Apunte 11 - JPA (Java Persistence API)

JPA es otra API de persistencia de Java pero trabaja a un nivel más alto de abstracción comparado con JDBC. Se basa en el concepto de ORM (Object-Relational Mapping) que mapea objetos en tu aplicación a tablas en una base de datos. JPA facilita el trabajo con bases de datos relacionales sin requerir que escribas consultas SQL explícitas para cada operación.

Características principales:

- **Entidades:** Las clases de tu modelo se anotan como entidades (@Entity), lo que permite a JPA saber qué clases deben ser mapeadas a tablas en la base de datos.
- **EntityManager:** El EntityManager es el punto de entrada para realizar operaciones CRUD en tus entidades.
- **JPQL:** JPA introduce un lenguaje de consulta llamado JPQL (Java Persistence Query Language) que se utiliza para realizar consultas de manera similar a SQL pero orientado a objetos.

Historia y Evolución

- **EJB 2.x Entity Beans:** Antes de JPA, los Entity Beans en EJB 2.x eran la opción estándar para la persistencia en Java, pero eran complejos y difíciles de manejar.
- **JPA 1.0:** Introducido en 2006 como parte de la especificación de Java EE 5, con el objetivo de simplificar la persistencia en Java.
- **JPA 2.0:** Lanzado en 2009 como parte de Java EE 6, agregó características como el Criteria API y las anotaciones de mapeo adicionales.
- **JPA 2.1:** Lanzado en 2013 con Java EE 7, incluyó nuevas características como los Stored Procedures, Converters y más.
- **JPA 2.2:** Lanzado en 2017, se centró en correcciones menores y actualizaciones.

Importancia de JPA en el Ecosistema Java

- **Independencia del Proveedor:** JPA permite a los desarrolladores cambiar fácilmente entre diferentes proveedores de persistencia como Hibernate, EclipseLink y OpenJPA.
- **Desacoplamiento:** Al separar los objetos de negocio de la lógica de persistencia, JPA facilita una arquitectura más limpia y mantenible.
- **Productividad:** Con anotaciones simples y un lenguaje de consulta intuitivo (JPQL), los desarrolladores pueden realizar tareas complejas de persistencia de manera más eficiente.
- **Estándar de la Industria:** JPA ha ganado aceptación y es ampliamente utilizado en aplicaciones empresariales, convirtiéndose en un estándar de facto para la persistencia en Java.

JPA Implementación y Uso

La especificación JPA se centra en 3 ejes fundamentales a la hora de formalizar el Mapeo Objeto Relacional en Java. Estos 3 ejes van a ser los 3 ejes fundamentales de estudio y a los que le vamos a tener que dedicar esfuerzo para comprender su funcionamiento y capacidades de control. Estas serán las capacidad que nos permitan modificar la Implementación subyacente para que responda de acuerdo con las decisiones de diseño que hemos tomado para el proyecto y se adecúe de la mejor manera a nuestro esquema funcional.

Estos 3 ejes fundamentales nos en órden de necesidad para la implementación:

- **Configuración** donde especificaremos la conexión a la base de datos, la implementación subyacente, el esquema de transacciones y las configuraciones específicas de la implementación.
- **Mapeo** donde realizaremos la conexión entre los atributos de las clases de entidad o Entidades JPA y las columnas de las tablas de la base de datos, además de marcar las restricciones o configuraciones específicas asociadas a dichos mapeos.
- **Administración de Entidades y Consultas** finalmente el código en el que aprovecharemos el uso del EntityManager para crear, actualizar o borrar entidades y las consultas JPQL para obtener listas de entidades a partir de los datos de la base de datos.

A continuación trabajaremos específicamente en cada uno de estos 3 ejes.

Configuración

Nuestro proyecto va a requerir algunos cambios para poder funcionar con JPA/<Implementación elegida> a saber:

1. **Inclusión de Dependencia:** Añade las dependencias necesarias para JPA en tu archivo de configuración de build (como pom.xml si estás utilizando Maven).

```
<!-- Dependencia para Hibernate (un proveedor popular de JPA) -->
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.x.x</version>
</dependency>
```

2. **Archivo persistence.xml:** Crea un archivo persistence.xml en el directorio `src/main/resources/META-INF`. Este archivo es esencial para configurar aspectos como el proveedor de JPA, la conexión a la base de datos y las configuraciones específicas de la implementación.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
    <persistence-unit name="nombreDeTuUnidadDePersistencia">
        <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
        <properties>
```

```
            <!-- Configuración de la base de datos -->
            <property name="javax.persistence.jdbc.url"
value="jdbc:mysql://localhost:3306/nombreDeLaBaseDeDatos"/>
            <property name="javax.persistence.jdbc.user"
value="nombreDeUsuario"/>
            <property name="javax.persistence.jdbc.password"
value="contraseña"/>
            <property name="javax.persistence.jdbc.driver"
value="com.mysql.cj.jdbc.Driver"/>

            <!-- Propiedades adicionales de Hibernate -->
            <property name="hibernate.dialect"
value="org.hibernate.dialect.MySQLDialect"/>
            <property name="hibernate.show_sql" value="true"/>
            <property name="hibernate.hbm2ddl.auto" value="update"/>
        </properties>
    </persistence-unit>
</persistence>
```

Configuración de `persistence.xml` para MySQL con Hibernate

Este archivo XML contiene la configuración necesaria para conectar la aplicación Java con una base de datos MySQL usando Hibernate como proveedor JPA.

Propiedades de Configuración

- **nombreDeTuUnidadDePersistencia**: Es el nombre que utilizarás para referenciar esta unidad de persistencia en tu código.
- **javax.persistence.jdbc.url**: URL de la base de datos.
- **javax.persistence.jdbc.user**: Nombre de usuario para acceder a la base de datos.
- **javax.persistence.jdbc.password**: Contraseña para acceder a la base de datos.
- **javax.persistence.jdbc.driver**: Clase del controlador JDBC.

Propiedades específicas de Hibernate

Las propiedades adicionales de Hibernate (como `hibernate.dialect`, `hibernate.show_sql`, `hibernate.hbm2ddl.auto`) son opcionales y pueden ajustarse según tus necesidades.

3. **Configuración de la Base de Datos:** Asegúrate de tener una base de datos instalada (como MySQL) y configura las credenciales y la URL en persistence.xml.
4. **Creación de Entidades:** Crea tu primera entidad Java y anótala adecuadamente con `@Entity`, `@Id`, etc.
5. **EntityManager:** Utiliza `EntityManager` para realizar operaciones CRUD básicas en tu entidad.

Mapeo: Entidades y Atributos

Entidades

Una entidad es una clase Java anotada con @Entity que representa una tabla en una base de datos relacional. Cada instancia de una entidad corresponde a una fila en esa tabla.

```
@Entity
public class Usuario {
    // código de la clase
}
```

Atributos

Los atributos de la clase entidad representan las columnas de la tabla. Estos pueden ser simples como números y cadenas, o complejos como otras entidades o colecciones de entidades.

```
@Entity
public class Usuario {
    @Id
    private Long id;
    private String nombre;
    private String email;
    // ...
}
```

Anotaciones en JPA

Las anotaciones en JPA juegan un papel crucial al simplificar la configuración de la persistencia y el mapeo objeto-relacional. A continuación se describen algunas de las anotaciones más importantes que debes conocer para trabajar con JPA.

`@Entity` Indica que una clase es una entidad y se debe mapear a una tabla en la base de datos.

```
@Entity
public class Persona {
    // ...
}
```

- `@Id` Identifica la propiedad que actúa como la clave primaria en la tabla de la base de datos.

  ```
  @Id
  private Long id;
  ```

- `@GeneratedValue` Se utiliza para especificar cómo se generan los valores de la clave primaria. Comúnmente se usa con @Id.

  ```
  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  ```

```
private Long id;
```

- `@Column` Define las propiedades de una columna en la tabla como el nombre de la columna, si es única, si es nullable, etc.

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

- `@Table` Se utiliza para especificar detalles sobre la tabla a la que se mapeará la entidad, como el nombre de la tabla, esquema, índices, etc.

```
@Entity
@Table(name = "personas")
public class Persona {
    // ...
}
```

EntityManager

EntityManager es la interfaz a través de la cual interactúas con el contexto de persistencia en JPA. Es responsable de las operaciones CRUD (Crear, Leer, Actualizar, Eliminar) y otras transacciones.

Obtener un EntityManager

Puedes obtener una instancia de EntityManager a través de EntityManagerFactory.

```
EntityManagerFactory emf =
Persistence.createEntityManagerFactory("nombreDeTuUnidadDePersistencia");
EntityManager em = emf.createEntityManager();
```

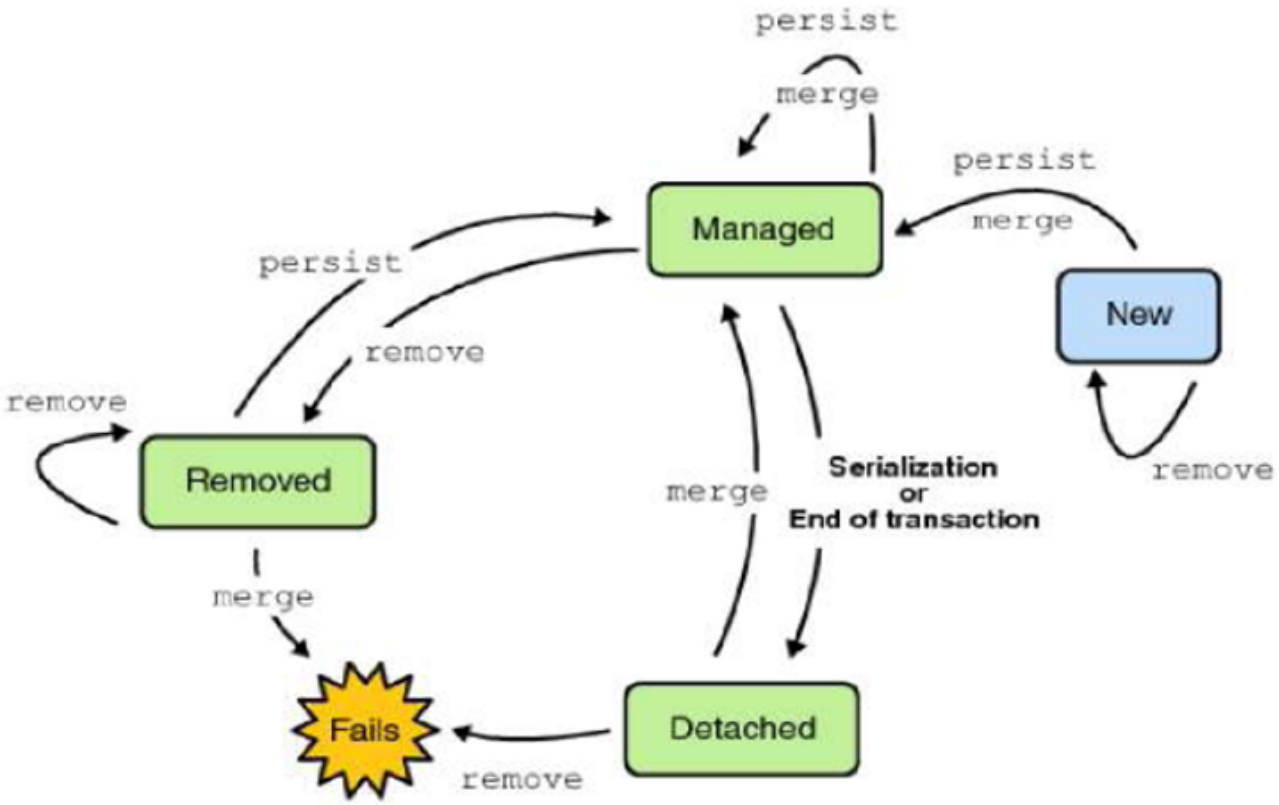
Operaciones CRUD Básicas

- **Crear:** em.persist(objeto)
- **Leer:** em.find(ClaseEntidad.class, id)
- **Actualizar:** em.merge(objeto)
- **Eliminar:** em.remove(objeto)

Contexto de Persistencia

El contexto de persistencia es un conjunto de entidades gestionadas por un EntityManager en un momento dado. Cuando una entidad es gestionada, cualquier cambio en su estado se sincroniza automáticamente con la base de datos.

Estados de una Entidad



- **New/Transient:** La entidad ha sido instanciada pero aún no está siendo gestionada por el EntityManager.
- **Managed/Persistent:** La entidad está siendo gestionada por el EntityManager y cualquier cambio se sincroniza con la base de datos.
- **Detached:** La entidad fue gestionada anteriormente pero ahora ya no está asociada con un EntityManager.
- **Removed:** La entidad está marcada para ser eliminada de la base de datos.

Operaciones CRUD con JPA (Create, Read, Update, Delete)

Las operaciones CRUD son las operaciones básicas de manipulación de datos en cualquier sistema de gestión de bases de datos. En JPA, estas operaciones se pueden realizar utilizando el `EntityManager`. A continuación se describen estas operaciones en detalle:

Create (Crear)

Para crear una nueva entidad y guardarla en la base de datos, utilizamos el método `persist`.

```
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();
Usuario nuevoUsuario = new Usuario("nombre", "email@example.com");
em.persist(nuevoUsuario);
em.getTransaction().commit();
```

Retrieve (Obtener)

Para leer una entidad de la base de datos, utilizamos el método `find`.

```
EntityManager em = emf.createEntityManager();
Usuario usuarioExistente = em.find(Usuario.class, id);
```

Update (Actualizar)

Para actualizar una entidad ya existente, utilizamos el método `merge`.

```
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();
usuarioExistente.setEmail("nuevo-email@example.com");
em.merge(usuarioExistente);
em.getTransaction().commit();
```

Delete (Eliminar)

Para eliminar una entidad de la base de datos, utilizamos el método `remove`.

```
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();
Usuario usuarioAEliminar = em.find(Usuario.class, id);
em.remove(usuarioAEliminar);
em.getTransaction().commit();
```

Estas son las operaciones CRUD básicas en JPA. Al comprender estos fundamentos, estarás bien equipado para manejar la persistencia de datos en tus aplicaciones Java.

Relaciones entre Entidades

Las relaciones entre entidades son uno de los aspectos más poderosos de JPA. Permiten mapear las relaciones complejas que existen entre las tablas de una base de datos a objetos en el mundo Java de una manera muy intuitiva.

Relación Uno a Uno (@OneToOne)

La anotación `@OneToOne` se utiliza para mapear una relación uno a uno entre dos entidades.

```
@Entity
public class Persona {
    @OneToOne
    private Pasaporte pasaporte;
}
```

Relación Uno a Muchos (@OneToMany) y Muchos a Uno (@ManyToOne)

La anotación `@OneToMany` se utiliza para mapear una relación uno a muchos, mientras que `@ManyToOne` se utiliza para mapear una relación muchos a uno.

```
@Entity
public class Departamento {
    @OneToMany(mappedBy = "departamento")
    private List<Empleado> empleados;
}

@Entity
public class Empleado {
    @ManyToOne
    private Departamento departamento;
}
```

Relación Muchos a Muchos (@ManyToMany)

La anotación `@ManyToMany` se utiliza para mapear una relación muchos a muchos entre dos entidades.

```
@Entity
public class Estudiante {
    @ManyToMany
    private List<Curso> cursos;
}

@Entity
public class Curso {
    @ManyToMany(mappedBy = "cursos")
    private List<Estudiante> estudiantes;
}
```

Al entender cómo utilizar estas anotaciones para representar relaciones entre entidades, podrás modelar de manera eficiente tu base de datos relacional y tu código Java.

Consultas en JPA

Las consultas en JPA se pueden realizar de diversas maneras, incluyendo JPQL (Java Persistence Query Language), Criteria API y consultas nativas SQL. A continuación, se exploran cada una de estas opciones.

JPQL (Java Persistence Query Language)

JPQL es un lenguaje de consultas similar a SQL pero orientado a objetos, lo cual lo hace más natural para los desarrolladores de Java.

Consulta Básica

```
List<Usuario> usuarios = em.createQuery("SELECT u FROM Usuario u",
Usuario.class).getResultList();
```

Consulta con Parámetros

```
List<Usuario> usuarios = em.createQuery("SELECT u FROM Usuario u WHERE u.nombre =
:nombre", Usuario.class)
                           .setParameter("nombre", "John")
                           .getResultList();
```

Criteria API

Criteria API proporciona una forma programática de crear consultas, lo cual es útil cuando las consultas son dinámicas.

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Usuario> query = cb.createQuery(Usuario.class);
Root<Usuario> root = query.from(Usuario.class);
query.select(root).where(cb.equal(root.get("nombre"), "John"));
List<Usuario> usuarios = em.createQuery(query).getResultList();
```

Consultas Nativas SQL

Para los casos en que se necesita un control más detallado que el ofrecido por JPQL o Criteria API, JPA también soporta consultas SQL nativas.

```
List<Usuario> usuarios = em.createNativeQuery("SELECT * FROM usuarios WHERE nombre
= 'John'", Usuario.class).getResultList();
```

Resultados de la Consulta

Los resultados de las consultas en JPA pueden devolverse como una lista de entidades, un único resultado o incluso como un conjunto de resultados.

```
Usuario usuario = em.createQuery("SELECT u FROM Usuario u WHERE u.id = :id",
Usuario.class)
                    .setParameter("id", 1L)
                    .getSingleResult();
```

Con un conocimiento sólido de cómo realizar consultas en JPA, puedes recuperar y manipular datos en tus aplicaciones Java de forma efectiva y eficiente.

Transacciones en JPA

Las transacciones son fundamentales para asegurar la integridad de los datos en aplicaciones que utilizan bases de datos. En JPA, el objeto `EntityManager` es el responsable de gestionar las transacciones. A continuación, se describen las operaciones básicas relacionadas con transacciones en JPA.

Iniciar una Transacción

Para iniciar una transacción, utilizamos el método `begin` del objeto `EntityManager`.

```
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();
```

Confirmar una Transacción (Commit)

Una vez que todas las operaciones han sido realizadas, la transacción se confirma utilizando el método `commit`.

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

Deshacer una Transacción (Rollback)

Si se encuentra un error o si se necesita revertir las operaciones realizadas, se puede utilizar el método `rollback`.

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

Transacciones en Bloques try-catch

Es una buena práctica manejar las transacciones dentro de bloques try-catch para manejar excepciones de forma efectiva.

```
try {
    em.getTransaction().begin();
    // Operaciones CRUD aquí
    em.getTransaction().commit();
} catch (Exception e) {
    em.getTransaction().rollback();
    e.printStackTrace();
} finally {
    em.close();
}
```

Atributos de Transacción en Contenedores (Opcional)

Si estás utilizando JPA en un entorno gestionado como un servidor de aplicaciones Java EE, los atributos de transacción podrían ser gestionados por el contenedor.

```
@Stateless
public class MiServicio {
    @PersistenceContext
    private EntityManager em;

    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public void miMetodo() {
        // Operaciones CRUD aquí
    }
}
```

Al comprender cómo se manejan las transacciones en JPA, podrás garantizar que tus aplicaciones sean robustas y mantengan la integridad de los datos.

Ejemplo JPA

```
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

public class JPASample {
  public static void main(String[] args) {
    // Crear EntityManagerFactory
    EntityManagerFactory emf =
Persistence.createEntityManagerFactory("persistence-unit");

    // Crear EntityManager
    EntityManager em = emf.createEntityManager();

    // Iniciar transacción
    em.getTransaction().begin();

    // Operación CRUD
    MyEntity entity = new MyEntity();
    entity.setName("Sample");
    em.persist(entity);

    // Commit
    em.getTransaction().commit();

    // Cerrar recursos
    em.close();
    emf.close();
  }
```

```
}
```

Enlaces relacionados

- Introducción a JPA
- Java Persistence API (JPA)