ORM

El mapeo objeto-relacional (**Object-Relational Mapping**) es una técnica de programación que permite trabajar con objetos Java en lugar de tablas de una Base de Datos. Esto permite abstraernos de los detalles concretos de la implementación de la BBDD, lo que facilita el desarrollo y el mantenimiento de los proyectos.

Para entender el potencial de ORM, supongamos que tenemos esta tabla con los Usuarios de una empresa:

ID	Nombre	Apellido
1	Ana	Sanz
2	Juan	Torres
3	Luis	Pérez

Para retornar la información en **JDBC** haríamos:

```
String sql = "SELECT * FROM usurios WHERE id = 1";
ResultSet rs = statement.executeQuery(sql);
Persona persona = new Persona();
persona.setId(rs.getInt("id"));
persona.setNombre(rs.getString("nombre"));
persona.setEdad(rs.getInt("edad"));
```

Pero, utilizando un **ORM (JPA)**, simplemente haríamos algo como:

```
Persona persona = entityManager.find(Persona.class, 1);
```

A efectos prácticos, puedes considerar que esta sentencia es equivalente a lo que has escrito en ORM. Accede a la tabla, recupera la fila de id = 1, recupera la información y la transforma en un objeto.

Estándares e implementaciones

El **JPA** (*Java Persistence API*) no es un ORM en sí, sino una especificación que define cómo deben comportarse los ORM en Java. Podríamos considerarlo el estándar que otros ORM deben implementar. El más utilizado es **Hibernate**, pero hay otros, como EclipseLink.

Nótese que, aunque usemos Hibernate para operar contra una BBDD, esto no impide utilizar las funciones y métodos disponibles para JPA.

Anotaciones en Hibernate

Para trabajar con un ORM moderno (y más tecnologías) necesitamos conocer un juego de **anotaciones** que podemos usar al incluir las librerías en nuestro proyecto. Estas anotaciones son instrucciones especiales en el código Java que le dicen al framework cómo debe mapear las clases y sus atributos a las tablas y columnas de la base de datos.

Es decir, siguiendo el ejemplo anterior, le dice al framework que el objeto Persona está relacionado con tabla Usuarios; que el atributo id equivale a la columna ID; etc.

Las anotaciones más importantes son:

- @Entity: La clase es una entidad persistente (tabla).
- @Table(name="nombre_tabla"): Define el nombre de la tabla.
- @Id: Define el atributo como una clave primaria.
- @OneToOne: Define un atributo como parte de una relación 1:1
- @OneToMany: Define un atributo como parte de una relación 1:N
- @ManyToOne: Define un atributo como parte de una relación N:1
- @ManyToMany: Define un atributo como parte de una relación N:M

Virtualmente cualquier cosa que puedas hacer con SQL en una Base de Datos relacional convencional, puede hacerse con la anotación adecuada, lo que incluye Join, claves auto incrementables, borrados en cascada…

Un ejemplo simple:

```
@Entity
@Table(name = "usurios")
public class Pedido {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String nombre;

    private int edad;

}
```

Estructura de Hibernate

A nivel fundamental, cuando incluimos Hibernate en nuestro proyecto tenemos que indicar lo siguiente:

- Las clases Java que mapean la BBDD.

- El fichero de mapeo (.xml) que especifica ese mapeo.

PERO si usas anotaciones, ya no es necesario el fichero de mapeo, porque las propias anotaciones se encargan de hacer el trabajo. Simplemente, que sepas que las implementaciones antiguas de Hibernate aún tienen este fichero.

La pieza más importante de Hibernate es la clase **Session**. Representa una conexión entre tu aplicación y la BBDD, gestionada por el framework. Es equivalente a una conexión de JDBC, pero con muchas más funcionalidades. Por ejemplo:

- Hibernate se encarga de abrir y cerrar **conexiones** físicas a la BBDD, por lo que el programador NO tiene que preocuparse de ello.

- Controla el **estado** de los objetos: si están guardados, modificados, etc.

- Administra **transacciones** con la BBDD.

- Permite hacer **consultas HQL**, **SQL nativo** o **Criteria API** contra la BBDD. Importante el HQL, el lenguaje de consultas de Hibernate. Muy parecido al SQL, trabaja con clases y atributos en lugar de con tablas y columnas.

Un ejemplo más elaborado de un insert con Hibernate:

```
SessionFactory sessionFactory = new Configuration()
    .configure().buildSessionFactory();
Session session = sessionFactory.openSession();
session.beginTransaction();

Persona p = new Persona("Ana", 25);
session.save(p);
session.getTransaction().commit();
session.close();
```

A simple vista, parece que el objeto Persona se guarda en la BBDD cuando hacemos save (), confirmamos con el commit, y el close () libera la conexión con la BBDD. Pero es más complicado que eso.

Estados de los objetos en Session

Los objetos mapeados contra la BBDD que están en una sesión tienen diferentes estados que hay que conocer:

- **Transient**: El objeto existe solo en memoria (no en BD).
- **Persistent**: Está siendo gestionado por la Session. Si cambia, Hibernate lo detecta y lo sincroniza.
- **Detached**: La sesión se cerró, el objeto ya no está sincronizado.
- **Removed**: Está marcado para ser eliminado al hacer commit.

Por tanto, volviendo al ejemplo de antes:

```
SessionFactory sessionFactory = new Configuration()
    .configure().buildSessionFactory();
Session session = sessionFactory.openSession();
session.beginTransaction();

Persona p = new Persona("Ana", 25); // Transient
session.save(p);                    // Ahora es Persistent
session.getTransaction().commit();
session.close();                    // Ahora es Detached
```

Cuando un objeto está **Persistent** significa que:

- Está **asociado a una Session activa** de Hibernate.
- Hibernate **vigila los cambios** que haces en ese objeto.
- Cuando se hace un **commit ()** o un **flush ()**, Hibernate **sincroniza automáticamente** esos cambios con la base de datos.

Esto es útil cuando, por ejemplo, realizas múltiples cambios sobre el objeto Persona. Hibernate estará al tanto de todo esto, y cuando se haga el commit, realizará un update () si es necesario, aunque NO se lo digas explícitamente. De la misma forma, si haces save () pero no un commit, todos los cambios que hagas a posteriori sobre Persona se considerarán un update en cuando se diga commit. Obviamente, mientras no hagas un close ().