



Lección 3

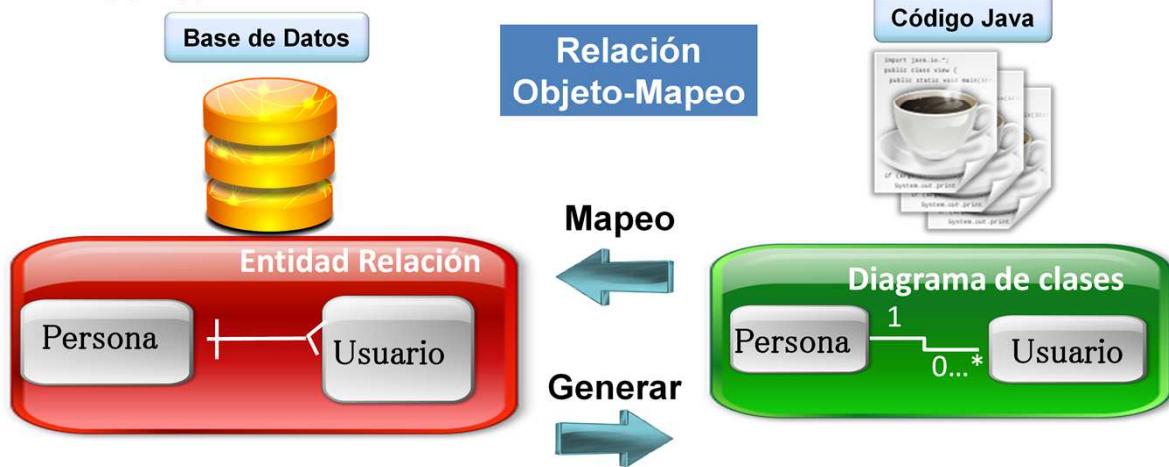
Java Persistence API (JPA)

www.globalmentoring.com.mx

© Derechos Reservados Global Mentoring

¿Qué es Java Persistence API?

- Java Persistence API, mejor conocido como JPA, es el estándar de persistencia de Java.
- JPA implementa conceptos de frameworks ORM (Object Relational Mapping)



La mayoría de la información de las aplicaciones empresariales es almacenada en bases de datos relacionales. La persistencia de datos en Java, y en general en los sistemas de información, ha sido uno de los grandes temas a resolver en el mundo de la programación.

Al utilizar únicamente JDBC tenemos el problema de crear demasiado código para poder ejecutar una simple consulta. Por lo tanto, para simplificar el proceso de interacción con una base de datos (select, insert, update, delete), se ha utilizado desde hace ya varios años el concepto de frameworks ORM (Object Relational Mapping), tales como Hibernate.

Como podemos observar en la figura, un framework ORM nos permite "mapear" una clase Java con una tabla de Base de Datos. Por ejemplo, la clase Persona, al crear un objeto en memoria, podemos almacenarlo directamente en la tabla de Persona, simplemente ejecutando una línea de código: **em.persist (persona)**. Esto ha simplificado enormemente la cantidad de código a escribir en la capa de datos de una aplicación empresarial.

En esta lección estudiaremos la tecnología Java Persistence API, mejor conocido como JPA. Esta tecnología de Java es el estándar de persistencia en Java, y la buena noticia es que cuenta con varias implementaciones, tales como Hibernate, EclipseLink, OpenJPA, entre algunas más. Así que si ya conoces Hibernate, o algún framework de persistencia Java, te será muy familiar muchos de los conceptos que estudiaremos en esta lección.

Aprenderemos temas como: Una visión general de JPA, Manejo y Ciclo de Vida de las entidades en JPA, JP Query Language para realizar consultas en la base de datos y Manejo de Transacciones utilizando EJB's y JPA, entre otros temas. Esta lección será crucial para generar una capa de datos robusta, flexible, y que pueda comunicarse con cualquier base de datos relacional.

Características de JPA

Características de Java Persistence API:



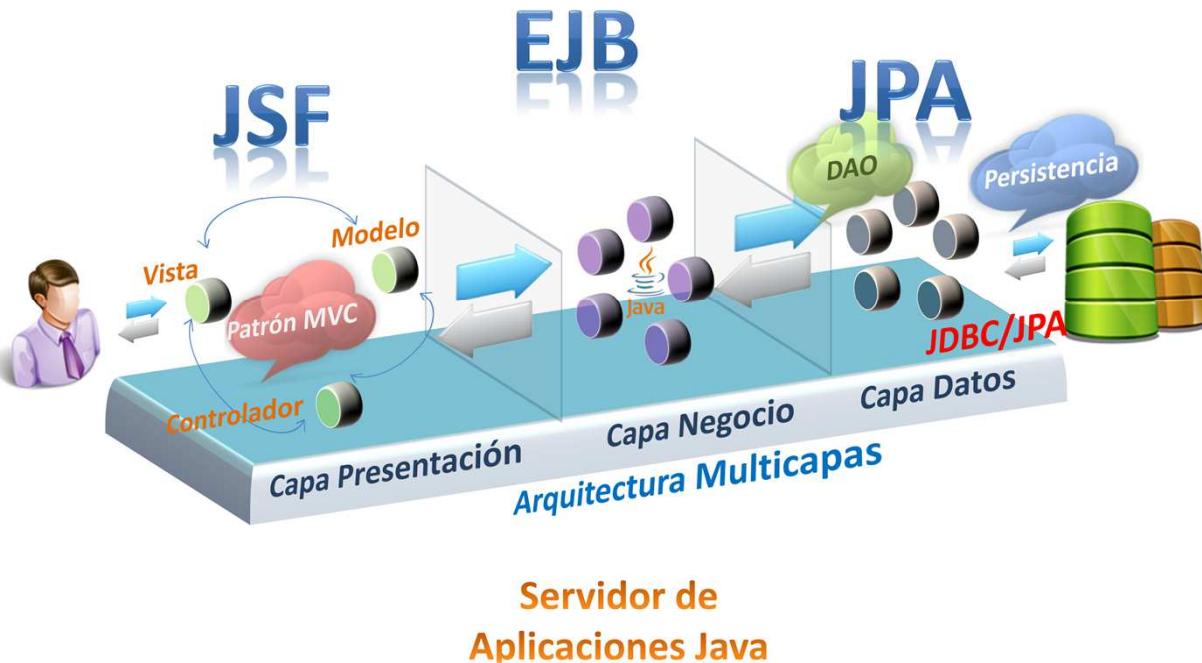
- ✓ Persistencia utilizando POJOs.
- ✓ No Intrusivo.
- ✓ Consultas utilizando Objetos Java.
- ✓ Configuración Simplificada.
- ✓ Integración.
- ✓ Testing.

La idea del API JPA es trabajar con objetos Java y no con código SQL, de tal manera que podamos enfocarnos en el código Java. JPA permite abstraer la comunicación con las bases de datos y crea un estándar para ejecutar consultas y manipular la información de una base de datos.

Características de Java Persistence API:

- ✓ **Persistencia utilizando POJOs:** Este es posiblemente el aspecto más importante de JPA, debido a que cualquier clase de Java podemos convertirla en una clase de entidad, simplemente agregando anotaciones y/o agregando un archivo xml de mapeo.
- ✓ **No Intrusivo:** JPA es una capa separada de los objetos a persistir. Por ello, las clases Java de Entidad no requieren extender ninguna funcionalidad en particular ni saber de la existencia de JPA, por ello es no intrusivo.
- ✓ **Consultas utilizando Objetos Java:** JPA permite ejecutar queries expresados en términos de objetos Java y sus relaciones, sin necesidad de utilizar el lenguaje SQL. Los queries son traducidos por el API de JPA en el código SQL equivalente.
- ✓ **Configuración simple:** Muchas de las opciones de JPA están configuradas con opciones por default, sin embargo si queremos personalizarlas, es muy simple, ya sea con anotaciones o a través de archivos xml de configuración.
- ✓ **Integración:** Debido a que las arquitecturas empresariales Java son por naturaleza multicapas, una integración transparente es muy valiosa para los programadores Java y JPA permite hacer la integración con las demás capas de manera muy simple.
- ✓ **Testing:** Con JPA ahora es posible realizar pruebas unitarias, o utilizar cualquier clase con un método main fuera del servidor, simplemente utilizando la versión estándar de Java. Esto permite reducir los tiempos de desarrollo de las aplicaciones empresariales de manera considerable.

Arquitectura Empresarial con JPA



En la figura podemos observar el rol de JPA en una arquitectura Java Empresarial. Esta tecnología aplica directamente en la capa de datos, la cual se encarga de tareas tales como:

- Recuperación de información a través de consultas (select)
- Manejo de información de objetos Java en las tablas de base de datos respectivas (insert, update, delete)
- Manejo de una unidad de persistencia (Persistence Unit) para la creación y destrucción de conexiones a la base de datos.
- Manejo de transacciones, respetando el esquema de propagación definido en la capa de negocio en los EJBs de Sesión.
- Portabilidad hacia otras bases de datos con un impacto menor, así como bajo acoplamiento con las otras capas empresariales.
- Entre varias tareas más.

Además, para realizar las tareas de persistencia, podemos utilizar patrones de diseño tales como:

- **DAO (Data Access Object):** Este patrón de diseño suele definir una interfaz y una implementación de dicha interfaz, para realizar las operaciones más comunes con la Entidad respectiva. Por ejemplo, para la entidad Persona, generaremos la interfaz DaoPersona, y agregaremos los métodos agregarPersona, modificarPersona, eliminarPersona, findAllPersonas, etc.
- **DTO (Data Transfer Object):** Este patrón de diseño permite definir una clase, que en ocasiones es muy similar a la clase de entidad, ya que contiene los mismos atributos, pero con el objetivo de transmitirla a las siguientes capas, incluso, hasta la capa Web. Por ello se les conoce como objetos de valor o de transferencia.

En la actualidad, y con la simplificación de las capas de una arquitectura empresarial, es opcional utilizar estos y otros patrones de diseño empresariales, sin embargo, muchas aplicaciones Java han sido construidas con estos patrones, así que vale la pena entender para qué se utilizan y cómo aplicarlos.

Para más información de los patrones de diseño les recomendamos los siguientes libros.

Para J2EE: <http://www.amazon.com/Core-J2EE-Patterns-Practices-Strategies/dp/0131422464>

Para Java EE: <http://www.amazon.com/Real-World-Patterns-Rethinking-Practices/dp/0557078326>

Entidades y Persistencia en JPA

- Una clase de entidad es un POJO y puede configurarse por medio de anotaciones o un archivo XML.
- Ejemplo de clase de Entidad con anotaciones:

```

@Entity
public class Persona {

    @Id
    @GeneratedValue
    private Long personaId;

    @Column(nullable = false)
    private String nombre;

    private String apePaterno;

    private String apeMaterno;
    private String email;
    private Integer telefono;

    // Constructores, getters, setters
}

```

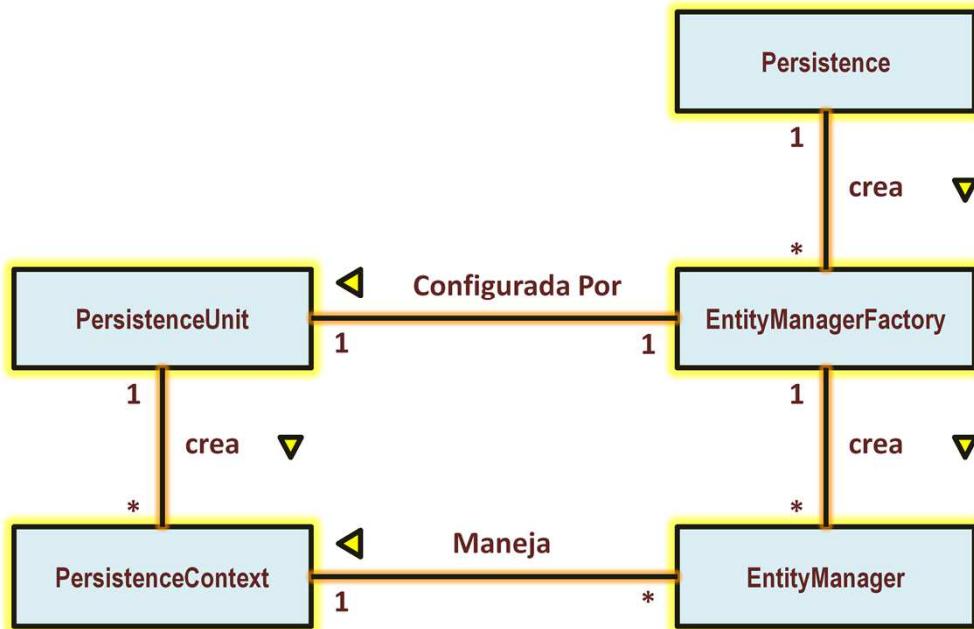
En las primeras versiones de J2EE, existía el concepto de Entity Beans para el manejo de persistencia. Sin embargo esta tecnología resultaba complicada para sistemas del mundo real, resultando en un bajo performance. Así mismo, realizar consultas (queries) utilizando los objetos Entity era muy complicado, se necesitaba un servidor de aplicaciones Java para ejecutar un EJB tipo Entity, no existían pruebas unitarias, y por lo tanto cualquier cambio en nuestro código implicaba un nuevo deploy, consumiendo mucho tiempo simplemente en probar nuestro código de persistencia, entre varios detalles más.

Sin embargo, el concepto de los Entity Beans era muy bueno desde el punto de vista de la programación orientada a objetos. Hibernate fue uno de los primeros frameworks en incorporar muchas de las características que podemos utilizar al día de hoy con JPA. Hibernate simplificó por mucho la tecnología de los EJB de Entidad, y aunque no es un estándar, en muchas aplicaciones Java, escoger Hibernate es garantía de un excelente framework de persistencia.

Al día de hoy, una clase conocida como Entidad es simplemente un POJO, y en combinación con el uso de anotaciones, es suficiente para convertirla en una clase de Entidad, la cual representa un registro de una tabla de base de datos. Este tipo de conceptos, heredados de frameworks como Hibernate, TopLink, JDO, entre otros, contribuyó en lo que conocemos al día de hoy como el estándar de persistencia Java conocido como JPA.

El API de JPA se puede utilizar en una aplicación estándar de Java o en un servidor Web o Empresarial Java. Ahora ya es posible realizar pruebas unitarias sobre nuestras clases de Entidad y Consultas sobre los objetos de Entidad, disminuyendo dramáticamente el tiempo de desarrollo de nuestras clases de entidad, consultas, y en general en la creación de la capa de datos de una aplicación empresarial.

API de JPA y Entity Manager



Para que una clase de Entidad pueda ser persistida, se debe realizar una llamada al API de JPA. De hecho muchas de las operaciones se realizan a través de esta API, la cual está separada de nuestras clases de Entidad.

En la figura podemos observar el API JPA, la cual tiene como elemento principal al objeto EntityManager, siendo este una interfaz. Una implementación de esta interfaz es la que realmente ejecuta el trabajo de persistencia, sincronización con la base de datos, transaccionalidad, validación de mapeo, conversión de código Java a SQL, entre muchas otras tareas.

Por lo tanto, una clase Entity, desde el punto de vista descrito anteriormente, es tan solo una clase Java normal, la cual al vincularse con un EntityManager, se persiste en la base de datos.

El objeto EntityManager se obtiene de una fábrica de objetos conocida como EntityManagerFactory, y este objeto se asocia con un proveedor JPA, pudiendo haber seleccionado entre varios proveedores según la implementación de JPA escogida (Hibernate, EclipseLink, OpenJPA, etc).

El objeto Persistence Unit, se encarga de realizar la configuración del proveedor seleccionado por medio de un archivo xml, además de definir otros elementos tales como: la forma de comunicarse con la Base de Datos, las clase de Entidad en la aplicación, si se va a utilizar JTA para el manejo transaccional, entre varias características más.

A su vez, al conjunto de objetos Entity administrados por JPA en un tiempo específico de la aplicación se le conoce como PersistenceContext, de esta manera JPA se asegura que no existan objetos de Entidad duplicados en memoria, entre otras tareas más.

Un ejemplo de cómo utilizar el API JPA para persistir un objeto de Entidad, se muestra a continuación:

```

EntityManagerFactory emf = Persistence.createEntityManagerFactory("PersonaService");
EntityManager em = emf.createEntityManager();
Personas persona = new Personas(15);
em.persist(persona);
  
```

Configuración de Unidad de Persistencia

Configuración de la Unidad de Persistencia (Persistence Unit):

Ejemplo de contenido del archivo persistence.xml:

```
<persistence>
  <persistence-unit name="PersonaService" transaction-type="RESOURCE_LOCAL">
    <class>domain.Persona</class>
    <properties>
      <property name="javax.persistence.jdbc.driver"
                value="org.apache.derby.jdbc.ClientDriver"/>
      <property name="javax.persistence.jdbc.url"
                value="jdbc:derby://localhost:1527/PersonaServDB;create=true"/>
      <property name="javax.persistence.jdbc.user" value="APP"/>
      <property name="javax.persistence.jdbc.password" value="APP"/>
    </properties>
  </persistence-unit>
</persistence>
```

Para realizar la configuración de la Unidad de Persistencia se debe utilizar un archivo xml llamado persistence.xml.

El nombre del elemento persistence-unit indica el nombre de la unidad de persistencia, y es muy importante recordarlo, ya que es el nombre que utilizaremos en nuestro código Java al momento de utilizar el objeto EntityManagerFactory.

El atributo transaction-type especifica el tipo de transaccionalidad que se utilizará, pudiendo seleccionar JTA como el proveedor.

Se pueden especificar también las clases de Entidad del sistema, pudiendo definir varias clases. Si la aplicación se despliega en un servidor Java, no es necesario declarar estas clases, sin embargo, para aplicaciones Java SE (Standard Edition) es necesario especificar las clases de Entidad del sistema.

La sección de propiedades especifica características del proveedor a utilizar, así como los datos de conexión a la base de datos.

Al momento de empaquetar una aplicación Java, el archivo persistence.xml se debe ubicar en la carpeta META-INF/persistence.xml del archivo .jar.

Nota: Los nombres de las propiedades JDBC se estandarizaron (`javax.persistence.jdbc.*`) en la versión JPA 2, previo a esta versión se debe revisar con el proveedor respectivo.

Referenciando la Unidad de Persistencia

Ejemplo de uso de la unidad de persistencia y del Entity Manager:

```

@Stateless
public class PersonaServiceBean implements PersonaService {

    @PersistenceContext(unitName="PersonaService")
    EntityManager em;

    public void agregarPersona(Persona persona) {
        em.persist(persona);
    }

    public Persona encontrarPersona(int idPersona) {
        return em.find(Persona.class, idPersona);
    }

    public Persona modificarNombrePersona(int idPersona, String nuevoNombre) {
        Persona persona = em.find(Persona.class, idPersona);
        if (persona != null) {
            persona.setNombre(nuevoNombre);
        }
        return persona;
    }

    public void eliminarPersona(int idPersona) {
        Employee emp = em.find(Employee.class, idPersona);
        em.remove(emp);
    }
}
  
```

Curso de Java EE

© Derechos Reservados Global Mentoring

En la figura podemos observar un ejemplo de un código donde hacemos uso de la unidad de persistencia para obtener el objeto EntityManager. Una vez que ya hemos obtenido una referencia al objeto EntityManager, podemos comenzar a realizar las operaciones con los objetos de Entidad.

Por ejemplo, podemos realizar tareas como:

- ✓ **Inserción:** Para persistir una entidad se utiliza el método **persist** del EntityManager. Con este método podemos generar un registro en la base de datos. Este registro no será guardado hasta haber concluido la transacción (commit). Recordemos que los métodos de un Session Bean son transaccionales por default, esto implica que al terminar de ejecutar el método **agregarPersona** y al ejecutarse en un contenedor empresarial Java, en automático se realizará el commit. En lecciones posteriores revisaremos el tema de transacciones dentro y fuera de un servidor de aplicaciones.
- ✓ **Búsqueda:** Una vez que tenemos un objeto persistido, podemos recuperar la información del registro de la base de datos utilizando el método **find** del objeto EntityManager, y basta con especificar el tipo (clase) y el id (llave primaria) que estamos buscando.
- ✓ **Modificación:** La modificación cambia un poco, debido a que JPA necesita primero saber con qué entidad se está trabajando, por ello necesitamos recuperar el objeto de entidad. Una vez recuperado, realizamos las modificaciones necesarias, y si el objeto se encuentra en una transacción activa, JPA revisará en automático si es necesario realizar alguna actualización sobre el registro. Lo interesante es que no es necesario volver a llamar al método **persist**, esta llamada es opcional.
- ✓ **Eliminación:** Similar a la modificación, primero se debe recuperar la entidad con el método **find**, y una vez en memoria, llamamos el método **remove**.

Estas son las operaciones básicas utilizando el objeto EntityManager sobre nuestros objetos de entidad. A continuación revisaremos algunos ejemplos para profundizar más en este tema.



Ejercicio 5 y 6

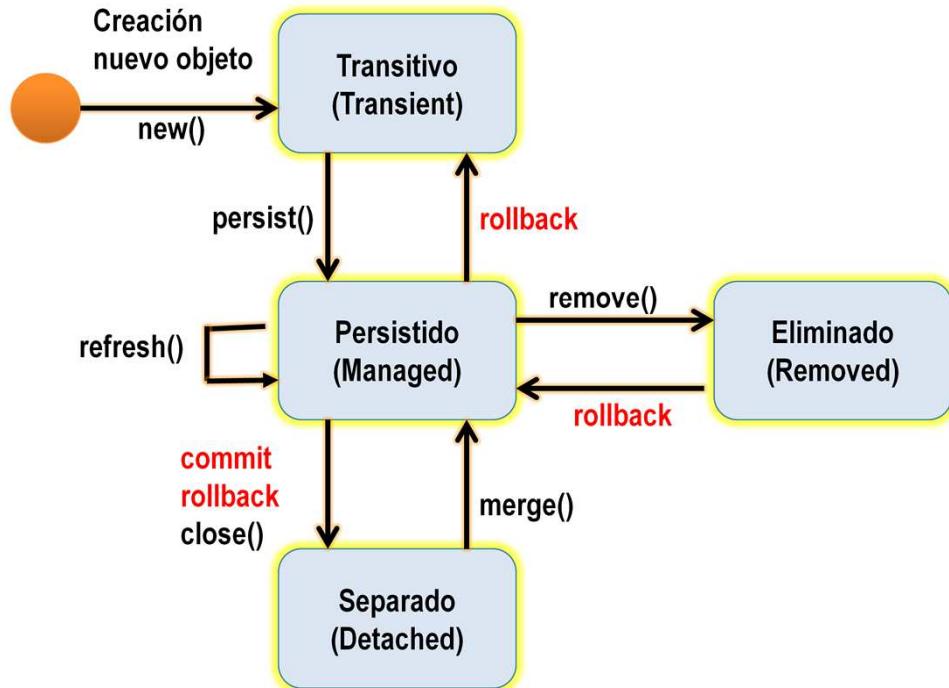
- Abrir el documento PDF de Ejercicios del cursos de Java EE.
- Realizar las siguientes prácticas:
- **Ejercicio 5:** Persistencia objetos con JPA y Testing.
- **Ejercicio 6:** SGA con JPA



Ejercicio 7 y 8

- Abrir el documento PDF de Ejercicios del cursos de Java EE.
- Realizar las siguientes prácticas:
- **Ejercicio 7:** Instalación de Hibernate Tools.
- **Ejercicio 8:** Ingeniería Inversa con JPA

Ciclo de Vida Entidad JPA



El API de JPA simplifica en gran medida la forma en que interactuamos con una base de datos. JPA agrega un ciclo de vida para la administración de los objetos de entidad. A continuación describimos los estados del ciclo de vida.

Estado Transitivo (Transient):

- ✓ Los objetos de entidad nuevos NO son guardados directamente en la Base de Datos (BD).
- ✓ No están asociados con un registro de BD.
- ✓ Se consideran NO transaccionales.

Estado Persistente (Managed):

- ✓ Un objeto persistente tiene asociado un registro en la BD.
- ✓ Los objetos persistentes siempre están asociados con una transacción. Su estado se sincroniza con la BD al terminar la transacción.

Estado Separado (Detached):

- ✓ Estos objetos tienen asociado un registro de BD, pero su estado no está sincronizado con la BD
- ✓ Todos los objetos recuperados en una transacción se convierten en detached una vez que termina la misma

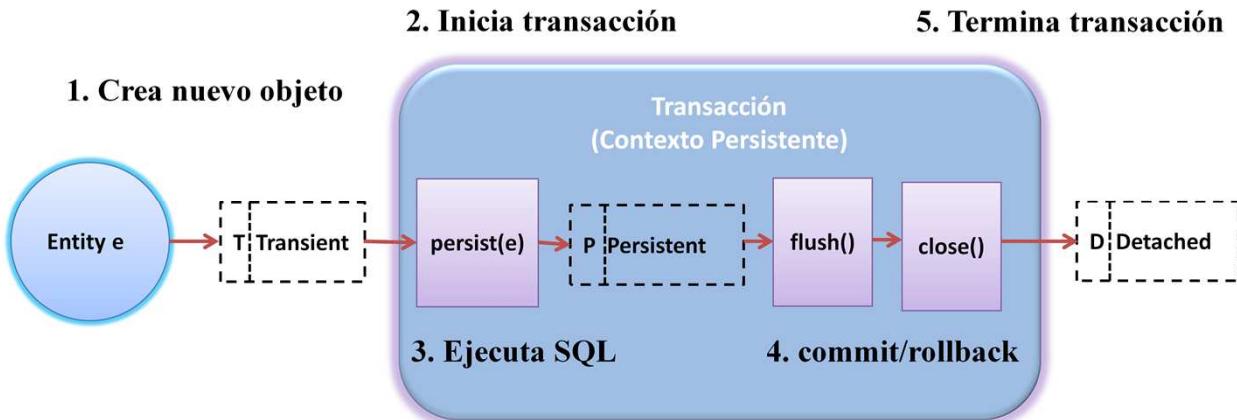
Estado Eliminado (Removed):

- ✓ Estos objetos ya no tiene una representación con la BD y al terminar la transacción, el registro asociado es eliminado.

Existen además anotaciones para complementar cualquier acción deseada entre los estados mostrados, tales como `@PrePersist` y `@PostPersist`, las cuales se utilizan al persistir un objeto, `@PreRemove` y `@PostRemove` al eliminar un objeto, `@PreUpdate` y `@PostUpdate` al actualizar

un objeto, así como @PostLoad al hacer refresh del objeto.

Persistir un objeto en JPA



Código ejemplo para persistir un objeto en JPA:

```
public void testPersistirObjeto() {
    //Paso 1. Crea nuevo objeto
    //Objeto en estado transitivo
    Persona persona1 = new
    Persona("Pedro","Luna",null,"pluna@mail.com","19292943");

    //Paso 2. Inicia transacción
    EntityTransaction tx = em.getTransaction();
    tx.begin();

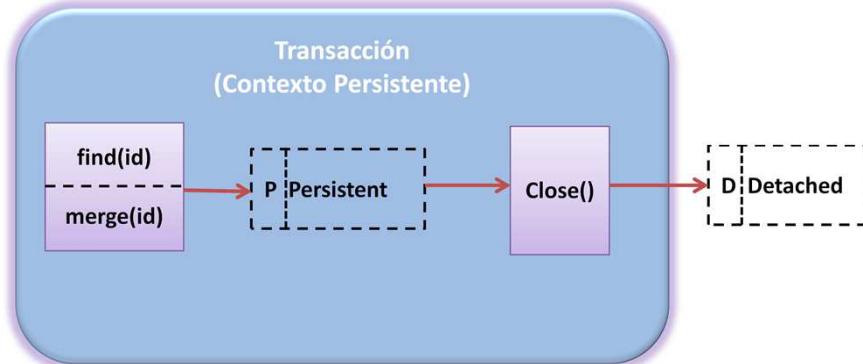
    //Paso 3. Ejecuta SQL
    em.persist(persona1);

    //Paso 4. Commit/rollback
    tx.commit();

    //Objeto en estado detached
    log.debug("Objeto persistido:" + persona1);
}
```

Recuperar un objeto de Entidad en JPA

1. Inicia transacción



3. Termina transacción

2. Ejecuta SQL

Código de ejemplo para recuperar un objeto en JPA:

```
public void testEncontrarObjeto() {
    //Paso 1. Inicia transacción
    EntityTransaction tx = em.getTransaction();
    tx.begin();

    //Paso 2. Ejecuta SQL de tipo select
    Persona persona1 = em.find(Persona.class, 23);

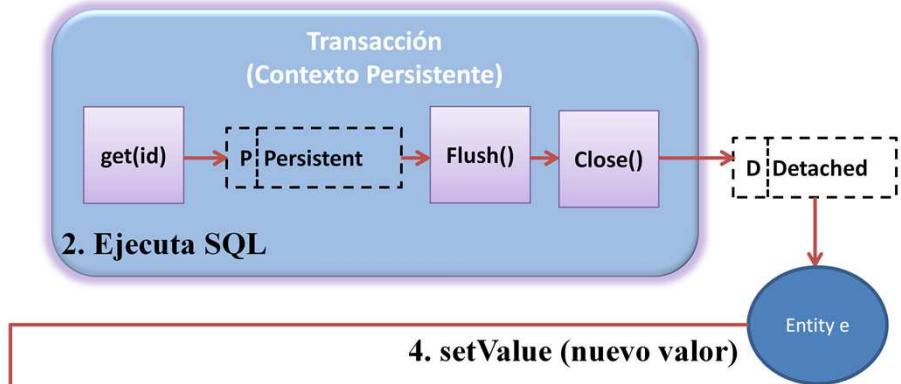
    //Paso 3. Termina transacción
    tx.commit();

    //Objeto en estado detached
    log.debug("Objeto recuperado:" + persona1);
}
```

Actualizar un Objeto Persistente en JPA

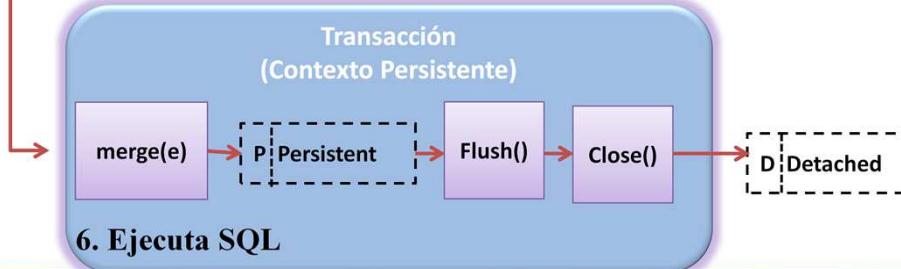
1. Inicia transacción I

3. Termina transacción I



5. Inicia transacción II

7. Termina transacción II



Proceso para actualizar un objeto en JPA:

```

public void testActualizarObjeto() {
    //Paso 1. Inicia transacción 1
    EntityTransaction tx1 = em.getTransaction();
    tx1.begin();

    //Paso 2. Ejecuta SQL de tipo select
    //El id proporcionado debe existir en la base de datos
    Persona persona1 = em.find(Persona.class, 23);

    //Paso 3. Termina transacción 1
    tx1.commit();

    //Objeto en estado detached
    log.debug("Objeto recuperado:" + persona1);

    //Paso 4. setValue (nuevoValor)
    persona1.setApeMaterno("Nava");

    //Paso 5. Inicia transacción 2
    EntityTransaction tx2 = em.getTransaction();
    tx2.begin();

    //Paso 6. Ejecuta SQL. Es un select, pero al estar modificado,
    //al terminar la transacción hará un update
    //Como ya tenemos el objeto hacemos solo un merge para resincronizar
    //el objeto a hacer merge, debe contar con el valor de la llave primaria
    em.merge(persona1);

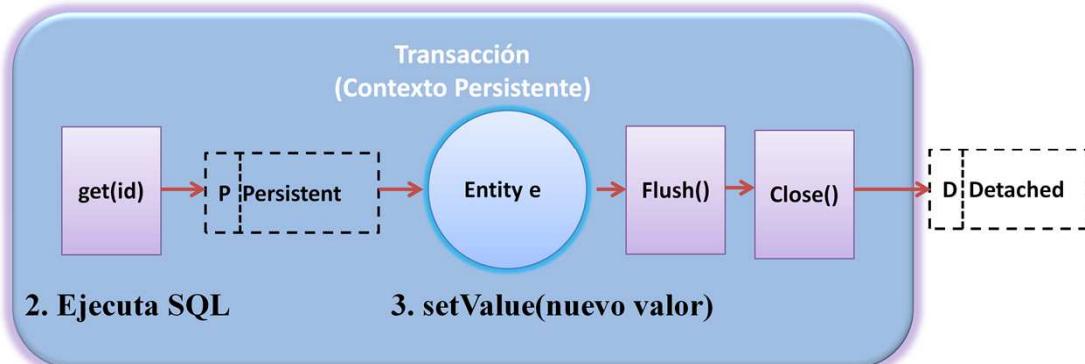
    //Paso 7. Termina transacción 2
    //Al momento de hacer commit, se revisan las diferencias
    //entre el objeto de la base de datos
    //y el objeto en memoria, y se aplican los cambios si los hubiese
    tx2.commit();

    //Objeto en estado detached ya modificado
    log.debug("Objeto recuperado:" + persona1);
}

```

Actualizar un Objeto Persistente con Sesión Larga

1. Inicia transacción



4. Termina transacción

No hay necesidad de hacer un UPDATE explícito, al terminar la transacción (commit) se ejecuta el update.

Persistir un objeto en JPA:

```
public void testActualizarObjetoSesionLarga() {
    //Paso 1. Inicia transacción 1
    EntityTransaction tx1 = em.getTransaction();
    tx1.begin();

    //Paso 2. Ejecuta SQL de tipo select
    //Puede ser un find o un merge si ya tenemos el objeto
    Persona persona1 = em.find(Persona.class, 23);

    //Paso 3. setValue (nuevoValor)
    persona1.setApeMaterno("Aguirre");

    persona1.setApeMaterno("Torres");

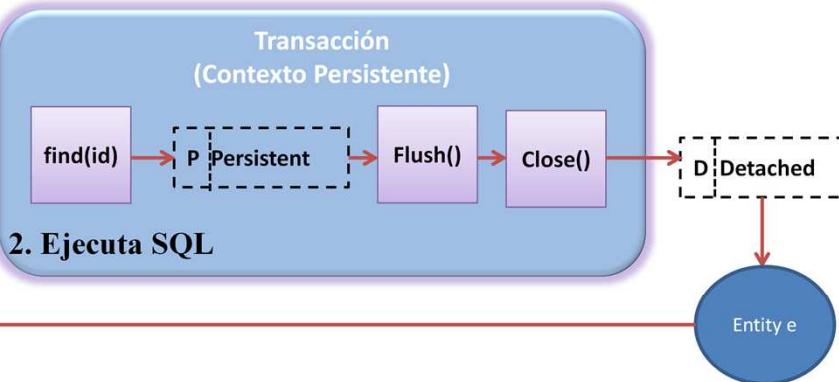
    //Paso 4. Termina transacción 1
    //Ejecuta el update, aunque hayamos hecho 2 cambios sobre el
    //objeto
    //unicamente persiste el último cambio del objeto
    //es decir, el valor de apeMaterno=Torres
    tx1.commit();

    //Objeto en estado detached
    log.debug("Objeto recuperado:" + persona1);
}
```

Eliminar un objeto en JPA

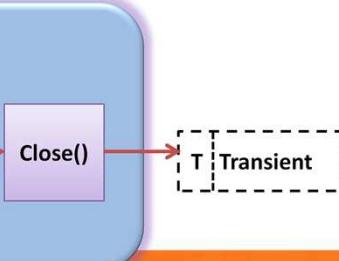
1. Inicia transacción I

3. Termina transacción I



2. Ejecuta SQL

4. Inicia transacción II



5. Ejecuta SQL

6. Termina transacción II

Persistir un objeto en JPA:

```

public void testActualizarObjetoSesionLarga() {
    // Paso 1. Inicia transacción 1
    EntityTransaction tx1 = em.getTransaction();
    tx1.begin();

    // Paso 2. Ejecuta SQL de tipo select
    Persona persona1 = em.find(Persona.class, 23);

    // Paso 3. Termina transacción 1
    tx1.commit();

    // Objeto en estado detached
    log.debug("Objeto recuperado:" + persona1);

    // Paso 4. Inicia transacción 2
    EntityTransaction tx2 = em.getTransaction();
    tx2.begin();

    // Paso 5. Ejecuta SQL (es un delete)
    em.remove(persona1);

    // Paso 6. Termina transacción 2
    // Al momento de hacer commit,
    // se realiza el delete
    tx2.commit();

    // Objeto en estado detached ya modificado
    // Ya no es posible resincronizarlo en otra transacción
    // Solo está en memoria, pero al terminar el método se eliminará
    log.debug("Objeto eliminado:" + persona1);
}

```

Laboratorio 1

Se deja como ejercicio probar cada caso creando las pruebas unitarias respectivas y validar cada uno de los casos anterior, según el código mostrado en las láminas. El código base es el siguiente:

```

package ciclovida;
import javax.ejb.embeddable.EJBContainer;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;
import mx.com.gm.sga.domain.Persona;
import org.apache.log4j.Logger;
import org.junit.After;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;

public class TestARealizarObjeto {
    static EntityManager em = null;
    static EntityTransaction tx = null;
    static EntityManagerFactory emf = null;
    Logger log = Logger.getLogger("TestEncontrarObjeto");

    @BeforeClass
    public static void init() throws Exception {
        EJBContainer.createEJBContainer();
        emf = Persistence.createEntityManagerFactory("PersonaPU");
    }

    @Before
    public void setup() {
        try {
            em = emf.createEntityManager();
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }

    @Test
    public void testARealizar() {
        //Agregar el código de la prueba
    }

    After
    public void tearDown() throws Exception {
        if (em != null) {
            em.close();
        }
    }
}

```

Relaciones en JPA

Tipos de Relaciones:

- ✓ **Uno a Uno:** @OneToOne
- ✓ **Uno a Muchos:** @OneToMany
- ✓ **Muchos a Uno:** @ManyToOne
- ✓ **Muchos a Muchos:** @ManyToMany

Direccionalidad en las relaciones:

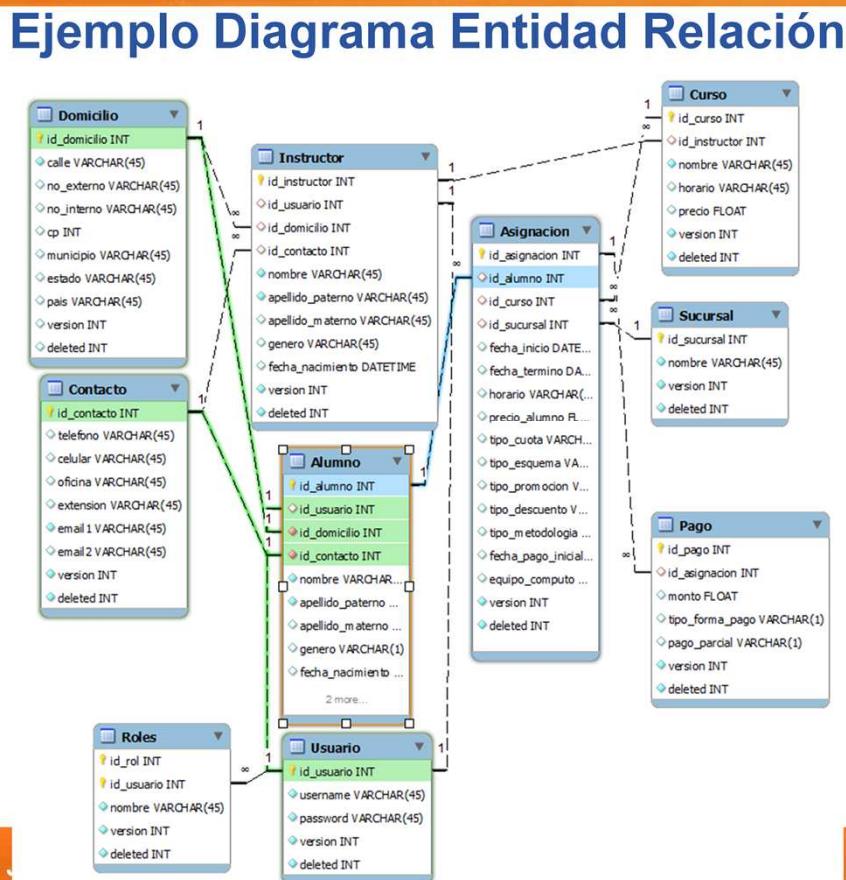
- ✓ **Unidireccional:** Se define el atributo de relación solo en una clase.
- ✓ **Bidireccional:** Se define los atributos de relación en ambas clases.

Normalmente los objetos de entidad, en un sistema con bases de datos relacionales, mantienen asociaciones con uno o más objetos. Los tipos de relaciones en JPA son las mismas que se manejan en la teoría de bases de datos relacionales.

- ✓ 1 a 1
- ✓ 1 a Muchos o Muchos a 1
- ✓ Muchos a Muchos

JPA soporta las relaciones mencionadas en los archivos de mapeo de cada clase de Entidad o en las clases Java utilizando anotaciones.

Las relaciones también tienen navegabilidad (directionality), esto quiere decir que podemos acceder a los objetos con los que tenemos relación de manera unidireccional o bidireccional. Esto lo logramos debido a que en los objetos de entidad manejamos un atributo que identifica el objeto(s) de entidad(es) con el que tenemos relación. Cuando cada objeto de entidad se apunta uno al otro por medio de este atributo o colección, se dice que es una relación bidireccional, y si por solamente una entidad apunta a la otra, la relación se conoce como unidireccional. Esto lo revisaremos en las láminas siguientes.



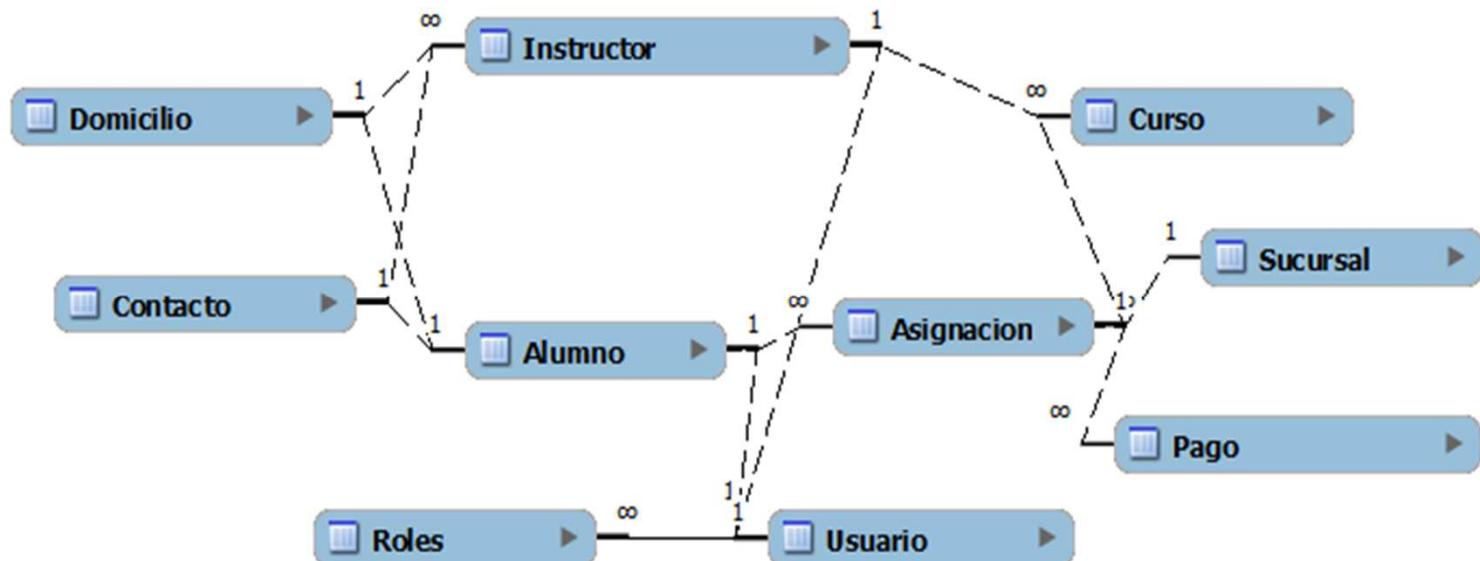
Curso de .

Mentoring

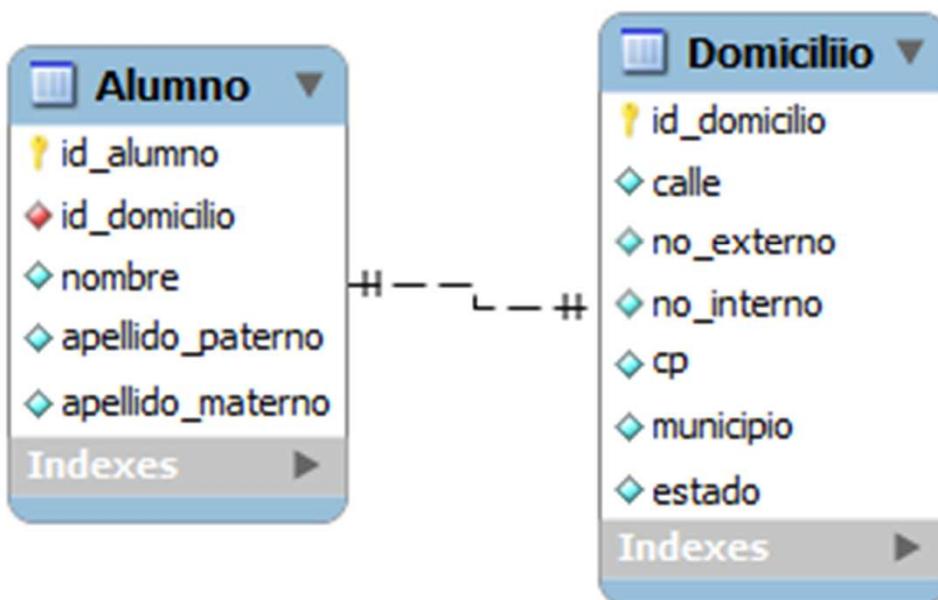
En la figura podemos observar un diagrama entidad relación con el que analizaremos las relaciones mencionadas:

- ✓ 1 a 1
- ✓ 1 a Muchos o Muchos a 1
- ✓ Muchos a Muchos

A continuación realizaremos un análisis de cada relación descrita.



Ejemplo de Relación 1 a 1 (Un Alumno tiene Un Domicilio)



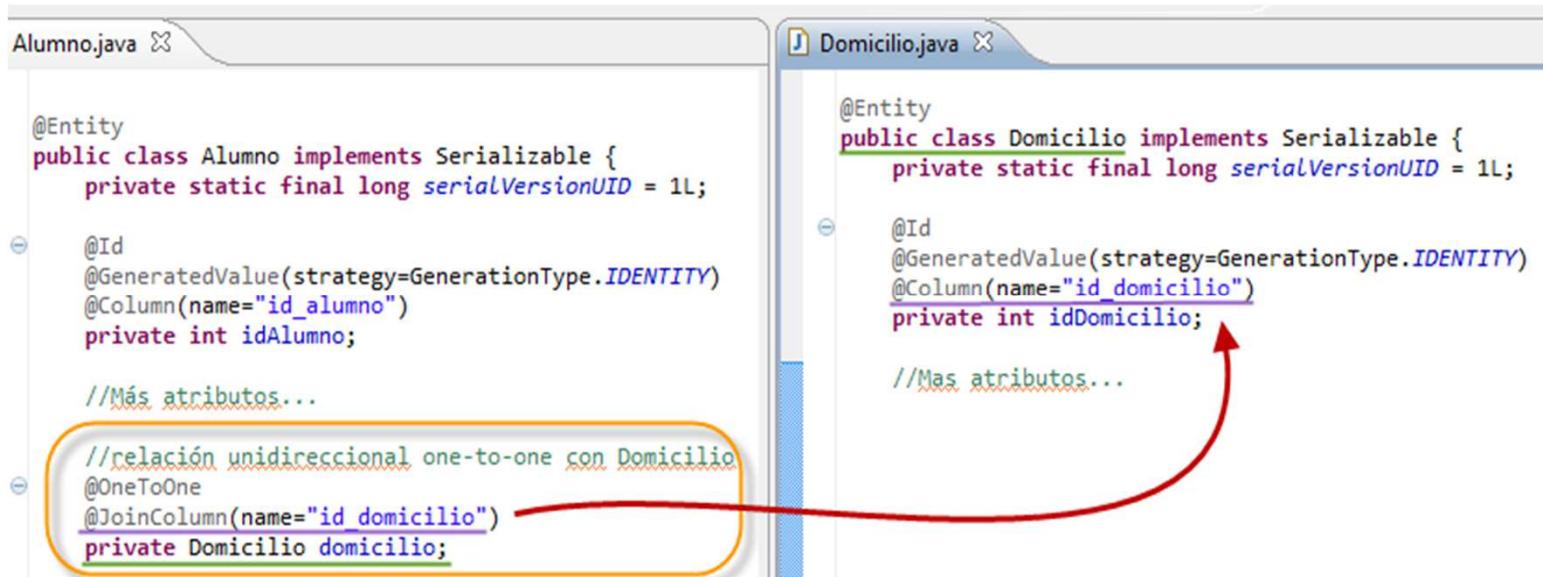
Curso de Java EE

© Derechos Reservados Global Mentoring

En la figura podemos observar una relación de 1 a 1, en la cual una entidad Alumno tiene una relación con sólo un domicilio, y viceversa, esto es, la cardinalidad entre las entidades es de 1 a 1.

Podemos observar que la clase de Alumno es la que guarda la referencia de un objeto Domicilio, para mantener una navegabilidad unidireccional y que a partir de un objeto Alumno podemos recuperar el objeto Domicilio asociado.

Es importante destacar que el manejo de relaciones es por medio de objetos, y no atributos aislados, esto nos permitirá ejecutar queries con JPQL que recuperen objetos completos. Este tema lo estudiaremos más adelante.



```

Alumno.java
@Entity
public class Alumno implements Serializable {
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    @Column(name="id_alumno")
    private int idAlumno;

    //Más atributos...

    //relación unidireccional one-to-one con Domicilio
    @OneToOne
    @JoinColumn(name="id_domicilio")
    private Domicilio domicilio;
}

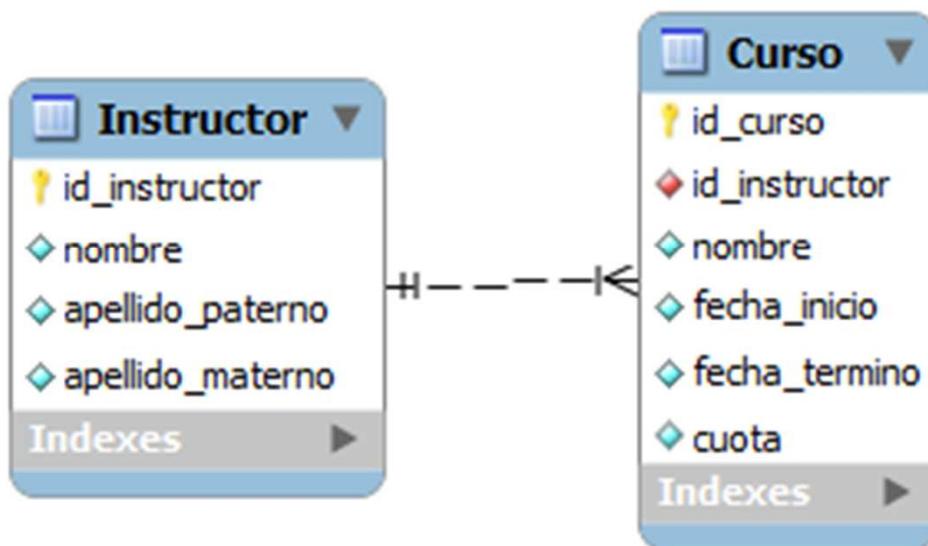
Domicilio.java
@Entity
public class Domicilio implements Serializable {
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    @Column(name="id_domicilio")
    private int idDomicilio;

    //Más atributos...
}
  
```

The screenshot shows the code for the **Alumno** and **Domicilio** entities. The **Alumno** entity has an attribute `domicilio` annotated with `@OneToOne` and `@JoinColumn`. The **Domicilio** entity has an attribute `idDomicilio` annotated with `@Id` and `@GeneratedValue`. A red arrow points from the `@OneToOne` annotation in the **Alumno** code to the `@JoinColumn` annotation in the **Domicilio** code, indicating the relationship mapping.

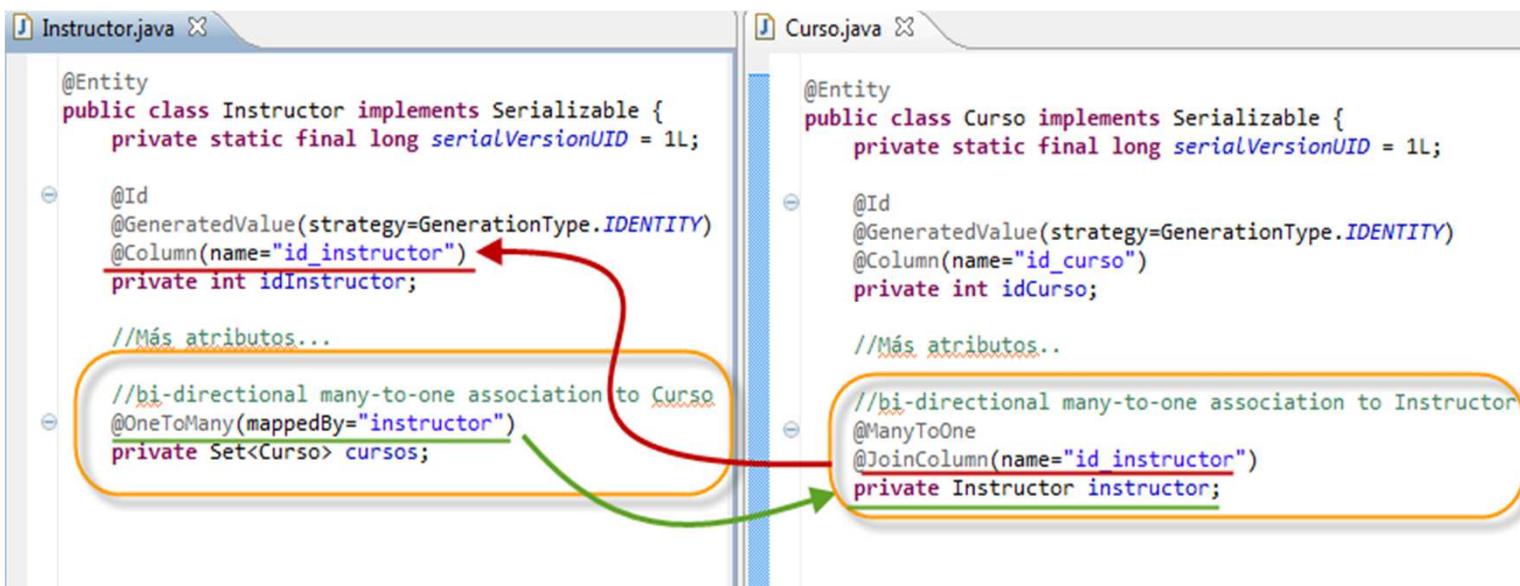
Ejemplo de Relación 1 a Muchos (Un Instructor imparte Muchos Cursos)



Cuando un objeto de Entidad está asociado con una colección de otros objetos de Entidad, es más común representar esta relación como una relación de Uno a Muchos. Por ejemplo, en la figura podemos observar la relación de un Instructor el cual puede tener asignados varios Cursos (relación de 1 a muchos).

Si queremos saber desde la clase **Curso** qué instructor tiene asociado, deberemos agregar el mapeo bidireccional (ManyToOne) hacia Alumno. Y en la clase Alumno, se especifica una relación uno a muchos (OneToMany) hacia una colección de objetos de tipo **Curso**, el cual puede ser una estructura de datos Set o un List, dependiendo si queremos manejar orden o no, respectivamente.

Si no queremos manejar una relación bidireccional, basta con eliminar la definición de alguna de las clases y así tendremos una relación unidireccional.



```

Instructor.java
@Entity
public class Instructor implements Serializable {
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    @Column(name="id_instructor")
    private int idInstructor;

    //Más atributos...

    //bi-directional many-to-one association to Curso
    @OneToMany(mappedBy="instructor")
    private Set<Curso> cursos;
}

Curso.java
@Entity
public class Curso implements Serializable {
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    @Column(name="id_curso")
    private int idCurso;

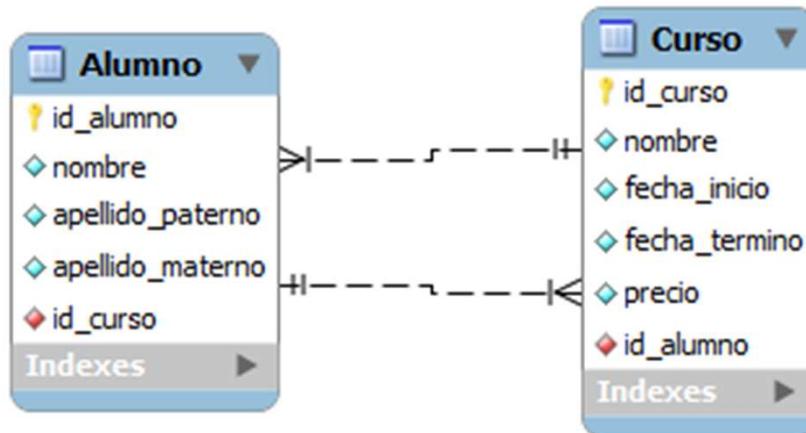
    //Más atributos...

    //bi-directional many-to-one association to Instructor
    @ManyToOne
    @JoinColumn(name="id_instructor")
    private Instructor instructor;
}
  
```

The screenshot shows the code for the **Instructor** and **Curso** entities. The **Instructor** entity has an **@Id** annotation with **GenerationType.IDENTITY**. It also has a **@OneToMany** annotation with **mappedBy="instructor"**, indicating a many-to-one relationship where the **instructor** field in the **Curso** entity maps back to the **Instructor** entity. The **Curso** entity has an **@Id** annotation with **GenerationType.IDENTITY**. It has a **@ManyToOne** annotation with **JoinColumn(name="id_instructor")**, indicating a one-to-many relationship where the **id_instructor** field in the **Instructor** entity maps to the **Curso** entity. Red arrows highlight the **mappedBy** and **JoinColumn** annotations to show the bidirectional nature of the relationship.

Ejemplo de Relación Muchos a Muchos (Un Alumno tiene Muchos Cursos y un Curso tiene Muchos Alumnos)

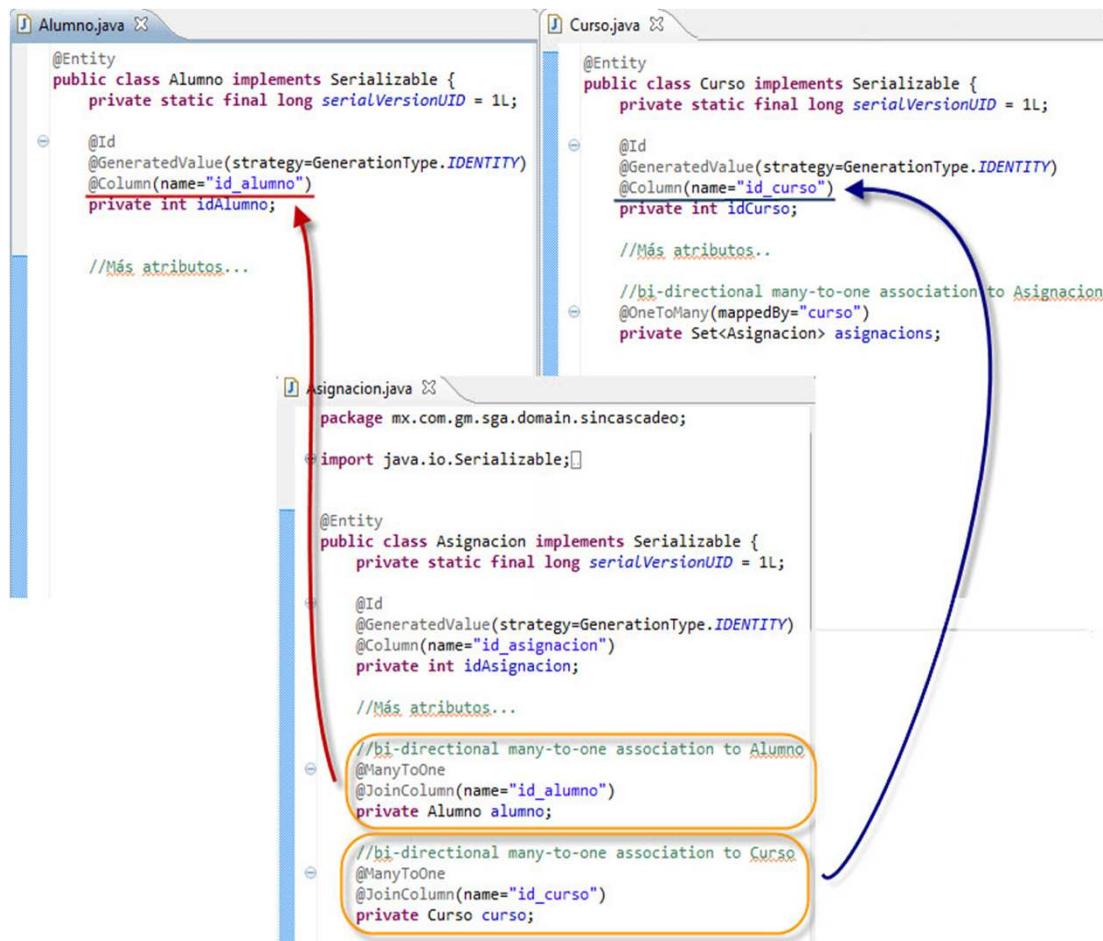
La siguiente figura muestra una relación muchos a muchos, posteriormente aplicaremos una normalización agregando una tabla transitiva llamada asignación y así convertir la relación de 1 a Muchos.



Curso de Java EE

© Derechos Reservados Global Mentoring

En la figura podemos observar una relación muchos a muchos @ManyToMany, pero ya está desnormalizada, esto con el objetivo de no utilizar relaciones muchos a muchos directamente, ya que no es recomendable.



```

Alumno.java
@Entity
public class Alumno implements Serializable {
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    @Column(name="id_alumno")
    private int idAlumno;

    //Más atributos...
}

Curso.java
@Entity
public class Curso implements Serializable {
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    @Column(name="id_curso")
    private int idCurso;

    //Más atributos..

    //bi-directional many-to-one association to Asignacion
    @OneToMany(mappedBy="curso")
    private Set<Asignacion> asignaciones;
}

Asignacion.java
package mx.com.gm.sga.domain.sincascadeo;

import java.io.Serializable;

@Entity
public class Asignacion implements Serializable {
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    @Column(name="id_asignacion")
    private int idAsignacion;

    //Más atributos...

    //bi-directional many-to-one association to Alumno
    @ManyToOne
    @JoinColumn(name="id_alumno")
    private Alumno alumno;

    //bi-directional many-to-one association to Curso
    @ManyToOne
    @JoinColumn(name="id_curso")
    private Curso curso;
}
  
```

The screenshot shows three Java code files: **Alumno.java**, **Curso.java**, and **Asignacion.java**. In **Alumno.java** and **Curso.java**, there are **@ManyToMany** annotations. In **Asignacion.java**, there are **@ManyToOne** annotations with **@JoinColumn** mappings. Red arrows indicate the relationship between the many-to-many annotations and the many-to-one annotations. Orange boxes highlight the **@ManyToOne** annotations in **Asignacion.java**.

Fetching en Relaciones

Lazy Loading: Carga Retardada

```

@Entity
public class Alumno implements Serializable {
    private static final long serialVersionUID = 1L;

    //Atributos...

    //bi-directional many-to-one association to Domicilio
    //relación de tipo Lazy, No se recuperan los datos
    //del objeto Domicilio, sino hasta que son solicitados
    @OneToOne(fetch=FetchType.LAZY)
    @JoinColumn(name="id_domicilio")
    private Domicilio domicilio;

```

Eager Loading: Carga Inmediata

```

@Entity
public class Alumno implements Serializable {
    private static final long serialVersionUID = 1L;

    //Atributos...

    //bi-directional many-to-one association to Domicilio
    //relación de tipo Eager, SI se recuperan los datos
    //del objeto Domicilio desde que se realiza la consulta
    @OneToOne(fetch=FetchType.EAGER)
    @JoinColumn(name="id_domicilio")
    private Domicilio domicilio;

```

Cuando recuperamos información de diferentes objetos de entidad (información de distintas tablas de base de datos), en ocasiones no hay necesidad de recuperar toda la información y sobrecargar la memoria Java, ya que el usuario NO necesita ver la información completa en ese momento, sino en consultas subsecuentes. Por ello JPA maneja el concepto de Lazy Loading (carga retardada), con lo cual logramos NO cargar las colecciones de entidades asociadas a cierta clase de Entidad y recuperar sólo la información que sea relevante para esa transacción.

Por ejemplo, si tenemos la relación de Instructor – Cursos (1 instructor imparte muchos Cursos), si NO queremos que la consulta recupere todos los objetos de entidad Curso asociado al objeto Instructor, deberemos configurar nuestra clase como se observar en la figura, o utilizar consultas de tipo Lazy Loading.

Un error común que nos podemos encontrar es ***Lazy Loading Exception***. Este error se produce cuando ejecutamos un query de tipo Lazy y queremos acceder a objetos que no fueron recuperados en la consulta inicial, por lo que se debe tener una transacción activa para poder recuperar los datos que no fueron cargados (puede ser durante la misma transacción o crear una nueva)

Ejemplo de consulta Lazy Loading con JPQL (NO se debe utilizar Fetch):

```
SELECT p FROM Persona p JOIN p.domicilio WHERE ...
```

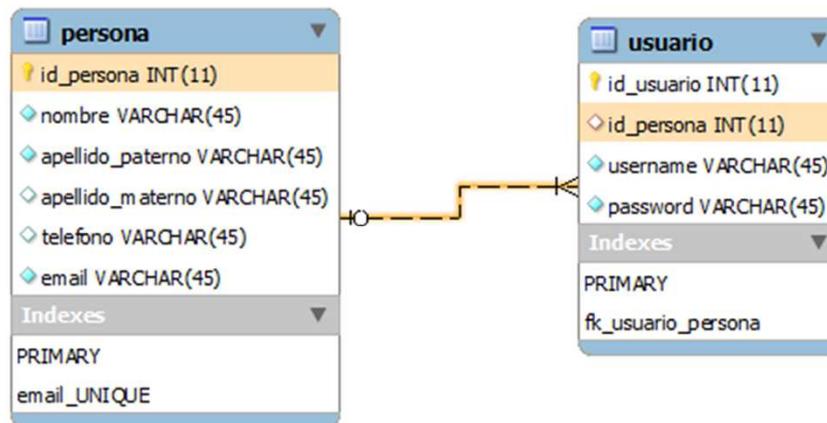
Por otro lado, tenemos el concepto de Eager Loading, esto es lo opuesto a Lazy Loading, y se utiliza cuando queremos recuperar todos los objetos de una colección asociada a un bean de Entidad. Con esto ya no necesitaremos volver a realizar consultas posteriores a la base de datos.

Ejemplo de consulta Eager Loading JPQL (se utiliza la palabra reservada Fetch):

```
SELECT p FROM Persona p JOIN FETCH p.domicilio WHERE ...
```

Como hemos comentado, el concepto de Lazy e Eager loading aplica también en las consultas con JPQL, de hecho sobreescriben el comportamiento definido en la clase de Entidad, sin embargo en ocasiones encontraremos que nuestras clases están configuradas con los parámetros Lazy e Eager, y es importante saber la diferencia y para qué funciona cada uno.

Guardado en Cascada



Otro concepto muy interesante en JPA es el concepto de persistencia en Cascada.

Por ejemplo, si queremos persistir un objeto de tipo Usuario, pero además queremos insertarlo con sus dependencias (Persona) y que JPA se encargue de generar las llaves primarias respectivas, así como la generación de las sentencias *insert* en el orden adecuado (primero inserta el objeto persona y después el objeto Usuario), podemos apoyarnos de la configuración de persistencia en cascada.

Para configurar nuestra clase de entidad Usuario con persistencia en cascada utilizamos la siguiente sintaxis:

```

@在这儿
@Entity
public class Usuario {

    //Más atributos...

    //bi-directional many-to-one association to Persona
    @ManyToOne(cascade={CascadeType.ALL})
    @JoinColumn(name="id_persona")
    private Persona persona;
}

```

Podemos all, persist, merge, remove, refresh

En todos estos casos es posible aplicar el concepto de persistencia en cascada.



Laboratorio 2

Se deja como ejercicio el guardado en cascada, creando una prueba unitaria y verificar que al guardar un objeto Usuario, se guarde en automático el objeto Persona asociado.

Además validar, ¿Qué pasa si no se agrega la configuración de Cascade al objeto Usuario y se pretende realizar el guardado? Este es el código a realizar.

```
@Test
public void testPersistenciaCascada() {

    EntityTransaction tx1 = em.getTransaction();
    tx1.begin();

    //Creamos el objeto persona (es el menos dependiente)
    Persona personal = new Persona("Hugo", "Sanchez", "Pinto", "hsanchez@mail.com", "09093090");

    //Creamos el objeto usuario (tiene dependencia con un objeto persona)
    Usuario usuario1 = new Usuario("hsanchez", "123", personal);

    //Solo Persistimos el objeto Usuario
    //No hay necesidad de persistir el objeto Persona
    em.persist(usuario1);

    tx1.commit();

    //Ya debe tener el objeto persona y su PK asociada
    log.debug("Objeto recuperado:" + usuario1);
}
```

Java Persistence Query Language (JPQL)

Java Persistence Query Language (JPQL):

- ✓ Lenguaje de Consulta, similar a SQL pero utilizando objetos Java.
- ✓ Queries Parametrizables.
- ✓ Consola de Ejecución en IDE's como Eclipse o MyEclipse.
- ✓ Consultas Avanzadas con recuperación de colecciones de datos.

Características de JPQL:

- ✓ Uso de select, from y where y subselects.
- ✓ Sensible a Mayúsculas/Minúsculas.
- ✓ Asociaciones, uso de joins y fetch.
- ✓ Uso de expresiones y operadores como: +, >, between, upper, etc.
- ✓ Uso de Funciones de agregación, tales como: avg, sum, count, etc.
- ✓ Uso de order by y group by

Java Persistence Query Language (JPQL) nos permite recuperar información de la Base de Datos. JPQL se enfoca en ejecutar queries que regresen objetos Java, en lugar de datos individuales.

¿Qué es Java Persistence Query Language (JPQL)?:

- ✓ Es un Lenguaje de Consulta, similar a SQL pero utilizando objetos Java.
- ✓ Permite ejecutar Queries Parametrizables.
- ✓ Cuenta con una Consola de Ejecución en IDE's como Eclipse o MyEclipse.
- ✓ Se pueden ejecutar Consultas Avanzadas para recuperar colecciones de datos.
- ✓ Ejemplo consulta con JPQL: **SELECT p FROM Persona p**

Características de JPQL:

- ✓ Uso de select, from y where y subselects.
- ✓ Sensible a Mayúsculas/Minúsculas.
- ✓ Asociaciones, uso de joins y fetch.
- ✓ Uso de expresiones y operadores como: +, >, between, upper, etc.
- ✓ Uso de Funciones de agregación, tales como: avg, sum, count, etc.
- ✓ Uso de order by y group by

JPA permite recuperar los objetos de diferentes maneras, tanto utilizando una sintaxis muy similar a SQL, pero también ofrece otras alternativas, utilizando código Java, conocido como API de Criteria y Query By Example.

Tipos de Queries:

- ✓ **Dynamic queries:** Consultas que reciben parámetros en tiempo de ejecución.
- ✓ **Named queries:** Consultas ya creadas previamente, y que se pueden ejecutar solo utilizando el nombre.
- ✓ **Native queries:** Consultas con SQL nativa, ya que hay casos de uso que lo requieren.
- ✓ **Criteria API queries:** Consultas con una sintaxis con código Java, en lugar de SQL.

Queries de Tipo Select en JPQL

A continuación revisaremos varios ejemplos utilizando JPQL:

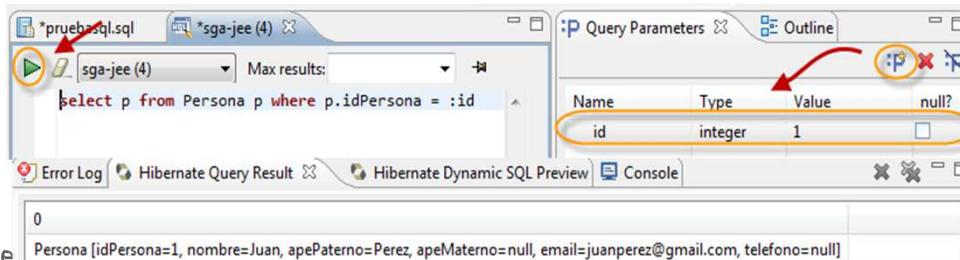
- 1) Consulta de todas las Personas: `select p from Persona p`
- 2) Consulta de la Persona con id = 1: `select p from Persona p where p.idPersona = 1`
- 3) Consulta de la Persona por nombre: `select p from Persona p where p.nombre = 'Juan'`
- 4) Consulta de datos individuales, se crea un arreglo (tupla) de tipo object de 3 columnas:
`select p.nombre as Nombre, p.apePaterno as Paterno, p.apeMaterno as Materno from Persona p`
- 5) Obtiene el objeto Persona y el id, se crea un arreglo de tipo Object con 2 columnas:
`select p, p.idPersona from Persona p`
- 6) Obtiene la lista de alumnos, utilizando el constructor del idPersona
`select new mx.com.gm.sga.domain.Persona(p.idPersona) from Persona p`
- 7) Regresa el valor mínimo y máximo del idPersona (Scalar results)
`select min(p.idPersona) as MinId, max(p.idPersona) as MaxId, count(p.idPersona) as Contador from Persona p`
- 8) Extrae los nombres de alumnos que son distintos
`select count(distinct p.nombre) from Persona p`

9) Concatena y Convierte a mayúsculas el nombre y apellido:

`select CONCAT (p.nombre, ' ', p.apePaterno) as Nombre FROM Persona p`

10) Obtiene el objeto alumno con id igual al parámetro:

`select p from Persona p where p.idPersona = :id`



11) Obtiene las personas que tienen nombre Juan Perez

`select p from Persona p where upper(p.nombre) like upper(:param1)`

12) Uso de between:

`select p from Persona p where p.idPersona between 1 and 2`

13) Uso del ordenamiento:

`select p from Persona p where p.idPersona > 3
order by p.nombre desc, p.apePaterno desc`

14) Uso de un subquery (el soporte de esta funcionalidad depende de la base de datos utilizada)

`select p from Persona p
where p.idPersona in (select min(p1.idPersona) from Persona p1)`

15) Uso de join con lazy loading: `select u from Usuario u join u.persona p`

16) Uso de left join con eager loading: `select u from Usuario u left join fetch u.persona`

Código Java para ejecutar un query JPQL

El siguiente código Java permite ejecutar los queries JPQL descritos:

```

    @Test
    public void testActualizarObjeto() {

        String jpql = null;
        List<Persona> personas;

        EntityTransaction tx1 = em.getTransaction();
        tx1.begin();

        //1) Consulta de todas las Personas:
        log.debug("1) Consulta de todas las Personas");

        jpql = "select p from Persona p";

        personas = em.createQuery(jpql).getResultList();
        for(Persona p : personas){
            log.debug(p);
        }

        tx1.commit();
    }

||0  [main] DEBUG TestJPQL - 1) Consulta de todas las Personas
360 [main] DEBUG TestJPQL - Persona [idPersona=1, nombre=Juan, apePaterno=Perez, apeMaterno=null, email=juanperez@gmail.com, tel
360 [main] DEBUG TestJPQL - Persona [idPersona=2, nombre=Oscar, apePaterno=Gomez, apeMaterno=Larios, email=ogomez@mail.com, tele
360 [main] DEBUG TestJPQL - Persona [idPersona=22, nombre=Angelica, apePaterno=Lara, apeMaterno=Gomez, email=alara@mail.com, tel
360 [main] DEBUG TestJPQL - Persona [idPersona=23, nombre=Hugo, apePaterno=Sanchez, apeMaterno=Pinto, email=hsanchez@mail.com, t

```

Curso de Java EE

© Derechos Reservados Global Mentoring

Como podemos observar en la figura, utilizando los queries descritos anteriormente podemos ejecutar con ayuda del API de JPA, las sentencias JPQL que hemos construido. Existen distintas formas de procesar las consultas dependiendo de la información a recuperar.

Por ejemplo, si queremos recuperar la lista de resultados:

```
String jpql = "select p from Persona p";
List<Persona> personas = em.createQuery(jpql).getResultList();
```

Si queremos recuperar sólo un registro:

```
String jpql = "select p from Persona p where p.idPersona = 1";
Persona persona = (Persona) em.createQuery(jpql).getSingleResult();
```

Si queremos procesar un conjunto de valores (tupla):

```
String jpql = "select p.nombre as Nombre, p.apePaterno as Paterno from Persona p";
iter = em.createQuery(jpql).getResultList().iterator();
while(iter.hasNext()){
    tupla = (Object[]) iter.next();
    String nombre = (String) tupla[0];
    String apePat = (String) tupla[1];
}
```

Si queremos enviar parámetros a un query:

```
String jpql = "select p from Persona p where p.idPersona = :id";
Query q = em.createQuery(jpql);
q.setParameter("id", 1);
persona = (Persona) q.getSingleResult();
```

Entre varios ejemplos más, los cuales incluimos en los ejercicios resueltos del curso.

API de Criteria en JPA

API Criteria en JPA (* Nuevo en JPA 2.0)

- ✓ El API de Criteria es una alternativa al uso de JPQL o SQL Nativo
- ✓ Permite la combinación de campos de criterio complejos (ej. Una pantalla de búsqueda avanzada)
- ✓ Permite crear queries dinámicos complejos más fácilmente, evitando el concatenado de cadenas.

Características del API de Criteria

- ✓ Son escritos en código Java
- ✓ Son typesafe (parámetros revisados en tiempo de compilación)
- ✓ Son queries portables (funcionan en cualquier base de datos)
- ✓ Se utilizan clases de Java en lugar de cadenas JPQL o SQL
- ✓ Permite utilizar expresiones, joins, ordenamiento, etc

La construcción de queries dinámicos conlleva mucha concatenación de cadenas si estamos utilizando JDBC directo, incluso en queries complejos utilizando JPQL podemos tener demasiados parámetros concatenados en la cadena JPQL.

Para simplificar este proceso, JPA en su versión 2.0 liberó el API de Criteria, el cual está basado en el API de Criteria de Hibernate, entre otros frameworks ORM, por lo tanto si ya se tiene conocimiento de Hibernate y su API de Criteria o alguno similar, será más sencillo aprender esta API.

Con el API de Criteria, es posible construir consultas dinámicas complejas, permitiendo configurar desde el mismo lenguaje de programación (y no de tipo SQL) los filtros necesarios para dicha consulta.

Sin embargo tanto JPQL como el API de Criteria tiene su propio lugar dependiendo del problema que se pretenda resolver. El API de Criteria es muy utilizado cuando tenemos pantallas con demasiados filtros de búsqueda, y el usuario tiene la opción de seleccionar uno o más filtros. En cambio las consultas con JPQL funcionan muy bien cuando tenemos un número establecido de parámetros y la mayoría de nuestro query es estático (no existe demasiada concatenación de parámetros).

La forma más rápida de aprender a utilizar el API de Criteria es comparando las consultas que realizamos con JPQL, ya que con ello podremos ver las diferencias y ver los pros y contras de cada API. Así que a continuación revisaremos y compararemos varios de estos queries utilizando las APIs mencionadas.

Código Java para ejecutar un query con API Criteria

El siguiente código Java permite ejecutar los queries API Criteria:

```

// Query utilizando el API de Criteria
// 1) Consulta de todas las Personas:

log.debug("\n1) Consulta de todas las Personas");

//1. El objeto EntityManager crea instancia CriteriaBuilder
CriteriaBuilder cb = em.getCriteriaBuilder();

//2. Se crear un objeto CriteriaQuery
CriteriaQuery<Persona> criteriaQuery = cb.createQuery(Persona.class);

//3. Creamos el objeto Raiz del query
Root<Persona> fromPersona = criteriaQuery.from(Persona.class);

//4. Seleccionamos lo necesario del from
criteriaQuery.select(fromPersona);

//5. Creamos el query typeSafe
TypedQuery<Persona> query = em.createQuery( criteriaQuery);

//6. Ejecutar la consulta
List<Persona> personas = query.getResultList();
mostrarPersonas(personas);

0  [main] DEBUG TestJPQL - 1) Consulta de todas las Personas
360 [main] DEBUG TestJPQL - Persona [idPersona=1, nombre=Juan, apePaterno=Perez, apeMaterno=null, email=juanperez@gmail.com, tel
360 [main] DEBUG TestJPQL - Persona [idPersona=2, nombre=Oscar, apePaterno=Gomez, apeMaterno=Larios, email=ogomez@mail.com, tel
360 [main] DEBUG TestJPQL - Persona [idPersona=22, nombre=Angelica, apePaterno=Lara, apeMaterno=Gomez, email=alaraj@mail.com, tel
360 [main] DEBUG TestJPQL - Persona [idPersona=23, nombre=Hugo, apePaterno=Sanchez, apeMaterno=Pinto, email=hsanchez@mail.com, t

```

Podemos observar el código para ejecutar consultas utilizando el API de Criteria:

```

// Query utilizando el API de Criteria
// 1) Consulta de todas las Personas:

log.debug("\n1) Consulta de todas las Personas");

//1. El objeto EntityManager crea instancia CriteriaBuilder
CriteriaBuilder cb = em.getCriteriaBuilder();

//2. Se crear un objeto CriteriaQuery
CriteriaQuery<Persona> criteriaQuery = cb.createQuery(Persona.class);

//3. Creamos el objeto Raiz del query
Root<Persona> fromPersona = criteriaQuery.from(Persona.class);

//4. Seleccionamos lo necesario del from
criteriaQuery.select(fromPersona);

//5. Creamos el query typeSafe
TypedQuery<Persona> query = em.createQuery( criteriaQuery);

//6. Ejecutar la consulta
List<Persona> personas = query.getResultList();
mostrarPersonas(personas);

```



Laboratorio 3

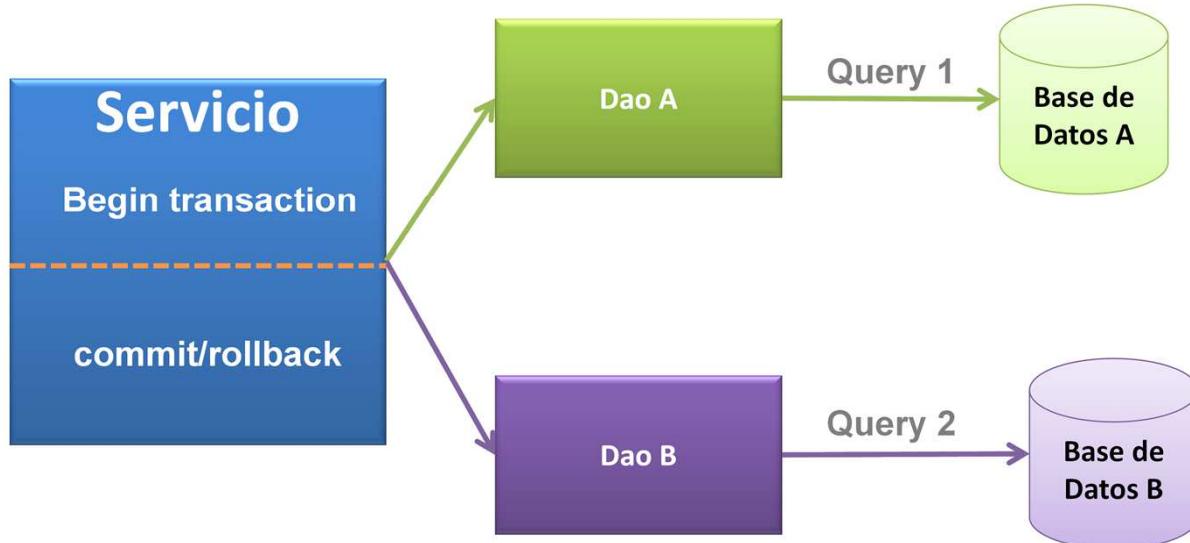
En la sección de ejercicios del curso se entregan los queries tanto de JPQL como del API de Criteria.

Se deja como ejercicio ejecutar los queries de JPQL en la consola de Eclipse (Hibernate) y también ejecutar las pruebas unitarias para comprobar el código Java que se integren adecuadamente los queries creados.

Además, se deja revisar a detalle cada query con el objetivo de ir resolviendo dudas al utilizar estas APIs.

¿Qué es una Transacción?

- Una transacción se conoce como una unidad de trabajo atómica, es decir, se realiza toda o nada del método transaccional.



El Manejo Transaccional es uno de los temas cruciales en cuanto a requerimientos para aplicaciones empresariales. Esta motivación surge debido a que en todo sistema empresarial nos interesa mantener la integridad de nuestra información, con esto en mente es que surge el tema de transacciones.

En la figura podemos observar un método de servicio que ejecuta llamadas a más de un DAO, y a su vez cada DAO modifica el estado de la base de datos al escribir y/o modificar su información.

El objetivo de una transacción es ejecutar todas las líneas de código de nuestro método y guardar finalmente la información en un repositorio, por ejemplo en nuestro caso, una base de datos. Esto se conoce como **commit** de nuestra transacción.

Si por alguna razón algo fallara en nuestro método de Servicio, se daría marcha atrás a los cambios realizados en la base de datos. Esto se conoce como **rollback**.

Lo anterior permite que nuestra información, ya sea que se una única base de datos o no, esté íntegra, y no exista posibilidad de datos corruptos por errores o fallas en la ejecución de nuestros métodos Java.

Por ejemplo, imaginemos un sistema de venta de boletos de avión por internet. En este caso, los pasos necesarios para reservar un boleto son:

- 1) Verificar boletos disponibles
- 2) Reservar un boleto
- 3) Realizar el pago
- 4) Recibir los datos del boleto

Si por alguna razón alguno de los pasos falla, entonces el boleto no se debe considerar como un boleto vendido, sino como un boleto disponible. En caso de que todo se haya realizado exitosamente, entonces el boleto ya no está disponible y se actualiza el estado de la base de datos para reflejar un boleto menos para los nuevos usuarios interesados en comprar un boleto de avión.

Así que los métodos que hagamos partícipes de una transacción se ejecutarán como una sola acción, lo que garantiza que se ejecutan por completo, o si fallan, no se persista ninguna de la información. Los EJB al ejecutarse en un contenedor Java empresarial, por default son transaccionales, por lo tanto, cualquier método de un EJB maneja este concepto en automático.

Características de una Transacción

Las características de una transacción tienen el acrónimo **ACID**:

Atomic: Las actividades de un método se consideran como una unidad de trabajo. Esto se conoce como *Atomicidad*.

Consistent: Una vez que termina una transacción la información queda en estado *consistente*, ya que se realiza todo o nada.

Isolated: Múltiples usuarios pueden utilizar los métodos transaccionales, sin embargo debemos prevenir errores por accesos múltiples, *aislando* en la medida de lo posible nuestros métodos transaccionales.

Durable: Sin importar si hay una caída del servidor, una transacción exitosa debe guardarse y *perdurar* posterior al término de una transacción.

Las características de una transacción tienen el acrónimo **ACID**:

Atomicidad: Las actividades de un método se consideran como una unidad de trabajo. Esto se conoce como *Atomicidad*. Este concepto asegura que todas las operaciones en una transacción se ejecuta todo o nada.

Si todas las instrucciones o líneas de código de un método transaccional son ejecutadas con éxito, entonces al finalizar se realiza un **commit**, es decir, guardado de la información.

Si alguna de las instrucciones falla se realiza un **rollback**, es decir, ninguna de la información es guardada en la base de datos o el repositorio donde se persiste dicha información.

Consistente: Una vez que termina una transacción (sin importar si ha sido exitosa o no) la información queda en estado *consistente*, ya que se realizó todo o nada, y por lo tanto los datos no deben estar corruptos en ningún aspecto.

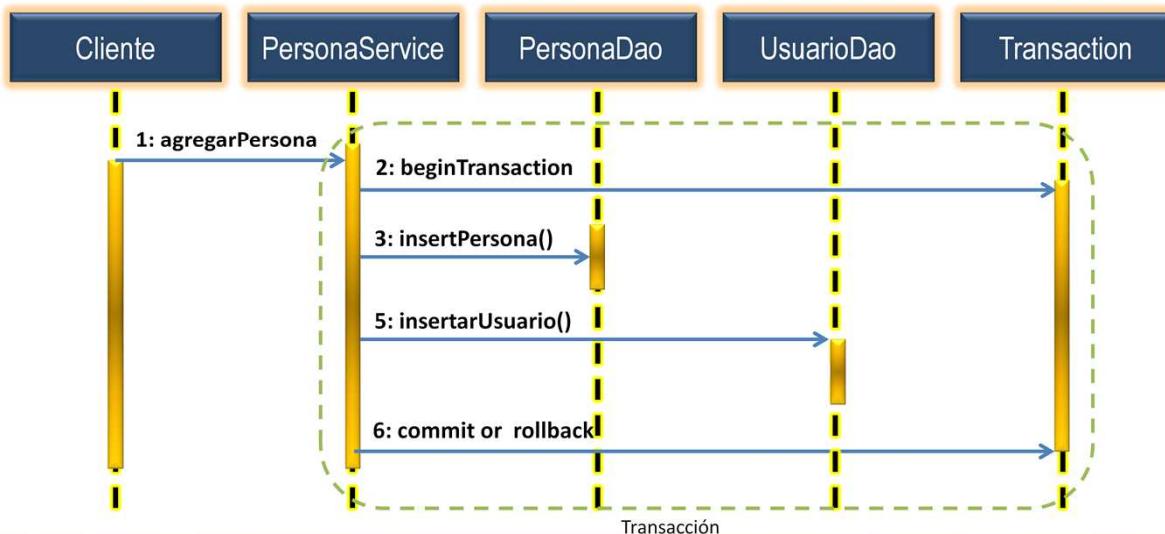
Aislado: Múltiples usuarios pueden utilizar los métodos transaccionales, sin afectar el acceso de otros usuarios. Sin embargo debemos prevenir errores por accesos múltiples, *aislando* en la medida de lo posible nuestros métodos transaccionales. El aislamiento normalmente involucra el bloqueo de registros o tablas de base de datos, esto se conoce como *locking*.

Durable: Sin importar si hay una caída del servidor, una transacción exitosa debe guardarse y *perdurar* posterior al término de una transacción.

En nuestro ejemplo de la compra de boletos, una transacción puede asegurar la **atomicidad** debido a que puede deshacer todos los cambios realizados si alguno de los pasos falla. También se aplicaría la **consistencia** de datos al asegurar que no se deja nada de manera parcial, ejecutando todo o nada. El **aislamiento** prevé que otro usuario tome el boleto reservado mientras todavía no termina la transacción. Y finalmente al momento de hacer commit o rollback de la transacción la información es almacenada sin inconsistencias, permitiendo que sea **durable** incluso posterior a la finalización de la transacción.

Manejo de Transacciones

Manejo de una transacción en una arquitectura de 3 capas Java Empresarial.



Al crear aplicaciones empresariales debemos poner especial atención en la persistencia. Además, una aplicación empresarial, según hemos estudiado, se divide en distintas capas, las cuales tienen responsabilidades bien definidas.

En la figura podemos observar en qué momento el objeto Transaction entra en participación. Observamos las 3 capas de una arquitectura empresarial (Cliente – Servicio – Acceso a Datos). En esta arquitectura la transacción comienza en la capa de servicio, la cual es la encargada de establecer la comunicación con la Capa de Datos (DAO).

La capa de datos es la responsable de establecer la comunicación con la base de datos a través del objeto Entity Manager. La transacción comienza desde la capa de servicio, y se propaga a la capa de datos. Esto se debe a que la capa de servicio puede tener comunicación con muchas clases Dao, y por lo tanto la transacción termina hasta que el método de negocio haya insertado, modificado, eliminado y seleccionado todos los datos requeridos de la base de datos, aplicando las características del manejo transaccional que hemos estudiado.

Los métodos de la capa de Servicio, son los que contienen mucha de la lógica de negocio de la aplicación, y por lo tanto es en este nivel donde definimos el manejo transaccional, ya que si lo aplicamos al nivel de la capa de datos, el manejo de commit/rollback por cada operación de un DAO, afectaría las operaciones restantes que tenga un método de negocio.

Configuración de la Propagación en Java EE

Demarcación de transacciones por medio de Container-Managed Transactions (CMTs)

| Tipo de Propagación | Significado |
|---------------------|---|
| MANDATORY | El método tiene que ejecutarse dentro de una transacción, de lo contrario se lanzará una excepción. |
| REQUIRED | El método DEBE ejecutarse dentro de una transacción. Si ya existe una transacción el método la utilizará, de lo contrario creará una nueva. |
| REQUIRES_NEW | El método DEBE ejecutarse dentro de una transacción. Si ya existe una transacción, se suspende durante la ejecución del método, de lo contrario creará una nueva. |
| SUPPORTS | Indica que el método no requiere el manejo transaccional, pero puede participar de una transacción si ya hay alguna ejecutándose. |
| NOT_SUPPORTED | El método NO debe ejecutarse en una transacción. Si ya existe una transacción, se suspenderá hasta la conclusión del método. |
| NEVER | El método NO debe ejecutarse en una transacción, de lo contrario lanza una excepción. |

La propagación indica cómo se comportará un método ante una transacción que ha sido iniciada previamente en otro método, es decir, cómo una transacción se propaga entre métodos transaccionales. El valor por defecto de la propagación es REQUIRED. Los tipos de propagación en una transacción son los siguientes:

- MANDATORY: Si no existe una transacción se lanza una excepción.
- REQUIRED: Este es el comportamiento por default. Si ya existe una transacción, se propaga y la utiliza, de lo contrario crea una nueva.
- REQUIRES_NEW: Únicamente debe usarse si la acción sobre la base de datos necesita confirmarse (commit) sin importar el resultado de la transacción en curso, por ejemplo, un registro en una bitácora de actividades, por cada intento de persistencia de información, sin importar si se persiste exitosamente o no.
- SUPPORTS: No necesita de una transacción, sin embargo si existe una la utiliza.
- NOT_SUPPORTED: Debido a que indica que no soporta transacciones, si hubiera alguna, la suspende hasta terminar la ejecución del método marcado con este atributo.
- PROPAGATION_NEVER : Lanza una excepción si está ejecutándose una transacción.

Para indicar si un método maneja un método distinto de programación se utiliza el código antes de la definición del método:

```
@TransactionAttribute(TransactionAttributeType.SUPPORTS)
```

Ejemplo Código de Transacciones

Ejemplo de código de transacciones administradas por el contenedor (CMT):

```

@Stateless
public class PersonaServiceImpl {

    @Resource
    private SessionContext contexto;

    public void modificarPersona(Persona persona) {
        try {
            personaDao.updatePersona(persona);
        } catch (Throwable t) {
            contexto.setRollbackOnly();
        }
    }

    //Más métodos...
}

```

Según hemos visto, el manejo de transacciones se lleva a cabo en los EJB de la capa de servicio o negocio. Como observamos en el código mostrado, el EJB PersonaService contiene métodos de servicio, los cuales por default son transaccionales al ejecutarse en un contenedor empresarial.

El comportamiento por default de un método EJB es de tipo REQUIRED. Sin embargo si queremos modificarlo podemos utilizar el código TransactionAttribute(TransactionAttributeType.SUPPORTS) ya sea a nivel de la clase para que afecte a todos los métodos, o al nivel del método que queremos modificar.

Cualquier código que espere arrojar una excepción, y si queremos aplicar rollback de manera explícita deberemos utilizar setRollbackOnly() del objeto SessionContext, el cual podemos injectar directamente utilizando la anotación @Resource.

Sin embargo, utilizar el método setRollbackOnly no provocará el rollback de la transacción de manera inmediata, es sólo una indicación de que en cuanto el contenedor termine de ejecutar el método, deberá realizarse el rollback.

Cada transacción JTA está asociada con la ejecución de un hilo (Thread), así que solo se puede ejecutar una transacción a la vez. De tal manera que si una transacción se encuentra activa, el usuario NO puede iniciar otra dentro del mismo hilo (a menos que dicha transacción sea suspendida).



Laboratorio 4

En la sección de ejercicios del curso se encuentra el proyecto modificado para el manejo transaccional.

Se deja como ejercicio hacer el test de la prueba unitaria del manejo transaccional.

Además, se deja revisar y probar con más métodos con el objetivo de ir resolviendo dudas al utilizar el manejo transaccional.

Conclusión

Los temas que estudiamos son bastante extensos, sin embargo con las bases que se han sentado, es mucho más sencillo atacar a más detalle estos temas. A continuación les dejamos algunas referencias para profundizar más en estos temas.

- Consultas con el JPQL:
<http://docs.oracle.com/javaee/6/tutorial/doc/bnbt1.html>
- Consultas con el API de Criteria:
<http://docs.oracle.com/javaee/6/tutorial/doc/gjivm.html>
- Otros ejemplos de JPA:
<http://www.objectdb.com/java/jpa/query/jpql/string>
- Tema de transacciones:
<http://docs.oracle.com/javaee/6/tutorial/doc/bncij.html>


Videotrainning >
Curso de Java EE



www.globalmentoring.com.mx

Pasión por la tecnología Java

Experiencia y Conocimiento para tu vida

© Derechos Reservados Global Mentoring

En Global Mentoring promovemos la Pasión por la Tecnología Java.

Te invitamos a visitar nuestro sitio Web donde encontrarás cursos Java Online desde Niveles Básicos, Intermedios y Avanzados.

Además agregamos nuevos cursos para que continúes con tu preparación como consultor Java de manera profesional.

A continuación te presentamos nuestro listado de cursos en constante crecimiento:

- ✓ Fundamentos de Java
- ✓ Programación con Java
- ✓ Java con JDBC
- ✓ HTML, CSS y JavaScript
- ✓ Servlets y JSP's
- ✓ Struts Framework
- ✓ Hibernate Framework
- ✓ Spring Framework
- ✓ JavaServer Faces
- ✓ Java EE (EJB, JPA y Web Services)
- ✓ JBoss Administration

Datos de Contacto:

Sitio Web: www.globalmentoring.com.mx

Email: informes@globalmentoring.com.mx

Ayuda en Vivo: www.globalmentoring.com.mx/chat.html