JPA / HIBERNATE

VÍCTOR CUSTODIO , 2015

ANTES DE JPA

- Con JDBC:
  - se escribía el SQL a mano de las operaciones
  - con una query específica para cada modificación
  - Había que ajustar todo si algo cambiaba
  - Creabamos objetos navegando por el ResulSet

¿QUÉ ES JPA?

- Java es orientado a objetos pero las bases de datos relacionales están basadas en tablas, filas y columnas
- *JPA* (Java Persistence API, API de Persistencia en Java) es una abstracción sobre JDBC que nos permite realizar una correlación de forma sencilla, realizando por nosotros toda la conversión entre nuestros objetos y las tablas de una base de datos y viceversa.
- Esta conversión se llama ORM (Object Relational Mapping - Mapeo Relacional de Objetos), y puede configurarse a través de metadatos (mediante xml o anotaciones).

¿QUÉ ES JPA?

- JPA establece una interface común que es implementada por un proveedor de persistencia de nuestra elección (como Hibernate, Eclipse Link, etc), de manera que podemos elegir en cualquier momento el proveedor que más se adecue a nuestras necesidades.
- El proveedor es quién realiza el trabajo, pero siempre funcionando bajo la API de JPA.
- Nosotros trabajaremos sobre Hibernate que esta demostrado es el proveedor más potente.

JPA- ENTIDADES

- Cuando un POJO entra en el ámbito de JPA y es persistido por JPA hablamos de Entidades
- Ejemplo de una Entitadad sencilla

```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

@Entity
public class Pelicula {
    @Id
    @GeneratedValue
    private Long id;
    private String titulo;
    private int duracion;

    // Getters y Setters
}
```

JPA- ENTIDADES

- La configuración de mapeo puede ser especificada mediante un archivo de configuración XML, o mediante anotaciones. A esta configuración del mapeo la llamamos metadatos. Trabajaremos con anotaciones ya que su uso es más simple y natural.
- **@Entity** informa al proveedor de persistencia que cada instancia de esta clase es una entidad. Para ser válida, toda entidad debe:
  - Proporcionar un constructor por defecto (ya sea de forma implícita o explícita)
  - Ser una clase de primer nivel (no interna)
  - No ser final
  - Implementar la interface java.io.Serializabe si va a ser accedida remotamente
- Todas las entidades tienen que poseer una identidad que las diferencie del resto, por lo que deben contener una propiedad marcada con la anotación @Id.

*es aconsejable que dicha propiedad sea de un tipo que admita valores null, como Integer en lugar de int

JPA- CONFIGURACIÓN POR DEFECTO

- JPA aplica a las entidades que maneja una configuración por defecto, de manera que una entidad es funcional con una mínima cantidad de información (las anotaciones @Entity y @Id en nuestro caso). Con esta configuración por defecto, todas las entidades del tipo Pelicula serán mapeadas a una tabla de la base de datos llamada PELICULA, y cada una de sus propiedades será mapeada a una columna con el mismo nombre (la propiedad id será mapeada en la columna ID, la propiedad titulo será mapeada en la columna TITULO, etc).
- No siempre es posible ceñirse a los valores de la configuración por defecto:
  - Tenemos que trabajar con una base de datos heredada, con nombres de tablas y filas ya definidos. En ese caso, podemos configurar el mapeo de manera que se ajuste a dichas tablas y filas:

```
@Entity
@Table(name = "TABLA_PELICULAS")
public class Pelicula {
    @Id
    @GeneratedValue
    @Column(name = "ID_PELICULA")
    private Long id;

    // ...
}
```

JPA- LECTURA TEMPRANA Y LECTURA DEMORADA

- Cuando leemos una entidad desde la base de datos, ciertas propiedades pueden no ser necesarias en el momento de la creación del objeto. JPA nos permite leer una propiedad desde la base de datos la primera vez que un cliente intenta leer su valor (lectura demorada), en lugar de leerla cuando la entidad que la contiene es creada (lectura temprana).
- Ejemplo lectura demorada

```
@Basic(fetch = FetchType.LAZY)
private Imagen portada;
```

- Ejemplo lectura temprana (por defecto)

```
@Basic(fetch = FetchType.EAGER)
private Imagen portada;
```

JPA- TIPOS ENUMERADOS

- JPA puede mapear los tipos enumerados (enum) mediante la anotación Enumerated:

```
@Enumerated
private Genero genero;
```

- La configuración por defecto de JPA mapeará cada valor ordinal de un tipo enumerado a una columna de tipo numérico en la base de datos. Por ejemplo, siguiendo el ejemplo anterior, podemos crear un tipo enum que describa el género de una película:

```
public enum Genero {
    TERROR,
    DRAMA,
    COMEDIA,
    ACCION,
}
```

- Si la propiedad genero del primer ejemplo tiene el valor Genero.COMEDIA, en la columna correspondiente de la base de datos de insertará el valor 2 (que es el valor ordinal de Genero.COMEDIA)

JPA- TRANSIENT

- Ciertas propiedades de una entidad pueden no representar su estado. Por ejemplo, imaginemos que tenemos una entidad que representa a una persona:

```
@Entity
public class Persona {
    @Id
    @GeneratedValue
    private Long id;
    private String nombre;
    private String apellidos
    private Date fechaNacimiento;
    private int edad;

    // getters y setters
}
```

- Podemos considerar que la propiedad edad no representa el estado de Persona, ya que si no es actualizada cada cumpleaños, terminará conteniendo un valor erróneo:

```
@Transient
private int edad;
```

- Ahora, para obtener el valor de edad utilizamos su método getter:

```
public int getEdad() {
    // calcular la edad y devolverla
}
```

JPA- COLECCIONES BÁSICAS

- Una entidad puede tener propiedades de tipo java.util.Collection y/o java.util.Map que contengan tipos básicos. Los elementos de estas colecciones serán almacenados en una tabla diferente a la que contiene la entidad donde están declarados. Podemos dejar los valores por defecto o modificarlos:

```
@ElementCollection(fetch = FetchType.LAZY)
@CollectionTable(name = "TABLA_COMENTARIOS")
private ArrayList<String> comentarios;
```

-

JPA- TIPOS INSERTABLES

- Los tipos insertables (embeddable types) son objetos que no tienen identidad, por lo que para ser persistidos deben ser primero insertados dentro de una entidad (u otro insertable que será a su vez insertado en una entidad, etc)
- Cada propiedad del tipo insertable será mapeada a la tabla de la entidad que lo contenga, como si fuera una propiedad declarada en la propia entidad. Definimos un tipo insertable con la anotación @Embeddable

```
@Embeddable
public class Direccion {
    private String calle;
    private int codigoPostal;

    // ...
}
```

- Y lo insertamos en una entidad:

```
@Entity
public class Persona {
    // ...
    @Embedded
    private Direccion direccion;
}
```

JPA- ASOSIACIONES

- Cuando queremos mapear colecciones de entidades, debemos usar asociaciones. Estas asociaciones pueden ser de dos tipos:
  - Unidireccionales reflejan un objeto que tiene una referencia a otro objeto (la información puede viajar en una dirección).
  - Bidireccionales representan dos objetos que mantienen referencias al objeto contrario (la información puede viajar en dos direcciones).
- Además del concepto de dirección, existe otro concepto llamado cardinalidad, que determina cuantos objetos pueden haber en cada extremo de la asociación.

JPA- ASOCIACIONES UNIDIRECCIONALES

```
@Entity
public class Cliente {
    @Id
    @GeneratedValue
    private Long id;
    @OneToOne
    private Direccion direccion;

    // Getters y setters
}

@Entity
public class Direccion {
    @Id
    GeneratedValue
    private Long id;
    private String calle;
    private String ciudad;
    private String pais;
    private Integer codigoPostal;

    // Getters y setters
}
```

JPA- ASOCIACIONES UNIDIRECCIONALES – UNO A UNO

```
@Entity
public class Cliente {
    @Id
    @GeneratedValue
    private Long id;
    @OneToOne
    private Direccion direccion;

    // Getters y setters
}
```

```
@Entity
public class Direccion {
    @Id
    GeneratedValue
    private Long id;
    private String calle;
    private String ciudad;
    private String pais;
    private Integer codigoPostal;

    // Getters y setters
}
```

- *Tambien podemos definir nosotros el nombre de la columna :

```
@OneToOne
@JoinColumn(name = "DIRECCION_FK")
private Direccion direccion;
```

JPA- ASOCIACIONES UNIDIRECCIONALES – UNO A MUCHOS

```
@Entity
public class Cliente {
   @Id
   @GeneratedValue
   private Long id;
   @OneToMany
   private List<Direccion> direcciones;

   // Getters y setters
}
```

- JPA utilizará por defecto una tabla de unión (join table). Cuando ocurre esto, las tablas donde se almacenan ambas entidades contienen una clave foránea a una tercera tabla con dos columnas; esta tercera tabla es llamada tabla de unión, y es donde se estata la asociación entre las entidades relacionadas
  - *Tambien podemos definir nosotros el nombre de las columnas de la tabla de unión:

```
@OneToMany
@JoinTable(name = ...,
      joinColumn = @JoinColumn(name = ...),
      inverseJoinColumn = @JoinColumn(name = ...))
private List direcciones;
```

JPA- ASOCIACIONES BIDIRECCIONALES (MAPPEDBY)

```
@Entity
public class Mujer {
    @Id
    @GeneratedValue
    private Long id;
    @OneToOne
    private Marido marido;

    // Getters y setters
}
```

```
@Entity
public class Marido {
    @Id
    @GeneratedValue
    private Long id;
    @OneToOne(mappedBy = "marido")
    private Mujer mujer;
}
```

- El atributo mappedBy puede ser usado en relaciones de tipo @OneToOne, @OneToMany y @ManyToMany. Únicamente el cuarto tipo de relación, @ManyToOne, no permite el uso de este atributo.

JPA- ASOCIACIONES BIDIRECCIONALES (MAPPEDBY)

- Por defecto la carga de objetos uno-a-muchos y muchos a muchos es demorada lazy.
- Como siempre podemos cambiarla:

```
@OneToMany(fetch = FetchType.EAGER)
private List<Pedido> pedidos;
```

JPA- ASOCIACIONES (ORDERBY)

- Puedes ordenar los resultados devueltos por una asociación mediante la anotación @OrderBy:

```
@OneToMany
@OrderBy("nombrePropiedad asc")
private List<Pedido> pedidos;
```

JPA- HERENCIA

- JPA nos permite gestionar la forma en que nuestros objetos son mapeados cuando en ellos interviene el concepto de herencia. Esto puede hacerse de maneras distintas:
  - Una tabla por familia (comportamiento por defecto)
  - Unión de subclases
  - Una tabla por clase concreta

JPA- HERENCIA – UNA TABLA POR FAMILIA

- El mapeo por defecto es *una tabla por familia*.
- Una familia no es, ni más ni menos, que todas las subclases que están relacionadas por herencia con una clase madre, inclusive.
- Todas las clases que forman parte de una misma familia son almacenadas en una única tabla. En esta tabla existe una columna por cada atributo de cada clase y subclase de la familia

JPA- HERENCIA – UNA TABLA POR FAMILIA

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
public class SuperClase {
    @Id
    @GeneratedValue
    private Long id;
    private int propiedadUno;
    private String propiedadDos;

    // Getters y Setters
}
```

```
@Entity
public class SubClase extends SuperClase {
    @Id
    @GeneratedValue
    private Long id;
    private float propiedadTres;
    private float propiedadCuatro;

    // Getters y setters
}
```

- Tanto las instancias de las entidades SuperClase y SubClase seran almancenadas en una única tabla que tendrá el nombre por defecto de la clase raíz (SuperClase). Dentro de esta tabla habrá seis columnas que se corresponderán con:
  - Una columna para la propiedad id (válida para ambas entidades, pues en ambas se mapearía a una columna con el mismo nombre)
  - Cuatro columnas para las propiedades propiedadUno, propiedadDos, propiedadTres y propiedadCuatro
  - Una última columna discriminatoria, donde se almacenará el tipo de clase al que hace referencia cada fila

JPA- HERENCIA – UNA TABLA POR CLASE UNIDA

- El segundo tipo de mapeo cuando existe herencia es unión de subclases, en el que cada clase y subclase (sea abstracta o concreta) será almacenada en su propia tabla:

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public class SuperClase { ... }
```

- La tabla raíz contiene una columna con una clave primaria usada por todas las tablas, así como la columna discriminatoria. Cada subclase almacenará en su propia tabla únicamente sus atributos propios (nunca los heredados), así como una clave foránea que hace referencia a la clave primaria de la tabla raíz.
  - La mayor ventaja de este sistema es que es intuitivo.
  - Su inconveniente es que, para construir un objeto de una subclase, hay que hacer una (o varias) operaciones JOIN en la base de datos.

JPA- HERENCIA – UNA TABLA POR CLASE INDEPENDIENTE

- El tercer y último tipo de mapeo cuando existe herencia es una tabla por clase concreta. Mediante este comportamiento, cada entidad será mapeada a su propia tabla (incluyendo todos los atributos propios y heredados). Con este sistema no hay tablas compartidas, columnas compartidas, ni columna discriminatoria:

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class SuperClase { ... }
```

- Solo comparten el valor de la clave primaria. También requiere operaciones JOIN en base de datos

PERSISTENCIA EN JPA

CRUD

PERSISTENCIA EN JPA

- JPA maneja todas las operaciones CRUD a través de la interface EntityManager.
- Comenzaremos definiendo una entidad de prueba:

```
import javax.persistence.*;
@Entity
public class Pelicula {
    @Id
    @GeneratedValue
    private Long id;
    private String titulo;
    private int duracion;

    // Getters y Setters
}
```

PERSISTENCIA EN JPA

- Para que JPA pueda persistir esta entidad, necesita un archivo de configuración XML llamado *persistence.xml*, el cual debe estar ubicado en el directorio *META-INF* de nuestra aplicación. En nuestro caso completaremos este archivo mediante eclipse y quedará con el siguiente aspecto:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
	<persistence-unit name="JPAPrueba" transaction-type="RESOURCE_LOCAL">
		<properties>
			<property name="hibernate.hbm2ddl.auto" value="create-drop"/>
			<property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/prueba"/>
			<property name="javax.persistence.jdbc.user" value="root"/>
			<property name="javax.persistence.jdbc.password" value="root"/>
			<property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
		</properties>
	</persistence-unit>
</persistence>
```

PERSISTENCIA EN JPA-TRANSACCIONES

- <persistence-unit name="introduccionJPA" transaction-type="RESOURCE_LOCAL">
- <!-- ... -->
- </persistence-unit>
- Dos tipos de gestión de transacciones:
  - Manejada por la aplicacion (RESOURCE_LOCAL)
  - Manejada por el contenedor (JTA)
- Nosotros hemos decidido manejar las transacciones desde la aplicación, ya que tomcat 8.5 no tiene implementación de JTA .Haremos la gestión de transacciones desde nuestro código a través de la interface EntityTransaction. En una aplicación real son utiles transacciones manejadas por el contenedor

PERSISTENCIA EN JPA – ESTADO DE LAS ENTIDADES

- Para JPA, una entidad puede estar en uno de los dos estados siguientes:
  - Managed (gestionada)
  - Detached (separada)
- Cuando persistimos una entidad, automáticamente se convierte en una entidad *gestionada*. Todos los cambios que efectuemos sobre ella dentro del contexto de una transacción se verán reflejados también en la base de datos, de forma transparente para la aplicación.
- En el estado *separado*, los cambios realizados en la entidad no están sincronizados con la base de datos. Una entidad se encuentra en estado separado antes de ser persistida por primera vez, y cuando tras haber estado gestionada es separada de su contexto de persistencia.

PERSISTIR UNA ENTIDAD

```
public class Main {
    public static void main(String[] args) {
        EntityManagerFactory emf =       Persistence.createEntityManagerFactory(“JPAPrueba");
        EntityManager em = emf.createEntityManager();
        EntityTransaction tx = em.getTransaction();

     Pelicula pelicula = new Pelicula();
        pelicula.setTitulo("Pelicula uno");
        pelicula.setDuracion(142);

        tx.begin();
        try {
            em.persist(pelicula);
            tx.commit();
        } catch(Exception e) {
            tx.rollback()
        }

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

PERSISTIR UNA ENTIDAD - SINCRONIZACIÓN

- Cuando llamamos a em.persist(pelicula) la entidad es persistida en nuestra base de datos, pelicula y queda gestionada por el proveedor de persistencia mientras dure la transacción. Por ello, cualquier cambio en su estado será sincronizado automáticamente y de forma transparente para la aplicación:

```
tx.begin();
em.persist(pelicula);
pelicula.setTitulo("otro titulo");
tx.commit();
```

- No sabemos cuando se realizará el volcado a base de datos, depende de la implementación de jpa(. Para forzar este volcado utilizamos el método flush:

```
tx.begin();
em.persist(pelicula);
pelicula.setTitulo("otro titulo");
em.flush();
// otras operaciones
tx.commit();
```

PERSISTIR UNA ENTIDAD - SINCRONIZACIÓN

- Podemos hacer la función contraria, actualizar nuestro objeto con los datos de la base de datos utilizando el metodo refresh():

```
tx.begin();
em.persist(pelicula);
pelicula.setTitulo("otro titulo");
em.refresh(pelicula);
tx.commit();
```

- Podemos conocer si una entidad esta gestionada con el método contains(entidad):

```
boolean gestionada = em.contains(pelicula);
// lógica de la aplicacion
```

PERSISTENCIA EN JPA- LEER UNA ENTIDAD

- Leer una entidad previamente persistida en la base de datos para construir un objeto Java. Podemos llevar a cabo esto de dos maneras distintas:
  - Obteniendo un objeto real

    Pelicula pelicula = em.find(Pelicula.class, id);
  - Obteniendo una referencia a los datos persistidos

    Pelicula pelicula = em.getReference(Pelicula.class, id);
    - Nos permite obtener una referencia a los datos almacenados en la base de datos, de manera que el estado de la entidad será leído de forma demorada, más concretamente en el primer acceso a cada propiedad y no en el momento de la creación de la entidad:

SEPARAR ENTIDADES DEL CONTEXTO DE PERSISTENCIA

- Podemos separar una o todas las entidades gestionadas actualmente por el contexto de persistencia mediante los métodos detach() y clear(), respectivamente

```
tx.begin();
em.persist(pelicula);
tx.commit();
em.detach(pelicula);
```

ACTUALIZAR UNA ENTIDAD

- Si queremos actualizar una entidad que este gestionada, valdrá con cambiar sus atributos, en cambio si queremos actualizar una entidad no gestionada debemos hacer uso del metodo merge() del EntityManager:

```
tx.begin();
em.persist(pelicula);
tx.commit();
em.detach(pelicula);
// otras operaciones
em.merge(pelicula);
```

ELIMINAR UNA ENTIDAD

- Cuando realizamos esta operación, la entidad es eliminada de la base de datos y separada del contexto de persistencia. Sin embargo, la entidad seguirá existiendo como objeto Java en nuestro código :

```
em.remove(pelicula);
pelicula.setTitulo("ya no soy una entidad, solo un objeto Java normal");
```

- Cuando existe una asociación uno-a-uno y uno-a-muchos entre dos entidades, y eliminamos la entidad dueña de la relación, la/s entidad/es del otro lado de la relación no son eliminada/s de la base de datos (este es el comportamiento por defecto), pudiendo dejar así entidades huerfana.
- podemos configurar nuestras asociaciones para que eliminen de manera automática todas las entidades subordinadas de la relación:

```
@Entity
public class Pelicula {
    ...
    @OneToOne(orphanRemoval = true)
    private Descuento descuento;

    // Getters y setters
}
```

OPERACIONES EN CASCADA

- La forma general de establecer como se realizarán estas operaciones en cascada se define mediante el atributo cascade de las anotaciones de asociación

```
@OneToOne(cascade = CascadeType.REMOVE)
private Descuento descuento;
```

- El tipo de operación en cascada que deseamos propagar (CascadeType.REMOVE) se indica mediante constantes de la clase CascadeType. Estas constantes pueden tener los siguientes valores:
  - PERSIST
  - REMOVE
  - MERGE
  - REFRESH
  - DETACH
  - ALL

JPQL

JPQL

- Al usar la interface EntityManager estamos limitados a realizar consultas en la base de datos proporcionando la identidad de la entidad que deseamos obtener, y solo podemos obtener una entidad por cada consulta que realicemos.
- JPQL nos permite realizar consultas en base a multitud de criterios

- Ejemplo:

```
SELECT p FROM Pelicula p
```

JPQL

```
SELECT p.titulo FROM Pelicula p
```

Notacion por puntos!

```
SELECT c.propiedad.subPropiedad.subSubPropiedad FROM Clase c
```

```
SELECT p.titulo, p.duracion FROM Pelicula p
```

Devuelve varias propiedades

```
SELECT COUNT(p) FROM Pelicula p
```

Funciones agregadas

JPQL

```
SELECT p FROM Pelicula p WHERE p.duracion < 120
```

```
SELECT p FROM Pelicula p
WHERE p.duracion < 120 AND p.genero = 'Terror'
```

```
SELECT p FROM Pelicula p
WHERE p.duracion BETWEEN 90 AND 150
```

```
SELECT p FROM Pelicula p
WHERE p.titulo LIKE 'El%'
```

Con condicionales!

JPQL

UPDATE Articulo a SET a.descuento = 15 WHERE a.precio > 50

Actualización

DELETE FROM Pelicula p WHERE p.duracion > 190

Borrado

Ver toda la sintaxis del lenguaje en http://docs.oracle.com/javaee/6/tutorial/doc/bnbuf.html

JPQL – EJECUCIÓN DE QUERYS

- El lenguaje JPQL es integrado a través de implementaciones de la interface Query. Dichas implementaciones se obtienen a través del EntityManager, mediante diversos métodos de factoría:
  - **Query**
  - NamedQuery
  - CriteriaQuery
  - NativeQuery

- Solamente estudiaremos las Querys más utilizadas, Query.
- Ver interfaz query en : https://docs.oracle.com/javaee/7/api/javax/persistence/Query.html

JPQL – EJECUCIÓN DE QUERYS BÁSICAS

```
public class Main {

    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence
                .createEntityManagerFactory("introduccionJPA");
        EntityManager em = emf.createEntityManager();

        String jpql = "SELECT p FROM Pelicula p";
        Query query = em.createQuery(jpql);
        List<Pelicula> resultados = query.getResultList();
        for(Pelicula p : resultados) {
            // ...
        }

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

JPQL – EJECUCIÓN DE QUERYS CON PARÁMETROS DINAMICOS

- Dos opciones:
  - Con interrogaciones:

```
String jpql = "SELECT p FROM Pelicula p WHERE p.duracion > ?1 AND p.genero = ?2"
Query query = em.createQuery(jpql);
query.setParameter(1, 180);
query.setParameter(2, "Accion");
List<Pelicula> resultados = query.getResultList();
```

  - Con nombres

```
public List<Pelicula> buscarPeliculaPorProtagonista(Actor actor){
String jpql = "SELECT p FROM Pelicula p WHERE p.actorPrincipal = :actor
Query query = em.createQuery(jpql);
query.setParameter("actor", actor);
List<Pelicula> resultados = query.getResultList();
return resultados;
}
```