CURSO DE PROGRAMACIÓN FULL STACK

APÉNDICE D

PERSISTENCIA CON JDBC Y JPA

EGG

CONECTIVIDAD A BASES DE DATOS DE JAVA (JDBC)

ESTABLECER UNA CONEXIÓN CON LA BASE DE DATOS

Para comunicar con una base de datos utilizando JDBC, se debe en primer lugar establecer una conexión con la base de datos a través del driver JDBC apropiado. El API JDBC especifica la conexión en la interfaz java.sql.Connection.

La clase *DriverManager* permite obtener objetos *Connection* con la base de datos.

Para conectarse es necesario proporcionar:

- URL de conexión, que incluye:
  - Nombre del host donde está la base de datos.
  - Nombre de la base de datos a usar.
- Nombre del usuario en la base de datos.
- Contraseña del usuario en la base de datos.

El siguiente código muestra un ejemplo de conexión JDBC a una base de datos MySQL:

```
Connection connection;
(…)
try {
      String url = “jdbc:mysql://hostname/database-name”;
      connection = DriverManager.getConnection(url, “user”, “passwd”);
} catch (SQLException ex) {
      connection = null;
      ex.printStackTrace();
      System.out.println(“SQLException: ” + ex.getMessage());
      System.out.println(“SQLState: ” + ex.getSQLState());
      System.out.println(“VendorError: ” + ex.getErrorCode());
}
```

En este ejemplo, la clase *DriverManager* intenta establecer una conexión con la base de datos *database-name* utilizando el driver JDBC que proporciona MySQL. Para poder acceder al RDBMS MySQL es necesario introducir un login y un password válidos. En el API JDBC, hay varios métodos que pueden lanzar la excepción SQLException.

Objeto Connection

Representa el contexto de una conexión con la base de datos, es decir:

- Permite obtener objetos Statement para realizar consultas SQL.
- Permite obtener metadatos acerca de la base de datos (nombres de tablas, etc.)
- Permite gestionar transacciones.

Objeto Statement

Los objetos Statement permiten realizar consultas SQL en la base de datos.

- Se obtienen a partir de un objeto Connection.
- Tienen distintos métodos para hacer consultas:
    - executeQuery: para sentencias SQL que recuperen datos de un único objeto ResultSet. Es usado para leer datos (típicamente consultas SELECT).
    - executeUpdate: para realizar actualizaciones que no devuelvan un ResultSet. Es usado para insertar, modificar o borrar datos (típicamente sentencias INSERT, UPDATE y DELETE).

Para crear una sentencia debemos ejecutar:

```
Statement statement = connectionn.createStatement();
```

Objeto ResultSet

Para realizar una consulta a la base de datos se utiliza el objeto *Statement* de la siguiente manera:

```
String query = "SELECT * FROM table_name"
Statement statement = connectionn.createStatement();
ResultSet rs = statement.executeQuery(query);
```

Devuelve un objeto *ResultSet* que representa el resultado de una consulta:

- Está compuesto por filas.
- Se leen secuencialmente las filas, desde el principio hacia el final.
- En cada fila se recuperan los valores de las columnas mediante métodos.
    - El método a usar depende del tipo de datos, y recibe el nombre o número (empezando en 1) de columna como parámetro: getString(), getInt(), getDate(), etc.

Para realizar una consulta al a base de datos se utiliza el objeto Statement de la siguiente manera:

```
Statement statement = connection.createStatement();

int inserts = statement.executeUpdate("INSERT INTO bd.table_name (id, col1, col2, col3,col4) VALUES('1234', ‘’, null, null, '')");

int updates = statement.executeUpdate("UPDATE bd.table_name set col1='nuevo valor' where id = '1234'");
```

Tipos de ResultSet

El tipo de un ResultSet especifica los siguiente acerca del ResultSet:

- Si el ResultSet es desplazable.
- Los tipos de los ResultSets de JDBC definidos por constantes en la interfaz ResultSet.

Las definiciones de estos tipos de ResultSet son las siguientes:

- TYPE_FORWARD_ONLY: Un cursor que solo puede utilizarse para procesar desde el principio de un ResultSet hasta el final del mismo. Este es el tipo por omisión.
- TYPE_SCROLL_INSENSITIVE: Un cursor que se puede emplear para desplazares a través de un ResultSet. Este tipo de cursor es insensible a los cambios efectuados en la base de datos mientras está abierto. Contiene filas que satisfacen la consulta cuando esta se procesa o cuando se extraen datos.
- TYPE_SCROLL_SENSITIVE: Un cursor que puede utilizarse para el desplazamiento en diversas formas a través de un ResultSet. Este tipo de cursor es sensible a los cambios efectuados en la base de datos mientras está abierto. Los cambios en la base de datos tienen un impacto directo sobre los datos del ResultSet.

Liberación de recursos

- Las consultas en progreso consumen recursos tanto en la base de datos como en el programa cliente.
- Se pueden liberar los recursos consumidos por objetos ResultSet y Statement mediante su método close().
- Los objetos ResultSet se cierran automáticamente cuando se cierra su objeto Statement asociado, o se hace una nueva consulta sobre él.

Por lo tanto, una vez que se tiene un objeto de tipo *Connection*, se pueden crear sentencias, *statements* ejecutables. Cada una de estas sentencias puede devolver cero o más resultados, que se devuelven como objetos de tipo *ResultSet*. La siguiente tabla muestra la misma lista de clases e interfaces junto con una breve descripción.

Clase/Interface	Descripción
Driver	Permite conectarse a una base de datos: cada gestor de base de datos requiere un driver distinto
DriverManager	Permite gestionar todos los drivers que están disponibles en una aplicación concreta. Es el objeto que mantiene las funciones de administración de las operaciones que se realizan con la base de datos
DriverPropertyInfo	Proporciona diversa información acerca de un driver
Connection	Representa una conexión con una base de datos. Representa la conexión con la base de datos. Es el objeto que permite realizar las consultas SQL y obtener los resultados de dichas consultas. Es el objeto base para la creación de los objetos de acceso a la base de datos.
DatabaseMetadata	Proporciona información acerca de una Base de Datos, como las tablas que contiene, etc.
Statement	Se utiliza para enviar las sentencias SQL simples, aquellas que no necesitan parámetros, a la base de datos.
PreparedStatement	Permite ejecutar sentencias SQL con parámetros de entrada. Tiene una relación de herencia con el objeto Statement, añadiéndole la funcionalidad de poder utilizar parámetros de entrada.
CallableStatement	Permite ejecutar sentencias SQL con parámetros de entrada y salida, típicamente procedimientos almacenados
ResultSet	Contiene las filas o registros obtenidos al ejecutar un SELECT

ResultSetMetadata	Permite obtener información sobre un ResultSet, como el número de columnas, sus nombres, etc.

¿CÓMO CONECTAR JAVA CON MYSQL EN NETBEANS?

Para conocer más acerca de cómo realizar la conexión de nuestra aplicación Java con MySQL usando Netbeans se puede consultar el siguiente link donde se muestra el paso a paso a través de un ejemplo: https://bit.ly/2woxKhs

PERSISTENCIA EN JAVA CON JPA

Clase entitymanager

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, y es mediante esta interface que se pueden realizar las operaciones básicas de una base de datos, como consultar, actualizar, borrar, crear. 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("namePersistentUnit").createEntityManager();
```

Ejemplo

```
EntityManager em =

Persistence.createEntityManagerFactory("PersonalPU").createEntityManager();


List<Empleado> empleados =

em.createQuery("SELECT * FROM Empleado e ORDER BY e.nombre").getResultList();
```

Transacciones del EntityManager

Además de recuperar objetos (consultas SELECT), JPQL soporta consultas de actualización (UPDATE) y borrado (DELETE). Para que tengan efecto deben estar dentro de una transacción.

```
em.getTransaction().begin();

int actualizados = em.createQuery("UPDATE Empleado e SET e.legajo = 2330").executeUpdate();

int eliminados = em.createQuery("DELETE Empleado e WHERE e.nombre = 'Martin Rumbo'").executeUpdate();

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

Por lo tanto, todas las operaciones que impliquen un cambio en los datos de la base de datos deben estar dentro del marco de una transacción.

Inicio:

```
em.getTransaction().begin();
```

Fin:

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

Revertir:

```
em.getTransaction().rollback();
```

Operaciones del EntityManager

Las entidades pueden ser cargadas, creadas, actualizadas y eliminadas a través del EntityManager:

Carga:

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

Creación:

```
em.persist(empleado);
```

Modificación:

```
em.merge(empleado);
```

Eliminación:

```
em.remove(empleado);
```

ANOTACIONES JPA

En la definición de la clase, se escribe la configuración con anotaciones. Las anotaciones se utilizan para las clases, propiedades y métodos. Las anotaciones comienzan con el símbolo "@". Las anotaciones son declaradas antes de una clase, propiedad o método. Todas las anotaciones de JPA se definen en el paquete *javax.persistence*.

Anotación	Descripción
`@Entity`	Declara la clase como una entidad o una tabla.
`@Table`	Declara nombre de la tabla.
`@Id`	Especifica la propiedad, el uso de la identidad (la clave principal de una tabla de la clase).
`@GeneratedValue`	Especifica el modo en que la identidad se puede inicializar como atributo automático, manual o valor tomado de la tabla de secuencias.
`@Transient`	Especifica la propiedad que no es constante, es decir, el valor nunca se almacena en la base de datos. Define un atributo como no persistente.
`@Column`	Declara el nombre de la columna para una tabla.
`@JoinColumn`	Especifica la entidad asociación o entidad colección. Esto se utiliza en asociaciones muchos-a-uno y uno-a-muchos. Declara el nombre de la columna de la tabla.
`@UniqueConstraint`	Especifica los campos y las restricciones unique para la clave primaria o la secundaria.
`@ColumnResult`	Hace referencia al nombre de una columna de la consulta SQL que utiliza cláusula select.
`@SequenceGenerator`	Especifica el modo en que la identidad se puede inicializar como atributo automático, manual o valor tomado de la tabla de secuencias

DECLARAR ENTIDADES CON @ENTITY

Una de las grandes ventajas de JPA es que nos permite manipular la base de datos a través de objetos, estos objetos son conocidos como Entity, las cuales son clases comunes y corrientes. Estas clases tiene la particularidad de ser clases que están mapeadas contra una tabla de la base de datos, dicho mapeo se lleva a cabo generalmente mediante Anotaciones, las cuales brindan los suficientes metadatos como para poder relacionar las clases contra las tablas y las propiedades contra las columnas. Es de esta forma que JPA es capaz de interactuar con la base de datos a través de las clases.

La anotación @Entity se debe de definir a nivel de clase y sirve únicamente para indicarle a JPA que esa clase es una Entity:

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

    /**
     * GETs and SETs
     */
}
```

MAPEO DE TABLAS CON @TABLE

La anotación @Table es utilizada para indicarle a JPA contra que tabla debe de mapear una entidad, de esta manera cuando se realice una persistencia, borrado o select de la entidad, JPA sabrá contra que tabla de la base de datos deberá interactuar.

```
@Entity
@Table(
    name = "empleados",
    schema = "personal",
    indexes = {@Index(nombre = "nombre_index", columnList = "nombre",unique =
true)}
)
public class Empleado {
    private Long id;
    private String nombre;
    /**
     * GETs and SETs
     */
}
```

Observemos que la clase *Empleado* mapea contra la tabla *empleados*, se le dice que la tabla está en el schema *personal* y que tiene un index llamado nombre_index para ordenar el nombre del empleado. En el caso del index para el nombre del empleado, estamos diciendo que el nombre deberá ser único

NOTA: La notación @Table no es obligatoria, pero el hecho de no ponerla supone que JPA determine que la tabla se llama igual que la clase y la buscara en el schema en que estas logueado.

DEFINIR LLAVE PRIMARÍA CON @ID

Al igual que en las tablas, las entidades también requieren un identificador(@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 utilizando por EntityManager a la hora de persistir un objeto, y es por este que puede determinar sobre que registro hacer el select, update o delete. JPA soporta ID simples de un solo campo o ID complejos, formados por más de un campo.

```
@Entity
@Table(
    name = "empleados",
    schema = "personal",
    indexes = {@Index(nombre = "nombre_index", columnList = "nombre",unique =
true)}
)
public class Empleado {
    @Id
    private Long id;
    private String nombre;
¡
    /**
     * GETs and SETs
     */
}
```

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 y MS SQL Server. JPA cuenta con la anotación @GeneratedValue para indicarle a JPA que regla de autogeneración de la lleve 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
@Table(
    name = "empleados",
    schema = "personal",
    indexes = {@Index(nombre = "nombre_index", columnList = "nombre",unique =
true)}
)
public class Empleado {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String nombre;

    /**
     * GETs and SETs
     */
}
```

DEFINICIÓN DE COLUMNAS CON @COLUMN

Una de las principales características cuando trabajamos con base de datos es que todas las tablas tienen Columnas, y dichas columnas esta mapeadas contra los atributos de las entidades, para lo cual es necesario que JPA identifique que columna mapea contra cada atributo de la clase y es aquí donde entra @Column. La anotación @Column permite definir aspectos muy importantes sobre las columnas de la base de datos como lo es el nombre, la longitud, constrains, etc. En caso de no definir esta anotación en los atributos, JPA determinara el nombre de la columna de forma automática mediante el nombre del atributo, por lo que siempre es recomendable establecer esta anotación en todos los atributos de la clase y evitarnos problemas.

```
public class Empleado {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "ID")
    private Long id;

    @Column(name = "NOMBRE", nullable = false, length = 150)
    private String nombre;

    @Column(name = "SALARIO", nullable = false, scale = 2)
    private double salario;

    @Column(name = "FECHA_REGISTRO", updatable = false, nullable = false)
    @Temporal(TemporalType.DATE)
    private Calendar fechaRegistro;

    /**
     * GETs and SETs
     */
}
```

Como se puede observar el atributo id, solo requiere definir la propiedad name, pues por default, ya es única, no nula, no actualizable. Por lo que con definir el nombre será necesario.

En el caso del nombre, se define la propiedad nombre, le indicamos que no puede ser nulo y le estamos definiendo una longitud de 150 caracteres.

En el caso del salario también ponemos el nombre, le indicamos que el sueldo puede tener 2 decimales y que no debe de ser nulo.

Finalmente, en la fecha de registro también establecemos el nombre, pero en este caso veamos que el nombre de la columna es diferente que el de la propiedad y es cuando es especialmente necesario definir la propiedad name, le indicamos que no debe de ser nulo y finalmente, la fecha de ingreso no debe de ser actualizable.

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, y es posible utilizar la clase Date o Calendar para estos fines. Se pueden establecer tres posibles valores para la anotación:

```
@Temporal(TemporalType.DATE)
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.TIMESTAMP)
TIMESTAMP: Toma la fecha y hora.
```

MAPEO DE ENUMERACIONES CON @ENUMERATED

Una de las ventajas de utilizar enumeraciones en Java, es podemos limitar los valores posibles para una propiedad, forzando a los desarrolladores a utilizar los valores ya definidos y evitando el margen de error.

Con JPA también es posible utilizar enumeraciones y pueden ser de mucha ayuda para asegurar que los programadores persistan un valor válido dentro de una lista previamente definida. JPA nos permite mediante la anotación @Enumerated definir la forma en que una enumeración será persistida, las cuales se explican a continuación:

- String: permite persistir la enumeración por su nombre, lo que significa que será una columna alfanumérica. La anotación quedaría así:

  ```
  @Enumerated(value = EnumType.STRING)
  ```

- Ordinal: esta estrategia persiste un valor entero que corresponde al valor ordinal o posición de valor en la enumeración. La anotación quedaría de la siguiente manera:

```
    @Enumerated(value = EnumType.ORDINAL)

public class Empleado {
    …
    @Column(name="estado", nullable = false, length = 8 )
    @Enumerated(value = EnumType.STRING)
    private Estado estado;

    /**
     * GETs and SETs
     */
}
```

La propiedad es de tipo *Estado*, la cual es una enumeración y tiene definida la estrategia String. También podemos ver que está definida la anotación @Column para definir las características de la columna. La enumeración *Estado* puede tener definidos los posibles valores ACTIVO e INACTIVO.

```
public enum Estado {
    ACTIVO,
    INACTIVO
}
```

ESTRATEGIAS DE CARGA CON @BASIC

@Basic es una anotación que nos permite controlar el momento en que una propiedad es cargada desde la base de datos, evitando que traer valores que no son necesario al momento de cargar el objeto. Esta anotación es utilizada generalmente para anotar objetos pesados, como una imagen o un archivo binario.

En JPA existe dos conceptos que son claves para entender cómo es que JPA carga los objetos desde la base de datos y estos son claves para mejorar el rendimiento de la aplicación, estos conceptos se explican a continuación:

- Lazy loading (Carga demorada): Los objetos de carga demorada no serán cargados desde la base de datos cuando el objeto sea creado, pero será cargado en cuanto se acceda a la propiedad. De esta manera JPA identifica cuando la propiedad es accedida por primera vez para cargar el valor desde la base de datos.

```
@Basic( fetch = FetchType.LAZY )
```

- Eager loading (Carga ansiosa o temprana): Este es la utilizada por default para la mayoría de las propiedades en JPA, a excepción de las colecciones.

```
@Basic( fetch = FetchType.EAGER )
```

```
public class Empleado {
    …
    @Column(name = "FOTO" ,nullable = true)
    @Basic(optional = false, fetch = FetchType.EAGER)
    private byte[] foto;
    …
    /**
     * GETs and SETs
     */
}
```

ATRIBUTOS VOLÁTILES CON @TRANSIENT

La anotación @Transient se utiliza para indicarle a JPA que un atributo de una Entidad no debe de ser persistente, de esta manera, JPA pasa por alto el atributo y no es tomado en cuenta a la hora de persistir el Objeto.

En la práctica no es común utilizar esta anotación, debido a que las Entidades por lo general solo tiene los atributos que mapean con la base de datos. Sin embargo, existen ocasiones en donde puede ser útil.

```
public class Empleado {
    …
    @Transient
    private float salario;
    …

    /**
     * GETs and SETs
     */
}
```

Si se ejecuta el programa no se creará una columna para la propiedad salario.

CARDINALIDAD Y RELACIONES CON JPA

Relación	***Descripción***
@ManyToMany	Define una relación many-to-many al unir tablas.
@ManyToOne	Define una relación de many-to-one al unir tablas.
@OneToMany	Define una relación one-to-many al unir tablas.
@OneToOne	Define una relación one-to-one al unir tablas.

RELACIÓN @MANYTOONE

Una de las grandes ventajas que tiene trabajar con JPA, es que te permite hacer relaciones con otras entidades, de esta forma, es posible agregar otras Entidades como atributos de clase y JPA se encargará de realizar el SELECT adicional para cargar esas Entidades.

```
@Entity
@Table(name="facturas")
public class Factura {

    @Id
    @Column(name="ID")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;

    @Column(name = "ESTADO", length = 20, nullable = false)
    @Enumerated(EnumType.STRING)
    private Status estado;

    @Column(name = "FECHA_REGISTRO", nullable = false)
    @Temporal(TemporalType.TIMESTAMP)
    private Calendar fechaRegistro;

    @ManyToOne(optional = false, cascade = CascadeType.ALL, fetch =
FetchType.EAGER)
    private Cliente cliente;

    /** GET and SET */
}
```

Como podemos apreciar, hemos utiliza la anotación @ManyToOne , la cual nos permite mapear una entidad con otra. Como única regla, es necesario la clase que sea una entidad, es decir, que también esté anotada con @Entity.

La anotación @ManyToOne cuenta con los siguientes atributos:

- Optional: indica si la relación es opcional, es decir, si el objeto puede ser null. Esta propiedad se utiliza optimizar las consultas. Si JPA sabe que una relación es opcional, entonces puede realizar un RIGHT JOIN o realizar la consulta por separado, mientras que, si no es opcional, puede realizar un INNER JOIN para realizar una solo consulta.

- Cascade: Esta propiedad le indica que operaciones en cascada puede realizar con la Entidad relacionada, los valores posibles son ALL , PERSIST , MERGE , REMOVE , REFRESH , DETACH y están definidos en la enumeración javax.persistence.CascadeType .
- Fetch: Esta propiedad se utiliza para determinar cómo debe ser cargada la entidad, los valores están definidos en la enumeración javax.persistence.FetchType y los valores posibles son:

    EAGER (ansioso): Indica que la relación debe de ser cargada al momento de cargar la entidad.

    LAZY (perezoso): Indica que la relación solo se cargará cuando la propiedad sea l eída por primera vez.

En el ejemplo se ha definido la propiedad optional en false, pues toda factura debe de tener forzosamente un Cliente. La propiedad cascade la definimos en ALL. La propiedad fetch la definimos en EAGER, aun que para este tipo de relación es su valor por default y no sería necesario definirlo.

PERSONALIZAR LAS RELACIONES CON @JOINCOLUMN

Mediante la anotación @JoinColumn es posible personalizar las columnas que será utilizadas como uniones con otras tablas. Cuando trabajamos con relaciones como @ManyToOne o @OneToOne, es necesario indicarle a JPA como es que tendrá que realizar la unión (JOIN) con la otra Entidad.

La anotación @JoinColumn se puede llegar a confundir con @Column, sin embargo, tiene una funcionalidad diferente, @JoinColumn se utiliza para marcar una propiedad la cual requiere de un JOIN para poder accederlas, mientas que @Column se utiliza para representar columnas simples que no están relacionadas con otra Entidad. A pesar de que se utiliza para cosas distintas, la realidad es que se parecen muchísimo, pues tiene casi las mismas propiedades, por que podríamos decir que @JoinColumn es el equivalente de @Column cuando utilizamos relaciones con Entidades.

Toda factura está relacionada a un cliente, y utilizamos la anotación @ManyToOne para crear la relación:

```
@Entity
@Table(name="facturas")
public class Factura {

    @Id
    @Column(name="ID")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;

    @Column(name = "ESTADO", length = 20, nullable = false)
    @Enumerated(EnumType.STRING)
    private Status estado;

    @Column(name = "FECHA_REGISTRO", nullable = false)
    @Temporal(TemporalType.TIMESTAMP)
    private Calendar fechaRegistro;

    @JoinColumn(name = "fk_cliente", nullable = false)
    @ManyToOne(optional = false, cascade = CascadeType.ALL, fetch =
FetchType.EAGER)
    private Cliente cliente;

    /** GET and SET */
}
```

PROPIEDADES DE @JOINCOLUMN

- name: Indica el nombre con el que se deberá de crear la columna dentro de la tabla.
- referencedColumnName: Se utiliza para indicar sobre que columna se realizará el Join de la otra tabla. Por lo general no se suele utilizar, pues JPA asume que la columna es el ID de la Entidad objetivo.
- unique: Crea un constraints en la tabla para impedir valores duplicados (default false).
- nullable: Crea un constraints en la tabla para impedir valores nulos (default true).

- insertable: Le indica a JPA si este valor deberá guardarse en la operación de inserción (default true)
- updatable: Le indica a JPA si el valor deberá actualizarse durante el proceso de actualización (default true)
- columnDefinition: Esta propiedad es utiliza para indicar la instrucción SQL que se deberá utilizar la crear la columna en la base de datos. Esta nos ayuda a definir exactamente como se creará la columna sin depender de la configuración de JPA.
- table: Le indicamos sobre que tabla deberá realizar el JOIN, normalmente no es utilizada, pues JPA asume la tabla por medio de la entidad objetivo.
- foreignKey: Le indica a JPA si debe de crear el Foreign Key, esta propiedad recibe uno de los siguientes valores CONSTRAINT , NO_CONSTRAINT , PROVIDER_DEFAULT definidos en la enumeración javax.persistence.ForeignKey .

RELACIONES @ONETOONE

Las relaciones Uno a Uno (@OneToOne) se caracterizan porque solo puede existir una y solo una relación con la Entidad de destino, de esta forma, la entidad marcada como @OnoToOne deberá tener una referencia a la Entidad destino y por ningún motivo podrá ser una colección. De la misma forma, la Entidad destino no podrá pertenecer a otra Instancia de la Entidad origen.

Las relaciones @OneToOne se utilizan cuando existe una profunda relación entre la Entidad origen y destino, de tal forma que la entidad destino le pertenece a la Entidad origen y solo a ella, por ende, la existencia de la entidad destino depende de la Entidad origen.

Entidad Factura

```
@Entity
@Table(name="facturas")
public class Factura {

    @Id
    @Column(name="ID")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;

    @Column(name = "ESTADO", length = 20, nullable = false)
    @Enumerated(EnumType.STRING)
    private Status estado;

    @Column(name = "FECHA_REGISTRO", nullable = false)
    @Temporal(TemporalType.TIMESTAMP)
    private Calendar fechaRegistro;

    @JoinColumn(name = "FK_CLIENTE", nullable = false)
    @ManyToOne(optional = false, cascade = CascadeType.ALL, fetch =
FetchType.EAGER)
    private Cliente cliente;

    @OneToOne(mappedBy = "factura", cascade = CascadeType.ALL)
    private Pago pago;

    /** GET and SET */
}
```

Entidad Pago

```
@Entity
@Table(name = "pagos")
public class Pago {

    @Id
    @Column(name = "ID")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Temporal(TemporalType.TIMESTAMP)
    @Column(name="FECHA_PAGO", nullable = false, updatable = false)
    private Calendar fechaPago;

    @Column(name = "CANTIDAD", nullable = false, updatable = false)
    private double cantidad;

    @Column(name = "METODO_PAGO ", nullable = false, updatable = false)
    private MetodoPago metodoPago;

    @OneToOne
    @JoinColumn(name = "FK_FACTURA", updatable = false, nullable = false)
    private Factura factura;

    /** GET and SET */
}
```

La tabla p*agos* tiene la columna FK_FACTURA que hace relación con la tabla *facturas*. Algo que hay que tomar en cuenta, es que tanto la Entidad *Factura* como *Pago* tiene anotada como @OneToOne a la otra entidad, sin embargo, solo pagos ha creado la columna para hacer el JOIN, esto se debe a que esta es la que tiene la anotación @JoinColumn, la cual nos permite establecer como se llamará la columna para realizar el JOIN, adicional, la Entidad *Factura* tiene definida la propiedad mappedBy de la anotación @OneToOne, esta propiedad es muy importante, pues le permite hacer una relación bidireccional, si no la pusiéramos, JPA agregaría una columna adicional en la tabla facturas, y guardaría el pago como un nuevo registro.

RELACIONES @ONETOMANY

Las relaciones uno a muchos (@OneToMany) se caracterizan por Entidad donde tenemos un objeto principal y colección de objetos de otra Entidad relacionados directamente. Estas relaciones se definen mediante colecciones, pues tendremos una serie de objetos pertenecientes al objeto principal.

En el ejemplo de la entidad Factura, se debe tener además una Entidad cabecera donde tengamos los datos principales de la factura, como podría ser serie, cliente, total, fecha de expedición, etc. Por otra parte, la factura tendrá una serie de líneas que representa cada uno de los productos vendidos.

Entidad Factura

```
@Entity
@Table(name="facturas")
public class Factura {

    @Id
    @Column(name="ID")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;

    @Column(name = "ESTADO", length = 20, nullable = false)
    @Enumerated(EnumType.STRING)
    private Status estado;

    @Column(name = "FECHA_REGISTRO", nullable = false)
    @Temporal(TemporalType.TIMESTAMP)
    private Calendar fechaRegistro;

    @JoinColumn(name = "FK_CLIENTE", nullable = false)
    @ManyToOne(optional = false, cascade = CascadeType.ALL, fetch =
FetchType.EAGER)
    private Cliente cliente;

    @OneToOne(mappedBy = "factura", cascade = CascadeType.ALL)
    private Pago pago;

    @OneToMany(cascade = CascadeType.ALL, mappedBy = "factura")
    private List<LineaFactura> lineas;

    /** GET and SET */
}
```

Como se puede observar se ha definido la propiedad lineas como una lista (List), lo cual nos permite relacionar la factura con un número indeterminado de líneas, por otro lado, hemos definido la propiedad mappedBy para indicar que es una relación bidireccional, es decir, la Entidad LineaFactura tendrá también una relación hacia la Entidad Factura.

Entidad LineaFactura

```
@Entity
@Table(name = "lineas_factura")
public class LineaFactura {
    @Id
    @Column(name = "ID")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "PRODUCTO", nullable = false)
    private String producto;

    @Column(name = "PRECIO", nullable = false)
    private double precio;

    @Column(name = "TOTAL", nullable = false)
    private double cantidad;

    @ManyToOne
    @JoinColumn(name = "FK_FACTURA", nullable = false, updatable = false)
    private Factura factura;

    /** GET and SET */
}
```

RELACIÓNES @MANYTOMANY

Las relaciones Mucho a Muchos (@ManyToMany) se caracterizan por Entidades que están relacionadas con a 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, 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 relaciones las dos Entidades, veremos más adelante como resolvemos eso.

Un ejemplo clásico de estas relaciones son los libros con sus autores, de esta forma, un libro puede tener varios autores, y a su vez, los autores pueden tener muchos libros. Pero para que quede más claro, veamos como quedarían las Entidades de Autor y Libro:

Entidad Libro

```
@Entity
@Table(name = "libros")
public class Libro {
    @Id
    @Column(name="ID")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "NOMBRE", nullable = false)
    private String nombre;

    @JoinTable(
        name = "rel_libro_autores",
        joinColumns = @JoinColumn(name = "FK_LIBRO", nullable = false),
        inverseJoinColumns = @JoinColumn(name="FK_AUTOR", nullable = false)
    )
    @ManyToMany(cascade = CascadeType.ALL)
    private List<Autor> autores;

    public void agregarAutor(Autor autor){
        if(this.autores == null){
            this.autores = new ArrayList<>();
        }

        this.autores.add(autor);
    }

    /** GET and SET */

}
```

Se ha creado una lista de tipo *Autor*, la cual es anotada con @ManyToMany, y adicionalmente, se ha definido la anotación @JoinTable, la cual nos sirve para definir la estructura de la tabla intermedia que contendrá la relación entre los libros y los autores.

La anotación @JoinTable no es obligatoria en sí, ya que en caso de no definirse JPA asumirá el nombre de la tabla, columnas, longitud, etc. Para no quedar a merced de la implementación de JPA, siempre es recomendable definirla, así, tenemos el control total sobre ella.

```
@Entity
@Table(name="authors")
public class Author {

    @Id
    @Column(name="ID")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name="NAME", nullable = false)
    private String name;

    @ManyToMany(mappedBy = "authors")
    private List<Book> books;

    /** GET and SET **/
}
```

El caso de la Entidad *Autor* es más simple, pues sólo se marca la colección con @ManyToMany, pero en este caso ya no es necesario definir la anotación @JoinTable, en su lugar, se define la propiedad mappedBy para indicar la relación bidireccional y al mismo tiempo, JPA puede tomar la configuración del @JoinTable de *Libros*.