# Gestión de reservas

Una introducción a la construcción de backends usando Spring Boot.

## Introducción

Este tema está dedicado a las herramientas de mapeo objeto-relacional.

En este tema aprenderemos:

1. Concepto de mapeo objeto relacional. Características de las herramientas ORM.
2. Herramientas ORM más utilizadas.
3. Instalación de una herramienta ORM. Configuración.
4. Estructura de un fichero de mapeo.
5. Elementos, propiedades. Clases persistentes. Mapeo de colecciones, relaciones y herencia.
6. Sesiones; estados de un objeto. Carga, almacenamiento y modificación de objetos.
7. Consultas SQL embebidas.
8. Lenguajes propios de la herramienta ORM.
9. Gestión de transacciones.

## ¿Qué es el mapeo objeto-relacional?

El mapeo objeto-relacional (más conocido por su nombre en inglés, Object-Relational mapping, o sus siglas O/RM, ORM, y O/R mapping) es una técnica de programación para convertir datos entre el sistema de tipos utilizado en un lenguaje de programación orientado a objetos y la utilización de una base de datos relacional como motor de persistencia. En la práctica esto crea una base de datos orientada a objetos virtual, sobre la base de datos relacional. Esto posibilita el uso de las características propias de la orientación a objetos (básicamente herencia y polimorfismo). FUENTE: Wikipedia

Básicamente se trata de una interfaz que nos sirve para trasnformar representaciones de datos del sistema gestor de base de datos relacional (SGBDR) a representaciones de objetos y viceversa.

## Herramientas más utilizadas

### Interfaz JPA

En Java, la Java Persistence API (JPA) es la manera más común de hacer persistentes objetos sin la necesidad de usar JDBC y gestionarlo manualmente.

En Python tenemos herramientas ORM como SQLAlchemy. Otros frameworks como Django incorporan su propia herramienta ORM.

### Implementaciones

Como ya hemos dicho antes, JPA es una interfaz, una capa más de abstracción que añadimos a nuestra aplicación para simplificar nuestro trabajo. Para llevarlo a cabo podremos elegir entre un vasto abanico de posibilidades de implementación como:

- Hibernate
- EclipseLink
- Enterprise JavaBeans
- MyBatis
- y un largo etcétera

Para finalizar, recuerda que probablemente también deberás indicar el driver de la base de datos (para el conector), pues en otra capa por debajo estará JDBC y también podríamos hacer uso de otras APIs de Java como JTA (Java Transaction API) y JNDI (Java Naming Directory Interface).

### Cómo funciona JPA

JPA se basa en un gestor de entidades o EntityManager que, **por introspección**, determina a partir de las clases entidad con las que trabaja, en qué tablas del sistema gestor de bases de datos relacional están almacenados.

**Ficheros de mapeo** Normalmente usaríamos un fichero XML donde se hace la *traducción* de clase a tabla y de atributos de clase a campos de la tabla. Ahora, gracias a las nuevas herramientas de mapeo, es posible hacerlo con anotaciones directamente en el propio código Java, lo que simplifica enormemente la tarea.

**Fichero persistence.xml:**

Para indicar al **EntityManager** qué base de datos usar, cómo conectarse a ella y qué clases entidad están relacionadas con qué tablas, necesitamos crear el fichero de persistencia *persistence.xml*:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" 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_1.xs
  <persistence-unit name="GestionPedidosv2PU" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>com.iesvdc.acceso.entidades.Pedido</class>
    <class>com.iesvdc.acceso.entidades.Cliente</class>
    <class>com.iesvdc.acceso.entidades.Producto</class>
    <class>com.iesvdc.acceso.entidades.LineaPedido</class>
    <exclude-unlisted-classes>false</exclude-unlisted-classes>
    <properties>
      <property name="javax.persistence.jdbc.url"
        value="jdbc:mysql://localhost:3306/damjdbc00?zeroDateTimeBehavior=convertToNull"/>
      <property name="javax.persistence.jdbc.user" value="damjdbc00"/>
      <property name="javax.persistence.jdbc.password" value="damuser"/>
      <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
    </properties>
  </persistence-unit>
</persistence>
```

El él definimos una unidad de persistencia *GestionPedidos2PU* por debajo gestionado por EclipseLink.

Después declaramos las clases entidad (a veces llamadas también POJO, aunque en este contexto no lo son realmente).

A continuación, fíjate cómo se declara la cadena de conexión, usuario y contraseña para conectar a la base de datos MySQL (driver MySQL).

**El EntityManager** Para trabajar con un gestor de entidades, primero hemos de cargar la configuración en una **factoría** que usaremos a lo largo de toda la ejecución del programa. Ejemplo de cómo crear la factoría a partir de la unidad de persistencia creada en el ejemplo anterior:

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("GestionPedidosv2PU");
```

Una vez definida la factoría, podemos usarla para instanciar el EntityManager que nos permitirá almacenar nuestros objetos en la base de datos. Ejemplo de insertar (CREATE en el CRUD):

```
public boolean persist(Object object) {
    boolean resultado=true;
    EntityManager em = emf.createEntityManager();
    try {
        em.getTransaction().begin();
        em.persist(object);
        em.getTransaction().commit();
    } catch (Exception e) {
        Logger.getLogger(getClass().getName()).log(Level.SEVERE, "exception caught", e);
        em.getTransaction().rollback();
        resultado=false;
    } finally {
        em.close();
    }
    return resultado;
}
```

Ejemplo de borrado (DELETE en el CRUD):

```
// Cliente c;
try {
    EntityManager em = emf.createEntityManager();
    em.getTransaction().begin();
    em.remove(c);
    em.getTransaction().commit();
} catch (Exception e) {
    em.getTransaction().rollback();
} finally {
    em.flush();
    em.close();
}
```

Ejemplo de lectura (READ en el CRUD):

```
EntityManager em = emf.createEntityManager();
List<Cliente> lc = em.createNamedQuery("Cliente.findAll").getResultList();
// hacemos algo con la lista....
em.close();
```

Ejemplo de actualización (UPDATE en el CRUD):

```
// Cliente c;
try {
    EntityManager em = emf.createEntityManager();
    em.getTransaction().begin();
    em.merge(c);
    em.getTransaction().commit();
} catch (Exception e) {
    em.getTransaction().rollback();
} finally {
    em.flush();
    em.close();
}
```

## Clases persistentes. Mapeo de colecciones, relaciones y herencia.

### Clases entidad:

Las clases de entidad son sencillas clases Java (de ahí que se confundan con los POJOs) donde hacemos una serie de anotaciones (por eso no es lo mismo que un POJO) para que el EntityManager sepa asociar los objetos con las tablas del SGBDR.

Ejemplo de clase entidad:

```
@Entity
@Table(name = "cliente")
@XmlRootElement
@NamedQueries({
    @NamedQuery(name = "Cliente.findAll",
        query = "SELECT c FROM Cliente c"),
    @NamedQuery(name = "Cliente.findByIdCliente",
        query = "SELECT c FROM Cliente c WHERE c.idCliente = :idCliente"),
    @NamedQuery(name = "Cliente.findByNombre",
        query = "SELECT c FROM Cliente c WHERE c.nombre = :nombre"),
    @NamedQuery(name = "Cliente.findByApellido",
        query = "SELECT c FROM Cliente c WHERE c.apellido = :apellido"),
    @NamedQuery(name = "Cliente.findByDireccion",
        query = "SELECT c FROM Cliente c WHERE c.direccion = :direccion")})
public class Cliente implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Basic(optional = false)
    @Column(name = "id_cliente", nullable = false)
    private Integer idCliente;
    @Basic(optional = false)
    @Column(name = "nombre", nullable = false, length = 25)
    private String nombre;
    @Basic(optional = false)
    @Column(name = "apellido", nullable = false, length = 50)
    private String apellido;
    @Basic(optional = false)
    @Column(name = "direccion", nullable = false, length = 80)
    private String direccion;
    @OneToMany(mappedBy = "idCliente")
    private List<Pedido> pedidoList;
    public Cliente() {
    }

    // GETTERs, SETTERs y constructores...

}
```

A continuación explicamos algunas de las anotaciones más importantes:

- @Table: enlaza la clase entidad con su tabla.
- @Entity: sirve para indicarle al EntityManager que es una clase entidad.
- @Id: Muy últil con los campos "ID" **AUTO_INCREMENT**.
- @Column: Mapeo atributo-columna tabla.

- @JoinColumn: Junto con las anotaciones @OneToOne o @ManyToOne, indica la clave primaria a quien se hace referencia.
- @OneToMany: Clave foránea, relación uno a muchos. Permite indicar el atributo mapeado en la entidad destino (en el lado "muchos").
- @ManyToOne: Relación muchos a uno.

### Consultas SQL embebidas

Con JPA es muy sencillo generar consultas embebidas. En el ejemplo anterior de Cliente.java, recuerda una anotación que había junto a la definición de la clase:

```
@NamedQuery(name = "Cliente.findByNombre", query = "SELECT c FROM Cliente c WHERE c.nombre = :nombre")
```

Una vez definida esta consulta, ahora es posible usarla desde el EntityManager como sigue:

```
List<Cliente> lc = em.createNamedQuery("Cliente.findByNombre").getResultList();
```

### Gestión de transacciones

Una transacción es un conjunto de operaciones contra una base de datos que queremos sean ejecutadas o todas o ninguna. Por ejemplo, queremos borrar el cliente 1 y añadir el cliente 2, pero queremos que se hagan las dos operaciones o ninguna. Observa cómo indicamos al EntityManager donde empieza la transacción, donde intentar salvar todos los cambios, donde volver atrás si hubo un error y cómo termina:

```
// Cliente c1, c2;
try {
    EntityManager em = emf.createEntityManager();
    em.getTransaction().begin();
    em.remove(c1);
    em.persist(c2);
    em.getTransaction().commit();
} catch (Exception e) {
    em.getTransaction().rollback();
} finally {
    em.flush();
    em.close();
}
```

##Lenguajes propios de la herramienta ORM

Por lo general, las implementaciones ORM como Hibernate, por ejemplo, son muy pesadas (muchos *megas* de dependencias) como para insertarlas en una aplicación móvil. Esto hace que, por lo general, sólo se usen en el backend en vez del frontend.

A su vez, las tecnologías del backend cada vez están más enfocadas a sistemas que hablan con otros sistemas (desacoplando totalmente el backend del frontend) y tendiendo a soluciones de Web Services tipo REST o RESTful (efectivamente este último es como llamamos un Web Service que implementa una arquiectura REST sobre HTTP).

Si juntamos la necesidad de implementar nuestro backend como un sevicio RESTful y usando una herramienta ORM, encontramos una tecnología muy extendida que integra Hibernate, JPA y JDO (Java DAta Objects), sin necesidad de aprender Hibernate. Por supuesto, estamos hablando de Spring.

### Instalación de la herramienta ORM

Para proyectos maven, instalar las dependencias necesarias es increiblemente sencillo haciendo uso del **pom.xml**.

### Sesiones, autenticación y mantenimiento del objeto en el tiempo

Autenticación básica, digest, por token...

Oauth 2.0

### Lombok

### Problema de JSON anidado en Spring (relaciones)