



Acceso a Datos

Tema 3 Bases de Datos Relacionales

José Luis González Sánchez



Contenidos

¿Qué voy a aprender?



Contenidos

1. Bases de datos Relacionales
2. El Desfase Objeto-Relacional
3. Acceso a Base de Datos Relacionales
4. CRUD con JDBC
5. ORMS
6. JPA e Hibernate
7. CRUD con JPA
8. Extra. Cuestiones de Arquitectura y Diseño. Patrones

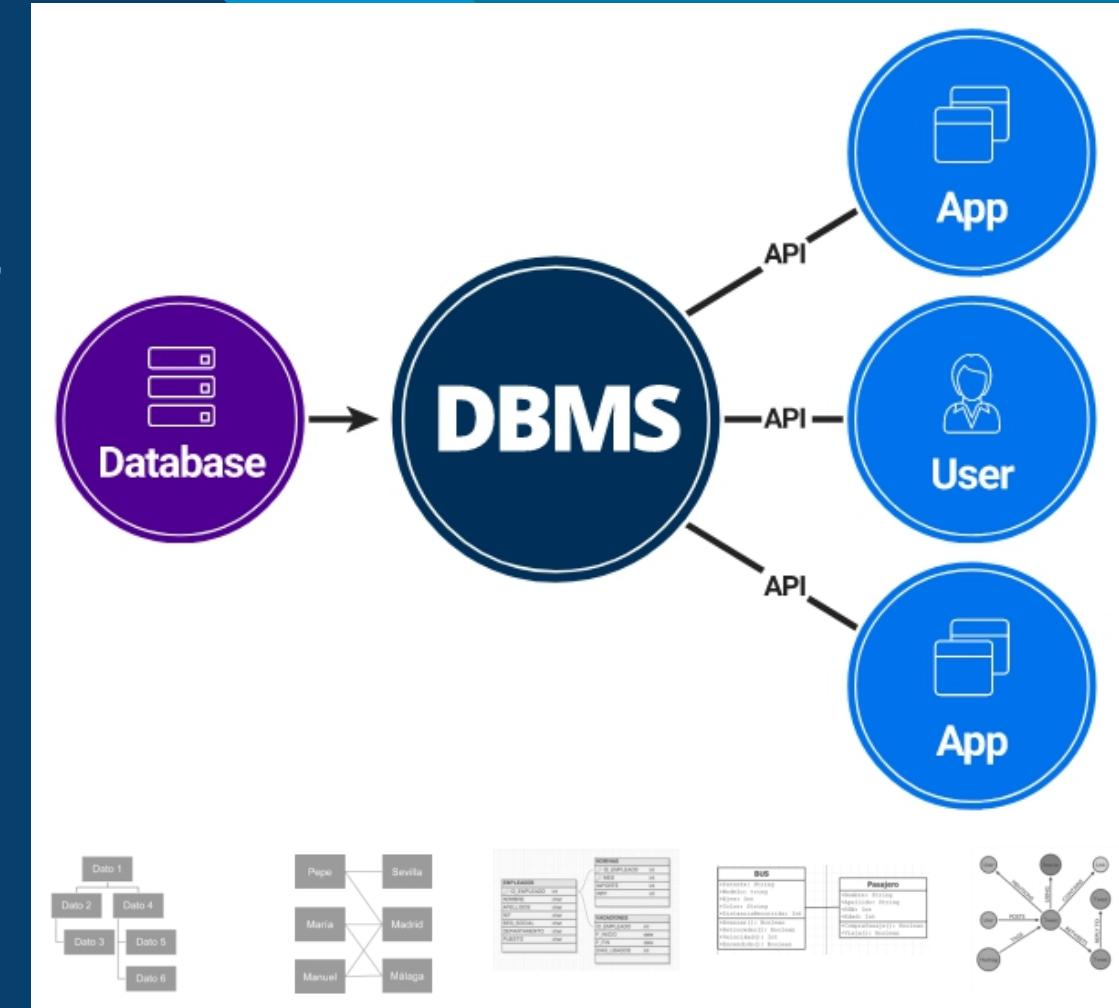
Bases de Datos Relacionales

La información entre tablas, filas y columnas



Bases de Datos Relacionales

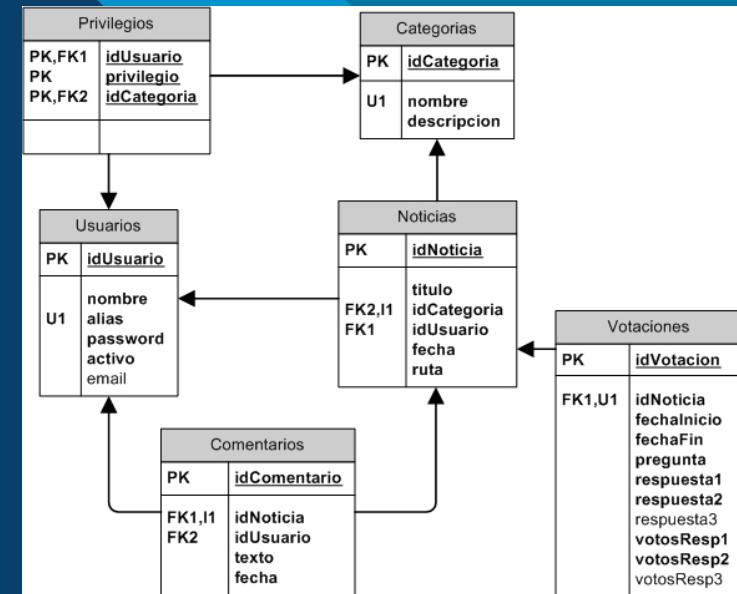
- Desde que ser humano existe, siempre hemos tenido la necesidad de almacenar la información, ya sea desarrollando la escritura, luego el papel, imprenta, libros, cintas perforadas, disco duro, etc.
- A la misma vez que almacenamos la información debemos organizarla y procesarla ya sea en ficheros de acceso aleatorio, indexados o usando bases de datos.
- Tenemos distintos DBMS según si esquema lógico:
 - Jerárquicos: En árbol, estructura padre/hijo.
 - En Red: Nodos y enlaces. Manejo complejo
 - Relacional. Uso de tablas como única estructura
 - Orientados a objetos: Origen en la POO, para uso concreto.
 - NoSQL, Not Only SQL, Orientado a documentos, no todo es relacional.





Bases de Datos Relacionales

- La base de datos relacional (BDR) es un tipo de base de datos (BD) que cumple con el modelo relacional (el modelo más utilizado actualmente para implementar las BD ya planificadas). Tras ser postuladas sus bases en 1970 por Edgar Frank Codd, de los laboratorios IBM en San José (California), no tardó en consolidarse como un nuevo paradigma en los modelos de base de datos.
- Virtualmente, todos los sistemas de bases de datos relacionales utilizan SQL (Structured Query Language) para consultar y mantener la base de datos.
 - Una base de datos se compone de varias tablas, denominadas relaciones.
 - No pueden existir dos tablas con el mismo nombre ni registro.
 - Cada tabla es a su vez un conjunto de campos (columnas) y registros (filas).
 - La relación entre una tabla padre y un hijo se lleva a cabo por medio de las llaves primarias y llaves foráneas (o ajenas).
 - Las llaves primarias son la clave principal de un registro dentro de una tabla y estas deben cumplir con la integridad de datos.
 - Las llaves ajenas se colocan en la tabla hija, contienen el mismo valor que la llave primaria del registro padre; por medio de estas se hacen las formas relacional



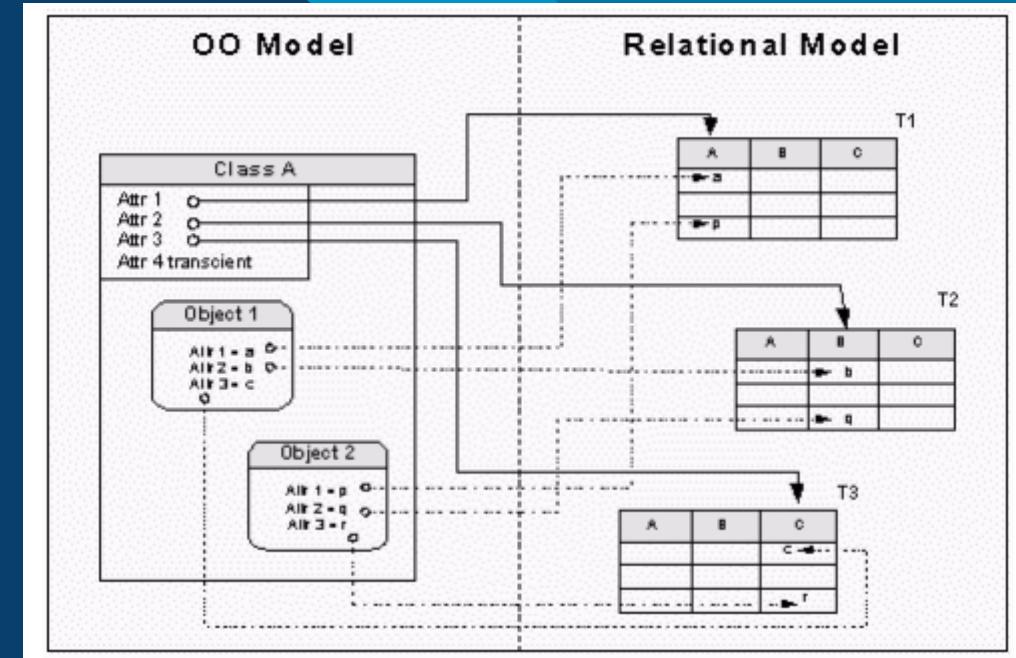
Desfase Objeto- Relacional

De tablas, filas y columnas a objetos



Desfase Objeto-Relacional

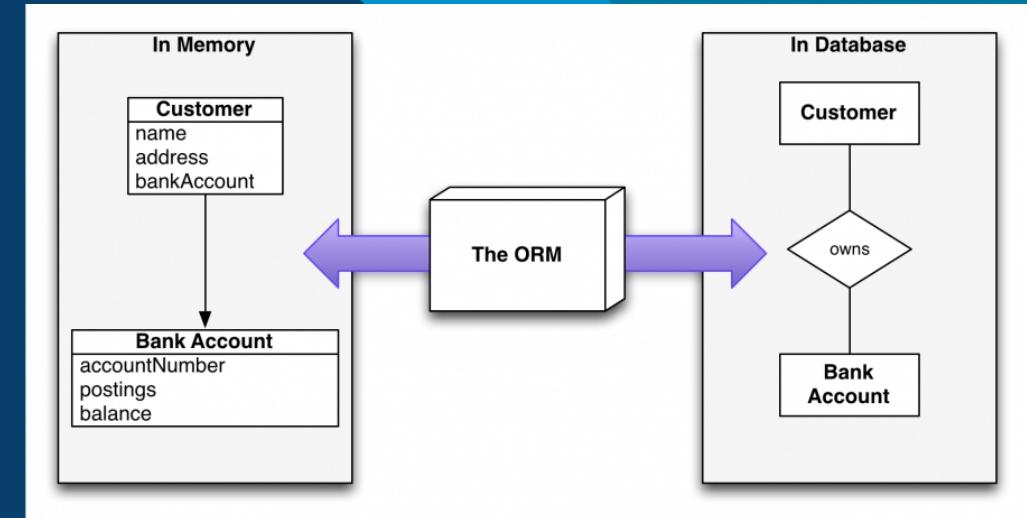
- También conocido como **impedancia objeto-relacional**, se trata de la diferencia existente entre la programación orientada a objetos y las bases de datos relacionales. Nos referimos a:
 - Lenguaje: El desarrollador debe conocer dos lenguajes: el de programación y el de la base de datos.
 - Tipos de datos: En la POO el tipo de datos es más complejo.
 - Paradigma de programación: Tiene que haber una traducción del modelo entidad-relación de la base de datos al modelo orientado a objetos del lenguaje de programación.
- Es decir, el modelo relacional trata de relaciones y conjuntos, mientras que POO trata los datos como objetos y asociaciones entre los mismos. Es decir al ahora de programar con BBDD Relacionales, debemos:
 - 1.Abrir una conexión.
 - 2.Crear sentencia SQL
 - 3.Copiar las propiedades del objeto y la sentencia y lanzar la consulta, o obtener de la consulta la información y **mapearla en el objeto**.





Desfase Objeto-Relacional. Mapeo Objeto Relacional

- Al trabajar con lenguajes orientados a objetos y bases de datos relacionales tenemos que estar continuamente descomponiendo los objetos para escribir la sentencia SQL para insertar, modificar o eliminar, o bien recomponer todos los atributos para formar el objeto cuando leamos algo de la base de datos.
- A este conjunto de técnicas que pueden desarrollarse de manera manual por el desarrollador o con apoyo de algún software se conoce como mapeo objeto-relacional.
- Aunque pueda parecer sencillo, es un problema si la base de datos cambia en estructura o tipo de datos, o simplemente el modelo de dominio de nuestro problema muta con el paso del tiempo, lo que puede provocar bastante cambios en cascada en nuestro código.



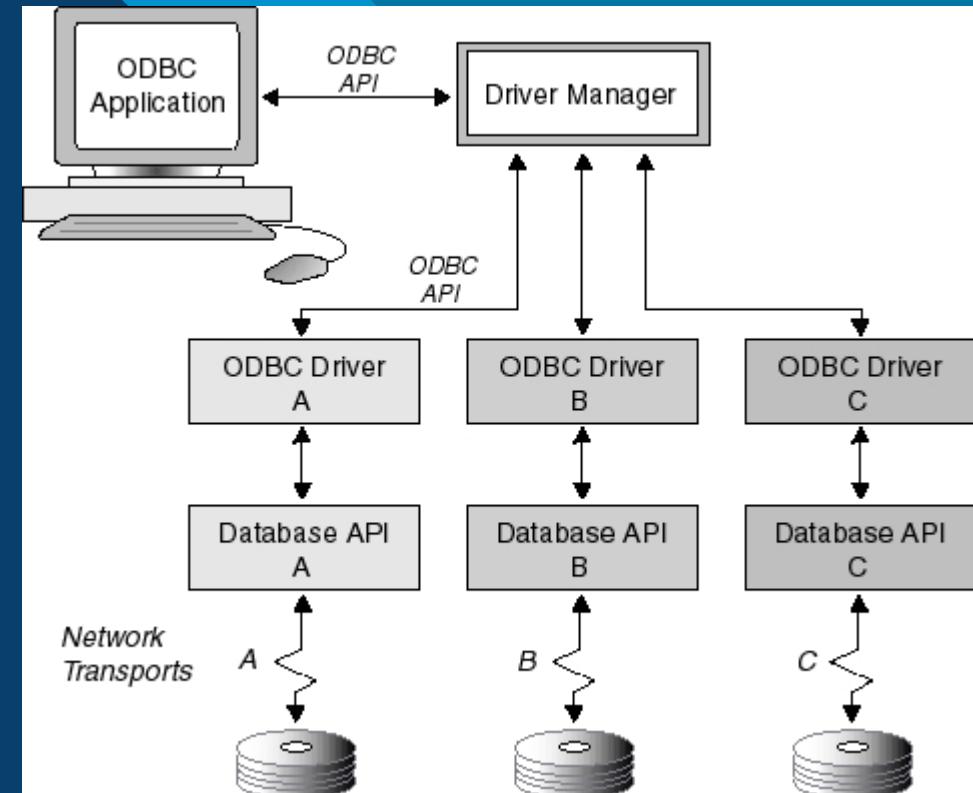
Acceso a Bases de Datos Relacionales

Trabajando con BBDD desde nuestro código



Acceso a Bases de Datos Relacionales. ODBC

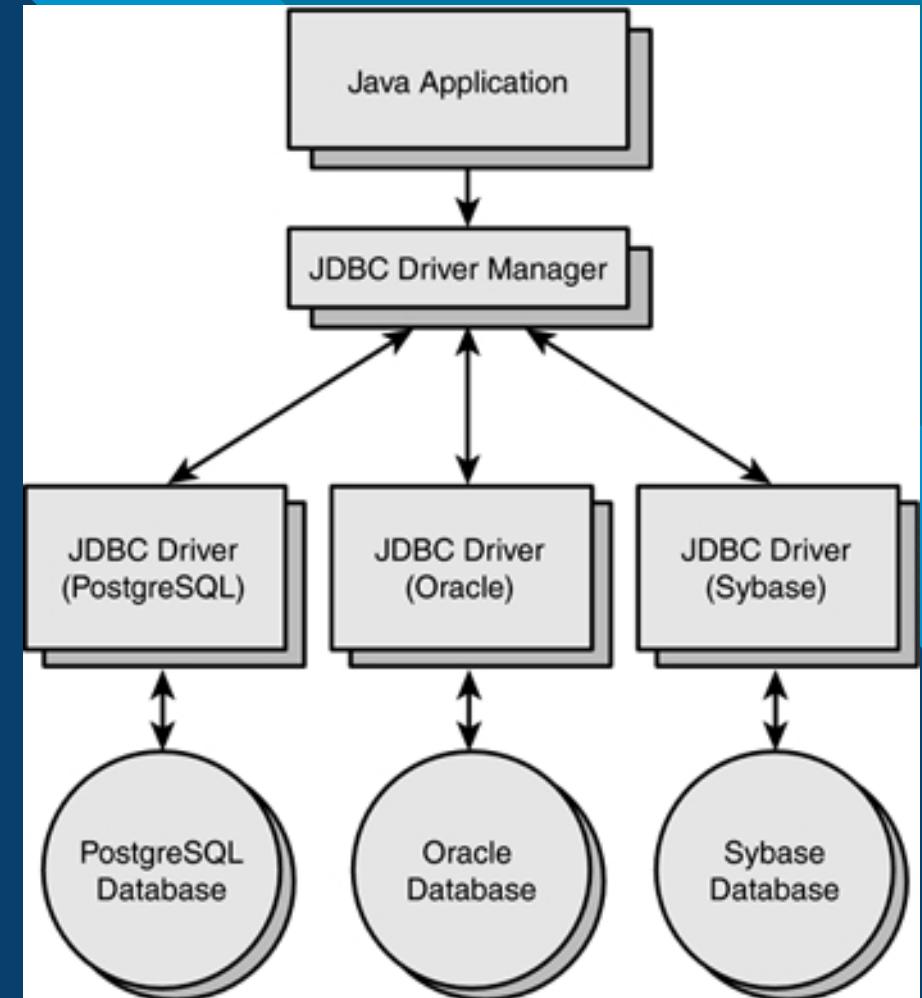
- Para acceder a BBDD Relacionales o de otro tipo podemos hacer uso de ODBC. Open DataBase Connectivity (ODBC) es un estándar de acceso a las bases de datos desarrollado por SQL Access Group (SAG) en 1992.
- El objetivo de ODBC es hacer posible el acceder a cualquier dato desde cualquier aplicación, sin importar qué sistema de gestión de bases de datos (DBMS) almacene los datos.
- ODBC logra esto al insertar una capa intermedia denominada nivel de Interfaz de Cliente SQL (CLI), entre la aplicación y el DBMS. El propósito de esta capa es traducir las consultas de datos de la aplicación en comandos que el DBMS entienda.
- Para que esto funcione tanto la aplicación como el DBMS deben ser compatibles con ODBC, esto es que la aplicación debe ser capaz de producir comandos ODBC y el DBMS debe ser capaz de responder a ellos. Desde la versión 2.0 el estándar soporta SAG (SQL Access Group) y SQL.





Acceso a Bases de Datos Relacionales. JDBC

- Java Database Connectivity, JDBC, es una API que permite la ejecución de operaciones sobre bases de datos desde el lenguaje de programación Java, independientemente del sistema operativo donde se ejecute o de la base de datos a la cual se accede, utilizando el dialecto SQL del modelo de base de datos que se utilice.
- El API JDBC se presenta como una colección de interfaces Java y métodos de gestión de manejadores de conexión hacia cada modelo específico de base de datos. Un manejador de conexiones hacia un modelo de base de datos en particular es un conjunto de clases que implementan las interfaces Java y que utilizan los métodos de registro para declarar los tipos de localizadores a base de datos (URL) que pueden manejar. Para utilizar una base de datos particular, el usuario ejecuta su programa junto con la biblioteca de conexión apropiada al modelo de su base de datos, y accede a ella estableciendo una conexión; para ello provee el localizador a la base de datos y los parámetros de conexión específicos. A partir de allí puede realizar cualquier tipo de tarea con la base de datos a la que tenga permiso: consulta, actualización, creación, modificación y borrado de tablas, ejecución de procedimientos almacenados en la base de datos, etc.





Acceso a Bases de Datos Relacionales. JDBC. Conector

- **Un conector o driver** es un conjunto de clases encargadas de implementar los interfaces del API y acceder a la base de datos.
- Las aplicaciones necesitan tener un driver para conectarse a una base de datos. Un conector suele ser un fichero .jar que contiene una implementación de todas las interfaces del API JDBC.
- Con JDBC queda oculta la capa de acceso a base de datos para el desarrollador, por lo que éste sólo debe preocuparse de la aplicación y no de cómo se acceden a los datos.
- El conector lo proporciona el fabricante de la base de datos, o un tercero. JDBC ofrece interfaces para:
 - Conectar a una BBDD
 - Ejecutar una consulta
 - Procesar los resultados



Acceso a Bases de Datos Relacionales. JDBC. Conector

- **Tipo 1. Denominados también JDBC-ODBC Bridge (puente JDBC-ODBC).**

- Proporcionan un puente entre el API JDBC y ODBC, traduciendo las llamadas JDBC a ODBC y las envía a la fuente de datos ODBC. Tiene las siguientes ventajas y desventajas:
 - Ventajas:
 - No se necesita driver específico de cada base de datos tipo ODBC.
 - Soportado por muchos fabricantes.
 - Desventajas:
 - Hay plataformas que no lo tienen implementado.
 - Rendimiento bajo debido a que se tiene que realizar una traducción de JDBC a ODBC
 - Se tiene que registrar manualmente en el gestor ODBC.

- **Tipo 2. API Nativa.**

- Convierten las llamadas JDBC a llamadas específicas de la base de datos, que pueden ser SQL Server, Informix, Oracle o Sybase.
- Este tipo de conector se comunica directamente con el servidor de base de datos, por lo que es necesario que haya código en la máquina cliente.
- Ventajas:
 - Rendimiento superior al Tipo 1.
- Desventajas:
 - La librería de la base de datos necesita cargarse en cada máquina cliente.
 - No pueden usarse para Internet.



Acceso a Bases de Datos Relacionales. JDBC. Conector

- **Tipo 3. JDBC-Net pure Java Driver.**

- Tiene una estructura de tres capas. Las peticiones JDBC a la base de datos se pasan a través de la red al servidor de la capa intermedia (middleware). Este servidor traduce el protocolo independiente del sistema gestor a protocolo específico del sistema gestor de BBDD y se envía a la base de datos. Los resultados vuelven al middleware y se dirigen al cliente.
- Ventajas:
 - Es útil para aplicaciones en internet
 - Basado en servidor, no se necesita ninguna librería de base de datos en la máquina cliente.

- **Tipo 4. Protocolo nativo.**

- Son conectores que convierten directamente las llamadas JDBC a protocolo de red usado por el sistema gestor de BBDD. Por tanto, podemos hacer llamadas directas desde la máquina cliente al servidor gestor de BBDD.
- Ventajas:
 - Solución excelente para acceso en intranets
 - No es necesaria traducción adicional o capa middleware, mejorando el rendimiento.
 - No hay que instalar ningún software adicional.
- Desventajas:
 - Necesita un driver diferente para cada base de datos. Ejemplo de este conector es Oracle-Thin.



Acceso a Bases de Datos Relacionales. JDBC vs otros

- 1.- ODBC no es apropiado para su uso directo con Java porque usa una interface C. Las llamadas desde Java a código nativo C tienen un número de inconvenientes en la seguridad, la implementación, la robustez y en la portabilidad automática de las aplicaciones.
- 2.- Una traducción literal del API C de ODBC en el API Java podría no ser deseable. Por ejemplo, Java no tiene punteros, y ODBC hace un uso copioso de ellos, incluyendo el notoriamente propenso a errores “void * ”. Se puede pensar en JDBC como un ODBC traducido a una interfaz orientada a objeto que es el natural para programadores Java.
- 3. ODBC es difícil de aprender. Mezcla características simples y avanzadas juntas, y sus opciones son complejas para ‘querys’ simples. JDBC por otro lado, ha sido diseñado para mantener las cosas sencillas mientras que permite las características avanzadas cuando éstas son necesarias.
- 4. Un API Java como JDBC es necesario en orden a permitir una solución Java “pura”. Cuando se usa ODBC, el gestor de drivers de ODBC y los drivers deben instalarse manualmente en cada máquina cliente. Como el driver JDBC esta completamente escrito en Java, el código JDBC es automáticamente instalable, portable y seguro en todas las plataformas Java.

CRUD con JDBC

Create, Read, Update & Delete

CRUD





CRUD con JDBC. Conexión

- Es necesario **instalar el conector o driver de la base de datos**. En una aplicación JAVA, se instala un conector JDBC, que es el que implementa la funcionalidad de las clases de acceso a datos, y proporciona comunicación entre el API JDBC y el sistema gestor de bases de datos. Traduce los comandos al protocolo nativo del SGBD.
- La versión 3.0 de JDBC proporciona un **pool de conexiones**. Al iniciar un servidor JAVA EE, el pool de conexiones crea un número de conexiones físicas iniciales. Cuando un objeto Java del servidor necesita una conexión (método `dataSource.getConnection()`), la fuente de datos `javax.sql.DataSource` habla con el pool de conexiones y éste le entrega una conexión lógica `java.sql.Connection`. Esta conexión es la que recibe el objeto Java.
- Para cerrar una conexión (método `connection.close()`) la fuente de datos `javax.sql.DataSource` habla con el pool de conexiones y devuelve la conexión lógica.
- Si hay una alta demanda de conexiones a la base de datos, el pool de conexiones crea más conexiones físicas de objetos tipo `Connection`.
- El código básico para conectarnos a una base de datos con pool de conexiones a través de JNDI podría ser:
 - `javax.naming.Context ctx = new InitialContext();`
 - `dataSource = (DataSource) ctx.lookup("java:comp/env/jdbc/Basededatos");`
- Para realizar una operación, obtenemos una conexión lógica:
 - `connection = dataSource.getConnection();`
- Y para cerrarla:
 - `connection.close();`



CRUD con JDBC. Conexión

```
...  
  
// Conexión a una BBDD  
Connection conexion = null;  
  
try {  
    conexion = DriverManager.getConnection(  
        "jdbc:mysql://localhost:3306/basededatos",  
        "usuario", "password");  
} catch (ClassNotFoundException cnfe) {  
    cnfe.printStackTrace();  
} catch (SQLException sqle) {  
    sqle.printStackTrace();  
} catch (InstantiationException ie) {  
    ie.printStackTrace();  
} catch (IllegalAccessException iae) {  
    iae.printStackTrace();  
}
```

```
...  
  
// Desconexión a una BBDD  
.  
.  
.  
try {  
    conexion.close();  
    conexion = null;  
} catch (SQLException sqle) {  
    sqle.printStackTrace();  
}  
.  
.
```



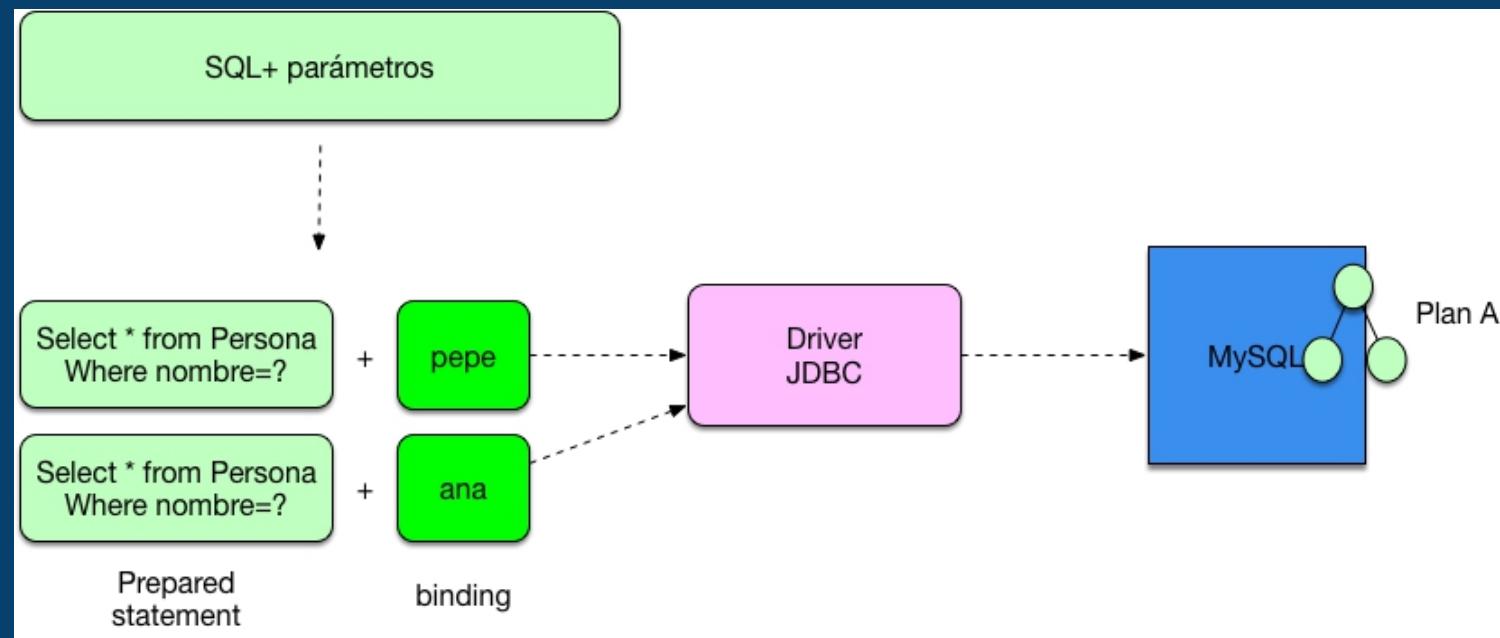
CRUD con JDBC. Consultas

- Para operar con una base de datos, nuestra aplicación debe:
 - Cargar el driver necesario.
 - Establecer una conexión con la base de datos.
 - Enviar consultas SQL, procesando el resultado.
 - Liberar los recursos al terminar.
 - Gestionar posibles errores.
- Se pueden usar tres tipos de sentencias:
 - Statement: Sentencias SQL
 - **PreparedStatement:** Consultas preparadas, como las que tienen parámetros. Recomendado
 - CallableStatement: Ejecutar procedimientos almacenados en la base de datos.
- JDBC distingue dos tipos de consultas:
 - Consultas: SELECT
 - Actualizaciones: UPDATE, INSERT, DELETE y sentencias DDL.



CRUD con JDBC. Consultas

- El uso de **JDBC Prepared Statement** es hoy en día prácticamente obligatorio. Son consultas precompiladas por lo que son más eficientes y pueden tener parámetros reutilizando el mismo plan de consulta y evitando ataques por inyección SQL. Se instancia de la siguiente manera (ejemplo con tabla medicamentos):
 - Si hay que emplear parámetros en una consulta, se puede hacer usando el carácter “?":
 - Establecemos los parámetros de una consulta utilizando métodos set que dependen del tipo SQL de la columna y el orden donde aparecen.
 - Finalmente ejecutamos la consulta con executeQuery() o executeUpdate(), dependiendo del tipo de consulta que sea.





CRUD con JDBC. Selección

- **Una consulta de selección** sobre una tabla de la base de datos devuelve un conjunto de datos organizados en filas y columnas (registros).
- **Usamos executeQuery**
- En Java almacenamos esa información mediante un objeto de la clase **ResultSet**, el cual está formado por filas y columnas.
- Un ResultSet solo puede recorrerse fila por fila, desde la primera hasta la última en una única dirección deon su método `next()`.
- Además, podemos acceder a cualquier columna de cada fila mediante algunos de sus métodos (`getInt()`, `getString()`, `getObject()`, etc). Las columnas se numeran empezando en el nº 1.
- Obtener las columnas de una consulta Select. Si en algún caso al realizar una consulta no sabemos cuántas columnas nos va a devolver, podemos usar el método `getMetaData()` que devuelve un objeto del tipo `ResultSetMetaData`. Se puede obtener antes si se usa `PreparedStatement`

```
    . . .  
  
    // Consulta tipo Select  
. . .  
  
    String sentenciaSql = "SELECT nombre, precio FROM productos " +  
                          "WHERE precio = ?";  
    PreparedStatement sentencia = null;  
    ResultSet resultado = null;  
  
    try {  
        sentencia = conexion.prepareStatement(sentenciaSql);  
        sentencia.setFloat(1, filtroPrecio);  
        resultado = sentencia.executeQuery();  
        while (resultado.next()) {  
            System.out.println("nombre: " + resultado.getString(1));  
            System.out.println("precio: " + resultado.getFloat(2));  
        }  
    } catch (SQLException sqle) {  
        sqle.printStackTrace();  
    } finally {  
        if (sentencia != null)  
            try {  
                sentencia.close();  
                resultado.close();  
            } catch (SQLException sqle) {  
                sqle.printStackTrace();  
            }  
    }  
. . .
```



CRUD con JDBC. Insercción, Actualización y Borrado

- Se les devolverá un entero con el número de registros afectados
- Se usa **executeUpdate()**

```
...  
  
// Consulta tipo insert  
.  
.  
.  
  
String sentenciaSql = "INSERT INTO productos (nombre, precio) VALUES (?, ?);";  
PreparedStatement sentencia = null;  
  
try {  
    sentencia = conexion.prepareStatement(sentenciaSql);  
    sentencia.setString(1, nombreProducto);  
    sentencia.setFloat(2, precioProducto);  
    sentencia.executeUpdate();  
} catch (SQLException sqle) {  
    sqle.printStackTrace();  
} finally {  
    if (sentencia != null)  
        try {  
            sentencia.close();  
        } catch (SQLException sqle) {  
            sqle.printStackTrace();  
        }  
}  
.  
.
```

```
...  
  
// Consulta tipo update  
.  
.  
.  
  
String sentenciaSql = "UPDATE productos SET nombre = ?, precio = ? " +  
                     "WHERE nombre = ?";  
PreparedStatement sentencia = null;  
  
try {  
    sentencia = conexion.prepareStatement(sentenciaSql);  
    sentencia.setString(1, nuevoNombreProducto);  
    sentencia.setFloat(2, precioProducto);  
    sentencia.setString(3, nombreProducto);  
    sentencia.executeUpdate();  
} catch (SQLException sqle) {  
    sqle.printStackTrace();  
} finally {  
    if (sentencia != null)  
        try {  
            sentencia.close();  
        } catch (SQLException sqle) {  
            sqle.printStackTrace();  
        }  
}  
.
```

```
...  
  
// Consulta tipo delete  
.  
.  
.  
  
String sentenciaSql = "DELETE productos WHERE nombre = ?";  
PreparedStatement sentencia = null;  
  
try {  
    sentencia = conexion.prepareStatement(sentenciaSql);  
    sentencia.setString(1, nombreProducto);  
    sentencia.executeUpdate();  
} catch (SQLException sqle) {  
    sqle.printStackTrace();  
} finally {  
    if (sentencia != null)  
        try {  
            sentencia.close();  
        } catch (SQLException sqle) {  
            sqle.printStackTrace();  
        }  
}  
.
```



CRUD con JDBC. Procedimientos almacenados

- La ejecución de procedimientos almacenados sigue la misma estructura que cualquiera de las sentencias SQL de los ejemplos anteriores, con la excepción de que usaremos la clase **CallableStatement** para representar al procedimiento y el método **execute()** de la misma para ejecutarlo.

```
    . . .
    // Llamando a un procedimiento almacenado
    . . .
    //Método para eliminar todos los productos

    String sentenciaSql = "call eliminar.todos.los.productos()";
    CallableStatement procedimiento = null;

    try {
        procedimiento = conexion.prepareCall(sentenciaSql);
        procedimiento.execute();
    } catch (SQLException sqle) {
        . . .
    }
    . . .
```



CRUD con JDBC. Funciones almacenadas

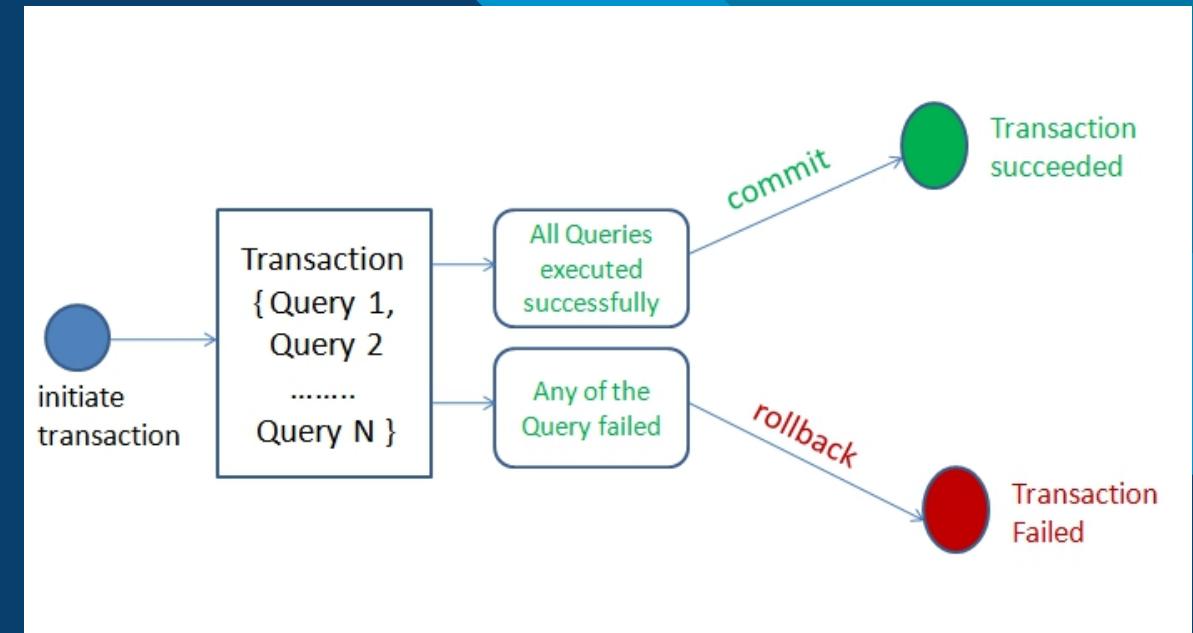
- En el caso de la ejecución de las funciones almacenadas, seguiremos la misma estructura que con las funciones de agregación de SQL (COUNT(), SUM(), AVG()), ya que nuestras funciones nos devolverán siempre un solo valor (o null en el caso de que no devuelvan nada).

```
• • •  
  
// Llamando a una función almacenada  
. . .  
String sentenciaSql = "SELECT get_precio_mas_alto()";  
PreparedStatement sentencia = null;  
ResultSet resultado = null;  
  
try {  
    sentencia = conexion.prepareStatement(sentenciaSql);  
    resultado = sentencia.executeQuery();  
    resultado.next();  
  
    if (resultado.wasNull())  
        System.out.println("No hay datos");  
    else {  
        float precioMasAlto = resultado.getFloat(1);  
        System.out.println("El producto más caro vale " + precioMasAlto);  
    }  
} catch (SQLException sqle) {  
. . .  
}  
. . .
```



CRUD con JDBC. Transacciones

- En el ámbito de las Bases de Datos, una transacción es cualquier conjunto de sentencias SQL que se ejecutan como si de una sola se tratara.
- La idea principal es poder ejecutar varias sentencias, que están relacionadas de alguna manera, de forma que si cualquiera de ellas fallara o produjera un error, no se ejecutara ninguna más e incluso se deshicieran todos los cambios que hayan podido efectuar las que ya se habían ejecutado dentro de la misma transacción.
- Para ello, tendremos que incorporar tres nuevas instrucciones a las que ya veníamos utilizando hasta ahora. Una instrucción para indicar que comienza una transacción (`conexion.setAutoCommit(false)`), otra para indicar cuando termina (`conexion.commit()`) y otra para indicar que la transacción actual debe abortarse y todos los cambios hasta el momento deben ser restaurados al estado anterior (`conexion.rollback()`).





CRUD con JDBC. Transacciones

```
...  
  
// Ejemplo de transacción  
...  
  
String nombreProducto = . . .;  
float precioProducto = . . .;  
int idCategoria = . . .;  
// El id del producto que vamos a registrar aún no se conoce  
  
String sqlAltaProducto = "INSERT INTO productos (nombre, precio)  
VALUES (?, ?);  
String sqlRelacionProducto =  
    "INSERT INTO producto_categoria(id_producto,  
id_categoria) "+  
    "VALUES (?, ?);  
  
try {  
    //Inicia transacción  
    conexion.setAutoCommit(false);  
  
    PreparedStatement sentenciaAltaProducto =  
        conexion.prepareStatement(sqlAltaProducto,  
        PreparedStatement.RETURN_GENERATED_KEYS);  
    sentenciaAltaProducto.setString(1, nombreProducto);  
    sentenciaAltaProducto.setFloat(2, precioProducto);  
    sentenciaAltaProducto.executeUpdate();  
  
    ...  
}
```

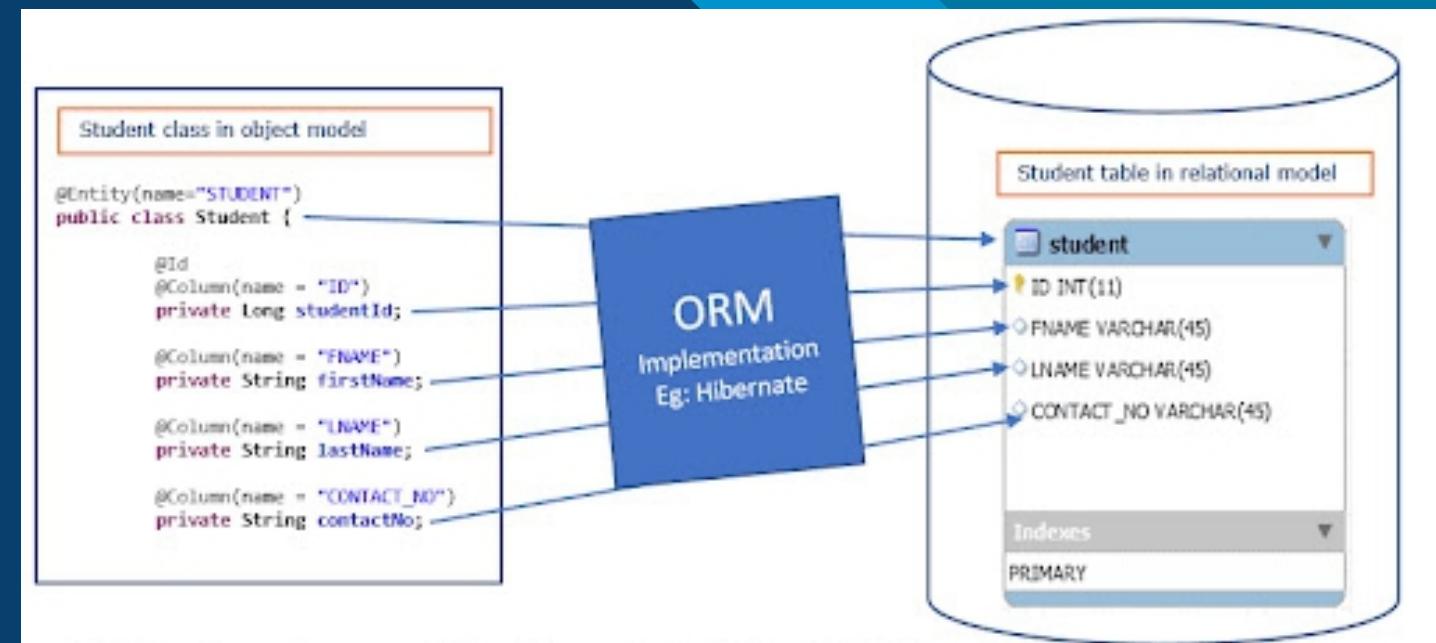
```
...  
  
// Obtiene el id del producto que se acaba de registrar  
ResultSet idsGenerados = sentenciaAltaProducto.getGeneratedKeys();  
idsGenerados.next();  
int idProducto = idsGenerados.getInt(1);  
sentenciaAltaProducto.close();  
  
PreparedStatement sentenciaRelacionProducto =  
    conexion.prepareStatement(sqlRelacionProducto);  
sentenciaRelacionProducto.setInt(1, idProducto);  
sentenciaRelacionProducto.setInt(2, idCategoria);  
sentenciaRelacionProducto.executeUpdate();  
  
// Valida la transacción  
conexion.commit();  
} catch (SQLException sqle) {  
    sqle.printStackTrace();  
} finally {  
    if (sentencia != null)  
        try {  
            sentencia.close();  
            resultado.close();  
        } catch (SQLException sqle) {  
            sqle.printStackTrace();  
        }  
}  
...  
...
```

ORMs

Automatizando el Mapeo Objeto Relacional

ORMS

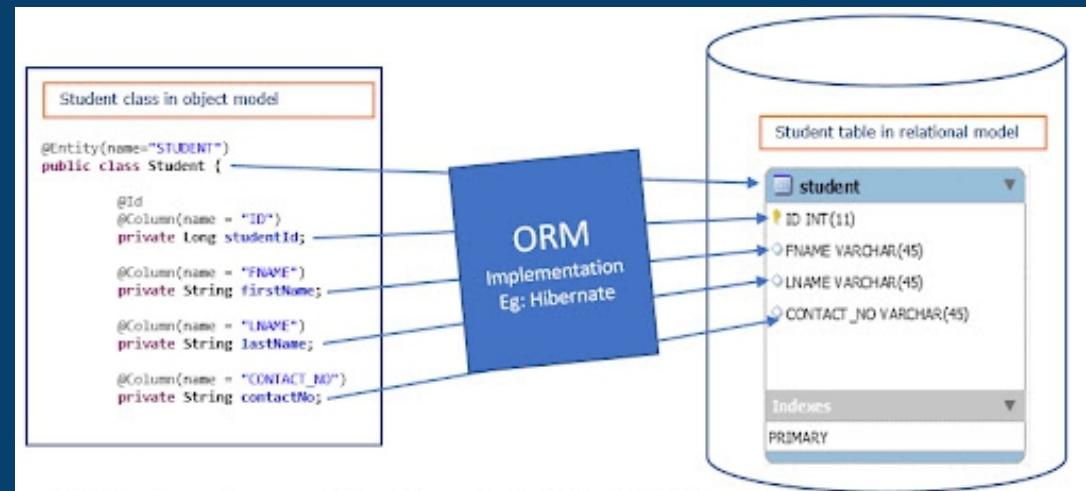
- Object Relational Mapping (ORM) es la herramienta que nos sirve para transformar representaciones de datos de los Sistemas de Bases de Datos Relacionales, a representaciones (Modelos) de objetos. Dado a que los RDBMS carecen de la flexibilidad para representar datos no escalares, la existencia de un ORM es fundamental para el desarrollo de sistemas de software robustos y escalables.
- Las herramientas ORM pues, actúan como un puente que conecta las ventajas de los RDBMS con la buena representación de estos en un lenguaje Orientado a Objetos, o, dicho en otras palabras, nos lleva de la base de datos al lenguaje de programación.





ORMS

- Las herramientas ORM facilitan el mapeo de atributos entre una base de datos relacional y el modelo de objetos de una aplicación, mediante archivos declarativos (XML) que permiten establecer estas relaciones. Gracias a las ORM, podemos conectar con una base de datos relacional para extraer la información contenida en objetos de programas que están almacenados.
- Para ello, sólo tendremos que definir la forma en la que establecer la correspondencia entre las clases y las tablas una sola vez (indicando qué propiedad se corresponde con cada columna, qué clase con cada tabla, etc.). Una vez hecho esto, podremos utilizar POJO's de nuestra aplicación e indicar a la ORM que los haga persistentes, consiguiendo que una sola herramienta pueda leer o escribir en la base de datos utilizando VO's directamente.





ORMS

- Ventajas de ORM.

- Ayudan a reducir el tiempo de desarrollo de software. La mayoría de las herramientas ORM disponibles, permiten la creación del modelo a través del esquema de la base de datos, es decir, el usuario crea la base de datos y la herramienta automáticamente lee el esquema de tablas y relaciones y crea un modelo ajustado.
- Abstracción de la base de datos.
- Reutilización.
- Permiten persistir objetos a través de un método `Orm.Save` y generar el SQL correspondiente.
- Permiten recuperar los objetos persistidos a través de un método `Orm.Load`.
- Lenguaje propio para realizar las consultas.
- Independencia de la base de datos.
- Incentivan la portabilidad y escalabilidad de los programas de software.

- Desventajas de ORM.

- Tiempo utilizado en el aprendizaje. Este tipo de herramientas suelen ser complejas por lo que su correcta utilización lleva un tiempo que hay que emplear en ver el funcionamiento correcto y ver todo el partido que se le puede sacar.
- Menor rendimiento (aplicaciones algo más lentas). Esto es debido a que todas las consultas que se hagan sobre la base de datos, el sistema primero deberá de transformarlas al lenguaje propio de la herramienta, luego leer los registros y por último crear los objetos.
- Sistemas complejos. Normalmente la utilidad de ORM desciende con la mayor complejidad del sistema relacional.



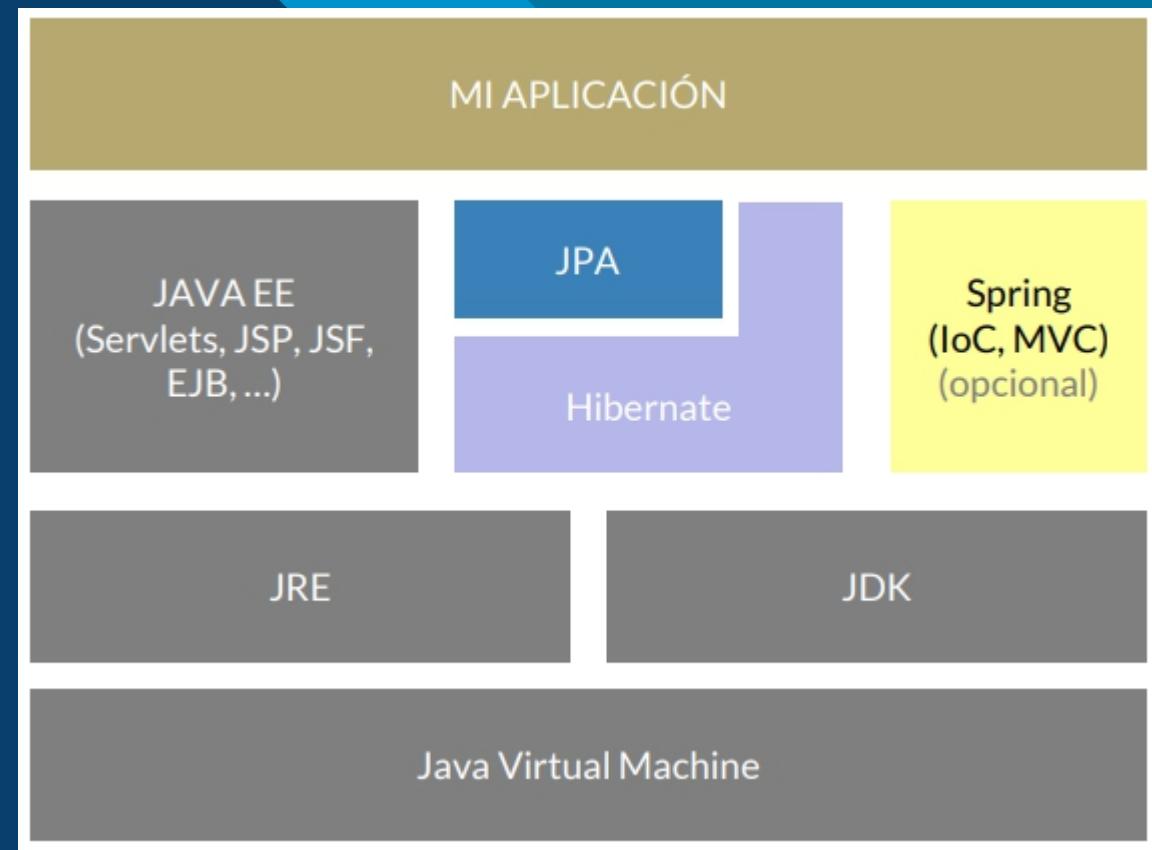
ORMS

- Entre las herramientas ORM más relevantes encontramos las siguientes:
- **Hibernate**: es una herramienta de Mapeo objeto-relacional (ORM) para la plataforma Java (y disponible también para .Net con el nombre de NHibernate) que facilita el mapeo de atributos entre una base de datos relacional tradicional y el modelo de objetos de una aplicación. Utiliza archivos declarativos (XML) o anotaciones en los beans de las entidades que permiten establecer estas relaciones. Hibernate es software libre, distribuido bajo los términos de la licencia GNU LGPL.
- **Java Persistence Api (JPA)**: El Java Persistence API (JPA) es una especificación de Sun Microsystems para la persistencia de objetos Java a cualquier base de datos relacional. Esta API fue desarrollada para la plataforma JEE e incluida en el estándar de EJB 3.0, formando parte de la Java Specification Request JSR 220. Para su utilización, JPA requiere de J2SE 1.5 (también conocida como Java 5) o superior, ya que hace uso intensivo de las nuevas características de lenguaje Java, como las anotaciones y los genéricos. Hibernate implementa y deja utilizar JPA, ampliando la forma de utilizarlo.
- **iBatis**: es un framework de persistencia desarrollado por la Apache software Foundation. Al igual que el resto de los proyectos desarrollados por la ASF, iBatis es una herramienta de código libre. iBatis sigue el mismo esquema de uso que Hibernate; se apoya en ficheros de mapeo XML para persistir la información contenida en los objetos en un repositorio relacional.



Hibernate y JPA

- **Productividad:** Nos permite centrarnos en el desarrollo de la lógica de negocio, ahorrándonos una gran cantidad de código.
- **Mantenibilidad:** El ORM de Hibernate, al reducir el número de líneas de código que tenemos que escribir, nos permite realizar refactorizaciones de forma más fácil.
- **Rendimiento:** Aunque la persistencia manual podría permitir un rendimiento mayor, Hibernate aporta muchas optimizaciones, como el almacenamiento en caché.
- **Independencia:** Hibernate nos permite aislarnos del RDBMS concreto que estemos usando.
- **JPA** nos ofrece facilidad para especificar como nuestros objetos Java se relaciona con el esquema de una base de datos en base a notaciones
- **JPA** nos da una api sencilla para realizar las operaciones CRUD
- **JPA** tiene un lenguaje y un API para realizar consultas sobre los datos (JPQL).
- **JPA** establece elementos de optimización (actualización de entidades, captura de asociaciones, caché, ...)





Hibernate y JPA. Instalación y Configuración

- Haremos uso de Maven para instalar el core de Hibernate con JPA y el Driver de la BBDD que queramos usar

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.5.3.Final</version>
</dependency>
```



Hibernate y JPA. Instalación y Configuración

- **Configuración de la Unidad de Persistencia:** `hibernate.cfg.xml` (o similar). El fichero de configuración de hibernate `hibernate.cfg.xml` se debe crear directamente dentro de la carpeta `src` del proyecto y el propio framework Hibernate será el encargado de leerlo para obtener las sesiones que permitan conectar con la Base de Datos en su apartado de **propiedades**.
- Si tenemos distintos valores de configuración para conectar con diferentes sistemas de bases de datos, necesitamos diferentes ficheros de configuración (p.e. `mysql.cfg.xml`, `postgre.cfg.xml`, `mssqlserver.cfg.xml`, etc) e instanciar desde doigo la que queremos.
- Las propiedades más importantes del fichero `Hibernate.cfg.xml` son:
 - `Hibernate.dialect`: Dialecto o lenguaje empleado. Por ejemplo, MySQL.
 - `Hibernate.connection.driver_class`: Driver utilizado para la conexión con la base de datos.
 - `Hibernate.connection.url`: Dirección de la base de datos con la que se va a conectar Hibernate.
 - `Hibernate.connection.username`: Nombre del usuario que va a realizar la extracción de información. Por defecto, el nombre de usuario es root.
 - `Hibernate.connection.password`: Contraseña del root.
 - `Hibernate.show_sql`: Para mostrar la herramienta. Por defecto, su valor es true.
- Además debemos definir las **clases o entidades** a mapear dentro de la unidad de persistencia en las etiquetas `class`.



Hibernate y JPA. Instalación y Configuración

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_2.xsd"
    version="2.2">

    <!-- Nombre de la unidad de persistencia -->
    <persistence-unit name="default">
        <description> Ejemplo Crud básico Hibernate JPA</description>

        <!-- Indicamos el "provider" que es la implementación de JPA que estamos usando.
En nuestro ejemplo hibernate, pero existen otros proveedores como EclipseLink: -->
        <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>

        <!-- Definiremos las clases que representan "entidades". Por cada clase
debemos utilizar la etiqueta <class> cuyo contenido debe incluir el paquete y el
nombre de la clase: -->

        <class>hibernate.Departamento</class>
        <class>hibernate.Empleado</class>
    <!-- Nuestras clases de persistencia -->
```

```
<!-- Añadimos las propiedades de conexión a la base de datos MySQL -->
<properties>
    <property name="hibernate.connection.url"
value="jdbc:mariadb://localhost:3306/mydb"/>
    <property name="hibernate.connection.driver_class"
value="org.mariadb.jdbc.Driver"/>
    <property name="hibernate.connection.user" value="mydb"/>
    <property name="hibernate.connection.password" value="mydb1234"/>
    <!-- Para ver las consultas -->
    <property name="hibernate.show_sql" value="true"/>
    <!-- Para trabajar con el esquema
        validate: validate the schema, makes no changes to the database.
        update: update the schema si detecta cambios.
        create: creates the schema, destroying previous data.
        create-drop: drop the schema when the SessionFactory is closed explicitly,
typically when the application is stopped.
        none: does nothing with the schema, makes no changes to the database-->
    <property name="hibernate.hbm2ddl.auto" value="update" />
</properties>
</persistence-unit>
</persistence>
```



Hibernate y JPA. Instalación y Configuración

- Todas las configuraciones anteriores también se pueden indicar mediante código Java en el siguiente fichero

```
public class HibernateUtil {  
  
    private static SessionFactory sessionFactory;  
    private static Session session;  
  
    /**  
     * Crea la factoria de sesiones  
     */  
    public static void buildSessionFactory() {  
  
        Configuration configuration = new Configuration();  
        // La llamada al método configure() carga los parámetros del fichero hibernate.cfg.xml  
        configuration.configure();  
  
        // Se registran las clases que hay que mapear con cada tabla de la base de datos  
        configuration.addAnnotatedClass(Clase1.class);  
        configuration.addAnnotatedClass(Clase2.class);  
        configuration.addAnnotatedClass(Clase3.class);  
        . . .  
        //Se crea una SessionFactory a partir del objeto Configuration  
        ServiceRegistry serviceRegistry = new StandardServiceRegistryBuilder().applySettings(  
            configuration.getProperties()).build();  
        sessionFactory = configuration.buildSessionFactory(serviceRegistry);  
    }  
}
```

```
/**  
 * Abre una nueva sesión  
 */  
public static void openSession() {  
    session = sessionFactory.openSession();  
}  
  
/**  
 * Devuelve la sesión actual  
 * @return  
 */  
public static Session getCurrentSession() {  
  
    if ((session == null) || (!session.isOpen()))  
        openSession();  
  
    return session;  
}  
  
/**  
 * Cierra Hibernate  
 */  
public static void closeSessionFactory() {  
  
    if (session != null)  
        session.close();  
  
    if (sessionFactory != null)  
        sessionFactory.close();  
}  
}
```



Hibernate y JPA. Mapeo de Entidades

- **Los mapeos de entidades se hacen sobre clases POJO usando anotaciones.**

- **@Entity** Indica que la clase es una tabla en la base de datos.
- **@Table(name = "nombre_tabla", catalog = "nombre_base_datos")** Indica el nombre de la tabla (si es distinto al de la entidad) y la base de datos a la que pertenece (este último parámetro no es necesario ya que esa información viene en el fichero de configuración). Por defecto, Hibernate toma una estrategia de generación de nombres para transformar el nombre de una clase (que en Java normalmente estará escrita en notación UpperCamelCase) al nombre de una tabla (por defecto, suele usar el mismo nombre en MAYÚSCULAS). Si no sigue esto, usar la notación **@Table**
- **@Id** Indica que un atributo es la clave. Como la anotación **@Id** es sobre un campo, por defecto, Hibernate habilitará por defecto todos los atributos de la clase como propiedades persistentes.
- Añadiendo la anotación **@GeneratedValue** a continuación de **@Id**, JPA asume que se va a generar un valor y a asignar el mismo antes de almacenar la instancia de la entidad. Existen diferentes estrategias de asignación:
 - **GenerationType.AUTO**: Hiberante escoge la mejor estrategia en función del dialecto SQL configurado (es decir, dependiendo del RDBMS).
 - **GenerationType.SEQUENCE**: Espera usar una secuencia SQL para generar los valores de todas las claves.
 - **GenerationType.IDENTITY**: Hibernate utiliza una columna especial dependiendo de cómo sea el RDBMS, puede ser autonumérica (MariaDB) o siguiendo otra política que asegure que esa clave es única para ese objeto.
 - **GenerationType.TABLE**: Hibernate usa una tabla extra en nuestra base de datos. Tiene una fila por cada tipo de entidad diferente, y almacena el siguiente valor a utilizar. **@Column(name = "nombre_columna")** Se utiliza para indicar el nombre de la columna en la tabla donde debe ser mapeado el atributo



Hibernate y JPA. Mapeo de Entidades

- **Cuando mapeamos una entidad, todos sus atributos son considerados persistentes por defecto. Las reglas por defecto son:**

- Si la propiedad es un tipo primitivo, un envoltorio de un tipo primitivo (Integer, Double, ...), String, BigInteger, BigDecimal, java.util.Date, java.util.Calendar, java.sql.Date, java.sql.Time, java.sql.Timestamp, byte[], Byte[], char[], o Character[] se persiste automáticamente con el tipo de dato SQL adecuado.
- Si es java.io.Serializable, se almacena con su representación serializada (esto no será lo que habitualmente deseemos).
- Si usamos @Embeddable (lo estudiaremos después), también lo persiste.
- En otro caso, lanzará un error en la inicialización. Hibernate escoge, dependiendo del dialecto configurado, la mejor correspondencia de tipos de dato en el RDBMS para los tipos Java que hayamos usado.
- **@Column(name = “nombre_columna”)** Esta anotación, sobre una propiedad, nos permitirá indicar algunas propiedades, entre las que se encuentran:
 - nullable: nos permite indicar si la columna mapeada puede o no almacenar valores nulos. En la práctica, es como marcar el campo como requerido.
 - name: permite modificar el nombre por defecto que tendrá la columna mapeada.
 - insertable, updatable: podemos modificar si la entidad puede ser insertada, modificada, ...
 - length: nos permite definir el número de caracteres de la columna.
 - ¿Dónde anotar? ¿En las propiedades o en los getter? Hibernate nos permite definir las anotaciones (@Id, @Column, ...) tanto sobre las propiedades como sobre los métodos getter (nunca sobre los setter). La pauta la marca la anotación @Id. Allá donde usemos esta anotación, marcaremos la estrategia a seguir.



Hibernate y JPA. Mapeo de Entidades

• Tipos Temporales

- Los tipos de datos temporales, de fecha y hora, tienen un tratamiento algo especial en Hibernate. Para un campo que contiene este tipo de información, se añade la propiedad **@Temporal**.
- Esta anotación la podemos usar con los tipos `java.util.Date`, `java.util.Calendar`, `java.sql.Date`, `java.sql.Time`, `java.util.Timestamp`. Hibernate también soporta los nuevos tipo de `java.time` disponibles en el JDK 8.
- Como propiedad, podemos indicar que tipo de dato temporal vamos a querer usar a través del atributo `TemporalType`, teniendo disponibles `DATE`, `TIME`, `TIMESTAMP`.
- **@CreationTimestamp**. La anotación `@CreationTimestamp` nos permite indicar a Hibernate que en el atributo anotado con esta anotación debe almacenarse la actual fecha y hora de la JVM cuando la entidad sea almacenada.

• Tipos Embebidos

- En ocasiones, nos puede interesar tratar un grupo de atributos como si fueran uno solo. Un ejemplo clásico suele ser la dirección (nombre de la vía, número, código postal, ...). Para este tipo de situaciones tenemos la posibilidad de convertir una clases en **@Embeddable**.

• Campos calculados

- **@Generated**. Estos valores son generados por la base de datos (y no por Java). Cuando Hibernate detecta una inserción o actualización, realiza una (re)generación del valor del campo calculado. Esta notación nos permite indicar cuando el campo será generado, a saber:
 - `INSERT`: Será generado en la inserción, pero no en las sentencias `UPDATE` posteriores.
 - `ALWAYS`: En cualquier modificación (`INSERT`, `UPDATE`).



Hibernate y JPA. Mapeo de Entidades

• Campos calculados

- **@Generated.** Estos valores son generados por la base de datos (y no por Java). Cuando Hibernate detecta una inserción o actualización, realiza una (re)generación del valor del campo calculado. Esta notación nos permite indicar cuando el campo será generado, a saber:
 - INSERT: Será generado en la inserción, pero no en las sentencias UPDATE posteriores.
 - ALWAYS: En cualquier modificación (INSERT, UPDATE).

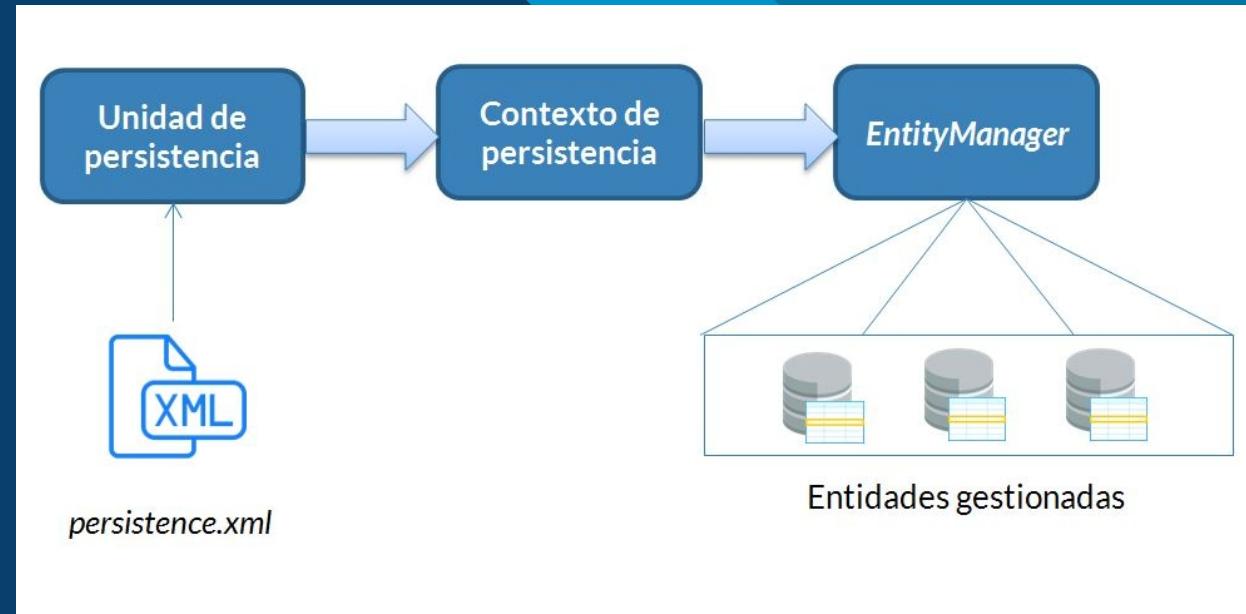
• Campos transformados

- Hibernate nos permite personalizar el código SQL que utiliza para leer o almacenar los valores de algunas columnas. Por ejemplo, si nuestro RDBMS tiene alguna función de encriptación o codificación, la podemos invocar para almacenar una columna (por ejemplo, una contraseña) usando **@ColumnTransformer** y el nombre de la función a usar del RDBMS.



Hibernate y JPA. Entidades y Unidad de Persistencia

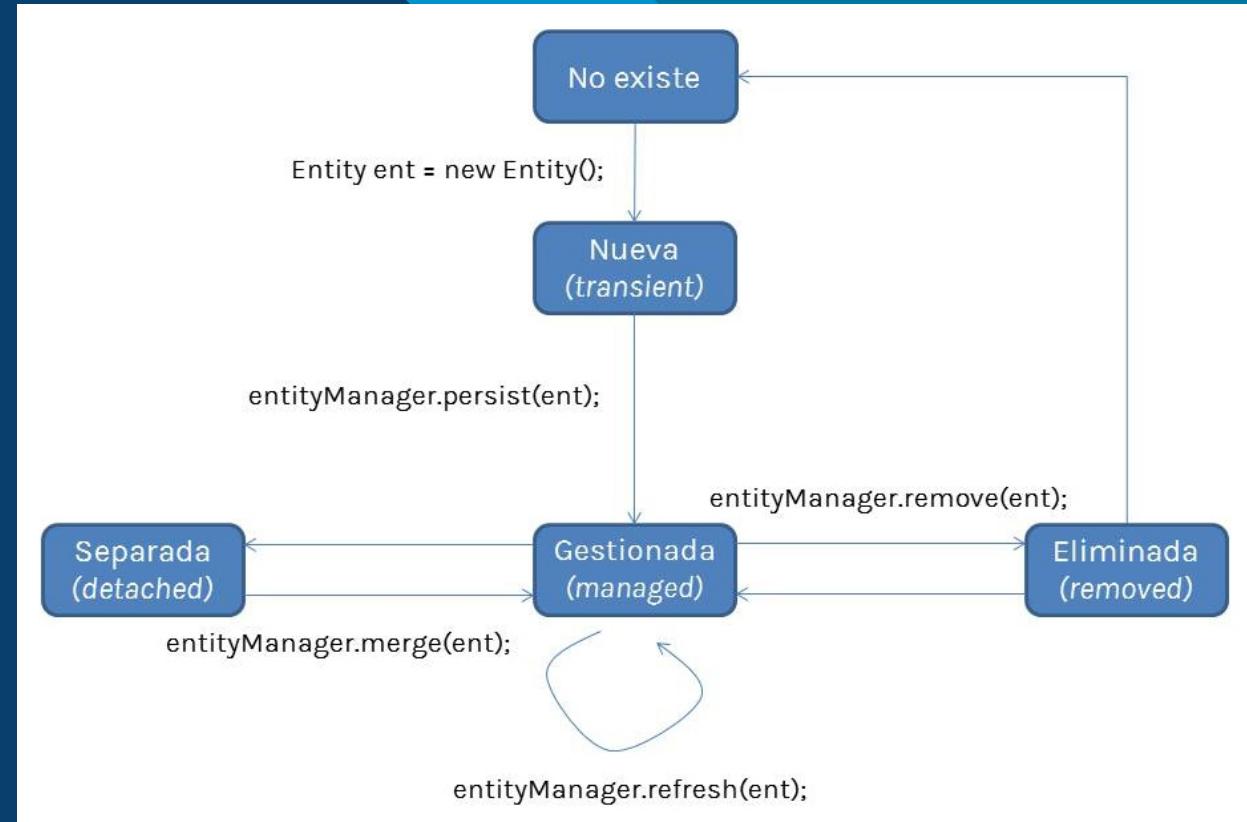
- Una **unidad de persistencia** representa un conjunto de entidades que pueden ser mapeadas en una base de datos, así como la información necesaria para que la aplicación se pueda conectar a la misma. Toda esta información viene definida en un fichero llamado persistence.xml.
- Dentro de este fichero podemos definir una o más unidades de persistencia, incluyendo siempre un nombre único y una fuente de datos (datasource).
- Un contexto de persistencia representa un conjunto de entidades que se encuentran gestionadas en un momento dado. Podríamos decir que es algo así como una instancia de una unidad de persistencia.
- **Lo manejamos con EntityManager.**
- **El EntityManager está asociado a una sesión y el ciclo de vida de las entidades gestionadas está ligado a esa sesión, si accedemos desde sesiones diferentes a una entidad puede que no esté cargada o veamos los cambios producidos.**





Hibernate y JPA. Unidad de Persistencia

- Ciclo de vida de una Entidad en JPA en la misma Sesión.
 - **transient (nueva)**: la entidad acaba de ser creada (posiblemente con el operador new) y aun no está asociada al contexto de persistencia. No tiene representación en la base de datos. Se hace con el método new.
 - **managed, persistent**: la entidad tiene un identificador y está asociada al contexto de persistencia. Puede estar almacenada en la base de datos, o aun no.
 - **detached**: la entidad tiene un identificador, pero no está asociada al contexto de persistencia (normalmente), porque hemos cerrado el contexto de persistencia.
 - **removed**: la entidad tiene un identificador y está asociada al contexto de persistencia, pero este tiene programada su eliminación.
- Obtención de una referencia a una entidad sin inicializar sus datos. JPA nos permite hacerlo a través del método getReference.
- Obtención de una referencia a una entidad inicializando sus datos, usaremos el método find
- Modificación de una entidad que ya está manejada/persistida Los cambios sobre las entidades manejadas por JPA serán detectados automáticamente, y persistidos cuando el contexto de persistencia sea flushed. Usaremos el método flush()
- Refresco del estado de una entidad. Podemos recargar una entidad desde el contexto de persistencia en cualquier momento con refresh()





Hibernate y JPA. Transacciones

- Las transacciones hacen que un conjunto de operaciones se realicen de forma atómica. Hibernate usa Resource Local es un tipo de transaccionalidad mediante la cual delegamos la responsabilidad de realizar las transacciones en el programador.
- Las transacciones se obtienen a partir del **EntityManager**, y son de tipo **javax.persistence.EntityTransaction**. Podemos manejar las transacciones a través de los métodos **begin()**, **commit()** y **rollback()**.
- Su uso está orientado sobre todo a aplicaciones de escritorio o que no se ejecutan en un servidor de aplicaciones.

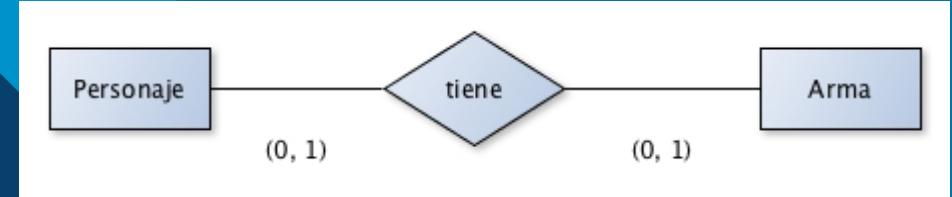


Hibernate y JPA. Relaciones

- Para el mapeo de las relaciones, además de crear el correspondiente objeto que permita mantener la relación entre las clases (de forma bidireccional), éstos atributos deben ser mapeados según convenga. Para todos los casos, en ambos casos se indicará el tipo de relación visto desde el lado correspondiente pero sólo se codificará la información de mapeo en uno de los lados:
 - **@OneToOne** Indica que el objeto es parte de una relación 1-1
 - **@ManyToOne** Indica que el objeto es parte de una relación N-1. En este caso el atributo sería el lado 1
 - **@OneToMany** Indica que el objeto es parte de una relación 1-N. En este caso el atributo sería el lado N
 - **@ManyToMany** Indica que el objeto es parte de una relación N-M. En este caso se indica la tabla que mantiene la referencia entre las tablas y los campos que hacen el papel de claves ajenas en la base de datos
- En el otro lado de la relación indicaremos el tipo de relación acompañado de la anotación **@MappedBy** añadiendo el atributo de la otra clase donde se especifica toda la información sobre el mapeo



Hibernate y JPA. Relaciones 1 a 1

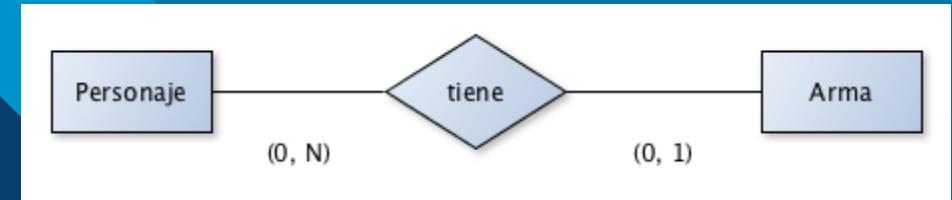


```
...  
@Entity  
@Table(name = "personajes")  
public class Personaje {  
  
    ...  
    private Arma arma;  
    ...  
    @OneToOne(cascade = CascadeType.ALL)  
    @PrimaryKeyJoinColumn  
    public Arma getArma() { return arma; }  
    ...  
}
```

```
...  
@Entity  
@Table(name = "armas")  
public class Arma {  
  
    ...  
    private Personaje personaje;  
    ...  
    @OneToOne(cascade = CascadeType.ALL)  
    @PrimaryKeyJoinColumn  
    public Personaje getPersonaje() { return personaje; }  
    ...  
}
```



Hibernate y JPA. Relaciones 1 a N

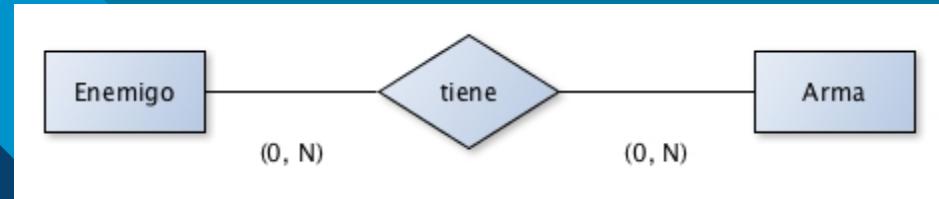


```
...  
@Entity  
@Table(name = "personajes")  
public class Personaje {  
    ...  
    private Arma arma;  
    ...  
    @ManyToOne  
    @JoinColumn(name="id_arma")  
    public Arma getArma() { return arma; }  
    ...  
}
```

```
...  
@Entity  
@Table(name = "armas")  
public class Arma {  
    ...  
    private List<Personaje> personajes;  
    ...  
    @OneToMany(mappedBy = "arma", cascade = CascadeType.ALL)  
    public List<Personaje> getPersonajes() { return personajes; }  
    ...  
}
```

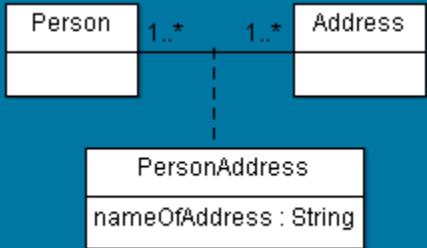


Hibernate y JPA. Relaciones N a M



```
...
@Entity
@Table(name = "enemigos")
public class Enemigo {
    ...
    private List<Arma> armas;
    ...
    // Cuando se elimine un personaje se desvinculará
    // el arma pero ésta no se borrará (DETACH)
    @ManyToMany(cascade = CascadeType.DETACH)
    @JoinTable(name="enemigo_arma",
        joinColumns={@JoinColumn(name="id_enemigo")},
        inverseJoinColumns={@JoinColumn(name="id_arma")})
    public List<Arma> getArmas() { return armas; }
    ...
}
```

```
...
@Entity
@Table(name = "armas")
public class Arma {
    ...
    private List<Enemigo> enemigos;
    ...
    @ManyToMany(mappedBy = "armas", cascade = CascadeType.DETACH)
    public List<Enemigo> getEnemigos() { return enemigos; }
    ...
}
```



Hibernate y JPA. Relaciones N a M con Atributos

- Algunos autores llaman a este tipo de asociación con atributos clase de asociación.
- Como este nuevo atributo no es ni de la dirección, ni de la persona, no lo podemos colocar en ninguna de las dos entidades, por lo que tenemos que seguir los siguientes pasos:
 - Generar una nueva entidad PersonAddress
 - **Romper la asociación @ManyToMany en ambos extremos en dos asociaciones que den el mismo resultado: @ManyToOne + @OneToMany.**
 - Manejar de forma conveniente la clave primaria de esta nueva entidad. Al ser una clave primaria compuesta, necesitaremos de una clase extra, PersonAddressId, y de la anotación @IdClass, para poder manejarla.
 - La anotación @Id solo está permitida, en primera instancia, para claves primarias simples, por lo que una entidad, por norma, no puede tener dos atributos anotados con @Id. Para poder manejar una clave primaria compuesta, JPA nos obliga a utilizar alguna estrategia diferente, como @IdClass o @EmbeddId. Nosotros optamos por la primera. Para ello, creamos una clase que tendrá los campos que conforman la clave primaria y que cumplirá con las siguientes características:
 - Debe ser una clase pública
 - Debe tener un constructor sin argumentos
 - Debe implementar Serializable
 - No debe tener clave primaria propia
 - Debe implementar los métodos equals y hashCode.



Hibernate y JPA. Relaciones y Bidireccionalidad

- Cuando realizamos **relaciones bi-direccionales** de las clases, desde Java podemos acceder a los elementos relacionados desde cualquier de los dos objetos de una relación. Para ello utilizaremos **atributos únicos en el lado de @ManyToOne y colecciones (List, Set, Map) para el lado de @OneToMany o @ManyToMany**. Debemos tener cuidado porque podemos entrar en bucles recursivos.
- Aunque las relaciones sean bidireccionales, **hibernate enfoca las relaciones desde una jerarquía padre-hijo**. Para asegurarnos de mantener la relaciones de forma correcta en la base de datos, añadiremos a las clases mapeadas una serie de métodos utilitarios.
- **Relaciones 1:N**. En las relaciones @OneToMany y @ManyToOne es recomendable que contenga algunos métodos utilitarios para mantener las relaciones:
 - En la clase muchos debemos tener un método que nos permita añadir al objeto de la relación 1
 - En la clase 1, debemos tener métodos que nos permita eliminar elementos de la colección de muchos
- **Relaciones N:M**
 - En las relaciones @ManyToMany ambas clases funcionan como padre y también como hijo. También creamos métodos utilitarios para mantener la integridad de las relaciones en la base de datos basada en la idea anterior.
 - Estos métodos se les conoce como Helpers
- Otro aspecto a tener en cuenta en las relaciones **@ManyToMany** es el uso de colecciones tipo **Set<>**, ya que favorecen el rendimiento respecto a colecciones de tipo **List<>**. Hibernate necesita lanzar más consultas para las eliminaciones o inserciones usando una lista, frente a un conjunto.
- Del mismo modo, si queremos crear el esquema de la base de datos a partir de las clases mapeadas, solo usando colecciones tipo Set se crean las claves primarias de la tabla de relación.



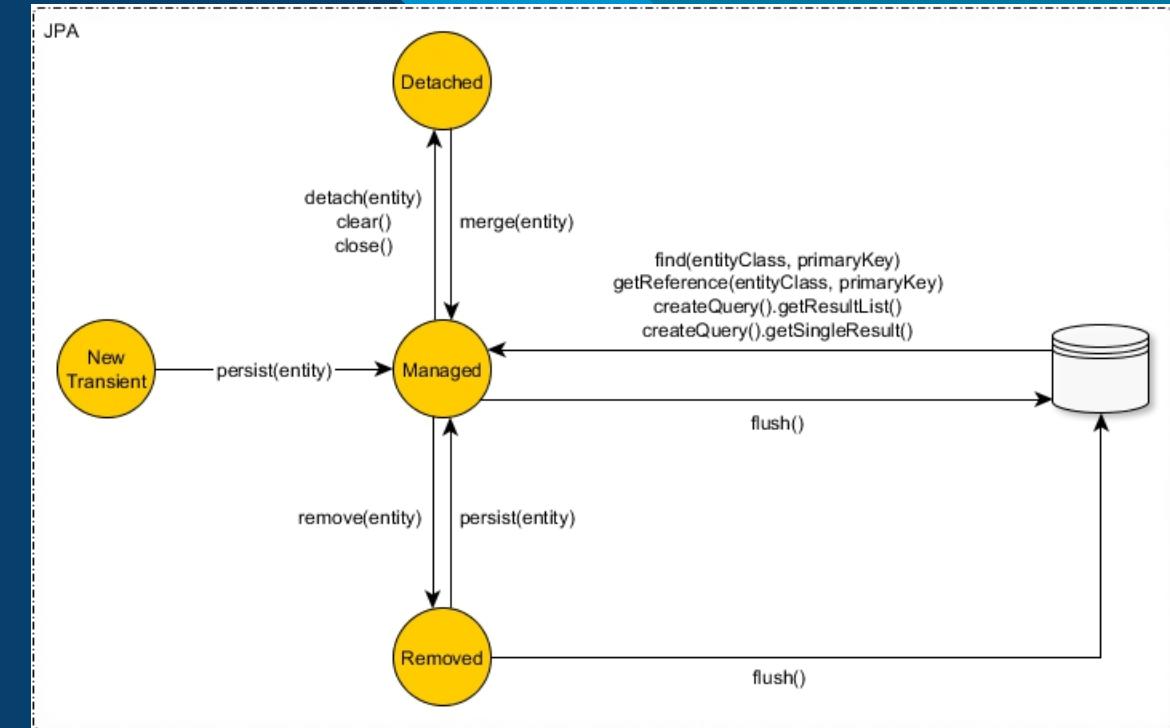
Hibernate y JPA. Relaciones, Transiciones y Carga de Datos

- A la hora de definir una relación, podemos indicar el plan de transición y de carga de datos
- Las transiciones en cascada son una herramienta potente, que nos permiten propagar un cambio de estado desde una entidad padre a otras entidades hijas relacionadas con la primera. Dentro del interfaz javax.persistence.CascadeType tenemos definidos varios tipos:
 - ALL: se propagan todos los cambios de estados.
 - PERSIST: se propagan las operaciones de persistencia.
 - MERGE: se propagan las operaciones merge.
 - REMOVE: se propagan las eliminaciones.
 - REFRESH: se propagan las actualizaciones.
 - DETACH: se propagan las separaciones.
- Fetch plans (planes de carga de datos). Como hemos visto, JPA+Hibernate, a través del contexto de persistencia, ponen a nuestra disposición los datos almacenados en una base de datos. Pero, ¿es necesario que tengamos siempre todos los datos de las entidades que manejamos? Podemos intervenir en este proceso, indicando el tipo de fetching que queremos realizar. Tenemos a nuestra disposición dos esquema de trabajo:
 - EAGER: Hibernate cargará todos los datos de las entidades que sean necesarias (incluidas las de las entidades hijas si hay asociaciones).
 - LAZY: Con el modo perezoso solo se cargarán los datos cuando estos sean realmente necesarios (es decir, cuando se vayan a utilizar).
 - ¡OJO!, debemos tener en cuenta en qué sesión estamos, porque quizás usando LAZY desde diferentes sesiones no podamos acceder a los elementos bajo demanda.



Hibernate y JPA. Relaciones, Transiciones y Carga de Datos

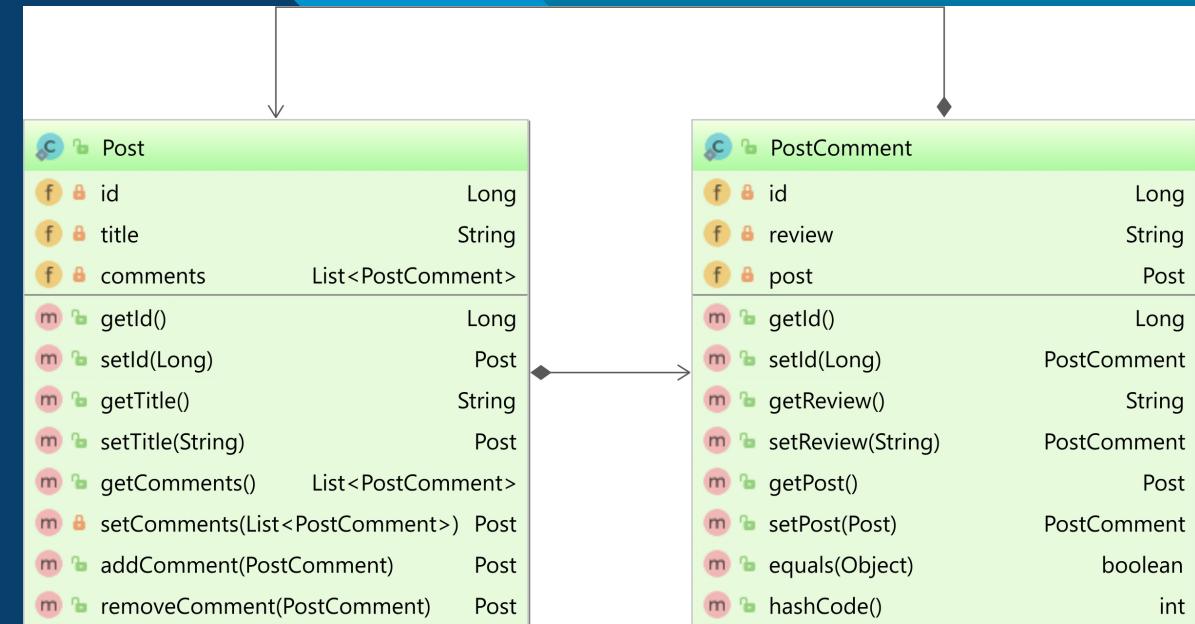
- **orphaRemoval** Nos ofrece la solución para eliminar a los hijos de los cuales no existe una referencia en relaciones Uno a Muchos.
- **orphanRemoval** está ligada al ORM, y “marca” a un hijo para ser eliminado, si **no está referenciado por un padre**. Por ejemplo, si hemos eliminado al hijo de una lista o colección existente en el objeto padre. Ideal para limpiar objetos que no deberían existir si no está el padre.
- **CascadeType.Remove, está ligada a operaciones en la base de datos.** Si borra un padre, se borran todos los hijos, y solo cuando se han borrado los hijos se borra finalmente el padre de la base de datos. Es decir, **solo borra si elemento padre ha sido eliminad**. Es menos agresiva. Pues no automatiza el borrado hasta que no es realizado por el programador.
- Debemos pensar que cuando insertamos los cambios no son vistos hasta que la entidad no se le hace un “Flushed”, esto es automático o manual. Pero cuando eliminamos no es así, no se realizan los cambios hasta que el Contexto de Persistencia es “Flushed”





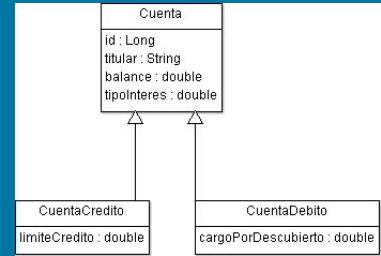
Hibernate y JPA. Relaciones, Transiciones y Carga de Datos

- En esta relación One to Many, uno a muchos.
- **CascadeType.REMOVE (ALL)**, El atributo de cascada le dice al proveedor de JPA que pase la transición del estado de la entidad desde la entidad principal Post a todas las entidades de PostComment contenidas en la colección de comentarios. Entonces, si eliminas la entidad Post, el proveedor de JPA eliminará primero la entidad PostComment, y cuando se eliminen todas las entidades secundarias, también eliminará la entidad Post.
- **orphanRemoval**, el proveedor de JPA programará una operación de eliminación cuando la entidad secundaria se elimine de la colección. El proveedor de JPA eliminará el registro post_comment asociado, ya que ya no se hace referencia a la entidad PostComment en la colección de comentarios.
- **DDL SQL ON DELETE CASCADE**. Todas las entidades post_comment asociadas son eliminadas automáticamente por el motor de base de datos. Sin embargo, esta puede ser una operación muy peligrosa si elimina una entidad raíz por error.



La ventaja de las opciones JPA cascade y orphanRemoval es que también puede beneficiarse del bloqueo optimista para evitar la pérdida de actualizaciones.

Si usas el mecanismo en cascada de JPA, no necesitas usar ON DELETE CASCADE de nivel DDL, que puede ser una operación muy peligrosa si elimina una entidad raíz que tiene muchas entidades secundarias en múltiples niveles.



Hibernate y JPA. Herencia

- **@MappedSuperclass**. En este primer esquema, la clase base no será trasladada a la base de datos. Es decir, la tendremos disponible en nuestra aplicación para trabajar con ella, pero no generará una nueva tabla. Si bien no hemos trabajado aun con consultas, cabe decir aquí que no podremos hacer uso de las consultas polimórficas, y por tanto tendríamos que consultar ambas entidades por separado.
- **@Inheritance(strategy = InheritanceType.SINGLE_TABLE)**, le indicamos a Hibernate que para todas las entidades que participen en la jerarquía de herencia, solamente tiene que crear una tabla en la base de datos. Esta estrategia es adecuada si el número de atributos que añaden las clases extendidas no es muy grande. Hay que tener en cuenta que si tenemos dos entidades, B y C, que extienden a A, la tabla generada tendrá los atributos de A, B y C. Si el número de atributos de B es muy grande, al insertar una entidad de tipo C, todos los atributos que correspondan a B serán nulos.
- **@Inheritance(strategy = InheritanceType.JOINED)** Este esquema de trabajo, también conocido como table per subclass, genera las siguientes tablas:
 - Una tabla para la entidad base de la jerarquía. Tendrá todos los atributos de la clase base.
 - Una tabla para cada entidad extendida de la jerarquía. Tendrá una referencia a la entidad base, y los atributos propios.
- **@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)**. La última opción es la de una tabla por clase. Cada entidad extendida tendrá los atributos de la clase base y los suyos propios (como en **@MappedSuperclass**), pero también existirá una tabla para la clase base. En caso de que queramos realizar consultas polimórficas, se realizaría un UNION entre los resultados de las consultas a las tablas.



Hibernate y JPA. Consultas. JPQL

- JPQL nos va a permitir realizar consultas en base a muchos criterios, así como obtener más de un valor por consulta permitiéndonos hacer uso de parámetros nombrados o con índice: `SELECT e FROM Employee e WHERE e.jobTitle = :jobTitle`
- JPA nos provee de dos interfaces, `javax.persistence.Query` (no va tipado) y `javax.persistence.TypedQuery` (el resultado va tipado), que se obtienen directamente desde el EntityManager.
- Joins explícitos. Como no podía ser de otra manera, JPQL permite realizar los mismos JOIN que en SQL (JOIN, INNER JOIN, LEFT | RIGHT JOIN), si bien uno de los más interesantes el JOIN FETCH. Este join sobrescribe la forma en que se recuperan determinadas asociaciones, evitando las N+1 consultas y sustituyéndolas por un JOIN.
- Joins Impícitos. Un JOIN implícito se realiza siempre a través de alguna de las asociaciones de una entidad, navegando a través de sus propiedades.
- JPQL también nos permite lanzar consultas de actualización de datos, update y delete, pero no insert

```
Query query = em.createQuery(  
    "SELECT e FROM Employee e WHERE e.jobTitle = :jobTitle"  
);  
  
query.setParameter("jobTitle", "Sales Rep");
```

https://www.tutorialspoint.com/es/jpa/jpa_jpql.htm



Hibernate y JPA. Consultas. JPQL

- **@NamedQueries.** Las consultas con nombre son un tipo de consultas especiales ya que, una vez que son definidas, no pueden ser modificadas. Son leídas y transformadas en SQL durante la inicialización del contexto de persistencia. Por ello, son más eficientes y ofrecen un rendimiento mayor. **Se suelen definir mediante anotaciones en las clases entidad**, si bien también pueden declararse en XML.
- Se pueden añadir más de una consulta con nombre, a través de la anotación **@NamedQueries**. Las consultas también pueden recibir parámetros.

```
...
@Entity
@Table(name="customers")
@NamedQueries({
    @NamedQuery(name="Customer.findAll", query="SELECT c FROM Customer c"),
    @NamedQuery(name="Customer.findByName", query="SELECT c FROM Customer c WHERE
c.customerName LIKE :name"),
    @NamedQuery(name="Customer.findByEmployee", query="SELECT c FROM Customer c WHERE
c.employee = :employee"),
})
public class Customer implements Serializable {

}
```

```
...
Query query = em.createQuery(
    "select c from Customer c"
);
List<Customer> listCustomer = (List<Customer>) query.getResultList()

Query query = em.createNamedQuery("Customer.findAll");
List<Customer> listCustomer = (List<Customer>) query.getResultList();
```



Hibernate y JPA. Consultas. SQL Nativo

- JPA e Hibernate también permiten la ejecución de SQL nativo (en particular, el dialecto del RDBMS que estemos usando). Esto es muy útil cuando nuestro sistema tiene funcionalidades específicas, o si nuestra experiencia en SQL nos permite implementar sentencias que sean realmente eficientes.
- JPA nos provee de dos interfaces, `javax.persistence.NativeQuery` (no va tipado) y `javax.persistence.NativeTypedQuery` (el resultado va tipado), que se obtienen directamente desde el EntityManager.
- Por último, decir que también podemos definir consultas con nombre, como ocurría con JPQL. Estas se definen a través de la anotación `@NamedNativeQuery`.

```
...  
  
@Entity  
@Table(name="employees")  
@NamedQuery(name="Employee.findAll", query="SELECT e FROM Employee e")  
@NamedNativeQuery(name="Employee.nativefindAll", query="SELECT * FROM employees",  
resultClass=Employee.class)  
public class Employee implements Serializable {  
    //Resto de código  
}
```

```
...  
  
List<Employee> employeesList = em.createNativeQuery(  
        "SELECT employeeNumber, lastName, firstName, extension, email, officeCode,  
        reportsTo, jobTitle FROM employees",  
        Employee.class).getResultList();  
...  
  
List<Employee> employeesList = em.createNamedQuery("Employee.nativefindAll").getResultList();
```

CRUD con Hibernate/JPA

Create, Read, Update & Delete

CRUD





CRUD con Hibernate/JPA. Entidades

```
Departamento

@Entity
@NamedQuery(name = "Departamento.finAll", query = "SELECT d FROM Departamento d")
public class Departamento {
    private long id;
    private String nombre;
    private Set<Empleado> empleados;

    @Id
    @GeneratedValue(strategy= GenerationType.IDENTITY)
    @Column(name = "id")
    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }

    @Basic
    @Column(name = "nombre")
    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    ...

    @OneToMany(mappedBy = "departamento")
    public Set<Empleado> getEmpleados() {
        return empleados;
    }

    ...
}
```

```
Empleado

@Entity
@Table (name="Empleados")
@NamedQuery(name = "Empleado.porDepartamentoNamed", query = "SELECT e FROM Empleado e WHERE e.departamento.nombre = ?1")
@NamedNativeQuery(
    name = "Empleado.porDepartamentoNative",
    query = "SELECT * FROM Empleado e, Departamento d WHERE d.nombre = ? and e.departamento_id = d.id",
    resultClass=Empleado.class)

public class Empleado {
    private long id;
    private String nombre;
    private String apellidos;
    private Departamento departamento;

    @Id
    @GeneratedValue(strategy= GenerationType.IDENTITY)
    @Column(name = "id")
    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }

    @Basic
    @Column(name = "nombre")
    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    @Basic
    @Column(name = "apellidos")
    public String getApellidos() {
        return apellidos;
    }

    public void setApellidos(String apellidos) {
        this.apellidos = apellidos;
    }

    ...

    @ManyToOne
    @JoinColumn(name = "departamento_id", referencedColumnName = "id", nullable = false)
    public Departamento getDepartamento() {
        return departamento;
    }

    ...
}
```





CRUD con Hibernate/JPA. EntityManager

```
...                                         EntityManager  
  
// Creamos las EntityManagerFactory para manejar las entidades con EntityManager  
// y transacciones con EntityTransaction  
EntityManagerFactory entityManagerFactory = Persistence.createEntityManagerFactory("default");  
EntityManager entityManager = entityManagerFactory.createEntityManager();  
EntityTransaction transaction = entityManager.getTransaction();  
...  
try {  
    ...  
} finally {  
    // Si hay un error y estamos en una transaccion aun activa  
    // No la hemos confirmado nosotros --> rollback  
    if (transaction.isActive()) {  
        transaction.rollback();  
    }  
    // Cerramos nuestra sesi n de EntityManager  
    entityManager.close();  
    entityManagerFactory.close();  
}
```



CRUD con Hibernate/JPA. Selección

Selección

```
// Selección
Query queryDepartamento = entityManager.createQuery("SELECT d FROM Departamento d");
List<Departamento> departamentos = queryDepartamento.getResultList();

TypedQuery<Departamento> namedQueryDepartamento =
entityManager.createNamedQuery("Departamento.finAll", Departamento.class);
List<Departamento> departamentos = namedQueryDepartamento.getResultList();

Departamento departamento = entityManager.find(Departamento.class, 1L);

List<Empleado> empleados = entityManager.createNamedQuery("Empleado.porDepartamentoNamed",
Empleado.class)
.setParameter(1, "TypeScript Departamento")
.getResultList();

List<Empleado> empleados = entityManager.createNamedQuery("Empleado.porDepartamentoNative",
Empleado.class)
.setParameter(1, "TypeScript Departamento")
.getResultList();
```



CRUD con Hibernate/JPA. Insercción

```
...  
Insercción  
  
// Insercción  
// Siempre que modificuemos un objeto y queremos que sea visible lo antes posible  
// debemos hacerlo con una transacción  
transaction.begin();  
Departamento dep = new Departamento();  
dep.setNombre("Prueba " + LocalDateTime.now().toString());  
entityManager.persist(dep);  
transaction.commit();  
  
transaction.begin();  
Empleado insert = new Empleado();  
// pedro.setId(6); --> es autonumérico  
insert.setNombre("Insert " + LocalDateTime.now().toString());  
insert.setApellidos(LocalDateTime.now().toString());  
insert.setDepartamento(departamento);  
entityManager.persist(insert);  
transaction.commit();
```



CRUD con Hibernate/JPA. Actualización

• • •

Actualización

```
// Actualización  
// Si no tenemos el objeto en la sesión, debemos buscarlo  
// Siempre que modifiquemos un objeto y queremos que sea visible lo antes posible  
// debemos hacerlo con una transacción  
Empleado update = entityManager.find(Empleado.class, 6L);  
update.setNombre("Update " + LocalDateTime.now().toString());  
update.setApellidos("Update " + LocalDateTime.now().toString());  
transaction.begin();  
entityManager.merge(update);  
transaction.commit();
```



CRUD con Hibernate/JPA. Eliminación



Eliminación

```
// Eliminación  
// Si no tenemos el objeto en la sesión, debemos buscarlo  
// Siempre que modifiquemos un objeto y queremos que sea visible lo antes posible  
// debemos hacerlo con una transacción  
Empleado delete = entityManager.find(Empleado.class, 6L);  
transaction.begin();  
entityManager.remove(delete);  
transaction.commit();
```

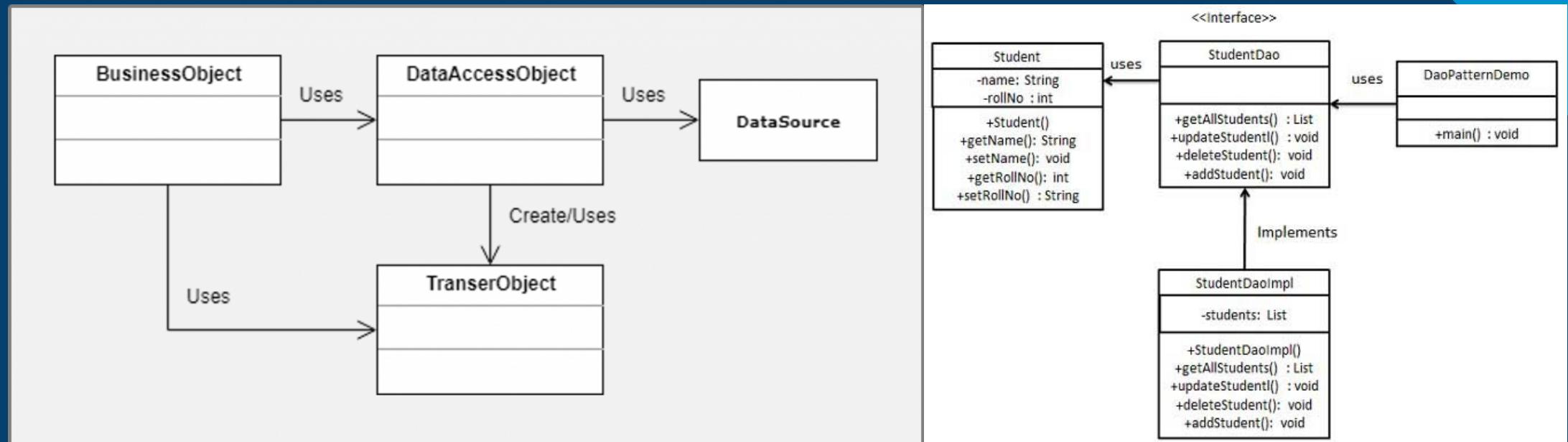
Extra. Cuestiones de Arquitectura y Diseño. Patrones

¿Por qué organizamos todo esto?



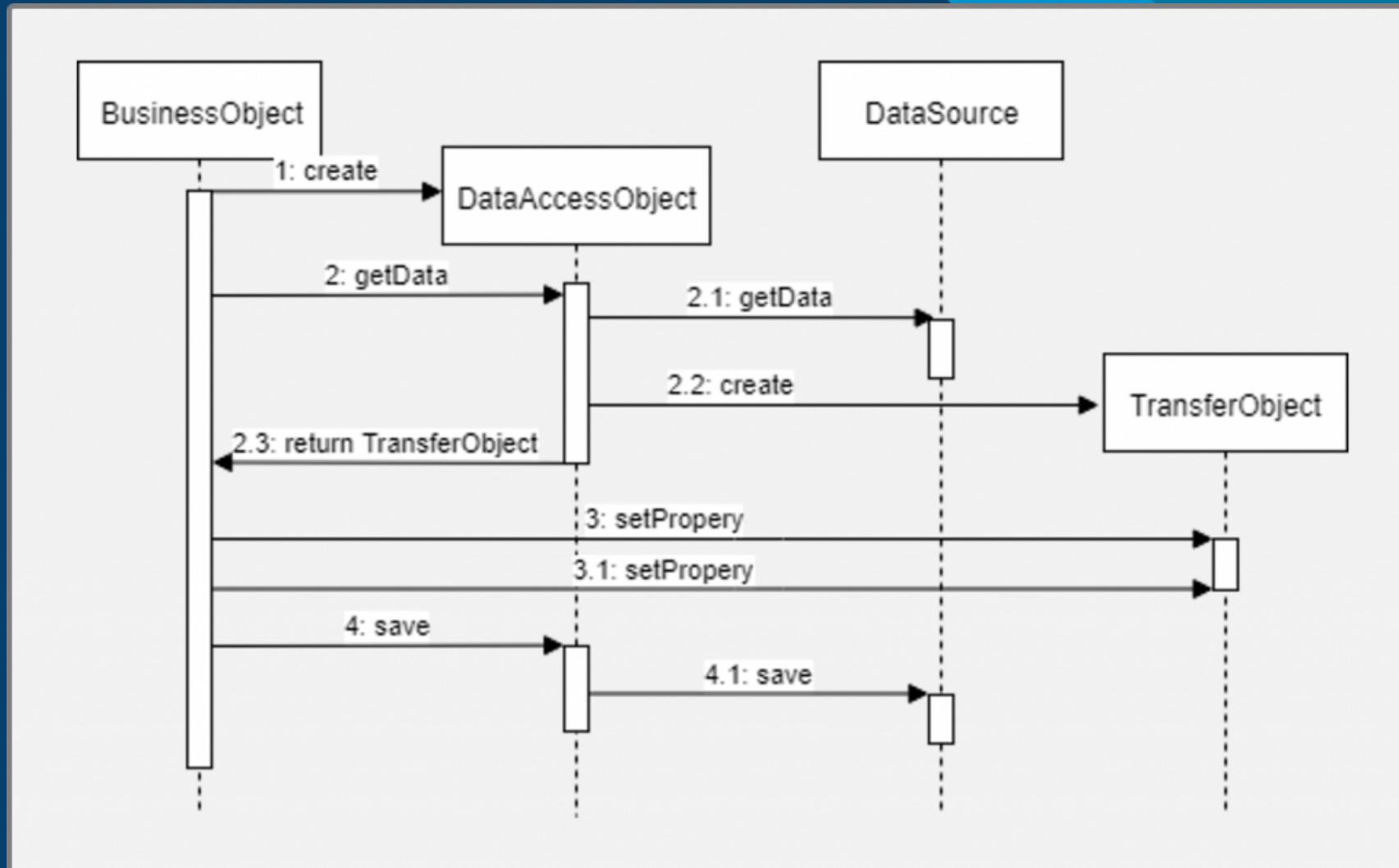
Patrón DAO

- Debemos tener en cuenta que la implementación y formato de la información puede variar según la fuente de los dato.
- El patrón 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.



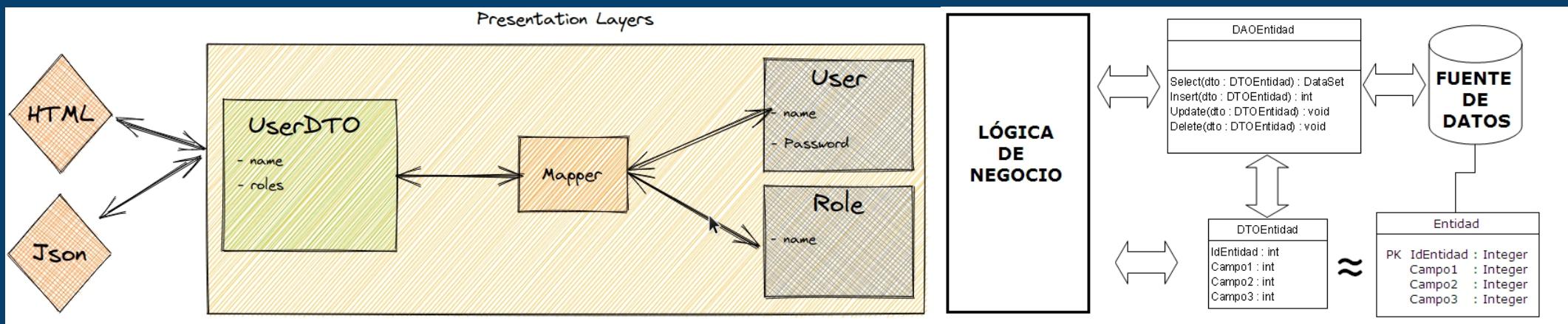


Patrón DAO



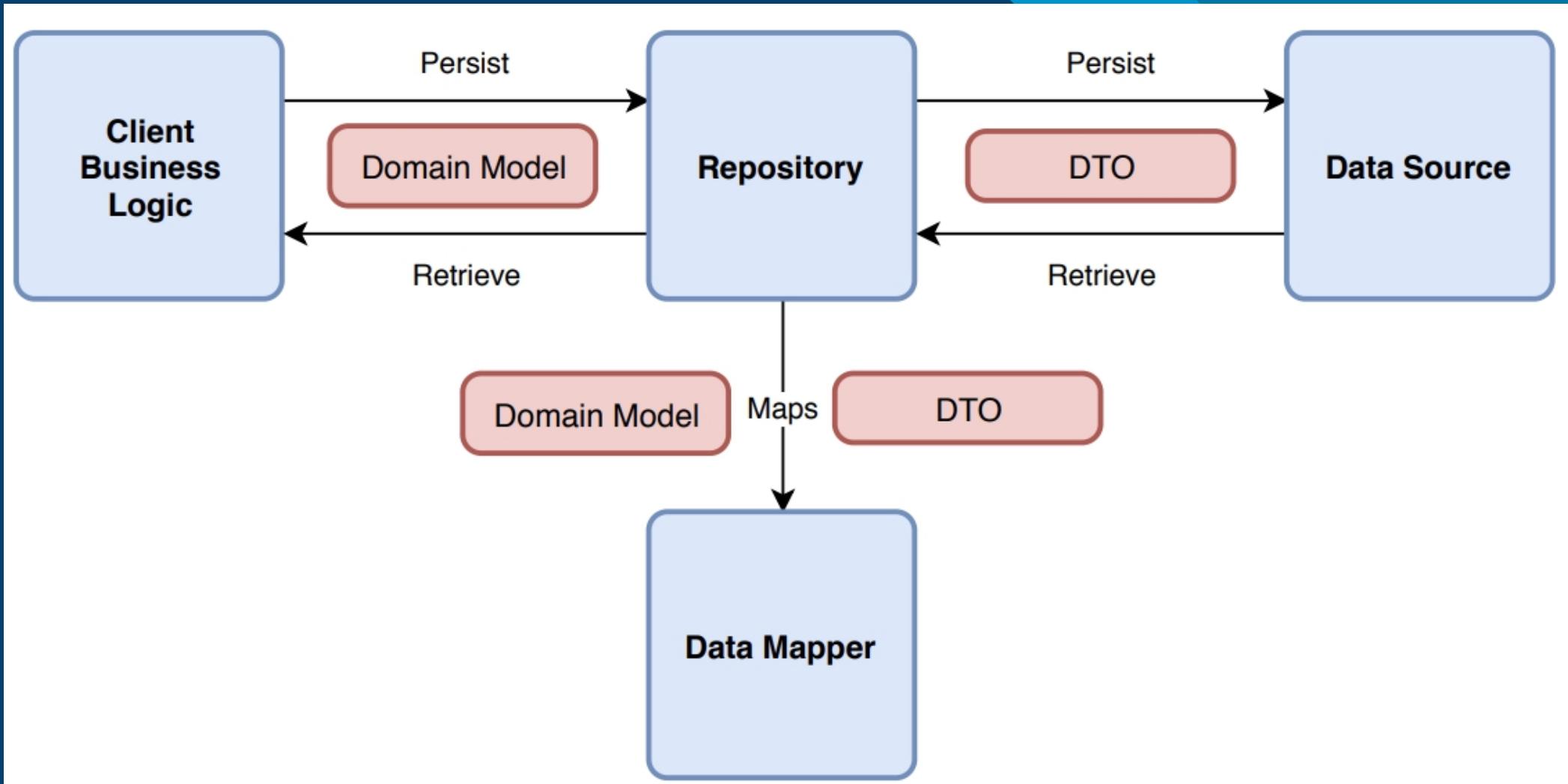
Patrón DTO

- El patrón DTO tiene como finalidad de crear un objeto plano (POJO) con una serie de atributos que puedan ser enviados o recuperados por nuestro servicio y enviados a capas superiores con el objetivo de condensar o adaptar la información para disminuir las trasferencia y con ello respetar nuestro modelo de datos, pues es en el objeto DTO donde realizamos operaciones de trasferencia de datos.
- Por ejemplo en nuestro modelo tenemos usuarios que escriben post. Podemos de una tacada traernos todos los usuarios y sus post en el DTO de usuarios.
- Por otro lado, los Mapper nos ayuda a ensamblar los DTO o desensamblarlos según el modelo de datos que tenemos. Es decir crear un objeto POJO a partir de un objeto DTO o POJO desde objetos DTO.





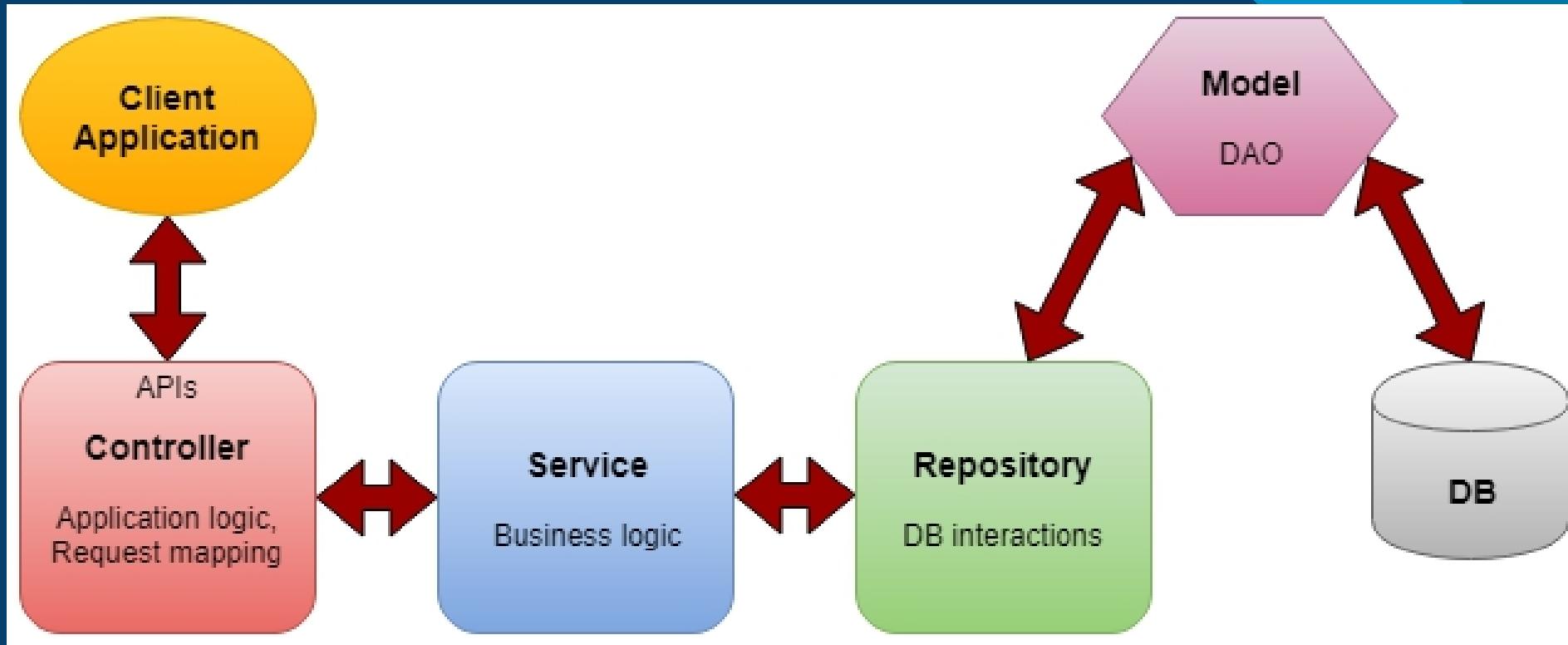
Patrón DTO y Mapper





Patrón CSR: Controlador, Servicio y Repositorio

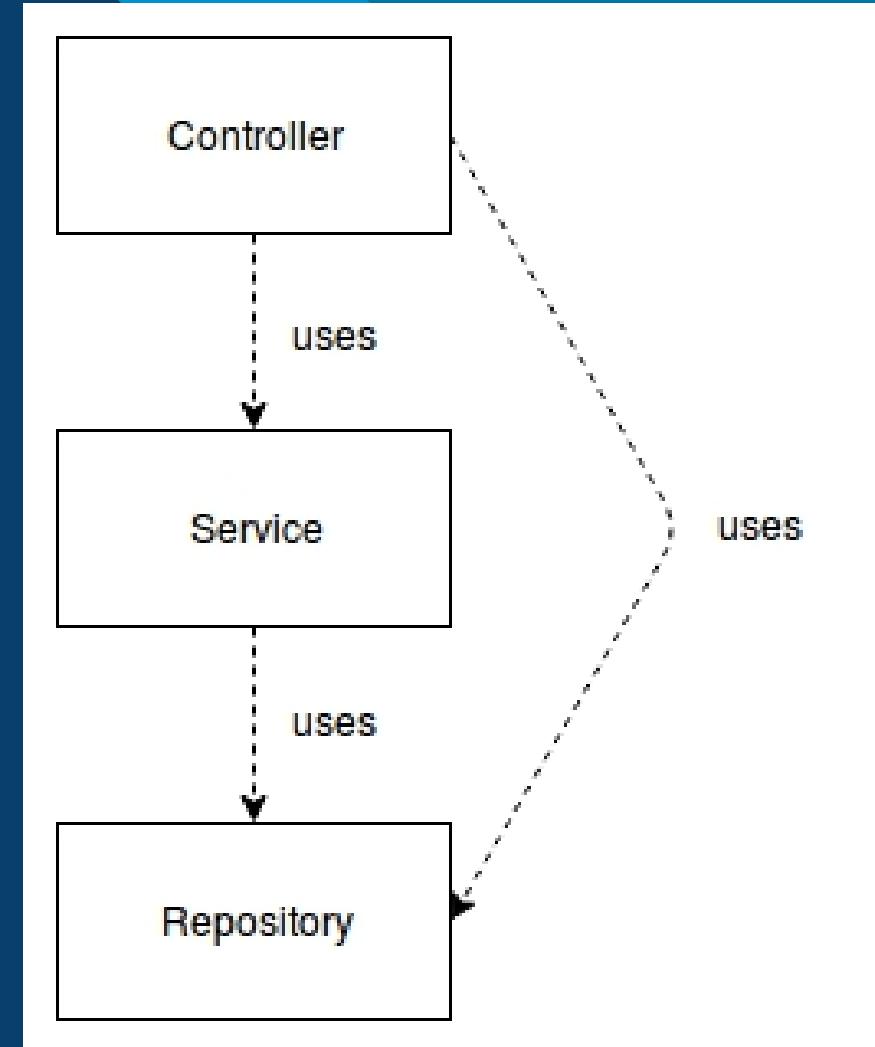
- La arquitectura que seguiremos es tipo CSR (Controladores -> Servicios -> Repositorios) de esta manera cualquier cambio no afectaría a la capa superior, manteniendo nuestra compatibilidad si por ejemplo pasamos de almacenamiento en ficheros XML a bases de datos relacionales o no relacionales. Es una de las arquitecturas más usadas para hacer Backend de aplicaciones distribuidas.





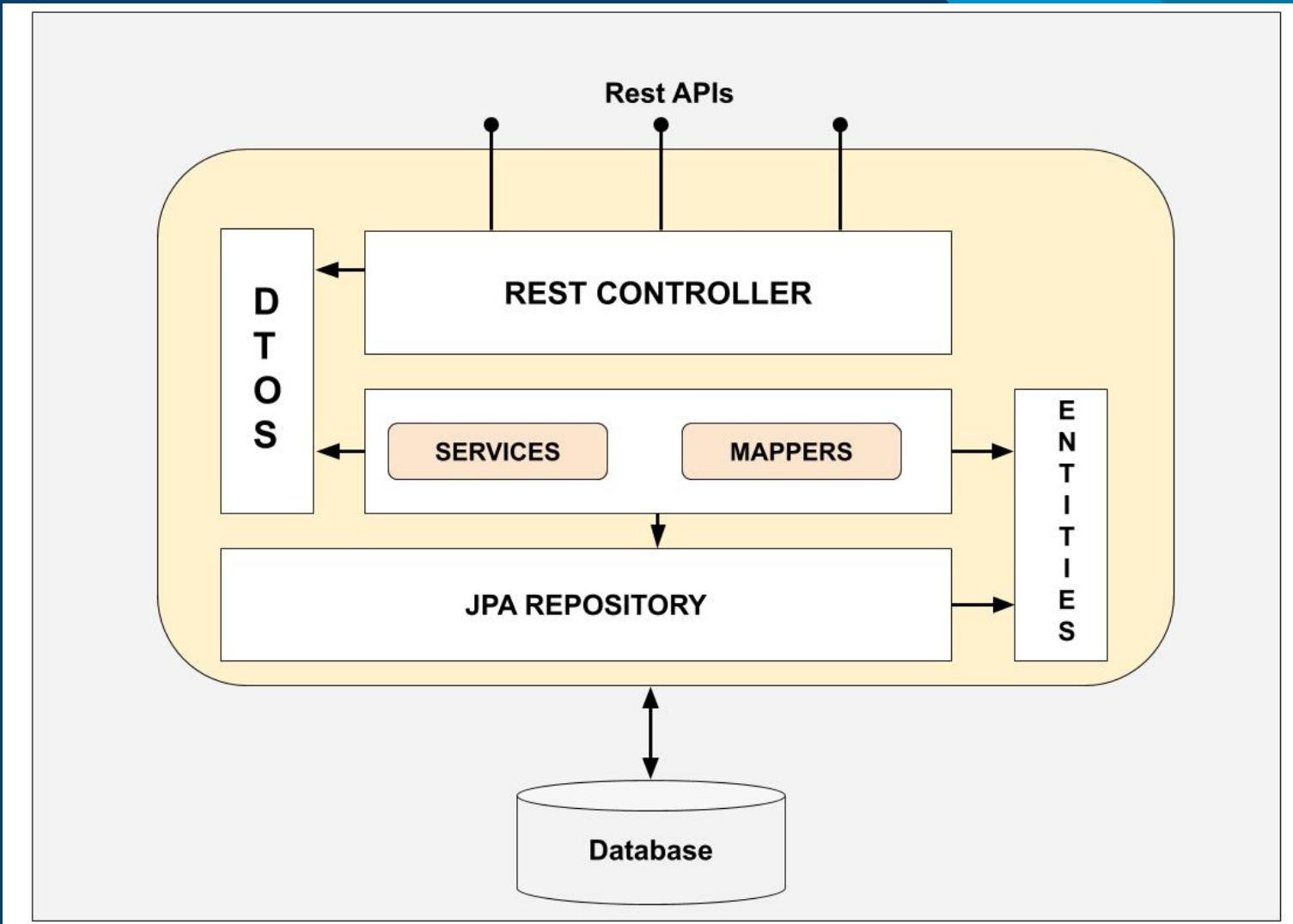
Patrón CSR: controlador, Servicio y Repositorio

- **Controlador.** Tiene la lógica de la aplicación y controla y redirige las distintas peticiones que se nos hacen. Puede hacer uso de uno o varios servicios para procesar la lógica de la aplicación y las distintas peticiones que nos llegan. Pertenece a la capa de presentación.
- **Servicio.** Implementa la lógica de negocio y procesan las peticiones que se nos hacen accediendo a los recursos necesarios para ello usando los repositorios. Son la capa intermedia.
- **Repositorio.** Implementan la lógica de acceso y manipulación de los datos encapsulando dichas operaciones.





Patrón CSR: controlador, Servicio y Repositorio





Repositorio vs DAO ¿Es lo mismo?

- **DAO** implementa las operaciones a más bajo nivel para persistencia y manipulación de la información. DAO es una abstracción de la persistencia de datos. DAO es un concepto de nivel inferior, más cercano a los sistemas de almacenamiento de datos. DAO funciona como una capa de mapeo/acceso de datos.
- **Repositorio** encapsula el propio sistema de almacenamiento, pero no suele estar tan ligado a dicho sistema de almacenamiento. Un repositorio es una abstracción de una colección de objetos. El repositorio es un concepto de nivel superior, más cercano a los objetos de dominio. Un repositorio es una capa entre dominios y capas de acceso a datos, que oculta la complejidad de recopilar datos y preparar un objeto de dominio.
- Se puede dar el caso que un mismo repositorio trabaje con distintos DAOS, por ejemplo las operaciones de manejo de datos de usuarios estén en una base de datos relacional (login, password) y en una NoSQL otra información (nombre, apellidos, email, etc). Es por ello que el repositorio para manejo de usuario llamará por debajo a dos DAOS separados. Pero si la correspondencia es 1 a 1, las ideas son muy similares y podemos optar por uno de ellos.
- Con Hibernate y JPA usamos repositorios, pues ellos encapsulan la lógica de acceso y mapeo objeto relacional, es decir nos pasan una tupla del modelo relacional a un objeto y viceversa.
- Si lo hiciésemos “a mano” hablaríamos de DAO pues somos nosotros los que accedemos a ese fichero XML o base de datos haciendo todo el proceso de implementación de operaciones básicas para el manejo de la información en dicha fuente de almacenamiento.

“

"La programación es una carrera entre los desarrolladores, intentando construir mayores y mejores programas a prueba de idiotas, y el universo, intentando producir mayores y mejores idiotas. Por ahora va ganando el Universo"

- Rich Cook

”



Recursos

- Twitter: <https://twitter.com/joseluisgonsan>
- GitHub: <https://github.com/joseluisgs>
- Web: <https://joseluisgs.github.io>
- Discord: <https://discord.gg/eFMKxfPY>
- Aula Virtual: <https://aulavirtual33.educa.madrid.org/ies.luisvives.leganes/course/view.php?id=247>



Gracias

José Luis González Sánchez

