



DIPLOMADO
**Desarrollo de sistemas con
tecnología Java**

Módulo 04
Persistencia con Hibernate

Mtro. Alfonso Gregorio Rivero Duarte



Acuerdos

- ★ Recesos de 10 minutos.
- ★ Preguntar en cualquier momento.
- ★ La participación en clase es muy importante.

Consideraciones

- ★ El moderador será el encargado de las sesiones de preguntas.
- ★ Hacer uso del ícono de  levantada para pedir la palabra.
- ★ Si por alguna circunstancia entra tarde a la sesión, no activar el micrófono para no interrumpir la sesión.
- ★ Se prohíbe la violencia y la agresión verbal en contra de cualquier persona.



¿Quién soy yo?

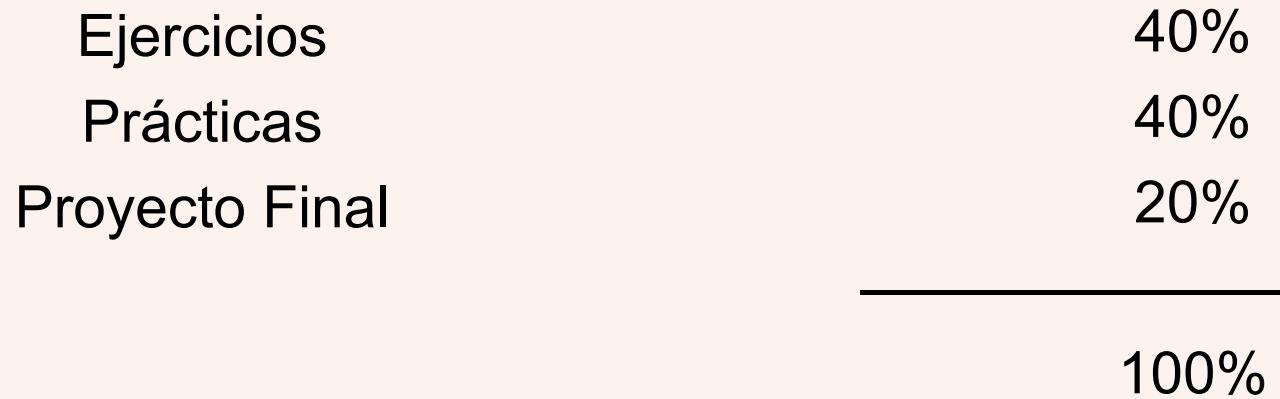
- ★ Experiencia profesional
- ★ Certificaciones
- ★ Academia
- ★ Hobbies
- ★ ¿Por qué elegí aportar con el diplomado?
- ★ Otros...



Módulos del Diplomado

Módulo	Duración (hrs)
Programación orientada a objetos con Java	40
Manejo de bases de datos con Java	20
Principios y patrones de diseño	20
Persistencia con Hibernate	20
JavaServer Faces	25
Enterprise Java beans y Servicios web	15
Introducción al desarrollo de aplicaciones empresariales con Spring Boot Framework	20
Persistencia con Spring Data	20
Desarrollo de aplicaciones Web con Spring Boot MVC	20
API RESTful con Spring Boot	20
Spring Security	20
Total	240

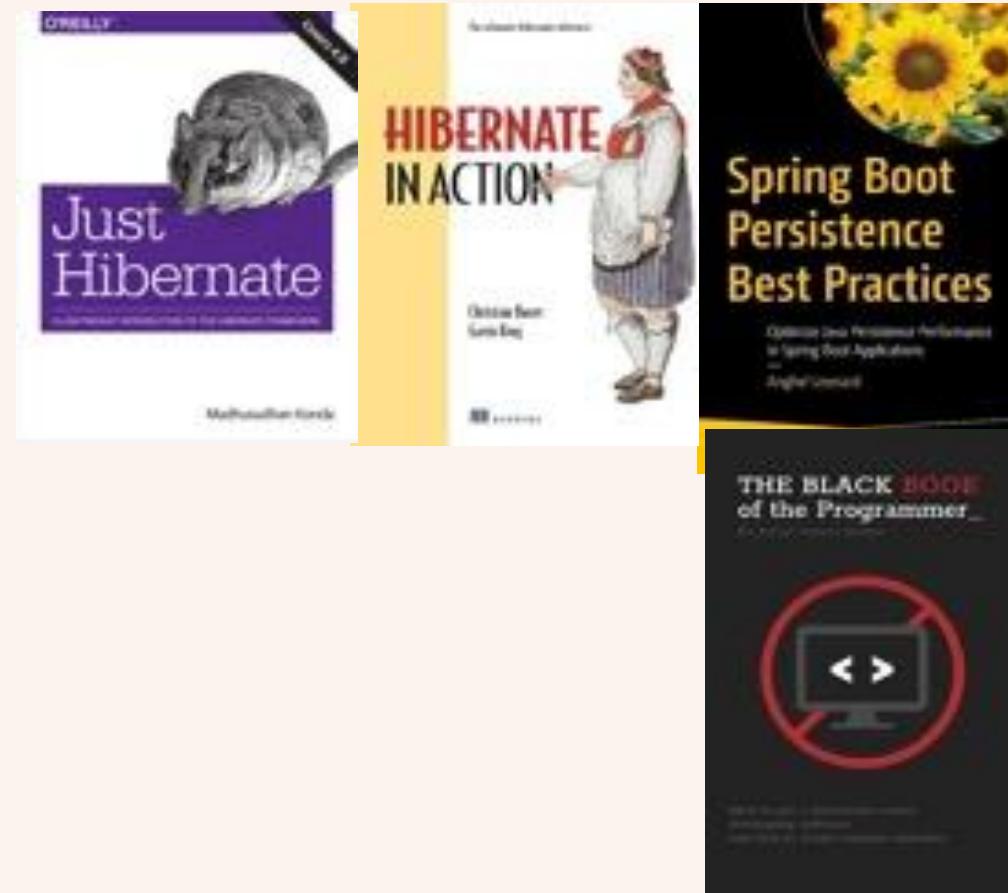
Evaluación



Todos los ejercicios y programas deben de subirse al apartado especial que se encuentra en Moodle.



Bibliografía



Temario

1. Introducción

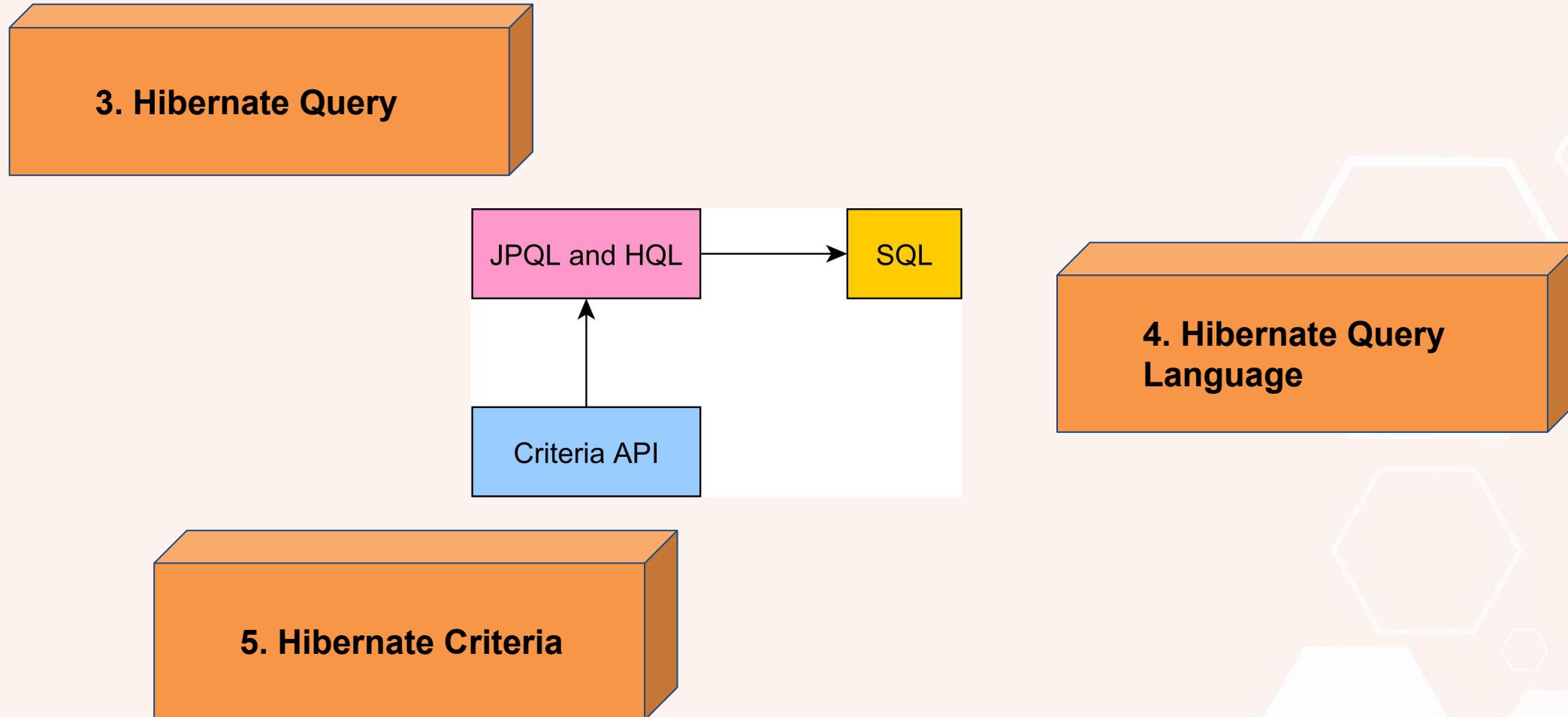
- Relaciones POO
- Persistencia de Datos
- ORM (Object Relational Mapping)



2. Hibernate

- Configuración
- Mapeo de Clases Persistentes
- Tipos de Datos
- Mapeo de Colecciones
- Transacciones
- Tipo de relaciones
- Claves primarias

Temario

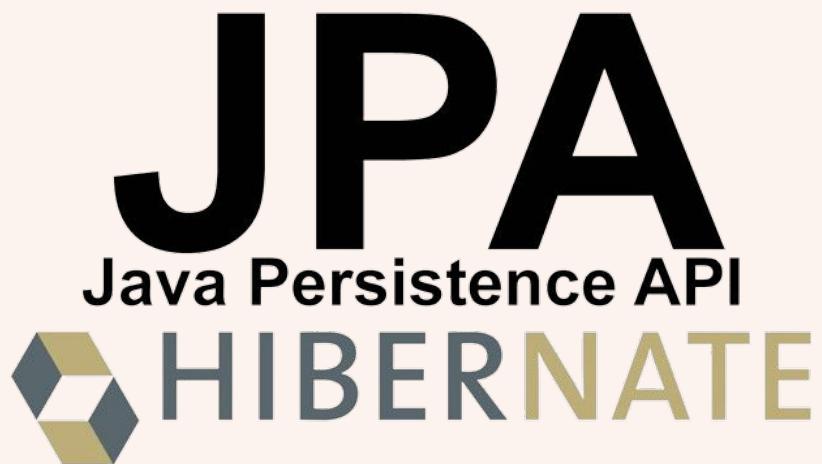


Objetivos Generales del Módulo de Hibernate

- Aplicar los principios de Hibernate y su uso en el desarrollo de arquitecturas empresariales.
- Enlistarse a configurar un proyecto con Hibernate desde cero.
- Identificar la importancia de la utilización del framework Hibernate en el Desarrollo de un proyecto en Java.
- Adquirir nuevos conceptos, habilidades y práctica para la creación de código empresarial.

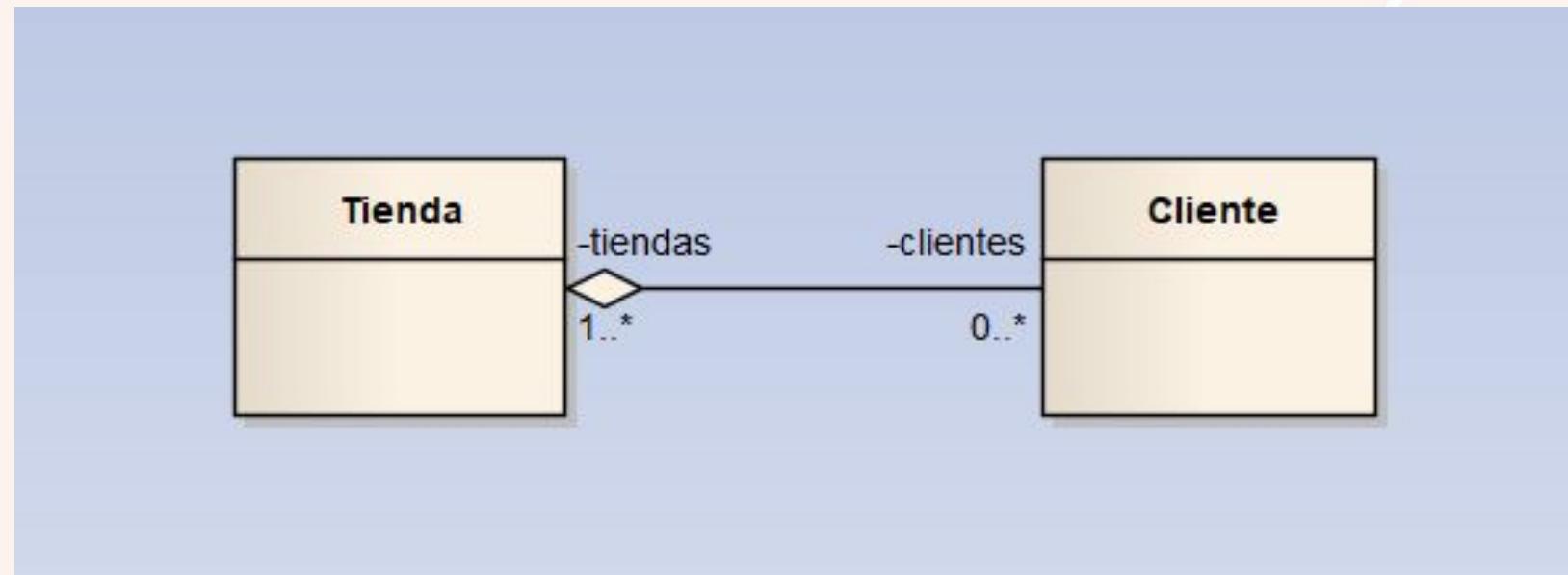
1. Introducción

1.1 Relaciones en POO



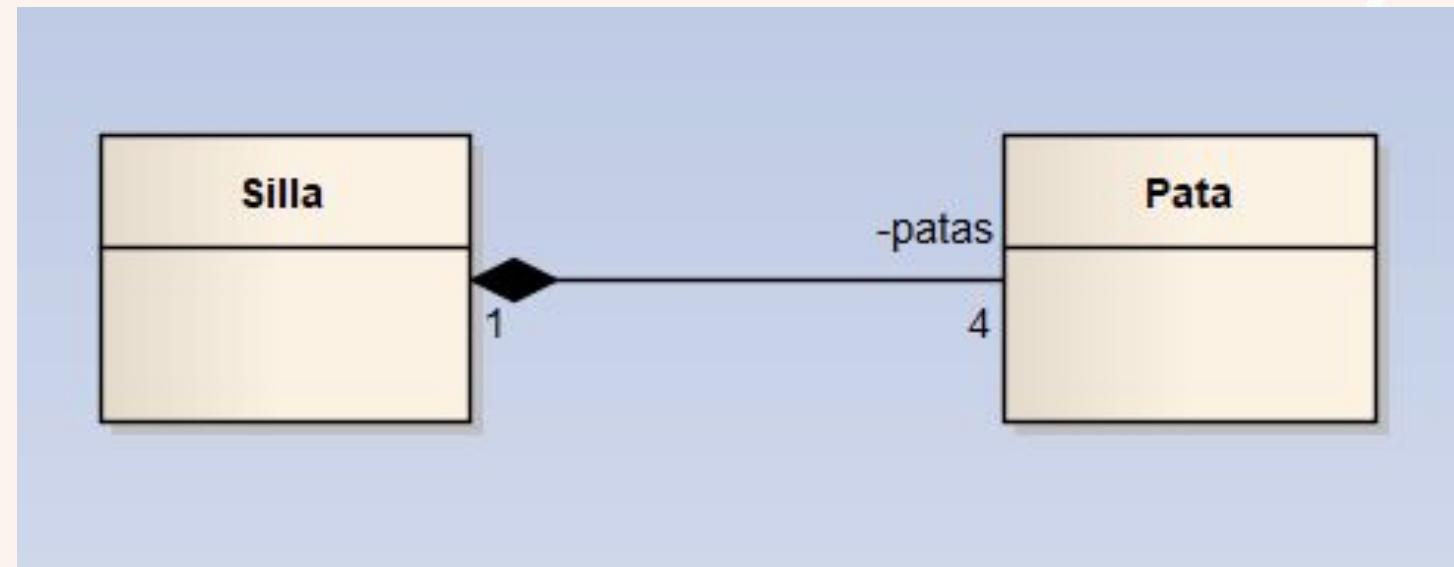
Agregación

La agregación es un tipo de relación que indica que un objeto forma parte o le pertenece a otro objeto, es prácticamente una asociación, pero se diferencian por la notación que se utiliza en UML y su funcionalidad dentro del código.



Composición

La composición es una relación como la agregación, pero más fuerte, es decir, un objeto no puede ser ese objeto sin otros objetos, por ejemplo: una silla no puede ser silla sin sus patas, un automóvil no puede ser automóvil sin sus ruedas o su motor, básicamente todos dependen de entre sí.

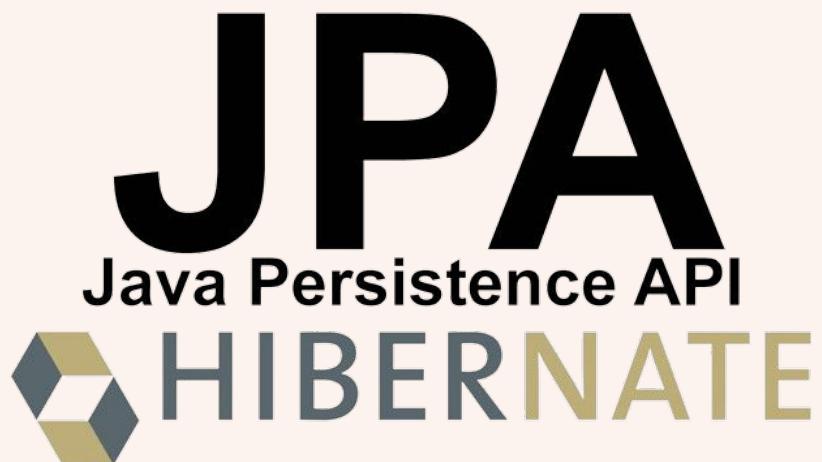


Composición

La forma de implementar esto es un poco diferente, ya que, a diferencia de las demás relaciones, en esta no queremos que, si el objeto Silla es destruido, sus patas anden vagando por la memoria, por decirse así; entonces, el objetivo es que la destrucción se lleve consigo a todas las patas que componen a la silla, y para lograr esto, debemos crear el objeto dentro de la clase Silla, siendo nuestra función de agregarPata una compuesta por parámetros del constructor de Pata.

1. Introducción

1.2 Persistencia de Datos

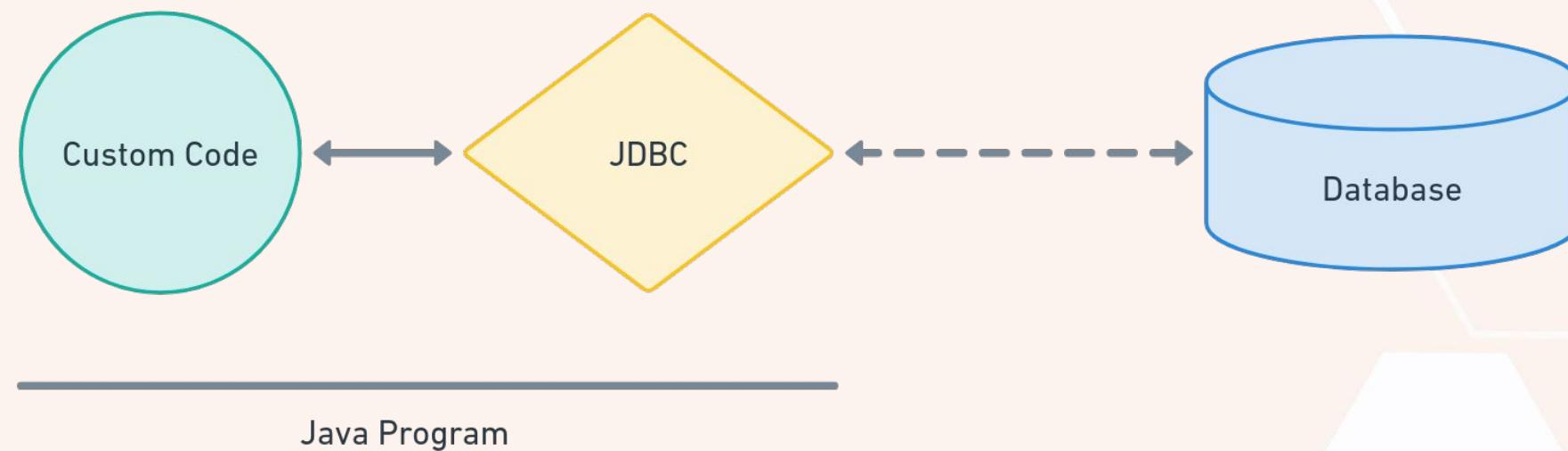


JDBC

API del lenguaje JAVA que permite la ejecución de operaciones sobre bases de datos.

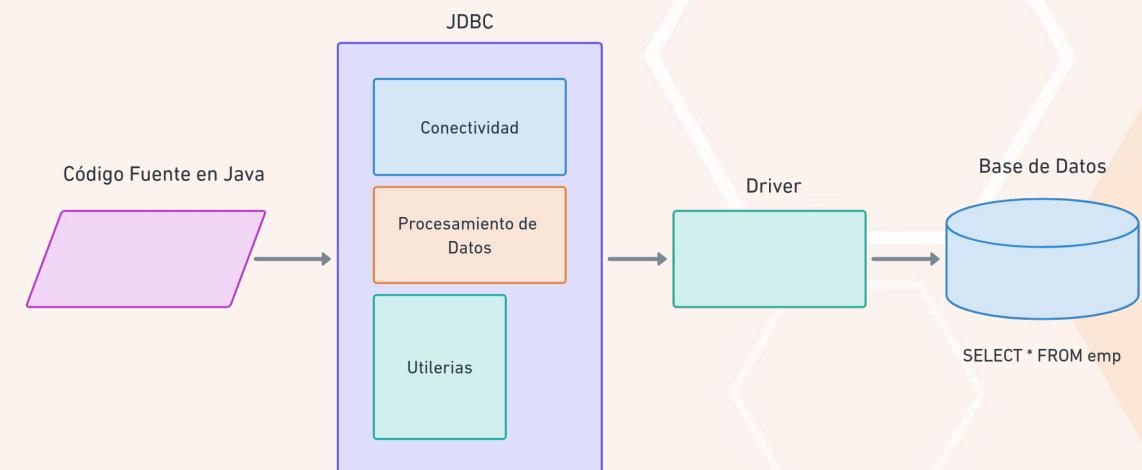
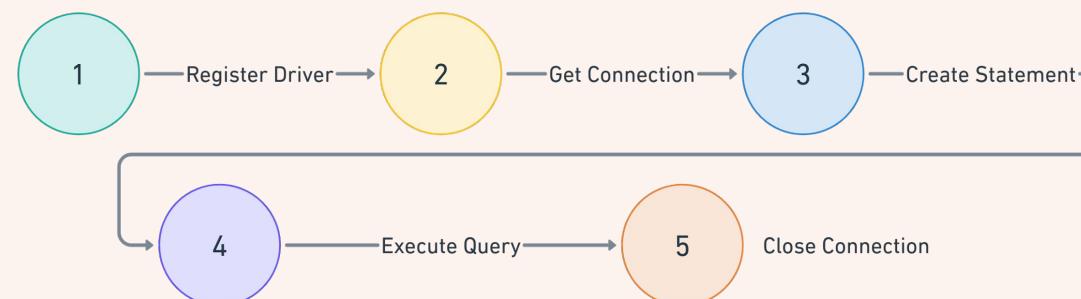
Independiente del sistema operativo y de la base de datos.

Utiliza el dialecto SQL del modelo de base de datos.



Conexión JDBC

La manera tradicional de acceder sería a través de JDBC directamente conectado a la BD mediante ejecución de sentencias SQL.



Desventaja

Esta primer aproximación puede ser útil para proyectos o arquitecturas con pocas clases de negocio, ya que el mantenimiento del código está altamente ligado a los cambios en el modelo de datos relacional de la BD.

Un mínimo cambio implica la revisión de casi todo el código así como de su compilación y nueva instalación en el cliente.

El Problema

Trabajar con software orientado a objetos y bases de datos relacionales puede ser incómodo y podría quitar demasiado tiempo.

Los costos de desarrollo son significativamente más altos debido a un desajuste de paradigma entre cómo se representan los datos en objetos y bases de datos relacionales.

Por ejemplo, considere una entrada de la libreta de direcciones que representa a una sola persona junto con cero o más números de teléfono y cero o más direcciones. Esto podría modelarse en una implementación orientada a objetos mediante un "objeto Persona" con atributos/campos para contener cada elemento de datos que comprende la entrada: el nombre de la persona, una lista de números de teléfono y una lista de direcciones.

El Problema

Sin embargo, muchos productos de bases de datos populares, como el sistema de gestión de bases de datos SQL (DBMS), sólo pueden almacenar y manipular valores escalares, como números enteros y cadenas organizadas dentro de tablas.

El meollo del problema consiste en traducir la representación lógica de los objetos a una forma atomizada que sea capaz de almacenarse en la base de datos, preservando al mismo tiempo las propiedades de los objetos y sus relaciones para que puedan recargarse como objetos cuando sea necesario. Si se implementa esta funcionalidad de almacenamiento y recuperación, se dice que los objetos son persistentes.

Persistencia de Datos

La persistencia de datos es un medio mediante el cual una aplicación puede recuperar información desde un sistema de almacenamiento no volátil y hacer que esta persista.

La tarea de persistir objetos Java en una base de datos relacional actualmente está siendo facilitada por un gran número de herramientas que permiten a los desarrolladores dirigir motores de persistencia para convertir objetos Java a columnas/registros de una base de datos y viceversa. Esta tarea implica serializar objetos Java estructurados en forma de árbol a una base de datos relacional estructurada de forma tabular y viceversa. Esencial para este esfuerzo es la necesidad de mapear los objetos Java a columnas y registros de la base de datos de una manera optimizada en velocidad y eficiencia.

Problemática con el acceso a datos

Una de las grandes problemáticas al momento de acceder a los datos, es que la implementación y formato de la información puede variar según la fuente de los datos, además, implementar la lógica de acceso a datos en la capa de lógica de negocio puedes ser un gran problema, pues tendríamos que lidiar con la lógica de negocio en sí, más la implementación para acceder a los datos, adicional, si tenemos múltiples fuentes de datos o estas pueden variar, tendríamos que implementar las diferentes lógicas para acceder las diferentes fuentes de datos, como podrían ser: bases de datos relacionales, No SQL, XML, archivos planos, servicios web o REST, etc).

Un programador inexperto o con pocos conocimientos sobre arquitectura podría optar por crear en un solo método la lógica de negocio y la lógica de acceso a datos, creando un método muy difícil de entender y mantener.

Problemática con el acceso a datos

Observa que las únicas líneas que obedecen a la lógica de negocio son de la 10 a la 11, donde validamos que no esté repetido el usuario, y podríamos decir que la 20 donde guardamos el nuevo usuario, todo lo demás es solo para establecer la conexión a la base de datos, y eso que solo estamos consultando un registro e insertando otro, imagina si esta operación fuera una transacción más grande que involucrar varias validaciones y entidades; seguramente este método crecería exponencialmente de tamaño y la complejidad para comprenderlo y darle mantenimiento también crecería.

```
1 public class UserService {  
2     public void createUser(NewUserRequestDTO dto) {  
3         Connection connection =  
4             DriverManager.getConnection("connection url ...");  
5         Statement queryStm = connection.createStatement();  
6         PreparedStatement usernameStm = connection.prepareStatement(  
7             "select 1 from users u where u.username = ?");  
8         usernameStm.setString(1, dto.getUsername());  
9         ResultSet usernameResult = usernameStm.executeQuery();  
10        if(usernameResult.next()) {  
11            throw new RuntimeException("Username already exists");  
12        }  
13        connection.setAutoCommit(false);  
14        PreparedStatement stm = connection.prepareStatement(  
15            "insert into users(id,username,password) values (?,?,?)");  
16        stm.setLong(1, dto.getId());  
17        stm.setString(2, dto.getUsername());  
18        stm.setString(3, dto.getPassword());  
19        stm.executeUpdate();  
20        connection.commit();  
21    }  
22 }
```

Modelo DAO

Dado lo anterior, el patrón Data Access Object (DAO) propone separar por completo la lógica de negocio de la lógica para acceder a los datos, de esta forma, el DAO proporcionará los métodos necesarios para insertar, actualizar, borrar y consultar la información; por otra parte, la capa de negocio solo se preocupa por lógica de negocio y utiliza el DAO para interactuar con la fuente de datos.

Un error común al implementar este patrón es no utilizar Entidades y en su lugar, regresar los objetos que regresan las mismas API's de las fuentes de datos, ya que esto obliga al BusinessObject tener una dependencia con estas librerías, además, si la fuente de datos cambia, también cambiarán los tipos de datos, lo que provocaría una afectación directa al BusinessObject.

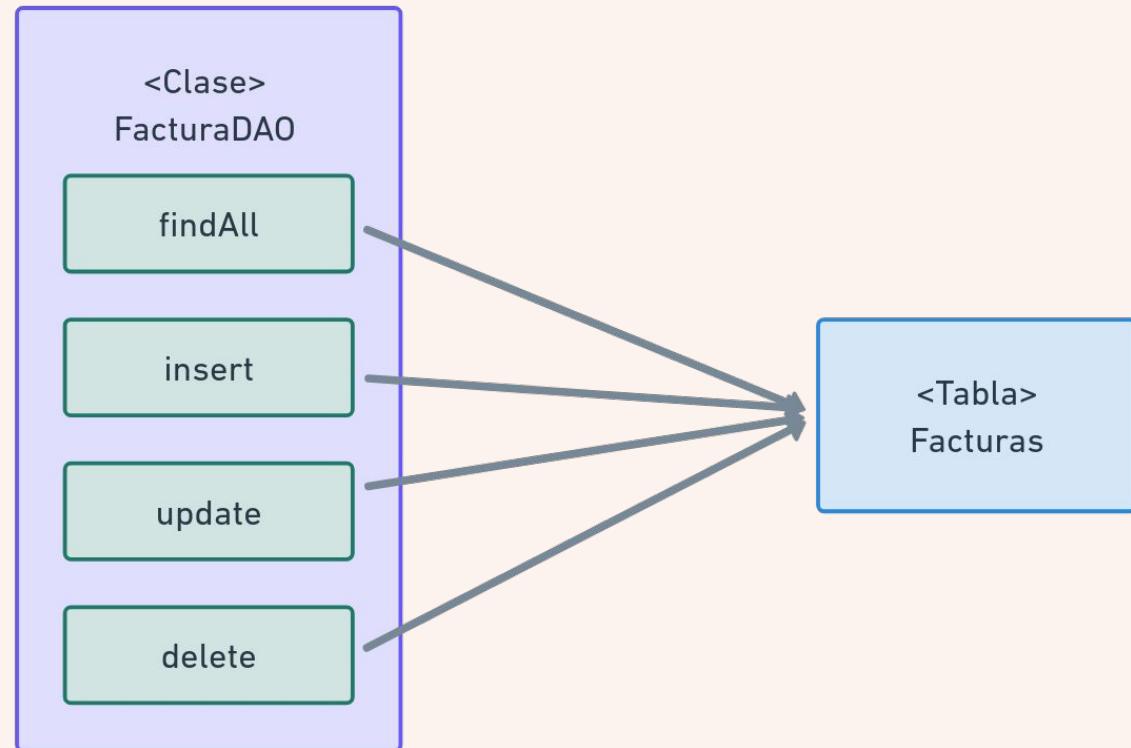
Modelo DAO

Si aplicamos el nuevo conocimiento adquirido para actualizar nuestro servicio de creación de usuario, podemos ver una clara diferencia en código:

Observa el cambio significativo que hemos logrado utilizando un DAO, podemos ver que en lugar de preocuparnos por cómo crear la conexión a la base de datos y cómo consultar y guardar los datos, solo nos centramos en las reglas de negocio, por lo tanto, el código se hace mucho más fácil de entender y mantener.

```
1 public void createUser(NewUserRequestDTO dto) {  
2     UserDAO userDAO = new UserDAOImpl();  
3     if(userDAO.findByUsername(dto.getUsername()) != null) {  
4         throw new RuntimeException("Username already exists");  
5     }  
6     UserConverter usrConverter = new UserConverter();  
7     User user = surConverter.fromDto(dto);  
8     usrDAO.save(user);  
9 }
```

Modelo DAO



<https://www.oscarblancarteblog.com/2018/12/10/data-access-object-dao-pattern/>

Si solo tenemos una fuente de datos esta podría ser la solución definitiva, sin embargo, seguimos teniendo un problema, y es como sabemos exactamente qué instancia del DAO instanciar, por ejemplo, en la línea 3 estamos creado una instancia de UserDAOImpl en código duro, por lo que si queremos cambiar de fuente de datos tendríamos que cambiar el código y con ello, un nuevo despliegue forzado, por ello, es inteligente utilizar el DAO en conjunto con el patrón Abstract Factory para que nos ayude a determinar y crear la instancia correcta del DAO según nuestra configuración.

Desventaja

Los problemas de esta implementación siguen siendo el mantenimiento de la misma así como su portabilidad. Lo único que podemos decir es que tenemos el código de transacciones encapsulado en las clases DAO.

Problemas Generales

Mapeo de objetos a bases de datos relacionales y viceversa.

Asociaciones, herencia y polimorfismo

Lo Deseado

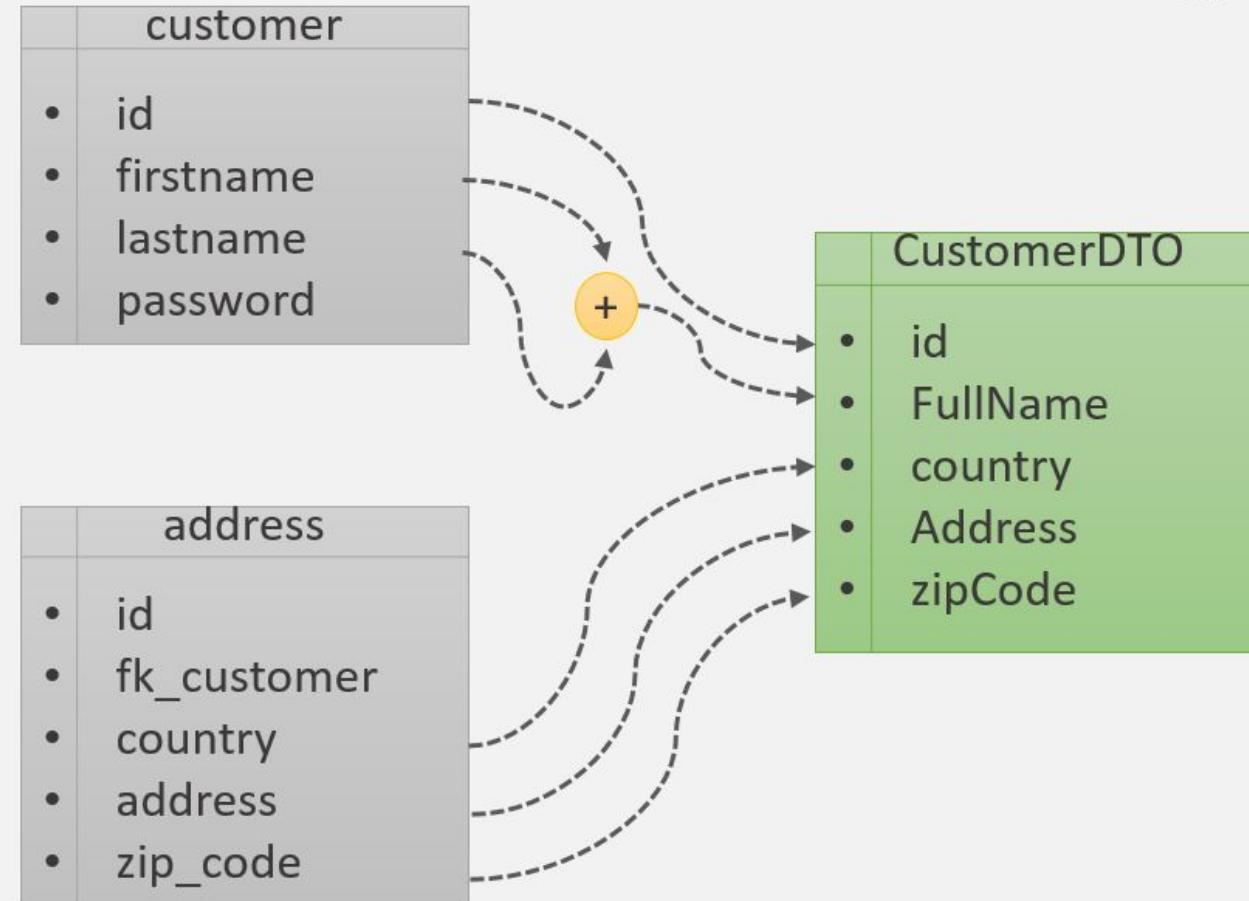
Lo que parece claro es que debemos separar el código de nuestras clases de negocio de la realización de nuestras sentencias SQL contra la Base de Datos.

Modelo DTO

Una de las problemáticas más comunes cuando desarrollamos aplicaciones, es diseñar la forma en que la información debe viajar desde la capa de servicios a las aplicaciones o capa de presentación, ya que muchas veces por desconocimiento o pereza, utilizamos las clases de entidades para retornar los datos, lo que ocasiona que retornemos más datos de los necesarios o incluso, tengamos que ir en más de una ocasión a la capa de servicios para recuperar los datos requeridos.

El patrón DTO tiene como finalidad de crear un objeto plano (POJO) con una serie de atributos que puedan ser enviados o recuperados del servidor en una sola invocación, de tal forma que un DTO puede contener información de múltiples fuentes o tablas y concentrarlas en una única clase simple.

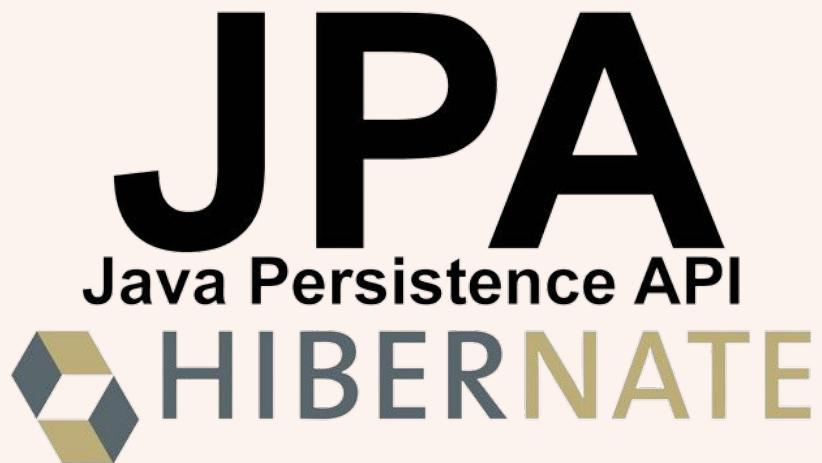
Modelo DTO



[https://www.oscarblancarteblog.com/
2018/11/30/data-transfer-object-dto-
patron-diseno/](https://www.oscarblancarteblog.com/2018/11/30/data-transfer-object-dto-patron-diseno/)

1. Introducción

1.3 ORM (Object Relational Mapping)



ORM Object Relational Mapping

Utiliza un motor de persistencia.

Hibernate es una solución ORM para ambientes Java.

El término Object Relational Mapping refiere a la técnica de mapear datos entre representación de objetos modelo hacia una representación modelo relacional.

Hibernate se encarga de mapear las clases en Java a tablas en base de datos, y de tipos de dato Java a tipos de dato SQL.

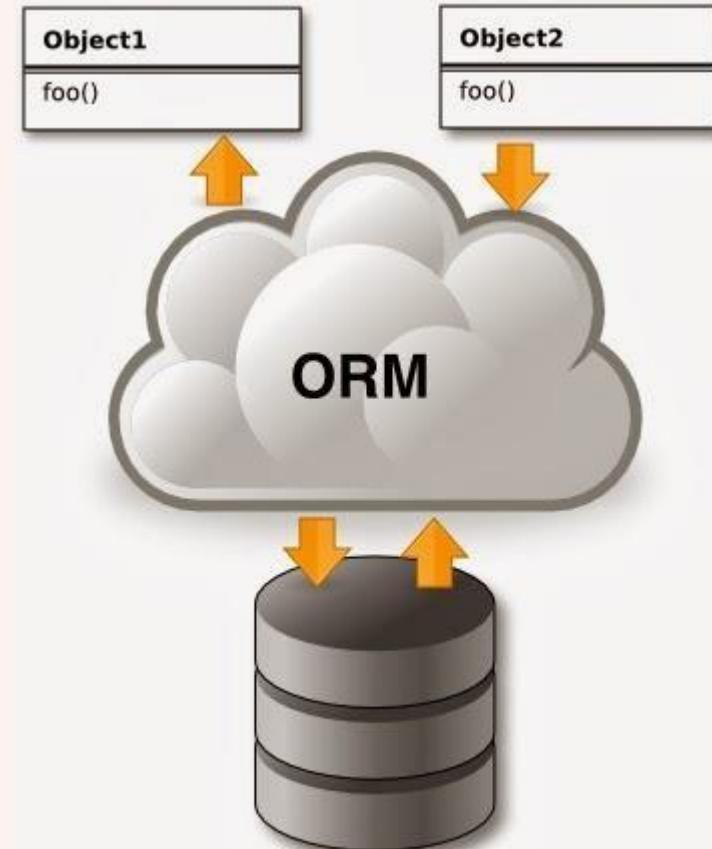
Objetivos de un ORM

Garantizar la persistencia de objetos.

- Conexiones JDBC + consultas SQL

Eliminar problemas

- Objetos con muchas propiedades
- Claves foráneas
- Restricciones de integridad



¿Qué ofrece un ORM?

Definir el mapeo en un solo punto.

Persistencia directa de objetos.

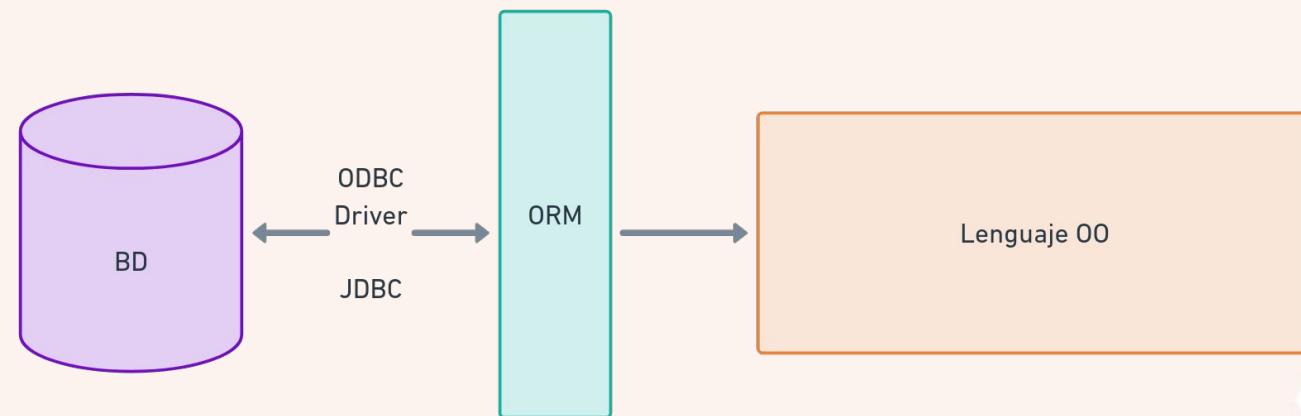
Carga automática de objetos.

Buen lenguaje de consultas (HQL, JPQL)

```
Query q = session.createQuery("from Item")
    .setReadOnly(true);

Criteria criteria = session.createCriteria(Item.class)
    .setReadOnly(true);

Query q = em.createQuery("select i from Item i")
    .setHint("org.hibernate.readOnly", true);
```



¿QUÉ ES UN ORM?

SQL Para **hacer consultas** a una base de datos un programador **necesita escribir SQL**.



ORM Un **ORM abstrae la base de datos** para que el programador haga consultas en el **lenguaje en el que está programando**, sin necesitar **SQL**.

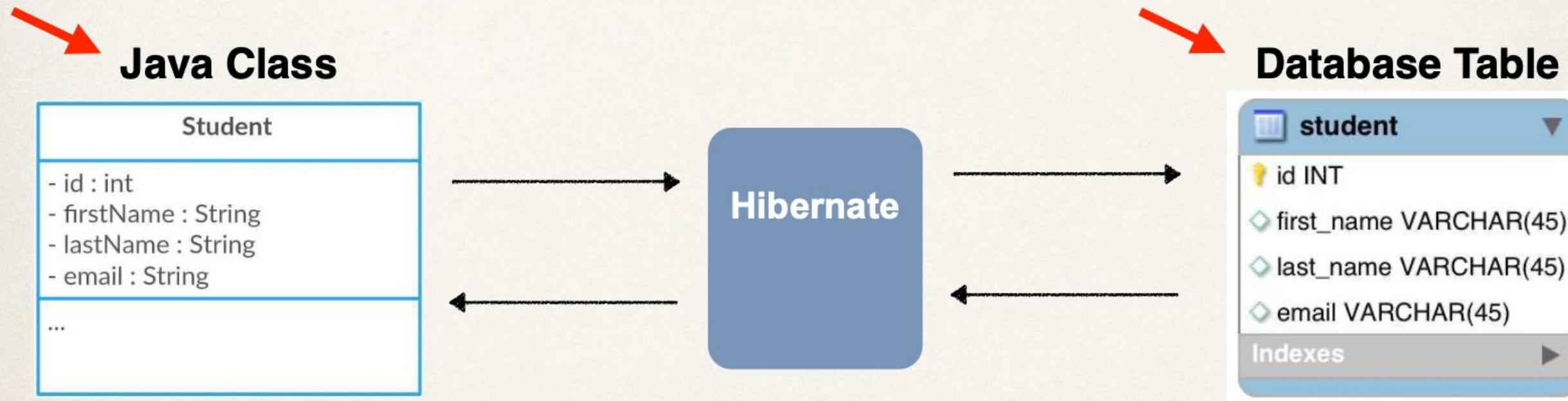


LOS ORM MÁS POPULARES SON:

Hibernate (Java)	←	JS TypeORM (JavaScript)
Entity Framework (C#)	←	SQL Alchemy (Python)
Eloquent (PHP)	←	Gorm (Go)

Aprende SQL para, diseñar, crear y administrar información en:  ed.team/cursos/sql

¡NUEVO CURSO!



Hibernate como API de Persistencia Java (JPA)

Hibernate ORM se preocupa por ayudar a que su aplicación logre persistencia. Entonces, ¿qué es la persistencia? La persistencia simplemente significa que nos gustaría que los datos de nuestra aplicación sobrevivan al proceso de utilización de datos.

Java Persistence API es una colección de clases y métodos para almacenar de forma persistente grandes cantidades de datos en una base de datos.

JPA es una interfaz que deben implementar los proveedores de persistencia. Hibernate es una de esas implementaciones de JPA. Cuando usas Hibernate con JPA, en realidad estás usando la implementación de Hibernate JPA.

Hibernate

Hibernate comenzó en el año 2001 por Gavin King con un equipo de Cirrus Technologies, como una alternativa al estilo de programación de EJB2, que usa beans como entidades. El objetivo original era ofrecer mejores capacidades de persistencia que las ofrecidas por EJB2, simplificar las complejidades, y complementar algunas características necesarias.



¿Qué ofrece Hibernate?

Es una capa de persistencia objeto-relacional y generador de sentencias SQL.

Facilita el mapeo entre BD relacional y modelo de objetos.

Funciona mediante archivos declarativos (xml) y anotaciones.

Lenguaje de consulta de datos HQL (Hibernate Query Language)

¿Por qué Hibernate?

Es Free Software+

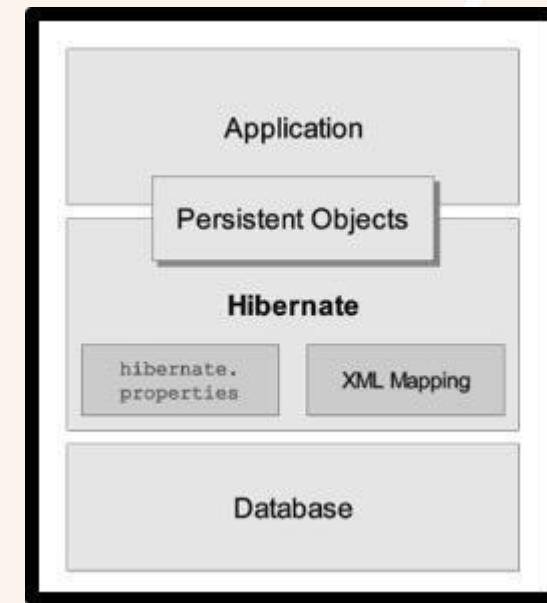
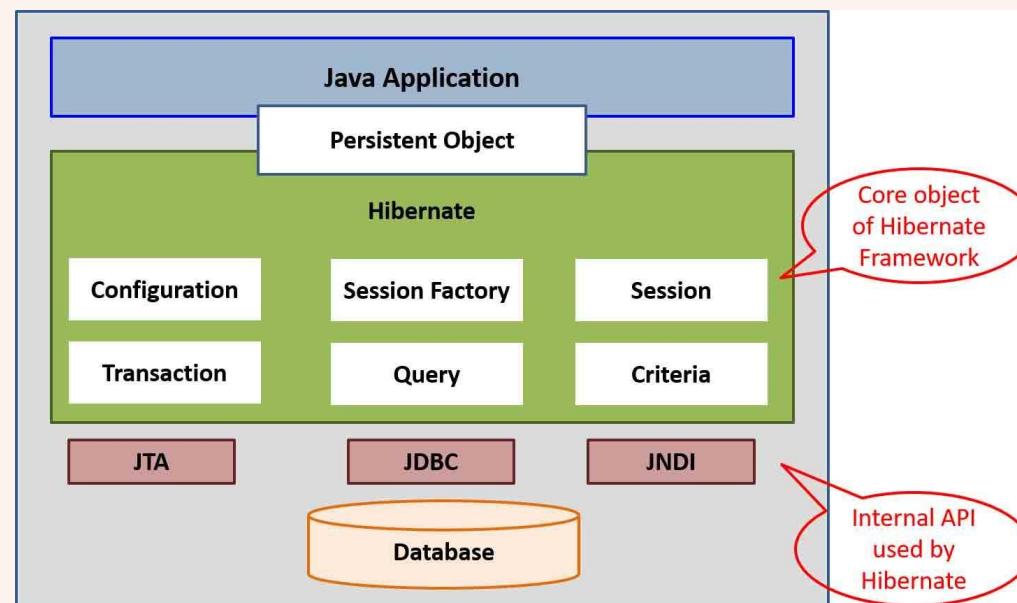
Buena documentación y estabilidad

Utiliza clases de forma directa

No depende de ningún manejador de base de datos.

Arquitectura

Hibernate se integra en cualquier tipo de aplicación justo por encima del contenedor de datos



Especificación JPA

JPA estandariza la importante tarea del mapeo relacional de objetos mediante el uso de anotaciones o XML para mapear objetos en una o más tablas de una base de datos. Para simplificar aún más el modelo de programación de persistencia

La API EntityManager puede persistir, actualizar, recuperar o eliminar objetos de una base de datos.

Especificación JPA

La API EntityManager y los metadatos de mapeo relacional de objetos manejan la mayoría de las operaciones de la base de datos sin necesidad de escribir código JDBC o SQL para mantener la persistencia.

JPA proporciona un lenguaje de consulta, que amplía el lenguaje de consulta independiente EJB (también conocido como JPQL), que puede utilizar para recuperar objetos sin escribir consultas SQL específicas de la base de datos con la que está trabajando.

Elementos JPA

Unidad de persistencia: define un modelo relacional de objetos completo que asigna clases Java (entidades + estructuras de soporte) con una base de datos relacional. EntityManagerFactory utiliza estos datos para crear un contexto de persistencia al que se puede acceder a través de EntityManager.

EntityManagerFactory: se utiliza para crear un EntityManager para interacciones de bases de datos. Los contenedores del servidor de aplicaciones normalmente proporcionan esta función, pero se requiere EntityManagerFactory si está utilizando la persistencia administrada por aplicaciones JPA. Una instancia de EntityManagerFactory representa un contexto de persistencia.

Elementos JPA

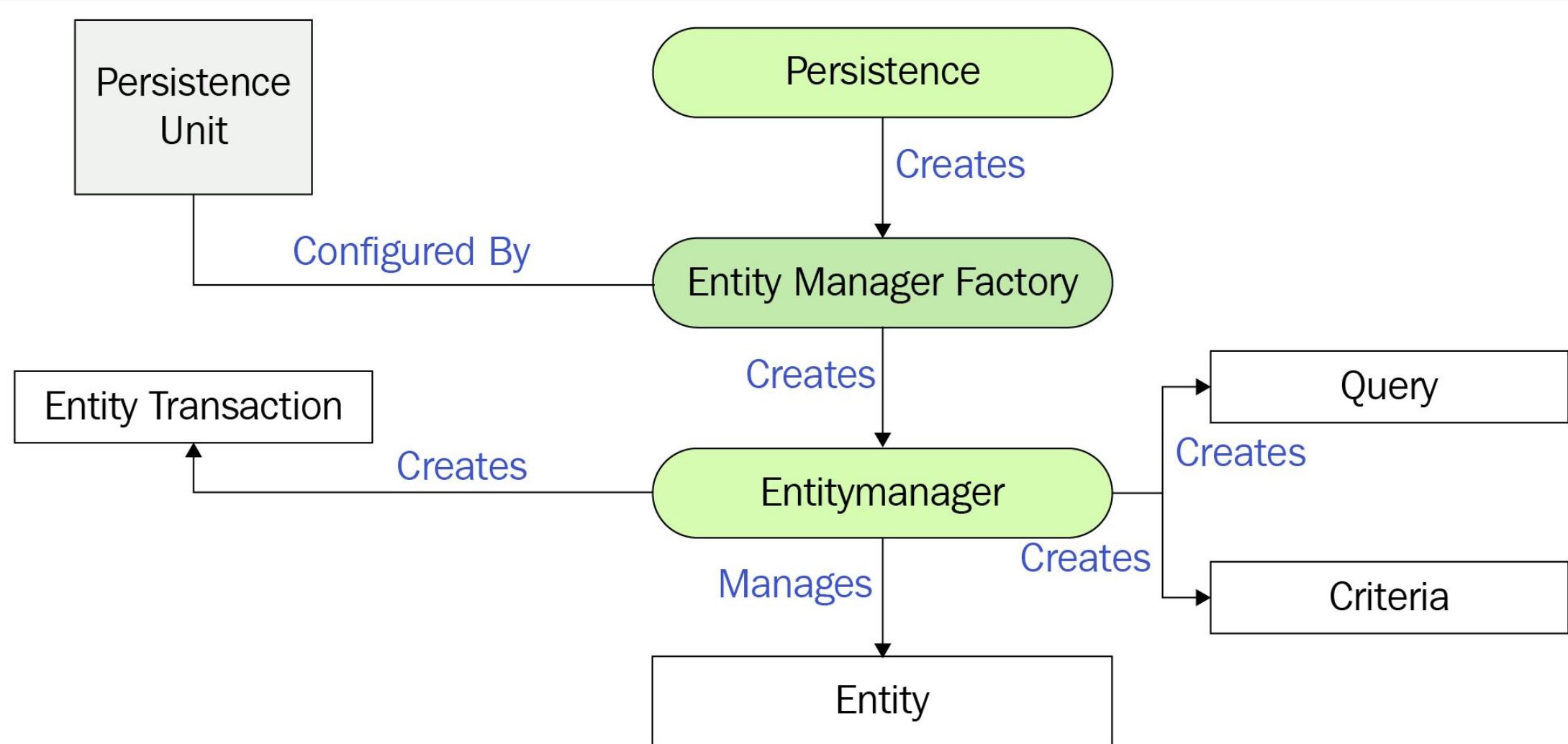
Contexto de persistencia: define el conjunto de instancias activas que la aplicación está manipulando actualmente. Puede crear el contexto de persistencia manualmente o mediante inyección.

EntityManager: el administrador de recursos que mantiene la colección activa de objetos de entidad que utiliza la aplicación. EntityManager maneja la interacción de la base de datos y los metadatos para asignaciones relacionales de objetos. Una instancia de un EntityManager mediante inyección en la aplicación o buscándola en el espacio de nombres del componente Java. Si la aplicación gestiona su persistencia, el EntityManager se obtiene de EntityManagerFactory.

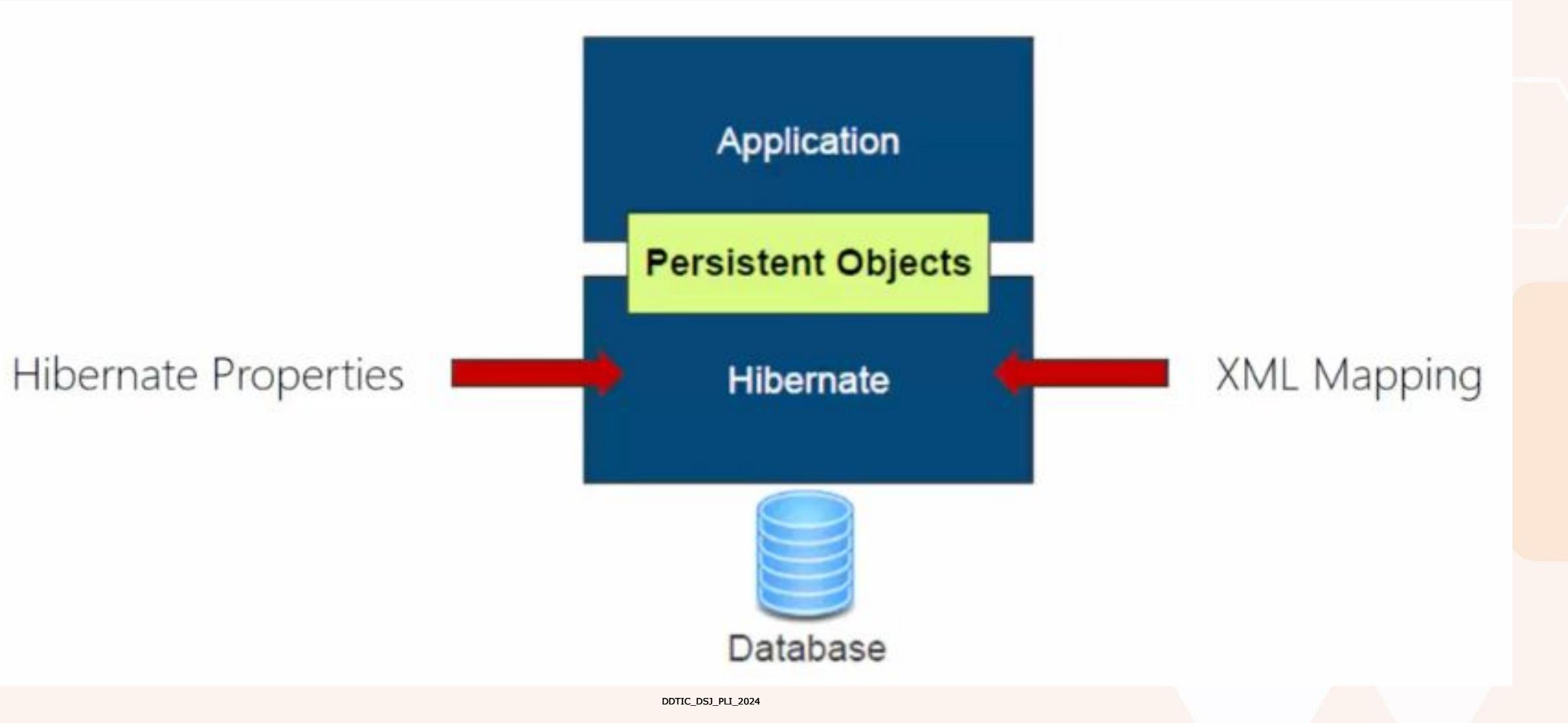
Elementos JPA

EntityObjects: una clase Java simple que representa una fila en una tabla de base de datos en su forma más simple. Los objetos de entidad pueden ser clases concretas o clases abstractas. Mantienen estados mediante el uso de propiedades o campos.

Elementos JPA



Elementos JPA



JPA

JPA (API de Persistencia en Java) es el estándar en Java que define una abstracción que nos permite realizar la integración entre el sistema orientado a objetos de Java y el sistema relacional de nuestra base de datos. Realiza por nosotros toda la conversión entre nuestros objetos y las tablas de una base de datos. Esta conversión se llama ORM (Mapeo Relacional de Objetos) y puede configurarse a través de metadatos (XML) o anotaciones.

JPA

Java Persistence API es una colección de clases y métodos para almacenar de forma persistente grandes cantidades de datos en una base de datos.

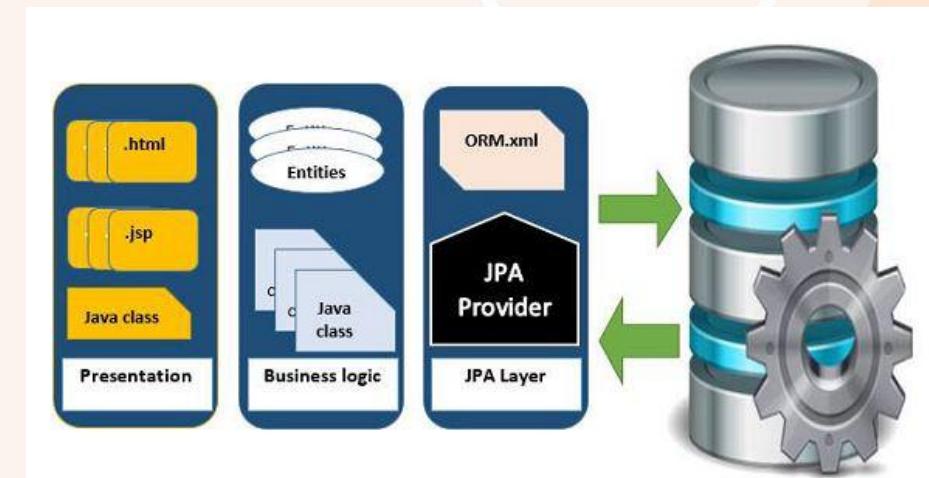
Normalmente, los desarrolladores de Java utilizan mucho código o utilizan el marco propietario para interactuar con la base de datos, mientras que al utilizar JPA, la carga de interacción con la base de datos se reduce significativamente. Forma un puente entre los modelos de objetos (programa Java) y los modelos relacionales (programa de base de datos).

Java Persistence API (JPA) es el estándar de Java para asignar objetos Java a una base de datos relacional.

¿Dónde se usa JPA?

Para reducir la carga de escribir códigos para la gestión de objetos relacionales, un programador sigue el marco "Proveedor JPA", que permite una fácil interacción con la instancia de la base de datos. Aquí el marco requerido es asumido por JPA.

JPA es una API de código abierto, por lo tanto, varios proveedores empresariales como Oracle, Redhat, Eclipse, etc. ofrecen nuevos productos al agregarles la versión de persistencia JPA. Algunos de estos productos incluyen Hibernate, Eclipselink, Toplink, Spring Data JPA, etc.



Arquitectura JPA a nivel de clase

EntityManagerFactory es una clase de fábrica de EntityManager. Crea y gestiona múltiples instancias de EntityManager.

EntityManager es una interfaz que gestiona las operaciones de persistencia en objetos. Funciona como fábrica para la instancia de consulta.

Las entidades son los objetos de persistencia, se almacenan como registros en la base de datos.

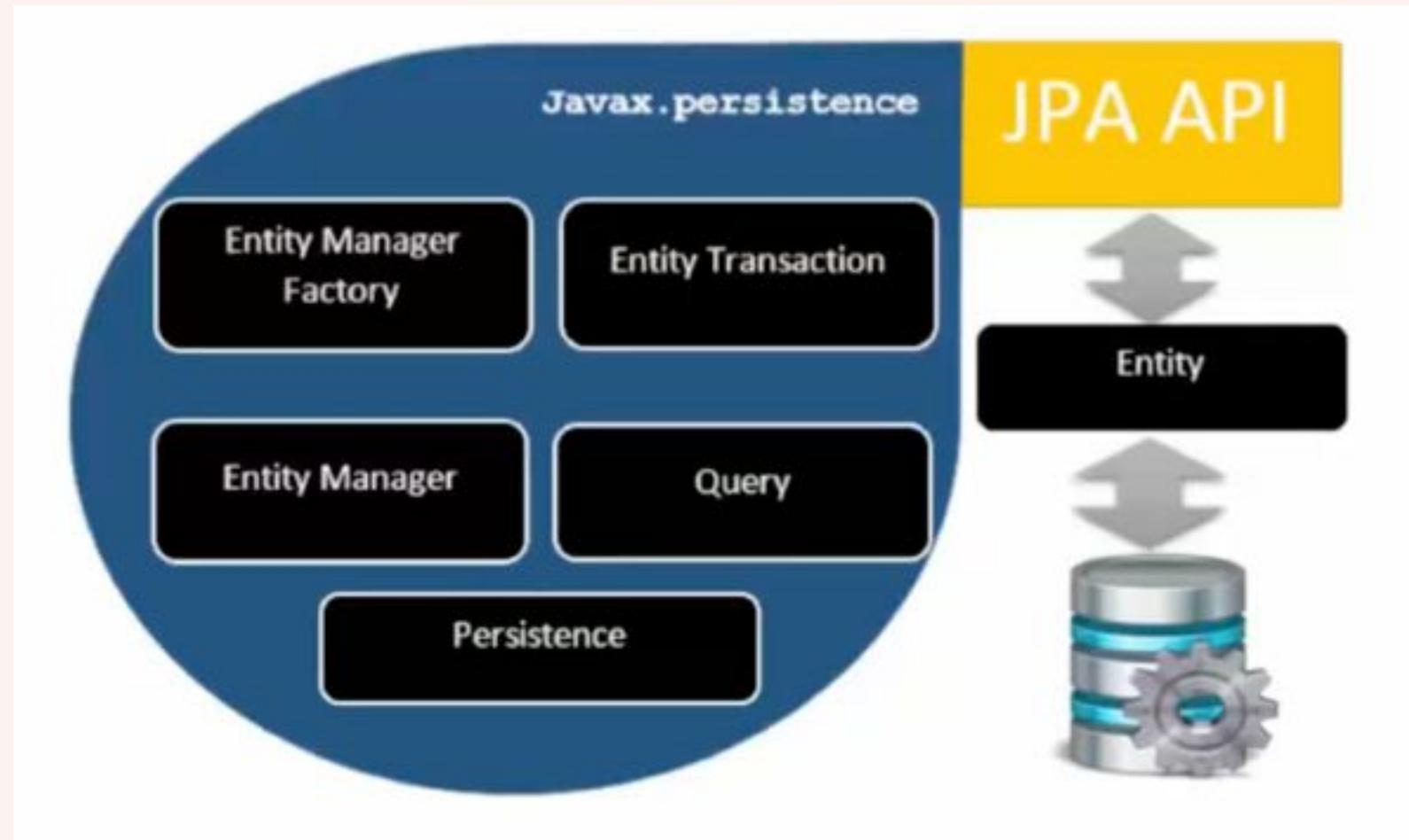
Arquitectura JPA a nivel de clase

EntityTransaction tiene una relación uno a uno con EntityManager

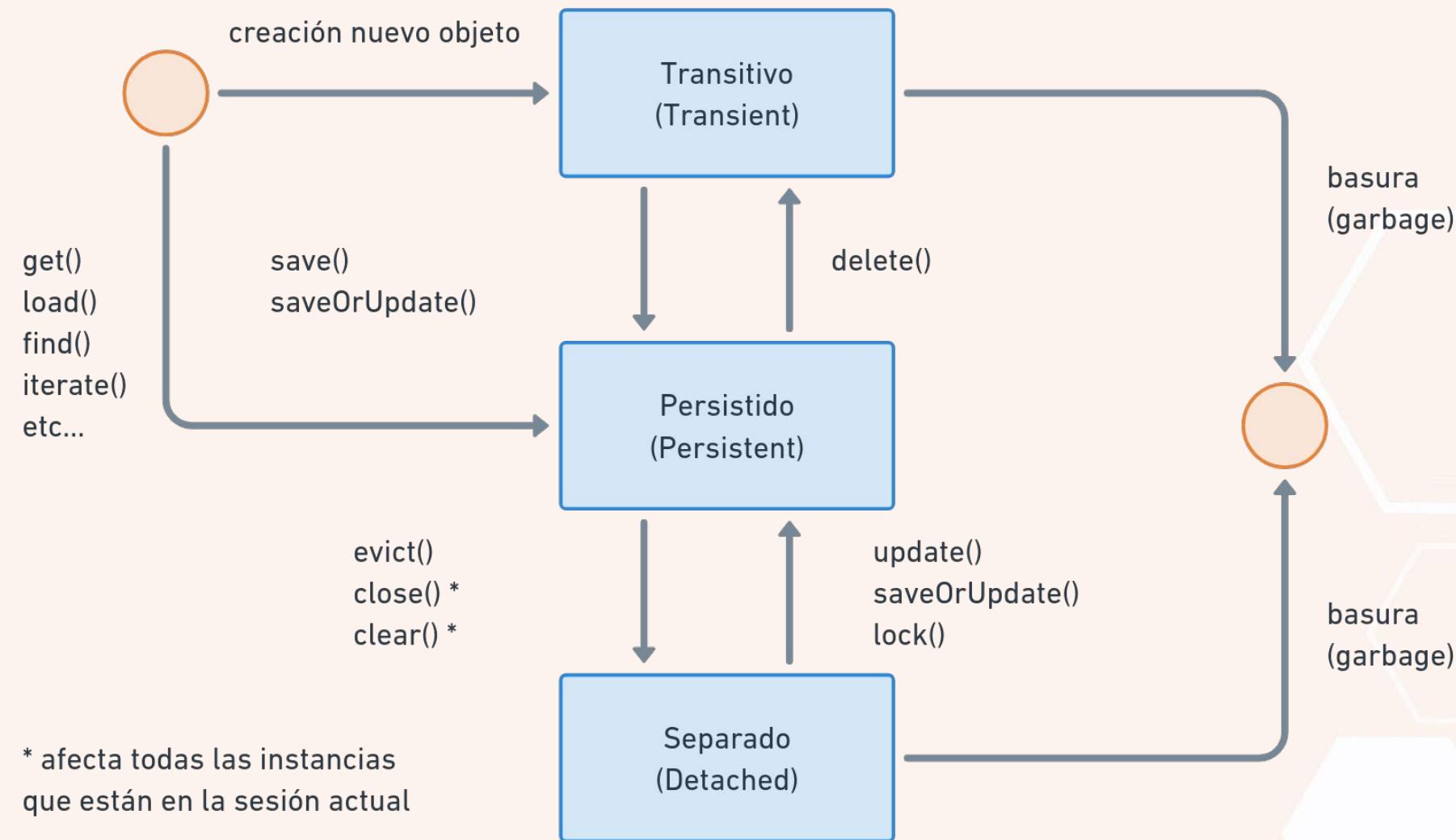
La clase de persistencia contiene métodos estáticos para obtener la instancia de EntityManagerFactory

La consulta se implementa para obtener objetos relacionales que cumplan con los criterios.

Arquitectura JPA a nivel de clase



Ciclo de Vida de Hibernate



Estados de los Objetos

Estado Transitivo (Transitive):

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

Estado Persistente (Persisted):

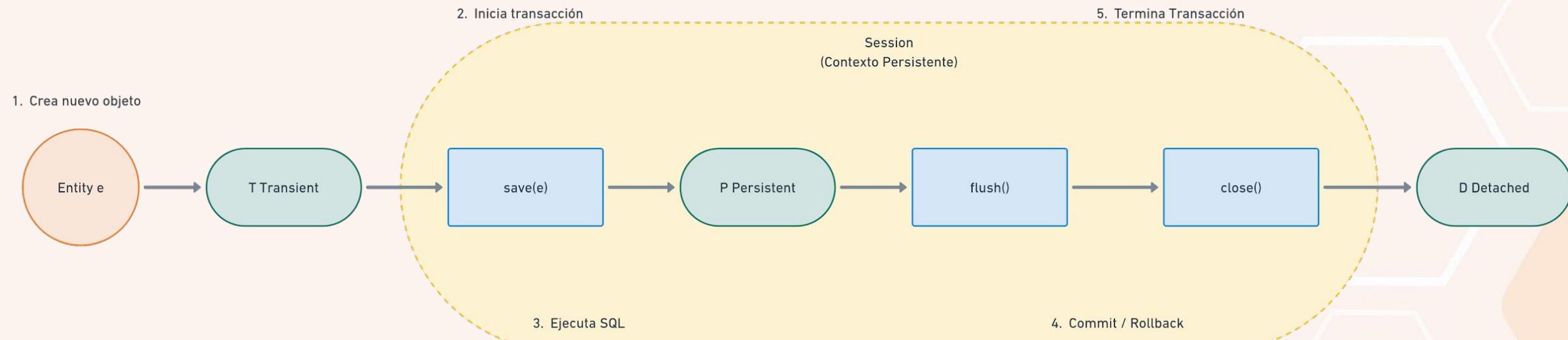
- Un objeto persistente tiene asociado un registro en la BD
- Los objetos persistentes siempre están asociados con una sesión y son transaccionales. Su estado se sincroniza con la DB al terminar la transacción

Estados de los Objetos

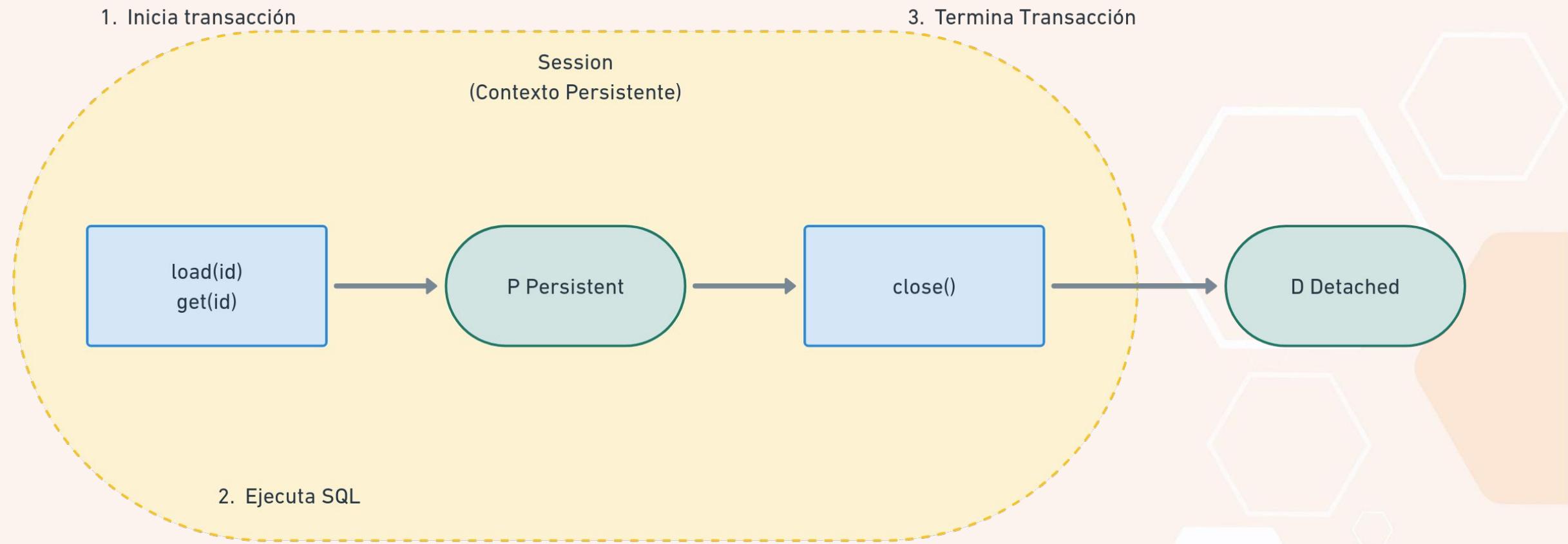
Estado Separado (Detached):

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

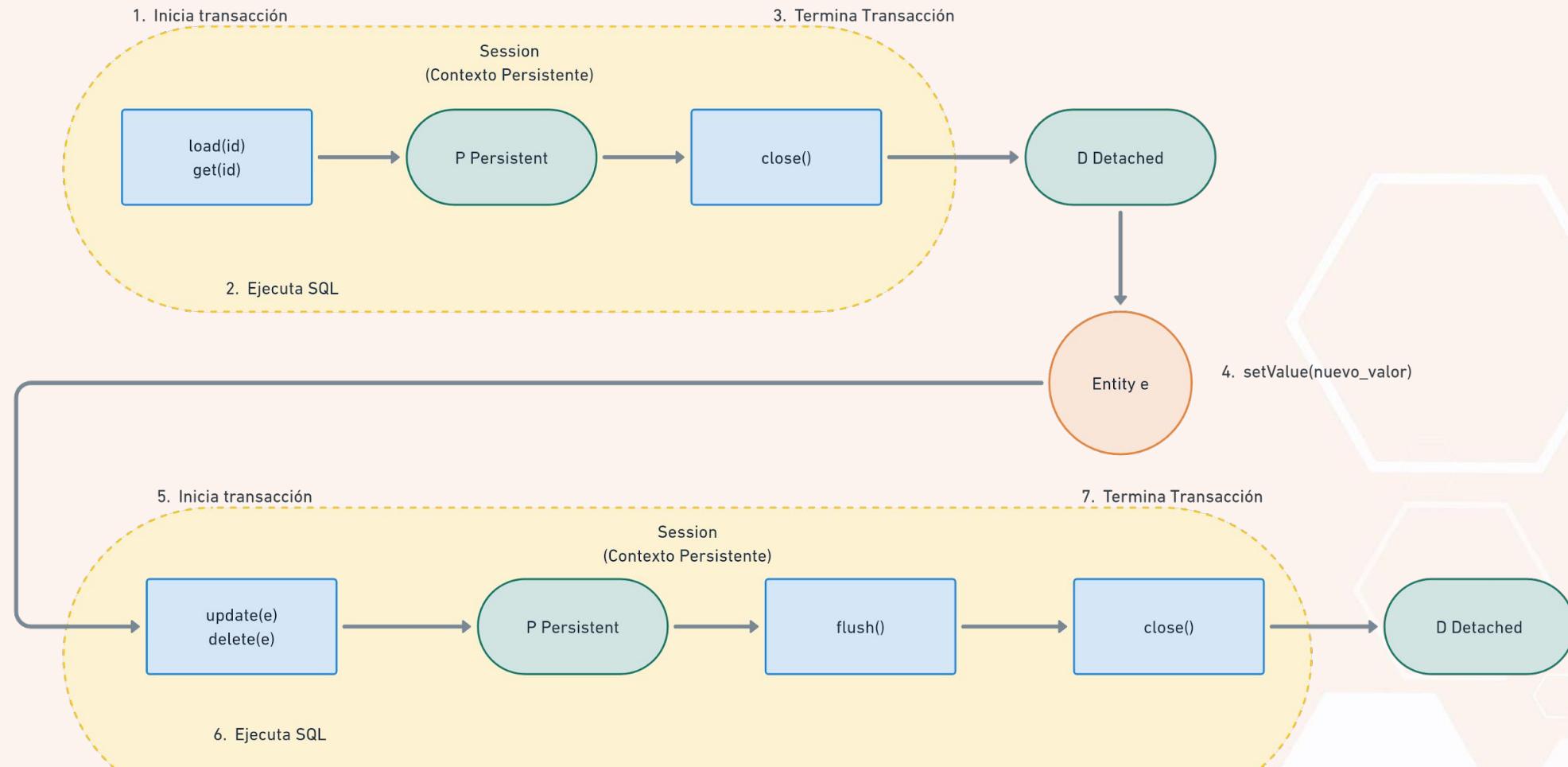
Persistir un objeto en hibernate



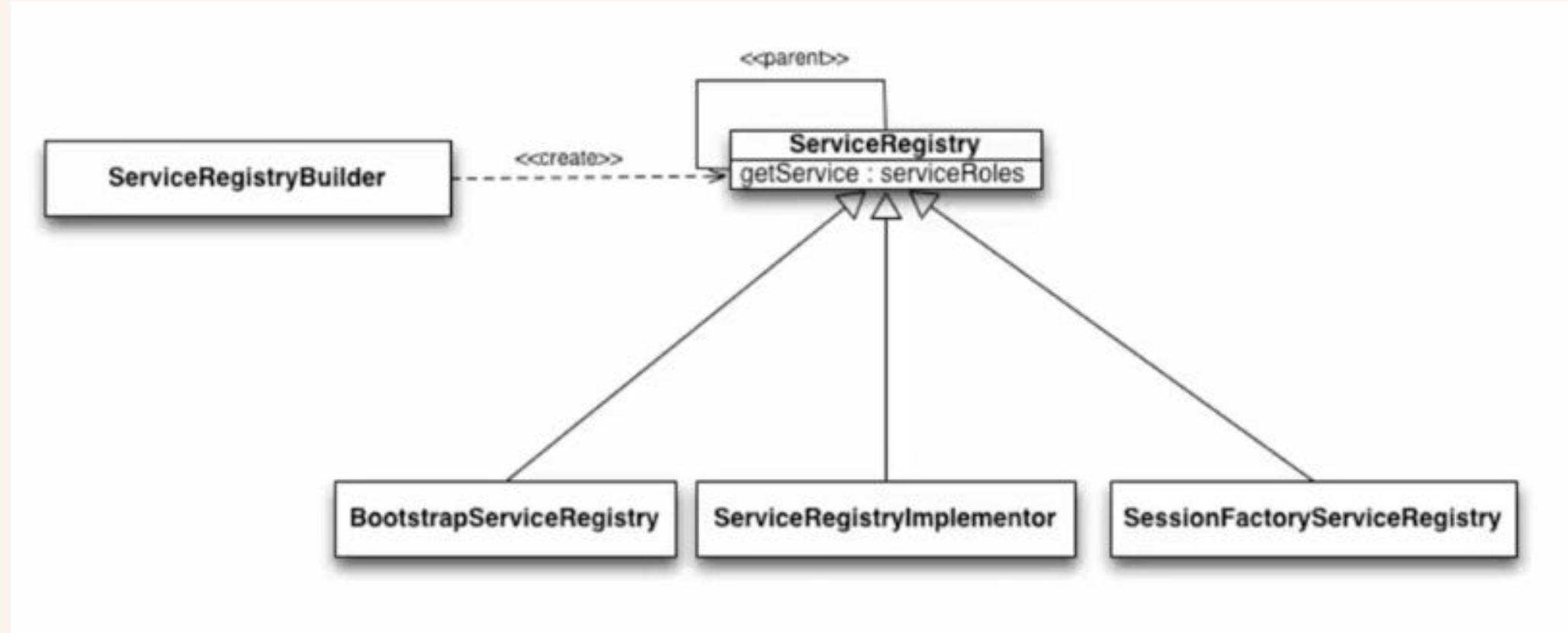
Recuperar un objeto en hibernate



Modificar o eliminar un objeto en hibernate



Hibernate Service Registries



Descripción General

Los servicios y registros son nuevos como concepto formalizado a partir de la versión 4.0

La funcionalidad proporcionada por los diferentes Servicios existe en Hibernate desde hace mucho, mucho más tiempo.

Lo que es nuevo es que su administración, sus ciclos de vida y dependencias se gestionan a través de un contenedor ligero y dedicado al que llamamos **ServiceRegistry**.

¿Qué es un servicio (capa de servicio)?

Los servicios proporcionan varios tipos de funcionalidades, de forma conectable.

Específicamente, son interfaces que definen cierta funcionalidad y luego implementaciones de esas interfaces de contrato de servicio.

La interfaz se conoce como rol de servicio; la clase de implementación se conoce como implementación del servicio.

La capacidad de conexión proviene del hecho de que la implementación del servicio se adhiere al contrato definido por la interfaz del rol de servicio y que los consumidores del servicio programan el rol de servicio, no la implementación.

¿Qué es un ServiceRegistry?

Un ServiceRegistry, en su forma más básica, aloja y gestiona Servicios. Su contrato está definido por la interfaz `org.hibernate.service.ServiceRegistry`.

Los servicios tienen un ciclo de vida. Tienen un alcance. Los servicios pueden depender de otros servicios. Y es necesario producirlos (elija utilizar una implementación en lugar de otra). ServiceRegistry satisface todas estas necesidades.

En una definición concisa, ServiceRegistry actúa como un contenedor de inversión de control (IoC). En OOP, IoC es un acoplamiento de objetos vinculado en tiempo de ejecución por un objeto ensamblador y, por lo general, no se conoce en tiempo de compilación mediante análisis estático.

¿Qué hace el ServiceRegistry?

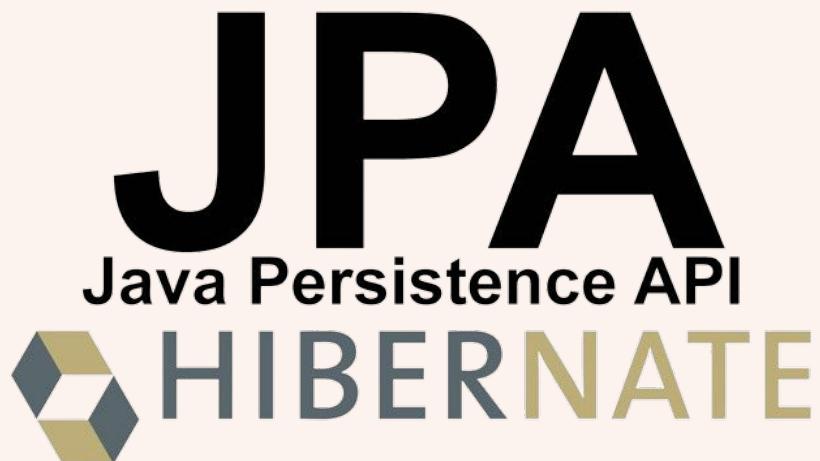
Un servicio está asociado con un ServiceRegistry. El ServiceRegistry abarca el Servicio.

ServiceRegistry gestiona el ciclo de vida del Servicio. ServiceRegistry maneja la inyección de dependencias en el Servicio (en realidad, se admiten tanto un enfoque de extracción como de inserción/inyección)

Los ServiceRegistries también son jerárquicos, lo que significa que un ServiceRegistry puede tener un ServiceRegistry principal. Los servicios de un registro pueden depender y utilizar los servicios de ese mismo registro, así como de cualquier registro principal.

2. Hibernate

2.1. Configuración



Configuración

La configuración de hibernate se puede realizar mediante el archivo **hibernate.cfg.xml**

Dentro de la configuración se encuentra el mapeo de las tablas de BD y las clases mediante archivos **hbm.xml**

La integración con la infraestructura java es con **datasource, jndi, jta**.

Configuración JDBC

hibernate.cfg.xml

Propiedades JDBC

- connection.driver_class
- connection.url
- connection.username
- connection.password
- connection.pool_size

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <!-- JDBC Database connection settings -->
        <property name="connection.driver_class">org.mariadb.jdbc.Driver</property>
        <property name="connection.url">jdbc:mariadb://localhost:3306/db</property>
        <property name="connection.username">root</property>
        <property name="connection.password">root</property>
        <!-- JDBCConnection pool settings ... using built-in test pool -->
        <property name="connection.pool_size">1</property>
        <!-- Select our SQL dialect -->
        <property name="dialect">org.hibernate.dialect.MariaDB103Dialect</property>
        <!-- Echo the SQL to stdout -->
        <property name="show_sql">true</property>
        <!-- Set the current session context -->
        <property name="current_session_context_class">thread</property>
        <!-- Drop and re-create the database schema on startup -->
        <property name="hbm2ddl.auto">create-drop</property>
        <!-- dbcp connection pool configuration -->
        <property name="hibernate.dbcp.initialSize">5</property>
        <property name="hibernate.dbcp.maxTotal">20</property>
        <property name="hibernate.dbcp.maxIdle">10</property>
        <property name="hibernate.dbcp.minIdle">5</property>
        <property name="hibernate.dbcp.maxWaitMillis">-1</property>
        <!-- entity mapping types -->
        <mapping resource="edu.unam.model.Entidad"/>
        <mapping class="Entidad"/>
    </session-factory>
</hibernate-configuration>
```

Dialecto Hibernate

Dialect es una clase y un puente entre los tipos de Java JDBC y los tipos de SQL, que contiene el mapeo entre el tipo de datos del lenguaje Java y el tipo de datos de la base de datos. Dialect permite a Hibernate generar SQL optimizado para una base de datos relacional en particular. Hibernate genera consultas para la base de datos específica basada en la clase Dialect.

- org.hibernate.dialect.MariaDB103Dialect

Mapeo de Archivo hbm.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class name="edu.unam.model.User" table="users" >
    <id name="id" column="id" type="integer">
      <generator class="assigned"/>
    </id>
    <property name="userName" column="name" type="string"/>
    <property name="password" type="string"/>
    <property name="emailAddress" type="string"/>
    <property name="lastLogon" type="date"/>
  </class>
</hibernate-mapping>
```

Mapeo de Archivo hbm.xml

Declaración de la DTD

- El documento DTD que se usa en los archivos **hbm.xml** viene en la distribución de hibernate (jar)

El elemento raíz es **<hibernate-mapping>**, se declara el elemento **<class>**.

Elemento **<class>** declara la clase persistente. Una clase persistente equivale a una tabla de base de datos

Elemento **<id>** permite definir el identificador del objeto, corresponde a la clave principal de la tabla de base de datos

Mapeo de Archivo hbm.xml

Elemento **<generator>** define qué clase se utilizará para generar los identificadores.

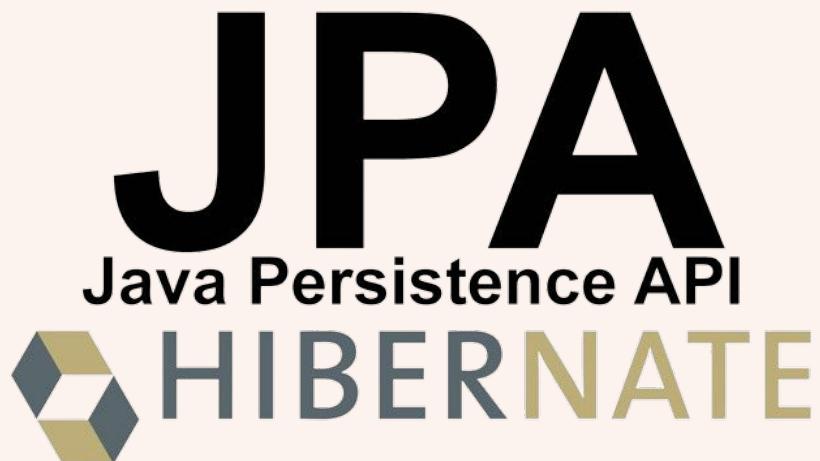
Elemento **<property>** declara una propiedad persistente de la clase que corresponde a una columna.

Tipos de relaciones (componentes y relaciones de objetos)

- 1 – 1, 1 – N, N – M, N – 1
- N – M y 1 – N son colecciones
- 1 – 1 y N – 1 son componentes

2. Hibernate

2.2. Mapeo de Clases Persistentes



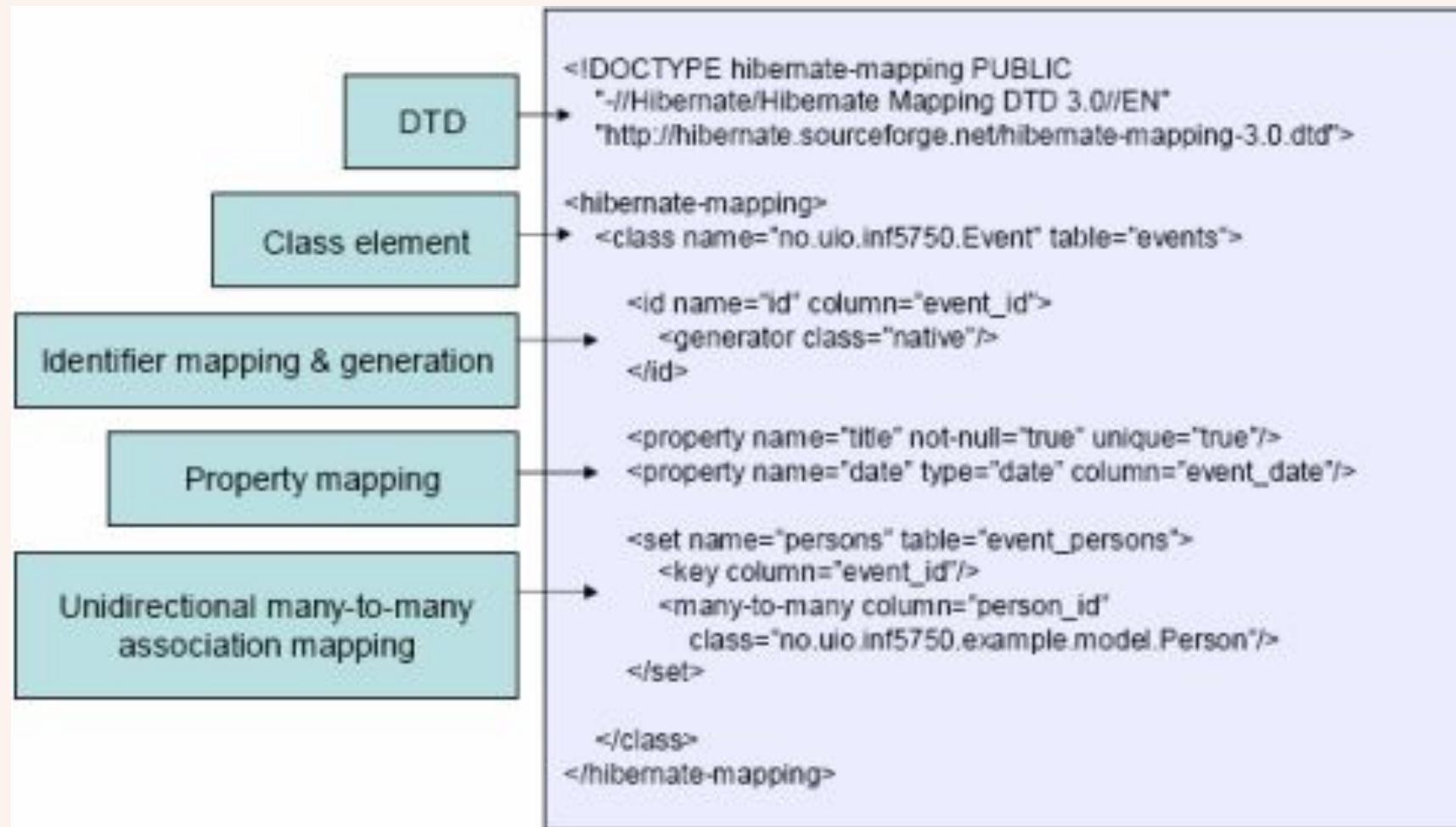
Opciones de Mapeo de una Clase

```
<hibernate-mapping
    schema="schemaName"                                (1)
    catalog="catalogName"                               (2)
    default-cascade="cascade_style"                   (3)
    default-access="field|property|ClassName"        (4)
    default-lazy="true|false"                          (5)
    auto-import="true|false"                           (6)
    package="package.name"                            (7)
/>
```

- ① schema (optional): the name of a database schema.
- ② catalog (optional): the name of a database catalog.
- ③ default-cascade (optional - defaults to none): a default cascade style.
- ④ default-access (optional - defaults to property): the strategy Hibernate should use for accessing all properties. It can be a custom implementation of PropertyAccessor.
- ⑤ default-lazy (optional - defaults to true): the default value for unspecified lazy attributes of class and collection mappings.
- ⑥ auto-import (optional - defaults to true): specifies whether we can use unqualified class names of classes in this mapping in the query language.
- ⑦ package (optional): specifies a package prefix to use for unqualified class names in the mapping document.

<https://docs.jboss.org/hibernate/orm/3.5/reference/en/html/mapping.html#mapping-declaration-property>

Opciones de Mapeo de una Clase



Opciones de Mapeo de una Clase

```

<class
    name="ClassName"                                (1)
    table="tableName"                               (2)
    discriminator-value="discriminator_value"      (3)
    mutable="true|false"                            (4)
    schema="owner"                                 (5)
    catalog="catalog"                             (6)
    proxy="ProxyInterface"                         (7)
    dynamic-update="true|false"                    (8)
    dynamic-insert="true|false"                   (9)
    select-before-update="true|false"              (10)
    polymorphism="implicit|explicit"             (11)
    where="arbitrary sql where condition"        (12)
    persister="PersisterClass"                   (13)
    batch-size="N"                                (14)
    optimistic-lock="none|version|dirty|all"       (15)
    lazy="true|false"                            (16)
    entity-name="EntityName"                      (17)
    check="arbitrary sql check condition"        (18)
    rowid="rowid"                                (19)
    subselect="SQL expression"                   (20)
    abstract="true|false"                          (21)
    node="element-name"
/>
  
```

- ➊ name (optional): the fully qualified Java class name of the persistent class or interface. If this attribute is missing, it is assumed that the mapping is for a non-POJO entity.
- ➋ table (optional - defaults to the unqualified class name): the name of its database table.
- ➌ discriminator-value (optional - defaults to the class name): a value that distinguishes individual subclasses that is used for polymorphic behavior. Acceptable values include null and not null.
- ➍ mutable (optional - defaults to true): specifies that instances of the class are (not) mutable.
- ➎ schema (optional): overrides the schema name specified by the root <hibernate-mapping> element.
- ➏ catalog (optional): overrides the catalog name specified by the root <hibernate-mapping> element.
- ➐ proxy (optional): specifies an interface to use for lazy initializing proxies. You can specify the name of the class itself.
- ➑ dynamic-update (optional - defaults to false): specifies that UPDATE SQL should be generated at runtime and can contain only those columns whose values have changed.
- ➒ dynamic-insert (optional - defaults to false): specifies that INSERT SQL should be generated at runtime and contain only the columns whose values are not null.
- ➓ select-before-update (optional - defaults to false): specifies that Hibernate should *never* perform an SQL UPDATE unless it is certain that an object is actually modified. Only when a transient object has been associated with a new session using update(), will Hibernate perform an extra SQL SELECT to determine if an UPDATE is actually required.
- ➔ polymorphism (optional - defaults to implicit): determines whether implicit or explicit query polymorphism is used.
- ➕ where (optional): specifies an arbitrary SQL WHERE condition to be used when retrieving objects of this class.
- ➖ persister (optional): specifies a custom ClassPersister.
- ➗ batch-size (optional - defaults to 1): specifies a "batch size" for fetching instances of this class by identifier.
- ➘ optimistic-lock (optional - defaults to version): determines the optimistic locking strategy.
- ➙ lazy (optional): lazy fetching can be disabled by setting lazy="false".
- ➚ entity-name (optional - defaults to the class name): Hibernate3 allows a class to be mapped multiple times, potentially to different tables. It also allows entity mappings that are represented by Maps or XML at the Java level. In these cases, you should provide an explicit arbitrary name for the entity. See [Section 4.4, "Dynamic models"](#) and [Chapter 18, XML Mapping](#) for more information.
- ➛ check (optional): an SQL expression used to generate a multi-row check constraint for automatic schema generation.
- ➜ rowid (optional): Hibernate can use ROWIDs on databases. On Oracle, for example, Hibernate can use the rowid extra column for fast updates once this option has been set to rowid. A ROWID is an implementation detail and represents the physical location of a stored tuple.
- ➝ subselect (optional): maps an immutable and read-only entity to a database subselect. This is useful if you want to have a view instead of a base table. See below for more information.
- ➞ abstract (optional): is used to mark abstract superclasses in <union-subclass> hierarchies.

Opciones de Mapeo de una Clase

```
<property
    name="propertyName"      (1)
    column="column_name"    (2)
    type="typename"          (3)
    update="true|false"      (4)
    insert="true|false"      (4)
    formula="arbitrary SQL expression" (5)
    access="field|property|ClassName" (6)
    lazy="true|false"        (7)
    unique="true|false"      (8)
    not-null="true|false"    (9)
    optimistic-lock="true|false" (10)
    generated="never|insert|always" (11)
    node="element-name|@attribute-name|element/@attribute|."
    index="index_name"
    unique_key="unique_key_id"
    length="L"
    precision="P"
    scale="S" />
```

- ① name: the name of the property, with an initial lowercase letter.
- ② column (optional - defaults to the property name): the name of the mapped database table column. This can also be specified by nested <column> element(s).
- ③ type (optional): a name that indicates the Hibernate type.
- ④ update, insert (optional - defaults to true): specifies that the mapped columns should be included in SQL UPDATE and/or INSERT statements. Setting both to false allows a pure "derived" property whose value is initialized from some other property that maps to the same column(s), or by a trigger or other application.
- ⑤ formula (optional): an SQL expression that defines the value for a *computed* property. Computed properties do not have a column mapping of their own.
- ⑥ access (optional - defaults to property): the strategy Hibernate uses for accessing the property value.
- ⑦ lazy (optional - defaults to false): specifies that this property should be fetched lazily when the instance variable is first accessed. It requires build-time bytecode instrumentation.
- ⑧ unique (optional): enables the DDL generation of a unique constraint for the columns. Also, allow this to be the target of a property-ref.
- ⑨ not-null (optional): enables the DDL generation of a nullability constraint for the columns.
- ⑩ optimistic-lock (optional - defaults to true): specifies that updates to this property do or do not require acquisition of the optimistic lock. In other words, it determines if a version increment should occur when this property is dirty.
- ⑪ generated (optional - defaults to never): specifies that this property value is actually generated by the database. See the discussion of [generated properties](#) for more information.

Opciones de Mapeo de una Clase

```
<id  
      name="propertyName"  
      type="typename"  
      column="column_name"  
      unsaved-value="null|any|none|undefined|id_value"  
      access="field|property|ClassName">  
      node="element-name|@attribute-name|element/@attribute|."  
  
      <generator class="generatorClass"/>  
</id>
```

```
<id name="id" type="long" column="cat_id">  
    <generator class="org.hibernate.id.TableHiLoGenerator">  
        <param name="table">uid_table</param>  
        <param name="column">next_hi_value_column</param>  
    </generator>  
</id>
```

(1)
(2)
(3)
(4)
(5)

- ① name (optional): the name of the identifier property.
- ② type (optional): a name that indicates the Hibernate type.
- ③ column (optional - defaults to the property name): the name of the primary key column.
- ④ unsaved-value (optional - defaults to a "sensible" value): an identifier property value that indicates an instance is newly instantiated (unsaved), distinguishing it from detached instances that were saved or loaded in a previous session.
- ⑤ access (optional - defaults to property): the strategy Hibernate should use for accessing the property value.

Opciones de Mapeo de una Clase

```
<many-to-one
    name="propertyName"          (1)
    column="column_name"        (2)
    class="ClassName"           (3)
    cascade="cascade_style"     (4)
    fetch="join|select"         (5)
    update="true|false"         (6)
    insert="true|false"         (6)
    property-ref="propertyNameFromAssociatedClass" (7)
    access="field|property|ClassName" (8)
    unique="true|false"         (9)
    not-null="true|false"        (10)
    optimistic-lock="true|false" (11)
    lazy="proxy|no-proxy|false"  (12)
    not-found="ignore|exception" (13)
    entity-name="EntityName"    (14)
    formula="arbitrary SQL expression" (15)
    node="element-name|@attribute-name|element/@attribute|."
    embed-xml="true|false"
    index="index_name"
    unique_key="unique_key_id"
    foreign-key="foreign_key_name" />
```

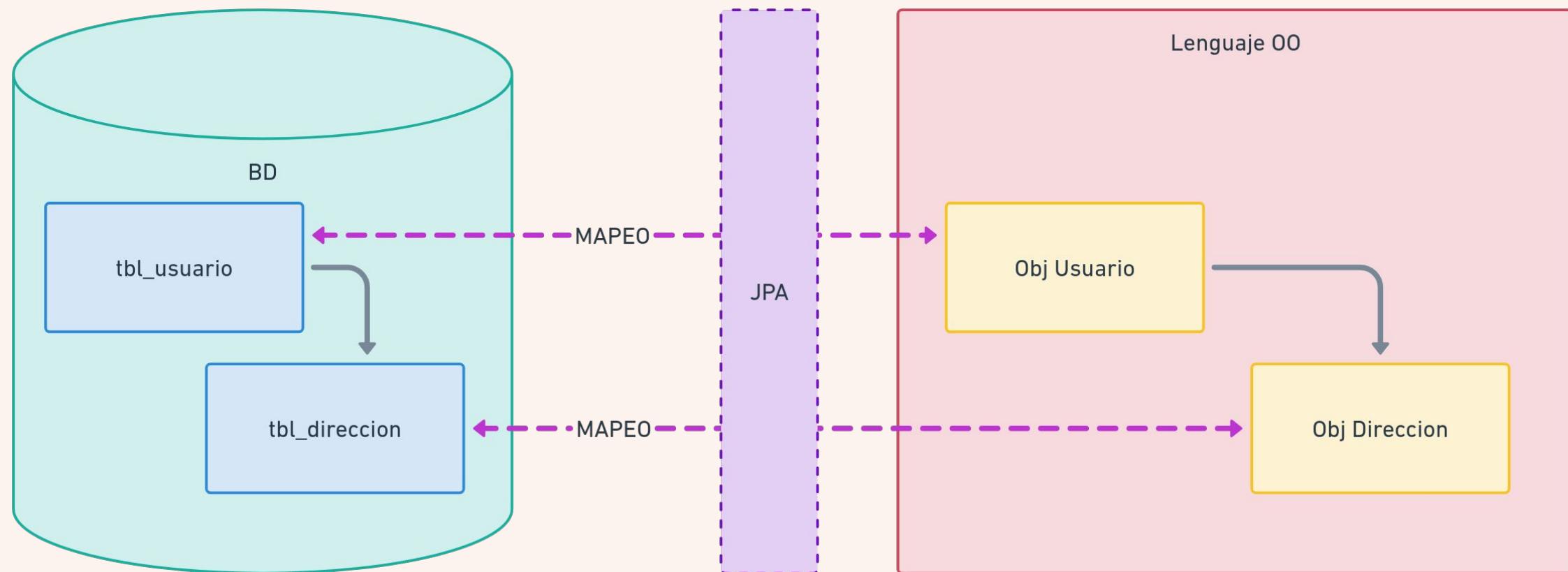
- ① name: the name of the property.
- ② column (optional): the name of the foreign key column. This can also be specified by nested <column> element(s).
- ③ class (optional - defaults to the property type determined by reflection): the name of the associated class.
- ④ cascade (optional): specifies which operations should be cascaded from the parent object to the associated object.
- ⑤ fetch (optional - defaults to select): chooses between outer-join fetching or sequential select fetching.
- ⑥ update, insert (optional - defaults to true): specifies that the mapped columns should be included in SQL UPDATE and/or INSERT statements. Setting both to false allows a pure "derived" association whose value is initialized from another property that maps to the same column(s), or by a trigger or other application.
- ⑦ property-ref (optional): the name of a property of the associated class that is joined to this foreign key. If not specified, the primary key of the associated class is used.
- ⑧ access (optional - defaults to property): the strategy Hibernate uses for accessing the property value.
- ⑨ unique (optional): enables the DDL generation of a unique constraint for the foreign-key column. By allowing this to be the target of a property-ref, you can make the association multiplicity one-to-one.
- ⑩ not-null (optional): enables the DDL generation of a nullability constraint for the foreign key columns.
- ⑪ optimistic-lock (optional - defaults to true): specifies that updates to this property do or do not require acquisition of the optimistic lock. In other words, it determines if a version increment should occur when this property is dirty.
- ⑫ lazy (optional - defaults to proxy): by default, single point associations are proxied. lazy="no-proxy" specifies that the property should be fetched lazily when the instance variable is first accessed. This requires build-time bytecode instrumentation. lazy="false" specifies that the association will always be eagerly fetched.
- ⑬ not-found (optional - defaults to exception): specifies how foreign keys that reference missing rows will be handled. ignore will treat a missing row as a null association.
- ⑭ entity-name (optional): the entity name of the associated class.
- ⑮ formula (optional): an SQL expression that defines the value for a *computed* foreign key.

Opciones de Mapeo de una Clase

```
<one-to-one
    name="propertyName" (1)
    class="ClassName" (2)
    cascade="cascade_style" (3)
    constrained="true|false" (4)
    fetch="join|select" (5)
    property-ref="propertyNameFromAssociatedClass" (6)
    access="field|property|ClassName" (7)
    formula="any SQL expression" (8)
    lazy="proxy|no-proxy|false" (9)
    entity-name="EntityName" (10)
    node="element-name|@attribute-name|element/@attribute|."
    embed-xml="true|false"
    foreign-key="foreign_key_name" />
```

- ① name: the name of the property.
- ② class (optional - defaults to the property type determined by reflection): the name of the associated class.
- ③ cascade (optional): specifies which operations should be cascaded from the parent object to the associated object.
- ④ constrained (optional): specifies that a foreign key constraint on the primary key of the mapped table and references the table of the associated class. This option affects the order in which save() and delete() are cascaded, and determines whether the association can be proxied. It is also used by the schema export tool.
- ⑤ fetch (optional - defaults to select): chooses between outer-join fetching or sequential select fetching.
- ⑥ property-ref (optional): the name of a property of the associated class that is joined to the primary key of this class. If not specified, the primary key of the associated class is used.
- ⑦ access (optional - defaults to property): the strategy Hibernate uses for accessing the property value.
- ⑧ formula (optional): almost all one-to-one associations map to the primary key of the owning entity. If this is not the case, you can specify another column, columns or expression to join on using an SQL formula. See org.hibernate.test.onetoonerformula for an example.
- ⑨ lazy (optional - defaults to proxy): by default, single point associations are proxied. lazy="no-proxy" specifies that the property should be fetched lazily when the instance variable is first accessed. It requires build-time bytecode instrumentation. lazy="false" specifies that the association will always be eagerly fetched. Note that if constrained="false", proxying is impossible and Hibernate will eagerly fetch the association.
- ⑩ entity-name (optional): the entity name of the associated class.

¿Qué es el Mapeo Relacional?



CRUD

¿QUÉ ES CRUD?

EDteam

Hace referencia a un acrónimo, en él unen las primeras letras de las cuatro operaciones principales de **aplicaciones en sistemas de bases de datos**:

The diagram features the word "CRUD" in large, bold, white letters. Surrounding it are four icons with corresponding labels: "CREATE" (a person icon with a pencil), "READ" (a laptop icon displaying a document), "UPDATE" (a pencil icon), and "DELETE" (a trash bin icon). Arrows point from each operation name to its respective icon.

CREATE
Crea Registros

READ
Lee Registros

UPDATE
Actualiza Registros

DELETE
Borra Registros

Gracias a los CRUDs podrás usar estas operaciones para manejar los datos de tus aplicaciones.
Prof. Pablo España

Ahora sabes que cuando te digan "haremos el CRUD de esta entidad", significa las operaciones que aquí te explicamos.

Crea aplicaciones que se ejecuten en la consola de tu sistema operativo en:

ed.team/cursos/tallerpython

POJO (Plain Old Java Object)

POJO son las iniciales de "Plain Old Java Object". Un POJO es una instancia de una clase que no extiende ni implementa nada en especial. Para los programadores Java sirve para enfatizar el uso de clases simples y que no dependen de un framework en especial.

```
1  public class ClienteBean implements java.io.Serializable {  
2      private String nombre;  
3      private int edad;  
4  
5      public ClienteBean() {  
6          //Constructor  
7      }  
8  
9      public String getNombre() {  
10         return nombre;  
11     }  
12  
13     public void setNombre(String nombre) {  
14         this.nombre = nombre;  
15     }  
16  
17     public int getEdad() {  
18         return edad;  
19     }  
20  
21     public void setEdad(int edad) {  
22         this.edad = edad;  
23     }  
24 }
```

Configuración de API

SessionFactory

Es una fábrica de sesiones. Un objeto configuration es capaz de crear una sessionfactory ya que tiene toda la información necesaria.

Normalmente, una aplicación sólo tiene una sessionfactory.

A partir del **hibernate.cfg.xml** podemos crear una sessionfactory.

```
1 if (sessionFactory == null) {  
2     try {  
3         // Create registry  
4         registry = new StandardServiceRegistryBuilder().configure().build();  
5         // Create <etadatasources  
6         MetadataSources sources = new MetadataSources(registry);  
7         // Create Metadata  
8         Metadata metadata = sources.getMetadataBuilder().build();  
9         // Create SessionFactory  
10        sessionFactory = metadata.getSessionFactoryBuilder().build();  
11    } catch (Exception e) {  
12        e.printStackTrace();  
13        if (registry != null) {  
14            StandardServiceRegistryBuilder.destroy(registry);  
15        }  
16    }  
17 }
```

Configuración de API

Session

La principal interfaz entre la aplicación java y hibernate. Es la que mantiene las conversaciones entre la aplicación y la base de datos. Permite añadir, modificar y borrar objetos en la base de datos.

```
...
1 try (Session session = HibernateUtil.getSessionFactory().openSession()) {
2     // Start a transaction
3     transaction = session.beginTransaction();
4     // save the student objects
5     session.save(student);
6     session.save(student1);
7     // commit transaction
8     transaction.commit();
9 }
```

Configuración de API

Transaction

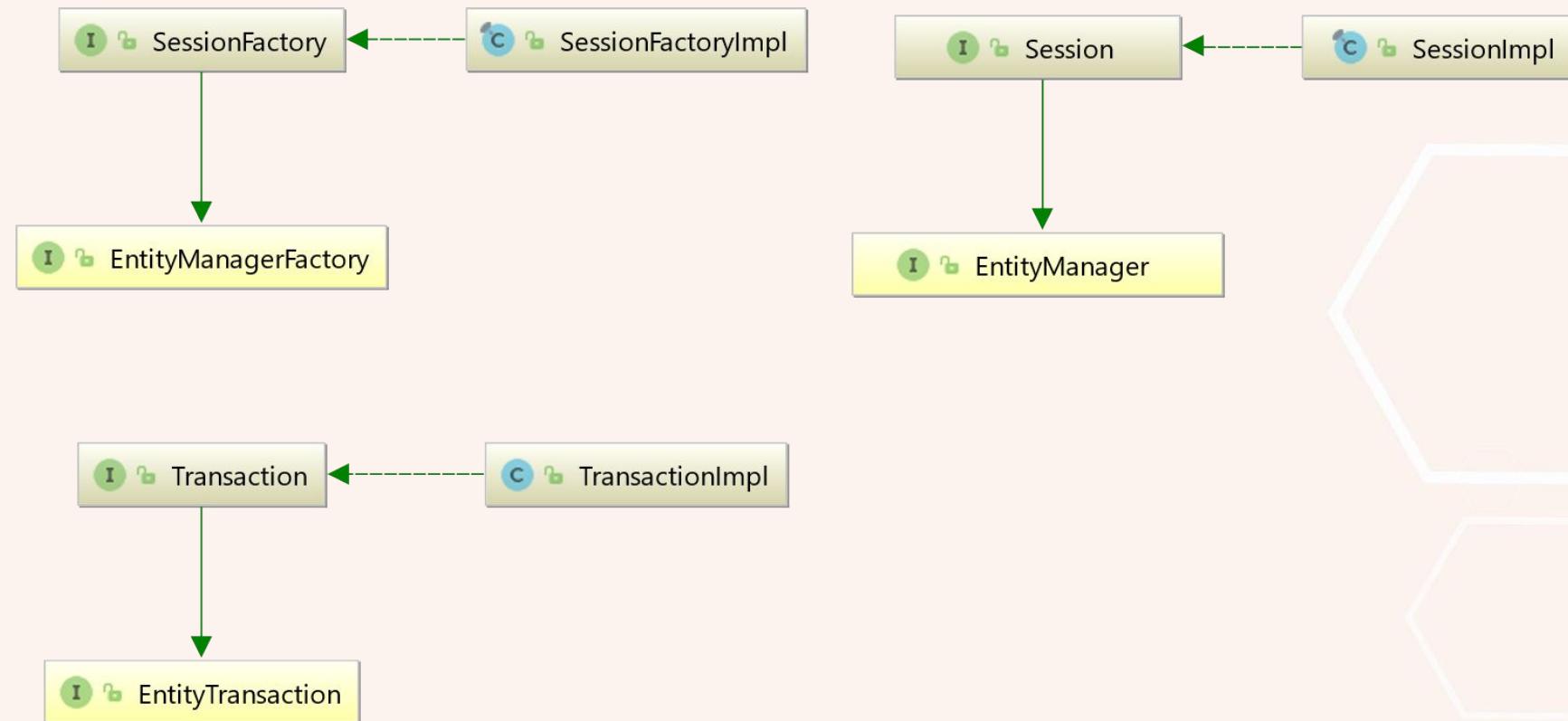
Se encarga de las transacciones hacia la base de datos

Commit

Rollback

```
...
1 try (Session session = HibernateUtil.getSessionFactory().openSession()) {
2     // Start a transaction
3     transaction = session.beginTransaction();
4     // save the student objects
5     session.save(student);
6     session.save(student1);
7     // commit transaction
8     transaction.commit();
9 }
```

Configuración de API



API Hibernate

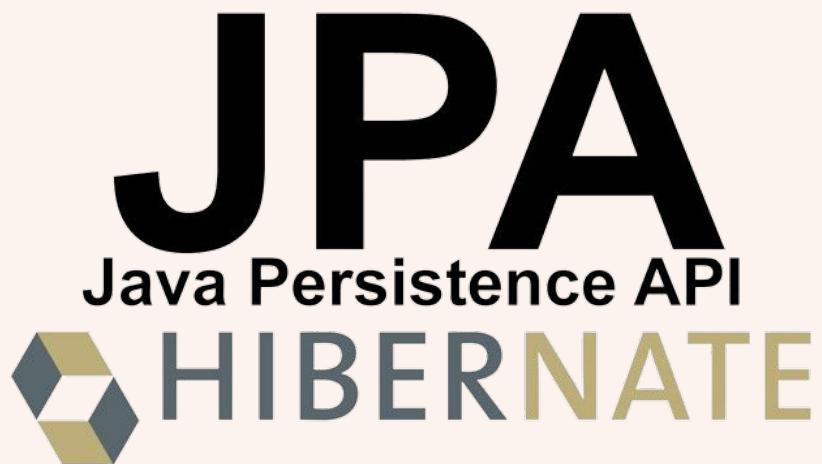
Hibernate provee de diferentes métodos para la persistencia de datos.

- save
- delete
- get
- load

Para las búsquedas en la base de datos se cuenta con HQL (Hibernate Query Language).

2. Hibernate

2.3. Tipos de Datos



Mapeo de Clases

Dentro del mapeo de clases persistentes con hibernate encontramos la compatibilidad de tipos de datos. Hibernate maneja los siguientes tipos de datos para la correspondencia con el SGBD.

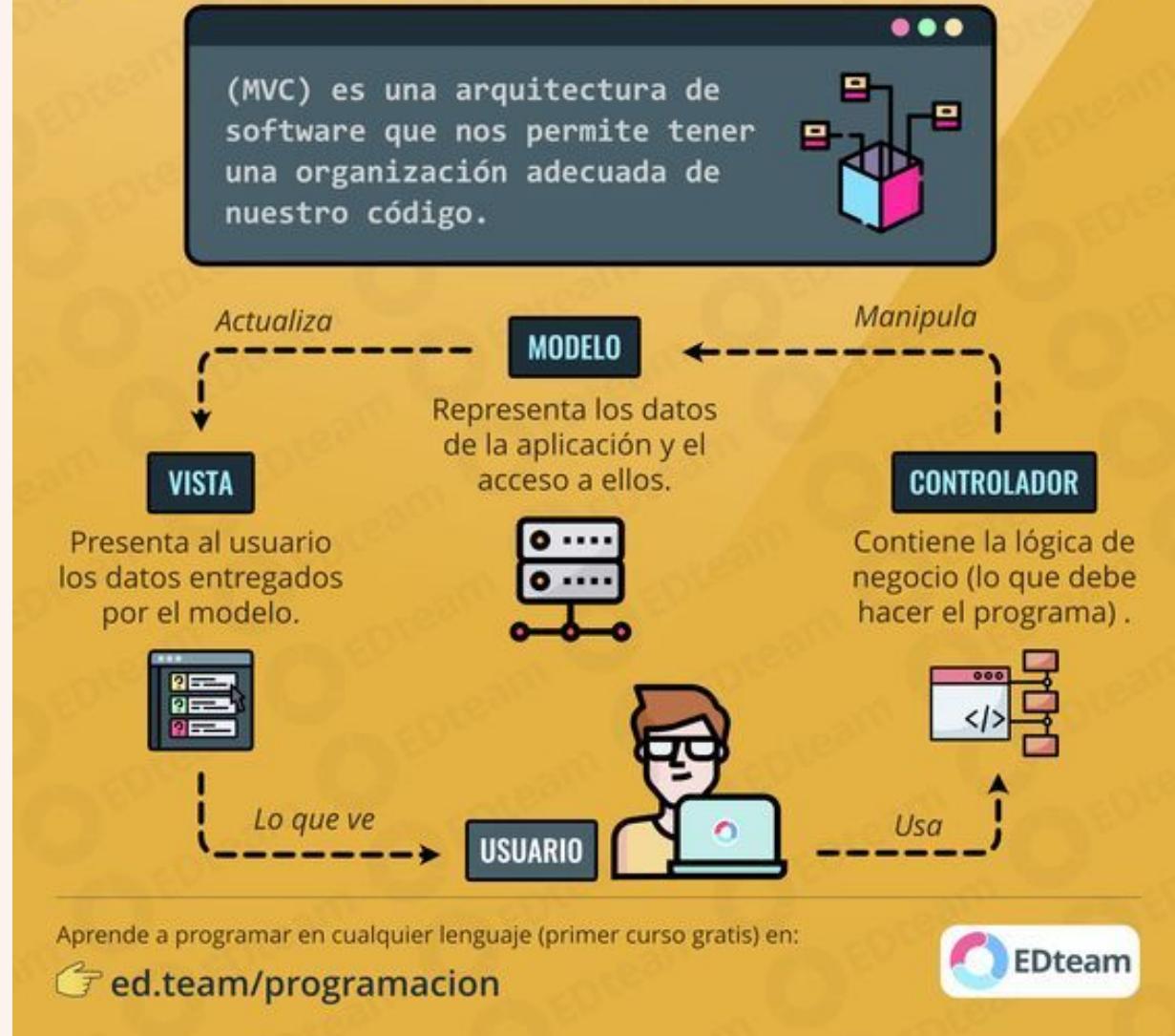
Hibernate type (org.hibernate.type package)	JDBC type	Java type	BasicTypeRegistry key(s)
StringType	VARCHAR	java.lang.String	string, java.lang.String
MaterializedClob	CLOB	java.lang.String	materialized_clob
TextType	LONGVARCHAR	java.lang.String	text
CharacterType	CHAR	char, java.lang.Character	character, char, java.lang.Character
BooleanType	BOOLEAN	boolean, java.lang.Boolean	boolean, java.lang.Boolean
NumericBooleanType	INTEGER, 0 is false, 1 is true	boolean, java.lang.Boolean	numeric_boolean
YesNoType	CHAR, 'N'/'n' is false, 'Y'/'y' is true. The uppercase value is written to the database.	boolean, java.lang.Boolean	yes_no

Mapeo de Clases

ByteType	TINYINT	byte, java.lang.Byte	byte, java.lang.Byte
ShortType	SMALLINT	short, java.lang.Short	short, java.lang.Short
IntegerType	INTEGER	int, java.lang.Integer	integer, int, java.lang.Integer
LongType	BIGINT	long, java.lang.Long	long, java.lang.Long
FloatType	FLOAT	float, java.lang.Float	float, java.lang.Float
DoubleType	DOUBLE	double, java.lang.Double	double, java.lang.Double
BigIntegerType	NUMERIC	java.math.BigInteger	big_integer, java.math.BigInteger
BigDecimalType	NUMERIC	java.math.BigDecimal	big_decimal, java.math.BigDecimal
TimestampType	TIMESTAMP	java.util.Date	timestamp, java.sql.Timestamp, java.util.Date
DbTimestampType	TIMESTAMP	java.util.Date	dbtimestamp
TimeType	TIME	java.util.Date	time, java.sql.Time
DateType	DATE	java.util.Date	date, java.sql.Date
CalendarType	TIMESTAMP	java.util.Calendar	calendar, java.util.Calendar, java.util.GregorianCalendar

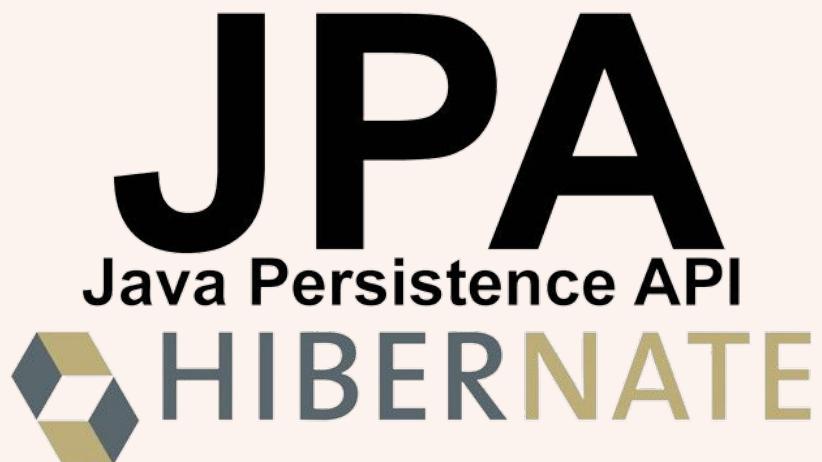
MVC

¿QUÉ ES EL MODELO VISTA CONTROLADOR?



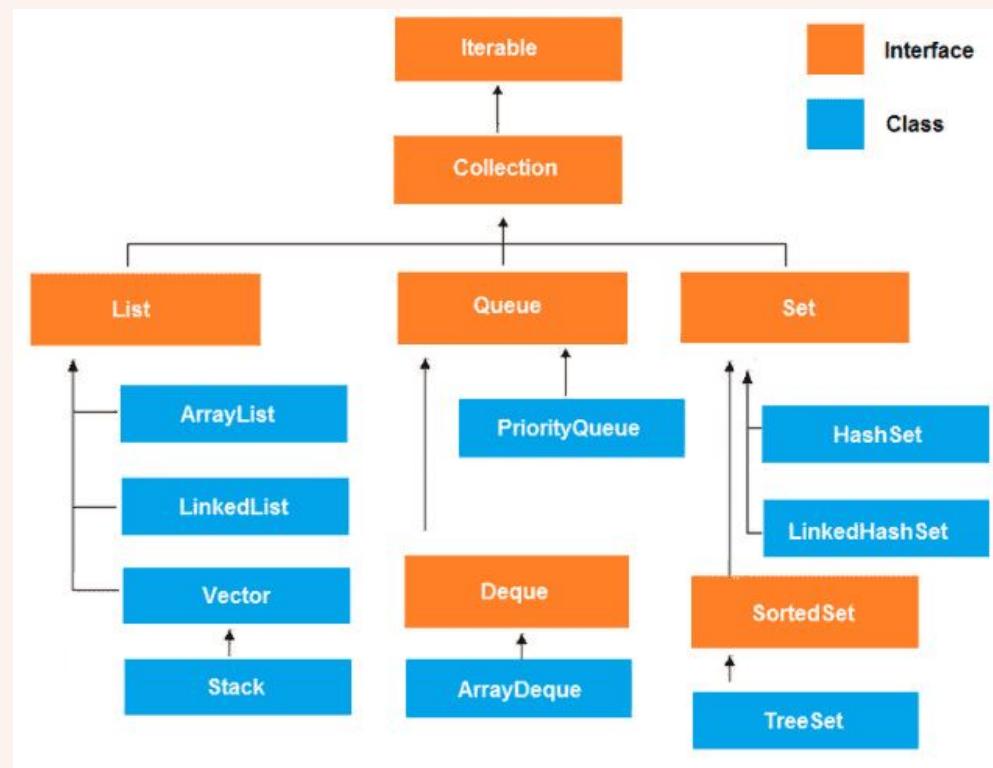
2. Hibernate

2.4. Mapeo de Colecciones



Colecciones

Las colecciones persistentes injectadas por Hibernate se comportan como HashMap, HashSet, TreeMap, TreeSet ó ArrayList, dependiendo del tipo de interfaz.



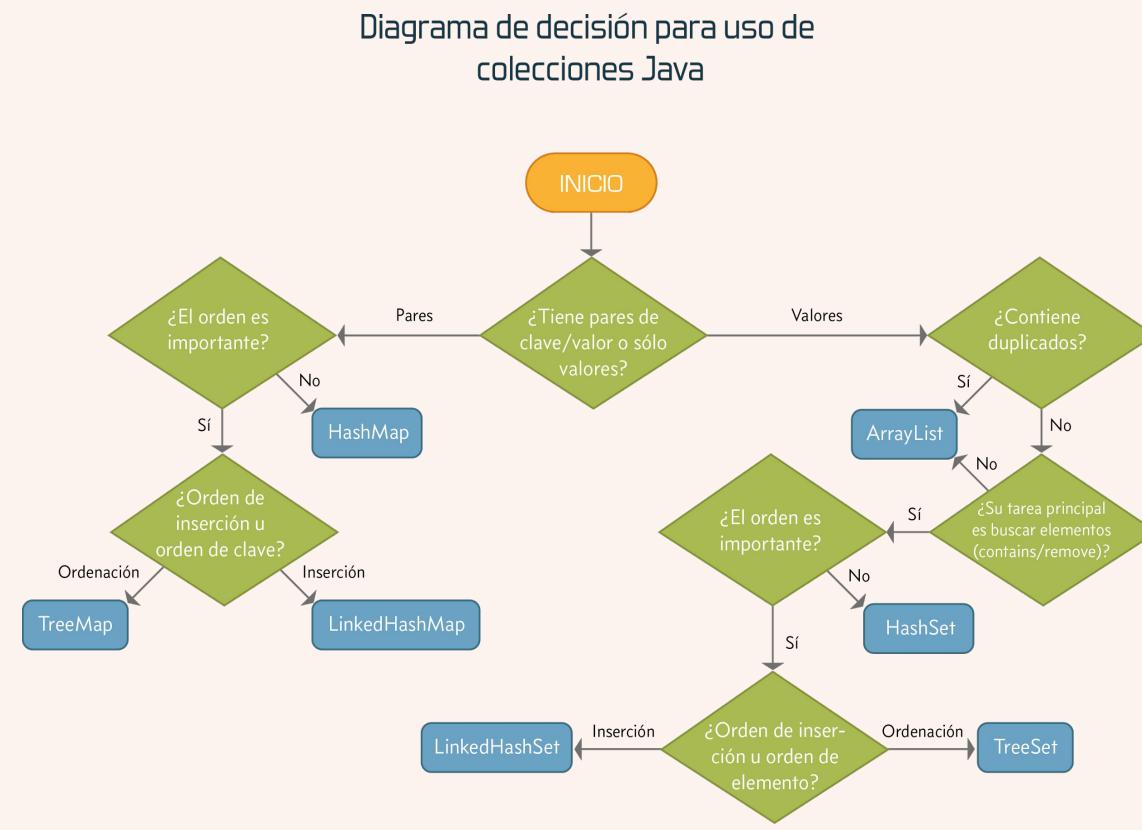
Colecciones

Las instancias de colecciones tienen el comportamiento usual de los tipos de valor. Son automáticamente persistidas al ser referenciadas por un objeto persistente y se borran automáticamente al perder la referencia. Si una colección se pasa de un objeto persistente a otro, puede que sus elementos se muevan de una tabla a otra. Dos entidades no pueden compartir una referencia a la misma instancia de colección. Debido al modelo relacional subyacente, las propiedades valuadas en colección no soportan la semántica de valor nulo.

Hibernate no distingue entre una referencia de colección nula y una colección vacía.

Colecciones

Las colecciones son declaradas utilizando <set>, <list>, <map>, <bag>, <array>.



Colecciones

Collection	Ordering	Random Access	Key-Value	Duplicate Elements	Null Element	Thread Safety
ArrayList	✓	✓	✗	✓	✓	✗
LinkedList	✓	✗	✗	✓	✓	✗
HashSet	✗	✗	✗	✗	✓	✗
TreeSet	✓	✗	✗	✗	✗	✗
HashMap	✗	✓	✓	✗	✓	✗
TreeMap	✓	✓	✓	✗	✗	✗
Vector	✓	✓	✗	✓	✓	✓
Hashtable	✗	✓	✓	✗	✗	✓
Properties	✗	✓	✓	✗	✗	✓
Stack	✓	✗	✗	✓	✓	✓
CopyOnWriteArrayList	✓	✓	✗	✓	✓	✓
ConcurrentHashMap	✗	✓	✓	✗	✗	✓
CopyOnWriteArraySet	✗	✗	✗	✗	✓	✓

2. Hibernate



2.5. Transacciones

Transacción

Una transacción es un conjunto de operaciones de datos que se realizan como una única unidad lógica de trabajo, que deben tener éxito o fallar por completo para garantizar la consistencia e integridad de los datos.

Transacción

La transacción tiene propiedades **ACID**:

Atómica: una transacción consiste en una o más acciones unidas como si fuera una sola unidad de trabajo. Atomicity asegura que todas las operaciones en una transacción ocurran o no ocurran.

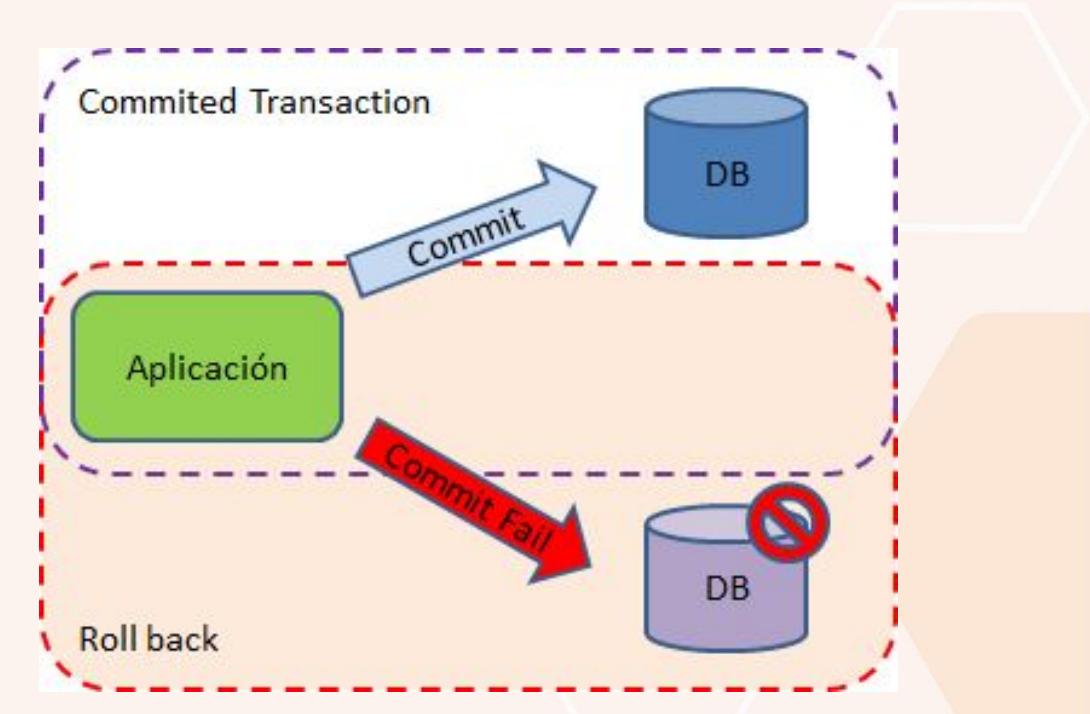
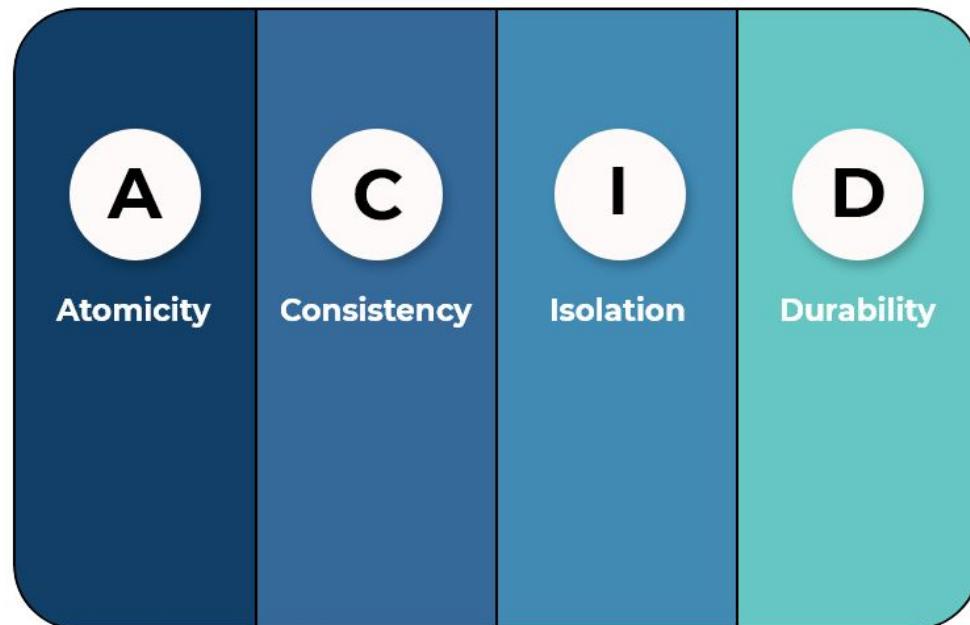
Consistencia (consistente): una vez que se completa una transacción (independientemente del éxito o el fracaso), el estado del sistema y sus reglas comerciales son consistentes. Es decir, los datos no deben destruirse.

Transacción

Aislado: la transacción debe permitir que varios usuarios operen con los mismos datos, y las operaciones de un usuario no se confundirá con las operaciones de otros usuarios.

Durable: una vez que se completa la transacción, el resultado de la transacción debe ser persistente.

Transacciones



Transacciones JDBC

En la API JDBC, la clase `java.sql.Connection` representa una conexión de base de datos. Proporciona los siguientes métodos para controlar las transacciones:

- **setAutoCommit (Boolean autoCommit)**: establece si se deben confirmar transacciones automáticamente.
- **commit()**: confirma la transacción
- **rollback()**: deshace la transacción

Transacciones JDBC

```
1 Connection con = null;
2 PreparedStatement psmt = null;
3 try {
4     con = DriverManager.getConnection(dbUrl, username, password);
5     //Establecer el modo de enviar transacciones manualmente
6     con.setAutoCommit(false);
7     psmt = ...;
8     psmt.executeUpdate();
9     //transaccion de confirmación
10    con.commit();
11 } catch(Exception e) {
12     //Deshacer transacciones
13     con.rollback();
14     ...
15 } finally {
16     ...
17 }
```

Transacciones en Hibernate

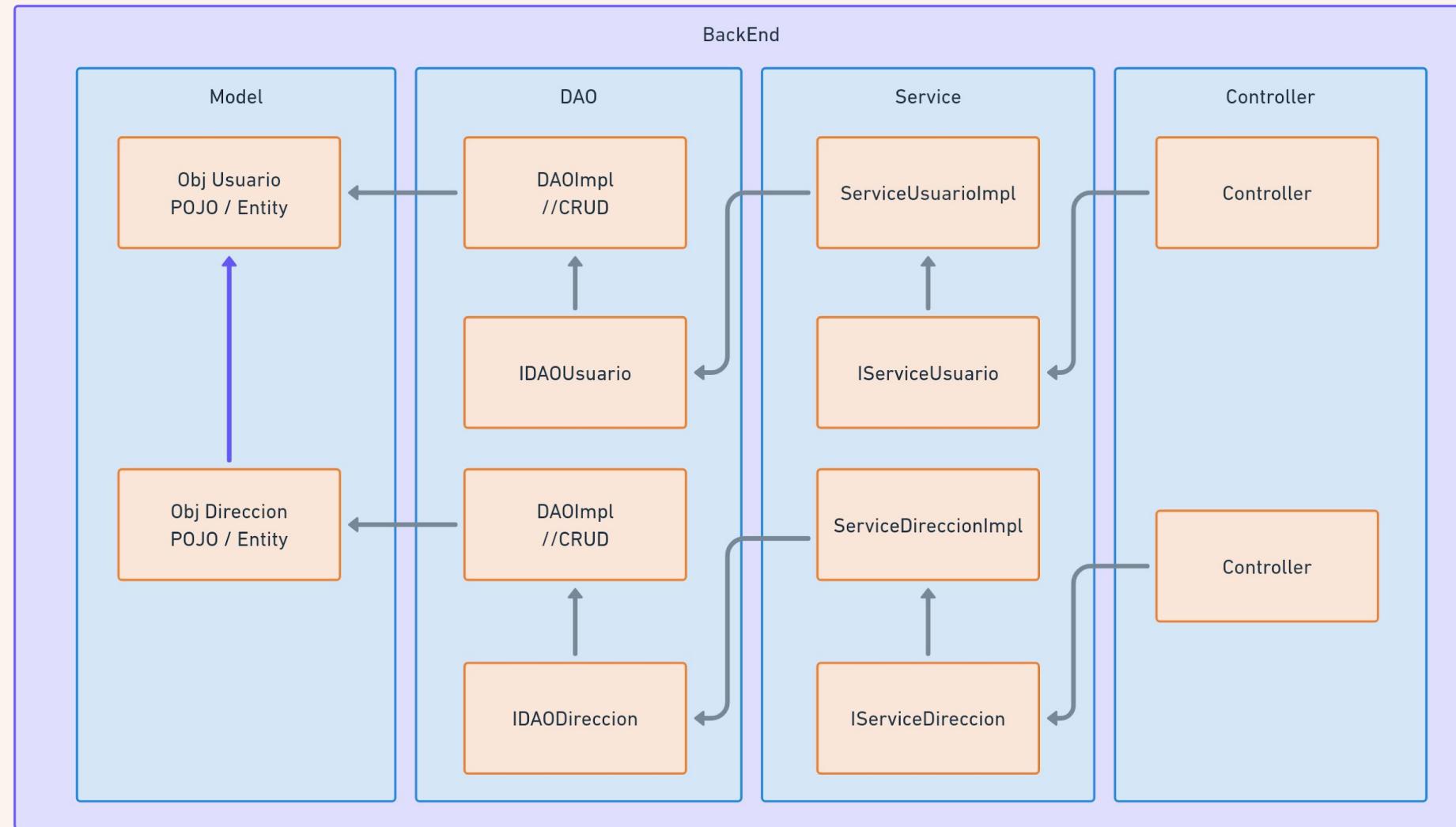
Hibernate implementa una ligera encapsulación de objetos de JDBC. Hibernate en sí no tiene funciones de procesamiento de transacciones en el momento del diseño. Las transacciones habituales de Hibernate solo encapsulan JDBCTransaction o JTATransaction subyacentes, y colocan Transaction and Session en el exterior. De hecho, la capa inferior es implementar la función de programación de la transacción delegando el JDBC o JTA subyacente.

Transacciones en Hibernate

- Hibernate usará transacciones JDBC por defecto
- Utiliza el procesamiento de transacciones JDBC de la siguiente manera:

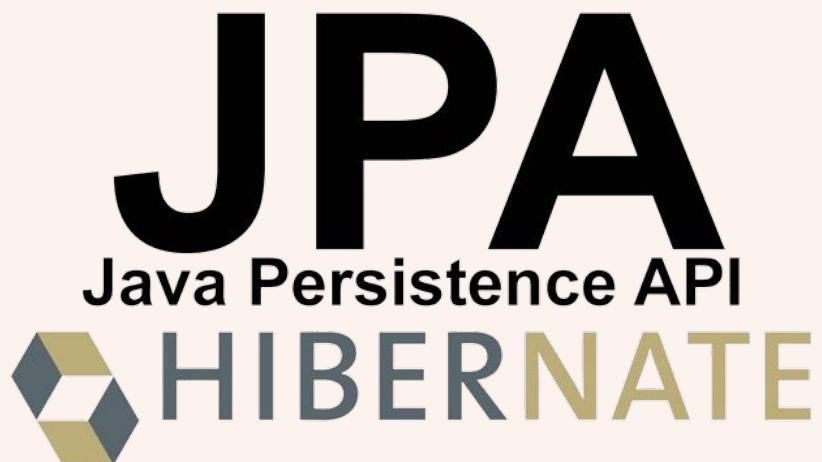
```
1 Transaction tx = null;
2 try {
3     tx = sess.beginTransaction();
4     // do some work
5     ...
6     tx.commit();
7 } catch(RuntimeException e) {
8     if (tx != null)
9         tx.rollback();
10    throw e; // or display error message
11 } finally {
12     sess.close();
13 }
```

Arquitectura de Encarpetado



2. Hibernate

2.6. Tipo de Relaciones



Tipo de Relaciones

- Uno a uno (unidireccional)
- Uno a uno (bidireccional)
- Uno a muchos (desordenada)
- Uno a muchos (ordenada)
- Muchos a muchos
- Cascade.

Uno a Uno (Unidireccional)

La relación uno a uno en Hibernate consiste simplemente en que un objeto tenga una referencia a otro objeto de forma que al persistir el primer objeto también se persista el segundo.

La relación va a ser unidireccional es decir que la relación uno a uno va a ser en un único sentido.

Las dos tablas deben tener la misma clave primaria y de esa forma se establece la relación.

Uno a Uno (Unidireccional)

Profesor
+ Integer: id
+ String: nombre
+ String: ape1
+ String: ape2

Direccion
+ Integer: id
+ String: calle
+ Integer: numero
+ String: poblacion
+ String: provincia

<<Tabla>>
Profesor
INTEGER id
VARCHAR nombre
VARCHAR ape1
VARCHAR ape2

<<Tabla>>
Direccion
INTEGER id
VARCHAR calle
INTEGER numero
VARCHAR poblacion
VARCHAR provincia



```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <class name="ejemplo01.Profesor" >
        <id column="Id" name="id" type="integer"/>
        <property name="nombre" />
        <property name="ape1" />
        <property name="ape2" />
        <one-to-one name="direccion" cascade="all" />
    </class>
</hibernate-mapping>
  
```

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <class name="ejemplo01.Direccion" >
        <id column="Id" name="id" type="integer"/>
        <property name="calle"/>
        <property name="numero"/>
        <property name="poblacion"/>
        <property name="provincia"/>
    </class>
</hibernate-mapping>
  
```

Uno a Uno (Unidireccional)

El tag **<one-to-one>** se utiliza para definir una relación uno a uno entre las dos clases Java. En su forma más sencilla contiene sólamente dos atributos:

- **name:** Este atributo contiene el nombre de la propiedad Java con la referencia al otro objeto con el que forma la relación uno a uno.
- **cascade:** Este atributo indicará a hibernate cómo debe actuar cuando realicemos las operaciones de persistencia de guardar, borrar, leer, etc.

Uno a Uno (Bidireccional)

Esta relación es muy similar a la relación Uno a Uno unidireccional, pero en éste caso la relación va a ser bidireccional.

Las dos tablas deben tener la misma clave primaria y de esa forma se establece la relación.

Uno a Uno (Bidireccional)

Profesor
+ Integer: id
+ String: nombre
+ String: ape1
+ String: ape2

Direccion
+ Integer: id
+ String: calle
+ Integer: numero
+ String: poblacion
+ String: provincia

<<Tabla>>
Profesor
INTEGER id
VARCHAR nombre
VARCHAR ape1
VARCHAR ape2

<<Tabla>>
Direccion
INTEGER id
VARCHAR calle
INTEGER numero
VARCHAR poblacion
VARCHAR provincia

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class name="ejemplo01.Profesor" >
    <id column="Id" name="id" type="integer"/>
    <property name="nombre" />
    <property name="ape1" />
    <property name="ape2" />
    <one-to-one name="direccion" cascade="all" />
  </class>
</hibernate-mapping>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class name="ejemplo01.Direccion" >
    <id column="Id" name="id" type="integer"/>
    <property name="calle" />
    <property name="numero"/>
    <property name="poblacion"/>
    <property name="provincia"/>
    <one-to-one name="profesor" cascade="all" />
  </class>
</hibernate-mapping>
```

Explicación



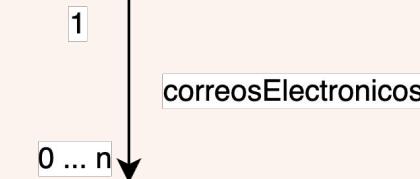
Uno a muchos (desordenada)

La relación uno a muchos consiste simplemente en que un objeto padre tenga una lista sin ordenar de otros objetos hijo de forma que al persistirse el objeto principal también se persista la lista de objetos hijo. Esta relación también suele llamarse *maestro-detalle* o *padre-hijo*.

En la tabla de origen contiene como clave ajena la clave primaria de la tabla destino y de esa forma se establece la relación uno a muchos.

Uno a muchos (desordenada)

Profesor	
+ Integer:	<code>id</code>
+ String:	<code>nombre</code>
+ String:	<code>ape1</code>
+ String:	<code>ape2</code>



CorreoElectronico	
+ Integer:	<code>idCorreo</code>
+ String:	<code>direccionCorreo</code>

<<Tabla>>	
Profesor	
INTEGER	<code>id</code>
VARCHAR	<code>nombre</code>
VARCHAR	<code>ape1</code>
VARCHAR	<code>ape2</code>

<<Tabla>>	
CorreoElectronico	
INTEGER	<code>id</code>
VARCHAR	<code>direccion_correo</code>
INTEGER	<code>id_profesor</code>

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <class name="ejemplo05.Profesor" >
        <id column="Id" name="id" type="integer"/>
        <property name="nombre" />
        <property name="ape1" />
        <property name="ape2" />
        <set name="correosElectonicos" cascade="all" inverse="true" >
            <key>
                <column name="idProfesor" />
            </key>
            <one-to-many class="ejemplo05.CorreoElectronico" />
        </set>
    </class>
</hibernate-mapping>
  
```

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <class name="ejemplo05.CorreoElectronico" >
        <id column="IdCorreo" name="idCorreo" type="integer"/>
        <property name="direccionCorreo" />
        <many-to-one name="profesor">
            <column name="idProfesor" />
        </many-to-one>
    </class>
</hibernate-mapping>
  
```

Uno a muchos (desordenada)

El tag **<set>** se utiliza para definir una relación uno a muchos desordenada entre las dos clases Java.

- **name:** Es el nombre de la propiedad Java del tipo Set en la cual se almacenan todos los objetos hijos.
- **cascade:** Este atributo indica que se realizan las mismas operaciones con el objeto padre que con los objetos hijos, es decir si uno se borra los otros también, etc. Su valor habitual es all.
- **inverse:** Este atributo se utiliza para minimizar las SQL's que Lanza Hibernate contra la Base de Datos. En este caso concreto debe establecerse a true para evitar una sentencia SQL de UPDATE por cada hijo.

Uno a muchos (desordenada)

Tags anidados

- **key:** Este tag contiene otro anidado llamado column con el atributo name que indica el nombre de una columna de la base de datos. Esta columna debe ser de la tabla hijo y ser el nombre de la clave ajena a la tabla padre.
- **one-to-many:** Este tag contiene el atributo class con el FQCN de la clase Java hija.
FQCN: full qualified class name of the object

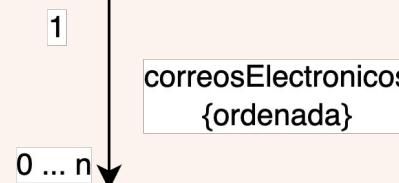
Uno a muchos (desordenada)

El tag **<many-to-one>** se utiliza para definir una relación muchos a uno entre las dos clases Java.

- **name:** Es el nombre de la propiedad Java que enlaza con el objeto padre.
- **column:** Este tag contiene el atributo name que indica el nombre de una columna de la base de datos. Esta columna debe ser de la tabla hijo y ser el nombre de la clave ajena a la tabla padre.

Uno a muchos (ordenada)

Profesor	
+ Integer:	id
+ String:	nombre
+ String:	ape1
+ String:	ape2



CorreoElectronico	
+ Integer:	idCorreo
+ String:	direccionCorreo

<<Tabla>>	
Profesor	
INTEGER	id
VARCHAR	nombre
VARCHAR	ape1
VARCHAR	ape2

<<Tabla>>	
CorreoElectronico	
INTEGER	id
VARCHAR	direccion_correo
INTEGER	id_profesor
INTEGER	idx

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
<class name="ejemplo05.Profesor" >
  <id column="Id" name="id" type="integer"/>
  <property name="nombre" />
  <property name="ape1" />
  <property name="ape2" />
  <list name="correosElectonicos" cascade="all" inverse="false" >
    <key>
      <column name="idProfesor" />
    </key>
    <list-index>
      <column name="Idx" />
    </list-index>
    <one-to-many class="ejemplo07.CorreoElectronico" />
  </list>
</class>
</hibernate-mapping>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
<class name="ejemplo05.CorreoElectronico" >
  <id column="IdCorreo" name="idCorreo" type="integer"/>
  <property name="direccionCorreo" />
  <many-to-one name="profesor">
    <column name="idProfesor" />
  </many-to-one>
</class>
</hibernate-mapping>
```

Uno a muchos (ordenada)

El tag **<list>** se utiliza para definir una relación uno a muchos entre las dos clases Java en las cuales hay un orden.

- **name:** Es el nombre de la propiedad Java del tipo List en la cual se almacenan todos los objetos hijos.
- **cascade:** Este atributo indica que se realizan las mismas operaciones con el objeto padre que con los objetos hijos, es decir si un se borra los otros también, etc. Su valor habitual es all.
- **inverse:** En este caso el atributo inverse debe tener el valor false ya que de esa forma se guardará en valor del orden de cada objeto. Si se estableciera a true no se guardaría el valor.

Uno a muchos (ordenada)

Tags anidados

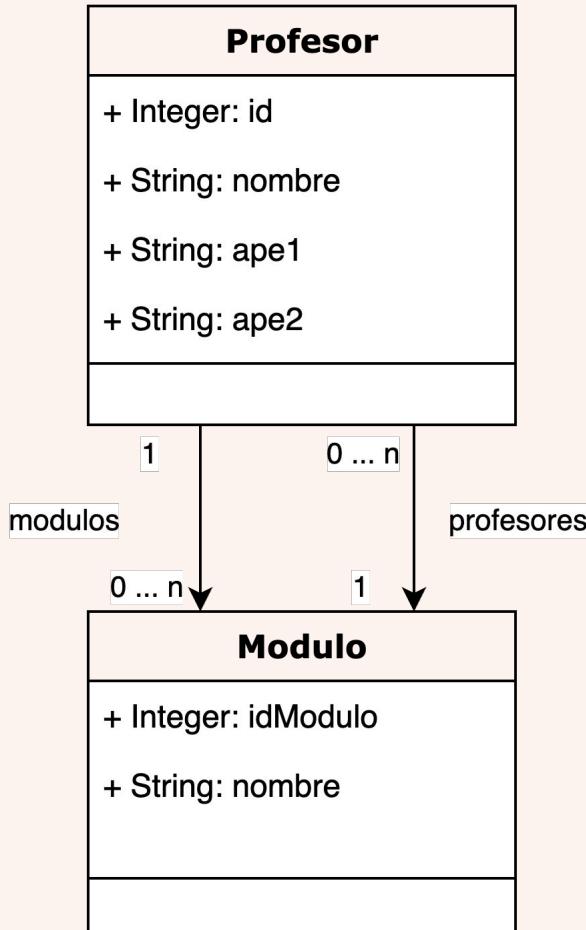
- **key:** Este tag contiene otro anidado llamado column con el atributo name que indica el nombre de una columna de la base de datos. Esta columna debe ser de la tabla hijo y ser el nombre de la clave ajena a la tabla padre.
- **list-index:** Este tag contiene otro anidado llamado column con el atributo name que indica el nombre de una columna de la base de datos. Esta columna debe ser de la tabla hijo y ser la columna donde se guarda el orden que ocupa dentro de la lista.
- **one-to-many:** Este tag contiene el atributo class con el FQCN de la clase Java hija.

Uno a muchos (ordenada)

El tag **<many-to-one>** se utiliza para definir una relación muchos a uno entre las dos clases Java.

- **name:** Es el nombre de la propiedad Java que enlaza con el objeto padre.
- **column:** Este tag contiene el atributo name que indica el nombre de una columna de la base de datos. Esta columna debe ser de la tabla hijo y ser el nombre de la clave ajena a la tabla padre.

Muchos a muchos



<<Tabla>>	
Profesor	
INTEGER id	
VARCHAR nombre	
VARCHAR ape1	
VARCHAR ape2	

1
0 ... n

<<Tabla>>	
ProfesorModulo	
INTEGER id_profesor	
INTEGER id_modulo	

0 ... n
1

<<Tabla>>	
Modulo	
INTEGER id	
VARCHAR nombre	

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
<class name="ejemplo09.Profesor" >
  <id column="Id" name="id" type="integer"/>
  <property name="nombre" />
  <property name="ape1" />
  <property name="ape2" />
  <set name="modulos" table="ProfesorModulo" cascade="all" inverse="true" >
    <key>
      <column name="idProfesor" />
    </key>
    <many-to-many column="IdModulo" class="ejemplo09.Modulo" />
  </set>
</class>
</hibernate-mapping>
  
```

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
<class name="ejemplo09.Modulo" >
  <id column="IdModulo" name="idModulo" type="integer"/>
  <property name="nombre" />
  <set name="profesores" table="ProfesorModulo" cascade="all" inverse="false" >
    <key>
      <column name="idModulo" />
    </key>
    <many-to-many column="IdProfesor" class="ejemplo09.Profesor" />
  </set>
</class>
</hibernate-mapping>
  
```

Muchos a muchos

El tag **<set>** se utiliza para definir una lista desordenada entre las dos clases Java.

- **name:** Es el nombre de la propiedad Java del tipo Set en la cual se almacenan todos los objetos relacionados.
- **table:** Es el nombre de la tabla de la base de datos que contiene la relación muchos a muchos.
- **cascade:** Como ya hemos explicado en anteriores lecciones, este atributo indica que se realizan las mismas operaciones con el objeto principal que con los objetos relacionados, es decir si uno se borra los otros también, etc. Su valor habitual es all.
- **inverse:** En el caso de las relaciones muchos a muchos es necesario poner un lado de la relación con el valor true y el otro con el valor false.

Muchos a muchos

Tags anidados

- **key:** Este tag contiene otro anidado llamado column con el atributo name, el cual contiene el nombre de una columna de la base de datos. Esta columna debe ser una de la tabla de la relación muchos a muchos y ser el nombre de la columna que contiene la clave ajena de tabla que estamos persistiendo.
- **many-to-many:** Este tag contiene el atributo class con el FQCN de la clase Java con la que se establece la relación.

Muchos a muchos

El tag **<set>** se utiliza para definir una lista desordenada entre las dos clases Java.

- **inverse:** En el caso de las relaciones muchos a muchos es necesario poner un lado de la relación con el valor true y el otro con el valor false.

Si ambos valores son true no se realizará ninguna inserción en la tabla intermedia y si ambos valores son false se realizará dos veces la inserción en dicha tabla dando un error de clave primaria duplicada.

Es decir, que en este caso inverse controla cuándo se realiza la inserción en la tabla intermedia si bien al guardarla en la tabla de izquierda o en la tabla de derecha.

Cascade

¿Cuál es el significado del atributo cascade? El significado es indicar qué debe hacer hibernate con las clases relacionadas cuando realizamos alguna acción con la clase principal.

Es decir, si borramos la clase principal, ¿debería borrarse la clase relacionada? La respuesta a ésta y otras preguntas depende de nuestro modelo de clases, por ello existen 11 valores distintos y deberemos elegir entre todos ellos.

Valores Cascade

Valor	Descripción
none	No se realiza ninguna acción en los objetos relacionados al hacerlo sobre el principal
save-update	Si se inserta o actualiza el objeto principal también se realizará la inserción o actualización en los objetos relacionados.
delete	Si se borra el objeto principal también se realizará el borrado en los objetos relacionados.
evict	Si se llama al método <code>JSession.evict(Object objeto)</code> para el objeto principal también se llamará para los objetos relacionados.
lock	Si se llama al método <code>JLockRequest.lock(Object objeto)</code> ¹⁰⁾ para el objeto principal también se llamará para los objetos relacionados.
merge	Si se llama al método <code>JSession.merge(Object objeto)</code> con el objeto principal también se llamará para los objetos relacionados.
refresh	Si se llama al método <code>JSession.refresh(Object objeto)</code> para el objeto principal también se llamará para los objetos relacionados.
replicate	Si se llama al método <code>JSession.replicate(Object objeto, ReplicationMode replicationMode)</code> para el objeto principal también se llamará para los objetos relacionados.
all	Si se realiza cualquiera de las anteriores acciones sobre el objeto principal también se realizará sobre los objetos relacionados.
delete-orphan	Este atributo sólo se usa si el objeto relacionado es una colección. Indica que si en la colección del objeto principal eliminamos un elemento , al persistir el objeto principal deberemos borrar de la base de datos el elemento de la colección que habíamos eliminado.
all-delete-orphan	Es la unión de los atributos all y delete-orphan ¹¹⁾

Claves Primarias

¿Qué propiedades debe tener una buena clave primaria?

- Debe ser única
- No puede ser null
- Nunca debe cambiar
- Debe ser una única columna
- Debe ser rápida de generar

Claves Primarias

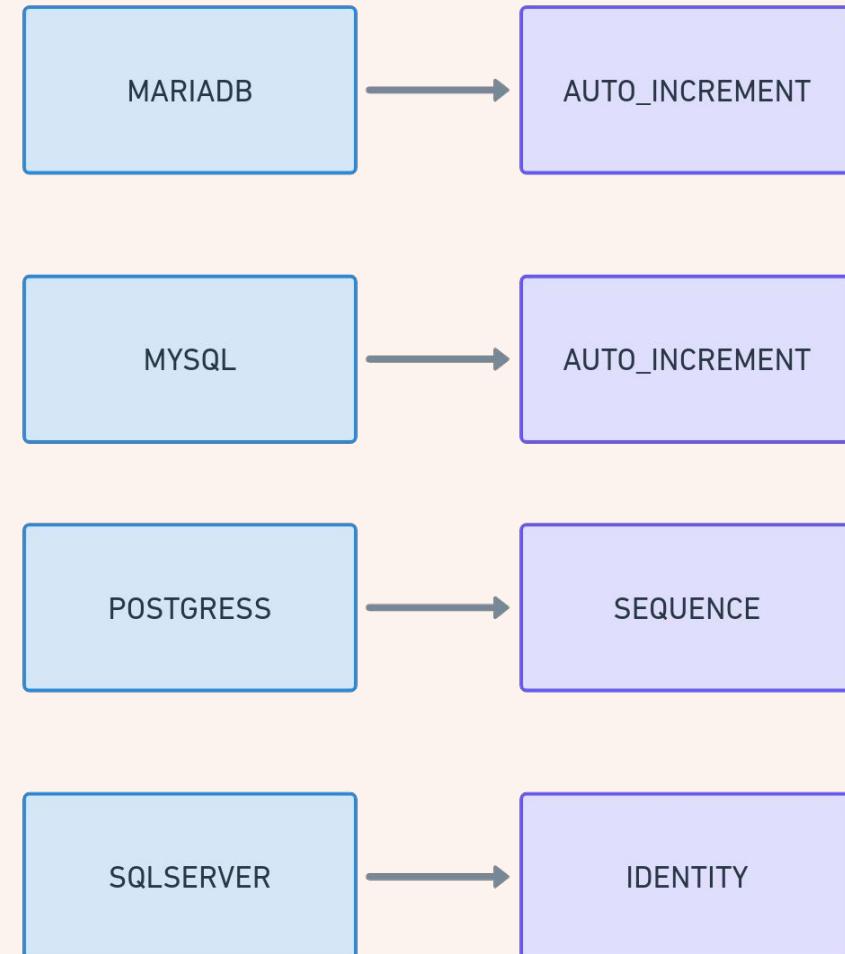
El tag **<generator>** se utiliza para indicar que la clave primaria será generada por el propio Hibernate en vez de asignarla directamente el usuario.

- **class:** Este atributo indica el método que usará Hibernate para calcular la clave primaria.

Claves Primarias

Valor	Descripción
native	Hibernate usará alguno de los siguientes métodos dependiendo de la base de datos. De esta forma si cambiamos de base de datos se seguirá usando la mejor forma de generar la clave primaria
identity	Hibernate usará el valor de la columna de tipo autoincremento. Es decir, que al insertar la fila, la base de datos le asignará el valor. La columna de base de datos debe ser de tipo autonumérico
sequence	Se utiliza una secuencia como las que existen en Oracle o PostgreSQL , no es compatible con MySQL. La columna de base de datos debe ser de tipo numérico
increment	Se lanza una consulta SELECT MAX() contra la columna de la base de datos y se obtiene el valor de la última clave primaria, incrementando el nº en 1.La columna de base de datos debe ser de tipo numérico
uuid.hex	Hibernate genera un identificador único como un String. Se usa para generar claves primarias únicas entre distintas bases de datos.La columna de base de datos debe ser de tipo alfanumérico.
guid	Hibernate genera un identificador único como un String pero usando las funciones que provee SQL Server y MySQL. Se usa para generar claves primarias únicas entre distintas bases de datos.La columna de base de datos debe ser de tipo alfanumérico.
foreign	Se usará el valor de otro objeto como la clave primaria. Un uso de ello es en relaciones uno a uno donde el segundo objeto debe tener la misma clave primaria que el primer objeto a guardar.

Claves Primarias



Claves Primarias

Al usar el método sequence es necesario indicar el nombre de la secuencia que va a usar Hibernate para obtener el valor.

```
<id column="Id" name="id" type="Integer">
    <generator class="sequence">
        <param name="sequence">secuencia_idProfesor</param>
    </generator?
</id>
```

Claves Primarias

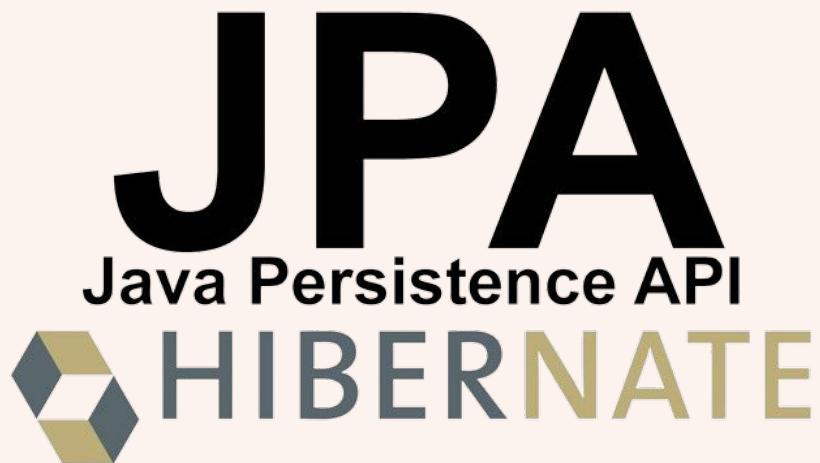
Al usar el método foreign es necesario indicar la propiedad del otro objeto del que se obtendrá su clave primaria.

Suponiendo que hay una relación uno a uno entre la tabla izquierda y tabla derecha como una relación uno a uno (bidireccional) la clave primaria de tabla derecha sería la misma que la de tabla izquierda.

```
<id column="Id" name="id" type="Integer">
    <generator class="foreign">
        <param name="property">profesor</param>
    </generator?
</id>
```

3. Hibernate Query

3.1. Hibernate Query



Hibernate Query

Hibernate tiene el objeto Query que nos da acceso a todas las funcionalidades para poder leer objetos desde la base de datos.

Lanzar una consulta con Hibernate es bastante simple. Usando la session llamamos al método `createQuery(String queryString)` con la consulta en formato HQL y nos retorna un objeto.

- Despues, sobre el objeto Query llamamos al método `list()` que nos retorna una lista de los objetos que ha retornado.

```
1 Query query = session.createQuery("SELECT p FROM Profesor p");
2 List<Profesor> profesores = query.list();
3 for (Profesor profesor : profesores) {
4     System.out.println(profesor.toString())
5 }
```

Hibernate Query

Lista de Objetos

- El método `list()` nos retorna una lista con todos los objetos que ha retornado la consulta. En caso de que no se encuentre ningún resultado se retornará una lista sin ningún elemento.

```
1 Query query = session.createQuery("SELECT p FROM Profesor p");
2 List<Profesor> profesores = query.list();
3 for (Profesor profesor : profesores) {
4     System.out.println(profesor.toString())
5 }
```

Hibernate Query

Lista de Array de objetos

- HQL también permite que las consultas retornen datos escalares en vez de clases completas.

```
SELECT p.id, p.nombre FROM Profesor p
```

- En estos casos el método `list()` retorna una lista con una Array de objetos con tantos elementos como propiedades hayamos puesto en el SELECT.

```
1 Query query = session.createQuery("SELECT p.id, p.nombre FROM Profesor p");
2 List<Object[]> listDatos = query.list();
3 for (Object[] datos : listDatos) {
4     System.out.println(datos[0] + "--" + datos[1]);
5 }
```

Hibernate Query

Hay otro caso cuando hay una única columna en el SELECT de datos escalares. Es ese caso, como el array a retornar dentro de la lista solo tendría un elemento, no se retorna una lista de arrays List<Object []> sino únicamente una lista de elementos List<Object>.

```
1 Query query = session.createQuery("SELECT p.nombre FROM Profesor p");
2 List<Object> listDatos = query.list();
3 for (Object datos : listDatos) {
4     System.out.println(datos;
5 }
```

Hibernate Query

uniqueResult

- En muchas ocasiones una consulta únicamente retornará cero o un resultado. En ese caso es poco práctico que nos retorne una lista con un único elemento. Para facilitarnos dicha tarea Hibernate dispone del método `uniqueResult()`.
- Este método retornará directamente el único objeto que ha obtenido la consulta. En caso de que no encuentre ninguno se retornará null.

```
1 Profesor profesor = (Profesor) session.createQuery("SELECT p FROM Profesor p WHERE id = 1001").uniqueResult();
2 System.out.println("Profesor con ID 1001 = " + profesor.getNombre());
```

Consultas con nombre

En cualquier libro sobre arquitectura del software siempre se indica que las consultas a la base de datos no deberían escribirse directamente en el código sino que deberían estar en un fichero externo para que puedan modificarse fácilmente.

Hibernate provee una funcionalidad para hacer ésto mismo de una forma sencilla. En cualquier fichero de mapeo de Hibernate se puede incluir el tag **<query>** con la consulta HQL que deseamos lanzar.

Consultas con nombre

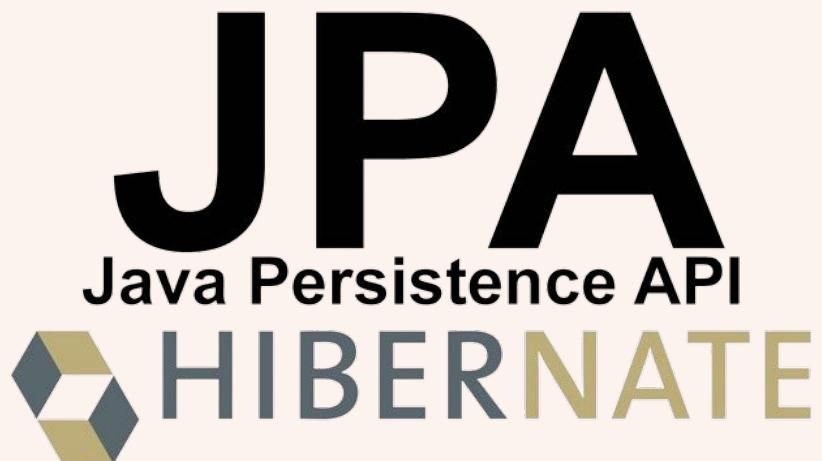
Tag <query>

- **name:** Este atributo define el nombre de la consulta. Es el nombre que posteriormente usaremos desde el código Java para acceder a la consulta.
- **contenido:** El contenido del tag <query> es la consulta en formato HQL que ejecutará Hibernate.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <class name="ejemplo01.Profesor" >
        <id column="Id" name="id" type="integer"/>
        <property name="nombre" />
        <property name="ape1" />
        <property name="ape2" />
    </class>
    <query name="findAllProfesores"><![CDATA[
        SELECT p FROM Profesor p
    ]]></query>
</hibernate-mapping>
```

4. Hibernate Query Language

4.1. Hibernate Query Language



Hibernate Query Language (HQL)

HQL es el lenguaje de consultas que usa Hibernate para obtener los objetos desde la base de datos. Su principal particularidad es que las consultas se realizan sobre los objetos java que forman nuestro modelo de negocio, es decir, las entidades que se persisten en Hibernate.

Hibernate Query Language (HQL)

Ésto hace que HQL tenga las siguientes características:

- Los tipos de datos son los de Java.
- Las consultas son independientes del lenguaje de SQL específico de la base de datos.
- Las consultas son independientes del modelo de tablas de la base de datos.
- Es posible tratar con las colecciones de Java.
- Es posible navegar entre los distintos objetos en la propia consulta.

Hibernate Query Language (HQL)

En Hibernate las consultas HQL se lanzan (o se ejecutan) sobre el modelo de entidades que hemos definido en Hibernate, esto es, sobre nuestras clases de negocio.

Nuestro modelo de tablas en HQL son las clases Java y NO las tablas de la base de datos. Es decir que cuando hagamos “SELECT columna FROM nombreTabla”, el “nombreTabla” será una clase Java y “columna” será una propiedad Java de dicha clase y nunca una tabla de la base de datos ni una columna de una tabla.

Hibernate Query Language (HQL)

```
SELECT c FROM Ciclo c ORDER BY nombre
```

¿Qué diferencias podemos ver entre HQL y SQL?

- Ciclo hace referencia a la clase Ciclo y NO a la tabla CicloFormativo. Nótese que la clase Java y la tabla tienen distinto nombre.
- Es necesario definir el alias c de la clase Java Ciclo.
- Tras la palabra SELECT se usa el alias en vez del “*”.
- Al ordenar los objetos se usa la propiedad nombre de la clase Ciclo en vez de la columna nombreCiclo de la tabla CicloFormativo.
- Hibernate soporta no incluir la parte del SELECT en la consulta HQL, quedando entonces la consulta de la siguiente forma:
 - FROM Ciclo

Hibernate Query Language (HQL)

Respecto a la sensibilidad de las mayúsculas y minúsculas

- Las palabras clave del lenguaje NO son sensibles a las mayúsculas o minúsculas.
 - `select count (*) from Ciclo`
 - `SELECT COUNT (*) FROM Ciclo`
- El nombre de las clases Java y sus propiedades SI son sensibles a las mayúsculas o minúsculas.
 - `SELECT c.nombre FROM Ciclo c WHERE nombre='Desarrollo de aplicaciones Web'`
- Al realizar comparaciones con los valores de las propiedades, éstas NO son sensibles a las mayúsculas o minúsculas.

Hibernate Query Language (HQL)

Filtrado

- Al igual que en SQL en HQL también podemos filtrar los resultados mediante la cláusula WHERE. La forma de usarla es muy parecida a SQL.
 - `SELECT p FROM Profesor p WHERE nombre = 'ISABEL' AND ape1 <> 'ORELLANA'`
- Al igual que con el nombre de la clase, el nombre de los campos del WHERE siempre hace referencia a las propiedades Java y nunca a los nombres de las columnas. De esa forma seguimos independizando nuestro código Java de la estructura de la base de datos.

Hibernate Query Language (HQL)

Para comparar los datos en una expresión se pueden usar las siguientes Operadores:

- **Signo igual "="**: La expresión será verdadera si los dos datos son iguales. En caso de comparar texto, la comparación no es sensible a mayúsculas o minúsculas.
- **Signo mayor que ">"**: La expresión será verdadera si el dato de la izquierda es mayor que el de la derecha.
- **Signo mayor que ">="**: La expresión será verdadera si el dato de la izquierda es mayor o igual que el de la derecha.
- **Signo mayor que "<"**: La expresión será verdadera si el dato de la izquierda es menor que el de la derecha.

Hibernate Query Language (HQL)

- **Signo mayor que "<=":** La expresión será verdadera si el dato de la izquierda es menor o igual que el de la derecha.
- **Signo desigual "<>":** La expresión será verdadera si el dato de la izquierda es distinto al de la derecha.
- **Signo desigual "!=":** La expresión será verdadera si el dato de la izquierda es distinto al de la derecha.
- **Operador "between":** La expresión será verdadera si el dato de la izquierda está dentro del rango de la derecha.
 - `SELECT tb FROM TiposBasicos tb WHERE inte BETWEEN 1 AND 10`

Hibernate Query Language (HQL)

- **Operador "in":** La expresión será verdadera si el dato de la izquierda está dentro de la lista de valores de la derecha.
 - `SELECT tb FROM TiposBasicos tb WHERE inte IN (1,3,5,7)`
- **Operador "like":** La expresión será verdadera si el dato de la izquierda coincide con el patrón de la derecha. Se utilizan los mismos signos que en SQL "%" y "_" .
 - `SELECT tb FROM TiposBasicos tb WHERE stri LIKE 'H_la%`
- **Operador "not":** Niega el resultado de una expresión.
- **Expresión "is null":** Comprueba si el dato de la izquierda es null.
 - `SELECT tb FROM TiposBasicos tb WHERE dataDate IS NULL`

Hibernate Query Language (HQL)

Operadores Lógicos

- Se puede hacer uso de los típicos operadores lógicos como en SQL:
 - AND
 - OR
 - NOT
- `SELECT p FROM Profesor p WHERE nombre = 'ANTONIO' AND (apel = 'LARA' OR ape2 = 'RUBIO')`

Hibernate Query Language (HQL)

Operadores Aritméticos

- Se puede hacer uso de los típicos operadores aritméticos:
 - Suma +
 - Resta –
 - Multiplicación *
 - División /
- `SELECT tb FROM TiposBasicos tb WHERE (((inte+1)*4)-10)/2=1`

Hibernate Query Language (HQL)

Funciones de agregación

- Las funciones de agregación que soporta HQL son:
 - **AVG()**: Calcula el valor medio de todos los datos.
 - **SUM()**: Calcula la suma de todos los datos.
 - **MIN()**: Calcula el valor mínimo de todos los datos.
 - **MAX()**: Calcula el valor máximo de todos los datos.
 - **COUNT()**: Cuenta el nº de datos.
- ```
SELECT AVG(c.horas), SUM(c.horas), MIN(c.horas), MAX(c.horas),
COUNT(*) FROM Ciclo c
```

# Hibernate Query Language (HQL)

## Funciones sobre escalares

- Algunas de las funciones que soporta HQL sobre datos escalares son:
  - **UPPER(s)**: Transforma un texto a mayúsculas.
  - **LOWER(s)**: Transforma un texto a minúsculas.
  - **CONCAT(s1, s2)**: Concatena dos textos
  - **TRIM(s)**: Elimina los espacio iniciales y finales de un texto.
  - **SUBSTRING(s, offset, length)**: Retorna un substring de un texto. El offset empieza a contar desde 1 y no desde 0.
  - **LENGTH(s)**: Calcula la longitud de un texto.
  - **ABS(n)**: Calcula el valor absoluto de un número.
  - **SQRT(n)**: Calcula la raíz cuadrada del número
  - **Operador "||"** : Permite concatenar texto.
- ```
SELECT p.nombre || ' ' || p.apel1 || ' ' || p.apel2 FROM Profesor p WHERE id=1001
```

Hibernate Query Language (HQL)

Ordenación

- Como en SQL también es posible ordenar los resultados usando ORDER BY. Su funcionamiento es como en SQL.
 - `SELECT p FROM Profesor p ORDER BY nombre ASC, ape1 DESC`
- Las palabras ASC y DESC son opcionales al igual que en SQL.
- El uso de funciones escalares y funciones de agrupamiento en la cláusula ORDER BY sólo es soportado por Hibernate si es soportado por el lenguaje de SQL de la base de datos sobre la que se está ejecutando.
- No se permite el uso de expresiones aritméticas en la cláusula ORDER BY.

Hibernate Query Language (HQL)

Agrupaciones

- Al igual que en SQL se pueden realizar agrupaciones mediante las palabras claves GROUP BY y HAVING
 - `SELECT nombre, count(nombre) FROM Profesor p GROUP BY nombre HAVING count(nombre)>1 ORDER BY count(nombre)`
- El uso de funciones escalares y funciones de agrupamiento en la cláusula HAVING sólo es soportado por Hibernate si es soportado por el lenguaje de SQL de la base de datos sobre la que se está ejecutando.
- No se permite el uso de expresiones aritméticas en la cláusula GROUP BY.

Hibernate Query Language (HQL)

Subconsultas

- HQL también soporta subconsultas como en SQL.
 - ```
SELECT c.nombre, c.horas FROM Ciclo c WHERE c.horas > (SELECT AVG(c2.horas) FROM Ciclo c2)
```

# Hibernate Query Language (HQL)

## Parámetros

- **Posicional:** Esta forma de definir parámetros es muy similar a la que usa SQL. Se basa en definir cada parámetro como un interrogante "?". Posteriormente estableceremos a la clase Query el valor de cada uno de los parámetros.

```
...
1 String nombre = "ISIDRO";
2 String ape1 = "CORTINA";
3 String ape2 = "GARCIA";
4 Query query = session.createQuery("SELECT p FROM Profesor p WHERE nombre = ? AND ape1 = ? AND ape2 = ?");
5 query.setString(0, nombre);
6 query.setString(1, ape1);
7 query.setString(2, ape2);
8 List<Profesor> profesores = query.list();
9 for (Profesor profesor : profesores) {
10 System.out.println(profesor.toString());
11 }
```

# Hibernate Query Language (HQL)

## Por nombre

- En este caso los parámetros se definen como nombre precedidos de dos puntos ":". Esto hace que el código sea más legible y menos propenso a error.

```
1 String nombre = "ISIDRO";
2 String ape1 = "CORTINA";
3 String ape2 = "GARCIA";
4 Query query = session.createQuery("SELECT p FROM Profesor p WHERE nombre = :nombre AND ape1 = :ape1 AND ape2 =:ape2");
5 query.setString("nombre", nombre);
6 query.setString("ape1", ape1);
7 query.setString("ape2", ape2);
8 List<Profesor> profesores = query.list();
9 for (Profesor profesor : profesores) {
10 System.out.println(profesor.toString());
11 }
```

# Hibernate Query Language (HQL)

## setParameter

- Existe otra forma de asignar valores a los parámetros que es independiente del tipo, según si el parámetro es por índice o por nombre. Los métodos son:
  - `setParameter(int position, Object val)`
  - `setParameter(String name, Object val)`

```
...
1 String nombre = "ISIDRO";
2 String ape1 = "CORTINA";
3 String ape2 = "GARCIA";
4 Query query = session.createQuery("SELECT p FROM Profesor p WHERE nombre = :nombre AND ape1 = :ape1 AND ape2 =:ape2");
5 query.setParameter("nombre", nombre);
6 query.setParameter("ape1", ape1);
7 query.setParameter("ape2", ape2);
8 List<Profesor> profesores = query.list();
9 for (Profesor profesor : profesores) {
10 System.out.println(profesor.toString());
11 }
```

# Hibernate Query Language (HQL)

## Navegación por propiedades

- Una característica de HQL es la posibilidad de navegar por las propiedades.
  - En HQL si referenciamos una propiedad de un clase Java, esta propiedad puede no ser un valor escalar sino una referencia a otro objeto Java, que a su vez tiene más propiedades, lo que a su vez nos puede llevar hasta otro objeto Java y así sucesivamente.
  - `SELECT p.nombre.apel FROM Profesor p`

# Hibernate Query Language (HQL)

SELECT p.nombre.apel, p.direccion.municipio.nombre FROM Profesor p

```
1 Profesor profesor = ;
2 String nombreMunicipio = profesor.getDireccion().getMunicipio().getNombre();
```

```
SELECT
 profesor0_apel AS col_0_0_,
 municipio2_.NombreMunicipio AS col_1_0_
FROM
 Profesor profesor0_,
 Direccion direccion1_,
 Municipios municipio2_
WHERE
 profesor0_.Id = direccion1_.Id AND
 direccion1_.idMunicipio = municipio2_.idMunicipio
```

# Hibernate Query Language (HQL)

## Colecciones

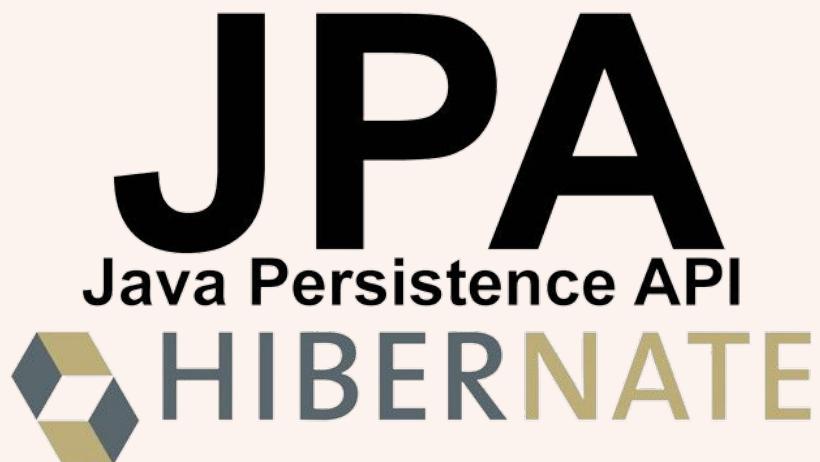
- Otra característica de HQL es la posibilidad de tratar con colecciones dentro de la propia consulta.
  - **SIZE(c)**: Obtiene el nº de elementos de una colección.
    - ```
SELECT p.nombre.apel, SIZE(p.correosElectronicos) FROM Profesor p GROUP BY p.nombre.apel
```
 - **c IS EMPTY**: Comprueba si una colección no tiene elementos.
 - ```
SELECT p.nombre FROM Profesor p WHERE p.correosElectronicos IS EMPTY
```
  - **IS NOT EMPTY c**: Comprueba si una colección sí tiene elementos.
    - ```
SELECT p.nombre FROM Profesor p WHERE p.correosElectronicos IS NOT EMPTY
```

Hibernate Query Language (HQL)

- **MINELEMENT(c)**: Retorna el menor de los elementos de la colección. Para averiguar cual es el menor siempre usa el campo que hace de id.
- **MAXELEMENT(c)**: Retorna el mayor de los elementos de la colección. Para averiguar cual es el mayor siempre usa el campo que hace de id.

5. Hibernate Criteria

5.1. Hibernate Criteria



Criteria

Hibernate proporciona formas alternativas de manipular objetos y, a su vez, datos disponibles en tablas RDBMS. Uno de los métodos es Criteria API, que le permite crear un objeto de consulta de criterios mediante programación donde puede aplicar reglas de filtrado y condiciones lógicas.

La interfaz Hibernate Session proporciona el método `createCriteria()`, que se puede utilizar para crear un objeto Criteria que devuelve instancias de la clase del objeto de persistencia cuando su aplicación ejecuta una consulta de criterios.

```
1 Criteria cr = session.createCriteria(Employee.class);
2 List results = cr.list();
```

Restricciones con Criteria

Puede utilizar el método `add()` disponible para el objeto `Criteria` para agregar restricción para una consulta de criterios. El siguiente es el ejemplo para agregar una restricción para devolver los registros con salario igual a 2000.

```
1 Criteria cr = session.createCriteria(Employee.class);
2 cr.add(Restrictions.eq("salary", 2000));
3 List results = cr.list();
```

```
1 Criteria cr = session.createCriteria(Employee.class);
2 //To get records having salary more than 2000
3 cr.add(Restrictions.gt("salary", 2000));
4 //To get records having salary less than 2000
5 cr.add(Restrictions.lt("salary", 2000));
6 //To get records having firstName starting with zara
7 cr.add(Restrictions.like("firstName", "zara%"));
8 //Case sensitive form of the above restriction
9 cr.add(Restrictions.ilike("firstName", "zara%"));
10 //To get records having salary in between 1000 and 2000
11 cr.add(Restrictions.between("salary", 1000, 2000));
12 //To check if the given property is null
13 cr.add(Restrictions.isNull("salary"));
14 //To check if the given property is not null
15 cr.add(Restrictions.isNotNull("salary"));
16 //To check if the given property is empty
17 cr.add(Restrictions.isEmpty("salary"));
18 //To check if the given property is not empty
19 cr.add(Restrictions.isNotEmpty("salary"));
```

Contacto

Mtro. Alfonso Gregorio Rivero Duarte
Senior Data Manager - CBRE

devil861109@gmail.com

Tels: (+52) 55 289970 69

Redes sociales:



<https://www.linkedin.com/in/alfonso-gregorio-rivero-duarte-139a9225/>