Ministerio de Educación Superior, Ciencia y Tecnología (Mescyt)

Centro de Tecnología y Educación Permanente TEP

Pontificia Universidad Católica Madre y Maestra (PUCMM)

Diplomado en Programación en Lenguaje JAVA

Laboratorio 02

Módulo III

Manejadores de persistencia

Facilitador: Ing. Eudris Cabrera

Fecha : 30 Julio del 2016

Santiago de los Caballeros, República Dominicana

Introducción

En Java solucionamos problemas de negocio a través de objetos, los cuales tienen estado y comportamiento.

Sin embargo, las bases de datos relacionales almacenan la información mediante tablas, filas, y columnas, de manera que para almacenar un objeto hay que realizar una correlación entre el sistema orientado a objetos de Java y el sistema relacional de nuestra base de datos.

El **Java Persistence API** (API de Persistencia en Java) (JPA) es una abstracción sobre JDBC que nos permite realizar dicha correlación de forma sencilla, realizando por nosotros toda la conversión entre nuestros objetos y las tablas de una base de datos.

Esta conversión se llama **ORM (Object Relational Mapping - Mapeo Relacional de Objetos)**, y puede configurarse a través de metadatos (mediante xml o anotaciones).

JPA también nos permite seguir el sentido inverso, creando objetos a partir de las tablas de una base de datos, y también de forma transparente. A estos objetos los llamaremos desde ahora entidades (entities).

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. Así, es el proveedor quién realiza el trabajo, pero siempre funcionando bajo la API de JPA.

Guía del laboratorio

Este laboratorio está enfocado en los conceptos fundamentales sobre los manejadores de persistencia en Java y su implementación en una aplicación web Java.

Escenario

Se desea crear la estructura de Entidades de un sistema de evaluación de peliculas.
Los datos que tenemos sobre cada uno de los actores que intervienen en nuestro escenario son los siguientes:

Película	Actor	Director
Título Descripción Idioma fecha de lanzamiento género Studio / Productor página web tiempo de duración **Puntuaciones** (*) **Actor** (*) **Director**(*)	Nombre Fecha de Nacimiento biografía lugar de origen **Roles** (*)	Nombre Fecha de Nacimiento biografía lugar de origen **Roles** (*)

Roles	Usuario	Puntuaciones
Nombre Descripción	Login nombre Contraseña Información personal **Roles** (*)	**Usuario** (*) **Película** (*) Estrellas Comentarios

Requerimientos

Java 1.7+ (Recomendado Java 8)

Netbeans 8.0.1+

Apache Maven 3.2.1+

Glassfish 4.1.1

Mysql Server 5.0+

Configuración de Ambiente

Crear una base de datos en mysql llamada **jpa_diplomado.**

```
create database jpa_diplomado;
```

Crear una aplicación **Maven** del tipo **Web Application**, ir a la opción: **New Project -> Maven -> Java Application.**

Información del proyecto

Nombre : **demo_jpa**

artifactId: valor por defecto

groupId: ***org.diplomado.pucmm.mescyt***

version: ***valor por defecto "1.0-SNAPSHOT"***

package: ***org.diplomado.pucmm.mescyt.java.jpa***

Modificar archivo pom.xml

Mysql

```
        <dependency>
                <groupId>mysql</groupId>
                <artifactId>mysql-connector-java</artifactId>
                <version>5.1.36</version>
        </dependency>
```

JPA

```
 <dependency>
                <groupId>org.eclipse.persistence</groupId>
                <artifactId>eclipselink</artifactId>
                <version>2.6.1</version>
</dependency>
```

Paso II

Crear la siguiente estructura de paquetes

1. ***org.diplomado.pucmm.mescyt.java.jpa*.entidades**

Clase que representan el dominio o negocio (POJO).

Introducción a Java Persistence API (JPA)

Creación de una entidad Simple usando JPA

Crear la entidad película sin las relaciones.

```
@Entity
public class Pelicula {
	@Id
	@GeneratedValue
	private Long id;
	private String titulo;
	private int duracion;
     ......
	// Getters y Setters
}
```

Las entidades suelen ser POJOs. La clase Película se ha anotado con **@Entity**, lo cual informa al proveedor de persistencia que cada instancia de esta clase es una entidad.

Para ser válida, toda entidad debe:

- Tener un constructor por defecto
- Ser una clase de primer nivel (no interna)
- No ser final
- Implementar la **interface java.io.Serializabe** si es accedida remotamente

La configuración de mapeo puede ser especificada mediante un archivo de configuración XML, o mediante anotaciones. A esta configuración del mapeo la llamamos metadatos.

Identidad:

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).

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 título será mapeada en la columna **titulo**, etc).

Sin embargo, no siempre es posible ceñirse a los valores de la configuración por defecto: imaginemos que 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;
	...
}
```

La anotación **@Table** nos permite configurar el nombre de la tabla donde queremos almacenar la entidad mediante el atributo name.

Así mismo, mediante la anotación **@Column** podemos configurar el nombre de la columna donde se almacenará una propiedad, si dicha propiedad puede contener un valor null, etc.

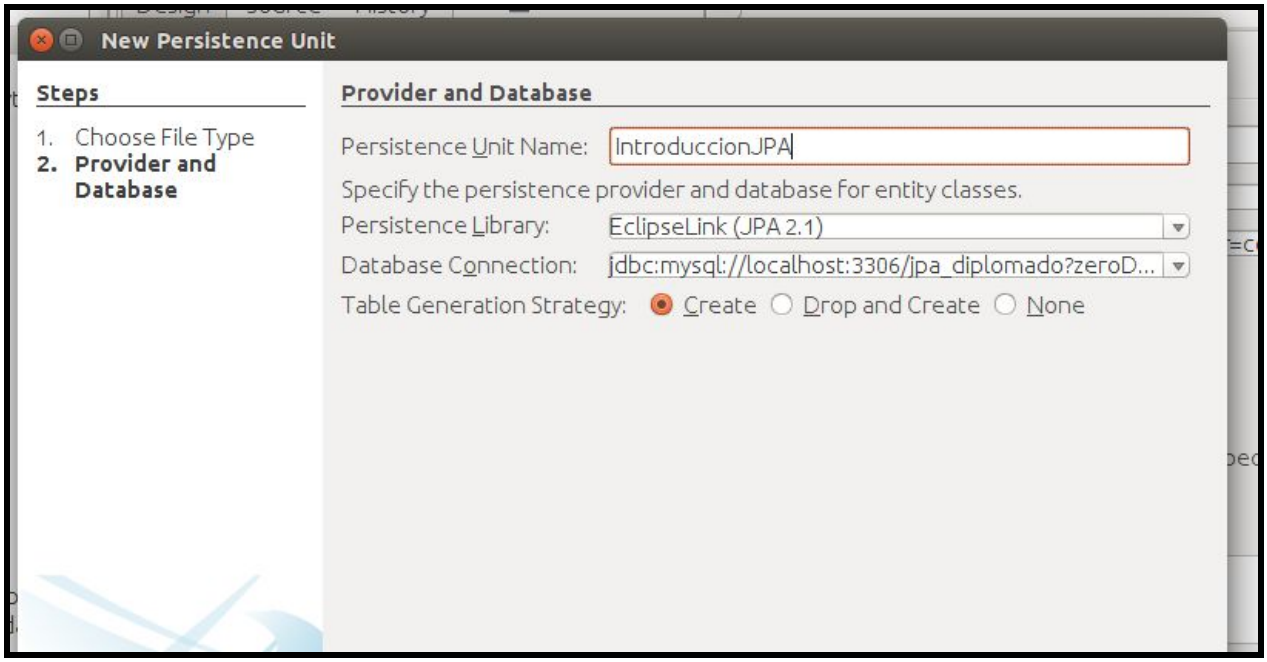
Persistencia

El concepto de persistencia implica el hecho de almacenar nuestras entidades (objetos Java de tipo POJO) en un sistema de almacenamiento, normalmente una base de datos relacional (tablas, filas, y columnas).

Más allá del proceso de almacenar entidades en una base de datos, todo sistema de persistencia debe permitir recuperar, actualizar y eliminar dichas entidades. Estas cuatro operaciones se conocen como operaciones CRUD (Create, Read, Update, Delete - Crear, Leer, Actualizar, Borrar).

JPA maneja todas las operaciones CRUD a través de la interfaz **EntityManager**. Antes de intentar persistir cualquier entidad debe haber creado la unidad de persistencia **“PersistenceUnit”** que es representado por un archivo xml llamado y comúnmente llamado **persitance.xml.**

Desde cualquier proyecto **New File > Other > Persistance > PersistanceUnit.** Completar los datos que nos pide el formulario. Esto es, un **nombre** para unidad de persistencia, conexión a la base de datos y la estrategia de generación de tablas



Persistir una Entidad

```
EntityManagerFactory emf =   Persistence.createEntityManagerFactory("introduccionJPA");
        EntityManager em = emf.createEntityManager();
        EntityTransaction tx = em.getTransaction();

        Pelicula pelicula = new Pelicula();
        pelicula.setTitulo("Pelicula uno");
       pelicula.setDescripcion("Esta es una pelicula de Prueba");
        pelicula.setDuracion(120);
        pelicula.setGenero("Drama");

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

Leer una Entidad

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

- Obteniendo un objeto real
- Obteniendo una referencia a los datos persistidos

De la primera manera, los datos son leídos (por ejemplo, con el método **find()**) desde la base de datos y almacenados en una instancia de la entidad:

```
Pelicula p = em.find(Pelicula.class, id);
```

La segunda manera de leer una entidad 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 (en el primer acceso a cada propiedad), no en el momento de la creación de la entidad:

```
Pelicula p = em.getReference(Pelicula.class, id);
```

Actualizar una Entidad

Mediante el método **merge()** de **EntityManager** podemos tener nuestra entidad gestionada y sincronizada

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

Eliminar una entidad

```
em.remove(pelicula);
```

Cuando existe una asociación uno-a-uno y uno-a-muchos entre dos entidades, y eliminas 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 persistidas sin ninguna entidad que haga referencia a ellas.

Estas entidades se conocen como entidades huérfanas. Sin embargo, podemos configurar nuestras asociaciones para que eliminen de manera automática todas las entidades subordinadas de la relación

Asociaciones

Cuando queremos mapear colecciones de entidades, debemos usar asociaciones. Estas asociaciones pueden ser de dos tipos:

- Asociaciones unidireccionales
- Asociaciones bidireccionales

Las asociaciones unidireccionales reflejan un objeto que tiene una referencia a otro objeto (la información puede viajar en una dirección).

Por el contrario, las asociaciones bidireccionales reflejan 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 cuántos objetos pueden haber en cada extremo de la asociación.

Asociaciones Unidireccionales

```
@Table(name = "pais")
@Entity
public class Pais implements Serializable {
	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	private Long id;

	private String nombre;
}

@Entity
@Table(name = "pelicula")
public class Pelicula implements Serializable {

	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	private Long id;
	@Column(name = "titulo")
	private String titulo;
	@Column(name = "descripcion")
	private String descripcion;
	@Column(name = "fecha_lanzamiento")
	@Temporal(javax.persistence.TemporalType.DATE)
	private Date fechaLanzamiento;
	@Column(name = "genero")
	private String genero;
	@Column(name = "pagina_web")
	private String paginaWeb;
	@Column(name = "duracion")
	private int duracion;

	@OneToOne
	private Pais paisOrigen;
}
```

Otro tipo de asociación muy común es la de tipo uno-a-muchos (one-to-many) unidireccional.

```
@Entity
@Table(name = "pelicula")
```

```
public class Pelicula implements Serializable {

        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        private Long id;
        @Column(name = "titulo")
        private String titulo;
        @Column(name = "descripcion")
        private String descripcion;
        @Column(name = "fecha_lanzamiento")
        @Temporal(javax.persistence.TemporalType.DATE)
        private Date fechaLanzamiento;
        @Column(name = "genero")
        private String genero;
        @Column(name = "pagina_web")
        private String paginaWeb;
        @Column(name = "duracion")
        private int duracion;

        @OneToOne
        private Pais paisOrigen;

        @OneToMany
        private List<Actor> actores;

}
```

Asociaciones Bidireccionales

En las asociaciones bidireccionales, ambos extremos de la relación mantienen una referencia al extremo contrario.

En este caso, el dueño de la relación debe ser especificado explícitamente, de manera que JPA pueda realizar el mapeo correctamente.

```
@Entity
@Table(name = "pelicula")
public class Pelicula implements Serializable {

    ....
```

```
        @OneToMany(mappedBy = "pelicula")
        private List<Actor> actores;

}

@Entity
@Table(name = "actor")
public class Actor implements Serializable {

        ....

        @ManyToOne(fetch = FetchType.LAZY)
        @JoinColumn(name = "pelicula_id")
        private Pelicula pelicula;

}
```

Operaciones en Cascada

Este tipo de operaciones permiten establecer la forma en que deben propagarse ciertos eventos (como persistir, eliminar, actualizar, etc) entre entidades que forman parte de una asociación.

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

```
@OneToOne(cascade = CascadeType.ALL)
private Pais paisOrigen;

CascadeType.PERSIST
        .REMOVE
        .MERGE
        .REFRESH
        .DETACH
        .ALL
```

La última constante, **CascadeType.ALL**, hace referencia a todas las posibles operaciones que pueden ser propagadas, y por tanto engloba en sí misma el comportamiento del resto de constantes.

También podemos configurar varias operaciones en cascada de la lista superior usando un array de constantes CascadeType:

```
@OneToOne(cascade = {
        CascadeType.MERGE,
        CascadeType.REMOVE,
        })
private Pais paisOrigen;
```

Lectura Temprana y Lectura Demorada

JPA nos permite leer una propiedad desde la base de datos la primera vez que un cliente intenta leer su valor (lectura demorada). Esto puede ser útil si la propiedad contiene un objeto de gran tamaño:

```
@OneToMany(fetch = FetchType.LAZY)
private List<Actor> actores;
```

El comportamiento por defecto de la mayoría de tipos Java es el contrario (lectura temprana).
Este comportamiento, a pesar de ser implícito, puede ser declarado explícitamente de la siguiente manera:

```
@OneToMany(fetch = FetchType.EAGER)
private List<Actor> actores;
```

JPQL (Java Persistence Query Language)

Lenguaje de Consulta de Persistencia en Java), un potente lenguaje de consulta orientado a objetos que va incluido con JPA.

JPQL BÁSICO

JPQL nos permite realizar consultas en base a multitud de criterios (como por ejemplo el valor de una propiedad, o condiciones booleanas), y obtener más de un objeto por consulta.

```
SELECT p FROM Pelicula p
```

Sentencias Condicionales

El primero de ellos es el de consulta condicional, el cual es aplicado añadiendo la cláusula WHERE en nuestra sentencia JPQL.

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

Parámetros Dinámicos

Podemos añadir parámetros dinámicamente a nuestras sentencias JPQL de dos formas: por posición y por nombre. La sentencia siguiente acepta un parámetro por posición (?1):

```
SELECT p FROM Pelicula p WHERE p.titulo = ?1
```

Y la siguiente, acepta un parámetro por nombre (:titulo):

```
SELECT p FROM Pelicula p WHERE p.titulo = :titulo
```

Ejecución de Sentencias JPQL

El lenguaje JPQL es integrado a través de implementaciones de la interfaz **Query**.

Dichas implementaciones se obtienen a través de nuestro querido amigo **EntityManager**, mediante diversos métodos de factoría.

De estos, los tres más usados (y los únicos que explicaremos aquí) son:

```
createQuery(String jpql)
createNamedQuery(String name)
createNativeQuery(String sql)
```

Consultas con Nombre (Estáticas)

Las consultas con nombre son diferentes de las sentencias dinámicas que hemos visto hasta ahora en el sentido de que una vez definidas, no pueden ser modificadas: son leídas y transformadas en sentencias SQL cuando el programa arranca por primera vez, en lugar de cada vez que son ejecutadas.

Este comportamiento estático las hace más eficientes, y por tanto ofrecen un mejor rendimiento. Las consultas con nombre son definidas mediante metadatos (recuerda que los metadatos se definen mediante anotaciones o configuración XML), como puedes ver en este ejemplo:

```
@Entity
@NamedQuery(name="buscarTodos", query="SELECT p FROM Pelicula p")
public class Pelicula { ... }
```

Consultas Nativas SQL

El tercer y último tipo de consultas que nos queda por ver requiere una sentencia SQL nativa en lugar de una sentencia JPQL:

```
String sql = "SELECT * FROM PELICULA";
Query query = em.createNativeQuery(sql);
```

Las consultas SQL nativas pueden ser definidas de manera estática como hicimos con las consultas con nombre, obteniendo los mismos beneficios de eficiencia y rendimiento.

Para ello, necesitamos utilizar, de nuevo, metadatos:

```
@Entity
@NamedNativeQuery(name=Pelicula.BUSCAR_TODOS, query="SELECT * FROM PELICULA")
public class Pelicula {
	public static final String BUSCAR_TODOS = "Pelicula.buscarTodos";
	...
}
```