Ejercicio Paso a Paso: CRUD de usuarios con Sqlite3 y ORM Sequelize.

Parte 1

Paso 1: Preparación del proyecto

En este paso inicializamos el proyecto e instalamos la librería `sqlite3` que usaremos para conectarnos a la base de datos.
Con el siguiente código, vamos a crear el directorio de trabajo, incializar el proyecto para node y agregar las liberiras que vamos a necesitar (sqlite3 y sequelize).

```
mkdir crud_usuarios
cd crud_usuarios
npm init -y
npm i sqlite3 sequelize
```

> Observaciones: **npm init -y** es un comando que se utiliza para iniciar un nuevo proyecto Node.js con un archivo package.json preconfigurado con valores predeterminados. La opción -y (abreviatura de "yes") indica que se deben aceptar automáticamente todas las opciones predeterminadas de configuración sin hacer preguntas adicionales. Esto significa que el comando creará un archivo package.json con valores predeterminados, como el nombre del proyecto, la versión, la descripción, el autor, la licencia, etc.
> El comando **npm i sqlite3 sequelize** se utiliza para instalar dos paquetes de Node.js que se utilizan comúnmente en aplicaciones web que utilizan una base de datos SQLite: sqlite3 y sequelize. Sqlite3 es un controlador para SQLite, que es un sistema de gestión de bases de datos relacionales ligero y autónomo que se ejecuta en la mayoría de los sistemas operativos. Sequelize es una biblioteca ORM (Object-Relational Mapping) para Node.js. Al ejecutar el comando *npm i sqlite3 sequelize*, se descargarán e instalarán los paquetes sqlite3 y sequelize y sus dependencias en el directorio actual de su proyecto de Node.js (carpeta *node_modules*). Una vez instalados, puede utilizar estos paquetes en su aplicación para conectarse y manipular una base de datos SQLite utilizando JavaScript.

Si estamos en linux podemos crear el archivo en blanco para posteriormente agregar el codigo en javascript dentro del mismo.

```
touch app.js db.js
```

En un equipo con windows, crear los archivos app.js y db.js con el explorador de windows o desde vscode desde el menú *archivo*, seleccionando *nuevo archivo*

Paso 2: Importar el paquete `sequelize` y conectarnos a la base de datos

En esta ejercitación paso a paso, vamos a trabajar con la importación de módulos de javascript estándar de node.js, es decir con la librería common.js

> Node.js trabaja de manera predeterminada con el sistema de módulos CommonJS para importar y exportar módulos. CommonJS es un estándar para trabajar con módulos en JavaScript, que proporciona una forma sencilla y coherente de organizar y reutilizar código en una aplicación. En Node.js, cada archivo JavaScript se considera un módulo y los módulos se pueden importar y exportar utilizando las palabras clave **require** y **module.exports**, respectivamente.

Verificar que el archivo package.json, no tenga la clave `"type"` o que esté establecida en `"type": "commonjs"` para hacer la guía paso a paso.

```
{
  "name": "crud_usuarios",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "sequelize": "^6.30.0",
    "sqlite3": "^5.1.6"
  }
}
```

Vamos a realizar la conexión con la base de datos en un módulo aparte: `db.js`

> Observar que si utilizamos `"sqlite::memory:"` usamos una base de datos `en memoria` , es decir, no persistirá entre una ejecución del programa y otra.

En nuestro ejemplo vamos hacer persistir la base de datos en un archivo físico en el disco llamado `db.sqlite` que va estar en la misma ruta donde ejecutamos nuestro proyecto.

En el archivo llamado `db.js` agregamos el código para conectarnos a la base de datos.

> Nota: sino está creada la base de datos **db.sqlite**, al enlazar el modelo, en una etapa posterior de esta guía, la misma se va a crear, y si ya existe, vamos a utilizar los datos que están almacenados en la misma.

En el archivo **db.js** agregar el siguiente código

```
const { Sequelize } = require("sequelize");

// const sequelize = new Sequelize("sqlite::memory:");
const sequelize = new Sequelize({
    dialect: "sqlite",
    storage: "./db.sqlite",
});

module.exports = sequelize;
```

En el archivo **app.js** Vamos a importar el módulo `db` .

```
const sequelize = require("./db");

async function main(){
  try{

    // Sincronizar la base de datos con el modelo
    await sequelize.sync();

    // Probar la conexión
    // await sequelize.authenticate();

    console.log("Base de Datos: lista");
  }
  catch (error){
    console.error("Ha ocurrido un error: ", error);
  }
};

main();
```

Si lo ejecutamos por primera vez desde la ventana de terminal de visual studio:

```
node app.js
```

vamos a observar que el archivo `db.sqlite` ha sido creado en el directorio de trabajo y se visualiza el resultado

```
Executing (default): SELECT 1+1 AS result
Base de Datos: lista
```

> Observación: Cuando se utiliza Sequelize para conectarse a una base de datos, es común realizar una "prueba de conexión" para asegurarse de que la conexión se ha establecido correctamente. Para realizar esta prueba, Sequelize ejecuta una consulta de base de datos simple que devuelve un resultado conocido. Sequelize utiliza **SELECT 1+1 AS result** para realizar una prueba de conexión simple que devuelve un valor constante y asegura que la conexión a la base de datos se ha establecido correctamente. Esta expresión es sintácticamente válida en la mayoría de los sistemas de base de datos y proporciona un valor simple y fácil de verificar como resultado de la consulta.

Paso 3: Crear el modelo

Ahora vamos a crear el modelo y para eso agregamos otro archivo: `usuario.js`

Si leemos la documentación (Model basics) veremos que tenemos dos alternativas sintácticas para crear nuestro modelo:

Usando `sequelize.define()`

En el achivo `usuario.js` agregar el siguiente código:

```
const { Model, DataTypes } = require("sequelize");
const sequelize = require("./db");

const Usuario = sequelize.define(
    "Usuario",
    {
        id: {
            type: DataTypes.INTEGER,
            autoIncrement: true,
            primaryKey: true
        },
        nombre: { type: DataTypes.STRING },
        apellido: { type: DataTypes.STRING },
        usuario: {
            type: DataTypes.STRING,
            unique: true
        },
        password: { type: DataTypes.STRING },
        email: { type: DataTypes.STRING },
        fecha_alta: { type: DataTypes.DATE, defaultValue: DataTypes.NOW }
    },
    {
        timestamps: false,
    }
);

module.exports = Usuario;
```

> Nota: *sequelize.define("Usuario", ...)* crea una definición de modelo llamada Usuario en la instancia de base de datos que está almacenada en la constante sequelize. La función *define* acepta dos argumentos: el nombre del modelo y un objeto que define los atributos de la tabla.
> Si no se define una clave primaria (PK) en un modelo de Sequelize, se asumirá que la tabla de la base de datos tiene una columna de clave primaria llamada **id** que es un número entero autoincrementable.
> Si la tabla de la base de datos tiene una clave primaria se utilizará esa en la definición del modelo.

La otra opción que nos provee el ORM Sequelize para definir el modelo, es extendiendo la clase `model` , que es una alternativa al anterior. Nosotros en la guía paso a paso usamos la primera opción pero si se desea se puede utilizar esta otra forma.

```
const { Model, DataTypes } = require("sequelize");
const sequelize = require("./db"); // instancia de Sequelize
class Usuario extends Model {}

Usuario.init({
        id: {
          type: DataTypes.INTEGER,
          autoIncrement: true,
          primaryKey: true
        },
        nombre: { type: DataTypes.STRING },
        apellido: { type: DataTypes.STRING },
        usuario: {
          type: DataTypes.STRING,
          unique: true
        },
        password: { type: DataTypes.STRING },
        email: { type: DataTypes.STRING },
        fecha_alta: { type: DataTypes.DATE, defaultValue: DataTypes.NOW  }
    },
    {
        sequelize,
        modelName: "usuario",
        timestamps: false,
    }
);

module.exports = Usuario;
```

> Observaciones: **{ sequelize, modelName: "usuario", timestamps: false }** define las opciones del modelo. sequelize especifica la instancia de Sequelize (la que definimos en el módulo db.js) a la que se asocia el modelo, modelName especifica el nombre del modelo (en la base de datos se creara en plural la tabla) y timestamps: false indica que no se deben agregar las marcas de tiempo createdAt y updatedAt a los registros de la tabla.

Para que nuestra aplicación conozca el modelo que vamos a utilizar, necesitamos indicar el módulo principal (app.js) que vamos hacer uso del modelo que definimos en usuario.js `const Usuario = require("./usuario");`

En `app.js` vamos a importar `usuario` . Tener en cuenta que lo importamos, y no hace falta establecer la extensión en .js ya que de manera predeterminada busca el archivo con dicha extensión, y lo asignamos a un variable Usuario (primera letra en mayúscula).

```
const sequelize = require("./db");
const Usuario = require("./usuario");

async function main(){
  try{

    // Sincronizar la base de datos con el modelo
    await sequelize.sync();

    // Probar la conexión
    // await sequelize.authenticate();

    console.log("Base de Datos: lista");
  }
  catch (error){
    console.error("Ha ocurrido un error: ", error);
  }

};

main();
```

Al ejecutar el programa deberías ver algo así como respuesta en la consola:

```
Executing (default): SELECT name FROM sqlite_master WHERE type='table' AND name='Usuarios';
Executing (default): CREATE TABLE IF NOT EXISTS `Usuarios` (`id` INTEGER PRIMARY KEY AUTOINCREMENT, `nombre` VA
Executing (default): PRAGMA INDEX_LIST(`Usuarios`)
Base de Datos: Lista
```

Y si examinás la base de datos con alguna herramienta apropiada, verás la tabla `Usuarios` creada.

> Cuando se utiliza Sequelize para definir un modelo de base de datos, por defecto se crea una tabla con un nombre en plural basado en el nombre del modelo en singular. Esto significa que, por ejemplo, un modelo llamado Usuario creará una tabla llamada Usuarios en la base de datos.
> Este comportamiento predeterminado es una convención en Sequelize y se basa en la suposición de que cada modelo representa una colección de registros en la base de datos. Al utilizar un nombre en plural para la tabla, se indica que la tabla contiene varios registros de ese modelo.
> Sin embargo, es posible cambiar el nombre de la tabla en la base de datos utilizando la opción tableName en la definición del modelo de Sequelize. `{ tableName: 'Usuario' }`

Paso 4: Crear un usuario

Teniendo la tabla vacía, lo primero que haremos será crear uno o dos instancias de nuestro modelo, esto es: usuarios.

En `app.js` agregamos la función agregarUsuarios() y la invocamos.

```
async function crearUsuarios(){

  //Crear usuario Juan Perez
  const datosJuanPerez = {
      nombre: "Juan",
      apellido: "Perez",
      usuario: "jperez",
      password: "MiPwd%%$$W",
      email: "jperez@gmail.com",
  };
  const juanPerez = await Usuario.create(datosJuanPerez);
  console.log("Usuario creado ID:", juanPerez.id);

  //Crear usuario María García
  const datosMariaGarcia = {
      nombre: "Maria",
      apellido: "García",
      usuario: "mgarcia",
      password: "Pwd%%$$Wss",
      email: "mgarcia@gmail.com",
  };
  const mariaGarcia = await Usuario.create(datosMariaGarcia);
  console.log("Usuario creado ID:", mariaGarcia.id);
}
```

De forma tal que el archivo `app.js` contiene el siguiente código:

```
const sequelize = require("./db");
const Usuario = require("./usuario");

async function crearUsuarios() {

    //Crear usuario Juan Perez
    const datosJuanPerez = {
        nombre: "Juan",
        apellido: "Perez",
        usuario: "jperez",
        password: "MiPwd%%$$W",
        email: "jperez@gmail.com",
    };
    const juanPerez = await Usuario.create(datosJuanPerez);
    console.log("Usuario creado ID:", juanPerez.id);

    //Crear usuario María García
    const datosMariaGarcia = {
        nombre: "Maria",
        apellido: "García",
        usuario: "mgarcia",
        password: "Pwd%%$$Wss",
        email: "mgarcia@gmail.com",
    };
    const mariaGarcia = await Usuario.create(datosMariaGarcia);
    console.log("Usuario creado ID:", mariaGarcia.id);
}

async function main() {
    try {

        // Sincronizar la base de datos con el modelo
        await sequelize.sync();
        await crearUsuarios(); //Creamos dos registros en la tabla usuarios

        // Probar la conexión
        //await sequelize.authenticate();

        console.log("Base de Datos: lista");
    }
    catch (error) {
        console.error("Ha ocurrido un error: ", error);
    }

};

main();
```

Si ejecutamos el proyecto con `node app.js` obtenemos el siguiente resultado, y observamos que se han creado dos usuarios con id 1 y 2

```
Executing (default): SELECT name FROM sqlite_master WHERE type='table' AND name='Usuarios';
Executing (default): PRAGMA INDEX_LIST(`Usuarios`)
Executing (default): PRAGMA INDEX_INFO(`sqlite_autoindex_Usuarios_1`)
Executing (default): INSERT INTO `Usuarios` (`id`,`nombre`,`apellido`,`usuario`,`password`,`email`,`fecha_alta`
Usuario creado ID: 1
Executing (default): INSERT INTO `Usuarios` (`id`,`nombre`,`apellido`,`usuario`,`password`,`email`,`fecha_alta`
Usuario creado ID: 2
Base de Datos: lista
```

Si volvemos a ejecutar el código con `node app.js` vemos que nos va a dar error:

```
Executing (default): SELECT name FROM sqlite_master WHERE type='table' AND name='Usuarios';
Executing (default): PRAGMA INDEX_LIST(`Usuarios`)
Executing (default): PRAGMA INDEX_INFO(`sqlite_autoindex_Usuarios_1`)
Executing (default): INSERT INTO `Usuarios` (`id`,`nombre`,`apellido`,`usuario`,`password`,`email`,`fecha_alta`
Ha ocurrido un error:  Error
    at Database.<anonymous> (C:\DLS\crud_usuarios\node_modules\sequelize\lib\dialects\sqlite\query.js:185:27)
    at C:\DLS\crud_usuarios\node_modules\sequelize\lib\dialects\sqlite\query.js:183:50
    at new Promise (<anonymous>)
    at Query.run (C:\DLS\crud_usuarios\node_modules\sequelize\lib\dialects\sqlite\query.js:183:12)
    at C:\DLS\crud_usuarios\node_modules\sequelize\lib\sequelize.js:315:28
    at async SQLiteQueryInterface.insert (C:\DLS\crud_usuarios\node_modules\sequelize\lib\dialects\abstract\que
    at async model.save (C:\DLS\crud_usuarios\node_modules\sequelize\lib\model.js:2490:35)
    at async Usuario.create (C:\DLS\crud_usuarios\node_modules\sequelize\lib\model.js:1362:12)
    at async crearUsuarios (C:\DLS\crud_usuarios\app.js:14:23)
    at async main (C:\DLS\crud_usuarios\app.js:34:9) {
  name: 'SequelizeUniqueConstraintError',
  errors: [
    ValidationErrorItem {
      message: 'usuario must be unique',
      type: 'unique violation',
      path: 'usuario',
      value: 'jperez',
      origin: 'DB',
      instance: [Usuario],
      validatorKey: 'not_unique',
      validatorName: null,
      validatorArgs: []
    }
  ],
  parent: [Error: SQLITE_CONSTRAINT: UNIQUE constraint failed: Usuarios.usuario] {
    errno: 19,
    code: 'SQLITE_CONSTRAINT',
    sql: 'INSERT INTO `Usuarios` (`id`,`nombre`,`apellido`,`usuario`,`password`,`email`,`fecha_alta`) VALUES (N
  },
  original: [Error: SQLITE_CONSTRAINT: UNIQUE constraint failed: Usuarios.usuario] {
    errno: 19,
    code: 'SQLITE_CONSTRAINT',
    sql: 'INSERT INTO `Usuarios` (`id`,`nombre`,`apellido`,`usuario`,`password`,`email`,`fecha_alta`) VALUES (N
  },
  fields: [ 'usuario' ],
  sql: 'INSERT INTO `Usuarios` (`id`,`nombre`,`apellido`,`usuario`,`password`,`email`,`fecha_alta`) VALUES (NUL
}
```

Esto se debe a que hay definido en el modelo (archivo usuario.js) un constraint en el que se indica que el nombre debe ser único, como se detalla a continuacion

```
.....
usuario: {
  type: DataTypes.STRING,
  unique: true
},
....
```

para continuar con el paso 5, una vez que se hayan creado los registros de los usuarios en la tabla, comentar las línea de código de crearUsuarios() de la siguiente manera:

`// await crearUsuarios();` ;

Paso 5: Recuperar todos los usuarios

Ahora que hemos creado un par de usuarios, vamos a recuperar todos los usuarios que hemos almacenado en la función `recuperarUsuarios()` :

Agregar en el archivo `app.js` la función asíncrona para recuperarUsuarios:

```
async function recuperarUsuarios(){
  // Recuperar todos los usuarios
  const usuarios = await Usuario.findAll();
  usuarios.forEach((u) => {
      console.log(`|${u.nombre}|${u.apellido}|${u.usuario}|${u.email}|`);
  });
}
```

y modificar la función Main() del mismo archivo e incluir el código para recuperar usuarios:

```
async function main() {
    try {

        // Sincronizar la base de datos con el modelo
        await sequelize.sync();
        //await crearUsuarios();
        await recuperarUsuarios();

        // Probar la conexión
        //await sequelize.authenticate();

        console.log("Base de Datos: lista");
    }
    catch (error) {
        console.error("Ha ocurrido un error: ", error);
    }

};
```

Ejecutar con ´node app.js´ y observar los resultados obtenidos:

```
Executing (default): SELECT name FROM sqlite_master WHERE type='table' AND name='Usuarios';
Executing (default): PRAGMA INDEX_LIST(`Usuarios`)
Executing (default): PRAGMA INDEX_INFO(`sqlite_autoindex_Usuarios_1`)
Executing (default): SELECT `id`, `nombre`, `apellido`, `usuario`, `password`, `email`, `fecha_alta` FROM `Usua
|Juan|Perez|jperez|jperez@gmail.com|
|Maria|García|mgarcia|mgarcia@gmail.com|
Base de Datos: lista
```

Paso 6: Recuperar un usuario específico.

También podemos recuperar un usuario específico, por ejemplo: el que tiene id = 3.

Para ello, modificar el archivo `app.js` y agregar el código:

```
async function recuperarUsuario(id){
    const u = await Usuario.findOne({ where: { id: id } });
    console.log(`|${u.nombre}|${u.apellido}|${u.usuario}|${u.email}|`);
}
```

y modificar la función main de app.js para obtener los datos del usuario con id = 1 y con id = 2:

```
async function main() {
    try {

        // Sincronizar la base de datos con el modelo
        await sequelize.sync();
        //await crearUsuarios();
        //await recuperarUsuarios();
        await recuperarUsuario(1);
        await recuperarUsuario(2);

        // Probar la conexión
        //await sequelize.authenticate();

        console.log("Base de Datos: lista");
    }
    catch (error) {
        console.error("Ha ocurrido un error: ", error);
    }
};
```

Paso 7: Modificar un usuario

Para modificar una instancia debemos primero recuperarla (por ejemplo por id), luego modificar las propiedades que deseamos actualizar y finalmente salvarla usando el método asíncrono: `save()`

`app.js`

```
async function actualizarUsuario(id, nombre){
  const u = await Usuario.findOne({ where: { id: id } });
  if (u){
    u.nombre = nombre;
    await u.save();
  }
}
```

Paso 8: Eliminar un usuario

Por último implementaremos la operación de eliminar un usuario invocando al método `destroy()`

app.js

```
async function eliminarUsuario(id){
  const u = await Usuario.findOne({ where: { id: id } });
  if (u){
    await u.destroy();
  }
}
```

Paso 9: Asociaciones. Definir una relación 1:N entre Usuarios y Tareas

Ahora vamos a crear una modelo `Tarea` que tendrá una relación con `Usuario` . Esta relación será de uno a muchos: un usuario puede tener muchas tareas asociadas.

Este es el tipo de relación más frecuente y la que vamos a utilizar en nuestro ejercicio. También se pueden definir relaciones *uno a uno (1:1)* y *muchos a muchos (N:M)*. Podés ver más detalles sobre cómo definir estas relaciones en la documentación: Associations

La relación se define a partir de los métodos `.hasMany()` y `.belongsTo()` . En nuestro caso será:

```
Usuario.hasMany(Tarea);
Tarea.belongsTo(Usuario);
```

Y si partimos de una base de datos existente. Especificamos la *clave foranea* de la relación.

```
Usuario.hasMany(Tarea, { foreignKey: "TareaId" });
Tarea.belongsTo(Usuario, { foreignKey: "TareaId" });
```

Entonces nuestro nuevo modelo `Tarea` queda definido así

Creamos el archivo `tarea.js` y agregamos el siguiente código de definición del modelo.

```
const { Model, DataTypes } = require("sequelize");
const sequelize = require("./db");

const Tarea = sequelize.define(
    "Tarea",
    {
        descripcion: { type: DataTypes.STRING },
    },
    {
        timestamps: false,
    }
);

module.exports = Tarea;
```

Para definir las relaciones entre el modelo de usuario y tareas agregamos el siguiento código al archivo app.js

app.js

```
const sequelize = require("./db");
const Usuario = require("./usuario");
const Tarea = require("./tarea");

Usuario.hasMany(Tarea);
Tarea.belongsTo(Usuario);
```

De esta forma, cuando invocamos en la función main `await sequelize.sync();` se va establecer la relación

Paso 10: Agregar tareas a un usuario

Al crear las asociaciones disponemos de un método que se crea de forma automática `createTarea()` que nos permite agregar una tarea a un `usuario` . En este caso vamos a agregar dos tareas al usuario cuyo `id` es `2`

app.js

```
async function agregarTareaAUsuario(usuarioId, descripcionTarea){
  const u = await Usuario.findOne({ where: { id: usuarioId } });
  if (u){
    await u.createTarea({
      descripcion: descripcionTarea
    });
  }
}
```

Paso 11: Listar las tareas de un usuario

El método `getTareas()` nos devuelve todas las tareas de un usuario. Aquí usamos el método `forEach()` para mostrarlas por consola.

app.js

```
async function listarTareasUsuario(usuarioId){
  const u = await Usuario.findOne({ where: { id: usuarioId } });
  if (u){
    const tareas = await u.getTareas();
    tareas.forEach((t) => {
        console.log(`|${t.descripcion}|`);
    });
  }
}
```

También podemos hacer algo similar usando **precarga** (*eager loading*) directamente cuando recuperamos el usuario podemos traer sus tareas si lo indicamos explicitamente con `include: Tarea`

app.js

```
async function listarTareasUsuarioPrecarga(usuarioId){
  const u = await Usuario.findOne({
    where: { id: usuarioId },
    include: Tarea,
  });
  if (u){
    u.Tareas.forEach((t) => {
        console.log(`|${t.descripcion}|`);
    });
  }
}
```

Paso 12 Remover las tareas de un usuario

Usando los métodos `removeTarea()` y `removeTareas()` podemos remover una o todas las tareas de un usuario específico. Vamos a ejemplificar cómo remover la tarea **Ejecutar** en este caso.
También demostramos el uso del método `countTareas()` para contar las tareas de un usuario espeífico.

app.js

```
async function eliminarTareaUsuario(usuarioId, descripcionTarea){
  const u = await Usuario.findOne({ where: { id: usuarioId } });
  if (u){
    // Recuperamos la tarea específica
    const t = await u.getTareas({
        where: { descripcion: descripcionTarea },
    });

    // Si la encontramos, la removemos
    if (t) {
        await u.removeTarea(t);
    }
  }
}
```