

# Java con Spring, JPA y MySQL

## Introducción

**Java Spring Boot** es una de las herramientas principales del ecosistema de desarrollo web con Java a nivel del backend en cuanto a entornos empresariales, principalmente en arquitecturas basadas en servicios web (REST y SOAP) y microservicios.

Primero que todo hay que hacer énfasis en que **Spring Boot NO es Spring** y que Spring Boot surge de la necesidad de realizar aplicaciones Java sin tantas complicaciones de configuración y toda la problemática que eso conlleva.

Dado lo anterior y por muchas más razones, es que nace Spring Boot y que junto a proyectos como Spring Framework, Spring Data, Spring Security, entre muchos otros, son la combinación perfecta para desarrollar, probar y desplegar nuestras aplicaciones en un entorno rápido, eficaz y bastante simple.

En el siguiente documento se explicará desde la creación de un proyecto con Spring Boot, las dependencias que usaremos y la conexión con MySQL.

## Requerimientos

Para poder empezar a trabajar debemos tener en cuenta los siguientes elementos; cada elemento como el IDE e incluso la base de datos, puede ser cambiada, solo hay que tener presente que algunos datos en la programación cambiarán con respecto a la base de datos, como garantizar que en el IDE elegido el proyecto de Java con Spring corra perfectamente y sin complicaciones.

1. Windows, Linux o Mac OS como sistema operativo
2. [IntelliJ IDEA Ultimate](#) como IDE
3. [MySQL 8](#) como base de datos
4. [Postman](#) para realizar pruebas a las API creadas

Una vez se tenga todo instalado, procederemos a realizar la creación de nuestro proyecto Java con Spring.

## Creación del proyecto

Para crear el proyecto Java con Spring tenemos dos opciones, la primera es entrar en la siguiente dirección: <https://start.spring.io/>, realizar la configuración pertinente, descargar el proyecto, el cual, viene en formato .zip, descomprimir y abrir la carpeta con el IDE indicado.

La segunda forma es crear el proyecto directamente en el IDE con ayuda del plugin Spring Boot, el cual, ya viene por defecto en IntelliJ IDEA Ultimate, en caso de no estarlo, solo es cuestión de instalar el plugin indicado anteriormente.

### Cómo crear el proyecto con Spring Initializr (página web)

Como ya se comentó anteriormente, debemos entrar en la siguiente página <https://start.spring.io/> la cual nos dará una página muy parecida a la siguiente:

The screenshot shows the Spring Initializr web form with the following numbered annotations:

- 1: Project type selection (Maven Project selected)
- 2: Language selection (Java selected)
- 3: Spring Boot version selection (2.6.4 selected)
- 4: Group ID (com.example)
- 5: Artifact ID (demo)
- 6: Name (demo)
- 7: Description (Demo project for Spring Boot)
- 8: Package name (com.example.demo)
- 9: Packaging (Jar selected)
- 10: Java version selection (17 selected)
- 11: Dependencies section (Spring Boot DevTools, Lombok, Spring Web, Spring Data JPA, MySQL Driver)
- 12: GENERATE button

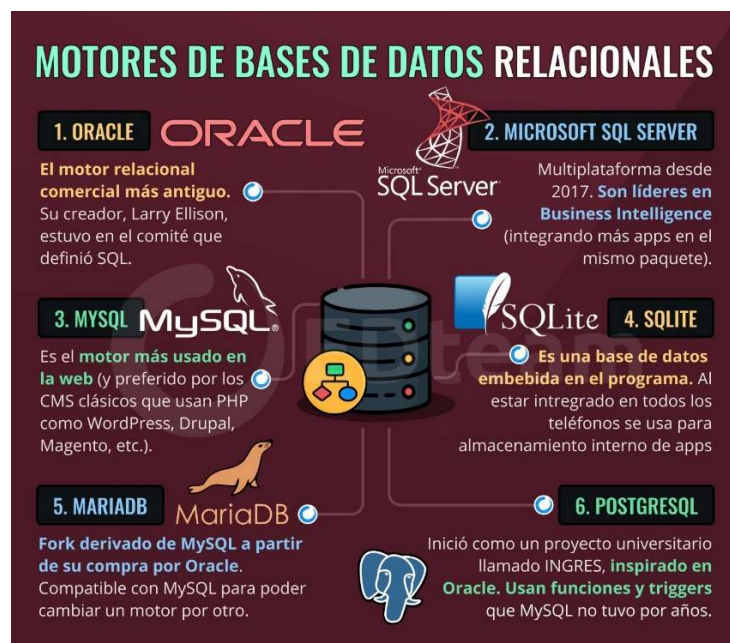
1. El proyecto puede ser creado utilizando Maven o Gradle, lo cual, son tecnologías que se usan en Java para la administración de las dependencias; en este caso usaremos **Maven**.
2. Ahora seleccionaremos el lenguaje con el cual se quiere trabajar, y podemos escoger entre tres lenguajes: Java, Kotlin o Groovy; en nuestro caso usaremos **Java**.

3. La idea en este punto es trabajar con la última versión estable de Spring Boot que se encuentre disponible al momento de crear nuestro proyecto; para nuestro caso usaremos la versión 2.6.4.

Desde el punto cuatro (4) al punto ocho (8) configuraremos tanto nombre del proyecto, como paquete al que pertenecerá y otros metadatos importantes para el mismo.

4. En **Group** daremos el nombre del paquete a usar; para nuestro ejercicio usaremos `com.sofka`
5. En **Artifact** daremos el nombre del artefacto que vamos a crear. En muchas ocasiones es habitual escribir el nombre del proyecto a usar; para nuestro ejemplo usaremos la palabra `contactos`.
6. En **Name** escribiremos el nombre del proyecto; en nuestro caso usaremos por nombre `Sistema DEMO de Contactos`.
7. En **Description** escribiremos la descripción de nuestro proyecto; para efectos de nuestro ejercicio, usaremos la siguiente descripción: *Este es un Sistema DEMO para manejar contactos personales*.
8. El **Package name** es escrito por defecto según el grupo y nombre del artefacto, pero hay quienes deciden dejar este punto igual que el grupo, el cual, no es problema; para nuestro ejemplo dejaremos el nombre generado de forma automática: `com.sofka.contactos`
9. Ahora seleccionaremos el tipo de empaquetado a usar, por defecto estará seleccionado el empaquetado de tipo **Jar**, el cual podemos usar y para este ejercicio será el que usaremos ya que no es necesario crear un empaquetado de tipo **War**, empaquetado que se requería anteriormente para las aplicaciones web.
10. Ahora seleccionaremos la versión de Java a utilizar, esta versión estará ligada a las directrices y exigencias del proyecto que se está desarrollando; en nuestro caso utilizaremos la versión 17.
11. Ahora seleccionaremos todas las dependencias que requiramos para la creación de nuestro proyecto. En nuestro caso seleccionaremos las siguientes dependencias:
  - a. **Spring Boot DevTools**, esto nos permite reiniciar más fácilmente nuestro servidor de aplicaciones.

- b. **Lombok**, esta librería es simple pero poderosa, ya que nos permite reducir escritura de código con los métodos Get y Set dentro de una clase de Java.
  - c. **Spring Web**, esta dependencia permite crear aplicaciones de tipo RESTful.
  - d. **Spring Data JPA**, esta dependencia se encargará de realizar las transacciones con nuestra base de datos y también la usaremos para crear la capa de lógica de negocio.
  - e. **MySQL Driver**, esta dependencia será la que en conjunto con Spring Data JPA se encargarán de realizar la conexión, en este caso, a la base de datos de MySQL.
- Aquí es donde puedes escoger el controlador de base de datos que requieras según el motor de bases de datos relacional.

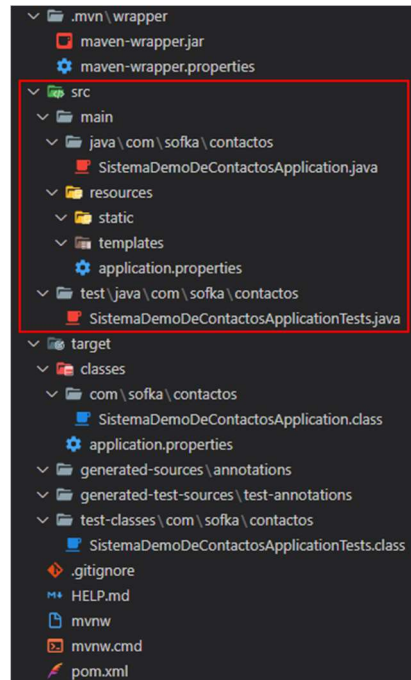


Fuente: <https://ed.team/comunidad/motores-de-bases-de-datos-relacionales>

## 12. Una vez se tenga el proyecto configurado, procedemos a realizar la descarga pertinente.

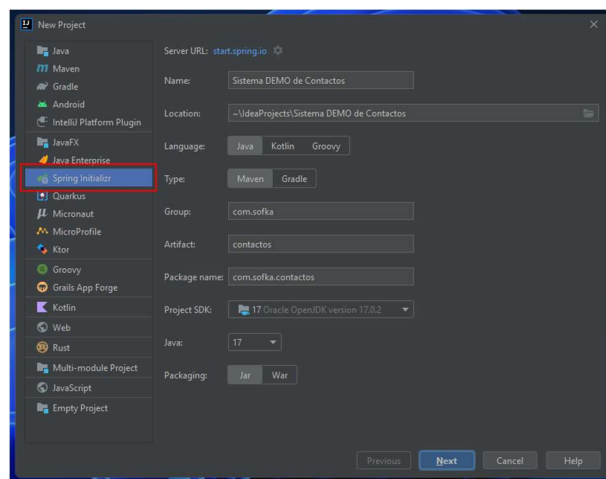
Cuando la descarga del archivo .zip se genere, se debe descomprimir y la carpeta que da como resultado la colocaremos en el lugar que deseamos para empezar a programar.

En nuestra aplicación de ejemplo, el resultado de carpetas y archivos sería el siguiente, siendo la carpeta *src* la que más estaremos usando que es donde reposará el código que escribiremos.



## Cómo crear el proyecto con IntelliJ IDEA Ultimate (plugin Spring Boot)

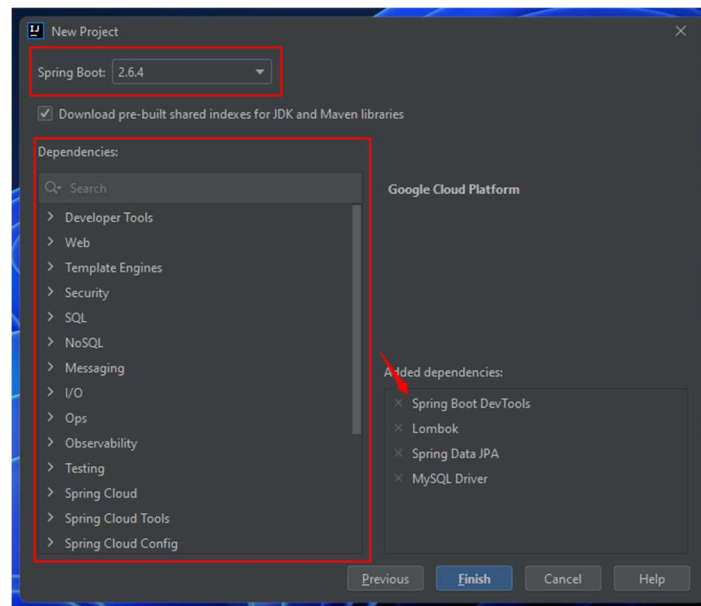
En el caso de IntelliJ IDEA Ultimate, indicamos que vamos a crear un nuevo proyecto y luego seleccionamos Spring Initializr como se puede observar en la siguiente imagen.



Como ya observamos en el apartado de [Cómo crear el proyecto con Spring Initializr \(página web\)](#), realizaremos las configuraciones pertinentes y daremos clic en el botón **Next** o **Siguiente**.

Paso seguido, seleccionaremos la **versión de Spring Boot a usar**, recuerde que siempre usar la versión estable o la que el proyecto demande, y luego buscamos y seleccionamos las

dependencias anteriormente mencionadas para nuestro ejemplo, es decir, **seleccionamos las dependencias que nuestro proyecto requiera.**



Y para terminar hacemos clic en el botón **Finish** o **Finalizar**.

### Notas para tener en cuenta

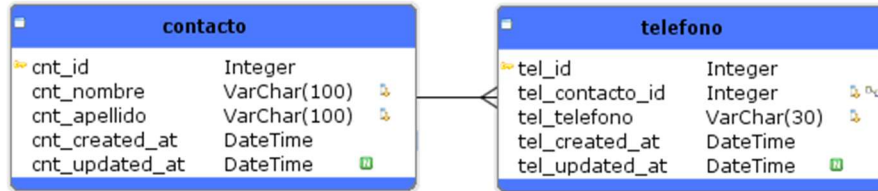
En el caso de **IntelliJ IDEA Ultimate**, cuando terminemos de crear el proyecto, el IDE hará una serie de recomendaciones para descargar librerías, el cual, se recomienda mucho aceptar las descargas e incluso un indexado de dichas librerías si el IDE lo sugiere.

El IDE también recomendará la instalación del plugin **JPA Buddy**, el cual se recomienda aceptar dicha instalación.

Dado que **IntelliJ IDEA Ultimate** es un IDE de pago y que cuenta con una DEMO totalmente funcional por 30 días, y si usted posee algún correo con la terminación **.edu.co**, ejemplo: **miCorreo@universidad.edu.co**, entonces puede solicitar una licencia educativa totalmente gratis en el siguiente enlace: <https://www.jetbrains.com/es-es/community/education/>. Tenga muy presente que dicha licencia es exclusivamente para uso educativo, no para uso comercial.

## Diagrama de Entidad-Relación

Para nuestro ejercicio usaremos una base de datos llamada *libreta*, en la cual contaremos con dos tablas: *contacto* y *teléfono*. Estas tablas contarán con una relación de uno a muchos, es decir, un contacto puede tener muchos teléfonos. También se tendrá presente el tema del borrado lógico y borrado físico.



A continuación, se muestra el script SQL para MySQL, el cual, corresponde al anterior diagrama entidad-relación.

```
/****** Add Table: contacto *****/
/* Build Table Structure */
CREATE TABLE contacto
(
  cnt_id INTEGER UNSIGNED AUTO_INCREMENT PRIMARY KEY NOT NULL,
  cnt_nombre VARCHAR(100) NOT NULL,
  cnt_apellido VARCHAR(100) NOT NULL,
  cnt_created_at DATETIME DEFAULT CURRENT_TIMESTAMP NOT NULL,
  cnt_updated_at DATETIME NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;


/* Add Indexes */
CREATE INDEX contacto_cnt_apellido_idx ON contacto (cnt_apellido) USING BTREE;
CREATE INDEX contacto_cnt_nombre_idx ON contacto (cnt_nombre) USING BTREE;
CREATE UNIQUE INDEX contacto_cnt_nombre_cnt_apellido_idx ON contacto (cnt_nombre, cnt_apellido) USING BTREE;

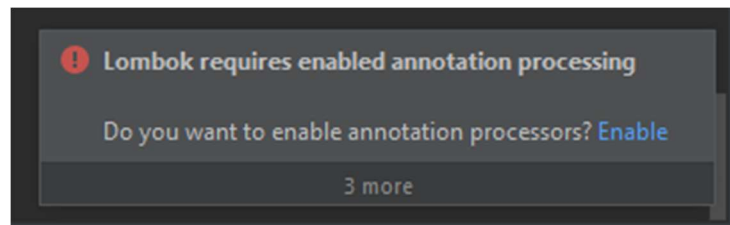
/****** Add Table: telefono *****/
/* Build Table Structure */
CREATE TABLE telefono
(
  tel_id INTEGER UNSIGNED AUTO_INCREMENT PRIMARY KEY NOT NULL,
  tel_contacto_id INTEGER UNSIGNED NOT NULL,
  tel_telefono VARCHAR(30) NOT NULL,
  tel_created_at DATETIME DEFAULT CURRENT_TIMESTAMP NOT NULL,
  tel_updated_at DATETIME NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;

/* Add Indexes */
CREATE UNIQUE INDEX telefono_tel_contacto_id_tel_telefono_idx ON telefono (tel_contacto_id, tel_telefono) USING BTREE;
CREATE INDEX telefono_tel_telefono_idx ON telefono (tel_telefono) USING BTREE;
CREATE INDEX telefono_tel_usuario_id_idx ON telefono (tel_contacto_id) USING BTREE;

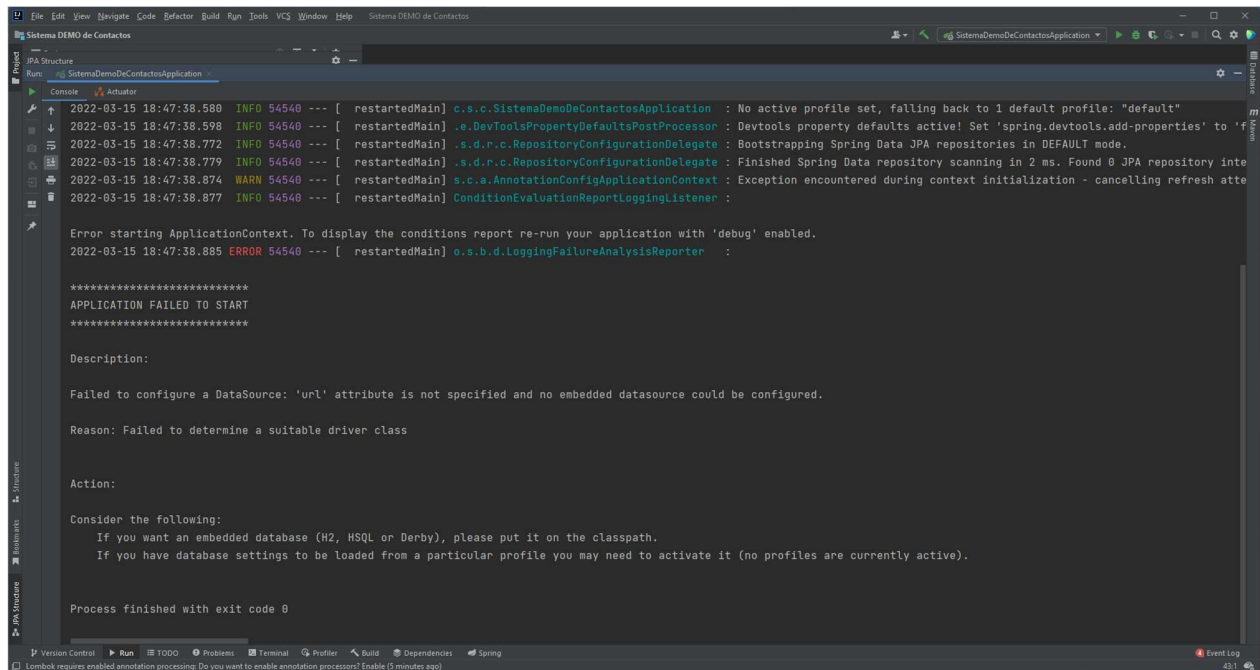
/****** Add Foreign Keys *****/
/* Add Foreign Key: fk_telefono_contacto */
ALTER TABLE telefono ADD CONSTRAINT fk_telefono_contacto
  FOREIGN KEY (tel_contacto_id) REFERENCES contacto (cnt_id)
  ON UPDATE CASCADE ON DELETE RESTRICT;
```

## Configuración inicial del aplicativo

Una de las primeras cosas que solemos hacer después de haber iniciado nuestro proyecto, es hacer clic en el botón de **play** , pero al realizar dicha acción, encontraremos varias cosas, la primera es que nos saldrá un mensaje como el siguiente, el cual recomiendo hacer clic en **Enable** para que se active el procesador de las anotaciones para nuestra dependencia **Lombok** en este caso.



Acto seguido encontraremos el siguiente error.

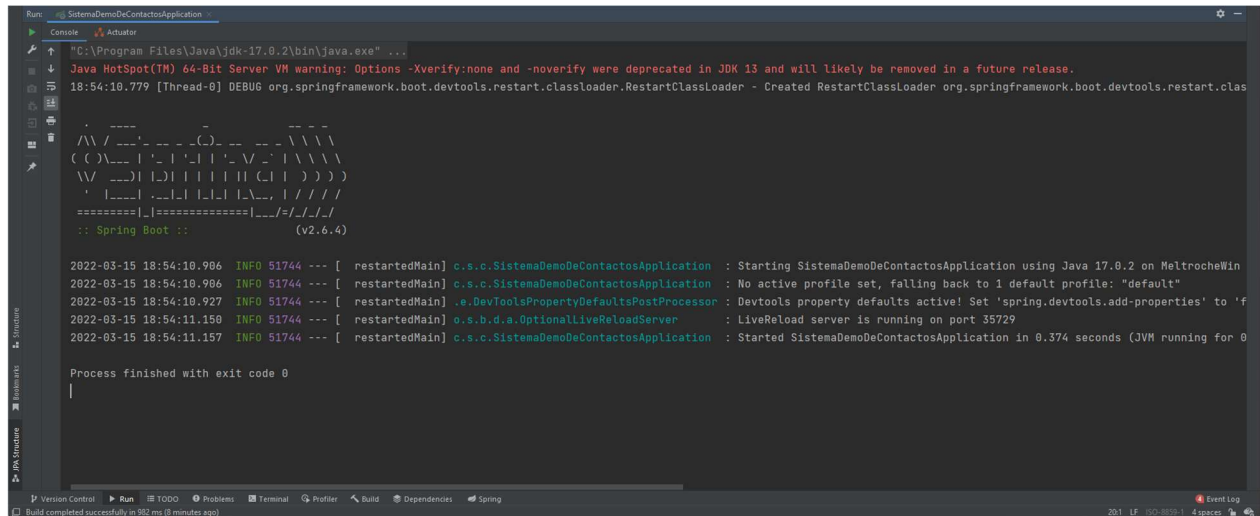


Lo anterior se puede solucionar fácilmente y de forma momentánea ya que en realidad requerimos realizar las configuraciones necesarias para con la base de datos. Para poder dar solución temporal, debemos abrir el archivo **application.properties** ubicado en **src/main/resources** y escribimos la siguiente línea.



```
spring.autoconfigure.exclude = org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration
```

Guardamos y volvemos a ejecutar. Con lo anterior deberíamos no tener problema alguno y obtener en la consola algo parecido a la siguiente imagen.



```
Run - SistemaDemoDeContactosApplication
Console
C:\Program Files\Java\jdk-17.0.2\bin\java.exe ...
Java HotSpot(TM) 64-Bit Server VM warning: Options -Xverify:none and -noverify were deprecated in JDK 13 and will likely be removed in a future release.
18:54:10.779 [Thread-0] DEBUG org.springframework.boot.devtools.restart.classloader.RestartClassLoader - Created RestartClassLoader org.springframework.boot.devtools.restart.clas

  ____ _
 / ___ \| | | |
/ /___ \| |_| |
 \___ \|____|_|_|
   Spring Boot :: (v2.6.4)

2022-03-15 18:54:10.906 INFO 51744 --- [ restartedMain] c.s.c.SistemaDemoDeContactosApplication : Starting SistemaDemoDeContactosApplication using Java 17.0.2 on MeltrocheWin
2022-03-15 18:54:10.906 INFO 51744 --- [ restartedMain] c.s.c.SistemaDemoDeContactosApplication : No active profile set, falling back to 1 default profile: "default"
2022-03-15 18:54:10.927 INFO 51744 --- [ restartedMain] .e.DevToolsPropertyDefaultsPostProcessor : Devtools property defaults active! Set 'spring.devtools.add-properties' to 'f
2022-03-15 18:54:11.150 INFO 51744 --- [ restartedMain] o.s.b.d.a.OptionalLiveReloadServer : LiveReload server is running on port 35729
2022-03-15 18:54:11.157 INFO 51744 --- [ restartedMain] c.s.c.SistemaDemoDeContactosApplication : Started SistemaDemoDeContactosApplication in 0.374 seconds (JVM running for 0

Process finished with exit code 0
```

## Nuestro archivo application.properties

Ahora bien, en este archivo pondremos en comentario la línea anteriormente digitada por medio del signo numeral al inicio de la línea y utilizaremos la siguiente configuración.

```
# Configuración inicial para evitar el error de que no se ha configurado una base de datos
# spring.autoconfigure.exclude=org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration

# Configuración del puerto que usará nuestra aplicación, por defecto es 8080
# server.port = 9090

# Habilitación del LiveReload cuando sea posible
spring.devtools.livereload.enabled = true

# ----- Configuración de conexión a MySQL -----

# Configuración para MySQL 8
spring.datasource.url = jdbc:mysql://localhost/libreta?useSSL=false&serverTimezone=UTC&allowPublicKeyRetrieval=true

# Usuario de base de datos
spring.datasource.username = root

# Contraseña para el usuario de la base de datos
spring.datasource.password = 1234567890

# Clase a usar para conectar con la base de datos
spring.datasource.driver-class-name = com.mysql.cj.jdbc.Driver

# Dialecto a usar de SQL, en este caso MySQL8
spring.jpa.database-platform = org.hibernate.dialect.MySQL8Dialect

# ----- Mostrar en consola el SQL que se está ejecutando -----

# Formatear salida del SQL en consola
spring.jpa.properties.hibernate.format_sql = true

# Hacer que se muestre en consola el SQL ejecutado
```

```
logging.level.org.hibernate.SQL = DEBUG  
  
# Hacer que se muestre en consola los valores que se inyectan a la sentencia SQL  
logging.level.org.hibernate.type.descriptor.sql.BasicBinder = TRACE
```

Sí se requiere configurar alguna variable de entorno para nuestra aplicación, este es el archivo indicado.

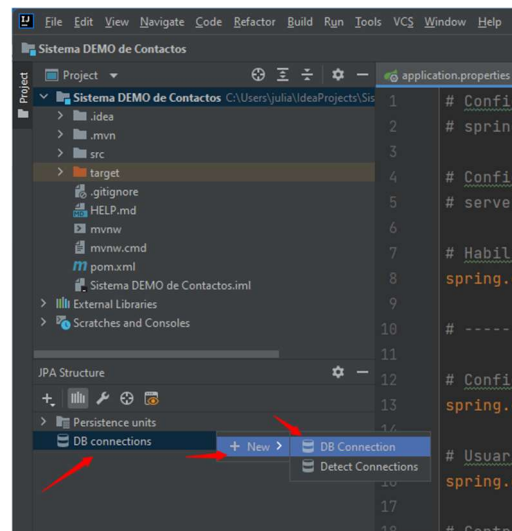
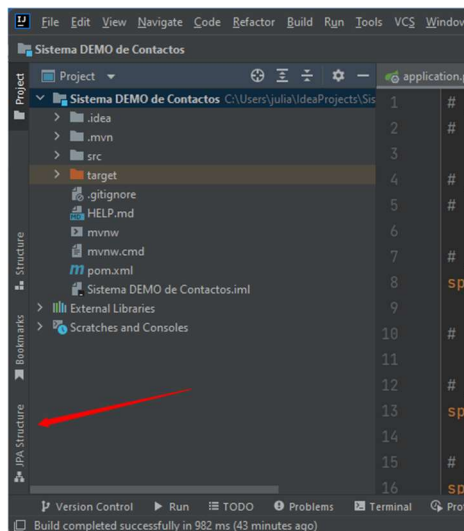
## Nuestro IDE conectado a la base de datos

Para tener una mejor integración de nuestro IDE con la base de datos del proyecto, es importante realizar dicha conexión ya que a futuro nos permitirá flexibilizar la creación de los archivos de conexión a la base de datos y los archivos necesarios para el patrón de diseño DTO (Data Transfer Object), el cual, explicaremos con más detalle más adelante.

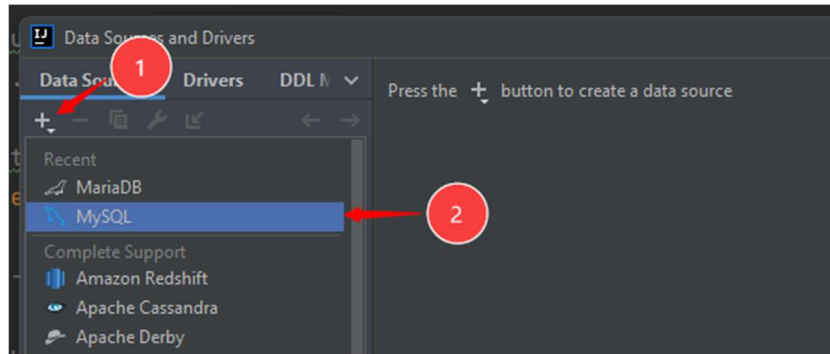
Para realizar dicha conexión tenemos dos opciones, crear dicha conexión desde cero o decirle a IDE que intente detectar la configuración de la conexión basado en nuestro archivo **application.properties**.

## Configurar la conexión a la base de datos desde cero

Para poder realizar dicha configuración, debemos entrar en el apartado **JPA Structure** y luego hacer clic derecho en **DB connections** -> **New** -> **DB Connection**, así como se muestra en las siguientes imágenes.

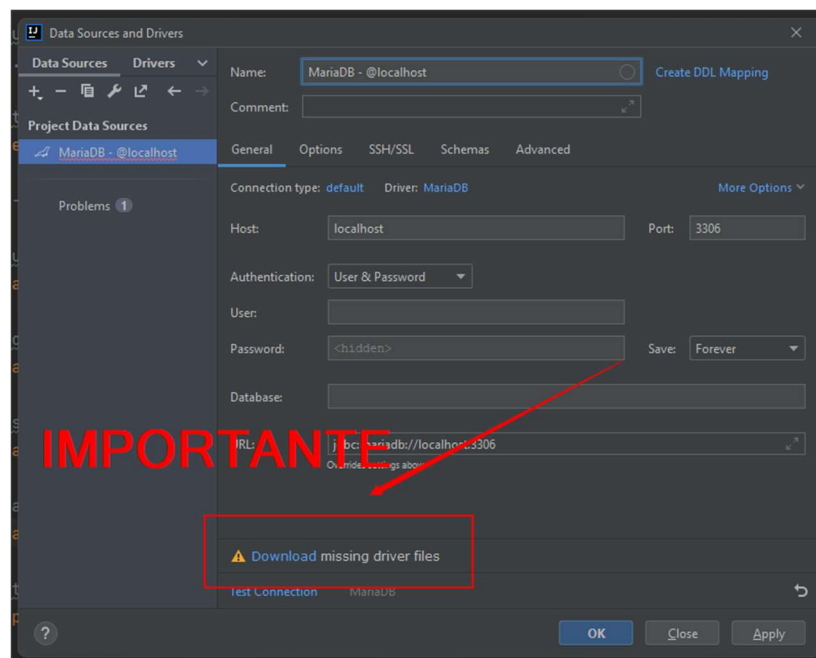


Cuando se ha realizado lo anterior, es momento de agregar la conexión a la base de datos y para ellos debemos indicar que se quiere agregar dicha conexión, así como se muestra en la siguiente imagen.

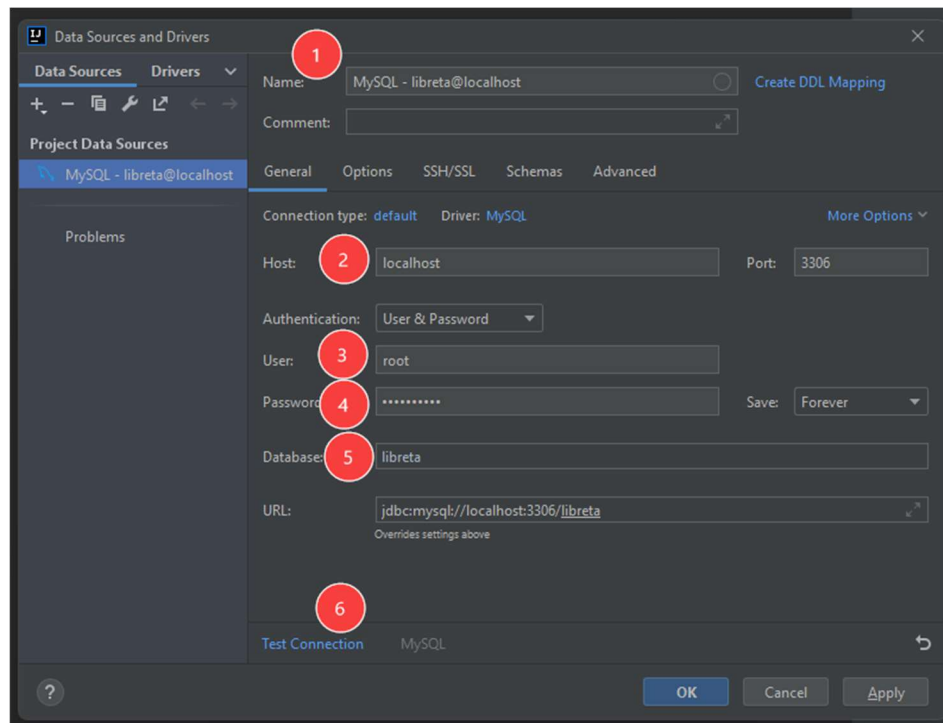


1. Se hace clic en el signo más ( + )
2. Se busca el motor de base de datos a usar

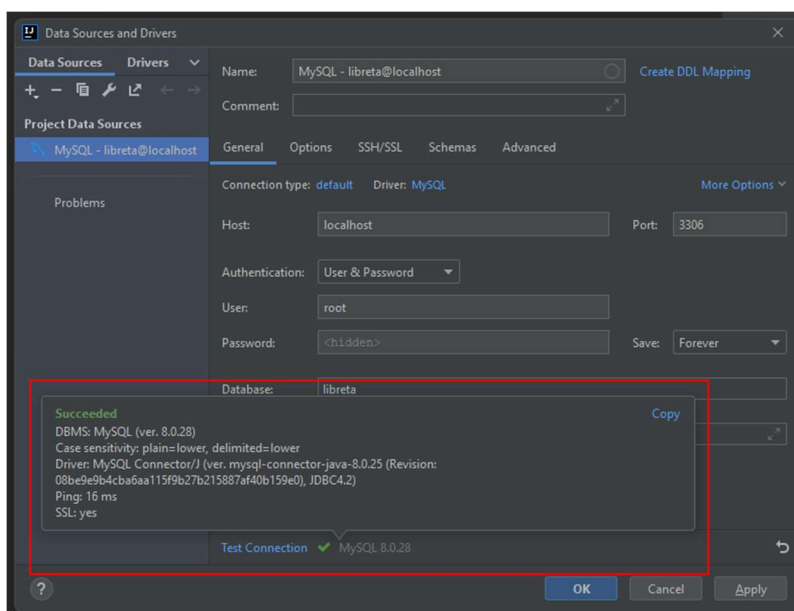
Antes de configurar no olvide descargar el controlador necesario para realizar la conexión del IDE a la base de datos deseada.



Una vez se haya descargado el controlador requerido, pasamos a configurar la conexión hacia la base de datos.

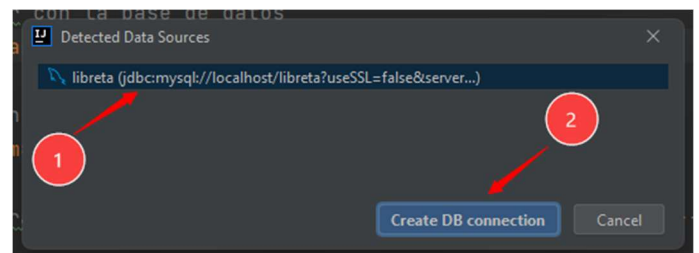
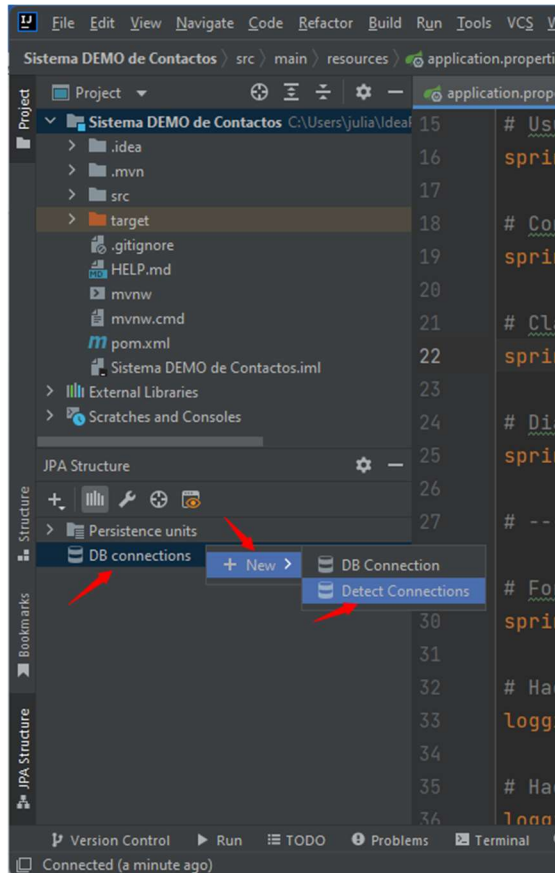


1. Nombre de la conexión
2. IP o nombre del host donde se encuentra la base de datos
3. Usuario de la base de datos
4. Contraseña del usuario de la base de datos
5. Nombre de la base de datos o esquema al cual nos conectaremos
6. Por último, testeamos la conexión esperando que dicha prueba sea exitosa



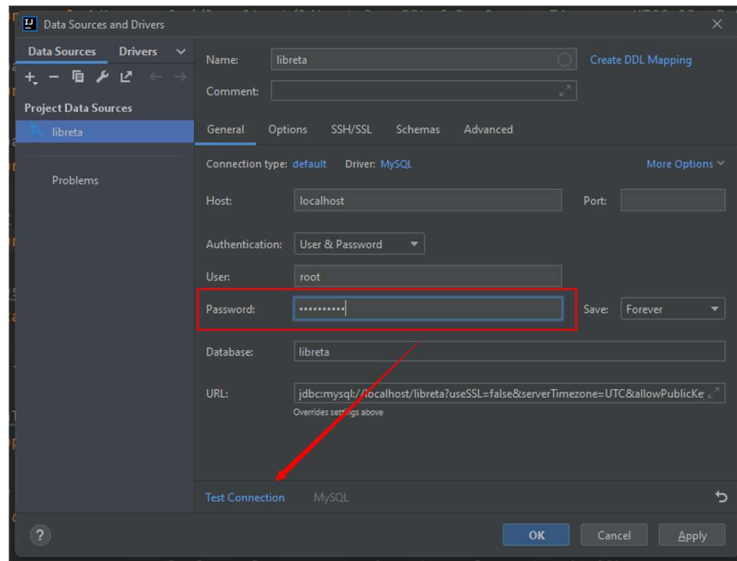
## Configurar la conexión a la base de datos mediante la detección del IDE

Para hacer que nuestro IDE detecte la conexión a nuestra base de datos basado en la configuración debemos ubicar el apartado **JPA Structure** y luego hacer clic derecho en **DB connections** -> **New** -> **Detect Connections**.




1. Verificamos que se haya detectado correctamente la conexión
2. Se hace clic en **Create DB connection**

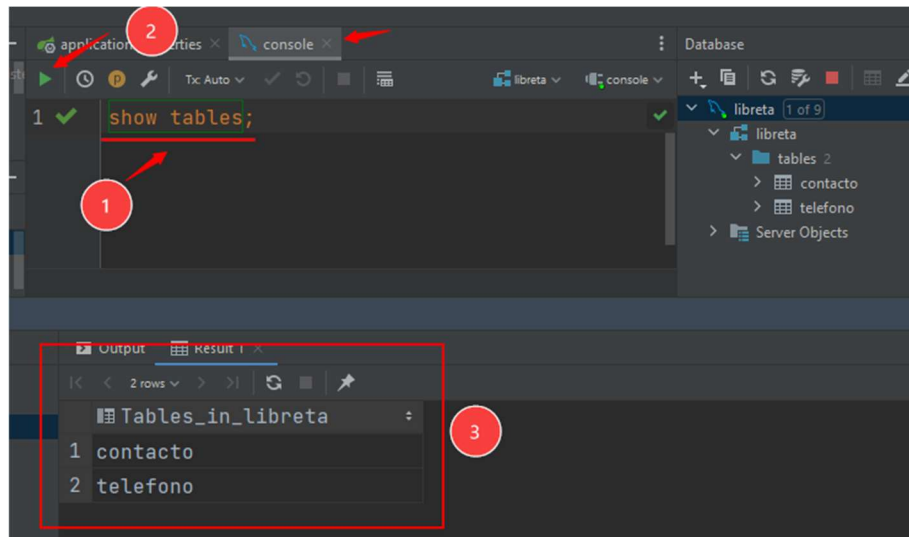
A diferencia de la configuración manual, este tipo de configuración detectará todo, menos la contraseña, la cual, se debe escribir y testear dicha conexión esperando que la prueba sea exitosa.



## Probando la configuración a la base de datos desde nuestro IDE

Cuando hemos terminado de realizar la configuración, se abrirá una pestaña más en nuestro IDE, donde podremos escribir sentencias SQL y ejecutarlas para poder ver su resultado.

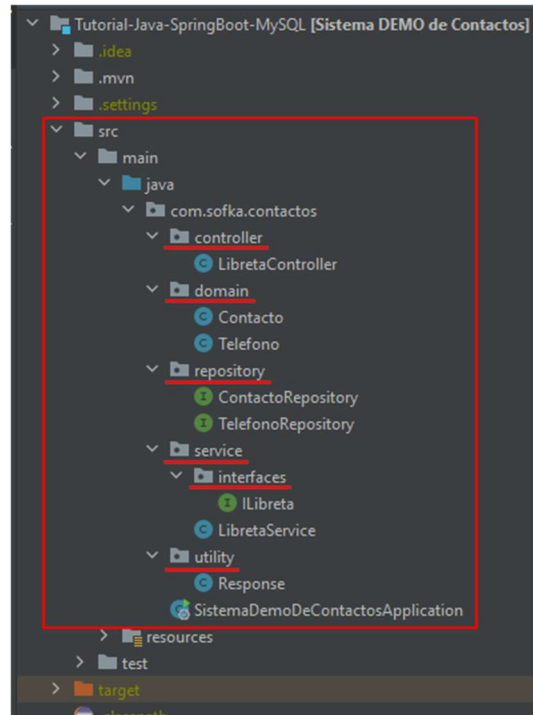
(1) En este caso digitaremos *show tables;* seguido de (Ctrl + Enter) o (2) clic en el botón **play**  para ejecutar la sentencia SQL y (3) poder ver su resultado como se muestra en la siguiente imagen.



Una vez logrado esto, estamos listos para empezar a programar.

## Sistema de paquetes

Para nuestro ejemplo usaremos el siguiente sistema de paquetes; este sistema de paquetes es solo una opción, en realidad esto se define en el grupo de trabajo cuando se inicializa el proyecto para así llevar una arquitectura acorde al proyecto a programar, muchas veces dictada por el arquitecto del proyecto o cualquier otra persona con el rol indicado para dicha tarea.



**Controller**, aquí almacenaremos los controladores que se encargarán de recibir, orquestar el proceso necesario y responder las solicitudes de nuestras API.

**Domain**, aquí pondremos todas las entidades correspondientes a las tablas de la base de datos. Si se desea usar patrón de diseño **DTO (Data Transfer Object)** este sería el espacio indicado para dichos archivos, lógicamente cambiaríamos el nombre de **domain** por **dto**.

**Repository**, aquí almacenaremos los archivos que se encargarán de tener acceso a la fuente de los datos, como también será el sitio indicado donde se almacenarán las consultas personalizadas que necesitamos realizar a la base de datos. Equivalente al patrón **DAO (Data Access Object)**.

**Service.Interfaces**, aquí estarán las interfaces que se implementarán en los diferentes servicios.

**Service**, aquí colocaremos los servicios que se prestan con relación a la base de datos y demás aspectos que se relacionan con la ejecución de este, en otras palabras, aquí escribiremos la lógica del negocio, la cual, cada servicio debería implementar su respectiva interfaz.

**Utility**, aquí guardaremos todas esas clases que iremos construyendo y que serán de utilidad para nuestro sistema en general, por ejemplo, la clase que usaremos para responder en las API.

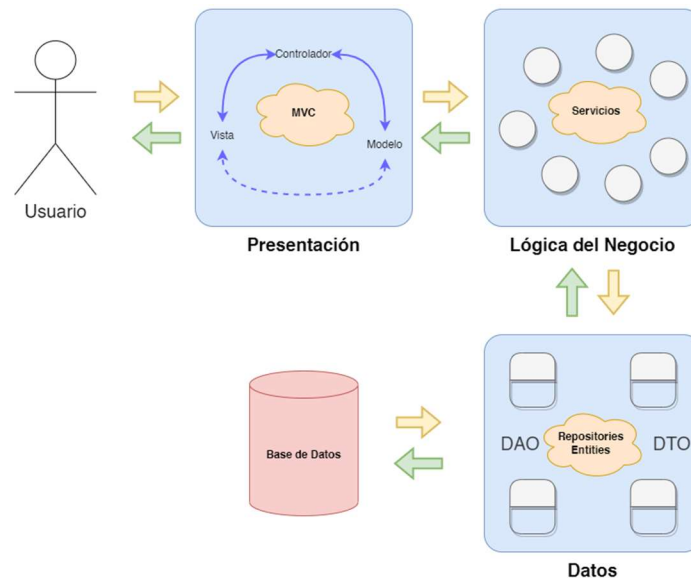
## **Lógica del sistema de contactos**

Nuestro sistema de ejemplo solo consta de dos tablas y el sistema deberá realizar las siguientes tareas:

1. Insertar (Post) un nuevo contacto con X cantidad de números de teléfono.
2. Obtener (Get) el listado de todos los contactos en el orden alfabético por nombre o apellido ya sea ascendente o descendente.
3. Obtener (Get) el listado de todos los contactos en el orden que fueron insertados en la base de datos.
4. Modificar (Put) toda la tupla de un contacto basado en su ID.
5. Modificar (Patch) un solo campo (teléfono, nombre o apellido) de un contacto basado en su ID.
6. Borrar (Delete) un contacto basado en su ID según se indique.
7. Buscar (Get) un contacto basado en el nombre o apellido.



## Explicación de capas a usar



Para nuestro ejemplo usaremos la siguiente arquitectura, la cual, consiste en 3 capas muy sencillas:

**Presentación**, en esta capa se trabajará el patrón MVC, aunque en realidad solo trabajaremos la parte del controlador.

**Lógica del Negocio**, esta capa será la encargada de las operaciones principales en el sistema como el guardar un nuevo dato, modificar, eliminar e incluso traer la información requerida de la base de datos.

**Datos**, en esta capa se trabajará todo lo relacionado a la información que viaja de un lado a otro con respecto a la base de datos, es decir, la capa de persistencia de datos.

## Manos a la obra, empecemos a programar

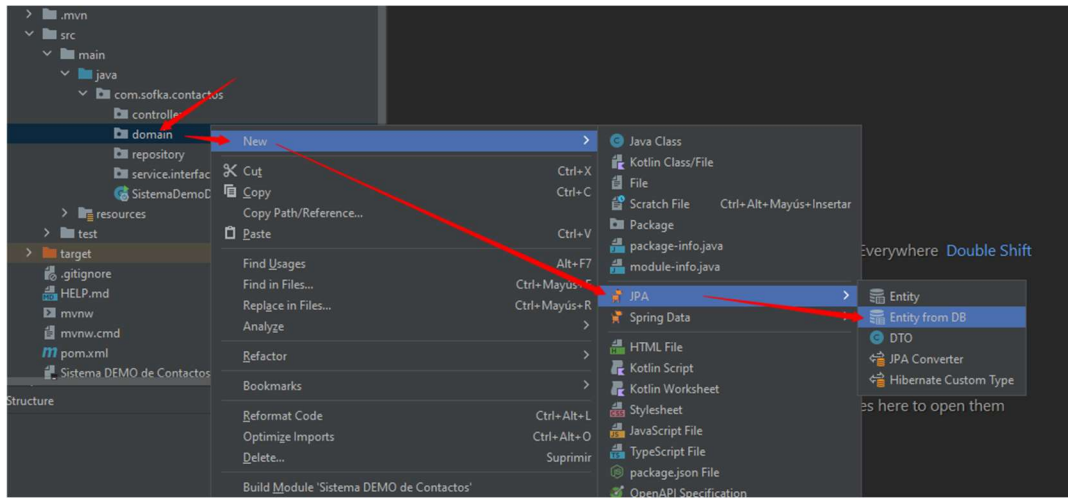
### Capa de persistencia de datos

#### Las entidades

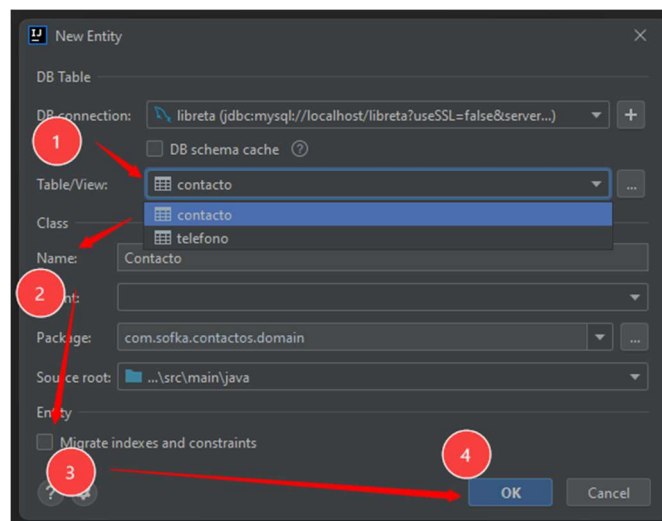
Existen dos maneras de crear las entidades que requerimos con respecto a la base de datos y el diagrama entidad-relación que ya tenemos diseñado, la **primera forma** es crear archivo por archivo de forma manual y la **segunda forma** es aprovechar las ventajas del plugin **JPA Buddy**

y realizar ingeniería inversa para mapear toda la base de datos, en cualquiera de los casos hay que personalizar cada entidad, así que para fines prácticos y pensando que a futuro trabajaremos no con dos tablas sino con muchas más, entonces haremos uso del plugin **JAP Buddy** pero personalizaremos entidad creada.

Para mapear la base de datos con **JPA Buddy** haremos clic derecho sobre el paquete *domain*, luego nos dirigimos a **New -> JPA -> Entity from DB**.

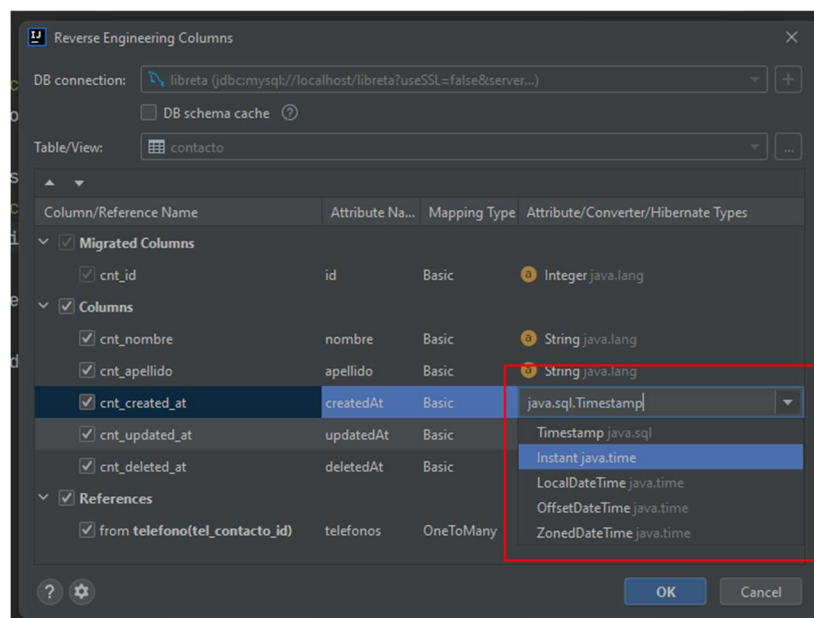
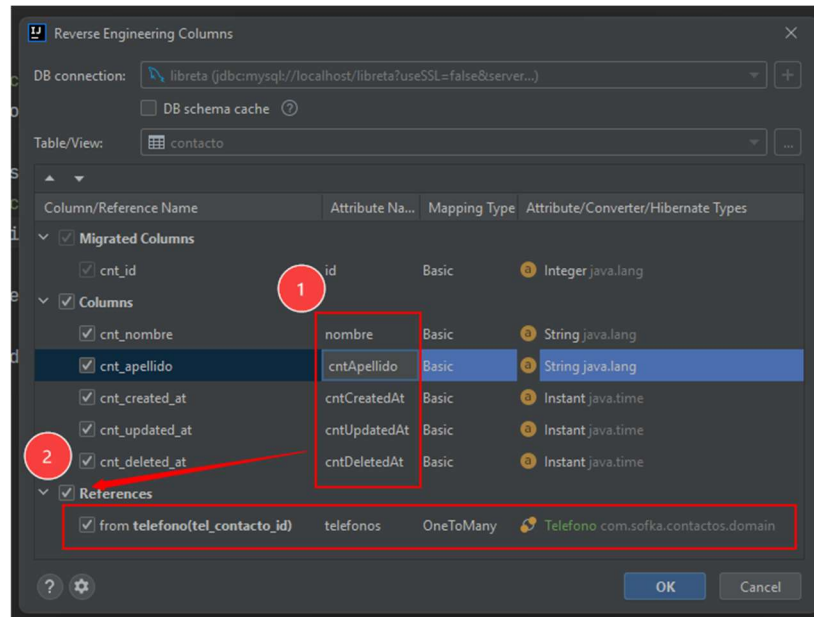


Luego en la siguiente ventana que aparece seleccionaremos la tabla que deseamos mapear, para este ejemplo (1) usaremos la tabla *contacto*, (2) automáticamente el nombre de la clase será el nombre de la tabla con el estilo **CamelCase**, (3) quitamos la selección de **Migrate indexes and constraints** si está marcada y (4) terminamos haciendo clic en el botón **OK**.



En la siguiente ventana es importante marcar (2) **References** en caso de existir referencias a otras tablas, esto permitirá mapear las demás tablas que se encuentren referencias a la seleccionada, de no hacerlo (de echo puede omitir este paso) deberá crear la referencia de forma manual.

También es aconsejable (1) corregir los nombres de cada atributo para evitar cualquier prefijo o sufijo en estilo **camelCase**.



También puede elegir el tipo de dato con el que será tratado cada campo, en el caso de los campos de tipo **DATETIME** usaremos para este ejemplo **Instant** del paquete **java.time**.

Una vez realizada esta operación, ahora tendremos dos archivos .java en el paquete *domain* **Contacto.java** y **Telefono.java**. El mapeo objetivo era la tabla Contacto y ahora debemos personalizar dicho resultado de la siguiente manera.

1. Borraremos todos los **setters** y **getters** creados ya que usaremos la anotación **@Data** de la librería **Lombok**.
2. Implementaremos, en la clase creada, la interfaz **Serializable** del paquete **java.io**
3. Y crearemos el atributo **serialVersionUID** de la siguiente manera:

```
private static final long serialVersionUID = 1L;
```

5. El plugin utiliza la siguiente forma `import javax.persistence.*;` para importar todos objetos, interfaces y demás del paquete, por favor y como buena práctica, borre esa línea e importe exclusivamente los paquetes a usar sin importar la cantidad que esté usando.

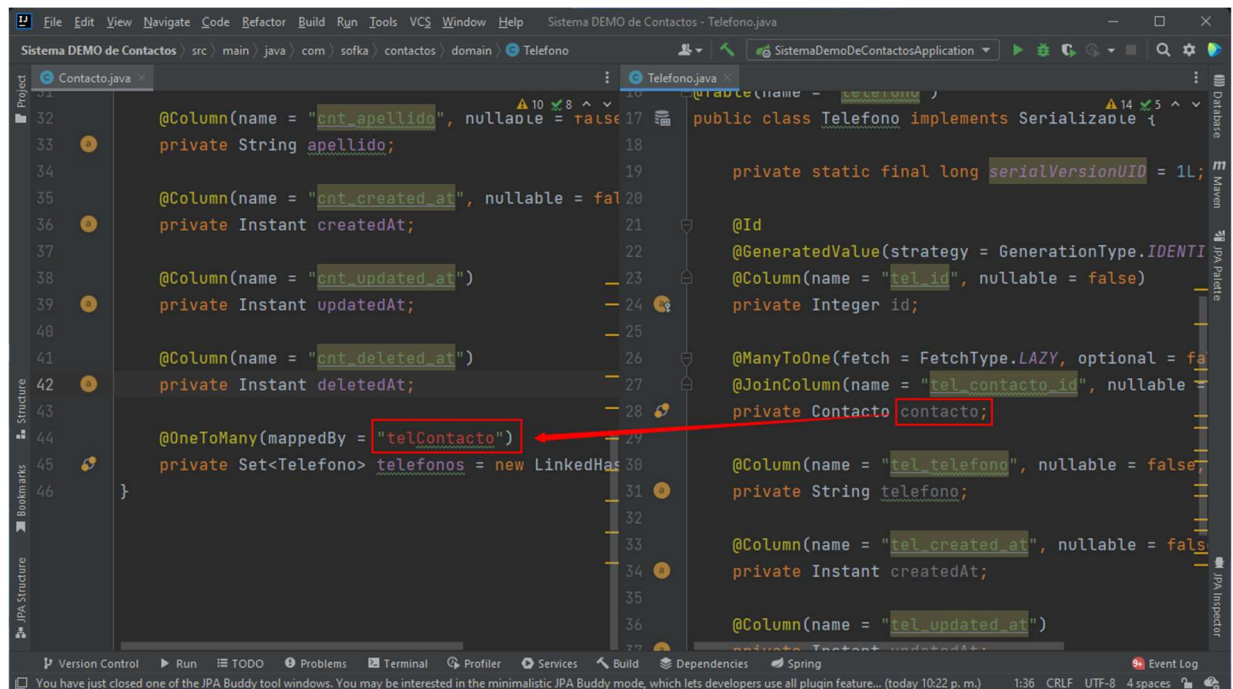
Dado lo anterior y como resultado en la entidad Contacto, tendríamos el código que se visualiza en el siguiente enlace: <https://github.com/training-practice-sofkau/Tutorial-Java-SpringBoot-MySQL/blob/main/src/main/java/com/sofka/contactos/domain/Contacto.java>

Ahora si vamos al archivo **Telefono.java**, encontraremos que dicho archivo está incompleto con respecto a la tabla en la base de datos y por ende tenemos dos opciones, completarlo manualmente o borrarlo, mapearlo nuevamente con **JPA Buddy** y personalizarlo, así como hicimos con el archivo anterior. ¡Usted decide!

Puede visualizar el resultado del archivo Telefono.java en el siguiente enlace: <https://github.com/training-practice-sofkau/Tutorial-Java-SpringBoot-MySQL/blob/main/src/main/java/com/sofka/contactos/domain/Telefono.java>

### **Nota para tener en cuenta**

Cuando cree el archivo Telefono.java, debe de tener presente actualizar en el archivo Contacto.java el campo que mapea la relación Uno a Muchos.



## Maapeo de las relaciones

Por medio de las anotaciones (`@Data`, `@Entity`, entre otras) que proporciona JPA cuando se usa Hibernate, se puede gestionar las relaciones entre dos tablas como si se tratase de objetos. Lo anterior facilita el mapeo de los atributos de la base de datos hacia el modelo de objetos de la aplicación, el dominio en este caso. Dependiendo de la lógica de negocio y cómo se haya modelado la base de datos, se podrán crear relaciones unidireccionales o bidireccionales.

**¿Qué es Hibernate?** Es una herramienta de software libre para el mapeo objeto-relacional, es decir, un **ORM** para la plataforma Java (aunque también está disponible para .Net con el nombre de NHibernate), el cual, facilita el mapeo de atributos desde una base de datos relacional hacia el modelo de objetos de una aplicación. Esta herramienta busca solucionar el problema de la diferencia entre los dos modelos de datos coexistentes en una aplicación: el usado en la memoria del computador (orientación a objetos) y el usado en las bases de datos (modelo relacional).

Para lograr lo anterior, Hibernate permite al desarrollador detallar cómo es su modelo de datos, qué relaciones existen y qué forma tienen. Con esta información el ORM le permite a la aplicación manipular los datos en la base de datos operando sobre objetos, con todas las

características de la POO. Esta herramienta convertirá los datos entre los tipos utilizados por Java y los definidos por SQL.

También genera las sentencias SQL y libera al desarrollador del manejo manual de los datos que resultan de la ejecución de dichas sentencias, manteniendo la portabilidad entre todos los motores de bases de datos con un ligero incremento en el tiempo de ejecución.

Hibernate está diseñado para ser flexible en cuanto al esquema de tablas utilizado, para así poder adaptarse a su uso sobre una base de datos ya existente. También tiene la funcionalidad de crear la base de datos a partir de la información disponible en los objetos y viceversa.

Este ORM ofrece también un lenguaje de consulta de datos llamado HQL (Hibernate Query Language), al mismo tiempo que una API para construir las consultas por medio de programación (conocida como "criteria").

Por último, esta herramienta para Java puede ser utilizada en aplicaciones Java independientes o en aplicaciones Java EE (Enterprise Edition), mediante el componente Hibernate Annotations que implementa el estándar JPA, que es parte de esta plataforma.

**¿Qué es JPA?** Java Persistence API es una especificación (concretamente la [JSR 338](#)), la cual, no es más que un documento en el que se define cómo se debe gestionar una X funcionalidad. En este caso una capa de persistencia con objetos Java.

Para poder trabajar con JPA, es necesario tener un ORM o Framework que implemente las especificaciones, uno de los más conocidos es Hibernate.

¿Existen más implementaciones de JPA? Algunas de las más conocidas son:

- [EclipseLink](#), una implementación muy usada ya que cuenta con el soporte de la fundación Eclipse y da soporte a JPA.
- [DataNucleus](#), otro ORM de persistencia que soporta JPA, pero incluye muchas más opciones y tipos de persistencia.
- [TopLink](#), esta es la implementación de Oracle para sus productos. Hoy en día está basada en EclipseLink.
- [ObjectDB](#), esta es otra implementación de JPA, el cual, promete un rendimiento alto comparado con sus competidores.

**¿Usar JPA o Hibernate?** Esta pregunta es un clásico en la programación de software en cuanto a Java, es decir, usar las especificaciones o usar directamente el ORM.

**¿Usar Hibernate?** Existen programadores que prefieren usar el ORM de forma directa ¿por qué? Porque el ORM tiene capacidades adicionales que JPA como estándar y especificación no soporta.

**¿Usar la especificación?** Para responder a esto, debemos ver un poco sus ventajas.

1. La especificación está muy difundida y a nivel cultural, muy extendida entre todos los desarrolladores, a comparación de que el uso de un ORM no lo es tanto (recuerde que estas son simplemente herramientas que van y vienen).
2. La especificación permitirá cambiar de implementación de forma transparente si tenemos la necesidad. Es raro tener que hacer algo como lo anterior, pero existe la posibilidad y esto permitirá portar de forma más sencilla la aplicación entre diferentes servidores Java EE que usen diferentes especificaciones.
3. La especificación es mucho más atemporal ya que es un documento y todo el mundo se ciñe a él. Por lo tanto, lo aprendido dura más en el tiempo y es más sólido al paso de las décadas.
4. Al ser algo tan inmutable es normal que aparezcan nuevos ORM o nuevas especificaciones que lo complementen. Por ejemplo, un caso claro de JPA es el uso de Spring Data con JPA que permite un trabajo mucho más cómodo y rápido con la implementación de Repositorios.

**Conclusión,** recuerde que usted está aprendiendo para laborar de forma profesional y por tanto es mejor trabajar con JPA como especificación; lo anterior da como resultado que usted puede usar cualquier implementación y para nuestro caso usaremos Hibernate, ya que es la implementación más conocida. No olvide usar solo anotaciones del estándar de JPA y evitar en lo posible hacer uso directo de las funciones de Hibernate, permitiendo mantener la portabilidad del código y ventajas de soluciones que salgan a futuro.

La anterior conclusión es una nota personal y puede cambiar dependiendo de las reglas dictadas para el proyecto a desarrollar en Java con JPA.

## Relaciones

Las relaciones que pueden llegar a existir en el mapeo de una base de datos relacional son:

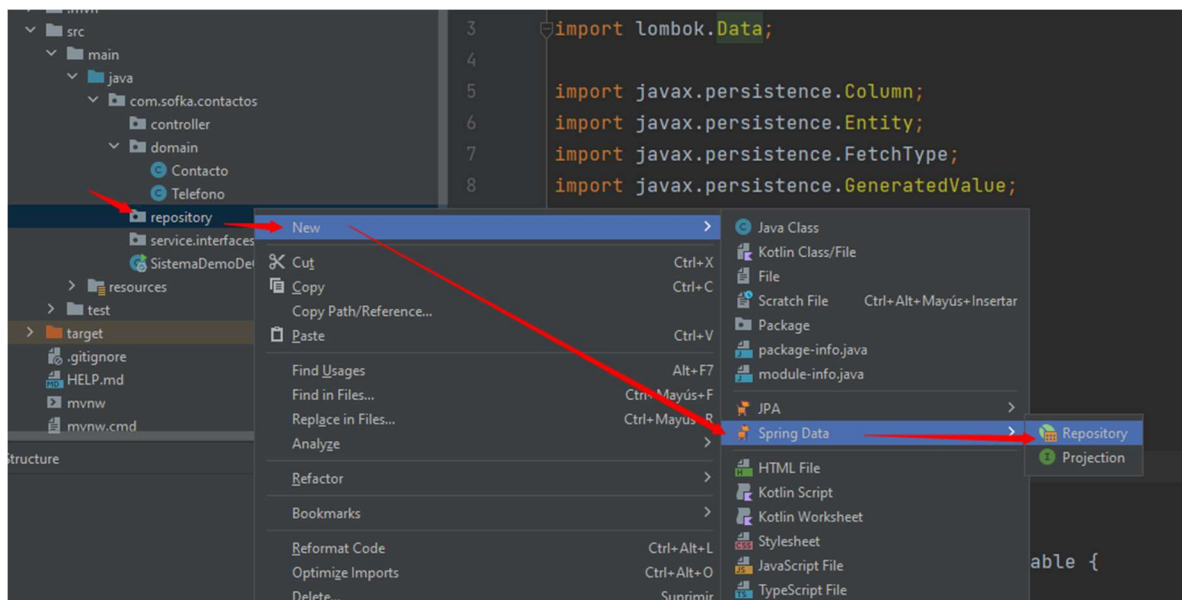
1. Uno a uno (@OneToOne) bidireccional.
2. Uno a muchos (@OneToMany) bidireccional.
3. Uno a muchos (@OneToMany) unidireccional.
4. Muchos a Muchos (@ManyToMany) bidireccional.

Para poder ampliar el conocimiento al respecto, se sugiere la lectura del contenido del siguiente enlace: <https://www.adictosaltrabajo.com/2020/04/02/hibernate-onetoone-onetomany-manytoone-y-manytomany/>

## Los repositorios

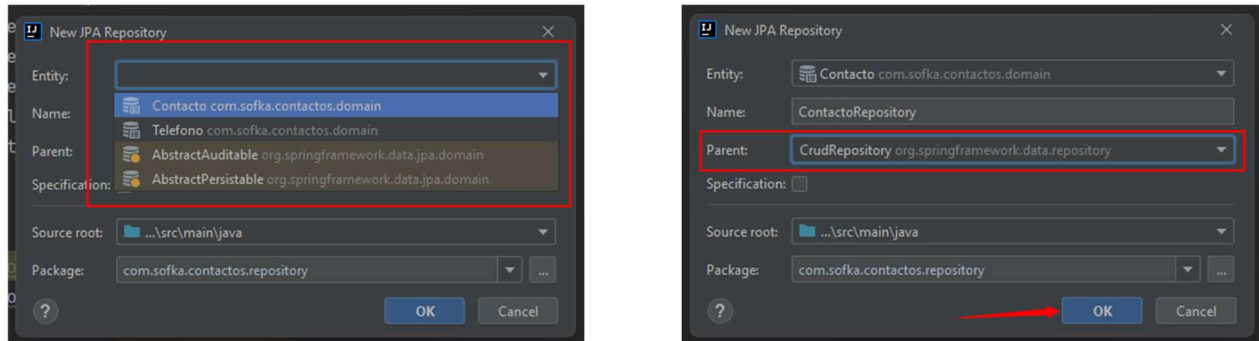
Al igual que cuando se crean las entidades, también existen dos formas de crear los repositorios, **la primera** de forma manual y **la segunda** haciendo uso del plugin **JPA Buddy**. Para hacer uso del plugin debemos hacer los siguientes pasos por cada repositorio a crear. La recomendación es crear un repositorio por cada entidad.

Para crear un repositorio con **JPA Buddy** haremos clic derecho sobre el paquete *repository*, luego nos dirigimos a **New -> Spring Data -> Repository**.





Acto seguido **seleccionaremos la entidad** sobre la cual crearemos el repositorio, luego en **Parent** seleccionaremos la opción **CrudRepository** y finalizamos haciendo clic en el botón **OK**.



Con lo anterior obtenemos un archivo llamado **ContactoRepository.java** con el siguiente código.

```
package com.sofka.contactos.repository;

import com.sofka.contactos.domain.Contacto;
import org.springframework.data.repository.CrudRepository;

public interface ContactoRepository extends CrudRepository<Contacto, Integer> {
}
```

Ahora repetiremos el proceso, creando un repositorio por cada entidad.

### Nota para tener en cuenta

No olvide que en este punto es donde pondremos tanto las consultas como inserciones, actualizaciones y borrados personalizados que tengan que ver con respecto a los datos de la base de datos.

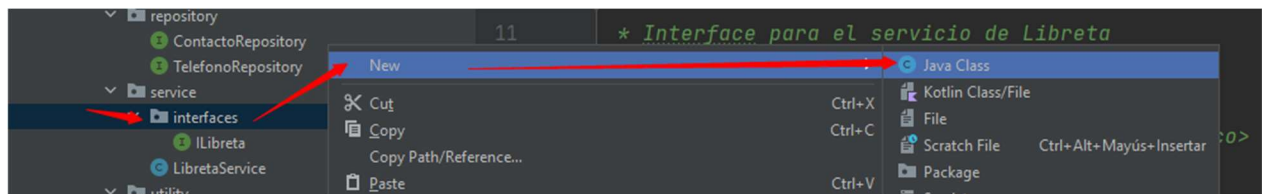
### Capa de servicios

Quizás una de las capas más importantes, sin menospreciar las demás, pero aquí es donde se concentra toda la lógica del negocio, es decir, todo lo que realiza el sistema debería estar concentrado en esta capa.

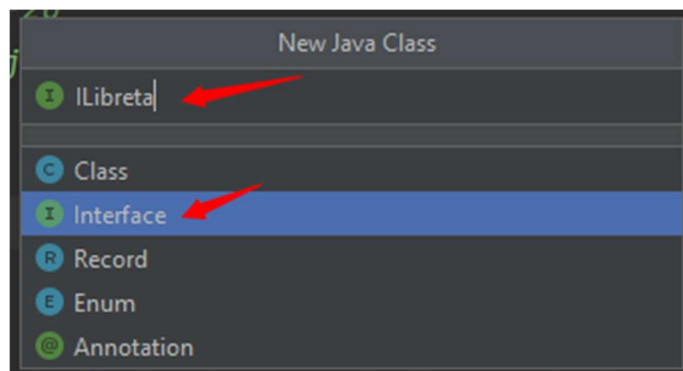
Aunque a veces se da la apariencia de que un servicio responde a una tabla de la base de datos, pero en realidad es mucho más que eso, ya que se puede crear servicios que no depende en forma literal de una tabla, sino que son X o Y proceso en toda la lógica del sistema y puede

involucrar una o más tablas haciendo uso de los repositorios para el tema enviar y recibir información de la base de datos.

Teniendo presente la lógica del sistema de contactos, crearemos una interface en el paquete *service.interfaces*, la cual llamaremos **ILibreta.java**, esta interface contendrá los métodos mínimos que requerimos en un futuro servicio llamado **LibretaService** al ser implementado. Para lo anterior se hará clic derecho sobre el paquete **interfaces** -> **New** -> **Java Class**.



Luego escribiremos *ILibreta*, la *I* mayúscula quiere decir que dicho archivo será una interfaz, después procedemos a seleccionar la opción **Interface** y terminamos pulsando **Enter**.



Con lo anterior obtendremos un código similar al siguiente.

```
package com.sofka.contactos.service.interfaces;  
public interface ILibreta {}
```

Ahora es momento de declarar los futuros métodos mínimos que requiere nuestro futuro servicio **LibretaService** a implementar según el uso que este puede ofrecer.

Nuestra interfaz declarará los métodos necesarios para cumplir entonces con lo siguiente:

1. Devolver una lista de Contactos con todos contactos del sistema.

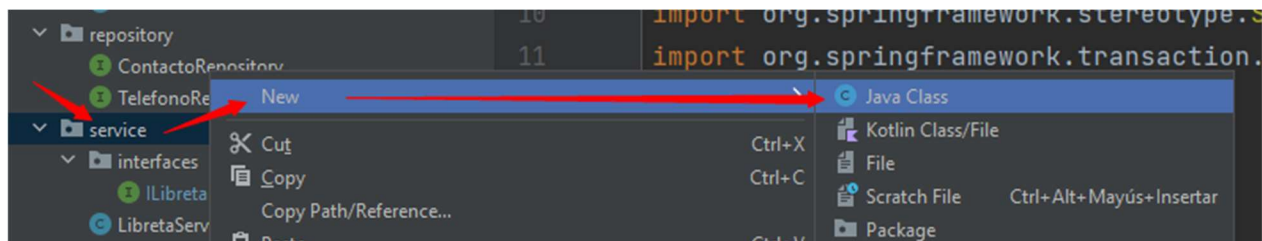
2. Devolver una lista de Contactos con todos contactos del sistema ordenados por el campo indicado (nombre o apellido) ya sea ascendente o descendente.
3. Buscar un dato dado entre el nombre y/o los apellidos en un contacto.
4. Crear un contacto en el sistema.
5. Crear un teléfono en el sistema a nombre de un contacto.
6. Actualizar una tupla completa de un contacto.
7. Actualizar el nombre de un contacto basado en su identificador.
8. Actualizar el apellido de un contacto basado en su identificador.
9. Actualizar la tupla completa de un teléfono en el sistema basado en su identificador.
10. Actualizar solamente el teléfono de un contacto a partir del ID de la tupla del teléfono.
11. Borrar un contacto del sistema basado en su identificador.
12. Borrar un teléfono del sistema basado en su identificador.

Como podrás leer los métodos que declararemos no están relacionados específicamente a una tabla sino a dos, es por esto que se indica que la interfaz de un servicio da la apariencia de que obedece a las acciones que se pueden realizar sobre una tabla, pero en realidad es mucho más que eso.

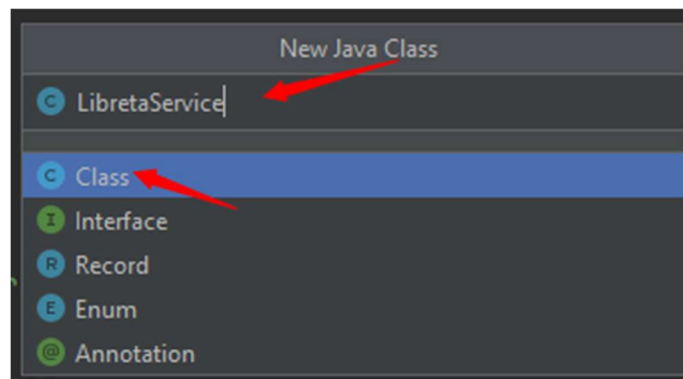
El código completo de la interfaz se puede consultar en el siguiente enlace:

<https://github.com/training-practice-sofkau/Tutorial-Java-SpringBoot-MySQL/blob/main/src/main/java/com/sofka/contactos/service/interfaces/ILibreta.java>

Para completar la capa de servicios ahora debemos crear el archivo **LibretaService.java** en el paquete *service*. Para lo anterior haremos clic derecho sobre **service** -> **New** -> **Java Class**.



En la ventana que sale a continuación, escribiremos **LibretaService**, seleccionaremos que se trata de una clase y finalizamos con un **Enter**.



Con lo anterior, abriremos el archivo e **implementaremos la interfaz ILibreta**, una vez hecho esto, el sistema nos indicará que debemos implementar los métodos de la interfaz, dando como resultado el siguiente código.

```
package com.sofka.contactos.service;

import com.sofka.contactos.domain.Contacto;
import com.sofka.contactos.domain.Telefono;
import com.sofka.contactos.repository.ContactoRepository;
import com.sofka.contactos.repository.TelefonoRepository;
import com.sofka.contactos.service.interfaces.ILibreta;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.Sort;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import java.util.List;

@Service
public class LibretaService implements ILibreta {

    @Autowired
    private ContactoRepository contactoRepository;

    @Autowired
    private TelefonoRepository telefonoRepository;

    @Override
    @Transactional(readOnly = true)
    public List<Contacto> getList() { }

    @Override
    @Transactional(readOnly = true)
    public List<Contacto> getList(String field, Sort.Direction order) { }

    @Override
    @Transactional(readOnly = true)
    public List<Contacto> searchContacto(String dataToSearch) { }

    @Override
    @Transactional
    public Contacto createContacto(Contacto contacto) { }

    @Override
    @Transactional
    public Telefono createTelefono(Telefono telefono) { }

    @Override
```

```

@Transactional
public Contacto updateContacto(Integer id, Contacto contacto) { }

@Override
@Transactional
public Contacto updateNombre(Integer id, Contacto contacto) { }

@Override
@Transactional
public Contacto updateApellidos(Integer id, Contacto contacto) { }

@Override
@Transactional
public Telefono updateTelefono(Integer id, Telefono telefono) { }

@Override
@Transactional
public Telefono updateOnlyTelefono(Integer id, Telefono telefono) { }

@Override
@Transactional
public Contacto deleteContacto(Integer id) { }

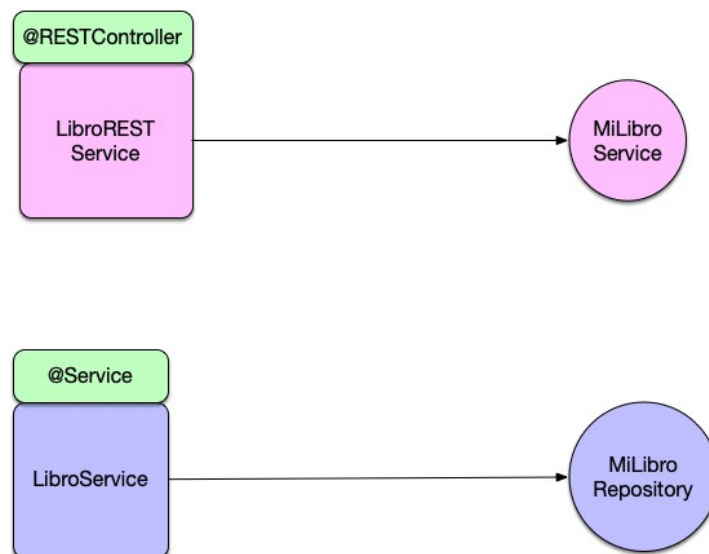
@Override
@Transactional
public Telefono deleteTelefono(Integer id) { }
}

```

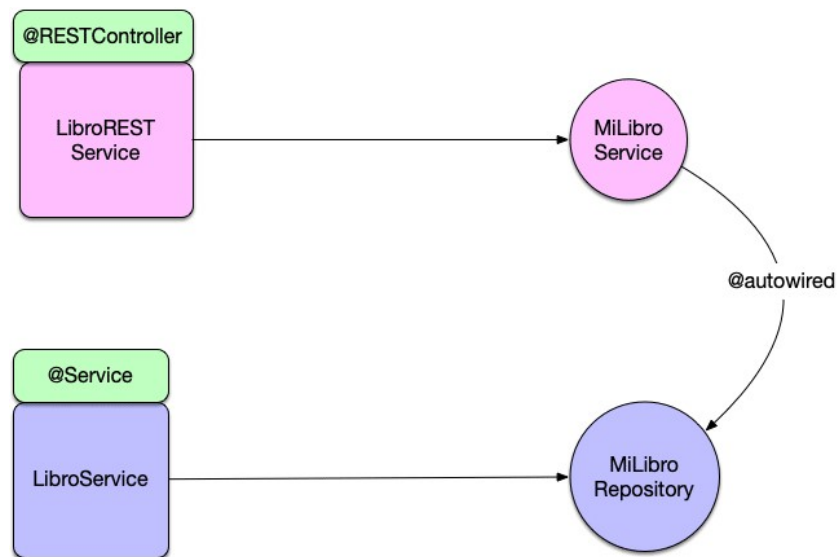
**@Service**, esta es una anotación que requiere la clase como tal para que sea reconocida por Spring como un servicio.

**@Autowired**, es una de las anotaciones más habituales cuando se trabaja con Spring Framework ya que se trata de una anotación que permite la inyección de dependencias.

Normalmente se acostumbra a usar **@Autowired** a nivel de la propiedad que se desea inyectar. Spring funciona como una [mega factoria de objetos](#). Cada clase se registra para instanciar objetos con alguna de las anotaciones **@Controller**, **@Service**, **@repository** o **@RestController**.



Una vez que los objetos están creados la anotación Spring **@Autowired** se encarga de construir las uniones entre los distintos elementos.



Lo anterior es habitual cuando se tiene un servicio y un repositorio como en nuestro ejercicio de ejemplo.

**@Override**, esta anotación en Java se usa para indicar la sobre escritura de un método ya sea heredado o implementado.

**@Transactional**, esta anotación indica que una interfaz, clase o método debe tener semántica transaccional, es decir, en el caso del método que lo implementa, hace que sus transacciones hacia la base de datos utilicen los términos de BEGIN TRANSACTION, COMMIT y ROLLBACK. Para ampliar el conocimiento y uso de estos términos en una base de datos relacional, sugiero consultar el siguiente vídeo: <https://youtu.be/keL9-EtE-zE>

**@Transactional(readOnly = true)**, esta anotación junto al atributo **readOnly** en verdadero hace las transacciones realizadas en ese método, sean exclusivamente de lectura, es decir, que no afecten la base de datos como tal, en otras palabras, se usa cuando estamos realizando una transacción del tipo SELECT.

El resultado final de este archivo lo podemos observar en la siguiente dirección:  
<https://github.com/training-practice-sofkau/Tutorial-Java-SpringBoot-MySQL/blob/main/src/main/java/com/sofka/contactos/service/LibretaService.java>

## El controlador

En el paquete *controller*, crearemos una clase de Java llamada **LibretaController** en el archivo **LibretaController.java** con la siguiente estructura.

```
package com.sofka.contactos.controller;

import com.sofka.contactos.domain.Contacto;
import com.sofka.contactos.domain.Telefono;
import com.sofka.contactos.service.LibretaService;
import com.sofka.contactos.utility.Response;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.dao.DataAccessException;
import org.springframework.data.domain.Sort;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.PatchMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestBody;

import javax.servlet.http.HttpServletResponse;

@Slf4j
@RestController
public class LibretaController {

    @Autowired
    private LibretaService libretaService;

    private Response response = new Response();

    private HttpStatus httpStatus = HttpStatus.OK;

    @GetMapping(path = "/")
    public ResponseEntity<Response> homeIndex1(HttpServletResponse httpResponse) { }

    @GetMapping(path = "/api/")
    public ResponseEntity<Response> homeIndex2(HttpServletResponse httpResponse) { }

    @GetMapping(path = "/api/v1/")
    public ResponseEntity<Response> homeIndex3(HttpServletResponse httpResponse) { }

    @GetMapping(path = "/api/v1/index")
    public ResponseEntity<Response> index() { }

    @GetMapping(path = "/api/v1/index/orderby/{orderBy}/{order}")
    public ResponseEntity<Response> indexOrderBy(
        @PathVariable(value="orderBy") String orderBy,
        @PathVariable(value="order") Sort.Direction order
    ) { }

    @GetMapping(path = "/api/v1/search/contact/{dataToSearch}")
    public ResponseEntity<Response> searchContactByNombreOrApellido(
        @PathVariable(value="dataToSearch") String dataToSearch
    ) { }

    @PostMapping(path = "/api/v1/contact")
    public ResponseEntity<Response> createContacto(@RequestBody Contacto contacto) { }

    @PostMapping(path = "/api/v1/phone")
    public ResponseEntity<Response> createTelefono(@RequestBody Telefono telefono) { }

    @PutMapping(path = "/api/v1/contact/{id}")
    public ResponseEntity<Response> updateContacto(
        @RequestBody Contacto contacto,
        @PathVariable(value="id") Integer id
    ) { }
```

```

@PutMapping(path = "/api/v1/phone/{id}")
public ResponseEntity<Response> updateTelefono(
    @RequestBody Telefono telefono,
    @PathVariable(value="id") Integer id
) { }

@PatchMapping(path = "/api/v1/contact/{id}/name")
public ResponseEntity<Response> updateNombreFromContacto(
    @RequestBody Contacto contacto,
    @PathVariable(value="id") Integer id
) { }

@PatchMapping(path = "/api/v1/contact/{id}/lastname")
public ResponseEntity<Response> updateApellidoFromContacto(
    @RequestBody Contacto contacto,
    @PathVariable(value="id") Integer id
) { }

@PatchMapping(path = "/api/v1/phone/{id}/number")
public ResponseEntity<Response> updateOnlyTelefono(
    @RequestBody Telefono telefono,
    @PathVariable(value="id") Integer id
) { }

@DeleteMapping(path = "/api/v1/contact/{id}")
public ResponseEntity<Response> deleteContacto(@PathVariable(value="id") Integer id) { }

@DeleteMapping(path = "/api/v1/phone/{id}")
public ResponseEntity<Response> deleteTelefono(@PathVariable(value="id") Integer id) { }

private ResponseEntity<Response> getResponseHome(HttpServletResponse httpResponse) { }

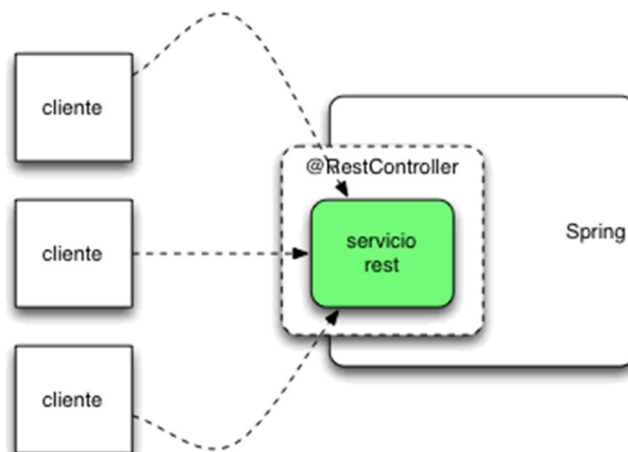
private void getErrorMessageInternal(Exception exception) { }

private void getErrorMessageForResponse(DataAccessException exception) { }
}

```

**@Slf4j** anotación usada (opcional) por parte de la librería de [Lombok](#), el cual usaremos para acceder a temas de [logging](#).

**@RestController** anotación usada y obligatoria para indicarle a Spring Framework que la clase que se ha definido actuará como un controlador de tipo RESTful y automáticamente se publicará como un Spring REST Service.

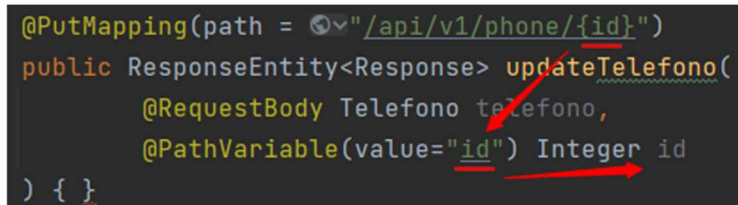




**@RequestBody**, es la anotación que le dice al sistema que todas las variables que llegan en el **Body** serán mapeadas al objeto que se indique, normalmente los datos que llegan por el Body son datos que se envían por el método **Post**, **Put** y **Patch**.

**@PathVariable**, es la anotación que indica por medio del atributo *value*, qué variable indicada en la URL será mapeada a la variable de Java.

```
@PutMapping(path = "/api/v1/phone/{id}")
public ResponseEntity<Response> updateTelefono(
    @RequestBody Telefono telefono,
    @PathVariable(value = "id") Integer id
) { }
```



**@GetMapping**, **@PostMapping**, **@PutMapping**, **@PatchMapping** y **@DeleteMapping** las anteriores etiquetas declaran que el método que los use será correlacionado con el método http que se espera por donde sea consultado. Cada anotación tiene varios atributos, pero su atributo más importante y usado es *path*, en el cual se define la ruta en la que el API responderá.

**Get**, método http usado para obtener información, como por ejemplo un listado. La forma coherente de transferir información al servidor por medio de este método es usando variables en la URL. No es normal hacer uso de transferir información en el Body, aunque no se descarta el hacerlo, pero tenga en cuenta que este movimiento no es normal.

**Post**, método http utilizado para crear información en el sistema, por ejemplo, crear un usuario, crear un registro de cualquier dato en el sistema.

**Put**, método http utilizado para cuando se quiere modificar una tupla en su totalidad, es decir, modificar todos sus campos de un registro basado normalmente en un ID. En este método es normal enviar el ID en la URL y los datos a modificar por medio del Body.

**Patch**, método http utilizado para cuando se requiere modificar una parte de la tupla, ya sea uno o varios campos sin que llegue a ser la totalidad, por ejemplo, se tiene una tupla conformada por nombre, apellidos y correo electrónico, en caso de modificar todos los campos, entonces lo ideal es usar el método Put, pero en caso de solo modificar uno o dos de esos campos,

entonces lo ideal es usar el método Patch. Este método al igual que el Put, lo ideal es enviar el ID por medio de la URL y la demás información por el Body.

**Delete**, método http utilizado para borrar información del servidor. Lo normal con este método es enviar información, específicamente el ID de lo que se desea borrar, por medio de la URL. Enviar información en el Body es factible al igual que en el método Get, pero no es normal realizar una acción de estas.

**ResponseEntity<Response>**, a pesar de que la clase **Response** no es una entidad, pero a través de **ResponseEntity** responderemos la API acompañados de un [código de estado http](#).

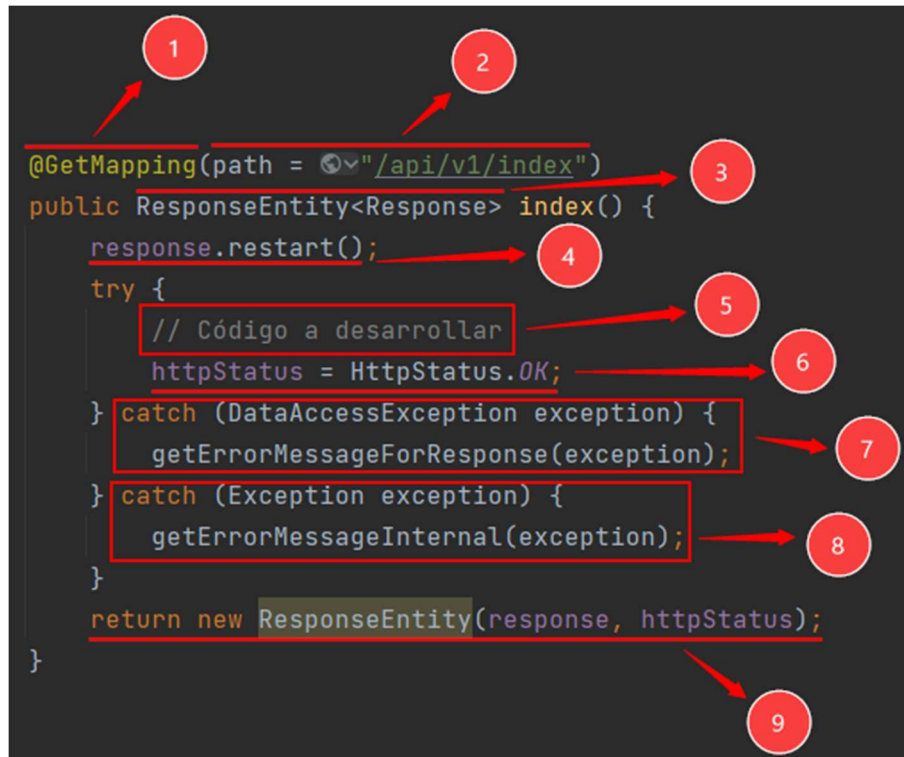


Como podrás notar las direcciones URL tienen una estructura detallada y eso tiene un por qué, el cual, consiste en que (1) la primera parte hacer alusión a que se trata de una API lo que ese está consumiendo, (2) la segunda parte hace referencia a la versión del API que se está consumiendo y (3) la última parte es la dirección como tal del API que se desea consumir en el sistema.

Estas direcciones no deberían estar repetidas dado que, si lo llegan a estar, el sistema dará error con tan solo iniciar.

**getErrorMessageInternal** y **getErrorMessageForResponse** para nuestro ejercicio serán dos métodos privados, el cual, manejarán los errores tanto del servidor interno como los que puedan ocurrir a nivel de bases de datos.

La estructura básica de cada método que es mapeado bajo un método http será la siguiente.



1. Anotación del método http que mapeará el método de la clase del controlador.
2. Dirección a la que responderá bajo el método configurado.
3. Tipo de respuesta que dará el método del controlador bajo el método http, en este caso se trata de la clase **ResponseEntity** junto al tipo de clase con el que realmente llevará la información, el cual será la clase **Response**.
4. Cuando iniciamos un método de un controlador, deberemos reiniciar la clase **Response** con respecto a sus atributos.
5. En este punto se desarrolla todo el contenido del método del controlador.
6. Al finalizar de forma correcta el método del controlador, deberíamos tener presente tres (3) cosas:
  - a. `response.message = "Mensaje para el frontend";`
  - b. `response.data = [Object];` // Listado, Arreglo u Objeto con información relevante para el frontend
  - c. `httpStatus = HttpStatus.OK;` // Estado (código http) con el que se responde la acción, normalmente es el código http 200
7. Dado que siempre deberemos controlar los errores que pueden existir en el sistema, este primer catch capturará las excepciones del tipo

**DataAccessException**, las cuales, son las que interceden cuando existen errores a nivel de ejecución de la base de datos.

8. Siguiendo la estela del numeral 7, este punto capturará las excepciones en general del sistema, es decir, errores internos en los cuales no deberían entorpecer la ejecución del frontend y es por esta razón que se deben controlar.
9. Por último, retornamos la instancia del objeto **ResponseEntity**, al cual le pasaremos tanto la instancia del objeto **Response**, como el código http de estado con el que se responderá la petición.

Puede observar el resultado final de este archivo en la siguiente dirección:

<https://github.com/training-practice-sofkau/Tutorial-Java-SpringBoot-MySQL/blob/main/src/main/java/com/sofka/contactos/controller/LibretaController.java>

Para consultar el código completo de este ejercicio, puede hacerlo en la siguiente dirección: <https://github.com/training-practice-sofkau/Tutorial-Java-SpringBoot-MySQL>

## Bibliografía

Álvarez Caules, C. (24 de Julio de 2015). *Arquitectura Java*. Obtenido de Spring REST Service con @RestController: <https://www.arquitecturajava.com/spring-rest-service-con-restcontroller/>

Álvarez Caules, C. (11 de Mayo de 2020). *Arquitectura Java*. Obtenido de Spring @Autowired y la inyección de dependencias: <https://www.arquitecturajava.com/spring-autowired-y-la-inyeccion-de-dependencias/>

Álvarez Caules, C. (19 de Junio de 2021). *Arquitectura Java*. Obtenido de ¿JPA vs Hibernate?: <https://www.arquitecturajava.com/jpa-vs-hibernate/>

de León, H. (21 de Agosto de 2020). *YouTube*. Obtenido de La magia de las transacciones SQL | Ejemplo en Sql Server: <https://youtu.be/keL9-EtE-zE>

EDTeams. (2020). *EDTeams*. Recuperado el 23 de Marzo de 2022, de Motores de bases de datos relacionales: <https://ed.team/comunidad/motores-de-bases-de-datos-relacionales>

*Marco de Desarrollo de la Junta de Andalucía*. (s.f.). Recuperado el 23 de Marzo de 2022, de Transacciones en la capa de negocio en Spring: <http://www.juntadeandalucia.es/servicios/madeja/contenido/recurso/170>

Romero, J. F. (02 de Abril de 2020). *Adictos al trabajo*. Obtenido de Hibernate – OneToOne, OneToMany, ManyToOne y ManyToMany: <https://www.adictosaltrabajo.com/2020/04/02/hibernate-onetoone-onetomany-manytoone-y-manytomany/>