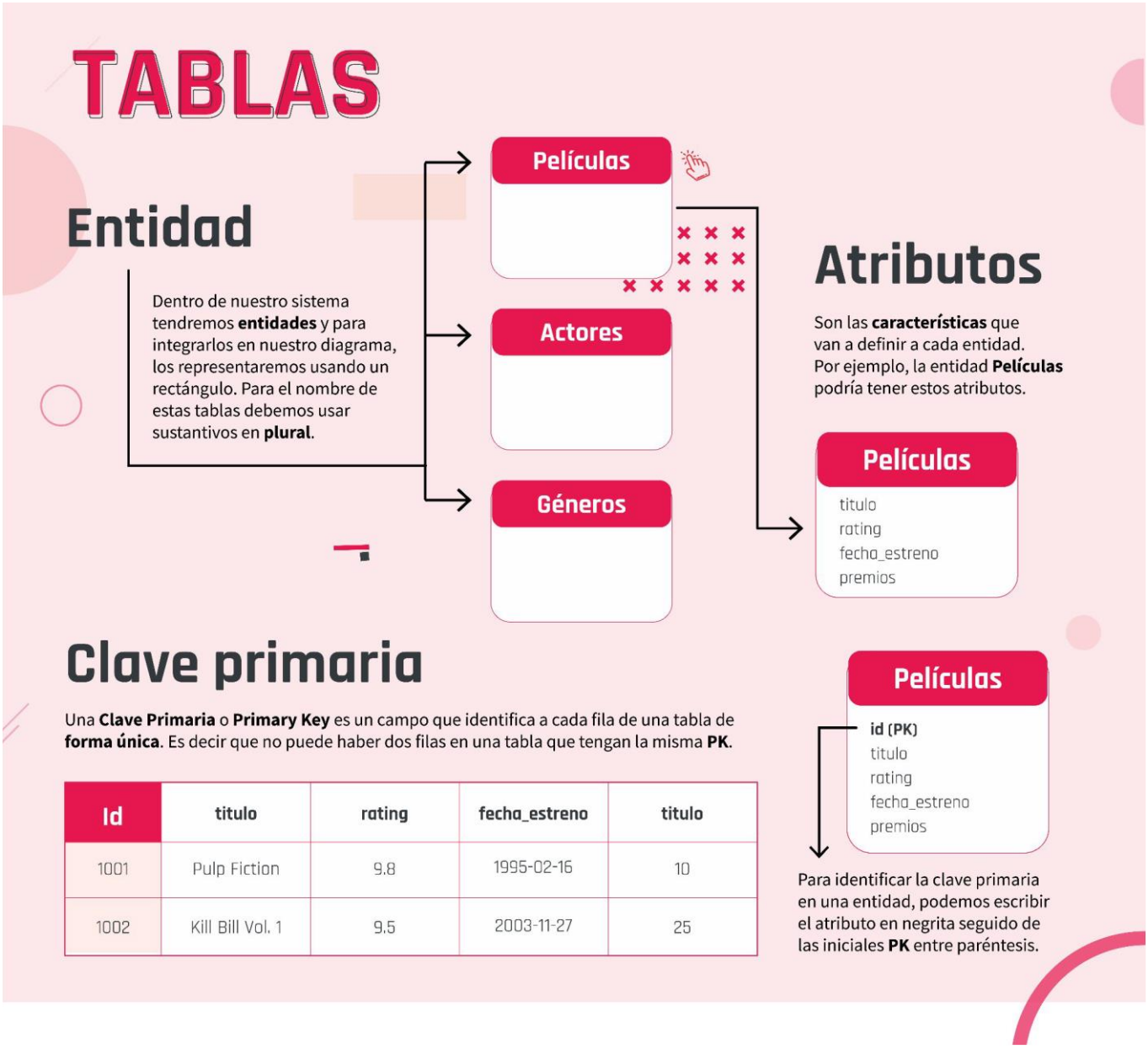
Concepto:

Base de datos: es un conjunto de datos pertenecientes a un mismo contexto, organizados para un propósito en específico. Nos permite almacenar, acceder, manipular, combinar y analizar datos.

Bases de datos relacional: es un tipo de base de datos que almacena datos en forma de tablas. Las tablas están formadas por filas y columnas. Las filas representan registros individuales de datos, y las columnas representan los atributos de esos datos.



Tablas

Una tabla de base de datos es la estructura principal y más general que existe para almacenar información; una base de datos contendrá una o más tablas.

La forma de identificar cada tabla será mediante un nombre único.

En concreto una tabla se divide en:

1. Las filas, que representan los registros de una base de datos
2. Cada fila tendrá su información dividida en columnas, con un nombre único dentro de una tabla.

Algunos de los tipos de datos que podemos almacenar en una tabla son:

Nombre del Tipo [Nombre SQL]	**Descripción**
Nota [LONGVARCHAR]	Campo para texto muy grande (2GB)
Texto (fijo) [CHAR]	Texto de tamaño fijo
Texto [VARCHAR]	Texto de tamaño variable
Texto [VARCHAR_IGNORECASE]	Texto de tamaño variable que no diferencia entre mayúsculas y minúsculas

Nombre del Tipo [Nombre SQL]	**Descripción**
TinyInteger [TINYINT]	Entero de 3 cifras. (Soporta valores entre -128 y 127)
Small Integer [SMALLINT]	Enteros de 5 cifras. (soporta valores entre -32768 y 32767)
Integer [INTEGER]	Entero de 10 cifras. (soporta valores entre -2147483648 y 21473647)
BigInt [BIGINT]	Entero de 19 cifras

Nombre del Tipo [Nombre SQL]	**Descripción**
Fecha [DATE]	Almacena valores del tipo día, mes y año
Hora [TIME]	Almacena valores del tipo hora, minuto y segundo
Fecha/Hora [TIMESTAMP]	Almacena valores del tipo día, mes, año, hora, minuto y segundo

Claves primarias y foraneas

Una clave primaria es una columna o un conjunto de columnas en una tabla cuyos valores identifican de forma exclusiva una fila de la tabla. Una base de datos relacional está diseñada para imponer la exclusividad de las claves primarias permitiendo que haya sólo una fila con un valor de clave primariaespecífico en una tabla.

Una clave foránea es una columna o un conjunto de columnas en una tabla cuyos valores corresponden a los valores de la clave primaria de otra tabla. Para poder añadir una fila con un valor de clave foránea específico, debe existir una fila en la tabla relacionada con el mismo valor de clave primaria.

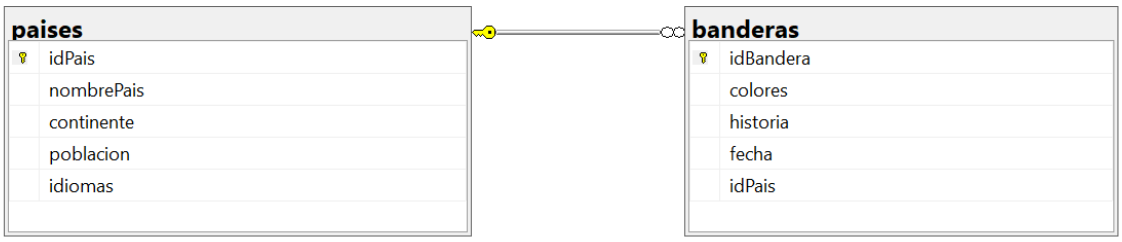
Relaciones

- **Relación uno a uno:** Relaciona un único registro de la tabla principal con un solo registro de la tabla relacionada. Este tipo de relación produce el mismo resultado que se si unieran los campos de ambas tablas en una sola tabla.

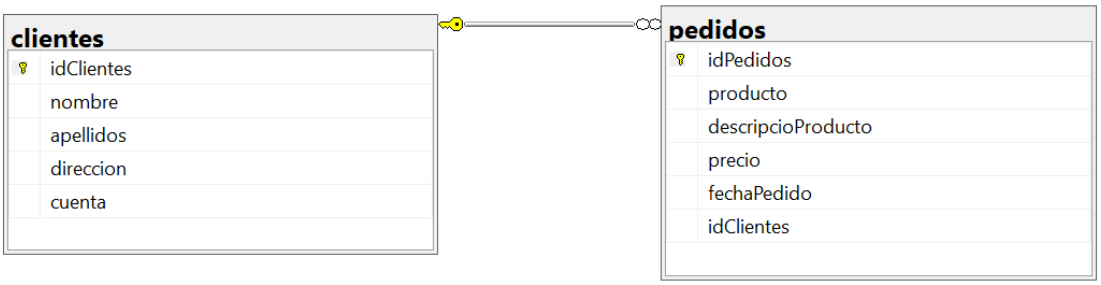
- **Relación uno a varios:** Es el tipo de relación mas frecuente. Un único registro de la tabla principal se puede relacionar con varios de la tabla relacionada.

- **Relación varios a varios:** un registro de la tabla principal se relaciona con varios de la tabla relacionada y, además, un registro de la tabla relacionada se relaciona con varios de la tabla principal. Este tipo de relaciones se puede transformar en dos relaciones de tipo uno a varios creando una tabla intermedia de unión.

Relación 1 a 1



Relación 1 a N



Relación N a N

Lenguaje SQL: es el lenguaje utilizado dentro de la base de datos para comunicar las peticiones y respuestas de datos.

El lenguaje de definición de datos (en inglés Data Definition Language, o **DDL**), es el que se encarga de la modificación de la estructura de los objetos de la base de datos. Incluye órdenes para modificar, borrar o definir las tablas en las que se almacenan los datos de la base de datos. Dentro de este grupo existen algunas operaciones muy conocidas como CREATE, ALTER, DROP.

Tambien podemos manipular o en su defecto insertar datos. Para esto, existen otros tipos de instrucciones/sentencias que podemos usar, las cuales forman parte de otro grupo llamado DML. Las sentencias **DML** permite a los usuarios introducir datos para posteriormente realizar tareas de consultas o modificación de los datos que contienen las Bases de Datos. Las sentencias DML son SELECT, INSERT, UPDATE, DELETE.

Creación de una base de datos

Se utiliza la sentencia **CREATE DATABASE**.

```
CREATE DATABASE db;
```

Para utilizarla en las siguientes sentencias, se utiliza:

```
USE db;
```

Creación de una tabla

Para crear una tabla se utiliza la sentencia **CREATE TABLE** de la siguiente manera:

```
CREATE TABLE tableName (
      id_table INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
      campo1 tipado,
      campo2 tipado,
);
```

Con la table se crean lo campos que contendrá. El primero será la clave primaria de la tabla, con los atributos de no vacío, auto incremental y entero ya definidas. Luego de agregan las demás columnas con el tipado aceptado para cada una.

Eliminar una tabla

Para eliminar una tabla especifica se utiliza la sentencia **DROP:**

```
DROP TABLE tableName;
```

Agragar cambios a tablas

Para realizar cambios en una tabla, debido a cambios en requerimientos o solo mejoraas que deseamos agregar, utilizamos la sentencia **ALTER TALBE**.

Para agregar campos (columnas) se utiliza:

```
ALTER TABLE tableName ADD campo tipadoDelCampo;
```

Insertar datos en tablas

Para insertar datos en los distintos campos de una tabla se utiliza la sentencia **INSERT**:

```
INSERT INTO table (campo1, campo2, campo3, …)
      VALUES (value1, value2, valu3, …),
            (value1', value2', value3', …);
```

Los valores que se insertan en una fila van dentro de paréntesis y en el orden que se especificó en la primera línea. Se pueden insertar varias filas en una misma sentencia, diferenciadas por estas entre paréntesis distintos.

Consultar datos de una tabla

La sintaxis utilizada para las consultas de datos es la siguiente:

```
SELECT columna, otra_columna, …
FROM tabla;
```

Devolverá las propiedades indicadas como columnas provenientes de la tabla especificada.

SELECT: palabra clave utilizada en declaraciones de consulta a datos de la base de datos.

FROM: indica la tabla que se desea consultar.

Con el carácter * se devolverá todas las columnas de datos de la tabla.

Filtrado de los resultados

```
SELECT column, another_column, …
FROM mytable
WHERE condition
    AND/OR another_condition
    AND/OR …;
```

WHERE: clausula aplicada en cada fila de datos para el chequeo de valores de columnas especificas para determinar cuales deben ser incluidas en el resultado o no.

AND / OR: utilizadas para agregar condiciones, de manera que se deban cumplir todas (AND) o al menos uno (OR).

Algunos operadores que se pueden utilizar para datos numéricos:

Operator	Condition	SQL Example
=, !=, < <=, >, >=	Standard numerical operators	col_name **!=** 4
BETWEEN ... AND ...	Number is within range of two values (inclusive)	col_name **BETWEEN** 1.5 **AND** 10.5
NOT BETWEEN ... AND ...	Number is not within range of two values (inclusive)	col_name **NOT BETWEEN** 1 **AND** 10
IN (...)	Number exists in a list	col_name **IN** (2, 4, 6)
NOT IN (...)	Number does not exist in a list	col_name **NOT IN** (1, 3, 5)

Algunos operadores comunes para datos de texto son:

Operator	Condition	Example
=	Case sensitive exact string comparison (**notice the single equals**)	col_name **=** "abc"
!= or <>	Case sensitive exact string inequality comparison	col_name **!=** "abcd"
LIKE	Case insensitive exact string comparison	col_name **LIKE** "ABC"
NOT LIKE	Case insensitive exact string inequality comparison	col_name **NOT LIKE** "ABCD"
%	Used anywhere in a string to match a sequence of zero or more characters (only with LIKE or NOT LIKE)	col_name **LIKE** "%AT%" (matches "AT", "ATTIC", "CAT" or even "BATS")
_	Used anywhere in a string to match a single character (only with LIKE or NOT LIKE)	col_name **LIKE** "AN_" (matches "AND", but not "AN")
IN (...)	String exists in a list	col_name **IN** ("A", "B", "C")
NOT IN (...)	String does not exist in a list	col_name **NOT IN** ("D", "E", "F")

Cuando tenemos datos repetidos en una tabla y queremos filtrarlos para que solo nos lo devuelvan una vez, se utiliza la siguiente sintaxis:

```
SELECT DISTINCT column, another_column, …
FROM mytable
WHERE condition(s);
```

DISTINCT: descarta filas que tienen el valor de alguna propiedad duplicado.

Para obtener resultados ordenados de forma ascendente o descendente, se utiliza la siguiente sintaxis:

```
SELECT column, another_column, …
FROM mytable
WHERE condition(s)
ORDER BY column ASC/DESC;
```

ORDER BY: especifica que columna debe ser ordenada de manera ascendente o descendente.

También se suelen utilizar sentencias donde se especifica el limite de resultados que se desean obtener y a partir de que posición de filas empezar a filtrar:

```
SELECT column, another_column, …
FROM mytable
WHERE condition(s)
ORDER BY column ASC/DESC
LIMIT num_limit OFFSET num_offset;
```

LIMIT: reduce el número de filas que devuelve la consulta.

OFFSET: especifica donde comienza el numero de filas a devolver.

Actualizar datos

Para modificar valores de uno o varios registros, se utiliza la sentencia **UPDATE:**

```
UPDATE table
SET column1 = value1, column2 = value2
WHERE id = num_id;
```

Eliminar datos

Para eliminar valores de una tabla se utiliza la sentencia **DELETE:**

```
DELETE FROM table_name WHERE condition;
```

Comunicación entre base de datos y la app

Se establece la comunicación con nuestra base de datos y la aplicación, utilizando Express y Sequelize.

Sequelize es un ORM basado en la promesa para Node.js y io.js. Es compatible con los dialectos PostgreSQL, MySQL, MariaDB, SQLite y MSSQL.

ORM es un modelo de programación que permite mapear las estructuras de una base de datos relacional sobre una estructura lógica de entidades con el objeto de simplificar y acelerar el desarrollo de nuestra aplicación.

Para la instalación, usamos el siguiente comando:

```
npm install –save sequelize mysql2
```

```
npm install --save-dev sequelize-cli
```

Una vez instalado, se debe ejecutar el comando init:

```
npm sequelize init
```

Al ejecutarlo dentro de la carpeta de nuestro proyecto se creará:

```
mi-proyecto/
├── config/
│   └── config.json
├── migrations/
├── models/
│   └── index.js
├── node_modules/
├── seeders/
├── app.js
├── package-lock.json
└── package.json
```

De estos archivos se destaca el archivo config donde se tendrá información para conectarnos a la base de datos. Sequelize nos permite diferenciar ambientes y en cada uno se debe especificar el usuario, password (del workbench), nombre de la base de datos, entre otros.

```
{
  "development": {
    "username": "root",
    "password": "123456",
    "database": "ejemplo-sequelize",
    "host": "127.0.0.1",
    "dialect": "mysql",
    "operatorsAliases": false
  },
  "test": {
    "username": "root",
    "password": "123456",
    "database": "ejemplo-sequelize",
    "host": "127.0.0.1",
    "dialect": "mysql",
    "operatorsAliases": false
  },
  "production": {
    "username": "root",
    "password": "123456",
    "database": "ejemplo-sequelize",
    "host": "127.0.0.1",
    "dialect": "mysql",
    "operatorsAliases": false
  }
}
```

- Creación de modelos

Estos modelos representaran a la tabla que esta en nuestra base de datos. El comando para crear un modelo será:

`sequelize model:create --name` *`nameModel`* `–attributes` *`field1:typeField1,field2:typeField2`*

Le indicamos a Sequelize que cree nuestro modelo *nameModel* con los campos que se especifican y su tipado (*field1:typeField1,field2:typeField2)*. Al ejecutar esta instrucción se creará una carpeta de nombre models dentro de nuestra aplicación y dentro encontraremos un archivo llamado *nameModel.js*

```
'use strict';
const {
  Model
} = require('sequelize');
module.exports = (sequelize, DataTypes) => {
  class nameModel extends Model {
    /**
     * Helper method for defining associations.
     * This method is not a part of Sequelize lifecycle.
     * The `models/index` file will call this method automatically.
     */
    static associate(models) {
      // define association here
    }
```

```
  }
  NameModel.init({
    id_model: {
      type: DataTypes.INTEGER,
      primaryKey: true,
      autoIncrement: true,
      allowNull: false
    },
    field1: DataTypes.STRING,
    field2: DataTypes.STRING
  }, {
    sequelize,
    modelName: 'NameModel',
  });
  return nameModel;
};
```

Dentro del método init() tendremos un objeto con cada campo de la tabla y sus características. El primer campo siempre será el id del modelo, con la configuración detallada en el código.

En caso de que uno de los campos sea una clave foránea, se agrega el siguiente objeto como valor del campo references dentro de una de las columnas de tipo id foráneo:

```
id_category:{
      type: DataTypes.INTEGER,
      references: {
        model: "category",
        key: "id_category"
      }
    },
```

En model se especifica la tabla a la que hace referencia y key será el nombre el campo de clave primaria de dicha tabla. Y se modifica al comienzo del archivo la siguiente sección associate:

```
static associate(models) {
      NameModel.belongsTo(models.Category, {foreignKey: "id_category"})

    }
```

Además de crear un archivo en models, se creará un archivo en migrations, el cual contendrá en su nombre la fecha y hora de creación seguida por el nombre del modelo. Podrá llamarse *20240219224953-create-nameModel.js*

```
'use strict';
/** @type {import('sequelize-cli').Migration} */
module.exports = {
  async up(queryInterface, Sequelize) {
    await queryInterface.createTable('NameModels', {
      id_model:{
        type: Sequelize.INTEGER,
        allowNull: false,
        primaryKey: true,
        autoIncrement: true
      },
      field1: DataTypes.STRING,
      field2: DataTypes.STRING,
      createdAt: {
        allowNull: false,
        type: Sequelize.DATE
      },
      updatedAt: {
        allowNull: false,
        type: Sequelize.DATE
      }
    });
  },
  async down(queryInterface, Sequelize) {
    await queryInterface.dropTable('NameModels');
  }
};
```

Dentro de createTable tendremos los mismos campos especificados en el archivo *nameModel.js*. Se agrega el createdAt y el updatedAt automáticamente que representan las fechas de creación y actualización respectivamente.

Hasta este paso, no hemos insertado nada en la base de datos. Solo hemos creado el modelo requerido y el archivo de migraciones para nuestro modelo. Ahora vamos a crear esta tabla en la base de datos ejecutando el siguiente comando:

```
npx sequelize-cli db:migrate
```

Para revertir una migración, se utiliza el comando:

```
npx sequelize-cli db:migrate:undo
```

Luego de hacer esto, podremos ver en el Worckbench la tabla con sus campos creados.

- Creación de Seeds:

Para agregar algunos datos por default podemos crear sedes (semillas). Lo realizaremos utilizando el comando:

```
npx sequelize-cli seed:generate –name demo-nameModel
```

Este commando creara un archive en la carpeta seeders con el formato *XXXXXXXXXXXXXX-demo-nameModel.js*

Luego debemos editar el archivo para insertar el demo:

```
module.exports = {
  up: (queryInterface, Sequelize) => {
    return queryInterface.bulkInsert('nameModels', [{
      firstName: 'John',
      lastName: 'Doe',
      email: 'example@example.com',
      createdAt: new Date(),
      updatedAt: new Date()
    }]);
  },
  down: (queryInterface, Sequelize) => {
    return queryInterface.bulkDelete('nameModels', null, {});
  }
};
```

Podremos cargar estas seed con el comando:

```
npx sequelize-cli db:seed:all
```

Podemos revertir una seed con el comando:

```
npx sequelize-cli db:seed:undo
```

- Adaptacion del controller:

Primero debe modificarse la importación del modelo dentro de controller, de manera que quede similar a:

```
const User = require("../models").User;
```

Ademas, las funciones que deban utilizar la base de datos, deberán ser funciones asíncronas y utilizar funciones especificas para acceder a los datos cargados en el modelo:

```
const getAllUsers = async (req, res) => {
  const users = await User.findAll();
  res.json(users);
};

async function getUserById(req, res) {
  const { id } = req.params;
  const user = await User.findByPk(id);
  if (!user) {
    return res.status(404).json({ message: "User not found" });
  }
  res.json(user);
}

const createUser = async (req, res) => {
  const { name, email, password } = req.body;
  const user = await User.create({ name, email, password });
  res.json(user);
};
```

Analizando cada uno:

- **create()** es una abreviatura para crear una instancia no guardada como Model.build() y guardar la instancia con instance.save().

- **findAll()** nos trae todos los registros de esta tabla y para acceder a ellos simplemente se ejecuta el método toJSON() que nos da un objeto con la información. Podemos hacer uso del objeto de configuración where que nos permite hacer una mejor selección de nuestros registros en nuestro método findAll o find.

- **findByPK()** se recibe el id que queremos buscar y como segundo parámetro un objeto donde especificamos diferentes opciones para mejorar nuestra consulta.

- **findOne()** obtiene la primera entrada que encuentra.

- **findOrCreate()** creara una entrada en la tabla a menos que no pueda encontrar una que cumpla con las opciones de la consulta. En ambos casos retornara una instancia y un booleano indicando si esa instancia fue creada o ya existía.

  La configuración where es utilizada para encontrar la entrada y la opción default es utilizada para definir que debe ser creado en caso de no encontrar la consulta. Si default no contiene valores en alguna columna, Sequelize tomara los valores dados en where.

```
const [user, created] = await User.findOrCreate({
  where: { username: 'sdepold' },
  defaults: {
    job: 'Technical Lead JavaScript'
  }
});
```

- **update()** actualizan una entrada. El primer parámetro especifica que entrada modificar y el segundo parámetro contiene el valor que debe introducirse.

```
await User.update({ lastName: "Doe" }, {
  where: {
    lastName: null
  }
});
```

- **delete()** elimina una entrada especificada en el parámetro:

```
await User.destroy({
  where: {
    firstName: "Jane"
  }
});
```