JPA con Hibernate

Índice

1. Introducción a JPA

1.1. Introducción a JPA

En la primera sesión del módulo de Java Persistence API (JPA) vamos a tratar una introducción a esta nueva tecnología Java que permite trabajar con entidades persistentes conectadas a una base de datos. Introduciremos los conceptos principales de JPA que iremos desarrollando en posteriores sesiones y proporcionaremos un ejemplo completo en el que describiremos la instalación básica de JPA utilizando Eclipse como entorno de desarrollo y Hibernate como implementación de JPA. Este ejemplo será la base de las prácticas de la sesión.

Entre los conceptos principales que trataremos sobre JPA destacamos los siguientes:

- uso de anotaciones para especificar propiedades
- entidades persistentes y relaciones entre entidades
- mapeado objeto-relacional
- gestión de contextos de persistencia y de transacciones
- diferencias entre JPA gestionado por la aplicación y gestionado por el contenedor
- lenguaje de *queries*

Estudiaremos estos conceptos en profundidad a lo largo del módulo. En la sesión de hoy realizaremos únicamente una introducción. Primero los explicaremos brevemente para después pasar a un ejemplo práctico en el que se podrán comprobar bastantes características de JPA.

1.1.1. Un poco de historia

El framework Hibernate, un conjunto de librerías que implementaba un mapeado ORM (Mapeado Objeto-Relacional), comenzó a ser desarrollado por Gavin King y un grupo de colaboradores a finales de 2001. Desde sus inicios se estableció como un proyecto Java open source. Pronto ganó popularidad y el grupo de desarrolladores fue contratado por JBoss, integrando el producto en el servidor de aplicaciones de la compañía. En la actualidad JBoss ha sido adquirido por RedHat, que ha incorporado su servidor de aplicaciones en algunas de sus distribuciones de Linux.

En paralelo al desarrollo y popularización de Hibernate, la especificación oficial de Java EE también intentaba definir *entidades persistentes*. En concreto, se definía en la arquitectura EJB (Enterprise JavaBeans) el uso de *entity beans*, objetos persistentes distribuidos gestionados por contenedores. Junto a los *entity beans*, Sun también apoyó la especificación de JDO (Java Data Objects), otro framework alternativo de gestión de entidades persistentes que no requiere el uso de contenedores EJB. Ninguno de los dos frameworks tuvo demasiado éxito. Los EJB de entidad siempre fueron denostados por ser muy poco eficientes y complejos de utilizar. JDO, por otra parte, tardó bastante en ser

implementado de una forma robusta y sencilla de manejar.

En este contexto se crea en Mayo de 2003 el grupo de trabajo que definirá la siguiente (actual) especificación de EJB (EJB 3.0). En este grupo de trabajo pronto se tiene que adoptar un modelo para la gestión de entidades persistentes y se decide apostar por la solución que ya ha adoptado de hecho la comunidad: el enfoque basado en POJOs de Hibernate. Tras tres años de trabajo, en Abril de 2006 se realiza la votación que aprueba la nueva especificación. En declaraciones de Gavin King, la especificación de JPA recoge el 95% de las funcionalidades de Hibernate.

En la actualidad Hibernate ofrece una implementación de la especificación de JPA. Una de las diferencias fundamentales con las versiones clásicas de Hibernate es que éstas utilizan ficheros de configuración XML para definir el mapeado de las entidades con la base de datos. JPA (y su implementación Hibernate) recurre a anotaciones y a opciones por defecto para simplificar la configuración.

1.1.2. JPA

Java Persistence API (JPA) es la tecnología estándar de Java para gestionar **entidades persistentes** que se incluye en la última versión de Java EE (Java EE 5). La descripción oficial del estándar está definida en el JSR 220 en el que se especifica la arquitectura completa EJB 3.0. JPA es una parte del estándar EJB 3.0, aunque está especificado en un documento separado y autocontenido. Si quieres obtener una visión completa y detallada de lo que hace JPA, te será muy útil consultar este documento. Como la mayoría de los documentos que especifican las JSR, es un documento bastante legible, muy bien estructurado, muy conciso y con bastante ejemplos. Además, por ser la especificación original, es completo. Cualquier característica de JPA debe estar reflejada en este documento. Te aconsejo, por tanto, que lo tengas a mano, que le eches un vistazo inicial (después de haber leído los apuntes de este módulo, por supuesto) y que lo utilices como primera referencia ante cualquier duda.

La idea de trabajar con entidades persistentes ha estado presente en la Programación Orientada a Objetos desde sus comienzos. Este enfoque intenta aplicar las ideas de la POO a las bases de datos, de forma que las clases y los objetos de una aplicación puedan ser almacenados, modificados y buscados de forma eficiente en unidades de persistencia. Sin embargo, aunque desde comienzos de los 80 hubo aplicaciones que implementaban bases de datos orientadas a objetos de forma nativa, la idea nunca ha terminado de cuajar. La tecnología dominante en lo referente a bases de datos siempre han sido los sistemas de gestión de bases de datos relacionales (RDBMS). De ahí que la solución propuesta por muchas tecnologías para conseguir entidades persistentes haya sido realizar un *mapeado* del modelo de objetos al modelo relacional. JPA es una de estas tecnologías.

1.1.3. Entidades persistentes

De la misma forma que en Programación Orientada a Objetos se trabaja con clases e

instancias, en JPA se definen los conceptos de **clase entidad** (*entity class*) e instancia de una clase entidad a la que llamaremos **instancia entidad** (*entity instance*). Una clase entidad define un conjunto de atributos persistentes que van a compartir todas sus instancias. Por ejemplo, si estamos escribiendo una aplicación para una agencia de viajes será normal que usemos clases entidades como `Hotel`, `Reserva` o `Vuelo`. Las clases entidades se mapean directamente en tablas de la base de datos. Por otro lado, las instancias entidad son objetos concretos de la clase entidad (un `hotel`, o un `vuelo`) y en el mapeado relacional se corresponden con filas concretas de las tablas definidas por la clase entidad.

A diferencia del anterior estándar de persistencia en Java EE (EntityBeans en EJB 2.1), las entidades JPA son POJOs (*Plain Old Java Object*), objetos Java estándar que se crean con una llamada a `new`, que pueden ser pasados como parámetros y que tienen un conjunto de métodos con los que acceder a sus atributos.

1.1.4. Mapeo entidad-relación

La forma de conseguir que las entidades sean persistentes es utilizando el denominado *mapeo entidad-relación* (ORM en inglés *object-relational mapping*).

Cada entidad definida con JPA se asocia a una tabla SQL con los mismos atributos que la entidad. Los elementos de la tabla son las instancias de la entidad. Por ejemplo, supongamos una entidad `Libro` con los siguientes atributos:

- `ISBN`: `String`
- `titulo`: `String`
- `autor`: `String`
- `fechaAlta`: `Date`

En el mapeado ORM esta clase se mapea con una tabla SQL `LIBRO` que tipos de columnas compatibles con los atributos de la entidad. Por ejemplo, podríamos utilizar las siguientes columnas

- `ISBN`: `varchar(13)`
- `titulo`: `varchar(255)`
- `autor`: `varchar(255)`
- `fechaAlta`: `datetime`

JPA se encarga de gestionar el mapeo y mantener sincronizados el modelo del dominio (entidades y sus instancias) y las tablas de la base de la base de datos.

Más adelante veremos las distintas posibilidades del mapeo ORM, que se extienden mucho más allá de este simple ejemplo. Entre las características que se implementan en JPA se encuentran el mapeo de relaciones de herencia entre entidades, las clases embebidas o las múltiples relaciones entre entidades que se mapean en relaciones con claves ajenas en las tablas.

1.1.5. Relaciones entre entidades y ORM

En JPA también es posible definir **relaciones entre entidades**, similares a las relaciones entre tablas en el modelo relacional. Así, una entidad puede estar relacionada con una o muchas otras entidades, definiendo relaciones **uno-a-uno**, **uno-a-muchos** o **muchos-a-muchos**. Estas relaciones se definen en JPA por medio de variables de instancia de las entidades y de métodos *getters* y *setters* que las actualizan. Por ejemplo, una propiedad de una entidad `Reserva` será el `Vuelo` sobre el que se ha hecho la reserva. Las instancias de `Reserva` tendrán asociadas instancias de `Vuelo` a las que podremos acceder fácilmente con un método como `getVuelo`. De esta forma, podremos obtener la(s) instancia(s) asociadas con una instancia dada llamando a un método de la entidad, sin tener que realizar ninguna consulta SQL. Esta es una de las ventajas fundamentales de JPA: es posible hacer de forma programativa (llamando a métodos definidos en las entidades) lo que en el modelo relacional sólo se puede hacer mediante consultas SQL.

Otra de las características principales de JPA frente a otros frameworks que realizan un ORM es su simplicidad. Hasta ahora otros frameworks (como Hibernate) han utilizado ficheros de configuración XML para especificar el mapeado entre clases Java y tablas de la BD relacional. Esto hacía complicado la definición y el mantenimiento de las entidades. La novedad de JPA en este sentido es la utilización de **anotaciones**, una importante característica de Java aportada en su release 5.0. Veremos que el uso de anotaciones simplifica bastante la definición de entidades y de sus relaciones.

1.1.6. Entity Manager y transacciones

El *Entity Manager* es el objeto de JPA que gestiona los contextos de persistencia y las transacciones. Una de las características más importantes de JPA es que no es invisible. Las entidades deben cargarse, borrarse, modificarse, etc. de forma activa por parte de la aplicación que está utilizando JPA (con la excepción, quizás, de las llamadas a los métodos `set`). Cuando una entidad se obtiene de la base de datos se guarda en una especie de caché en memoria que mantiene JPA. Esta caché se denomina **contexto de persistencia**, y se mantiene en el objeto `EntityManager` que utiliza la aplicación.

Es posible utilizar JPA en dos configuraciones. Por un lado puede ser usado en aplicaciones de escritorio implementadas en Java SE y en aplicaciones web sin utilizar ningún servidor de aplicaciones. En este caso, la gestión de la persistencia la realiza la aplicación Java y se habla de JPA gestionado por la aplicación (*application-managed*). En el segundo es el servidor de aplicaciones Java EE el que proporciona el *entityManager* y se dice que JPA está gestionado por el contenedor (*container-managed*).

Una diferencia fundamental entre ambas configuraciones es la gestión de las transacciones realizada por el *Entity Manager*. En el primer caso (persistencia gestionada por la aplicación) JPA implementa la gestión de transacciones utilizando el gestor de transacciones local del recurso con el que está trabajando, transacciones nativas definidas

por el driver de JDBC. Las transacciones de este tipo son locales y no pueden participar en una conversación gestionada por una transacción de mayor ámbito. En el segundo caso (persistencia gestionada por el contenedor) se utiliza el API de gestión de transacciones JTA. Con él es posible utilizar transacciones extendidas y hacer que distintas transacciones elementales participen en una transacción común a todas ellas. Para trabajar con JPA gestionado por un contenedor necesitamos un servidor de aplicaciones que de soporte a la gestión de componentes Enterprise JavaBeans (EJB).

1.1.7. Implementaciones de JPA

JPA es un estándar que necesita ser implementado por desarrolladores o empresas. Al ser una especificación incluida en Java EE 5 cualquier servidor de aplicaciones compatible con Java EE debe proporcionar una implementación de este estándar. Por otro lado, dado que también es posible utilizar JPA en Java SE, existen bastantes implementaciones de JPA en forma de librerías Java (ficheros JAR) disponibles para incluir en aplicaciones de escritorio o aplicaciones web. La más popular es Hibernate, que también se incluye en el servidor de aplicaciones JBoss. Otras implementaciones gratuitas son Apache OpenJPA (incluida en el servidor de aplicaciones Jeronimo) y Oracle TopLink (incluida en el servidor de aplicaciones GlassFish de Sun). La única implementación comercial de JPA existente en la actualidad es CocoBase PURE POJO.

La implementación de Hibernate es la más popular del estándar. La gran aceptación de Hibernate en la comunidad de desarrolladores Java se refleja en que en la actualidad hay muchas empresas que utilizan Hibernate como capa de persistencia y no han dado todavía el salto a JPA. Es previsible que lo hagan próximamente.

1.2. Un ejemplo completo con Hibernate

Vamos a analizar los distintos elementos que componen una aplicación JPA con un ejemplo práctico. Vamos a centrarnos en una aplicación de escritorio desarrollada en Java SE. Más adelante veremos como se incorporan estos elementos en una aplicación web. El ejemplo que vamos a usar es muy sencillo, una aplicación que gestiona mensajes creados por autores. Se definen dos entidades: `Autor` y `Mensaje` así como una relación uno a muchos entre ellos. Un autor tiene asociado la lista de mensajes escritos. Queremos que sea una relación denominada padre-hijo, en la que el `Autor` sea el propietario de la relación. Si se elimina un autor de la base de datos, se deben eliminar los mensajes que ha escrito.

Comenzamos en esta sección listando los ficheros que vamos a utilizar y después comentaremos los elementos necesarios para configurar y ejecutar la aplicación en Eclipse. En el apartado siguiente analizaremos los conceptos más importantes de JPA basándonos en este ejemplo.

Fichero `es/ua/jtech/jpa/Autor.java`:

```
package es.ua.jtech.jpa;

import java.util.Collection;
import javax.persistence.*;

@Entity
public class Autor {
   @Id
   private String nombre;
   private String correo;
   @OneToMany(mappedBy = "autor", cascade=CascadeType.ALL)
   private Set<Mensaje> mensajes = new HashSet<Mensaje>();

   public Autor() { }
   public String getNombre() { return nombre; }
   public void setNombre(String nombre) { this.nombre = nombre; }
   public String getCorreo() { return correo; }
   public void setCorreo(String correo) { this.correo = correo; }
   public Set<Mensaje> getMensajes() { return mensajes; }
   public void setMensajes(Set<Mensaje> mensajes) {
      this.mensajes = mensajes; }

   //Metodo para añadir un mensaje
   public void addMensaje(Mensaje mensaje) {
      this.getMensajes().add(mensaje);
      mensaje.setAutor(this);
   }

}
```

El fichero `Autor.java` define la clase entidad `Autor`. La entidad tiene los atributos `nombre` (nombre del autor, debe ser único por ser el identificador de la entidad), `correo` (su dirección de correo electrónico) y `mensajes`. Los mensajes están precedidos con la anotación `@OneToMany` para indicar que el `Autor` tiene una relación uno-a-muchos con las entidades `Mensaje` (más adelante explicaremos la anotación `mappedBy`). La anotación `cascade` indica que las acciones de borrado, *persist* y *merge* se propagan en cascada a los mensajes hijos.

También se define en el bean un método distinto de un *getter* o un *setter* que gestiona la operación de relacionar un autor con un mensaje. Es necesario actualizar tanto la lista de mensajes en el autor, como el atributo `autor` del mensaje.

Fichero `es/ua/jtech/jpa/Mensaje.java`

```
package es.ua.jtech.jpa;

import java.util.Date;

import javax.persistence.*;

@Entity
public class Mensaje {

   @Id @GeneratedValue
   private long id;
```

```
   private String texto;
   private Date fecha;
   @ManyToOne
   private Autor autor;

   public Mensaje() { }
   public long getId() { return id; }
   public void setId(long id) { this.id = id; }
   public String getTexto() { return texto; }
   public void setTexto(String texto) { this.texto = texto; }
   public Autor getAutor() { return autor; }
   public void setAutor(Autor autor) { this.autor = autor; }
   public Date getFecha() { return fecha; }
   public void setFecha(Date fecha) { this.fecha = fecha; }
}
```

El fichero `Mensaje.java` define la clase entidad `Mensaje`. La entidad tiene los atributos `id` (identificador único del mensaje), `texto` (el texto del mensaje) y `autor` (el autor del mensaje, una instancia entidad de tipo `Autor` con la que se define la relación inversa a la definida en autor).

A continuación el fichero `HolaMundo.java` define el programa principal que usa las entidades anteriores. Se pide al usuario un mensaje y un nombre de autor, se busca el nombre de autor y si no existe se crea uno nuevo y se crea el nuevo mensaje asociado al autor indicado. Por último, se listan todos los mensajes asociados a ese autor.

Fichero `es/ua/jtech/jpa/HolaMundo.java`

```
package es.ua.jtech.jpa;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Collection;
import java.util.Date;
import java.util.Iterator;

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

public class HolaMundo {
   public static void main(String[] args) {
      String mensStr;
      String autorStr;

      EntityManagerFactory emf = Persistence
                  .createEntityManagerFactory("simplejpa");

      try {
         BufferedReader in = new BufferedReader(
               new InputStreamReader(System.in));
         System.out.print("Nombre: ");
         autorStr = in.readLine();
         System.out.print("Mensaje: ");
         mensStr = in.readLine();
```

```
      } catch (IOException e) {
         autorStr = "Error";
         mensStr = "Error";
      }

      // Primera operación

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

      // Buscamos la entidad Autor en la BD y la
      // creamos si no existe

      Autor autor = em.find(Autor.class, autorStr);
      if (autor == null) {
         autor = new Autor();
         autor.setNombre(autorStr);
         autor.setCorreo(autorStr + "@ua.es");
         em.persist(autor);
      }

      // Creamos la entidad Mensaje

      Mensaje mensaje = new Mensaje();
      mensaje.setTexto(mensStr);
      mensaje.setFecha(new Date());
      mensaje.setAutor(autor);
      em.persist(mensaje);

      // Una vez que las entidades están creadas, actualizamos
      // la colección de mensajes del autor

      autor.addMensaje(mensaje);
      em.getTransaction().commit();
      System.out.println("Mensaje y autor añadidos");
            System.out.println("El autor " + autor.getNombre() + "
ha escrito "
                        + autor.getMensajes().size() + "
mensajes\n");

      em.close();

      // Suponemos que ha pasado un tiempo y que hacemos
      // una segunda operación. Consultamos los mensajes de un
      // autor utilizando la colección de mensajes.
      // JPA hace una query automáticamente.

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

      autor = em.find(Autor.class, autor.getNombre());
      Collection<Mensaje> mensajes = autor.getMensajes();

      System.out.println("Mensajes:");

      Iterator<Mensaje> it = mensajes.iterator();
      while (it.hasNext()) {
         Mensaje mens = it.next();
         System.out.println(mens.getTexto() + " - "
```

```
                + mens.getFecha().toString());
        }
        em.getTransaction().commit();

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

Por último, el fichero `persistence.xml` es el fichero de configuración de JPA. En él se especifica el driver SQL que se utiliza, la URL de la conexión a la base de datos, el gestor de base de datos (MySQL) y se configura el pool de conexiones `c3p0` implementado por Hibernate. Este fichero de configuración debe encontrarse en el directorio `META-INF` del classpath.

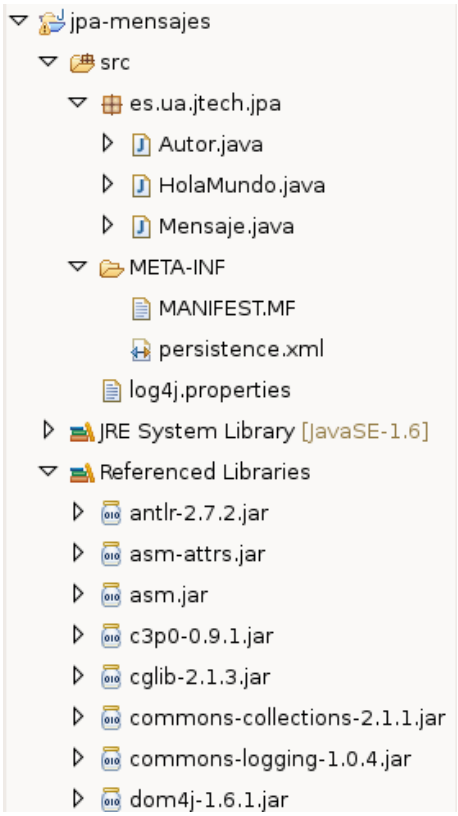
Al poner la propiedad `hibernate.hbm2ddl.auto` a `update` estamos indicando que Hibernate cree automáticamente las tablas necesarias para el mapeo de las entidades. Y la propiedad `hibernate.show_sql` hace que Hibernate muestre por la salida estándar las sentencias SQL que se lanzan al proveedor de base de datos.

Fichero `META-INF/persistence.xml`:

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
version="1.0">

<persistence-unit name="simplejpa">
   <properties>
      <property name="hibernate.archive.autodetection"
                value="class, hbm" />
      <property name="hibernate.connection.driver_class"
                value="com.mysql.jdbc.Driver" />
      <property name="hibernate.connection.url"
                value="jdbc:mysql://localhost:3306/jpa" />
      <property name="hibernate.connection.username"
                value="root" />
      <property name="hibernate.connection.password"
                value="especialista" />
      <property name="hibernate.c3p0.min_size"
                value="5" />
      <property name="hibernate.c3p0.max_size"
                value="20" />
      <property name="hibernate.c3p0.timeout"
                value="300" />
      <property name="hibernate.c3p0.max_statements"
                value="50" />
      <property name="hibernate.c3p0.idle_test_period"
                value="3000" />
      <property name="hibernate.dialect"
                value="org.hibernate.dialect.MySQL5Dialect" />
      <property name="hibernate.hbm2ddl.auto"
                value="update" />
      <property name="hibernate.show_sql"
                    value="true"/>
```

```
          <property name="hibernate.format_sql"
                    value="false"/>
   </properties>
</persistence-unit>
</persistence>
```

Para poder ejecutar el ejemplo en Eclipse, debemos crear un proyecto, incluir en él las librerías de Hibernate JPA (disponibles en esta dirección) y configurar en Eclipse la siguiente estructura de directorios:



Por completitud, a continuación listamos el fichero de configuración de Log4Java `log4j.properties` que debe estar en el CLASSPATH y con el que se configuran los mensajes de *log* generados por Hibernate:

Fichero `log4j.properties`:

```
# Direct log messages to stdout
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target=System.out
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE} %5p
%c{1}:%L - %m%n

# Root logger option
log4j.rootLogger=WARN, stdout
```

```
# Hibernate logging options (INFO only shows startup messages)
#log4j.logger.org.hibernate=INFO

# Log JDBC bind parameter runtime arguments
#log4j.logger.org.hibernate.type=DEBUG
```

Para ejecutar la aplicación en Eclipse hay que lanzar la clase `HolaMundo` como una aplicación Java.

1.3. Anatomía de la aplicación JPA

Terminamos la sesión explicando los elementos de JPA más importantes presentes en el ejemplo anterior. Más adelante profundizaremos en estos conceptos.

1.3.1. Entidades

En la aplicación se definen dos entidades: `Mensaje` y `Autor`. El siguiente código define la entidad `Mensaje`.

```
package es.ua.jtech.jpa;

import java.util.Date;

import javax.persistence.*;

@Entity
public class Mensaje {

   @Id @GeneratedValue
   private long id;
   private String texto;
   private Date fecha;
   @ManyToOne
   private Autor autor;

   public Mensaje() { }
   public long getId() { return id; }
   public void setId(long id) { this.id = id; }
   public String getTexto() { return texto; }
   public void setTexto(String texto) { this.texto = texto; }
   public Autor getAutor() { return autor; }
   public void setAutor(Autor autor) { this.autor = autor; }
   public Date getFecha() { return fecha; }
   public void setFecha(Date fecha) { this.fecha = fecha; }
}
```

Vemos que la entidad se define como un JavaBean: una clase Java con un conjunto de atributos privados con sus métodos de acceso (*getters* y *setters*) y con un constructor sin parámetros. La definición como entidad JPA se realiza con la anotación `@Entity`. Esta anotación hará que la clase se mapee automáticamente en una tabla en la base de datos definida por el gestor JPA. JPA también realiza automática la conversión de los atributos

de la entidad a columnas de la tabla (aunque podríamos especificar con anotaciones los nombres de las columnas asociadas a cada atributo; lo veremos más adelante).

En el ejemplo definimos los siguientes atributos en la entidad `Mensaje`.

- **`Long id`**: identificador único que va a tener cada `mensaje`. Las instancias entidad deben ser únicas y debe existir algún atributo que las identifique de forma inequívoca. Este campo juega este papel. El hecho de definir el tipo del atributo como `Long` (una clase Java, en lugar del tipo primitivo `long`) permite que en la variable de instancia podamos guardar `null` (aunque no en este caso, porque estamos definiendo una clave primaria que no puede estar vacía). Vemos también en el código las anotaciones de JPA `@Id` y `@GeneratedValue`. La primera identifica este campo como el que define la identidad de las instancias e indica la clave primaria en la tabla generada por el mapeo entidad-relación (lo veremos más adelante). La segunda anotación indica que el valor del atributo se genera de forma automática cada vez que se crea una nueva instancia.
- **`String texto`**: texto del mensaje. Se trata de un atributo de tipo `String`.
- **`Date fecha`**: fecha de creación del mensaje. Se trata de un atributo de tipo `java.util.Date` que se mapea con una columna de la tabla SQL de tipo `DATETIME`. El tipo concreto de fecha SQL depende del gestor de base de datos que utilicemos (Mysql, Oracle, etc.); el gestor de persistencia de Hibernate es convierte el tipo Java al tipo SQL correspondiente.
- **`Autor autor`**: autor del mensaje. Con este atributo se está definiendo una relación con otra entidad, indicada por el tipo del atributo (un `Autor`). La anotación `@ManyToOne` especifica que existe una relación muchos-a-uno entre mensajes y autores (muchos mensajes pueden pertenecer al mismo autor).

La tabla generada en el mapeo entidad-relación es la siguiente (utilizando el gestor de base de datos MySQL):

```
CREATE TABLE Mensaje (
  id bigint(20) NOT NULL auto_increment,
  texto varchar(255) default NULL,
  fecha datetime default NULL,
  autor_nombre varchar(255) default NULL,
  PRIMARY KEY  (id),
  KEY FK9BDD22BF8FD5366D (autor_nombre),
  CONSTRAINT FK9BDD22BF8FD5366D FOREIGN KEY (autor_nombre)
REFERENCES autor (nombre)
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=latin1;
```

La columna `autor_nombre` define una clave ajena a la otra tabla `Autor` definida en la aplicación.

1.3.2. Relaciones

En la aplicación ejemplo definimos una relación una-a-muchos bidireccional entre mensajes y autores. Un autor puede haber escrito muchos mensajes.

La relación es bidireccional porque podemos obtener el autor asociado a un mensaje en

las instancias `mensaje`, y podemos obtener la colección de mensajes asociados a un autor en las instancias `autor`. Para ello en la entidad `Mensaje` definimos el método `getAutor` y en la entidad `Autor` definimos el método `getMensajes` que devuelve la colección de mensajes asociada al autor.

En las dos clases implicadas en la relación debemos marcar con anotaciones los atributos que intervienen en la misma. La entidad `Mensaje` es la *propietaria* de la relación y contiene la clave ajena a la entidad `Autor`. Esto lo indicamos con la anotación `@ManyToOne` que indica que muchos mensajes pueden pertenecer al mismo autor. El método `setAutor` asocia un mensaje con un autor. Hay que pasarle como parámetro una instancia entidad.

```
@Entity
public class Mensaje {
   @Id
   @GeneratedValue
   private Long id;
   private String texto;
   @ManyToOne
   private Autor autor;

   ...

   public void setAutor(Autor autor) { this.autor = autor; }
   public Autor getAutor() { return this.autor  ; }
}
```

Por otro lado, en la entidad `Autor` declaramos el atributo `mensajes` de tipo `Set<Mensaje>` precedido por la anotación `@OneToMany(mappedBy = "autor")`. Lo inicializamos a un `HashSet` vacío, para indicar que no pueden haber elementos repetidos en la colección. De esta forma definimos la relación inversa a la anterior. El atributo `mappedBy="autor"` indica que la clave ajena en la entidad `Mensaje` es el campo `autor`. Se define también el método de acceso `getMensajes` que devuelve los mensajes asociados al autor. JPA transformará este método en una consulta SQL que realiza la búsqueda en la tabla de mensajes. Y se define el método de actualización `setMensajes` con el que inicializar la colección de mensajes.

```
@Entity
public class Autor {
   @Id
   private String nombre;
   private String correo;
   @OneToMany(mappedBy = "autor", cascade=CascadeType.ALL)
   private Set<Mensaje> mensajes = new HashSet();

   ...

   public Set<Mensaje> getMensajes() { return this.mensajes; }
   public void setMensajes(Set<Mensaje> mensajes) { this.mensajes =
mensajes}
   }
```

Para añadir un nuevo mensaje a un autor hay que obtener su colección de mensajes y

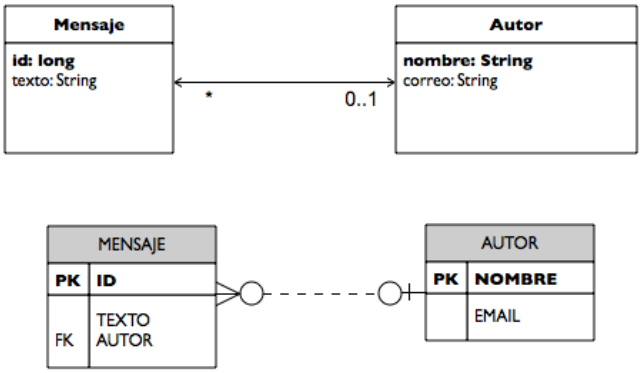
añadirle (con el método `add` de la colección) el nuevo mensaje.

Hay que hacer notar que en lo referido a la persistencia, el único campo que es importante (es el que se guarda en la base de datos) es el atributo `autor` de la entidad `Mensaje`. Bastaría con actualizar ese campo para que la relación en se actualizara en la base de datos. Sin embargo, en la aplicación debemos asegurarnos de que el mensaje también se añada en el campo del autor, para que la relación se mantenga correctamente en memoria. El siguiente fragmento de código de la aplicación es el que hace esta actualización:

```
mensaje.setAutor(autor);
Set<Mensaje> mensajes = autor.getMensajes();
mensajes.add(mensaje);
```

Es exactamente el mismo código que escribiríamos si las entidades fueran objetos normales Java no persistentes.

La siguiente figura muestra la relación muchos-a-uno entre mensajes y autores y las tablas en las que se mapea. Notar que el elemento `mappedBy` definido en la entidad `Autor` es el que define la clave ajena en la entidad propietaria de la relación (el atributo `autor` en la entidad `Mensaje`).



1.3.3. Unidad de trabajo JPA

El elemento fundamental del framework JPA es la clase `EntityManager`. De forma similar a las conexiones JDBC, los objetos de esta clase son los encargados de gestionar la persistencia de las entidades declaradas en la aplicación. La unidad de trabajo habitual en JPA con Java SE consiste en: (1) crear un *entity manager*, (2) comenzar una transacción, (3) realizar operaciones sobre las entidades, (4) cerrar la transacción y (5) cerrar el *entity manager*. Todas las entidades que se crean en un entity manager son gestionadas por él y viven en su *contexto de persistencia*. Cuando el entity manager se cierra, las entidades siguen existiendo como objetos Java, pero a partir de ese momento se encuentran desconectadas (*detached*) de la base de datos.

Los cambios en las entidades no se propagan automáticamente a la base de datos. Es

cuando se realiza un commit de la transacción cuando JPA chequea el contexto de persistencia, detecta los cambios que se han producido en las entidades, utiliza el proveedor de persistencia para generar las sentencias SQL asociadas a los cambios y vuelca (*flush*) esas sentencias en la base de datos.

A continuación copiamos un ejemplo de una unidad de trabajo JPA. En una aplicación es habitual hacer una unidad de trabajo completo en cada caso de uso. Hibernate define un *pool* de entity managers, de forma que su creación no es costosa.

```
// (1) Creamos el Entity Manager
EntityManager em = emf.createEntityManager();

// (2) Creamos la transacción
em.getTransaction().begin();

// (3) Operaciones sobre las entidades
Autor autor = em.find(Autor.class, "ritchie");
Mensaje mensaje = new Mensaje("Hola mundo");
em.persist(mensaje);
mensaje.setAutor(autor);
autor.getMensajes().add(mensaje);

// (4) Cerramos la transaccion
em.getTransaction().commit();

// (5) Cerramos el Entity Manager
em.close();
```

1.4. Aplicación web ejemplo

Vamos a terminar convirtiendo el ejemplo en una aplicación web. Definimos el servlet `AddMensajeServlet` que procesa una petición con los parámetros `mensaje` y `autor`. El servlet realiza la unidad de trabajo JPA de la misma forma que vimos en la aplicación de escritorio, obteniendo las entidades persistentes y añadiendo el mensaje y el autor. Una vez terminada la transacción y hechos persistentes los cambios, el servlet coloca la entidad `Autor` y la colección de entidades `Mensaje` en la petición y la redirige a la página JSP `listaMensaje.jsp` que es la encargada de mostrar el autor y su lista de mensajes.

El siguiente listado muestra el código del servlet `AddMensajeServlet.java`:

Fichero **AddMensajeServlet.java**:

```
package es.ua.jtech.jpa.servlet;

import java.io.*;
import java.util.Collection;

import javax.persistence.*;
import javax.servlet.*;
import javax.servlet.http.*;


public class AddMensajeServlet extends HttpServlet {
```

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

   public void doGet(HttpServletRequest request,
         HttpServletResponse response)
   throws ServletException, IOException   {

      String mensStr =
         request.getParameter("mensaje");
      String autorStr =
         request.getParameter("autor");

      // El EMF lo deberíamos abrir al iniciar la aplicación.
      // Veremos cómo hacerlo en el proyecto de integración.
      EntityManagerFactory emf = Persistence
                        .createEntityManagerFactory("simplejpa");

      // Comienza la sesión

      EntityManager em = emf.createEntityManager();

      em.getTransaction().begin();
      Autor autor = em.find(Autor.class, autorStr);
      if (autor == null) {
         autor = new Autor();
         autor.setNombre(autorStr);
         autor.setCorreo(autorStr+"@ua.es");
         em.persist(autor);
      }
      Mensaje mensaje = new Mensaje();
      mensaje.setTexto(mensStr);
      mensaje.setFecha(new Date());
      mensaje.setAutor(autor);
      em.persist(mensaje);

      autor.addMensaje(mensaje);
      em.getTransaction().commit();

      Collection<Mensaje> mensajes = autor.getMensajes();

      // Cerramos la sesión
      em.close();

      // El emf lo deberíamos cerrar abierto mientras que la
aplicación
      // web esté funcionando
      emf.close();

      // Una vez cerrada la sesión pasamos el control a la capa
      // de presentación para que pinte los resultados.
      // Cuidado con las relaciones entre entidades, porque ahora
      // están desconectadas.

      request.setAttribute("autor", autor);
      request.setAttribute("mensajes", mensajes);

getServletContext().getRequestDispatcher("/listaMensajes.jsp")
                        .forward(request, response);
      }
}
```

La página inicial HTML en la que el usuario introduce el nombre del autor y el texto del mensaje es la siguiente:

Fichero **index.html**

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
      <title>Añadir mensaje</title>
   </head>
<body>

   <form action="servlet/AddMensajeServlet">
      Autor: <input type="text" name="autor"><br>
      Mensaje:  <input type="text" name="mensaje"> <br>
     <input type="submit" value="Enviar">
   </form>
</body>
</html>
```

El siguiente código muestra la página JSP `listaMensajes.jsp` que muestra el autor y sus mensajes. Utiliza la librería de tags JSTL.

Fichero **listaMensajes.jsp**

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
        <%@ taglib uri="http://java.sun.com/jsp/jstl/core"
prefix="c"%>
        <html>
          <head>
            <title>Lista Mensajes</title>
          </head>
             <body>
            <table>
              <thead>
                <tr>
                  <th>Autor</th>
                  <th>Mensaje</th>
                  <th>Fecha</th>
                </tr>
             </thead>
              <tbody>
                <c:forEach items="${mensajes}" var="mens">
                  <tr>
                    <td><c:out value="${autor.nombre}"/></td>
                    <td><c:out value="${mens.texto}"/></td>
                    <td><c:out value="${mens.fecha}"/></td>
                  </tr>
                </c:forEach>
              </tbody>
            </table>
          </body>
        </html>
```

Y, por último, la parte del fichero `web.xml` que mapea el servlet es la siguiente:

Fichero **web.xml**:

```
<servlet>
   <servlet-name>AddMensajeServlet</servlet-name>
   <servlet-class>es.ua.jtech.jpa.AddMensajeServlet</servlet-class>
</servlet>

<servlet-mapping>
   <servlet-name>AddMensajeServlet</servlet-name>
   <url-pattern>/AddMensajeServlet</url-pattern>
</servlet-mapping>
```

2. Ejercicios sesión 1: Introducción a JPA

2.1. Puesta en marcha de la aplicación de escritorio HolaMundo

1. Crea el proyecto `jpa-mensajes` con la aplicación `HolaMundo` vista en la sesión de teoría. Necesitarás las librerías con la implementación de JPA de Hibernate. Las puedes encontrar en este enlace.

2. La aplicación crea todas las tablas en la base de datos `jpa`. Utiliza el administrador de MySQL para crear ese nuevo esquema.

3. Prueba a ejecutar varias veces la aplicación, introduciendo varios autores y mensajes. Comprueba con el QueryBrowser de MySQL las tablas y los datos creados.Comprueba en la salida estándar las sentencias SQL ejecutadas por Hibernate.

4. Escribe en un fichero `respuestas-sesion1.txt` las respuestas a las preguntas que hay a continuación.

5. ¿Qué tablas se han creado? ¿Cómo se ha mapeado la relación uno-a-muchos entre autor y mensajes?

6. Vamos a comprobar el funcionamiento de la anotación `cascade`. Modifica el código de la aplicación `HolaMundo` para que no se haga una llamada al método `persist` del mensaje. ¿Se sigue creando el nuevo mensaje? Comenta ahora la anotación `cascade` y pruébalo de nuevo. Vuelve por último a dejar la anotación activa.

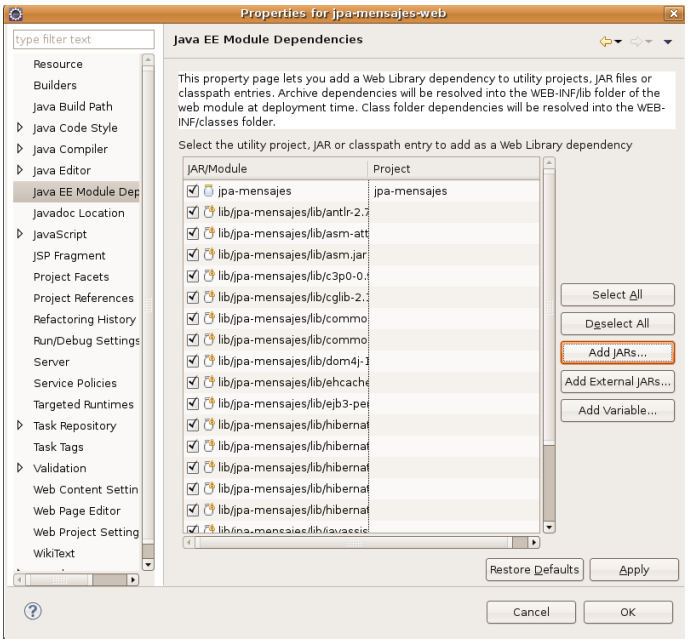
Escribe un programa `DeleteAutor` que pida un nombre de autor por la entrada estándar y lo borre. Comprueba que se borran en cascada todos los mensajes asociados a ese autor. Para borrar el autor debes llamar al método `remove()` del entity manager:

```
em.remove(autor);
```

2.2. Puesta en marcha del servlet AddMensaje

1. Vamos a crear el proyecto web `jpa-mensajes-web` con la aplicación web vista en la sesión de teoría. Crea un proyecto web dinámico (versión 2.5). Necesitarás las librerías JSTL `standard.jar` y `jstl.jar`. Si no las tienes las puedes encontrar en este enlace. Guarda estos JARs en el directorio `WEB-INF/ib`. Ahora vamos a enlazar (no copiar) en este nuevo proyecto todas las librerías de JPA y las clases del proyecto anterior.

2. Para enlazar las librerías del proyecto `jpa-mensajes` debemos hacer lo mismo que hicimos en el proyecto de integración entre el proyecto común y el proyecto web (ver el apartado creación del proyecto web de la sesión 3 del proyecto de integración): añadimos el proyecto `jpa-mensajes` en el classpath del proyecto web y en las *Java EE Module Dependencies* añadimos también el mismo proyecto así como todas las librerías de JPA.



4. Una vez configuradas las librerías se habrán importado todas las clases Java y el fichero `etc/META-INF/persistence.xml` en la aplicación web, listas para ser desplegadas en el servidor web. Creamos ahora el servlet `AddMensajeServlet` y las páginas `index.html` y `listaMensajes.jsp` tal y como están definidas en los apuntes de teoría. Creamos también un servidor en el que desplegar el proyecto.

5. Despliega el proyecto en el servidor web y carga la página **http://localhost:8080/jpa-sesion1-web/**. Prueba a dar de alta un autor y un mensaje. La página JSP debe mostrar un listado de los mensajes del autor introducido, incluyendo el nuevo mensaje. Al igual en el proyecto anterior, podrás ver (esta vez en la salida de la consola) las sentencias SQL que lanza Hibernate a la base de datos. Repite las pruebas varias veces para comprobar que todo funciona correctamente.

3. Conceptos básicos de JPA

3.1. Entidades

Como hemos visto en la sesión anterior, una entidad es esencialmente un nombre, o un conjunto de estados (atributos) asociados juntos en una unidad. Puede participar en relaciones con otras entidades, en las que una entidad puede estar relacionada con una o muchas instancias de otra entidad.

Las entidades integran las ventajas de la Programación Orientada a Objetos y de las bases de datos. La característica más importante de una entidad es su persistencia. ¿Qué otras características podemos destacar en las entidades?

3.1.1. Persistencia

La primera y más importante característica de las entidades es que son persistentes. Esto quiere decir que su estado puede volcarse un almacén de datos y que puede ser consultado con posterioridad, quizás justo después del proceso que ha creado la entidad.

Podríamos llamarlas objetos persistentes, y mucha gente lo hace, pero no es técnicamente correcto. Hablando estrictamente, un objeto persistente se vuelve persistente en el momento en que se instancia. Si existe un objeto persistente, entonces por definición ya es persistente. Una entidad, sin embargo, es persistente porque *puede* guardarse en un almacén persistente. La diferencia es que no se hace persistente automáticamente, sino que debemos invocar a algún método del API para iniciar el proceso. Esto representa una distinción importante, ya que deja el control sobre la persistencia en manos de la aplicación. De esta forma se ofrece a la aplicación la flexibilidad de manipular los datos, realizar lógica de negocio sobre la entidad y decidir cuándo es el momento correcto de efectuar la persistencia. O sea, que las entidades pueden manipularse sin tener necesariamente repercusiones persistentes.

3.1.2. Identidad

Como cualquier objeto Java, una entidad tiene una identidad que lo distingue de otros objetos, pero cuando existen en el almacén de datos también tiene una identidad persistente. La identidad persistente, definida por el identificador (id) de la entidad, es la clave única que identifica a un instancia entidad y la diferencia de otras instancias del mismo tipo de entidad.

Una entidad tiene una identidad persistente cuando existe una representación de ella en el almacén de datos, esto es, una fila en una tabla de una base de datos. Si no está en la base de datos, entonces incluso aunque en memoria tenga su campo identidad definido, no tienen una identidad persistente. El identificador de la entidad es, por tanto, equivalente a

la clave primaria de la tabla de la base de datos en la que se almacena la entidad.

3.1.3. Transaccionalidad

Las entidades son transaccionales. Normalmente se crean, actualizan y borran dentro de una transacción, y se requiere una transacción para que los cambios se actualicen (*commit*) en la base de datos. Los cambios realizados en la base de datos o bien fallan o bien tienen éxito de forma atómica.

En memoria, sin embargo, la historia es diferente. Las entidades pueden cambiarse sin que los cambios sean ni siquiera hechos persistentes. Incluso cuando está incluida en una transacción una entidad puede quedar en un estado inconsistente si se produce un *rollback* o falla la transacción. Las entidades en memoria son simples objetos Java que obedecen todas las reglas y restricciones que la máquina virtual Java aplica a todos los objetos.

3.1.4. Granularidad

También podemos aprender cosas sobre las entidades describiendo lo que *no* son. No son objetos primitivos, ni *wrappers*. Un `string`, por ejemplo, no puede ser una entidad. Su granularidad es demasiado fina para ser parte de un dominio dado. Sin embargo, un `string` será parte normalmente de una entidad en forma del tipo de datos de alguno de sus atributos.

Las entidades son objetos de grano fino que tienen un conjunto de atributos que normalmente se almacenan en un único lugar, como una fila de una tabla. ¿Cuál es la dimensión correcta de una entidad? Varía según la aplicación, pero definitivamente no es correcto ni hacer una entidad de 100 columnas, ni una entidad de una única columna. En general, lo más habitual es utilizar entidades más bien pequeñas, en forma de objetos ligeros, relacionados unos con otros, que pueden ser gestionados eficientemente por la aplicación.

3.1.5. Metadatos

Los metadatos de la entidad son características asociadas a la entidad necesarias para su configuración. Son usados por JPA para reconocer, interpretar y gestionar correctamente la entidad desde el momento en que se carga hasta el momento de invocación en tiempo de ejecución.

Deben ser unos datos mínimos, pero la forma de definirlos debe ser flexible para poder configurar opciones avanzadas. En JPA existen dos posibles formas de definir los metadatos: anotaciones y XML. Hasta la llegada de JPA, cuando la herramienta de persistencia más común era Hibernate, los ficheros XML eran la forma más habitual de configurar los metadatos de las entidades. En JPA se ha hecho un gran esfuerzo en simplificar la configuración de las entidades y se ha dado gran importancia a las anotaciones (aunque se sigue manteniendo la posibilidad de utilizar ficheros XML).

Las anotaciones se introdujeron en Java SE 5 y se han convertido en una parte fundamental de la especificación Java EE, en la que se utilizan para especificar los metadatos de las entidades JPA y de los *Enterprise JavaBeans*. Las anotaciones son sentencias con parámetros opcionales que se colocan justo antes de elementos de programación Java como clases, métodos, campos y variables. A continuación mostramos un ejemplo sencillo con bastantes anotaciones JPA, en el que se define una entidad con un identificador autogenerado y con un atributo asociado a una columna específica de la base de datos.

```
@Entity
public class Empleado {
   @Id @GeneratedValue
   private int id;
   @Column(name="E_NOMBRE")
   private String nombre;
}
```

El compilador Java procesa las anotaciones, y las añade a los ficheros de clases o las elimina, dependiendo del tipo de anotación. Cuando se mantienen en el fichero de clases, pueden consultarse en tiempo de ejecución mediante un API basada en la reflexión.

Junto al uso de anotaciones, otro principio determinante de la simplicidad de JPA es la idea de *configurar excepcionalmente* (*configuration by exception* en inglés). Esto significa que el motor de persistencia de JPA define opciones por defecto que funcionan correctamente en la mayoría de las ocasiones, y que los usuarios necesitan realizar una configuración explícita sólo cuando necesiten modificar el valor por defecto. En otras palabras, el que el usuario deba proporcionar valores de configuración debe ser la excepción, no la regla.

El uso extendido de los valores por defecto tiene, sin embargo, un coste. Cuando los valores por defecto se incluyen en el API y no tienen que ser especificados, tampoco son visibles ni evidentes a los usuarios. Éstos pueden olvidar que están ahí y que el buen funcionamiento de la aplicación se debe a ellos, haciendo algo más complicado realizar una depuración o modificar el comportamiento de la aplicación cuando sea necesario.

Los valores por defecto no deberían, por tanto, servir para hacer olvidar a los usuarios las complejidades del desarrollo de aplicaciones con entidades persistentes. Deberían servir para permitir al desarrollador comenzar fácil y rápidamente con algo que funcione y para después poder ir modificándolo iterativamente conforme el proyecto lo requiere.

3.1.6. Definición de entidades en JPA

Vimos en la sesión anterior varios ejemplos de entidades. Vamos a detallar algo su construcción.

Vamos a comenzar con una sencilla clase Java con un constructor sin argumentos. Por ejemplo, una clase normal Java que define un empleado:

```
public class Empleado {
   private int id;
   private String nombre;
   private long salario;

   public Empleado() {}
   public Empleado(int id) { this.id = id; }

   public int getId() { return id; }
   public void setId(int id) { this.id = id; }
   public String getNombre() { return nombre; }
   public void setNombre(String nombre) { this.nombre = nombre; }
   public long getSalario() { return salario; }
   public void setSalario(long salario) { this.salario = salario; }
}
```

Como ya habrás notado, estamos definiendo una clase al estilo JavaBean con tres propieades: `id`, `nombre` y `salario`. Cada una de estas propiedades se representan por un par de métodos de acceso para obtener y definir la propiedad y está respaldada por una variable de instancia privada en la clase. Estas propiedades o atributos son las unidades de estado dentro de la entidad que deseamos hacer persistentes en el almacén de datos.

Para convertir `Empleado` en una entidad debemos comenzar por anotar la clase con `@Entity`. Este marcador indica al motor de persistencia que la clase es una entidad.

La segunda anotación que debemos añadir es `@Id`. Esta anotación define que una propiedad o atributo particular es la que va a contener la identidad persistente de la entidad (la clave primaria) y es necesaria para que motor conozca qué campo usar como clave primaria en la tabla. Esta anotación debe colocarse o bien antes de la definición del campo o en el método de acceso. Normalmente utilizaremos la primera forma. De esta forma, la entidad que definida como sigue:

```
@Entity
public class Empleado {
   @Id
   private int id;
   private String nombre;
   private long salario;
   ...
}
```

Los campos de la entidad son hechos persistentes automáticamente utilizando el principio comentado de usar valores por defecto. ¿Cuáles son estos valores por defecto?.

Para contestar a la pregunta debemos profundizar en la anotación `@Entity`. Allí nos encontramos un elemento llamado `name` que identifica de forma única el tipo de la entidad. Este elemento puede ser definido explícitamente por el programador, como `@Entity(name="Emp")`. El valor por defecto de este elemento es el propio nombre de la entidad (`Empleado`, en este caso). Este valor será utilizado para dar nombre a la tabla en la que se almacenarán los datos de las entidades.

Cada uno de los atributos de la entidad define una columna de la tabla. El nombre de la

columna será por defecto el del propio campo. Es posible cambiar este nombre utilizando el elemento `name` de la anotación (lo veremos en la sesión 3). La posibilidad de especificar los nombres de las tablas y de las columnas de la base de datos resultante del mapeado ORM hace posible reutilizar bases de datos ya existentes y adaptarlas a JPA.

3.2. Entity Manager

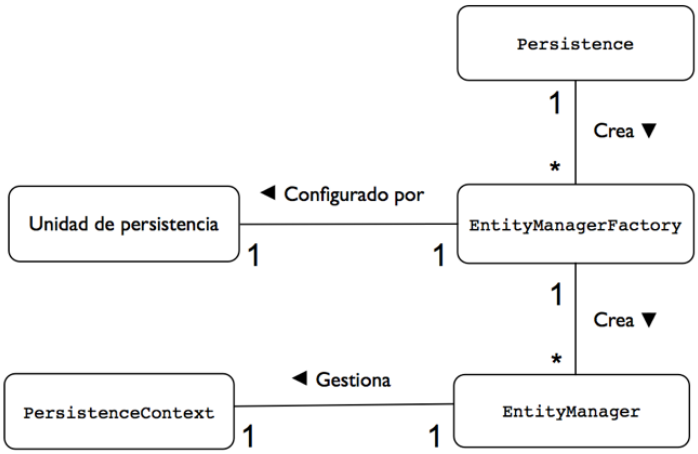
Hemos comentado que es necesario invocar un API para hacer persistentes las entidades en la base de datos. De hecho, son necesarias llamadas a este API para realizar la mayor parte de operaciones sobre las entidades. Este API está implementado por el *gestor de entidades* (`EntityManager` en inglés) y está especificado en una única interfaz llamada `EntityManager` (enlace a javadoc).

Cuando un `EntityManager` obtiene una referencia a una entidad, se dice que la entidad está gestionada (una *managed entity* en inglés) por el `EntityManager`. Al conjunto de entidades gestionadas por un `EntityManager` se le denomina su *contexto de persistencia* (*persistence context* en inglés). Los entity managers trabajan con la base de datos mediante un *proveedor de persistencia* (*persistence provider* en inglés). El proveedor es el que proporciona al `EntityManager` compatibilidad con el tipo de base de datos con la que trabaja a través de la implementación de la interfaz `Query` y de la generación de SQL.

Los entity managers se obtienen a través de factorías del tipo `EntityManagerFactory`. El `EntityManager` se configura mediante la especificación de una *unidad de persistencia* (*persistence unit* en inglés) definida en un fichero de propiedades (XML normalmente) cuyo nombre se pasa a la factoría. La unidad de persistencia define las características concretas de la base de datos con la que van a trabajar todos los gestores de persistencia obtenidos a partir de esa factoría y queda asociada a ella en el momento de su creación. Existe, por tanto, una relación uno-a-uno entre una unidad de persistencia y su `EntityManagerFactory` concreto.

Por último, para obtener una factoría `EntityManagerFactory` debemos llamar a un método estático de la clase `Persistence` .

Las relaciones entre las clases que intervienen en la configuración y en la creación de *entity managers* se muestran en la siguiente figura.



3.2.1. Obtención de un EntityManager

Un `EntityManager` siempre se obtiene a partir de un `EntityManagerFactory`. La forma de obtener la factoría varía dependiendo de si estamos utilizando JPA gestionado por la aplicación (cuando utilizamos JPA en Java SE) o si estamos utilizando un contenedor de persistencia (proporcionado por un servidor de aplicaciones). En el segundo caso se utiliza un método denominado *inyección de dependencias* que analizaremos en detalle en el módulo del Especialista en el que hablemos de la arquitectura EJB. En el primer caso, el que vamos a utilizar durante todo este módulo, debemos usar el método estático `createEntityMaagerFactory()` de la clase `Persistence`. En este método se debe proporcionar el nombre de la unidad de persistencia que vamos a asociar a la factoría. Por ejemplo, para obtener un `EntityManagerFactory` asociado a la unidad de persistencia llamada "SimpleJPA" hay que escribir lo siguiente:

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

El nombre "SimpleJPA" indica el nombre de la unidad de persistencia en la que se especifican los parámetros de configuración de la conexión con la base de datos (URL de la conexión, nombre de la base de datos, usuario, contraseña, gestor de base de datos, características del *pool* de conexiones, etc.). Esta unidad de persistencia se especifica en el fichero estándar de JPA `META-INF/persistence.xml`.

Una vez que tenemos una factoría, podemos obtener fácilmente un `EntityManager`:

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

Esta llamada no es demasiado costosa, ya que las implementaciones de JPA (como Hibernate) 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.

Repetimos a continuación un ejemplo típico de uso que ya hemos visto previamente:

```
public class AutorTest {
   public static void main(String[] args) {
      EntityManagerFactory emf =
         Persistence.createEntityManagerFactory("SimpleJPA");
      EntityManager em = emf.createEntityManager();
      EntityTransaction tx = em.getTransaction();
      tx.begin();
      Autor autor = em.find(Autor.class, "ritchie");
      Mensaje mensaje = new Mensaje("Hola mundo");
      em.persist(mensaje);
      autor.addMensaje(mensaje);
      tx.commit();
      em.close();
   }
}
```

Cuando estamos definiendo las capas de una aplicación las operaciones anteriores suelen implementarse en métodos de los DAO. En cada método del DAO se obtiene un entity manager, se abre una transacción y se utiliza el entity manager y se cierra la transacción y el entity manager. Estas operaciones son operaciones del estilo *findAutorNombre(String nombre)* o *listaMensajesDeAutor(String nombre)*. Cogen como parámetros datos planos y devuelven entidades desconectadas una vez que ha terminado la transacción. Al final de este tema veremos un ejemplo completo de un DAO implementado con JPA.

Es muy importante considerar que los objetos `EntityManager` no son *thread-safe*. Cuando los utilicemos en servlets, por ejemplo, deberemos crearlos en cada petición HTTP. Esto también es correcto, además, para evitar que distintas sesiones accedan al mismo contexto de persistencia. Si queremos que una sesión HTTP utilice un único entity manager, podríamos guardarlo en el objeto `HtttpSession` y acceder a él al comienzo de cada petición. El objeto `EntityManagerFactory` a partir del que obtenemos los entity managers sí que es *thread-safe*. Podemos implementar un *Singleton* al que acceden todos los threads para obtener entity managers.

3.2.2. Haciendo persistente una entidad

Hacer persistente una entidad consiste en tomar una entidad transitoria, que no tiene representación persistente en la base de datos, y almacenar su estado de forma que podamos recuperarlo más tarde. Ésta es la base de la persistencia; crear un estado que pueda sobrevivir al proceso que lo creó. Veamos cómo hacerlo de forma sencilla:

```
Empleado emp = new Empleado(146);
em.persist(emp);
```

El método `persist` es el que se encarga de comenzar a hacer persistente la entidad. Veremos más adelante que la entidad se vuelca realmente a la base de datos cuando se

realiza un *flush* del contexto de persistencia y se generan las instrucciones SQL que se ejecutan en la base de datos (normalmente al hacer un *commit* de la transacción actual). Si el `EntityManager` encuentra algún problema al ejecutar el método, se lanza la excepción no chequeada `PersistenceException`. Cuando termine la ejecución del método, si no se ha producido ninguna excepción, `emp` será a partir de ese momento una entidad gestionada dentro del contexto de persistencia del `EntityManager`. El siguiente código muestra como definir un sencillo método que crea un nuevo empleado y lo hace persistente en la base de datos:

```
public Empleado createEmpleado(int id, String nombre, long salario)
{
   Empleado emp = new Empleado(id);
   emp.setNombre(nombre);
   emp.setSalario(salario);
   em.persist(emp);
   return emp;
}
```

El método asume la existencia de un `EntityManager` en la variable de instancia `em` y lo usa para hacer persistente el empleado recién creado. La llamada a `persist()` podría generar una excepción de tiempo de ejecución de tipo `PersistenceException` que se propagaría al llamador del método.

3.2.3. Búsqueda de entidades

Una vez que la entidad está en la base de datos, lo siguiente que podemos hacer es encontrarla de nuevo. Para ello basta con escribir una línea de código:

```
Empeado emp = em.find(Empleado.class, 146);
```

Pasamos la clase de la entidad que estamos buscando (en el ejemplo estamos buscando una instancia de la clase `Empleado`) y el identificador o clave primaria que identifica la entidad. El entity manager buscará esa entidad en la base de datos y devolverá la instancia buscada. La entidad devuelta será una entidad gestionada que existirá en el contexto de persistencia actual asociado al entity manager.

En el caso en que no existiera ninguna entidad con ese identificador, se devolvería simplemente `null`.

La llamada a `find` puede devolver dos posibles excepciones de tiempo de ejecución, ambas de la clase `PersistenceException`: `IllegalStateException` si el entitiy manager ha sido previamente cerrado o `IllegalArgumentException` si el primer argumento no contiene una clase entidad o el segundo no es el tipo correcto de la clave primaria de la entidad.

El método para buscar entidades de tipo empleado sería tan sencillo como sigue:

```
public Empleado findEmpleado(int id) {
   return em.find(Employee.class, id);
}
```

3.2.4. Borrado de entidades

Un borrado de una entidad realiza una sentencia `DELETE` en la base de datos. Esta acción no es demasiado frecuente, ya que las aplicaciones de gestión normalmente conservan todos los datos obtenidos y marcan como no activos aquellos que quieren dejar fuera de vista de los casos de uso. Se suele utilizar para eliminar datos que se han introducido por error en la base de datos o para trasladar de una tabla a otra los datos (se borra el dato de una y se inserta en la otra). En el caso de entidades esto último sería equivalente a un cambio de tipo de una entidad.

Para eliminar una entidad, la entidad debe estar gestionada, esto es, debe existir en el contexto de persistencia. Esto significa que la aplicación debe obtener la entidad antes de eliminarla. Un ejemplo sencillo es:

```
Empleado emp = em.find(Empleado.class, 146);
em.remove(emp);
```

Un posible problema del código anterior es que podría darse el caso de que la llamada a `find()` no encontrara la entidad, lo que causaría una excepción en la llamada a `remove()`. Podemos incluir un chequeo de que la entidad no es `null` en nuestro método para borrar un empleado:

```
public void removeEmpleado(int id) {
   Empleado emp = em.find(Empleado.class, id);
   if (emp != null) {
      em.remove(emp);
   }
}
```

3.2.5. Actualización de entidades

Una entidad puede ser actualizada en unas pocas formas diferentes, pero veremos ahora el caso más común y sencillo. Es el caso en el que tenemos una entidad gestionada y queremos hacer cambios en ella. Para actualizar una entidad, primero debemos obtenerla para convertirla en gestionada. Después podremos colocar los nuevos valores en sus atributos utilizando los métodos `set` de la entidad. Por ejemplo, supongamos que queremos subir el sueldo del empleado 146 en 1.000 euros. Tendríamos que hacer lo siguiente:

```
Empleado emp = em.find(Empleado.class, 146);
emp.setSueldo(emp.getSueldo() + 1000);
```

Nótese la diferencia con las operaciones anteriores, en las que el `EntityManager` era el responsable de realizar la operación directamente. Aquí no llamamos al `EntityManager` sino a la propia entidad. Estamos, por así decirlo, trabajando con una caché de los datos de la base de datos. Posteriormente, cuando se finalice la transacción, el `EntityManager` hará persistentes los cambios mediante las correspondientes sentencias SQL.

Podemos definir un método que aumente el sueldo de un empleado en una determinada cantidad:

```
public Empleado subeSueldoEmpleado(int id, long aumento) {
   Empleado emp = em.find(Empleado.class, id);
   if (emp != null) {
      emp.setSueldo(emp.getSueldo + aumento);
   }
}
```

La otra forma de actualizar una entidad es con el método `merge()` del `EntityManager`. A este método se le pasa como parámetro una entidad no gestionada. El `EntityManager` busca la entidad en su contexto de persistencia (utilizando su identificador) y actualiza los valores del contexto de persistencia con los de la entidad no gestionada. En el caso en que la entidad no existiera en el contexto de persistencia, se crea con los valores que lleva la entidad no gestionada.

Veamos el siguiente código para aclararlo un poco.

```
em.getTransaction().begin();
Empleado emp1 = new Empleado();
emp1.setId(200);
emp1.setNombre("Juan");
em.persist(emp1);
Empleado emp2 = new Empleado();
emp2.setId(200);
emp2.setNombre("Luisa");
em.merge(emp2);
em.getTransaction().commit();
```

En el código creamos un empleado con el nombre "Juan" y lo hacemos persistente, con lo que la entidad se incorpora al contexto de persistencia. Después creamos otro empleado con el nombre "Luisa" y el mismo identificador. Al no hacerlo persistente, se trata de una entidad no gestionada (si hubiéramos llamado a `persist()` hubiéramos tenido un error porque ya hay una entidad con ese identificador). Cuando llamamos al método `merge()`, la entidad en el contexto de persistencia (referenciada por la variable `emp1`) es actualizada con los valores de la entidad `emp2` (se actualiza su nombre a "Luisa"). Por último, al grabar la transacción los datos que se hacen persistentes en la base de datos son los últimos actualizados.

3.2.6. Transacciones

Cualquier operación que conlleve una creación, modificación o borrado de entidades debe hacerse dentro de una transacción. En JPA las transacciones se gestionan de forma distinta dependiendo de si estamos en un entorno Java SE o en un entorno Java EE. La diferencia fundamental entre ambos casos es que en un entorno Java EE las transacciones se manejan con JTA (Java Transaction API), un API que implementa el *two face commit* y que permite gestionar operaciones sobre múltiples recursos transaccionales o múltiples operaciones transaccionales sobre el mismo recurso. En el caso de Java SE las

transacciones se implementan con el gestor de transacciones propio del recurso local (la base de datos) y se especifican en la interfaz `EntityTransaction`.

El gestor de transacciones locales se obtiene con la llamada `getTransaction()` al `EntityManager`. Una vez obtenido, podemos pedirle cualquiera de los métodos definidos en la interfaz: `begin()` para comenzar la transacción, `commit()` para actualizar los cambios en la base de datos (en ese momento JPA vuelca las sentencias SQL en la base de datos) o `rollback()` para deshacer la transacción actual.

El siguiente listado muestra un ejemplo de uso de transacciones con los métodos que estamos definiendo:

```
em.getTransaction().begin();
createEmpleado(146, "Juan Garcia", 30000);
em.getTransaction().commit();
```

En los métodos que hemos ido construyendo se asume que la gestión de las transacciones se va a realizar fuera de ellos, que se ejecutan en una transacción ya comenzada y que la aplicación se encargará de hacer el *commit* de la transacción. Esto nos permite mayor flexibilidad en el uso de los métodos, ya que podremos encadenar más de uno de estos métodos en una única transacción.

3.2.7. Queries

Uno de los aspectos fundamentales de JPA es la posibilidad de realizar consultas sobre las entidades, muy similares a las consultas SQL. El lenguaje en el que se realizan las consultas se denomina *Java Persistence Query Language* (JPQL).

Una consulta se implementa mediante un objeto `Query`. Los objetos `Query` se construyen utilizando el `EntityManager` como una factoría. La interfaz `EntityManager` proporciona un conjunto de métodos que devuelven un objeto `Query` nuevo. Veremos algún ejemplo ahora, pero profundizaremos en el tema más adelante.

Una consulta puede ser estática o dinámica. Las consultas estáticas se definen con metadatos en forma de anotaciones o XML, y deben incluir la consulta propiamente dicha y un nombre asignado por el usuario. Este tipo de consulta se denomina una consulta con nombre (*named query* en inglés). El nombre se utiliza en tiempo de ejecución para recuperar la consulta.

Una consulta dinámica puede lanzarse en tiempo de ejecución y no es necesario darle un nombre, sino especificar únicamente las condiciones. Son un poco más costosas de ejecutar, porque el proveedor de persistencia (el gestor de base de datos) no puede realizar ninguna preparación, pero son muy útiles y versátiles porque pueden construirse en función de la lógica del programa, o incluso de los datos proporcionados por el usuario.

El siguiente código muestra un ejemplo de consulta dinámica:

```
Query query = em.createQuery("SELECT e FROM Empleado e " +
                             "WHERE e.sueldo > :sueldo");
query.setParameter("sueldo", 20000);
List emps = query.getResultList();
```

En el ejemplo vemos que, al igual que en JDBC, es posible especificar consultas con parámetros y posteriormente especificar esos parámetros con el método `setParameter()`. Una vez definida la consulta, el método `getResultList()` devuelve la lista de entidades que cumplen la condición. Este método devuelve un objeto que implementa la interfaz `List`, una subinterfaz de `Collection` que soporta ordenación. Hay que notar que no se devuelve una `List<Empleado>` ya que no se pasa ninguna clase en la llamada y no es posible parametrizar el tipo devuelto. Sí que podemos hacer un casting en los valores devueltos por los métodos que implementan las búsquedas, como muestra el siguiente código:

```
public List<Empleado> findEmpleadosSueldo(long sueldo) {
   Query query = em.createQuery("SELECT e FROM Empleado e " +
                                "WHERE e.sueldo > :sueldo");
   query.setParameter("sueldo", 20000);
   return (List<Empleado>) query.getResultList();
}
```

Más adelante veremos las consultas con mayor profundidad.

3.3. Implementando un DAO con JPA

Veamos a continuación una primera implementación de un DAO típico utilizando JPA. En temas sucesivos incluiremos más adelante algunas mejoras en el código. En el DAO los objetos que se pasan como parámetros y que se devuelven son valores no persistentes. El patrón DAO también obliga a operaciones atómicas. En cada método se debe abrir y cerrar el *entity manager*:

```
import javax.persistence.*;
import java.util.List;

public class EmpleadoDAO {
   private EntityManagerFactory emf;

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

   public Empleado createEmpleado(int id, String nombre, long
sueldo)  {
      EntityManager em = emf.createEntityManager();
      em.getTransaction().begin();
      Empleado emp = em.find(Empleado.class, id);
      if (emp == null) {
         emp = new Empleado(id);
      }
      emp.setNombre(nombre);
      emp.setSueldo(sueldo);
```

```
      em.persist(emp);
      em.getTransaction().commit();
      em.close();
      return emp;
   }

   public void removeEmpleado(int id) {
      EntityManager em = emf.createEntityManager();
      em.getTransaction().begin();
      Empleado emp = em.find(Empleado.class, id);
      if (emp != null) {
         em.remove(emp);
      }
      em.getTransaction().commit();
      em.close();
   }

   public void subeSueldoEmpleado(int id, long aumento) {
      EntityManager em = emf.createEntityManager();
      em.getTransaction().begin();
      Empleado emp = em.find(Empleado.class, id);
      if (emp != null) {
         emp.setSueldo(emp.getSueldo() + aumento);
      }
      em.getTransaction().commit();
      em.close();
   }

   public void cambiaNombreEmpleado(int id, String nuevoNombre) {
      EntityManager em = emf.createEntityManager();
      em.getTransaction().begin();
      Empleado emp = em.find(Empleado.class, id);
      if (emp != null) {
         emp.setNombre(nuevoNombre);
      }
      em.getTransaction().commit();
      em.close();
   }

   public Empleado findEmpleado(int id) {
      EntityManager em = emf.createEntityManager();
      em.getTransaction().begin();
      Empleado emp= em.find(Empleado.class, id);
      em.getTransaction().commit();
      em.close();
      return emp;
   }

   public List<Empleado> findEmpleadosSueldoMayorQue(long sueldo) {
      EntityManager em = emf.createEntityManager();
      em.getTransaction().begin();
      Query query = em.createQuery("SELECT e FROM Empleado e " +
                                   "WHERE e.sueldo > :sueldo");
      query.setParameter("sueldo", sueldo);
      List<Empleado> list = (List<Empleado>) query.getResultList();
      em.getTransaction().commit();
      em.close();
      return list;
   }
```

```
}
```

El siguiente listado proporciona un ejemplo de utilización de este DAO:

```
package es.ua.jtech.jpa;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Collection;

public class EmpleadoDAOTest {

   public static void main(String[] args) {
      int numEmpleado=0;
      String nombre=null;

      EntityManagerFactory emf =
Persistence.createEntityManagerFactory("simplejpa");
      EmpleadoDAO empDAO = new EmpleadoDAO(emf);

      try {
         BufferedReader in = new BufferedReader(
               new InputStreamReader(System.in));
         System.out.print("Id: ");
         numEmpleado = Integer.parseInt(in.readLine());
         System.out.print("Nombre: ");
         nombre = in.readLine();
      } catch (IOException e) {
         System.exit(-1);
      }

      // crear y hacer persistente el empleado
      Empleado emp = empDAO.createEmpleado(numEmpleado,nombre,
50000);
      System.out.println("Empleado" + emp + "hecho persistente");

      // busca un empleado
      emp = empDAO.findEmpleado(numEmpleado);
      System.out.println("Encontrado empleado" + emp);

      // lista empleados
      Collection<Empleado> empleados =
empDAO.findEmpleadosSueldoMayorQue(30000);
      System.out.println("Hay " + empleados.size() + "
ejecutivos");
   }
}
```

3.4. Contextos de persistencia

La clave para entender el entity manager es entender el contexto de persistencia. La inclusión o no de una entidad en el contexto de persistencia determinará el resultado de cualquier operación de persistencia. Si el contexto de persistencia participa en una transacción, entonces el estado en memoria de las entidades gestionadas se sincronizará

con la base de datos. Sin embargo, a pesar del importante papel que juega, el contexto de persistencia nunca es realmente visible a la aplicación. Siempre se accede a él indirectamente a través del entity manager y asumimos que está ahí cuando lo necesitamos.

Es también fundamental entender que el contexto de persistencia hace el papel de *caché* de las entidades que están realmente en la base de datos. Cuando actualizamos una instancia en el contexto de persistencia estamos actualizando una caché, una copia que sólo se hace persistente en la base de datos cuando el entity manager realiza un *flush* de las instancias en la base de datos. Simplificando bastante, podemos pensar que el entity manager realiza el siguiente proceso para todas las entidades:

1. Si la aplicación solicita una entidad (mediante un `find`, o accediendo a un atributo de otra entidad en una relación), se comprueba si ya se encuentra en el contexto de persistencia. Si no se ha recuperado previamente, se obtiene la instancia de la entidad de la base de datos.
2. La aplicación utiliza las instancias del contexto de persistencia, accediendo a sus atributos y (posiblemente) modificándolos. Todas las modificaciones se realizan en la memoria, en el contexto de persistencia.
3. En un momento dado (cuando termina la transacción, se ejecuta una query o se hace una llamada al método `flush`) el entity manager comprueba qué entidades han sido modificadas y vuelca los cambios a la base de datos.

Es muy importante darse cuenta de la diferencia entre el contexto de persistencia y la base de datos propiamente dicha. No se encuentran sincronizados hasta que el entity manager vuelca los cambios a la base de datos. La aplicación debe ser consciente de esto y utilizar razonablemente los contextos de persistencia.

Veamos un ejemplo sencillo que puede ilustrar esto. Supongamos que tenemos una relación uno-a-muchos entre `Autor` y `Mensaje`, que la entidad `Mensaje` es la propietaria de la relación (contiene la clave ajena hacia `Autor`) y que la aplicación ejecuta el siguiente código para añadir un mensaje a un autor:

```
em.getTransaction().begin();
Autor autor = em.find(Autor.class, "kirai");
System.out.println(autor.getNombre() + " ha escrito " +
                   autor.getMensajes().size() +
                   " mensajes");
Mensaje mens = new Mensaje("Nuevo mensaje");
mens.setAutor(autor);
em.persist(mens);
em.getTransaction().commit();
System.out.println(autor.getNombre() + " ha escrito " +
                   autor.getMensajes().size() +
                   " mensajes");
```

Si comprobamos qué sucede veremos que no aparecerá ningún cambio entre el primer mensaje de la aplicación y el segundo, ambos mostrarán el mismo número de mensajes. ¿Por qué? ¿Es que no se ha actualizado el nuevo mensaje de Kirai en la base de datos?. Si miramos en la base de datos, comprobamos que la transacción sí que se ha completado

correctamente. Sin embargo cuando llamamos al método `getMensajes()` en la colección resultante no aparece el nuevo mensaje que acabamos de añadir.

Este es un ejemplo del tipo de errores que podemos cometer por trabajar con contextos de persistencia pensando que estamos conectados directamente con la BD. El problema se encuentra en que la primera llamada a `getMensajes()` (antes de crear el nuevo mensaje) ha generado la consulta a la base de datos y ha cargado el resultado en memoria. Cuando hacemos una segunda llamada, el proveedor detecta que esa información ya la tiene en la caché y no la vuelve a consultar.

Una posible solución es hacer que la aplicación modifique el contexto de persistencia para que esté sincronizado con la base de datos. Lo haríamos con el siguiente código:

```
em.getTransaction().begin();
Autor autor = em.find(Autor.class, "kirai");
System.out.println(autor.getNombre() + " ha escrito " +
                   autor.getMensajes().size() +
                   " mensajes");
Mensaje mens = new Mensaje("Nuevo mensaje");
mens.setAutor(autor);
em.persist(mens);
em.getTransaction().commit();
autor.getMensajes().add(Mensaje);
System.out.println(autor.getNombre() + " ha escrito " +
                   autor.getMensajes().size() +
                   " mensajes");
```

La llamada al método `add()` de la colección añade un mensaje nuevo a la colección de mensajes del autor existente en el contexto de persistencia. De esta forma estamos reflejando en memoria lo que hemos realizado en la base de datos. Lo habitual (tal y como hicimos en la sesión 1) es definir en la entidad `Autor` un método de conveniencia `addMensaje(mensaje)` que realice ambas operaciones: añadir el mensaje a la colección de mensajes del autor y establecer el autor del mensaje en el método `setAutor()` del mensaje.

Otra posible solución es obligar al entity manager a que sincronice la entidad y la base de datos. Para ello podemos llamar al método `refresh()` del entity manager:

```
em.getTransaction().begin();
// ... añado un mensaje al autor
em.getTransaction().commit();
em.refresh(autor);
System.out.println(autor.getNombre() + " ha escrito " +
                   autor.getMensajes().size() +
                   " mensajes");
```

Estos ejemplos ponen en evidencia que para trabajar bien con JPA es fundamental entender que el contexto de persistencia es una caché de la base de datos propiamente dicha.

3.4.1. Operaciones y contexto de persistencia

Una vez vistos los conceptos de entity manager y contexto de persistencia, vamos a repasar las distintas operaciones que podemos realizar con las entidades y los entity managers, desde el punto de vista de qué sucede en el contexto de persistencia y cómo se realiza la sincronización con la BD. Introduciremos también algunas operaciones no vistas hasta ahora como `getReference()` y `merge()`.

3.4.1.1. Persist

El método `persist()` del `EntityManager` acepta una nueva instancia de entidad y la convierte en gestionada. Si al entidad que se va a hacer persistir ya está gestionada en el contexto de persistencia, la llamada se ignora. La operación `contains()` puede usarse para comprobar si una entidad está gestionada, pero es muy raro el tener que llamar a esta operación. La aplicación debería saber qué entidades están gestionadas y cuáles no. El diseño de la aplicación define cuándo las entidades pasan a ser gestionadas o desconectadas.

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. Lo más normal es que esto suceda cuando se realiza 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. 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.

Si se llama a `persist()` fuera de una transacción la entidad se incluirá en el contexto de persistencia, pero no se realizará ninguna acción hasta que la transacción comience y el contexto de persistencia se sincronice con 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 existen 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 conexto de persistencia con la base de datos, al encontrar una clave primaria duplicada.

Un ejemplo completo de utilización de `persist()` es el siguiente:

```
Departamento dept = em.find(Departamento.class, 30);
Empleado emp = new Empleado();
emp.setId(53);
emp.setNombre("Pedro");
emp.setDepartamento(dept);
dept.getEmpleados().add(emp);
em.persist(emp);
```

En el ejemplo comenzamos obteniendo una instancia que ya existe en la base de datos de la entidad `Departamento`. Se crea una nueva instancia de `Empleado`, proporcionando la clave primaria y algún atributo. Después asignamos el empleado al departamento,

llamando al método `setDepartamento()` del empleado y pasándole la instancia de `Departamento` que habíamos recuperado. Actualizamos el otro lado de la relación llamando al método `add()` de la colección para que el contexto de persistencia mantenga correctamente la relación bidireccional. Y por último realizamos la llamada al método `persist()` que convierte la entidad en gestionada. Cuando el contexto de persistencia se sincroniza con la base de datos, se añade la nueva entidad en la tabla y se actualiza al mismo tiempo la relación. Hay que hacer notar que sólo se actualiza la tabla de `Empleado`, que es la propietaria de la relación y la que contiene la clave ajena a `Departamento`.

3.4.1.2. Find y GetReference

El método `find()` se utiliza para localizar una entidad por su clave primaria. La llamada devuelve una entidad gestionada asociada al contexto de persistencia del entity manager.

Existe una versión especial de `find()` que se puede usar cuando se quiere añadir un objeto con una clave primaria conocida a una relación. Para ello se puede usar el método `getReference()` del entity manager. Ya que únicamente estamos creando una relación, no es necesario cargar todo el objeto de la base de datos. Sólo se necesita su clave primaria. Veamos la nueva versión del ejemplo anterior:

```
Departamento dept = em.getReference(Departamento.class, 30);
Empleado emp = new Empleado();
emp.setId(53);
emp.setNombre("Pedro");
emp.setDepartamento(dept);
dept.getEmpleados().add(emp);
em.persist(emp);
```

Esta versión es más eficiente que la anterior porque no se realiza ningún `SELECT` en la base de datos para buscar la instancia del `Departamento`. Cuando se llama a `getReference()`, el proveedor devolverá un *proxy* al `Departamento` sin recuperarlo realmente de la base de datos. En tanto que sólo se acceda a la clave primaria, no se recuperará ningún dato. Y cuando se haga persistente el `Empleado`, se guardará en la clave ajena correspondiente el valor de la clave primaria del `Departamento`.

Un posible problema de este método es que, a diferencia de `find()` no devuelve `null` si la instancia no existe, ya que realmente no realiza la búsqueda en la base de datos. Únicamente se debe utilizar el método cuando estamos seguros de que la instancia existe en la base de datos. En caso contrario estaremos guardando en la variable `dept` una referencia (clave primaria) de una entidad que no existe, y cuando se haga persistente el empleado se generará una excepción porque el `Empleado` estará haciendo referencia a una entidad no existente.

En general, la mayoría de las veces llamaremos al método `find()` directamente. Las implementaciones de JPA hacen un buen trabajo con las cachés y si ya tenemos la entidad en el contexto de persistencia no se realiza la consulta a la base de datos.

3.4.1.3. Merge

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. Un ejemplo:

```
public void subeSueldo(Empleado emp, long inc)
   Empleado empGestionado = em.merge(emp);
   empGestionado.setSueldo(empGestionado.getSueldo()+inc);
}
```

Si una entidad con el mismo identificador que `emp` existe en el contexto de persistencia, se devuelve como resultado y se actualizan sus atributos. Si el objeto que se le pasa a `merge()` es un objeto nuevo, se comporta igual que `persist()`, con la única diferencia de que la entidad gestionada es la devuelta como resultado de la llamada.

3.4.1.4. Remove

Borrar una entidad no es una tarea compleja, pero puede requerir algunos pasos, dependiendo del número de relaciones en la entidad que vamos a borrar. En su forma más simple, el borrado de una entidad se realiza pasando la entidad como parámetro del método `remove()` del entity manager que la gestiona. En el momento en que el contexto de persistencia se sincroniza con una transacción y se realiza un commit, la entidad se borra. Hay que tener cuidado, sin embargo, con las relaciones en las que participa la entidad para no comprometer la integridad de la base de datos.

Veamos un sencillo ejemplo. Consideremos las entidades `Empleado` y `Despacho` y supongamos una relación unidireccional uno-a-uno entre `Empleado` y `Despacho` que se mapea utilizando una clave ajena en la tabla `EMPLEADO` hacia la tabla `DESPACHO` (lo veremos en la sesión siguiente). Supongamos el siguiente código dentro de una transacción en el que borramos el despacho de un empleado:

```
Empleado emp = em.find(Empleado.class, empId);
Despacho desp = emp.getDespacho();
em.remove(desp);
```

Cuando se realice un commit de la transacción veremos una sentencia `DELETE` en la tabla `DESPACHO`, pero en ese momento obtendremos una excepción con un error de la base de datos referido a que hemos violado una restricción de la clave ajena. Esto se debe a que existe una restricción de integridad referencial entre la tabla `EMPLEADO` y la tabla `DESPACHO`. Se ha borrado una fila de la tabla `DESPACHO` pero la clave ajena correspondiente en la tabla `EMPLEADO` no se ha puesto a `NULL`. Para corregir el problema, debemos poner explícitamente a `null` el atributo `despacho` de la entidad `Empleado` antes de que la transacción finalice:

```
Empleado emp = em.find(Empleado.class, empId);
Despacho desp = emp.getDespacho();
emp.setDespacho(null);
em.remove(desp);
```

El mantenimiento de las relaciones es una responsabilidad de la aplicación. Casi todos los problemas que suceden en los borrados de entidades tienen relación con este aspecto. Si la entidad que se va a borrar es el objetivo de una clave ajena en otras tablas, entonces debemos limpiar esas claves ajenas antes de borrar la entidad.

3.4.1.5. Clear

En ocasiones puede ser necesario limpiar (`clear`) contexto de persistencia y vaciar las entidades gestiondas. Esto puede suceder, por ejemplo, en conextos extendidos gestionados por la aplicación que han crecido demasiado. Por ejemplo, consideremos el caso de un entity manager gestionado por la aplicación que lanza una consulta que devuelve varios cientos de instancias entidad. Una vez que ya hemos realizado los cambios a unas cuantas de esas instancias y la transacción se termina, se quedan en memoria cientos de objetos que no tenemos intención de cambiar más. Si no queremos cerrar el contexto de persistencia en ese momento, entonces tendremos que limpiar de alguna forma las instancias gestionadas, o el contexto de persistencia irá creciendo cada vez más.

El método `clear()` del interfaz `EntityManager` se utiliza para limpiar el contexto de persistencia. En muchos sentidos su funcionamiento es similar a un rollback de una transacción en memoria. Todas las instancias gestionadas por el contexto e persistencia se desconectan del contexto y quedan con el estado previo a la llamada a `clear()`. La operación `clear()` es del tipo todo o nada. No es posible cancelar selectivamente la gestión de una instancia particular cuando el contexto de persistencia está abierto.

3.4.1.6. Modificación de atributos de la instancia

La última forma con la que podemos modificar relacionados con una entidad en la base de datos es modificando los atributos de una instancia gestionada. En el momento en que se haga un commit de la transacción los cambios se actualizarán en la base de datos mediante una sentencia `UPDATE`.

Es muy importante notar que no está permitido modificar la clave primaria de una entidad gestionada. Si intentamos hacerlo, en el momento de hacer un commit la transacción lanzará una excepción `RollbackException`. Para reforzar esta idea, es conveniente definir las entidades sin un método `set` de la clave primaria. En el caso de aquellas entidades con una generación automática de la clave primaria, ésta se generará en tiempo de creación de la entidad. Y en el caso en que la aplicación tenga que proporcionar la clave primaria, lo puede hacer en el constructor.

3.4.1.7. Operaciones en cascada

Por defecto, las operaciones del entity manager se aplican únicamente a las entidades proporcionadas como argumento. La operación no se propagará a otras entidades que tienen relación con la entidad que se está modificando. Lo hemos visto antes con la llamada a `remove()`. Pero no sucede lo mismo con operaciones como `persist()`. Es bastante probable que si tenemos una entidad nueva y tiene una relación con otra entidad, las dos deben persistir juntas.

Consideremos la secuencia de operaciones del siguiente códgo que muestran cómo se crea un nuevo `Empleado` con una entidad `Direccion` asociada y cómo se hacen los dos persistentes. La segunda llamada a `persist()` sobre la `Direccion` es algo redundante. Una entidad `Direccion` se acopla a la entidad `Empleado` que la almacena y tiene sentido que siempre que se cree un nuevo `Empleado`, se propague en cascada la llamada a `persist()` para la `Direccion`.

```
Empleado emp = new Empleado(12, "Rob");
Direccion dir = new Direccion("Alicante");
emp.setDireccion(dir);
em.persist(emp);
em.persist(dir);
```

El API JPA proporciona un mecanismo para definir cuándo operaciones como `persist()` deben propagarse en cascada. Para ello se define el elemento `cascade` en todas las anotaciones de relaciones (`@OneToOne`, `@OneToMany`, `@ManyToOne` y `@ManyToMany`).

Las operaciones a las que hay que aplicar la propagación se identifican utilizando el tipo enumerado `CasacadeType`, que puede tener como valor `PERSIST`, `REFRESH`, `REMOVE`, `MERGE` y `ALL`.

Persist en cascada

Para activar la propagación de la persistencia en cascada debemos añadir el elemento `cascade=CascadeType.PERSIST` en la declaración de la relación. Por ejemplo, en el caso anterior, si hemos definido una relación muchos-a-uno entre `Empleado` y `Direccion`, podemos escribir el siguiente código:

```
@Entity
public class Empleado {
   // ...
   @ManyToOne(cascadeCascdeType.PERSIST)
   Direccion direccion;
   // ...
}
```

Para invocar la persistencia en cascada sólo nos tenemos que asegurar de que la nueva entidad `Direccion` se ha puesto en el atributo `direccion` del `Empleado` antes de llamar a `persist()` con él. La definición de la operación en cascada es unidireccional, y tenemos que tener en cuenta quién es el propietario de la relación y dónde se va a actualizar la misma antes de tomar la decisión de poner el elemento en ambos lados. Por ejemplo, en el caso anterior cuando definamos un nuevo empleado y una nueva dirección pondremos la

dirección en el empleado, por lo que el elemento `cascade` tendremos que definirlo únicamente en la relación anterior.

Borrado en cascada

A primera vista, la utilización de un borrado en cascada puede parecer atractiva. Dependiendo de la cardinalidad de la relación podría eliminar la necesidad de eliminar múltiples instancias de entidad. Sin embargo, aunque es un elemento muy interesante, debe utilizarse con cierto cuidado. Hay sólo dos situaciones en las que un `remove()` en cascada se puede usar sin problemas: relaciones uno-a-uno y uno-a-muchos en donde hay una clara relación de propiedad y la eliminación de la instancia propietaria debe causar la eliminación de sus instancias dependientes. No puede aplicarse ciegamente a todas las relaciones uno-a-uno o uno-a-muchos porque las entidades dependientes podrían también estar participando en otras relaciones o podrían tener que continuar en la base de datos como entidades aisladas.

Habiendo realizado el aviso, veamos qué sucede cuando se realiza una operación de `remove()` en cascada. Si una entidad `Empleado` se elimina, no tiene sentido eliminar el despacho (seguirá existiendo) pero sí sus cuentas de correo (suponiendo que le corresponde más de una). El siguiente código muestra cómo definimos este comportamiento:

```
@Entity
public class Empleado {
   // ...
   @OneToOne(cascade={CascadeType.PERSIST})
   Despacho despacho;
   @OneToMany(mappedBy="empleado",
              cascade={CascadeType.PERSIST, CascadeType.REMOVE})
   Collection<CuentaCorreo> cuentasCorreo;
   // ...
}
```

Cuando se llama al método `remove()` el entity manager navegará por las relaciones entre el empleado y sus cuentas de correo e irá eliminando todas las instancias asociadas al empleado.

Hay que hacer notar que este borrado en cascada afecta sólo a la base de datos y que no tiene ningún efecto en las relaciones en memoria entre las instancias en el contexto de persistencia. Cuando la instancia de `Empleado` se desconecte de la base de datos, su colección de cuentas de correo contendrá las mismas instancias de `CuentaCorreo` que tenía antes de llamar a la operación `remove()`. Incluso la misma instancia de `Empleado` seguirá existiendo, pero desconectada del contexto de persistencia.

3.4.2. Sincronización con la base de datos

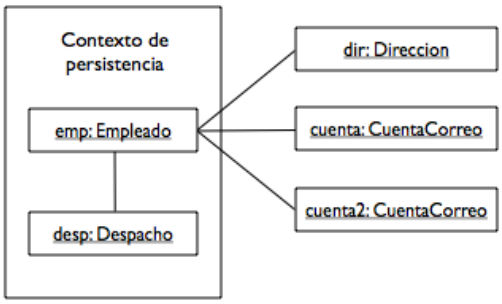
Cada vez que el proveedor de persistencia genera sentencias SQL y las escribe en la base de datos a través de una conexión JDBC, decimos que se ha volcado (*flush*) el contexto de

persistencia. Todos los cambios pendientes que requieren que se ejecute una sentencia SQL en la transacción se escriben en la base de datos cuando ésta realiza un commit. Esto significa que cualquier operación SQL que tenga lugar después de haberse realizado el volcado ya incorporará estos cambios. Esto es particularmente importante para consultas SQL que se ejecutan en una transacción que también está realizando cambios en los datos de la entidad.

¿Qué sucede exactamente cuando se realiza un volcado del contexto de persistencia? Un volcado consiste básicamente en tres componentes: entidades nuevas que necesitan hacerse persistentes, entidades modificadas que necesitan ser actualizadas y entidades borradas que deben ser eliminadas de la base de datos. Toda esta información es gestionada por el contexto de persistencia.

Cuando ocurre un volcado, el entity manager itera primero sobre las entidades gestionadas y busca nuevas entidades que se han añadido a las relaciones y que tienen activada la opción de persistencia en cascada. Esto es equivalente lógicamente a invocar a `persist()` con cada una de las entidades gestionadas antes de que se realice el volcado. El entity manager también comprueba la integridad de todas las relaciones. Si una entidad apunta a otra que no está gestionada o que ha sido eliminada, entonces se puede lanzar una excepción.

Las reglas que determinan si un volcado falla o no en presencia de entidades no gestionadas pueden ser complicadas. Veamos un ejemplo que demuestra los asuntos más comunes. La siguiente figura muestra un diagrama de objetos para una instancia de `Empleado` y algunos objetos con los que está relacionado.



Las instancias `emp` y `desp` están gestionadas en el contexto de persistencia. El objeto `dir` es una entidad desconectada de una transacción previa y los objetos `CuentaCorreo` son objetos nuevos que no han formado parte de ninguna relación hasta el momento. Supongamos que se va a volcar la instancia `emp`. Para determinar el resultado de este volcado, debemos mirar primero las características de la opción `cascade` en la definición de la relación. Supongamos que la entidad `Empleado` se define de la siguiente forma:

```
@Entity
public class Empleado {
   // ...
   @OneToOne
```

```
    Despacho despacho;
    @OneToMany(mappedBy="empleado", cascade=CascadeType.PERSIST)
    Collection<CuentaCorreo> cuentasCorreo;
    @ManyToOne
    Direccion direccion;
    // ...
}
```

Vemos que sólo la relación `cuentasCorreo` tiene una opción de persistencia en cascada. El resto de relaciones tienen la opción de cascada por defecto, por lo que no la tienen activada.

Comenzando por el objeto `emp` vamos a recorrer el proceso de volcado como si fuéramos el proveedor de persistencia. El objeto `emp` está gestionado y está enlazado con otros cuatro objetos. El primer paso en el proceso es recorrer las relaciones desde esta entidad como si fuéramos a invocar a `persist()` con ella. El primer objeto que encontramos en este proceso es el objeto `desp` en la relación una-a-una `despacho`. Al ser una instancia gestionada, no tenemos que hacer nada más. Después vamos a la relación `cuentasCorreo` con dos objetos `CuentaCorreo`. Estos objetos son nuevos y esto causaría normalmente una excepción, pero debido a que se ha definido `PERSIST` como opción de cascada, hacemos lo equivalente a invocar a `persist()` en cada objeto `CuentaCorreo`. Esto hace que los objetos sean gestionados, haciéndolos formar parte del contexto de persistencia. Los objetos `CuentaCorreo` no tienen ninguna otra relación que hacer persistente en cascada, por lo que hemos terminado por este lado. Después alcanzamos el objeto `dir` a través de la relación `direccion`. Ya que este objeto está desconectado, lanzaríamos normalmente una excepción, pero esta relación es un caso especial el algoritmo de volcado. Si el objeto desconectado es el destino de una relación uno-a-uno o muchos-a-uno no se lanzará una excepción y se procederá al volcado. Esto es debido a que el acto de hacer persistente la entidad propietaria de la relación no depende del objetivo. La entidad propietaria contiene una clave ajena y sólo necesita almacenar el valor de la clave primaria de la entidad con la que está relacionada. No hay que modificar nada en la entidad destino. Con esto hemos terminado de volcar el objeto `emp`. Ahora debemos ir al objeto `desp` y comenzar de nuevo. Terminaremos cuando no queden nuevos objetos que hacer persistentes.

Si en el proceso de volcado alguno de los objetos a los que apunta la instancia que estamos haciendo persistente no está gestionado, no tiene el atributo de persistencia en cascada y no está incluido en una relación uno-a-uno o muchos-a-uno entonces se lanzará una excepción `IllegalStateException`.

3.4.3. Desconexión de entidades

Como resultado de una consulta o de una relación, obtendremos una colección de entidades que deberemos tratar, pasar a otras capas de la aplicación y, en una aplicación web, mostrar en una página JSP o JSF. En este apartado vamos a ver cómo trabajar con las entidades obtenidas y vamos a reflexionar sobre su desconexión del contexto de persistencia.

Una entidad desconectada (*detached entity* en inglés) es una entidad que ya no está asociada a un contexto de persistencia. En algún momento estuvo gestionada, pero el contexto de persistencia puede haber terminado, o la entidad puede haberse transformado de forma que ha perdido su asociación con el contexto de persistencia que la gestionaba. Cualquier cambio que se realice en el estado de la entidad no se hará persistente en la base de datos, pero todo el estado que estaba en la entidad antes de desconectarse sigue estando ahí para ser usado por la aplicación.

Hay dos formas de ver la desconexión de entidades. Por una parte, es una herramienta poderosa que puede utilizarse por las aplicaciones para trabajar con aplicaciones remotas o para soportar el acceso a los datos de la entidad mucho después de que la transacción ha concluido. Otra posible interpretación es que puede ser una fuente de problemas frustrantes cuando las entidades contienen una gran cantidad de atributos que se cargan de forma perezosa y los clientes que usan estas entidades desconectadas necesitan acceder a esta información.

Existen muchas condiciones en las que una entidad se convierte en desconectada. Cada una de las situaciones siguientes generarán entidades desconectadas:

- Cuando el contexto de persistencia se cierra con una llamada a `close()` del entity manager
- Cuando se llama al método `clear()` del entity manager
- Cuando se produce un rollback de la transacción
- Cuando una entidad se serializa

Todos los casos se refieren a contextos de persistencias gestionados por la aplicación (Java SE y aplicaciones web sin contenedor de EJB).

En temas siguientes veremos el tipo de recuperación `LAZY` que puede aplicarse a los mapeos básicos o las relaciones. Este elemento tiene como efecto indicar al proveedor que la carga de los atributos de la entidad no debe hacerse hasta que se acceden por primera vez. Aunque no se suele utilizar para los atributos básicos, sí que es muy importante utilizar con cuidado esta característica en las relaciones para mejorar el rendimiento de la aplicación.

Tenemos que considerar por tanto, el impacto de la desconexión en la carga perezosa. Veamos un ejemplo. Supongamos que tenemos la siguiente definición de `Empleado`:

```
@Entity
public class Empleado {
   // ...
   @ManyToOne
   private Direccion direccion;
   @OneToOne(fetch=FetchType.LAZY)
   private Departamento departamento;
   @OneToMany(mappedBy="employee")
   private Collection<CuentaCorreo> cuentasCorreo;
   // ...
}
```

La relación `dirección` se cargará de forma ávida (*eager*, en inglés) debido a que no hemos especificado ninguna característica de carga y esa es la opción por defecto en las relaciones muchos-a-uno. En el caso de la relación `departamento`, que se cargaría también de forma ávida, hemos especificado una opción `LAZY`, por lo que la referencia se cargará de forma perezosa. Las cuentas de correo, por ser una relación uno-a-muchos se cargará también de forma perezosa por defecto.

En tanto en que la entidad `Empleado` esté gestionada todo funciona como es de esperar. Cuando la entidad se recupera de la base de datos, sólo la instancia `Direccion` se cargará en ella. El proveedor obtendrá las entidades necesarias cuando la aplicación acceda a las relaciones `cuentasCorreo` o `departamento`.

Si la entidad se desconecta, el resultado de acceder a las relaciones anteriores es ya algo más complidado. Si se accedió a las relaciones cuando la entidad estaba gestionada, entonces las entidades pueden también ser recuperadas de forma segura aunque la entidad `Empleado` esté desconectada. Si, sin embargo, no se accedió a las relaciones, entonces tenemos un problema.

El comportamento del acceso a atributos no cargados cuando la entidad está desconectada no está definido en la especificación. Algunas implemenentaciones pueden intentar resolver la relación, mientras que otros simplemente lanzan una excepción y dejan el atributo sin inicializar. En el caso de Hibernate, se lanza una excepción de tipo `org.hibernate.LazyInitializationException`. Si la entidad ha sido desconectada debido a una serialización entonces no hay virtualmente ninguna esperanza de resolver la relación. La única forma portable de gestionar estos casos es no utilizando estos atributos.

En el caso en el que las entidades no tengan atributos de carga perezosa, no debe haber demasiados problemas con la desconexión. Todo el estado de la entidad estará todavía disponible para su uso en la entidad. Vamos a ver a continuación las dos posibles estrategias para trabajar con entidades desconectadas: preparar las entidades para la desconexión y no desconectar en absoluto. Vamos a centrarnos en cómo aplicar estas estrategias a aplicaciones web que usan servlets, páginas JSP y peticiones HTTP.

3.5. Contextos de persistencia y aplicaciones web

Una de las características fundamentales de las aplicaciones web es que están basadas en peticiones y que no tienen estado. Es cierto que es posible definir una sesión en la aplicación y que los servlets proporcionan soporte para ello. Pero muchas veces esta sesión únicamente se utiliza a efectos de autentificación del usuario. Casi toda la lógica de negocio de la aplicación es *no conversacional*: el usuario hace una petición y, el servidor la contesta y el navegador la muestra. Vamos a considerar este escenario.

Por ser más específicos, supongamos que un servlet realiza llamada a un método de negocio que implementa una query JPA y que recibe una colección de entidades como resultado. El servlet coloca entonces estas entidades en el `request` y lo envía a un JSP

para presentación. Este patrón se denomina el Page Controller. En el contexto de la arquitectura MVC, el método de consulta proporciona el modelo, la página JSP es la vista y el servlet el controlador. En este caso el modelo es el DAO que devuelve la lista de empleados:

```
public class EmpleadoDAO {
   private EntityManagerFactory emf;

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

   public List<Empleado> findAll() {
      EntityManager em = emf.createEntityManager();
      List<Empleado> listaEmp = em.createQuery("SELECT e FROM
Employee e").getResultList();
          em.close();
      return listaEmp;
   }

   // ...
}
```

Vemos que el método `findAll()` abre y cierra el entity manager, por lo que devuelve una lista de entidades desconectadas.

Veamos cómo sería el servlet que realiza la llamada a una instancia de la clase `EmpleadoService`, obtendría los resultados y se los pasaría a la página JSP que pintaría los resultados:

```
public class EmployeeServlet extends HttpServlet {
   protected void doGet(HttpServletRequest request,
HttpServletResponse response)
            throws ServletException, IOException {

      EntityManagerFactory emf =
Persistence.createEntityManagerFactory("SimpleJPA");

      EmpleadoDAO empDAO = new EmpleadoDAO();
      empDAO.setEMF(emf);
      List emps = empDAO.findAll();
      request.setAttribute("empleados", emps);
getServletContext().getRequestDispatcher("/listaEmpleados.jsp")
                        .forward(request, response);

    }
}
```

Por último, veamos la página JSP. Usa JSTL para iterar sobre la colección de instancias de `Empleado` y para mostrar el nombre del empleado y el nombre del departamento al que está asignado. La variable `empleados` a la que se accede con la etiqueta `<c:forEach/>` es la lista de instancias de `Empleado` que el servlet ha colocado en el request.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
```

```
<html>
  <head>
    <title>Lista empleados</title>
  </head>
          <body>
    <table>
      <thead>
        <tr>
          <th>Nombre</th>
          <th>Departamento</th>
        </tr>
     </thead>
      <tbody>
        <c:forEach items="${empleados}" var="emp">
          <tr>
            <td><c:out value="${emp.nombre}"/></td>
            <td><c:out value="${emp.departmento.nombre}"/></td>
          </tr>
        </c:forEach>
      </tbody>
    </table>
  </body>
</html>
```

Hay que hacer notar la línea del código JSP en la que se accede al nombre del departamento asociado a un empleado recorriendo las relaciones entre los objetos. Si probamos el ejemplo tal y como lo hemos definido, veremos que al intentar pintar la página JSP aparece un error debido a que las entidades empleado están desconectadas del contexto de persistencia.

El problema tiene que ver con la carga perezosa de las entidades y su desconexión. La relación `departamento` está configurada para utilizar carga perezosa. Cuando se el método `findAll()` devuelve las entidades desconectadas, sus atributos `departamento` están vacíos. Cuando la página JSP recorre estas instancias e intenta acceder a esos atributos se produce un error.

Veamos a continuación las dos posibles soluciones a este problema. En el primer escenario dejaremos cargaremos en las entidades los datos que necesitamos mostrar en la página JSP. En el segundo escenario dejaremos abierta la sesión del entity manager y la cerraremos después de llamar a la página JSP.

3.5.1. Entidades desconectadas

En este escenario definimos en el DAO un método `findAllConNombreDepto()` que devuelve un conjunto de entidades desconectadas de la base de datos en las que hemos cargado el departamento y el nombre. El código parece un poco raro, porque se descartan los valores devueltos por los métodos por el método `get()`, pero el resultado es correcto:

```
public List<Empleado> findAllConNombreDepto() {
   EntityManager em = emf.createEntityManager();
   List<Empleado> listaEmp = em.createQuery("SELECT e FROM Employee
e").getResultList();
```

```
   for (Employee emp : emps) {
       Departmento dept = emp.getDepartmento();
       if (dept != null) {
           dept.getNombre();
       }
   }
   em.close();
   return listaEmp;
}
```

Hay que hacer notar que no sólo se invoca el método `getDepartamento()` en la instancia de `Empleado`, sino que se invoca también el método `getName()` en la propia instancia `Departamento`. Esto es necesario porque las entidades devueltas en una relación con carga perezosa pueden ser proxies y necesitamos acceder a alguno de sus atributos para que sean cargadas totalmente.

El servlet debe llamar a este método, en lugar de al método genérico `findAll()`.

3.5.2. Entidades gestionadas

La otra solución al escenario planteado es no desconectar las entidades. De esta forma no deberemos preocuparnos de si los datos están cargados o no, ya que será la página JSP la que acceda a la base de datos a través de las entidades gestionadas.

Para ello debemos buscar una forma de mantener el contexto de persistencia abierto durante la llamada al método de negocio y a la página JSP. La forma de hacerlo dependerá mucho de la arquitectura de la aplicación y de si estamos utilizando algún framework, tipo Struts o Spring. En general, debemos encontrar una forma de: (1) abrir un entity manager antes de la llamada al método de negocio, (2) pasarle el mismo entity manager (y su contexto de persistencia) al método de negocio y a la página JSP y (3) cerrar el entity manager una vez terminado el proceso.

En el ejemplo planteado no es demasiado complicado. Redefinimos la clase `EmpleadoDAO` de la siguiente forma:

```
public class EmpleadoDAO {
   private EntityManager em;
   private EntityManagerFactory emf;

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

   public setEM(EntityManager em) {
      this.em = em;
   }

   public List<Empleado> findAllManaged() {
      List<Empleado> listaEmp = em.createQuery("SELECT e FROM
Employee e").getResultList();
      return listaEmp;
   }
```

```
   public List<Empleado> findAll() {
      EntityManager em = emf.createEntityManager();
      List<Empleado> listEmp = findAllManaged();
      em.close();
      return listEmp;
   }
   // ...
}
```

Vemos que hemos añadido un método setter para el entity manager, dando la posibilidad de que se inyecte desde fuera de la clase. También hemos definido el método `findAllManaged()` que utiliza el entity manager actual y devuelve entidades gestionadas. Por último, hemos reescrito el método `findAll()` para que llame al método anterior, evitando código duplicado.

Modificamos el servlet para controlar la apertura y el cierre del entity manager fuera del DAO y poder llamar a la página JSP con la sesión abierta:

```
public class EmployeeServlet extends HttpServlet {
   protected void doGet(HttpServletRequest request,
HttpServletResponse response)
            throws ServletException, IOException {

      EntityManagerFactory emf =
Persistence.createEntityManagerFactory("SimpleJPA");
      EntityManager em = emf.createEntityManager();

      EmpleadoDAO empDAO = new EmpleadoDAO();
      empDAO.setEM(em);

      List emps = empleadoDAO.findAllManaged();

      request.setAttribute("empleados", emps);
getServletContext().getRequestDispatcher("/listaEmpleados.jsp")
                           .forward(request, response);

      em.close();
    }
}
```

Creamos al comienzo de la petición un `EntityManager` y se lo pasamos al DAO con el método `setEM()`. De esta forma, el método `findAllManaged()` utilizará el entity manager que hemos creado en la petición y no lo cerrará al terminar. Así las instancias que se devuelvan estarán gestionadas y lo seguirán estando cuando se pase la llamada a la página JSP. Al final es muy importante cerrar el entity manager.

La solución anterior se denomina a veces *entity manager por petición*, ya que asociamos la vida del entity manager a la duración de la petición HTTP. Es una solución bastante común y es la que es usada por defecto por frameworks como Spring.

Es difícil utilizar un entity manager que se extienda más allá de una petición, implementando lo que se denominan *conversaciones*. Podríamos guardarlo en un objeto

HTTPSession para utilizarlo en más de una página de la misma sesión (supongamos un carrito de la compra, por ejemplo), pero se hace complicado manejar diversos aspectos como la transaccionalidad o la concurrencia. Lo habitual es desconectar las entidades en cada petición y que la sesión trabaje con datos desconectados que volvemos a conectar (con merge()) en un nuevo entity manager cuando sea necesario.

La arquitectura EJB proporciona componentes denominados *EJB con estado* que están pensados precisamente para guardar el estado conversacional en una sesión. Si nuestra aplicación lo requiere, debemos considerar su utilización, en lugar de usar la sesión HTTP.

4. Ejercicios sesión 2: Conceptos básicos de JPA

4.1. Servlet DoAction

Vamos a modificar la aplicación web para que sea más sencillo añadir nuevas funcionalidades. En la página HTML vamos a incluir un nuevo campo de texto en el que el usuario escriba la acción que quiere realizar (añadir autor, listar mensajes, etc.). También vamos que cambiar el servlet para que lea la acción y llame al método correspondiente que procesa esa acción.

1. Cambia el fichero `index.html`, modificando el nombre del servlet y añadiendo el nuevo parámetro `accion` ligado a un nuevo campo de entrada. En ese nuevo campo escribiremos la acción que queremos probar. Así nos será más sencillo añadir nuevas funcionalidades sin tener que retocar demasiado las páginas HTML de la aplicación.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
   <meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
   <title>Aplicación web Mensajes</title>
</head>
<body>

   <form action="DoActionServlet">
   Mensaje:  <input type="text" name="mensaje"><br>
   Autor: <input type="text" name="autor"><br>
   Acción (add,lista,cuenta): <input type="text" name="accion"><br>
   <input type="submit" value="Enviar">
   </form>
</body>
</html>
```

2. Define el servlet `DoActionServlet` como sigue:

```
package es.ua.jtech.jpa;

import java.io.*;
import java.util.Collection;

import javax.persistence.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class DoActionServlet extends HttpServlet {
   private static final long serialVersionUID = 1L;

   public void doGet(HttpServletRequest request,
HttpServletResponse response)
         throws ServletException, IOException {
```

```
      String accion = request.getParameter("accion");

      if (accion.equals("add")) {
         doAdd(request, response);
      } else if (accion.equals("lista")) {
         doLista(request, response);
      } else if (accion.equals("cuenta")) {
         doCuenta(request, response);
      } else {
         PrintWriter out = response.getWriter();
         out.println("Error. La acción " + accion + " no está
implementada");
      }
   }

   private void doCuenta(HttpServletRequest request,
         HttpServletResponse response) throws ServletException,
IOException {

           // Cuenta los mensajes de un autor

   }

   private void doLista(HttpServletRequest request,
                        HttpServletResponse response)
         throws ServletException, IOException {

      // Lista todos los mensajes de un autor

   }

   private void doAdd(HttpServletRequest request,
                      HttpServletResponse response)
         throws ServletException, IOException {

      // Añade un mensaje a un autor. Mismo código que el servlet
AddMensajeServlet
   }
}
```

3. Cambia por último el fichero `web.xml` para añadir este servlet.

4. Comprueba que todo funciona correctamente. Ahora podremos añadir fácilmente nuevas funciones. Sólo tendremos que añadir un nuevo caso en el `switch`, el código de la nueva función en un método privado y una nueva página JSP que haga la presentación de los resultados.

4.2. Implementación de funciones adicionales

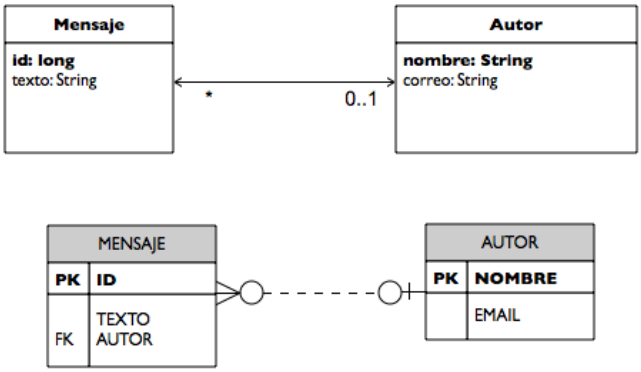
Implementa las acciones que están vacías en el servlet (`doLista` y `doCuenta`) y las páginas JSP siguientes. Supongamos que queremos cerrar el entity manager antes de llamar a las páginas JSP (igual que en `addMensaje()`). ¿Funciona correctamente el listado o la cuenta de mensajes? ¿Cómo arreglarlo? (contesta en `respuestas02.txt`)

- `addMensaje.jsp`: muestra el nombre del usuario y el texto del mensaje añadido

- `cuentaMensajes.jsp`: muestra el nombre del usuario y el número de mensajes que ha añadido
- `listaMensajes.jsp`: muestra el nombre del usuario y la lista de mensajes que ha añadido (**ya está implementada**)

4.3. Contexto de persistencia

En este ejercicio vamos a probar la diferencia entre el contexto de persistencia y la base de datos y a reforzar la característica principal del contexto de persistencia: es una caché de la base de datos. Lo vamos a hacer estudiando la relación uno-a-muchos entre autor y mensajes. Recordemos que la relación es propiedad de la entidad `Mensaje` y que allí es donde se guarda la clave ajena a la entidad `Autor` (ver figura).



Debido a que la entidad propietaria es el `Mensaje` si no actualizamos esa clave ajena (con el método `setAutor()`) no se guardará la relación en la base de datos. También debemos recordar que en la entidad `Autor` se guarda una colección de mensajes. Esta colección reside en el contexto de persistencia. El responsable de recuperar la colección de mensajes de la base de datos es el `EntityManager`, que intercepta la llamada al método `getMensajes()` y realiza una consulta SQL si lo considera necesario.

1. Vamos a comenzar probando algunas modificaciones en el programa Java `HolaMundo`. Modifica el programa para que no se actualice la colección de mensajes del autor y prueba a imprimir el número de mensajes antes y después de actualizar la relación entre el mensaje y el autor:

```
Autor autor = em.find(Autor.class, autorStr);
if (autor == null) {
   autor = new Autor();
   autor.setNombre(autorStr);
   autor.setCorreo(autorStr + "@ua.es");
   em.persist(autor);
}

System.out.println(autor.getNombre() + " ha escrito " +
                   autor.getMensajes().size() + " mensajes");
```

```
// Creamos la entidad Mensaje

Mensaje mensaje = new Mensaje();
mensaje.setTexto(mensStr);
mensaje.setFecha(new Date());
mensaje.setAutor(autor);
em.persist(mensaje);

// No actualizamos la colección de mensajes
//autor.addMensaje(mensaje);

em.getTransaction().commit();

System.out.println("Mensaje y autor añadidos");
System.out.println("El autor " + autor.getNombre() + " ha escrito "
                   + autor.getMensajes().size() + " mensajes\n");

em.close();
```

Prueba la función introduciendo un mensaje y un autor nuevo. ¿Se muestra correctamente la lista de mensajes? ¿Se ha hecho realmente la modificación en la base de datos? ¿Qué está pasando? (contesta en `respuestas02.txt`)

2. Por último, añade la siguiente instrucción justo antes de obtener la colección de mensajes antes de cerrar la transacción.

```
em.refresh(mensajes);
```

¿Funciona ahora correctamente el listado? ¿Por qué? (contesta en `respuestas02.txt`)

3. Déjalo todo como al principio.

4.4. Carga perezosa

1. Vamos a comprobar el problema que surge cuando recorremos las relaciones entre entidades y éstas están desconectadas. Para ello vamos a utilizar la aplicación web. Cambia la página JSP que muestra la lista de mensajes para que muestre el nombre del autor a partir de la relación con el mensaje:

```
<c:forEach items="${mensajes}" var="mens">
  <tr>
    <td><c:out value="${mens.autor.nombre}"/></td>
    <td><c:out value="${mens.texto}"/></td>
    <td><c:out value="${mens.fecha}"/></td>
  </tr>
</c:forEach>
```

2. Implementa ahora la acción `lista-todos` que hace una query para obtener todos los mensajes de la base de datos y llama a esta página JSP. ¿Funciona? (contesta en `respuestas02.txt`)

```
@SuppressWarnings("unchecked")
private void doListaTodos(HttpServletRequest request,
```

```
HttpServletResponse response) throws ServletException, IOException
{

   EntityManagerFactory emf = Persistence
                .createEntityManagerFactory("simplejpa");
   EntityManager em = emf.createEntityManager();
   em.getTransaction().begin();
   Query query = em.createQuery("SELECT m FROM Mensaje m");
   List<Mensaje> mensajes = query.getResultList();
   em.close();
   emf.close();

   System.out.println("Cerrado el Entity Manager");

   request.setAttribute("mensajes", mensajes);
getServletContext().getRequestDispatcher("/listaMensajes.jsp").forward(
                request, response);
}
```

3. Cambia ahora el tipo de acceso en la relación entre mensaje y autor, para que sea *lazy*:

```
@Entity
public class Mensaje {
   @Id @GeneratedValue
   private long id;
   private String texto;
   private Date fecha;
   @ManyToOne(fetch=FetchType.LAZY)
   private Autor autor;
```

Vuelve a probarlo ¿Qué mensaje de error aparece? ¿Por qué? (contesta en `respuestas02.txt`). Prueba las dos soluciones que se han explicado en teoría. Comienza por la más sencilla (cerrar el entity manager después de crear la página HTML) y prueba también la segunda de añadir un nuevo método al DAO que recupere los mensajes de la base de datos.

4.5. Construcción de clases DAO (*)

Construye las clases `AutorDAO` y `MensajeDAO`, siguiendo el esquema visto en la clase de teoría (la versión en la que se devuelven entidades desconectadas). Modifica los métodos del servlet `DoActionServlet` para que haga las llamadas a estas clases, en lugar de a las entidades.

5. Mapeado entidad-relación: tablas

En este tema vamos a comenzar a tratar en profundidad el mapeado entidad-relación (ORM, *Object-Relational Mapping* en inglés), uno de los aspectos principales del API de persistencia. Veremos los aspectos referidos al mapeo de entidades en tablas: cómo acceder al estado de una entidad, como se mapea la entidad en una tabla, como se mapean los distintos tipos simples de Java en tipos de la base de datos, cómo se maneja el identificador de la identidad y se mapea en la clave primaria de la tabla. Terminaremos introduciendo un par de conceptos avanzados, los objetos embebidos y la herencia.

Como hemos comentado en temas anteriores, la especificación del ORM se realiza mediante metadatos especificados por el desarrollador. JPA soporta dos formas de especificar estos metadatos, mediante anotaciones y mediante ficheros XML. A lo largo de todo el tema (y del módulo) utilizaremos las anotaciones por ser mucho más legibles y estar integradas en el propio código de la aplicación.

5.1. Acceso al estado de la entidad

El proveedor de la base de datos debe poder acceder al estado mapeado en memoria en tiempo de ejecución, de forma que en el momento de escribir los datos, éstos puedan obtenerse de la instancia de la entidad. De la misma forma, cuando se carga el estado de la base de datos, el proveedor debe poder crear una nueva instancia e insertar en ella los datos obtenidos de la base de datos.

Existen dos formas diferentes con las que podemos especificar el estado persistente: podemos anotar los campos de la entidad o anotar sus propiedades (métodos JavaBean *getters* y *setters*). El mecanismo del proveedor para acceder al estado dependerá entonces del tipo de anotación utilizada. En el primer caso, el proveedor leerá y establecerá los campos de la entidad utilizando reflexión. Si las anotaciones se realizan en las propiedades, entonces el proveedor utilizará estos métodos para acceder al estado.

5.1.1. Acceso por campo

Si anotamos los campos de una entidad, el proveedor accederá a estos campos mediante reflexión (aunque estén declarados como privados). A esta forma de acceder a la entidad por parte del proveedor se denomina *acceso por campo*. Los métodos *getters* y *setters* serán ignorados por el proveedor. Todos los campos deben declararse como `protected` o `private`. Se desaconseja siempre el uso de campos públicos, porque podrían accederse directamente desde otras clases. Este acceso público podría incluso estropear el funcionamiento del proveedor.

El ejemplo siguiente muestra el uso de las anotaciones en los campos. La anotación `@Id` indica no sólo que el campo `id` es el identificador o clave primaria de la entidad, sino

también que se va a utilizar un acceso por campo. Los campos `nombre` y `sueldo` son hechos persistentes en columnas con el mismo nombre.

```
@Entity
public class Empleado {
@Id private int id;
private String nombre;
private Long sueldo;

public Empleado() {}

public int getId() { return id; }
public void setId(int id) { this.id = id; }
public String getNombre() { return nombre; }
public void setNombre(String nombre) { this.nombre = nombre; }
public Long getSueldo() { return sueldo; }
public void setSueldo(Long sueldo) { this.sueldo = sueldo; }}
```

5.1.2. Acceso por propiedad

Cuando anotamos las propiedades, se aplican las mismas condiciones que cuando se define un JavaBean, y debe haber *getters* y *setters* para las propiedades que queremos hacer persistentes. En este caso se dice que el proveedor utiliza un acceso por propiedad a la entidad. El tipo de la propiedad viene determinado por el tipo devuelto por el método *getter* y debe ser el mismo que el único parámetro pasado al método *setter*. Ambos métodos deben tener visibilidad `public` o `protected`. La anotación de mapeado debe realizarse en el método *getter*.

En el código siguiente, la clase `Empleado` tiene una anotación `@Id` en el método *getter* `getId()`, por lo que el proveedor podrá utilizar el acceso a la propiedad `id` para obtener y establecer el estado de la entidad. Las propiedades `nombre` y `sueldo` se hacen persistentes también automáticamente y se mapearán en columnas con el mismo nombre. Nótese que es posible utilizar nombres distintos para los campos. En el ejemplo, la propiedad `sueldo` está respaldada por el campo `sueldo`. El proveedor ignora esta diferencia, ya que únicamente utiliza los *getters* y *setters*.

```
@Entity
public class Empleado {
private int id;
private String nombre;
private Long sueldo;

public Empleado {}

@Id public int getId() { return id; }
public void setId(int id) { this.id = id; }
public String getNombre() { return nombre; }
public void setNombre(String nombre) { this.nombre = nombre; }
public Long getSueldo() { return sueldo; }
public void setSueldo(Long sueldo) { this.sueldo = sueldo; }}
```

5.2. Mapeado de entidades

Ya hemos visto en el tema anterior que es muy sencillo mapear entidades en tablas. Sólo se necesitan las anotaciones `@Entity` y `@Id` para crear y mapear una entidad en una tabla de la base de datos.

En esos casos el nombre que se utiliza para la tabla es el propio nombre de la entidad. Podría darse el caso de que necesitáramos especificar un nombre distinto para la tabla, por ejemplo si no estamos desarrollando la aplicación desde cero y partimos de un modelo de datos ya creado. Podemos hacerlo con la anotación `@Table` en la que incluimos el nombre de la tabla. El siguiente código muestra un ejemplo.

```
@Entity
@Table(name="EMP")
public class Empleado { ... }
```

Los nombres por defecto son los nombres de las clases, que en Java comienzan por mayúscula y continúan con minúscula. ¿Cómo se mapean las mayúsculas y minúsculas en la base de datos? Depende de la base de datos. Muchas bases de datos no distinguen mayúsculas y minúsculas, por lo que en estos casos el nombre se convierte en mayúsculas. En el caso de *MySQL*, sí que se distinguen entre mayúsculas y minúsculas, por lo que el nomre de las tablas será idéntico al de las clases.

La anotación `@Table` proporciona la posibilidad no sólo de nombrar la tabla, sino de especificar un esquema o catálogo de la base de datos. El nombre del esquema se utiliza normalmente para diferenciar un conjunto de tablas de otro y se indica en la anotación con el elemento `schema`. Se muestra en el siguiente ejemplo.

```
@Entity
@Table(name="EMP", schema="IT")
public class Empleado { ... }
```

Cuando se especifica de esta forma, el proveedor colocará el nombre del esquema como prefijo del de la tabla cuando acceda a los datos. En este caso, los accesos se harán a la tabla `IT.EMP`.

5.3. Mapeado de tipos

La especificación de JPA define un gran número de tipos Java que pueden ser hechos persistentes. Son los siguientes:

- **Tipos primitivos Java**: `byte`, `int`, `short`, `long`, `boolean`, `char`, `float`, `double`
- **Clases *wrapper* de los tipos primitivos**: `Byte`, `Integer`, `Short`, `Long`, `Boolean`, `Character`, `Float`, `Double`
- **Arrays de bytes y char**: `byte[]`, `Byte[]`, `char[]`, `Character[]`
- **Tipos numéricos largos**: `java.math.BigInteger`, `java.math.BigDecimal`
- **Strings**: `java.lang.String`
- **Tipos temporales de Java**: `java.util.Date`, `java.util.Calendar`
- **Tipos temporales de JDBC**: `java.sql.Date`, `java.sql.Time`,

`java.sql.Timestamp`
- **Tipos enumerados**: cualquier tipo enumerado del sistema o definido por el usuario
- **Objetos serializables**: cualquier tipo serializable del sistema o definido por el usuario

En algunos casos el tipo de la columna que está siendo mapeada no es exactamente el mismo que el tipo Java. En casi todos los casos, el runtime del proveedor puede convertir el tipo devuelto por la consulta JDBC en el tipo Java correcto del atributo. Si el tipo de la capa JDBC no puede convertirse en el tipo Java del campo o la propiedad se lanza una excepción.

Cuando se hace persistente un campo o una propiedad, el proveedor comprueba que su tipo es uno de los que está en la lista anterior. Si lo está, el proveedor lo transformará en el tipo JDBC apropiado y lo pasará al driver JDBC.

Se puede usar la anotación opcional `@Basic` en el campo o la propiedad para marcarlos como persistentes. Esta anotación se utiliza únicamente para efectos de documentación o para especificar algún detalle sobre la persistencia (lo veremos más adelante).

Ahora que hemos comprobados que los campos (definidos en las variables de instancia) y las propiedades (definidas en los *getters* y *setters*) son equivalentes en términos de persistencia, los llamaremos de ahora en adelante *atributos*. Consideramos *atributo* un campo o una propiedad de una clase estilo JavaBean.

5.3.1. Mapeo de columnas

Es posible anotar las características físicas de la columna de la base de datos en la que se mapea un atributo utilizando la anotación `@Column`. Aunque es posible especificar bastantes elementos, vamos a comentar sólo alguno de ellos (consultar la especificación de JPA para obtener más información).

La primera característica que hay que mencionar es el nombre de la columna. Al igual que con las tablas, es posible especificar los nombres de las columnas con las que se va a mapear cada atributo. El siguiente código muestra un ejemplo.

```
@Entity
public class Empleado {
   @Id
   @Column(name="EMP_ID")
   private int id;
   private String nombre;
   @Column(name=SAL)
   private Long sueldo;
   @Column(name=COM)
   private String comentario;
   // ...
}
```

La tabla resultante del mapeo se llamaría `EMPLEADO` y tendría como columnas `EMP_ID`, `NOMBRE`, `SAL` y `COM`. La primera columna sería la clave primaria de la tabla. La siguiente figura muestra la tabla resultante en la base de datos.

Es posible también obligar a que un atributo no pueda dejarse a `null` utilizando el elemento `nullable=false`. En la columna de la tabla se incluiría la restricción SQL `NOT NULL`. Por ejemplo en el siguiente código obligaríamos a que el nombre del empleado nunca pudiera ser `null`.

```
@Entity
public class Empleado {
   @Id private int id;
   @Column(nullable=false)
   private String nombre;
   // ...
}
```

Cuando no se especifica la longitud de una columna que almacena cadenas (`String`, `char[]` o `Character[]`), el valor por defecto es 255. Para definir otro tamaño hay que utilizar el elemento `length`, como en el siguiente ejemplo:

```
@Entity
public class Empleado {
   @Id private int id;
   @Column(length=40)
   private String nombre;
   // ...
}
```

5.3.2. Recuperación perezosa

El concepto de recuperación perezosa (*lazy fetching* en inglés) es muy importante para gestionar de forma eficiente la base de datos. Ya veremos más adelante que también se aplica a las relaciones entre entidades.

En ocasiones, sabemos que hay algunos atributos de la entidad a los que se accede con muy poca frecuencia. En este caso podemos optimizar el rendimiento de los accesos a la base de datos obteniendo sólo los datos que vamos a necesitar con frecuencia. Existen muchos nombres para esta idea, entre los que se incluyen (en inglés) *lazy loading*, *lazy fetching*, *on-demand fetching* o *indirection*. Todos significan lo mismo, que es que algunos datos no se cargan en el objeto cuando éste es leído inicialmente de la base de datos sino que serán recuperados sólo cuando sean referenciados o accedidos.

El comportamiento de un atributo de la entidad se puede especificar con el elemento `fetch` de la anotación `@Basic`. El tipo enumerado `FetchType` especifica los posibles valores de este elemento, que pueden ser `EAGER` (ávido, recupera el dato cuando se

obtiene la entidad de la base de datos) o `LAZY` (perezoso, recupera el dato cuando se accede al atributo). El comportamiento por defecto es el primero.

El siguiente código muestra un ejemplo, en el que el atributo `comentario` se define con un mapeo de recuperación perezosa:

```
@Entity
public class Empleado {
   // ...
   @Basic(fetch=FetchType.LAZY)
   @Column(name=COM)
   private String comentario;
   // ...
}
```

Antes de usar esta característica se debería tener claro unos cuantos aspectos. Lo primero es que la declaración de un atributo como de recuperación perezosa no obliga a nada al proveedor de persistencia. Sólo es una indicación para que pueda agilizar ciertas acciones sobre la base de datos. El proveedor no está obligado a respetar la petición, ya que el comportamiento de la entidad no queda comprometido haga una cosa u otra el proveedor.

Segundo, aunque en principio pueda parecer interesante definir ciertos atributos como de carga perezosa, en la práctica no es correcto hacerlo con tipos simples (no relaciones a otras entidades). La razón es que se gana poco haciendo que la base de datos devuelva parte de una fila. Únicamente se gana algo y debería considerarse la recuperación perezosa cuando tenemos muchas (decenas o cientos) columnas o cuando algunas columnas ocupan mucho (por ejemplo, cadenas muy largas o *lobs*).

La recuperación perezosa sí que es muy importante cuando hablemos de mapeo de relaciones, como veremos más adelante.

5.3.3. LOBs

El nombre que habitualmente se les da en base de datos a los objetos de tipo byte o caracter que son muy grandes es el de *large object* o *LOB* como abreviatura. Las columnas de la base de datos que almacenan estos tipos de objetos se deben acceder desde Java con llamadas JDBC especiales. Para indicarle al proveedor que debería usar métodos de tipo LOB en el driver de JDBC para acceder a ciertas columnas se debe utilizar la anotación `@Lob`.

En la base de datos se pueden encontrar dos tipos de LOBs: objetos grandes de tipo carácter, llamados *CLOBs* y objetos grandes de tipo binario, llamados *BLOBs*. Como su nombre indica, una columna CLOB guarda una larga secuencia de caracteres, mientras que un BLOB guarda una larga secuencia de bytes no formateados. Los tipos Java que se mapean con columnas CLOB son `String`, `char[]` y `Character[]`, mientras que `byte[]`, `Byte[]` y `Serializable` se mapean con columnas de tipo BLOB.

El siguiente código muestra el ejemplo de mapeo de una columna con un BLOB imagen. Suponemos que la columna PIC guarda una imagen del empleado, que se mapea en el

atributo `foto` de tipo `byte[]`.

```
@Entity
public class Empleado {
   @Id private int id;
   @Basic(fetch=FetchType.LAZY)
   @Lob @Column(name="PIC")
   private byte[] foto;
   // ...
}
```

5.3.4. Tipos enumerados

Otro tipo Java que puede ser mapeado en la base de datos es cualquier tipo enumerado del sistema o definido por el usuario.

Al igual que en otros lenguajes de programación, a los valores de los tipos enumerados en Java se les asigna un ordinal implícito que depende del orden de creación. Este ordinal no puede ser modificado en tiempo de ejecución y es el que se utiliza para representar y almacenar el valor del tipo enumerado. El proveedor, por tanto, mapeará un tipo enumerado en una columna de tipo entero y sus valores en números enteros específicos.

Por ejemplo, consideremos el siguiente tipo enumerado:

```
public enum TipoEmpleado {
   EMPLEADO_TIEMPO_PARCIAL,
   EMPLEADO_TIEMPO_COMPLETO,
   EMPLEADO_EXTERNO
}
```

Los ordinales asignados en tiempo de compilación a los valores de este tipo enumerado son 0 para `EMPLEADO_TIEMPO_PARCIAL`, 1 para `EMPLEADO_TIEMPO_COMPLETO` y 2 para `EMPLEADO_EXTERNO`. El siguiente código utiliza este tipo para definir un atributo de la entidad:

```
@Entity
public class Empleado {
   @Id private int id;
   private TipoEmpleado tipo;
   // ...
}
```

Podemos ver que el mapeado es trivial, ya que no hay que hacer nada especial y el proveedor se encarga de realizar la transformación del tipo enumerado al tipo entero de la base de datos.

Sin embargo, hay que tener cuidado con una cosa. Si en algún momento cambiamos el tipo enumerado podemos tener problemas, ya que puede cambiar el orden de los valores en el tipo enumerado y no corresponderse con los ya existentes en la base de datos. Por ejemplo, supongamos que necesitamos añadir un nuevo tipo de empleado a tiempo completo: `EMPLEADO_TIEMPO_COMPLETO_EXCEDENCIA` y supongamos que lo añadimos

justo después de `EMPLEADO_TIEMPO_COMPLETO`. Esto causaría un cambio en el ordinal asociado a `EMPLEADO_EXTERNO`, que pasaría de 2 a 3. Los empleados existentes en la base de datos, sin embargo, no cambiarían y los datos grabados con el ordinal 2 pasarían de ser `EMPLEADO_EXTERNO` a ser `EMPLEADO_TIEMPO_COMPLETO_EXCEDENCIA`.

Podríamos modificar la base de datos y ajustar todos las entidades, pero si los ordinales se utilizan en algún otro lugar tendríamos que arreglarlo también. No es una buena política de mantenimiento.

Una solución mejor sería almacenar el nombre del valor como una cadena en lugar de almacenar el ordinal. Esto nos aislaría de los cambios en la declaración y nos permitiría añadir nuevos tipos sin tener que preocuparnos sobre los datos existentes. Podemos hacer esto añadiendo una anotación `@Enumerated` en el atributo y especificando un valor de `STRING`. El siguiente código muestra cómo hacerlo:

```
@Entity
public class Empleado {
   @Id private int id;
   @Enumerated(EnumType.STRING)
   private TipoEmpleado tipo;
   // ...
}
```

Hay que hacer notar de esta forma no arreglamos el problema completamente. Ahora en la base de datos se guardan las cadenas `EMPLEADO_TIEMPO_COMPLETO` y demás. Si en algún momento modificamos el nombre de los valores del tipo enumerado también deberíamos cambiar los datos de la base de datos. Pero esto es menos frecuente, ya que un cambio en los valores de un tipo enumerado nos obliga a cambiar todo el código en el que aparezcan los valores, y esto es bastante más serio que cambiar los datos de una columna de la base de datos.

En general, definir el tipo enumerado como un ordinal es la forma más eficiente de trabajar, pero siempre que no sea probable tener que añadir nuevos valores en medio de los ya existentes.

5.3.5. Tipos temporales

Los tipos temporales son el conjunto de tipos basados en tiempo que pueden usarse en el mapeo entidad-relación. La lista de tipos temporales soportados incluye los tres tipos `java.sql` `java.sql.Date`, `java.sql.Time` y `java.sql.Timestamp`, e incluye también los tipos `java.util` `java.util.Date` y `java.util.Calendar`.

El mapeo de los tipos `java.sql` no plantea ningún problema en absoluto y se almacenan en la base de datos sin cambios. Los dos tipos `java.util` necesitan metadatos adicionales, para indicar qué tipo JDBC `java.sql` hay que usar cuando el proveedor haga persistente la entidad. Esto se consigue anotándolos con la anotación `@Temporal` y especificando el valor del tipo JDBC utilizando el valor correspondiente del tipo enumerado `TemporalType`. Hay tres valores enumerados: `DATE`, `TIME` y `TIMESTAMP` que

representan cada uno de los tipos `java.sql`.

EL código siguiente muestra cómo `java.util.Date` y `java.util.Calendar` pueden mapearse a columnas de la base de datos.

```
@Entity
public class Empleado {
   @Id private int id;
   @Temporal(TemporalType.DATE)
   private java.util.Date fechaNacimiento;
   @Temporal(TemporalType.TIMESTAMP)
   private java.util.Date horaSalida;
   // ...
}
```

5.3.6. Estado transitorio

Es posible definir en la entidad atributos que no se hacen persistentes utilizando la palabra clave de Java `transient` o el atributo `@Transient`. Si se especifica alguna de estas propiedades, el proveedor no aplicará las reglas por defecto al atributo marcado.

Los campos transitorios son útiles, por ejemplo, para cachear un estado en memoria que no queremos recalcular o reinicializar. En el ejemplo siguiente usamos el campo transitorio `traduccion` para guardar la traducción de la palabra "Empleado" en el locale actual, de forma que se imprima correctamente el nombre. El uso del modificador Java `transient` hace que el atributo sea temporal no sólo para la persistencia sino también para la máquina virtual. Si el `Empleado` se serializa y se envía desde una MV a otra el valor del atributo `traduccion` no se enviaría.

```
@Entity
public class Empleado {
   @Id private int id;
   private String nombre;
   private Long sueldo;
   transient private String traduccion;
   // ...

   public String toString() {
      if (traduccion == null) {
         traduccion =
ResourceBundle.getBundle("EmpResources").getString("Empleado");
      }
      return traduccion + ": " + id + " " + nombre;
   }
}
```

5.4. Mapeo de la clave primaria

Cualquier entidad mapeada en una base de datos relacional debe definir un mapeo a una clave primaria de la tabla. Vamos a explorar con algo más de detalle las claves primarias y cómo conseguir que el proveedor de persistencia las genere para nosotros.

Los identificadores en entidades que después se mapean a claves primarias en la tabla se restringen a los siguientes tipos:

- **Tipos Java primitivos**: `byte`, `int`, `short`, `long`, `char`
- **Clases wrapper de tipos primitivos**: `Byte`, `Integer`, `Short`, `Long`, `Character`
- **Arrays de tipos primitivos o de clases wrappers**
- **Cadenas**: `java.lang.String`
- **Tipos numéricos grandes**: `java.math.BigInteger`
- **Tipos temporales**: `java.util.Date`, `java.sql.Date`

Igual que con los mapeos básicos, la anotación `@Column` puede utilizarse para modificar el nombre con el que el atributo identificador se hace persistente. Si no se utiliza sucede igual que con los mapeos básicos y el campo se guarda en la columna con el mismo nombre.

5.4.1. Generación del identificador

En muchas ocasiones no queremos preocuparnos de definir un nombre único para las instancias de algunas entidades de nuestro modelo de dominio y nos interesa que los identificadores se generen de forma automática. A esto se le llama generación de id y se especifica con la anotación `@GeneratedValue`.

Cuando se utiliza la generación de id, el proveedor de persistencia generará un identificador único para cada instancia de un tipo dado. Una vez que se obtiene este identificador, el proveedor lo insertará en la entidad; sin embargo, dependiendo de cómo ha sido generado, puede ser que no se esté realmente presente en el objeto hasta que la entidad haya sido insertada en la base de datos. En otras palabras, la aplicación no puede acceder al identificador hasta que se haya hecho un flush o se haya completado la transacción.

Existen cuatro estrategias de generación de id que se seleccionan mediante el elemento `strategy` de la anotación. Son `AUTO`, `TABLE`, `SEQUENCE` o `IDENTITY`, en la que se utilizan valores enumerados del tipo `GenerationType`. Vamos a ver solamente la primera.

5.4.1.1. Generación de id automática

Una forma rápida de obtener generación de id automática es mediante la estrategia `AUTO`. Con esta estrategia, dejamos que sea el proveedor de persistencia el que se ocupe de cómo generar los identificadores. El siguiente código muestra un ejemplo:

```
@Entity
public class Empleado {
   @Id @GeneratedValue(strategy=GenerationType.AUTO)
   private int id;
   // ...
}
```

Un problema con esta estrategia es que es poco portable, ya que depende del proveedor de

persistencia. En muchas ocasiones hay que recurrir al administrador de la base de datos para que configure alguna opción en la base de datos para habilitar o configurar esta característica en la base de datos que queramos utilizar.

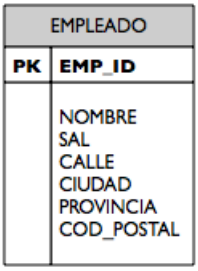
5.5. Objetos embebidos

Un *objeto embebido* es uno que no tiene identidad propia, y que está ligado a una entidad. Es meramente una parte de una entidad que ha sido separada y almacenada en un objeto Java independiente para adaptar mejor el modelo de datos al dominio. En la definición de la entidad aparece un atributo con un tipo no básico. A primera vista parecería que se está definiendo una relación con otra entidad. Sin embargo, el tipo embebido no tiene entidad suficiente como para definir una entidad persistente por él mismo. Sus datos se almacenan con el resto de la entidad en la misma fila de la base de datos.

Un ejemplo muy común es el tipo `Direccion`. Puede ser que en nuestro dominio una dirección no tenga las características que le hagan definir una entidad persistente (no vamos a hacer búsquedas por direcciones, ni identificadores de direcciones). Sin embargo, queremos guardar los datos que forman la dirección como un atributo del empleado. Y además obtener los beneficios del modelo de objetos considerándolos uno único dato.

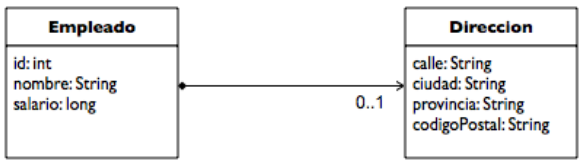
Las ventajas de agrupar un conjunto de campos en un nuevo tipo de datos Java son múltiples. En primer lugar, abstraemos el modelo físico (representación en la tabla de la base de datos) y obtenemos una representación más cercana al dominio de la aplicación. Podremos utilizar objetos `Direccion` en distintas partes de la lógica de negocio. En segundo lugar, podemos reutilizar este tipo en más de una entidad, dando consistencia a nuestro modelo físico. Por último, es una forma muy portable de conseguir una características de SQL que nunca se ha llegado a estandarizar: el uso de tipos definidos por el usuario.

Vamos a ver el ejemplo con más detalle. La siguiente figura muestra una tabla `EMPLEADO` que contiene una mezcla de información propia del empleado y de columnas que definen su dirección:



Las columnas `CALLE`, `CIUDAD`, `PROVINCIA` y `COD_POSTAL` se combinan lógicamente para formar la dirección. En el modelo de objetos podríamos perfectamente abstraer esta

información en tipo embebido `Direccion`. La clase entidad tendría entonces una atributo `direccion` que referenciaría un objeto embebido de tipo `Direccion`. La siguiente figura muestra la relación entre `Empleado` y `Direccion`. Utilizamos la asociación UML *composición* para denotar que el `Empleado` posee completamente la `Direccion` y que una instancia de `Direccion` no debe ser compartida con ningún otro objeto salvo la instancia de `Empleado` que lo posee.



Con esta representación, no sólo la información de la dirección se encapsula de forma limpia dentro de un objeto, sino que otras entidades como `Empresa` pueden también utilizar el tipo y tener sus propios atributos de con objetos embebidos de tipo `Direccion`.

Para definir el tipo embebido debemos utilizar la anotación `@Embeddable` en la definición de la clase. Esta anotación sirve para diferenciar la clase de otras clases normales Java. Una vez que la clase ha sido definida como embebible, sus atributos se harán persistentes como parte de la entidad. Los atributos de mapeo de columnas `@Basic`, `@Temporal`, `@Enumerado`, `@Lob` y `@Column` pueden añadirse a los atributos de la clase embebida. El código siguiente muestra un ejemplo.

```
@Embeddable
public class Direccion {
   private String calle;
   private String ciudad;
   private String provincia;
   @Column(name="COD_POSTAL")
   private String codigoPostal;
   // ...
}
```

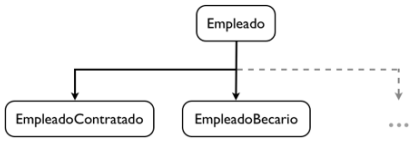
Para usar esta clase en una entidad hay que declararla con la anotación `@Embedded`. Se muestra a continuación.

```
@Entity
public class Empleado {
   @Id private int id;
   private String nombre;
   private Long sueldo;
   @Embedded private Direccion direccion;
   // ...
}
```

Cuando el proveedor realice la persistencia de una instancia de `Empleado` accederá a los atributos del objeto `Direccion` como si estuvieran presentes en la propia instancia. El mapeado de las columnas del tipo `Direccion` se realiza realmente en la tabla `EMPLEADO`.

5.6. Mapeado de las relaciones de herencia

Una de las diferencias fundamentales entre un modelo orientado a objetos y un modelo relacional es la existencia en el primero de herencia entre clases o entidades. La definición de herencia es algo muy natural y útil en modelos orientados a objetos. Por ejemplo, siguiendo con nuestro ejemplo de `Empleado`, supongamos que queremos definir dos tipos de empleado: `EmpleadoContratado` y `EmpleadoBecario`, cada uno de ellos con sus propios atributos adicionales. Supongamos también que cualquier empleado deba ser de uno de los subtipos, de forma que no se permita crear instancias del tipo padre. Para ello, en programación OO deberíamos definir la clase `Empleado` como *abstracta* y las dos clases deberían ser subclases de ella.



JPA permite mapear estas relaciones de herencia en tablas de la base de datos. Existen tres posibles estrategias para realizar este mapeo:

- Tabla única
- Tablas join
- Una tabla por clase

Vamos a ver la estrategia más común, la de tabla única.

En la estrategia de tabla única, todas las clases en la jerarquía de herencia se mapean en una única tabla. Esta tabla contiene almacenadas todas las instancias de todos los posibles subtipos. Los distintos objetos en la jerarquía OO se identifican utilizando una columna especial denominada columna discriminante (*disriminator column*). Esta columna contiene un valor distinto según la clase a la que pertenezca el objeto. Además, las columnas que no se correspondan con atributos de un tipo dado se rellenan con NULL.

Supongamos que un `EmpleadoBecario` tiene un atributo `SeguroMedico` de tipo `Long` con la aportación que debe realizar la empresa para el seguro del empleado. Y el `EmpleadoContratado` tienen un atributo `PlanPensiones` de tipo `Long` con la aportación para el plan de pensiones.

En la figura siguiente aparece la tabla única que guardaría entidades de ambos tipos. La columna discriminante es la columna `Tipo`. La tabla contiene todos los registros. Los registros que se corresponden con empleados contratados tienen el valor `Contrato` en la columna discriminante y los empleados becarios `Beca`. Las columnas que no se correspondan con el tipo de entidad están a NULL.

ID	Nombre	Salario	Tipo	PlanPensiones	SeguroMedico
1	Antonio	2.300	Contrato	230	NULL
2	Juan	1.200	Beca	NULL	150
3	María	2.400	Contrato	240	NULL

Para implementar la herencia en JPA se utiliza la anotación `@Inheritance` en la clase padre. En esta anotación hay que indicar la estrategia de mapeado utilizada con el elemento `strategy`. También hay que añadir a la clase padre la anotación `@DiscriminatorValue` que indica la columna discriminante. El tipo de la columna discriminante se define con el elemento `discriminatorType` que puede tomar como valor las constantes `DiscriminatorType.STRING`, `DiscriminatorType.INTEGER` o `DiscriminatorType.CHAR`.

El siguiente código muestra cómo sería la definición de la clase `Empleado`. Hay que hacer notar la declaración de la clase como `abstract` que impide crear instancias.

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="Tipo",
discriminatorType=DiscriminatorType.STRING)
public abstract class Empleado {
   ...
}
```

Las subclases de la jerarquía se definen igual que en Java estándar (recordemos que las entidades son clases Java normales) con la palabra clave `extends`. Lo único que hay que añadir es el valor en la columna discriminante que se le asigna a esta clase. Para ello se utiliza la anotación `DiscriminatorValue` y su atributo `value`. Por ejemplo, si el `EmpleadoBecario` va a tener como valor la cadena `Beca`, hay que indicar:

```
@DiscriminatorValue(value="Beca")
```

Las definiciones de las subclases las mostramos a continuación. Primero la clase `EmpleadoContratado`, que se asocia al valor `Contrato` de la columna discriminante. En la nueva clase se define al atributo específico `Long planPensiones`.

```
import javax.persistence.*;

@Entity
@DiscriminatorValue(value="Contrato")
public class EmpleadoContratado extends Empleado {
   private Long planPensiones;

   public Long getPlanPensiones() {
      return planPensiones;
   }

   public void setPlanPensiones(Long planPensiones) {
      this.planPensiones = planPensiones;
   }
}
```

Por último, la clase `EmpleadoBecario` se distingue por el valor `Beca`. En la clase se define el nuevo atributo `Long seguroMedico`

```
@Entity
@DiscriminatorValue(value="Beca")
public class EmpleadoBecario extends Empleado {
   private Long seguroMedico;

   public Long getSeguroMedico() {
      return seguroMedico;
   }

   public void setSeguroMedico(Long seguroMedico) {
      this.seguroMedico = seguroMedico;
   }
}
```

Con esto es suficiente. Una vez definidas, las entidades se usan como clases Java normales:

```
Empleado emp = new EmpleadoContratado();
emp.setId(id);
emp.setNombre(nombre);
emp.setSueldo(sueldo);
emp.setPlanPensiones(sueldo/10);
```

6. Ejercicios sesión 3: Mapeado entidad-relación, tablas

En esta sesión de ejercicios vamos a practicar con el mapeado de entidades en tablas y columnas. El mapeado de relaciones lo dejamos para la sesión siguiente.

Vamos a continuar con el ejemplo anterior en el que definíamos autores y mensajes. En esta sesión de ejercicios añadiremos algunas funcionalidades más, introduciendo la posibilidad de que los autores definan recursos (ficheros PDF, imágenes, páginas HTML, ...) y etiquetas. La aplicación que estamos construyendo permitirá para anotar la autoría de estos recursos y etiquetarlos. Este escenario nos servirá para practicar el mapeado de las relaciones de herencia, introduciendo dos tipos de recursos: páginas HTML y documentos PDF.

Vamos a usar los mismos proyectos (`jpa-mensajes` y `jpa-mensajes-web`).

6.1. Creación de la nueva entidad Tag

Vamos a añadir nuevas entidades a las que ya tenemos. Vamos a definir recursos y etiquetas. En la próxima sesión relacionaremos los autores con los recursos, definiendo una relación uno-a-muchos entre autores y recursos y una relación muchos-a-muchos entre recursos y tags.

La idea es definir las bases para una aplicación web con la que los autores pueden crear mensajes y recursos, y añadir etiquetas a estos últimos.

1. Comenzamos creando la entidad `Tag` junto a las otras entidades ya existentes (en el proyecto `jpa-mensajes`). Las instancias de esta entidad van a servir para etiquetar mensajes y recursos con una cadena de texto. La entidad debe tener los siguientes atributos:

- `int id`: identificador de la etiqueta (autogenerado)
- `String cadena`: cadena de texto que representa la etiqueta. Puede contener espacios.
- `java.util.Date fCreacion`: fecha en la que se creo por primera vez la etiqueta
- `int ocurrencias`: número de documentos etiquetados con ese tag

Algunos ejemplos de cadenas de etiquetas: `'curso'`, `'java'`, `'tutorial'`

Define la clase entidad `Tag` con los atributos anteriores. Define un constructor vacío `protected` y los setters y getters necesarios. En la aplicación web que ya tienes montada del ejercicio anterior, crea una función privada `addTag()` (sin parámetros) que cree una nueva `Tag` y que la haga persistente en la base de datos. Despliega la aplicación web, ejecuta la nueva acción y comprueba la nueva tabla con el administrador de MySQL. ¿Qué tipos de columna se han definido?

2. Define la clase `TagDAO` en el mismo paquete con los métodos que vamos a usar para trabajar con la entidad persistente:

- `Tag createTag(String cadena)`: crea una etiqueta, la inicializa con la fecha actual del sistema, le pone el numero de ocurrencias a 0 y la hace persistente.
- `Tag cambiaCadena(Tag tag, String cadena): sustituye la etiqueta de la cadena por una nueva.`
- `List<Tag> listaTags()`: devuelve todas las etiquetas creadas

6.2. Creación de la nueva entidad Recurso

Vamos ahora a realizar la construcción de la entidad con la que representamos los recursos.

1. Crea un tipo embebido `DatosFichero` con los siguientes atributos:

- `String nombre`: nombre del fichero (incluyendo su ruta y extensión)
- `Double size`: tamaño en Kbytes del fichero
- `java.util.Date fCreacion`: fecha de creación del fichero

2. Crea la nueva entidad `Recurso` con los siguientes atributos:

- `int idRecurso`: identificador del recurso (autogenerado)
- `DatosFichero fichero`: datos del fichero
- `String resumen`: descripción larga con el contenido del recurso. Debe ser un tipo LOB.
- `int visitas`: número de veces que se ha consultado el recurso

Define un constructor vacío y los getters y setters necesarios. En la aplicación web que ya tienes montada, crea una función `addRecurso()` (sin parámetros) que cree un nuevo `Recurso` y que lo haga persistente en la base de datos. Despliega la aplicación web, ejecuta la nueva acción y comprueba la nueva tabla con el administrador de MySQL. ¿Qué tipos de columna se han definido?

6.3. Creación de las entidades hijas

1. Crea las nuevas entidades `PaginaHtml` y `FicheroPDF`, subentidades de `Recurso`. Utiliza el método de la tabla única, y escoge tu mismo la columna y el valor discriminante.

En la entidad `PaginaHTML` define los siguientes atributos:

- `String contenido`: Contenido HTML de la página. Debe ser un tipo LOB
- `boolean imagenes`: True o false, dependiendo de si la página enlaza a imágenes

En la entidad `FicheroPDF` define los siguientes atributos adicionales:

- `int páginas`: número de páginas
- `String titulo`: título del PDF

2. Implementa ambas clases y modifica la función `addRecurso` para comprobar la tabla

creada en el mapeado.

3. Implementa una clase `RecursoDAO` que implemente los siguientes métodos de acceso a las entidades persistentes:

- Recurso createRecurso(String nombreFichero, String resumen, TipoRecurso tipo)
- void updatePaginaHTML(PaginaHTML pagina, String contenido, boolean imagenes)
- void updateFicheroPDF(FicheroPDF pdf, int paginas, String titulo)
- Recurso findRecursoById(int idRecurso)
- Collection<Recurso> findRecursoByName(String name)

6.4. Interfaz web para etiquetas (*)

Define un servlet `doActionTag`, similar al implementado en la sesión 2, con el que podamos probar las distintas funcionalidades del `TagDAO`.

Define una página HTML con la que podamos introducir los parámetros necesarios para el servlet. Debes pedir la cadena del tag y una acción a realizar.

Implementa como mínimo las acciones:

- `add-tag`: añade una etiqueta
- `listar-tags`: lista todas las etiquetas
- `buscar-tag`: busca una etiqueta

Implementa una página JSP que muestre el resultado de cada acción.

6.5. Interfaz web para recursos (*)

Haz lo mismo para el `RecursoDAO`.

Define una páginas HTML con las que podamos introducir nuevos recursos, llamando al método `createRecurso` del RecursoDAO. Implementa como mínimo las acciones:

- `add-pagina`: añade una página
- `buscar-pagina`: busca una página por el nombre

Implementa una página JSP que muestre el resultado de cada acción.

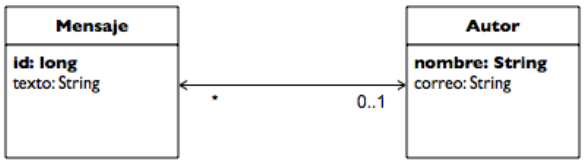
7. Mapeo entidad-relación: relaciones

7.1. Conceptos previos

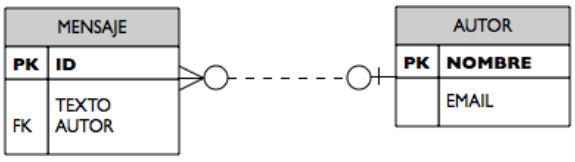
Antes de comenzar a detallar los aspectos del mapeo de las relaciones entre entidades vamos a repasar algunos conceptos básicos y alguna terminología. Es muy importante tener claro estos conceptos antes de intentar entender los entresijos a los que nos vamos a enfrentar más adelante.

Vamos a ilustrar estos conceptos con la relación que vimos en la sesión 1. Allí definíamos dos entidades, `Autor` y `Mensaje` y una relación muchos-a-uno bidireccional de `Mensaje` a `Autor`. Un autor puede escribir muchos mensajes.

La relación la representamos con la figura siguiente, en donde se muestran las dos entidades unidas con una flecha con dos puntas que indica que la relación es bidireccional y con la anotación de la cardinalidad de cada entidad bajo la flecha.



El modelo físico que se genera con el mapeado es el que se muestra en la siguiente figura. La tabla `MENSAJE` contiene una clave ajena hacia la tabla `AUTOR`, en su columna `AUTOR`. Los valores de esta columna guardan claves primarias de autores. De esta forma relacionamos cada mensaje con el autor que lo ha escrito. Cuando queramos obtener todos los mensajes de un determinado autor se deberá hacer un `SELECT` en la tabla `MENSAJE` buscando todas las filas que tengan como clave ajena el autor que buscamos.



Vamos a ver las distintas características que hay que definir para especificar completamente una relación en el modelo de entidades, utilizando como ejemplo esta relación entre `Mensaje` y `Autor`.

7.1.1. Direccionalidad

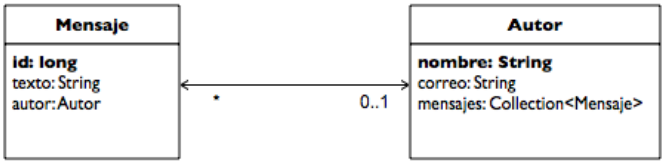
En primer lugar debemos considerar la *direccionalidad* de la relación. Nos indica si desde

una entidad podemos obtener la otra. Una relación puede ser *unidireccional* cuando desde la *entidad origen* se puede obtener la *entidad destino* o *bidireccional*, como es el caso del ejemplo, cuando desde ambas partes de la relación se puede obtener la otra parte.

En un diagrama UML, la direccionalidad viene indicada por la dirección de la flecha. En el caso del ejemplo anterior tenemos una relación bidireccional. Podemos pedirle a una instancia de `Mensaje` que nos diga con que `Autor` está relacionado. En la entidad de `Mensaje` habrá un método que devuelve el autor. Y al ser la relación bidireccional podemos hacer lo mismo al revés, podemos pedirle a un `Autor` que nos diga con qué `Mensajes` está relacionado. En la entidad `Autor` hay un método que devuelve una colección de mensajes. Por tanto, los métodos definidos por la relación son:

- En la entidad `Mensaje`: `public Autor getAutor()`
- En la entidad `Autor`: `public Collection<Mensaje> getMensajes()`

Podríamos haber escrito estos atributos en la figura anterior, en las entidades, como se muestra en la siguiente figura:



Ambos diagramas son equivalentes. El segundo proporciona más información porque indica el nombre que le damos a los atributos de la relación.

7.1.2. Cardinalidad

La cardinalidad de una relación define el número de instancias de una entidad que pueden estar relacionada con otras. Atendiendo a la cardinalidad, una relación puede ser *uno-a-uno*, *uno-a-muchos*, *muchos-a-uno* o *muchos-a-muchos*.

En el caso del ejemplo tenemos una relación muchos-a-uno entre `Mensaje` y `Autor`. Al ser una relación bidireccional, también podríamos considerar esta relación como una relación uno-a-muchos entre `Autor` y `Mensaje`.

7.1.3. Entidad propietaria de la relación

¿Cómo se realiza el mapeo de una relación? La forma más habitual es definir una columna adicional en la tabla asociada a una de las entidades con la clave primaria de la otra tabla con la que está relacionada. Esta columna hace de *clave ajena* hacia la otra tabla. Decimos que esta entidad cuya tabla contiene la clave ajena hacia la otra es la *propietaria* de la relación.

En el ejemplo visto, la relación se mapea haciendo que la tabla `MENSAJE` contenga una clave ajena hacia la tabla `AUTOR`, como hemos visto en la figura. Decimos entonces que la

entidad `Mensaje` es la propietaria de la relación.

Para especificar las columnas que son claves ajenas en una relación se utilizan las anotaciones `@JoinColumn` y `mappedBy`. La primera es opcional y se puede identificar con los valores por defecto. La segunda es obligatoria y es un atributo de la anotación que define la relación: `OneToOne`, `OneToMany` o `ManyToMany`. Como veremos más adelante, este elemento debe indicar el nombre del atributo que representa la clave ajena en la otra entidad.

También es posible mapear una relación utilizando una *tabla join*, una tabla auxiliar con dos claves ajenas hacia las tablas de la relación. No nos da tiempo en este tema de explicar cómo hacerlo. Si estás interesado puedes consultar en la especificación de JPA.

7.1.4. Sincronización con la base de datos

Ya hemos comentado previamente que una de las características fundamentales del funcionamiento de JPA es que las instancias de las entidades viven en el contexto de persistencia asociado al entity manager hasta que se realiza una operación de *flush* sobre la base de datos. Normalmente esto sucede cuando se cierra la transacción. En ese momento es cuando el proveedor de persistencia consulta las instancias del contexto y genera las sentencias SQL que actualizan la base de datos. Es interesante tener activo el debug de Hibernate que muestra las sentencias generadas para comprobar en qué momento se realizan y cuáles son.

En el caso de las relaciones, la especificación de JPA avisa literalmente de lo siguiente (el énfasis es del documento original):

Es particularmente importante asegurarse de que los cambios en el lado inverso de una relación se actualizan de forma apropiada en el lado propietario, para asegurarse de que los cambios no se pierden cuando se sincronizan en la base de datos.

Esto es, para que una relación se sincronice correctamente en la base de datos, se debe actualizar la instancia a la que se refiere en la parte propietaria de la relación. En el caso del ejemplo, para añadir una relación entre un `Mensaje` y un `Autor`, debemos añadir el autor asociado al mensaje en la instancia `Mensaje` (la propietaria de la relación). No te preocupes si no te queda demasiado claro. Lo explicamos mejor más adelante con algunos ejemplos, cuando veamos tipos de relaciones concretos.

7.2. Definición de relaciones

Detallamos a continuación cómo se especifican las relaciones utilizando anotaciones.

7.2.1. Relación uno-a-uno unidireccional

En una relación uno-a-uno unidireccional entre la entidad A y la entidad B, una instancia de la entidad A referencia una instancia de la entidad B. En la relación A pondremos la

anotación `@OneToOne` en el atributo referente. En la entidad B no hay que añadir ninguna anotación.

Un ejemplo podría ser la relación entre un `Empleado` y un `Despacho` (asumiendo que `Despacho` es una entidad, y no un objeto embebido). El código sería el siguiente:

```
@Entity
public class Empleado {
   // ...
   @OneToOne
   private despacho Despacho;
   // ...

   public void setDespacho(Despacho despacho) {
      this.despacho = despacho;
   }
}

@Entity
public class Despacho {
   @Id
   String idDespacho;

   // ...
}
```

La entidad `Empleado` es la propietaria de la relación y en su tabla hay una clave ajena (en la columna `despacho`) que apunta a la otra tabla. Se puede especificar esta clave ajena con la anotación `@JoinColumn`:

```
public class Empleado {
        // ...
   @OneToOne
   @JoinColumn(name="despacho",
   referencedColumnName="idDespacho, updatable=false)
   private Despacho despacho;
   //...
```

En esta anotación `@JoinColumn` se especifica el nombre de la columna, el nombre de la columna con la clave primaria en la tabla a la que se apunta y se indica que la columna no puede ser modificada mediante un update.

En el caso de no utilizar la anotación, JPA realizará el mapeo utilizando como nombre de columna el nombre del atributo que guarda la otra entidad y el nombre de la columna con la clave primaria en la otra entidad, uniéndolas con un subrayado ("_"). En este caso, se llamaría `empleado_idDespacho`.

Para actualizar la relación, basta con llamar al método `setDespacho()`:

```
em.getTrasaction().begin();
Empleado emp = new Empleado("John Doe");
Despacho desp = new Despacho("C1");
desp.setEmpleado(emp);
em.persist(desp);
```

```
em.persist(emp);
em.getTransaction().commit();
```

7.2.2. Relación uno-a-uno bidireccional

Supongamos que necesitamos modificar la relación anterior para que podamos obtener el empleado a partir de su despacho. Basta con hacer que la relación anterior sea bidireccional. Para ello, debemos añadir el atributo `empleado` en la entidad `Despacho` y anotarlo con `@OneToOne(mappedBy="despacho")`. De esta forma estamos indicando que la entidad `Empleado` es la propietaria de la relación y que contiene la clave ajena en el atributo `despacho`.

```
@Empleado
public class Empleado {
   // ...
   @OneToOne
   private despacho Despacho;
   // ...

   public void setDespacho(Despacho despacho) {
      this.despacho = despacho;
   }
}

@Entity
public class Despacho {
   // ...
   @OneToOne(mappedBy="despacho")
   private empleado Empleado;
   // ...

   public Empleado getEmpleado() {
      return this.empleado;
   }
}
```

Hay que destacar la utilización del elemento `mappedBy` en el atributo `despacho`. Con ese elemento estamos indicando al proveedor de persistencia cuál es la clave ajena en la entidad `Empleado`.

Para actualizar la relación hay que hacer lo mismo que antes, utilizar el método `setDespacho()` del objeto `Empleado` pasándole el `Despacho` con el que está relacionado. Hay que hacer notar que es muy importante actualizar el lado de la relación que es el propietario de la misma y que contiene la clave ajena. Si lo hiciéramos al revés y actualizáramos el atributo `empleado` del `Despacho` tendríamos la desagradable sorpresa de que la relación **no se actualiza** en la base de datos. Por ello **sería incorrecto hacer lo siguiente**:

```
em.getTransaction().begin();
Empleado emp = new Empleado("John Doe");
Despacho desp = new Despacho("C1");
desp.setEmpleado(emp);
```

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

Sería incorrecto porque estamos actualizando el lado inverso de la relación, no el lado propietario. Aunque la relación es bidireccional, la actualización sólo se puede hacer en el lado del propietario de la relación. Para evitar posibles errores, lo mejor es implementar únicamente el método `set` en el lado del propietario de la relación y en el otro lado (en la entidad `Empleado`) poner como `private` el atributo de la relación y no definir ningún método `set`.

7.2.3. Relación uno-a-muchos/muchos-a-uno bidireccional

En una relación uno-a-muchos bidireccional entre la entidad A y la B (que es equivalente a una relación muchos-a-uno bidireccional entre B y A), una instancia de la entidad A referencia a una colección de instancias de B y una instancia de B referencia a una instancia de A. La entidad B debe ser la propietaria de la relación, la que contiene la clave ajena a la entidad A.

En la entidad B se define un atributo del tipo de la entidad A, con la anotación `@ManyToOne`. En la entidad A se define un atributo del tipo `Collection<B>` con la anotación `@OneToMany(mappedBy="a")` que indica el nombre de la clave ajena.

El ejemplo que presentamos al comienzo del capítulo es de este tipo. Otro ejemplo podría ser el de la relación entre un departamento y los empleados. Un `Departamento` está relacionado con muchos `Empleados`. Y un `Empleado` pertenece a un único `Departamento`. De esta forma el `Empleado` hace el papel de propietario de la relación y es quien llevará la clave ajena hacia `Departamento`. Lo vemos en el siguiente código:

```
@Entity
public class Empleado {
   // ...
   @ManyToOne
   @JoinColumn(name="departamento",
referencedColumnName="idDepartamento")
   private departamento Departamento;
   // ...
   public void setDepartamento(Departamento departamento) {
      this.departamento = departamento;
   }
}

@Entity
public class Departamento {
   // ...
   @OneToMany(mappedBy="departamento")
   private Collection<Empleado> empleados = new HashSet();
   // ...

   public Collection<Empleado> getEmpleados() {
      return this.empleados;
   }
}
```

En la declaración se podría eliminar la anotación `@JoinColumn` y el nombre de la columna con la clave ajena se obtendría de la misma forma que hemos visto en la relación uno-a-uno.

Como también hemos dicho anteriormente, la relación se actualiza insertando el nuevo `Departamento` en la instancia `Empleado` (la propietaria de la relación) con el método `setDepartamento()`, de esta forma nos aseguraremos que la relación se guarda en la base de datos. Sin embargo, también es importante actualizar la colección en el otro lado de la relación. La llamada a `getEmpleados()` es sobreescrita por JPA y es implementada por una consulta a la base de datos, pero si queremos utilizar inmediatamente la colección de empleados, debemos actualizar su contenido:

```
Collection empleados = depto.getEmpleados();
emp.setDepartemnto(depto);
empleados.add(emp);
```

7.2.4. Relación muchos-a-uno unidireccional

En una relación muchos-a-uno unidireccional entre las entidades A y B, una instancia de A está relacionada con una instancia de B (que puede ser la misma para distintas instancias de A). Para definir la relación hay que definir en la entidad propietaria A un atributo que define la clave ajena hacia la entidad B. Al ser una relación unidireccional no podremos obtener desde la entidad B su entidad relacionada A, por lo que no hay que definir ningún atributo adicional.

Por ejemplo, podríamos definir una relación entre `Empleado` y `Categoria` en la que muchos empleados pueden pertenecer a la misma categoria y no nos interesa listar los empleados de una determinada cualificacion:

```
@Empleado
public class Empleado {
   // ...
   @ManyToOne
   @JoinColumn(name="categoria",
referencedColumnName="idCategoria")
   private categoria Categoria;
   // ...
   public void setCategoria(Categoria categoria) {
      this.categoria = categoria;
   }
}

@Entity
public class Categoria {
   // ...
}}
```

7.2.5. Relación muchos-a-muchos bidireccional

En una relación muchos-a-muchos bidireccional entre la entidad A y la entidad B, una instancia de A está relacionada con una o muchas instancias de B (que pueden ser las mismas para distintas instancias de A). Desde la instancia A se puede obtener la colección de instancias de B con las que está relacionada y viceversa.

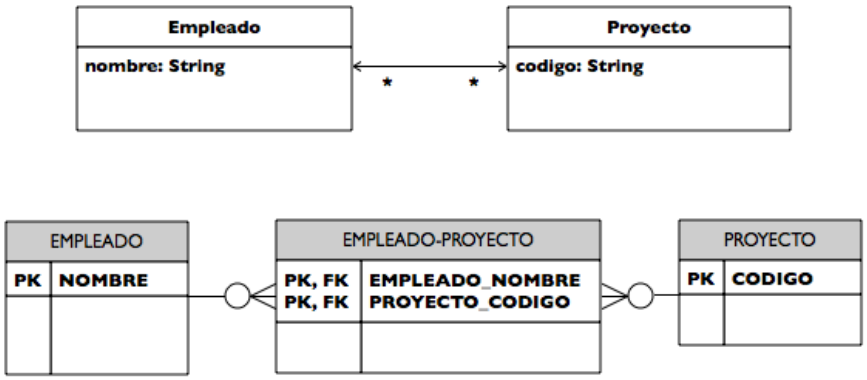
Por ejemplo, un `Empleado` puede participar en más de un `Proyecto`. Dado un empleado se puede obtener la lista de proyectos en los que participa y dado un proyecto podemos obtener la lista de empleados asignados a él.

Se implementa con las siguientes anotaciones:

```
@Entity
public class Empleado {
   @Id String nombre;
   @ManyToMany
   private Collection<Projecto> proyectos = new HashSet();

   public Collection<Proyecto> getProyectos() {
      return this.proyectos;
   }

   public void setProyectos(Collection<Proyecto> proyectos) {
      this.proyectos = proyectos;
   }

   // ...
}

@Entity
public class Proyecto {
   @Id String codigo;
   @ManyToMany(mappedBy="proyectos");
   private Collection<Empleado> empleados = new HashSet();

   public Collection<Empleado> getEmpleados() {
      return empleados;
   }
   // ...
}
```

La anotación `mappedBy` apunta a la entidad propietaria de la relación. En este caso es la entidad `Empleado`.

La relación se mapea, a diferencia de los casos anteriores, utilizando una *tabla join*. Es una tabla con dos columnas que contienen claves ajenas a las tablas de las entidades. La primera columna apunta a la tabla propietaria de la relación y la segunda a la otra tabla. La siguiente imagen representa el mapeo de la relación anterior.



Al igual que en las otras relaciones, para actualizar la relación hay que modificar el lado propietario de la relación, que es el que se sincronizará con la base de datos. En este caso, en el lado propietario (`Empleado`) tenemos una colección de proyectos. Para añadir un nuevo empleado a un proyecto hay que obtener la referencia a la colección de proyectos en los que participa el empleado (con el método `getProyectos()`) y añadir el nuevo proyecto con el método `add()` de la colección:

```
Proyecto proyecto = em.find(Proyecto.class, "P04");
Collection<Proyectos> proyectos = empleado.getProyectos();
proyectos.add(proyecto);
```

Otro ejemplo que podría ser útil es el siguiente, en el que se muestra cómo cambiar a un empleado del proyecto "P01" al proyecto "P04":

```
Empleado empleado = em.find(Empleado.class, "Miguel Garcia");
Proyecto proyectoBaja = em.find(Proyecto.class, "P04");
Proyecto proyectoAlta = em.find(Proyecto.class, "P01");
empleado.getProyectos().remove(proyectoBaja);
empleado.getProyectos().add(proyectoAlta);
```

Es muy interesante analizar un poco más el código. ¿Por qué funciona correctamente el método `remove()`? Hay que recordar que que para que funcione correctamente el método, el objeto que se pasa como parámetro debe ser igual (con el método `equals()`) que el objeto que hay en la colección. Hibernate se encarga de sobrecargar el método `equals()` para que devuelva `true` cuando se refieren a la misma instancia de entidad. De esta forma la instancia que se devuelve con el método `find()` es la misma que hay en la colección obtenida con `getProyectos()`.

7.2.6. Relación muchos-a-muchos unidireccional

En una relación muchos-a-muchos unidireccional entre una entidad A y otra entidad B, cada instancia de la entidad A está relacionada con una o muchas instancias de B (que pueden ser las mismas para distintas instancias de A). Al ser una relación unidireccional, dada una instancia de B no nos interesa saber la instancia de A con la que está

relacionada.

Por ejemplo, un `Empleado` tiene una colección de `Patentes` que ha desarrollado. Distintos empleados pueden participar en la misma patente. Pero no nos interesa guardar la información de qué empleados han desarrollado una patente determinada. La forma de especificarlo es:

```
@Entity
public class Empleado {
   @Id String nombre;
   @ManyToMany
   private Collection<Patentes> patentes = new HashSet();

   public Collection<Patentes> getPatentes() {
      return this.patentes;
   }

   public void setPatentes(Collection<Patentes> patentes) {
      this.patentes = patentes;
   }

   // ...
}

@Entity
public class Patente {
   // ...
}
```

En el mapeo de la relación se crea una tabla join llamada `EMPLEADO_PATENTE` con las dos claves ajenas hacia `EMPLEADO` y `PATENTE`.

La forma de actualizar la relación es la misma que en la relación bidireccional.

7.3. Carga perezosa

Hemos comentado que una de las características de JPA es la *carga perezosa* (*lazy fetching* en inglés). Este concepto es de especial importancia cuando estamos trabajando con relaciones.

Supongamos los ejemplos anteriores, con la relación uno-a-muchos entre `Departamento` y `Empleado` y la relación muchos-a-muchos entre `Empleado` y `Proyecto`. Cuando recuperamos un departamento de la base ese departamento está relacionado con un conjunto de empleados, que a su vez están relacionados cada uno de ellos con un conjunto de proyectos. La carga perezosa consiste en que no se cargan todos los objetos en el contexto de persistencia, sino sólo la referencia al departamento que se ha recuperado. Es cuando se realiza una llamada al método `getEmpleados()` cuando se recuperan de la base de datos todas las instancias de `Empleado` con las que ese departamento está relacionado. Pero sólo los empleados. Los proyectos de cada empleado no se cargan hasta que no se llama al método `getProyectos()` del empleado que nos interese.

Esta funcionalidad hace que la carga de instancias sea muy eficiente, pero hay que saber utilizarla correctamente. Un posible problema puede surgir cuando la instancia original (el `Departamento`) queda desconectada (*detached*) del *EntityManager* (lo veremos en el tema siguiente). Entonces ya no podremos acceder a ningún atributo que no se haya cargado.

Es posible que quereamos desactivar la propiedad por defecto de la carga perezosa. Para ello debemos definir la opción `fetch=FetchType.EAGER` en la anotación que define el tipo de relación:

```
@Entity
public class Empleado {
   @Id String nombre;
   @ManyToMany(fetch=FetchType.EAGER)
   private Collection<Patentes> patentes = new HashSet();
   // ...
}
```

También es posible configurar el proveedor de persistencia para que la opción por defecto sea `EAGER`. En este caso podemos definir una relación como perezosa con la anotación `fetch=FetchType.LAZY`.

8. Ejercicios sesión 4: Mapeado entidad-relación, relaciones

En esta sesión de ejercicios vamos a añadir relaciones a las entidades creadas en la sesión anterior.

8.1. Relación uno-a-muchos entre Autor y Recurso

1. Define una relación uno-a-muchos entre `Autor` y `Recurso`. Modifica las entidades para implementar la relación. Crea una función nueva (por ejemplo, `add-autor-pagina()` (sin parámetros) que busque un autor y una página ya creados y que los relacione. Despliega la aplicación en el servidor web, y comprueba las tablas de la base de datos. ¿Qué nuevas columnas se han añadido? ¿Se ha creado alguna tabla adicional?

2. Añade en la clase `RecursoDAO` el método `setAutorRecurso` que actualice la relación.

3. Modifica la página `index.html` y añade parámetros a la función nueva, para que podamos añadir una página a un autor, introduciendo el nombre de la página y el nombre del autor. Comprueba que funciona correctamente.

8.2. Relación muchos-a-muchos entre Tag y Recurso

1. Define una relación muchos-a-muchos entre `Tag` y `Recurso`. Modifica las entidades, añadiendo la anotación `@ManyToMany(mappedBy="tags")` junto al atributo `recursos` de la entidad `Tag` y la anotación `@ManyToMany` junto al atributo `tags` de la entidad `Recurso`. Crea una función nueva (por ejemplo, `add-tag-recurso()` sin parámetros) que busque una tag cualquiera y una página ya creadas y que los relacione. Despliega la aplicación en el servidor web, y comprueba las tablas de la base de datos. ¿Qué nuevas tablas se han creado? ¿Qué columnas tiene y qué representan?

2. Cambia ahora el sentido del *mappedBy*, definiéndolo en la entidad `Recurso`. Borra con el administrador de base de datos la tabla creada en el punto anterior y vuelve a ejecutar la aplicación. ¿Has tenido que cambiar el código? ¿Qué tabla se crea ahora?. Elimina por último la anotación *mappedBy* de ambas tablas y vuelve a comprobar qué tabla se crea.

8.3. DAOs

Modifica los DAOs para incluir los métodos de actualización de las relaciones anteriores.

Haz que la función del DAO que asocie una etiqueta con una página aumente también el número de ocurrencias de la etiqueta.

8.4. Interfaz web para tags (*)

Modifica las páginas HTML, los servlets y las páginas JSP para poder añadir tags a un recurso y, dado un recurso, listar sus tags asociados.

9. Consultas

9.1. Definición de consultas

El API JPA proporciona la interfaz `Query` para configurar y ejecutar consultas. Podemos obtener una instancia que implemente esa interfaz mediante un dos de 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.

9.1.1. Consultas dinámicas

Se puede crear una consulta dinámica pasándole la cadena con la consulta al método `createQuery`. El proveedor de persistencia transforma en ese momento la consulta en el código SQL que se ejecutará en la base de datos. Veamos un ejemplo de consulta, que utiliza el paso de parámetros por nombre visto anteriormente:

```
public class EmpleadoDAO {
           private static final String QUERY =
              "SELECT e.salary " +
              "FROM Empleado e " +
              "WHERE e.departamento.nombre = : deptNombre AND " +
              "      e.nombre = : empNombre";

           // ...
           public long queryEmpleadoSalario(String empNombre,
String deptNombre) {
              return (Long) em.createQuery(QUERY)
                              .setParameter("deptNombre",
deptNombre)
                              .setParameter("empNombre", empNombre)
                              .getSingleResult();
           }
        }
```

Un inconveniente de las consultas dinámicas es que se procesan en tiempo de ejecución de la aplicación, con la consiguiente pérdida de rendimiento. Su ventaja principal es que se pueden definir con mucha comodidad.

9.1.2. Consultas con nombre

Las consultas con nombre se definen junto con la entidad, utilizando la anotación

`@NamedQuery`. Mejoran el rendimiento con respecto a las consultas dinámicas, ya que son procesadas una única vez. El siguiente código muestra un ejemplo:

```
@NamedQuery(name="salarioPorNombreDepartamento",
                  query="SELECT e.salary " +
                        "FROM Empleado e " +
                        "WHERE e.departamento.nombre = : deptNombre
AND " +
                        "      e.nombre = : empNombre")
```

Si se necesita más de una consulta para una entidad, deben incluirse en la anotación `@NamedQueries`, que acepta una array de una o más anotaciones `@NamedQuery`:

```
@NamedQueries({
           @NamedQuery(name="Empleado.findAll",
                       query="SELECT e FROM Empleado e"),
           @NamedQuery(name="Empleado.findById",
                       query="SELECT e FROM Empleado e WHERE e.id =
:id"),
           @NamedQuery(name="Empleado.findByNombre",
                       query="SELECT e FROM Empleado e WHERE
e.nombre = :nombre")
        })
```

Para ejecutar la consulta hay que llamar al método `createNamedQuery` pasándole como parámetro el nombre de la consulta. El siguiente código muestra un ejemplo:

```
public class EmpleadoDAO {
           // ...
           public Empleado findEmpleadoByNombre(String nombre) {
              return (Empleado)
em.createNamedQuery("Empleado.findByNombre")
                                    .setParameter("nombre", nombre)
                                    .getSingleResult();
           }
           public List<Empleado> findAll() {
              return (List<Empleado>)
em.createNamedQuery("Empleado.findAll")
                                       .getResultList();
           }
        }
```

9.2. Ejecución de consultas

Veamos por último los métodos del interfaz `Query` para ejecutar las consultas definidas. Son los métodos `getSingleResult()` y `getResultList()`.

Ambos métodos se deben lanzar sobre una `Query` ya construida y en la que se han introducido los parámetros. El método `getSingleResult()` se utiliza con consultas que devuelven un único resultado. Devuelve un `Object` que contiene el resultado de la consulta. Después de llamarlo conviene hacer un casting al tipo (entidad o tipo básico) que esperamos. Puede suceder que la consulta ejecutada no devuelva ningún resultado o devuelva más de uno. En el primer caso se genera la excepción `NoResultException` y en

el segundo `NonUniqueResultException`.

Si no tenemos seguridad de una consulta vaya a devolver un único valor deberíamos llamar a `getResultList()`. Este método devuelve una `List` de `Object`. La utilización de la interfaz `List` en lugar de `Collection` es para soportar la devolución de colecciones ordenadas, generadas por consultas con la cláusula `ORDER BY`. En el caso en que la consulta no obtenga resultados, se devuelve una lista vacía. El siguiente código muestra un ejemplo de utilización de una consulta que devuelve una lista ordenada:

```
public void muestraEmpleadosProyecto(String nombreProyecto) {
           List<Empleado> result = em.createQuery(
                                       "SELECT e " +
                                       "FROM Proyecto p JOIN
p.empleados e " +
                                       "WHERE p.nombre = ?1 " +
                                       "ORDER BY e.name")
                                       .setParameter(1,
nombreProyecto)
                                       .getResultList();
           for (empleado : result) {
                      System.out.println(empleado.nombre);
           }
       }
```

9.3. Java Persistence Query Language

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

9.3.1. Consultas básicas

La consulta más sencilla en JPQL es la que selecciona todas las instancias de un único

tipo de entidad:

```
SELECT e FROM Empleado e
```

La sintaxis de JPQL es similar a la de SQL. De esta forma los desarrolladores con experiencia en SQL pueden comenzar a utilizarlo rápidamente. La diferencia principal es que en SQL se seleccionan filas de una tabla mientras que en JPQL se seleccionan instancias de un tipo de entidad del modelo del dominio. La cláusula `SELECT` es ligeramente distinta a la de SQL listando sólo el alias `e` del `Empleado`. Este tipo indica el tipo de datos que va a devolver la consulta, una lista de cero o más instancias `Empleado`.

A partir del alias, podemos obtener sus atributos y recorrer las relaciones en las que participa utilizando el operador punto (.). Por ejemplo, si sólo queremos los nombres de los empleados podemos definir la siguiente consulta:

```
SELECT e.nombre FROM Empleado e
```

Ya que la entidad `Empleado` tiene un campo persistente llamado `nombre` de tipo `String`, esta consulta devolverá una lista de cero o más objetos de tipo `String`.

Podemos también seleccionar una entidad cuyo tipo no listamos en la cláusula `SELECT`, utilizando los atributos relación. Por ejemplo:

```
SELECT e.departamento FROM Empleado e
```

Debido a la relación muchos-a-uno entre `Empleado` y `Departamento` la consulta devolverá una lista de instancias de tipo `Departamento`.

9.3.2. Filtrando los resultados

Igual que SQL, JPQL contiene una cláusula `WHERE` para especificar condiciones que deben cumplir los datos que se devuelven. La mayoría de los operadores disponibles en SQL están en JPQL, incluyendo las operaciones básicas de comparación: `IN`, `LIKE` y `BETWEEN`, funciones como `SUBSTRING` o `LENGTH` y subqueries. Igual que antes, la diferencia fundamental es que se utilizan expresiones sobre entidades y no sobre columnas. El siguiente código muestra un ejemplo:

```
SELECT e
   FROM Empleado e
   WHERE e.departamento.name = 'NA42' AND
         e.direccion.provincia IN ('ALC','VAL')
```

9.3.3. Parámetros de las consultas

JPQL soporta dos tipos de sintaxis para la ligadura (*binding*) de parámetros: posicional y por nombre. Vemos un ejemplo del primer caso:

```
SELECT e
   FROM Empleado e
```

```
   WHERE e.departamento = ?1 AND
         e.salario > ?2
```

Veremos en el siguiente apartado como ligar parámetros determinados a esas posiciones. La segunda forma de definir parámetros es por nombre:

```
SELECT  e
   FROM Empleado e
   WHERE e.departamento = :dept AND
         e.salario > :sal
```

9.3.4. Proyectando los resultados

Es posible recuperar sólo alguno de los atributos de las instancias. Esto es útil cuando tenemos una gran cantidad de atributos (columnas) y sólo vamos a necesitar listar algunos. Por ejemplo, la siguiente consulta selecciona sólo el nombre y el salario de los empleados:

```
SELECT e.nombre, e.salario
   FROM Empleado e
```

El resultado será una colección de cero o más instancias de arrays de tipo `Object`. Cada array contiene dos elementos, el primero un `String` y el segundo un `Double`:

```
List result = em.createQuery(
   "SELECT e.nombre, e.salario " +
   "FROM Empleado e WHERE e.salario > 30000 " +
   "ORDER BY e.nombre").getResultList();
Iterator empleados = result.iterator();
while (empleados.hasNext()) {
   Object[] tupla = (Object[]) empleados.next();
   String nombre = (String) tupla[0];
   int salario = ((Integer) tupla[1]).intValue();
   ...
}
```

9.3.5. Joins entre entidades

Al igual que en SQL es posible definir consultas que realicen una selección en el resultado de unir (*join*) entidades entre las que se ha establecido una relación. Por ejemplo, el siguiente código muestra un join entre las entidades `Empleado` y `CuentaCorreo` para recuperar todos los correos electrónicos de un departamento específico:

```
SELECT c.correo
   FROM Empleado e, CuentaCorreo c
   WHERE e = c.empleado AND
         e.departamento.nombre = 'NA42'
```

En JPQL también es posible especificar joins en la cláusula `FROM` utilizando el operador `JOIN`. Una ventaja de este operador es que el join puede especificarse en términos de la

propia asociación y que el motor de consultas proporcionará automáticamente el criterio de join necesario cuando genere el SQL. El siguiente código muestra la misma consulta reescrita para utilizar este operador:

```
SELECT c.correo
   FROM Empleado e JOIN e.cuentasCorreo c
   WHERE e.departamento.nombre = 'NA42'
```

JPQL soporta múltiples tipos de joins, incluyendo inner y outer joins, left joins y una técnica denominada fetch joins para cargar datos asociados a las entidades que no se devuelven directamente. No tenemos tiempo de detallar todos ellos. Vamos a ver algunos ejemplos más, para tener una idea de la potencia de la sintaxis.

Selecciona todos los departamentos distintos asociados a empleados:

```
SELECT DISTINCT e.departamento FROM Empleado e
```

Otra forma de hacer la misma consulta utilizando el operador `JOIN`:

```
SELECT DISTINCT d FROM Empleado e JOIN e.departamento d
```

Selecciona los departamentos distintos que trabajan en Alicante y que participan en el proyecto 'BlueBook':

```
SELECT DISTINCT e.departamento
   FROM Proyecto p JOIN p.empleados e
   WHERE p.nombre = 'BlueBook' AND
         e.direccion.localidad = 'ALC'
```

Selecciona los proyectos distintos que pertenecen a empleados de un departamento:

```
SELECT DISTINCT p
   FROM Departamento d JOIN d.empleados JOIN e.proyectos p
```

Selecciona todos los empleados y recupera (para evitar el lazy loading) la entidad `Direccion` con la que está relacionado cada uno:

```
SELECT e
   FROM Empleado e JOIN FETCH e.direccion
```

9.3.6. Paginación de resultados

Es posible realizar un paginado de los resultados, definiendo un número máximo de instancias a devolver en la consulta y un número de instancia a partir del que se construye la lista:

```
Query q = em.createQuery("SELECT e FROM Empleado e");
q.setFirstResult(20);
q.setMaxResults(10);
List empleados = q.getResultList();
```

9.3.7. Subqueries

Es posible anidar múltiples queries. Podemos utilizar el operador `IN` para obtener las instancias que se encuentran en el resultado de la subquery. Por ejemplo, la siguiente expresión devuelve los empleados que participan en proyectos de tipo 'A'.

```
SELECT e FROM Empleado e
WHERE e.proyecto IN (SELECT p
   FROM Proyecto p
   WHERE p.tipo = 'A')
```

9.3.8. Valores nulos y colecciones vacías

Es posible utilizar los operadores `IS NULL` o `IS NOT NULL` para chequear si un atributo es o no nulo:

```
WHERE e.departamento IS NOT NULL
```

En el caso de colecciones hay que utilizar los operadores `IS EMPTY` o `IS NOT EMPTY`:

```
WHERE e.proyectos IS EMPTY
```

9.3.9. Funciones

Es posible utilizar llamadas a funciones predefinidas en JPQL. Veamos unos cuantos ejemplos:

- `CONCAT` concatena dos cadenas

  ```
  CONCAT(string1, string2)
  ```

  Por ejemplo:

  ```
  WHERE CONCAT(e.nombre, e.apellido) LIKE 'Ju%cia'
  ```

- `SUBSTRING` extra una subcadena

  ```
  SUBSTRING(string, position, length)
  ```

- `LOWER` devuelve los caracteres de una cadena convertidos en minúsculas

  ```
  LOWER(string)
  ```

- `UPPER` devuelve los caracteres de una cadena convertidos en mayúsculas

  ```
  UPPER(string)
  ```

- `LENGTH` devuelve la longitud de una cadena

  ```
  LENGTH(string)
  ```

- `SIZE` devuelve el tamaño de una colección

  ```
  WHERE SIZE (e.proyectos) > 4
  ```

- `CURRENT_TIME`, `CURRENT_DATE`, `CURRENT_TIMESTAMP` devuelven el tiempo del momento actual

9.3.10. Agrupaciones

JPQL proporciona las siguientes funciones de agrupación: `AVG`,`COUNT`,`MAX`,`MIN`,`SUM` que pueden ser aplicadas a un número de valores. El tipo devuelto por las funciones es `Long` o `Double`, dependiendo de los valores implicados en la operación.

Por ejemplo, para obtener el mayor sueldo de un conjunto de empleados:

```
SELECT MAX(e.sueldo)
FROM Empleado e
```

Para obtener el número de elementos que cumplen una condición:

```
SELECT COUNT(e)
FROM Empleado e
WHERE ...
```

Podemos realizar consultas más complicadas utilizando `GROUP BY` y `HAVING`. Por ejemplo, podemos obtener los empleados y el número de proyectos en los que participan de la siguiente forma:

```
SELECT p.empleado, COUNT(p)
FROM Proyecto p
GROUP BY p.empleado
```

El código para recorrer la lista resultante sería el siguiente:

```
List result = em.createQuery("SELECT p.empleado, COUNT(p)" +
"FROM Proyecto p GROUP BY p.empleado").getResultList();
Iterator res = result.iterator();
while (res.hasNext()) {
  Object[] tupla = (Object[]) res.next();
  Empleado emp = (Empleado) tupla[0];
  long count = ((Long) tupla[1]).longValue();
  ...
}
```

La siguiente consulta combina todos los elementos. Primero se filtrarían los resultados con la cláusula `WHERE`, después los resultados son agrupados y por último se comprueba la cláusula `HAVING`. Devuelve los empleados que participan en más de 5 proyectos creados entre dos fechas dadas.

```
SELECT p.empleado, COUNT(p)
FROM Proyecto p
WHERE p.fechaCreacion is BETWEEN :date1 and :date2
GROUP BY p.empleado
HAVING COUNT(p) > 5
```

10. Ejercicios sesión 5: Consultas

Vamos a completar el código de algunos DAO, añadiendo unas cuantas consultas. Comenzamos añadiendo los siguientes métodos a la clase `PaginaHtmlDAO`.

Método 1:

```
Collection<PaginaHtml> findAllPaginasHtml()
```

El método `findAllPaginasHtml()` devuelve una colección con todas las instancias `PaginaHtml` creadas.

Método 2:

```
Collection<PaginaHtml> findPaginasHtmlTag(String cadena)
```

El método `findPaginasHtmlTag(cadena)` devuelve una colección con todas las páginas que tienen asociadas una etiqueta que comienza por la `cadena`. En JPQL es posible hacer consultas sobre atributos de texto utilizando expresiones regulares. Por ejemplo, la siguiente expresión devolvería todas las revistas cuyo título comienza por 'J'.

```
SELECT x FROM Magazine x WHERE x.title LIKE 'J%'
```

Método 3:

```
Collection<PaginaHtml> findPaginasHtmlTagStrict(String cadena)
```

El método `findPaginasHtmlTagStrict(cadena)` devuelve una colección con todas las páginas que tienen asociadas una etiqueta cuyo texto coincide con la `cadena`.

Método 4:

```
Collection<PaginaHtml> findPaginasHtmlConMasTagsDe(int veces)
```

El método `findPaginasHtmlConMasTagsDe(veces)` devuelve una colección con aquellas páginas que tienen asociadas más etiquetas que las indicadas en el parámetro `veces`.

3. Añade al DAO `TagDAO` los siguientes métodos, implementando los que consideres necesarios como consultas JPQL:

Método 1:

```
Collection<Tag> findTagsComienzanPor(String cadena)
```

El método `findTagsComienzanPor(cadena)` devuelve una colección de aquellas etiquetas cuyo texto comienza por la cadena que pasamos como parámetro.

Método 2:

```
Tag findTagMasUsada()
```

El método `findTagMasUsada()` devuelve la etiqueta con más ocurrencias.

Método 3:

```
List<Tag> findTagsUsadasMasDe(int veces)
```

El método `findTagsUsadasMasDe(veces)` devuelve una lista ordenada por número de ocurrencias (de mayor a menor) con aquellas etiquetas que tienen más ocurrencias que el argumento `veces` que se pasa por parámetro.

4. Añade por último en el DAO `AutorDAO` las siguientes consultas:

Método 1:

```
Collection<Tag> findTagsAutor(Autor autor)
```

El método `findTagsAutor(autor)` devuelve una colección con todos los tags (distintos) usados por el autor que se pasa como parámetro.

Método 2:

```
Collection<Mensaje> findMensajeAutorTag(Autor autor, String cadena)
```

El método `findMensajeAutorTag(autor,tag)` devuelve una colección de los mensajes asociados al autor que están asociados también a alguna página con una etiqueta que comienza por `cadena`.

11. Transacciones

La gestión de transacciones es fundamental en cualquier aplicación que trabaja con de datos. Idealmente, una transacción define una *unidad de trabajo* que debe cumplir los criterios ACID: Atomicidad, Consistencia, Aislamiento (la "I" viene del inglés *Isolation*) y Durabilidad. La consistencia la debe proporcionar el programador, realizando operaciones que lleven los datos de un estado consistente a otro. La durabilidad la proporciona el hecho de que usemos un sistema de bases de datos. Las dos características restantes, la *atomicidad* y el *aislamiento* son las más interesantes, y las que vienen determinadas por el sistema de gestión de persistencia que usemos. Veámoslas con más detalle.

Una transacción debe ser atómica. Esto es, todas las operaciones de una transacción deben terminar con exito o la transacción debe abortar completamente, dejando el sistema en el mismo estado que antes de comenzar la transacción. En el primer caso se dice que la transacción ha realizado un commit y en el segundo que ha efectuado un rollback. Por esta propiedad, una transacción se debe tratar como una unidad de computación.

Para garantizar la atomicidad en JDBC se llama al método `setAutoCommit(false)` de la conexión para demarcar el comienzo de la transacción y a `commit()` o `rollback()` al final de la transacción para confirmar los cambios o deshacerlos. Veremos que JPA que utiliza una demarcación explícita de las transacciones, marcando su comienzo con una llamada a un método `begin()` y su final también con llamadas a `commit()` o `rollback()`. Siempre debemos tener muy presente que JPA (cuando no utilizamos un servidor de aplicaciones) se basará completamente en la implementación de JDBC para tratar con la base de datos.

Un transacción también debe ejecutarse de forma aislada. Esto es, no se deben exponer a otras transacciones concurrentes datos que todavía no se han consolidado con un commit. La concurrencia de acceso a los recursos afectados en una transacción hace muy complicado mantener esta propiedad. Veremos que JPA proporciona un enfoque moderno basado en *bloqueos optimistas* (*optimistic locking*) para tratar la concurrencia entre transacciones.

11.1. Atomicidad

JPA define la interfaz `EntityTransaction` para gestionar las transacciones. Esta interfaz intenta imitar el API de Java para gestión de transacciones distribuidas: JTA. Pero tengamos siempre en cuenta que para su implementación se utiliza el propio sistema de transacciones de la base de datos sobre la que trabaja JPA, utilizando los métodos de transacciones de la interfaz `Connection` de JDBC. Estas transacciones son locales (no distribuidas) y no se pueden anidar ni extender.

Para definir una transacción hay que obtener un `EntityTransaction` a partir del entity

manager. El método del entity manager que se utiliza para ello es `getTransaction()`. Una vez obtenida la transacción, podemos utilizar uno de los métodos de su interfaz:

```
public interface EntityTransaction {
   public void begin();
   public void commit();
   public void rollback();
   public void setRollbackOnly();
   public boolean getRollbackOnly();
   public boolean isActive();}
```

Sólo hay seis métodos en la interfaz `EntityTransaction`. El método `begin()` comienza una nueva transacción en el recurso. Si la transacción está activa, `isActive()` devolverá `true`. Si se intenta comenzar una nueva transacción cuando ya hay una activa se genera una excepción `IllegalStateException`. Una vez activa, la transacción puede finalizarse invocando a `commit()` o deshacerse invocando a `rollback()`. Ambas operaciones fallan con una `IllegalStateException` si no hay ninguna transacción activa. El método `setRollbackOnly()` marca una transacción para que su único final posible sea un `rollback()`.

Se lanzará una `PersistenceException` si ocurre un error durante el rollback y se lanzará una `RollbackException` (un subtipo de `PersistenceException`) si falla el commit. Tanto `PersistenceException` como `IllegalException` son excepciones de tipo `RuntimeException`.

El método `setRollbackOnly()` se utiliza para marcar la transacción actual como inválida y obligar a realizar un rollback cuando se ejecuta JPA con transacciones gestionadas por el contenedor de EJB (CMT: *Containter Managed Transactions*). En ese caso la aplicación no define explícitamente las transacciones, sino que es el propio componente EJB el que abre y cierra una transacción en cada método.

Hemos comentado que en JPA todas las excepciones son de tipo `RunTimeException`. Esto es debido a que son fatales y casi nunca se puede hacer nada para recuperarlas. En muchas ocasiones ni siquiera se capturan en el fragmento de código en el que se originan, sino en el único lugar de la aplicación en el que se capturan las excepciones de este tipo.

La forma habitual de definir las transacciones en JPA es la definida en el siguiente código:

```
EntityManager em = emf.createEntityManager();
EntityTransaction tx = em.getTransaction();
   try {
      tx.begin();
      // Operacion sobre entidad 1
      // Operacion sobre entidad 2
      tx.commit();
   } catch (RunTimeException ex) {
      tx.rollback();
   } finally {
      em.close();
}
```

Primero se obtiene la transacción y se llama al método `begin()`. Después se realizan todas las operaciones dentro de la transacción y se realiza un `commit()`. Todo ello se engloba en un bloque de `try/catch`. Si alguna operación falla lanza una excepción que se captura y se ejecuta el `rollback()`. En cualquier caso se cierra el entity manager.

Cuando se deshace una transacción en la base de datos todos los cambios realizados durante la transacción se deshacen también. La base de datos vuelve al estado previo al comienzo de la transacción. Sin embargo, el modelo de memoria de Java no es transaccional. No hay forma de obtener una instantánea de un objeto y revertir su estado a ese momento si algo va mal. Una de las cuestiones más complicadas de un mapeo entidad-relación es que mientras que podemos utilizar una semántica transaccional para decidir qué cambios se realizan en la base de datos, no podemos aplicar las mismas técnicas en el contexto de persistencia en el que viven las instancias de entidades.

Siempre que tenemos cambios que deben sincronizarse en una base de datos, estamos trabajado con un contexto de persistencia sincronizado con una transacción. En un momento dado durante la vida de la transacción, normalmente justo antes de que se realice un commit, esos cambios se traducirán en sentencias SQL y se enviarán a la base de datos.

Si la transacción hace un rollback pasarán entonces dos cosas. Lo primero es que la transacción en la base de datos será deshecha. Lo siguiente que sucederá será que el contexto de persistencia se limpiará (*clear*), desconectando todas las entidades que se gestionaban. Tendremos entonces un montón de entidades desconectadas de la base de datos con un estado no sincronizado con la base de datos.

11.2. Concurrencia y niveles de aislamiento

La gestión de la concurrencia en transacciones es un problema complejo. Por ejemplo, supongamos un sistema de reservas de vuelos. Sería fatal que se vendiera el mismo asiento de un vuelo a dos personas distintas por el hecho de que se han realizado dos accesos concurrentes al siguiente método:

```
public void reservaAsiento(Pasajero pasajero, int numAsiento, long
numVuelo) {
   EntityManager em = emf.createEntityManager();
   AsientoKey key = new AsientoKey(numAsiento, numVuelo);
   em.getTransaction().begin();
   Asiento asiento = em.find(Asiento.class, key);
   if (! asiento.getOcupado()) {
      asiento.setOcupado(true);
      asiento.setPasajero(pasajero);
      pasajero.setAsiento(asiento);
   }
   em.getTransaction().commit();
   em.close();
}
```

Si no se controlara el acceso concurrente de las transacciones a la tabla de asientos, podría

suceder que dos transacciones concurrentes accedieran al estado del asiento a la misma vez (antes de haberse realizado el UPDATE de su atributo `ocupado`), lo vieran libre y se lo asignaran a un pasajero y después a otro. Uno de los pasajeros se quedaría sin billete.

Se han propuesto múltiples soluciones para resolver el acceso concurrente a los datos. El estándar SQL define los llamados *niveles de aislamiento* (isolation levels) que tratan este problema. Los niveles más bajos solucionan los problemas más comunes y permiten al mismo tiempo que la aplicación responda sin generar bloqueos. El nivel más alto alto garantiza que todas las transacciones son serilizables, pero obliga a que se definan un número excesivo de bloqueos.

El estándar SQL define cuatro posibles problemas que pueden generarse cuando dos transacciones concurrentes realizan SELECTS y UPDATES en la base de datos:

- **Actualizaciones perdidas** (*lost updates*): este problema ocurre cuando dos transacciones hacen un UPDATE sobre el mismo dato y una de ellas aborta, perdiéndose los cambios de ambas transacciones. Un ejemplo: (1) un dato tiene un valor V0; (2) la transacción T1 comienza; (3) la transacción T2 comienza y actualiza el dato a V2; (4) la transacción T1 lo actualiza a V1; (5) la transacción T2 hace un commit, guardando V2; (6) por último, la transacción T1 se aborta realizando un rollback y devolviendo el dato a su estado inicial de V0.
- **Lecturas de datos sucios** (*dirty readings*): sucede cuando una transacción hace un SELECT y obtiene un dato modificado por un UPDATE de otra transacción que no ha hecho commit. El dato es *sucio* porque todavía no ha sido confirmado y puede cambiar. Por ejemplo: (1) un dato tiene un valor V0; (2) la transacción T1 lo actualiza a V1; (3) la transacción T2 lee el valor V1 del dato; (4) la transacción T1 hace un rollback, volviendo el dato al valor V0, y quedando sucio el dato contenido en la transacción T2.
- **Lecturas no repetibles** (*urepeatable read*): sucede cuando una transacción lee un registro dos veces y obtiene valores distintos por haber sido modificado por otro UPDATE confirmado. Por ejemplo: (1) una transacción T1 lee un dato; (2) comienza otra transacción T2 que lo actualiza; (3) T2 hace un commit; (4) T1 vuelve a leer el dato y obtiene un valor distinto al primero.
- **Lecturas fantasmas** (*phantom read*): una transacción ejecuta dos consultas y en la segunda aparecen resultados que no son compatibles con la primera (registros que se han borrado o que han aparecido). El mismo ejemplo anterior, cambiando la lectura de T1 por una consulta.

Las bases de datos SQL definen cuatro posibles niveles de aislamiento que corresponden a modos de funcionamiento de la base de datos en el que se garantiza que no sucede alguno de los problemas anteriores. Tradicionalmente se han utilizado bloqueos para asegurar estos niveles. Vamos a describirlos, indicando la estrategia de bloqueo utilizada en cada caso. De menor a mayor nivel de seguridad son los siguientes:

- **READ_UNCOMMITTED**: Es el método que aísla menos las transacciones, ya que permite lecturas de escrituras de las que no se han hecho *commit*. Aún así, garantiza

que no ocurren actualizaciones perdidas. Si la base de datos utiliza bloqueos, un UPDATE de un dato lo bloqueara para escritura. De esta forma, otras transacciones podrán leerlo (y se podrán producir cualquiera de los otros problemas) pero no escribirlo. Cuando se realiza un commit se libera el bloqueo.

- **READ_COMMITTED**: garantiza que no existen las lecturas sucias ya que sólo permite leer datos que han sido confirmados. Utilizando bloqueos, se podría resolver haciendo que un UPDATE de un dato lo bloqueara para lectura y escritura. Cualquier otra transacción que intente leer el dato quedará en espera hasta que se confirme la transacción.
- **REPEATABLE_READ**: garantiza que dentro de la misma transacción no cambia el valor de un dato leído. Para asegurar este nivel utilizando bloqueos, un SELECT sobre un dato lo bloquea frente a otras actualizaciones. Muchas bases de datos no utilizan bloqueos para resolver este problema, sino que guardan múltiples versiones de un dato y muestran a una transacción sólo el dato previamente leído por ella.
- **SERIALIZABLE**: garantiza que no se producen lecturas fantasmas. Es el nivel máximo de seguridad y para garantizarlo utilizando bloqueos se deben bloquear tablas enteras, no solo registros.

Como hemos dicho, los niveles de aislamiento se han garantizado tradicionalmente utilizando distintos tipos de bloqueos en los accesos a los datos. Recientemente, sin embargo, bases de datos como Oracle, Postgress o MySQL con InnoDB proporcionan alternativas distintas a los bloqueos para garantizar estos niveles. En concreto, permiten usar el llamado *Multiversion Concurrency Control* (MVCC) en el que se utilizan versiones de los datos para garantizar las condiciones de aislamiento. Con esta estrategia, en el nivel READ_COMMITTED cuando una transacción lee de un dato que no ha sido confirmado se lee la versión anterior del dato (la última que ha sido confirmada). Incluso en el modo REPEATABLE_READ el primer SELECT y el segundo obtendrían el mismo valor anterior a la modificación a la modificación de la transacción 2. En general, con MVCC las operaciones de lectura no bloquean, sino que obtienen la última versión confirmada del dato.

¿Cuál es el nivel de aislamiento recomendable para una aplicación? El nivel READ_UNCOMMITTED es demasiado permisivo, ya que permite que una transacción lea datos que no han sido confirmados por otra transacción. El SERIALIZABLE es demasiado restrictivo y hace que la aplicación no escale correctamente. Se producirían demasiados bloqueos para resolver un problema que no sucede demasiado a menudo (lecturas fantasmas). Esto nos deja con los niveles READ_COMMITTED y REPEATABLE_READ. La mayoría de aplicaciones en producción usan alguno de estos niveles.

Una solución muy frecuente es utilizar el nivel READ_COMMITTED como nivel por defecto y realizar bloqueos puntuales en aquellas ocasiones peligrosas en las que puede suceder un problema como el de la butaca de cine. En SQL se puede utilizar la instrucción

```
SELECT ... FOR UPDATE
```

para bloquear un determinado registro explícitamente para lectura y escritura hasta realizar un commit.

En resumen, cuando estemos trabajando con una base de datos nos tenemos que plantear las siguientes preguntas (las respuestas varían dependiendo de la base de datos):

- ¿Cuál es el nivel de aislamiento por defecto de la base de datos?
- ¿Qué niveles de aislamiento soporta?
- ¿Cómo se implementa cada nivel de aislamiento, utilizando bloqueos o múltiples versiones (MVCC)?

11.3. Gestión concurrencia con JPA

Los niveles de aislamiento vistos en el apartado anterior se gestionan por la propia base de datos y afectan a JPA en el momento en que el entity manager hace un flush y se generan las sentencias SELECT y UPDATE. En ese momento el sistema de base de datos toma el control de la concurrencia utilizando el nivel de seguridad definido por defecto.

En la versión estándar de JPA no es posible definir el nivel de aislamiento de la base de datos subyacente. Esta opción es dependiente de la implementación de JPA. En la versión de Hibernate, se puede definir en el fichero `peristence.xml`, con la propiedad `hibernate.connection.isolation`. Los posibles valores son:

- 1: READ_UNCOMMITTED
- 2: READ_COMMITTED
- 4: REPEATABLE_READ
- 8: SERIALIZABLE

JPA utiliza por defecto un sistema optimista de gestión de la concurrencia. Para que funcione correctamente necesita que el nivel de aislamiento de la base de datos sea READ_COMMITTED. Recordemos que este nivel no protege del problema de las lecturas no repetidas (una transacción lee un dato y otra transacción lo modifica antes de que la primera haga un commit).

El control optimista de la concurrencia asume que no suceden conflictos y no se realizan bloqueos sobre los datos. Sin embargo, si al final de la transacción se ha producido un error se lanza una excepción.

En un control optimista de concurrencia todos los objetos tienen un atributo adicional que guarda su número de versión (una columna adicional en la tabla). Las lecturas leen el número de versión y las escrituras lo incrementan. Cuando una transacción intenta escribir en una versión de un objeto que no corresponde con la que había leído previamente se aborta la actualización y se genera una excepción.

Por ejemplo, supongamos que una transacción T1 realiza una lectura sobre un objeto. Se obtiene automáticamente su número de versión. Supongamos que otra transacción T2 modifica el objeto y realiza un commit. Automáticamente se incrementa su número de

versión. Si ahora la transacción T1 intenta modificar el objeto se comprobará que su número de versión es mayor que el que tiene y se generará una excepción.

En este caso el usuario de la aplicación que esté ejecutando la transacción T1 obtendrá un mensaje de error indicando que alguien ha modificado los datos y que no es posible confirmar la operación. Lo deberá intentar de nuevo.

Para que JPA pueda trabajar con versiones es necesario que los objetos tengan un atributo marcado con la anotación `@Version`. Este atributo puede ser del tipo `int`, `Integer`, `short`, `Short`, `long`, `Long` y `java.sql.Timestamp`.

```
@Entity
public class Autor {
   @Id
   private String nombre;
   @Version
   private int version;
   private String correo;
   ...
}
```

Aunque el estándar no lo permite, en Hibernate es posible definir un comportamiento optimista en entidades que no definen una columna de versión. Para ello basta con definir la entidad de la siguiente forma:

```
@Entity
@org.hibernate.annotations.Entity (
   optimisticLock = OptimisticLockType.ALL,
   dynamicUpdate = true
)
public class Autor {
   @Id
   private String nombre;
   private String correo;
   ...
}
```

De forma complementaria al comportamiento optimista, también es posible en JPA definir bloqueos explícitos sobre objetos. Para ello hay que utilizar el método `lock(objeto,LockModeType)` del entity manager. Este método bloquea un objeto (registro) para lectura y escritura. Hay que pasarle como parámetro el objeto sobre el que se realiza el bloqueo y el tipo de bloqueo. El tipo de bloqueo puede ser `LockModeType.READ` y `LockModeType.WRITE`. La diferencia entre ambos es que el segundo incrementa automáticamente el número de versión del objeto, independientemente de que se haga después una actualización o no. Cualquier intento de escritura de una versión anterior del objeto generará una excepción.

Esta característica se ha estandarizado en la versión 2.0 de JPA aprobada recientemente. En versiones anteriores su definición dependía del gestor de JPA. En el caso de Hibernate para que funcione correctamente hay que configurar el nivel de aislamiento de la base de datos en nivel 4 (REPETABLE_READ):

```
><property name="hibernate.connection.isolation"
    value="4"/>
```

Por ejemplo, si se quisiera evitar el problema del asiento del vuelo bloqueando explícitamente el registro, habría que escribir el siguiente código:

```
public void reservaAsiento(Pasajero pasajero, int numAsiento, long
numVuelo) {
   EntityManager em = emf.createEntityManager();
   AsientoKey key = new AsientoKey(numAsiento, numVuelo);
   em.getTransaction().begin();
   Asiento asiento = em.find(Asiento.class, key);
   em.lock(asiento,LockType.WRITE);
   if (! asiento.getOcupado()) {
      asiento.setOcupado(true);
      asiento.setPasajero(pasajero);
      pasajero.setAsiento(asiento);
   }
   em.getTransaction().commit();
   em.close();
}
```

Una llamada a `lock` genera inmediatamente una instrucción SQL `SELECT ... FOR UPDATE`, sin esperar a que se realice un flush del contexto de persistencia. Esta instrucción SQL hace que el gestor de base de datos realice un bloqueo del registro.

12. Ejercicios sesión 6: Transacciones

Vamos a probar las distintas características de atomicidad y aislamiento de las transacciones JPA. Utilizaremos la entidad `Autor` creada en la sesión 1 y algunos programas de prueba que escribiremos en esta sesión. Puedes guardar los programas de prueba en el mismo proyecto `jpa-mensajes`.

12.1. Atomicidad

En este primer ejercicio vamos a probar que las transacciones funcionan correctamente. Generaremos un error y probaremos el *rollback* de la transacción.

1. Introduce en el código la gestión de transacciones vista en los apuntes de teoría, de forma que si hay algún error, se captura y se realice un rollback();

2. Produce algún error justo después de haber actualizado la relación entre el autor y el mensaje (por ejemplo, podemos lanzar una excepción haciendo un `throw new RuntimeErrorException(new Error())`). Ejecuta el programa creando un nuevo autor, comprueba en las sentencias SQL que se realizan los inserts en la base de datos. Comprueba que se realizado el rollback y el autor y el mensaje han dejado de estar en la BD. Es posible que no funcione el rollback porque las tablas se hayan creado de tipo `MyISAM`. Utiliza el administrador de MySQL para actualizar el tipo de las tablas a `InnoDB` (el tipo de tabla de MySQL que soporta las transacciones).

12.2. Programas de prueba

Utilizaremos en los ejercicios un conjunto de programas de prueba que nos van a permitir simular los distintos problemas y niveles de aislamiento:

El programa **Escritor** busca el autor 'Optimista', cambia su correo a un determinado valor ('AAA'), se queda esperando a que el usuario pulse 'INTRO' y después es cuando hace el commit de la transacción:

```
public class Escritor {
   public static void main(String[] args) {

      EntityManagerFactory emf = Persistence
            .createEntityManagerFactory("simplejpa");
      EntityManager em = emf.createEntityManager();

      pulsaIntro("Pulsa INTRO para Begin + Find");
      em.getTransaction().begin();
      Autor autor = em.find(Autor.class, "Optimista");
      System.out.println("Valor:" + autor.getCorreo());

      pulsaIntro("Pulsa INTRO para Set AAA");
      autor.setCorreo("AAA");
```

```
      em.flush();
      System.out.println("Nuevo valor: " + autor.getCorreo());

      pulsaIntro("Pulsa INTRO para COMMIT");
      em.getTransaction().commit();

      System.out.println("Transacción terminada. Valor: " +
autor.getCorreo());
      em.close();
      emf.close();
   }
```

El programa **LectorEscritor** busca el mismo autor que el programa anterior, lee dos veces el valor de su correo (esperando que se pulse 'INTRO' entre una y otra lectura) y cambia el valor del correo a 'BBB':

```
public class LectorEscritor {
   public static void main(String[] args) {

      EntityManagerFactory emf = Persistence
            .createEntityManagerFactory("simplejpa");
      EntityManager em = emf.createEntityManager();

      pulsaIntro("Pulsa INTRO para Begin + Find");
      em.getTransaction().begin();
      Autor autor = em.find(Autor.class, "Optimista");
      String c = autor.getCorreo();
      System.out.println("Valor:" + c);

      pulsaIntro("Pulsa INTRO para actualizar");
      autor.setCorreo(c + "BBB");
      em.getTransaction().commit();

      System.out.println("Transacción terminada. Valor: " +
autor.getCorreo());
      em.close();
      emf.close();
   }
```

Función **private static void pulsarIntro(String msg)** que se utiliza en todos ellos:

```
private static void pulsaIntro(String msg) {
   try {
      BufferedReader in = new BufferedReader(
            new InputStreamReader(System.in));
      System.out.println(msg);
      in.readLine();
   } catch (IOException e) {
   }
}
```

12.3. Gestión optimista de la concurrencia

Vamos a probar cómo gestiona JPA la concurrencia de forma optimista. Recordemos que JPA asume que la BD está funcionando con un nivel de aislamiento

READ_COMMITED. En Hibernate podemos definir el nivel de aislamiento SQL de la siguiente forma:

```
<property name="hibernate.connection.isolation"
          value="2"/>
```

El valor puede ser:

- 1: READ_UNCOMMITTED
- 2: READ_COMMITTED (valor por defecto)
- 4: REPEATABLE_READ
- 8: SERIALIZABLE

La forma de implementar estos niveles de aislamiento es dependiente de la BD. En el caso de MySQL se utiliza la estrategia MVCC (MultiVersion Concurrency Control) para asegurar el nivel READ_COMMITED.

Para que funcione la gestión optimista de concurrencia en JPA debemos incluir en las entidades un atributo `Integer` con la anotación `@Version`:

```
@Entity
public class Autor {
   @Id
   private String nombre;
   @Version
   private Integer version;
   private String correo;
   ...
}
```

Para comprobarlo haz lo siguiente y escribe lo que sucede en el fichero `respuestas06.txt`:

Con el programa `HolaMundo` creamos un autor llamado 'Optimista'.

- Lanzamos el prgrama QueryBrowser y nos aseguramos de que exista el autor 'Optimista'. Su correo debe ser 'optimista@ua.es'.
- Lanzamos el programa `Escritor` y hacemos que escriba 'AAA' en el autor. No pulsamos 'INTRO', para evitar que se realice un commit. ¿Se ha ejecutado el INSERT en la BD?¿Se ha cambiado el valor en el QueryBrowser?
- Lanzamos el programa `LectorEscritor` y comprobamos qué valor lee.
- Hacemos commit con `Escritor`.
- Continuamos con `LectorEscritor` para que modifique el valor del correo. ¿Se modifica en la BD? ¿Es correcto?

Ahora vuelve a ejecutar los pasos anteriores, pero lanzando primero `LectorEscritor` y cambiando el valor con `Escritor` antes que `LectorEscritor` haya escrito el nuevo valor. De esta forma, intentará sobreescribir un valor en un campo que ha sido modificado por otra transacción concurrente. ¿Qué sucede?

12.4. Bloqueos JPA (*)

Por último, comprobamos que añadiendo un bloque explícito de JPA en ambos programas evitamos que el Escritor realice la primera lectura sobre el valor:

```
pulsaIntro("Pulsa INTRO para Begin + Find");
em.getTransaction().begin();
Autor autor = em.find(Autor.class, "Optimista");
em.lock(autor, LockModeType.READ);
```

¿Qué sucede en este caso? Explícalo en el fichero `respuestas06.txt`.

13. De JPA a Hibernate

13.1. De JPA a Hibernate

Una vez que hemos visto en profundidad el enfoque de JPA para solucionar el problema de la persistencia en Java, vamos a dedicar un par de sesiones a explorar cómo se puede hacer todo esto utilizando Hibernate Core, sin las extensiones para implementar JPA. En la actualidad hay muchos proyectos en funcionamiento que fueron implementados en Hibernate y que no se han portado a JPA. Si un proyecto Hibernate funciona correctamente y no hay que añadir demasiadas características nuevas, no es necesario moverse a JPA.

También hay que decir que JPA, al ser un estándar, es una versión relativamente simplificada de Hibernate. Hibernate es más potente y flexible. En palabras de Gavin King, el creador y líder del proyecto Hibernate, la especificación de JPA cubre el 95% de las necesidades cubiertas con Hibernate. Hay un 5% de funcionalidades que ofrece Hibernate y que no es posible conseguir en JPA. Sin embargo, estas funcionalidades van más allá del tiempo que tenemos para comentar Hibernate. En este capítulo y el siguiente resaltaremos principalmente las equivalencias entre ambos frameworks, y explicaremos cómo hacer en Hibernate lo que ya sabemos hacer en JPA.

Por último, una ventaja de Hibernate es la buena integración con frameworks como Spring y en otros proyectos Open Source. Esto es debido, básicamente, a la madurez del framework. Es de suponer que con el tiempo aumentará la integración de JPA en Spring y otros frameworks populares, así como en entornos de desarrollo.

Repaso: hola mundo en JPA y Hola mundo en Hibernate.

Recordemos el programa `HolaMundo` en JPA:

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

      em.getTransaction().begin();
         // Busco el autor y lo creo si no existe
      autor = em.find(Autor.class, autorStr);
      if (autor == null) {
         autor = new Autor();
         autor.setNombre(autorStr);
         autor.setCorreo(autorStr + "@ua.es");
         em.persist(autor);
      }
      // Miro si el autor ya un mensaje
      if (autor.getMensaje() == null) {
         // Nuevo mensaje
```

```
         mensaje = new Mensaje();
         mensaje.setAutor(autor);
         mensaje.setTexto(mensStr);
         em.persist(mensaje);
      } else {
         // Modifico el mensaje que ya existe
         Long mensId = autor.getMensaje().getId();
         mensaje = em.find(Mensaje.class, mensId);
         mensaje.setTexto(mensStr);
      }
         em.getTransaction().commit();
      em.close();
      emf.close();
   }
}
```

Y esta es la versión del mismo programa en Hibernate:

```
import org.hibernate.*;

import entity.Autor;
import entity.Mensaje;

public class HolaMundo {

   public static void main(String[] args) {
      Autor autor;
      Mensaje mensaje;
      String autorStr, mensStr;

      SessionFactory sessionFactory =
         new Configuration().configure().buildSessionFactory();
      Session session = sessionFactory.getCurrentSession();
      // Leo el mensaje y el autor
      try {
           BufferedReader in = new BufferedReader(
              new InputStreamReader(System.in));
         System.out.print("Nombre: ");
         autorStr = in.readLine();
         System.out.print("Mensaje: ");
         mensStr = in.readLine();
      } catch (IOException e) {
         autorStr = "Error";
         mensStr = "Error";
      }

      session.beginTransaction();

      // Busco el autor y lo creo si no existe
      autor = (Autor) session.get(Autor.class, autorStr);
      if (autor == null) {
         autor = new Autor();
         autor.setNombre(autorStr);
         autor.setCorreo(autorStr + "@ua.es");
         session.save(autor);
      }

      // Creo el mensaje
```

```
      mensaje = new Mensaje();
      mensaje.setAutor(autor);
      mensaje.setTexto(mensStr);
      session.save(mensaje);

      // Lo añado al autor
      Collection<Mensaje> mensajes = autor.getMensajes();
      mensajes.add(mensaje);
      session.getTransaction().commit();

      // Imprimimos todos los mensajes del autor
      System.out.println(autor.getNombre() + " ha escrito " +
mensajes.size() + " mensajes:");
      Iterator<Mensaje> it = mensajes.iterator();
      while (it.hasNext()) {
         Mensaje mens = it.next();
         System.out.println(mens.getTexto());
      }
      sessionFactory.close();
   }
}
```

Como se puede comprobar, el funcionamiento de Hibernate es muy similar al de JPA. Existe una factoría de la que se obtiene una sesión de Hibernate (objeto `Session` equivalente al `EntityManager` de JPA). Con la sesión podemos abrir y cerrar la transacción, obtener entidades y guardarlas en la base de datos.

En lo que respecta a las entidades, son también similares a las de JPA. Se trata de POJOs en forma de Java Beans (objetos Java con atributos accedidos por los *setters* y *getters*). Se recuperan de la base de datos llamando al método `get` de la sesión (en lugar del `find` sobre el *entityManager*). Se graban en la base de datos utilizando el método `save` de la sesión (en lugar del `persist` del entityManager).

Las entidades siguen funcionado como objetos Java que están relacionados y que pueden recorrerse. El código Java propiamente dicho es el mismo en la versión JPA y en la versión Hibernate.

En cuanto a las diferencias más importantes, veremos más adelante que la definición de entidades en Hibernate es mucho más complicada que en JPA, ya que Hibernate define el mapeo de las entidades a tablas mediante ficheros XML.

La configuración de la conexión de Hibernate con la base de datos se define también en en un fichero XML similar al `persistence.xml` de JPA.

Todos estos detalles los veremos más adelante. En esta primera sesión explicaremos cómo Hibernate define el mapeo entidad-relación. La segunda sesión la utilizaremos para detallar el uso de las sesiones y del funcionamiento dinámico de Hibernate, así como para repasar algunos ejemplos de queries.

13.2. Configuración de Hibernate

Para poder utilizar Hibernate en nuestra aplicación hay que instalar un conjunto de librerías en el CLASSPATH y configurar sus características.

Las librerías necesarias son las propias del proyecto *Hibernate Core* más un conjunto de librerías de otros proyectos open source que utiliza Hibernate. Todas están disponibles en el paquete *Hibernate Core* de la web de Hibernate. La versión más reciente es la 3.3.1.GA, lanzada en septiembre de 2008. En la distribución se encuentran las siguientes librerías necesarias:

- antlr-2.7.5.jar
- commons-collections-3.1.jar
- dom4j-1.6.1.jar
- hibernate3.jar
- hibernate-testing.jar
- javassist-3.4.GA.jar
- jta-1.1.jar
- slf4j-api-1.5.2.jar
- slf4j-simple-1.5.2.jar
- mysql-connector-java-5.1.6-bin.jar

La librería `slf4j-simple-1.5.2.jar` no está incluida en la distribución original y hay que añadirla si se quiere mostrar los mensajes de log por la salida estándar.

El fichero `hibernate.cfg.xml` define la configuración de Hibernate. Debe llamarse de esa forma y encontrarse en la raíz del *classpath*.

```
<?xml version='1.0' encoding='utf-8'?>
  <!DOCTYPE hibernate-configuration PUBLIC
   "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
  <session-factory>
    <property
name="connection.url">jdbc:mysql://localhost/jpa</property>
    <property name="connection.username">root</property>
    <property name="connection.password">root</property>
    <property name="connection.driver_class">com.mysql.jdbc.Driver
           </property>
    <property name="dialect">org.hibernate.dialect.MySQLDialect
           </property>
    <property
name="current_session_context_class">thread</property>
    <property name="transaction.factory_class">
org.hibernate.transaction.JDBCTransactionFactory</property>
    <property name="cache.provider_class">
           org.hibernate.cache.NoCacheProvider</property>
    <property name="hibernate.show_sql">true</property>
    <property name="hbm2ddl.auto">update</property>
    <mapping resource="mappings.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```

El elemento `mapping` define la localización en el *classpath* de los ficheros de mapeo. Se pueden definir tantos ficheros de mapeo como se desee. Las dos opciones más utilizadas son crear un único fichero de mapeo en el que se definen todos los mapeos de todas las clases o separar en tantos ficheros de mapeo como clases se definan en la aplicación. En este último caso se suele nombrar los ficheros de mapeo con el nombre de la clase y la extensión `.hbm.xml` (por ejemplo: `Autor.hbm.xml`).

13.3. Mapeo de entidades

El concepto de *entidad* en Hibernate es el mismo que en JPA: una entidad existe independientemente de las referencias que otros objetos mantienen hacia ella. Esto contrasta con el modelo usual de Java en el que los objetos no referenciados se eliminan con la recolección de basura. Las entidades deben ser borradas y grabadas explícitamente (excepto en el caso de que sea un borrado o un grabado en cascada desde una entidad padre a su hijo).

En Hibernate el código Java de las entidades es el mismo que el de JPA. Son clases Java con *setters* y *getters*, un identificador y un constructor vacío.

Las clases deben cumplir cuatro condiciones:

- La clase debe tener un constructor sin argumento.
- Un atributo debe hacer de identificador. Se recomienda que el este identificador sea de un tipo *nullable* (esto es, no primitivo).
- Clases no finales. Todos los métodos deben ser públicos y no finales, para que Hibernate pueda sobrecargarlos e implementar los *proxies* que permiten acceder de forma *lazy* a otras entidades.
- Declarar métodos de acceso para los atributos persistentes. Los métodos pueden llamarse `getFoo`, `isFoo` y `setFoo` y no es necesario que sean públicos.

Por ejemplo, la clase persistente `Autor` es la siguiente:

```
public class Autor {
   private String nombre;
   private String correo;
   private Integer edad;
   private Set<Mensaje> mensajes = new HashSet<Mensaje>();

   public Autor() {}
   public String getNombre() {return nombre;}
   public void setNombre(String nombre) {this.nombre = nombre;}
   public String getCorreo() {return correo;}
   public void setCorreo(String correo) {this.correo = correo;}
   public Set<Mensaje> getMensajes() {return mensajes;}
   public void setMensajes(Set<Mensaje> mensajes) {
      this.mensajes = mensajes;}
}
```

```
package entity;
```

```
import java.util.Date;

public class Mensaje {
   private long id;
   private String texto;
   private Date creado;
   Autor autor;

   public Mensaje() {}
   public long getId() {return id;}
   public void setId(long id) {this.id = id;}
   public String getTexto() {return texto;}
   public void setTexto(String texto) {this.texto = texto;}
   public Autor getAutor() {return autor;}
   public void setAutor(Autor autor) {this.autor = autor;}
   public Date getCreado() {return creado;}
   public void setCreado(Date creado) {this.creado = creado;}
}
```

Vemos que se cumplen todas las condiciones. La clase tiene un constructor vacío. El atributo `nombre` hace de identificador. Los métodos no son finales. Y los métodos de los atributos que van a ser persistentes (todos) son del tipo `get` y `set`.

El fichero de mapeo es

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
   <class name="entity.Autor" table="Autor">
      <id name="nombre" type="string" column="nombre"/>
         <property name="correo" type="string" column="correo"/>
         <property name="edad" type="integer" column="edad"/>
         <set name="mensajes" inverse="true">
            <key column="autor_nombre"/>
               <one-to-many class="entity.Mensaje"/>
         </set>
   </class>

   <class name="entity.Mensaje" table="Mensaje">
      <id name="id" column="id">
         <generator class="native"/>
      </id>
      <property name="texto" type="string" column="texto"
not-null="true"/>
      <property name="creado" type="date" column="creado"/>
      <many-to-one name="autor"
         column="autor_nombre"
         class="entity.Autor"
         not-null="true"/>
    </class>
</hibernate-mapping>
```

Más adelante explicaremos con más detalle el mapeo de las relaciones entre entidades.

13.3.1. Tipos de valores

Una entidad Java puede estar compuesta de valores primitivos, componentes o referencias a otras entidades (en forma de relaciones).

En la aplicación Java, todos estos elementos son tipos y objetos Java. El trabajo de Hibernate es mapearlos a tipos de la base de datos relacional. Para ello, se definen los siguientes mapeos básicos de Hibernate:

- `integer`,`long`,`short`,`float`,`double`,`character`,`byte`,`boolean`,`yes_no`,`true_false`: estos mapeos transforman los tipos básicos de Java (y sus clases wrappers) en los tipos SQL apropiados (dependientes del vendedor). Los mapeos `boolean`, `yes_no` y `true_false` son distintos nombres para booleanos.
- `string`: mapeo de `java.lang.String` a `VARCHAR` (o `VARCHAR2` de Oracle).
- `date`,`time`,`timestamp`: mapeo de la clase `java.util.Date` y sus subclases a los tipos SQL `DATE`,`TIME` y `TIMESTAMP` (o equivalentes).
- `calendar`,`calendar_date`: mapeo de `java.util.Calendar` a los tipos SQL `TIMESTAMP` y `DATE` (o equivalentes).
- `locale`,`timezone`,`currency`: mapeo de `java.util.Locale`,`java.util.Timezone` y `java.util.Currency` a `VARCHAR` (o `VARCHAR2` de Oracle). Las instancias de `Locale` y `Currency` se mapean a sus códigos ISO. Las instancias de `TimeZone` se mapean a su ID.
- `binary`: mapeo de array de bytes al tipo binario SQL apropiado.
- `text`: mapeo de cadenas largas de Java al tipo SQL `TEXT` o `CLOB`.
- `serializable`: mapeo de un tipo Java serializable a un tipo binario SQL.

13.3.2. Métodos equals y hashMap

Una característica importante de las clases persistentes es que debemos definir los métodos `equals` y `hashCode` para que funcione correctamente la comprobación de igualdad entre dos entidades y el contrato con la clase `Set` en las entidades desconcectadas.

Por defecto, Java define la igualdad `equals` entre dos objetos como igualdad de referencia. Dos variables son iguales en términos `equals` si apuntan al mismo objeto. Sin embargo, cuando estamos tratando con objetos que contienen datos en sus atributos, la mayoría de las veces nos interesa interpretar como iguales a objetos que tienen el mismo contenido.

Cuando los objetos se encuentran en la misma sesión de Hibernate y están gestionados, las instancias que se refieren a un mismo registro no se duplican, y la comparación con el == es correcta. Por ejemplo:

```
Autor autor1 = (Autor) session.get(Autor.class, autorStr);
Autor autor2 = (Autor) session.get(Autor.class, autorStr);
if (autor1 == autor2) {
```

```
   System.out.println("Iguales en referencia");
}
if (autor1.equals(autor2) {
   System.out.println("Iguales en contenido");
}
```

Sin embargo, si en algún momento vamos a desconectar las entidades es muy recomendable sobrecargar `equals` para que se realice la comparación mediante el contenido de los objetos. Por ejemplo:

```
Collection<Mensaje> mensajes = Collection<Mensaje>
autorDAO.allMensajesAutor(autor);
Mensaje mejorMensaje = mensajeDAO.findMejorMensaje();
if (mensajes.contains(mejorMensaje)) {
    autor.setMejor(true);
    ...
}
```

La solución recomendada por Hibernate es implementar `equals` en terminos de una *clave de negocio* (*business key*) que identifica de forma unívoca el objeto. Por ejemplo, en el caso del `Autor` sería el `nombre` (se debería llamar `login`), un identificador único. En el caso en que el identificador sea autogenerado por Hibernate, deberíamos obtener otro propio del objeto, ya que el identificador autogenerado sólo se crea en el momento de hacer persistente el objeto. ¿Qué sucedería si queremos comprar un objeto antes de insertarlo en un registro de la tabla? Por ello es necesario definir el método `equals` de forma que se utilicen sólo atributos del objeto.

Ejemplo:

```
public class Cat {
    ...
    public boolean equals(Object other) {
       if (this == other) return true;
       if ( !(other instanceof Cat) ) return false;
       final Cat cat = (Cat) other;
       if ( !cat.getId().equals( getId() ) ) return false;
       if ( !cat.getMother().equals( getMother() ) ) return false;
                return true;
    }

    public int hashCode() {
       int result;
       result = getMother().hashCode();
       result = 29 * result + getLitterId();
       return result;
    }
}
```

La única condición para la implementación de `hashCode` es que dos objetos deben devolver el mismo *hashCode* cuando `equals()` devuelve `true`.

13.3.3. Componentes

En una entidad pueden existir atributos que son de clases definidas por el programador, pero que no se mapean en entidades. Queremos que estos atributos se mapeen directamente en columnas de la tabla. En JPA teníamos la anotación `@embedded`, en Hibernate tenemos el elemento `component`.

Veamos el siguiente ejemplo. Definimos una clase `Persona` en la que uno de los atributos de tipo `Nombre`. A su vez, la clase `Nombre` está formada por tres atributos de tipo `String`. No queremos que los objetos de tipo `Nombre` sean entidades (no queremos que un nombre pueda estar relacionado con más de una persona), sino que representen valores primitivos que se guardan en la base de datos, como un conjunto de columnas.

```
public class Persona {
   private String key;
   private java.util.Date cumpleanyos;
   private Nombre nombre;

   public Persona() {}

   private void setKey(String key) {
      this.key=key;
   }
   public String getKey() {
      return key;
   }
   public java.util.Date getCumpleanyos() {
      return cumpleanyos;
   }
   public void setCumpleanyosy(java.util.Date cumpleanyos) {
      this.cumpleanyos = cumpleanyos;
   }
   public Nombre getNombre() {
      return nombre;
   }
   public void setName(Nombre nombre) {
      this.nombre = nombre;
   }
}
```

```
public class Nombre {
   String nombre;
   String primerApellido;
   String segundoApellido;

   public String getPrimerApellido() {
      return primerApellido;
   }
   void setPrimerApellido(String apellido) {
      this.primerApellido = apellido;
   }
   public String getSegundoApellido() {
      return segundoApellido;
   }
   void setLast(String segundoApellido) {
      this.segundoApellido = segundoApellido;
   }
   public String getNombre() {
```

```
      return nombre;
   }
   void setInitial(String nombre) {
      this.nombre = nombre;
   }
}
```

Para guardar los atributos de los `Nombre` embebidos en la tabla `Persona`, Hibernate utiliza el elemento `component` en la descripción del mapeo.

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
   <class name="entity.Persona" table="Persona">
   <id name="Key" column="pid" type="string">
      <generator class="uuid"/>
   </id>
   <property name="cumpleanyos" type="date"/>
   <component name="Nombre" class="entity.Nombre">
      <property name="nombre"/>
      <property name="primerApellido"/>
      <property name="segundoApellido"/>
   </component>
   </class>
</hibernate-mapping>
```

De esta forma, se la tabla `Persona` se define con los campos correspondientes a los atributos de la clase `Nombre`. Cuando se lee un registro de la tabla, se crea un objeto de clase `Nombre` y se guarda una referencia a él en el nuevo objeto de tipo `Persona`.

13.4. Mapeo de relaciones

Vamos a presentar dos ejemplos para comprobar cómo se realiza el mapeo de relaciones en Hibernate: el ejemplo clásico de relación uno-a-muchos y la relación muchos-a-muchos.

Básicamente existen dos formas de mapear una relación entre dos entidades. La primera es utilizando una clave ajena en una de las tablas que referencia a la clave primaria de la otra tabla. La segunda es utilizando una *tabla join* con claves ajenas a las claves primarias de ambas tablas. Dependiendo del tipo de relación definida se usa una estrategia u otra. Por ejemplo, para definir una relación uno-a-muchos bidireccional se usa una clave ajena, mientras que una relación muchos-a-muchos bidireccional se implementa con una tabla join. Para conocer todas las posibles combinaciones y formas de construir relaciones, es aconsejable consultar el manual de referencia de Hibernate.

13.4.1. Relación uno-a-muchos

Veamos por el caso más sencillo, una relación uno-a-muchos bidireccional de tipo

padre-hijo. Es la relación que definimos en el ejemplo entre un autor y los mensajes que ha escrito. En una relación padre-hijo la entidad padre tiene una referencia a una colección de entidades que, conceptualmente, forman parte de él. Por ejemplo, es la relación que se podría definir entre un producto y sus partes o entre una factura y sus artículos. Las relaciones de este tipo se definen en cascada y si se borra o graba el padre también se deben borrar los hijos asociados.

Vamos a utilizar el ejemplo de la relación entre `Autor` y `Mensaje`. En la entidad `Autor` debemos definir un atributo que sea una colección de objetos `Mensaje`. Esta colección puede ser cualquiera de las interfaces del framework de colecciones Java: `Set`, `Collection`, `List`, `Map` o `Map`. Escogeremos una u otra dependiendo del tipo de colección Java que necesitemos utilizar en la aplicación. Hibernate puede mapear cualquiera de estos tipos. Si queremos que la relación sea bidireccional en `Mesaje` debemos definir un atributo en el que se guardará el `Autor` con el que cada mensaje está relacionado.

El siguiente fichero de mapeo muestra cómo se define la relación:

```
<hibernate-mapping>
   <class name="Autor">
      <id name="autorId">
            <generator class="sequence"/>
      </id>
      <property name="nombre" type="string"/>

      <set name="mensajes">
            <key column="autorId"/>
            <one-to-many class="Mensaje"/>
         </set>

   </class>

   <class name="Mensaje">
   <id name="mensajeId" column="mensajeId">
      <generator class="sequence"/>
   </id>
   <property name="texto" type="string"/>

   <many-to-one name="autor"
      column="autorId"
      class="Autor"
      not-null="true"/>

   </class>
</hibernate-mapping>
```

Esto se mapea con las siguiente definiciones SQL de tablas:

```
create table autor (autorId bigint not null primary key,
nombre varchar(255))
create table mensaje (mensajeId bigint not null primary key,
texto varchar(255), autorId bigint )
alter table mensaje add constraint mensajefk0 (autorId) references
autor
```

Es muy importante recordar que cuando se define una relación bidireccional, hay que marcar uno de los lados con la etiqueta *inverse*. Básicamente, esto le dice a Hibernate que no ignore esa relación y que la obtenga a partir de su relación *espejo* en el otro lado. Cuando actualicemos una relación bidireccional, hay que asegurarse de actualizar siempre el lado no inverso. Aunque una recomendación mejor es actualizar siempre ambos lados de la relación (evitamos errores y mantenemos consistente la relación en memoria por si no se hace un *flush*).

Vemos que es una típica relación definida por una clave ajena de `mensaje` a `autor`.

Si queremos hacer la relación bidireccional debemos añadir al mensaje el atributo que apunta a su autor. En el fichero de mapeo declaramos que este atributo se corresponde a una relación muchos-a-uno (la inversa de la definida en autor) y en el mapeo de autor debemos indicar que se ha definido la relación bidireccional poniendo el elemento `inverse` a `true`. También ponemos el elemento `not-null` de `mensaje` a `true` para indicar que no puede haber mensajes sin autores.

```
<hibernate-mapping>
   <class name="Autor">
      <id name="autorId">
            <generator class="native"/>
      </id>
      <property name="nombre" type="string"/>

      <set name="mensajes">
            <key column="autorId" inverse="true"/>
            <one-to-many class="Mensaje"/>
         </set>

   </class>

   <class name="Mensaje">
   <id name="id" column="id">
      <generator class="native"/>
   </id>
   <property name="texto" type="string"/>

   <many-to-one name="autor"
      column="autorId"
      class="Autor"
      not-null="true"/>

   </class>
</hibernate-mapping>
```

La definición de clases anteriores genera las siguientes tablas:

```
create table parent ( id bigint not null primary key )
create table child ( id bigint not null
                     primary key,
                     name varchar(255),
                     parent_id bigint not null )
alter table child add constraint childfk0 (parent_id) references
 parent
```

Al igual que en JPA, para hacer persistente esta relación hay que actualizar el campo

Un ejemplo de cómo añadir objetos a la relación:

```
Cat cat = new DomesticCat();
Cat kitten = new DomesticCat();
....
Set kittens = new HashSet();
kittens.add(kitten);
cat.setKittens(kittens);
session.persist(cat);
kittens = cat.getKittens(); // Okay, kittens collection is a Set
(HashSet) cat.getKittens(); // Error!
```

13.4.2. Relación muchos-a-muchos

Por otro lado, si un mensaje puede tener más de un autor debemos definir una relación muchos-a-muchos. Supongamos que la relación es unidireccional y que en la definición de clases un `Autor` tiene una colección de mensajes escritos y que más de un autor puede haber sido autor del mismo mensaje. Debemos definir la relación como `many-to-many` en el lado del autor y utilizar una tabla más en el modelo relacional. Se trata de la tabla `autor-mensaje` que define la relación muchos-a-muchos relacionando las parejas de `autor` y `mensaje`

```
<hibernate-mapping>
   <class name="Autor">
      <id name="autorId">
             <generator class="native"/>
      </id>
      <property name="nombre" type="string"/>

      <set name="mensajes" table="autor-mensaje">
      <key column="autorId"/>
         <many-to-many class="mensaje" column="mensajeId"/>
      </set>
   </class>

   <class name="Mensaje">
      <id name="mensajeId">
         <generator class="sequence"/>
      </id>
      <property name="texto"/>
   </class>
</hibernate-mapping>
```

Las definiciones de tablas generadas son:

```
create table autor (autorId bigint not null primary key,
   nombre varchar(255))
create table mensaje (mensajeId bigint not null primary key,
   texto varchar(255))
create table autor-nombre (autorId bigint not null,
   mensajeId bigint not null,
   primary key (autorId, mensajeId))
```

13.5. Relaciones de herencia

Al igual que JPA, Hibernate permite tres estrategias para implementar el mapeo de las relaciones de herencia:

- Tabla única por jerarquía de clases
- Tabla por subclase
- Tabla por clase concreta

Veamos un ejemplo de la primera opción, la más común. Supongamos la misma jerarquía de clases que vimos en JPA.

La entidad `EmpleadoContratado` extiende la clase `Empleado`, añadiendo el atributo `planPensiones`:

```
public class EmpleadoContratado extends Empleado {
   private Long planPensiones;

   public Long getPlanPensiones() {
      return planPensiones;
   }

   public void setPlanPensiones(Long planPensiones) {
      this.planPensiones = planPensiones;
   }
}
```

La entidad `EmpleadoBecario` extiende la clase `Empleado`, añadiendo el atributo `seguroMedico`:

```
public class EmpleadoBecario extends Empleado {
   private Long seguroMedico;

   public Long getSeguroMedico() {
      return seguroMedico;
   }

   public void setSeguroMedico(Long seguroMedico) {
      this.seguroMedico = seguroMedico;
   }
}
```

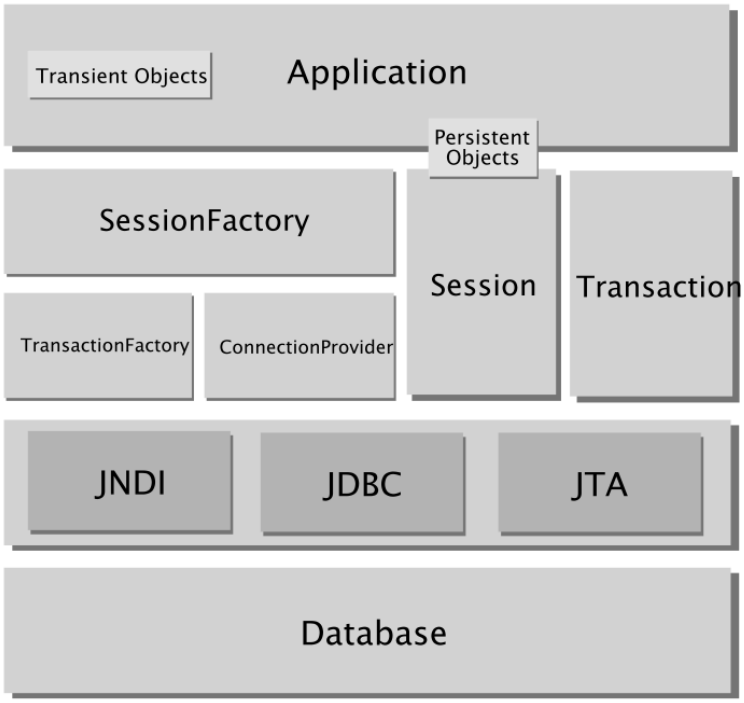
Para definir el mapeo, al igual que en JPA, hay que indicar la tabla que va a contener toda la jerarquía, la columna y los valores discriminantes. Un elemento `class` define toda la jerarquía:

```
<class name="Empleado" table="EMPLEADO">
    <id name="empleadoId" type="long" column="EMPLEADO_ID">
        <generator class="native"/>
    </id>
    <discriminator column="TIPO" type="string"/>
    <property name="nombre" type="string" column="NOMBRE"/>
    ...
    <subclass name="EmpleadoContratado"
```

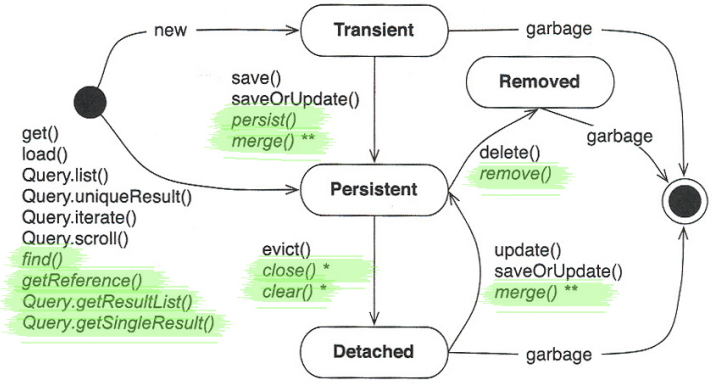
```
discriminator-value="contrato">
        <property name="planPensiones" type="long"
column="PLAN_PENSIONES"/>
    </subclass>
    <subclass name="EmpleadoBecario" discriminator-value="beca">
        <property name="seguroMedico" type="long"
column="SEGURO_MEDICO"/>
    </subclass>
</class>
```

13.6. Arquitectura de Hibernate

La arquitectura de Hibernate es idéntica a la de JPA, cambiando los nombres de los elementos:



El ciclo de vida de una entidad es el mismo que en JPA, cambiando los nombres de los métodos:



* Hibernate & JPA, affects all instances in the persistence context
** Merging returns a persistent instance, original doesn't change state

14. Roadmap Java Persistence API

14.1. Puntos destacados

- JPA es un API basado en la tecnología ORM (*Object Relational Mapping*) para trabajar con entidades persistentes
- Existen distintas implementaciones del API, en el módulo hemos trabajado con la implementación de Hibernate
- JPA es parte de la especificación EJB 3.0 de Java EE, pero también puede usarse en aplicaciones Java SE y aplicaciones web.
- JPA se basa en JDBC y es necesario tener en funcionamiento una base de datos SQL (como MySQL). Para usar el API debemos instalar el conjunto de librerías que lo implementan y el fichero de configuración `META-INF/persistence.xml`.
- Una de las características fundamentales de JPA es su simplicidad. Muchas opciones de configuración se definen por defecto.
- Las entidades en JPA son POJOs, clases Java sencillas a las que se les añaden una anotaciones para indicar las características de persistencia.
- En el mapeado entidad-relación las entidades se convierten en tablas y sus atributos en columnas.
- Es posible definir en JPA todas las posibles asociaciones entre entidades: uno-a-uno, uno-a-muchos, muchos-a-uno y muchos-a-muchos. El mapeo de muchas de estas asociaciones se implementa utilizando claves ajenas en la tabla propietaria de la relación. La definición de estas claves ajenas se realiza desde la entidad opuesta con el elemento `mappedBy` de la anotación que define la relación.
- El `EntityManager` es el elemento fundamental del API. Es el encargado de hacer persistentes las entidades que gestiona y de mantener la sincronización entre el contexto de persistencia (estado en memoria de las entidades) y la base de datos.
- Cuando el entity manager se cierra, todas las entidades del contexto de persistencia quedan desconectadas (*detached*) de la base de datos.
- La sincronización del contexto de persistencia y la base de datos se realiza mediante el volcado (*flush*) de sentencia SQL. Este volcado se realiza por defecto cuando se cierra una transacción o cuando se realiza una consulta.
- Por defecto, las colecciones y ciertos atributos de las entidades se cargan de forma perezosa (*lazy loading*), de forma que se guardan referencias y sólo se recuperan de la base de datos cuando se accede a ellos.
- En JPA es necesario abrir y cerrar una transacción siempre que se realiza una operación de actualización. Las transacciones que se utilizan en Java SE son las del recurso (el `autocommit` de JDBC). En Java EE se utilizan transacciones distribuidas JTA.
- En JPA se define un potente lenguaje de consulta basado en SQL llamado JPQL.

14.2. Certificación Sun

Los conocimientos de JPA son necesarios para superar la certificación de desarrollador de componentes de negocio (*Sun Certified Business Component Developer (SCBCD)*), que es la siguiente nivel una vez obtenida la certificación de desarrollador de componentes web.

La página oficial con los requisitos de la certificación se puede encontrar en el siguiente enlace: Sun Certified Business Component Developer (SCBCD). En concreto, se listan los siguientes objetivos:

Entidades JPA

- Identify correct and incorrect statements or examples about the characteristics of Java Persistence entities.
- Develop code to create valid entity classes, including the use of fields and properties, admissible types, and embeddable classes.
- Identify correct and incorrect statements or examples about primary keys and entity identity, including the use of compound primary keys.
- Implement association relationships using persistence entities, including the following associations: bidirectional for @OneToOne, @ManyToOne, @OneToMany, and @ManyToMany; unidirectional for @OneToOne, @ManyToOne, @OneToMany, and @ManyToMany.
- Given a set of requirements and entity classes choose and implement an appropriate object-relational mapping for association relationships.
- Given a set of requirements and entity classes, choose and implement an appropriate inheritance hierarchy strategy and/or an appropriate mapping strategy.
- Describe the use of annotations and XML mapping files, individually and in combination, for object-relational mapping.

Operaciones sobre las entidades

- Describe how to manage entities, including using the EntityManager API and the cascade option.
- Identify correct and incorrect statements or examples about entity instance lifecycle, including the new, managed, detached, and removed states.
- Identify correct and incorrect statements or examples about EntityManager operations for managing an instance's state, including eager/lazy fetching, handling detached entities, and merging detached entities.
- Identify correct and incorrect statements or examples about Entity Listeners and Callback Methods, including: @PrePersist, @PostPersist, @PreRemove, @PostRemove, @PreUpdate, @PostUpdate, and @PostLoad, and when they are invoked.
- Identify correct and incorrect statements about concurrency, including how it is managed through the use of @Version attributes and optimistic locking.

Unidades y contextos de persistencia

- Identify correct and incorrect statements or examples about JTA and resource-local

entity managers.
- Identify correct and incorrect statements or examples about container-managed persistence contexts.
- Identify correct and incorrect statements or examples about application-managed persistence contexts.
- Identify correct and incorrect statements or examples about transaction management for persistence contexts, including persistence context propagation, the use of the EntityManager.joinTransaction() method, and the EntityTransaction API.
- Identify correct and incorrect statements or examples about persistence units, how persistence units are packaged, and the use of the persistence.xml file.
- Identify correct and incorrect statements or examples about the effect of persistence exceptions on transactions and persistence contexts.

JPQL

- Develop queries that use the SELECT clause to determine query results, including the use of entity types, use of aggregates, and returning multiple values.
- Develop queries that use Java Persistence Query Language syntax for defining the domain of a query using JOIN clauses, IN, and prefetching.
- Use the WHERE clause to restrict query results using conditional expressions, including the use of literals, path expressions, named and positional parameters, logical operators, the following expressions (and their NOT options): BETWEEN, IN, LIKE, NULL, EMPTY, MEMBER [OF], EXISTS, ALL, ANY, SOME, and functional expressions.
- Develop Java Persistence Query Language statements that update a set of entities using UPDATE/SET and DELETE FROM.
- Declare and use named queries, dynamic queries, and SQL (native) queries.
- Obtain javax.persistence.Query objects and use the javax.persistence.Query API.

Hemos cubierto bastantes de estos objetivos en el módulo. Muchos de los que no han sido cubiertos aquí los trataremos en el módulo de EJB.

14.3. Recursos adicionales

14.3.1. Bibliografía

- Mike Keith, Merrick Schincariol, *Pro EJB 3. Java Persistence API*, Apress, 2006
- Christian Bauer, Gavin King, *Java Persistence with Hibernate*, Manning, 2006

14.3.2. Enlaces

Referencias

- Paquete javax.persistence
- Manual de referencia de Hibernate-JPA

- Dali - Entorno IDE para JPA incluido en Eclipse

Tutoriales y artículos

- Using the Java Persistence API in Desktop Applications
- Dali Object-Relational Mapping Tool
- Artículo en TheServerSide: Defining your object model with JPA
- Blog de Gavin King
- Presentación de Martin Krajcy