Entity Manager y contexto de persistencia

Índice

1. Entity Manager y contexto de persistencia

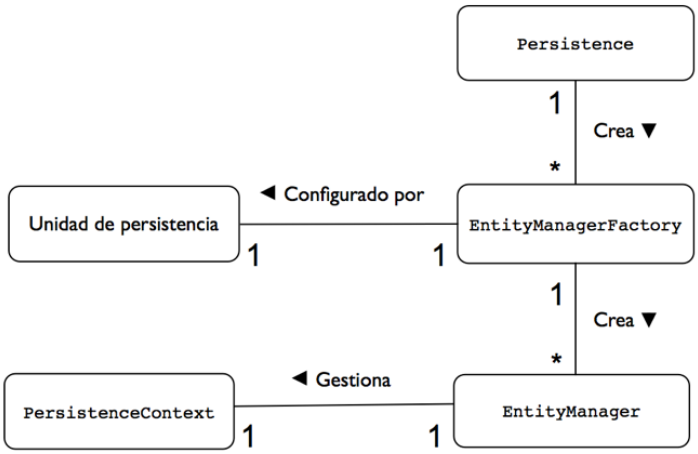
En las sesiones anteriores hemos visto que todas las operaciones relacionadas con la persistencia de las entidades se realizan a través de un *gestor de entidades* (*entity manager* en inglés). El funcionamiento del *entity manager* está especificado en una única interfaz llamada `EntityManager` (enlace a javadoc).

El *entity manager* tiene dos responsabilidad fundamentales:

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

El *entity manager* se obtiene a través de una factoría del tipo `EntityManagerFactory`, que se configura mediante la especificación de una *unidad de persistencia* (*persistence unit* en inglés) definida en el fichero XML `persistence.xml`. En el fichero pueden haber definidas más de una unidad de persistencia, cada una con un nombre distinto. El nombre de la unidad de persistencia escogida 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 *entity managers* 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. 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.



Una vez creado el *entity manager* lo utilizaremos para realizar todas las operaciones de recuperación, consulta y actualización de entidades. Cuando un *entity manager* obtiene una referencia a una entidad, se dice que la entidad está gestionada (una *managed entity* en inglés) por él. El *entity manager* guarda internamente todas las entidades que gestiona y las utiliza como una caché de los datos en la base de datos. Por ejemplo, cuando va a recuperar una entidad por su clave primaria, lo primero que hace es consultar en su caché si esta entidad ya la ha recuperado previamente. Si es así, no necesita hacer la búsqueda en la base de datos y devuelve la propia referencia que mantiene. Al conjunto de entidades gestionadas por un *entity manager* se le denomina su *contexto de persistencia* (*persistence context* en inglés).

En un determinado momento, el *entity manager* debe volcar a la base de datos todos los cambios que se han realizado sobre las entidades. También debe ejecutar las consultas JPQL definidas. Para ello el *entity manager* utiliza un *proveedor de persistencia* (*persistence provider* en inglés) que es el responsable de generar todo el código SQL compatible con la base de datos.

1.1. Obtención de un EntityManager

La forma de obtener un *entity manager* varía dependiendo de si estamos utilizando JPA gestionado por la aplicación (cuando utilizamos JPA en Java SE) o si estamos utilizando JPA en una aplicación gestionada por un servidor de aplicaciones Java EE. En el segundo caso se utiliza un método denominado *inyección de dependencias* y el servidor de aplicaciones será el responsable de obtener el *entity manager* e *inyectarlo* en una variable que tiene una determinada anotación. Lo veremos más adelante.

En el primer caso, cuando estamos usando JPA gestionado por la aplicación, el *entity*

manager se obtiene a partir de un `EntityManagerFactory`. Para obtener la factoría se debe llamar al método estático `createEntityMagerFactory()` 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, "dennis.ritchie@gmail.com");
      Mensaje mensaje = new Mensaje("Hola mundo", autor);
      em.persist(mensaje);
      autor.getMensajes().add(mensaje);

      tx.commit();
      em.close();
   }
}
```

Primero se obtiene el *entity manager* a partir de la llamada al método `createEntityManager` de la clase `Persistence`. Después se marca el comienzo de una transacción y se recupera un autor de la base de datos con la llamada a `find`. Después se crea un nuevo mensaje, que se incorpora al contexto de persistencia con el método `persist`. Por último se actualiza la colección con los mensajes creados por el autor y se cierra la transacción y la entidad de persistencia.

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. De esta forma se evita 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 acceder para obtener *entity managers*. Lo llamamos `PersistenceManager` y lo definimos en el paquete *persistence* en el que vamos a crear la capa de persistencia:

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

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

public class PersistenceManager {
   static private final String PERSISTENCE_UNIT_NAME = "simplejpa";
   protected static PersistenceManager me = null;
   private EntityManagerFactory emf = null;

   private PersistenceManager() {
      if (emf == null) {
         emf =
Persistence.createEntityManagerFactory(PERSISTENCE_UNIT_NAME);
         this.setEntityManagerFactory(emf);
      }
   }

   public static PersistenceManager getInstance() {
      if (me == null) {
         me = new PersistenceManager();
      }
      return me;
   }

   public void setEntityManagerFactory(EntityManagerFactory emf) {
      this.emf = emf;
   }

   public EntityManagerFactory getEntityManagerFactory() {
      return this.emf;
   }

   public EntityManager createEntityManager() {
      return emf.createEntityManager();
   }
}
```

En la clase se define una cadena estática con el nombre de la unidad de persistencia que carga el singleton, en este caso `simplejpa`.

Para obtener el entity manager debemos hacer ahora lo siguiente:

```
public class AutorTest {
   public static void main(String[] args) {
      EntityManager em =
PersistenceManager.getInstance().createEntityManager();
      em.getTransaction().begin();

      Autor autor = em.find(Autor.class, "dennis.ritchie@gmail.com");
      Mensaje mensaje = new Mensaje("Hola mundo", autor);
```

```
        em.persist(mensaje);
        autor.getMensajes().add(mensaje);

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

1.2. Contexto de persistencia

Una cuestión muy importante para entender el funcionamiento del *entity manager* es comprender su contexto de persistencia. Contiene la colección de entidades gestionadas por el *entity manager* que están conectadas y sincronizadas con la base de datos. Cuando el entity manager cierra una transacción, su contexto de persistencia se sincroniza automáticamente 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 es así, se devuelve su referencia. Si no se ha recuperado previamente, se obtiene la instancia de la entidad de la base de datos.
2. La aplicación utiliza las entidades 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. La sincronización no se realiza 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.

1.3. Operaciones del entity manager

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

- `void clear()`: borra el contexto de persistencia, desconectando todas sus entidades

- `boolean contains(Object entity)`: comprueba si una entidad está gestionada en el contexto de persistencia
- `Query createNamedQuery(String name)`: obtiene una consulta JPQL precompilada
- `void detach(Object entity)`: elimina la entidad del contexto de persistencia, dejándola desconectada de la base de datos
- `<T> T find(Class<T>, Object key)`: busca por clave primaria
- `void flush()`: sincroniza el contexto de persistencia con la base de datos
- `<T> T getReference(Class<T>, Object key)`: obtiene una referencia a una entidad, que puede haber sido recuperada de forma *lazy*
- `EntityTransaction getTransaction()`: devuelve la transacción actual
- `<T> T merge(T entity)`: incorpora una entidad al contexto de persistencia, haciéndola gestionada
- `void persist(Object entity)`: hace una entidad persistente y gestionada
- `void refresh(Object entity)`: refresca el estado de la entidad con los valores de la base de datos, sobreescribiendo los cambios que se hayan podido realizar en ella
- `void remove(Object entity)`: elimina la entidad

Vamos a estudiarlas con más detalle.

1.3.1. Persist para hacer persistente una entidad

El método `persist()` del `EntityManager` acepta una nueva instancia de entidad y la convierte en gestionada. Si la entidad que se pasa como parámetro ya está gestionada en el contexto de persistencia, la llamada se ignora. La operación `contains()` puede usarse para comprobar si una entidad está gestionada.

El hecho de convertir una entidad en gestionada no la hace persistir inmediatamente en la base de datos. La verdadera llamada a SQL para crear los datos relacionales no se generará hasta que el contexto de persistencia se sincronice con la base de datos. 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 existe en la base de datos el proveedor de persistencia puede detectarlo y lanzar una excepción `EntityExistsException`. Si no lo hace, entonces se lanzará la excepción cuando se sincronice el 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.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 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`.

El código asume la existencia de un `EntityManager` en la variable de instancia `em` y lo usa para hacer persistente el empleado recién creado.

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`.

Una cuestión a tener en cuenta es que el identificador del empleado puede no estar disponible en un tiempo, hasta que se realice el *flush* del contexto de persistencia.

Una mejora del código anterior, que garantiza que el identificador del empleado se carga en la entidad sería la siguiente:

```
...
em.persist(empleado);
em.flush(empleado);
em.refresh(empleado);
```

La llamada a `flush` asegura que se ejecuta el `insert` en la BD y la llamada a `refresh` asegura que el identificador se carga en la instancia.

1.3.2. Find para buscar entidades

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

```
Empeado empleado = 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.

Existe una versión especial de `find()` que sólo recupera una referencia a la entidad, sin obtener los datos de los campos de la base de datos. Se trata del método `getReference()`. Es últil cuando se quiere añadir un objeto con una clave primaria conocida a una relación. 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.

1.3.3. 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.

1.3.4. Remove para borrar 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 empleado = em.find(Empleado.class, 146);
em.remove(emp);
```

La llamada a `remove` asume que el empleado existe. En el caso de no existir se lanzaría una excepción.

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.

1.3.5. 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.

1.3.6. Actualización de entidades

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 empleado = em.find(Empleado.class, 146);
empleado.setSueldo(empleado.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.

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.

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.

1.3.7. 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`.

1.3.7.1. 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.

1.3.7.2. 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.

1.3.8. 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 una transacción:

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

1.3.9. 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();
}
```

2. Trabajando con el contexto de persistencia

El trabajo con el contexto de persistencia es uno de los aspectos más complicado con JPA. Veamos algunos ejemplos que pueden aclarar algunas cuestiones.

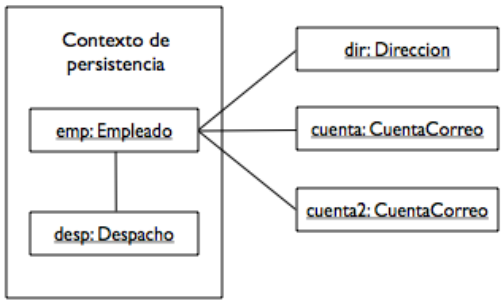
2.1. Sincronización con la base de datos

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`.

2.2. Actualización de las colecciones en las relaciones a-muchos

En una relación uno-a-muchos el campo que mantiene la colección es un *proxy* que nos permite acceder a los elementos propiamente dichos, pero sólo si la entidad está gestionada. Hay que tener cuidado porque si la parte inversa de la relación se actualiza con un nuevo elemento, JPA no tiene forma de saberlo por si solo. La aplicación debe actualizar la colección también.

Veamos un ejemplo. Supongamos que tenemos la relación uno-a-muchos entre `AutorEntity` y `MensajeEntity` definida en la sesión anterior, que la entidad `MensajeEntity` contiene la clave ajena hacia `AutorEntity` y que la aplicación ejecuta el siguiente código para añadir un mensaje a un autor:

```
em.getTransaction().begin();
AutorEntity autor = em.find(AutorEntity.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");
MensajeEntity mens = new MensajeEntity("Nuevo mensaje");
mens.setAutor(autor);
em.persist(mens);
em.getTransaction().commit();
autor.getMensajes().add(mens);
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.

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.

2.3. 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. Además, son objetos que pueden hacer el papel de DTOs (*Data Transfer Objets*) y ser devueltos por la capa de lógica de negocio y utilizados por la capa de presentación. La 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.

3. Cómo implementar y usar un DAO con JPA

El patrón DAO (*Data Access Object*) se ha utilizado históricamente para ocultar los detalles de implementación de la capa de persistencia y ofrecer una interfaz más sencilla a otras partes de la aplicación. Esto es necesario cuando se trabaja con un API de persistencia de bajo nivel, como JDBC, en la que las operaciones sobre la base de datos conlleva la utilización de sentencias SQL que queremos aislar de otras capas de la aplicación.

Sin embargo, en desarrollos en los que utilizamos un API de persistencia de mayor nivel como JPA, la utilización de un patrón DAO no es tan necesaria. Muchos argumentan incluso que se trata de un *antipatrón*, un patrón que conviene evitar. No creemos que sea así, porque un patrón DAO bien diseñado y adaptado a JPA permite eliminar errores derivados de la mala utilización del contexto de persistencia y agrupar en una única clase todas las consultas relacionadas con una misma clase de dominio.

Presentamos a continuación una adaptación del patrón DAO propuesto por Adam Bien, uno de los autores y expertos sobre Java EE más importantes en la actualidad.

Se define una clase DAO para cada una de las entidades definidas como en el dominio de la aplicación. Los DAO trabajarán con entidades gestionadas y proporcionarán las

operaciones básicas CRUD sobre la entidad (*Create*, *Read*, *Update* y *Delete*), así como las consultas JPQL.

Los DAO trabajarán con un *entity manager* y una transacción abierta, de forma que podremos englobar distintas operaciones de distintos DAOs en una misma transacción y un mismo contexto de persistencia. La capa responsable de abrir la transacción y el *entity manager* será la capa que utiliza los DAO, la que implementa la lógica de negocio. Si se intenta llamar a cualquiera de los métodos del DAO sin que se haya abierto una transacción en el *entity manager* se deberá lanzar una excepción.

Vamos a definir todas las clases DAO en el paquete `persistence` y las clases que implementan la lógica de negocio en el paquete `service`.

La implementación concreta se especifica a continuación. Agrupamos todas las funciones CRUD comunes a todos los DAO en una clase abstracta genérica, que tiene como parámetro la entidad que va a gestionar y el tipo de su clave primaria de la entidad. Después se define una clase DAO específica por cada una de las entidades de dominio que extiende esta clase genérica.

Clase DAO genérica:

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

import javax.persistence.EntityManager;

abstract class Dao<T, K> {
   EntityManager em;

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

   public EntityManager getEntityManager() {
      return this.em;
   }

   public abstract T find(K id);

   public T create(T t) {
      checkTransaction();
      em.persist(t);
      em.flush();
      em.refresh(t);
      return t;
   }

   public T update(T t) {
      checkTransaction();
      return (T) em.merge(t);
   }

   public void delete(T t) {
      checkTransaction();
      t = em.merge(t);
      em.remove(t);
   }

   private void checkTransaction() {
```

```
      if (!em.getTransaction().isActive())
         throw new RuntimeException("Transacción inactiva");
   }
}
```

Algunos comentarios sobre los métodos:

- `checkTransaction`: método privado que comprueba que existe una transacción abierta y si no es así, lanza una excepción. Todos los métodos del DAO lo llaman.
- `find`: busca una entidad utilizándose su clave primaria, devolviendo `null` si no se encuentra. La debe implementar la subclase, porque el método `find` del *entity manager* utiliza como primer parámetro la clase de la entidad a buscar y Java no permite obtenerla a partir del tipo paramétrico `L`.
- `create`: hace persistente una entidad que se pasa como parámetro y la devuelve. En su código se llama a los métodos `flush` y `refresh` para asegurar que la clave primaria autogenerada por la base de datos se carga en la entidad. Se devuelve una entidad gestionada.
- `update`: se actualiza el valor de una entidad en el contexto de persistencia, llamando al método `merge` para asegurarse de que la entidad se convierte en gestionada. Si la entidad ya está gestionada no se hace nada. Se devuelve la propia entidad. Cuando se termine la transacción y el *entity manager* realice un *flush*, el valor de la entidad se volcará a la base de datos.
- `delete`: se elimina una entidad de la base de datos, llamando al método `remove` una vez nos hemos asegurado de que la entidad está gestionada llamando al método `merge`.

Un ejemplo concreto de DAO, `PeliculaDao`, que extiende la clase anterior:

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

import java.util.List;
import javax.persistence.EntityManager;
import es.ua.jtech.jpa.filmoteca.domain.Pelicula;

public class PeliculaDao extends Dao<Pelicula, Long> {

   public PeliculaDao(EntityManager em) {
      super(em);
   }

   public Pelicula find(Long id) {
      EntityManager em = this.getEntityManager();
      return em.find(Pelicula.class, id);
   }

   @SuppressWarnings("unchecked")
   public List<Pelicula> getPeliculasActor(String nombreActor) {
      EntityManager em = this.getEntityManager();
      return (List<Pelicula>) em
            .createNamedQuery("Pelicula.peliculasActor")
            .setParameter("actorNombre", nombreActor).getResultList();

   }
}
```

Veamos ahora un ejemplo de clase que contiene lógica de negocio, la clase `PeliculaService`. Definimos en ella algunos métodos de negocio, como:

- `creaPelicula(titulo, fechaEstreno, pais, presupuesto)`
- `getPeliculasActor(nombreActor)`
- `actualizaRecaudacionPelicula(idPelicula, recaudacion)`

Todos los métodos de lógica de negocio son exportados a otras capas (posiblemente remotas) de la aplicación. A diferencia de los métodos de los DAO, que toman y devuelven entidades gestionadas, los métodos de negocio toman y devuelven objetos planos, desconectados de cualquier transacción o gestor de persistencia. Son métodos atómicos. Cuando terminan podemos estar seguros de que se ha realizado la operación. En el caso en que haya algún problema, la operación no se realizará en absoluto (el estado de los objetos quedará como estaba antes de llamar a la operación) y se lanzará una excepción runtime.

Resaltamos en el código las llamadas a los DAO.

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

// imports

public class PeliculaService {

   public void actualizaRecaudacionPelicula(Long idPelicula,
         Double recaudacion) {
      EntityManager em = PersistenceManager.getInstance()
            .createEntityManager();
      em.getTransaction().begin();

      PeliculaDao daoPelicula = new PeliculaDao(em);

      Pelicula pelicula = daoPelicula.find(idPelicula);
      pelicula.setRecaudacion(recaudacion);
      daoPelicula.update(pelicula);

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

   public Long creaPelicula(String titulo, Date fechaEstreno,
                            String pais, Double presupuesto) {
      if (titulo == null || fechaEstreno == null)
         throw new IllegalArgumentException(
               "Falta título o fecha de estreno de película");
      EntityManager em = PersistenceManager.getInstance()
            .createEntityManager();
      em.getTransaction().begin();

      PeliculaDao daoPelicula = new PeliculaDao(em);

      Pelicula pelicula = new Pelicula(titulo, fechaEstreno);
      pelicula.setPresupuesto(presupuesto);
      pelicula.setPais(pais);

      daoPelicula.create(pelicula);

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

```
      return pelicula.getId();
   }

   public List<Pelicula> getPeliculasActor(String nombreActor) {
      EntityManager em = PersistenceManager.getInstance()
            .createEntityManager();

      PeliculaDao daoPelicula = new PeliculaDao(em);

      List<Pelicula> peliculas = daoPelicula
            .getPeliculasActor(nombreActor);

      em.close();
      return peliculas;
   }
}
```

Todos los métodos de negocio tienen la misma estructura:

1. Se obtiene un *entity manager* y se abre la transacción si se va a hacer algún cambio en las entidades. No es necesario abrir una transacción cuando se va a hacer sólo una consulta.
2. Se obtienen los DAO de las entidades que intervienen en la operación.
3. Se realizan las consultas y modificaciones de las entidades que participan en la operación, actualizando su estado en la base de datos mediante los DAO.
4. Se realiza el commit de la transacción y se cierra el *entity manager*.

Presentamos a continuación un ejemplo algo más complicado, en el que se utilizan varios DAOs desde una única transacción y entity manager. Se trata de una supuesta operación `transferencia(cantidad, idCuentaOrigen, idCuentaDestino)` en la que se pasa una cantidad de dinero de una cuenta a otra. Se realiza también una notificación de un mensaje de texto a cada una de las cuentas. Toda la operación es atómica, o se realiza completamente o no se realiza nada:

```
public class CuentaCorrienteService {

   public void transferencia(Long cantidad,
                             Long idCuentaOrigen,
                             Long idCuentaDestino) {

      EntityManager em = PersistenceManager.getInstance()
          .createEntityManager();
      em.getTransaction().begin();

      CuentaDao cuentaDao = new CuentaDao(em);
      NotificacionDao notificiacionDao = new NotificacionDao(em);

      // Obtenemos las entidades de cada cuenta
      Cuenta cuentaOrigen = cuentaDao.find(idCuentaOrigen);
      Cuenta cuentaDestino = cuentaDao.find(idCuentaDestino);

      // Actualizamos sus saldos
      Long saldoOrigen = cuentaOrigen.getSaldo();
      Long saldoDestino = cuentaDestino.getSaldo();
      cuentaOrigen.setSaldo(saldoOrigen - cantidad);
      cuentaDestino.setSaldo(saldoDestino + cantidad);
      cuentaDao.update(cuentaOrigen);
      cuentaDao.update(cuentaDestino);
```

```
      // Creamos las notificaciones
      Notificacion notificacion1 =
            new Notificacion(cuentaOrigen, "TRANSFERENCIA A "
                                          +
cuentaDestino.getDescripcion()
                                          + " DE "
                                          + cantidad.toString());
      notificacionDao.create(notificacion1);
      Notificacion notificacion2 =
            new Notificacion(cuentaDestino, "TRANSFERENCIA DE "
                                          + cuentaOrigen.getDescripcion()
                                          + " DE "
                                          + cantidad.toString());
      notificacionDao.create(notificacion2);

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