ORM con JAVA

Juan Carlos Pérez Rodríguez

Sumario

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

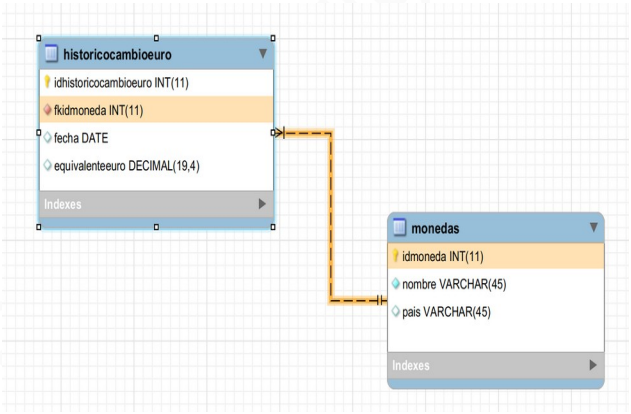
Aplicación que seguiremos para el aprendizaje

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(idhistoricocambioeuro,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

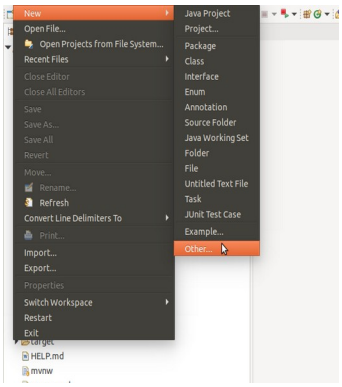
Contenido de la tabla historicocambioeuro:

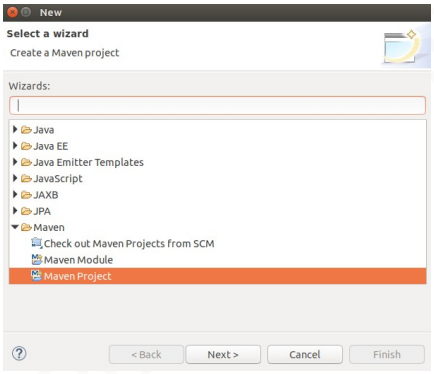
idhistoricocambioeuro	fkidmoneda	fecha	equivalenteeuro
1	1	2020-01-01	1.1200
2	2	2020-03-25	0.2500
3	2	2019-01-14	0.5000
4	1	2019-12-27	0.9000
5	1	2020-12-30	0.9500
6	1	2020-11-23	0.8000

Ilustración 2: tabla historicocambioeuro

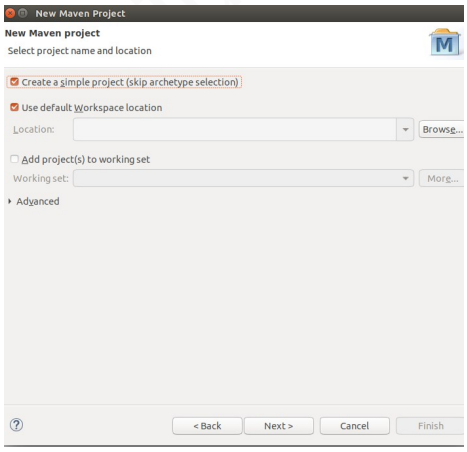
Ahora vamos a crear una aplicación maven en Eclipse:

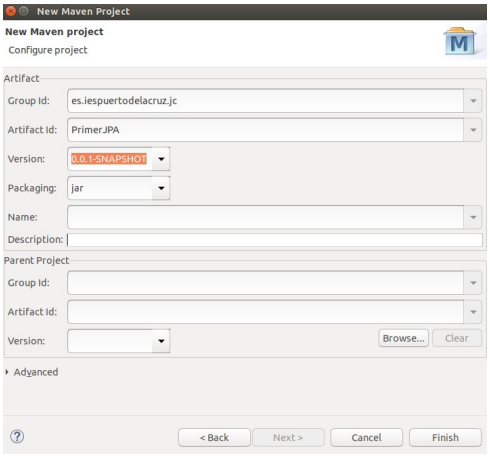
File → New → Other → maven → Maven Project





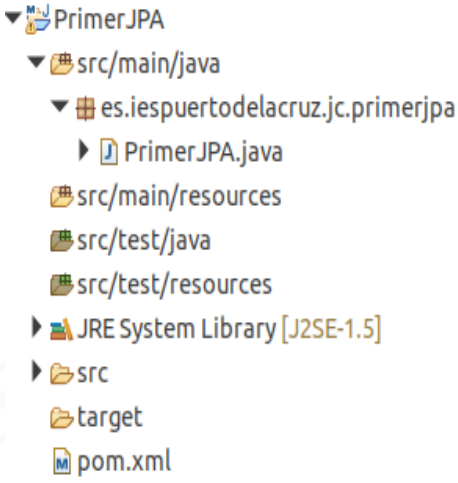
Después en el wizard nos preguntan y elegimos que sea un proyecto simple (no seleccionamos el archetype ahora) En la siguiente ventana especificamos el Group id (recomendable poner algo estructurado del estilo de nuestra organización: es.iespuertodelacruz concatenado con nuestro nombre o un acrónimo)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

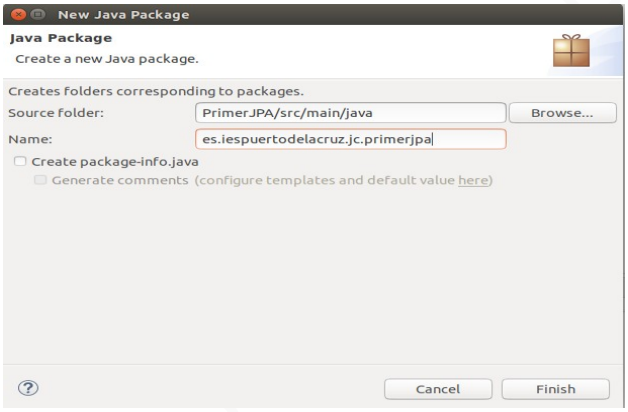


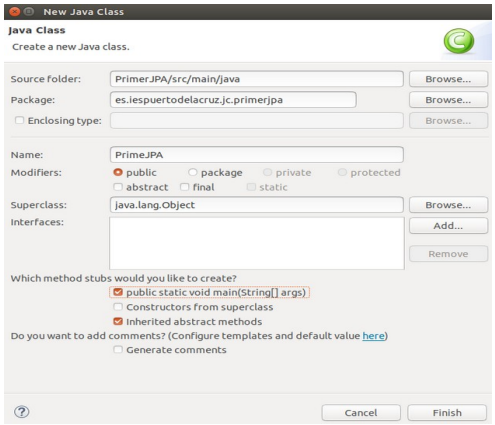


Nos creará la estructura del proyecto maven

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()



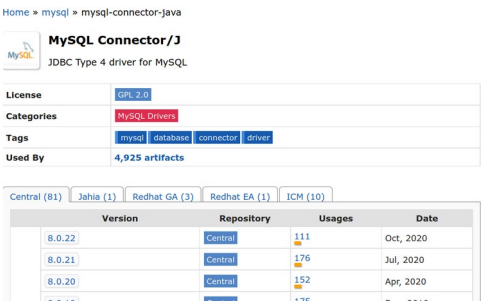




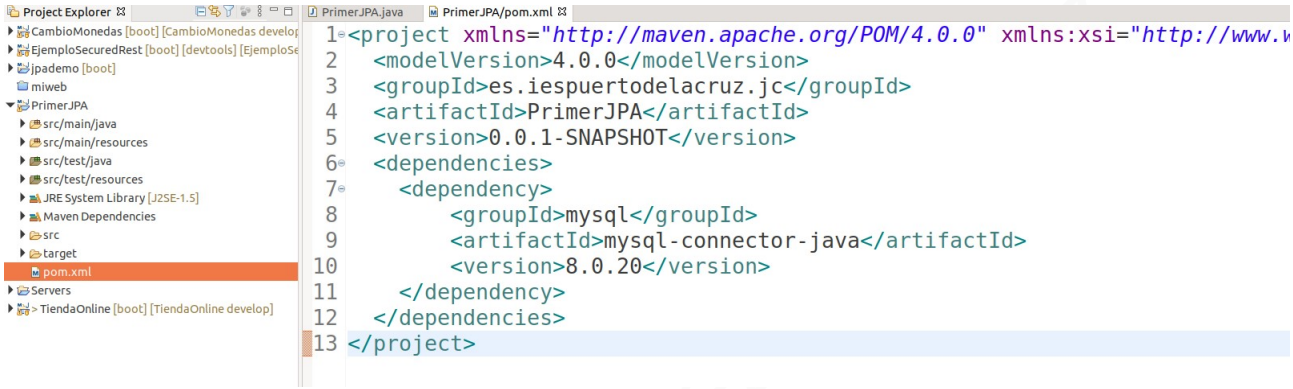
Una vez creado vamos a agregar un par de cosas al fichero pom.xml (el fichero de maven) .

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)



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:



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>
```

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:

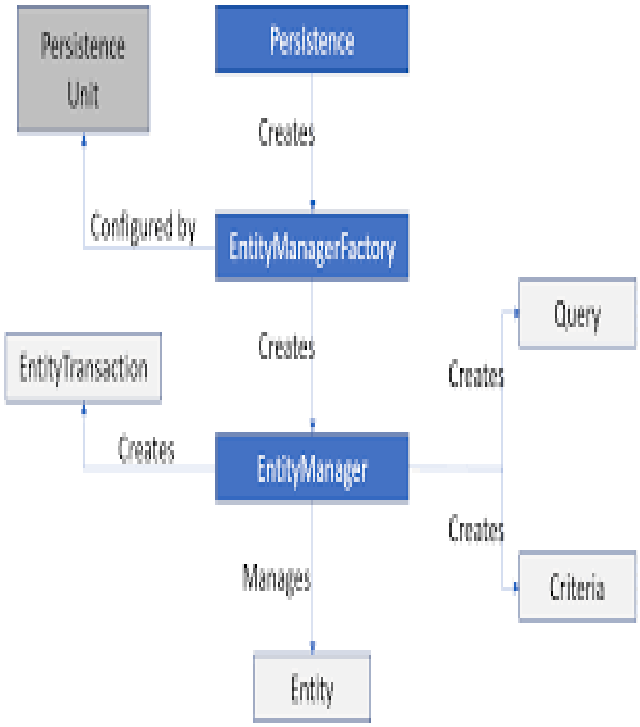
```
@Entity
@Table(name = "usuario")
public class Usuario implements Serializable {

    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 … blablaba")
		.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/seguimientomonedas?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 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.

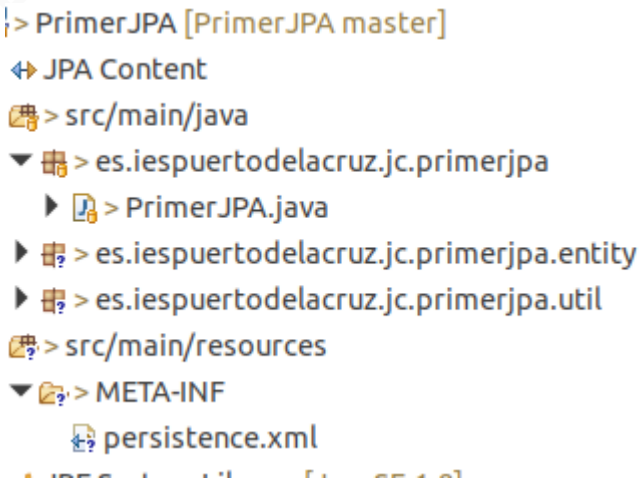
Dicho lo anterior, vemos que habitualmente NO HAY MÁS NECESIDAD QUE UN ÚNICO OBJETO EntityManagerFactory, una única unidad de persitencia y que tendrá vida para toda la aplicación. Es por eso que en nuestra aplicación de ejemplo vamos a apoyarnos en el patrón singleton para mantener la entitymanagerfactory.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



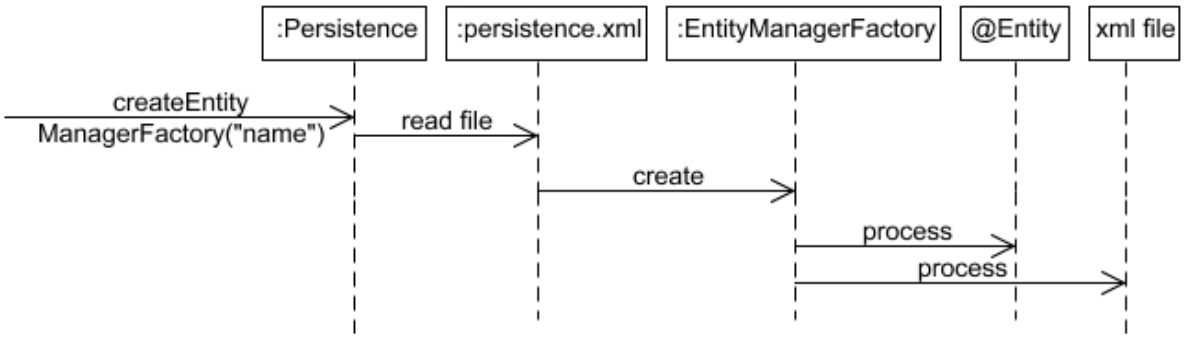
Como hemos dicho ya, podemos ahora crear un EntityManagerFactory Para ello sólo tenemos que escribir:

```
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 lóngitud 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 wraper (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 lleve 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      */
//bi-directional many-to-one association to Historicoca
@OneToMany(mappedBy="moneda")
private List<Historicocambioeuro> historicocambioeuros;
```

```
/*      CLASE Historicocambioeuro PARTE N DE LA RELACIÓN 1:N
//bi-directional many-to-one association to Moneda
@ManyToOne
@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)

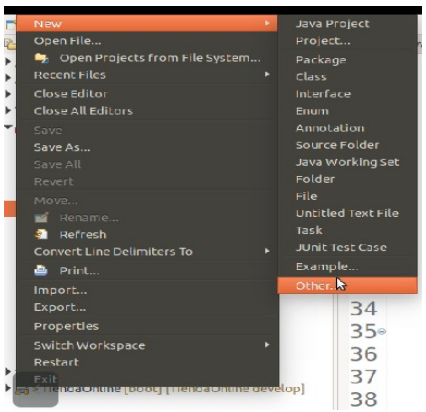
- En la Entity de la relación que es la parte N ponemos la notacón @ManyToOne y se indica el nombre del campo de la tabla que tiene la foreign key mediante @JoinColumn: @JoinColumn(name="fkidmoneda", nullable=false) sobre el atributo de la otra entidad (moneda en este caso). ¡¡ importante !! Observar 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)

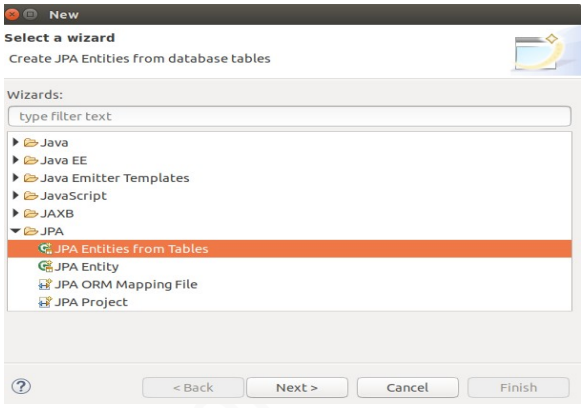
Debemos conocer las anotaciones y en algunos casos hay que modificarlas (la veremos que es lazy y eager), pero muchas veces podemos apoyarnos en el IDE para generar las entities

Procedimiento para generación Entities de forma automática

En Eclipse haremos:

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

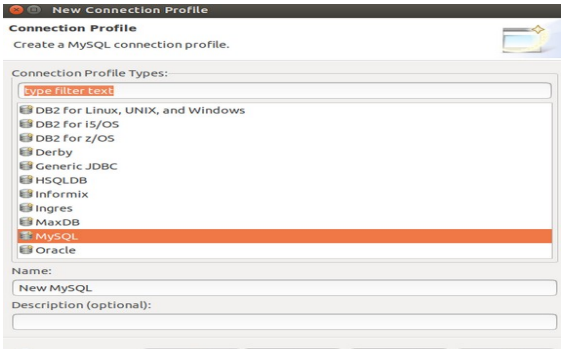




- Se nos abre un wizard con la pantalla:

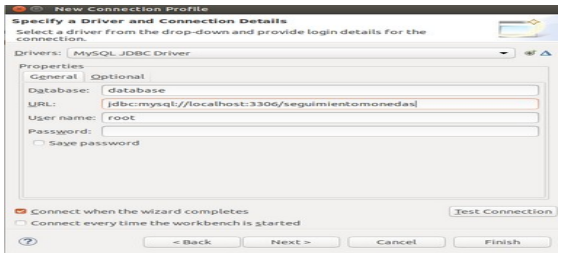
En esta pantalla debemos establecer la conexión a la base de datos. (observar donde dice Connection) hay un botón a la derecha que nos permite crear una nueva conexión si fuera pertinente

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



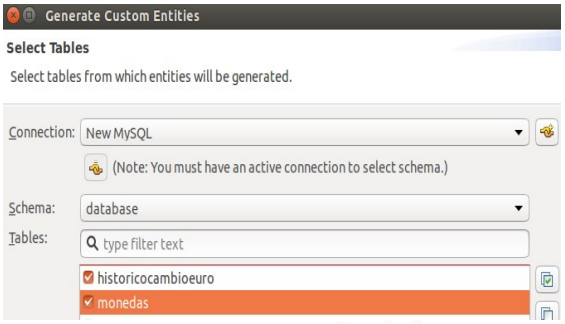
Elegiremos la versión del driver que tengamos ya descargado (por ejemplo la versión 5.1) y elegimos en el disco duro la ubicación donde tengamos ese driver (por ejempl la 5.1.49)

- 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

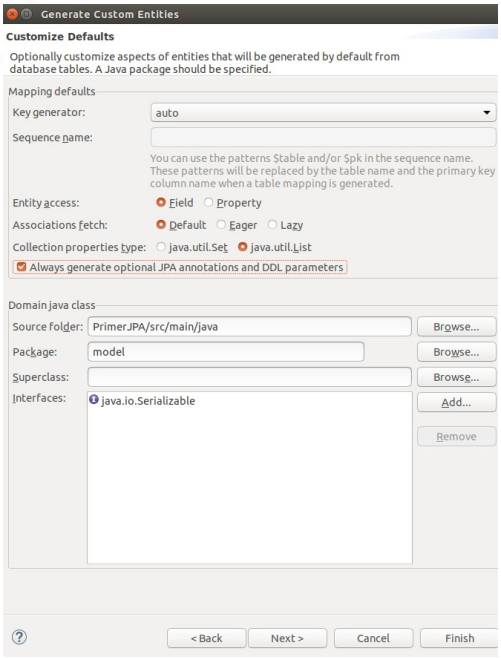
- Al pulsar next nos oferta las tablas que encuentra en la base de datos y de las que nosotros queramos crear entities. Elegiremos todas



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 nombamos 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 EntityManagers 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 realizado 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 DDBB ? 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("Lira1");
moneda.setPais("Turquía");
em.persist(moneda);
em.getTransaction().begin();
moneda.setNombre("lira2");
em.getTransaction().commit();
moneda.setNombre("lira3");
em.close();
```

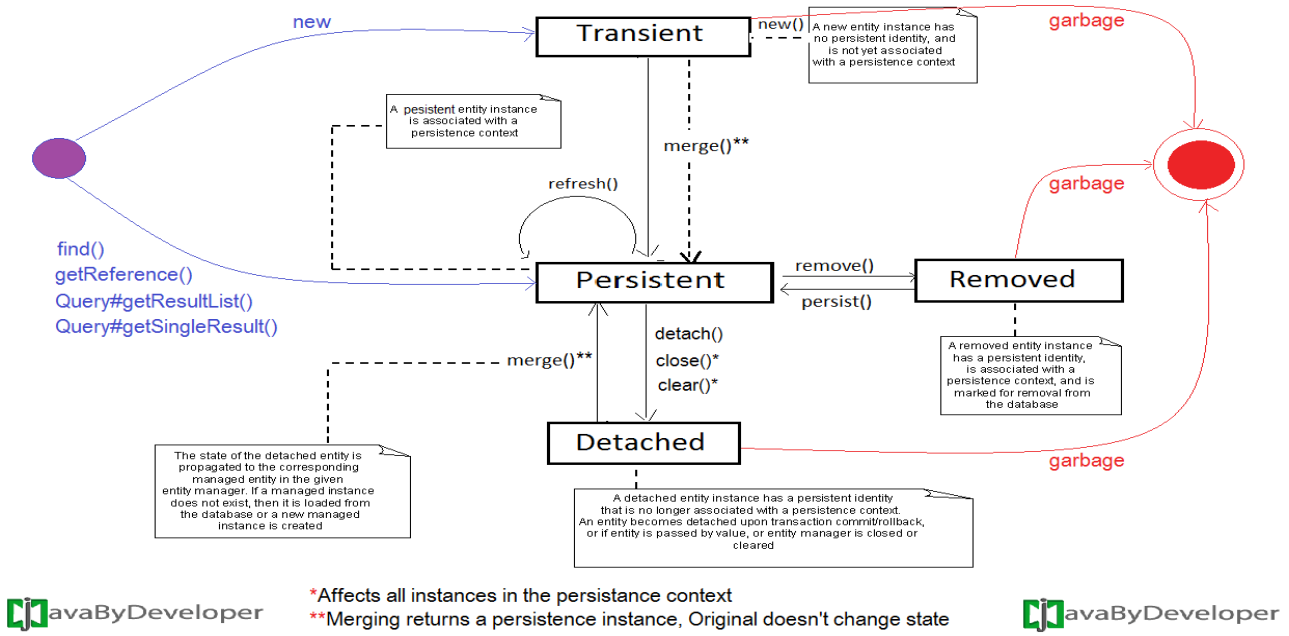
Práctica 4: ¿ qué nombre queda guardado en la base de datos con el código anterior?

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 autogenere) 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 memoría) que esté mapeada pon 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

detach()

Podemos considerar detach() como el inverso de merge(). Nos saca una entidad del contexto de persistencia (deja de estar vigilada) Eso significa que los cambios que se le hagan a la entidad no tendrán reflejo en la base de datos

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: Historiocambioeuro ¿qué ocurrirá ?

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

```
            Historicocambioeuro h = new Historicocambioeuro();
            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 Historicocambioeuro 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 CascadaType. 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

Creación de objetos con relaciones

El siguiente código es el de un método que recibe un id moneda y un EntityManager y crea Historicos de forma automática:

```
	public	static	Historicocambioeuro	crearHistoricoDeMoneda(Integer idmoneda,EntityManager em) {

		Moneda moneda = em.find(Moneda.class, idmoneda);
		Historicocambioeuro h = new Historicocambioeuro();
		h.setEquivalenteeuro(new BigDecimal(Math.random()*9));
		h.setFecha(new Date());
		h.setMoneda(moneda);

		EntityTransaction tr= (em.isJoinedToTransaction())? null:em.getTransaction();
		if( tr != null)
			tr.begin();
		em.persist(h);

		if( tr != null)
			tr.commit();
		System.out.println(h);
		return h;
	}
```

Del código anterior, lo único nuevo es: isJoinedToTransaction()

Usar esa instrucción nos permite que el método funcione en casos que estamos dentro de transacción o no lo estemos

Recordar que modificaciones en la DDBB implica que hay que usar transacción. Lo que se hace es garantizarse que estamos dentro de una transacción. Si no lo estamos se genera una nueva

Puntualizar que NO se puede crear una transacción si ya estamos dentro de una. Es por eso la importancia de: isJoinedToTransaction()

Ahora veamos el trozo del código del main():

```
            EntityManager em = emf.createEntityManager();
            Moneda moneda = new Moneda();
            moneda.setNombre("Lira56");
            moneda.setPais("Turquía");


            em.getTransaction().begin();
            em.persist(moneda);
            for (int i = 0; i <3; i++) {
                  crearHistoricoDeMoneda(moneda.getIdmoneda(),em);
            }

            em.getTransaction().commit();


            em.close();

            System.out.println(moneda);
            System.out.println("lista de histórico: " +
moneda.getHistoricocambioeuros());
```

Práctica 9: Ejecutar el código anterior Comprobar (con captura pantalla) si se han creado las entidades en la base de datos. ¿ el último println devuelve el listado o devuelve null ? ¿ por qué ?

Lo primero y más importante es que nosotros en principio si usamos SQL, debiéramos primero guardar la moneda en la base de datos y así obtener un IDmoneda para esa moneda. Una vez tenemos ese IDmoneda podemos ir a la tabla historico y guardar los historicos con esa foreign key

Ese comportamiento nos puede hacer creer que es necesario hacer primero una transact para guardar la (recordar que sin transact no se guarda) y después en subsiguientes transact guardar las Entities historico (cuando ya tenemos un idmoneda generado por la base de datos)

Pero el código anterior funciona Y se ha usado una única transact.

El motor de persistencia se da cuenta de lo que queremos hacer y funcionará

Ahora bien, hemos observado que la lista (el getHistorico) nos la muestra nula. Eso es porque no es capaz de darse cuenta que al insertar un Historico tiene que actualizar la lista en RAM moneda.getHistoricocambioeuro() (En base de datos sí ha quedado todo bien)

Vamos a ver ahora una variante del código anterior:

```
            EntityManager em = emf.createEntityManager();
            Moneda moneda = new Moneda();
            moneda.setNombre("Lira63");
            moneda.setPais("Turquía");

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

            for (int i = 0; i <3; i++) {
                  Historicocambioeuro h =
crearHistoricoDeMoneda(moneda.getIdmoneda(),em);

            }
            em.flush();

            //em.refresh(moneda);
            moneda.getHistoricocambioeuros().size();
            em.getTransaction().commit();

            em.close();

            System.out.println(moneda);
            System.out.println("lista       de       histórico:       "       +
moneda.getHistoricocambioeuros());
```

Práctica 10: Ejecutar el código anterior Comprobar (con captura pantalla) si se han creado las entidades en la base de datos. ¿ el último println devuelve el listado o devuelve null ? ¿ por qué ? ¿ se genera algún error ?
Ahora quitar el comentario en em.refresh() ¿ funciona ahora ?

Aquí se están dando tres circunstancias. Si usamos **refresh()** vemos que nos funciona (ese es uno de los detalles) gracias a que obliga a sincronizar DESDE la base de datos las entities (descarta lo que está en RAM y toma info de la DDBB). Su vínculo con **flush()** (flush() obliga a sincronizar inmediatamente HACIA la base de datos lo que hay en la transacción actual) y el otro detalle es llamar a: .size() en la collection Esto último es por el comportamiento Lazy por defecto en persistencia. Ya hablaremos de lazy.

El código anterior funciona porque con los flush() obligamos a que todos los Historico pasen inmediatamente a la base de datos. Eso implica que si hacemos un refresh(moneda) El motor de

persistencia se ve obligado a tomar la información actualizada de la DDBB y hace que moneda tenga una lista en: getHistoricocambioeuro().

Esa lista no está accesible hasta que la llamemos dentro de una transacción (por eso se ejecuta el método size()) . La lista tiene ese comportamiento porque por defecto se usa Lazy entre las entidades relacionadas (no se ejecutan sentencias SQL en las entidades relacionadas hasta que no queda otro remedio para evitar más trabajo del necesario a la DDBB)

Es relevante que la instrucción refresh() nos haya resuelto el problema. Vamos a ver su funcionamiento

refresh()

Este método de entitymanager nos permite forzar recargar la información de la base de datos en nuestras entities del contexto de persistencia. No hay que olvidar que el motor de persistencia tiene que mirar a dos sitios: el SGBD y nuestas entities en ram. La sincronización de ambos ámbitos de forma automática no es posible desde que el SGBD puede estar aceptando múltiples solicitudes de diferentes ámbitos ¿ cómo puede saber el motor de persistencia que otra aplicación con otra conexión está cambiando en un momento dado una fila de la base de datos ?

Sobre todo en ambientes concurrentes puede ocurrir que el estado de nuestras entities esté desfasado con respecto a la realidad de nuestra base de datos. El comando refresh(mientidad) obliga a que se sincronice: mientidad con la información de la base de datos (descartando todo lo que la entidad pudiera haber realizado y tomando lo que existe en ese momento en la base de datos)

Bien. El código que hemos visto nos ha ayudado a entender el funcionamiento de refresh() y flush(). Ahora bien, eso no significa que el código sea eficiente. Hemos obligado a realizar múltiples consultas SQL que podíamos haber evitado. Vamos a ver una versión alternativa de código:

```
            EntityManager em = emf.createEntityManager();
            Moneda moneda = new Moneda();
            moneda.setNombre("Lira65");
            moneda.setPais("Turquía");

            em.getTransaction().begin();
            em.persist(moneda);
      moneda.setHistoricocambioeuros(new ArrayList<Historicocambioeuro>());
            for (int i = 0; i <3; i++) {
      Historicocambioeuro h = crearHistoricoDeMoneda(moneda.getIdmoneda(),em);
                  moneda.getHistoricocambioeuros().add(h);
            }
            //em.flush();

            //em.refresh(moneda);
            //moneda.getHistoricocambioeuros().size();

            em.getTransaction().commit();

            em.close();
```

Como observamos en el código lo que hacemos es cargar nosotros mismos la información en RAM (hemos creado nosotros el ArrayList y le hemos agregado los objetos) Se han dejado las líneas comentadas que ya no son necesarias. Observar que implican varias sentencias SQL en la base de datos que hemos evitado

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 pasarle 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 clausula 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 paréntisis. 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 hace 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

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