Frameworks de persistencia - JPA

Sesión 2 - Entity Manager y contexto de persistencia

Índice

Entidades persistentes

- Una entidad agrupa un conjunto de datos de forma coherente. Se corresponde con una tabla de una base de datos. Las instancias de la entidad se corresponden con filas de la tabla.
- Ejemplos: Hotel, Vuelo, Estudiante, ...
- Modelo de persistencia
  - En bases de datos orientadas a objetos puras una entidad se crea en la base de datos cuando se hace un `new()`
  - En JPA no pasa así: una entidad es un POJO (*Plain Old Java Object*) con unas anotaciones. La creación de objetos se hace a través de llamadas explícitas al *entity manager*.

Un ejemplo de entidad (1)

```java
@Entity
public class Autor {

    @Id
    @GeneratedValue
    @Column(name = "autor_id")
    Long id;
    @Column(name="email", nullable = false, unique = true)
    private String correo;
    private String nombre;
    @OneToMany(mappedBy = "autor", cascade = CascadeType.ALL, fetch = FetchType.EAGER)
    private Set<Mensaje> mensajes = new HashSet<Mensaje>();
    @Version
    private int version;

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

    public String getCorreo() { return correo; }
    public void setCorreo(String correo) { this.correo = correo; }

    public String getNombre() { return nombre; }
    public void setNombre(String nombre) { this.nombre = nombre; }

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

Un ejemplo de entidad (2)

```
    public Autor() {
    }

    public Autor(String nombre, String correo) {
        this.nombre = nombre;
        this.correo = correo;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        Autor autor = (Autor) o;

        return !(id != null ? !id.equals(autor.id) : autor.id != null);

    }

    @Override
    public int hashCode() {
        return id != null ? id.hashCode() : 0;
    }

    @Override
    public String toString() {...}
}
```

Entity manager

- Similar a una conexión JDBC
- Hay que crearlo antes de comenzar a trabajar con las entidades
- Las recuperaciones y búsquedas se hacen a través de él
- Proporciona métodos para trabajar con transacciones
- Hay que cerrarlo al final de realizar el trabajo con las entidades

Contexto de persistencia

- Conjunto de entidades conectadas a la base de datos que son gestionadas por un *entity manager*
- Contiene todas las entidades que han sido creadas (`persist`), recuperadas o mezcladas (`merge`) por el *entity manager*
- Cuando el *entity manager* se cierra con una llamada a `close()`, las entidades permanecen en memoria como objetos Java, pero desconectadas de la BD
- Se puede considerar una caché de primer nivel de la BD. Cuando el *entity manager* busca una entidad (`find`) donde primero mira es en su contexto de persistencia.

EntityManagerFactory

- Factoría de la que se obtienen entity managers
- La creación es costosa, ya que conlleva el mapeado de las tablas con la BD. Normalmente se hace una única vez.
- Se pas como parámetro el nombre de una unidad de persistencia definida en el fichero `META-INF/persistence.xml`
- En la unidad de persistencia se define la configuración de la base de datos a la que va a acceder el *entity manager*: nombre, driver, contraseñas, ...
- El *entity manager* se obtiene con un método de la clase `EntityManagerFactory`

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

Singleton que almacena el entityManagerFactory

- Es muy habitual usar un *singleton* en el que guardar el entityManagerFactory para asegurarnos de que sólo se carga una vez

```
public class EmfSingleton {
    private static EmfSingleton ourInstance =
            new EmfSingleton();
    static private final String PERSISTENCE_UNIT_NAME = “mensajes-mysql";
    private EntityManagerFactory emf = null;

    public static EmfSingleton getInstance() {
        return ourInstance;
    }

    private EmfSingleton() {
    }

    public EntityManagerFactory getEmf() {
        if (this.emf == null)
            this.emf = Persistence
                    .createEntityManagerFactory(PERSISTENCE_UNIT_NAME);
        return this.emf;
    }
}

// Uso: EntityManagerFactory emf = EmfSingleton.getInstance().getEmf();
```

Operaciones del entity manager

- El entity manager es el responsable de mantener las entidades sincronizadas con la base de datos
- Operaciones sobre las entidades:
  - Hacer persistente una entidad
  - Buscar una entidad en la BD
  - Borrar entidades
  - Actualizar entidades
  - Queries a la BD
- Operaciones sobre el contexto de persistencia:
  - Sincronización de la base de datos
  - Desconexión de entidades
  - Mezcla de entidades

API EntityManager

Enlace al API EntityManager

- `void clear()`
- `boolean contains(Object entity)`
- `Query createNamedQuery(String name)`
- `void detach(Object entity)`
- `<T> T find(Class<T>, Object key)`
- `void flush()`
- `<T> T getReference(Class<T>, Object key)`
- `EntityTransaction getTransaction()`
- `<T> T merge(T entity)`
- `void persist(Object entity)`
- `void refresh(Object entity)`
- `void remove(Object entity)`

Creación de entidades

- Una vez creada una instancia entity como un objeto Java normal, hay que llamar al método `persist(entity)` para hacerla persistente
- El método no devuelve nada, pero la entidad pasará a formar parte del contexto de persistencia y será gestionada por el entity manager
- El entity manager generará una sentencia `SQL INSERT` la siguiente vez que se realice un *flush*
- Si la entidad tiene un identificador ya existente en la base de datos se generará una excepción `runtime javax.persistence.EntityExistsException`
- **CUIDADO**: Cuando el identificador es generado por la base de datos, el objeto no tiene identificador hasta que se realiza la sincronización con la BD

Persist

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

// Añadimos un nuevo empleado al departamento 1
Departamento depto = em.find(Departamento.class, 1L);
Empleado empleado = new Empleado();
empleado.setNombre("Pedro");
empleado.setSueldo(38000.0);
empleado.setDepartamento(depto);
em.persist(empleado);

// Actualizamos la colección en memoria de empleados del departamento
depto.getEmpleados().add(empleado);

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

	id	nombre	sueldo	departamento_id
1	1	Domingo Gallardo	2100	1
2	2	Otto Colomina	3000	1
3	3	Francisco Moreno	3500	2
4	4	Pedro	38000	1

Búsqueda de entidades

- Para buscar una entidad en la base de datos hay que llamar al método `find(clase, id)`, pasando como parámetro la clase de entidad que se busca y el identificador de la instancia
- Si no existe ninguna entidad con ese identificador, se devuelve `null`

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

getReference()

- Obtiene una referencia a una entidad sin recuperar sus datos de la BD. Se devuelve una entidad recuperada de forma perezosa (*lazy fetched*).
- Mejoramos la eficiencia, porque las consultas no tienen que recuperar todos los campos
- Cuidado: no devuelve `null` si la entidad no existe, genera una excepción

```
Departamento depto = em.getReference(Departamento.class, 1L);
Empleado empleado = new Empleado();
empleado.setNombre("Pedro");
empleado.setSueldo(38000.0);
empleado.setDepartamento(depto);
em.persist(empleado);
```

Borrado de entidades

- El *entity manager* puede borrar una entidad gestionada con el método **`remove()`**
- Se genera una sentencia **`DELETE`** la siguiente vez que se hace un *flush*

```
Empleado empleado = em.find(Empleado.class, 1L);
em.remove(emp);
```

- Hay que tener cuidado con las relaciones para que la BD no lance excepciones por estado inconsistente. Antes de borrar hay que poner a **`null`** las relaciones en las que participa:

```
Empleado empleado = em.find(Empleado.class, 3L);
Despacho desp = empleado.getDespacho();
empleado.setDespacho(null);
em.remove(desp);
```

Modificación de entidades

- Una vez que el *entity manager* gestiona una entidad, para modificar uno de sus atributos no hay más que llamar a un método *setter* de la entidad
- La siguiente vez que se haga un *flush* se sincroniza el estado de la entidad con la BD con una sentencia `UPDATE`

```
em.getTransaction().begin();
Empleado empleado = em.find(Empleado.class, 2L);
double sueldo = empleado.getSueldo();
empleado.setSueldo(sueldo + 1000.0);
em.getTransaction().commit();
em.close();
```

Sincronización con la BD

- El *entity manager* es responsable de mantener la sincronización entre la BD y el contexto de persistencia. El contexto de persistencia define una caché que el entity manager debe volcar (flush) en la BD.
- Simplificando, el funcionamiento del entity manager es el siguiente:
  1. Si la aplicación solicita una entidad (mediante un `find`, o accediendo a una relación con otra entidad), 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 hace una consulta 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.

Merge para actualizar entidades

```
EntityManager em = emf.createEntityManager();
Empleado empleado = em.find(Empleado.class, 2L);
em.close();

empleado.setNombre("Pepito Pérez");

em = emf.createEntityManager();
em.getTransaction().begin();
empleado = em.merge(empleado);
empleado.setSueldo(20000.0);
em.getTransaction().commit();
em.close();
```

Queries a la BD

Enlace al API Query

- Es posible realizar consultas elaboradas utilizando el lenguaje JPQL y el API Criteria
- Se devuelve un objeto o una colección a los que hay que hacer un casting (o null si no hay resultados)

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

Operaciones en cascada

- Es posible definir operaciones en cascada de forma que una única llamada a una operación del *entity manager* sobre una entidad se propague a todo el grafo de entidades con la que está relacionada.
- Es posible restringir las operaciones que se van a hacer en cascada: `PERSIST`, `REFRESH`, `REMOVE`, `MERGE` y `ALL`

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

Flush

- Es posible obligar al *entity manager* a sincronizar el contexto de persistencia con la base de datos usando el método `flush()`
- Con el método `setFlushMode(FlushModeType)` se le puede indicar al *entity manager* cuándo hacer la sincronización
- Los posibles valores de `FlushModeType` son:
  - `FlushModeType.AUTO` (valor por defecto): se sincroniza previamente a una consulta y al hacer un commit
  - `FlushModeType.COMMIT`: sólo se sincroniza cuando la transacción hace un commit

Transacciones

- Todas las operaciones de actualización del *entity manager* hay que hacerlas dentro de una transacción, en caso contrario se genera una excepción runtime de tipo `javax.persistence.TransactionRequiredException`
- Se obtiene una transacción con el método `beginTransaction()` del entity manager
- Dada una transacción es posible:
  - Cerrarla con `commit()`
  - Deshacerla con `rollback()`

Ejemplo transacción

```
    public static void main(String[] args) {
        Autor autor;

        EntityManagerFactory emf = Persistence
                .createEntityManagerFactory("mensajes-mysql");
        EntityManager em = emf.createEntityManager();

        try {
            em.getTransaction().begin();
            String email = leerTexto("Introduce el correo electrónico: ");
            String nombre = leerTexto("Introduce nombre: ");
            autor = new Autor(nombre, email);
            em.persist(autor);
            em.flush();
            System.out.println("Identificador del autor: " + autor.getId());

            String mensajeStr = leerTexto("Introduce mensaje: ");
            Mensaje mens = new Mensaje(mensajeStr, autor);
            mens.setFecha(new Date());
            em.persist(mens);

            System.out.println("Identificador del mensaje: " + mens.getId());

            em.getTransaction().commit();
        } catch (Exception ex) {
            em.getTransaction().rollback();
            System.out.println("Error: " + ex.getMessage() + "\n\n");
            ex.printStackTrace();
        } finally {
            em.close();
            emf.close();
        }
    }
```

Desconexión de entidades

- Una vez cerrado el contexto de persistencia (cuando se llama al método `close`), las entidades quedan en memoria como objetos Java desconectados de la base de datos.
- Sólo tendremos acceso a los atributos que se hayan cargado durante la transacción.
- Hay que tener cuidado con las colecciones relacionadas: si no las hemos traído a memoria no podremos acceder a ellas. Siempre que sea posible es recomendable marcar la relación con EAGER para que JPA traiga la colección relacionada automáticamente cuando se recupera la entidad.

Cuidado con el lazy loading

- En JPA es fundamental el concepto de carga perezosa (*lazy loading*): un objeto está cargado de forma perezosa cuando sus datos no están en memoria, sino que se recuperan (mediante una consulta la BD) al acceder a sus campos o sus elementos
- En general JPA recupera los grafos de objetos de la siguiente forma:
  - Cuando recupera una entidad, carga en memoria las entidades con las que tiene una relación a-uno
  - Las relaciones a-muchos las carga de forma perezosa. La colección que se devuelve realmente no contiene datos, sólo un "proxy" de JPA que generará una consulta a la BD cuando recuperemos sus elementos
- Si el entity manager se cierra, no se pondrá acceder a los objetos *lazy*, hay que tener cuidado de cargarlos en memoria antes
- Las colecciones resultantes de las queries SI que se cargan en memoria

Un ejemplo que no funcionaría

```
public class ListaMensajes {

   public static void main(String[] args) {
      EntityManagerFactory emf = Persistence
            .createEntityManagerFactory(“mensajes-mysql");
      EntityManager em = emf.createEntityManager();
      em.getTransaction().begin();
      System.out.println("--Listando mensajes de un usuario");
      String correo = leerTexto("Introduce correo identificador de usuario: ");
      Autor autor = em
            .find(Autor.class, correo);
      if (autor == null) {
         System.out.println("Usuario no existente");
      } else {
         System.out.println("Usuario encontrado");
         System.out.println("Nombre: " + autor.getNombre());
         // Obtenemos los mensajes asociados al autor
         Set<Mensaje> mensajes = autor.getMensajes();
      }
      em.getTransaction.commit();
      em.close;

      // La colección de mensajes se ha cargado de forma perezosa
      // y ahora está desconectada. Lo siguiente dará un error
      for (Mensaje mensaje : mensajes) {
         System.out.println(mensaje.getId() + " " mensaje.getTexto() + " "
                              + mensaje.getFecha());
      }
```

Una solución

```
public class ListaMensajes {

     ...
         System.out.println("Usuario encontrado");
         System.out.println("Nombre: " + autor.getNombre());
         // Obtenemos los mensajes asociados al autor
         Set<Mensaje> mensajes = autor.getMensajes();
         // Realizamos una operación que obliga a cargar los mensajes
         // en memoria
         mensajes.size()
      }
      em.getTransaction.commit();
      em.close;

      // Ahora sí tenemos los mensajes en memoria y funciona lo siguiente
      for (MensajeEntity mensaje : mensajes) {
         System.out.println(mensaje.getId() + " " mensaje.getTexto() + " "
                              + mensaje.getFecha());
      }
```

Las relaciones a-uno y los resultados de las queries sí se cargan en memoria

```
public class BuscaMensajes {

   @SuppressWarnings("unchecked")
   public static void main(String[] args) {
      EntityManagerFactory emf = Persistence
            .createEntityManagerFactory(“mensajes-mysql");
      EntityManager em = emf.createEntityManager();
      em.getTransaction().begin();
      System.out.println("--Buscando en los mensajes");
      String palabra = leerTexto("Introduce una palabra: ");

      System.out.println("Mensajes con la palabra: " + palabra);
      String patron = "%"+palabra+"%";
      List<Mensaje> mensajes = em.createNamedQuery(
            "mensajesConPatron").setParameter("patron", patron).getResultList();
      em.getTransaction().commit();
      em.close();

      // Hemos cerrado el entity manager pero las entidades y las relaciones
      // a uno (el autor del mensaje) se han cargado en memoria
      for (MensajeEntity mensaje : mensajes) {
         System.out.println(mensaje.getTexto() + " -- " +
               mensaje.getAutor().getNombre());
      }
}
```

Otra solución mejor: hacer la colección EAGER

```
public class Empleado {
    @Id
    @GeneratedValue
    private Long id;
    ...

    @OneToMany(mappedBy = "empleado", fetch = FetchType.EAGER)
    private Set<CuentaCorreo> correos;
   ...
}
```

```
// Cargamos una entidad y cerramos la conexión
// del entity manager
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();
Empleado empleado = em.find(Empleado.class, 1L);
em.getTransaction().commit();
em.close();

// Comprobamos que los correos se han cargado en memoria
// por ser el tipo Eager Fetch

assertTrue(empleado.getCorreos().size()==2);
```

JPA no actualiza las relaciones en memoria

- El siguiente código daría un resultado incorrecto

La colección se carga en memoria

```
em.getTransaction().begin();
Autor autor = em.find(AutorEntity.class, 1L);
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");
```

Como ya se ha cargado en memoria el segundo acceso a **`getMensajes()`** no se recupera de la DB y devolvería el mismo tamaño las dos veces

Solución: actualizar las colecciones a mano

Actualizamos a mano la colección, añadiendo el nuevo mensaje

```
em.getTransaction().begin();
Autor autor = em.find(Autor.class, 1L);
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(mens);
System.out.println(autor.getNombre() + " ha escrito " +
                   autor.getMensajes().size() + " mensajes");
```

Data Access Object con JPA

- El patrón DAO permite ocultar las complejidades de la capa de datos
- Hay quien argumenta que JPA ya es lo suficientemente de alto nivel como para necesitar un DAO. Depende del nivel de formación del equipo de desarrollo.
- Proporcionamos un DAO adaptado a JPA por si resulta de interés su utilización.
- Ventajas del DAO:
  - Unifica el API de acceso a las entidades, mejorando el API de JPA (por ejemplo las funciones persist y merge de JPA funcionan de distinta forma: una devuelve la entidad y la otra no)
  - Agrupa las consultas y las ofrece como métodos con parámetros
  - Proporciona un API muy sencillo y estándar para todo el equipo
- Características de la propuesta:
  - Se define una clase DAO para cada una de las entidades
  - Trabajan con entidades gestionadas y proporcionarán las operaciones básicas CRUD sobre la entidad (Create, Read, Update y Delete), así como las consultas JPQL.
  - Trabajan 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.
  - Se define una clase genérica de la que heredan todos los DAOs específicos
  - Incluimos todos los DAOs en el package “persistencia”

Clase DAO genérica

```
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) {
      em.persist(t);
      em.flush();
      em.refresh(t);
      return t;
   }
```

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

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

Clase DAO específica: EmpleadoDao

```
public class EmpleadoDao extends Dao<Empleado, Long> {
    String FIND_ALL_EMPLEADOS = "SELECT e FROM Empleado e ";

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

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

    public List<Empleado> listAllEmpleados() {
        EntityManager em = this.getEntityManager();
        Query query = em.createQuery(FIND_ALL_EMPLEADOS);
        return (List<Empleado>) query.getResultList();
    }
}
```

Capa de negocio

- Define los métodos de negocio: funciones que aceptan objetos planos como parámetros (no entidades) y devuelven otros objetos como resultado
- Cada método de negocio tiene la misma estructura:
  1. Se abre un entity manager y una transacción
  2. Se crean los DAO pasando como parámetro el entity manager
  3. Se llaman a las funciones de los DAO para realizar la lógica de negocio con las entidades gestionadas
  4. Se cierra la transacción
  5. Se cierra el entity manager
- Una llamada a un método de negocio realiza una operación atómica. Cuando termina estamos seguros de que se han actualizado los cambios en la BD. Si hay algún error, el método devuelve una excepción y la operación no se realiza en absoluto.
- Convenciones:
  - Los métodos de negocio se agrupan en clases con el sufijo “servicio”
  - Definimos las clases dentro del package “servicio”

Clase EmpleadoServicio

```java
public class EmpleadoServicio {

    private EntityManagerFactory emf;

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

    public Empleado findPorId(Long idEmpleado) {
        EntityManager em = emf.createEntityManager();
        Empleado empleado = null;
        EmpleadoDao empleadoDao = new EmpleadoDao(em);
        try {
            em.getTransaction().begin();
            empleado = empleadoDao.find(idEmpleado);
            em.getTransaction().commit();
        } catch (Exception e){
            e.printStackTrace();
        } finally {
            em.close();
        }
        return empleado;
    }
```

```java
    public void intercambiaDespachos(Long idEmpleado1,
                                     Long idEmpleado2) {
        EntityManager em = emf.createEntityManager();
        EmpleadoDao empleadoDao = new EmpleadoDao(em);
        try {
            em.getTransaction().begin();
            Empleado emp1 = empleadoDao.find(idEmpleado1);
            Empleado emp2 = empleadoDao.find(idEmpleado2);
            Despacho despachoAux = emp1.getDespacho();
            emp1.setDespacho(emp2.getDespacho());
            emp2.setDespacho(despachoAux);
            empleadoDao.update(emp1);
            empleadoDao.update(emp2);
            em.getTransaction().commit();
        } catch (Exception e){
            e.printStackTrace();
        } finally {
            em.close();
        }
    }
}
```

¿Preguntas?