ACCESO A DATOS

TÉCNICO EN DESARROLLO DE APLICACIONES MULTIPLATAFORMA

El mapeo objeto relacional
ORM
Hibernate

Concepto de mapeo objeto relacional

Java es un lenguaje de programación **orientado a objetos** el cual puede ser **representado en un gráfico de objetos**; mientras que una **base de datos relacional** se representa en un formato tabular usando **filas y columnas**.

Cuando se trata de **grabar un objeto en base de datos**, existen algunas diferencias obvias entre estos dos sistemas; por lo tanto, para solventar este tipo de problemas o diferencias de datos a almacenar, tenemos la **ayuda** del **Mapeo Objeto Relacional**.

Un **ORM** es un **framework** que facilita el almacenamiento de los **objetos** a una **base de datos relacional**.

En una base de datos, el **nivel de elemento más alto** es la **tabla**, dividida a su vez en **filas y columnas**. Una **columna** contiene **valores en un tipo** determinado, y una **fila** contendrá un **conjunto de información** de una tabla determinada.

Un **objeto plano** es equivalente a **una fila en una base de datos relacional**, y básicamente esto es lo que se mapea en un sentido y en otro.

> ***ORM Hibernate Espejo***
>
> *En un ORM como Hibernate son una **réplica a nivel de definición** **de datos:***
>
> - *Lo que tenemos en el lado de **base de datos relacional**,*
> - *Lo que tenemos en el lado de **aplicación.***
>
> *Deberán de **definirse las entidades tal y como** están definidas en la **base de datos** real para que no haya **ningún tipo de discordancia** y se puedan efectuar las operaciones sin ningún tipo de incongruencia.*

Flujo del mapeo objeto relacional

Aplicación Java → Objeto Java → ORM (framework) → Base de Datos

*(objetos en **entidades fáciles de persistir**)*

Ventajas e inconvenientes del mapeo objeto relacional

Java posee una **API de conexión a base de datos**: los drivers de conexión (**JDBC**) para acceder a la base de datos, lo que nos facilita la forma de **consultar dicha información** en base de datos.

Escribiríamos ciertas consultas SQL nativas y obtendríamos un Resulset con el resultado de nuestra Query. Para hacer esto, el desarrollador debe conocer la base de datos a fondo, y saber qué tipo de **relaciones existen entre tablas**, así como el **nombre exacto de las columnas**, **Constrains**, etc.

Si consiguiéramos **realizar las mismas operaciones** con los datos **desde la parte de Java**, la perspectiva cambiaría totalmente. De esta forma tendríamos **abstracción, herencia, composición, identidad** y **muchas características** más en la parte de la aplicación Java que se podrían conseguir igualmente por medio de un framework (algunas de las razones por las que se usan los **ORM**).

Existen **distintos tipos de base de datos**, cada una con diferentes tipos de funciones y tipología de datos definidos. Cuando usamos la conexión **JDBC** debemos tener **en cuenta** este tipo de **diferencias**.

Algunas de las **ventajas de usar un ORM**:

- Mejora en la **eficiencia** del desarrollo.
- Desarrollo más **orientado a objetos**. Mejora del **desarrollo orientado a objetos.**
- Mejora de la **manejabilidad** de los **objetos** de la aplicación.
- Mejora a la hora de realizar diferentes **consultas y procedimientos**.
- Facilidad para **introducir nuevas funciones** (cacheo de información…).
- Mejora de la **manejabilidad**

Algunas de los **inconvenientes**:

- El mapeado automático de las bases de datos **consumen muchos recursos** de sistema.
- Aumento de la memoria de la carga de recursos al introducir esta capa intermedia del ORM.
- La **sintaxis de los ORM** a veces puede **complicarse** si realizamos **consultas muy complejas** mediante las que crucemos varias **tablas** y con diversas **condiciones**.

Fases de mapeo objeto relacional

Arquitectura funcional de un framework ORM

Tienen tres fases de funcionamiento:

- **Fase 1 (dentro de la app)**: Estructura...

  Nos centramos en los **datos del objeto**. Esta fase contiene los **POJO** (**Plain Old Java Object**), las clases simples de Java, las clases de implementación, clases e interfaces con su correspondiente **capa de negocio** de cada aplicativo (a esta capa la podemos llamar **capa servicio** y en ella también encontraremos las distintas **clases DAO**), además de clases orientadas a la **capa de datos** con métodos como **crearObjeto()**, **encontrarObjeto()**, **borrarObjeto()**, etc.

- **Fase 2 (dentro de la app):** Los objetos se transforman para la base de datos...

  Llamada también de **persistencia o mapeo**. Contiene los siguientes **agentes**:

  • **Proveedor JPA**: librería que hace posible toda la funcionalidad de JPA: **javax.persistence**.

  • **Archivo de asignación**: es un **fichero XML** donde se almacena la **configuración** de la asignación de los **datos de una clase JAVA** (**POJO**) y los **datos reales** de la base de datos relacional.

  • **JPA Cargador**: realmente esta parte funciona como una **memoria caché**, **cargará los datos** de la base de datos proporcionando algo parecido a una **copia**; para de esa forma, realizar **interacciones rápidas** con las clases de servicio.

  • **Reja de objeto**: es el lugar donde se **almacenan temporalmente una copia** de los datos de nuestra base de datos relacional. Se le llama objeto **grid**, por lo que todas las **consultas pasarán por este punto**, y una vez realizadas las **verificaciones** pasará a la **base de datos** principal.

- **Fase 3 (exterior de la app – BBDD extrerna)**:

  Llamada **fase de datos relacionales**. Una vez pasada la reja de objetos y todo haya ido bien, se irá directamente **a base de datos**. **Hasta entonces** como hemos mencionado antes, se permanecerá en ese **espacio temporal de caché**.

Esquema arquitectura ORM

FASE 1	FASE 2	FASE 3
Capa servicio	Object Grid	BDD
POJO	JPA Loader	
Entidades	Archivo asign.	
DAO	JPA provider	
	Mapping	

Herramientas ORM

Los **ORM** más **usados en Java**:

• **EBEAN**:

Algunas características son:

• Soporte en bases de datos: soporta bases de datos como **H2, Postgres, Mysql, NuoDB, PostGis, MariaDB, SQL Server, Oracle, SAP**, etc.

• Múltiples niveles de abstracción: **consultas** de tipo **ORM** mezcladas con **SQL**, además de consultas **DTO** (Data Transfer Object).

• **Beneficios**: evita automáticamente N+1, usa **caché** de tipo **L2** para reducir carga de base de datos, realiza consultas **mezclando base de datos y cacheado L2**, ajusta automáticamente las consultas ORM, contiene **tecnología Elasticsearch para caché L3**, etc. Elasticsearch es un motor de búsqueda y analítica de RESTful distribuido capaz de abordar un número creciente de casos de uso.

• **IBATIS**:

Un ORM que aparece de la mano de **Apache Software Foundation**. En 2010 el desarrollo se centralizó en "Google Code", usando el nuevo nombre MyBatis. Posee **soporte para Java y .Net**. Algunas de sus características son:

• Posee la opción de **dividir** la **capa de persistencia** en:

capa de **abstracción**,
capa de **framework persistente**,
capa de **Driver**.

• Una de sus virtudes es la facilidad de **interactuar con los objetos** y los **datos** de las bases de datos **relacionales**.

• Ofrece **abstracción** a nivel de la **capa de persistencia** de **objetos**.

• **HIBERNATE**:

El ORM **más extendido y más usado**. Disponible para lenguaje Java y también **para .Net** (denominándose para éste, **Nhibernate**). Facilita el **mapeo relacional** de los distintos objetos entre una base de datos relacional y el modelo de objetos de la aplicación; para lo que se apoya en un **fichero .xml** que representa y establece dichas relaciones o también por medio de **anotaciones** donde se establecen las relaciones. Algunas de sus características:

• **Simplicidad**: al disponer de un solo fichero .xml para establecer las relaciones, es muy sencillo e intuitivo tanto dirigirnos a éste como consultar cualquier tipo de relación entre entidades o atributos.

• **Robusto**: dispone de muchas **características adaptadas al lenguaje Java**: colecciones, herencia, abstracción, orientación a objetos, etc. En la capa de abstracción ofrece una propia capa de **consultas SQL llamada HQL**, orientada a **facilitar la sintaxis** y a mejorar la **eficiencia** de estas.

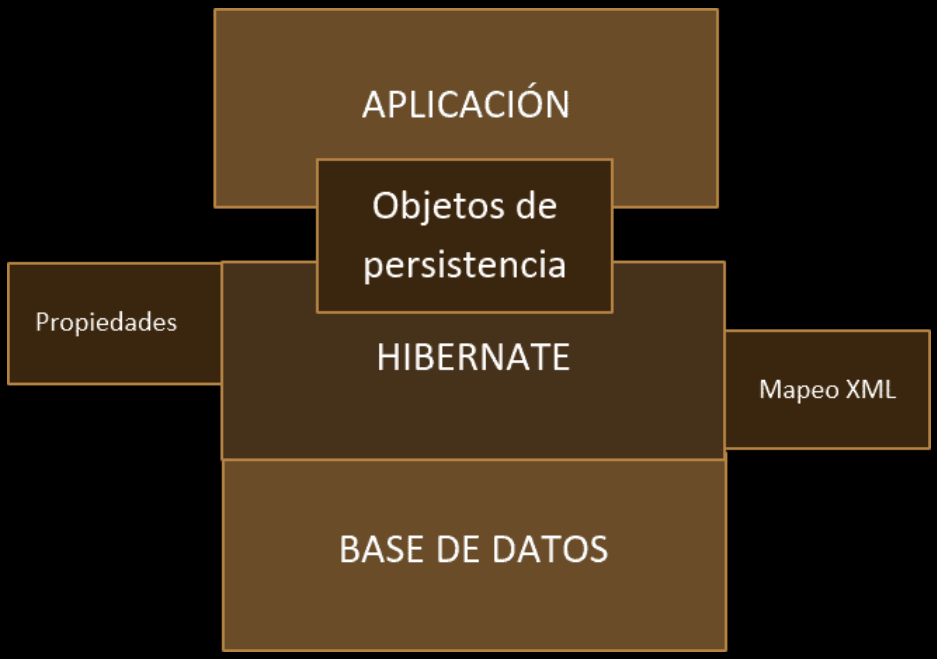
Definición de la arquitectura de Hibernate

Servicio de **alta eficiencia** de **mapeo** objeto-relacional y con **funcionalidades** bastante **potentes de consulta**. Empezaremos aprendiendo sobre su **arquitectura** y avanzaremos en la **instalación y configuración** de este Framework en aplicación Java.

La arquitectura de Hibernate incluye **distintos objetos** como son los objetos de **persistencia**, **sesiones**, **transacciones**, entre otros.

Usa el **principio** de **reflexión Java:** Permite el **análisis y la modificación** de los distintos **atributos** y características de las distintas clases **en tiempo de ejecución**.

Arquitectura Hibernate y diferentes capas que lo conforman:



Capas de **Aplicación e Hibernate unidas** por los **Objetos de persistencia:**

> porque en una **parte específica de la aplicación** se da cierta **conversión** (fichero de **mapeo**), dónde la información fluye y es mapeada desde dichos ficheros persistentes **a la base de datos**.

Capa **Hibernate:**

> **→** Realiza la **conexión con el driver**,
>
> → Se cargan:
>
> > **configuraciones Hibernate**,
> >
> > todas las **entidades** previamente diseñadas.

Componentes de Hibernate

Piezas clave que hacen **posible el mapeo objeto relacional**:

Algunos de sus **componentes principales** son:

- **SessionFactory Object**: mediante el que se permitirá el uso de **objetos** de tipo **Session**. Este objecto Java se puede instanciar de diferentes formas: normalmente **coge la configuración** existente en el **fichero de configuración** establecido por defecto. Utilizaremos un **objeto SessionFactory** por **cada base de datos** que tengamos en la aplicación.

- **Session Object**: utilizaremos dicho objeto para instanciar una **conexión directa** con nuestra base de datos relacional. Es un objeto no muy pesado y su función principal es **interactuar con la base de datos**. Este tipo de objetos **no deben permanecer abiertos mucho tiempo** por temas de **seguridad**.

- **Transaction Object**: es un objeto opcional, que básicamente **maneja las transacciones** directamente con las **bases de datos** relacionales. Si no se quiere hacer uso de dicho objeto, también **podemos indicar manualmente** aquellos **bloques** que queremos que sean **transaccionales**. Recordemos de temas anteriores la definición de transacción orientada al bloque de operación/es, cuyo objetivo consistía en **persistir** todas y cada una de las operaciones que contenían dicho bloque, **o** por el contrario, realizar **rollback** (marcha atrás) en dicha operación.

- **Query Object**: en Hibernate disponemos de varias formas de realizar **consultas** a la base de datos. Este tipo de objetos utilizan consultas de tipo SQL o de tipo **Hibernate Query** (**HQL**). Con este tipo de objetos **enlazaremos los distintos parámetros de nuestra consulta**, podremos realizar ciertas restricciones como controlar el número de resultados, y ejecutar la consulta. Es un modo de realizar consultas **mucho más dinámico que** las consultas **nativas**.

- **Criteria Object**: desaparecerá el lenguaje nativo de SQL para dar paso a las **consultas** por medio de **objetos Java**, y por medio de las funciones que nos ofrece Hibernate, que más tarde serán **traducidas a sentencias SQL**.

Instalación de Hibernate

Vamos con la instalación del framework **ORM** (Object Relational Mapping) que nos acompañará en el tema, y que nos ayudará a desarrollar ese concepto de **mapeo objeto – relacional**, denominado **Hibernate**, con **SpringBoot** en una **aplicación web Java:**

Dependiendo del **tipo de proyecto** Java que tengamos construido, podremos instalar Hibernate **de diferentes formas**.

Básicamente podremos añadir el .**jar a nuestro proyecto** con la versión específica de Hibernate que nos hayamos descargado. Si por el contrario estamos trabajando en un **proyecto web** con gestor de **dependencias maven**, podremos **agregar nuestra dependencia** de la **versión** determinada de **Hibernate**, y al **compilar**, se **descargará** dicha **librería** y tendremos las **clases del framework** disponibles para su uso.

Ahora, realizaremos la instalación agregando la dependencia determinada usando **Spring Boot**, considerando que estamos realizando una **aplicación desde cero**, a la cual queremos agregar Hibernate. Si éste **no fuese el caso**, nos podríamos quedar simplemente con la parte de **agregar la dependencia**.

Como ya hemos hecho anteriormente con las bases de datos embebidas, nos dirigiremos a la **web inicializadora** de **arquetipos** de **Spring Boot**. Hoy en día disponemos de la siguiente URL:

https://start.spring.io/

Una vez hemos ingresado en la URL, nos centraremos solo en la parte de añadir el framework que vamos buscando, ya que la composición del proyecto SpringBoot la vimos en temas anteriores. Hacemos clic en **Add dependencies** (añadir dependencias).

Buscando por Hibernate, añadiremos directamente **JPA con Sping Data**, e **Hibernate** y nuestras dependencias serían:

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

Aquí podemos observar la **dependencia agregada**. Si por el contrario quisiéramos agregar Hibernate **sin tener a Spring Boot** de mediador, podríamos dirigirnos a la web oficial de **Hibernate** y encontrar allí la **última versión para la dependencia** adecuada.

Configuración de Hibernate

Continuaremos con nuestra aplicación en Spring Boot y, por lo tanto, para realizar cambios de **configuración** nos dirigiremos al fichero "**application.properties**" de nuestro proyecto. A continuación, comentaremos algunos de los aspectos más importantes y usados en la configuración del framework instalado Hibernate.

Comunicación con base de datos:

```
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
```

Con esta línea estamos estableciendo un nexo de unión entre nuestro framework Hibernate y la base de datos.

Una vez situados en el **fichero application.properties** dispondremos de la línea que hemos colocado en la arriba, comunicando nuestro **framework con la base de datos** que tenemos instalada. En este caso es una **base de datos H2**, pero si tuviéramos una base de datos distinta, cambiaríamos y **adaptaríamos la línea en función a la base de datos que tengamos**.

Con la siguiente configuración, estaremos indicando que **habilitamos las trazas de tipo SQL**, y además, que se muestren con el **formato SQL** correspondiente.

Trazas SQL:

```
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
```

Con las siguientes líneas de configuración, estaremos habilitando los **logs de Hibernate a true**; de esta forma podremos ver un **log** extenso sobre los **errores que se vayan dando**, así como las distintas **líneas de log** que **queramos** mostrar por nuestra cuenta. En la imagen también podemos observar que el **nivel de log** está establecido a **debug** (también podremos establecerlo a **nivel info**).

Logs:

```
spring.jpa.properties.hibernate.generate_statistics=true
logging.level.org.hibernate.type=trace
logging.level.org.hibernate.stat=debug
```

Con esta última línea, estaremos permitiendo a Hibernate **crear manualmente distintas entidades** que queramos generar en un fichero que habría que crear ("**schema.sql**"), y más tarde popular dichas tablas con otro script ("**data.sql**").

Inicialización BDD:

```
spring.jpa.hibernate.ddl-auto=none
```