GOBIERNO DE ESPAÑA
MINISTERIO DE EDUCACIÓN Y FORMACIÓN PROFESIONAL
GENERALITAT VALENCIANA
Conselleria d'Educació, Investigació, Cultura i Esport
cefire
Fons Social Europeu
L'FSE inverteix en el teu futur
UNIÓ EUROPEA

3. Acceso a bases de datos relacionales: MariaDB/MySQL

En esta sesión trataremos cómo conectar con un sistema SQL tradicional, como MariaDB/MySQL. En primer lugar, daremos unas nociones básicas sobre cómo acceder a MariaDB a través de la herramienta *phpMyAdmin*, y después veremos qué librería(s) utilizar en Node.js para acceder a estos sistemas de bases de datos.

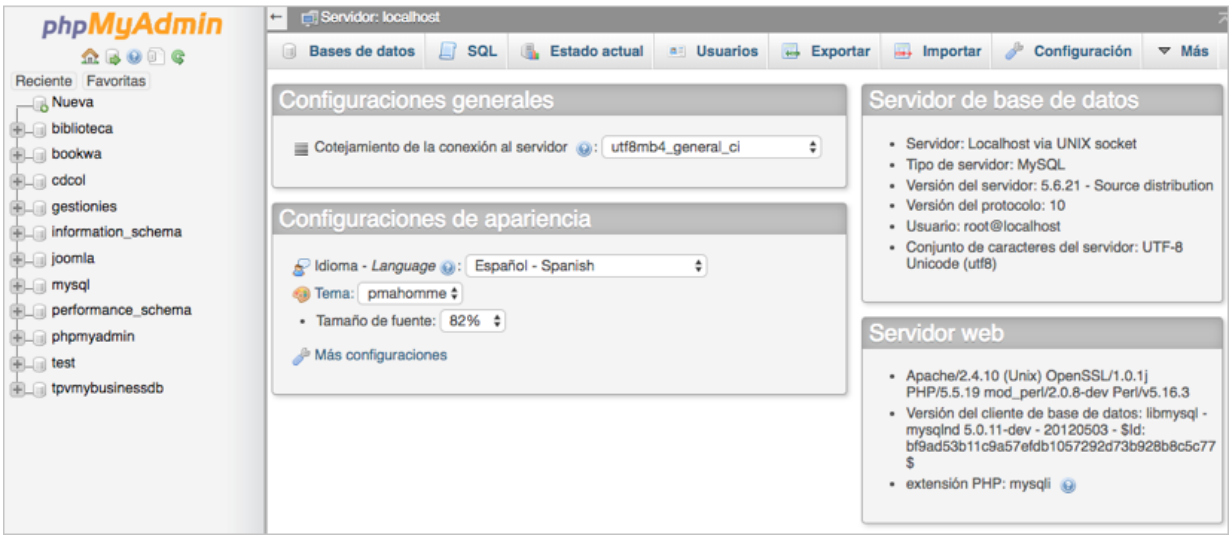
3.1 Gestión de bases de datos MariaDB/MySQL

Para trabajar con bases de datos MariaDB y poderlas gestionar cómodamente, vamos a utilizar un sistema XAMPP que, además de instalar el propio servidor de bases de datos, también instala la aplicación *phpMyAdmin*, que nos permite acceder al servidor vía web para crear nuevas bases de datos, editarlas o hacer copias de seguridad y restauraciones. En el documento de instalación se han descrito los pasos para instalar y configurar XAMPP en distintos sistemas, y también lo tienes listo para usarse en la máquina virtual del curso, con todo el software instalado.

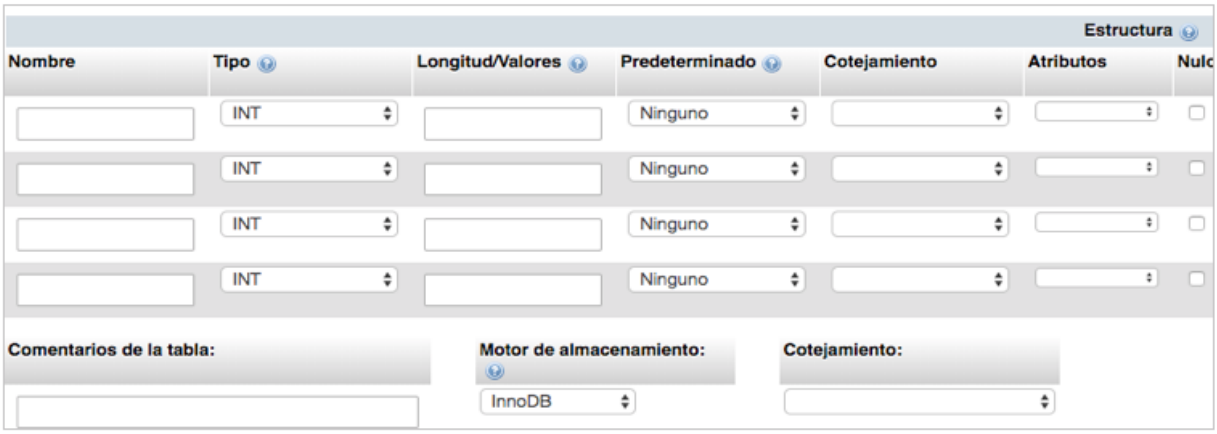
3.1.1. Uso básico de *phpMyAdmin*

Antes de ver cómo acceder a bases de datos MariaDB desde Node.js, vamos a dar las pautas necesarias para el trabajo que vamos a hacer con este SGBD: crear bases de datos, añadir tablas y campos, y exportar/importar bases de datos ya hechas.

Si ponemos en marcha *Apache* y el servidor *MySQL/MariaDB* desde el manager de XAMPP, y accedemos a la URL *http://localhost/phpMyAdmin* veremos el cliente web para gestionar las bases de datos MySQL:



En la parte izquierda aparecerán las bases de datos que tengamos ya creadas en el sistema, y un enlace "Nueva" para crear nuevas bases de datos. Después, podemos definir nuevas tablas, y los campos de cada tabla:



También podemos importar una base de datos ya hecha, desde el botón "*Importar*" de la barra de herramientas superior. Con el botón de "*Exportar*" podemos crear un *backup* de nuestra base de datos en un archivo de texto con extensión `.sql`, que luego podremos importar con *phpMyAdmin* en otro servidor (o el mismo).

3.2. La librería *mysql*

Conviene tener presente que la combinación de Node.js y MySQL no es demasiado habitual. Es bastante más frecuente el uso de bases de datos MongoDB empleando este framework, ya que la información que se

maneja, en formato BSON, es muy fácilmente exportable entre los dos extremos.

Sin embargo, también existen herramientas para poder trabajar con MySQL desde Node.js. Una de las más populares es la librería `mysql`, disponible en el repositorio NPM. Para ver cómo utilizarla, comenzaremos por crear un proyecto llamado "*PruebaContactosMySQL*" en nuestra carpeta de "*ProyectosNode/Pruebas*". Crea dentro un archivo `index.js`, y ejecuta el comando `npm init` para inicializar el archivo `package.json`. Después, instalamos la librería con el correspondiente comando `npm install`:

```
npm install mysql
```

3.2.1. Conexión a la base de datos

Una vez instalado el módulo, en nuestro archivo `index.js` lo importamos (con `require`), y ejecutamos el método `createConnection` para establecer una conexión con la base de datos, de acuerdo a los parámetros de conexión que facilitaremos en el propio método:

- `host`: nombre del servidor (normalmente `localhost`)
- `user`: nombre del usuario para conectar
- `password`: password del usuario para conectar
- `database`: nombre de la base de datos a la que acceder, de entre las que haya disponibles en el servidor al que conectamos.
- `port`: un parámetro opcional, a especificar si el servidor de bases de datos está escuchando por un puerto que no es el puerto por defecto
- `charset`: también opcional, para indicar un juego de codificación de caracteres determinado (por ejemplo, "utf8").

En el caso de *phpMyAdmin*, salvo que creemos otro usuario, el que viene por defecto es "*root*" con contraseña vacía (""). Para las pruebas que haremos en este proyecto de prueba, utilizaremos una base de datos llamada "*contactos*" que puedes descargar e importar desde los recursos de esta sesión. Teniendo en cuenta todo lo anterior, podemos dejar los parámetros de conexión así:

```
const mysql = require('mysql');

let conexion = mysql.createConnection({
    host: "localhost",
    user: "root",
    password: "",
    database: "contactos"
});
```

Después, podemos establecer la conexión con:

```
conexion.connect((error) => {
    if (error)
        console.log("Error al conectar con la BD:", err);
    else
        console.log("Conexión satisfactoria");
});
```

En el caso de que se produzca algún error de conexión, lo identificaremos en el parámetro `error` y podremos actuar en consecuencia. En este caso se muestra un simple mensaje por la consola, pero también podemos almacenarlo en algún *flag* booleano o algo similar para impedir que se hagan operaciones contra la base de datos, o se redirija a otra página.

3.2.2. Consultas

La base de datos "contactos" tiene una tabla del mismo nombre, con los atributos *id*, *nombre* y *telefono*.

id	nombre	telefono
1	Nacho Iborra	966112233
2	Arturo Bernal	965665544
3	Alex Amat	966998877

Vamos a definir una consulta para obtener resultados y recorrerlos. Por ejemplo, mostrar todos los contactos:

```
conexion.query("SELECT * FROM contactos",
(error, resultado, campos) => {
    if (error)
        console.log("Error al procesar la consulta");
    else
    {
        resultado.forEach((contacto) => {
            console.log(contacto.nombre, ":",
                contacto.telefono);
        });
    }
});
```

Notar que el método `query` tiene dos parámetros: la consulta a realizar, y un *callback* que recibe otros tres parámetros: el error producido (si lo hay), el conjunto de resultados (que se puede procesar como un vector de objetos), e información adicional sobre los campos de la consulta.

Notar también que el propio método `query` nos sirve para conectar (dispone de su propio control de error), por lo que no sería necesario el paso previo del método `connect`. En cualquier caso, podemos hacerlo si queremos asegurarnos de que hay conexión, pero cada *query* que hagamos también lo puede verificar.

Existen otras formas de hacer consultas, como son:

- Utilizando marcadores (*placeholders*) en la propia consulta. Estos marcadores se representan con el símbolo `?`, y se sustituyen después por el elemento correspondiente de un vector de parámetros que se coloca en segunda posición. Por ejemplo:

```
conexion.query("SELECT * FROM contactos WHERE id = ?", [1],
    (error, resultado, campos) => {
        ...
```

- Utilizando como parámetro del método `query` un objeto con diferentes propiedades de la consulta: la instrucción SQL en sí, los parámetros embebidos mediante *placeholders*... de forma que podemos proporcionar información adicional como *timeout*, conversión de tipos, etc.

```
conexion.query({
    sql: "SELECT * FROM contactos WHERE id = ?",
    values: [1],
    timeout: 4000
}, (error, resultado, campos) => {
    ...
```

3.2.3. Actualizaciones (inserciones, borrados, modificaciones)

Si lo que queremos es realizar alguna modificación sobre los contenidos de la base de datos (INSERT, UPDATE o DELETE), estas operaciones se realizan desde el mismo método `query` visto antes. La diferencia está en que en el parámetro `resultado` del callback ya no están los registros de la consulta, sino datos como el número de filas afectadas (en el atributo `affectedRows`), o el *id* del nuevo elemento insertado (atributo `insertId`), en el caso de inserciones con id autonumérico.

Por ejemplo, si queremos insertar un nuevo contacto en la agenda y obtener el *id* que se le ha asignado, lo podemos hacer así:

```
conexion.query("INSERT INTO contactos" +
"(nombre, telefono) VALUES " +
"('Fernando', '966566556')", (error, resultado, campos) => {
    if (error)
        console.log("Error al procesar la inserción");
    else
        console.log("Nuevo id = ", resultado.insertId);
});
```

También podemos pasar un objeto JavaScript como dato a la consulta, y automáticamente se asigna cada campo del objeto al campo correspondiente de la base de datos (siempre que los nombres de los campos coincidan). Esto puede emplearse tanto en inserciones como en modificaciones:

```
conexion.query("INSERT INTO contactos SET ?",
    {nombre: 'Nacho C.', telefono: '965771111'},
    (error, resultado, campos) => {
        ...
});
```

Si hacemos un borrado o actualización, podemos obtener el número de filas afectadas, de esta forma:

```
conexion.query("DELETE FROM contactos WHERE id > 10",
    (error, resultado, campos) => {
        if (error)
            console.log("Error al realizar el borrado");
        else
            console.log(resultado.affectedRows,
                "filas afectadas");
});
```

Intenta realizar en este punto el Ejercicio 1 del final de esta sesión.

3.3. Uso del ORM Sequelize

Sequelize es un popular ORM (*Object Relational Mapping*) que permite trabajar con bases de datos relacionales definiendo por encima nuestros propios modelos de objetos, de forma que, en nuestro código, trabajamos con los datos como si fueran objetos, pero internamente se almacenan y extraen de tablas relacionales. Actualmente, Sequelize soporta distintos SGBD relacionales, como MySQL/MariaDB, PostgreSQL o SQLite, entre otros.

Para el ejemplo que vamos a seguir en los siguientes apartados, vamos a crear un proyecto llamado "*PruebaContactosSequelize*" en nuestra carpeta de pruebas, y una base de datos vacía llamada

"contactos_sequelize" desde phpMyAdmin. Ejecuta también el comando `npm init` en el proyecto *PruebaContactosSequelize* para dejar el archivo `package.json` preparado.

3.3.1. Instalación y primeros pasos

La instalación de Sequelize es igual de sencilla que la de cualquier otro módulo de Node, a través del comando `npm`. Además, Sequelize se apoya en otras librerías para poder comunicarse con la base de datos correspondiente, y convertir así los registros en objetos y viceversa. Es lo que la propia librería denomina "dialectos" (*dialects*), y debemos incluir la(s) librería(s) del dialecto o SGBD que hayamos seleccionado.

En nuestro caso, vamos a trabajar con bases de datos MariaDB, por lo que necesitaremos incluir la librería con el *driver* correspondiente para conectar: `mariadb`, además de la propia `sequelize`.

```
npm install sequelize mariadb
```

Después, debemos incorporar Sequelize a los archivos fuente que lo necesiten en nuestro proyecto, con la correspondiente instrucción `require`. Creamos un archivo `index.js` en nuestro proyecto de pruebas creado anteriormente, y añadimos este código:

```
const Sequelize = require('sequelize');
```

A continuación, debemos establecer los parámetros de conexión a la base de datos en cuestión:

```
const sequelize = new Sequelize('nombreBD', 'usuario', 'password', {
    host: 'nombre_host',
    dialect: 'mariadb'
});
```

En el último parámetro se admiten otros campos de configuración. Podemos, por ejemplo, configurar un *pool* de conexiones a la base de datos, de forma que se auto-gestionen las conexiones que quedan libres y se reasignen a nuevas peticiones entrantes.

En nuestro caso, si conectamos con una base de datos MariaDB llamada "contactos_sequelize" usando el usuario y contraseña por defecto de *phpMyAdmin*, en el servidor local, nos quedaría una instrucción así (configurando un *pool* de 10 conexiones):

```
const sequelize = new Sequelize('contactos_sequelize', 'root', '', {
    host: 'localhost',
    dialect: 'mariadb',
    pool: {
        max: 10,
        min: 0,
        acquire: 30000,
        idle: 10000
    }
});
```

3.3.2. Definiendo los modelos

El modelo o modelos de nuestra aplicación definen las distintas clases o estructuras de datos que vamos a necesitar para gestionar la información de dicha aplicación. Para definir el modelo de nuestro ejemplo, vamos a crear una subcarpeta `models` en nuestro proyecto. Dentro, vamos a crear un archivo `contacto.js`, con la estructura que va a tener la tabla de contactos:

```
module.exports = (sequelize, Sequelize) => {

    let Contacto = sequelize.define('contactos', {
        nombre: {
            type: Sequelize.STRING,
            allowNull: false
        },
        telefono: {
            type: Sequelize.STRING,
            allowNull: false
        }
    });

    return Contacto;
};
```

Observa que a la propiedad `module.exports` le asociamos una función que recibe dos parámetros, que hemos llamado `sequelize` y `Sequelize`. Serán dos datos que llegarán de fuera cuando carguemos estos archivos, y proporcionarán la conexión a la base de datos y el acceso a la API de Sequelize, respectivamente.

Como puedes ver, hemos exportado cada modelo para poder ser utilizado desde otros archivos de nuestra aplicación. En esta página puedes consultar los tipos de datos disponibles para definir los modelos en Sequelize. También puedes ver aquí algunos validadores que podemos aplicar a cada campo, como por ejemplo comprobar si es un e-mail, si tiene un valor mínimo y/o máximo, etc.

Una vez definido el modelo (o los modelos), podemos importarlo(s) al archivo principal `index.js` con el correspondiente `require`, aunque en este caso deberemos pasarle como parámetros los objetos `sequelize` y `Sequelize` creados previamente:

```
const Sequelize = require('sequelize');
const sequelize = new Sequelize('contactos_sequelize', 'root', '', {
    ...
};
const ModeloContacto = require(__dirname + '/models/contacto');
const Contacto = ModeloContacto(sequelize, Sequelize);
```

El objeto `Contacto` que obtendremos al final nos permitirá hacer operaciones sobre la tabla "contactos" utilizando objetos en lugar de registros, como veremos a continuación.

También es posible definir relaciones entre modelos, en el caso de tener varios, para así establecer conexiones *uno a uno*, *uno a muchos* o *muchos a muchos*. Es algo que no veremos en esta sesión por requerir más tiempo del que disponemos, pero se puede consultar información al respecto aquí.

3.3.3. Aplicando los cambios

Todos los pasos que hemos definido antes no se han materializado aún en la base de datos. Para ello, es necesario sincronizar el modelo de datos con la base de datos en sí, utilizando el método `sync` del objeto `sequelize`, una vez establecida la conexión y el modelo. Esto lo haremos desde el archivo principal `index.js`.

Podemos pasarle como parámetro un objeto `{force: true}` para forzar a que se creen de cero todas las tablas y relaciones, borrando lo que haya previamente. Si no se pone dicho parámetro, no se eliminarán los datos existentes, simplemente se añadirán o modificarán las estructuras nuevas que se hayan añadido al modelo.

```
const sequelize = new Sequelize('contactos_sequelize', 'root', '', {
    ...
});
const ModeloContacto = require(__dirname + '/models/contacto');
const Contacto = ModeloContacto(sequelize, Sequelize);

sequelize.sync(/*{force: true}*/)
.then(() => {
    // Aquí ya está todo sincronizado
    // Nuestro código a continuación iría aquí
}).catch (error => {
    console.log(error);
});
```

Tras sincronizar, observa que en cada tabla que hayamos definido (en nuestro caso, sólo la tabla de "contactos") se han creado automáticamente:

- Un *id* autonumérico como clave primaria (no lo habíamos especificado en el esquema)
- Un par de campos adicionales de tipo fecha, que nos permiten almacenar la fecha de creación y de última modificación de cada registro. Estos datos se auto-actualizan cuando insertemos o modifiquemos registros utilizando los métodos proporcionados por Sequelize, que veremos más tarde.

3.3.4. Operaciones sobre los modelos

Para terminar nuestro ejemplo, veamos cómo realizar distintas operaciones sobre la base de datos con Sequelize: listados, inserciones, borrados y modificaciones.

3.3.4.1. Inserciones

Para hacer una inserción de un objeto Sequelize, podemos emplear el método estático `create`, asociado a cada modelo. Recibe como parámetro un objeto JavaScript con los campos del objeto a insertar. Después, el método `create` se comporta como una promesa, por lo que podemos añadir las correspondientes cláusulas `then` y `catch`, o bien emplear la especificación *async/await*.

Por ejemplo, así realizaríamos la inserción de un contacto en la tabla de contactos:

```
Contacto.create({
    nombre: "Nacho",
    telefono: "966112233"
}).then(resultado => {
    if (resultado)
        console.log("Contacto creado con estos datos:", resultado);
    else
        console.log("Error insertando contacto");
}).catch(error => {
    console.log("Error insertando contacto:", error);
});
```

3.3.4.2. Búsquedas

Para realizar búsquedas, Sequelize proporciona una serie de métodos estáticos de utilidad. Por ejemplo, el método `findAll` se puede emplear para obtener todos los elementos de una tabla, o bien indicar algún parámetro que permita filtrar algunos de ellos.

De esta forma implementaríamos el listado general de contactos:

```
Contacto.findAll().then(resultado => {
    console.log("Listado de contactos: ", resultado);
}).catch(error => {
    console.log("Error listando contactos: ", error);
});
```

Si quisiéramos, por ejemplo, quedarnos con el contacto "Nacho", podríamos hacer algo así:

```
Contacto.findAll({
    where: {
        nombre: "Nacho"
    }
}).then...
```

Otra búsqueda que podemos hacer de forma habitual es la búsqueda por clave, a través del método `findByPk` (*buscar por clave primaria*). Le pasaremos como parámetro en este caso el *id* del objeto a buscar. Para obtener los datos de un contacto a partir de su *id*, puede quedar así:

```
Contacto.findByPk(1).then(resultado => {
    if (resultado)
        console.log("Contacto encontrado: ", resultado);
    else
        console.log("No se ha encontrado contacto");
}).catch(error => {
    console.log("Error buscando contacto: ", error);
});
```

Aquí podéis consultar otros tipos de operadores y alternativas para hacer búsquedas filtradas.

3.3.4.3. Modificaciones y borrados

Para realizar modificaciones y borrados, primero debemos obtener los objetos a modificar o borrar. En nuestro caso vamos a realizar modificaciones y borrados individuales, por lo que emplearemos el método `findByPk` para localizar el objeto a modificar o borrar, y una vez obtenido el objeto, podemos directamente llamar al método `update` para cambiar los campos que queramos, o al método `destroy` para eliminar el objeto. En ambos casos, recogeremos los resultados de la promesa correspondiente.

Así quedaría la modificación de datos de contactos:

```
Contacto.findByPk(1).then(contacto => {
    if (contacto)
        return contacto.update({ nombre: "Otro nombre",
            telefono: "612345678" });
    else
        reject ("Error actualizando contacto");
}).then(resultado => {
    console.log("Contacto actualizado: ", resultado);
}).catch(error => {
    console.log("Error actualizando contacto: ", error);
});
```

Y así quedaría el borrado:

```
Contacto.findByPk(1).then(contacto => {
    if (contacto)
        return contacto.destroy();
    else
        reject ("Error borrando contacto");
}).then(resultado => {
    console.log("Contacto borrado: ", resultado);
}).catch(error => {
    console.log("Error borrando contacto: ", error);
});
```

> **NOTA**: si añades estas operaciones una tras otra en el archivo `index.js` de nuestro proyecto de pruebas *PruebaContactosSequelize*, debes tener en cuenta que son asíncronas (trabajan con promesas), y por tanto, no se van a ejecutar secuencialmente. Dicho de otro modo, si hacemos una inserción y a continuación un listado, es posible que dicha inserción no salga en el listado porque aún no se ha ejecutado del todo. Es preferible que ejecutes las instrucciones una a una, dejando comentadas el resto de pruebas, para verificar su funcionamiento.

Intenta realizar en este punto el Ejercicio 2 del final de esta sesión.

3.4. Ejercicios propuestos

Para los ejercicios de esta sesión, crea una subcarpeta llamada "**Sesion3**" en tu carpeta "**ProyectosNode/Ejercicios**", para dentro ir creando un proyecto para cada ejercicio.

Ejercicio 1

Crea una carpeta llamada "**Ejercicio_3_1**" dentro de la carpeta anterior "*ProyectosNode/Ejercicios/Sesion3*". Importa desde *phpMyAdmin* el *backup* de la base de datos "libros" que tienes disponible en los recursos de

esta sesión.

En el proyecto, inicializa el archivo `package.json` con `npm init`, e instala el módulo *mysql*. Crea un archivo `index.js` que cargue este módulo (con `require`), y realice estas operaciones sobre la base de datos de libros:

- Insertar 3 libros cualesquiera, con los datos que se te ocurran (observa qué campos tiene la tabla).
- Listar los libros de más de 10 euros.
- Modificar el precio del libro 1 a 35 euros.
- Borrar el libro 3.

Ejercicio 2

Crea una carpeta llamada "**Ejercicio_3_2**" dentro de la carpeta anterior "*ProyectosNode/Ejercicios/Sesion3*". Crea también una base de datos llamada "canciones" con *phpMyAdmin*.

En el proyecto, inicializa el archivo `package.json` con `npm init`, e instala los módulos *sequelize* y *mariadb*.

Ahora, vamos a definir una carpeta `models` en el proyecto, con un archivo llamado `cancion.js`. De cada canción vamos a almacenar su título (texto sin nulos), su duración en segundos (sin nulos) y el nombre del artista que la interpreta (texto sin nulos).

En el programa principal `index.js`, conecta con la base de datos, carga el modelo, sincroniza los datos y realiza las siguientes operaciones:

- Crea 2 nuevas canciones con los datos que quieras
- Muestra los datos de la primera canción
- Modifica la duración de alguna de las canciones
- Borra alguna de las dos canciones