Documentación de JPA

Alejandro León Marín

May 12, 2024

Introducción

En este documento se daran las bases de la documentación de la JPA, explicada por el master Carlos Carranza Blanco. El cual explica JPA y JPQL en las clases de Programación II, como bases para el manejo de las bases de datos en Java. Las cuales serán usadas durante el curso.

Persistencia

En Java manipulamos objetos los cuales normalmente son instacias de clases, dentro de estos objetos podemos tener referencias a otros objetos o a colecciones de los mismos objetos. Podemos llegar a referenciarlos de manera recursiva como lo podemos ver en los singleton, que podemos declarar una variable del mismo tipo de clase.

Estos objetos encapsulan datos y comportamientos, y esto porque almacenan los atributos y los metodos de las clases. El problema es qyue este estado es accesible solo mientras la JVM (Java Virtual Machine) este en ejecución.

Ahora si la JVM se detiene o si el garbage collector destruye el objeto el estado desaparece. Y algunos objetos necesitan que sus datos perduren la vida de la JVM, es decir que perduren más allá de la ejecución de la JVM o del aplicativo.

Un objeto puede almacenar su estado mediante un nedio permanente y durable,s e dice que es un objeto persistente. En las aplicaciones empresiarales, el estado persistente sule almacenarse en base de datos relacionales. Existen diversos mecanismos para lograr esto

- Utilizar mecanisos manuales basados en JDBC
- Realizar mapeo automático entre instacias de clase y tuplas de una base de datos, a través de un ORM(Object Relational Mapping)

JPA es la especificacion de una solucon de ORM para la plataforma Java EE. Y para poder realizar el mapeo y todo el manejo de persistencia se estara usando JPA(Java Persistence API).

Entidades

Cuando hablamos de un objeto que se mapea a una table de una base de datos relacional, utilizamos el termino entidad en lugar de objeto. Es decir es un objeto que se mapea una tabla de base de datos con una serie de anotaciones a una base de datos especifica. Los objetos son instacias que solo viven en memoria. Las entidades son objetos que viven corto tiempo en memoria mas sin embargo persisten en la base de datos.

Las entidades tienen la habilidad de ser mapeadas a la base de datos, estas pueden ser concretas o abstractas. tambien pueden soportar herencia y relaciones.

Una vez que una entidad sea mapeada a la base de datos, la misma puede

ser gestionada por JPA. Esto significa que podemos persistir, eliminar y consultar una entidad de la base de datos.

En el mundo de JPA, es una POJO (Plain Old Java Object), solo que en realidad es un objeto de java que tiene metadatos y algunas anotaciones, estas manejan un estado el cual puede ser accesidiodos por getters y modificados por setters, y el estado de la entidad esta representado por los valores de los atributos de la misma.

Aqui un ejemplo de un objeto

```
public class Book{
    private Long id;

    private String title;
    private String price;
    private String description;
    private String isbn;
    private String nbOfPage;
    private boolean illustrations;

    public Book(){}

    //Getters and Setters
}
```

Y aquí tenemos un ejemplo de la entidad Book

```
@Entity
public class Book{
    @Id@GeneratedValue
    private Long id;

    private String title;
    private String price;
    private String description;
    private String isbn;
    private String nbOfPage;
    private boolean illustrations;

    public Book(){}

    //Getters and Setters
}
```

Notesé de las anotaciones el @Entity indica que esa clase la vamos a usar como una entidad y que va a estar mapeada a una clase en especifico, el @Id

que indica que es el atributo clave o la llave principal de la tabla y el @GeneratedValue que indica que el valor de la llave principal se va a generar automaticamente. Todas las entidades estan anotadas con @javafx.persistence.Entity .
Debemos marcar un atributo como la clave primaria usando la anotacion javafx.persistence.Id .
Se debe de tener un construtor por defecto ya sea public o protected y debe ser una clase clase top-level es decir que una sea una interfaz o un enumerado.

La clase no puede estar marcada como final. A la misma vez que ningún metodo o atributo puede estar marcado como final. Si la entidad deber ser pasada como parámetro a través una interfaz remota, entonces debe implementar la interfaz java.io.Serializable .

A través del uso de metadatos (XML o anotaciones) JPA puede mapear una entidad a una tabla de base de datos. Un proovedor de persistencia (implemtacion de JPA) es el encargado de usar estos metadatos para sincronizar el estado entre los atributos de la entidad y la tabla.

Una vez que tenemos las entidades mapeadas a la base de datos, podemos utilizar JPA para acceder a las entidades. Podemos consultar entidades y sus relaciones, sin necesitar conocer la estructura de base subyacente. El elemento central de JPA que permite realizar esto, es el API: javax.persistence.EntityManager .

Entity Manager

El EntityManager es la pieza central de JPA y su rol es permitir dentro un contexto de persistencia lo siguiente:

- Gestionar entidades
- Leer y escribir de una base de datos
- Permitir operaciones CRUD simples
- Localizar entidades por su clave primaria
- Permitir lockear el acceso a en(esto en forma pesismista o optimista)
- Permite crear y ejecutar que en JPQL o usando el API de Criteria

Tambien funciona como uan interfaz cuya implemtacion la brinda un proovedor de persistencia. Ahora bien, mientar una entidad no entre en contacto con

una instancia de un EntityManager, se dice que la entidad no esta siendo administrada, pero en cuanto entre en contacto con el EntityManager (creacion, querie, modificaion,etc.) ahi se dice que la entidad paso un estado MANAGED. Y en este estado el EntityManager se encarga de sincronizar los cambios de la entidad con la base de datos automaticamente.

Existen dos tipos de EntityManager:

- Application Managed EntityManager
- Container Managed EntityManager

El primer caso se da cuando usamos el EntityManager fuerea de un sevidor de aplicaciones y el segundo caso se da cuando estamos dentro de un servidor, como GlassFish.

Por ejemplo, podemos usar el EntityManager de esta forma :

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory
EntityManager em = emf.createEntityManager();
em.persist(book);
```

Extension del EntityManager a la entidad Book

```
@Entity
@NameQuery(name = "findBookH2G2", query = "SELECT b FROM Book
b WHERE b.title = 'H2G2'")

public class Book{
    @Id@GeneratedValue
    private Long id;

    private String title;
    private String price;
    private String description;
    private String isbn;
    private String nbOfPage;
    private boolean illustrations;

    //Construtors ,getters, setters
}
```

Main de la App

```
public class Main{
    public static void main(String[] args){
```

```
        Book book = new Book();
        book.setTitle("H2G2");
        book.setPrice(12.5F);
        book.setDescription("The Hitchhiker's Guide to the Galaxy");
        book.setIsbn("1-84023-742-2");
        book.setNbOfPage(354);
        book.setIllustrations(false);

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

        EntityTransaction tx = em.getTransaction();
        tx.begin();
        em.persist(book);
        tx.commit();

        Query query = em.createNamedQuery("findBookH2G2", Book.class);
        book = query.getSingleResult();

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

Persistence Context

Es un conjunto de entidades que administradas(Managed), en un momento dado y para una transacción de un usuario dado. Solo una instacia de una entidad, con la misma entidad persistente(PK) puede existir en un persistence context. Solo las entidades contenidas en un persistence context son manejadas por el EntityManager.

El EntityManager actualiza o consulta el contexto de persistencia cuando el método del API de EntityManager es invocado, por ejemplo cuando se invoca el metodo persist(), la entidad pasada como parámetro se agrega al contexto de persistencia, si no existe en el mismo. Cuando buscamos una entidad por ID, primero verificamos si no esta cargada en el contexto de persistencia.

Por estos motivos, el contexto de persistencia se denomida tambien como "cache de primer nivel". Y por defecto los objetos solo viven en el contexto de persistencia, solo mientras dure la transacción actual.

Unidad de Persistencia

Algo que nos falta en los códigos anteriormente mostrados es cómo nos conectamos a la base de datos, a qué base de datos nos vamos a conectar, a qué servidor, con qué usuario nos vamos a conectar y qué driver de JDBC vamos a usar. Toda esta información está almacenada en un elemento llamado Unidad de Persistencia.

Cuando creamos un EntityManagerFactory, le pasamos como parámetro el nombre de la Unidad de Persistencia. La Unidad de Persistencia es un conjunto de información que se encuentra en un archivo llamado persistence.xml. En este archvo se encuentra la información para establecer la conexión con la base de datos. Este archivo se encuentra en la carpeta META-INF de nuestro proyecto. Este es el archivo que permite configruar el contexto de persistencia. Viene dado por el archivo persistence.xml . Este se encuentra en el META-INF. Permite establecer diferentes tipos de parámetros para configurar el contexto de persistencia creado. Y se puede utilizar propiedades estándar y/o propiedades del proovedor de persistencia como el Hibernate.

Bean Validation

Además de la validacion manual, es posible utilizar Bean Validation automaticamente con una entidad JPA. Las constraints pueden ser aplicadas en la clases de entidad, clases embeddable y mapped superclases. La validacion ocurre automaticamente siempre luego de los eventos

- Prepersist()
- PreUpdate()
- PreRemove()

```
@Entity
public class Book{
    @Id@GeneratedValue
    private Long id;

    @NotNull
    private String title;
    private String price;

    @Size(min = 10, max = 2000)
    private String description;
    private String isbn;
    private String nbOfPage;
    private boolean illustrations;
```

```
    public Book(){}

    //Getters and Setters
}
```

ORM

Para establecer el vinculo entre las entidades en el mundo Java y los elementos a nivel realcion, debemos definir las anotaciones de ORM para la entidad. En JPA utilizamos una configuracion por execpción. A menos que especifiquemos lo contrario, se asumen los valores por defecto. En ciertas situaciones esto no nos sirve como por ejemplo para poder acceder a un esquema legado

Las anotaciones de ORM son las siguientes:

@Table

Poder defecto el nombre de la tabla y la entidad coinciden. Para configurar esto, debemos usar la anotacion @javax.persistence.Table . El elemento mas basico que podemos cambiar es el nombre de la table

```
@Table(name = "t_book").
```

Podemos especificar el schema y el catalog de la tabla.

```
@Entity
@Table(name = "t_book")
public class Book{
    @Id
    private Long id;
    private String title;
    private String price;
    private String description;
    private String isbn;
    private String nbOfPage;
    private boolean illustrations;

    public Book(){}

    //Getters and Setters
}
```

Primary Keys

Esta identifican unívocamente una tupla en una tabla de la base de datos, esta puede ser una columna o un conjunto de columnas. JPA requiere que

las entidades tengan un identificador mapeado a una clave primaria. Una vez actualizado en la base de datos, el valor de una clave primaria no puede ser actualizado.

@Id

El @Id es la anotacion que marca un atributo simple como la clave primaria de la entidad. Pueden ser de los siguientes tipos:

- Tipos primitivos : byte, int, short, long, char
- Tipos Wrapper: Byte, Integer, Short, Long, Character
- String , numero y fechas: String, BigInteger, durante
- Arrays de primitivos/wrappers: int[], Integer[]

@GeneratedValue

El valor de la clave primaria pued ser genrado de forma manual o automaticamente. Para poder realizar esta generacion usamos la anotacion @GeneratedValue. La cual puede tomar cuatro posibles valores:

- AUTO
- IDENTITY
- SEQUENCE
- TABLE

El valor asignado por defecto es AUTO.

```
@Entity
@Table(name = "t_book")
public class Book{
    @Id@GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String title;
    private String price;
    private String description;
    private String isbn;
    private String nbOfPage;
    private boolean illustrations;

    public Book(){}

    //Getters and Setters
}
```

Campos Identity

La base debe de soportar columnas de tipo autogeneradas, como SQL Server y MySQL. Debemos hacer

```
@Entity
public class Inventory implements Serializable{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
}
```

Estrategia por defecto

Cuando especificamos que la estrategia es AUTO, estamos indicando que el proveedor de persistencia puede seleccionar la estrategia que crea conveniente. Generalmente la selección es TABLE, ya que es el mecanismo garantizado para cualquier manejador de base de datos. En el caso de utilizar Hibernate como proveedor de persistencia, el nombre de la tabla por defecto, es "hibernate_sequences"

Primary Key compuestas

Cuando mapeados un entidad es una buena practica usar una unica columna como clave, y que esta columna sea de im tipo integral (INTEGER,LONG, etc). Pero aveces esto no es posible porque si tengo que aherir la clave a ciertas reglas de negocio, o si tengo que mapear un esquema legado, Entonces en estos casos tenemos dos posibles soluciones:

- @EmbeddedId
- @IdClass

@Embedded

La anotacion @Embedded se usa para representar objetos que estan embebidos en otros objetos. Un onjeto embebido no tiene identidad esto quiere decir que no PK por si mismo. Sus atributos terminaran formando parte delas columnas que contiene el objeto embebido. Un Objeto embebido debe de :

- Tener un constructor sin parametros
- Debe de etenre setters y getters para los atributos allí presentes
- Debe de definir el metodo equals y el metodo hashcode()
- La clase no tiene una identificación propia, esto es que no tiene tiene atributos marcados como @Id

Un ejemplo :

```
@Embeddable
public class NewsId{
    private String title;
    private String language;

    //constructors
    //Getters and Setters
    // equals and hashcode
}
```

@EmbeddedId

Esta notacion se utliza para cuando representamos los campos de una clave compuesta de un @Embedded. En este caso no necesitamos la clave primaria como @Id

```
@Entity
public class News{
    @EmbeddedId
    private NewsId id;
    private String content;
}
```

En este caso, la clave primaria no es un atributo simple, sino que esta representada por una instancia del objeto embebido. Y para poder usarla debemos de hacer lo siguiente:

```
NewsId pk = new NewsId(
    "Richard Wright has died on September 2008",
    "EN");
    News news = em.find(News.class, pk);
```

@IdClass

Esta notacion es parecida al mecanisno anterior, pero los atributos que componene la clave primaria, deben de especificarse en la clase utilizando @Id pero por separado

```
@Entity
@IdClass(NewsId.class)
public class News{
    @Id
    private String title;
    @Id
    private String language;
    private String content;
```

```
    //constructors, setters and getters
}
```

Atributos

Los atributos componene el estado de la entidad, entre los atributos podemos mapear los siguientes tipos :

- Tipos primitivos y sus wrappers
- String , tipos numericos y tipos temporales
- Array de bytes y chars
- Enumerados
- Tipos que implementat Serializable
- Colecciones de tipo basicos y embeddables

Una entidad tambien puede tener atributos de tipo entidad, colecciones de entidades o clases embebidas. Este tipo de atributos requieren del uso de relaciones entre entidades. Como con los nombres de las tablas, se usa una configuracion por excepción.

@Basic

Es la notacion mas simple para mapear un coampo a la base de datos. Peromite hacer un override a la forma en como se levantan los datos a la base

```
@TARGET({METHOD, FIELD}) @RETENTION(RUNTIME)
public @interface Basic{
    FetchType fetch() default EAGER;
    boolean optional() default true;
}
```

- Optional

  Permite indicar si el atributo puede ser nulo o no.

  No aplica para los tipos de datos primitivos

- fetch

  Puede tomar dos valores : EAGER o LAZY

  Le indica al proovedor cuando debe de cargar el dato en cuestion

  EAGER : Carga el atributo cuando se carga la entidad

  LAZY : Carga el atributo cuando se accede a el

```
@Entity
public class Track{
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String title;
    private Float duration;

    @Basic(fetch = FetchType.LAZY)
    @Lob
    private byte[] wav;
    private String description;

    //constructors, getters and setters
}
```

@Column

Esta anotacion se usa para mapear un atributo a una columna de la base de datos. Permite hacer un override a la forma en como se mapea el atributo a la base de datos.

```
@TARGET({METHOD, FIELD} @ RETENTION(RUNTIME)
public @interface Column{
    String name() default "";
    boolean unique() default false;
    boolean nullable() default true;
    boolean insertable() default true;
    boolean updatable() default true;
    String columnDefinition() default "";
    String table() default "";
    int length() default 255;
    int precision() default 0; //decimal precision
    int scale() default 0; //decimal scale
}

@Entity
public class Book{
    @Id@GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @Column(name = "book_title", nullable = false, updatable = false)
    private String title;
    private Float price;
    @Column(length = 2000)
    private String description;
    private String isbn;
```

```
    @Column(name = "nb_of_page" , nullable = false)
    private String nbOfPage;
    private boolean illustrations;

    //constructors, getters and setters
}
```

@Temporal

Los tipos de datos Date(java.sql y java.util) pueden ser llevados a la base de diferentes representaciones. Para poder controlar esto usamos la anotacion @Temporal que puede tomar tres valores:

- DATE: almacena la fecha
- TIME: almacena la hora
- TIMESTAMP: almacena la fecha y la hora completa

```
@Entity
public class Empleados{
    @Id @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String nombre;
    private String apellido;
    private String email;
    private String telefono;
    @Temporal(TemporalType.DATE)
    private Date fechaDeNacimiento;
    @Temporal(TemporalType.TIMESTAMP)
    private Date fechaDeContratacion;
}
```

@Transient

En JPA, cuando anotamos la clase con @Entity, esto hace que todos sus campos formen parte del estado persistente. Si no queremos persistir un atributo a la entidad, podemos colocarle la anotacion @Transient. Esto indica que el atributo no sera persistido en la base de datos. Y colocarle el modificador de accesso transient al atributo.

```
@Entity
public class Empleado{
    @Id@GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String nombre;
    private String apellido;
```

```
    private String email;
    private String telefono;
    @Temporal(TemporalType.DATE)
    private Date fechaDeNacimiento;
    @Transient
    private Integer age;
    @Temporal(TemporalType.TIMESTAMP)
    private Date fechaDeContratacion;
}
```

Relaciones

Las relaciones entre entidades son una parte fundamental de JPA. Las relaciones entre entidades se pueden representar de dos formas:

- Relaciones unidireccionales
- Relaciones bidireccionales

Las relaciones tiene cardinalidad, esto se refiere a la cantidad de participantes en cada extremo de la relacion. Las relaciones puede ser establecidas con columnas de join(Foreing Key). O tambien pueden ser establecidas por tablas de mapeo.

Ahora veremos los diferentes tipos de relaciones que podemos tener en JPA.

- @OneToOne

  En estos casos, se tiene una referencia a un solo elemento de la otra entidad.

```
@TARGET({METHOD, FIELD} @ RETENTION(RUNTIME)
public @interface OneToOne{
    Class targetEntity() default void.class;
    CascadeType[] cascade() default {};
    FetchType fetch() default EAGER;
    boolean optional() default true;
    String mappedBy() default "";
    boolean orphanRemoval() default false;
}
```

  @JoinColumn

  Es similar a @Column, pero se usa para customizar la columna de foreing key.

  Siempre debe utilizarse en el lado del "dueño" de la relacion. Es decir el que no contiene mappedBy.

```
@Entity
public class Employee{
```

```
    @Id@GeneratedValue
    private Long id;
    private String name;
    private String lasName;
    private String email;
    private String phone;
    @OneToOne
    @JoinColumn(name = "address_fk")
    private Address address;
}

@Entity
public class Address{
    @Id@GeneratedValue
    private Long id;
    private String street1;
    private String street2;
    private String city;
    private String state;
    private String zipCode;
    private String country;
    @OneToOne(mappedBy = "address")
    private Employee employee;
}
```

- @OneToMany unidireccional

  En este caso, el objeto origen mantiene una colección de objetos destino. En este tipo de relaciones de JPA, se generan tablas intermedias de mapeo entre entidades.

  @JoinTable

  Podemos usar esta anotacion para customizar la table de mapeo entre ambas entidades.

```
@TARGET({METHOD, FIELD} @ RETENTION(RUNTIME)
public @interface JoinTable{
    String name() default "";
    String catalog() default "";
    String schema() default "";
    JoinColumn[] joinColumns() default {};
    JoinColumn[] inverseJoinColumns() default {};
    UniqueConstraint[] uniqueConstraints() default {};
    Index[] indexes() default {};
}

@Entity
```

```
public class Order{
    @Id@GeneratedValue
    private Long id;
    @Temporal(TemporalType.TIMESTAMP)
    private Date creationDate;
    @OneToMany
    @JoinTable(name = "order_item",
    joinColumns = @JoinColumn(name = "order_fk"),
    inverseJoinColumns = @JoinColumn(name = "order_line_fk"))
    private List<OrderLine> orderLines;
}
```

La tabla obtenida, en este caso caambia a lo siguiente:

```
CREATE TABLE JOIN_ORD_LINE(
    ORDER_FK BIGINT not null,
    ORDER_LINE_FK BIGINT not null,
    primary key (ORDER_FK, ORDER_LINE_FK),
    foreing key (ORDER_LINE_FK) references ORDER_LINE(ID),
    foreing key (ORDER_FK) references ORDER(ID)
);
```

Para relaciones de este estilo la accion por defecto es usar una tabla de mapeo, sin embargo es posible modificar esto para utilizar una columna de join en la tabla subordinada. Basta indicar en la entidad padre que vamos a utilizar una @JoinColumn en lugar de @JoinTable.

```
@Entity
public class Order{
    @Id@GeneratedValue
    private Long id;
    @Temporal(TemporalType.TIMESTAMP)
    private Date creationDate;
    @OneToMany(fecth=FetchType.EAGER)
    @JoinColumn(name = "order_fk")
    private List<OrderLine> orderLines;
}
```

- @OneToMany bidireccional

  Para transformar esta realcion en una bidireccional, en realidad debemos hacer que la lunea de la orden referencia a la orden. Para esto, agregamos una referencia a la orden y utlizamos la anotacion @ManyToOne. Es importante en este caso. que la anotacion tenga el atributo "mappedBy" para indicar que es la inversa de la anterior.

```
@Entity
public class Artist{
```

```
    @Id@GeneratedValue
    private Long id;
    private String name;
    private String lastName;
    @OneToMany
    @JoinTable(name = "jnd_art_cd",
    joinColumns = @JoinColumn(name = "artist_fk"),
    inverseJoinColumns = @JoinColumn(name = "cd_fk"))
    private List<CD> cds;

    //constructors, getters and setters
}
```

Ahora un ejemplo de la clase @ManyToMany bidireccional

```
@Entity
public class CD{
    @Id@GeneratedValue
    private Long id;
    private String title;
    private Float price;
    private String description;
    private String musicCompany;
    @ManyToMany(mappedBy = "cds")
    private List<Artist> artists;

    //constructors, getters and setters
}
```

Fecthing

Todas la anotaciones anteriores definin un mecanis de fecthing(recuperacion de datos). Los objetos asociados puede ser cargados de dos formas:

- Inmediamente: FetchType.EAGER
- Cuando se requiera: FetchType.LAZY

Por ejemplo en caso donde necesitamos levantar todos los datos de las entidades por ejemplo en la busqueda por ID, si los cargamos todos de una vez, el impacto en el performance es enorme. O si si hacemos esto :

```
class1.getClass2().getClass3().getClass4()
```

Entonces en ese momento se can a cargar los datos de las entidades relacionadas. Aqui una tabla de los valores por defecto de FetchType segun la relacion:

Anotaciones	Estrategia de fetching por defecto
`@OneToOne`	EAGER
`@ManyToOne`	EAGER
`@OneToMany`	LAZY
`@ManyToMany`	LAZY

```
@Entity
public class Order{
    @Id@GeneratedValue
    private Long id;
    @Temporal(TemporalType.TIMESTAMP)
    private Date creationDate;
    @OneToMany(fetch = FetchType.EAGER)
    private List<OrderLine> orderLines;

    //constructors, getters and setters
}
```

Persistiendo una entidad

Si los datos no existen en la base datos, los datos van a ser insertados, de los contraio se propaga una excepcion llamada EntityExistException.

```
Costumer costumer = new Costumer("Anthony",
"Giddens", "tgiddens@mail.com);
Address address = new Address("Ritherdon Rd", "London",
"8QE", "UK");
costumer.setAddress(address);
tx.begin();
em.persist(costumer);
em.persist(address);
tx.commit();
```

Buscando por ID

Para realizar la busqueda por id, usamos el metodo find()

```
Costumer costumer = em.find(Costumer.class, 1L);
if(costumer != null){
    //procesar objeto
}
```

Lo que va a hacer este metodo es buscar en la base de datos, el id, si existe lo retorno y no si no exite retorna un null. Otra forma de de hacerlo es usando el metodo getReference(), el cual levanta los datos de la entidad en forma LAZY.

```
try{
    Costumer costumer = em.getReference(Costumer.class, 1L);
    //procesar objeto

}catch(EntityNotFoundException e){
    //Entity no encontrada
}
```

En el caso de getReference, si la entidad no existe, se genera la excepcion EntityNotFoundException. Los datos de la entidad se recuperan en forma LAZY, por lo que debe de hacerse dentro del contexto de persistencia. Y su la entidad se devuelde detached, ya no podemos recuperar los datos, generando una LazyInitializationException.

Borrando una entidad

Para borrar una entidad usamos el metodo remove(). Al borrar una entidad esta misma se elimina de la base de datos, se desvincula del persistence context, el estado pasa a ser detached y no puede volver a ser sincronizada. Sin embargo el objeto aun es accesible desde Java ya que es un POJO tradicional.

```
Customer customer = new Customer ("Anthony",
"Giddens", "tgiddens@mail.com);
Address address = new Address("Ritherdon Rd", "London",
"8QE", "UK");

customer.setAddress(address);
tx.begin();
em.persist(customer);
em.persist(address);
tx.commit();

tx.begin();
em.remove(customer);
tx.commit();

// ESTO ES true

assertNotNull(customer);
```

Sincronizacion a la base de datos

Para sincronizar los cambios hechos a la base de datos, usamos el EntityManager, recordemos que el EntityManager funciona como un cache de primer nivel, por lo tanto todas las transacciones a las que se commitearon estar esperando en ese cache de primer nivel a ser flusheados, a la base de datos. Por lo tanto cuando

realizamos multiples operaciones a un contexto de persistencia, por ejemplo persistir dos entidades, las sentencias se hacen permanentes cuando se commitean los cambios

Flushing

El metodo flush(), puede hacer que los datos sean enviados a la base, pero aun no se commitee la transaccion. El problema que podemos tener al flushear manualmente es que podemos ejecutar senticas que pueden violar una restriccion de integridad. Por ejemplo :

```
tx.begin();
em.persist(customer);
em.flush();
em.persist(address);
tx.commit();
```

Refresh

Este metodo es unsado para sincronizar datos, pero direccion opuesta al flush. Esta operacion sobre escribe el estado persistente de la entidad en el cache de primer vicel. Es util para cuando queremos deshacer cambios que hicimos en memoria.

```
Customer costumer = em.find(Customer.class, 1L);
assertEquals(costumer.getFirstName() , "Anthony");
customer.setFirstName("John");
em.refresh(customer);
assertEquals(costumer.getFirstName() , "Anthony");
```

Contains

El metodo contains() retorna un booleando(true o false), indicando su en el contexto de persistemcia actual, una entidad esta siendo managed.

```
Customer customer = new Customer("Anthony", "Giddens",
"tgiddens@mail.com);

tx.begin();
em.persist(customer);
tx.commit();

assertTrue(em.contains(customer));

tx.begin();
em.remove(customer);
tx.commit();
```

```
assertFalse(em.contains(customer));
```

Clear y Detach

El metodo clear() limpia el contexto ded persistencia, por lo tanto todas las entidades que estaban siendo managed pasan automaticamente a un estado detached. Y el metodo detached() recibe una entidad y la desconecta del contexto de persistencia. Cualquier otro cambio que ea haga sobre la misma no sera sincronizada contra la base de datos.

```
Customer customer = new Customer("Anthony", "Giddens",
"tgiddens@mail.com);

tx.begin();
em.persist(customer);
tx.commit();

assertTrue(em.contains(customer));

em.detached(customer);

assertFalse(em.contains(customer));
```

Merging

Para poder asociar una entidad que esta deconectad de un contexto de persistencia debemos de re-attachearla es decir mergearla. Esta situacion es comun cuando

- Una entidad es devuelta por componente de negocio a presentacion
- Se le hace cambios en presentacion
- Es enviada al componente de nogocio para ser actualizada en la base

```
Customer customer = new Customer("Anthony", "Giddens",
"tgiddens@mail.com);

tx.begin();
em.persist(customer);
tx.commit();

em.clear();

//Setteamos el valor a la entidad detached
customer.setFirstName("John");
```

```
tx.begin();
em.merge(customer);
tx.commit();
```

Actualizacion de una entidad

Para poder actualizar un entidad tenemos dos formas de hacerlo:

- La anterior, en la cual la entidad detached , es mergeada al contexto de persistencia actual.
- Pero si la entidad ya esta siendo managed, los cambios se efectuaran automaticamente sin necesidad de mergearla explicitamente.

```
Customer customer = new Customer("Anthony", "Giddens",
"tgiddens@mail.com);

tx.begin();
em.persist(customer);

customer.setFirstName("John");

tx.commit();
```

Cascading

Existen situaciones en las cuales las operaciones aplicadas sobre una entidad, deben ser programadas a las entidades relacionadas. A esto se le denomida "Cascade de eventos". Por ejemplo si el Customer y Address estan vinculados podemos hacer esto:

```
Customer customer = new Customer("Anthony", "Giddens",
"tgiddens@mail.com);
Address address = new Address("Ritherdon Rd", "London",
"8QE", "UK");

customer.setAddress(address);

tx.begin();
em.persist(customer);
em.persist(address);
tx.commit();
```

En este caso, guardamos ambos objetos explícitamente

```
Customer customer = new Customer("Anthony", "Giddens",
"tgiddens@mail.com);
Address address = new Address("Ritherdon Rd", "London",
"8QE", "UK");

customer.setAddress(address);

tx.begin();
em.persist(customer);
tx.commit();
```

En este caso, usamos cascade al persistir

```
@Entity
public class Customer{
    @Id@GeneratedValue
    private Long id;
    private String firstName;
    private String lastName;
    private String email;
    @OneToOne( fetch = FetchType.LAZY
        cascade = {CascadeType.PERSIST,
        CascadeType.REMOVE})
    @JoinColumn(name = "address_fk")
    private Address address;

    //constructors, getters and setters
}
```

Cascading - Eventos

Evento	Descripcion
PERSIST	Se persiste la entidad y se propaga a las entidades relacionadas
MERGE	Se mergea la entidad y se propaga a las entidades relacionadas
REMOVE	Se remueve la entidad y se propaga a las entidades relacionadas
REFRESH	Se refresca la entidad y se propaga a las entidades relacionadas
DETACH	Se desconecta la entidad y se propaga a las entidades relacionadas
ALL	Se propaga a todas las operaciones

JPQL

Es un lenguaje de consulta usado para definir busquedas contra laas entidades, independientemente del motor de base de datos utlizado. JPQL es muy similar a SQL, pero tiene esto de diferencia

- En vez de tablas, usamos clases
- En vez de columnas, usamos atributos
- En vez de joins por FKs, usamos navegacion

el ejemplo mas basico de JPQL es el siguiente:

```
Select b FROM Book b
```

Donde Book representa la entidad y usamos la b la cual representa un alis utilizado en la consulta.

Un ejemplo general de JPQL es el siguiente:

```
SELECT <select clause>
FROM <from clause>
 [WHERE <where clause>]
 [ORDER BY <order by clause>]
 [GROUP BY <group by clause>]
 [HAVING <having clause>]
```

Select clause

Esta clausula permite seleccionar los resultados de la consulta. Permite devolver lo siguiente:

- Una entidad
- Un atributo de una entidad

- Una "constructor expression"
- Una funcion agregada
- Una expresion de navegacion(usando el ".")

La forma mas general es la siguiente:

```
SELECT [DISTINCT] <select expression> [[AS] <identification variable>
expression::= {NEW|TREAT|AVG|MAX|MIN|SUM|COUNT}
```

Por ejemplo, si el Customer es una entidad, a la cual se le da el alias "c", entonces estos son unos ejemplos simples de seleccion:

```
SELECT c FROM Customer c
SELECT c.firstName FROM Customer c
SELECT c FROM Customer c WHERE c.firstName = 'Anthony'
SELECT c FROM Customer c WHERE c.address.city = 'London'
```

Desde JPA 2.0, se soporta en la seleccion el operador CASE-WHEN-THEN-ELSE-END. Por ejemplo:

```
SELECT CASE WHEN 'Apress'
 THEN b.price *0.5
 ELSE b.price * 0.8
 END FROM Book b
```

Si Customer tiene una relacion (a 1) con Address, entonces podemos hacer lo siguiente:

```
SELECT c.address FROM Customer c
```

Si Address tiene una relacion con Country el cual tiene el codigo(code), entonces podemmos usar la siguiente navegacion con "." para recuperar ese valor

```
SELECT c.address.country.code FROM Customer c
```

Para remover duplicados en la seleccion, usamos la palabra DISTINCT

```
SELECT DISTINCT FROM Customer c

SELECT DISTINCT c.firstName FROM Customer c
```

En el select podemos usar funciones de agregacion como AVG, MAX, MIN, SUM, COUNT. Los resultados pueden ser agrupados usando GROUP BY y filtrados usando HAVING. Tambien tenemos funciones esclares que podemos usar en el SELECT, en el WHERE y en el HAVING.

Sobre valores numericos, podemos usar las siguientes funciones:

- ABS
- SQRT
- MOD

- SIZE
- LENGTH
- AVG
- MAX
- MIN
- SUM
- COUNT
- INDEX

Sobre valores String, podemos usar las siguientes funciones:

- CONCAT
- SUBSTRING
- TRIM
- LOWER
- UPPER
- LOCATE
- LENGTH

Sobre valores de fecha, podemos usar las siguientes funciones:

- CURRENT_DATE
- CURRENT_TIME
- CURRENT_TIMESTAMP

From clause

Esta clausula es usada para definir las entidades que seran usadas en la consulta. Una variable de identificacion es un alias, permite utilizar dicho alias en los demas elementos de la consulta. (SELECT,WHERE)

```
SELECT c FROM Customer c
```

Where clause

Es una expresion boolenada usada para restringir el resultado de un Select, un Update o un Delete.

```
SELECT c FROM Customer c WHERE c.firstName = 'Anthony'
```

```
SELECT c FROM Customer c WHERE c.firstName = 'Anthony'
AND c.lastName = 'Giddens'
```

Esta clausula soporta los siguientes operadores logicos:

- =,¿,¡,¿=,¡=,¡¿
- NOT[BETWEEN], NOT[LIKE], NOT[IN]
- IS [NOT] NULL, IS [NOT] EMPTY

NOT MEMBER OF

- LIKE

Aqui unos ejemplo:

```
SELECT c FROM Customer c WHERE c.Age > 18
```

```
SELECT c FROM Customer c WHERE c.Age NOT BETWEEN 40 AND 50
```

```
SELECT c FROM Customer c WHERE c.address.country IN ('UK', 'USA')
```

```
SELECT c FROM Customer c WHERE c.email LIKE '%mail.com'
```

Parametros

JPQL soporta paramteros posicionales. Estos se especifican con un "?", seguido de las posicion (1,2,...). Cuando se ejecuta la query, los parametros deben ser reemplazados

```
SELECT c FROM
Customer c WHERE c.firstName = ?1 AND c.address.city = ?2
```

Tambien soporta parametros nombrados, se especifican con un ":" seguido de un nombre logico. Como en el caso anterior, al ejecutar la query, los parametros deben ser reemplazados.

```
SELECT c FROM
Customer c WHERE c.firstName = :firstName
AND c.address.city = :city
```

Subqueries

Es una query que puede ser embebida en un WHERE o en un HAVING. La evaluacion ocurre como parte de la evaluacion de la query principal.

```
SELECT c FROM Customer c WHERE
c.age = (SELECT MIN(cust.age) FROM Customer cust)
```

Order By clause

Como en el caso de SQL, permite ordenar los valores devueltos

```
SELECT c FROM Customer c WHERE c.age > 18 ORDER BY c.age DESC
```

Group By / Having clause

```
SELECT c.address.city, count(c)
FROM Customer c GROUP BY c.address.city

SELECT c.address.city, count(c)
FROM Customer c
GROUP BY c.address.country
HAVING c.address.country <> 'UK'
```

Bulk Delete

Retorna la cantidad de tuplas eliminadas

```
DELETE FROM <entity name>
[[AS] <identification variable>]
[WHERE <where clause>]

DELETE FROM Customer c WHERE
c.age < 18
```

Bulk Update

```
UPDATE <entity name>
[[AS] <identification variable>]
SET <update statement> {, <update statement>}*
[WHERE <where clause>]
```

```
UPDATE Customer c
SET c.firstName = 'TOO YOUNG'
WHERE c.age < 18
```

Tipos De Queries

- Dinamic Query
- Named Query
- Criteria API (Desde JPA 2.0)
- Native Query
- Stored Procedure Query (Desde JPA 2.1)

Recuperacion de resultados

- getResultList() : Retorna una lista de resultados
- getSingleResult() : Retorna un unico resultado

Para ejecutar un update o un delete, usamos el metodo executeUpdate(). Este retorna el numero de tuplas afectadas. Para poder estableceer los valores de los parametros, debemos usar el metodo setParameter() correspondiente al tipo de parametro que estamos usando. Podemos usar los siguientes metodos:

- setFirstResult() : Establece el primer resultado a recuperar
- setMaxResults() : Establece el maximo de resultados a recuperar

Queries Dinamicas

Son queries creadas cuando la aplicacion las necesita, utilizando el metodo EntityManager.createQuery(). Este metodo retorna un objeto Query, el cual puede ser usado para ejecutar la query.

```
Query query = em.createQuery("SELECT c FROM Customer c");
List<Customer> customers = query.getResultList();

TypedQuery<Customer> query = em.createQuery("SELECT c FROM Customer c
Customer.class);

List<Customer> customers = query.getResultList();
```

Si no conocemos la estructura de la consulta antemano podemos usar concatenaciones de strings para poder armar la query. Pero esto puede generar problemas de seguridad.

```
String jpqlQuery = "SELECT c FROM Customer c";
if(someCriteria){
    jpqlQuery += " WHERE c.firstName = 'Anthony'";
}
Query query = em.createQuery(jpql);
List<Customer> customers = query.getResultList();
```

Podemos utilizar parametros (nombrados o posicionales)

```
query = em.createQuery("SELECT c FROM Customer c WHERE c.firstName =
query.setParameter("firstName", "Anthony");
List<Customer> customers = query.getResultList();

query = em.createQuery("SELECT c FROM Customer c WHERE c.firstName =
query.setParameter(1, "Anthony");
List<Customer> customers = query.getResultList();
```

Es importante tener en cuenta el costo de procesar estas consultas por parte del proovedor de performance. Ya que cada consulta de estas:

- Debe ser procesada, no puede ser precedida
- Debe parsearse a JPQL
- Debe accederse a los metadatos
- Debe generarse en el SQL apropiado

Named Queries

Estas queries son estaticas e incambiables a nivel estructural. Estas aceptan parametros, son mas eficientes porque el proveedor de persistencia puede transformar las consultas en SQL cuando inicia la aplicacion. Se usa la notacion @NamedQuery y generalmente se definen en la entidad asociada.

```
@Entity
@NamedQueries({
    @NamedQuery(name = "findAll", query = "SELECT c FROM Customer c")
    @NamedQuery(name = "Anthony",
    query = "SELECT c FROM Customer c WHERE c.firstName = 'Anthony'")
    @NamedQuery(name = "findWithParam",
    query = "SELECT c FROM Customer c WHERE c.firstName = :firstName"
})

@Entity
@NamedQuery(name = "findAll", query = "SELECT c FROM Customer c")
public class Customer{...}

Query query = em.createNamedQuery("findAll");
```

```
TypedQuery<Customer> query =
em.createNamedQuery("findAll", Customer.class);
```

El nombre de la query nombrada, debe ser unico con el scope de persistencia que se esta utlizando. Por ejemplo, solo podemos tener una query nombrada "findAll" en el scope de la persistencia. Como podemos tener multiples entidad que requieran la misma query, podemos hacer esto:

```
@Entity
@NamedQuery(name = "Customer.findAll", query = "SELECT c FROM Custome
public class Customer{

public statis final String FIND_ALL = "Customer.findAll";

// atributos, constructores, getters y setters
}
```

Concurrencia

El problema de la concurrencia:

```
tx1.begin();
// The price of the book is $10
Book book = em.find(Book.class, 12);
book.raisePriceByTwoDollars();
tx1.commit();

tx2.begin();
// The price of the book is $10
Book book = em.find(Book.class, 12);
book.raisePriceByFiveDollars();
tx2.commit();
```

El ganador es el ultimo que commitea, esto no es un problema de JPA en especifico. Tampoco es un problema nuevo, por lo que exitsen multiples soluciones para este problema. Estas son:

- Optimistic Locking
- Pessimistic Locking

Versionado

Permite adjuntar un numero de version a las entiades. La primera vez que se persiste una entidad, esta tendra el numero de version 1. cuando se actualice la entidad y se commiteeen los cambios a la base de datos, el numero incrementara a 2 y asi sucesivamente. Para esot debemos color en la entidad un atributo para manipular la version. Este atributo debe ser de tipo Integer, Long, Short o Timestamp.

```
@Entity
public class Book{
    @Id@GeneratedValue
    private Long id;
    @Version
    private Integer version;
    private String title;
    private Float price;
    private String description;
    private String isbn;
    private String nbOfPage;
    private boolean illustrations;

    //constructors, getters and setters
}
```

@Version

La entidad puede acceder a su calor de version mas sin embargo no puede modificarlo. Si la entidad es modificada, el valor de la version se incrementara automaticamente por el proovedor de persistencia.

```
Book book =
new Book("H2G2" , 12.5F, "The Hitchhiker's Guide to the Galaxy" , "123-45
tx.begin();
em.persist(book);
tx.commit();
assertEquals(1, book.getVersion());

tx.begin();
book.raisePriceByTwoDollars();
tx.commit();
assertEquals(2, book.getVersion());
```

Locking Optimista

Asume que las transacciones no presentan muchos conflictos entre si. Cuando una transaccion viola el bloqueo optimista, se genera una excepcion OptimisticLockException. Y como se genera?

- Bloqueando la entidad con el LockModeType apropiado
- Dejando que el provider chequee el atributo anotado con @Version

```
tx1.begin();
// the price is $10
Book book = em.find(Book.class, 12);
book.getVersion() ==1
book.raisePriceByTwoDollars();
tx1.commit();
//The price now is $12
book.getVersion() == 2

tx2.begin();
// the price is $10
Book book = em.find(Book.class, 12);
book.getVersion() ==1
book.raisePriceByFiveDollars();
tx2.commit();
//The price now is $15
book.getVersion() == 2
```

Locking Pesimista

En este caso, el lock es obtenido antes de realizar las modificaciones sobre la entidad. Es restrictivo, generando degradacion de performance, sobre todo en sistema con alto nivel de concurrencia. La base de datos es la que provee el mecanismo de locking apropiado (SELECT ... FOR UPDATE). Si se viola se propaga una excepcion PessimisticLockException.

Conclusion

Java Persistence API (JPA) es una especificación de Java que proporciona un conjunto de estándares para mapear objetos Java a datos en una base de datos relacional. Está diseñada para simplificar el desarrollo de aplicaciones empresariales que interactúan con bases de datos.

Algunos puntos clave sobre JPA incluyen:

- Mapeo Objeto-Relacional (ORM): JPA permite mapear clases Java a tablas en una base de datos relacional y viceversa. Esto simplifica el desarrollo al eliminar la necesidad de escribir consultas SQL manualmente y facilita el mantenimiento del código.
- Entidades y Relaciones: JPA define el concepto de entidades, que son clases Java que representan objetos almacenados en la base de datos. Además, permite definir relaciones entre entidades, como relaciones uno a uno, uno a muchos y muchos a muchos, utilizando anotaciones como @OneToOne, @OneToMany, @ManyToOne y @ManyToMany.
- API de EntityManager: JPA proporciona la interfaz EntityManager para interactuar con la capa de persistencia. Esta interfaz permite realizar operaciones CRUD (crear, leer, actualizar y eliminar) en entidades, así como también realizar consultas utilizando el lenguaje de consultas de objetos Java (JPQL).
- Transacciones: JPA admite transacciones, lo que garantiza la integridad de los datos y la consistencia de la base de datos. Las transacciones se manejan utilizando la interfaz EntityTransaction, que permite comenzar, confirmar o revertir transacciones.
- Consultas JPQL: JPA proporciona un lenguaje de consultas llamado JPQL que permite realizar consultas orientadas a objetos en lugar de consultas SQL.
- Anotaciones: JPA utiliza anotaciones para mapear clases Java a tablas de base de datos y definir relaciones entre entidades. Algunas de las anotaciones más comunes incluyen @Entity, @Table, @Id, @GeneratedValue, @Column, @OneToOne, @OneToMany, @ManyToOne y @ManyToMany.
- Locking Optimista y Pesimista: JPA admite dos estrategias de bloqueo para manejar la concurrencia en entornos multiusuario. El bloqueo optimista se basa en un campo de versión
- Portabilidad: Una de las ventajas de JPA es su portabilidad entre diferentes proveedores de persistencia. Puedes cambiar fácilmente entre proveedores JPA, como Hibernate, EclipseLink o Apache OpenJPA, sin necesidad de cambiar el código de la aplicación.

En resumen, JPA simplifica el desarrollo de aplicaciones empresariales al proporcionar un modelo de programación orientado a objetos para interactuar con bases de datos relacionales. Facilita el mapeo de objetos a tablas de base de datos, el manejo de transacciones y la escritura de consultas, lo que hace que el desarrollo de aplicaciones sea más eficiente y menos propenso a errores.

Informacion Personal

- Nombre: Alejandro León Marín
- Correo: aleleonmarin01@gmail.com