

### 3. Java Persistence Api (JPA)

- 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.

### 3. Java Persistence Api (JPA)

- 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.

# 3. Java Persistence Api (JPA)

## ¿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)



## 3. Java Persistence Api (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.
- Si ya conoces Hibernate, o algún framework de persistencia Java, te será muy familiar muchos de los conceptos que estudiaremos en esta lección.

# 3. Java Persistence Api (JPA)

## Características de JPA

### Características de Java Persistence API:

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



### 3. Java Persistence Api (JPA)

- 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.

# 3. Java Persistence Api (JPA)

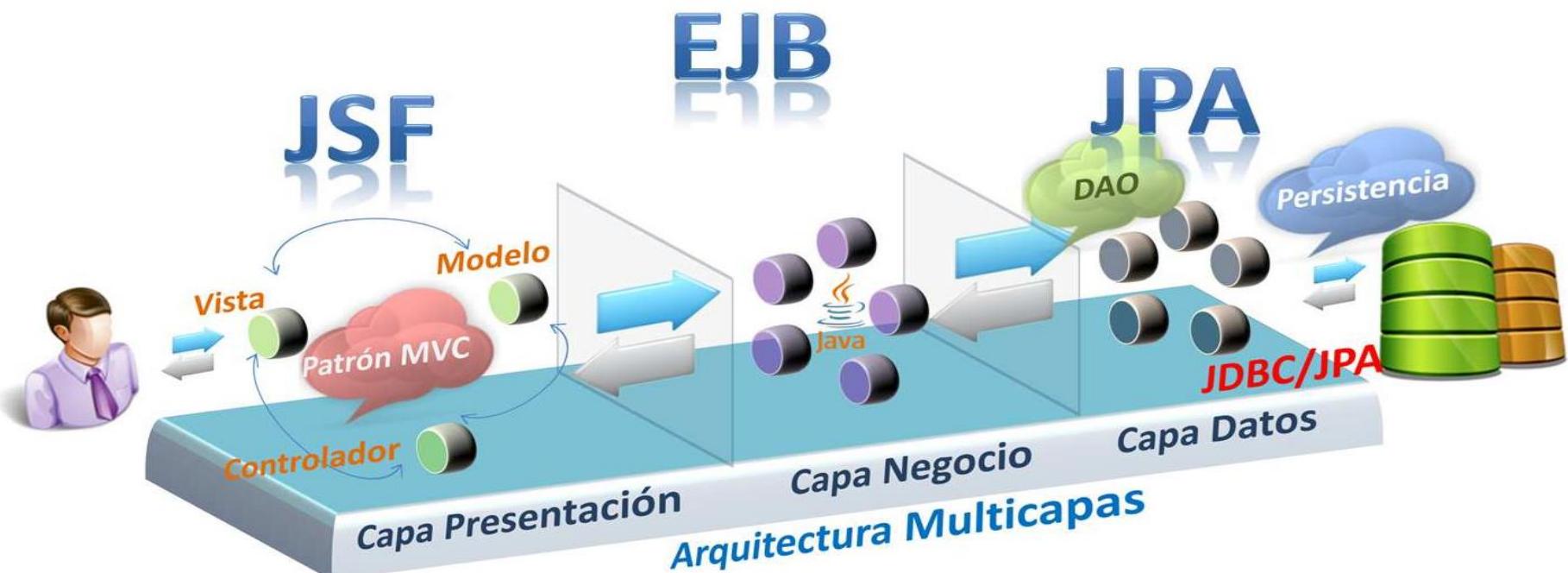
- Características de Java Persistence API:
  - **Persistencia utilizando POJOs:** Este es posiblemente el aspecto más importante de JPA, debido a que cualquier clase de Java podemos convertirla en una clase de entidad, simplemente agregando anotaciones y/o agregando un archivo xml de mapeo.
  - **No 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 expresadas en términos de objetos Java y sus relaciones, sin necesidad de utilizar el lenguaje SQL. Las queries son traducidos por el API de JPA en el código SQL equivalente.

# 3. Java Persistence Api (JPA)

- Características de Java Persistence API:
  - **Configuración simple**: Muchas de las **opciones** de JPA están configuradas con opciones **por defecto**, 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.

# 3. Java Persistence Api (JPA)

## Arquitectura Empresarial con JPA



Servidor de  
Aplicaciones Java

### 3. Java Persistence Api (JPA)

- 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 (Persistance 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.

### 3. Java Persistence Api (JPA)

- 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**.

# 3. Java Persistence Api (JPA)

## 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  
}
```

### 3. Java Persistence Api (JPA)

- 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 rendimiento. 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

### 3. Java Persistence Api (JPA)

- 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.

### 3. Java Persistence Api (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 así 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.

### 3. Java Persistence Api (JPA)

- 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.

### 3. Java Persistence Api (JPA)

- 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.

### 3. Java Persistence Api (JPA)

- 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).

### 3. Java Persistence Api (JPA)

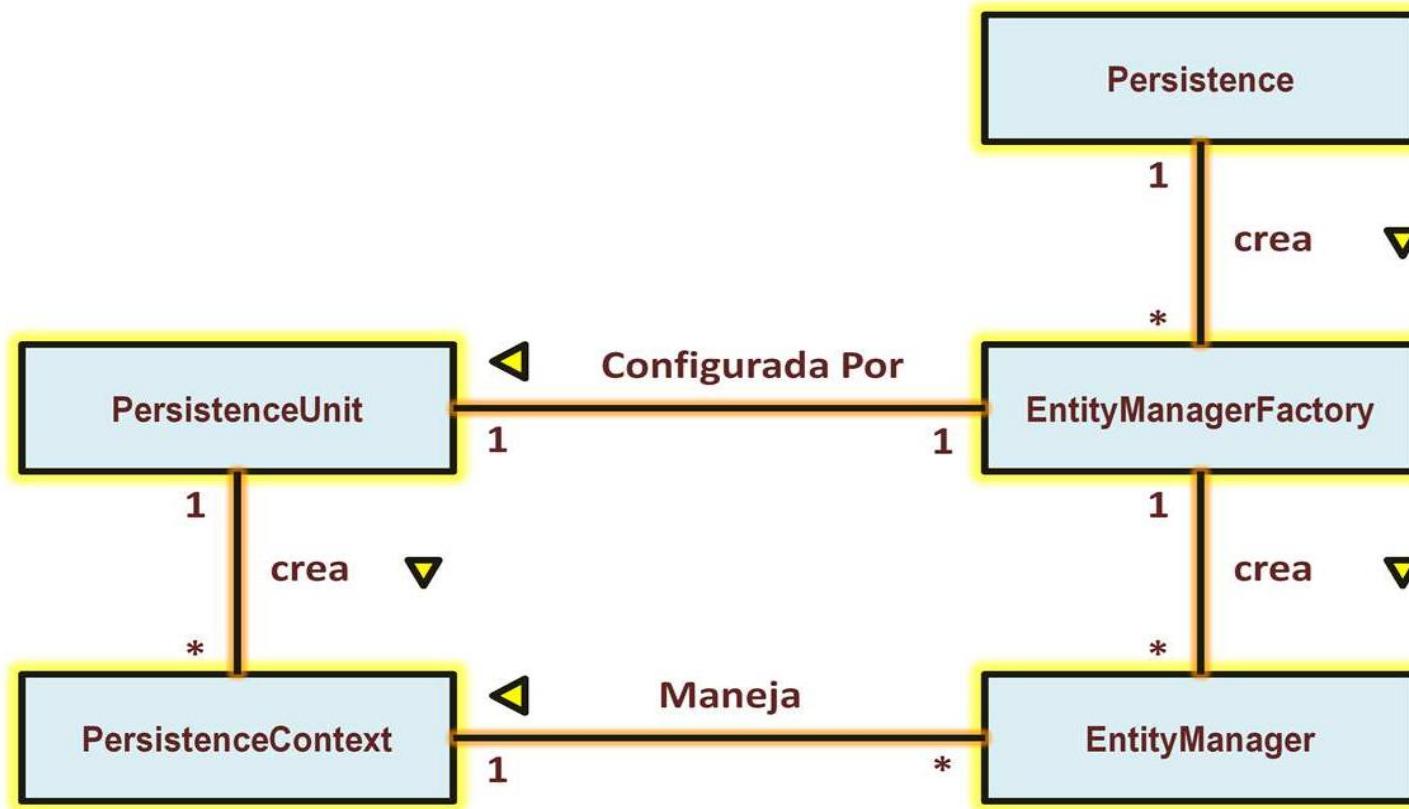
- El objeto Persistance Unit, se encarga de realizar la configuración del proveedor seleccionado por medio de un archivo xml, además de definir otros elementos tales como: la forma de comunicarse con la Base de Datos, las clase de Entidad en la aplicación, si se va a utilizar JTA para el manejo transaccional,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

### 3. Java Persistence Api (JPA)

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

# 3. Java Persistence Api (JPA)

## API de JPA y Entity Manager



## 3. Java Persistence Api (JPA)

- 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();  
Persona persona = new Persona(15);  
em.persist( persona );
```

# 3. Java Persistence Api (JPA)

## 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>
```

### 3. Java Persistence Api (JPA)

- 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.

### 3. Java Persistence Api (JPA)

- 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.

### 3. Java Persistence Api (JPA)

- 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.

# 3. Java Persistence Api (JPA)

## 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);
    }
}
```

# 3. Java Persistence Api (JPA)

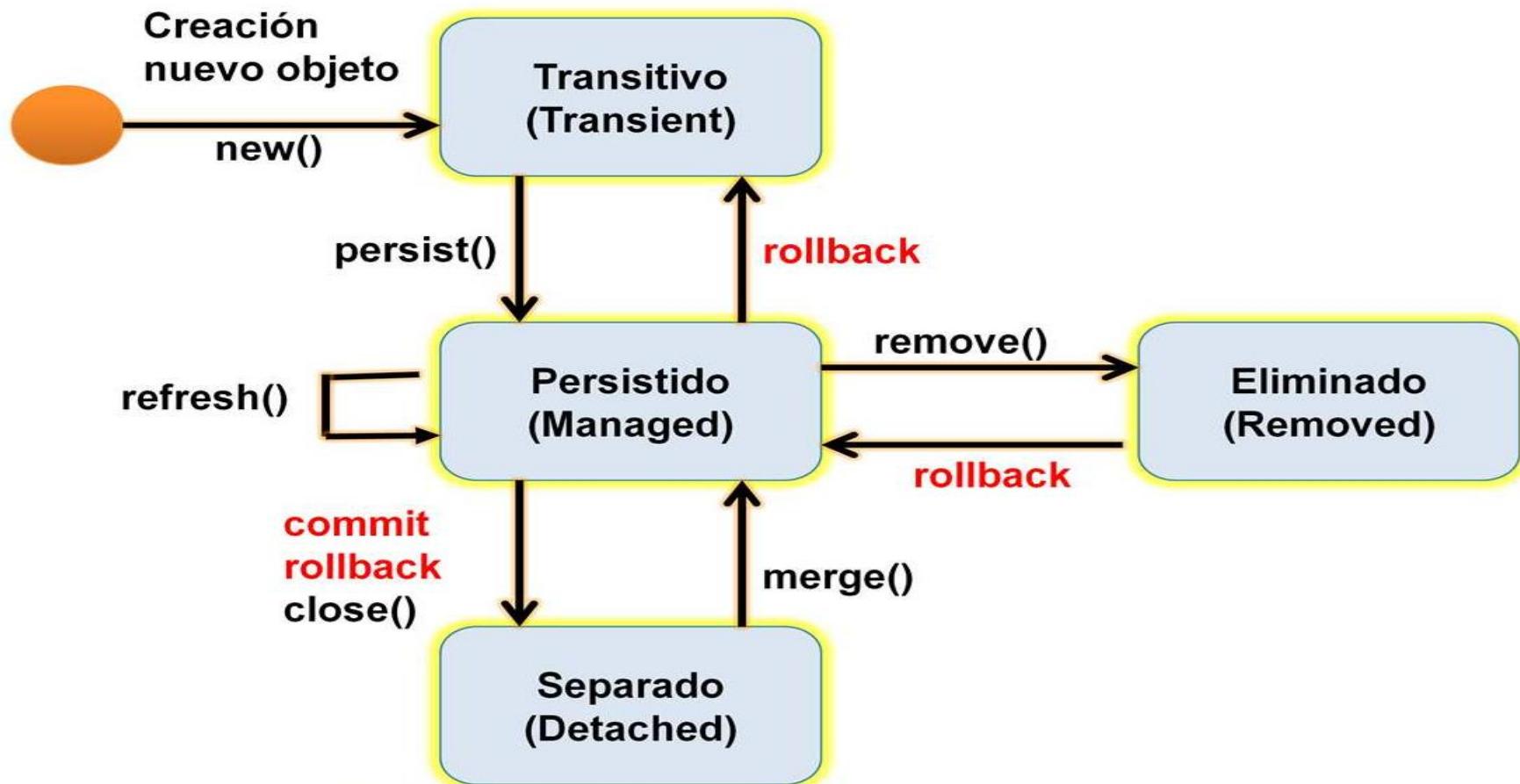
- Por ejemplo, podemos realizar tareas como:
  - **Inserción:** Para persistir una entidad se utiliza el método **persist** del EntityManager. Con este método podemos generar un registro en la base de datos. Este registro no será guardado hasta haber concluido la transacción (commit). Los métodos de 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.
  - **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.

### 3. Java Persistence Api (JPA)

- **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**.

# 3. Java Persistence Api (JPA)

## Ciclo de Vida Entidad JPA



### 3. Java Persistence Api (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.
- Los estados del ciclo de vida son.
  - **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.

# 3. Java Persistence Api (JPA)

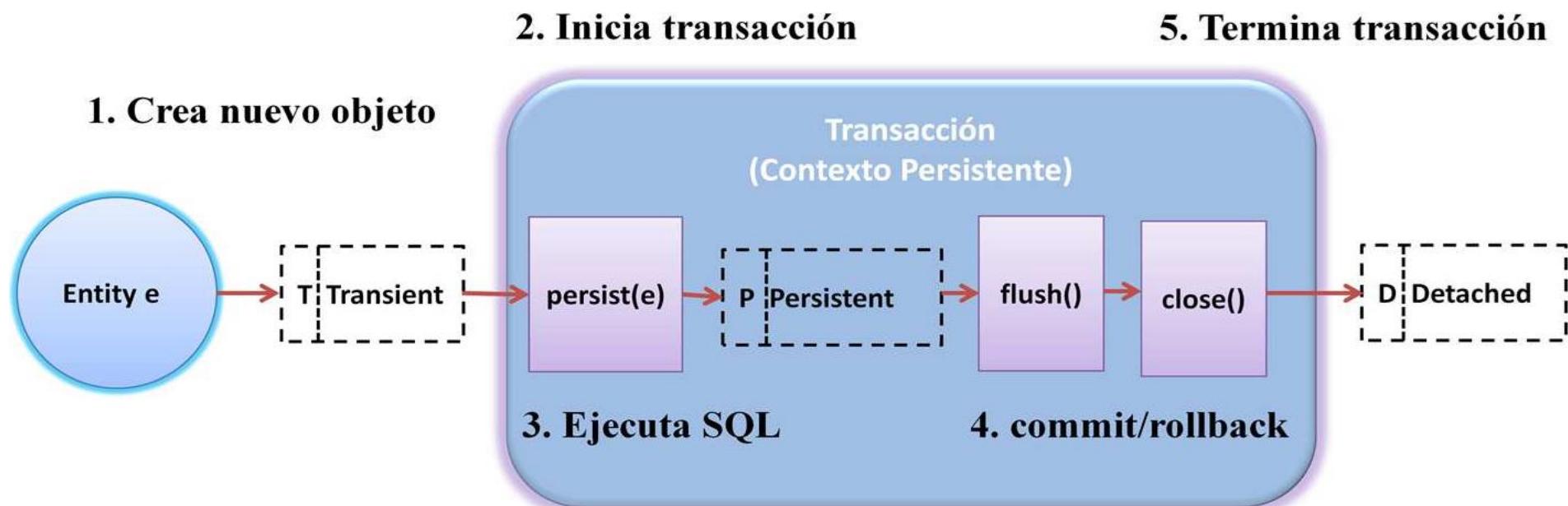
- **Estado Separado (Detached):**
  - Estos objetos tienen asociado un registro de BD, pero su estado no está sincronizado con la BD
  - Todos los objetos recuperados en una transacción se convierten en detached una vez que termina la misma
- **Estado Eliminado (Removed):**
  - Estos objetos ya no tiene una representación con la BD y al terminar la transacción, el registro asociado es eliminado.

### 3. Java Persistence Api (JPA)

- 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.

# 3. Java Persistence Api (JPA)

## Persistir un objeto en JPA



# 3. Java Persistence Api (JPA)

- Código ejemplo para persistir un objeto en JPA:

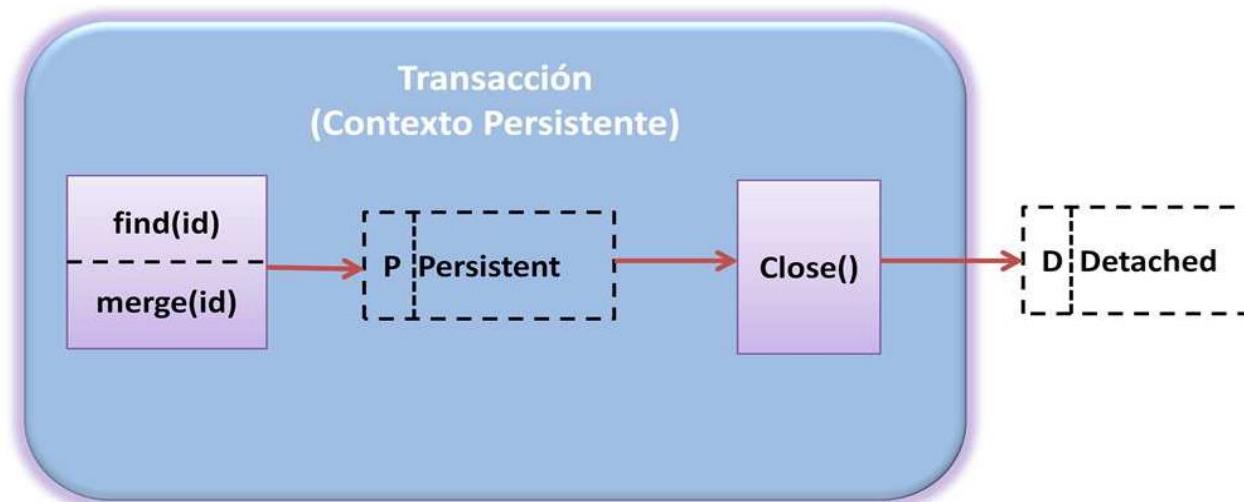
```
public void testPersistirObjeto() {  
    //Paso 1. Crea nuevo objeto  
    //Objeto en estado transitivo Persona persona1 = new  
    Persona("Pedro","Luna",null,"pluna@mail.com","19292943");  
  
    //Paso 2. Inicia transacción  
    EntityTransaction tx = em.getTransaction();  
    tx.begin();  
  
    //Paso 3. Ejecuta SQL  
    em.persist(persona1);  
  
    //Paso 4. Commit/rollback  
    tx.commit();  
  
    //Objeto en estado detached  
    log.debug("Objeto persistido:" + persona1);  
}
```

# 3. Java Persistence Api (JPA)

## Recuperar un objeto de Entidad en JPA

1. Inicia transacción

3. Termina transacción



2. Ejecuta SQL

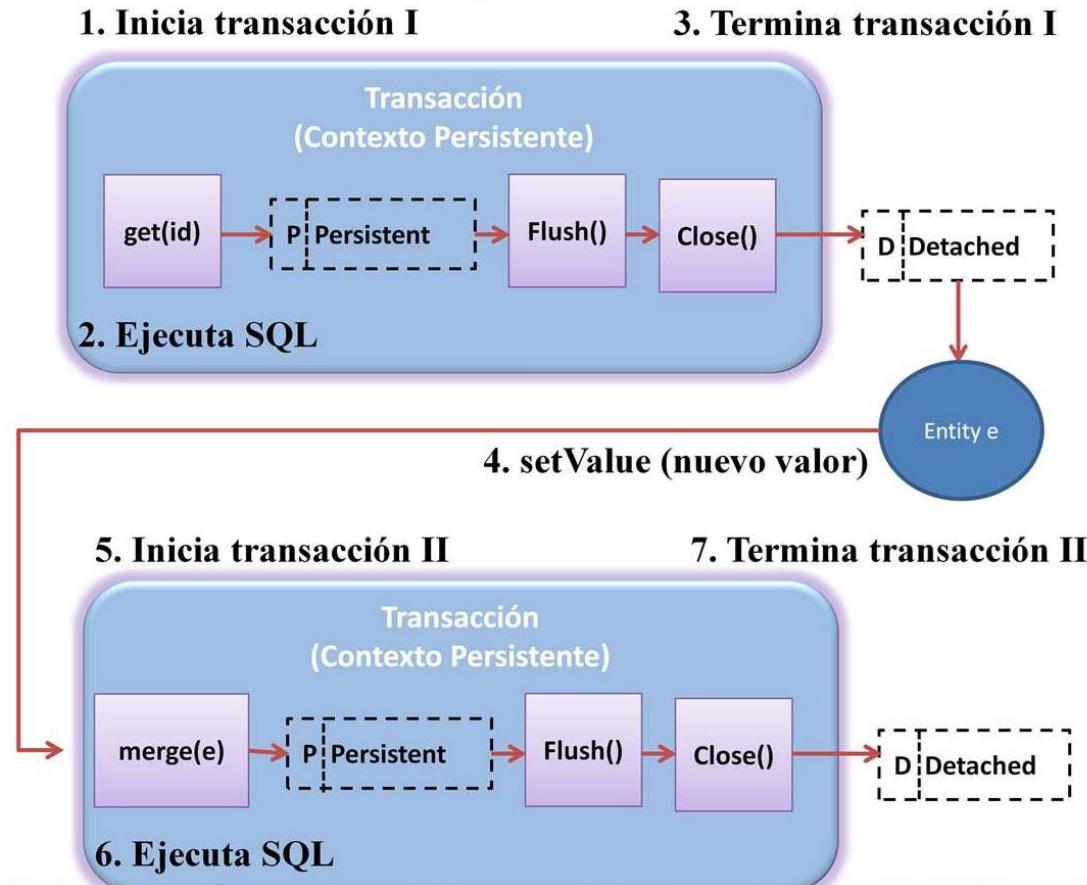
# 3. Java Persistence Api (JPA)

- 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);  
}
```

# 3. Java Persistence Api (JPA)

## Actualizar un Objeto Persistente en JPA



# 3. Java Persistence Api (JPA)

- 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 personal = em.find(Persona.class, 23);  
  
    //Paso 3. Termina transacción 1  
    tx1.commit();  
  
    //Objeto en estado detached |  
    log.debug("Objeto recuperado:" + personal);  
  
    //Paso 4. setValue (nuevoValor)  
    personal.setApellido("Nava");  
  
    //Paso 5. Inicia transacción 2  
    EntityTransaction tx2 = em.getTransaction();  
    tx2.begin();
```

### 3. Java Persistence Api (JPA)

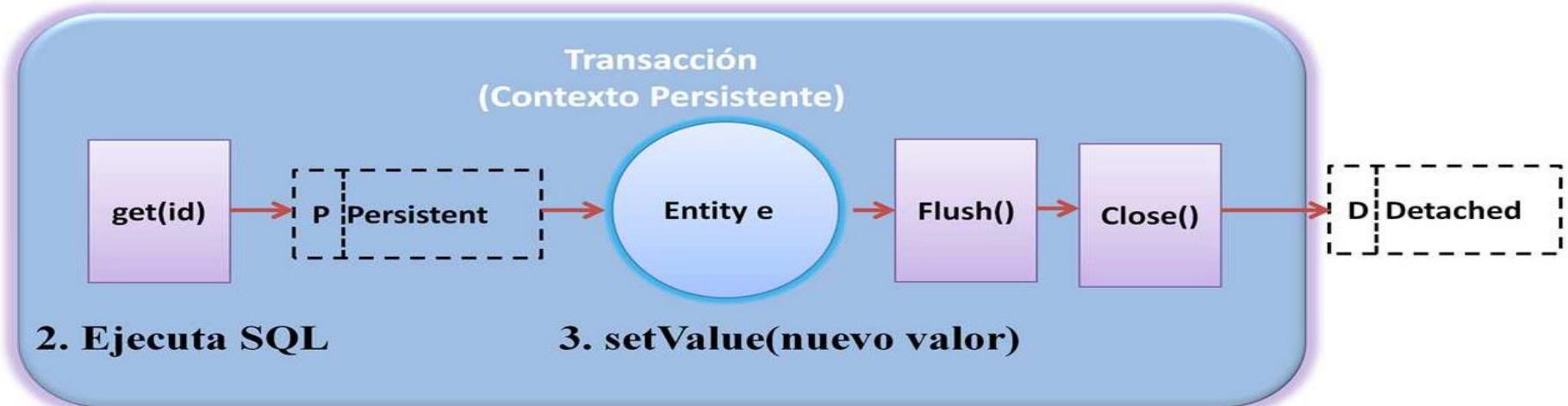
```
//Paso 6. Ejecuta SQL. Es un select, pero al estar modificado,  
//al terminar la transacción hará un update  
//Como ya tenemos el objeto hacemos solo un merge para resincronizar  
//el objeto a hacer merge, debe contar con el valor de la llave primaria  
em.merge(persona1);  
  
//Paso 7. Termina 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);  
}
```

# 3. Java Persistence Api (JPA)

## 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.

# 3. Java Persistence Api (JPA)

- 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);  
}
```

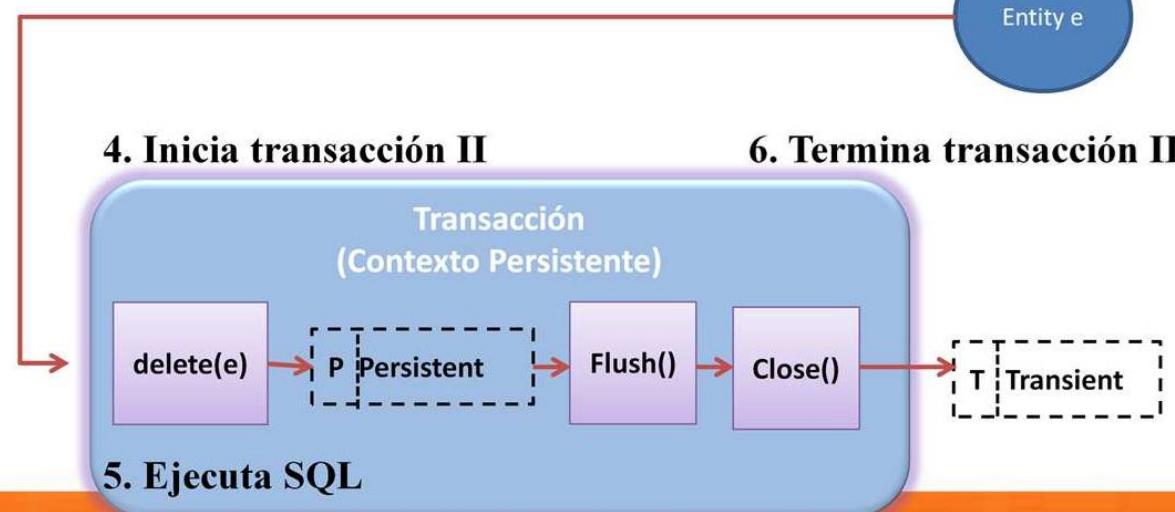
# 3. Java Persistence Api (JPA)

## Eliminar un objeto en JPA

1. Inicia transacción I



3. Termina transacción I



### 3. Java Persistence Api (JPA)

```
public void testActualizarObjetoSesionLarga() {  
    // Paso 1. Inicia transacción 1  
    EntityTransaction tx1 = em.getTransaction();  
    tx1.begin();  
  
    // Paso 2. Ejecuta SQL de tipo select  
    Persona personal1 = em.find(Persona.class, 23);  
  
    // Paso 3. Termina transacción 1  
    tx1.commit();  
  
    // Objeto en estado detached  
    log.debug("Objeto recuperado:" + personal1);  
  
    // Paso 4. Inicia transacción 2  
    EntityTransaction tx2 = em.getTransaction();  
    tx2.begin();  
  
    // Paso 5. Ejecuta SQL (es un delete)  
    em.remove(personal1);  
  
    // 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:" + personal1);  
}
```

### 3. Java Persistence Api (JPA)

```
public void testActualizarObjetoSesionLarga() {  
    // Paso 1. Inicia transacción 1  
    EntityTransaction tx1 = em.getTransaction();  
    tx1.begin();  
  
    // Paso 2. Ejecuta SQL de tipo select  
    Persona personal1 = em.find(Persona.class, 23);  
  
    // Paso 3. Termina transacción 1  
    tx1.commit();  
  
    // Objeto en estado detached  
    log.debug("Objeto recuperado:" + personal1);  
  
    // Paso 4. Inicia transacción 2  
    EntityTransaction tx2 = em.getTransaction();  
    tx2.begin();  
  
    // Paso 5. Ejecuta SQL (es un delete)  
    em.remove(personal1);  
  
    // 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:" + personal1);  
}
```

# 3. Java Persistence Api (JPA)

## 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.

### 3. Java Persistence Api (JPA)

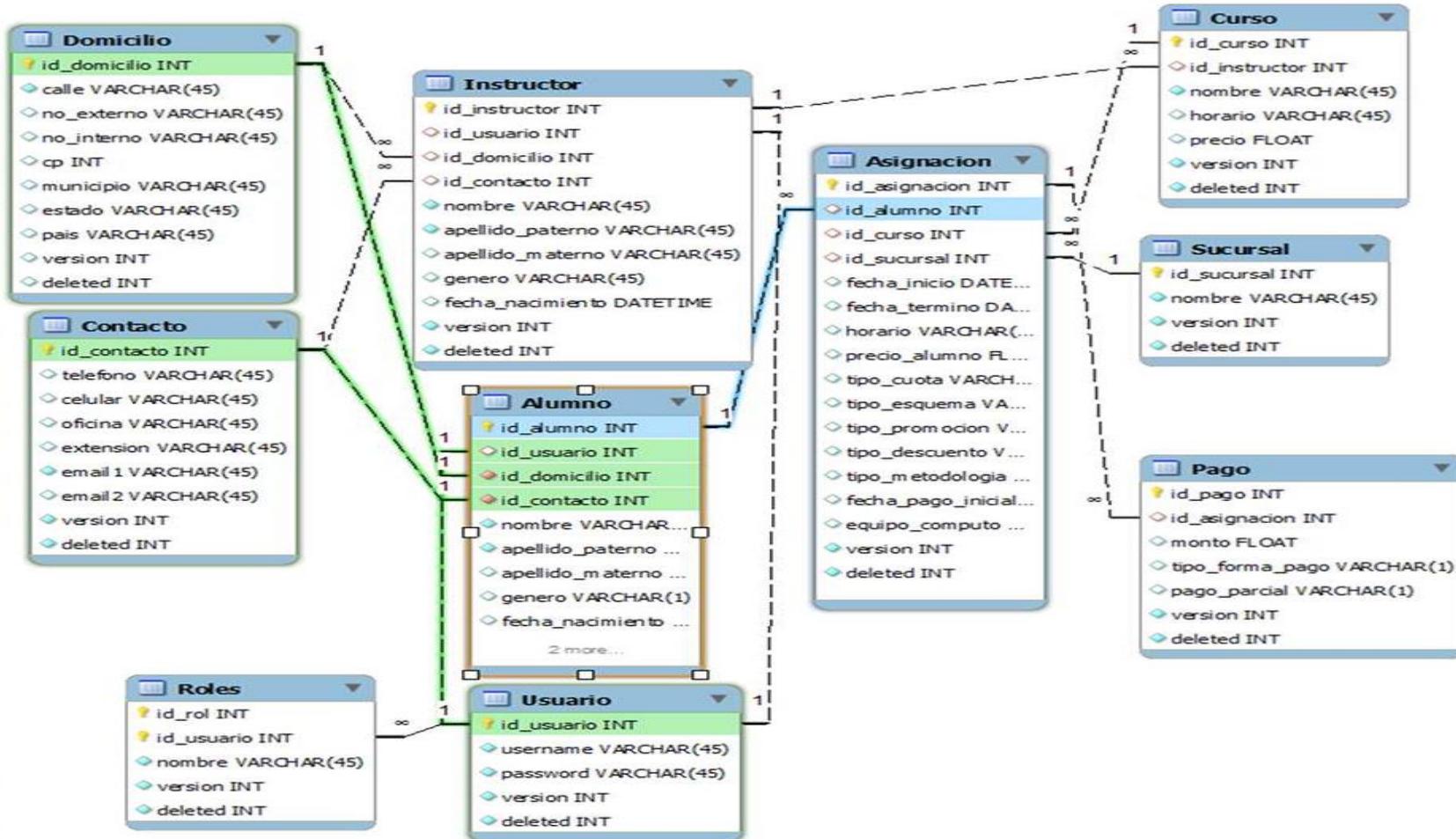
- Normalmente los objetos de entidad, en un sistema con bases de datos relacionales, mantienen asociaciones con uno o más objetos. Los tipos de relaciones en JPA son las mismas que se manejan en la teoría de bases de datos relacionales.
  - 1 a 1
  - 1 a Muchos o Muchos a 1
  - Muchos a Muchos

### 3. Java Persistence Api (JPA)

- Las relaciones también tienen **navegabilidad** (directionalidad), 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**.

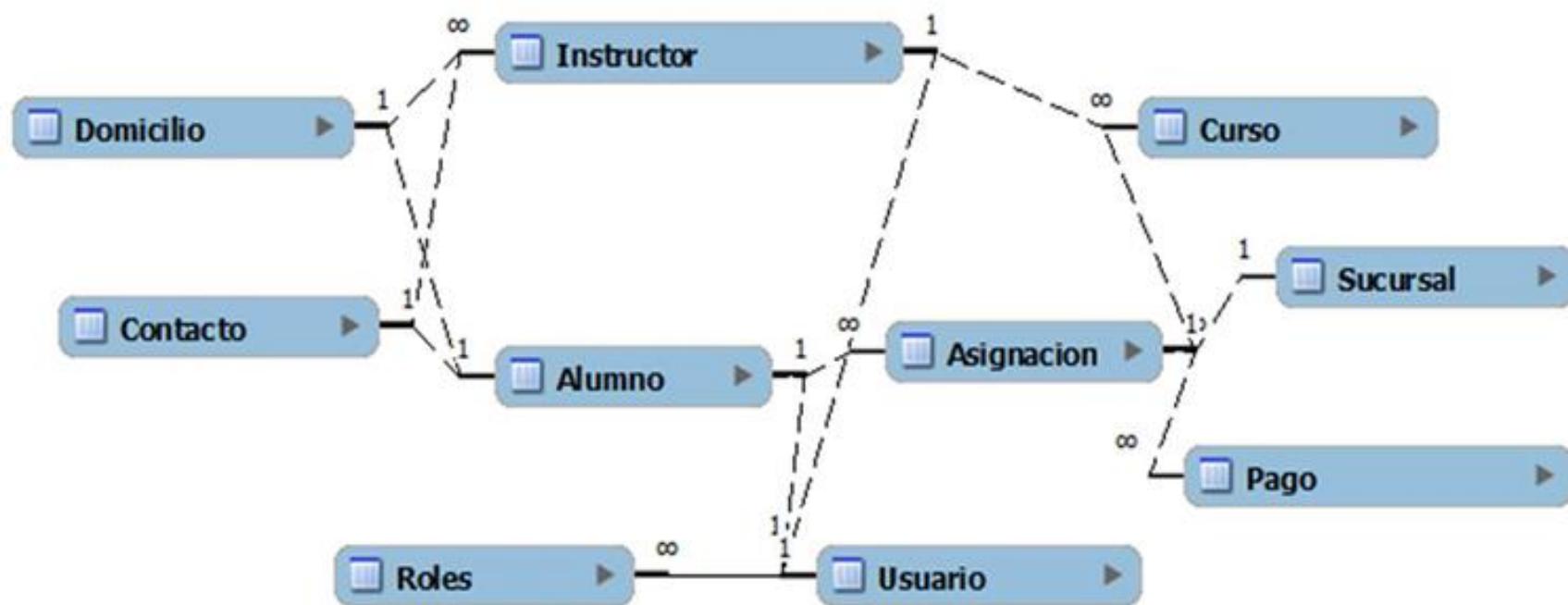
# 3. Java Persistence Api (JPA)

## Ejemplo Diagrama Entidad Relación



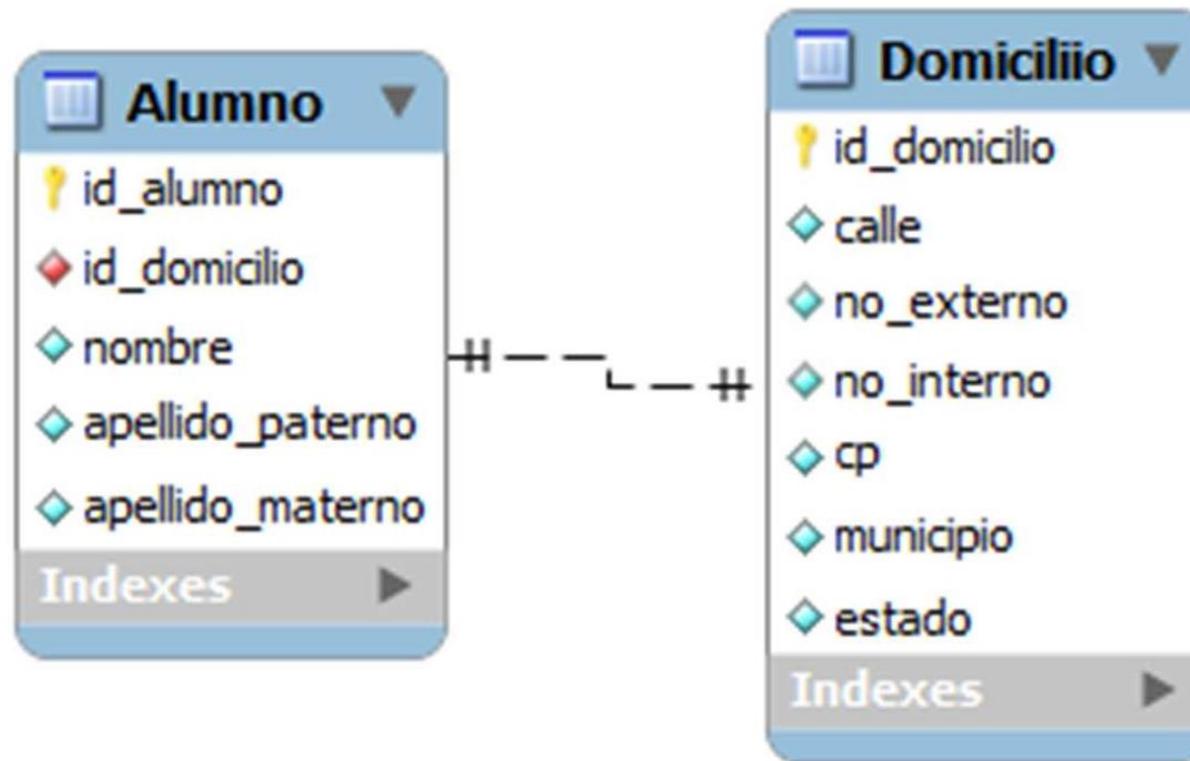
### 3. Java Persistence Api (JPA)

- A continuación realizaremos un análisis de cada relación descrita.



### 3. Java Persistence Api (JPA)

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



### 3. Java Persistence Api (JPA)

- 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 podamos 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 (Java Persistence query language) que recuperen objetos completos.

### 3. Java Persistence Api (JPA)

```
Alumno.java X
```

```
@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 X
```

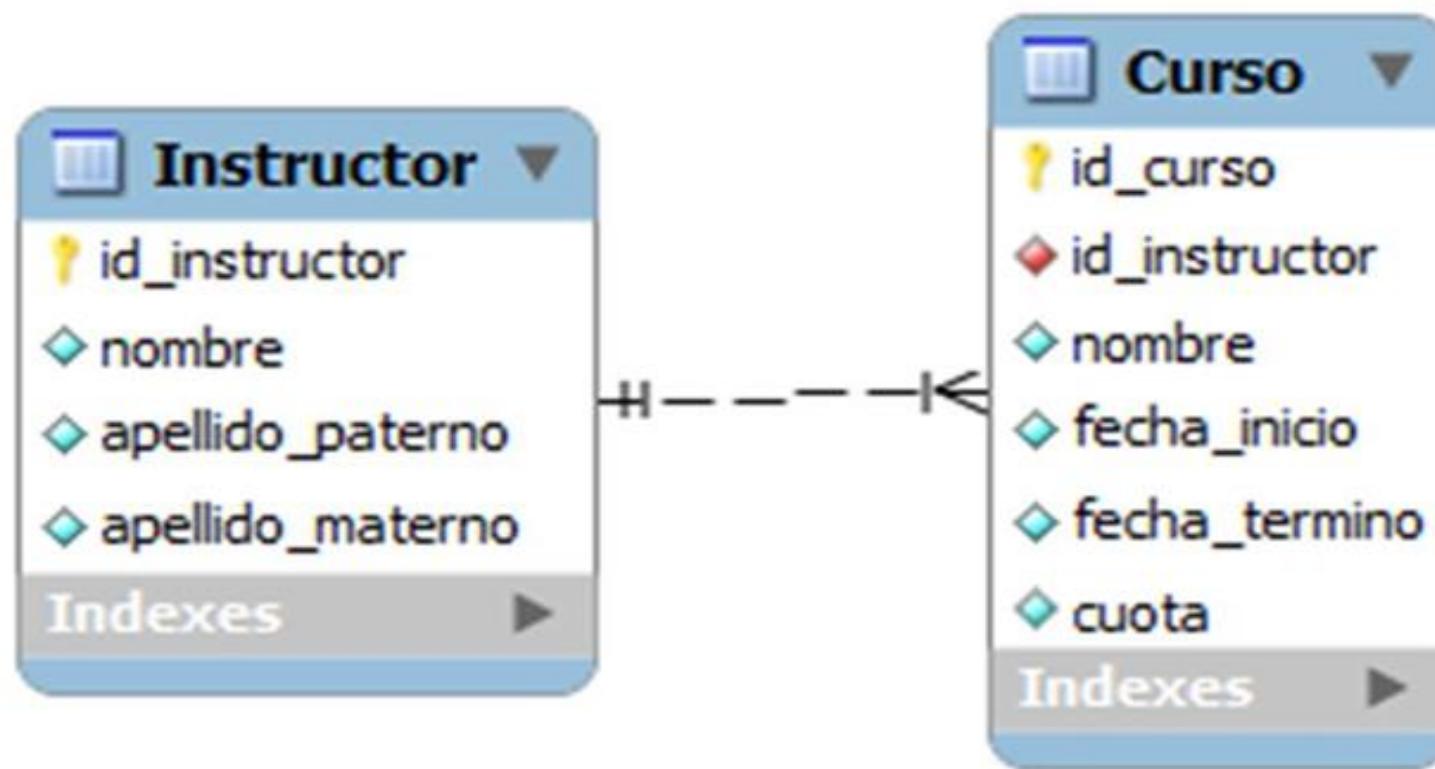
```
@Entity
public class Domicilio implements Serializable {
    private static final long serialVersionUID = 1L;

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

    //Mas atributos...
```

### 3. Java Persistence Api (JPA)

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



### 3. Java Persistence Api (JPA)

- 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**.
- 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.

### 3. Java Persistence Api (JPA)

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

The diagram illustrates two Java entity classes, `Instructor.java` and `Curso.java`, shown in an IDE interface. Both classes are annotated with `@Entity`. They each have an `@Id` field named `id_instructor` with `GenerationType.IDENTITY`. The `Instructor.java` class has a many-to-one association to `Curso` with the annotation `@OneToOne(mappedBy="instructor")` and a `Set<Curso> cursos` collection. The `Curso.java` class has a one-to-many association to `Instructor` with the annotation `@ManyToOne` and a `private Instructor instructor;` field. Red arrows highlight the `mappedBy` and `JoinColumn` annotations, indicating they are the focus of the explanation.

```
J Instructor.java X
@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
    @OneToOne(mappedBy="instructor")
    private Set<Curso> cursos;
}

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

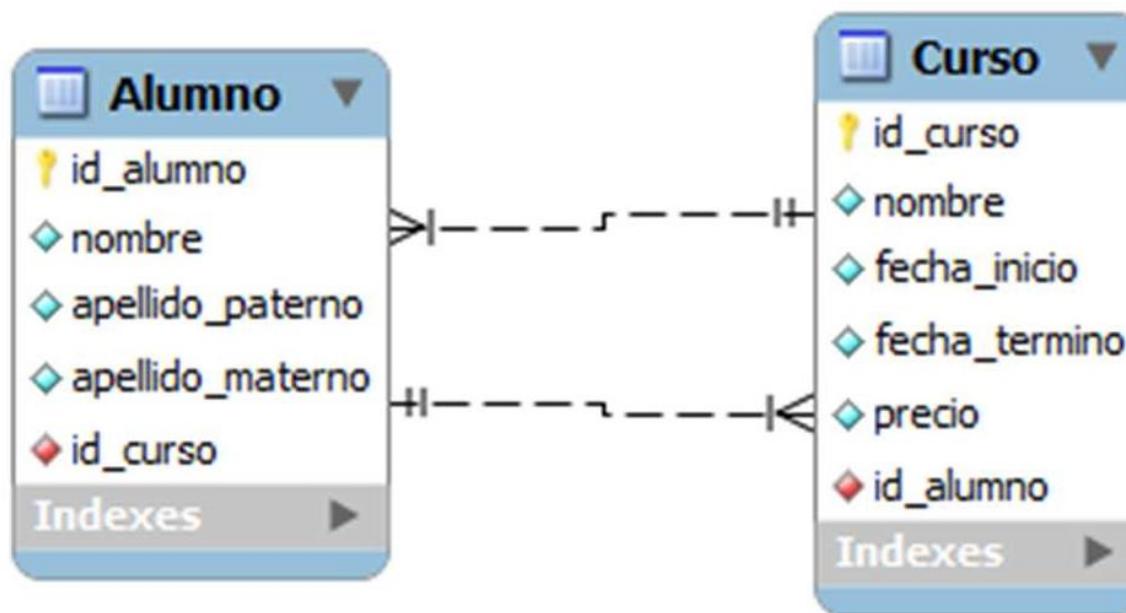
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    @Column(name="id_curso")
    private int idCurso;

    //Más atributos..

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

### 3. Java Persistence Api (JPA)

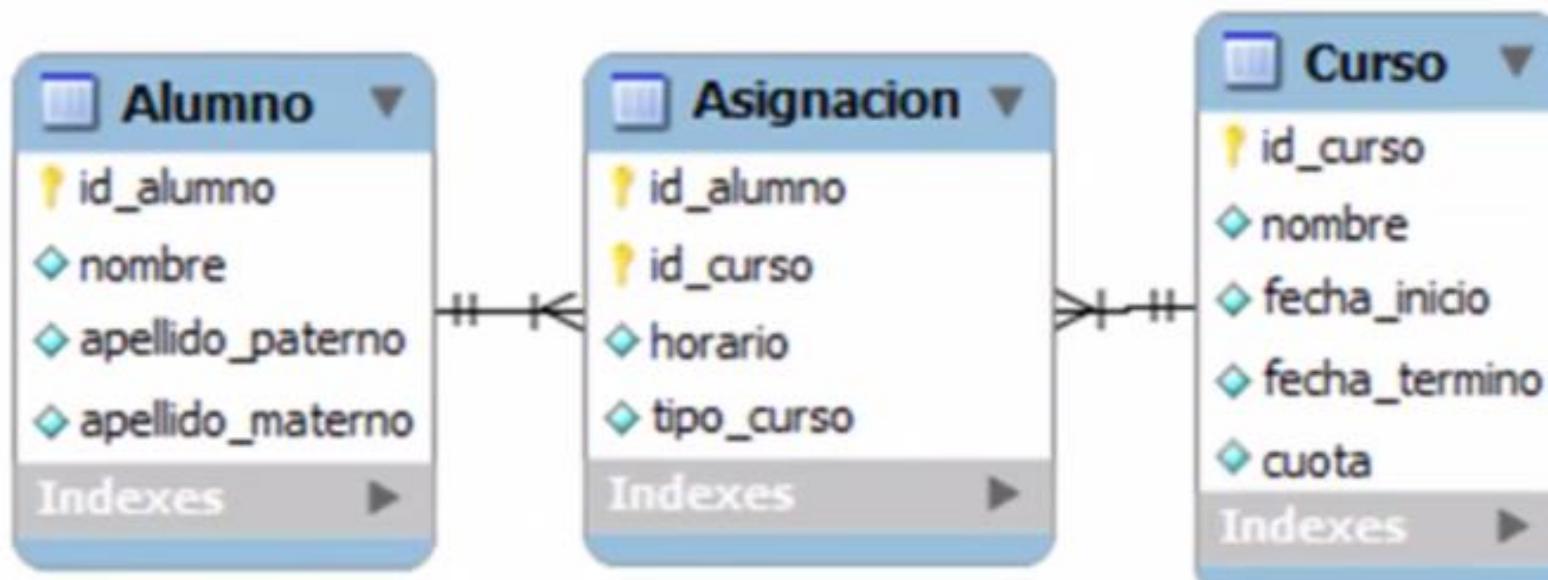
Ejemplo de Relación Muchos a Muchos  
(Un Alumno tiene Muchos Cursos y un Curso tiene  
Muchos Alumnos)



### 3. Java Persistence Api (JPA)

- 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.
- Aplicamos una normalización agregando una tabla transitiva llamada asignación y así convertir la relación de 1 a Muchos.

### 3. Java Persistence Api (JPA)



### 3. Java Persistence Api (JPA)

```
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;
}
```

# 3. Java Persistence Api (JPA)

## 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;
```

### 3. Java Persistence Api (JPA)

- 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.

### 3. Java Persistence Api (JPA)

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

### 3. Java Persistence Api (JPA)

- 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):

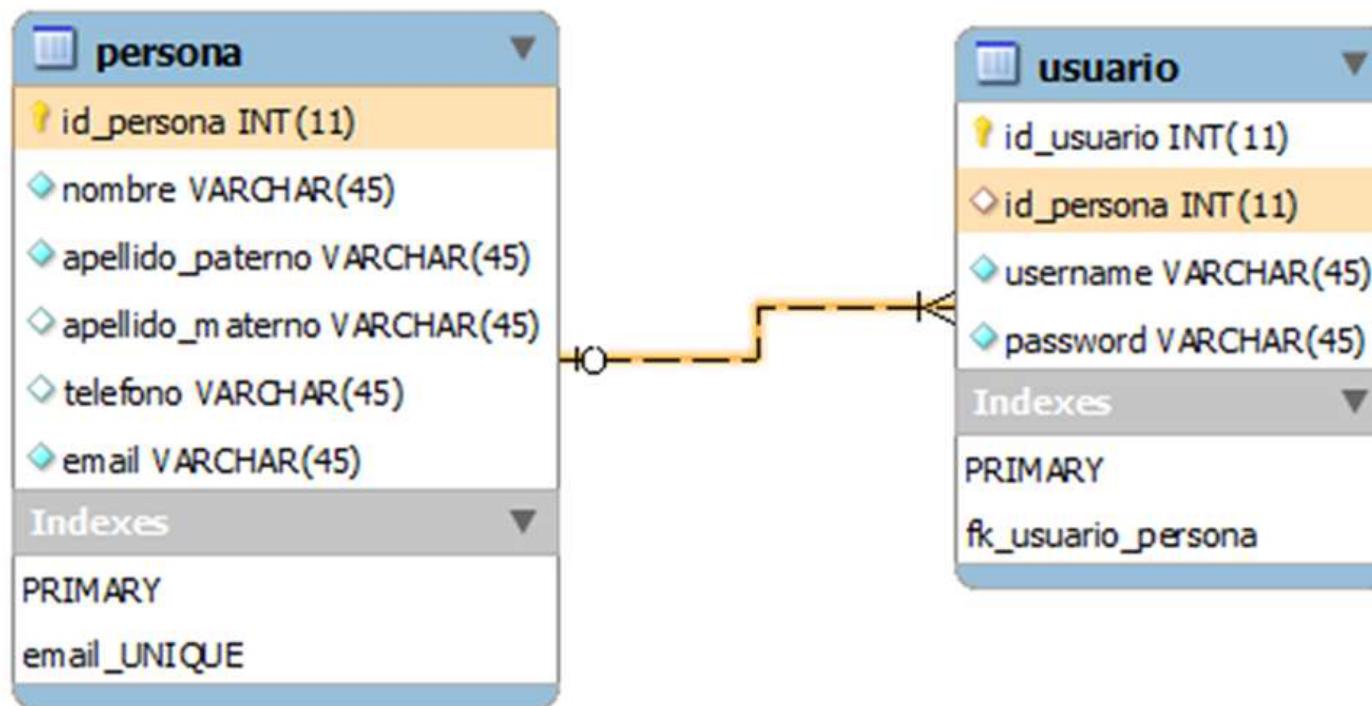
```
SELECT p FROM Persona p JOIN FETCH p.domicilio WHERE ...
```

### 3. Java Persistence Api (JPA)

- 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.

# 3. Java Persistence Api (JPA)

## Guardado en Cascada



### 3. Java Persistence Api (JPA)

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

```
@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;
```

- Tenemos los siguientes valores
  - all, persist, merge, remove, refresh
- En todos estos casos es posible aplicar el concepto de persistencia en cascada.

# 3. Java Persistence Api (JPA)

## 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

### 3. Java Persistence Api (JPA)

- 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**

### 3. Java Persistence Api (JPA)

- 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.

# 3. Java Persistence Api (JPA)

## 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 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`

# 3. Java Persistence Api (JPA)

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 p

```
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, sin importar mayúscula/minúsculas:  
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`

# 3. Java Persistence Api (JPA)

## 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();
}
```

### 3. Java Persistence Api (JPA)

- 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();
```

### 3. Java Persistence Api (JPA)

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();
```

# 3. Java Persistence Api (JPA)

## 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

### 3. Java Persistence Api (JPA)

- La construcción de queries dinámicas conlleva mucha concatenación de cadenas si estamos utilizando JDBC directo, incluso en queries complejas 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.

### 3. Java Persistence Api (JPA)

- 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.
- 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).

### 3. Java Persistence Api (JPA)

- 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 varias de estas queries utilizando las APIs mencionadas.

# 3. Java Persistence Api (JPA)

## 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);

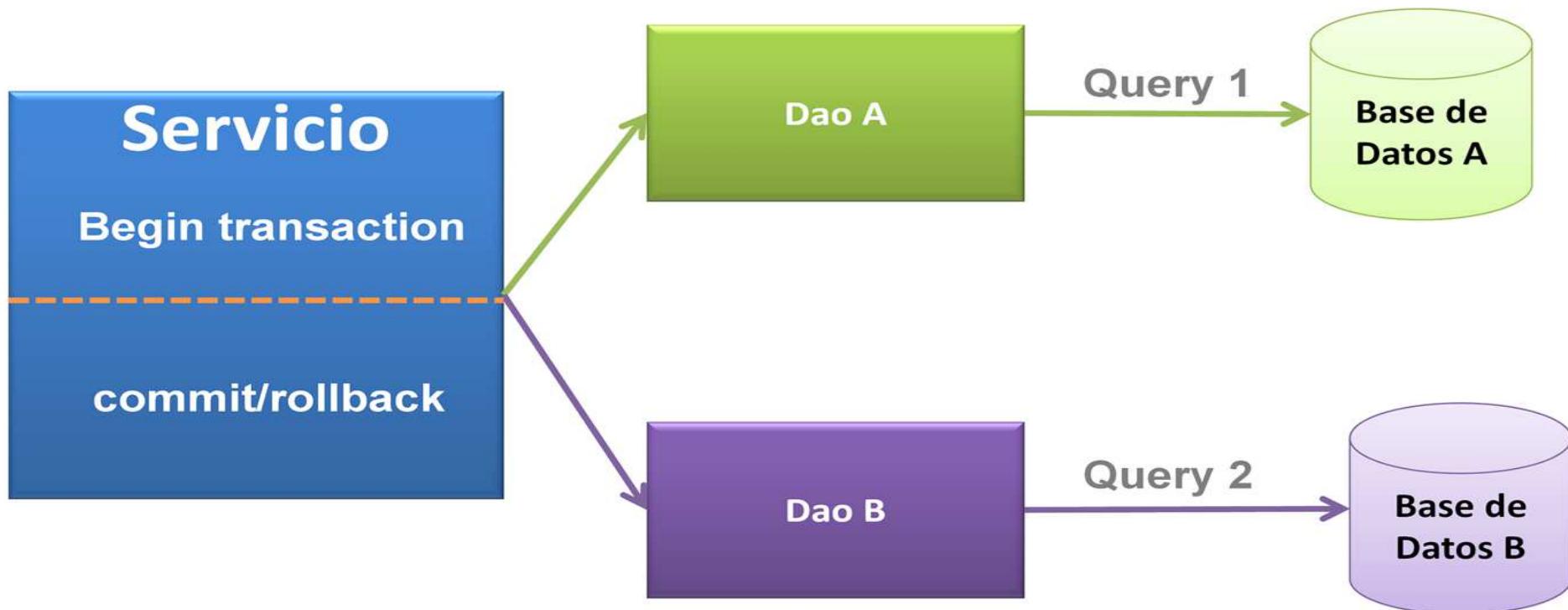
//5. Creamos el query typeSafe
TypedQuery<Persona> query = em.createQuery( criteriaQuery);

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

# 3. Java Persistence Api (JPA)

## ¿Qué es una Transacción?

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



### 3. Java Persistence Api (JPA)

- 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.
- 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.

## 3. Java Persistence Api (JPA)

- 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 fallos en la ejecución de nuestro código Java.

### 3. Java Persistence Api (JPA)

- 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**.
  - **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**.

### 3. Java Persistence Api (JPA)

- **Aislado:** (Isolated) 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.