en función de algunos aspectos:

- Code First
  - El equipo de desarrollo maneja mejor el código que las bases de datos
  - El proyecto no trabaja sobre una BD previa, sino que la crea desde el inicio
- Database First
  - El equipo de desarrollo se defiende bien con las bases de datos
  - El proyecto se ha de adaptar a una base de datos existente

Spring Data JPA - Code First

Este enfoque es el más sencillo de implementar en los proyectos Spring.

El propio framework se encargará de crear las tablas en la BD de aquellas entidades que no tengan tabla correspondiente.

Igualmente, se encargará de ir adaptando los campos de las entidades con las tablas existentes en la BD.

Para que se puedan crear la tabla en una BD, para un modelo determinado, se requiere de lo siguiente:

- La clase del modelo ha de estar anotada con @Entity
- Se ha de haber definido correctamente la conexión con la BD en el archivo “application.properties”

Al lanzar el proyecto se harán automáticamente las comprobaciones y actualizaciones necesarias para mantener una correspondencia entre entidades y tablas.

Ejercicio 4

Realizar un proyecto Spring MVC, que incluya el modelo Animal del Ejercicio 3. Esta vez habrá que utilizar los mecanismos proporcionados por Spring Data JPA para persistir los datos en una BD MySql.

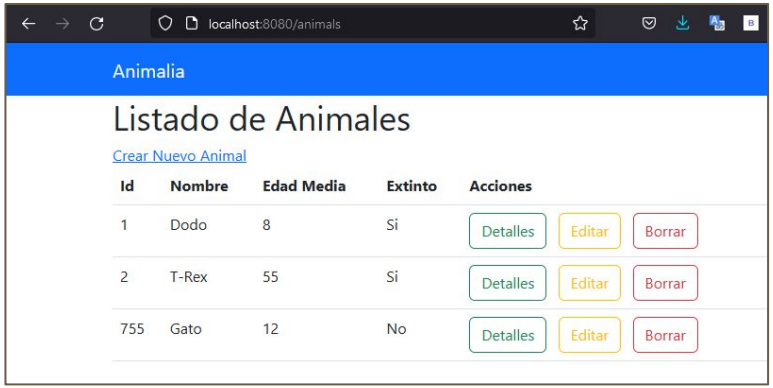
Se tendrá que respetar la estructura y arquitectura propuesta en clase y reflejada en las diapositivas.

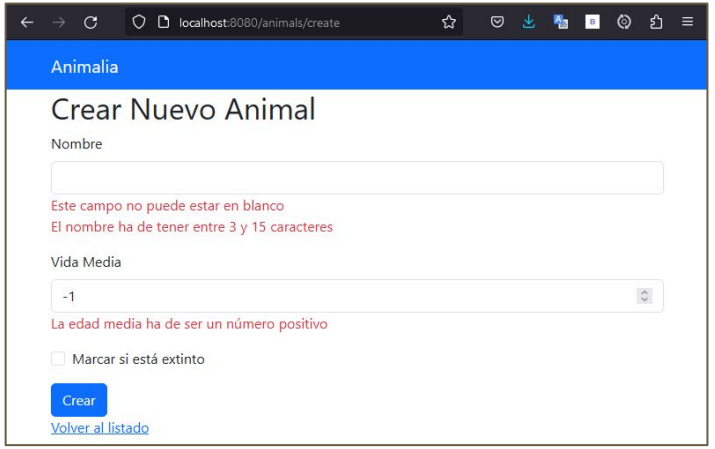
Se podrá tener que alternar entre el servicio que ofrece almacenamiento de elementos de tipo Animal en memoria (como en el Ejercicio 3) con la persistencia de este tipo de entidad en la BD. Todo esto sin tener que modificar ningún archivo que quede fuera de la capa de servicios.

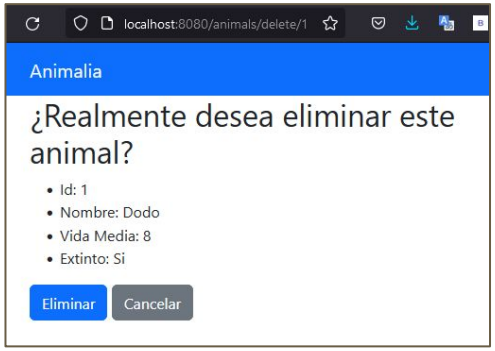
Se tienen que poder crear, detallar, editar y borrar elementos de tipo Animal, tanto en memoria como en BD.

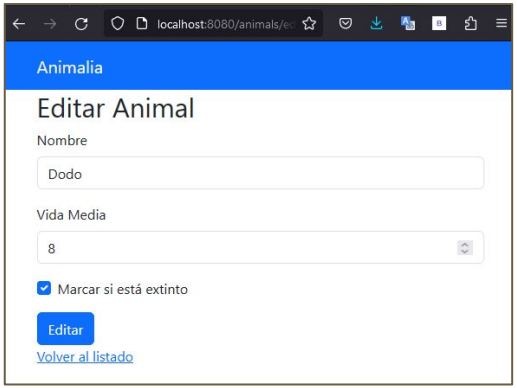
Una propuesta de solución se corresponde con las siguientes capturas:

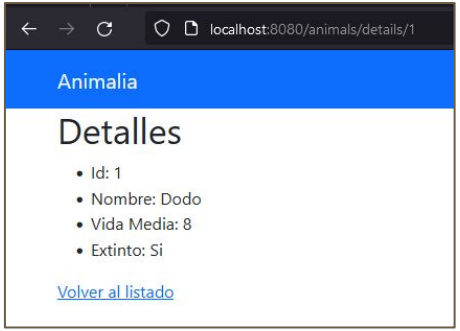
Ejercicio 4


localhost:8080/animals
Animalia
Listado de Animales
Crear Nuevo Animal
Id
Nombre
Edad Media
Extinto
Acciones
1
Dodo
8
Si
Detalles
Editar
Borrar
2
T-Rex
55
Si
Detalles
Editar
Borrar
755
Gato
12
No
Detalles
Editar
Borrar


localhost:8080/animals/create
Animalia
Crear Nuevo Animal
Nombre
Este campo no puede estar en blanco
El nombre ha de tener entre 3 y 15 caracteres
Vida Media
-1
La edad media ha de ser un número positivo
Marcar si está extinto
Crear
Volver al listado


localhost:8080/animals/delete/1
Animalia
¿Realmente desea eliminar este animal?
Id: 1
Nombre: Dodo
Vida Media: 8
Extinto: Si
Eliminar
Cancelar


Animalia
Editar Animal
Nombre
Dodo
Vida Media
8
Marcar si está extinto
Editar
Volver al listado


localhost:8080/animals/details/1
Animalia
Detalles
Id: 1
Nombre: Dodo
Vida Media: 8
Extinto: Si
Volver al listado

Relaciones Entre Entidades

La forma en que las entidades se relacionan en una base de datos (de ahí que son bases de datos relacionales) ha de ser reflejado de alguna manera en las clases de los modelos del proyecto.

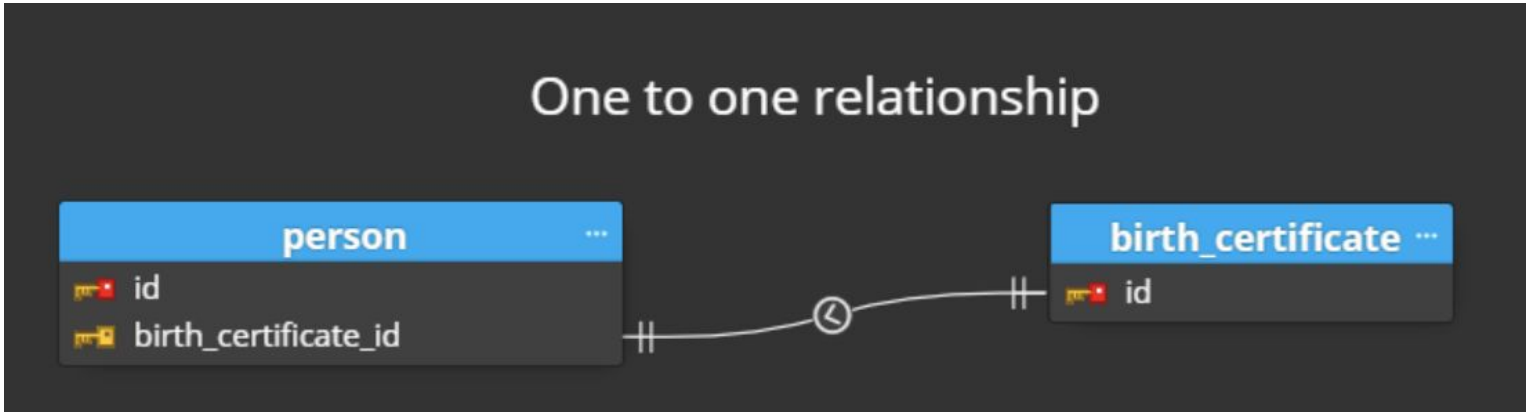
Las formas en que se pueden definir estas relaciones en los modelos son múltiples, pero todas hacen referencia a 3 formas básicas de relación entre entidades:

- One to One
- One to Many
- Many to One
- Many to Many

Relaciones Entre Entidades

One to One

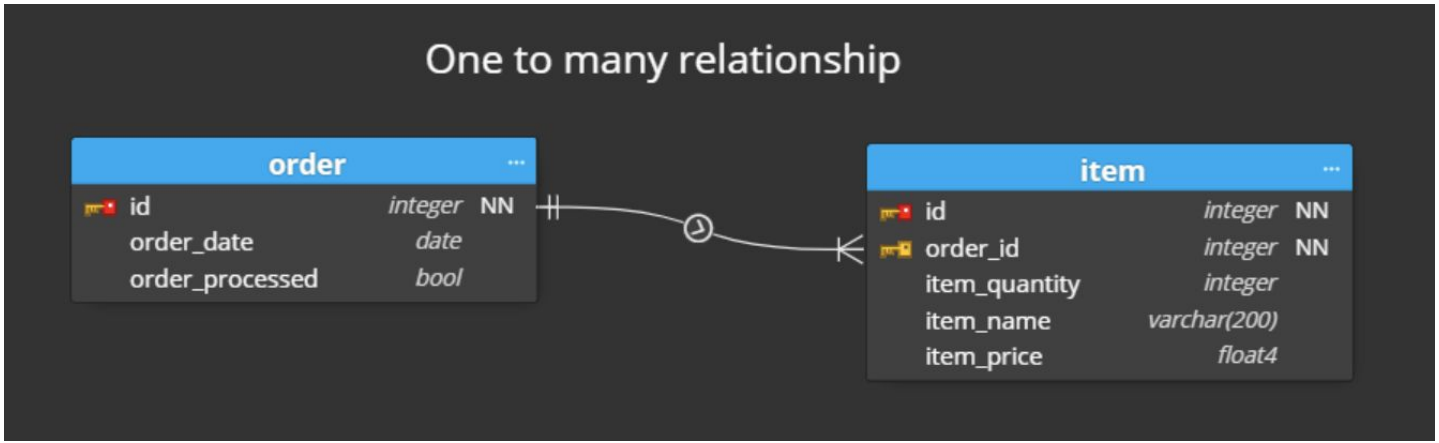
Son aquellas relaciones entre dos tablas donde un registro de una entidad hace referencia a otro registro de otra entidad, y viceversa.



Relaciones Entre Entidades

One to Many

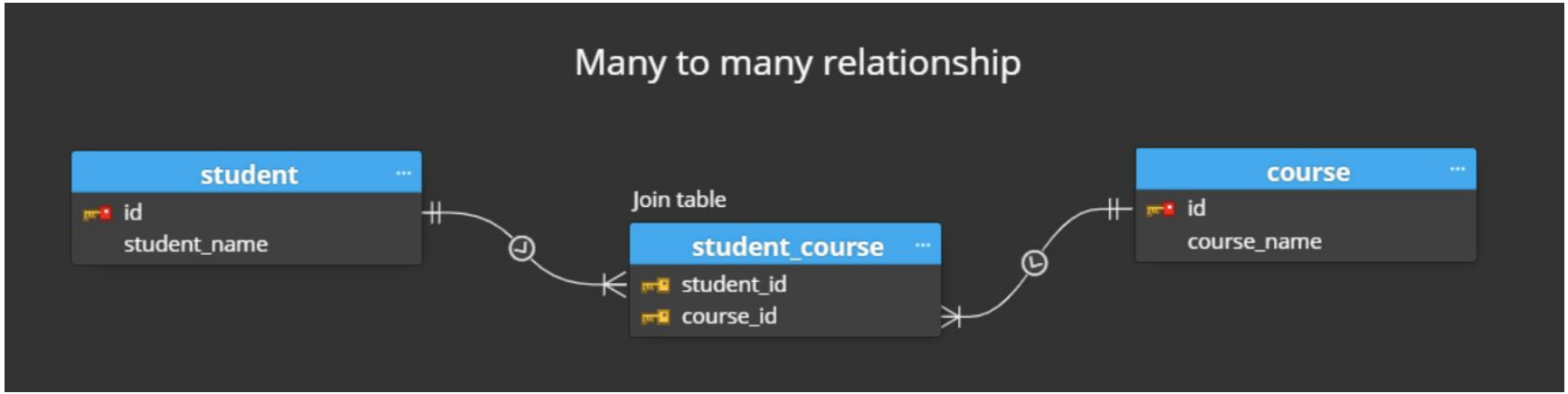
Son aquellas relaciones entre dos tablas donde un registro de una entidad hace referencia a otro registro de otra entidad, y el contrario puede ser referenciado por varios registros.



Relaciones Entre Entidades

Many to Many

Son aquellas relaciones entre dos tablas donde un registro de una entidad puede ser referenciado por varios de otra entidad, y viceversa.



Relaciones Entre Entidades

¡OJO con esto, que cascade y orphanremoval podrían ser redundantes, investigarlo!

Ejemplo de relación One to Many

Este tipo de relación se implementa de la siguiente forma en los modelos.

```java
@Entity
public class Album {

    //Id y otros atributos

    @OneToMany(mappedBy = "album",
            cascade = CascadeType.ALL,
            orphanRemoval = true)
    private List<Track> tracks;

    //Constructores
    //Getter/seeter
}
```

```java
@Entity
public class Track {

    //Id y otros atributos

    @ManyToOne
    private Album album;

    //Constructores
    //Getter/seeter
    //etc
}
```

Relaciones Entre Entidades

Ejemplo de relación One to Many

Una vez definida la relación entre los modelos, se actualizará la BD en el siguiente lanzamiento del proyecto.

Siguiendo con el ejemplo anterior, la tabla de “track” contendrá una FK que apuntará a la tabla “album”. El nombre de esta columna, si no damos ningún otro de forma explícita, sería “album_id”.

Spring Data JPA hará que cada vez que recuperemos elementos de cualquiera de las dos entidades se incluyan los valores de los campos que referencian (tanto el listado de Tracks en caso del Album, como un elemento de tipo Album en un Track).

Relaciones Entre Entidades

Ejemplo de relación One to Many

A la hora de crear un elemento del lado “many” (en nuestro caso un Track) habrá que mostrar un listado de Album en el **formulario** de creación del Track.

Lo más común será mostrar un elemento de tipo “select” con los valores posibles a elegir (en este caso los distintos álbumes a los que podría pertenecer un track).

En el controller habrá que pasar ese listado de elementos en el model.

Relaciones Entre Entidades

Ejemplo de relación One to Many

El desplegable del formulario para la creación del elemento del lado *many* de la relación (Track en este caso), tendrá un <select> similar al siguiente:

```
<select th:field="*{album}" >
  <option th:each="album: ${albumes}"
          th:value="${album.id}"
          th:text="${album.nombre}"></option>
</select>
```

Nombre del atributo anotado con @ManyToOne, del lado *many* de la relación

Listado de elementos recuperado en el *controller* e incluido en el *Model*

Ejercicio 5

Crear un nuevo proyecto Spring MVC, con las dependencias *starter* correspondiente, que incluya toda la funcionalidad del Ejercicio 4 más todo lo siguiente:

- Incluir una nueva entidad llamada Clase (que representa las clases de Animales que hay) que tenga los siguientes atributos: Id y Nombre.
- Entre la entidad Animal y Clase ha de existir una relación *OneToMany* (varios animales pueden pertenecer a una misma clase)
- Se han de poder crear, detallar, editar y eliminar entidades de los dos tipos.
- Se utilizarán vistas parciales, mediante Thymeleaf, para generar un único *header* que sea utilizado en todas las vistas.
- Se tendrá que crear una *home* del sitio, desde la que se podrá navegar al listado de animales o clases desde los enlaces del *header*.
- Al mostrar los detalles de un elemento de tipo Clase, se tendrán que listar todos los animales que pertenecen a esta clase.

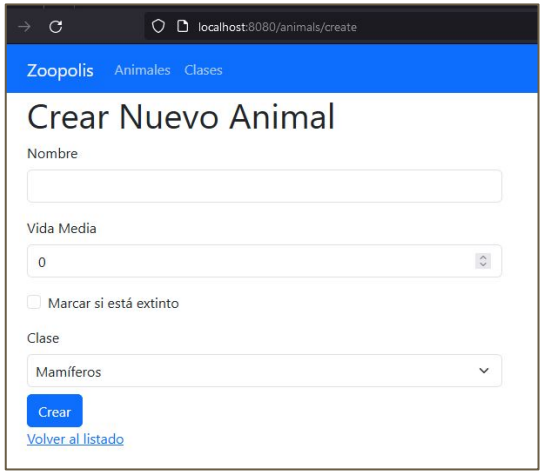
La apariencia del producto final sería similar a lo siguiente:

Ejercicio 5


localhost:8080/clases
Zoopolis
Animales
Clases
Listado de Clases de Animales
Crear Nueva Clase
Id
Nombre
Acciones
1
Mamíferos
Detalles
Editar
Borrar
2
Peces
Detalles
Editar
Borrar
3
Reptiles
Detalles
Editar
Borrar
4
Aves
Detalles
Editar
Borrar


localhost:8080
Zoopolis
Animales
Clases
Biblioteca de Zoopolis


localhost:8080/animals
Zoopolis
Listado de Animales
Crear Nuevo Animal
Id
Nombre
Edad Media
Extinto
Clase
Acciones
4
Gato
15
No
Mamíferos
Detalles
Editar
Borrar
52
Iguana
12
No
Reptiles
Detalles
Editar
Borrar
53
T-Rex
50
Si
Reptiles
Detalles
Editar
Borrar
54
Perro
16
No
Mamíferos
Detalles
Editar
Borrar


localhost:8080/animals/create
Zoopolis
Animales
Clases
Crear Nuevo Animal
Nombre
Vida Media
0
Marcar si está extinto
Clase
Mamíferos
Crear
Volver al listado


localhost:8080/clases/details/1
Zoopolis
Animales
Clases
Detalles
Id: 1
Nombre: Mamíferos
Animales de esta clase:
Gato
Perro
Volver al listado