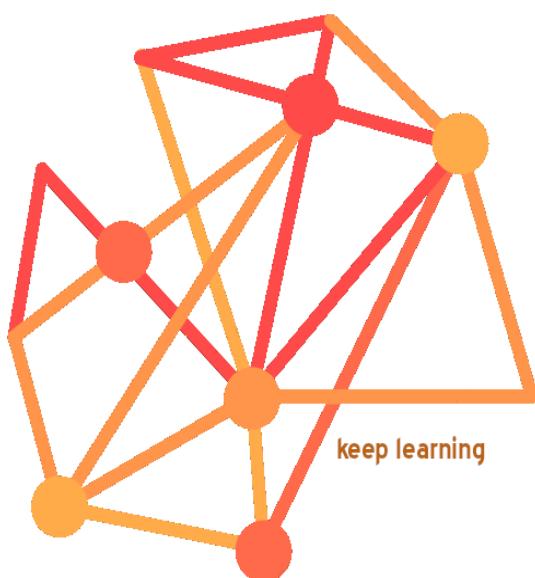


ORM con JAVA



Juan Carlos Pérez Rodríguez

Sumario

.....	2
Introducción.....	4
Aplicación que seguiremos para el aprendizaje.....	5
JPA.....	7
Definición teórica de Entidades (entities).....	8
¿ Usar anotaciones o xml ?.....	8
Funcionamiento básico y estructura de JPA.....	10
EntityManagerFactory.....	12
persistence.xml (alojado en src/main/resources/META-INF).....	14
Entidades (Entities).....	17
POJO vs BEAN.....	19
Anotaciones en las Entities.....	21
@ManyToMany.....	24
@Transient.....	26
@JsonBackReference,@JsonManagedReference.....	26
Procedimiento para generación Entities de forma automática.....	27
EntityManager.....	30
Responsabilidades del EntityManager.....	32
Operaciones del EntityManager.....	32
persist() (Guardando una nueva Entity).....	34
Actualizando una entidad (y uso de merge()).....	36
persist y CascadeType.PERSIST.....	41
detach().....	43
Borrar una entidad en la DDBB (remove()).....	43
Creación de objetos con relaciones.....	46
refresh().....	50
Gestión de excepciones en las transacciones y uso de rollback.....	52
Creación de consultas – createQuery().....	53
JPQL.....	55
JPQL subquery.....	57
JPQL join.....	57
named query.....	58
Native query.....	59
Capa Repositorio y relación con DAO.....	60
Anexo: Tests con maven surefly.....	62
Anexo: Paso a paso Proyecto maven-eclipse-jpa (ya sea web o no) con mysql.....	64
Caso proyecto no web, simple maven (sin archetype).....	64
Poner dependencias Maven para JPA (caso no web jdk1.8).....	68
Poner dependencias Maven para JPA (caso web jdk1.8).....	69

Actualizar proyecto con dependencias nuevas.....	71
Convertir proyecto en JPA.....	72
Anexo: Crear tablas en la DDBB habiendo creado las entities.....	85
Anexo: Hacer que Eclipse genere correctamente las anotaciones @ManyToMany.....	90
Anexo Eclipse crear aplicación jdk1.8.....	91
Anexo: Dependencias maven para un proyecto web jdk 11 con JPA, junto con plugin jetty.....	95

Introducción

El mapeo objeto-relacional (más conocido por su nombre en inglés, Object-Relational mapping, o sus siglas O/RM, **ORM**, y O/R mapping) es una técnica de programación para convertir datos entre el sistema de tipos utilizado en un lenguaje de programación orientado a objetos y la utilización de una base de datos relacional como motor de persistencia. En la práctica esto crea una base de datos orientada a objetos virtual, sobre la base de datos relacional. Esto posibilita el uso de las características propias de la orientación a objetos (básicamente herencia y polimorfismo). Hay paquetes comerciales y de uso libre disponibles que desarrollan el mapeo relacional de objetos

fuente: wikipedia

Es evidente que hay una dificultad para llevar un modelo orientado a objetos a un modelo relacional. Cuando programamos sin un ORM se invierte siempre tiempo y recursos para generar una capa de abstracción que nos permita llevar nuestros objetos a tablas relacionales. La idea es separar nuestro código de la persistencia en base de datos, de tal forma que sea fácil la transición de un SGBD a otro por ejemplo.

La impedancia para trasladar nuestros objetos a tablas relacionales ha motivado estas herramientas de mapeo. Son bastante usadas en diferentes lenguajes, si bien, tienen ventajas e inconvenientes

En java tenemos varias herramientas, pero la más usada (o quizás la que empezó antes) es Hibernate

Podríamos abordar todo desde Hibernate, sin embargo desde que ese software empezó a implementar **JPA** (**estandar java para ORM**) no tiene mucho sentido abundar en las características propias del framework, sino trabajar con el estándar. Usando el estándar no nos importa demasiado si debajo tenemos EclipseLink, Hibernate, o cualquier otro. Es cierto que hay particularidades que perdemos al usar el estándar. Pero para el trabajo habitual es más que suficiente. Cualquier detalle más avanzado-particular de estos frameworks queda como investigación propia del alumno

Aplicación que seguiremos para el aprendizaje

Para hacer el seguimiento y trabajar con ejemplos, tomaremos el siguiente supuesto:

Queremos tener una aplicación Java que nos permita mantener la información de tipo cambiario de diferentes monedas respecto al euro a lo largo del tiempo.

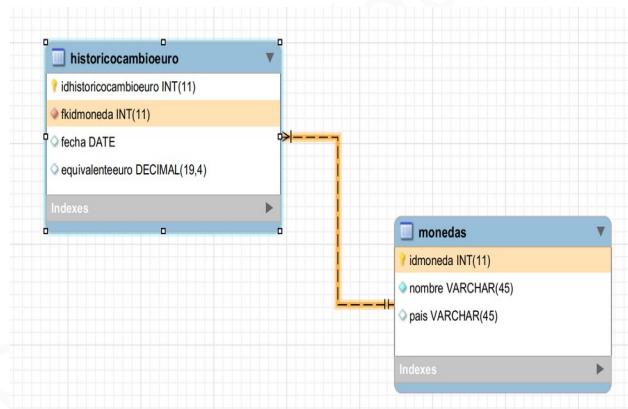
Básicamente lo que nosotros haremos será una aplicación que nos permita hacer un CRUD ("Crear, Leer, Actualizar y Borrar". En original, en inglés: Create, Read, Update and Delete) con las tablas de la base de datos.

Estructura de la base de datos:

Se compone de dos tablas únicamente (hasta que pongamos seguridad). Las tablas:

monedas(idmoneda,nombre,pais)

historicocambioeuro(idhistoricocambi
oeuro,fecha,equivalenteeuro)



Hay una foreign key en historicocambioeuro llamada fkidmoneda que hace referencia a: idmoneda/monedas

Contenido de la tabla monedas:

idmoneda	nombre	pais
1	dolar	Estados Unidos
2	yen	Japón
3	libra	UK

Ilustración 1: tabla monedas

Contenido de la tabla historicocambioeuro:

idhistoricocambioeuro	fkidmoneda	fecha	equivalenteeuro
1	1	2020-01-01	1.1200
2	2	2020-03-25	0.2500
3	2	2019-01-14	0.5000
4	1	2019-12-27	0.9000
5	1	2020-12-30	0.9500
6	1	2020-11-23	0.8000

Ilustración 2: tabla historicocambioeuro

Para un proyecto web se recomienda (si se usa eclipse) crear primero un dynamic webproject, luego convertirlo a maven y agregar las dependencias (están en anexos)

Dependencias maven y configuración proyecto jpa con test h2 en memoria

Desde eclipse: File → new → JPA project nos generará la estructura para un proyecto jpa en eclipse (si se está con un proyecto web es mejor empezar creando un proyecto web y luego transformarlo a jpa que está detallado en los anexos).

En el wizard de la creación del proyecto nos pedirá driver de conexión a base de datos etc
Los pasos a seguir son los mismos que en el anexo: Convertir proyecto en JPA

Dependencias maven incluyendo mysql y h2:

```
<dependencies>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.26</version>
    </dependency>
    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-databind</artifactId>
        <version>2.12.2</version>
    </dependency>
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>5.4.3.Final</version>
    </dependency>
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-entitymanager</artifactId>
        <version>5.4.3.Final</version>
    </dependency>
    <dependency>
        <groupId>javax.persistence</groupId>
        <artifactId>javax.persistence-api</artifactId>
        <version>2.2</version>
    </dependency>
    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <version>2.1.214</version>
    </dependency>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-engine</artifactId>
        <version>5.8.0-M1</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

A lo anterior hay que poner el plugin para maven-surefire que nos hace test maven

Tenemos que ponerlo dentro de: <build> <plugins>

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>3.0.0-M5</version>
    </plugin>
  </plugins>
</build>
```

También hay que configurar el fichero: META-INF/persistence.xml para que pueda acceder a mysql en el caso habitual y a h2 si hay test. Veamos un ejemplo:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.2" 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_2.xsd">
  <persistence-unit name="MatriculasJPA">
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
    <class>es.iespuertodelacruz.jc.entities.Alumno</class>
    <class>es.iespuertodelacruz.jc.entities.Asignatura</class>
    <class>es.iespuertodelacruz.jc.entities.Matricula</class>

    <properties>
      <property name="javax.persistence.jdbc.url"
        value="jdbc:mysql://localhost:3306/instituto?serverTimezone=UTC"/>
      <property name="javax.persistence.jdbc.user" value="root"/>
      <property name="javax.persistence.jdbc.driver"
        value="com.mysql.cj.jdbc.Driver"/>
      <property name="javax.persistence.jdbc.password" value="1q2w3e4r"/>

      <property name="hibernate.dialect" value="org.hibernate.dialect.MySQLDialect" />
      <property name="hibernate.show_sql" value="true" />
    </properties>
  </persistence-unit>

  <persistence-unit name="TEST" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
    <properties>
      <property name="javax.persistence.jdbc.url"
        value="jdbc:h2:mem:test;DB_CLOSE_ON_EXIT=TRUE;DB_CLOSE_DELAY=0;MODE=Mysql;DATABASE_TO_LOWER=TRUE;INIT=RUNSCRIPT FROM 'src/test/resources/instituto.sql';"/>
      <property name="javax.persistence.jdbc.driver" value="org.h2.Driver"/>
      <property name="hibernate.dialect" value="org.hibernate.dialect.H2Dialect"/>
      <property name="hibernate.id.new_generator_mappings" value="true"/>
      <property name="hibernate.globally_quoted_identifiers" value="true" />
      <property name="hibernate.format_sql" value="true"/>
      <property name="hibernate.show_sql" value="true"/>
    </properties>
  </persistence-unit>
</persistence>
```

En la parte de test (la persistence-unit name="TEST") se ha puesto una conexión h2 en memoria (h2:mem:test) con un modo de compatibilidad Mysql, haciendo que no haya problema con mayúsculas y minúsculas (más compatible con mysql) y un fichero sql: instituto.sql que se lanzará en el inicio que se espera encontrar en: src/test/resources/

JPA

Java Persistence API, más conocida por sus siglas JPA, es la API de persistencia desarrollada para la plataforma Java EE.

Persistencia en este contexto cubre tres áreas:

- La [API](#) en sí misma, definida en el paquete `javax.persistence`
- El lenguaje de consulta Java Persistence Query Language (JPQL).
- Metadatos objeto/relacional.

El objetivo que persigue el diseño de esta API es no perder las ventajas de programar orientado a objetos al interactuar con la base de datos (eso ocurría con aproximaciones anteriores como EJB2). Adicionalmente permitir usar objetos sencillos: POJO (Plain Old Java Object: clases java sencillas que no dependen de ningún framework especial: < no usar clases que implementen interfaces de esos frameworks etc >). Vamos a ver como son esas clases:

Definición teórica de Entidades (entities)

Una entidad de persistencia (entity) es una clase de Java ligera, cuyo estado es persistido de manera asociada a una tabla en una base de datos relacional. Las instancias de estas entidades corresponden a un registro (conjunto de datos representados en una fila) en la tabla. Normalmente las entidades están relacionadas con otras entidades, y estas relaciones son expresadas a través de metadatos objeto/relacional. Los metadatos del objeto/relacional pueden ser especificados directamente en el fichero de la clase, usando las anotaciones de Java (annotations), o en un documento descriptivo XML.

¿ Usar anotaciones o xml ?

Nos vamos a detener un momento en lo último que hemos dicho ¿ anotaciones o XML ? Nos inclinaremos por anotaciones al ser más sencillo (menor tamaño) y mejorar la legibilidad. Ahora bien, no está claro que siempre sea lo mejor. Posiblemente la mejor aproximación sea un abordaje mixto (principalmente anotaciones pero para determinadas tareas XML) Vamos a ver ventajas y desventajas de ambos

(tomado del paper: https://www.abis.be/html/en2012-06_XMLvsAnnotations.html)

Ventajas de la anotación:

- 1) Los ficheros XML son mucho más grandes (es su gran desventaja)
- 2) Toda la información está en un solo archivo (no es necesario abrir dos archivos para configurar un comportamiento determinado)
- 3) Cuando la clase cambia, no es necesario modificar el archivo xml

Ventajas del archivo xml:

- 1) Separación clara entre el POJO y su comportamiento.
- 2) Cuando no sabe qué POJO es responsable del comportamiento, es más fácil encontrar ese POJO (buscar en un subconjunto de archivos en lugar de en todo el código fuente)

JPA hace uso de entidades para el mapeo-objeto relacional. Estas clases llevan anotaciones que hacen que se mapee correctamente con las tablas de la base de datos. Veamos un pequeño ejemplo:

```
@Entity  
@Table(name = "usuario")  
public class Usuario implements Serializable {  
  
    private static final long serialVersionUID = 1L;  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    @Basic(optional = false)  
    @Column(name = "idusuario")  
    private Integer idusuario;  
    @Basic(optional = false)  
    @Column(name = "nombre")  
    private String nombre;  
}
```

Basta con fijarse en las anotaciones para entender lo que está ocurriendo sin recurrir casi a la documentación. La clase se ha declarado como una Entity y así nuestro motor de persistencia la tomará como tal. Se le indica que esta entidad corresponde con la tabla usuario de la base de datos.

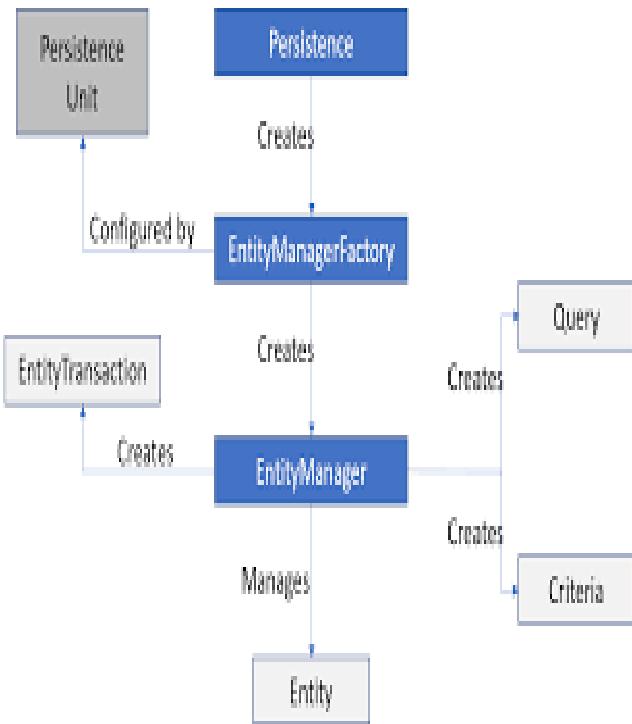
Con: @Id y @GeneratedValue le estamos diciendo que el atributo de la clase: idusuario es clave primaria en la tabla relacional y que es autoincremental. Adicionalmente con: @Column le decimos que corresponde con el campo en la base de datos que se llama: usuario.idusuario (recordar que la entity ya estaba establecida a la tabla usuario, luego el campo debe pertenecer a dicha tabla)

@Basic(optional) se está utilizando para indicar que el campo no es opcional (no admite nulos)

Funcionamiento básico y estructura de JPA

`javax.persistence` es el paquete donde está la API de persistencia de Java JPA

Podemos observar que desde ese espacio creamos un objeto llamado: `EntityManagerFactory`. Ese es un objeto fundamental para el trabajo que vayamos a desarrollar. Realmente a nivel de código lo iniciaremos únicamente al principio de nuestro trabajo con Entidades (típicamente lo iniciamos cuando arrancamos la aplicación) y lo terminaremos cerrando al finalizar todo uso de persistencia en nuestra aplicación (típicamente cuando cerramos la aplicación). Este objeto se basa en la configuración que se haya establecido en la unidad de persistencia: `Persistence Unit`, que básicamente es un fichero xml donde definimos cosas como el driver que se conecta a la base de datos, etc.



La gestión real la haremos con los objetos que le solicitamos al `EntityManagerFactory`, que son los `EntityManager`

Como su propio nombre indica, los `EntityManager` son los que gestionan nuestras entidades. En el grafismo vemos que dice que los `EntityManager` “manages” las `Entity`. Lo cierto es que esa gestión involucra todo lo que nos imaginamos: `persist()` nos permitiría guardar un objeto (una entidad) como una tabla en la base de datos, `remove()` nos permitiría borrar el objeto de la base de datos, etc.

Finalmente también vemos que las query y criteria dependen del `EntityManager`, esto lo que nos viene a decir es que cualquier consulta (query) la haremos desde el `EntityManager`

ej.

```
void mimetodopersistencia( EntityManager em ){
    List<MiEntity> lista = em.createQuery(
        "SELECT c FROM ... blablab")
        .getResultList();
}
```

Ahora iremos poco a poco desgranando los elementos anteriores y agregando cosas que no hemos incluido en el grafismo como las transacciones.

EntityManagerFactory

Los Factory son estrategias que han surgido en los últimos tiempos en Java, como “fábricas” de objetos en lugar de recurrir a los clásicos: new MiClase() que se usan en programación objetos.

Es fácil entender que EntityManagerFactory tiene como principal objetivo ser una fábrica de objetos (los entitymanager que veremos más adelante) Pero es una pieza fundamental en el trabajo con JPA.

Partimos de esta factoría para que nos cree los entitymanager en función de una configuración que le hayamos dado en persistence.xml . Luego serán realmente los entitymanager los que nos hagan el trabajo habitual

Como hemos dicho, para el uso de JPA hace falta crear un EntityManagerFactory (se recomienda desde inicio de la aplicación) y no lo cerraremos hasta que finalice

Esto es porque es un recurso costoso y es mejor dejarlo cargado en memoria todo el tiempo

El EntityManagerFactory toma la información de su configuración del fichero persistence.xml (la unidad de persistencia)

Un EntityManagerFactory es normalmente único y es con el que nosotros gestionamos todas las entidades (mediante los entitymanager que genere). Ahora bien si tenemos varias conexiones a base de datos distintas (nuestra app ataca varias bases de datos) deberemos definir un concepto que nos permite clarificar que tenemos dos EntityManagerFactories distintos. Este concepto es el que se conoce como PersistenceUnit o unidad de persistencia. Cada PersistenceUnit tiene asociado un EntityManagerFactory diferente que gestiona un conjunto de entidades distinto.

Para entender mejor lo dicho, veamos el proceso cuando nosotros creamos un EntityManagerFactory:

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("unidadPersistencia");
```

Podemos observar que llamamos a la librería Persistence (pertenece a javax) y le decimos que nos cree el factory BASÁNDONOS EN UNA UNIDAD DE PERSISTENCIA que nosotros le decimos cuál es mediante un parámetro.

Vamos a ver el fichero: persistence.xml que es donde está la unidad de persistencia que se solicita para crear el EntityManagerFactory anterior:

Nota: este fichero persistence.xml nos vale para su uso en general si queremos trabajar con hibernate y mysql

Si queremos usarlo en otras aplicaciones hibernate mysql tendremos que modificar principalmente:

- la url para establecer donde está el SGBD
- user para decir el usuario con el que nos conectamos
- password para decir la clave de ese usuario)

persistence.xml (alojado en src/main/resources/META-INF)

```
<?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="unidadPersistencia" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>

    <exclude-unlisted-classes>false</exclude-unlisted-classes>
    <properties>
      <property name="javax.persistence.jdbc.url"
value="jdbc:mysql://localhost:3306/seguidormonedas?serverTimezone=UTC"value="com.mysql.cj.jdbc.Driver"value="org.hibernate.dialect.MySQLDialect" />
        <property name="hibernate.show_sql" value="true" />
    </properties>
  </persistence-unit>
</persistence>
```

Hemos destacado, primero el nombre de la persistence-unit que es precisamente lo que se pasa como parámetro a la hora de crear el EntityManagerFactory

Después hemos destacado los tres datos que habitualmente modificaremos en este fichero persistence.xml cuando ataquemos una base de datos Mysql (url, nombreusuario, password)

Observando el fichero podemos darnos cuenta que efectivamente la unidad de persistencia queda asociada a una base de datos en concreto. Cuando nosotros creamos un EntityManagerFactory (que hemos visto que tenemos que pasar como parámetro la unidad de persistencia) necesariamente ese factory esta indisolublemente ligado a una única base de datos. Y por tanto puede haber proyectos que necesitemos varias EntityManagerFactory.

Dicho lo anterior, vemos que habitualmente NO HAY MÁS NECESIDAD QUE UN ÚNICO OBJETO EntityManagerFactory, una única unidad de persistencia y que tendrá vida para toda la aplicación. Es por eso que en nuestra aplicación de ejemplo vamos a apoyarnos en el patrón singleton para mantener la entitymanagerfactory. Es importante que tengamos en cuenta que eso no

siempre será lo correcto por lo que hemos dicho antes... Hay que considerar la posibilidad de que estemos atacando varias bases de datos desde la misma aplicación.

¿ dónde ponemos ese fichero persistence.xml ?

Vamos a crear una carpeta: **src/main/resources**

Y dentro de esa carpeta, otra llamada: **src/main/resources/META-INF**

Es en esa ubicación donde pondremos el fichero persistence.xml



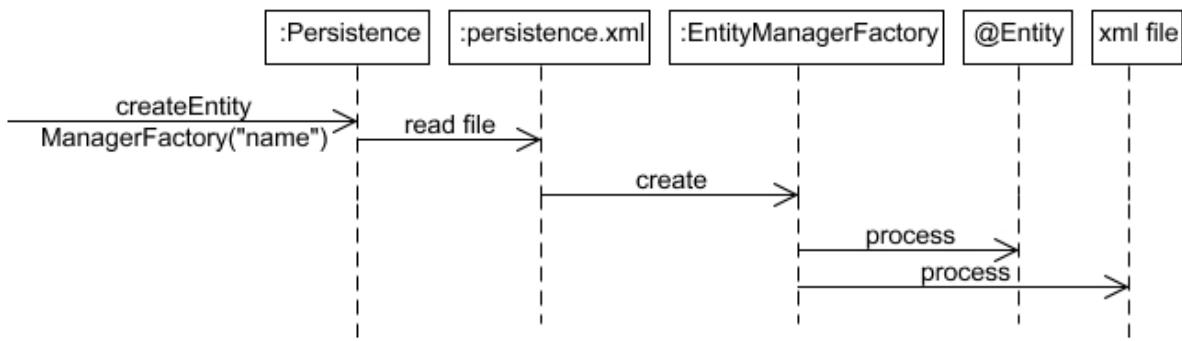
Como hemos dicho ya, podemos ahora crear un EntityManagerFactory Para ello sólo tenemos que escribir:

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("unidadPersistencia");
```

Es importante que observemos que el nombre que ponemos: "unidadPersistencia" debe ser el mismo que hayamos puesto en persistence.xml

Hemos nombrado que ya podemos crear nuestro EntityManagerFactory pero con eso no llegaremos muy lejos... Si el sistema no tiene información de las entidades y las correspondientes tablas en base de datos no podremos hacer mucho.

Veamos el esquema de nuevo:



Ya hemos visto la parte en la que: `ManagerFactory("name")` toma de `persistence.xml` la información y obtenemos el `EntityManagerFactory`. Ahora debe procesar las diferentes `@Entity` (las diferentes entidades) para poder hacer el mapeo entre las clases (entidades) y las tablas de la base de datos.

Antes ya vimos un poco las entidades. Vamos a abordar las primeras anotaciones y otras consideraciones.

Entidades (Entities)

Las entidades las pondremos en su propio paquete. En la aplicación de ejemplo el paquete aparece con el nombre:

es.iespuertodelacruz.jc.primerjpa.entity

Como sabemos el patrón MVC (modelo vista controlador) nos dice que tenemos que tener todos los accesos de persistencia (ficheros, bases de datos) en la parte del modelo. Eso ya implicaba directamente que las entidades tener un espacio separado del resto de la aplicación. Pero daremos un paso más y las pondremos en su propio paquete

Empecemos por ver como nos genera un IDE la Clase/Entidad Moneda (recordar la tabla monedas de la base de datos)

```
@Entity  
@Table(name="monedas")  
@NamedQuery(name="Moneda.findAll", query="SELECT m FROM Moneda m")  
public class Moneda implements Serializable {  
    private static final long serialVersionUID = 1L;  
  
    @Id  
    @GeneratedValue(strategy=GenerationType.IDENTITY)  
    @Column(unique=true, nullable=false)  
    private Integer idmoneda;  
  
    @Column(nullable=false, length=45)  
    private String nombre;  
  
    @Column(length=45)  
    private String pais;  
  
    //bi-directional many-to-one association to Historicocambioeuro  
    @OneToMany(mappedBy="moneda")  
    private List<Historicocambioeuro> historicocambioeuros;  
  
    public Moneda() {  
    }  
  
    public int getIdmoneda() {  
        return this.idmoneda;  
    }  
  
    // se omiten los getter y los setter  
}
```

Primero y lo más importante. Ya hemos dicho que una Entity es una clases Java simple (un POJO) De hecho cumple todos los requisitos de un JavaBean: <https://docs.oracle.com/javaee/6/tutorial/doc/bnbqa.html#:%7E:text=An%20entity%20is%20a%20lightweight,entities%20can%20use%20helper%20classes>.

Adicionalmente tiene que tener la anotación: @Entity Podemos ver la descripción completa en la documentación oficial: <https://docs.oracle.com/javaee/6/tutorial/doc/bnbqa.html#:%7E:text=An%20entity%20is%20a%20lightweight,entities%20can%20use%20helper%20classes>.

Muchas veces veremos que se usan términos como POJO y BEAN siendo tratadas a veces indistintamente (de hecho nosotros a veces lo haremos así) Vamos a ver de qué estamos hablando

POJO vs BEAN

Bien, volvamos de nuevo a wikipedia y la definición de **POJO**:

Un objeto POJO es una instancia de una clase que no extiende ni implementa nada en especial. Por ejemplo, un [Servlet](#) tiene que extender de `HttpServlet` y sobrescribir sus métodos, por lo tanto no es un POJO. En cambio, si se define una clase 'Persona', con sus atributos privados y sus correspondientes getters y setters públicos, una instancia de esta simple clase es un objeto POJO

Ahora veamos lo que dice wikipedia respecto a **JavaBean**:

Para funcionar como una clase JavaBean, una clase debe obedecer ciertas convenciones sobre nomenclatura de métodos, construcción y comportamiento.

Estas convenciones permiten tener herramientas que puedan utilizar, reutilizar, sustituir y conectar JavaBeans.

Las convenciones requeridas son:

- Debe tener un constructor sin argumentos.
- Sus atributos de clase deben ser privados.
- Sus propiedades deben ser accesibles mediante métodos get y set que siguen una convención de nomenclatura estándar.
- Debe ser serializable

Vistas las definiciones de wikipedia de ambos elementos Observamos que hay similitudes. Pero podemos concluir que:

Los Beans son tipos especiales de POJOS. Hay algunas restricciones en POJO para que puedan ser Beans.

1. **Todos los JavaBeans son POJO pero no todos los POJOs son JavaBeans.**
2. **Serializables**, es decir, deben implementar la interfaz Serializable. **Aún algunos POJOs que no implementan interfaz Serializable se llaman POJOs** porque Serializable es una interfaz de marcadores y por lo tanto no de mucha carga.
3. **Los campos deben ser privados.** Esto es para proporcionar el control completo en los campos.
4. Los campos deben tener getters o setters o ambos.
5. Un constructor sin argumentos debe existir en un bean.
6. Se accede a los campos sólo por constructor o getter o setters.

Por qué se usa la interfaz serializable?

Por lo general la información que se persiste debe viajar mediante la red a un servidor por lo que un objeto que se envía a guardar debe ser descompuesto en bytes, la interfaz serializable permite que un objeto sea descompuesto a bytes y que al otro lado pueda ser reconstruido

Como observamos en la entidad que nos ha creado el IDE automáticamente vemos que cumple con las condiciones de un BEAN, y por tanto también de POJO Es interesante tener en cuenta que al ser JavaBeans también son componentes Java:
https://cgrw01.cgr.go.cr/rup/RUP.es/LargeProjects/tech.j2ee/guidances/concepts/javabean_D488CF3B.html

Pasemos ahora a detallar las anotaciones:

Anotaciones en las Entities

@Entity Como ya hemos dicho antes, es requisito para que una clase sea una entidad, que sea anotada de esta forma: @Entity

@Table(name="monedas") Estamos informando de la tabla de la base de datos a la que mapea la Entidad. Hay comportamientos por defecto (si no especificamos el nombre de la tabla), que es que el nombre de la tabla será coincidente con el de la clase. Nosotros, siempre pondremos el nombre de la tabla a la que mapea la entidades. También permite establecer los índices de las tablas:

```
@Table(  
    name = "EMPLOYEES" ,  
    schema = "jpatutorial",  
    indexes = {@Index(name = "name_index", columnList = "name",unique = true)})
```

@Column Nos permite establecer características particulares del campo de la tabla correspondiente

- **name** Permite especificar un nombre diferente en el atributo de la clase al campo de la tabla (en otro caso deben coincidir en el nombre)
- **length** Permite establecer una longitud del campo. En nuestro ejemplo es de 45 (varchar 45 es lo que hemos puesto exactamente en la tabla de la base de datos)
- **nullable** Permite especificar si el atributo admite nulos Vemos que en nuestro ejemplo el nombre de la moneda no admite nulos
- **unique** Permite establecer que no se permiten repeticiones del valor del campo en la tabla (equivalente a esa misma cláusula en SQL)

@Id Nos permite establecer cuál es la primary key. El ID puede ser cualquier tipo de datos soportado por JPA, como puede ser, todos los tipos primitivos y clases wrapper (Integer, Long,etc).

Si bien existe libertad es recomendable usar los wrapper porque aceptarán circunstancias de nulos, por ejemplo.

Hay otras anotaciones relacionadas:

- **@GeneratedValue** JPA cuenta con la anotación @GeneratedValue para indicarle a JPA que regla de autogeneración de la llave primaria vamos a utilizar. JPA soporta 4 estrategias de autogeneración de la llave primaria:
 - **Identity:** Esta estrategia es la más fácil de utilizar pues solo hay que indicarle la estrategia y listo, no requiere nada más, **JPA cuando persista la entidad no enviará este valor, pues asumirá que la columna es auto generada.** Es apropiado para los autoincrementales de Mysql
 - **Sequence:** Mediante esta estrategia le indicamos a JPA que el ID de generar a través de una secuencia de la base de datos. De esta manera, cuando se realice un insert, esta agregara la instrucción para que en el ID se inserte el siguiente valor de la secuencia. Es apropiado para los sequence de Oracle
 - **Auto:** Esta estrategia lo único que hace es decirle a JPA que utilice la estrategia por default para la base de datos con la que estamos trabajando.

@OneToMany, ManyToOne: En nuestro ejemplo dice:

```
//bi-directional many-to-one association to Historicocambioeuro
@OneToMany(mappedBy="moneda")
```

Si nos fijamos en lo que indican los propios comentarios autogenerados nos está diciendo que la información de la relación uno a muchos estaba informado en la relación muchos a uno. Luego cuando leemos la sentencia exactamente: `@OneToMany(mappedBy="moneda")` nos dice que es una relación uno a muchos que está establecida en el atributo “moneda” ¿ A qué atributo hace referencia ? Se refiere al atributo llamado: “moneda” que está en la Entity `Historicocambioeuro` (que es la parte de muchos en la relación uno a muchos)

Así que realmente para que el motor de persistencia sepa como comportarse con esta relación hay que mirar en la otra Entity:

```
//bi-directional many-to-one association to Moneda
@ManyToOne
@JoinColumn(name="fkidmoneda", nullable=false)
private Moneda moneda;
```

Lo primero que observamos es lo que hemos comentado antes: El atributo se llama: moneda, que es el que aparece en el onetomany: @OneToMany(mappedBy="moneda")

Lo siguiente que observamos es que aparece una nueva anotación:

- **@JoinColumn** Esta anotación nos indica cuál es el nombre del campo de la tabla que es foreign key. En este caso nos dice que ese campo es: fkidmoneda. También nos dice que no puede ser nulo

¿ entonces cuál es el procedimiento para relación 1:N ?

```
/* CLASE Moneda PARTE 1 DE LA RELACIÓN 1:N */ /* CLASE Historicocambioeuro PARTE N DE LA RELACIÓN 1:N */
//bi-directional many-to-one association to Historicoca //bi-directional many-to-one association to Moneda
@OneToMany(mappedBy="moneda") @ManyToOne
private List<Historicocambioeuro> historicocambioeuros; @JoinColumn(name="fkidmoneda", nullable=false)
private Moneda moneda;
```

- En la Entity de la relación que es la parte 1 ponemos la notación @OneToMany y se indica que campo de la otra Entity define la relación: mappedBy="moneda" sobre la colección (List, Set)

- En la Entity de la relación que es la parte N ponemos la notación @ManyToOne y se indica el nombre del campo de la tabla que tiene la foreign key mediante @JoinColumn: @JoinColumn(name="fkidmoneda", nullable=false) sobre el atributo de la otra entidad (moneda en este caso). **¡¡ importante !! Observar que ES EL OBJETO COMPLETO, NO ÚNICAMENTE EL ID (en tablas relacionales sabemos que la foreign key es un id de la tabla origen. En este caso no es así, es un objeto completo)**

Debemos conocer las anotaciones y en algunos casos hay que modificarlas (la veremos que es lazy y eager), pero muchas veces podemos apoyarnos en el IDE para generar las entities

Un caso especial es cuando queremos saltarnos la tabla intermedia entre una relación N:M vamos a verlo

@ManyToMany

Supongamos el caso de una base de datos de matrículas de alumnos de un instituto. Una asignatura participa de múltiples matrículas de alumno y una matrícula de alumno se realiza con múltiples asignaturas. Estamos pues, en un caso ManyToMany

Cuando hay atributos adicionales en la relación (como por ejemplo, número de convocatoria del alumno para una asignatura_matricula) habrá de crearse el objeto intermedio: AsignaturaMatricula pero si no hay nada relevante sino es únicamente una tabla de unión podemos “puentejar” la creación de ese objeto-entity en Java y quedará únicamente un mapeo para que JPA sepa como trabajar con la tabla intermedia. El siguiente código está puesto en la entity: Asignatura

Entity Asignatura:

```
@ManyToMany  
 @JoinTable( name="asignatura_matricula",  
             joinColumns = @JoinColumn(name="idasignatura"),  
             inverseJoinColumns = @JoinColumn(name="idmatricula")  
 )  
  
private List<Matricula> matriculas;
```

y en la entity: Matricula la parte de código que corresponde es:

```
@ManyToMany(mappedBy="matriculas")  
private List<Asignatura> asignaturas;
```

Como vemos la relación mediante **@JoinTable** se establece en una única Entity, en este caso: Asignatura **NO** debemos poner la relación en las dos entities o nos arriesgamos a problemas de referencias cruzadas. Si ponemos en una tabla la forma de unión: **@JoinTable** en la otra entity escribiremos **mappedBy**

IMPORTANTE: Para que nos genere entradas en la tabla intermedia tienen que estar bien establecidas las relaciones en nuestros objetos Entity creados.

Veamos ejemplo para matrícula y asignatura:

```
EntityManager em = emf.createEntityManager();

Alumno alumno = new Alumno();
alumno.setDni("12345678Z");
alumno.setNombre("alumnoPrueba");
alumno.setApellidos("alumnoApellido");

em.getTransaction().begin();
em.persist(alumno);
em.getTransaction().commit();

Matricula matricula1 = new Matricula();
matricula1.setAlumno(alumno);
matricula1.setYear(2022);
matricula1.setAsignaturas(new ArrayList());

Asignatura asignatura = new Asignatura();
asignatura.setCurso("3ºDAM");
asignatura.setNombre("FCTppp");
asignatura.setMatriculas(new ArrayList());

//Ahora las llamadas cruzadas entre los dos objetos:
asignatura.getMatriculas().add(matricula1);
matricula1.getAsignaturas().add(asignatura);

em.getTransaction().begin();
em.persist(matricula1);
em.persist(asignatura); // esta linea si no hay cascade persist
em.getTransaction().commit();

em.close();
```

@Transient

En ocasiones queremos que JPA no tenga en cuenta algún campo para su interacción con la base de datos.

El siguiente ejemplo es en Sakila respecto a Staff y campos de imagen:

```
@Transient  
@JsonIgnore  
@Lob  
private byte[] picture;
```

JPA ignorará el campo. El equivalente en JSON es: **@JsonIgnore**

@JsonBackReference, @JsonManagedReference

En ocasiones se nos dan referencias cruzadas en JSON una opción es usar **@JsonIgnore** y entonces no “existirá” el campo para json pero la anotación **@JsonBackReference** es más apropiada.

Veamos ejemplo, observar que el backreference va en el oneToMany y el managed en manyToOne

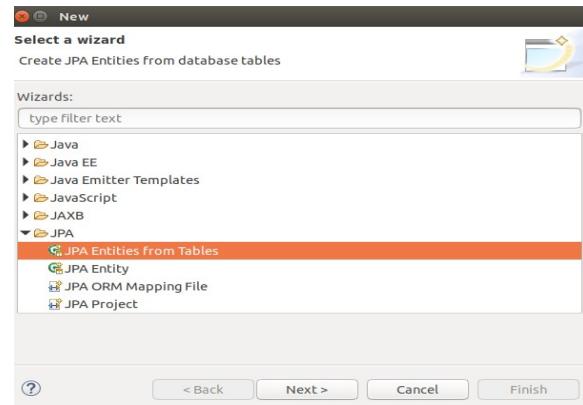
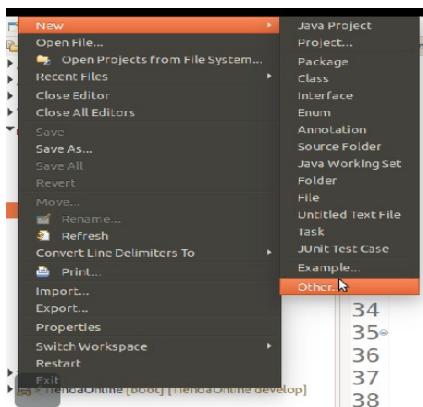
```
1  @QClass(name = "clase")  
2  @Entity  
3  @Table(name = "b_clazz")  
4  public class Clazz extends BaseEntity {  
5      @OneToMany(mappedBy = "clazz")  
6      @JsonBackReference  
7      private Set<Student> students; //estudiante  
8  }
```

```
1  @QClass(name = "estudiante")  
2  @Entity  
3  @Table(name = "b_student")  
4  public class Student extends BaseEntity {  
5      @ManyToOne  
6      @JoinColumn(name = "clazz_id")  
7      @JsonManagedReference  
8      private Clazz clazz; // Clase perteneciente  
9  }
```

Procedimiento para generación Entities de forma automática

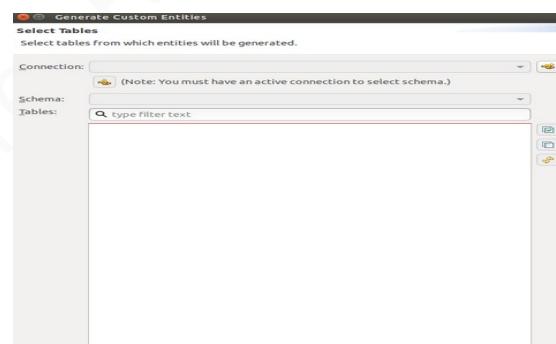
En Eclipse haremos:

- File → new → other → JPA → JPA Entities from tables



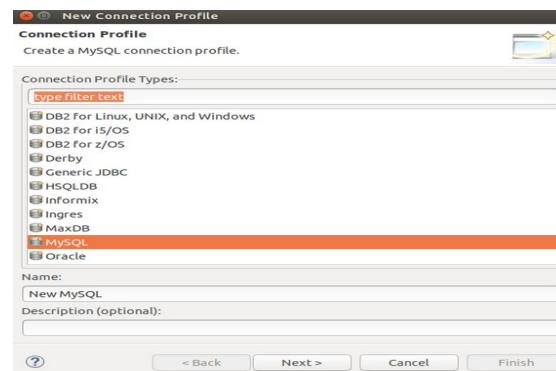
- Se nos abre un wizard con la pantalla:

En esta pantalla debemos establecer la conexión a la base de datos. (observar donde dice Connection) hay un botón a la derecha que nos permite crear una nueva conexión si fuera pertinente

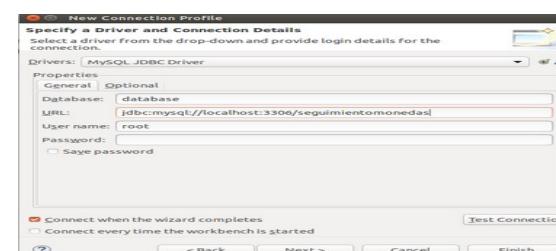


- Se nos abre una nueva ventana para especifica esa nueva conexión. Elegimos MySQL y pulsamos next.

Elegiremos la versión del driver que tengamos ya descargado (por ejemplo la versión 5.1) y elegimos en el disco duro la ubicación donde tengamos ese driver (por ejempl la 5.1.49)

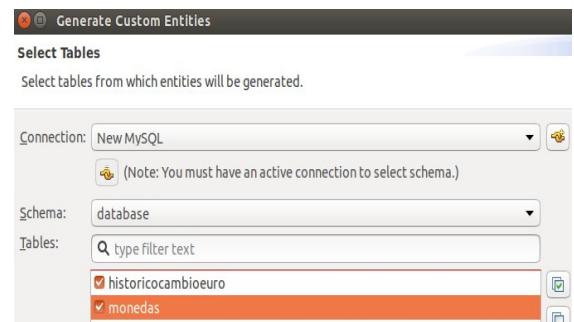


- Cuando hayamos establecido ese driver volvemos a una pantalla donde especificamos



los parámetros para la conexión. Vemos que especificamos la url de la conexión jdbc (básicamente agregamos el nombre de la base de datos) . También damos el nombre del usuario y la clave

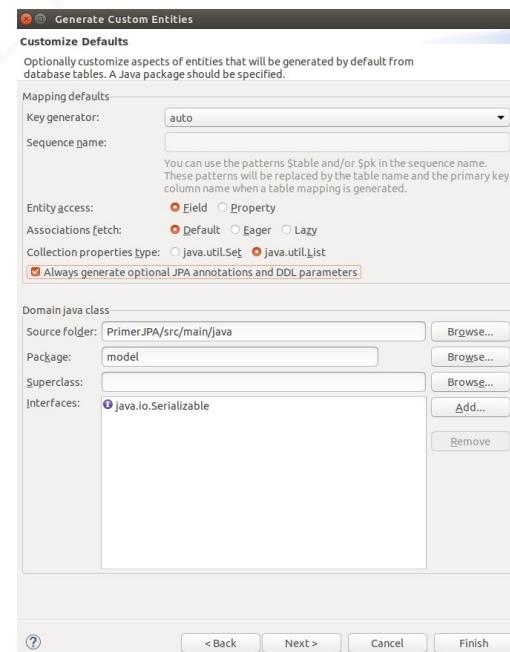
- Al pulsar next nos oferta las tablas que encuentra en la base de datos y de las que nosotros queremos crear entities. Elegiremos todas



Finalmente nos preguntará cosas como el nombre de la ubicación donde queremos que nos genere las entities (ya hemos explicado en páginas anteriores que es interesante un subpaquete llamado entity) .

En esa ventana nos interesa marcar que nos genere las anotaciones opcionales y parámetros DDL

Ahí aparecen cosas que nombramos antes (elegir entre Lazy Eager) esa parte la dejaremos como está y pulsamos en finalizar



Con lo anterior habremos creado automáticamente las entities

Vale, en este punto ya tenemos todo prácticamente preparado para ya poder trabajar. Ahora ya pasaremos a trabajar directamente con código usando EntityManager (es lo que genera el EntityManagerFactory que ya hemos realizado)

EntityManager

Vamos a empezar con un ejemplo de uso e iremos explicando. El siguiente código lo podemos poner en el método main de nuestro proyecto

```
public static void main(String[] args) {  
  
    List<Moneda> monedas = null;  
    EntityManagerFactory emf =  
Persistence.createEntityManagerFactory("unidadPersistencia");  
    EntityManager em = emf.createEntityManager();  
    EntityTransaction tr = em.getTransaction();  
  
    tr.begin();  
    String query = "SELECT m FROM Moneda m";  
    monedas = em.createQuery(query, Moneda.class)  
        .getResultList();  
    tr.commit();  
    em.close();  
  
    monedas.stream()  
        .forEach(p->System.out.println(p));  
  
    emf.close();  
}
```

Importante: NO es buena costumbre crear y cerrar EntityManagerFactory en cada acción, es un recurso costoso para el sistema. Como ya hemos dicho anteriormente debe ser una única vez al crear la aplicación y únicamente cerrarla cuando acaba la aplicación. Lo que ocurre es que en éste ejemplo en concreto, es coincidente con el inicio de nuestra aplicación y con su finalización (todo empieza y acaba en el método main)

Lo siguiente a la creación del EntityManagerFactory que vemos en el código de ejemplo es que creamos un EntityManager: `EntityManager em = emf.createEntityManager();`

Esta llamada no es demasiado costosa, ya que las implementaciones de JPA implementan pools de entity managers. El método `createEntityManager` no realiza ninguna reserva de memoria ni de otros recursos sino que simplemente devuelve alguno de los entity managers disponibles. Esto significa que no debe suponernos demasiado problema crear y eliminar EntityManager ni que eso nos influya en la cantidad de vida que tenga ese instancia de objeto.

La vida de los EntityManager muchas veces es coincidente con la transacción que hayamos pensando realizar, pero esto no tiene por qué ser siempre así. De hecho hay aplicaciones que tienen

una única EntityManager durante toda su ejecución. Ahora bien, es importante tener en cuenta que los EntityManager no son thread-safe (esto significa que hay peligro de corrupción si varios hilos de ejecución tratan de acceder al mismo EntityManager. Esta es una situación que se puede dar fácilmente en aplicaciones web, con múltiples peticiones simultáneas de los usuarios) Una buena práctica en esos ambientes web es crear el EntityManager para cada petición web y eliminarlo al finalizar, de esa forma no habría peligro de corrupción.

La siguiente parte de código, vemos que usamos el EntityManager para obtener una transacción (nos acostumbraremos a hacer todas las acciones de Entities mediante transacciones, aunque no siempre sea estrictamente necesario).

```
tr.begin();
String query = "SELECT m FROM Moneda m";
monedas = em.createQuery(query, Moneda.class)
    .getResultList();
tr.commit();
```

En el trozo de código vemos que la transacción nos cubre todo el recorrido desde que se declara el `.begin()` hasta el `commit()` de tal forma que tiene un comportamiento atómico todo lo que se haya realizado en medio (Esto es, si insertas una fila en la tabla: monedas y modificas una fila en la tabla de historico y al final algo falla, y una de las acciones no tiene lugar, ninguna tendrá lugar. En este sentido es atómico: o se validan y todas salen ok o ninguna quedará registrada en la base de datos)

Dentro de la transacción ejecutamos mediante el entitymanager una consulta (`query`) a la base de datos. El resultado nos lo devuelve en forma de lista. El lenguaje que vemos no es SQL sino JPQL Es una variante que muestra consultas desde un punto orientado a Objetos. Si nos fijamos no usamos el nombre de la tabla: monedas en el FROM lo que usamos es el nombre de la Entidad: Moneda. Por otro lado también vemos que el registro (el equivalente a `select *`) se trata como un objeto de tipo Moneda (lo que se muestra en la query como la variable: `m`)

Ya con lo anterior nos vemos competentes para hacer pequeñas operaciones contra la base de datos mediante JPA, pero debemos seguir profundizando en el concepto de EntityManager

Responsabilidades del EntityManager

El entity manager tiene dos responsabilidad fundamentales:

- Define una conexión transaccional con la base de datos que debemos abrir y mantener abierta mientras estamos realizando operaciones. En este sentido realiza funciones similares a las de una conexión JDBC.
- Además, mantiene en memoria una caché con las entidades que gestiona y es responsable de sincronizarlas correctamente con la base de datos cuando se realiza un flush. El conjunto de entidades que gestiona un entity manager se denomina su contexto de persistencia.

Operaciones del EntityManager

El API de la interfaz EntityManager define todas las operaciones que debe implementar un entity manager. Destacamos las siguientes:

- **void clear()**: borra el contexto de persistencia, desconectando todas sus entidades Entity Manager y contexto de persistencia
- **boolean contains(Object entity)**: comprueba si una entidad está gestionada en el contexto de persistencia
- **Query createNamedQuery(String name)**: obtiene una consulta JPQL precompilada
- **void detach(Object entity)**: elimina la entidad del contexto de persistencia, dejándola desconectada de la base de datos
- **<T> T find(Class<T>, Object key)**: busca por clave primaria
- **void flush()**: sincroniza el contexto de persistencia con la base de datos
- **<T> T getReference(Class<T>, Object key)**: obtiene una referencia a una entidad, que puede haber sido recuperada de forma lazy

- **EntityTransaction getTransaction()**: devuelve la transacción actual
- <T> **T merge(T entity)**: incorpora una entidad al contexto de persistencia, haciéndola gestionada
- **void persist(Object entity)**: hace una entidad persistente y gestionada
- **void refresh(Object entity)**: refresca el estado de la entidad con los valores de la base de datos, sobreescribiendo los cambios que se hayan podido realizar en ella
- **void remove(Object entity)**: elimina la entidad

Vamos a ver varias con más detalle. Pero antes comentar que **las operaciones que tengan repercusión en la base de datos (cambios) debemos hacer uso de transacciones porque en otro caso no tendrá lugar. Únicamente las lecturas no será necesario usar transacciones**

persist() (Guardando una nueva Entity)

El método **persist()** del EntityManager acepta una nueva instancia de entidad y la convierte en gestionada ¡¡cuidado no la pone en la base de datos. Lo que hace es ponerla en el contexto de persistencia!!!. La operación **contains()** puede usarse para comprobar si una entidad está gestionada.

El hecho de convertir una entidad en gestionada no la hace persistir inmediatamente en la base de datos. La verdadera llamada a SQL para crear los datos relacionales no se generará hasta que el contexto de persistencia se sincronice con la base de datos. Esto habitualmente es un commit de la transacción. En el momento en que la entidad se convierte en gestionada, los cambios que se realizan sobre ella afectan al contexto de persistencia (si le modificas un atributo: moneda.setNombre("nuevonombre") el contexto de persistencia lo tiene en cuenta). Y en el momento en que la transacción termina, el estado en el que se encuentra la entidad es volcado en la base de datos. **La operación persist() se utiliza con entidades nuevas que no existen en la base de datos. Si se le pasa una instancia con un identificador que ya existe en la base de datos el proveedor de persistencia puede detectarlo y lanzar una excepción EntityExistsException . Si no lo hace, entonces se lanzará la excepción cuando se sincronice el contexto de persistencia con la base de datos, al encontrar una clave primaria duplicada (esto es cuando hacemos commit en la transacción)**



Práctica 1: En el proyecto de monedas crear entitymanagerfactory, entitymanager , crear todo lo necesario y persistir una moneda: nombre:libra, pais:uk pero sin usar transacciones. ¿ se ha generado en la base de datos ?. ¿ lanza error ? Comentar lo ocurrido y tomar captura de pantalla

Seguramente parte del código realizado anterior se parezca a:

```
Moneda moneda = new Moneda();
moneda.setNombre("Lira");
moneda.setPais("Turquía");
em.persist(moneda);
em.close();
```

Lo anterior por lo tanto no funciona. Se cierra la conexión y no guarda. Debemos hacer uso de transacciones

● **Práctica 2:** Modificar lo anterior pero esta vez dentro de una transacción y hacer que funcione Tomar captura de la base de datos mostrando el registro creado. Poner también el código creado

Sea el siguiente código:

```
Moneda moneda = new Moneda();
moneda.setNombre("Lira");
moneda.setPais("Turquía");
em.persist(moneda);
em.getTransaction().begin();
em.getTransaction().commit();
em.close();
```

● **Práctica 3:** Probar con el código anterior. Observar que no se ejecuta nada dentro de la transacción . ¿ Se ha guardado la moneda en la DDDB ? Escribir lo que ha ocurrido si hay error detallarlo etc.

Como vemos lo importante es ejecutar una transacción ¿ Entonces que ocurrirá si ponemos el siguiente código ?

```
Moneda moneda = new Moneda();
moneda.setNombre("Liral");
moneda.setPais("Turquía");
em.persist(moneda);
em.getTransaction().begin();
moneda.setNombre("lira2");
em.getTransaction().commit();
moneda.setNombre("lira3");
em.close();
```

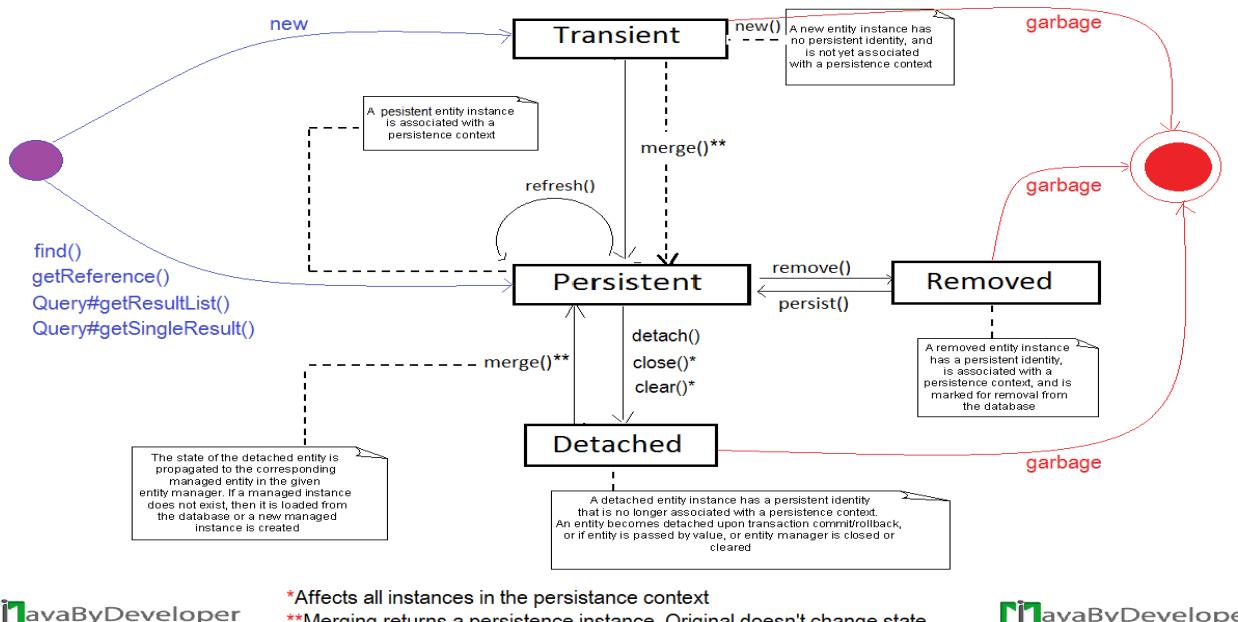
● **Práctica 4:** ¿ qué nombre queda guardado en la base de datos con el código anterior?

Actualizando una entidad (y uso de merge())

¿ En qué situaciones actualizamos ?

Antes observamos que después de hacer un persist() podemos modificar todas las veces que queramos una entidad que mientras siga abierto el entity manager donde se ejecutó el persist terminará trascendiendo los cambios (si ejecutamos una transacción) Entonces los casos de actualizaciones que nos quedan por analizar son aquellos cuando la entity manager que ejecutó el persist() se ha cerrado: entitymanager.close()

Al hacer un entiymanager.close() la entity pasa del estado managed (persistent) al detached veamos un gráfico de los estados:



Vemos en el grafismo que al ejecutar persist() la Entidad pasa a pertenecer al contexto de persistencia (está vigilada y se tiene en cuenta cualquier cambio) Ese es el estado de managed (persistent)

En el grafismo vemos que si se hace un entitymanager.close() o entitymanager.clear() la entidad entra en **Detached** ¿ qué significa ? Es una entidad que ya no está vigilada por el motor de persistencia. Hay una fila correspondiente a la entity en la base de datos pero si modificamos la entidad **NO TIENE REFLEJO EN LA FILA DE LA BASE DE DATOS**. Esto es porque la entidad ya no está vigilada. Así pues para que los cambios tengan lugar hay que volver a managed

El método **merge()** permite volver a incorporar en el contexto de persistencia del entity manager una entidad que había sido desconectada. Debemos pasar como parámetro la entidad que queremos incluir. Hay que tener cuidado con su utilización, porque el objeto que se pasa como parámetro no pasa a ser gestionado. Hay que usar el objeto que devuelve el método

```
public static void mostrarMonedaEnDDBB(  
    Integer idmoneda,  
    EntityManagerFactory emf  
) {  
    EntityManager em = emf.createEntityManager();  
    Moneda moneda = em.find(Moneda.class, idmoneda);  
    System.out.println(moneda);  
    em.close();  
  
}  
  
public static void main(String[] args) {  
    EntityManagerFactory emf =  
    Persistence.createEntityManagerFactory("unidadPersistencia");  
    EntityManager em = emf.createEntityManager();  
    Moneda moneda = new Moneda();  
    moneda.setNombre("Lira32");  
    moneda.setPais("Turquía");  
  
    em.getTransaction().begin();  
    em.persist(moneda); // moneda vigilada → está en contexto  
    em.getTransaction().commit(); //moneda guardada en DDBB  
  
    System.out.println("1: en base de datos: ");  
    mostrarMonedaEnDDBB(moneda.getIdmoneda(), emf);  
    em.close(); //moneda separada→ detached Cambios no persisten en DDBB  
  
    em = emf.createEntityManager(); // nueva conexión a la DDBB  
    em.getTransaction().begin(); //transact abierta para grabar en DDBB  
    moneda.setNombre("Lira34"); //la moneda detach... ;; no se grabará !!  
    em.getTransaction().commit(); // da igual el commit.. no hay efecto  
  
    System.out.println("2: en base de datos: ");  
    mostrarMonedaEnDDBB(moneda.getIdmoneda(), emf);
```

```

Moneda monedaVigilada = em.merge(moneda); //monedaVigilada
//debemos trabajar con monedaVigilada no con moneda para cambios
//aunque en apariencia el system.out nos las muestra iguales
System.out.println("monedavigilada: " + monedaVigilada);
System.out.println("moneda: " + moneda);
monedaVigilada.setNombre("lira35"); //entity managed
moneda.setNombre("lira36"); //entity detached

System.out.println("3: en base de datos antes de transact: ");
mostrarMonedaEnDDBB(moneda.getIdmoneda(), emf);

em.getTransaction().begin();
em.getTransaction().commit(); //ahora sí cambia la DDBB

System.out.println("4: en base de datos: después de transact");
mostrarMonedaEnDDBB(moneda.getIdmoneda(), emf);

em.close();
emf.close();
}

```

● Práctica 5: Ejecutar el código. Tomar los mensajes que muestra el sout. Que número de lira se muestra en el último mensaje ?

Vale, parece que entonces usaremos merge() cuando queramos actualizar una fila de la base de datos que ya no tenemos vigilada. También vemos que hemos usado un par de cosas nuevas en el método mostrarMonedaEnDDBB() la instrucción: **find() de entitymanager**. Esa instrucción ataca directamente la base de datos (no precisa estar bajo una transacción) y busca una Entity por id. El objeto que nos devuelve estará en el contexto de persistencia del entitymanager que lo creó y por tanto, las modificaciones que le hagamos tendrán consecuencias en la base de datos (si se usan transacciones). Pero para eso hemos creado previamente otra entitymanager diferente. Las modificaciones de una Entity en memoria (contexto de persistencia) no queremos que estén afectando a la entity obtenida con find (al ser dos entitymanager tiene cada uno su espacio en memoria... son dos conexiones distintas a la base de datos)

Eso significa que hemos encontrado otro camino para modificar registros de la base de datos. Hacer búsquedas por la información que queramos allí y el objeto que nos devuelva estará en el contexto de persistencia... disponible para aceptar modificaciones.

Lo anterior lo podemos traducir en lo siguiente: Podemos reemplazar la orden merge() de una moneda por un find del id. Veámoslo:

```
Moneda monedaVigilada1 = em.merge(moneda);
Moneda monedaVigilada2 = em.find(Moneda.class, moneda.getIdmoneda());
```

En ambos casos son objetos vigilados por el contexto de persistencia

No pasa nada si con la orden merge usamos el mismo nombre de variable (salvo que pasa a estar managed)

```
moneda = em.merge(moneda);
```

Para terminar con merge() y entender mejor las diferencias con persist(), vamos a suponer que tenemos:

```
EntityManager em = emf.createEntityManager();
Moneda moneda = new Moneda();
moneda.setNombre("Lira35");
moneda.setPais("Turquía");

em.getTransaction().begin();
em.persist(moneda);
em.getTransaction().commit();
em.close();

em = emf.createEntityManager();
Moneda moneda2 = new Moneda();
moneda2.setNombre("Lira36");
moneda2.setPais("Turquía");
moneda2.setIdmoneda(moneda.getIdmoneda());

em.getTransaction().begin();
//moneda2=em.merge(moneda2);
em.persist(moneda2);
em.getTransaction().commit();

em.close();
```

● **Práctica 6:** Ejecutar el código. ¿ funciona el persist() ? ¿ da error ? ¿ cuál ? (tomar captura del error) comentar el persist() y quitar comentario en merge() ¿ ha cambiado algo ?

De lo anterior observamos que una Entidad nueva con un id coincidente con un id de la base de datos hace que la podamos tomar como una entidad detached

También podemos concluir que merge() nos permite guardar una entidad (un insert) sin usar persist()



Práctica 7: Crear una nueva moneda (por supuesto, no ponerle id, que se autogenera) y guardarla en base de datos mediante merge (no usar persist) ¿ funciona ? ¿ da error ?

De lo anterior vemos que podríamos hacer uso de **merge()** y prescindir de usar **persist()** ¿ entonces por qué usar persist ? Pues por circunstancias de rendimiento en el SQL subyacente y porque persist() lanza una excepción si una entidad ya está guardada y la volvemos a guardar. Nos puede ayudar a detectar errores, ya que merge simplemente actualizaría sin más.

Bien, en definitiva podemos afirmar que para modificar una fila en la base de datos debemos tener una Entity en el contexto de persistencia (en memoria) que esté mapeada por la fila. Luego hacer uso de los: set() de la entity y finalmente ejecutar una transacción:

Hacer update fila en DDBB

1. Poner una Entity vigilada (en contexto de persistencia)

que apunte a la fila de la DDBB:

- merge(), find(), ...

2. Hacer los: entity.set() correspondientes

3. ejecutar una transacción

persist y CascadeType.PERSIST

Si queremos guardar entidades que están relacionadas (por ejemplo si guardamos un pedido queremos que los detalles de pedido se guarden a la vez) Tenemos dos aproximaciones posibles:

- Sin usar cascade:

```
EntityManager em = emf.createEntityManager();
em.persist(order);

for( OrderDetail orderdetail: order.getOrderDetails() ) {
    orderdetail.setOrder(order);
    em.persist(orderdetail);
}

em.getTransaction().begin();
em.getTransaction().commit();
em.close();
```

Order — 1: N → OrderDetail

Si nos fijamos primero “vigilamos” (ejecutamos persist) de la Entity Order para que se tenga en cuenta el id autogenerado en la base de datos para usarlo como una foreign key en la Entity OrderDetail. Después garantizamos que cada orderdetail tenga especificado como foreign key la order (orderdetail.setOrder(order)) y finalmente le decimos que también lo persista: em.persist(orderdetail)

Una alternativa a lo anterior es especificar CascadeType.PERSIST de esa forma JPA tratará de persistir todos los orderdetail al persistir order.

Veamos la clase Order de esa forma:

```
public class Order implements Serializable {
    @OneToMany(mappedBy="order" ,cascade=CascadeType.PERSIST )
    private List<OrderDetail> orderDetails;
```

Veamos como quedaría ahora el grabado de un nuevo pedido:

```
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();

for( OrderDetail orderdetail: order.getOrderDetails() ) {
    orderdetail.setOrder(order);
    //em.persist(orderdetail);
}

em.persist(order);
em.getTransaction().commit();
em.close();
```

Vemos que está comentado el persist(orderdetail) ya no es necesario, porque al hacer la persistencia de order también se hace de los orderdetails

Borrar una entidad en la DDBB (remove())

Como hemos dicho, salvo las lecturas de DDBB todo lo demás debemos hacerlo mediante una transacción. En este caso, el borrado de las entidades debe ser también de esa forma (comando remove())

A este respecto es relevante tener en cuenta los efectos de las entidades vinculadas. Por ejemplo, si hay una moneda que tiene registros vinculados en la tabla/entidad: Historicambioeuro ¿qué ocurrirá ?

Vamos a ver la creación de un objeto Historicambioeuro que esté relacionado con una moneda (recordar que se hizo que tuviera un foreign key de Moneda)

```
Historicambioeuro h = new Historicambioeuro();
h.setMoneda(em.find(Moneda.class, 15)); //tomamos la moneda id 15
h.setEquivalenteeuro(new BigDecimal(0.4));
h.setFecha(new Date());
em.getTransaction().begin();
em.persist(h);
em.getTransaction().commit()
```



Práctica 8: Crear una moneda y al menos un Historicambioeuro vinculado a la moneda. Se debe aportar el código en la práctica y captura de lo insertado en la base de datos Luego intentar hacer un remove de la moneda ¿ qué error da ? Tomar captura pantalla

Sabemos que la integridad referencial impide borrar un elemento y eso nos genera error. Hay varias formas de solución. Incluso es posible que no haya fallado si el IDE creó CascadeType. Veámoslo

Si hemos usado un IDE para la creación de las entities es posible que en las anotaciones aparezca algo como:

```
@OneToMany(cascade = CascadeType.ALL, mappedBy = "moneda")
private Collection ...
```

Con esto se consigue borrar en cascada y no hay error pero sin embargo, aplicar un remove() en cascada puede hacer que la lista de entities Historicambioeuro siga reflejando la existencia de la entity moneda que ya no existe. Con los consiguientes problemas en nuestro código.

El siguiente código nos funcionará independientemente de CascadeType (pensado para un id de moneda 12)

```
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();
Moneda m = em.find(Moneda.class, 12);
m.getHistoricocambioeuros()
    .stream()
    .forEach(h-> em.remove(h));

em.remove(m);
em.getTransaction().commit();
```

La parte importante del código anterior es eliminar tanto en monedas (la Entity Moneda) como en historicocambioeuros (la colección de Entity Historicocambioeuro) Todo dentro de la misma transacción

Al ser en la misma transacción el motor de persistencia se dará cuenta de lo que queremos hacer y borrará convenientemente dejando nuestras Entities en un estado consistente

En general es preferible que seamos nosotros quienes controlemos lo que ocurre en las diferentes tablas. Evitaremos el uso de CascadeType salvo que tengamos muy claro que conforman casi una unidad: Hospital – 1:N → Uci La desaparición (borrado) de un hospital implica el borrado de las unidades de cuidados intensivos (uci) que tenía el hospital

Veamos un ejemplo que realiza el borrado en Northwind. Queremos borrar un pedido (Order) y eso implica borrar también todos los detalles del pedido: (OrderDetail)

```
@Override
public boolean delete(Integer id) {
    EntityManager em = emf.createEntityManager();
    em.getTransaction().begin();
    Order order = em.find(Order.class, id);
    if(order != null) {
        if( order.getOrderDetails() != null) {
```

```

        OrderDetail[] array =
order.getOrderDetails().toArray(new
OrderDetail[order.getOrderDetails().size()]);
        for (OrderDetail orderDetail : array) {
            //orderDetail = em.merge(orderDetail);
            //orderDetail.setOrder(null);
            order.getOrderDetails().remove(orderDetail);
            em.remove(orderDetail);
        }
    }

    em.remove(order);
    em.getTransaction().commit();
}

em.close();
// TODO Auto-generated method stub
return false;
}

```

Observar que lo hacemos bajo una única transacción. Primero quitamos de RAM (de la lista de OrderDetails que tiene el pedido: Order.getOrderDetails()) el objeto OrderDetail. Después lo eliminamos de la base de datos: em.remove(orderDetail) y una vez hemos eliminado todos los orderdetail de la base de datos ya podemos borrar el pedido(Order) en la base de datos: em.remove(order)

Creación/modificación de objetos con relaciones

Pongamos el caso de una Moneda con id: 1 => moneda1 y los históricos con id: 3, id: 4 => historico3, historico4

Así si moneda1.historicos = [historico3, historico4]

también tiene que ocurrir que: historico3.moneda = moneda1, historico4.moneda = moneda1

Los pasos que seguiríamos son:

- Hacemos una transacción
- persistimos la entidad de cardinalidad 1: em.persist(moneda1)
- persistimos la entidad con cardinalidad N: em.persist(historico1) ; em.persist(historico2);
- hacemos commit

Es importante que la entidad de cardinalidad 1 tenga reflejada la lista de objetos de la entidad N y que la entidad de cardinalidad N tenga reflejada la entidad de cardinalidad 1:

```
Moneda moneda = new Moneda();
moneda.setNombre("una moneda");
moneda.setPais("un país");
Historicocambioeuro historico1 = new Historicocambioeuro();
historico1.setFecha(new java.util.Date());
historico1.setEquivalenteuro(0.23);
historico1.setMoneda(moneda);
ArrayList<Historicocambioeuro> lista = new ArrayList<Historicocambioeuro>();
lista.add(historico1);
moneda.setHistoricocambioeuros(lista);
em.getTransaction().begin();
em.persist(moneda);
em.persist(historico1);
em.getTransaction().commit();
```

Gestión de excepciones en las transacciones y uso de rollback

Cuando hacemos: `em.getTransaction().commit()` puede tener lugar una excepción (por ejemplo si guardamos una persona con el mismo dni (presuponiendo que hemos puesto que dni sea único). Se desencadena: `javax.persistence.RollbackException;` un rollback automático. Veamos un ejemplo de gestión:

```
public Alumno save(Alumno a) {
    EntityManager em = emf.createEntityManager();
    Alumno alumno = null;
    try {
        em.getTransaction().begin();
        alumno = new Alumno();
        alumno.setDni(a.getDni());
        alumno.setNombre(a.getNombre());
        alumno.setApellidos(a.getApellidos());
        alumno.setFechanacimiento(a.getFechanacimiento());
        em.persist(alumno);
        em.getTransaction().commit();
    } catch (RollbackException ex) {
        ex.printStackTrace();
        alumno = null;
    }
    return alumno;
}
```

Si en un momento dado necesitamos parar el proceso de transacción podemos desencadenar nosotros el rollback: `em.getTransaction().rollback();`

Creación de consultas – `createQuery()`

El API JPA proporciona la interfaz Query para configurar y ejecutar consultas. Podemos obtener una instancia que implemente esa interfaz mediante los métodos del entity manager: `createQuery()` y `createNamedQuery()`. El primer método se utilizar para crear una consulta dinámica y el segundo una consulta con nombre.

Una vez obtenida la consulta podemos pasárle los parámetros con `setParameter()` y ejecutarla. Se definen dos métodos para ejecutar consultas: el método `getSingleResult()` que devuelve un Object que es la única instancia resultante de la consulta y el método `getResultList()` que devuelve una lista de instancias resultantes de la consulta.

Vamos a ver un ejemplo:

```
Moneda m = em.find(Moneda.class,10);
String query = "select h from Historicocambioeuro h where h.moneda = :moneda";
List<Historicocambioeuro> list = em.createQuery(query,Historicocambioeuro.class)
    .setParameter("moneda", m)
    .getResultList();
```

Hablaremos con más detalle de JPQL pero la idea es trabajar con entidades. Observar que en la query hablamos de: `Historicocambioeuro` que es la Entity (la tabla era en minúsculas)

En la cláusula `from` declaramos un objeto-entity que representa cada fila para poder referenciarlo. Así:

from Historicocambioeuro h

Nos está diciendo que: `h` es una entity de `Historicocambioeuro`.

Por lo tanto el:

select h

Pretende obtener Entities de tipo `Historicocambioeuro`.

La parte del `where` nos dice:

where h.moneda = :moneda

Observar que accedemos a los atributos del objeto. Así h.moneda es la Entity Moneda que coincide con la foreign key para Historicocambioeuro. Luego vemos que se iguala a:

:moneda

con esto le estamos diciendo que queremos recibir un parámetro llamado: "moneda" (**siempre usaremos dos puntos: ":" para especificar el nombre del parámetro**)

Vemos que nosotros le pasamos esa información mediante:

```
.setParameter("moneda", m)
```

En definitiva, nos va a obtener los Historicocambioeuro que tengan como foreign key el objeto Moneda que le pasemos como parámetro. En concreto, la lista de Historicocambioeuro que sean para la moneda con id: 10

Eso significa que la consulta debiera coincidir con:

```
Moneda m = em.find(Moneda.class, 10);
List<Historicocambioeuro> list = m.getHistoricocambioeuros();
```

Ej:

Obtener los ids de la tabla historicocambioeuro para la moneda 1

```
Moneda m = em.find(Moneda.class, 1);

String query = "select h.idhistoricocambioeuro from Historicocambioeuro h where
h.moneda = :moneda";

List<Integer> list = em.createQuery(query, Integer.class)
    .setParameter("moneda", m)
    .getResultList();
```

Observar que los ids son de tipo Integer y por eso ponemos: Integer.class en el: em.createQuery()

JPQL

Ya hemos visto varios ejemplos vamos a ver teóricamente el lenguaje de consultas:

JPQL es el lenguaje en el que se construyen las queries en JPA. Aunque en apariencia es muy similar a SQL, en la realidad operan en mundos totalmente distintos. En JPQL las preguntas se construyen sobre clases y entidades, mientras que SQL opera sobre tablas, columnas y filas en la base de datos.

Una consulta JPQL puede contener los siguientes elementos

```
SELECT c
FROM Category c
WHERE c.categoryName LIKE :categoryName
ORDER BY c.categoryId
```

- Una cláusula SELECT que especifica el tipo de entidades o valores que se recuperan
- Una cláusula FROM que especifica una declaración de entidad que es usada por otras cláusulas
- Una cláusula opcional WHERE para filtrar los resultados devueltos por la query
- Una cláusula opcional ORDER BY para ordenar los resultados devueltos por la query
- Una cláusula opcional GROUP BY para realizar agregación
- Una cláusula opcional HAVING para realizar un filtrado en conjunción con la agregación

Ej:

Obtener el cambio medio entre la moneda de id1 y el Euro:

```
String query = "select avg(h.equivalenteeuro) from Historicocambioeuro h where
h.moneda = :moneda";

Double result = em.createQuery(query,Double.class)
.setParameter("moneda", em.find(Moneda.class, 1))
.getSingleResult();
```

Aquí vemos que se pueden usar funciones acumuladores como la media: avg()

Observar que no tenemos que hacer uso de los getter() hacemos referencia a: h.equivalenteeuro

Ej2:

Vamos a variar el anterior para hacer agrupamientos: Obtener el cambio medio para cada moneda respecto al euro:

```
String query = "select avg(h.equivalenteeuro) "
    + "from Historicocambioeuro h "
    + "group by h.moneda";

em.createQuery(query,Double.class)
    .getResultList()
    .stream()
    .forEach(System.out::println);
```

Observamos que soporta group by En esta ocasión hemos tenido que obtener un resultlist por ese motivo en lugar de singleresult

JPQL subquery

JPA también permite subconsultas en where y having. Las subconsultas deben ir dentro de paréntesis. Veamos un ejemplo:

```
String query = "select h.fecha "
    + "from Historicocambioeuro h "
    + "where h.moneda in ( "
    + "select m from Moneda m where m.nombre like 'dolar' "
    + ")";

em.createQuery(query, Date.class)
    .getResultList()
    .stream()
    .forEach(System.out::println);
```

La sentencia obtiene las fechas de datos cambiarios respecto al euro para todas las monedas dólar (dólar Usa, dólar Canadiense,...)

JPQL join

La ventaja es principalmente el uso de left y right. En general y para evitar problemas de versiones se puede hacer el producto cartesiano y filtrar en un where:

```
from Moneda m, Historicocambioeuro h
where ...
```

Ej. obtener las monedas a las que no se les ha apuntado ningún historico de tipos de cambio con el euro

```
String query = "select m "
+ "from Historicocambioeuro h "
+ "right join Moneda m "
+ "on h.moneda = m "
+ "where "
+ "h.idhistoricocambioeuro is null";

em.createQuery(query,Moneda.class)
.getResultList()
.stream()
.forEach(System.out::println);
```

Observar que el join habla de entidades en lugar de elementos primitivo

Traer entidades vinculadas en relaciones 1:N

Imaginemos que estamos en casos de fetch= Lazy que es el habitual. En algunas consultas podemos querer rellenar el array de objetos de la tabla con cardinalidad: N Para eso usaremos un join con fetch. El siguiente ejemplo obtiene un alumno por dni y si tiene matrículas se llenará su array de matriculas:

```
"select a from Alumno a left join fetch a.matriculas where a.dni = :dni"
```

Observar que al hacer left join también seleccionamos los alumnos sin matrículas. Pero si tiene matrículas, al haber usado la cláusula: fetch nos trae el array de matriculas:

fetch: a.matriculas

named query

La principal ventaja es que al no ser dinámica (se genera en tiempo de compilación) tiene un rendimiento mayor.

Adicionalmente si tenemos un sitio centralizado donde se realizan las diferentes consultas respecto a una misma entidad nos ayuda a que otro desarrollador no vaya a generar su propia query si ve que ya está entre las incluidas. Es más eficiente en cuanto a localización y centralización de las queries

Native query

JPQL únicamente abarca a un subconjunto de lo que se puede alcanzar con SQL Así aún precisaremos ejecutar consultas SQL directamente. Para eso tenemos native query. Veamos un ejemplo y explicamos su funcionamiento:

Buscamos los alumnos que se han matriculado en alguna ocasión:

```
EntityManager em = emf.createEntityManager();
String query="select alumnos.* from alumnos inner join matriculas on
matriculas.dni = alumnos.dni";
List<Alumno> rl = em.createNativeQuery(query, Alumno.class).getResultList();
for (Alumno object : rl) {
    System.out.println(object);
}
em.close();
```

Vemos que a createNativeQuery() se le pasa la consulta. Adicionalmente se le puede pasar un segundo campo que es el tipo de dato que devuelve (en este caso Alumno) si no damos esa segunda información devuelve: List<Object []> esto es, una lista en la que cada elemento es una fila de la consulta. Y cada fila de la consulta es un array de Object

Capa Repositorio y relación con DAO

Repository y DAO son dos patrones conceptualmente diferentes. Erick Evans (uno de los autores que desarrolló la idea) dice sobre Repository:

“Para cada tipo de objeto que necesite acceso global, un Repositorio crea la ilusión de una colección en memoria de todos los objetos de ese tipo. Configura el acceso a través de una interfaz global bien conocida. Provee métodos para **agregar y remover objetos**. (...) Provee métodos que **seleccionan** objetos basados **en algún criterio** y que devuelvan objetos o colecciones de objetos completamente instanciados cuyos valores de atributos cumplen los criterios”

Observar la parte que habla de criterios. Así por ejemplo, para el caso de facturas, un posible método de repositorio podría ser: getFacturasSinCobrar()

Adicionalmente a la anterior definición, **los repositorios son una capa más alta. Un repositorio puede basarse en DAOs pero no puede ocurrir al revés, un DAO no se puede basar en repositorios.** Los DAO están más pegados a la base de persistencia (bases de datos,...)

Pongamos el caso de que tenemos información remota y nosotros queremos mantener una caché. De tal forma que no dependamos tanto de la conexión externa. Ahí tendremos dos orígenes de información, no es simplemente un DAO que consulte a una base de datos. Se debe decidir cuándo se va a la fuente de datos externa o cuando se toma la información local. Ese es un caso claramente de un repositorio. El ejemplo descrito lo podemos ver en android developers: <https://developer.android.com/jetpack/guide/data-layer?hl=es-419>

Una vez vistas las definiciones procede etiquetar que es un repositorio y qué no lo es. Cuándo usamos algunos frameworks ORM , son tan completos que ya incluyen criterios, y pudiera considerarse que por tanto deban llamarse repositorios en el sentido que le dieron inicialmente Fowler/Evans.

En general, nosotros usaremos la capa repositorio para aislar los objetos de nuestra lógica de negocio de circunstancias como consultar dos bases de datos distintas (una local y otra remota) decidiendo en cada caso a cuál acceder. O para tener objetos que realmente se componen de información almacenada en bases de datos distintas. Pero para ir introduciendo la idea de Repository, la vamos a introducir desde ya, en el siguiente ejemplo, aunque podría ser discutible si debiera llamarse DAO en lugar de Repository. Para mantener coherencia con el framework Spring.

```

public class MonedaRepository implements ICrud<Moneda, Integer>{

    private EntityManagerFactory emf;

    public MonedaRepository(EntityManagerFactory emf) {
        this.emf = emf;
    }

    @Override
    public List<Moneda> findAll() {
        EntityManager em = emf.createEntityManager();
        em.getTransaction().begin();
        List<Moneda> lista = em.createNamedQuery("Moneda.findAll", Moneda.class)
            .getResultList();
        em.getTransaction().commit();
        em.close();
        return lista;
    }

    @Override
    public Moneda findById(Integer id) {
        EntityManager em = emf.createEntityManager();
        em.getTransaction().begin();
        Moneda Moneda = em.find(Moneda.class, id);
        em.getTransaction().commit();
        em.close();
        return Moneda;
    }

    public Moneda findByName(String nombre) {
        EntityManager em = emf.createEntityManager();
        em.getTransaction().begin();
        Moneda Moneda = em.createNamedQuery("Moneda.findByName", Moneda.class)
            .setParameter("name", nombre)
            .getSingleResult();

        em.getTransaction().commit();
        em.close();
        return Moneda;
    }
}

```

Anexo: Tests con maven surefly

En el pom.xml agregamos las dependencias:

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>5.8.1</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.8.1</version>
    <scope>test</scope>
</dependency>
```

En los plugins de build agregamos surefly:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>2.22.0</version>
        </plugin>
    </plugins>
</build>
```

Como siempre, nuestros test estarán en: src/test/java

Surefly espera localizar las clases con el prefijo o sufijo: Test. Así por ejemplo: `AlumnoRepositoryTest`

Tener cuidado con los imports, que sean de jupiter. Así para assertNull etc nos quedarían los siguientes:

```
import static org.junit.jupiter.api.Assertions.assertNull;
import static org.junit.jupiter.api.Assertions.assertNotEqual;
import static org.junit.jupiter.api.Assertions.assertTrue;
```

Si queremos establecer un orden en los test tenemos que usar la anotación: `@TestMethodOrder` veamos un trozo de ejemplo que testea el repositorio Alumno:

```
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
public class AlumnoRepositoryTest {

    /**
     * @throws java.lang.Exception
     */
    @BeforeAll
    static void setUpBeforeClass() throws Exception {
        EMFSingleton emfSingleton = EMFSingleton.getSingleton();
        EntityManagerFactory emf = emfSingleton.getEmf();
        alumnoRepository = new AlumnoRepository(emf);
    }

    static AlumnoRepository alumnoRepository;
    /**
     * @throws java.lang.Exception
     */
    @AfterAll
    static void tearDownAfterClass() throws Exception {
    }

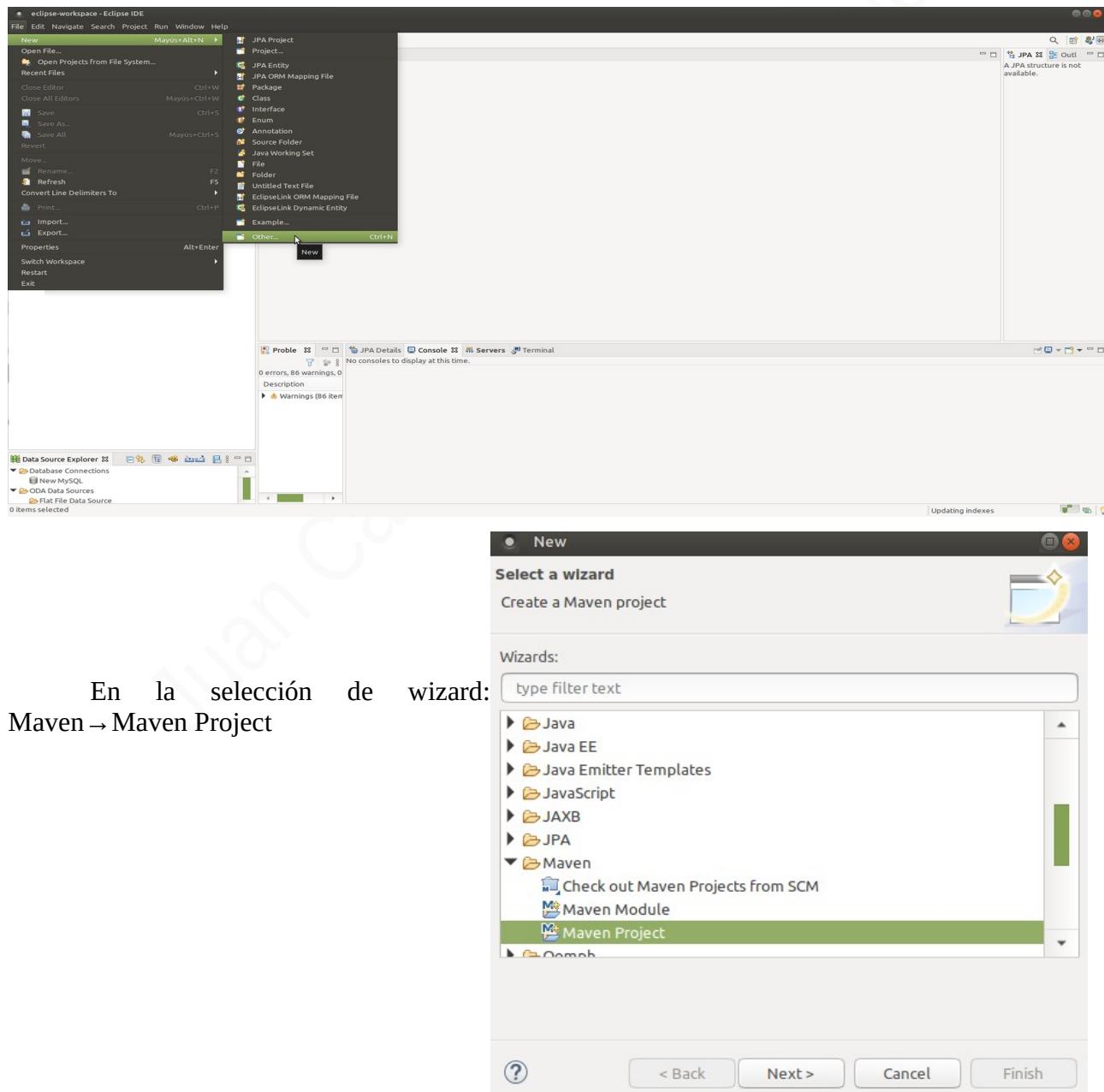
    /**
     * Test method for {@link
     es.iespuertodelacruz.jc.instituto.repository.AlumnoRepository#findById(java.lang.String)}
     */
    @Test()
    @Order(2)
    void testFindById() {
    }
}
```

Anexo: Paso a paso Proyecto maven-eclipse-jpa (ya sea web o no) con mysql

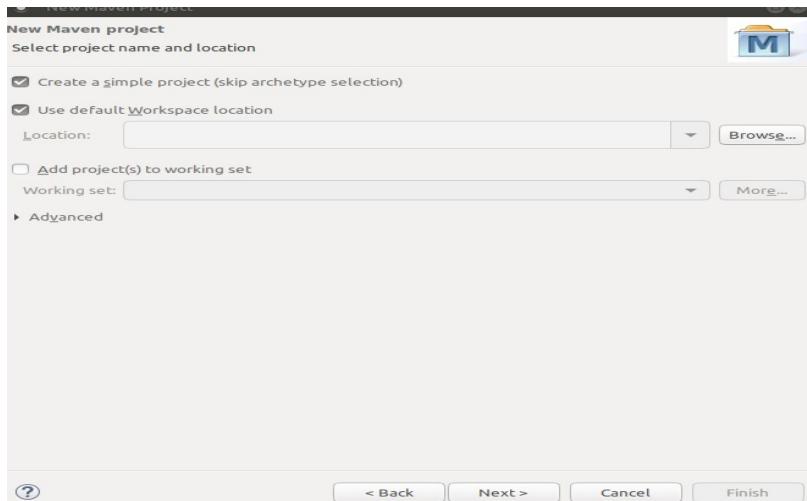
Nota: Los primeros pasos son para un **proyecto maven simple**. Saltar y hacer procedimiento web si correspondiera

Caso proyecto no web, simple maven (sin archetype)

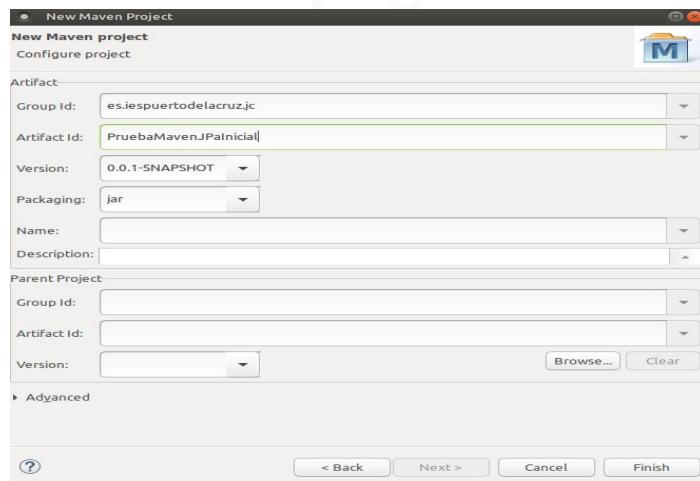
Creamos un proyecto maven en eclipse: File->New->Other



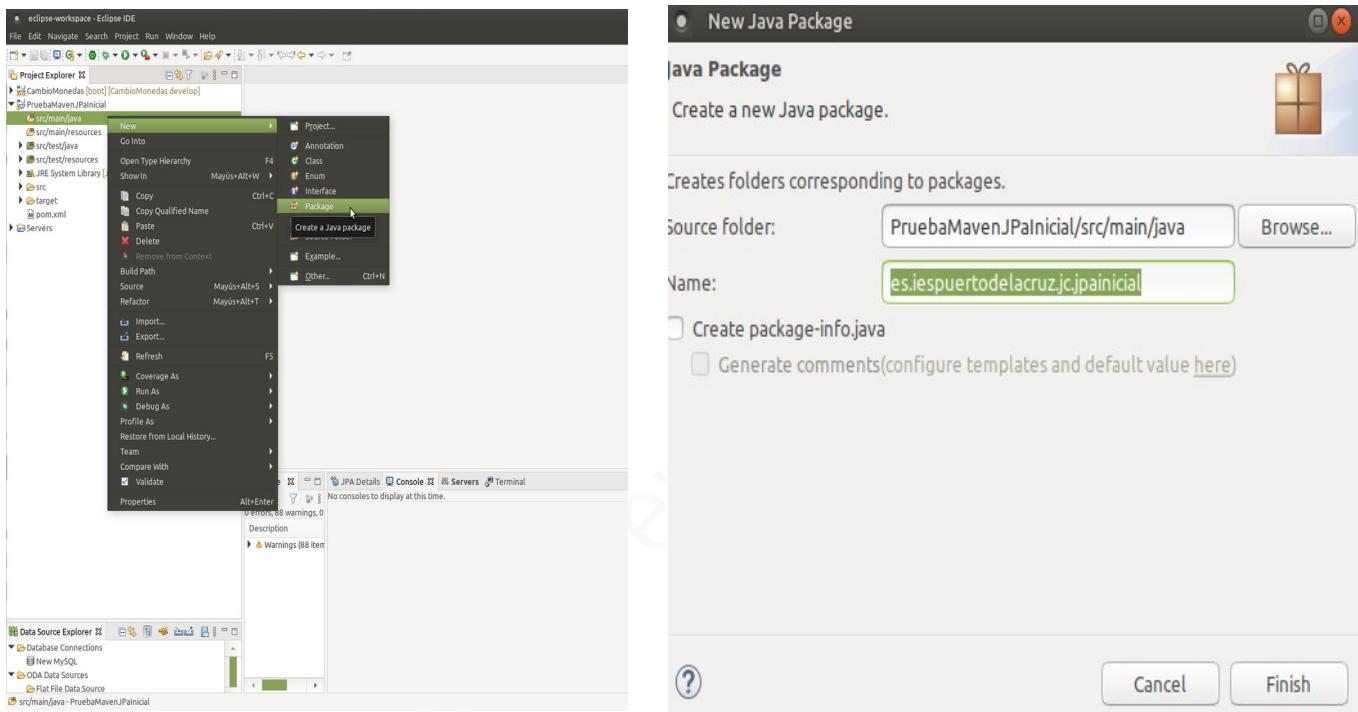
Marcamos el checkbox: Create a simple project (skip archetype selection)



Damos los datos maven de nuestro proyecto (group id y Artifact id) donde Artifact id es el nombre del proyecto en sí mismo



Creamos el paquete raíz a partir del cual vamos a trabajar. Nos ponemos encima de: src/main/java y le damos a agregar un paquete:



Vamos a establecer una clase con un método main (en este ejemplo que no tiene mayor interés se llamará: Main)

The screenshot shows the Eclipse IDE interface. The title bar reads "eclipse-workspace - PruebaMavenJPainicial/src/main/java/es/iespuertodelacruz/jc/jpainicial/Main.java - Eclipse IDE". The menu bar includes File, Edit, Source, Refactor, Navigate, Search, Project, Run, Window, and Help. The toolbar has various icons for file operations like Open, Save, Cut, Copy, Paste, and Find. The Project Explorer view on the left shows a project named "PruebaMavenJPainicial" with a sub-project "CambioMonedas [boot] [CambioMonedas develop]". Inside "PruebaMavenJPainicial", there are folders for "src/main/java" containing "es.iespuertodelacruz.jc.jpainicial" and "src/main/resources", and "src/test/java", "src/test/resources", and "target" with "pom.xml". The JRE System Library is set to "J2SE-1.5". The right-hand side shows the code editor for "Main.java" with the following content:

```
1 package es.iespuertodelacruz.jc.jpainicial;
2
3 public class Main {
4
5     public static void main(String[] args) {
6         // TODO Auto-generated method stub
7
8     }
9
10}
11
```

Poner dependencias Maven para JPA (caso no web jdk1.8)

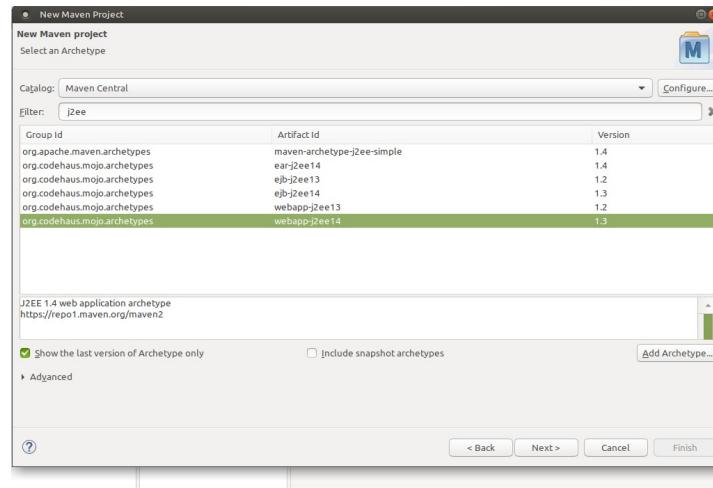
Ahora vamos a agregar al pom.xml que nos ha generado la información de mysql, hibernate:

pom.xml

```
<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.iespuertodelacruz.jc</groupId>
  <artifactId>PruebaMavenJPaInicial</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <properties>
    <maven.compiler.target>1.8</maven.compiler.target>
    <maven.compiler.source>1.8</maven.compiler.source>
  </properties>
  <dependencies>
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>8.0.20</version>
    </dependency>
    <!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-core -->
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-core</artifactId>
      <version>5.4.3.Final</version>
    </dependency>
    <!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-entitymanager -->
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-entitymanager</artifactId>
      <version>5.4.3.Final</version>
    </dependency>
    <!-- https://mvnrepository.com/artifact/javax.persistence/javax.persistence-api -->
    <dependency>
      <groupId>javax.persistence</groupId>
      <artifactId>javax.persistence-api</artifactId>
      <version>2.2</version>
    </dependency>
  </dependencies>
</project>
```

Poner dependencias Maven para JPA (caso web jdk1.8)

Para crear el proyecto Java Web maven se recomienda al elegir la parte de archetype en eclipse elegir: webapp-j2ee14



En el pom.xml aparte de las dependencias para el archetype webapp-j2ee14, hay que subir la versión de java que viene puesta por defecto. Esto lo hacemos en la parte de plugins:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.6.0</version>
      <configuration>
        <source>1.8</source>
        <target>1.8</target>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Ahora veremos la parte de las dependencias de ese fichero pom.xml

```

<dependencies>
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>3.1.0</version>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>javax.servlet.jsp</groupId>
    <artifactId>javax.servlet.jsp-api</artifactId>
    <version>2.3.0</version>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>javax.el</groupId>
    <artifactId>javax.el-api</artifactId>
    <version>3.0.0</version>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jstl</artifactId>
    <version>1.2</version>
</dependency>
<dependency>
    <groupId>taglibs</groupId>
    <artifactId>standard</artifactId>
    <version>1.1.2</version>
</dependency>

<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.26</version>
</dependency>

<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.12.2</version>
</dependency>

<dependency>
    <groupId>commons-dbcp</groupId>
    <artifactId>commons-dbcp</artifactId>
    <version>1.4</version>
</dependency>

<dependency>
<groupId>org.hibernate</groupId>
<artifactId>hibernate-core</artifactId>
<version>5.4.3.Final</version>
</dependency>
<dependency>
<groupId>org.hibernate</groupId>
<artifactId>hibernate-entitymanager</artifactId>
<version>5.4.3.Final</version>
</dependency>

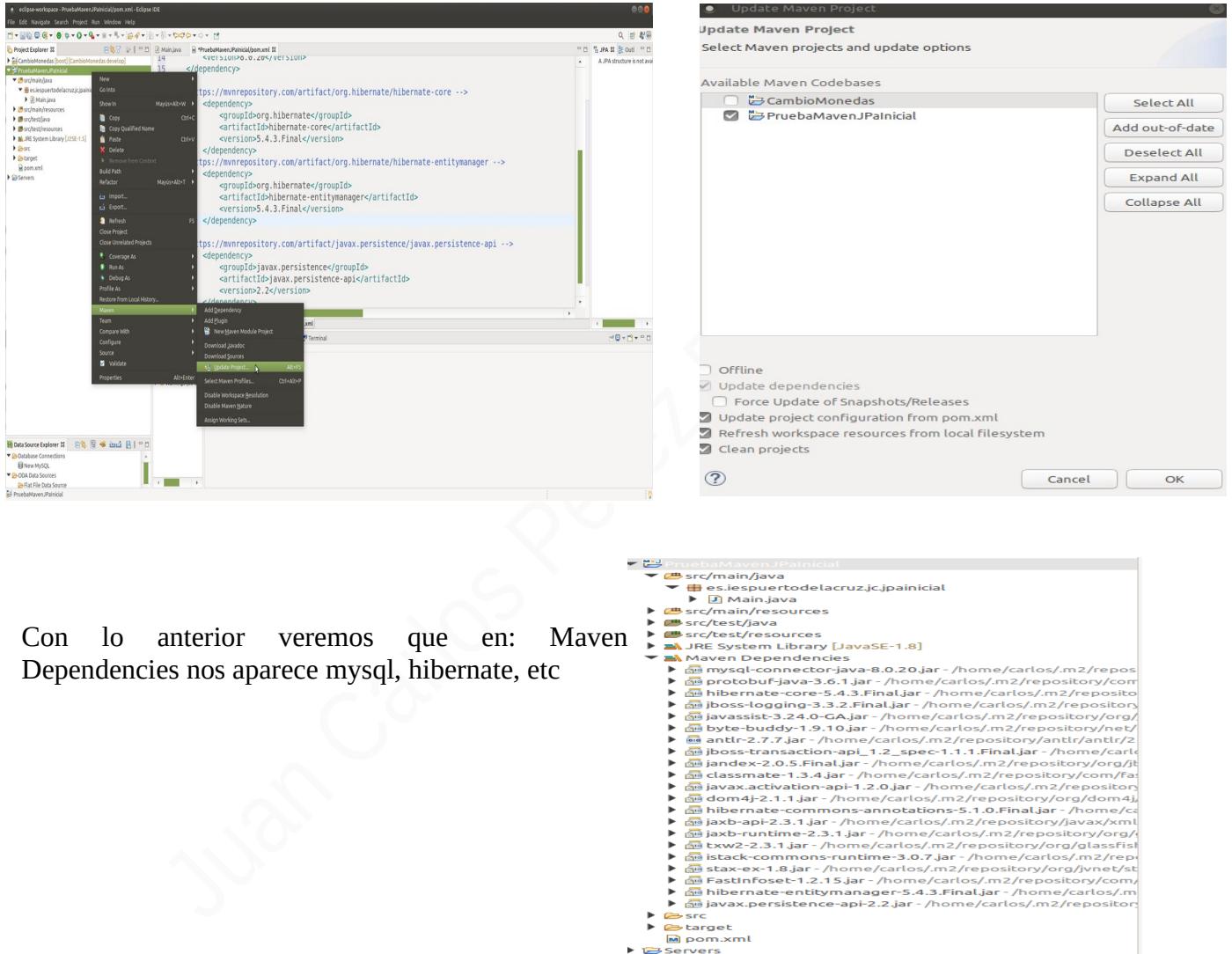
<dependency>
<groupId>javax.persistence</groupId>
<artifactId>javax.persistence-api</artifactId>
<version>2.2</version>
</dependency>
</dependencies>

```

Actualizar proyecto con dependencias nuevas

Para que actualice correctamente le decimos a eclipse que actualice Maven

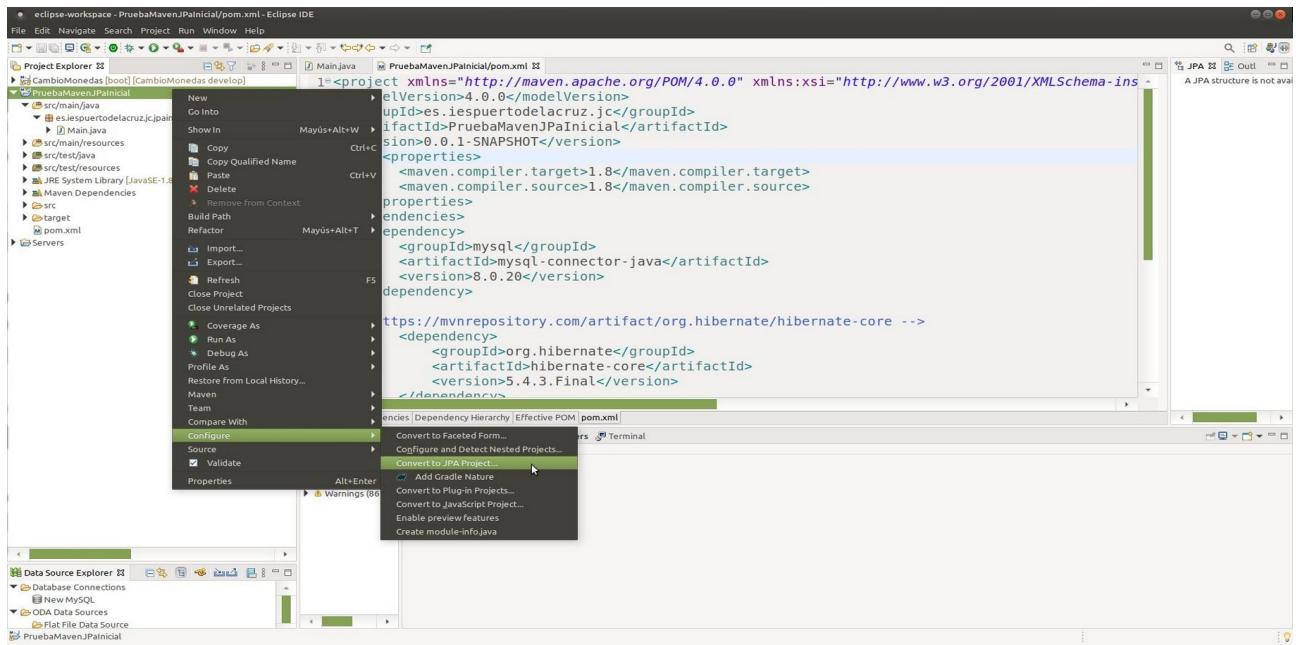
Para ello nos posicionamos sobre el proyecto botón derecho → Maven → update project



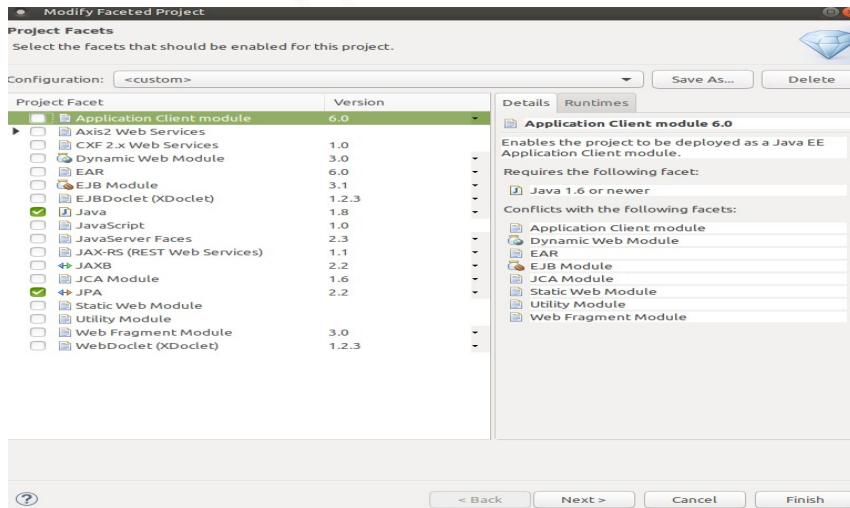
Con lo anterior veremos que en: Maven Dependencies nos aparece mysql, hibernate, etc

Convertir proyecto en JPA

Ahora para la generación automática de las Entities vamos a convertir el proyecto en un proyecto JPA: Botón derecho sobre proyecto → Configure → Convert to jpa project

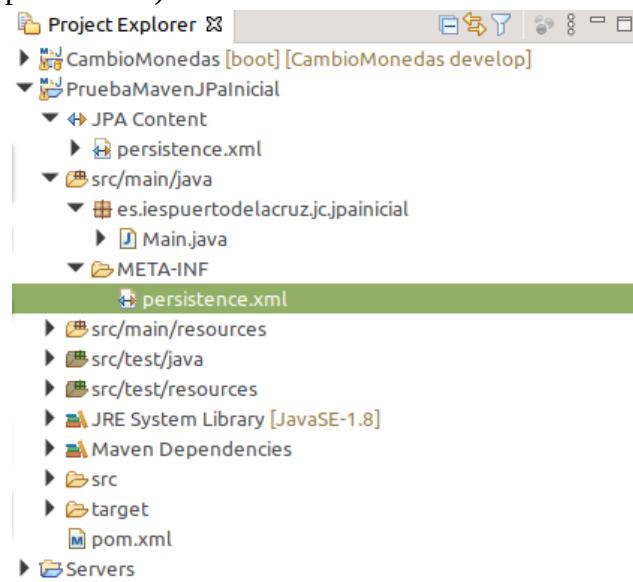


En la ventana seleccionamos que esté seleccionado JPA

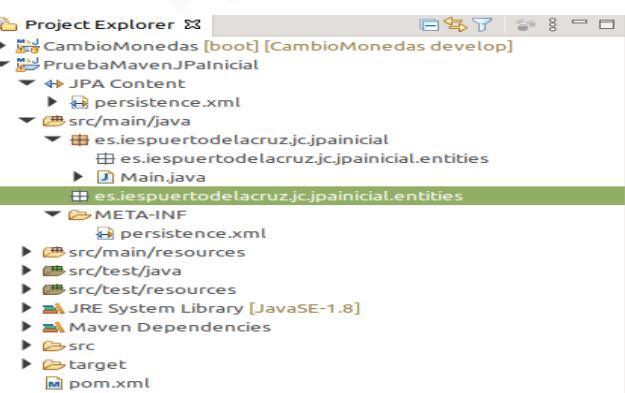


Pulsamos en el botón finish (no hace falta pulsar next)

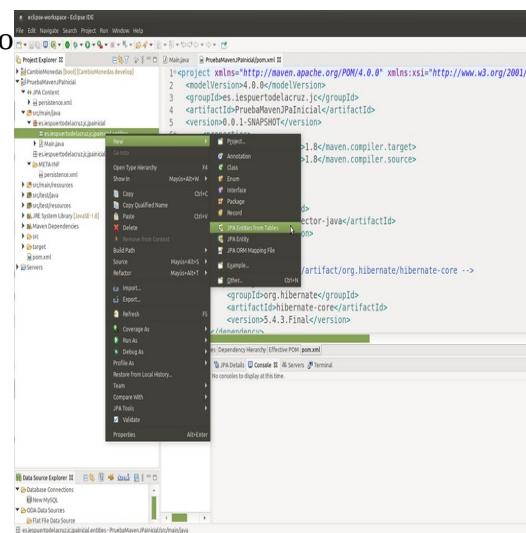
Nos habrá creado un fichero persistence.xml



Ahora crearemos el paquete donde vamos a poner nuestras entities. Por ejemplo:
es.iespuertodelacruz.jc.jpainicial.entities



Nos ponemos encima del paquete, botón derecho
→ New → JPA entities from tables

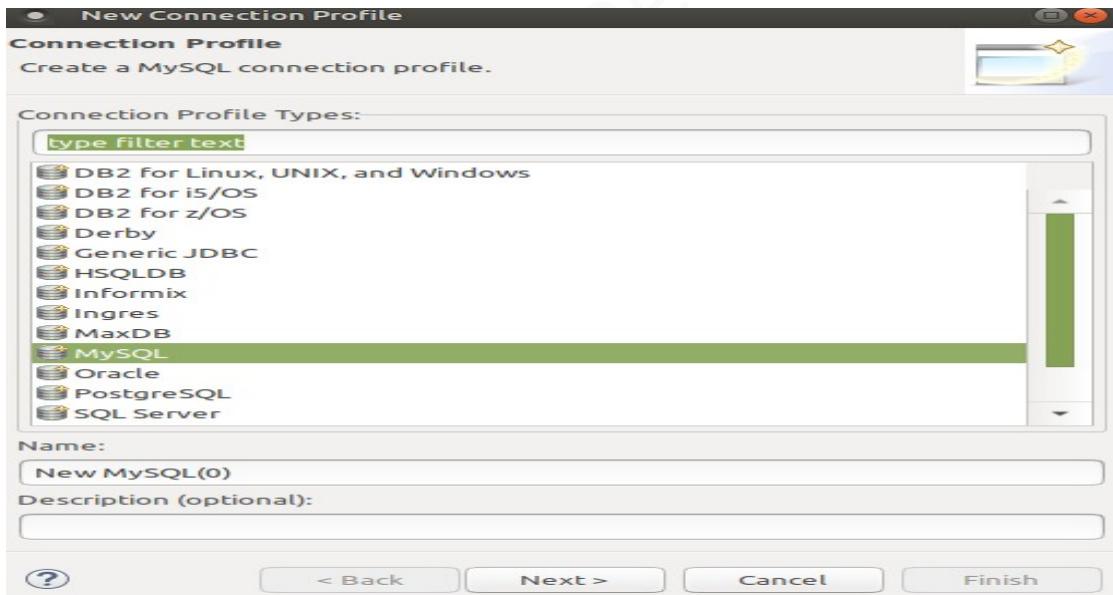


Nos aparece un wizard donde habrá que seleccionar una conexión. Como no tenemos aún ninguna pulsamos sobre el icono amarillo de nueva conexión (observar que lo hemos rodeado con un círculo para detectarlo fácilmente)

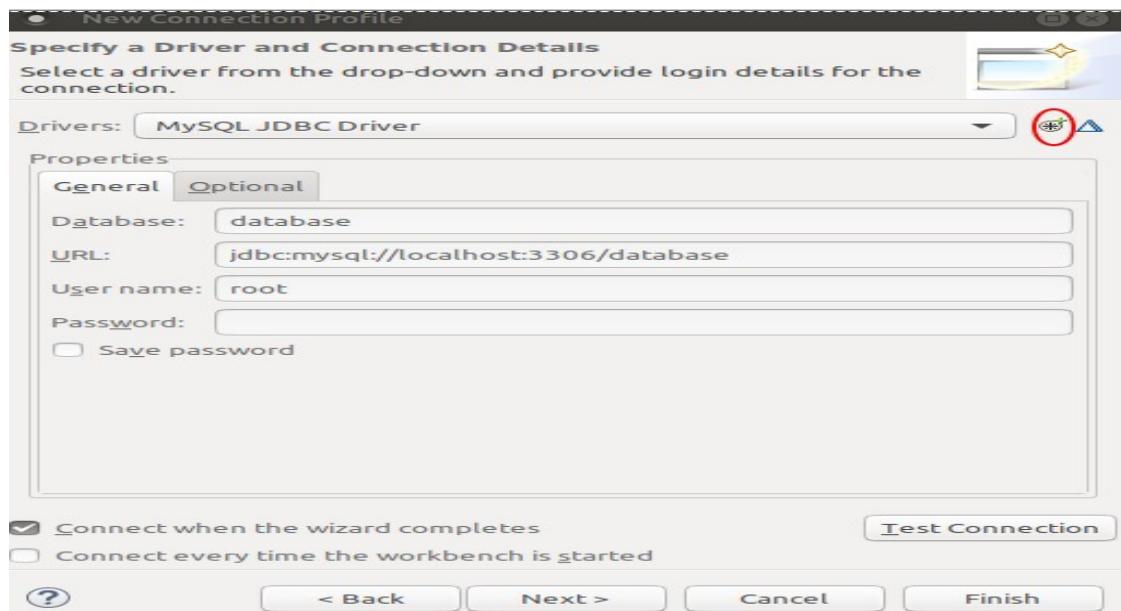
nota: cuidado! Si no tenemos arrancado el servicio de mysql no va a terminar conectando nuestro proyecto



Abre un nuevo wizard para crear la nueva conexión. Seleccionamos Mysql y pulsamos next

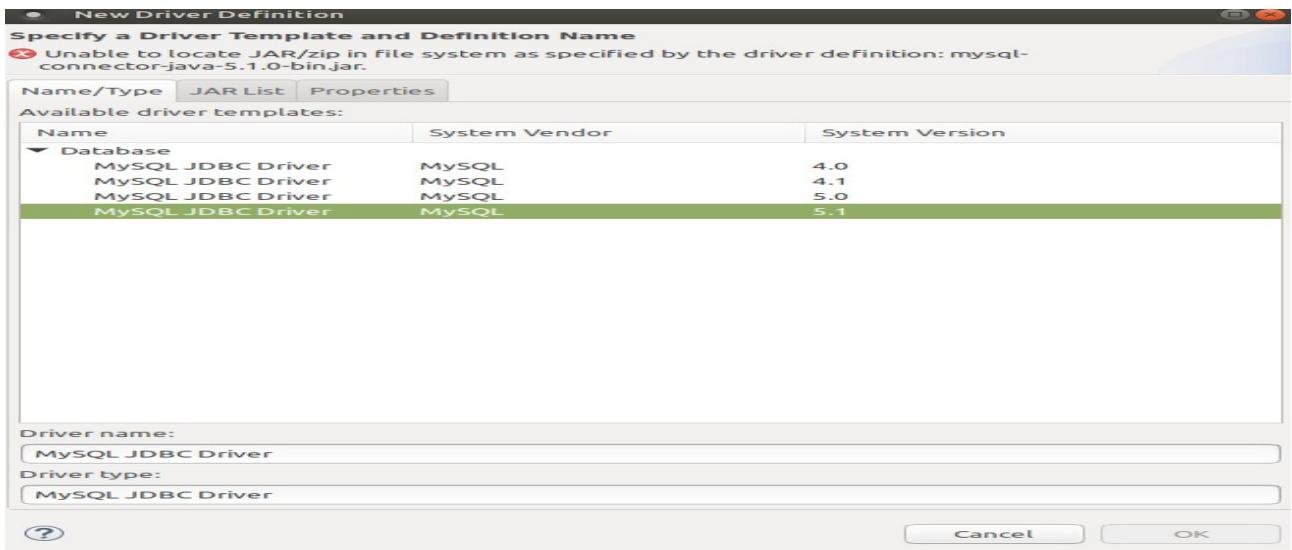


En la siguiente ventana del wizard se detallarán las especificidades de la conexión (nombre de la base de datos, usuario, etc) pero adicionalmente nosotros vamos a tener que hacer un paso más ya que aún no hemos cargado un driver mysql para que Eclipse pueda conectar con la base de datos

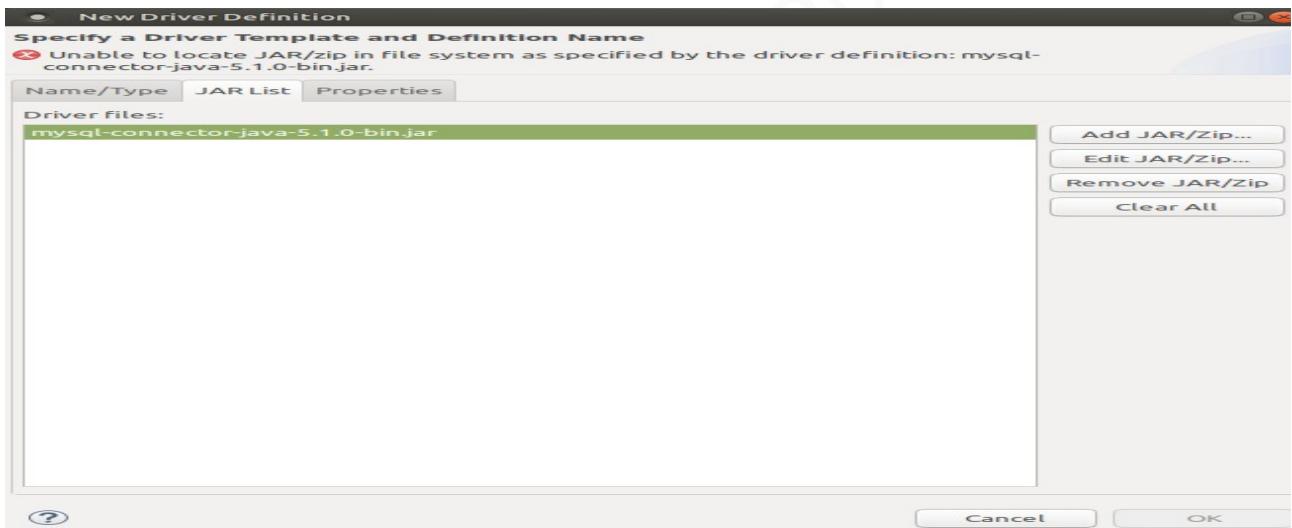


Se ha marcado con un círculo rojo el botón que hay que pulsar para decirle a eclipse donde tenemos el driver para que se pueda conectar a la base de datos (en contextual el icono dice: New driver definition)

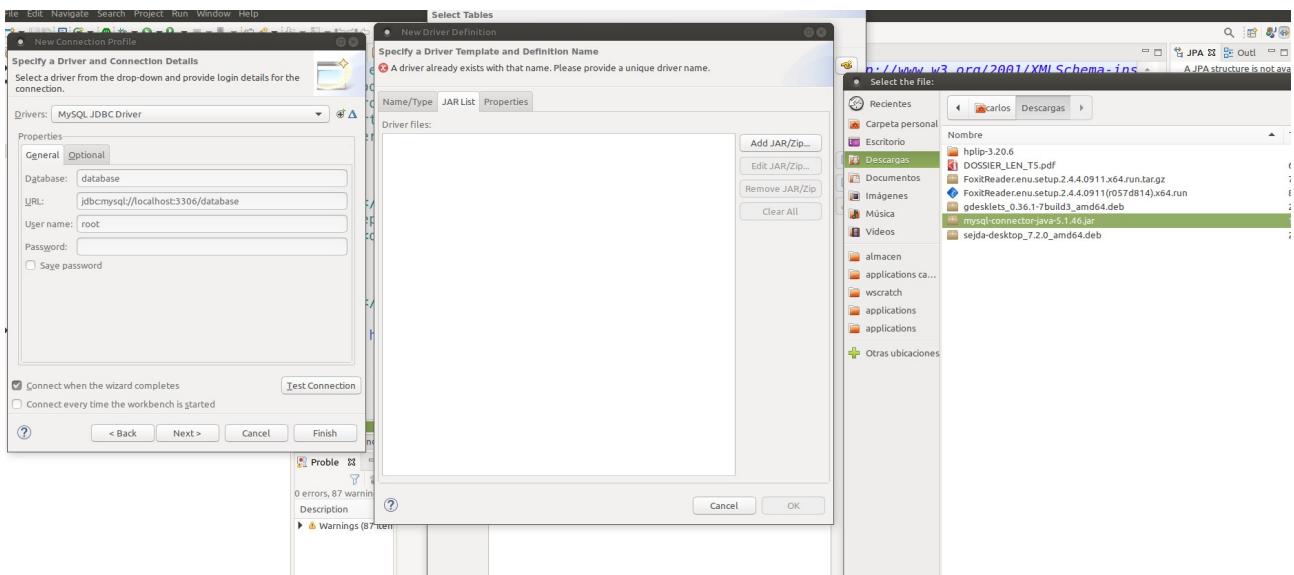
Al pulsar en el botón nos aparece un nuevo wizard. En la primera pestaña seleccionamos la versión 5.1 del driver



Luego pulsamos sobre la pestaña: Jar List y borramos el driver que tiene establecido:



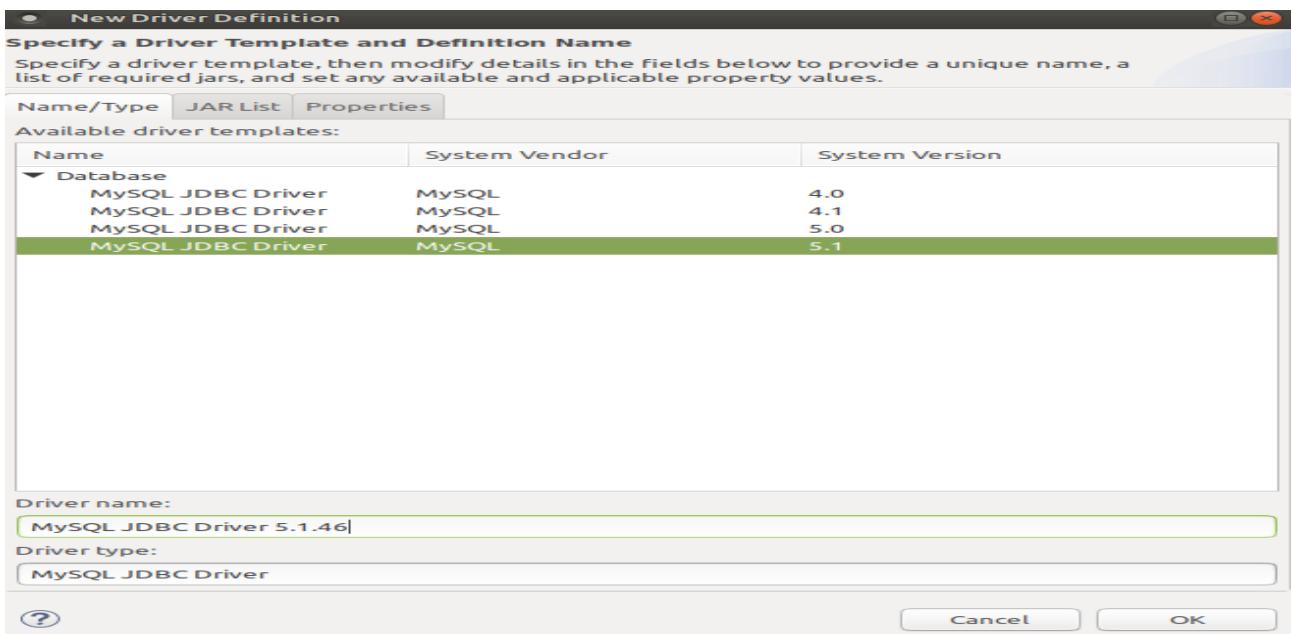
Después agregamos el driver que tengamos descargado en el disco duro. Para eso pulsamos en el botón: Add Jar/Zip y buscamos la ubicación del driver (en nuestro caso la versión 5.1.46)



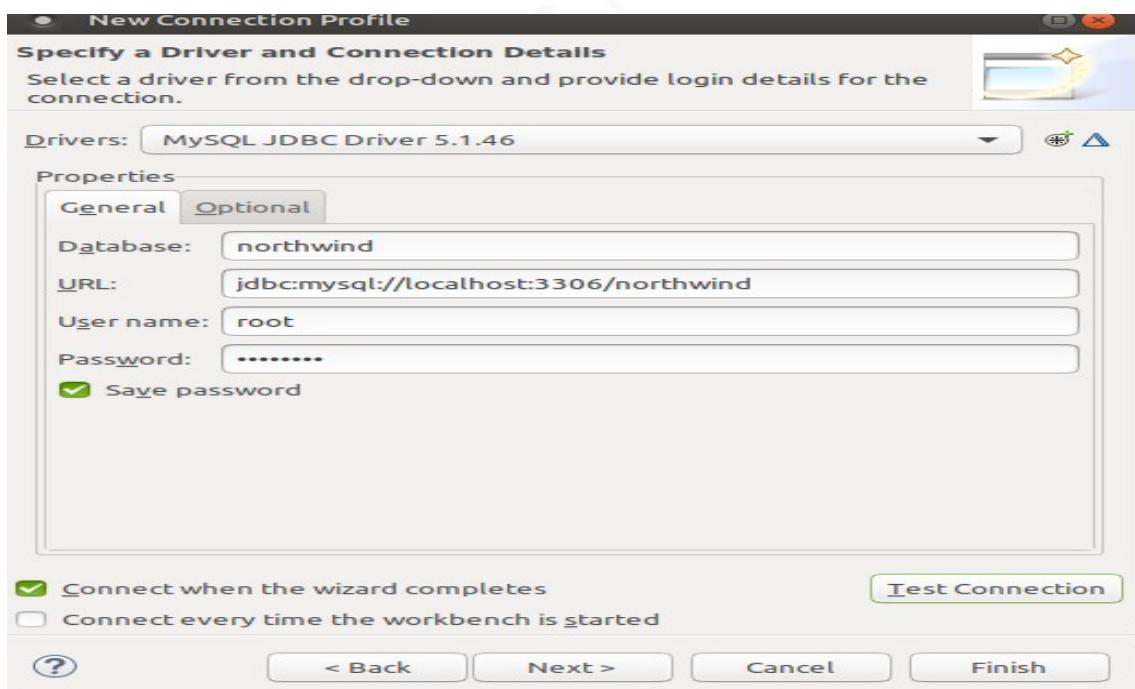
Ahora la pestaña Jar List nos quedará:



Observar que no nos permitirá pulsar en OK si no le hemos puesto un nombre correcto al Driver. En Este caso se muestra un ejemplo con el nombre 5.1.46:



Una vez cargado el driver para que eclipse se conecte terminamos de llenar los datos de conexión en el wizard. Vamos a ver un ejemplo de conexión a base de datos northwind en local (Mysql está instalado en el mismo ordenador que tenemos eclipse)



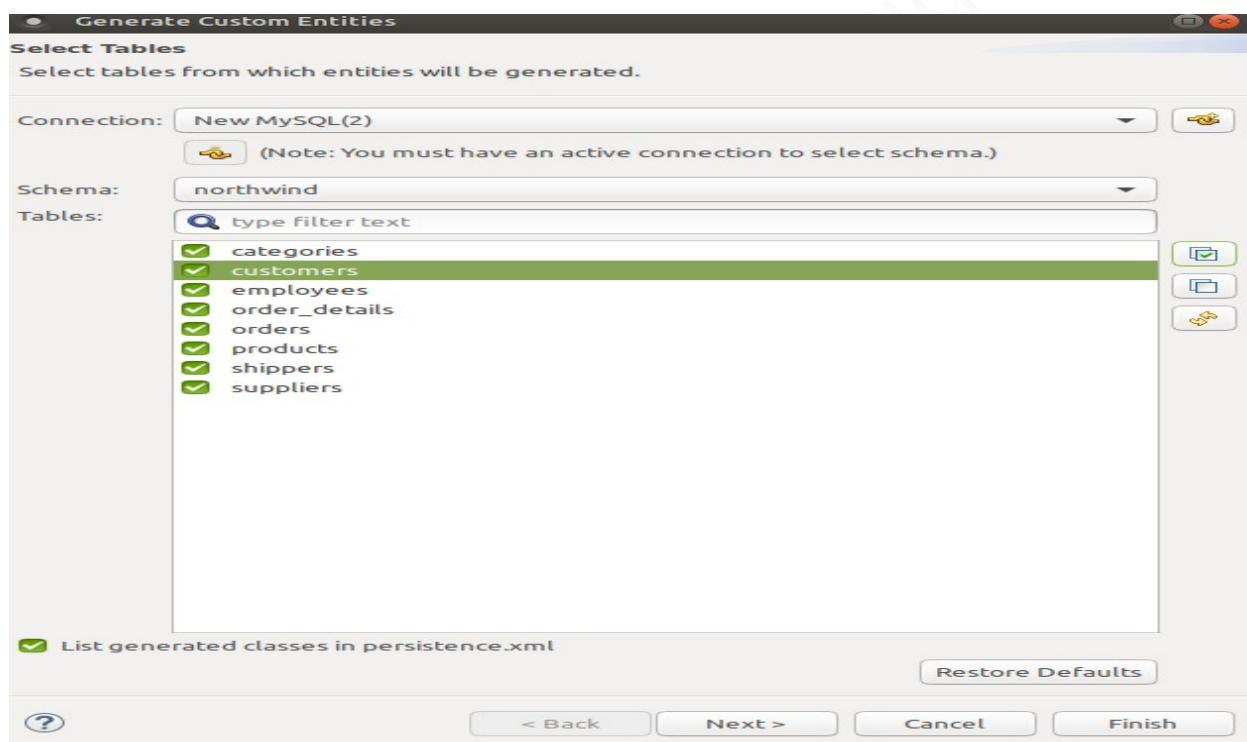
Una vez establecido no es necesario pulsar en next. Basta con pulsar en: Finish

Con lo anterior se cierra el wizard y nos devuelve al wizard inicial que era el de generación de entities desde la base de datos.

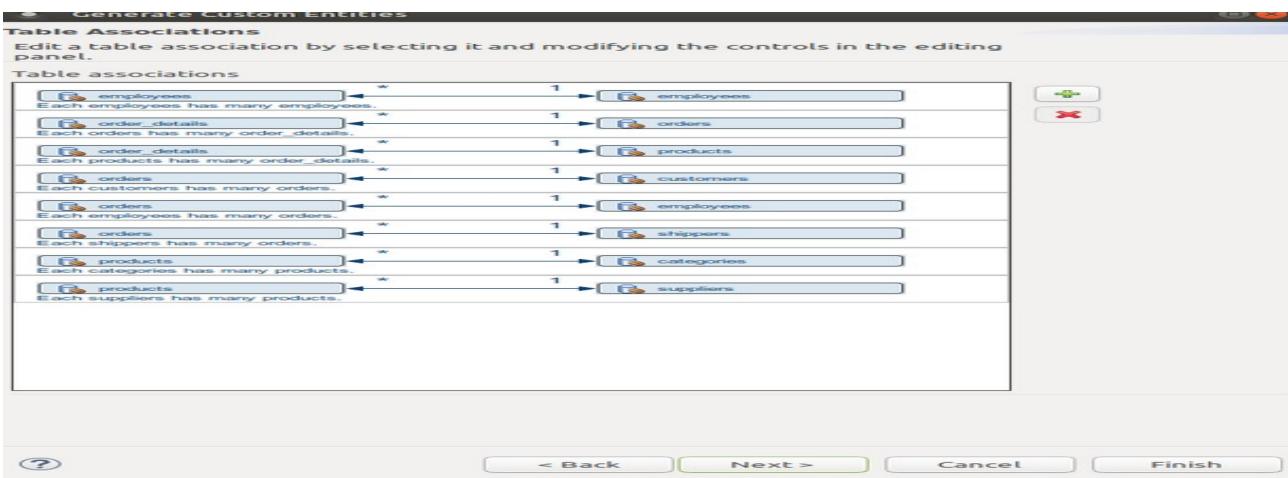
Tenemos que tener seleccionado en: Connection la conexión que hemos creado hace un momento (en el ejemplo se llamó New Mysql(2))

Tenemos que tener seleccionado en: Schema la base de datos que queremos atacar. En este caso de ejemplo: northwind

Nos mostrará las diferentes tablas de la base de datos. Nosotros seleccionaremos aquellas que queramos incluir. En este ejemplo las tomaremos todas



Avanzamos el wizard con next:



En esa ventana no hay nada que hacer ahora y pulsamos en next:

Esta ventana tiene varias opciones importantes. Como sabemos que northwind tiene autonuméricos vamos a seleccionar en: Key generator la opción: identity

También vemos que podemos elegir entre Eager y Lazy eso es algo importante pero ahora no lo vamos a tocar

De igual forma también podemos elegir entre listas y conjuntos (list, set) de momento también lo dejamos todo como viene y pulsamos en: Finish

Ahora podemos observar que nos ha creado las Entities:



Para que todo funcione correctamente nos falta agregar un par de cosas en el fichero **persistence.xml**

Después de los wizard nos mostrará algo parecido a:

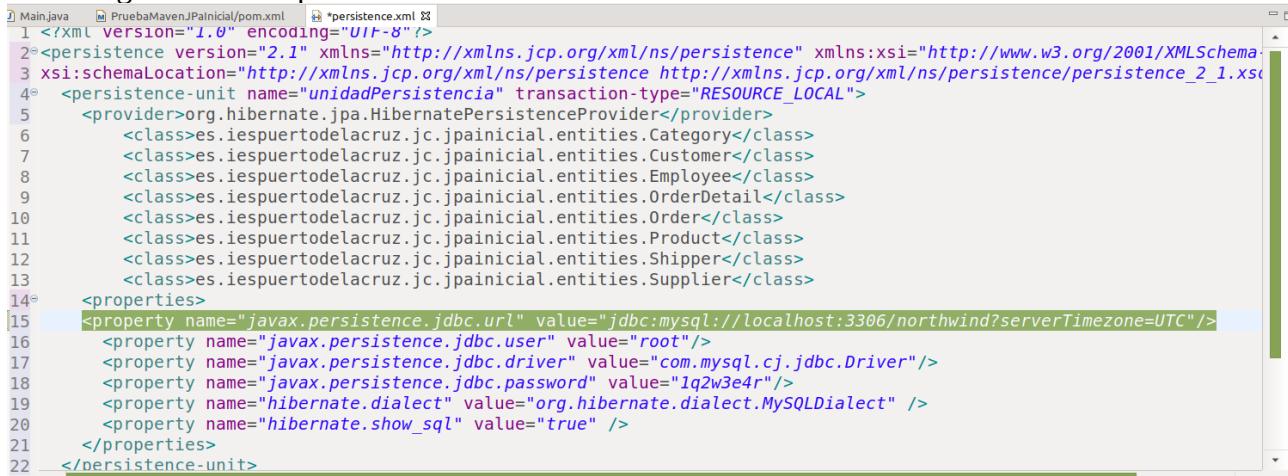
```
<persistence version="2.2" xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<persistence-unit name="PruebaMavenJPaInicial">
<class>es.iespuertodelacruz.jc.jpainicial.entities.Category</class>
<class>es.iespuertodelacruz.jc.jpainicial.entities.Customer</class>
<class>es.iespuertodelacruz.jc.jpainicial.entities.Employee</class>
<class>es.iespuertodelacruz.jc.jpainicial.entities.OrderDetail</class>
<class>es.iespuertodelacruz.jc.jpainicial.entities.Order</class>
<class>es.iespuertodelacruz.jc.jpainicial.entities.Product</class>
<class>es.iespuertodelacruz.jc.jpainicial.entities.Shipper</class>
<class>es.iespuertodelacruz.jc.jpainicial.entities.Supplier</class>
</persistence-unit>
</persistence>
```

Nosotros le vamos a agregar la información de hibernate y los datos de conexión:

En este caso que nos conectamos a la propia máquina (localhost) la base da datos northwind y le ponemos el parámetro de serverTimezone:

```
<property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/northwind?serverTimezone=UTC"/>
```

Una imagen de como quedaría es:



The screenshot shows a code editor with the file "persistence.xml" open. The XML code defines a persistence unit named "unidadPersistencia" using the RESOURCE_LOCAL transaction type. It lists several entity classes: Category, Customer, Employee, OrderDetail, Order, Product, Shipper, and Supplier. The properties section specifies the JDBC URL as "jdbc:mysql://localhost:3306/northwind?serverTimezone=UTC", the user as "root", the driver as "com.mysql.cj.jdbc.Driver", and the password as "1q2w3e4r". The dialect is set to "org.hibernate.dialect.MySQLDialect" and show_sql is set to "true".

```
<?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="unidadPersistencia" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
    <class>es.iespuertodelacruz.jc.jpainicial.entities.Category</class>
    <class>es.iespuertodelacruz.jc.jpainicial.entities.Customer</class>
    <class>es.iespuertodelacruz.jc.jpainicial.entities.Employee</class>
    <class>es.iespuertodelacruz.jc.jpainicial.entities.OrderDetail</class>
    <class>es.iespuertodelacruz.jc.jpainicial.entities.Order</class>
    <class>es.iespuertodelacruz.jc.jpainicial.entities.Product</class>
    <class>es.iespuertodelacruz.jc.jpainicial.entities.Shipper</class>
    <class>es.iespuertodelacruz.jc.jpainicial.entities.Supplier</class>
  <properties>
    <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/northwind?serverTimezone=UTC"/>
    <property name="javax.persistence.jdbc.user" value="root"/>
    <property name="javax.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver"/>
    <property name="javax.persistence.jdbc.password" value="1q2w3e4r"/>
    <property name="hibernate.dialect" value="org.hibernate.dialect.MySQLDialect" />
    <property name="hibernate.show_sql" value="true" />
  </properties>
  </persistence-unit>
</persistence>
```

Para poder copiar lo vemos como texto el fichero persistence.xml

```
<?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="unidadPersistencia" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
    <class>es.iespuertodelacruz.jc.jpainicial.entities.Category</class>
    <class>es.iespuertodelacruz.jc.jpainicial.entities.Customer</class>
    <class>es.iespuertodelacruz.jc.jpainicial.entities.Employee</class>
    <class>es.iespuertodelacruz.jc.jpainicial.entities.OrderDetail</class>
    <class>es.iespuertodelacruz.jc.jpainicial.entities.Order</class>
    <class>es.iespuertodelacruz.jc.jpainicial.entities.Product</class>
    <class>es.iespuertodelacruz.jc.jpainicial.entities.Shipper</class>
    <class>es.iespuertodelacruz.jc.jpainicial.entities.Supplier</class>
  <properties>
    <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/northwind?serverTimezone=UTC"/>
    <property name="javax.persistence.jdbc.user" value="root"/>
    <property name="javax.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver"/>
    <property name="javax.persistence.jdbc.password" value="1q2w3e4r"/>
    <property name="hibernate.dialect" value="org.hibernate.dialect.MySQLDialect" />
    <property name="hibernate.show_sql" value="true" />
  </properties>
  </persistence-unit>
</persistence>
```

Hemos marcado en amarillo aquellas cosas que nosotros podemos modificar habitualmente

Por ejemplo, se ha elegido ponerle el nombre: unidadPersistencia

De esa forma cuando queramos crear un entitymanagerfactory escribimos:

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("unidadPersistencia");
```

También aparece marcado en amarillo los datos de la conexión que antes detallamos:

```
jdbc:mysql://localhost:3306/northwind?serverTimezone=UTC"
```

Hemos marcado en amarillo la forma en la que ponemos el nombre del usuario que conectará el programa a la base de datos:

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

La contraseña para ese usuario:

```
<property name="javax.persistence.jdbc.password" value="1q2w3e4r"/>
```

Y finalmente hemos marcado una línea que se pone para hacer debug Lo que hace es mostrarnos que ejecuta hibernate al actuar sobre la base de datos:

```
<property name="hibernate.show_sql" value="true" />
```

Con lo anterior ya lo tenemos todo configurado. Ahora para comprobarlo vamos a mostrar en pantalla los datos del primer empleado (empleado con id 1)

Podemos poner en el main() (para el caso de un proyecto simple, no Web) :

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

import es.iespuertodelacruz.jc.jpainicial.entities.Employee;

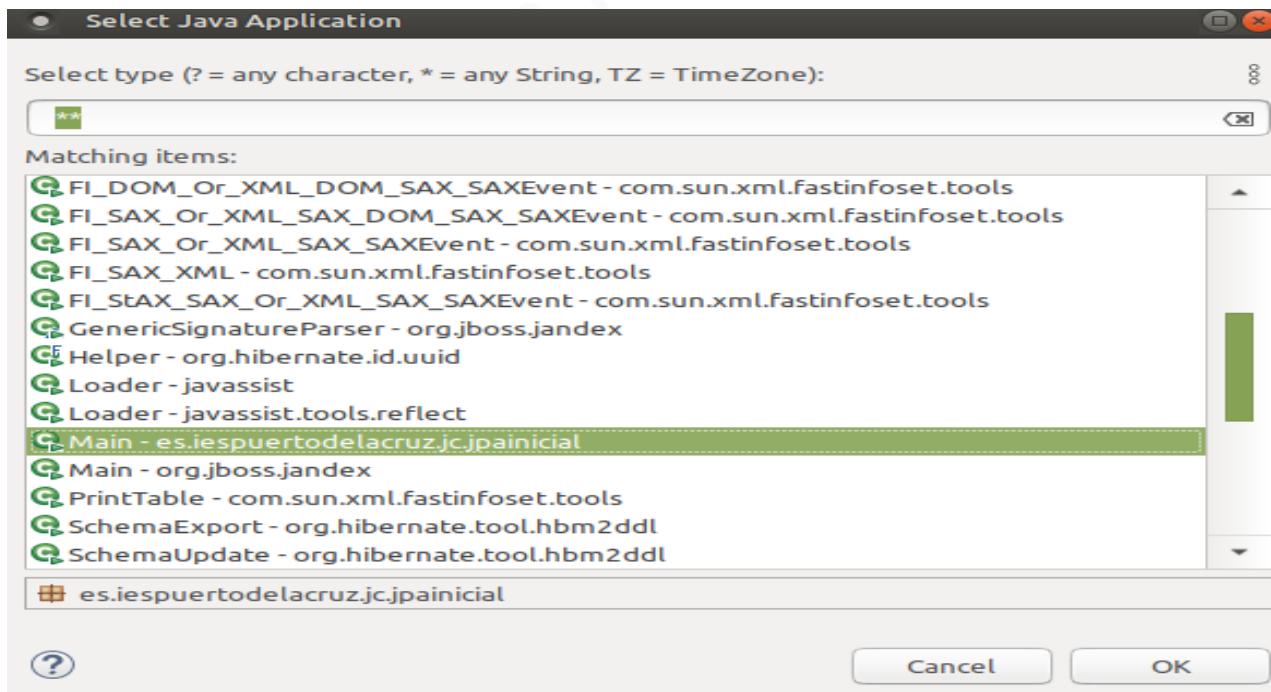
public class Main {

    public static void main(String[] args) {
        EntityManagerFactory emf =
Persistence.createEntityManagerFactory("unidadPersistencia");
        EntityManager em = emf.createEntityManager();

        Employee emp = em.find(Employee.class, 1);

        System.out.println( emp.getFirstName() + " " + emp.getLastName());
    }
}
```

Para ejecutar la aplicación elegimos: Run as: Java application y buscamos la clase Main que tenemos en nuestro proyecto:



Anexo: Crear tablas en la DDBB habiendo creado las entities

Las aplicaciones Java que atacan a bases de datos suelen ser aplicaciones empresariales grandes. En esos casos, casi seguro estamos en una situación en la que ya hay una base de datos establecida y, en todo caso, se mapean los objetos relacionales de la DDBB en objetos Java. Pero vamos a ver el procedimiento si fuéramos a crear la DDBB desde código

Son dos puntos los que hay que tener en cuenta: `property schema-generation` en la unidad de persistencia:

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

y luego ejecutar:

```
Persistence.generateSchema("creardbjpa", null);
```

Para que funcione primero hay que crear la DDBB vacía en el SGBD.

Veamos ejemplo completo:

- 1 Creamos en mysql la database: valormonedas
- 2 Creamos un paquete para las entities y las ponemos allí:
(veamos dos entities)

```

@Entity
@Table(name = "monedas")
@XmlRootElement
public class Monedas implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Basic(optional = false)
    @Column(name = "idmoneda")
    private Integer idmoneda;
    @Basic(optional = false)
    @Column(name = "nombre")
    private String nombre;
    @Column(name = "pais")
    private String pais;
    @OneToMany(      mappedBy = "fkidmoneda"      )
    private Collection<Historicocambioeuro> historicos;

    public Monedas() { }
    public Monedas(Integer idmoneda) { this.idmoneda = idmoneda; }
    public Monedas(Integer idmoneda, String nombre) {
        this.idmoneda = idmoneda;
        this.nombre = nombre;
    }

    public Integer getIdmoneda() { return idmoneda; }
    public void setIdmoneda(Integer idmoneda) { this.idmoneda = idmoneda; }
    public String getNombre() { return nombre; }
    public void setNombre(String nombre) { this.nombre = nombre; }
    public String getPais() { return pais; }
    public void setPais(String pais) { this.pais = pais; }
    public Collection<Historicocambioeuro> getHistoricos() {
        return historicos;
    }

    public void setHistoricos(Collection<Historicocambioeuro> historicos) {
        this.historicos = historicos;
    }
    @Override
    public int hashCode() {
        int hash = 0;
        hash += (idmoneda != null ? idmoneda.hashCode() : 0);
        return hash;
    }
    @Override
    public boolean equals(Object object) {
        if (!(object instanceof Monedas)) {
            return false;
        }
        Monedas other = (Monedas) object;
        if ((this.idmoneda == null && other.idmoneda != null) || (this.idmoneda != null
&& !this.idmoneda.equals(other.idmoneda))) {
            return false;
        }
        return true;
    }

    @Override
    public String toString() {
        return "entitiesmonedas.Monedas[ idmoneda=" + idmoneda + " ]";
    }
}

```

Y ahora la otra entity:

```
@Entity
@Table(name = "historicocambioeuro")
@XmlRootElement
public class Historicocambioeuro implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Basic(optional = false)
    @Column(name = "idhistoricocambioeuro")
    private Integer idhistoricocambioeuro;
    @Column(name = "fecha")
    @Temporal(TemporalType.DATE)
    private Date fecha;

    @Column(name = "equivalenteeuro")
    private BigDecimal equivalenteeuro;

    @JoinColumn(name = "fkidmoneda", referencedColumnName = "idmoneda", foreignKey =
@ForeignKey(name = "fk_moneda"))
    @ManyToOne(optional = false)
    private Monedas fkidmoneda;

    public Historicocambioeuro() { }
    public Historicocambioeuro(Integer idhistoricocambioeuro) {
        this.idhistoricocambioeuro = idhistoricocambioeuro;
    }
    public Integer getIdhistoricocambioeuro() { return idhistoricocambioeuro; }
    public void setIdhistoricocambioeuro(Integer idhistoricocambioeuro) {
        this.idhistoricocambioeuro = idhistoricocambioeuro;
    }
    public Date getFecha() { return fecha; }
    public void setFecha(Date fecha) { this.fecha = fecha; }
    public BigDecimal getEquivalenteeuro() {return equivalenteeuro; }
    public void setEquivalenteeuro(BigDecimal equivalenteeuro) {
        this.equivalenteeuro = equivalenteeuro;
    }
    public Monedas getFkidmoneda() { return fkidmoneda; }
    public void setFkidmoneda(Monedas fkidmoneda) {
        this.fkidmoneda = fkidmoneda;
    }
    @Override
    public int hashCode() {
        int hash = 0;
        hash += (idhistoricocambioeuro != null ? idhistoricocambioeuro.hashCode() : 0);
        return hash;
    }

    @Override
    public boolean equals(Object object) {
        if (!(object instanceof Historicocambioeuro)) {
            return false;
        }
        Historicocambioeuro other = (Historicocambioeuro) object;
        if ((this.idhistoricocambioeuro == null && other.idhistoricocambioeuro != null) ||
(this.idhistoricocambioeuro != null && !this.idhistoricocambioeuro.equals(other.idhistoricocambioeuro))) {
            return false;
        }
        return true;
    }
}
```

3 Se tiene que establecer el persistence.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.2" 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_2.xsd">
  <persistence-unit name="creardbjpa">
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>

    <class>es.iespuertodelacruz.jc.creardbjpa.entity.Historicocambioeuro</class>
    <class>es.iespuertodelacruz.jc.creardbjpa.entity.Monedas</class>

    <properties>
      <property name="hibernate.show_sql" value="true" />
      <property name="hibernate.dialect"
        value="org.hibernate.dialect.MySQL5Dialect" />
      <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="1q2w3e4r" />
      <property name="javax.persistence.jdbc.url"
        value="jdbc:mysql://localhost:3306/valormonedas" />

      <property name="javax.persistence.schema-
generation.database.action"
        value="drop-and-create" />
    </properties>
  </persistence-unit>
</persistence>
```

Se ha marcado las líneas más importantes: el nombre de la unidad de persistencia que vamos a ejecutar: `<persistence-unit name="creardbjpa">`

El nombre de la conexión hacia la ddbb que hemos creado vacía:

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

La propiedad que hay que establecer que permite la creación de objetos en la DDBB relacional:

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

4 Finalmente hay que ejecutar el siguiente código (método main o cualquier método que lancemos) Observar que se tiene que dar el nombre correcto de la unidad de persistencia

```
Persistence.generateSchema("creardbjpa", null);
```

Juan Carlos Pérez Rodríguez

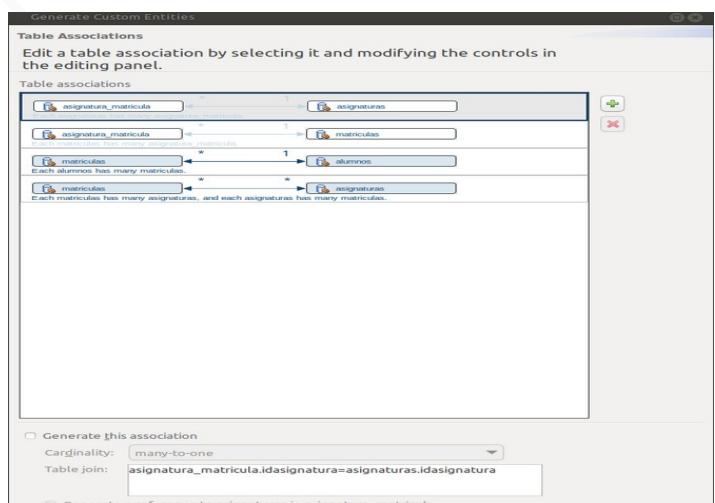
Anexo: Hacer que Eclipse genere correctamente las anotaciones @ManyToOne

En el wizard incluimos (si nos la oferta) la tabla intermedia



Desactivamos (uncheck Generate this association) en las dos relaciones 1:N que se establecen con la tabla intermedia asignatura_matricula

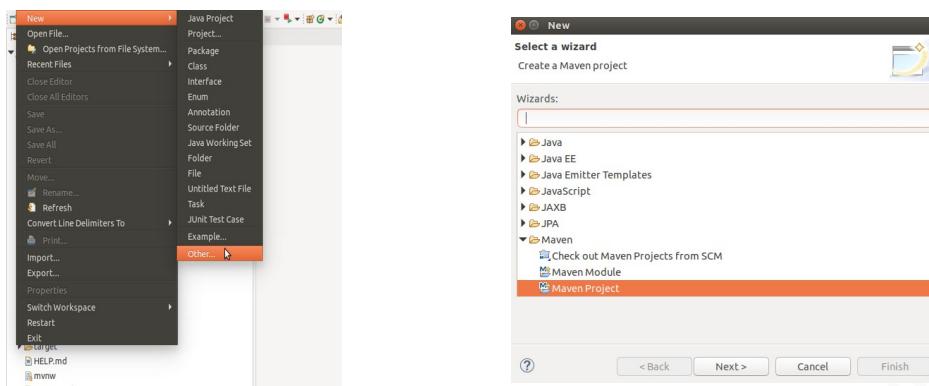
Dejando activada la N:M



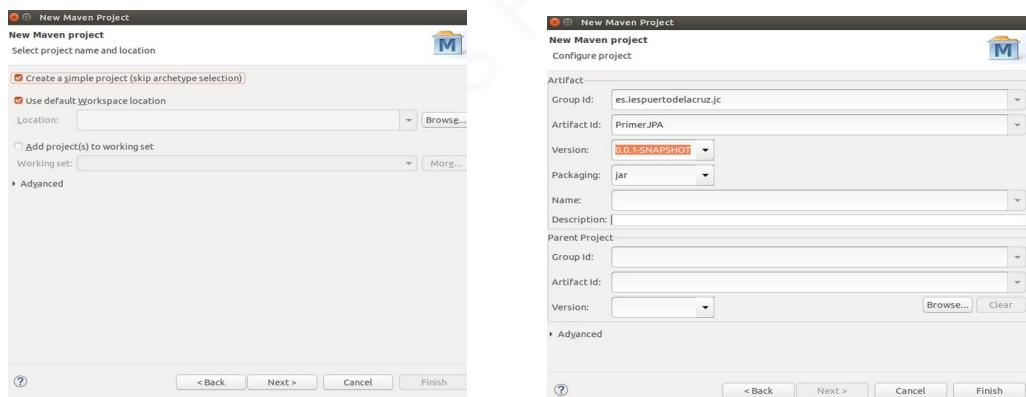
Al finalizar el wizard nos genera correctamente las anotaciones en las entities creadas

Anexo Eclipse crear aplicación jdk1.8

File → New → Other → maven → Maven Project

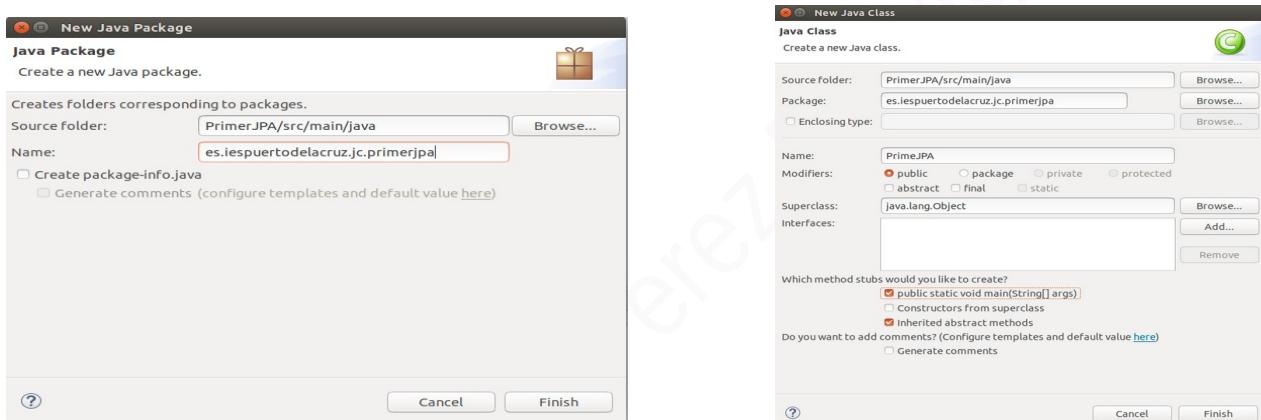


Después en el wizard nos preguntan y elegimos que sea un proyecto simple (no seleccionamos el archetype ahora) En la siguiente ventana especificamos el Group id (recomendable poner algo estructurado del estilo de nuestra organización: es.iespuertodelacruz concatenado con nuestro nombre o un acrónimo) En Artifact id pondremos el nombre de nuestro proyecto



Nos creará la estructura del proyecto
maven

Nosotros vamos a crear un paquete que coincidirá con lo que pusimos en nuestro artefacto, grupo de maven y una clase donde vamos a ubicar nuestro main()



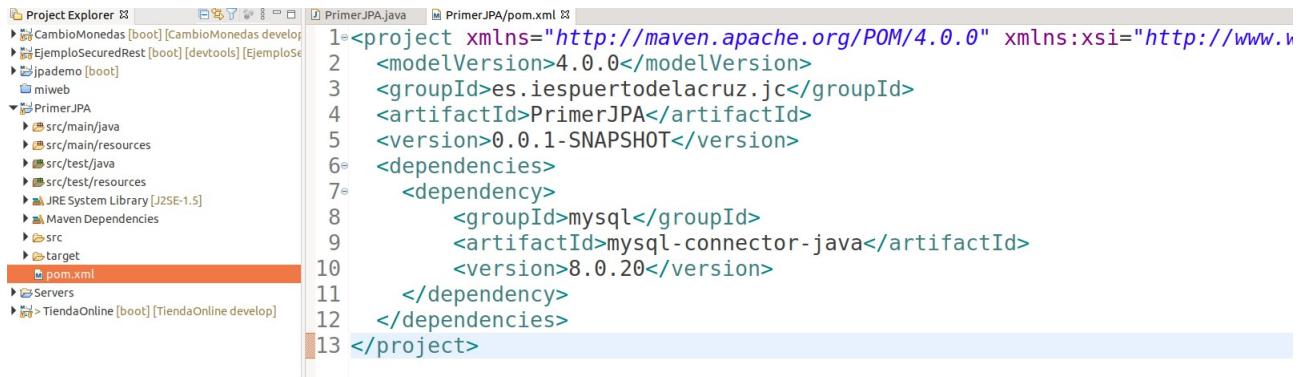
Una vez creado vamos a agregar un par de cosas al fichero pom.xml (el fichero de maven).

Si bien nosotros podemos configurar los IDE para la búsqueda de dependencias maven desde el propio IDE vamos a hacer la búsqueda directamente en la web. Eso nos independiza del IDE utilizado y por otro lado le quita carga al IDE (si se pone que actualice el índice de los repositorios etc se ralentiza).

Buscaremos por: maven central mysql (maven central es nuestro repositorio de dependencias)

Central (81)	Jahia (1)	Redhat GA (3)	Redhat EA (1)	ICM (10)
Version	Repository	Usages	Date	
8.0.22	Central	111	Oct, 2020	
8.0.21	Central	176	Jul, 2020	
8.0.20	Central	152	Apr, 2020	

Pulsamos en este caso, respecto a alguno más o menos actual (8.0.20) y tomamos el texto que nos da. Lo ponemos en nuestro fichero pom.xml:



The screenshot shows the Eclipse IDE interface. On the left, the Project Explorer view lists several Java projects: CambioMonedas [boot], EjemploSecuredRest [boot], EjemploSecuredRest [devtools], Jpademo [boot], miweb, PrimerJPA, and TiendaOnline [boot]. The 'PrimerJPA' project is selected. In the center, the code editor displays the 'pom.xml' file with the following content:

```
<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.iespuertodelacruz.jc</groupId>
    <artifactId>PrimerJPA</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <dependencies>
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>8.0.20</version>
        </dependency>
    </dependencies>
</project>
```

Vemos que le estamos agregando dependencias. En este caso en concreto la del driver mysql.

También le diremos que use java 8:

```
<properties>
    <maven.compiler.target>1.8</maven.compiler.target>
    <maven.compiler.source>1.8</maven.compiler.source>
</properties>
```

Adicionalmente hay que poner las librerías de persistencia (en este caso las de hibernate)

```
<!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-core -->
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.4.3.Final</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-entitymanager -->
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-entitymanager</artifactId>
    <version>5.4.3.Final</version>
</dependency>

<!-- https://mvnrepository.com/artifact/javax.persistence/javax.persistence-api -->
<dependency>
    <groupId>javax.persistence</groupId>
    <artifactId>javax.persistence-api</artifactId>
    <version>2.2</version>
</dependency>
```

Veamos el fichero pom.xml que nos queda completo(en general nos interesa copiar properties y dependencies):

```
<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.iespuertodelacruz.jc</groupId>
  <artifactId>PrimerJPA</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <properties>
    <maven.compiler.target>1.8</maven.compiler.target>
    <maven.compiler.source>1.8</maven.compiler.source>
  </properties>
  <dependencies>
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>8.0.20</version>
    </dependency>
    <!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-core -->
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-core</artifactId>
      <version>5.4.3.Final</version>
    </dependency>
    <!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-
entitymanager -->
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-entitymanager</artifactId>
      <version>5.4.3.Final</version>
    </dependency>
    <!--
https://mvnrepository.com/artifact/javax.persistence/javax.persistence-api -->
    <dependency>
      <groupId>javax.persistence</groupId>
      <artifactId>javax.persistence-api</artifactId>
      <version>2.2</version>
    </dependency>
  </dependencies>
</project>
```

Anexo: Dependencias maven para un proyecto web jdk 11 con JPA, junto con plugin jetty

```
<dependencies>
    <!-- dependencias servlet. Para versión mayor a 2 servlet y con soporte filtros -->
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>javax.servlet-api</artifactId>
        <version>4.0.1</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>javax.servlet.jsp</groupId>
        <artifactId>javax.servlet.jsp-api</artifactId>
        <version>2.3.0</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>javax.el</groupId>
        <artifactId>javax.el-api</artifactId>
        <version>3.0.0</version>
        <scope>provided</scope>
    </dependency>

    <!-- plantillas jstl para jsp -->
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>jstl</artifactId>
        <version>1.2</version>
    </dependency>
    <dependency>
        <groupId>taglibs</groupId>
        <artifactId>standard</artifactId>
        <version>1.1.2</version>
    </dependency>
    <!-- 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/com.fasterxml.jackson.core/jackson-core -->
```

```

<dependency>
<groupId>com.fasterxml.jackson.core</groupId>
<artifactId>jackson-databind</artifactId>
<version>2.12.2</version>
</dependency>
<!--
https://mvnrepository.com/artifact/org.hibernate/hibernate-core
-->
<dependency>
<groupId>org.hibernate</groupId>
<artifactId>hibernate-core</artifactId>
<version>5.4.3.Final</version>
</dependency>
<!--
https://mvnrepository.com/artifact/org.hibernate/hibernate-entitymanager
-->
<dependency>
<groupId>org.hibernate</groupId>
<artifactId>hibernate-entitymanager</artifactId>
<version>5.4.3.Final</version>
</dependency>
<!--
https://mvnrepository.com/artifact/javax.persistence/javax.persistence-api
-->
<dependency>
<groupId>javax.persistence</groupId>
<artifactId>javax.persistence-api</artifactId>
<version>2.2</version>
</dependency>
<!-- generar hash password -->
<dependency>
<groupId>org.mindrot</groupId>
<artifactId>jbcrypt</artifactId>
<version>0.4</version>
</dependency>
</dependencies>
<build>
<plugins>
<plugin>
<artifactId>maven-compiler-plugin</artifactId>
<version>3.8.1</version>
<configuration>

```

```
<release>11</release>
</configuration>
</plugin>
<plugin>
<artifactId>maven-war-plugin</artifactId>
<version>3.2.3</version>
</plugin>
<!-- Jetty Plugin. Default port is 8080 -->
<plugin>
<groupId>org.eclipse.jetty</groupId>
<artifactId>jetty-maven-plugin</artifactId>
<version>9.4.28.v20200408</version>
<configuration>
<httpConnector>
<port>8080</port>
</httpConnector>
<!-- scans your project for any changes to hot swap it to server -->
<scanIntervalSeconds>5</scanIntervalSeconds>
</configuration>
</plugin>
</plugins>
</build>
```

Para convertir el proyecto luego en JPA y obtener enties desde las tablas, instrucciones en el anexo:
: [#Convertir proyecto en JPA|outline](#)