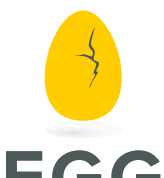
CURSO DE PROGRAMACIÓN FULL STACK

ACCESO A BASES DE DATOS DESDE JAVA: JPA



GUÍA DE PERSISTENCIA CON JPA

PERSISTENCIA EN JAVA CON JPA

JPA (Java Persistence API) es la propuesta estándar que ofrece Java para implementar un Framework Object Relational Mapping (ORM), que permite interactuar con la base de datos por medio de objetos, de esta forma, JPA es el encargado de convertir los objetos Java en instrucciones para el Manejador de Base de Datos (MDB). El objetivo que persigue el diseño de esta API es no perder las ventajas de la orientación a objetos al interactuar con una base de datos (siguiendo el patrón de mapeo objeto-relacional).

Cuando empezamos a trabajar con bases de datos en Java utilizamos el API de JDBC el cual nos permite realizar consultas directas a la base de datos a través de consultas SQL nativas. JDBC por mucho tiempo fue la única forma de interactuar con las bases de datos, pero representaba un gran problema y es que Java es un lenguaje orientado a objetos y se tenía que convertir los atributos de las clases en una consulta SQL como SELECT, INSERT, UPDATE, DELETE, etc. Lo que ocasionaba un gran esfuerzo de trabajo y provocaba muchos errores en tiempo de ejecución, debido principalmente a que las consultas SQL se tenían que generar frecuentemente al vuelo.

JPA es una especificación, es decir, no es más que un documento en el cual se plasman las reglas que debe de cumplir cualquier proveedor que desee desarrollar una implementación de JPA, de tal forma que cualquier persona puede tomar la especificación y desarrollar su propia implementación de JPA. Existen varios proveedores como lo son los siguientes:

- Hibernate
- ObjectDB
- TopLink
- EclipseLink
- OpenJPA

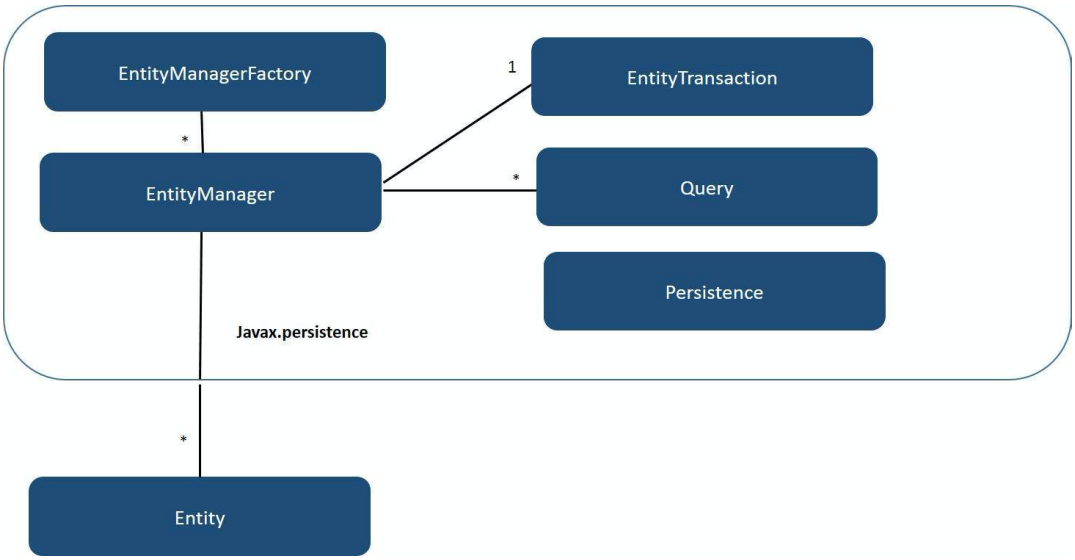
PERSITENCIA DE OBJETOS

JPA representa una simplificación del modelo de programación de persistencia. La especificación JPA define explícitamente la correlación relacional de objetos, en lugar de basarse en implementaciones de correlación específicas del proveedor. JPA crea un estándar para la importante tarea de la correlación relacional de objetos mediante la utilización de anotaciones o XML para correlacionar objetos con una o más tablas de una base de datos. Para simplificar aún más el modelo de programación de persistencia:

- La API EntityManager puede actualizar, recuperar, eliminar o aplicar la persistencia de objetos de una base de datos.
- JPA proporciona un lenguaje de consulta, que amplía el lenguaje de consulta EJB independiente, conocido también como JPQL, el cual puede utilizar para recuperar objetos sin grabar consultas SQL específicas en la base de datos con la que está trabajando.
- El programador no necesita programar código JDBC ni consultas SQL.
- El entorno realiza la conversión entre tipos Java y tipos SQL.

- El entorno crea y ejecuta las consultas SQL necesarias.

ARQUITECTURA JPA



La arquitectura de JPA está diseñada para gestionar Entidades y las relaciones que hay entre ellas. A continuación, detallamos los principales componentes de la arquitectura

Entity: Clase Java simple que representa una fila en una tabla de base de datos con su formato más sencillo. Los objetos de entidades pueden ser clases concretas o clases abstractas. Podemos decir que cada Entidad corresponderá con una tabla de nuestra Base de Datos

Persistence: Clase con métodos estáticos que nos permiten obtener instancias de EntityManagerFactory.

EntityManagerFactory: Es una factoría de EntityManager. Se encarga crear y gestionar múltiples instancias de EntityManager

EntityManager: Es una interface que gestiona las operaciones de persistencia de las entidades, ya sea crear, editar, eliminar, traer de la base de datos una entidad, etc. Es la base de todo proyecto de JPA. A su vez trabaja como factoría de las Querys.

Query: Es una interface para obtener la relación de objetos que cumplen un criterio

EntityTransaction: Agrupa las operaciones realizadas sobre un EntityManager en una única transacción de Base de Datos

MAPEO CON ANOTACIONES

Como sabemos 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. JPA nos permite realizar dicha correlación de forma sencilla, realizando 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 (anotaciones). A estos objetos, las cuales son clases comunes y corrientes, los llamaremos desde ahora entidades.

Las anotaciones nos permiten configurar el mapeo de una entidad dentro del mismo archivo donde se declara la clase, de este modo, relaciona las clases contra las tablas y los atributos contra las columnas.

Las anotaciones comienzan con el símbolo "@" seguido de un identificador. Las anotaciones son utilizadas antes de la declaración de clase, propiedad o método. A continuación, se detallan las principales:

@Entity: Declara la clase como una Entidad

@Table: Declara el nombre de la Tabla con la que se mapea la Entidad

@Id: Declara un atributo como la clave primaria de la Tabla

@GeneratedValue: Declara como el atributo que va a ser al clave primaria va a ser inicializada. Manualmente, Automático o a partir de una secuencia

@Column: Declara que un atributo se mapea con una columna de la tabla

@Enumerated: Declara que un atributo es de alguno de los valores definidos en un Enumerado (lista de valores constantes). Los valores de un tipo enumerado tienen asociado implícitamente un tipo ordinal que será asociada a la propiedad de este tipo.

@Temporal: Declara que se está tratando de un atributo que va a trabajar con fechas, entre paréntesis, debemos especificarle que estilo de fecha va a manejar en la base de datos: @Temporal(TemporalType.DATE), @Temportal(TemporalType.TIME), @Temporal(TemporalType.TIMESTAMP)

DECLARAR ENTIDADES CON @ENTITY

Como ya discutimos hace un momento, las entidades son simples clases Java como cualquier otra, sin embargo, JPA debe de ser capaz de identificar que clases son entidades para de esta forma poder administrarlas. Es aquí donde nace la importancia de la anotación @Entity, esta anotación se debe de definir a nivel de clase y sirve únicamente para indicarle a JPA que esa clase es una Entity, veamos el siguiente ejemplo:

```
public class Empleado {
private Long id;
private String nombre;
}
```

En el ejemplo vemos una clase común y corriente la cual representa a un Empleado, hasta este momento la clase Empleado, no se puede considerar una entidad, pues a un no tiene la anotación @Entity que la señale como tal. Ahora bien, si a esta misma clase le agregamos la anotación @Entity le estaremos diciendo a JPA que esta clase es una entidad y deberá ser administrada por el EntityManager, veamos el siguiente ejemplo:

```
@Entity
public class Empleado {
private Long id;
private String nombre;
}
```

En este punto la clase ya se puede considerar una Entidad.

DEFINIR LLAVE PRIMARÍA CON @ID

Al igual que en las tablas, las entidades también requieren un identificador o clave primaria(ID). Dicho identificador deberá de diferenciar a la entidad del resto. Como regla general, todas las entidades deberán definir un ID, de lo contrario provocaremos que el EntityManager marque error a la hora de instanciarlo.

El ID es importante porque será utilizado por el EntityManager a la hora de persistir un objeto, y es por este que puede determinar sobre que registro hacer el select, update o delete.

```
@Entity
public class Empleado {

@Id
private Long id;

private String nombre;

}
```

Se ha agregado @Id sobre el atributo id, de esta manera, cuando el EntityManager inicie sabrá que el campo id es el Identificador de la clase Empleado.

ANOTACIÓN @GENERATEDVALUE

Esta anotación se utiliza cuando el ID es autogenerado (Identity) como en el caso de MySQL. JPA cuenta con la anotación @GeneratedValue para indicarle a JPA que regla de autogeneración de la lleva primaria vamos a utilizar.

Identity

Esta estrategia es la más fácil de utilizar pues solo hay que indicarle la estrategia y listo, no requiere nada más, JPA cuando persista la entidad no enviará este valor, pues asumirá que la columna es auto generada. Esto provoca que el contador de la columna incremente en 1 cada vez que un nuevo objeto es insertado.

```
@Entity
public class Empleado {

@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;

private String nombre;

}
```

MAPEO DE FECHAS CON @TEMPORAL

Mediante la anotación @Temporal es posible mapear las fechas con la base de datos de una forma simple. Una de las principales complicaciones cuando trabajamos con fecha y hora es determinar el formato empleado por el manejador de base de datos. Sin embargo, esto ya no será más problema con @Temporal.

Mediante el uso de @Temporal es posible determinar si el atributo almacena Hora, Fecha U hora y Fecha. Para esto podemos utilizar la clase Date o Calendar. Se pueden establecer tres posibles valores para la anotación:

DATE: Acotara el campo solo a la Fecha, descartando la hora.

```
@Temporal(TemporalType.DATE)
```

TIME: Acotara el campo solo a la Hora, descartando a la fecha.

```
@Temporal(TemporalType.TIME)
```

TIMESTAMP: Toma la fecha y hora.

```
@Temporal(TemporalType.TIMESTAMP)
```

Ejemplo:

```
@Entity
public class Persona {
@Id
private Long id;
private String nombre;
@Temporal(TemporalType.DATE)
private Date fechaNacimiento;
}
```

LAS RELACIONES

Como sabemos en Java, los objetos pueden estar relacionados entre sí mediante las relaciones entre clases y sabemos que en MySQL las tablas también están relacionadas entre sí. Es por esto que JPA, nos da 4 anotaciones para cuando tenemos una relación entre dos clases en Java y le queremos explicar a la base de datos la relación entre las tablas y sus registros.

Estas anotaciones solo van a afectar a las tablas, sirven para especificar como se van a relacionar los registros de una tabla, con los registros de otra tabla. Recordemos que las anotaciones cumplen el propósito de "traducir" nuestro código de Java para que lo entienda la base de datos, por lo que las anotaciones, no van a afectar nunca a nuestro código.

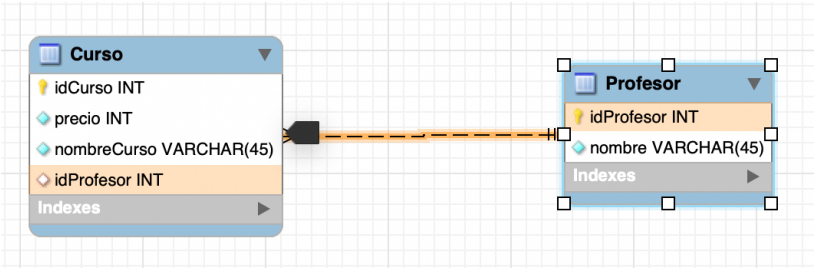
Entonces, supongamos que tenemos dos clases, Curso y Profesor, entre las cuales existe una relación de `1` a `1`. Un Curso por lo tanto tiene `1` Profesor y un Profesor pertenece a un Curso. Esto en nuestro código Java sería algo así:

```
@Entity
public class Profesor {
@Id
private Long id;
private String nombre;
}
@Entity
public class Curso {
@Id
private Long id;
private Integer precio;
private String nombreCurso;
private Profesor profesor;
}
```

Por ahora lo único que hemos creado es una referencia a la clase Profesor sin utilizar JPA para nada. El siguiente paso será anotar la clase con anotaciones de JPA para que se construya la relación a nivel de persistencia.

```
@Entity
public class Curso {

@Id
private Long id;

private Integer precio;

private String nombreCurso;

@OneToOne
private Profesor profesor;

}
```

Con esta anotación nos quedará unas tablas en MySQL de la siguiente manera:



Como podemos observar en la tabla Curso, existe una llave foránea de la tabla Profesor, de la misma manera que en nuestra clase Curso existe un objeto de tipo Profesor.

La relación OneToOne en nuestra tablas va a especificar que para un registro de un Curso, solo hay un registro de un Profesor. En otras palabras sería que un Curso no puede tener dos Profesores o que a cada Curso, solo podemos asignarle/persistir un Profesor.

A partir de estos conceptos definimos 4 tipos principales de relaciones entre entidades:

@OneToOne: usamos esta anotación cuando tenemos una relación de `1 a 1` entre dos clases / tablas.

@ManyToOne: usamos esta anotación cuando tenemos una relación de `n a 1` entre dos clases / tablas. Por ejemplo, muchos Álbumes pueden pertenecer a un Autor.

```
@ManyToOne
private Autor autor;
```

La relación ManyToOne en nuestra tablas va a especificar que para uno o varios registros de Álbumes va a haber un Autor. En otras palabras sería que uno o más Álbumes van a tener el mismo Autor. Esto nos daría la posibilidad de que, a uno o muchos Álbumes asignarle el mismo Autor, esta posibilidad no existe con la OneToOne, ya que a cada registro solo le podemos asignar un registro.

@OneToMany: usamos esta anotación cuando tenemos una relación de `1 a n` entre dos clases / tablas. Por ejemplo, un Curso tiene muchos Alumnos. Es importante recordar que cuando hablamos de que una clase tiene **muchos**(Many) o n de algo, usamos una colección para representar esa relación en Java, sino usaríamos un solo objeto.

```
@OneToMany
private List<Alumno> alumnos;
```

La relación OneToMany en nuestra tablas va a especificar que para un registro de un Curso, va a haber varios registros de Alumnos. En otras palabras sería que un Curso puede tener uno o más Alumnos. Esto nos daría la posibilidad de, al mismo registro de Curso asignarles varios Alumnos.

@ManyToMany: usamos esta anotación para entidades que están relacionadas con muchos elementos de un tipo determinado, pero al mismo tiempo, estos últimos registros no son exclusivos de un registro en particular, si no que pueden ser parte de varios. Por lo tanto, tenemos una Entidad A, la cual puede estar relacionada como muchos registros de la Entidad B, pero al mismo tiempo, la Entidad B puede pertenecer a varias instancias de la Entidad A.

Algo muy importante a tomar en cuenta cuando trabajamos con relaciones @ManyToMany o @OneToMany, es que en realidad este tipo de relaciones no existen físicamente en la base de datos, y en su lugar, es necesario crear una tabla intermedia que relacione las dos entidades.

```
@ManyToMany
private List<Alumno> alumnos;
```

Nota: que clase va a tener la referencia a la otra clase va a ser decisión del programador.

Tabla intermedia

El concepto de tabla intermedia se presenta cuando tenemos una entidad que va a pertenecer a varias instancias de otra entidad, por ejemplo, un Curso tiene muchos Alumnos. Esto en SQL sería que un registro tiene varios registros asignados a el mismo.

El problema es que en SQL solo podemos poner un dato por columna, supongamos que el Curso tiene 3 alumnos con los identificadores(id) 1,2,3. Nosotros no podemos tener una columna que tenga 3 valores separados. Lo que podríamos hacer es que se repita el registro de Curso 3 veces con los 3 identificadores, pongamos un ejemplo de una tabla que cumpla ese requisito:

ID	Precio	NombreCurso	IdAlumnos
1	1000	Programación	1
1	1000	Programación	2
1	1000	Programación	3

Esto parecería estar bien, pero si pensamos en las reglas de SQL, no podemos tener dos identificadores iguales en la misma y en nuestra tabla Curso hay 3 veces el mismo identificador para el Curso, ya que necesitamos que se repita.

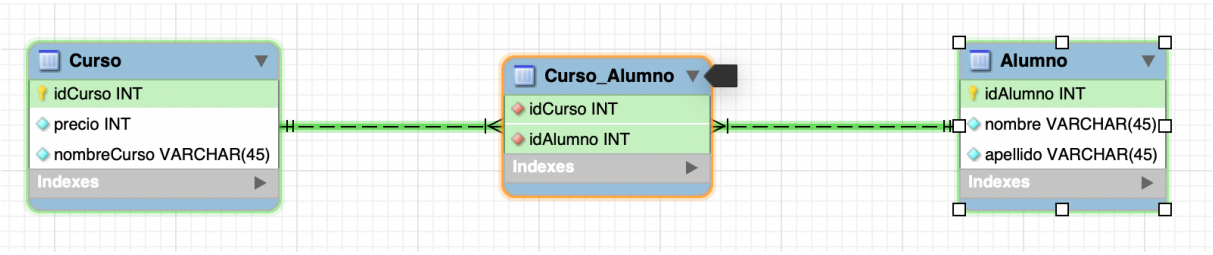
Aquí es donde entra la tabla intermedia, la tabla intermedia va a ser una tabla que se va a crear, además de las dos tablas que ya tenemos y que va a encargarse de relacionar estas dos tablas.

La tabla intermedia se denomina típicamente "tabla de unión". Esta tabla se utiliza para vincular las otras dos tablas. Para ello, tiene dos campos que hacen referencia a la clave principal de cada una de las otras dos tablas. Veamos un ejemplo de una tabla intermedia:

Id_curso	Id_alumno
1	1
1	2
1	3

Como podemos observar la tabla intermedia solo tiene dos columnas, el id del curso y el id del alumno. Esta tabla no toma las columnas como llaves primarias, sino como llaves foráneas, esto nos permite repetir el identificador de Curso para asignárselo a los 3 alumnos.

Las tablas en MySQL se verían así:



Notemos que en la tabla Curso no tiene una columna que haga referencia a Alumno, ni Alumno a Curso, si no que es la tabla intermedia la encargada de hacer el cruce entre las dos tablas.

Nota: recordemos que esta tabla intermedia se va a generar con las anotaciones @OneToMany y @ManyToMany.

RELACIONES JPA Y UML

En Java, nosotros tenemos dos posibles relaciones entre clases, uno a uno o de uno a muchos, además son las dos que podemos representar en código. Por ejemplo, la relación muchos a uno, no existe en Java porque no podemos poner que una clase va a ser un List para representar el muchos. Java al ver que hay una referencia a una clase con un objeto solo, la va a tomar como una relación de uno a uno.

Esto nos va a generar que cuando veamos un UML de nuestro proyecto JPA, no veamos las relaciones ManyToOne o ManyToMany, ya que como dijimos solo existen las uno a uno o uno a muchos.

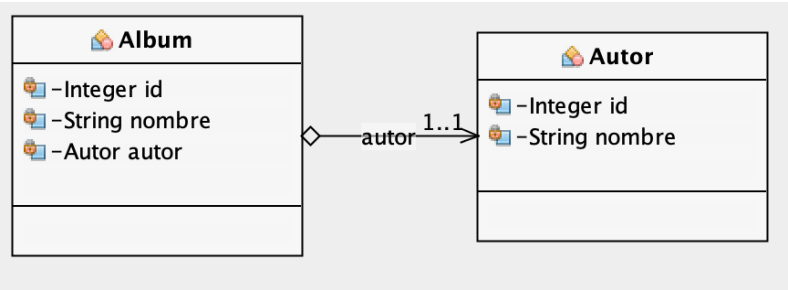
Pongamos un ejemplo, tenemos la Clase Autor y la clase Álbum, vamos a decir que muchos Álbumes existe el mismo autor, por lo que sería una ManyToOne. Si la representamos en código sería así:

```
@ManyToOne
private Autor autor;
```

Ahora si tuviéramos una OneToOne, la representación en codigo sería esta:

```
@OneToOne
private Autor autor;
```

Si miramos el código, podemos observar que para las dos relaciones escribimos el mismo código, entonces para Java la relación ManyToOne la va a representar como una OneToOne (1…1) en UML y lo mismo nos pasaría con la OneToMany y la ManyToMany.



Por lo que si nos encontramos con un UML que tiene una relación uno a uno (1…1), pensar que puede ser una OneToOne o una ManyToOne, o que si tiene una relación uno a muchos (1…n), puede ser una OneToMany o una ManyToMany.

Esto es porque las relaciones muchos a uno y muchos a muchos son propias de MySQL, no de Java. Entonces es importante, que a la hora de pensar en que anotaciones le vamos a poner a nuestras entidades, pensemos en las tablas, ya que estamos trabajando como va a ser la relación entre las tablas y no las clases.

PERSISTENCIA EN JPA CON ENTITYMANAGER

Ahora que entendemos como a través de las anotaciones del ORM podemos unificar las tablas de la base de datos con los objetos, que pasan a llamarse entidades. Ahora tenemos que ver como hacemos para poder guardar, editar, eliminar, etc. a esas entidades en la base de datos.

JPA tiene como interface medular al EntityManager, el cual es el componente que se encarga de controlar el ciclo de vida de todas las entidades definidas en la unidad de persistencia. El EntityManager nos dará la posibilidad de poder crear, borrar, actualizar y consultar todas estas entidades de la base de datos. También es la clase por medio de la cual se controlan las transacciones. Los EntityManager son configurados siempre a partir de las unidades de persistencia definidas en el archivo persistence.xml.

```
EntityManager em =
Persistence.createEntityManagerFactory("nombreUnidadDePersistencia").creat
eEntityManager();
```

En esta línea se puede ver que se obtiene una instancia de la Interfaz EntityManagerFactory, mediante la clase Persistence, esta última recibe como parámetro el nombre de la unidad de persistencia que definimos en el archivo persistence.xml. Una vez con el EntityManagerFactory se obtiene una instancia de EntityManager para finalmente ser retornada para ser utilizada.

OPERACIONES ENTITYMANAGER

Las entidades pueden ser cargadas, creadas, actualizadas y eliminadas a través del EntityManager. Vamos a mostrar los métodos del EntityManager que nos permiten lograr estas operaciones.

Persist()

Este método nos deja persistir una entidad en nuestra base de datos. Persistir es la acción de preservar la información de un objeto de forma permanente, en este caso en una base de datos, pero a su vez también se refiere a poder recuperar la información del mismo para que pueda ser nuevamente utilizado.

Antes de ver como persistimos un objeto, también tenemos que entender el concepto de transacciones, ya que para persistir un objeto en la base de datos, la operación debe estar marcada como una transacción.

Una transacción es un conjunto de operaciones sobre una base de datos, que suelen crear, editar o eliminar un registro de la base de datos, que se deben ejecutar como una unidad. Por lo que una consulta a la base de datos no se la considera una transacción.

Entendiendo esto veamos un ejemplo del método Persist():

```
// Creamos un EntityManager
EntityManager em =
Persistence.createEntityManagerFactory("nombreUnidadDePersistencia").createEnt
ityManager();
//Creamos un objeto Alumno y le asignamos un nombre
Alumno alumno = new Alumno();
a1.setNombre(“Nahuel”);
//Iniciamos una transacción con el método getTransaction().begin();
em.getTransaction().begin();
//Persistimos el objeto
em.persist(alumno);
//Terminamos la transacción con el método commit. Commit en programación
significa confirmar un conjunto de cambios, en este caso persistir el objeto
em.getTransaction().commit();
```

Find()

Este método se encarga de buscar y devolver una Entidad en la base de datos, a través de su clave primaria(Id). Para ello necesita que le pasemos la clave y el tipo de Entidad a buscar.

```
// Creamos un EntityManager
EntityManager em =
Persistence.createEntityManagerFactory("nombreUnidadDePersistencia").createEnt
ityManager();
// Usamos el método find para buscar una persona con el id 123 en nuestra base
de datos
Persona persona = em.find(Persona.class, 123);
```

De esta manera podremos obtener una Persona de la base de datos para usar ese objeto como queramos.

Merge()

Este método funciona igual que el método persist pero, sirve para actualizar una entidad en la base de datos.

```
EntityManager em =
Persistence.createEntityManagerFactory("nombreUnidadDePersistencia").createEnt
ityManager();
//Usamos el método find para buscar el alumno a editar
Alumno alumno = em.find(Alumno.class,1234);
//Le asignamos un nuevo nombre
alumno.setNombre(Francisco);
em.getTransaction().begin();
//Actualizamos el alumno
em.merge(alumno);
em.getTransaction().commit();
```

Remove()

Este método se encarga de eliminar una entidad de la base de datos.

```
EntityManager em =
Persistence.createEntityManagerFactory("nombreUnidadDePersistencia").createEnt
ityManager();
//Usamos el método find para buscar el alumno a borrar
Alumno alumno = em.find(Alumno.class,1234);
em.getTransaction().begin();
//Borramos el alumno
em.remove(alumno);
em.getTransaction().commit();
```

JAVA PERSISTENCE QUERY LANGUAGE (JPQL)

Es un lenguaje de consulta orientado a objetos independiente de la plataforma definido como parte de la especificación Java Persistence API (JPA). JPQL es usado para hacer consultas contra las entidades almacenadas en una base de datos relacional. Está inspirado en gran medida por SQL, y sus consultas se asemejan a las consultas SQL en la sintaxis, pero opera con objetos entidad de JPA en lugar de hacerlo directamente con las tablas de la base de datos.

CLAUSULAS SELECT – FROM

La cláusula FROM define de qué entidades se seleccionan los datos. Cualquier implementación de JPA, mapea las entidades a las tablas de base de datos correspondientes. Esto significa que vamos a utilizar el nombre de las entidades en vez del nombre de las tablas y los atributos de las entidades en vez de las columnas de las tablas.

La sintaxis de una cláusula FROM de JPQL es similar a SQL pero usa el modelo de entidad en lugar de los nombres de tabla o columna. El siguiente fragmento de código muestra una consulta JPQL simple en la que selecciono todas las entidades Autor.

```
SELECT a FROM Autor a;
```

En la query se ve que, se hace referencia a la entidad Autor en lugar de la tabla de autor y se le asigna la variable de identificación **a**. La variable de identificación a menudo se llama alias y es similar a una variable en su código Java.

Se utiliza en todas las demás partes de la consulta para hacer referencia a esta entidad. Por ejemplo, si queremos seleccionar un atributo de la entidad Autor, en vez de todos, usaríamos el alias asi:

```
SELECT a.nombre, a.apellido FROM Autor a;
```

CLAUSULA WHERE

La sintaxis es muy similar a SQL, pero JPQL admite solo un pequeño subconjunto de las características de SQL.

JPQL admite un conjunto de operadores básicos para definir expresiones de comparación. La mayoría de ellos son idénticos a los operadores de comparación admitidos por SQL, y puede combinarlos con los operadores lógicos AND, OR y NOT en expresiones más complejas.

Operadores:

- Igual: `author.id = 10`
- Distinto: `author.id <> 10`
- Mayor que: `author.id > 10`
- Mayor o Igual que: `author.id => 10`
- Menor que: `author.id < 10`
- Menor o igual que: `author.id <= 10`
- Between: `author.id BETWEEN 5 and 10`
- Like: `author.firstName LIKE :'%and%'`
- Is null: `author.firstName IS NULL .`
  Se lo puede negar con el operador NOT, para traer todos los que no son nulos
- In: `author.firstName IN ('John','Jane')`
  Va a traer todos los autores con el nombre John o Jane.

UNIR ENTIDADES

Si necesitamos seleccionar datos de más de una entidad, por ejemplo, todos los libros que ha escrito un autor, debe unir las entidades en la cláusula FROM. La forma más sencilla de hacerlo es utilizar las asociaciones definidas de una entidad como en el siguiente fragmento de código.

```
SELECT a FROM Libro a JOIN a.autor b;
```

La definición de la entidad Libro proporciona toda la información necesaria para unirla a la entidad Autor, y no es necesario que proporcione una declaración ON adicional.

También podemos utilizar el operador ".", para navegar a través del atributo de autor de la entidad Libro y traer los libros que tengan un autor con un nombre a elección. Esto generaría una relación implícita entre las dos entidades, sin la necesidad de usar un Join.

```
SELECT a FROM Libro a WHERE a.autor.nombre LIKE : "Homero";
```

PREGUNTAS DE APRENDIZAJE

1) ¿Que significa el acrónimo JPA?

   a) Java Practise API
   b) Java Persitence API
   c) Java Persist API
   d) Ninguna de las anteriores

2) El ORM nos permite interactuar con la base de datos mediante:

   a) Objetos
   b) Clases e Interfaces
   c) Métodos
   d) Solo Clases

3) La anotación @Entity:

   a) Declara la clave primaria de una entidad
   b) Declara la clase como entidad
   c) Declara que el atributo es una columna de la tabla
   d) Ninguna de las anteriores

4) La anotación @Id:

   a) Declara la clave primaria de una entidad
   b) Declara la clase como entidad
   c) Declara que el atributo es una columna de la tabla
   d) Ninguna de las anteriores

5) La anotación @Column:

   a) Declara la clave primaria de una entidad
   b) Declara la clase como entidad
   c) Declara que el atributo es una columna de la tabla
   d) Ninguna de las anteriores

6) La anotación @Temporal:

   a) Declara que se está tratando de un atributo que va a trabajar con fechas
   b) Declara la clase como entidad
   c) Declara que el atributo es una columna de la tabla
   d) Ninguna de las anteriores

7) ¿Cuál de estos no es un tipo de la anotación @Temporal?

   a) DATE
   b) TIME
   c) DATETIME
   d) TIMESTAMP

8) Elegir que anotación se aplica a la siguiente relación, un Perro pertenece a un Dueño:

a) @OneToOne
b) @ManyToMany
c) @ManyToOne
d) @OneToMany

9) Elegir que anotación se aplica a la siguiente relación, un Cliente tiene muchas Facturas:

a) @OneToOne
b) @ManyToMany
c) @ManyToOne
d) @OneToMany

10) Elegir que anotación se aplica a la siguiente relación, muchos Alumnos tienen un Profesor:

a) @OneToOne
b) @ManyToMany
c) @ManyToOne
d) @OneToMany

11) Elegir que anotación se aplica a la siguiente relación, muchas Alumnos tienen muchas Clases:

a) @OneToOne
b) @ManyToMany
c) @ManyToOne
d) @OneToMany

12) ¿Qué interfaz se encargar de persistir, actualizar y borrar las entidades?

a) Persistence
b) EntityManagerFactory
c) EntityManager
d) DriverManager

13) El método que se encarga de persistir las entidades es el método:

a) Persist()
b) Merge()
c) Remove()
d) Find()

14) ¿Cuál es el lenguaje de consultas en JPA?

a) SQL
b) MYSQL
c) JPQL
d) Mongo

EJERCICIOS DE APRENDIZAJE

Para la realización de los ejercicios que se describen a continuación, sigue siendo necesario el conector de MySQL y es necesario tener descargado el Instructivo Unidad de Persistencia.

VER VIDEOS:

A. Introducción JPA – Parte 1
B. Introducción JPA – Parte 2
C. Capa de acceso a datos

1. Sistema de Guardado de una Librería

El objetivo de este ejercicio es el desarrollo de un sistema de guardado de libros en JAVA utilizando una base de datos MySQL y JPA como framework de persistencia.

Creación de la Base de Datos MySQL:

Lo primero que se debe hacer es crear la base de datos sobre el que operará el sistema de reservas de libros. Para ello, se debe abrir el IDE de base de datos que se está utilizando (Workbench) y ejecutar la siguiente sentencia:

```
CREATE  DATABASE libreria;
```

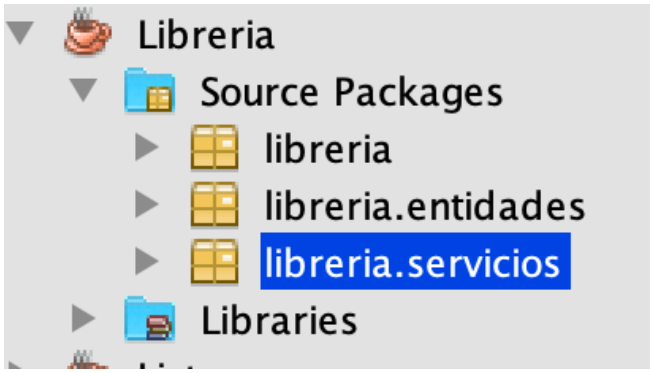
De esta manera se habrá creado una base de datos vacía llamada librería.

Paquetes del Proyecto Java:

Los paquetes que se utilizarán para este proyecto son los siguientes:

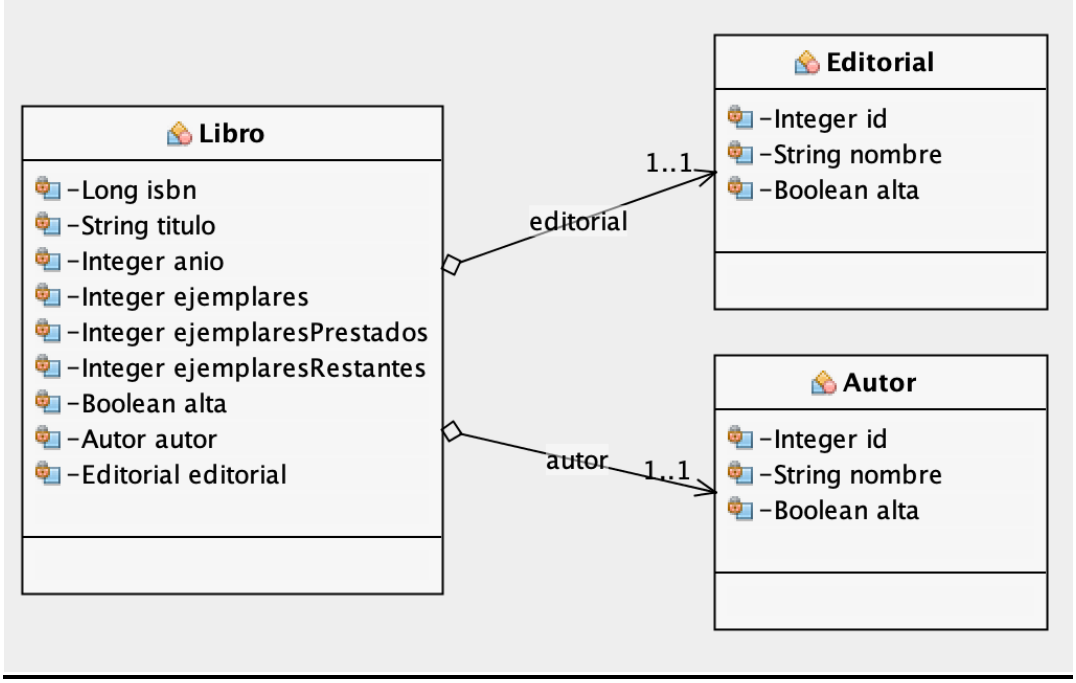
- **entidades**: en este paquete se almacenarán aquellas clases que se quiere persistir en la base de datos.
- **servicios:** en este paquete se almacenarán aquellas clases que llevarán adelante la lógica del negocio. En general se crea un servicio para administrar las operaciones CRUD (Create, Remove, Update, Delele) cada una de las entidades y las consultas de cada entidad.

Nota: En este proyecto vamos a eliminar entidades, pero no es considerado una buena practica. Por esto, además de eliminar nuestras entidades, vamos a practicar que nuestras entidades estén dados de alta o de baja. Por lo que las entidades tendrán un atributo alta booleano, que estará en true al momento de crearlas y en false cuando las demos de baja, que sería cuando se quiere eliminar esa entidad.



a) Entidades

Crearemos el siguiente modelo de entidades:



Entidad Libro

La entidad libro modela los libros que están disponibles en la biblioteca para ser prestados. En esta entidad, el atributo "ejemplares" contiene la cantidad total de ejemplares de ese libro, mientras que el atributo "ejemplaresPrestados" contiene cuántos de esos ejemplares se encuentran prestados en este momento y el atributo "ejemplaresRestantes" contiene cuántos de esos ejemplares quedan para prestar.

Entidad Autor

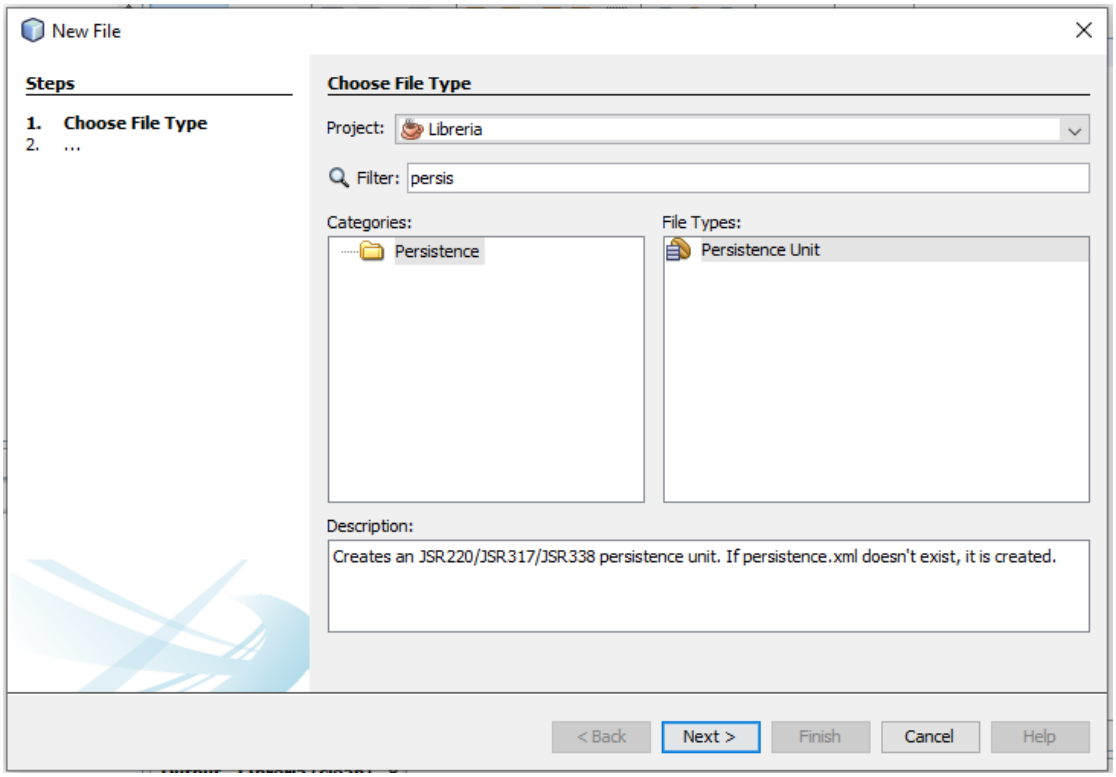
La entidad autor modela los autores de libros.

Entidad Editorial

La entidad editorial modela las editoriales que publican libros.

b) Unidad de Persistencia

Para configurar la unidad de persistencia del proyecto, se recomienda leer el Instructivo Unidad de Persistencia recuerde hacer click con el botón derecho sobre el proyecto y seleccionar nuevo. A continuación, se debe seleccionar la opción de Persistence Unit como se indica en la siguiente imagen.



Base de Datos

Para este proyecto nos vamos a conectar a la base de datos Librería, que creamos previamente.

Generación de Tablas

La estrategia de generación de tablas define lo que hará JPA en cada ejecución, si debe crear las tablas faltantes, si debe eliminar todas las tablas y volver a crearlas o no hacer nada. Recomendamos en este proyecto utilizar la opción: "Create"

Librería de Persistencia

Se debe seleccionar para este proyecto la librería "EclipseLink".

c) Servicios

AutorServicio

Esta clase tiene la responsabilidad de llevar adelante las funcionalidades necesarias para administrar autores (consulta, creación, modificación y eliminación).

EditorialServicio

Esta clase tiene la responsabilidad de llevar adelante las funcionalidades necesarias para administrar editoriales (consulta, creación, modificación y eliminación).

LibroServicio

Esta clase tiene la responsabilidad de llevar adelante las funcionalidades necesarias para administrar libros (consulta, creación, modificación y eliminación).

d) Main

Esta clase tiene la responsabilidad de llevar adelante las funcionalidades necesarias para interactuar con el usuario. En esta clase se muestra el menú de opciones con las operaciones disponibles que podrá realizar el usuario.

e) Tareas a Realizar

Al alumno le toca desarrollar, las siguientes funcionalidades:

1) Crear base de datos Librería
2) Crear unidad de persistencia
3) Crear entidades previamente mencionadas (excepto Préstamo)
4) Generar las tablas con JPA
5) Crear servicios previamente mencionados.
6) Crear los métodos para persistir entidades en la base de datos librería
7) Crear los métodos para dar de alta/bajo o editar dichas entidades.
8) Búsqueda de un Autor por nombre.
9) Búsqueda de un libro por ISBN.
10) Búsqueda de un libro por Título.
11) Búsqueda de un libro/s por nombre de Autor.
12) Búsqueda de un libro/s por nombre de Editorial.
13) Agregar las siguientes validaciones a todas las funcionalidades de la aplicación:
    - Validar campos obligatorios.
    - No ingresar datos duplicados.

EJERCICIOS EXTRAS

Estos van a ser ejercicios para reforzar los conocimientos previamente vistos. Estos pueden realizarse cuando hayas terminado la guía y tengas una buena base sobre lo que venimos trabajando. Además, si ya terminaste la guía y te queda tiempo libre en las mesas, podes continuar con estos ejercicios extra, recordando siempre que no es necesario que los termines para continuar con el tema siguiente. Por ultimo, recordá que la prioridad es ayudar a los compañeros de la mesa y que cuando tengas que ayudar, lo más valioso es que puedas explicar el ejercicio con la intención de que tu compañero lo comprenda, y no sólo mostrarlo. ¡Muchas gracias!

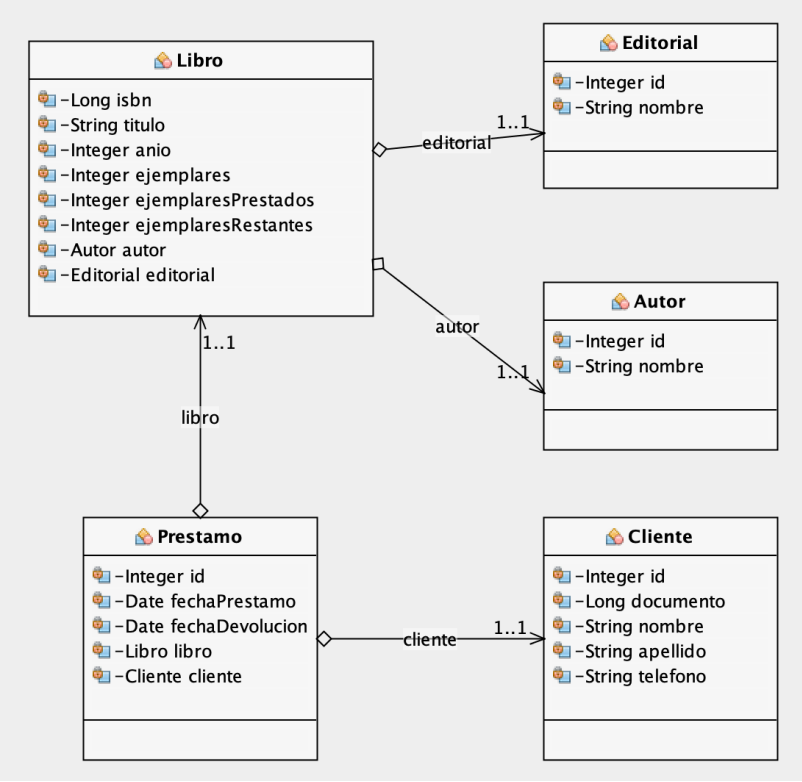
1. Sistema de Reservas de una Librería

Vamos a continuar con el ejercicio anterior. Ahora el objetivo de este ejercicio es el desarrollo de un sistema de reserva de libros en JAVA. Para esto vamos a tener que sumar nuevas entidades a nuestro proyecto en el paquete de entidades y crearemos los servicios de esas entidades.

Usaremos la misma base de datos y se van a crear las tablas que nos faltan. Deberemos agregar las entidades a la unidad de persistencia.

a) Entidades

Crearemos el siguiente modelo de entidades:



Entidad Cliente

La entidad cliente modela los clientes (a quienes se les presta libros) de la biblioteca. Se almacenan los datos personales y de contacto de ese cliente.

Entidad Préstamo

La entidad préstamo modela los datos de un préstamo de un libro. Esta entidad registra la fecha en la que se efectuó el préstamo y la fecha en la que se devolvió el libro. Esta entidad también registra el libro que se llevo en dicho préstamo y quien fue el cliente al cual se le prestaron.

b) Servicios

ClienteServicio

Esta clase tiene la responsabilidad de llevar adelante las funcionalidades necesarias para administrar clientes (consulta, creación, modificación y eliminación).

PrestamoServicio

Esta clase tiene la responsabilidad de llevar adelante las funcionalidades necesarias para generar prestamos, va a guardar la información del cliente y del libro para persistirla en la base de datos. (consulta, creación, modificación y eliminación).

c) Tareas a Realizar

Al alumno le toca desarrollar, las siguientes funcionalidades:

1. Creación de un Cliente nuevo
2. Crear entidad Préstamo
3. Registrar el préstamo de un libro.
4. Devolución de un libro
5. Búsqueda de todos los prestamos de un Cliente.
6. Agregar validaciones a todas las funcionalidades de la aplicación:
    - Validar campos obligatorios.
    - No ingresar datos duplicados.
    - No generar condiciones inválidas. Por ejemplo, no se debe permitir prestar más ejemplares de los que hay, ni devolver más de los que se encuentran prestados. No se podrán prestar libros con fecha anterior a la fecha actual, etc.