



Anexo II Tema 3 – Hibernate, JPA y Maven

HIBERNATE, JPA Y MAVEN
STUDIUM

1. INTRODUCCIÓN

Hibernate es una herramienta que intenta hacer casar el modelo de datos que se usa durante la ejecución del programa (que está en memoria y cuando programamos en Java es orientado a objetos) con la base de datos (por lo general almacenada en el disco y con un modelo relacional).

En 2006 se lanzaron las especificaciones de la API (Interfaz de Programación) de persistencia de Java, conocida como JPA.

Se llama persistencia, a la capa de las aplicaciones que se encarga de gestionar los datos que se guardan, que persisten incluso cuando el programa no está funcionando, en las bases de datos.

En este tema crearemos las clases necesarias para realizar el Mapeo Objeto-Relacional (ORM) entre Java y la base de datos. Utilizaremos como ejemplo, un gestor de pedidos y facturas.

2. HERRAMIENTAS QUE NECESITAREMOS

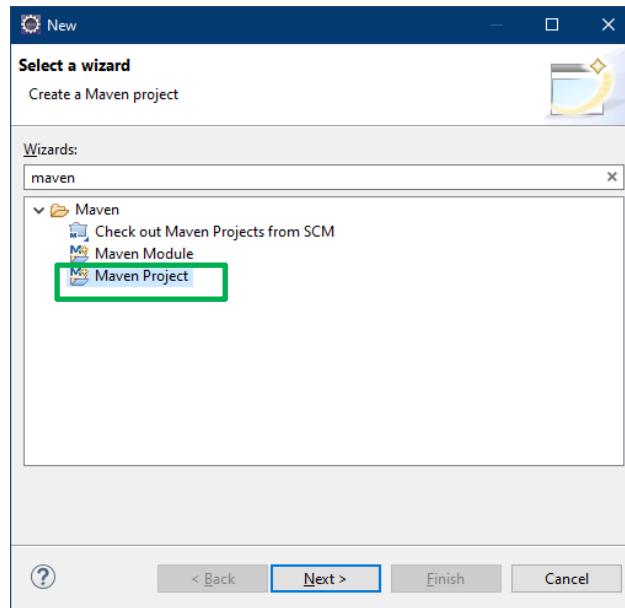
A continuación, detallaremos las aplicaciones que necesitaremos para llevar a cabo nuestro proyecto.

- Servidor de bases de datos MySQL.
- Eclipse IDE for Enterprise Java Developers (JEE).
 - La última versión disponible es Eclipse 2021-06 R.
 - Lo podéis descargar desde el siguiente enlace:
<https://www.eclipse.org/downloads/packages/>
- Hibernate versión 5.4 o la última versión estable.
- Maven para gestionar las dependencias, por ello no tendremos que descargarnos ni el connector de mysql ni las librerías de Hibernate.
 - Maven ya está integrado en Eclipse.
- MySQL Workbench lo utilizaremos como cliente para visualizar y manejar la base de datos.

3. CONFIGURACIÓN DEL PROYECTO

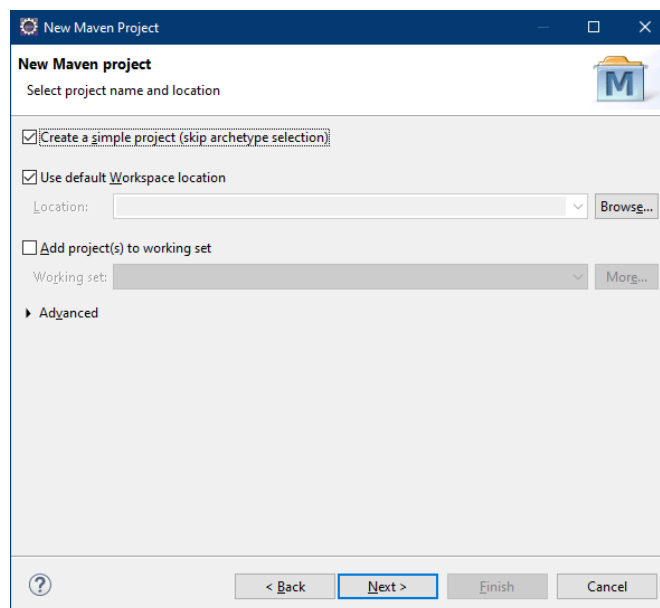
Crearemos en Eclipse un proyecto tipo **Maven** utilizando el **arquetipo quickstart**.

Desde Eclipse vamos a las opciones de menú **File - New – Other** en la ventana que nos aparece, en el **campo Wizard** ponemos *maven* y nos aparecerá el contenido de la carpeta **Maven**.



Seleccionamos la opción **Maven Project** para crear nuestro proyecto tipo Maven.

Next y nos aparece la siguiente pantalla donde marcamos la primera opción **“Create a simple project (skip archetype selection)”** y pulsamos **Next**.



En la ventana que nos aparece, indicamos los datos del proyecto Maven.

En Group id tenemos que indicar el nombre del paquete que vamos a utilizar en el proyecto.

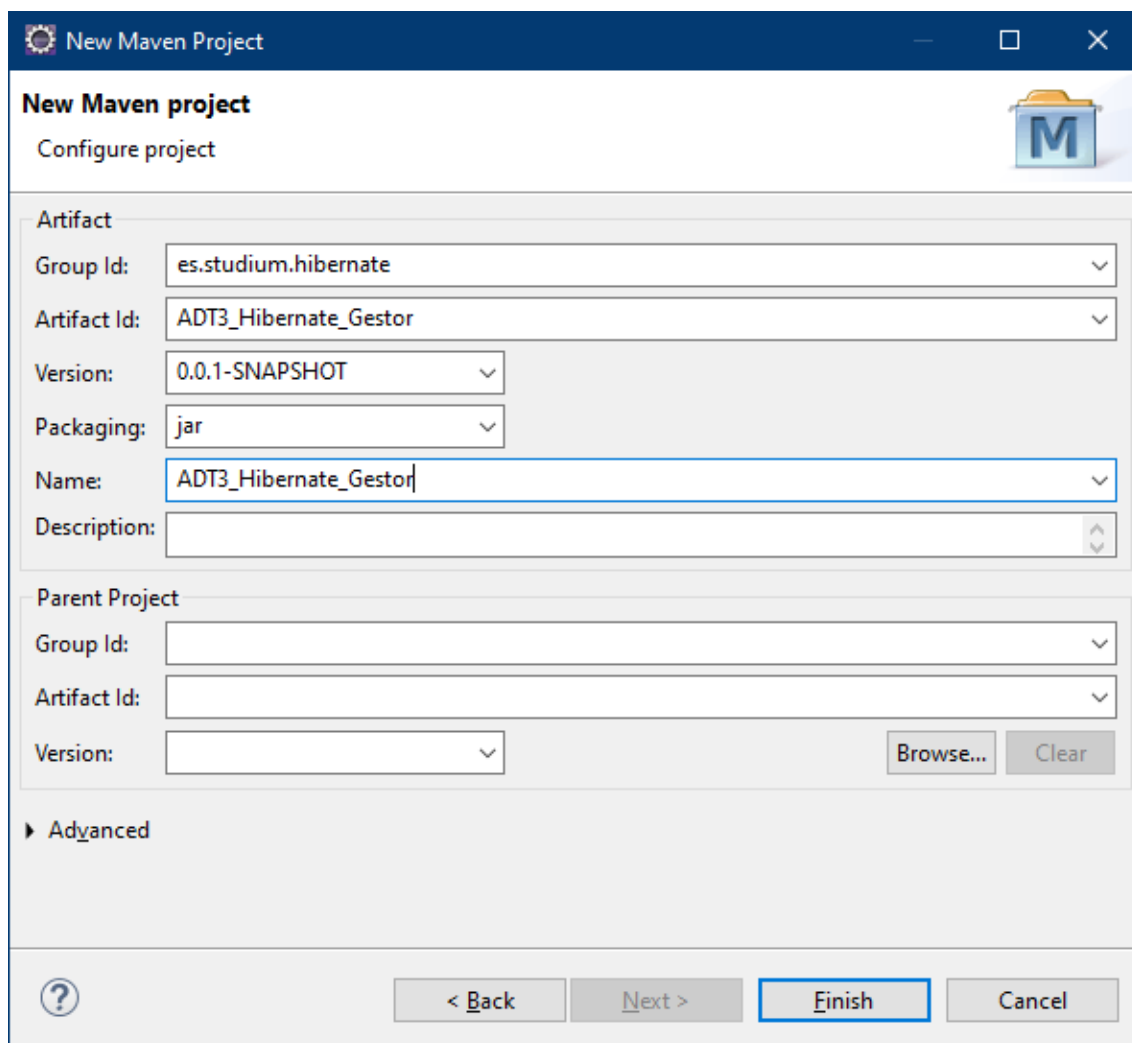
Group id: es.studium.hibernate

El nombre del proyecto lo tenemos que indicar en el campo Artifact id.

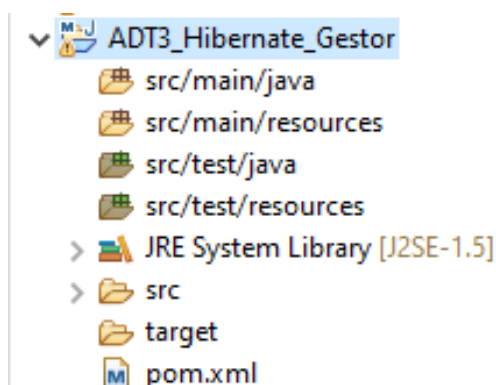
Artifact id: ADT3_Hibernate_Gestor

En el campo Name indicaremos el nombre del proyecto, es decir, lo mismo que hemos puesto en el campo Artifact id.

Name: ADT3_Hibernate_Gestor



Finish y observamos que nuestro proyecto Maven se ha creado correctamente en Eclipse y que tiene la siguiente estructura.



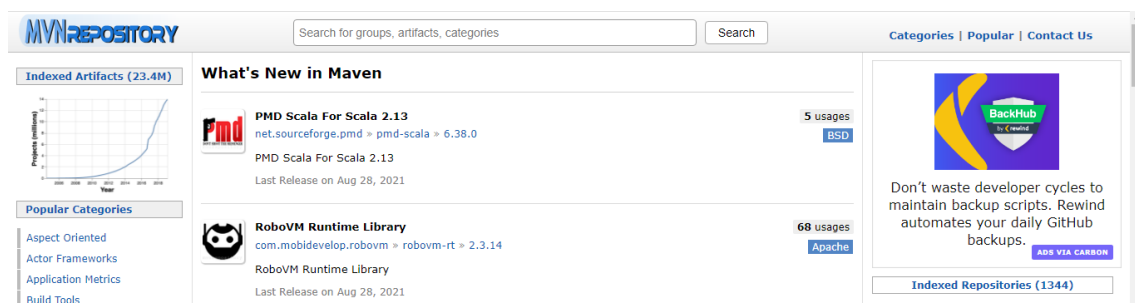
4. MAVEN

Maven es una herramienta para la gestión de proyectos y la utilizaremos como repositorio de librerías. Nos va a permitir **gestionar las dependencias con librerías que tenga nuestro proyecto**.

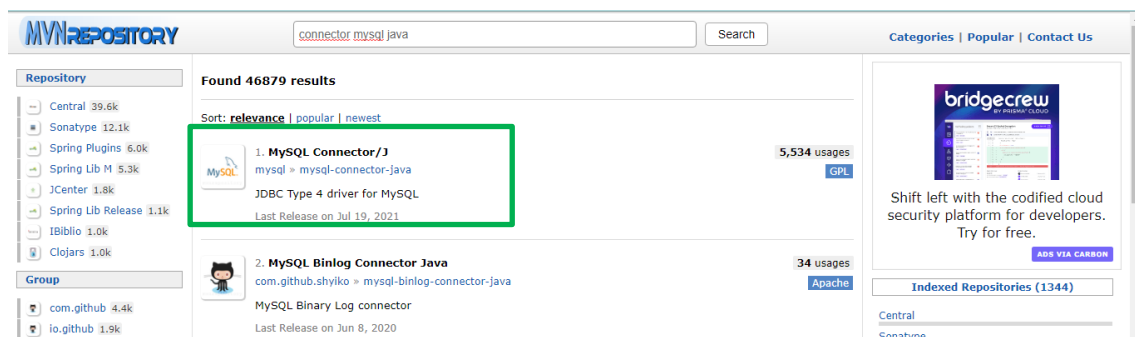
Podemos acceder a la página **web** del **repositorio** de **Maven** para encontrar las librerías que vamos a utilizar en nuestros proyectos y poder trabajar con Hibernate: <https://mvnrepository.com/>

Maven nos permite gestionar las dependencias a través de un fichero de configuración **pom.xml**.

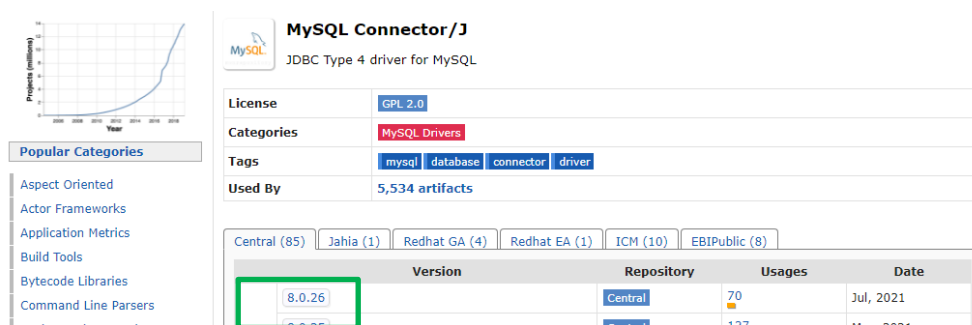
Accediendo al repositorio Maven, buscamos la dependencia que vamos a utilizar en nuestro proyecto: Hibernate y el connector de MySQL para Java.



- Indicamos, en el campo de búsqueda: **connector mysql java** y nos aparece la siguiente página.



Seleccionamos la primera opción: **MySQL Connector/J**



Seleccionamos la última versión disponible **8.0.26**.

Home » mysql » mysql-connector-java » 8.0.26

MySQL Connector/J >> 8.0.26

JDBC Type 4 driver for MySQL

License: GPL 2.0

Categories: MySQL Drivers

Organization: Oracle Corporation

HomePage: <http://dev.mysql.com/doc/connector-j/en/>

Date: (Jul 19, 2021)

Files: pom (2 KB) | jar (2.3 MB) | View All

Repositories: Central

Used By: 5,534 artifacts

Maven | Gradle | Gradle (Short) | Gradle (Kotlin) | SBT | Ivy | Grape | Leiningen | Buildr

```
<!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>8.0.26</version>
</dependency>
```

☒ Include comment with link to declaration

En la página que nos aparece, copiamos el código que nos aparece en la pestaña Maven y lo pegamos en el fichero pom.xml de nuestro proyecto Maven.

Home » mysql » mysql-connector-java » 8.0.26

MySQL Connector/J >> 8.0.26

JDBC Type 4 driver for MySQL

License: GPL 2.0

Categories: MySQL Drivers

Organization: Oracle Corporation

HomePage: <http://dev.mysql.com/doc/connector-j/en/>

Date: (Jul 19, 2021)

Files: pom (2 KB) | jar (2.3 MB) | View All

Repositories: Central

Used By: 5,534 artifacts

Maven | Gradle | Gradle (Short) | Gradle (Kotlin) | SBT | Ivy | Grape | Leiningen | Buildr

```
<!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>8.0.26</version>
</dependency>
```

☒ Include comment with link to declaration

Copied to clipboard!

```
<!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>8.0.26</version>
</dependency>
```

- Para conseguir las **dependencias de Hibernate**, tenemos que indicarlo en el campo de búsqueda del repositorio Maven.

En concreto utilizaremos el **módulo hibernate-agroal** luego, como criterio de búsqueda indicaremos **hibernate-agroal** y en la página que nos aparece, lo buscaremos desplazándonos hacia abajo.

MIT 93
TERRACOTTA 38
BSD 23
EPL 19
GPL 12

Last Release on Aug 4, 2010

7. Java Persistence API, Version 2.1
org.hibernate.javax.persistence » hibernate-jpa-2.1-api
969 usages
EPL | EDL
Clean-room definition of JPA APIs intended for use in developing Hibernate JPA implementation. See README.md for details
Last Release on Jan 23, 2018

8. Hibernate Agroal Relocation
org.hibernate » hibernate-agroal
1 usages
LGPL
Integration for Agroal as a ConnectionProvider for Hibernate ORM
Last Release on Aug 4, 2021

9. Hibernate HikariCP Relocation
org.hibernate » hibernate-hikaricp
431 usages
LGPL
Integration for HikariCP into Hibernate O/RM
Last Release on Aug 4, 2021

10. JPA 2.0 API
org.hibernate.javax.persistence » hibernate-jpa-2.0-api
983 usages
Hibernate definition of the Java Persistence 2.0 (JSR 317) API.
Last Release on Jun 15, 2011

Previous 1 2 3 4 5 6 7 8 9 10 Next

Pinchamos en la opción **Hibernate Agroal Relocation** y nos desplazamos hacia abajo para localizar la última versión FINAL disponible **5.5.7 Final**.

MVNREPOSITORY Search for groups, artifacts, categories Search

Home » org.hibernate » hibernate-agroal

Hibernate Agroal Relocation
Integration for Agroal as a ConnectionProvider for Hibernate ORM

License: LGPL 2.1
Tags: hibernate persistence
Used By: 1 artifacts

Note: This artifact was moved to:
org.hibernate.orm » hibernate-agroal » 6.0.0.Alpha9

Central (82)

	Version	Repository	Usages	Date
6.0.x	6.0.0.Alpha9	Central	0	Aug, 2021
	6.0.0.Alpha8	Central	0	May, 2021
	6.0.0.Alpha7	Central	0	Mar, 2021
	6.0.0.Alpha6	Central	0	Aug, 2020
	6.0.0.Alpha5	Central	0	Apr, 2020
	6.0.0.Alpha4	Central	0	Dec, 2019

Version	Repository	Downloads	Released
6.0.0.Alpha8	Central	0	May, 2021
6.0.0.Alpha7	Central	0	Mar, 2021
6.0.0.Alpha6	Central	0	Aug, 2020
6.0.0.Alpha5	Central	0	Apr, 2020
6.0.0.Alpha4	Central	0	Dec, 2019
6.0.0.Alpha3	Central	0	Nov, 2019
6.0.0.Alpha2	Central	0	Apr, 2019
5.6.0.Beta1	Central	0	Aug, 2021
5.5.7.Final	Central	0	Aug, 2021
5.5.6.Final	Central	0	Aug, 2021
5.5.5.Final	Central	0	Jul, 2021
5.5.4.Final	Central	0	Jul, 2021
5.5.3.Final	Central	0	Jun, 2021
5.5.2.Final	Central	0	Jun, 2021
5.5.0.Final	Central	0	Jun, 2021
5.5.0.CR1	Central	0	May, 2021
5.5.0.Beta1	Central	0	May, 2021

Hibernate Agroal Relocation » 5.5.7.Final
Integration for Agroal as a ConnectionProvider for Hibernate ORM

License LGPL 2.1
Organization Hibernate.org
HomePage <https://hibernate.org/orm>
Date (Aug 25, 2021)
Files [pom \(2 KB\)](#) [jar \(6 KB\)](#) [View All](#)
Repositories Central
Used By 1 artifacts

Note: There is a new version for this artifact

New Version 6.0.0.Alpha9

Maven [Gradle](#) [Gradle \(Short\)](#) [Gradle \(Kotlin\)](#) [SBT](#) [Ivy](#) [Grape](#) [Leiningen](#) [Buildr](#)

```
<!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-agroal -->
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-agroal</artifactId>
  <version>5.5.7.Final</version>
</dependency>
```

Copiamos el código que nos aparece en la pestaña Maven y lo pegamos en el fichero **pom.xml** de nuestro proyecto Maven.

```
<!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-agroal -->
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-agroal</artifactId>
  <version>5.5.7.Final</version>
</dependency>
```

5. DEPENDENCIAS

Incluimos las dependencias en nuestro proyecto Maven: **ADT3_Hibernate_Gestor**.

Para ello, en el fichero **pom.xml** tenemos que incluir el siguiente código:

- **Connector Java para MySQL**


```
<!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>8.0.26</version>
</dependency>
```

- **Hibernate**

```
<!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-agroal -->
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-agroal</artifactId>
  <version>5.5.7.Final</version>
</dependency>
```

IMPORTANTE: cuando añadamos esta dependencia al pom.xml debemos agregar la siguiente etiqueta: `<type>pom</type>` después de la etiqueta `<version>` quedando de la forma siguiente.

```
<!--
https://mvnrepository.com/artifact/org.hibernate/hibernate-
agroal -->
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-agroal</artifactId>
    <version>5.5.7.Final</version>
    <type>pom</type>
  </dependency>
```

De esta forma especificamos que el tipo de la dependencia debe ser **pom** y no el valor que tomaría por defecto que sería `<type>jar</type>`.

Al crear nuestro proyecto Maven, el fichero **pom.xml** aparece de la siguiente forma:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>es.studium.hibernate</groupId>
  <artifactId>ADT3_Hibernate_Gestor</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>ADT3_Hibernate_Gestor</name>
</project>
```

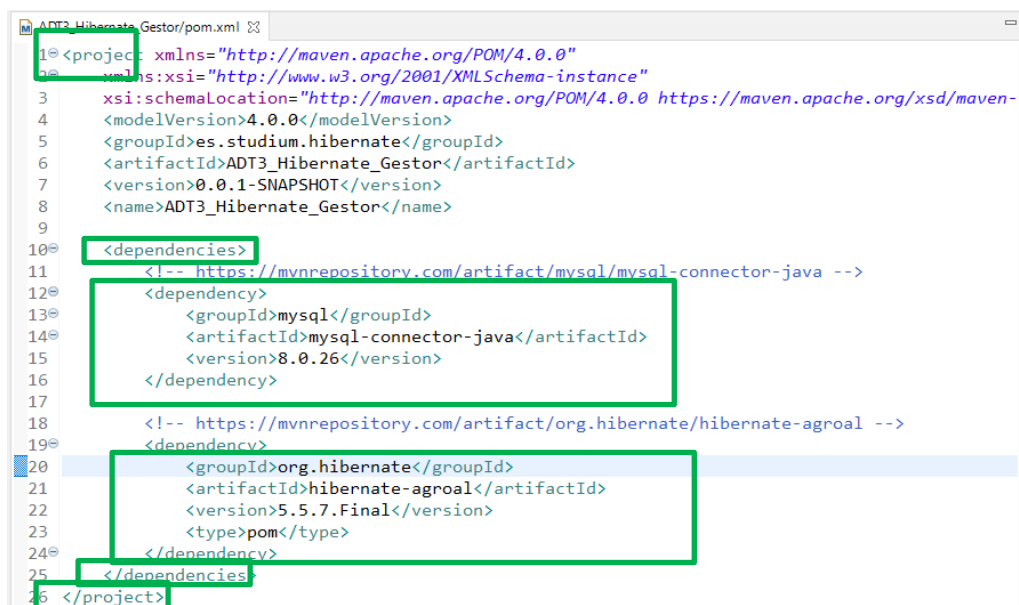
Y nosotros, tenemos que añadir en él, las dependencias que vamos a utilizar en nuestro proyecto: Connector de MySQL e Hibernate.

Para ello tenemos que incluir las etiquetas `<dependencies>` `</dependencies>` dentro de las etiquetas `<project>``</project>` entre las que incluiremos nuestras dependencias de la forma siguiente:

```
<dependencies>
  <!-- https://mvnrepository.com/artifact/mysql/mysql-
connector-java -->
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.26</version>
  </dependency>

  <!--
https://mvnrepository.com/artifact/org.hibernate/hibernate-
agroal -->
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-agroal</artifactId>
    <version>5.5.7.Final</version>
    <type>pom</type>
  </dependency>
</dependencies>
```

Nuestro fichero **pom.xml** quedará configurado de la forma siguiente:

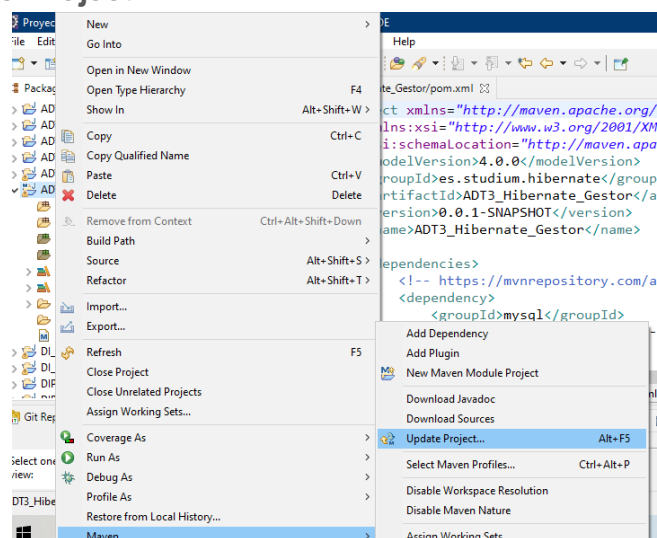


```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

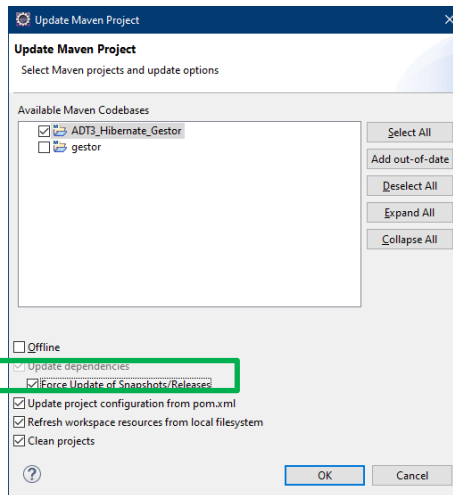
```
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>es.studium.hibernate</groupId>
  <artifactId>ADT3_Hibernate_Gestor</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>ADT3_Hibernate_Gestor</name>

  <dependencies>
    <!--
https://mvnrepository.com/artifact/mysql/mysql-connector-java
-->
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>8.0.26</version>
    </dependency>
  <!--
https://mvnrepository.com/artifact/org.hibernate/hibernate-
agroal -->
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-agroal</artifactId>
      <version>5.5.7.Final</version>
      <type>pom</type>
    </dependency>
  </dependencies>
</project>
```

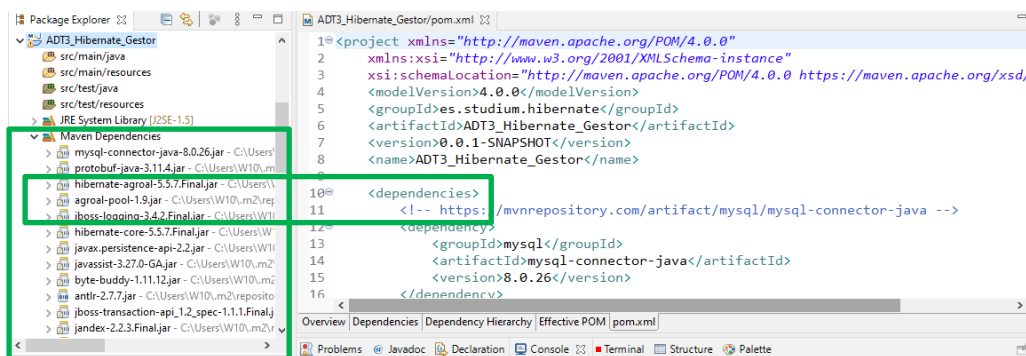
Una vez hemos incluido las dependencias en el fichero pom.xml, **actualizamos el proyecto** para hacer efectivos los cambios en el fichero **pom.xml**. Para ello **sobre el nombre del proyecto** con el botón derecho del ratón pinchamos en **Maven – Update Project**.



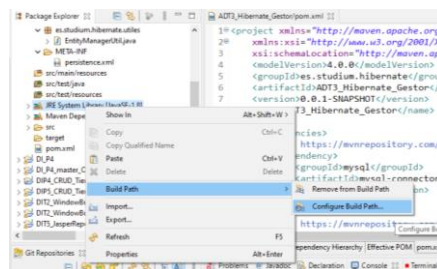
Nos aparece la siguiente pantalla en la que, vemos que nuestro proyecto ya está seleccionado, además seleccionamos la opción “**Force Update of Snapshots/Releases**” y ok.



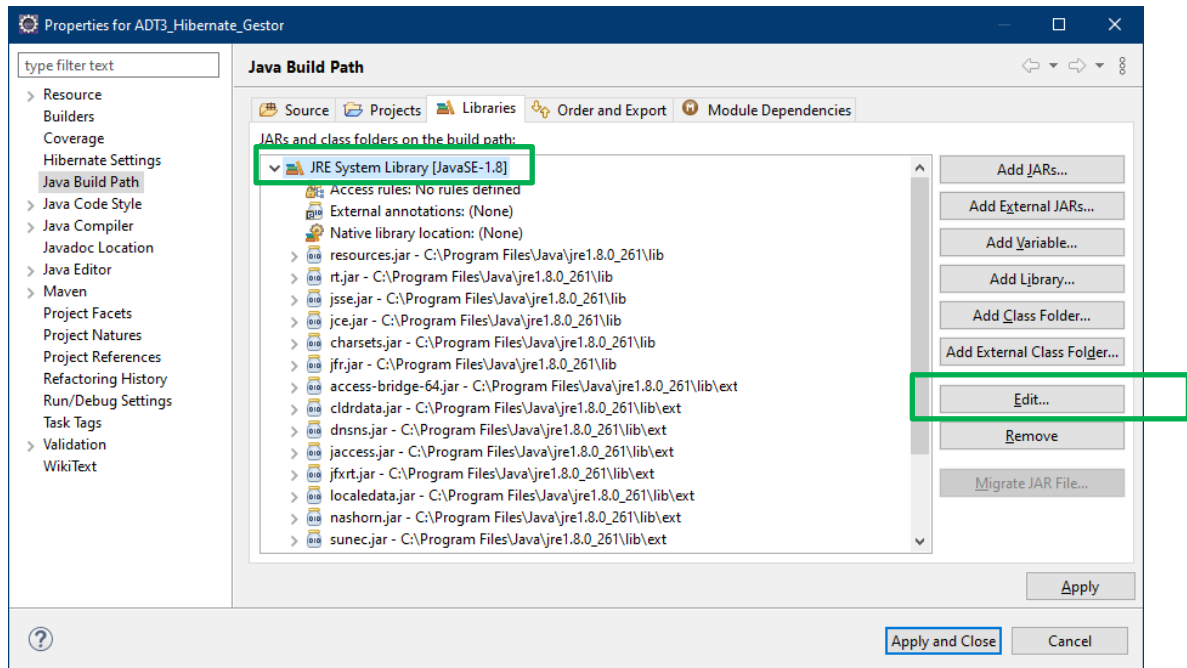
Observamos que nuestro proyecto debe tener la siguiente estructura:



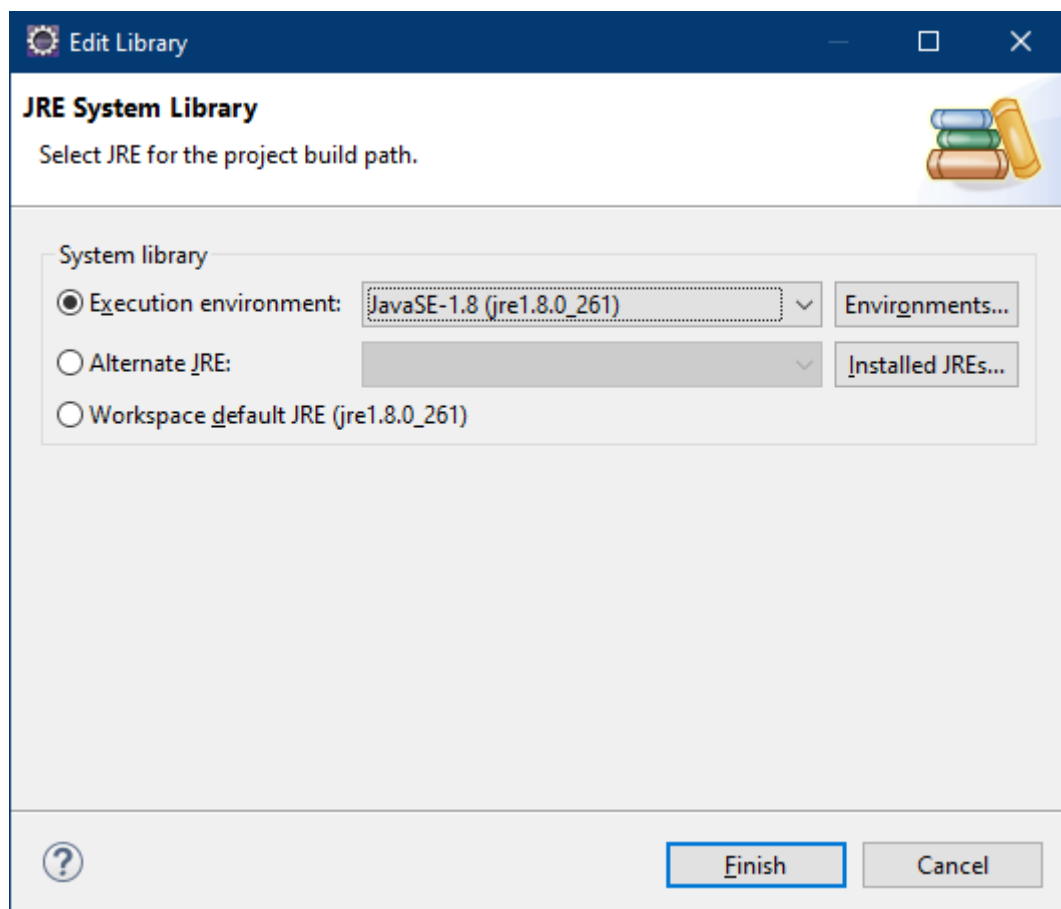
Nota: debemos tener en cuenta, que cuando hacemos esta actualización en nuestro proyecto, la versión de java se modifica a la 1.5 y debemos escoger la versión con la que deseamos trabajar en nuestro proyecto. Para ello, desde “JRE System Library” en nuestro proyecto, con el botón derecho del ratón seleccionamos las opciones de menú **Build Path – Configura Build Path**.



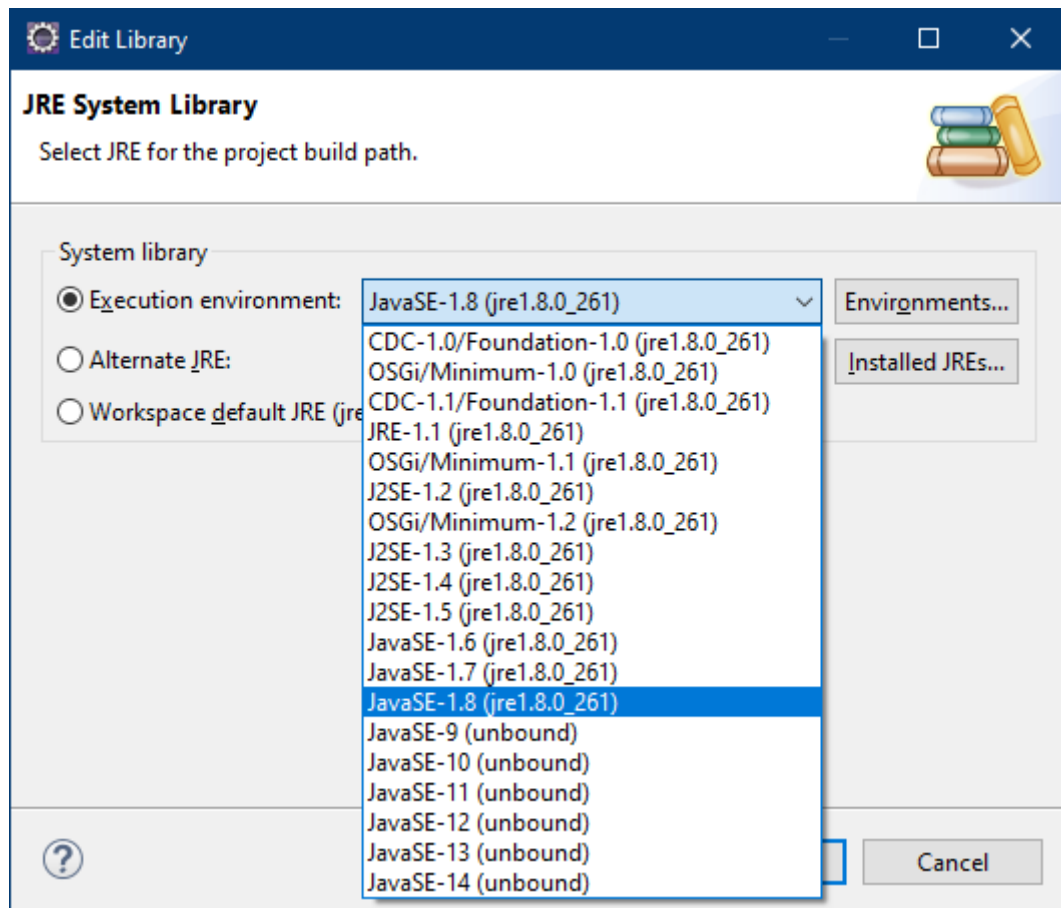
En la pantalla que nos aparece, seleccionamos la versión de Java con la que queremos trabajar en nuestro proyecto.



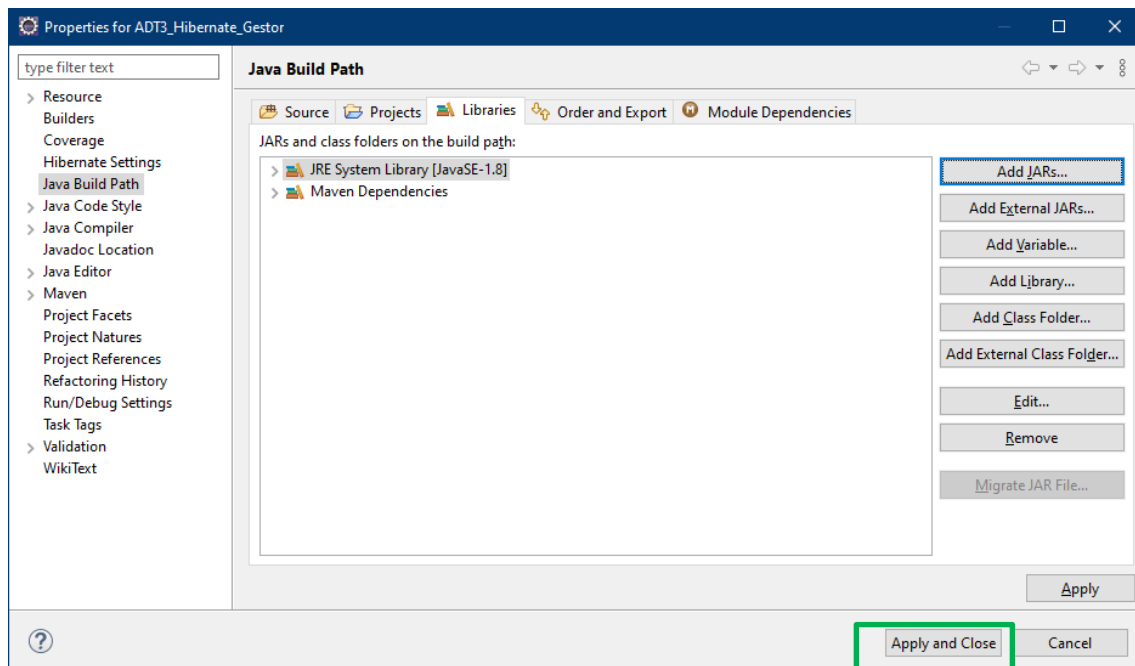
Nos posicionamos sobre **“JRE System Library”** y pinchamos en **Edit**. Nos aparece la siguiente pantalla.



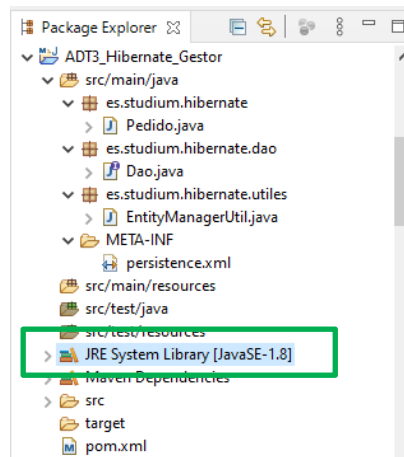
En el desplegable, podemos seleccionar la versión de Java que queramos.



Una vez la hemos seleccionado, pinchamos en Finish y después en Apply and Close.



Y podemos comprobar que los cambios se han hecho efectivos en nuestro proyecto.



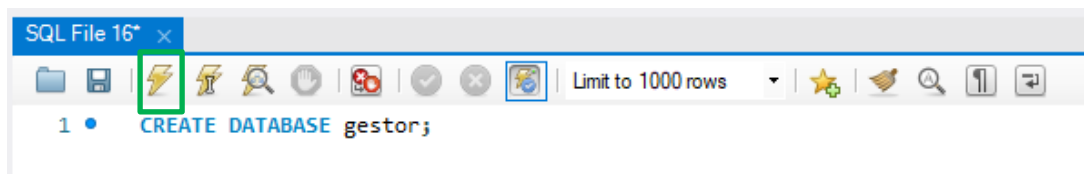
6. CREACIÓN DE LA BASE DE DATOS

Para Conectar nuestro código Java con una base de datos, primero tenemos que crear el esquema de la base de datos.

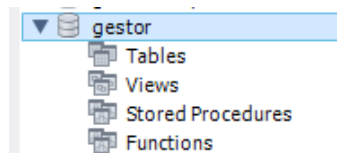
En nuestro ejemplo dejaremos que Hibernate cree de forma automática las tablas, por eso nosotros solamente vamos a crear el esquema de la base de datos al que vamos a llamar **gestor**.

Utilizando MySQL Workbench crearemos el esquema de la base de datos:

CREATE DATABASE gestor;



Pinchamos en ejecutar la sentencia SQL y comprobamos que se ha creado nuestro esquema de base de datos con la siguiente estructura:



Si queremos usar nuestra base de datos, debemos ejecutar la siguiente consulta:

use gestor;



7. CONFIGURACIÓN DE HIBERNATE

Ahora configuraremos la persistencia en nuestro proyecto, es decir, indicaremos a Hibernate la base de datos a la que tiene que conectarse, con qué datos, etc... Hay distintas formas de hacerlo.

Como **Hibernate** (que es una implementación en concreto) fue **anterior** a **JPA** (que es un estándar a seguir), hay proyectos que utilizan la forma de hacerlo con **Hibernate**: que es utilizando el **fichero de configuración** llamado **hibernate.cfg.xml** y que tenéis desarrollado en la documentación disponible en la plataforma:

<https://campustudium.com/mod/book/view.php?id=8332>

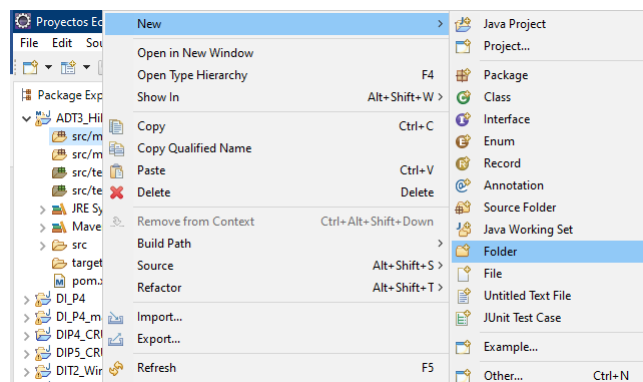
Pero **JPA** propone otra forma que es la que desarrollaremos en esta documentación.

El modo de hacerlo según **JPA** es con un **fichero de configuración** en formato XML llamado **persistence.xml** y que tiene que estar dentro de la **carpeta META-INF** de nuestro proyecto.

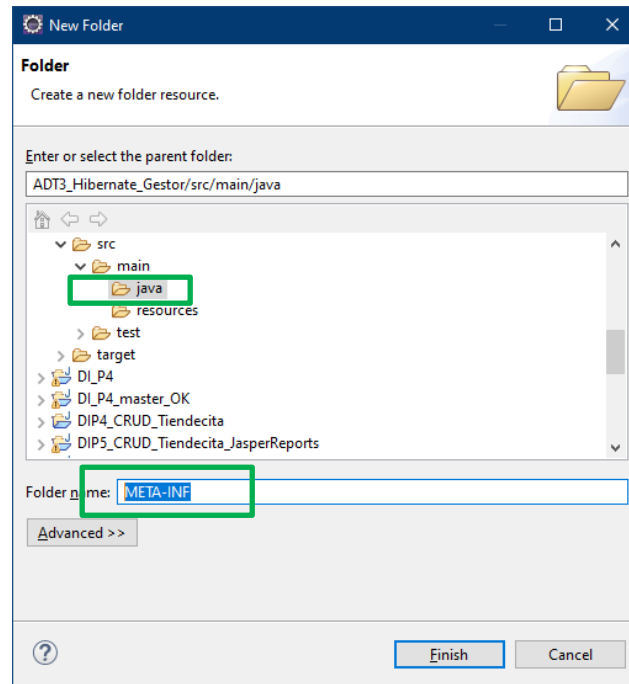
Creemos, paso a paso, el fichero de configuración **persistence.xml**.

- Creamos una carpeta **META-INF** dentro de la carpeta **src/main/java**.

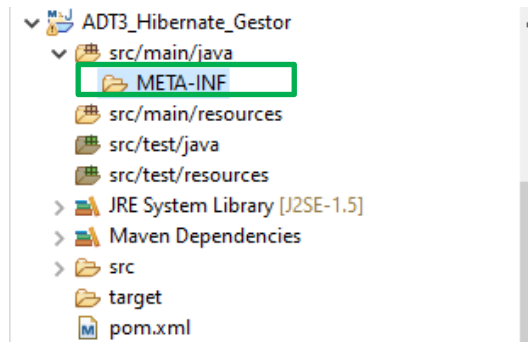
Sobre la carpeta **src/main/java** con el botón derecho del ratón seleccionamos las opciones de menú **New - Folder**



En la pantalla que aparece, nos aseguramos de que está seleccionada la carpeta **src/main/java** y le damos el nombre **META-INF** a la carpeta que vamos a crear.

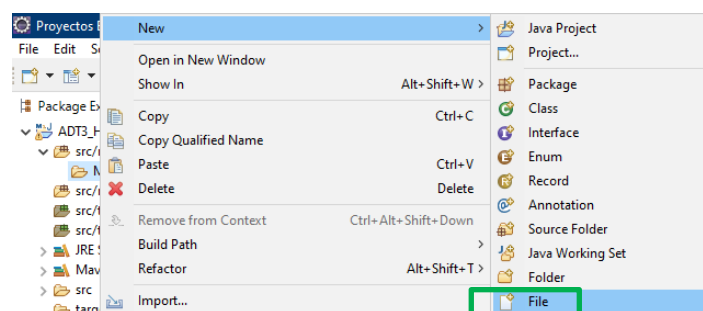


Finish y observamos que se ha creado nuestra **carpeta META-INF** dentro de **src/main/java**. Es muy importante mantener tanto el nombre de la carpeta como su ubicación.

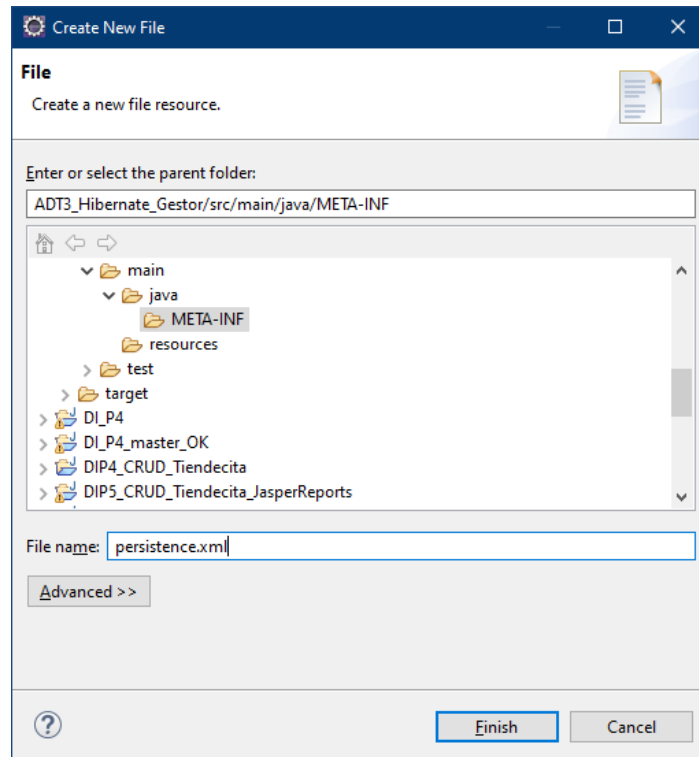


- Dentro de esta carpeta, creamos un fichero de texto que llamaremos **persistence.xml**.

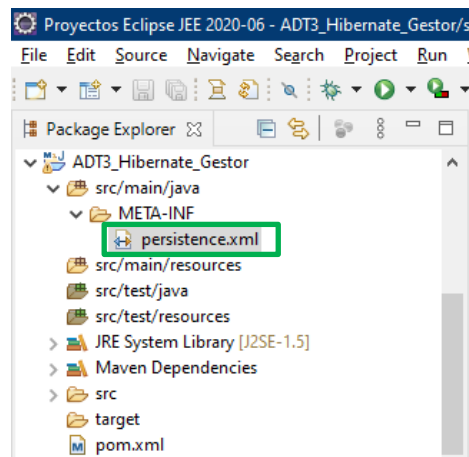
Sobre la carpeta **META-INF** con el botón derecho del ratón seleccionamos las opciones de menú **New – File**.



En la pantalla que aparece, nos aseguramos de que tenemos seleccionada la carpeta **META-INF** y le damos el nombre a nuestro fichero **persistence.xml**.



Finish y observamos que se ha creado nuestro fichero.



- En la primera línea del fichero xml debemos definir versión y encoding, como es todos los ficheros xml.

```
<?xml version="1.0" encoding="UTF-8"?>
```

- La etiqueta raíz en este fichero debe ser `<persistence>` y en ella indicamos la ruta del esquema y el espacio de nombres que vamos a usar.

```
<persistence version="2.1"
xmlns="http://xmlns.jcp.org/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">

</persistence>
```

- Dentro de la etiqueta `persistence` metemos otra etiqueta llamada `persistence-unit` en la que empezaremos ya a personalizar.

```
<persistence-unit name="gestor" transaction-
type="RESOURCE_LOCAL">

</persistence-unit>
```

Le pondremos el nombre que queramos `name="gestor"` pero tenemos que recordarlo, porque luego lo referenciaremos en el código Java. Llamaremos `gestor` a esta unidad de persistencia `persistence-unit` para recordarlo más fácilmente.

- Dentro de `persistence-unit` indicamos el proveedor que usaremos. En nuestro caso el de Hibernate.

```
<provider>org.hibernate.jpa.HibernatePersistenceProvider</pro
vider>
```

- Luego indicaremos las propiedades: una etiqueta que las engloba, `<properties>` en plural, y una etiqueta por cada una de las propiedades, `<property>` en singular, dentro de la anterior.

```
<properties>
  <property name="javax.persistence.jdbc.url"
value="jdbc:mysql://localhost:3306/gestor"/>
  <property name="javax.persistence.jdbc.driver"
value="com.mysql.jdbc.Driver"/>
  <property name="javax.persistence.jdbc.user"
value="root"/>
  <property name="javax.persistence.jdbc.password"
value="RootRoot"/>
  <property name="javax.persistence.schema-
generation.database.action" value="update"/>
  <property name = "hibernate.show_sql" value = "true" />
</properties>
```

- Entre las propiedades tenemos que especificar la URL de nuestra base de datos que será la siguiente en nuestro caso, ya que estamos utilizando MySQL como motor de base de datos.

```
<property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/gestor"/>
```

Donde tenemos que la URL llevará el **protocolo** `jdbc` el **subprotocolo** `mysql` (el protocolo y subprotocolo reemplazan lo que en una URL de una web sería http o https), `://servidor` `://localhost`, puerto `:3306` y el nombre de nuestra base de datos `gestor`.

- Ahora indicaremos el **driver** que será el de mysql.

```
<property name="javax.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver"/>
```

- A continuación indicamos el nombre de usuario con el que vamos a acceder a nuestra base de datos, usaremos root ya que no hemos creado otro usuario diferente.

```
<property name="javax.persistence.jdbc.user" value="root"/>
```

- Ahora indicamos la contraseña del usuario con el que accederemos a la base de datos.

```
<property name="javax.persistence.jdbc.password" value="Studium2019;"/>
```

- Debemos especificar cómo queremos que Hibernate gestione la base de datos. Lo pondremos en `value="update"` que significa que si hacemos cambios en el código de nuestro mapeo, añadir o quitar columnas por ejemplo, el propio Hibernate se encargará de modificar la base de datos.

```
<property name="javax.persistence.schema-generation.database.action" value="update"/>
```

Con estos datos, ya tenemos configurado Hibernate para poder trabajar.

Nuestro fichero **persistence.xml** completo, queda de la forma siguiente.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1"
xmlns="http://xmlns.jcp.org/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
  <persistence-unit name="gestor" transaction-
type="RESOURCE_LOCAL">
```

```
<provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>

    <properties>
        <property name="javax.persistence.jdbc.url"
value="jdbc:mysql://localhost:3306/gestor"/>
        <property name="javax.persistence.jdbc.driver"
value="com.mysql.cj.jdbc.Driver"/>
        <property name="javax.persistence.jdbc.user"
value="root"/>
        <property
name="javax.persistence.jdbc.password" value="Stodium2019;"/>
        <property name="javax.persistence.schema-
generation.database.action" value="update"/>
        <property name = "hibernate.show_sql" value =
"true" />
    </properties>
</persistence-unit>
</persistence>
```

8. ENTIDADES

El objetivo del mapeo entre Java y la base de datos es lograr representar en programación orientada a objetos la estructura de datos que tenemos en la base de datos.

Para ello, recurrimos a una clases llamadas **entidades**.

En la mayoría de los casos, habrá una correspondencia casi directa entre clases y tablas, es decir, por cada tabla habrá una clase.

Para hacer el mapeo, hay dos opciones:

- Por el fichero de configuración XML, que es la primera forma que hubo de hacerlo.
- Por anotaciones, que es una forma más cómoda de hacer el mapeo, porque las anotaciones van metidas dentro del código Java, justo por encima de cada elemento.

NOTA: Las anotaciones en Java se utilizan para muchas cosas, no solo para el mapeo objeto relacional. Son esas arrobas (@) que se ponen antes de ciertos elementos. Las anotaciones están implementadas en clases y para poderlas utilizar, necesitamos importarlas en nuestro programa.

- Para **marcar como entidad una clase Java**, es decir, para **vincularla con una tabla de la base de datos**, utilizaremos la **anotación @Entity**. De esta

forma, le estamos diciendo al sistema de persistencia, que esta clase es una entidad que tiene que mapear.

Observaremos en nuestro programa, que al incluir esta anotación en la clase, se importa de forma automática el paquete `javax.persistence.Entity`.

Debemos tener cuidado a la hora de realizar los import, debemos fijarnos en el paquete que vamos a importar, ya que como las cosas se pueden hacer por Hibernate o por JPA, hay muchas anotaciones con nombres coincidentes que están disponibles en ambas paquetaciones.

Las anotaciones de **Hibernate** están en el paquete `org.hibernate.annotations`; las de **JPA** en el paquete `javax.persistence`.

Así que cada vez que tengamos que elegir el paquete que queremos importar, debemos recordar que estamos haciéndolo por JPA. Debemos mantener la coherencia o no nos funcionará el programa.

- Ahora tenemos que decirle con qué tabla queremos mapear esta clase, esta entidad. Para ello debajo del `@Entity` meteremos la **anotación de tabla**: `@Table(name = "nombre_tabla")`
- Si una clase se mapea a una tabla, un atributo se mapea a una columna, a cada atributo le pondremos la anotación `@Column(name = "nombre_columna")` indicando el nombre de la columna en la tabla. Por cada atributo tenemos los métodos *getter* y *setter*, para que Hibernate acceda a ellos.

Cuando las columnas se llaman igual que los atributos nos ahorramos las anotaciones `@Column(name = "nombre_columna")`.

- Para marcar la clave primaria, utilizaremos la anotación `@Id` sin nada más.
- Si queremos que alguna columna sea autogenerada, lo indicaremos usando la anotación `@GeneratedValue(strategy = GenerationType.IDENTITY)`

8.1 Entidad Pedido

Realizaremos nuestro primer mapeo empezando por la **clase Pedido** que tendrá solamente tres atributos: id, referencia y fecha, sin relacionarla aún con otras tablas.

Creamos el paquete `es.studium.hibernate` dentro de la ruta `src/main/java` y dentro de él crearemos nuestra **clase Pedido**.

Clase Pedido

- Atributos privados:
 - id (tipo int)
 - referencia (tipo String)
 - fecha (tipo LocalDateTime)
- Métodos constructores
 - Constructor por defecto
 - Constructor con dos parámetros referencia y fecha
- Métodos inspectores:
 - Getters
 - Setters
- Método toString

```
package es.studium.hibernate;

import java.time.LocalDateTime;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

public class Pedido {

    private int id;
    private String referencia;
    private LocalDateTime fecha;

    public Pedido() {

    }

    public Pedido(String referencia, LocalDateTime fecha) {
        this.referencia = referencia;
        this.fecha = fecha;
    }

    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getReferencia() {
        return referencia;
    }
    public void setReferencia(String referencia) {
        this.referencia = referencia;
    }
}
```

```
}  
public LocalDateTime getFecha() {  
    return fecha;  
}  
public void setFecha(LocalDateTime fecha) {  
    this.fecha = fecha;  
}  
@Override  
public String toString() {  
    return "Pedido [id=" + id + ", referencia=" +  
referencia + ", fecha=" + fecha + "];"  
}  
}
```

Incluimos las anotaciones indicadas anteriormente en nuestra entidad Pedido y nos quedaría así.

```
package es.studium.hibernate;  
  
import java.time.LocalDateTime;  
import javax.persistence.Column;  
import javax.persistence.Entity;  
import javax.persistence.GeneratedValue;  
import javax.persistence.GenerationType;  
import javax.persistence.Id;  
import javax.persistence.Table;  
  
@Entity  
@Table(name = "pedido")  
public class Pedido {  
  
    @Column(name = "id")  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private int id;  
  
    @Column(name = "referencia")  
    private String referencia;  
  
    @Column(name = "fecha")  
    private LocalDateTime fecha;  
  
    public Pedido() {  
  
    }  
  
    public Pedido(String referencia, LocalDateTime fecha) {  
        this.referencia = referencia;  
    }  
}
```



```
        this.fecha = fecha;
    }

    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getReferencia() {
        return referencia;
    }
    public void setReferencia(String referencia) {
        this.referencia = referencia;
    }
    public LocalDateTime getFecha() {
        return fecha;
    }
    public void setFecha(LocalDateTime fecha) {
        this.fecha = fecha;
    }
    @Override
    public String toString() {
        return "Pedido [id=" + id + ", referencia=" +
referencia + ", fecha=" + fecha + "]";
    }
}
```

9. OBJETOS DE ACCESO A DATOS (DAO)

Hoy en día, el patrón de diseño DAO (objetos de acceso a datos) es muy utilizado para acceder a los datos.

Este patrón nos invita a tener una clase con los métodos para acceder a los datos.

Si por un lado tenemos las entidades que se emparejan con las tablas de la base de datos, por el otro están los métodos del DAO que emparejan con las operaciones CRUD (creación, petición, actualización y borrado de datos) de SQL.

En las clases DAO, que habrá una por entidad o tabla, tendremos métodos para realizar consultas, creaciones, actualizaciones, borrados... Pero estos métodos serán muy parecidos para unas y otras tablas, como por ejemplo encontrar una persona por id, encontrar una factura por id, eliminar un expediente por id, etc. Por ello en los proyectos se suele crear una parte común, es decir, una Interfaz DAO que defina todas las operaciones comunes para cualquier entidad

(recuperar todo, recuperar por id, guardar, actualizar, borrar), para luego crear una Clase Abstracta que implemente la parte común para todas las entidades y finalmente crear una Clase DAO por cada Entidad en la que añadiremos métodos particulares a dicha entidad.

9.1 Interfaz Dao<T>

Creemos un **subpaquete** `es.studium.hibernate.dao` dentro del que crearemos la **Interfaz** llamada **Dao**. Como vamos a tratar distintos tipos de datos, uno por cada Entidad que tengamos, utilizaremos en esta interfaz un **tipo genérico T**.

Esta interfaz tendrá los métodos siguientes:

- **get**: recupera un registro concreto, según su id. Como puede que no encuentre ninguno, retornamos un objeto *Optional*. Recibirá el id y devolverá, o no, un objeto del tipo genérico de la clase.

Nota: Una variable cuyo tipo es *Optional* nunca debe ser nula; siempre debe apuntar a una instancia *Optional*.

```
Optional<T> get(long id);
```

- **getAll**: Por lo general, también necesitamos recuperar todos los registros: no recibe nada y devuelve una lista de objetos del tipo genérico.

```
List<T> getAll();
```

- **save**: No devuelve nada y recibe como parámetro uno de los objetos de tipo genérico T. **void** save(T t);

- **update**: no devolverá nada y recibe el objeto a actualizar.

```
void update(T t);
```

- **delete**: no devuelve nada y recibe el objeto a eliminar. **void** delete(T t);

Interfaz Dao

```
package es.studium.hibernate.dao;

import java.util.List;
import java.util.Optional;

public interface Dao<T> {

    Optional<T> get(long id);
    List<T> getAll();
    void save(T t);
```

```
void update(T t);  
void delete(T t);  
  
}
```

9.2 Gestor de entidades

Partiendo de la Interfaz Dao podemos implementar una clase abstracta para acceder a la base de datos a través de Hibernate.

Antes de implementar nuestra clase abstracta, implementaremos una clase que nos ayudará a establecer y manejar la conexión con la base de datos. Es una clase muy sencilla, pero muy importante para acceder a la base de datos.

- La llamaremos `EntityManagerUtil` y la crearemos dentro del subpaquete `es.studium.hibernate.utiles`.

```
package es.studium.hibernate.utiles;
```

```
public class EntityManagerUtil {...}
```

- En ella, crearemos un único método estático llamado `getEntityManager()` que no recibe nada pero devuelve un objeto de tipo `EntityManager` que pertenece a la librería de persistencia de Java EE.

```
public static EntityManager getEntityManager() {...}
```

- Dentro del método `getEntityManager()`
 - Creamos un objeto factoría, `EntityManagerFactory`, de la misma librería.

```
EntityManagerFactory factory =  
Persistence.createEntityManagerFactory("gestor");
```

A la hora de crear la factoría, es importante que le indiquemos el nombre correcto que le hayamos dado a la unidad de persistencia en el fichero de configuración `persistence.xml` que en nuestro caso es `gestor`.

- Ahora que ya tenemos la `factory`, le pedimos el gestor de entidades, `EntityManager` que hay que devolver.

```
EntityManager manager = factory.createEntityManager();
```

De manera que el método `getEntityManager()` quedaría de la forma siguiente:

```
public static EntityManager getEntityManager() {  
    EntityManagerFactory factory =  
Persistence.createEntityManagerFactory("gestor");  
    EntityManager manager =  
factory.createEntityManager();  
    return manager;  
}
```

Nuestra clase `EntityManagerUtil` quedaría así:

```
package es.studium.hibernate.utiles;  
  
import javax.persistence.EntityManager;  
import javax.persistence.EntityManagerFactory;  
import javax.persistence.Persistence;  
  
public class EntityManagerUtil {  
  
    public static EntityManager getEntityManager() {  
        EntityManagerFactory factory =  
Persistence.createEntityManagerFactory("gestor");  
        EntityManager manager =  
factory.createEntityManager();  
        return manager;  
    }  
}
```

Pero para comprobar que funciona correctamente vamos a probarla, ya que hemos comentado anteriormente que esta clase es vital para el funcionamiento de nuestro programa, y para ello añadimos un pequeño método `main`, simplemente para comprobar que todo está bien conectado.

El código es el siguiente:

```
public static void main(String[] args) {  
    EntityManager manager =  
EntityManagerUtil.getEntityManager();  
    System.out.println("EntityManager class ==> " +  
manager.getClass().getCanonicalName());  
}
```

Donde vemos que llamamos al método que hemos creado y luego pintamos por pantalla.

Por lo tanto, el código completo de nuestra clase `EntityManagerUtil`, será el siguiente:

Clase EntityManagerUtil

```
package es.studium.hibernate.utiles;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

public class EntityManagerUtil {

    public static EntityManager getEntityManager() {
        EntityManagerFactory factory =
        Persistence.createEntityManagerFactory("gestor");
        EntityManager manager =
        factory.createEntityManager();
        return manager;
    }

    public static void main(String[] args) {
        EntityManager manager =
        EntityManagerUtil.getEntityManager();
        System.out.println("EntityManager class ==> " +
        manager.getClass().getCanonicalName());
    }
}
```

9.2.1 Código del método getEntityManager()

```
public static EntityManager getEntityManager() {
    EntityManagerFactory factory =
    Persistence.createEntityManagerFactory("gestor");
    EntityManager manager =
    factory.createEntityManager();
    return manager;
}
```

En la implementación de este método, creamos un objeto factoría, `EntityManagerFactory`, de la librería de persistencia de Java EE.

```
EntityManagerFactory factory =
Persistence.createEntityManagerFactory("gestor");
```

`EntityManagerFactory` es una Interfaz que podemos encontrar en el **paquete** `javax.persistence` de la API Java EE.

El objeto `EntityManagerFactory` lo creamos al llamar al método `createEntityManagerFactory("gestor")` método abstracto de la **clase** `Persistence` que encontraremos también en el paquete `javax.persistence` de la API Java EE.

<https://javaee.github.io/javaee-spec/javadocs/>

The screenshot shows the Java EE API documentation for the `javax.persistence` package. The left sidebar lists various interfaces and classes. The main content area is titled "Package javax.persistence" and includes a description: "Java Persistence is the API for the management for persistence and object/relational mapping." Below this, there is a table titled "Interface Summary" with two columns: "Interface" and "Description".

Interface	Description
<code>AttributeConverter<X,Y></code>	A class that implements this interface can be used to convert entity attribute state into database column representation and back again.
<code>AttributeNode<T></code>	Represents an attribute node of an entity graph.
<code>Cache</code>	Interface used to interact with the second-level cache.
<code>EntityGraph<T></code>	This type represents the root of an entity graph that will be used as a template to define the attribute nodes and boundaries of a graph of entities and entity relationships.
<code>EntityManager</code>	Interface used to interact with the persistence context.
<code>EntityManagerFactory</code>	Interface used to interact with the entity manager factory for the persistence unit.
<code>EntityTransaction</code>	Interface used to control transactions on resource-local entity managers.
<code>Parameter<T></code>	Type for query parameter objects.
<code>PersistenceUnitUtil</code>	Utility interface between the application and the persistence provider managing the persistence unit.

Observando en la API de Java EE, vemos que el método `createEntityManagerFactory("gestor")` devuelve un objeto `EntityManagerFactory` para la unidad de persistencia que le pasamos como parámetro.

The screenshot shows the Java EE API documentation for the `EntityManagerFactory` interface. The left sidebar lists various interfaces and classes. The main content area is titled "Method Summary" and includes a table with two columns: "Modifier and Type" and "Method and Description".

Modifier and Type	Method and Description
<code>static EntityManagerFactory</code>	<code>createEntityManagerFactory(String persistenceUnitName)</code> Create and return an <code>EntityManagerFactory</code> for the named persistence unit.
<code>static EntityManagerFactory</code>	<code>createEntityManagerFactory(String persistenceUnitName, Map properties)</code> Create and return an <code>EntityManagerFactory</code> for the named persistence unit using the given properties.
<code>static void</code>	<code>generateSchema(String persistenceUnitName, Map map)</code> Create database schemas and/or tables and/or create DDL scripts as determined by the supplied properties.
<code>static PersistenceUnitUtil</code>	<code>getPersistenceUnitUtil()</code> Return the <code>PersistenceUnitUtil</code> instance

En nuestro caso sería:

```
EntityManagerFactory factory = Persistence.createEntityManagerFactory("gestor");
```

Con este objeto `factory` llamamos al método `createEntityManager()` que pertenece a la Interfaz `EntityManagerFactory` del paquete `javax.persistence`.

Method Summary		
All Methods	Instance Methods	Abstract Methods
Modifier and Type	Method and Description	
<T> void	addNamedEntityGraph(String graphName, EntityGraph<T> entityGraph) Add a named copy of the EntityGraph to the EntityManagerFactory.	
void	addNamedQuery(String name, Query query) Define the query, typed query, or stored procedure query as a named query such that future query objects can be created from it using the createNamedQuery or createNamedStoredProcedureQuery method.	
void	close() Close the factory, releasing any resources that it holds.	
EntityManager	createEntityManager() Create a new application-managed EntityManager.	
EntityManager	createEntityManager(Map map) Create a new application-managed EntityManager with the specified Map of properties.	
EntityManager	createEntityManager(SynchronizationType synchronizationType)	

Vemos que este método nos permite crear el objeto **EntityManager** que hemos llamado **manager**.

Hemos podido comprobar, que todos los objetos y métodos utilizados en la implementación del método **getEntityManager()** están en la API de persistencia de Java EE en el paquete **javax.persistence**.

Ejecutamos nuestra clase y observamos la siguiente salida por consola.

```
ago 31, 2021 1:02:02 PM
org.hibernate.jpa.internal.util.LogHelper
logPersistenceUnitInformation
INFO: HHH000204: Processing PersistenceUnitInfo [name: gestor]
ago 31, 2021 1:02:02 PM org.hibernate.Version logVersion
INFO: HHH000412: Hibernate ORM core version 5.5.7.Final
ago 31, 2021 1:02:03 PM
org.hibernate.annotations.common.reflection.java.JavaReflectionManager <clinit>
INFO: HCANN000001: Hibernate Commons Annotations {5.1.2.Final}
ago 31, 2021 1:02:03 PM
org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl configure
WARN: HHH10001002: Using Hibernate built-in connection pool (not for production use!)
ago 31, 2021 1:02:03 PM
org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl buildCreator
INFO: HHH10001005: using driver [com.mysql.cj.jdbc.Driver] at URL [jdbc:mysql://localhost:3306/gestor]
ago 31, 2021 1:02:03 PM
org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl buildCreator
INFO: HHH10001001: Connection properties: {user=root, password=****}
ago 31, 2021 1:02:03 PM
org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl buildCreator
INFO: HHH10001003: Autocommit mode: false
```

```
ago 31, 2021 1:02:03 PM
org.hibernate.engine.jdbc.connections.internal.DriverManagerCo
nnectionProviderImpl$PooledConnections <init>
INFO: HHH000115: Hibernate connection pool size: 20 (min=1)
ago 31, 2021 1:02:04 PM org.hibernate.dialect.Dialect <init>
INFO: HHH000400: Using dialect:
org.hibernate.dialect.MySQL8Dialect
ago 31, 2021 1:02:05 PM
org.hibernate.resource.transaction.backend.jdbc.internal.DdlTr
ansactionIsolatorNonJtaImpl getIsolatedConnection
INFO: HHH10001501: Connection obtained from
JdbcConnectionAccess
[org.hibernate.engine.jdbc.env.internal.JdbcEnvironmentInitiat
or$ConnectionProviderJdbcConnectionAccess@3bc735b3] for (non-
JTA) DDL execution was not in auto-commit mode; the Connection
'local transaction' will be committed and the Connection will
be set into auto-commit mode.
ago 31, 2021 1:02:05 PM
org.hibernate.engine.transaction.jta.platform.internal.JtaPlat
formInitiator initiateService
INFO: HHH000490: Using JtaPlatform implementation:
[org.hibernate.engine.transaction.jta.platform.internal.NoJtaP
latform]
EntityManager class ==> org.hibernate.internal.SessionImpl
```

Todas las líneas salvo la última, se ven en rojo y se generan al llamar al método `getEntityManager()`.

Observando estos mensajes, reconocemos algunos de los datos de configuración de nuestro proyecto, como el nombre de la unidad de persistencia gestor, el driver mysql, el usuario root, etc. Pero no parece que hay ningún error, sino muchos mensajes de información **INFO** y un solo warning **WARN** que nos advierte de que la configuración no es adecuada para aplicaciones en producción.

9.3 El DAO abstracto

Ya **tenemos** la **Interfaz Dao** y la **clase EntityManagerUtil**. A continuación implementaremos la **clase** abstracta Dao a la que llamaremos **AbstractDao**.

- Empezaremos creando la clase **AbstractDao** en el paquete **es.studium.hibernate.dao**. Esta clase debe implementar la Interfaz Dao y debe ser una clase abstracta.

Primero indicamos que implemente la interfaz Dao e indicamos a Eclipse que nos genere el esqueleto de todos los métodos que debemos implementar de la interfaz Dao.

Siguiendo las indicaciones anteriores, nuestra **clase AbstractDao** debe tener la siguiente implementación:

```
package es.studium.hibernate.dao;

import java.util.List;
import java.util.Optional;

public abstract class AbstractDao<T> implements Dao<T> {

    @Override
    public Optional<T> get(long id) {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    public List<T> getAll() {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    public void save(T t) {
        // TODO Auto-generated method stub
    }

    @Override
    public void update(T t) {
        // TODO Auto-generated method stub
    }

    @Override
    public void delete(T t) {
        // TODO Auto-generated method stub
    }

}
```

- Indicamos los **atributos** de la clase **AbstractDao**, que son dos.

- Un objeto **EntityManager** que lo vamos a utilizar en todos los métodos. Declaramos e inicializamos el atributo:

```
private EntityManager entityManager =  
EntityManagerUtil.getEntityManager();
```

Vemos que utilizamos para ello el método estático `getEntityManager()` de la clase `EntityManagerUtil` que hemos creado anteriormente.

Al ser privado este atributo, generamos sus métodos getter y setter.

```
public EntityManager getEntityManager() {  
    return entityManager;  
}  
  
public void setEntityManager(EntityManager  
entityManager) {  
    this.entityManager = entityManager;  
}
```

Cuando declaremos el objeto `entityManager` debemos tener en cuenta que tendremos que importar el paquete en el que se encuentre la clase `EntityManagerUtil` que en nuestro caso será `import es.studium.hibernate.utiles.EntityManagerUtil;`

- Otro **atributo** que vamos a necesitar es uno **que lleve la clase (entidad) con la que vamos a trabajar**. Lo podemos declarar de la forma siguiente:

```
private Class<T> clazz;
```

pero no podemos inicializarlo, porque en la clase abstracta no sabemos qué valor tendrá la clase abstracta no sabemos todavía qué valor tendrá este atributo.

Al ser privado este atributo, generamos sus métodos getter y setter.

```
public Class<T> getClazz() {  
    return clazz;  
}  
  
public void setClazz(Class<T> clazz) {  
    this.clazz = clazz;  
}
```

- A continuación veremos las **consultas**. Recordemos que las consultas a la base de datos no la modifican.

Tendremos **dos métodos de consulta**, por **id** y **todos**.

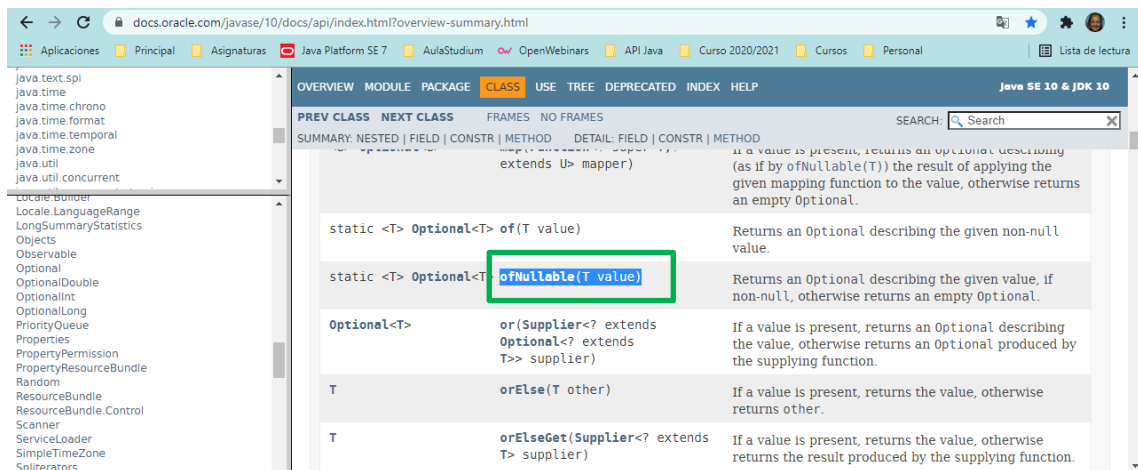
- Empezamos por la **búsqueda por id**:

```
@Override
public Optional<T> get(long id) {
    return Optional.ofNullable(entityManager.find(clazz, id));
}
```

Le pedimos al **entityManager** que busque, llamando al método **find**. Le pasamos la clase de la entidad, para que sepa en qué tabla buscar, y el id, para que sepa darnos el registro que estamos buscando.

Lo envolvemos todo con **Optional.ofNullable** para asegurarnos de que se maneja bien en caso de que no encuentre nada.

ofNullable es un método estático de la clase **Optional** que se encuentra en el paquete **java.util**.



Observamos que el método **ofNullable** devuelve un Opcional que describe el valor dado, si no es nulo; de lo contrario, devuelve un Opcional vacío.

La clase **Optional** suele utilizarse como tipo de retorno de un método donde hay posibilidad de que el uso de *null* cause errores.

- Vemos el método **get** que nos **devuelve todos los registros de una tabla**.

```
@Override
public List<T> getAll() {
    String qlString = "FROM " + clazz.getName();
    Query query = entityManager.createQuery(qlString);
    return query.getResultList();
}
```

En este método, lo primero que hacemos es preparar una **query**, que será como un trocito de lo que sería un SELECT en SQL: **FROM** y el **nombre de la tabla**, que lo obtenemos de **clazz.getName()**.

```
String qlString = "FROM " + clazz.getName();
```

Nuestra **query** se la pasamos al método **createQuery(qlString)** del **entityManager** para crear la query, y una vez tenemos la query le pedimos la lista de resultados.

El método **createQuery(qlString)** es un método de la **interfaz EntityManager** que podemos encontrar en el paquete **javax.persistence** de la API JEE como vemos en la imagen siguiente:

<https://javaee.github.io/javaee-spec/javadocs/>



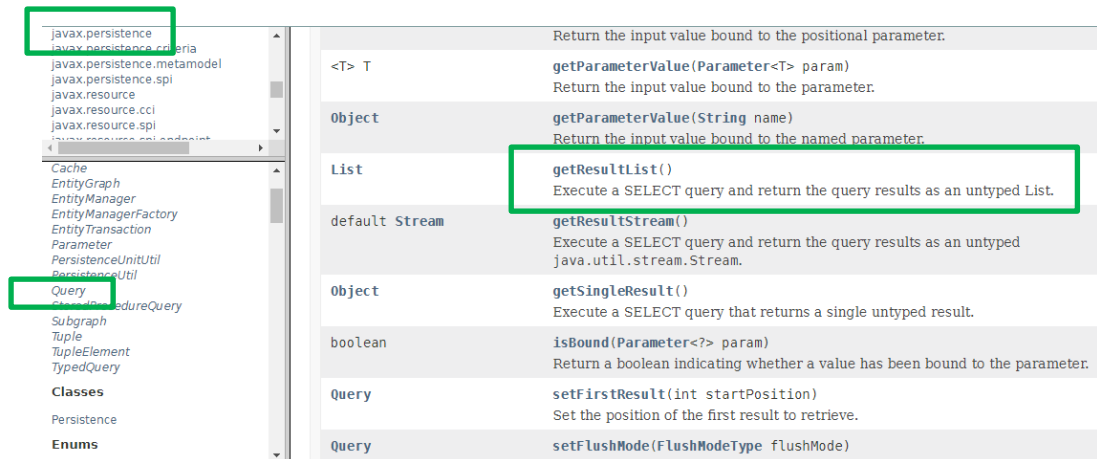
Donde vemos que le pasamos un String con la query, que en nuestro caso es **qlString**, y nos devuelve un objeto Query.

Como este método **createQuery(qlString)** pertenece a la interfaz **EntityManager** lo llamamos con el objeto **entityManager** que habíamos creado anteriormente como atributo de la clase **AbstractDao<T>**.

```
Query query = entityManager.createQuery(qlString);
```

De esta manera, tenemos creada la **query** y llamando al método **getResultList()** de la interfaz Query, tenemos la lista de resultados.

Consultamos la API Java EE y en el paquete **javax.persistence** encontramos la **interfaz Query**.



Luego con `query.getResultList()` nos devuelve un objeto **List** que contiene la lista de todos los resultados obtenidos al ejecutar nuestra consulta `qlString`.

- **Métodos de modificación: guardar, actualizar, borrar.**

Como estos métodos modifican la base de datos, deberíamos ser capaces de deshacer si hay algún problema y también de indicar cuándo empezamos y terminamos la transacción.

Y como eso va a ser lo mismo, hagamos la operación que hagamos, lo sacaremos a un método privado que implementaremos de la forma siguiente:

```
private void
executeInsideTransaction(Consumer<EntityManager> action) {
    EntityManager tx =
    entityManager.getTransaction();
    try {
        tx.begin();
        action.accept(entityManager);
        tx.commit();
    } catch (RuntimeException e) {
        tx.rollback();
        throw e;
    }
}
```

Observamos en este método que le pedimos una transacción al `entityManager.getTransaction()` llamando al método `getTransaction()` con el objeto `entityManager`. Este método `getTransaction()` devuelve un objeto `EntityManager`.

```
EntityManager tx = entityManager.getTransaction();
```

javax.persistence	Get the properties and hints and associated values that are in effect for the entity manager.
javax.persistence.criteria	
javax.persistence.metamodel	
javax.persistence.spi	
javax.resource	
javax.resource.cci	
javax.resource.spi	
javax.resource.spe	
Cache	
EntityGraph	
EntityManager	
EntityManagerFactory	
EntityTransaction	
Parameter	
PersistenceUnitUtil	
PersistenceUtil	
Query	
StoredProcedureQuery	
Subgraph	
Tuple	
TupleElement	
TupleQuery	
Classes	
Persistence	
Enums	

<T> T	getReference(Class<T> entityClass, Object primaryKey) Get an instance, whose state may be lazily fetched.
EntityTransaction	getTransaction() Return the resource-level EntityTransaction object.
boolean	isJoinedToTransaction() Determine whether the entity manager is joined to the current transaction.
boolean	isOpen() Determine whether the entity manager is open.
void	joinTransaction() Indicate to the entity manager that a JTA transaction is active and join the persistence context to it.
void	lock(Object entity, LockModeType lockMode) Lock an entity instance that is contained in the persistence context with the specified lock mode type.
void	lock(Object entity, LockModeType lockMode, Map<String, Object> properties) Lock an entity instance that is contained in the persistence context with the specified

A continuación, dentro de un **bloque try/catch**, para gestionar las excepciones que puedan producirse, **intentamos iniciar la transacción** que hemos creado **tx.begin()**; ya que la interfaz **EntityTransaction** tiene el método **begin()**.

javax.persistence

javax.persistence.criteria

javax.persistence.metamodel

javax.persistence.spi

javax.resource

javax.resource.cci

javax.resource.spi

javax.resource.spi.adapter

Cache

EntityGraph

EntityManager

EntityManagerFactory

EntityTransaction

Parameter

PersistenceUnitUtil

PersistenceUtil

Query

StoredProcedureQuery

Subgraph

Tuple

TupleElement

TupleQuery

Classes

Persistence

Enums

Method Summary

All MethodsInstance MethodsAbstract Methods

Modifier and Type	Method and Description
void	begin() Start a resource transaction.
void	commit() Commit the current resource transaction, writing any unflushed changes to the database.
boolean	getRollbackOnly() Determine whether the current resource transaction has been marked for rollback.
boolean	isActive() Indicate whether a resource transaction is in progress.
void	rollback() Roll back the current resource transaction.
void	setRollbackOnly() Mark the current resource transaction so that the only possible outcome of the transaction is for the transaction to be rolled back.

Seguidamente **ejecutamos la acción**, ya sea la acción de guardar, actualizar o borrar y hacemos el **commit de la transacción** para que se haga efectiva.

```
action.accept(entityManager);  
tx.commit();
```

Y si hay algún problema, deshacemos los cambios llamando al método **tx.rollback()**; y lanzamos la excepción para manejarla más arriba en las llamadas al método **executeInsideTransaction(Consumer<EntityManager> action)**.

Implementando este método, veremos cómo los **métodos save, update y delete** quedan más sencillos.

Nota: El objeto que le pasamos como parámetro al método **executeInsideTransaction(Consumer<EntityManager> action)** es **Consumer<EntityManager> action**.

Consumer<T> es una **Interfaz Funcional** de Java que se encuentra en el paquete **java.util.function**.

The first screenshot shows the Java SE 10 & JDK 10 documentation for the **java.util.function** package. The package is highlighted, and the **Consumer** interface is listed under the **Interfaces** section.

The second screenshot shows the details of the **Consumer<T>** interface. The interface is highlighted, and its details are shown, including its type parameters, subinterfaces, and functional interface status.

action.accept(entityManager); el método **accept()** es uno de los métodos de **Consumer<T>**.

The screenshot shows the Java SE 10 & JDK 10 documentation for the **Consumer<T>** interface. The **accept(T t)** method is highlighted, and its details are shown, including its modifier and type, method name, and description.

Este método realiza esta operación, indicada por **action**, en el argumento dado como parámetro, que es **entityManager**. Recordemos que **action** representa guardar, actualizar o borrar.

La **<T>** de **Consumer<T>** en nuestro caso es **<EntityManager>**.

- Primera acción: Método **save(T t)**


```
@Override
public void save(T t) {
    executeInsideTransaction(entityManager ->
        entityManager.persist(t));
}
```

Al `entityManager` le pedimos que haga persistir el objeto recibido, y esa acción se la pasamos al método que hemos hecho para gestionar las transacciones.

En el método `save(T t)` estamos utilizando el método `executeInsideTransaction(...)` que hemos creado antes. Al llamar a este método, le tenemos que pasar un objeto de tipo `Consumer<EntityManager>` que como ya sabemos, al ser una Interfaz Funcional, podemos pasarle una expresión lambda en la que el tipo de dato del parámetro debe ser `<EntityManager>`. La expresión lambda que hemos utilizado es `entityManager -> entityManager.persist(t)` con la que, utilizando el método `persist(t)` de la interfaz `EntityManager`, conseguimos la persistencia del objeto recibido como parámetro.

Y el método `save(T t)` queda implementado de esta forma tan sencilla.

- Segunda acción: Método **update(T t)**

```
public void update(T t) {
    executeInsideTransaction(entityManager ->
        entityManager.merge(t));
}
```

Este método es igual al anterior, la diferencia es que utilizaremos el método `merge(t)` de la interfaz `EntityManager`.

- Tercera acción: Método **delete(T t)**

```
@Override
public void delete(T t) {
    executeInsideTransaction(entityManager ->
        entityManager.remove(t));
}
```

Este método es igual a los anteriores, la diferencia es que utilizaremos el método `remove(t)` de la interfaz `EntityManager`.

Clase AbstractDao

```
package es.studium.hibernate.dao;
```



```
import java.util.List;
import java.util.Optional;
import java.util.function.Consumer;

import javax.persistence.EntityManager;
import javax.persistence.EntityTransaction;
import javax.persistence.Query;

import es.studium.hibernate.utiles.EntityManagerUtil;

public abstract class AbstractDao<T> implements Dao<T> {

    private EntityManager entityManager =
EntityManagerUtil.getEntityManager();
    private Class<T> clazz;

    public EntityManager getEntityManager() {
        return entityManager;
    }

    public void setEntityManager(EntityManager
entityManager) {
        this.entityManager = entityManager;
    }

    public Class<T> getClazz() {
        return clazz;
    }

    public void setClazz(Class<T> clazz) {
        this.clazz = clazz;
    }

    @Override
    public Optional<T> get(long id) {
        return
Optional.ofNullable(entityManager.find(clazz, id));
    }

    @Override
    public List<T> getAll() {
        String qlString = "FROM " + clazz.getName();
        Query query =
entityManager.createQuery(qlString);
        return query.getResultList();
    }

    private void
executeInsideTransaction(Consumer<EntityManager> action) {
```

```
        EntityTransaction tx =
entityManager.getTransaction();
        try {
            tx.begin();
            action.accept(entityManager);
            tx.commit();
        } catch (RuntimeException e) {
            tx.rollback();
            throw e;
        }
    }

    @Override
    public void save(T t) {
        executeInsideTransaction(entityManager ->
entityManager.persist(t));
    }

    @Override
    public void update(T t) {
        executeInsideTransaction(entityManager ->
entityManager.merge(t));
    }

    @Override
    public void delete(T t) {
        executeInsideTransaction(entityManager ->
entityManager.remove(t));
    }
}
```

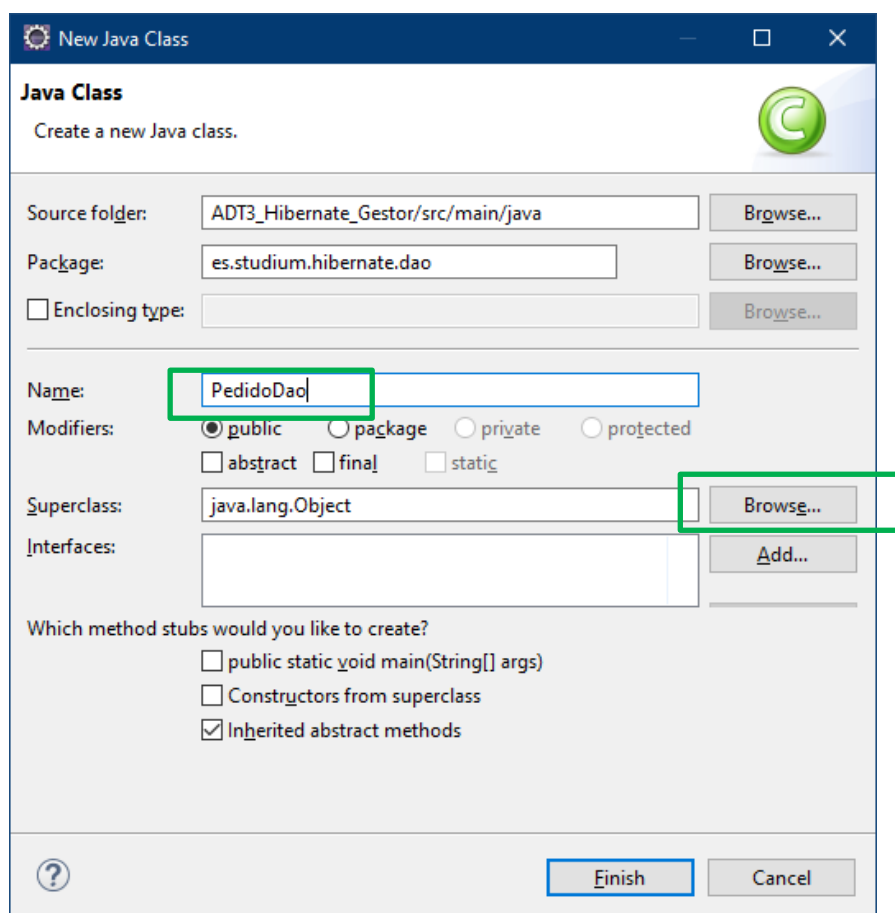
9.4 Los DAO del gestor

Ahora tenemos que crear una clase DAO por cada Entidad, ya que tenemos la **interfaz Dao** y la **clase AbstractDao**.

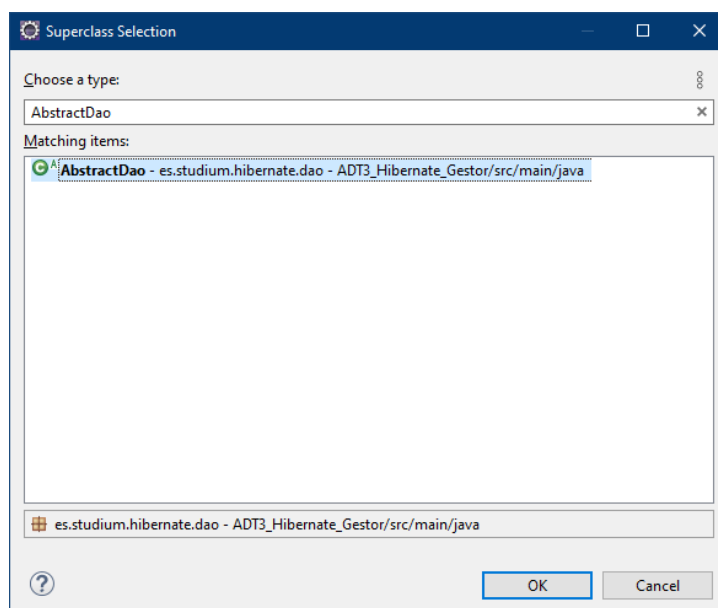
Debemos tener en cuenta, que todas las **clases XxxDao** que vamos a crear ahora, vana heredar de la clase **AbstractDao**.

Como ya tenemos creada la entidad Pedido, vamos a comenzar por crear la **clase PedidoDao** que extiende de **AbstractDao**.

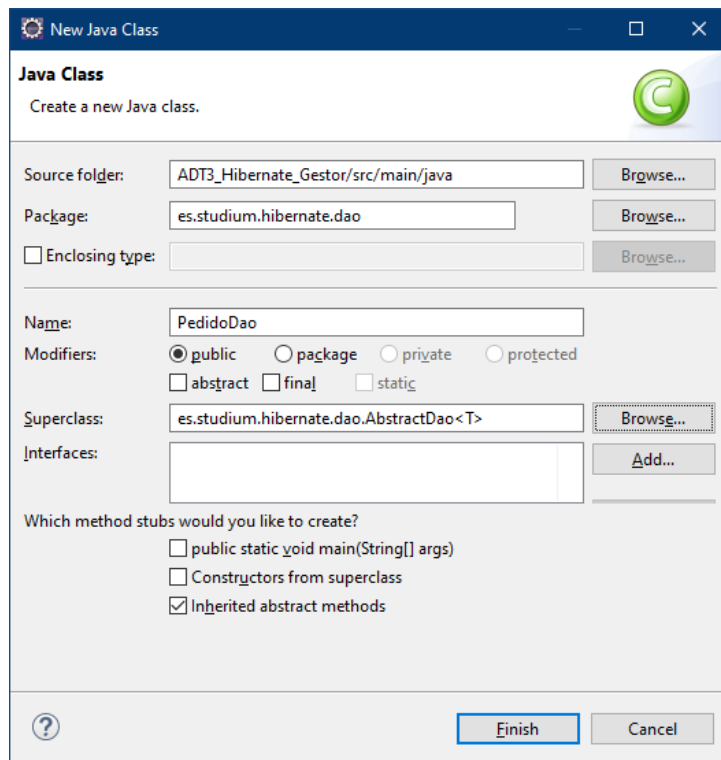
- Creamos la **clase PedidoDao** dentro del paquete **es.studium.hibernate.dao** y podemos utilizar el asistente de Eclipse para ello.



Introducimos el nombre de la clase y antes de darle a Finish, pinchamos en Browse para buscar la superclase de PedidoDao, que será la clase AbstractDao. Lo indicamos en el campo **Choose a type**, de la pantalla que aparece y seleccionamos la superclase:



Ok



Finish y se crea la **clase PedidoDao**.

```
package es.studium.hibernate.dao;  
  
public class PedidoDao extends AbstractDao<T> {  
}
```

- Observamos que aparece un error en la <T> al crear la clase PedidoDao, y es debido a que debemos concretar. Tenemos que sustituir esta T genérica por una clase concreta, que será la clase de la entidad a la que estemos haciéndole el Dao, en este caso Pedido.

```
PedidoDao.java  
1 package es.studium.hibernate.dao;  
2  
3 public class PedidoDao extends AbstractDao<T> {  
4  
5 }
```

De manera que nuestra clase PedidoDao queda de la siguiente forma.

```
package es.studium.hibernate.dao;  
import es.studium.hibernate.Pedido;  
  
public class PedidoDao extends AbstractDao<Pedido> {  
  
}
```

```
PedidoDao.java
1 package es.studium.hibernate.dao;
2
3 import es.studium.hibernate.Pedido;
4
5 public class PedidoDao extends AbstractDao<Pedido> {
6
7 }
```

- Recordemos que en la clase abstracta nos falta determinar la clase con la que trabajamos, por ello en la clase PedidoDao añadimos el constructor por defecto de PedidoDao e informamos de nuestra clase Pedido.

```
public PedidoDao() {
    setClazz(Pedido.class);
}
```

Donde utilizamos el método `setClazz (Class<T> clazz)` de la clase abstracta, para establecer la clase que estamos utilizando como entidad.

- Ya tenemos creado nuestro PedidoDao. Podemos probarlo, creando una clase Principal en el paquete `es.studium.hibernate` con el siguiente código:

Clase Principal para probar la clase PedidoDao.

```
package es.studium.hibernate;

import java.util.List;
import es.studium.hibernate.dao.PedidoDao;

public class Principal {

    public static void main(String[] args) {

        PedidoDao pedidoDao = new PedidoDao();

        List<Pedido> pedidos = pedidoDao.getAll();

        System.out.println("Lista de pedidos: " +
pedidos);
    }
}
```

Creamos un PedidoDao y llamamos al método `getAll()` de la clase AbstractDao para que nos devuelva el listado de los pedidos que tenemos en la base de datos.

Ejecutamos nuestra clase Principal y observamos que no devuelve nada, porque la base de datos está vacía, pero tampoco nos da ningún error.

```
sep 02, 2021 11:08:07 AM
org.hibernate.jpa.internal.util.LogHelper
logPersistenceUnitInformation
INFO: HHH000204: Processing PersistenceUnitInfo [name: gestor]
sep 02, 2021 11:08:08 AM org.hibernate.Version logVersion
INFO: HHH000412: Hibernate ORM core version 5.5.7.Final
sep 02, 2021 11:08:08 AM
org.hibernate.annotations.common.reflection.java.JavaRefleccio
nManager <clinit>
INFO: HCANN000001: Hibernate Commons Annotations {5.1.2.Final}
sep 02, 2021 11:08:08 AM
org.hibernate.engine.jdbc.connections.internal.DriverManagerCo
nnectionProviderImpl configure
WARN: HHH10001002: Using Hibernate built-in connection pool
(not for production use!)
sep 02, 2021 11:08:08 AM
org.hibernate.engine.jdbc.connections.internal.DriverManagerCo
nnectionProviderImpl buildCreator
INFO: HHH10001005: using driver [com.mysql.cj.jdbc.Driver] at
URL [jdbc:mysql://localhost:3306/gestor]
sep 02, 2021 11:08:08 AM
org.hibernate.engine.jdbc.connections.internal.DriverManagerCo
nnectionProviderImpl buildCreator
INFO: HHH10001001: Connection properties: {user=root,
password=****}
sep 02, 2021 11:08:08 AM
org.hibernate.engine.jdbc.connections.internal.DriverManagerCo
nnectionProviderImpl buildCreator
INFO: HHH10001003: Autocommit mode: false
sep 02, 2021 11:08:08 AM
org.hibernate.engine.jdbc.connections.internal.DriverManagerCo
nnectionProviderImpl$PooledConnections <init>
INFO: HHH000115: Hibernate connection pool size: 20 (min=1)
sep 02, 2021 11:08:09 AM org.hibernate.dialect.Dialect <init>
INFO: HHH000400: Using dialect:
org.hibernate.dialect.MySQL8Dialect
sep 02, 2021 11:08:10 AM
org.hibernate.resource.transaction.backend.jdbc.internal.DdlTr
ansactionIsolatorNonJtaImpl getIsolatedConnection
INFO: HHH10001501: Connection obtained from
JdbcConnectionAccess
[org.hibernate.engine.jdbc.env.internal.JdbcEnvironmentInitiat
or$ConnectionProviderJdbcConnectionAccess@2e1792e7] for (non-
JTA) DDL execution was not in auto-commit mode; the Connection
'local transaction' will be committed and the Connection will
be set into auto-commit mode.
Hibernate: create table pedido (id integer not null
auto_increment, fecha datetime(6), referencia varchar(255),
primary key (id)) engine=InnoDB
```

```
sep 02, 2021 11:08:10 AM
org.hibernate.engine.transaction.jta.platform.internal.JtaPlatformInitiator initiateService
INFO: HHH000490: Using JtaPlatform implementation:
[org.hibernate.engine.transaction.jta.platform.internal.NoJtaPlatform]
Hibernate: select pedido0_.id as id1_0_, pedido0_.fecha as fecha2_0_, pedido0_.referencia as referenc3_0_ from pedido
pedido0_
Lista de pedidos: []
```

Al ejecutar la clase Principal, observamos que se ha creado la **tabla Pedido** en la **base de datos gestor**, pero está vacía.



- Añadimos en la clase Principal, la creación y guardado de un pedido en la base de datos y la ejecutamos observando los resultados.

Clase Principal

```
package es.studium.hibernate;

import java.util.Date;
import java.util.List;
import es.studium.hibernate.dao.PedidoDao;

public class Principal {

    public static void main(String[] args) {

        PedidoDao pedidoDao = new PedidoDao();

        Pedido pedido = new Pedido();
        pedido.setFecha(new Date());
        pedido.setReferencia("001");
        pedidoDao.save(pedido);

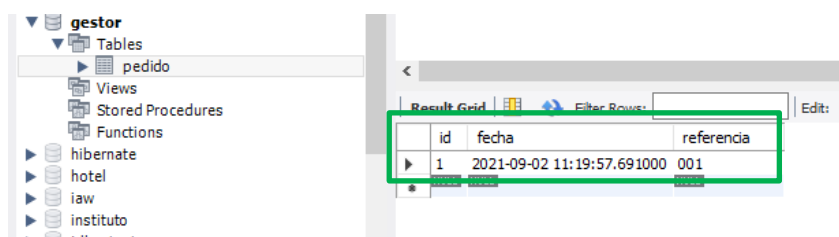
        List<Pedido> pedidos = pedidoDao.getAll();
        System.out.println("Lista de pedidos: " +
pedidos);
    }
}
```

Ejecutamos nuestra clase Principal y observamos la siguiente salida:

Lista de pedidos: [Pedido [id=1, referencia=001, fecha=Thu Sep 02 11:19:57 CEST 2021]]

Gracias al método toString que tenemos implementado en la clase Pedido, se ha impreso de forma legible el pedido que hemos creado en la tabla Pedido de nuestra base de datos gestor, al ejecutar la clase Principal.

Podemos ir a MySQL Workbench y comprobar que se ha generado el registro de este pedido:



Como podemos comprobar, nuestra clase PedidoDao funciona correctamente. Se ha creado la tabla Pedido y se han registrado los pedidos en la tabla.

9.5 Consultas simples

Con la implementación que tenemos de PedidoDao concretando AbstractDao, podemos recuperar un registro por id, recuperar todos los registros y guardar, modificar o borrar un registro, de la tabla Pedido de nuestra base de datos gestor.

Pero puede ser que necesitemos realizar otro tipo de operaciones, sobre todo de consultas, como podría ser recuperar la lista de pedidos de la semana pasada o el pedido más reciente.

Para atender estas dos peticiones, tendremos que añadir dos métodos nuevos a la clase PedidoDao.

9.5.1 Consulta del pedido más reciente

Añadiremos el método `pedidoMasReciente()` en la clase PedidoDao con la siguiente implementación:

```
public Pedido pedidoMasReciente() {  
    String queryString = "FROM " + Pedido.class.getName()  
    + " WHERE fecha < now() order by fecha desc";  
}
```



```
Query query =  
getEntityManager().createQuery(qlString).setMaxResults(1);  
    return (Pedido) query.getSingleResult();  
}
```

Este método nos **devolverá un Pedido**.

- Preparamos la *query*:

```
String qlString = "FROM " + Pedido.class.getName() + " WHERE  
fecha < now() order by fecha desc";
```

Queremos recuperar de la clase/tabla Pedido, aquellos registros con fecha inferior a la actual, y ordenados por fecha, para obtener el más reciente de los que cumplan esa condición.

Para saber la fecha actual, podemos usar la función `now()` en la *query*.

- Ahora tenemos que usar la *query qlString* que acabamos de crear. Y lo hacemos de la misma forma que hemos visto anteriormente, le pasamos la *query qlString* como parámetro al método `createQuery(qlString)` que llamamos con el objeto `entityManager` es decir:

```
Query query = entityManager.createQuery(qlString)
```

Pero si lo ponemos así, Eclipse nos da un error porque nos indica que `entityManager` es **private** es decir no es accesible, tenemos que acceder a él a través de su método `getEntityManager()` que tenemos creado en la clase `AbstractDao`.

Luego la forma correcta de implementarlo, es la siguiente, sustituimos `entityManager` por `getEntityManager()`:

```
Query query = getEntityManager().createQuery(qlString)
```

Todavía nos queda, decirle a la *query* que solo queremos un resultado y eso lo hacemos con el método `setMaxResults(1)` que establece el máximo número de resultados que se recuperan.

Luego nuestra *query* quedará de la forma siguiente:

```
Query query = getEntityManager().createQuery(qlString).setMaxResults(1);
```

Ya tenemos la *query*.

- Nos falta el **return** que tiene que devolvernos el método, que será un `Pedido`. Para ello, con la *query* llamamos al método `getSingleResult()`

que nos permite ejecutar una consulta SELECT que devuelve un único resultado.

```
query.getSingleResult();
```

el método `getSingleResult()` devuelve un **Object** por eso tendremos que hacer un casting a tipo (Pedido).

```
return (Pedido) query.getSingleResult();
```

Probamos el método que acabamos de crear, para ello tenemos que asegurarnos de que existen pedidos creados en la tabla Pedido, con distintas fechas y de esa forma nuestro método tendrá que devolvernos, solamente uno que será, el más reciente.

Luego nuestra clase Principal tendrá que crear pedidos con distintas fechas y para ello lo implementaremos así:

Clase Principal

```
package es.studium.hibernate;

import java.time.LocalDateTime;
import java.time.temporal.ChronoUnit;
import es.studium.hibernate.dao.PedidoDao;

public class Principal {

    public static void main(String[] args) {
        PedidoDao pedidoDao = new PedidoDao();

        /*Pedido creado con la fecha actual*/
        Pedido pedido2 = new Pedido("001",
LocalDateTime.now());
        pedidoDao.save(pedido2);

        /*Pedido creado con fecha de pasado mañana, 2 días
más de la fecha actual*/
        Pedido pedido3 = new Pedido("pedidoFuturo",
LocalDateTime.now().plus(2, ChronoUnit.DAYS));
        pedidoDao.save(pedido3);

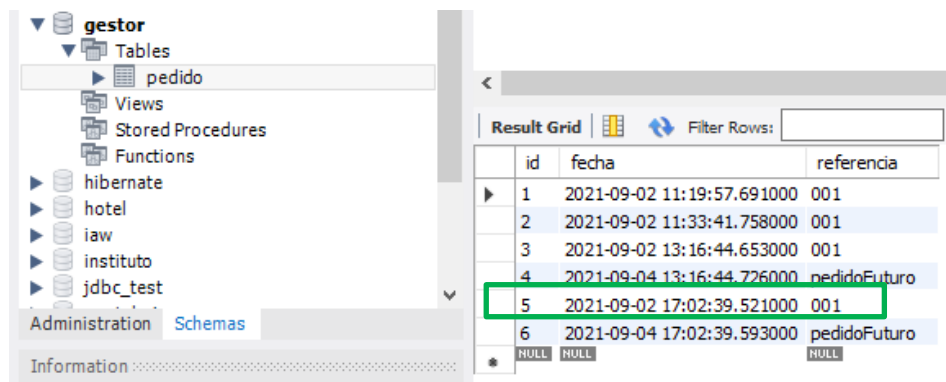
        /*Llamamos al método pedidoMasReciente() e
imprimimos el resultado*/
        Pedido masReciente = pedidoDao.pedidoMasReciente();
        System.out.println("Pedido más reciente: " +
masReciente);
    }
}
```

Ejecutamos la clase Principal y el resultado es el pedido más reciente:

Pedido más reciente: Pedido [id=5, referencia=001, fecha=2021-09-02T17:02:39.521]

Que es el de referencia 001 con id=5.

Si observamos los pedidos que tenemos registrados en la tabla pedido de nuestra base de datos gestor, con MySQL Workbench, vemos el siguiente resultado, porque he ejecutado dos veces la clase Principal:



	id	fecha	referencia
1	2021-09-02 11:19:57.691000	001	
2	2021-09-02 11:33:41.758000	001	
3	2021-09-02 13:16:44.653000	001	
4	2021-09-04 13:16:44.726000	pedidoFuturo	
5	2021-09-02 17:02:39.521000	001	
6	2021-09-04 17:02:39.593000	pedidoFuturo	
*	NULL	NULL	NULL

- ¿Qué pasaría si la query no encuentra nada?

Pongamos el caso en que nuestra query fuese la siguiente:

```
String qlString = "FROM " + Pedido.class.getName() + " WHERE  
fecha < now() and id > 100 order by fecha desc";
```

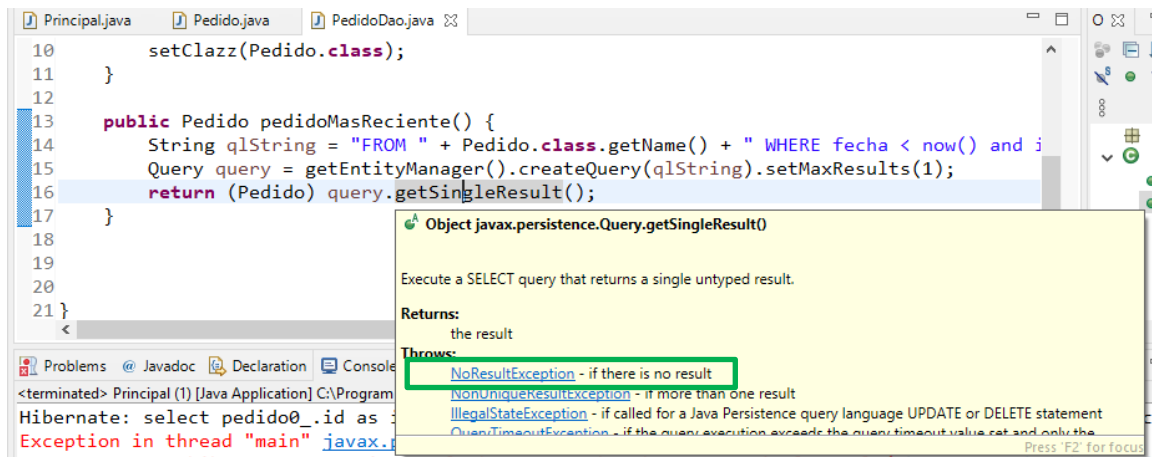
Ejecutamos nuestra clase Principal, con esta nuevo query, y el resultado es una excepción:

```
Exception in thread "main"  
javax.persistence.NoResultException: No entity found for  
query  
    at  
    org.hibernate.query.internal.AbstractProducedQuery.getSingle  
Result(AbstractProducedQuery.java:1667)  
    at  
    es.studium.hibernate.dao.PedidoDao.pedidoMasReciente(PedidoD  
ao.java:16)  
    at  
    es.studium.hibernate.Principal.main(Principal.java:32)
```

Vemos que el problema está en la línea 16 de la clase PedidoDao:

```
return (Pedido) query.getSingleResult();
```

que es la línea en la que llamamos al método `getSingleResult()`, nos ponemos sobre el método y el javadoc nos indica las posibles excepciones que puede lanzar y cuándo, este método.



Vemos que la excepción `NoResultException` es lanzada cuando no hay resultados.

Luego lo que tenemos que hacer es tratarla en nuestro código metiendo un bloque `try/catch` en el método `main` en la llamada al método `getSingleResult()`.

```
try {  
    /*Llamamos al método pedidoMasReciente() e imprimimos el resultado*/  
    Pedido masReciente = pedidoDao.pedidoMasReciente();  
    System.out.println("Pedido más reciente: " + masReciente);  
} catch (NoResultException e) {  
    System.out.println("No hay pedidos recientes.");  
}
```

La **clase Principal** queda de la siguiente forma, **tratando la excepción** en un bloque `try/catch`:

```
package es.studium.hibernate;  
  
import java.time.LocalDateTime;  
import java.time.temporal.ChronoUnit;  
  
import javax.persistence.NoResultException;  
  
import es.studium.hibernate.dao.PedidoDao;  
  
public class Principal {  
  
    public static void main(String[] args) {  
        PedidoDao pedidoDao = new PedidoDao();  

```

```
        /*Pedido creado con la fecha acrtual*/
        Pedido pedido2 = new Pedido("001",
LocalDateTime.now());
        pedidoDao.save(pedido2);

        /*Pedido creado con fecha de pasado mañana, 2
días más de la fecha actual*/
        Pedido pedido3 = new Pedido("pedidoFuturo",
LocalDateTime.now().plus(2, ChronoUnit.DAYS));
        pedidoDao.save(pedido3);
        try {
            /*Llamamos al método pedidoMasReciente() e
imprimimos el resultado*/
            Pedido masReciente =
pedidoDao.pedidoMasReciente();
            System.out.println("Pedido más reciente: " +
masReciente);
        } catch (NoResultException e) {
            System.out.println("No hay pedidos
recientes.");
        }
    }
}
```

Ejecutamos de nuevo nuestra clase Principal y el resultado es el siguiente:

```
No hay pedidos recientes.
```

9.5.2 Consulta de pedidos de la semana pasada

Añadiremos el método `pedidosSemanaPasada()` en la clase `PedidoDao` con la siguiente implementación:

```
public List<Pedido> pedidosSemanaPasada() {
    String qlString = "FROM " + Pedido.class.getName()
+ " WHERE fecha between ?1 and ?2";
    Query query =
getEntityManager().createQuery(qlString);
    LocalDate esteLunes = getEsteLunes();
    LocalDate lunesAnterior = esteLunes.minusWeeks(1);
    query.setParameter(1,
lunesAnterior.atStartOfDay());
    query.setParameter(2, esteLunes.atStartOfDay());
    return query.getResultList();
}

private static LocalDate getEsteLunes() {
```

```
    LocalDate now = LocalDate.now();
    DayOfWeek diaSemana = now.getDayOfWeek();
    return now.minusDays(diaSemana.getValue() - 1);
}
```

- Este método se llama `pedidosSemanaPasada()` y devuelve una lista de pedidos `List<Pedido>`.

```
public List<Pedido> pedidosSemanaPasada() {
}
```

- Preparamos la query:

```
String qlString = "FROM " + Pedido.class.getName() + " WHERE  
fecha between ?1 and ?2";
```

La query será esta, porque nos interesa recuperar todos los pedidos cuya fecha esté entre otras dos, por ello la query tendrá estos dos parámetros que marcamos con `?1` and `?2`. Cuando construyamos la query, le pasamos los valores a estos dos parámetros.

- Ahora, como ya hemos visto anteriormente, le pedimos la query al `entityManager` que como es un atributo privado, accedemos a él utilizando su método `get`.

```
Query query = getEntityManager().createQuery(qlString);
```

- Como tenemos que pasarle los dos parámetros a la query, buscamos la fecha del lunes de esta semana, utilizando un método auxiliar que llamaremos `getEsteLunes()` porque no es inmediato de calcular:

```
LocalDate esteLunes = getEsteLunes();
```

El valor de `esteLunes` lo obtenemos del método `getEsteLunes()`.

```
private static LocalDate getEsteLunes() {
    /*Obtenemos la fecha actual del sistema*/
    LocalDate now = LocalDate.now();

    /*Obtiene el día de la semana*/
    DayOfWeek diaSemana = now.getDayOfWeek();

    /* (diaSemana.getValue() - 1) Obtiene el día de la semana
    en número entero y le resta 1 y el método minusDays devuelve la
    fecha con el número de días restado.
    Ej.: diaSemana es jueves diaSemana.getValue() es 4,
    (diaSemana.getValue() - 1) es (4-1)=3,
```

```
luego minusDays(diaSemana.getValue() - 1) = minusDays(3) nos  
devuelve la fecha del día correspondiente a 3 días anteriores al  
actual, es decir tres días antes al jueves, es el lunes.*/  
    return now.minusDays(diaSemana.getValue() - 1);  
}
```

- Utilizaremos **esteLunes** para calcular la fecha del lunes anterior

```
LocalDate lunesAnterior = esteLunes.minusWeeks(1);
```

Llamando al método **minusWeeks(1)** que nos devuelve la fecha con el número de semanas pasada como parámetro, restada. En este caso, a la fecha obtenida en **esteLunes** que es el lunes, le restamos 1 semana, luego estamos obteniendo el lunes de la semana pasada, es decir **lunesAnterior**.

- A continuación lo que hacemos es pasarle los parámetros a la query:

```
query.setParameter(1, lunesAnterior.atStartOfDay());  
query.setParameter(2, esteLunes.atStartOfDay());
```

con el método **atStartOfDay()** lo que hacemos es conseguir que la fecha indicada en **lunesAnterior** comience a contar desde la media noche de la fecha en cuestión.

- Lo último que nos queda del método **pedidosSemanaPasada()** es el **return**.

```
return query.getResultList();
```

con todo lo anterior, ya tenemos la query lista para pedirle resultados y **getResultList()** nos devuelve la lista de resultados.

Ahora probaremos si nuestro método funciona.

Para ello, añadimos el siguiente código en nuestra **clase Principal**:

- Creamos un pedido de hace una semana.

```
Pedido pedido3 = new Pedido("pedPas",  
    LocalDateTime.now().minus(1, ChronoUnit.WEEKS));  
pedidoDao.save(pedido3);
```

- Recuperamos la lista de pedidos de la semana pasada y lo imprimimos.

```
List<Pedido> pedidosSemanaPasada =  
pedidoDao.pedidosSemanaPasada();  
System.out.println("Pedidos de la semana pasada: " +  
pedidosSemanaPasada);
```

Clase Principal

```
package es.studium.hibernate;

import java.time.LocalDateTime;
import java.time.temporal.ChronoUnit;
import java.util.List;
import javax.persistence.NoResultException;
import es.studium.hibernate.dao.PedidoDao;

public class Principal {

    public static void main(String[] args) {

        PedidoDao pedidoDao = new PedidoDao();

        Pedido pedido = new Pedido();
        pedido.setFecha(LocalDateTime.now());
        pedido.setReferencia("001");
        pedidoDao.save(pedido);

        List<Pedido> pedidos = pedidoDao.getAll();
        System.out.println("Lista de pedidos: " + pedidos);

        /*Pedido creado con la fecha actual*/
        Pedido pedido2 = new Pedido("001",
LocalDateTime.now());
        pedidoDao.save(pedido2);

        /*Pedido creado con fecha de pasado mañana, 2 días
más de la fecha actual*/
        Pedido pedido3 = new Pedido("pedidoFuturo",
LocalDateTime.now().plus(2, ChronoUnit.DAYS));
        pedidoDao.save(pedido3);

        try {
            /*Llamamos al método pedidoMasReciente() e
imprimimos el resultado*/
            Pedido masReciente = pedidoDao.pedidoMasReciente();
            System.out.println("Pedido más reciente: " +
masReciente);
        } catch (NoResultException e) {
            System.out.println("No hay pedidos
recientes.");
        }

        /*Creamos un pedido de hace una semana.*/
    }
}
```



```
Pedido pedido4 = new Pedido("pedPas",  
    LocalDateTime.now().minus(1, ChronoUnit.WEEKS));  
pedidoDao.save(pedido4);  
  
/*Recuperamos la lista de pedidos de la semana  
pasada y lo imprimimos.*/  
List<Pedido> pedidosSemanaPasada =  
pedidoDao.pedidosSemanaPasada();  
System.out.println("*** Pedidos de la semana  
pasada: " + pedidosSemanaPasada);  
}  
}
```

Ejecutamos nuestro programa y observamos que tenemos un pedido de la semana pasada en nuestra tabla Pedido de la base de datos gestor:

```
Pedidos de la semana pasada: [Pedido [id=32, referencia=pedPas,  
fecha=2021-08-27T19:33:06.605]]
```

9.6 Entidad Albaran. Relaciones 1:N

Añadiremos la **entidad Albaran** a nuestro programa en el paquete **es.studium.hibernate**.

Clase Abaran

- Atributos privados:
 - id (tipo int)
 - referencia (tipo String)
 - fechaEmision (tipo LocalDateTime)
 - fechaRecepcion (tipo LocalDateTime)
- Métodos constructores
 - Constructor por defecto
 - Constructor con el parámetro referencia
- Métodos inspectores:
 - Getters
 - Setters
- Método toString

El **id** será autogenerado y numérico.

Para la **referencia** tomaremos la del pedido y le concatenaremos el prefijo **ALB-**. La **fecha de emisión** se establecerá al crear el albarán y la **fecha de recepción** en algún momento posterior.

Con las indicaciones anteriores, nuestra clase Albaran quedará configurada de la forma siguiente.

Clase Albaran

```
package es.studium.hibernate;

import java.time.LocalDateTime;

public class Albaran {

    private static final String PREFIJO = "ALB-";

    private int id;
    private String referencia;
    private LocalDateTime fechaEmision;
    private LocalDateTime fechaRecepcion;

    public Albaran() {
    }

    public Albaran(String refPedido) {
        referencia = PREFIJO + refPedido;
        fechaEmision = LocalDateTime.now();
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getReferencia() {
        return referencia;
    }

    public void setReferencia(String referencia) {
        this.referencia = referencia;
    }

    public LocalDateTime getFechaEmision() {
        return fechaEmision;
    }

    public void setFechaEmision(LocalDateTime fechaEmision) {
        this.fechaEmision = fechaEmision;
    }

    public LocalDateTime getFechaRecepcion() {
        return fechaRecepcion;
    }
}
```

```
    }

    public void setFechaRecepcion(LocalDateTime
fechaRecepcion) {
        this.fechaRecepcion = fechaRecepcion;
    }

    @Override
    public String toString() {
        return "Albaran [id=" + id + ", referencia=" +
referencia + ", fechaEmision=" + fechaEmision
        + ", fechaRecepcion=" + fechaRecepcion +
        "]";
    }
}
```

Realizamos el mapeo en nuestra clase Albaran incluyendo las anotaciones correspondientes, de la misma manera que hemos hecho anteriormente con la entidad Pedido y el código de nuestra clase Albaran quedará de la forma siguiente.

```
package es.studium.hibernate;

import java.time.LocalDateTime;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name = "albaran")
public class Albaran {

    private static final String PREFIJO = "ALB-";

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String referencia;
    private LocalDateTime fechaEmision;
    private LocalDateTime fechaRecepcion;

    public Albaran() {
    }

    public Albaran(String refPedido) {
        referencia = PREFIJO + refPedido;
    }
}
```

```
        fechaEmision = LocalDateTime.now();
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getReferencia() {
        return referencia;
    }

    public void setReferencia(String referencia) {
        this.referencia = referencia;
    }

    public LocalDateTime getFechaEmision() {
        return fechaEmision;
    }

    public void setFechaEmision(LocalDateTime fechaEmision) {
        this.fechaEmision = fechaEmision;
    }

    public LocalDateTime getFechaRecepcion() {
        return fechaRecepcion;
    }

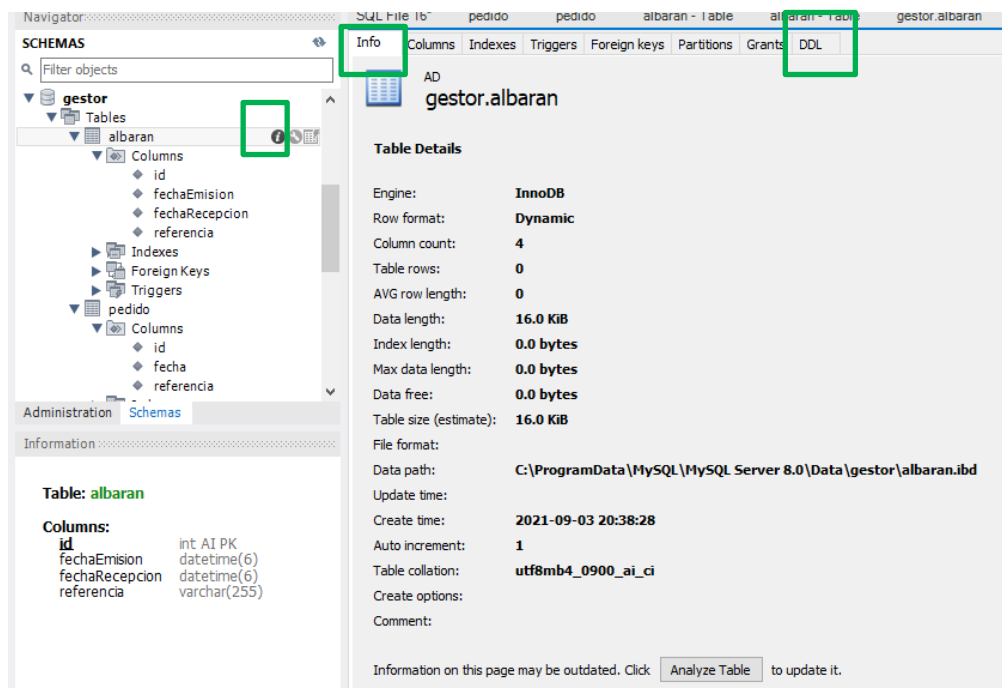
    public void setFechaRecepcion(LocalDateTime
fechaRecepcion) {
        this.fechaRecepcion = fechaRecepcion;
    }

    @Override
    public String toString() {
        return "Albaran [id=" + id + ", referencia=" +
referencia + ", fechaEmision=" + fechaEmision
        + ", fechaRecepcion=" + fechaRecepcion +
        "]\n";
    }
}
```

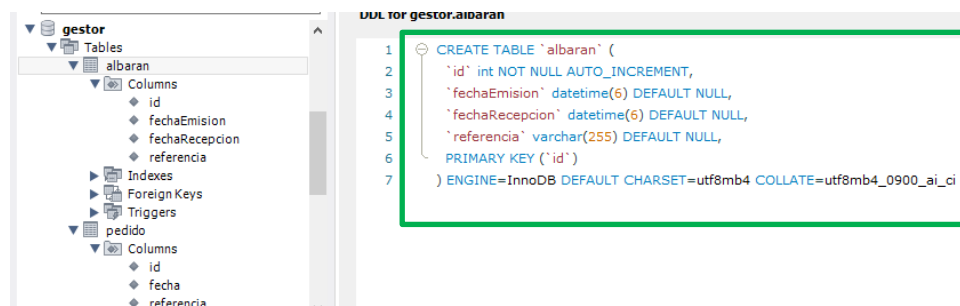
Ejecutamos la clase Principal, sin hacer ningún cambio, y vemos que Hibernate ha creado la tabla Albaran en nuestra base de datos gestor, desde MySQL Workbench.



También vemos que podemos consultar su DDL para ver la sentencia con la que se crearía esta tabla albaran desde el icono de información de la tabla, nos aparece la siguiente pantalla:



Seleccionado la pestaña DDL, nos aparece la siguiente pantalla:



Ya tenemos dos entidades, Pedido y Albaran, ahora tenemos que relacionarlas para que nuestro programa funcione correctamente.

Para relacionar Pedido y Albaran, lo primero es ver qué tipo de relación tienen y vemos que cada albarán está asociado a un pedido, pero de cada pedido se pueden generar varios albaranes. Luego la relación será 1:N (un pedido genera varios albaranes y un albarán está asociado a un único pedido).

La forma de representar esta relación en la base de datos, es añadir una columna en la tabla Albaran con el id de la tabla Pedido (idPedido).

Traducido a entidades, tenemos que **añadir un atributo Pedido en la entidad Albaran:**

```
@ManyToOne(fetch = FetchType.LAZY)
private Pedido pedido;

public Pedido getPedido() {
    return pedido;
}

public void setPedido(Pedido pedido) {
    this.pedido = pedido;
}
```

Indicamos el parámetro `fetch = FetchType.LAZY` porque no queremos llevarnos todos los datos del pedido, cuando recuperamos la información de la base de datos.

Para que Hibernate entienda todo bien, tenemos que mapear la relación al revés. Es decir, en la entidad Pedido tenemos que crear un atributo que será una lista de Albaranes, ya que la relación es 1:N.

Por lo tanto, en la entidad Pedido tendremos que añadir:

```
@OneToMany(mappedBy = "pedido", cascade = CascadeType.ALL)
private List<Albaran> albaranes = new ArrayList<Albaran>();

public List<Albaran> getAlbaranes() {
    return albaranes;
}

public void setAlbaranes(List<Albaran> albaranes) {
    this.albaranes = albaranes;
}
```

Observamos que la anotación utilizada en la entidad Albaran ha sido `@ManyToOne` y en la entidad Pedido hemos usado `@OneToMany` porque, como ya hemos comentado, la relación es que un pedido genera varios albaranes.

En el atributo `mappedBy` de `@OneToMany` indicaremos el nombre del atributo del otro lado de la relación, en este caso es `pedido` de `private Pedido pedido;` que tenemos en la clase `Albaran`.

```
@OneToMany(mappedBy = "pedido", cascade = CascadeType.ALL)
```

Nuestras clases `Pedido` y `Albaran`, quedan así:

Clase `Pedido`

```
package es.studium.hibernate;

import java.time.LocalDateTime;
import java.util.ArrayList;
import java.util.List;
import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToMany;
import javax.persistence.Table;

@Entity
@Table(name = "pedido")
public class Pedido {

    @Column(name = "id")
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

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

    @Column(name = "fecha")
    private LocalDateTime fecha;

    @OneToMany(mappedBy = "pedido", cascade =
CascadeType.ALL)
    private List<Albaran> albaranes = new
ArrayList<Albaran>();

    public Pedido() {
        id = 0;
        referencia = "";
        fecha = LocalDateTime.now();
    }
}
```

```
public Pedido(String referencia, LocalDateTime fecha) {
    this.referencia = referencia;
    this.fecha = fecha;
}

public int getId() {
    return id;
}

public void setId(int id) {
    this.id = id;
}

public String getReferencia() {
    return referencia;
}

public void setReferencia(String referencia) {
    this.referencia = referencia;
}

public LocalDateTime getFecha() {
    return fecha;
}

public void setFecha(LocalDateTime fecha) {
    this.fecha = fecha;
}

public List<Albaran> getAlbaranes() {
    return albaranes;
}

public void setAlbaranes(List<Albaran> albaranes) {
    this.albaranes = albaranes;
}

@Override
public String toString() {
    return "Pedido [id=" + id + ", referencia=" +
referencia + ", fecha=" + fecha + "];"
}
}
```

Clase Albaran

```
package es.studium.hibernate;

import java.time.LocalDateTime;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
```



```
import javax.persistence.ManyToOne;
import javax.persistence.Table;

@Entity
@Table(name = "albaran")
public class Albaran {

    private static final String PREFIJO = "ALB-";

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String referencia;
    private LocalDateTime fechaEmision;
    private LocalDateTime fechaRecepcion;

    @ManyToOne(fetch = FetchType.LAZY)
    private Pedido pedido;

    public Albaran() {
    }

    public Albaran(String refPedido) {
        referencia = PREFIJO + refPedido;
        fechaEmision = LocalDateTime.now();
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getReferencia() {
        return referencia;
    }

    public void setReferencia(String referencia) {
        this.referencia = referencia;
    }

    public LocalDateTime getFechaEmision() {
        return fechaEmision;
    }

    public void setFechaEmision(LocalDateTime fechaEmision) {
        this.fechaEmision = fechaEmision;
    }
}
```

```
}

    public LocalDateTime getFechaRecepcion() {
        return fechaRecepcion;
    }

    public void setFechaRecepcion(LocalDateTime
fechaRecepcion) {
        this.fechaRecepcion = fechaRecepcion;
    }

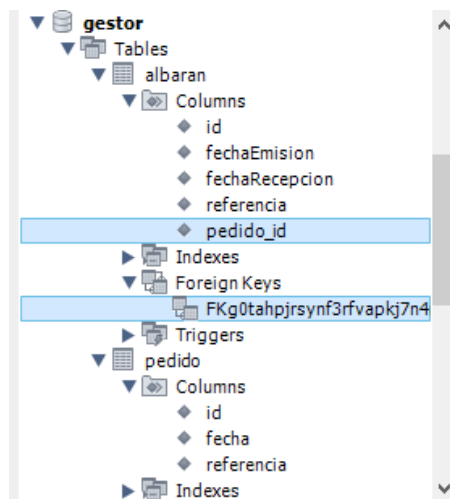
    public Pedido getPedido() {
        return pedido;
    }

    public void setPedido(Pedido pedido) {
        this.pedido = pedido;
    }

    @Override
    public String toString() {
        return "Albaran [id=" + id + ", referencia=" +
referencia + ", fechaEmision=" + fechaEmision
        + ", fechaRecepcion=" + fechaRecepcion +
        "]\n";
    }
}
```

Ejecutamos nuestro programa, sin hacer ninguna modificación en la clase Principal y observamos los cambios que se han producido en la base de datos.

Vemos que la tabla pedido sigue teniendo las mismas columnas, pero en la tabla albaran ha aparecido una columna pedido_id y en la sección de claves foráneas, ha aparecido una como podemos ver en la siguiente imagen.

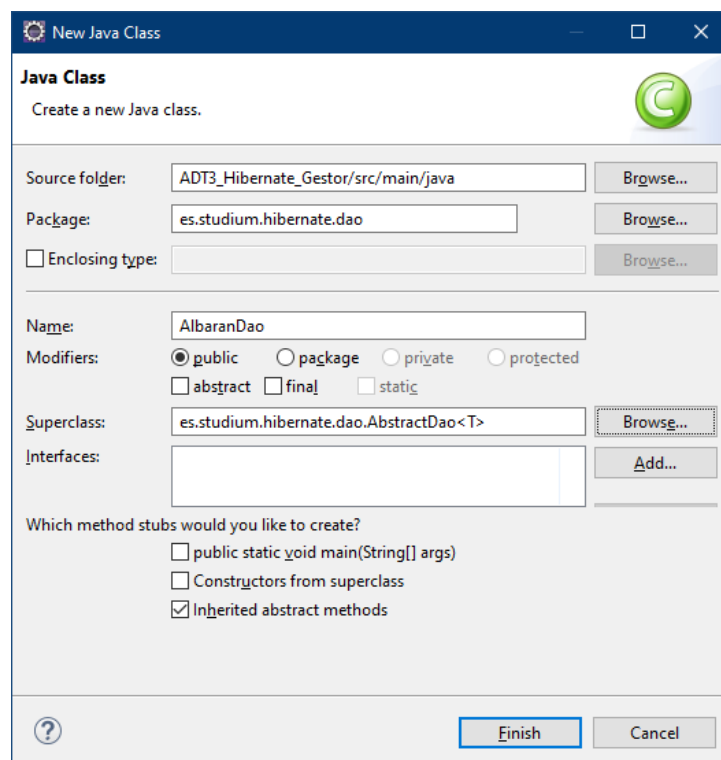


Esta sería nuestra tabla albaran, después de añadir la relación 1:N entre Pedido y Albaran.

9.6.1 El DAO de Albaran

Ahora tenemos que crear la **clase AlbaranDao** en el paquete **es.studium.hibernate.dao** para poder realizar operaciones con esta entidad.

Esta clase debe heredar de la clase AbstractDao.



Debemos tener en cuenta, que en este caso <T> será <Albaran>.

Clase AlbaranDao

```
package es.studium.hibernate.dao;

import es.studium.hibernate.Albaran;

public class AlbaranDao extends AbstractDao<Albaran> {
    public AlbaranDao() {
        setClazz(Albaran.class);
    }
}
```

Solo con esta implementación de la clase AlbaranDao, podemos desde la clase Principal, crear albaranes, modificarlos y eliminarlos.

9.7 Entidad Factura. Relaciones 1:1

Añadiremos la **entidad Factura** a nuestro programa en el paquete **es.studium.hibernate**.

Clase Factura

- Atributos privados:
 - id (tipo int)
 - numero (tipo String)
- Métodos constructores
 - Constructor por defecto
 - Constructor con el parámetro referencia
- Métodos inspectores:
 - Getters
 - Setters
- Método toString

El **id** será autogenerado y numérico.

El **numero** de factura será un String al que agregaremos el prefijo **FAC-**.

Clase Factura

```
package es.studium.hibernate;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name = "factura")
public class Factura {

    private static final String PREFIJO = "FAC-";

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String numero;

    public Factura() {
    }
}
```

```
public Factura(Pedido pedido) {
    this.numero = PREFIJO + pedido.getReferencia();
}

public int getId() {
    return id;
}

public void setId(int id) {
    this.id = id;
}

public String getNumero() {
    return numero;
}

public void setNumero(String numero) {
    this.numero = numero;
}

@Override
public String toString() {
    return "Factura [id=" + id + ", numero=" + numero +
    "]\n";
}
```

Ejecutamos la clase Principal y observamos en MySql Workbench que se ha creado la tabla factura con los dos campos: id y numero.

Pero Facturas y Pedidos está relacionados. Luego debemos tenerlo en cuenta en las entidades Pedido y Factura. La relación que existe entre ellas es 1:1, porque un pedido tiene una única factura y una factura es de un pedido.

Por lo tanto, en la clase Pedido tenemos que añadir un atributo Factura:

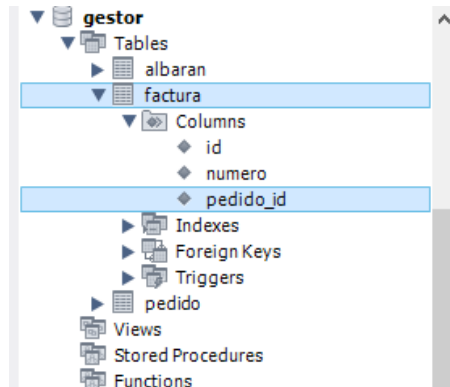
```
@OneToOne(mappedBy = "pedido")
private Factura factura;
```

En este caso, la anotación que utilizamos es @OneToOne mapeada con el nombre del atributo que haga referencia a la factura en la clase Pedido y que llamaremos `pedido`.

En la clase Factura, añadimos un atributo Pedido al que tenemos que llamar `pedido`:

```
@OneToOne
@JoinColumn
private Pedido pedido;
```

En este caso, en la anotación `@OneToOne` en lugar de ponerle información de mapeo le añadimos una segunda anotación `@JoinColumn` que indica que el vínculo se hará por esta columna. Es decir, la nueva columna **pedido_id** debe aparecer en la **tabla factura** de la base de datos.



Nuestras clases Pedido y Factura, quedan configuradas de la forma siguiente:

Clase Pedido

```
package es.studium.hibernate;

import java.time.LocalDateTime;
import java.util.ArrayList;
import java.util.List;
import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToMany;
import javax.persistence.OneToOne;
import javax.persistence.Table;

@Entity
@Table(name = "pedido")
public class Pedido {

    @Column(name = "id")
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

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

    @Column(name = "fecha")
```

```
private LocalDateTime fecha;

@OneToMany(mappedBy = "pedido", cascade =
CascadeType.ALL)
private List<Albaran> albaranes = new
ArrayList<Albaran>();

@OneToOne(mappedBy = "pedido")
private Factura factura;

public Pedido() {
    id = 0;
    referencia = "";
    fecha = LocalDateTime.now();
}

public Pedido(String referencia, LocalDateTime fecha) {
    this.referencia = referencia;
    this.fecha = fecha;
}

public int getId() {
    return id;
}

public void setId(int id) {
    this.id = id;
}

public String getReferencia() {
    return referencia;
}

public void setReferencia(String referencia) {
    this.referencia = referencia;
}

public LocalDateTime getFecha() {
    return fecha;
}

public void setFecha(LocalDateTime fecha) {
    this.fecha = fecha;
}

public List<Albaran> getAlbaranes() {
    return albaranes;
}

public void setAlbaranes(List<Albaran> albaranes) {
    this.albaranes = albaranes;
}

public Factura getFactura() {
```

```
        return factura;
    }

    public void setFactura(Factura factura) {
        this.factura = factura;
    }

    @Override
    public String toString() {
        return "Pedido [id=" + id + ", referencia=" +
referencia + ", fecha=" + fecha + "]";
    }
}
```

Clase Factura

```
package es.studium.hibernate;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.OneToOne;
import javax.persistence.Table;

@Entity
@Table(name = "factura")
public class Factura {

    private static final String PREFIJO = "FAC-";

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String numero;

    @OneToOne
    @JoinColumn
    private Pedido pedido;

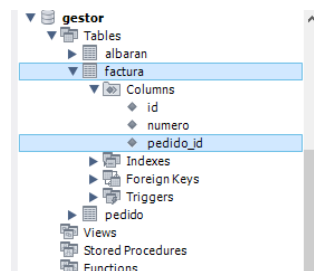
    public Factura() {
    }

    public Factura(Pedido pedido) {
        this.numero = PREFIJO + pedido.getReferencia();
        this.pedido = pedido;
    }
}
```



```
public int getId() {  
    return id;  
}  
public void setId(int id) {  
    this.id = id;  
}  
public String getNumero() {  
    return numero;  
}  
public void setNumero(String numero) {  
    this.numero = numero;  
}  
  
public Pedido getPedido() {  
    return pedido;  
}  
  
public void setPedido(Pedido pedido) {  
    this.pedido = pedido;  
}  
  
@Override  
public String toString() {  
    return "Factura [id=" + id + ", numero=" + numero +  
    "];"  
}  
}
```

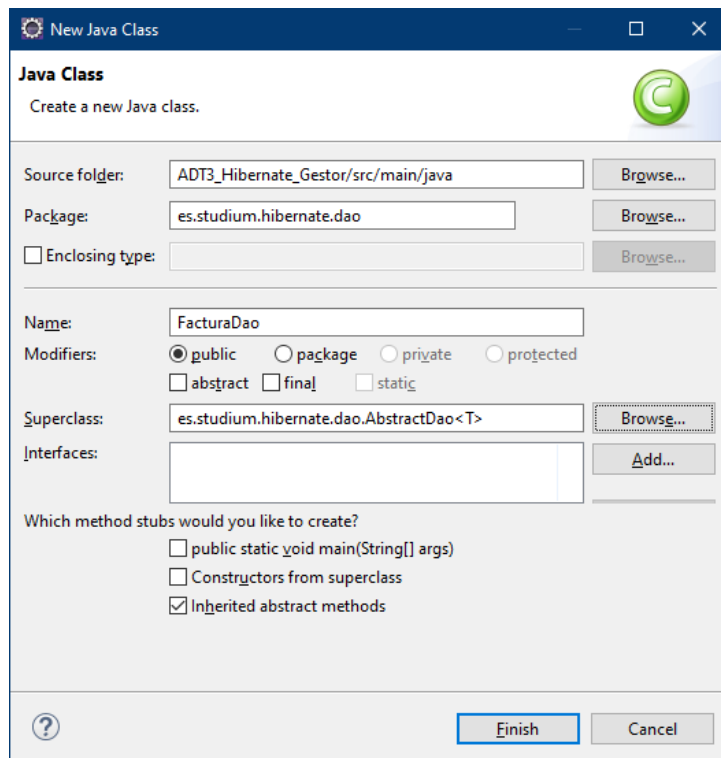
Ejecutamos nuestra clase principal y vemos qué en la tabla factura ha aparecido la columna pedido_id.



9.7.1 El DAO de Factura

Ahora tenemos que crear la **clase FacturaDao** en el paquete **es.studium.hibernate.dao** para poder realizar operaciones con esta entidad.

Esta clase debe heredar de la clase AbstractDao.



Debemos tener en cuenta, que en este caso <T> será <Factura>.

Clase FacturaDao

```
package es.studium.hibernate.dao;

import es.studium.hibernate.Factura;

public class FacturaDao extends AbstractDao<Factura> {
    public FacturaDao() {
        setClazz(Factura.class);
    }
}
```

Solo con esta implementación de la clase FacturaDao, podemos desde la clase Principal, crear facturas, modificarlas y eliminarlas.

9.8 Relaciones M:N

Tenemos las entidades Pedido, Albaran y Factura. Nos falta crear la entidad Producto.

La relación que existe entre pedidos y productos es M:N ya que en un pedido podemos tener muchos productos y un producto puede estar en muchos pedidos.

Cuando tenemos una relación M:N la solución es generar tabla. Es decir, a parte de la tabla pedido y de la tabla producto, que todavía tenemos que generar, generaremos una tabla productos_pedido en la que pondremos el id del pedido y el id del producto.

9.9 Entidad Producto

Añadiremos la **entidad Producto** a nuestro programa en el paquete **es.studium.hibernate**.

Clase Producto

- Atributos privados:
 - id (tipo int)
 - referencia (tipo String)
 - descripcion (tipo String)
- Métodos constructores
 - Constructor por defecto
 - Constructor con el parámetro referencia
- Métodos inspectores:
 - Getters
 - Setters
- Método toString

El **id** será autonumérico.

La **referencia** del producto será un String con la referencia interna del producto y una descripción del producto.

Nuestra clase Producto con las anotaciones incluidas, queda de la forma siguiente.

Clase Producto

```
package es.studium.hibernate;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
```

```
@Table (name = "producto")
public class Producto {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    private String referencia;

    private String descripcion;

    public Producto() {
    }

    public Producto(String referencia, String descripcion) {
        this.referencia = referencia;
        this.descripcion = descripcion;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getReferencia() {
        return referencia;
    }

    public void setReferencia(String referencia) {
        this.referencia = referencia;
    }

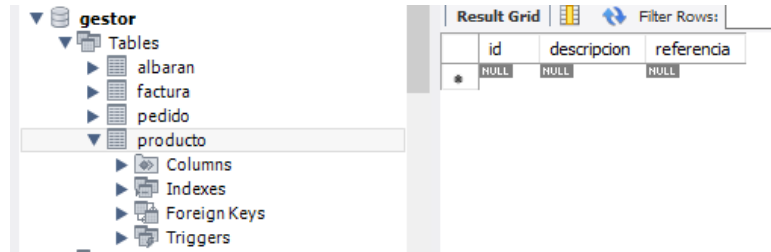
    public String getDescripcion() {
        return descripcion;
    }

    public void setDescripcion(String descripcion) {
        this.descripcion = descripcion;
    }

    @Override
    public String toString() {
        return "Producto [id=" + id + ", referencia=" +
referencia + ", descripcion=" + descripcion + "];"
    }
}
```

```
}
```

Ejecutamos la clase Principal y observamos que se ha creado la tabla Producto con las columnas id, referencia y descripción.



Nos falta incluir la relación M:N.

- En la **entidad Pedido** necesitamos un conjunto de productos:

```
@ManyToMany(mappedBy = "pedidos", cascade = CascadeType.ALL)
private Set<Producto> productos = new HashSet<Producto>();
```

```
public Set<Producto> getProductos() {
    return productos;
}
```

```
public void setProductos(Set<Producto> productos) {
    this.productos = productos;
}
```

- Y en la **entidad Producto**, necesitamos un conjunto de pedidos:

```
@ManyToMany
private Set<Pedido> pedidos = new HashSet<Pedido>();
```

```
public Set<Pedido> getPedidos() {
    return pedidos;
}
```

```
public void setPedidos(Set<Pedido> pedidos) {
    this.pedidos = pedidos;
}
```

Nota: También podríamos haber utilizado **List** en lugar de **Set**, pero vamos a utilizar Set por cambiar.

Nuestras clases Pedido y Producto quedan configuradas de la forma siguiente, tras incluir la relación M:N.

Clase Pedido

```
package es.studium.hibernate;

import java.time.LocalDateTime;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;
import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.ManyToMany;
import javax.persistence.OneToOne;
import javax.persistence.Table;

@Entity
@Table(name = "pedido")
public class Pedido {

    @Column(name = "id")
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

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

    @Column(name = "fecha")
    private LocalDateTime fecha;

    @OneToMany(mappedBy = "pedido", cascade =
CascadeType.ALL)
    private List<Albaran> albaranes = new
ArrayList<Albaran>();

    @OneToOne(mappedBy = "pedido")
    private Factura factura;

    @ManyToMany(mappedBy = "pedidos", cascade =
CascadeType.ALL)
    private Set<Producto> productos = new
HashSet<Producto>();

    public Pedido() {
        id = 0;
        referencia = "";
    }
}
```

```
        fecha = LocalDateTime.now();
    }

    public Pedido(String referencia, LocalDateTime fecha) {
        this.referencia = referencia;
        this.fecha = fecha;
    }

    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getReferencia() {
        return referencia;
    }
    public void setReferencia(String referencia) {
        this.referencia = referencia;
    }
    public LocalDateTime getFecha() {
        return fecha;
    }
    public void setFecha(LocalDateTime fecha) {
        this.fecha = fecha;
    }

    public List<Albaran> getAlbaranes() {
        return albaranes;
    }

    public void setAlbaranes(List<Albaran> albaranes) {
        this.albaranes = albaranes;
    }

    public Factura getFactura() {
        return factura;
    }

    public void setFactura(Factura factura) {
        this.factura = factura;
    }

    public Set<Producto> getProductos() {
        return productos;
    }

    public void setProductos(Set<Producto> productos) {
        this.productos = productos;
    }
}
```

```
    }

    @Override
    public String toString() {
        return "Pedido [id=" + id + ", referencia=" +
referencia + ", fecha=" + fecha + "]";
    }
}
```

Clase Producto

```
package es.studium.hibernate;

import java.util.Set;
import java.util.HashSet;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.ManyToMany;
import javax.persistence.Table;

@Entity
@Table (name = "producto")
public class Producto {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    private String referencia;

    private String descripcion;

    @ManyToMany
    private Set<Pedido> pedidos = new HashSet<Pedido>();

    public Producto() {
    }

    public Producto(String referencia, String descripcion) {
        this.referencia = referencia;
        this.descripcion = descripcion;
    }

    public int getId() {
        return id;
    }
}
```



```
public void setId(int id) {
    this.id = id;
}

public String getReferencia() {
    return referencia;
}

public void setReferencia(String referencia) {
    this.referencia = referencia;
}

public String getDescripcion() {
    return descripcion;
}

public void setDescripcion(String descripcion) {
    this.descripcion = descripcion;
}

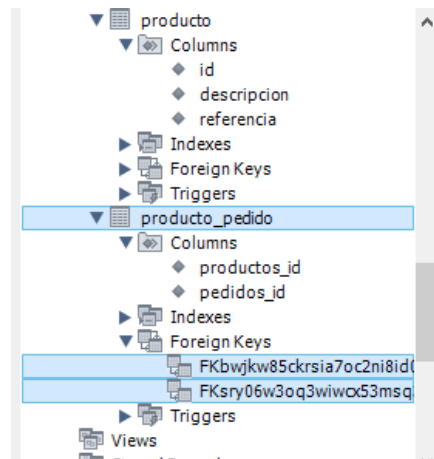
public Set<Pedido> getPedidos() {
    return pedidos;
}

public void setPedidos(Set<Pedido> pedidos) {
    this.pedidos = pedidos;
}

@Override
public String toString() {
    return "Producto [id=" + id + ", referencia=" +
referencia + ", descripcion=" + descripcion + "]";
}
}
```

Ejecutamos nuestra clase Principal y observamos que ha aparecido la **tabla producto_pedido**, porque hemos puesto el `mappedBy = "pedidos"` en la clase Pedido. Si lo hubiésemos puesto en la entidad Producto, la tabla se habría llamado `pedido_producto`.

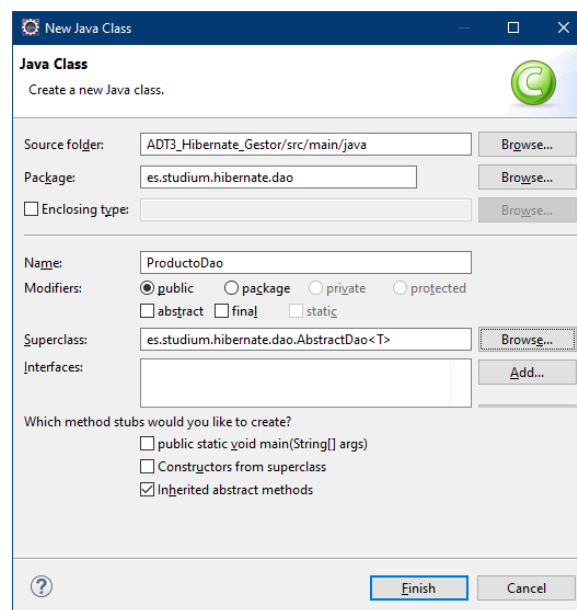
Esta nueva tabla, tiene dos foreign key, que referencian a las columnas id, una de la tabla pedido y la otra de la tabla producto.



9.9.1 El DAO de Producto

Ahora tenemos que crear la **clase ProductoDao** en el paquete **es.studium.hibernate.dao** para poder realizar operaciones con esta entidad.

Esta clase debe heredar de la clase AbstractDao.



Debemos tener en cuenta, que en este caso <T> será <Producto>.

Clase ProductoDao

```
package es.studium.hibernate.dao;

import es.studium.hibernate.Producto;

public class ProductoDao extends AbstractDao<Producto> {
    public ProductoDao() {
        setClazz(Producto.class);
    }
}
```

```
}  
}
```

Solo con esta implementación de la clase ProductoDao, podemos desde la clase Principal, crear facturas, modificarlas y eliminarlas.

Para la nueva tabla creada producto_pedido, no es necesario crear una clase DAO.

9.10 Gestor de Pedidos

Creamos una nueva **clase GestorPedidos**, dentro del **paquete es.studium.hibernate**, con método main para hacer nuevas pruebas con nuestro proyecto.

Crearemos también nuevos métodos en la **clase Pedido** para poder añadir productos uno a uno, generar albarán y generar factura.

- Añadir productos uno a uno.

```
public void addProducto(Producto producto) {  
    productos.add(producto);  
}
```

- Generar albarán.

```
public Albaran generaAlbaran() {  
    Albaran albaran = new Albaran();  
    albaranes.add(albaran);  
    return albaran;  
}
```

- Generar factura.

```
public Factura generaFactura() {  
    factura = new Factura(this);  
    return factura;  
}
```

Nuestra clase Pedido queda implementada de la forma siguiente.

Clase Pedido

```
package es.studium.hibernate;  
  
import java.time.LocalDateTime;  
import java.util.ArrayList;  
import java.util.List;
```

```
import java.util.Set;
import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.ManyToMany;
import javax.persistence.OneToOne;
import javax.persistence.Table;

@Entity
@Table(name = "pedido")
public class Pedido {

    @Column(name = "id")
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

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

    @Column(name = "fecha")
    private LocalDateTime fecha;

    @OneToMany(mappedBy = "pedido", cascade =
CascadeType.ALL)
    private List<Albaran> albaranes = new
ArrayList<Albaran>();

    @OneToOne(mappedBy = "pedido")
    private Factura factura;

    @ManyToMany(mappedBy = "pedidos", cascade =
CascadeType.ALL)
    private Set<Producto> productos;

    public Pedido() {
        id = 0;
        referencia = "";
        fecha = LocalDateTime.now();
    }
    public Pedido(String referencia, LocalDateTime fecha) {
        this.referencia = referencia;
        this.fecha = fecha;
    }
}
```

```
public int getId() {
    return id;
}
public void setId(int id) {
    this.id = id;
}
public String getReferencia() {
    return referencia;
}
public void setReferencia(String referencia) {
    this.referencia = referencia;
}
public LocalDateTime getFecha() {
    return fecha;
}
public void setFecha(LocalDateTime fecha) {
    this.fecha = fecha;
}

public List<Albaran> getAlbaranes() {
    return albaranes;
}
public void setAlbaranes(List<Albaran> albaranes) {
    this.albaranes = albaranes;
}
public Albaran generaAlbaran() {
    Albaran albaran = new Albaran(referencia);
    albaranes.add(albaran);
    return albaran;
}

public Factura getFactura() {
    return factura;
}
public void setFactura(Factura factura) {
    this.factura = factura;
}
public Factura generaFactura() {
    factura = new Factura(this);
    return factura;
}

public Set<Producto> getProductos() {
    return productos;
}
public void setProductos(Set<Producto> productos) {
    this.productos = productos;
}
public void addProducto(Producto producto) {
```

```
        productos.add(producto);  
    }  
  
    @Override  
    public String toString() {  
        return "Pedido [id=" + id + ", referencia=" +  
referencia + ", fecha=" + fecha + "];"  
    }  
}
```

En el main de la **clase GestorPedidos**, hacemos las siguientes pruebas:

Clase GestorPedidos

```
package es.studium.hibernate;  
  
import java.time.LocalDateTime;  
import es.studium.hibernate.dao.PedidoDao;  
  
public class GestorPedidos {  
  
    public static void main(String[] args) {  
        /*Creamos un objeto de tipo PedidoDao*/  
        PedidoDao pedidoDao = new PedidoDao();  
  
        /*Creamos tres objetos de tipo Producto*/  
        Producto libro = new Producto("libJava", "Manual  
Imprescindible Java");  
        Producto cuaderno = new Producto("cuaRojo",  
"Cuaderno rojo");  
        Producto lapiz = new Producto("lapHB", "Lápiz HB");  
  
        /*Creamos un obeto de tipo Pedido y le añadimos los  
Productos*/  
        Pedido vueltaAlCole = new Pedido("153947",  
LocalDateTime.now());  
        vueltaAlCole.addProducto(libro);  
        vueltaAlCole.addProducto(cuaderno);  
        vueltaAlCole.addProducto(lapiz);  
  
        /*Guardamos el Pedido*/  
        pedidoDao.save(vueltaAlCole);  
  
        /*Generamos un Albaran y una Factura*/  
        Albaran albaran = vueltaAlCole.generaAlbaran();  
        Factura factura = vueltaAlCole.generaFactura();  
  
        /*Mostramos los Pedidos*/  
        System.out.println("Pedido:\n" + vueltaAlCole);  
    }  
}
```

```
}  
}
```

Modificamos los métodos toString() de todas las entidades para que nos muestre las salidas deseadas:

Clase Pedido

```
@Override  
public String toString() {  
    return "Pedido [id=" + id + ", referencia=" + referencia  
+ ", fecha=" + fecha + ", albaranes=" + albaranes + ",  
factura=" + factura + ", productos=" + productos + "];"  
}
```

Clase Albaran

```
@Override  
public String toString() {  
    return "Albaran [id=" + id + ", referencia=" + referencia  
+ ", fechaEmision=" + fechaEmision + ", fechaRecepcion=" +  
fechaRecepcion + "];"  
}
```

Clase Factura

```
@Override  
public String toString() {  
    return "Factura [id=" + id + ", numero=" + numero + "];"  
}
```

Clase Producto

```
@Override  
public String toString() {  
    return "Producto [id=" + id + ", referencia=" +  
referencia + ", descripcion=" + descripcion + "];"  
}
```

Hechas estas modificaciones, **ejecutamos** la clase **GestoPerdidos** y la salida debe ser similar a la que mostramos a continuación:

```
Pedido:  
Pedido      [id=230,      referencia=153947,      fecha=2021-09-  
06T12:09:21.672,  albaranes=[Albaran  [id=0,      referencia=ALB-
```

```
153947,          fechaEmision=2021-09-06T12:09:21.803,
fechaRecepcion=null]],  factura=Factura  [id=0,  numero=FAC-
153947],  productos=[Producto  [id=55,  referencia=cuaRojo10,
descripcion=Cuaderno  rojo],          Producto  [id=56,
referencia=libJava10, descripcion=Manual Imprescindible Java],
Producto  [id=57, referencia=lapHB10, descripcion=Lápiz HB]]]
```

Pedido tiene id porque ha sido guardado, pero albaran y factura cuando los estamos imprimiendo, aún no lo tienen.

Vemos que gracias al atributo `cascade = CascadeType.ALL` que hemos indicado en las anotaciones de las relaciones `@OneToOne` , `@OneToMany` y `@ManyToMany` conseguimos que al guardar el pedido también se guarden en cascada todos sus elementos.

Clase Pedido

```
@OneToMany(mappedBy = "pedido", cascade = CascadeType.ALL)
private List<Albaran> albaranes = new ArrayList<Albaran>();

@OneToOne(mappedBy = "pedido", cascade = CascadeType.ALL)
private Factura factura;

@ManyToMany(mappedBy = "pedidos", cascade = CascadeType.ALL)
private Set<Producto> productos = new HashSet<Producto>();
```

Clase Factura

```
@OneToOne (cascade = CascadeType.ALL)
@JoinColumn
private Pedido pedido;
```

Añadimos en la clase `GestorPedidos`, las siguientes líneas de código para guardar los cambios en el pedido:

```
pedidoDao.update(vueltaAlCole);
System.out.println("Pedido actualizado:\n" + vueltaAlCole);
```

Volvemos a ejecutar nuestra **clase GestorPedidos**:

```
Pedido actualizado:
Pedido      [id=240,      referencia=153947,      fecha=2021-09-
06T13:17:11.837,  albaranes=[Albaran  [id=64,  referencia=ALB-
153947,
                        fechaEmision=2021-09-06T13:17:11.927,
fechaRecepcion=null]],  factura=Factura  [id=15,  numero=FAC-
153947],  productos=[Producto  [id=85,  referencia=cuaRojo10,
descripcion=Cuaderno  rojo],          Producto  [id=86,
```



```
referencia=libJava10, descripcion=Manual Imprescindible Java],  
Producto [id=87, referencia=lapHB10, descripcion=Lápiz HB]]]
```

Vemos ya que todas las tablas tienen los valores de los id ya que hemos hecho efectiva su actualización en la base de datos con el método `update`.

9.11 Relaciones M:N bidireccionales

Para rellenar la tabla `producto_pedido` debemos implementar que se puedan añadir pedidos a los productos, se añade el producto automáticamente, y que se puedan añadir productos a los pedidos, en este caso se añade el pedido al producto.

Vemos el código que tenemos que añadir en las entidades `Pedido` y `Producto`.

Clase Pedido modificamos el método `addProducto` de la forma siguiente:

```
public void addProducto(Producto producto) {  
    productos.add(producto);  
  
    if (!producto.getPedidos().contains(this)) {  
        producto.addPedido(this);  
    }  
}
```

Clase Producto modificamos el método `addPedido` de la forma siguiente:

```
public void addPedido(Pedido pedido) {  
    pedidos.add(pedido);  
  
    if (!pedido.getProductos().contains(this)) {  
        pedido.addProducto(this);  
    }  
}
```

Para probar correctamente si funciona nuestra implementación, debemos eliminar los métodos `setProductos` y `setPedidos` y asegurarnos de que las colecciones están inicializadas:

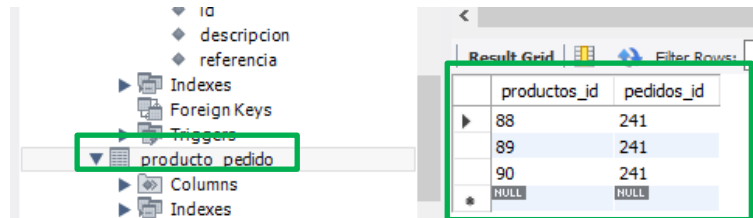
Clase Pedido

```
@ManyToMany(mappedBy = "pedidos", cascade = CascadeType.ALL)  
private Set<Producto> productos = new HashSet<Producto>();
```

Clase Producto

```
@ManyToMany
private Set<Pedido> pedidos = new HashSet<Pedido>();
```

Ejecutamos la **clase GestorPedidos** y comprobamos que la tabla **producto_pedido** ya contiene valores:



10. CLASES DE NUESTRO PROYECTO

A continuación indicamos cómo queda el código de las clases que hemos utilizado en nuestro proyecto Maven.

10.1 Entidades

Clase Albaran

```
package es.studium.hibernate;

import java.time.LocalDateTime;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.ManyToOne;
import javax.persistence.Table;

@Entity
@Table(name = "albaran")
public class Albaran {

    private static final String PREFIJO = "ALB-";

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
private int id;
private String referencia;
private LocalDateTime fechaEmision;
private LocalDateTime fechaRecepcion;

@ManyToOne(fetch = FetchType.LAZY)
private Pedido pedido;

public Albaran() {
}
public Albaran(String refPedido) {
    referencia = PREFIJO + refPedido;
    fechaEmision = LocalDateTime.now();
}

public int getId() {
    return id;
}
public void setId(int id) {
    this.id = id;
}

public String getReferencia() {
    return referencia;
}
public void setReferencia(String referencia) {
    this.referencia = referencia;
}

public LocalDateTime getFechaEmision() {
    return fechaEmision;
}
public void setFechaEmision(LocalDateTime fechaEmision) {
    this.fechaEmision = fechaEmision;
}

public LocalDateTime getFechaRecepcion() {
    return fechaRecepcion;
}
public void setFechaRecepcion(LocalDateTime
fechaRecepcion) {
    this.fechaRecepcion = fechaRecepcion;
}

public Pedido getPedido() {
    return pedido;
}
public void setPedido(Pedido pedido) {
    this.pedido = pedido;
}
```

```
    }

    @Override
    public String toString() {
        return "Albaran [id=" + id + ", referencia=" +
referencia + ", fechaEmision=" + fechaEmision
        + ", fechaRecepcion=" + fechaRecepcion +
        "]" ;
    }
}
```

Clase Factura

```
package es.studium.hibernate;

import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.OneToOne;
import javax.persistence.Table;

@Entity
@Table(name = "factura")
public class Factura {

    private static final String PREFIJO = "FAC-";

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String numero;

    @OneToOne (cascade = CascadeType.ALL)
    @JoinColumn
    private Pedido pedido;

    public Factura() {
    }

    public Factura(Pedido pedido) {
        this.numero = PREFIJO + pedido.getReferencia();
        this.pedido = pedido;
    }

    public int getId() {
        return id;
    }
}
```

```
public void setId(int id) {
    this.id = id;
}

public String getNumero() {
    return numero;
}

public void setNumero(String numero) {
    this.numero = numero;
}

public Pedido getPedido() {
    return pedido;
}

public void setPedido(Pedido pedido) {
    this.pedido = pedido;
}

@Override
public String toString() {
    return "Factura [id=" + id + ", numero=" + numero +
    "]" ;
}
}
```

Clase Pedido

```
package es.studium.hibernate;

import java.time.LocalDateTime;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;
import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.ManyToMany;
import javax.persistence.OneToOne;
import javax.persistence.Table;

@Entity
@Table(name = "pedido")
public class Pedido {
```

```
@Column(name = "id")
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private int id;

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

@Column(name = "fecha")
private LocalDateTime fecha;

@OneToMany(mappedBy = "pedido", cascade =
CascadeType.ALL)
private List<Albaran> albaranes = new
ArrayList<Albaran>();

@OneToOne(mappedBy = "pedido", cascade = CascadeType.ALL)
private Factura factura;

@ManyToMany(mappedBy = "pedidos", cascade =
CascadeType.ALL)
private Set<Producto> productos = new
HashSet<Producto>();

public Pedido() {
    id = 0;
    referencia = "";
    fecha = LocalDateTime.now();
}
public Pedido(String referencia, LocalDateTime fecha) {
    this.referencia = referencia;
    this.fecha = fecha;
}

public int getId() {
    return id;
}
public void setId(int id) {
    this.id = id;
}

public String getReferencia() {
    return referencia;
}
public void setReferencia(String referencia) {
    this.referencia = referencia;
}

public LocalDateTime getFecha() {
```

```
        return fecha;
    }
    public void setFecha(LocalDateTime fecha) {
        this.fecha = fecha;
    }

    public List<Albaran> getAlbaranes() {
        return albaranes;
    }
    public void setAlbaranes(List<Albaran> albaranes) {
        this.albaranes = albaranes;
    }
    public Albaran generaAlbaran() {
        Albaran albaran = new Albaran(referencia);
        albaranes.add(albaran);
        return albaran;
    }

    public Factura getFactura() {
        return factura;
    }
    public void setFactura(Factura factura) {
        this.factura = factura;
    }
    public Factura generaFactura() {
        factura = new Factura(this);
        return factura;
    }

    public Set<Producto> getProductos() {
        return productos;
    }
    public void addProducto(Producto producto) {
        productos.add(producto);
        if (!producto.getPedidos().contains(this)) {
            producto.addPedido(this);
        }
    }

    @Override
    public String toString() {
        return "Pedido [id=" + id + ", referencia=" +
referencia + ", fecha=" + fecha + ", albaranes=" + albaranes
            + ", factura=" + factura + ",
productos=" + productos + "];"
    }
}
```

Clase Producto

```
package es.studium.hibernate;

import java.util.HashSet;
import java.util.Set;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.ManyToMany;
import javax.persistence.Table;

@Entity
@Table (name = "producto")
public class Producto {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String referencia;
    private String descripcion;

    @ManyToMany
    private Set<Pedido> pedidos = new HashSet<Pedido>();

    public Producto() {
    }
    public Producto(String referencia, String descripcion) {
        this.referencia = referencia;
        this.descripcion = descripcion;
    }

    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }

    public String getReferencia() {
        return referencia;
    }
    public void setReferencia(String referencia) {
        this.referencia = referencia;
    }

    public String getDescripcion() {
        return descripcion;
    }
}
```



```
}  
public void setDescripcion(String descripcion) {  
    this.descripcion = descripcion;  
}  
  
public Set<Pedido> getPedidos() {  
    return pedidos;  
}  
public void addPedido(Pedido pedido) {  
    pedidos.add(pedido);  
    if (!pedido.getProductos().contains(this)) {  
        pedido.addProducto(this);  
    }  
}  
  
@Override  
public String toString() {  
    return "Producto [id=" + id + ", referencia=" +  
referencia + ", descripcion=" + descripcion + "];"  
}  
}
```

10.2 Clases de prueba (main)

Son las dos clases que tienen **método main**, con las que hemos estado probando nuestro programa. Son las **clases Principal** y **GestorPedidos**.

Clase Principal

```
package es.studium.hibernate;  
  
import java.time.LocalDateTime;  
import java.time.temporal.ChronoUnit;  
import java.util.List;  
import javax.persistence.NoResultException;  
  
import es.studium.hibernate.dao.AlbaranDao;  
import es.studium.hibernate.dao.FacturaDao;  
import es.studium.hibernate.dao.PedidoDao;  
import es.studium.hibernate.dao.ProductoDao;  
  
public class Principal {  
  
    public static void main(String[] args) {  
        /*OPERACIONES CON PEDIDOS*/  
        /*Creamos Pedidos*/  
        PedidoDao pedidoDao = new PedidoDao();  
  
        Pedido pedido = new Pedido();
```

```
pedido.setFecha(LocalDateTime.now());
pedido.setReferencia("001");
pedidoDao.save(pedido);

List<Pedido> pedidos = pedidoDao.getAll();
System.out.println("Lista de pedidos: " + pedidos);

/*Pedido creado con la fecha actual*/
Pedido pedido2 = new Pedido("001",
LocalDateTime.now());
pedidoDao.save(pedido2);

/*Pedido creado con fecha de pasado mañana, 2 días
más de la fecha actual*/
Pedido pedido3 = new Pedido("pedidoFuturo",
LocalDateTime.now().plus(2, ChronoUnit.DAYS));
pedidoDao.save(pedido3);
try {
/*Llamamos al método pedidoMasReciente() e
imprimimos el resultado*/
Pedido masReciente = pedidoDao.pedidoMasReciente();
System.out.println("Pedido más reciente: " +
masReciente);
} catch (NoResultException e) {
System.out.println("No hay pedidos
recientes.");
}

/*Creamos un pedido de hace una semana.*/
Pedido pedido4 = new Pedido("pedPas",
LocalDateTime.now().minus(1, ChronoUnit.WEEKS));
pedidoDao.save(pedido4);

/*Recuperamos la lista de pedidos de la semana
pasada
* y lo imprimimos.*/
List<Pedido> pedidosSemanaPasada =
pedidoDao.pedidosSemanaPasada();
System.out.println("*** Pedidos de la semana pasada:
" + pedidosSemanaPasada);
}
}
```

```
package es.studium.hibernate;

import java.time.LocalDateTime;
import es.studium.hibernate.dao.PedidoDao;

public class GestorPedidos {

    public static void main(String[] args) {
        /*Creamos un objeto de tipo PedidoDao*/
        PedidoDao pedidoDao = new PedidoDao();

        /*Creamos tres objetos de tipo Producto*/
        Producto libro = new Producto("libJava10", "Manual
Imprescindible Java");
        Producto cuaderno = new Producto("cuaRojo10",
"Cuaderno rojo");
        Producto lapiz = new Producto("lapHB10", "Lápiz
HB");

        /*Creamos un obeto de tipo Pedido y le añadimos los
Productos*/
        Pedido vueltaAlCole = new Pedido("153947",
LocalDateTime.now());
        vueltaAlCole.addProducto(libro);
        vueltaAlCole.addProducto(cuaderno);
        vueltaAlCole.addProducto(lapiz);

        /*Guardamos el Pedido*/
        pedidoDao.save(vueltaAlCole);

        /*Generamos un Albaran y una Factura*/
        Albaran albaran = vueltaAlCole.generaAlbaran();
        Factura factura = vueltaAlCole.generaFactura();

        /*Mostramos los Pedidos*/
        System.out.println("Pedido:\n" + vueltaAlCole);

        /*Actualizamos el Pedido y lo mostramos
actualizado.*/
        pedidoDao.update(vueltaAlCole);
        System.out.println("Pedido actualizado:\n" +
vueltaAlCole);
    }
}
```

10.3 Recursos DAO

Clases e interfaces utilizadas para los objetos de acceso a datos.

Interfaz Dao

```
package es.studium.hibernate.dao;

import java.util.List;
import java.util.Optional;

public interface Dao<T> {
    Optional<T> get(long id);
    List<T> getAll();
    void save(T t);
    void update(T t);
    void delete(T t);
}
```

Clase AbstractDao

```
package es.studium.hibernate.dao;

import java.util.List;
import java.util.Optional;
import java.util.function.Consumer;
import javax.persistence.EntityManager;
import javax.persistence.EntityTransaction;
import javax.persistence.Query;
import es.studium.hibernate.utiles.EntityManagerUtil;

public abstract class AbstractDao<T> implements Dao<T> {

    private EntityManager entityManager =
EntityManagerUtil.getEntityManager();
    private Class<T> clazz;

    public EntityManager getEntityManager() {
        return entityManager;
    }
    public void setEntityManager(EntityManager entityManager)
    {
        this.entityManager = entityManager;
    }

    public Class<T> getClazz() {
        return clazz;
    }
    public void setClazz(Class<T> clazz) {
        this.clazz = clazz;
    }
}
```

```
@Override
public Optional<T> get(long id) {
    return
Optional.ofNullable(entityManager.find(clazz, id));
}

@Override
public List<T> getAll() {
    String qlString = "FROM " + clazz.getName();
    Query query = entityManager.createQuery(qlString);
    return query.getResultList();
}

private void
executeInsideTransaction(Consumer<EntityManager> action) {
    EntityTransaction tx =
entityManager.getTransaction();
    try {
        tx.begin();
        action.accept(entityManager);
        tx.commit();
    } catch (RuntimeException e) {
        tx.rollback();
        throw e;
    }
}

@Override
public void save(T t) {
    executeInsideTransaction(entityManager ->
entityManager.persist(t));
}

@Override
public void update(T t) {
    executeInsideTransaction(entityManager ->
entityManager.merge(t));
}

@Override
public void delete(T t) {
    executeInsideTransaction(entityManager ->
entityManager.remove(t));
}
}
```

10.3.1 DAO de las Entidades

Clases DAO para poder realizar operaciones con las Entidades implementadas en el apartado 10.1.

Son cuatro clases PedidoDao, AlbaranDao, FacturaDao y ProductoDao.

Clase PedidoDao

```
package es.studium.hibernate.dao;

import java.time.DayOfWeek;
import java.time.LocalDate;
import java.util.List;
import javax.persistence.Query;
import es.studium.hibernate.Pedido;

public class PedidoDao extends AbstractDao<Pedido> {

    public PedidoDao() {
        setClazz(Pedido.class);
    }

    public Pedido pedidoMasReciente() {
        /*Query que devuelve el pedido más reciente*/
        String qlString = "FROM " + Pedido.class.getName()
+ " WHERE fecha < now() order by fecha desc";

        /*Query que no devuelve ningún pedido*/
        // String qlString = "FROM " + Pedido.class.getName()
+ " WHERE fecha < now() and id > 100 order by fecha desc";

        Query query =
getEntityManager().createQuery(qlString).setMaxResults(1);
        return (Pedido) query.getSingleResult();
    }

    public List<Pedido> pedidosSemanaPasada() {
        String qlString = "FROM " + Pedido.class.getName()
+ " WHERE fecha between ?1 and ?2";

        Query query =
getEntityManager().createQuery(qlString);

        LocalDate esteLunes = getEsteLunes();

        LocalDate lunesAnterior = esteLunes.minusWeeks(1);

        query.setParameter(1,
lunesAnterior.atStartOfDay());
        query.setParameter(2, esteLunes.atStartOfDay());
    }
}
```

```
        return query.getResultList();
    }
    private static LocalDate getEsteLunes() {
        /*Obtenemos la fecha actual del sistema*/
        LocalDate now = LocalDate.now();

        /*Obtiene el día de la semana*/
        DayOfWeek diaSemana = now.getDayOfWeek();

        /*Obtiene el día de la semana en número entero y le
        resta 1 y el método minusDays devuelve la fecha con el número
        de días restado. Ej.: diaSemana es jueves diaSemana.getValue()
        es 4, (diaSemana.getValue() - 1) es (4-1)=3, luego
        minusDays(diaSemana.getValue() - 1) = minusDays(3) nos
        devuelve la fecha del día correspondiente a 3 días anteriores
        al actual, es decir tres días antes al jueves, es el lunes.*/
        return now.minusDays(diaSemana.getValue() - 1);
    }
}
```

Clase AlbaranDao

```
package es.studium.hibernate.dao;

import es.studium.hibernate.Albaran;

public class AlbaranDao extends AbstractDao<Albaran> {
    public AlbaranDao() {
        setClazz(Albaran.class);
    }
}
```

Clase FacturaDao

```
package es.studium.hibernate.dao;

import es.studium.hibernate.Factura;

public class FacturaDao extends AbstractDao<Factura> {
    public FacturaDao() {
        setClazz(Factura.class);
    }
}
```

Clase ProductoDao

```
package es.studium.hibernate.dao;

import es.studium.hibernate.Producto;

public class ProductoDao extends AbstractDao<Producto> {
    public ProductoDao() {
        setClazz(Producto.class);
    }
}
```

10.3.2 Clase EntityManagerUtil

Clase de utilidad que nos va a permitir establecer y manejar la conexión con la base de datos.

Clase EntityManagerUtil

```
package es.studium.hibernate.utiles;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

public class EntityManagerUtil {

    public static EntityManager getEntityManager() {
        EntityManagerFactory factory =
        Persistence.createEntityManagerFactory("gestor");
        EntityManager manager =
        factory.createEntityManager();
        return manager;
    }

    public static void main(String[] args) {
        EntityManager manager =
        EntityManagerUtil.getEntityManager();
        System.out.println("EntityManager class ==> " +
        manager.getClass().getCanonicalName());
    }
}
```

10.3.3 Posibles Excepciones

Posibles excepciones al ejecutar nuestro programa.

Si ejecutamos nuestro programa, la clase Principal, y se produce la excepción:

javax.persistence.**PersistenceException**:org.hibernate.**PersistentObjectException**

la solucionaremos anulando la anotación

`@GeneratedValue(strategy = GenerationType.IDENTITY)`

del id de la entidad que esté dando esta excepción.