Material descargable

Java Persistence API

egg

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 (DBMS). 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 tenían 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
- EclipseLink
- OpenJPA

Persistencia de objetos

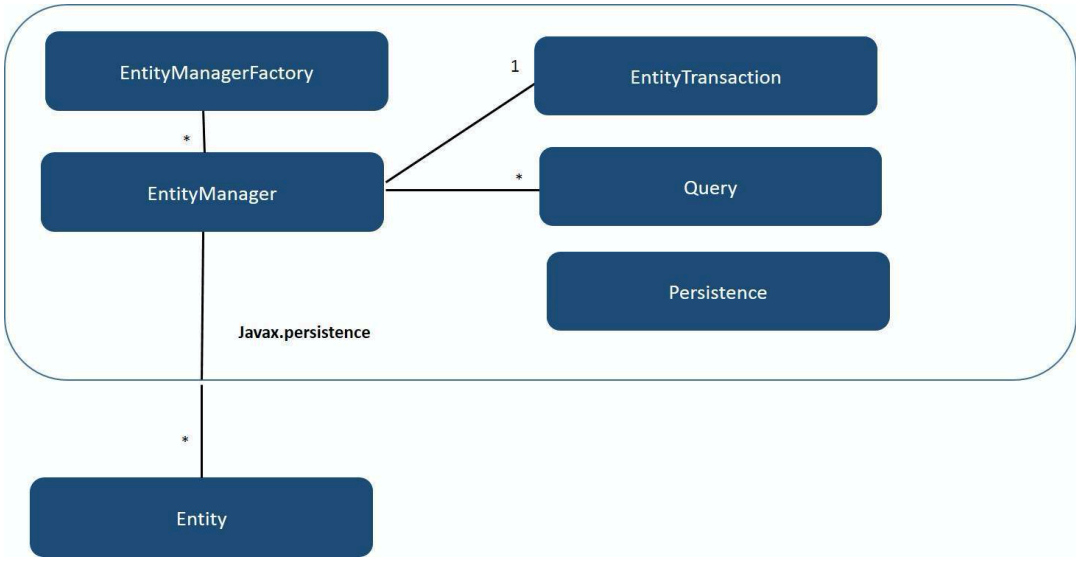
JPA representa una simplificación del modelo de programación de persistencia. La especificación JPA define explícitamente la correlación relacional de objetos, en lugar de basarse en implementaciones de correlación específicas del proveedor.

JPA crea un estándar para la importante tarea de la correlación relacional de objetos mediante la utilización de anotaciones o XML para relacionar 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 - Componentes



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

Entidad: En JPA, una entidad es una clase Java que representa una fila en una tabla de base de datos. Cada instancia de la entidad corresponde a una fila en la tabla, y cada atributo de la clase mapea una columna en la tabla. Las entidades son siempre clases concretas (no abstractas) que deben estar anotadas con `@Entity`. Estas clases también pueden ser usadas para definir relaciones entre tablas y sus datos. En resumen, cada entidad en JPA está asociada a una tabla en la base de datos, y sus atributos corresponden a las columnas de esa tabla.

Persistence: Es una clase que proporciona métodos estáticos para obtener instancias de `EntityManagerFactory`, que es el punto de entrada a la unidad de persistencia.

EntityManagerFactory: Es una factoría de EntityManager. Se encarga de crear y gestionar instancias de EntityManager, y a menudo se configura una sola vez durante la vida del proceso de la aplicación.

EntityManager: Es una interfaz que gestiona las operaciones de persistencia de las entidades, como crear, leer, actualizar y eliminar. Es fundamental en cualquier proyecto que use JPA y también actúa como una fábrica para la creación de consultas (`Query`).

Query: Es una interfaz que se utiliza para ejecutar consultas JPQL (Java Persistence Query Language) y obtener resultados basados en criterios específicos.

EntityTransaction: Representa una transacción en la base de datos que agrupa múltiples operaciones realizadas a través de un `EntityManager`. Permite controlar el inicio, confirmación y reversión de transacciones.

Mapeo con Anotaciones

En el contexto de las bases de datos relacionales, la información se almacena mediante tablas, filas y columnas. Para almacenar un objeto en una base de datos, es necesario correlacionar el sistema orientado a objetos de Java con el sistema relacional de la base de datos. JPA facilita esta tarea realizando la conversión entre objetos y

tablas de manera automática mediante ORM (Object-Relational Mapping - Mapeo Relacional de Objetos).

Las anotaciones en JPA permiten configurar el mapeo de una entidad dentro del mismo archivo donde se declara la clase, especificando cómo se deben relacionar las clases con las tablas y los atributos con las columnas. Estas anotaciones comienzan con el símbolo `@` y se utilizan antes de la declaración de la clase, propiedad o método. Aquí se detallan las principales anotaciones:

Anotación	Propiedades	Descripción
@Entity	-	Declara la clase como una entidad que será mapeada a una tabla en la base de datos.
@Table	**name** (String): Nombre de la tabla en la base de datos. **schema** (String): Esquema de la base de datos. **catalog** (String): Catálogo de la base de datos. **uniqueConstraints** (UniqueConstraint[]): Restricciones de unicidad.	Especifica el nombre de la tabla en la base de datos con la que se mapea la entidad. Si no se usa, JPA utilizará el nombre de la clase como el nombre de la tabla por defecto.
@Id	-	Declara que un atributo de la entidad es la clave primaria de la tabla.
@GeneratedValue	**strategy** (GenerationType): Estrategia de generación de la clave primaria. **generator** (String): Nombre del generador de secuencia.	Define cómo se generará el valor de la clave primaria. Puede ser configurado para ser generado manualmente, automáticamente o a partir de una secuencia.

@Column	**name** (String): Nombre de la columna. **length** (int): Longitud de la columna. **nullable** (boolean): Si la columna puede ser nula. **unique** (boolean): Si la columna es única.	Indica que un atributo se mapea con una columna de una tabla en la base de datos. Se puede usar para especificar detalles adicionales sobre la columna, como el nombre o la longitud.
@Enumerated	**value** (EnumType): Tipo de valor que se almacenará (ORDINAL o STRING).	Especifica que un atributo es de un tipo enumerado (enum). Los valores de un tipo enumerado se almacenan en la base de datos como su posición ordinal o como su nombre.
@Temporal	**value** (TemporalType): Tipo de fecha que se usará (DATE, TIME, TIMESTAMP).	Indica que un atributo maneja fechas y permite especificar el tipo de fecha que se usará en la base de datos.

Estas anotaciones permiten definir cómo se deben transformar las entidades en tablas de base de datos, facilitando así la persistencia y recuperación de datos en aplicaciones Java.

Declarar entidades con @Entity

Como hemos discutido, las entidades en JPA son clases Java comunes y corrientes. Sin embargo, para que JPA pueda **identificarlas y convertirlas en tablas de base de datos**, debes usar la anotación `@Entity`. Esta anotación se coloca a nivel de clase y le indica a JPA que la clase es una entidad que debe ser administrada y mapeada a una tabla.

Veamos un ejemplo para aclarar esto:

Supongamos que tienes una clase llamada `Empleado`, la cual actualmente no se considera una entidad porque aún no tiene la anotación `@Entity`. Aquí está cómo se vería la clase sin la anotación:

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

    public Long getId() {
        return id;
    }
    // Otros métodos y atributos
}
```

En este estado, `Empleado` es una clase común y no está vinculada a ninguna tabla en la base de datos.

Ahora, para convertir esta clase en una entidad, simplemente debes agregar la anotación `@Entity` sobre la clase:

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

    public Long getId() {
        return id;
    }
    // Otros métodos y atributos
}
```

Al agregar `@Entity`, le estás indicando a JPA que esta clase `Empleado` es una entidad y que debe ser administrada por el `EntityManager`. En este punto, la clase `Empleado` ya puede ser considerada una entidad y será mapeada a una tabla en la base de datos.

Definir llave primaria con @Id

Al igual que en las tablas de bases de datos, las entidades en JPA también necesitan un identificador o clave primaria (`ID`). Este identificador es crucial para diferenciar cada entidad del resto. Por regla general, debes definir un `ID` para todas las entidades; de lo contrario, el `EntityManager` generará un error al intentar instanciar la entidad.

El `ID` es importante porque el `EntityManager` lo utiliza para realizar operaciones de persistencia, como `select`, `update` o `delete`. Es el campo que permite al `EntityManager` identificar y gestionar los registros en la base de datos.

Aquí tienes un ejemplo de cómo definir el `ID` en una entidad:

```
@Entity
public class Empleado {
    @Id
    private Long id;

    private String nombre;
}
```

En este caso, al agregar `@Id` sobre el atributo `id`, le estás indicando al `EntityManager` que este campo es el identificador de la clase `Empleado`.

Anotación @GeneratedValue

La anotación `@GeneratedValue` se utiliza para autogenerar el valor del `ID`, similar a cómo funciona el auto incremento en MySQL. Esta anotación le dice a JPA qué estrategia de generación de claves primarias utilizar. Una de las estrategias comunes es `GenerationType.IDENTITY`.

Identity

La estrategia `GenerationType.IDENTITY` es fácil de usar. Solo necesitas especificarla, y JPA se encargará del resto. Cuando persistencias una entidad, JPA no envía el valor del `ID` porque asume que la columna es autogenerada. Cada vez que

insertas un nuevo objeto, el contador de la columna se incrementa automáticamente en 1.

Aquí tienes un ejemplo de cómo usar `@GeneratedValue` con la estrategia `IDENTITY`:

```
@Entity
public class Empleado {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String nombre;
}
```

Mapeo de fechas con @Temporal

Con la anotación `@Temporal`, puedes mapear fechas en tu base de datos de manera sencilla. Uno de los desafíos al trabajar con fechas y horas es asegurar que el formato sea compatible con el manejador de base de datos. Sin embargo, `@Temporal` simplifica este proceso.

Utilizando `@Temporal`, puedes especificar si el atributo debe almacenar solo la fecha, solo la hora, o ambos. Puedes usar esta anotación con las clases `Date` o `Calendar`. Aquí están los tres valores posibles para la anotación:

- **DATE**: Solo almacena la fecha, descartando la hora. Utiliza `@Temporal(TemporalType.DATE)`.
- **TIME**: Solo almacena la hora, descartando la fecha. Utiliza `@Temporal(TemporalType.TIME)`.
- **TIMESTAMP**: Almacena tanto la fecha como la hora. Utiliza `@Temporal(TemporalType.TIMESTAMP)`.

Aquí tienes un ejemplo de cómo usar `@Temporal`:

```java
@Entity
public class Empleado {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String nombre;

    @Temporal(TemporalType.DATE)
    private Date fechaNacimiento;
}
```

En este ejemplo, `@Temporal(TemporalType.DATE)` indica que el campo `fechaNacimiento` solo almacenará la fecha, sin incluir la hora.

Las Relaciones

En Java, los objetos pueden estar relacionados entre sí mediante relaciones entre clases. De manera similar, en bases de datos relacionales como MySQL, las tablas también pueden tener relaciones entre ellas. JPA te ofrece herramientas para reflejar estas relaciones en tu modelo de datos.

Cuando tienes dos objetos relacionados y quieres que esta relación se represente en las tablas de la base de datos, JPA proporciona cuatro anotaciones principales para definir estos tipos de relaciones. Estas anotaciones afectan cómo se relacionan los registros entre las tablas, pero no alteran tu código Java.

Aquí están las anotaciones que JPA ofrece para definir las relaciones entre tablas:

- **@OneToOne**: Relación uno a uno entre tablas.
- **@OneToMany**: Relación uno a muchos entre tablas.
- **@ManyToOne**: Relación muchos a uno entre tablas.
- **@ManyToMany**: Relación muchos a muchos entre tablas.

Es importante entender cómo aplicar estas anotaciones. La primera palabra en la anotación se refiere a la clase que tiene la relación, y la segunda palabra se refiere a la clase que es el atributo de la primera.

@OneToOne

Supongamos que tienes dos clases, `Curso` y `Profesor`, con una relación uno a uno en Java: un curso tiene un único profesor y un profesor pertenece a un único curso. En Java, esto se representa así:

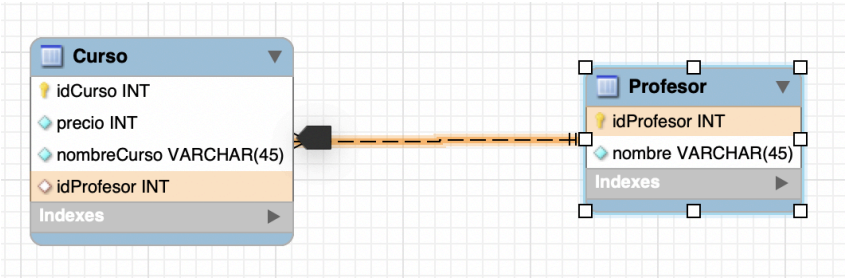
```java
@Entity
public class Curso {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @OneToOne
    private Profesor profesor;

    // Otros atributos y métodos
}
```

Aquí, al agregar `@OneToOne`, indicas que la relación entre `Curso` y `Profesor` debe ser uno a uno. En la base de datos, esto se traduce en una tabla `Curso` con una llave foránea que apunta a la tabla `Profesor`. Esto asegura que cada registro de `Curso` está asociado a un único `Profesor`.

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



@ManyToOne

Usa esta anotación cuando tienes una relación de muchos a uno entre tablas. Por ejemplo, si tienes muchos álbumes que pueden pertenecer a un solo autor, lo representas en Java así:

```
@Entity
public class Album {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToOne
    private Autor autor;

    // Otros atributos y métodos
}
```

Aquí, `@ManyToOne` indica que varios álbumes pueden tener el mismo autor. En la base de datos, esto significa que la tabla `Album` tendrá una columna que referencia la tabla `Autor`, permitiendo que múltiples registros en `Album` se asocien con un único `Autor`.

@ManyToMany

Esta anotación se usa para relaciones muchos a muchos. Por ejemplo, si cada tarea puede ser asignada a muchos empleados y cada empleado puede tener muchas tareas, lo representas en Java así:

```
@Entity
public class Empleado {
    @Id
```

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

    @ManyToMany
    private List<Tarea> tareas;

    // Otros atributos y métodos
}
```

Es fundamental tener en cuenta que, cuando trabajas con relaciones @ManyToMany o @OneToMany, estas no existen físicamente en la base de datos de manera directa. En su lugar, es necesario crear una tabla intermedia que se encargue de establecer la relación entre las dos entidades.

Cuál 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 Empleado o varios Empleados pueden hacer una o varias Tareas y a su vez, una o varias Tareas pueden ser realizadas por uno o varios Empleados. Esto en SQL sería que un registro tiene varios registros asignados a él mismo.

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

Empleado			
ID	Nombre	Apellido	IdTarea
1	Adriana	Cardello	1

1	Adriana	Cardello	2
1	Adriana	Cardello	3

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

Por lo que, para este tipo de relación se crea una tabla intermedia conocida como tabla asociativa.

Por convención, el nombre de esta tabla debe estar formado por el nombre de las tablas participantes (en singular y en orden alfabético) separados por un guión bajo (_). A excepción de que la relación tenga algún nombre significativo, por ejemplo: Un alumno puede estar inscripto en muchos cursos y un curso tiene muchos alumnos, a esa relación alumno - curso la podemos llamar Inscripción.

Esta tabla está compuesta por las claves primarias de las tablas que se relacionan con ella, así se logra que la relación sea de uno a muchos, en los extremos, de modo tal que la relación se lea:

Un empleado tiene muchas tareas y una tarea puede ser hecha por muchos empleados.

Las tablas se verían así:

Empleado	
ID	Nombre
95	Adriana
28	Mariela
31	Agustín

Tarea	
ID	Nombre
915	Ordenar
624	Limpiar
567	Cocinar

Tabla intermedia

Empleado_Tarea	
ID_Empleado	ID_Tarea
95	624
95	915
28	567
28	624

Como podemos observar la tabla intermedia solo tiene dos columnas, el id del empleado y el id de la tarea. Esta tabla no toma las columnas como llaves primarias, sino como llaves foráneas, esto nos permite repetir los valores para asignar las tareas al empleado.

Las tablas en MySQL se verían así:

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

Las anotaciones de JPA (Java Persistence API) son una parte fundamental del mapeo objeto-relacional (ORM) en Java. Estas anotaciones se utilizan para definir cómo las clases de Java y sus atributos se relacionan con las tablas y columnas de una base de datos relacional. Para profundizar en el uso de estas anotaciones, te ofrecemos un recurso útil en el siguiente enlace: **link.**