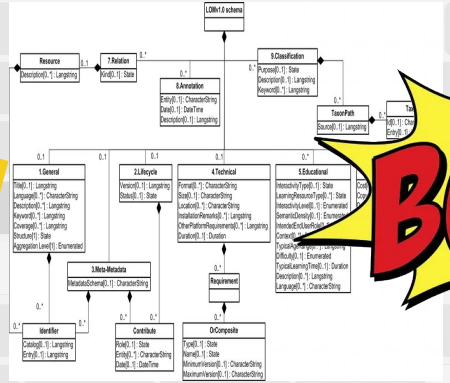

ORM

Object Relational Mapping

ORM

Hay una divergencia entre la representación de los datos que se utiliza en Java (POO) y la forma en la que se almacenan en las BD relacionales (Tablas).

Un ORM nos permite "conectar" esas dos representaciones de forma simple para el programador.

ORMLite

Proporciona funcionalidad de forma y ligera para prersistir objetos Java en BD SQL evitando la complejidad y sobrecarga de otros paquetes ORM más potentes.

NO OLVIDES AÑADIR LA DEPENDENCIA EN EL POM DE MAVEN (ORM LITE JDBC)

Bases de ORMLite

Preparación de las clases añadiendo anotaciones Java

Clases DAO incorporadas

Constructor de consultas flexible para construir consultas complicadas

Soporte para MySQL, Postgres, Ms SQL Server H2 ...

Maneja consultas precompiladas para consultas repetidas.

Soporte básico para transaciones

...

Uso de anotaciones para las clases a persistir

Hay que añadir en la parte superior de la clase la anotación (pueden ser optativos los paréntesis...)

```
@DatabaseTable(tableName = “nombre de la tabla”)
```

y en cada uno de los campos de la clase a persistir la anotación

```
@DatabaseField(id = true | canBeNull = false)
```

TODAS las clases a persistir deben contar con un constructor sin argumentos (no son necesarios los getters y setters

Primer ejemplo. Código de la clase a persistir

```java
@DatabaseTable
public class Alumno {

    2 usages
    @DatabaseField(id = true)
    private String nombre;
    2 usages
    @DatabaseField
    private String apellido;
    2 usages
    @DatabaseField
    private int edad;
    1 usage
    public Alumno(String nombre, String apellido, int edad){
        this.nombre = nombre;
        this.apellido = apellido;
        this.edad = edad;
    }
    public Alumno(){}
```

Primer ejemplo. Código de uso

```
    try (ConnectionSource conSrc = new JdbcConnectionSource( url: "jdbc:mysql://db-programacion-ej1.cbved7bhnvqz.us-e

        Dao<Alumno, String> alumnoDao = DaoManager.createDao(conSrc, Alumno.class);

        Alumno alumno1 = new Alumno( nombre: "Perico",  apellido: "de los Palotes",  edad: 64);

        alumnoDao.create(alumno1);

        Alumno alumno2 = alumnoDao.queryForId("John");

        System.out.println(alumno2);

    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Consultar la documentación

La documentación es extensa y "espartana"

https://ormlite.com/javadoc/ormlite-core/doc-files/ormlite.html

También podéis verla en un formato más "friendly" en pdf

https://ormlite.com/docs/ormlite.pdf

Y por supuesto, tenemos la API formato javadoc

https://ormlite.com/javadoc/ormlite-core/

Consideraciones sobre las anotaciones

Muchas de las anotaciones admiten el uso de parámetros entre paréntesis. El formato habitual es

@Anotación (clave 1 = valor1, clave2 = valor2 …)

Puede que necesitemos utilizar un argumento clave = valor, varios o ninguno. Si no utilizamos ninguno, se aplicarán valores por defecto. Es cuestión de mirar la documentación.

@DatabaseTable

Para indicar la tabla en la que persiste el objeto, justo antes de la declaración de la clase

Supongamos que configuramos la clase Alumno para persistir

@DatabaseTable -> la tabla asociada a la clase se llamará "igual" que la clase (alumno)

@DatabaseTable(tableName = "alumnos") ->

la tabla asociada será alumnos, independientemente del nombre de la clase

@DatabaseField

Con ella marcamos cada uno de los campos de la clase que queremos persistir. Justo delante de la declaración del campo. Hay muchas opciones disponibles. Supongamos que es numExpediente de alumno.

@DatabaseField -> El campo se asociará a la tabla como un campo con ese mismo nombre (alumno.numExpediente).

@DatabaseField(id = true) -> identifica este campo corresponde a una clave primaria. NECESARIO para update, delete... Solo puede haber uno.

@DatabaseField(canBeNull = false) -> indica que el campo correspondiente no puede ser nulo (por defecto true)

Configurando un DAO

ORMLite utiliza el patrón DAO para aislar la capa de acceso a datos de la lógica de negocio. Cada DAO proporciona operaciones CRUD.

El objeto Dao utiliza genéricos en la forma Dao<T, ID>, -> T es la clase donde operará el código e ID es la clase de la columna ID asociada con esta clase (la clave primaria)

Ej, para crear un DAO que trabaje sobre alumnos se declarará (suponiendo que el id es el campo String numExpediente):

```
Dao<Alumno, String> daoAlumno;
```

Instanciando un Dao

Para crear un objeto de la interfaz Dao, utilizamos el método estático createDao de la clase DaoManager.

A este hay que pasarle como argumentos un objeto de tipo ConnectionSource que representa la conexión a la base de datos y un objeto Class, que representa la clase sobre la que trabajamos:

```
DaoManager.createDao(ConnectionSource connectionSource,
Class<T> clase)
```

Necesitaremos una instancia de ConnectionSource que obtendremos de JdbcConnectionSource, que implementa esa interfaz

```
ConnectionSource conSrc = new JdbcConnectionSource(dbURL,
user, passwd);
```

Ejemplo de creación del Dao

Para declarar e instanciar nuestro objeto Dao para persistir la clase Alumno con id numExpediente en la base de datos típica de AWS RDS

```
ConnectionSource conSrc = new
JdbcConnectionSource("jdbc:mysql://db-programacion-ej1.cbved7bhnvqz.
us-east-1.rds.amazonaws.com/alumnos","admin","piramide");

Dao<Alumno, String) daoAlumno =
DaoManager.createDao(conSrc,Alumno.class);
```

Cuando terminemos de utilizar la conexión deberemos cerrar el ConnectionSource invocando su método close();

```
conSrc.close();
```

Utilizando el objeto Dao para operar en la BD

Ahora utilizaremos ese objeto Dao para ejecutar operaciones en la base de datos. Algunas de las operaciones:

```
create(objeto)

queryForId(valor buscado del campo id)

update(objeto)

delete(objeto)

...
```

Creación de tablas desde ORMLite

Podemos automatizar la creación de las tablas necesarias para persistir los objetos que hemos definidos mediante la clase `TableUtils` y sus métodos estáticos, que incorpora ORMLite

Simplemente tendremos que ejecutar sobre el Dao correspondiente las operaciones:

Ej

```
TableUtils.createTable(alumnoDao);
```

También existe un método `createTableIfNot Exists`

Persistencia de campos de tipo no primitivo

Podemos indicar que un atributo de una clase se corresponde con un objeto indicándolo con la anotación

```
@DatabaseField(foreign=true)
```

Si no indicamos parámetros en la anotación, esta "conectará" con el campo de la tabla correspondiente a través de una clave foránea identificada como nombreDeLaClase_id

Por ejemplo, si la clase es Profesor y el id de esa clase (la clave primaria) es nrp, en la tabla que "enlaza" con la tabla que representa esa clase se utilizará el campo `profesor_id`

Consultas personalizadas

Los Dao incorporan métodos para realizar consultas como: `queryForId`, `queryForAll` o iterator para poder moverse por todos los datos de la tabla (y algunos más)

Pero también podemos crear nuestras propias consultas personalizadas a través de `queryBuilder`

La primera recomendación, como buena práctica es utilizar constantes Java para definir los nombres de las columnas de las tablas para que las consultas sean más claras. Por ejemplo

```
public static final String NOMBRE_COLUMNA_PASSWORD = “password”

@DatabaseField(columnName = NOMBRE_COLUMNA_PASSWORD)
```

queryBuilder

`QueryBuilder<T,ID>` -> *T es la clase sobre la que actúa e ID el tipo de identificador*

Ej

```
QueryBuilder<Usuario, String> queryBuilder = usuarioDao.queryBuilder();
```

A partir de aquí hay que ir añadiendo las cláusulas a la consulta

queryBuilder.where.eq(Usuario.NOMBRE_COLUMNA_PASSWORD, "1234");

https://ormlite.com/javadoc/ormlite-core/com/j256/ormlite/stmt/QueryBuilder.html

PreparedQuery

Una vez construida la consulta sobre el queryBuilder, hay que proceder a "generar" la consulta, lo que dará como resultado un objeto de tipo `PreparedQuery`

```
PreparedQuery<Clase> preparedQuery = queryBuilder().prepare();
```

Por último ejecutaremos la consulta y recogeremos los resultados en un objeto de tipo `List<Clase>`

```
List<Usuario> listaUsuarios = usuarioDao.query(preparedQuery);
```

Preparando consultas personalizadas

QueryBuilder se ha implementado para poder construir consultas "fácilmente"

Las consultas simples pueden construirse de forma literal

```
QueryBuilder<Account, String> queryBuilder =  accountDao.queryBuilder();
Where<Account, String> where = queryBuilder.where();
where.eq(Account.NAME_FIELD_NAME, "foo");
where.and();
where.eq(Account.PASSWORD_FIELD_NAME, "_secret");
PreparedQuery<Account> preparedQuery = queryBuilder.prepare();
```

Equivale a la consulta `SELECT * FROM account  WHERE (name = 'foo' AND password = '_secret')`