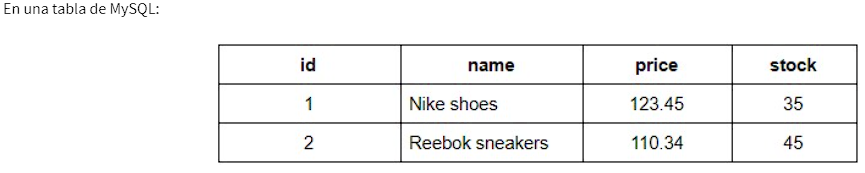
1. Introducción: ORN: sequelize
2. Promesas
3. Sequelize
4. Modelos
5. findAll, findByPk, findOne
6. Where y operadores
7. Order y Limit

1.

¿Cómo trabajamos con la DB en nuestro proyecto de express? Utilizaremos una herramienta llamada **SEQUELIZE**. La cual es un **ORM (Object Relational Mapping).** Esta ORM nos va a permitir “transformar” una tabla de BD en una serie de entidades para facilitar el acceso a los datos.





En nuestro programa, estas entidades, tienen forma de objeto literal. Como MySQL se usa ya hace mucho tiempo este tipo de herramientas tiene bastante soporte y son muy bien recibidas.

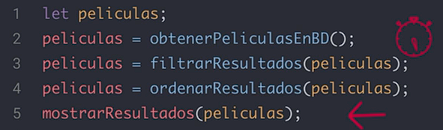
2.

Promesas: son un tipo de funciones especiales para trabajar de forma asincrónica

ejecuta un bloque de código el cual tardará un cierto tiempo y luego, en base a la respuesta obtenida, hará determinada cosa.

La mayoría de lenguajes de programación se frenen en una línea y hasta no resolverla no continúan con la siguiente línea. **JS** es un **lenguaje asincrónico** donde si una línea de pedido al BD tarda mucho tiempo (línea 2), continuará con la siguiente línea (línea 3, 4, 5) pero sin la seguridad de que el pedido haya finalizado.

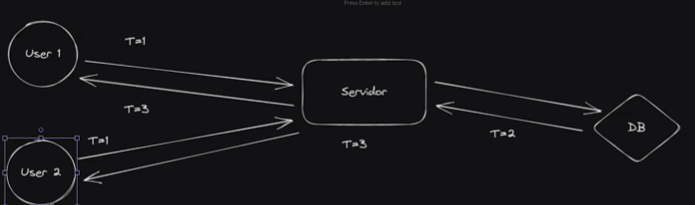
Estaría tratando de mostrar datos que todavía no tengo.



¿Por qué necesitamos un lenguaje asincrónico?

Un usuario hace un pedido al servidor, el servidor hace el pedido a la DB, la DB le da una respuesta y el servidor hacer da una respuesta al usuario. Estoy tomará 5 segundos.

Otro usuario hace, al mismo tiempo, un pedido similar al servidor, luego a DB, luego al servidor y respuesta al usuario. Si fuera un lenguaje **sincrónico,** el segundo usuario tendría que esperar que termine toda la ejecución del primer pedido del primer usuario para que recién el servidor tome el pedido del segundo.

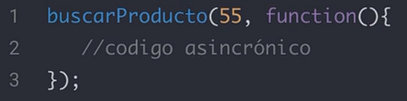


Y si llegara un tercer o cuarto usuario estaría esperando mucho más tiempo.

Es como un restaurante con mucha gente y un solo mesero.

Con las promesas podemos solucionar esto.

**Antes** utilizábamos un código como el siguiente para hacer **trabajos asincrónicos**, donde dentro de **la callback** poníamos todo el código asincrónico que necesitáramos, el callback **es la consecuencia**.



Una función registra una compra, actualiza el stock y repone un producto (sin promesas):



Esta forma de escribir código se hacía difícil de leer. Por esto **se empezaron a utilizar las promesas**.

Siguiendo con el ejemplo sin promesas, a continuación, una función con promesas que registra una compra:



La primera diferencia es que **no recibe una callback**, si no que aparece un **.then** donde escribimos la consecuencia.

Si agregamos la funcionalidad de actualizar productos y reponer producto se vería así:

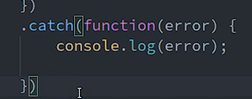


Se crea una **cadena de. then**, cuando termina una promesa comienza la siguiente.

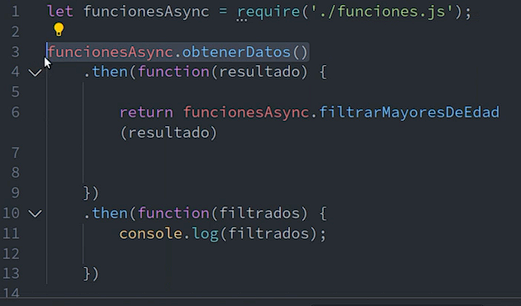
Si las promesas llegaran a fallar por algún **error** (conexión de DB por ejemplo) usamos **.catch** para atajar el error. Donde lo que este dentro del .catch ejecutara en vez de las promesas. .catch recibe un error como parámetro que nos dirá porque falló.



Si algún .then falla, atajamos ese error con un .catch al final de la función, que recibe un el error como parámetro y lo podemos imprimir en consola para ver que pasó



Ejemplo práctico: en un archivo llamado promesas.js



Lo que retorne el primer .then lo recibirá el segundo .then. Luego ponemos siempre un. catch.

Existen también el **Promise.all()**. A veces necesitaremos que dos o más promesas se resuelvan para realizar cierta acción en otra función. **Promise.all()** contendrá una array de promesas, una vez que se resuelvan todas se ejecutara un .then.

Lo primero es guardar en varaibles las promesas que necesitemos:

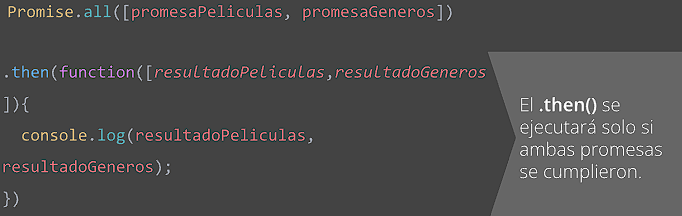




Luego usamos el método **Promise.all()** y guardamos las varaibles:



El .then recibe el array con las prmesas cumplidas y recién ahí se ejecuta:



Documentación:

<https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Using_promises>

<https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Promise/all>

3.

**Sequelize:** como dijimos el un ORM que nos ayuda a interactuar con la DB. Nos conectará con la DB con código de JS.

**Configuración:** (una sola vez por proyecto)

Primero lo instalamos usando **npm install sequelize sequelize-cli.**

**(**NO instalamos paquetes de forma global porque podría afectar a proyecto creados anteriormente que funcionen con versiones anteriores del paquete, por esto es mejor instalaros como dependencia de ese proyecto.**)**

También instalamos un paquete que indique que tipo de DB vamos a usar en este caso MySQL. El Código seria así: **npm install mysql2.**

En la carpeta raíz del proyecto crearemos un archivo llamado .**sequelizerc.** En este archivo indicaremos en que carpeta estarán almacenados todos los archivos relacionados con la DB.

Ejemplo:



Línea 4 a 7 indica donde va a guardar los archivos de sequelize

La carpeta **database** contendrá todos estos archivos.

Para crear esta carpeta y archivos usamos en código **sequelize init** si es que lo instalamos de forma global a sequelize. Si no lo instalamos de forma global usamos el código **npx sequelize-cli init** .

Dentro de todo lo que nos crea el anterior código no centraremos en dos archivos importantes: **config.js y index.js**.

A continuación, modificaremos el archivo **config.js** para establecer el nombre de la DB y otros detalles.

También es importante el archivo **index.js** que hará toda la conexión con la DB. Este archivo exporta una variable llamada **db.** Esta variable la usaremos para hacer todas las consultas a la DB dentro de los controladores.

El archivo config.js se creará con sintaxis JSON, hay dos opciones: **1.** Cambiar de .js a .json (no queremos hacer eso porque vamos a querer que esa info sea dinámica), y **2.** Agregar un **module.exports** para exportar el objeto.

También modificamos el nombre de la DB la contraseña si es que tiene, etc. Etc.

Documentación adicional:

<https://sequelize.org/>

4.

**Modelos**: son la capa del programa que permite interactuar con la base de datos, con sintaxis JS.

Para que sequelize pueda trabajar debe saber y entender como es la DB, por esto configuraremos una capa de modelo. Dentro de la arquitectura MVC, tenemos M o modelos que hacer referencia a la representación de los datos que tenemos en el sistema. El Modelo es una representación de la tabla con el nombre de la tabla o las columnas que vamos a usar.

Ejemplo:

modelo del usuario: cremao el archivo **Usuario.js** dentro de la carpeta models creada con el comando **npx sequelize-cli init.** (el nombre del archivo se recomienda que debe empezar con mayúscula y estar en singular.). este archivo exporta una función que recibe dos parámetros: el primero que es la conexión con la DB “sequelize”, y el segundo son los tipos de bases de datos para trabajar “dataType”



Todos los modelos empiezan igual, cambiando el nombre de la variable.

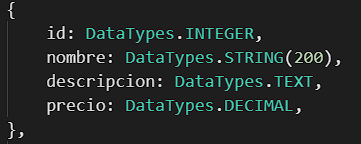
**Nosotros completaremos lo parámetros que recibe define() que son 3 importantes: alias, cols y config**

****

El **primero alias** es un string con el nombre del modelo (se suele utilizar el nombre de la tabla, es decir, el nombre del archivo en plural).



El **segundo** es un objeto literal con los nombres de los campos. Por cada atributo o campo hay que especificar más detalles en otro objeto literal.



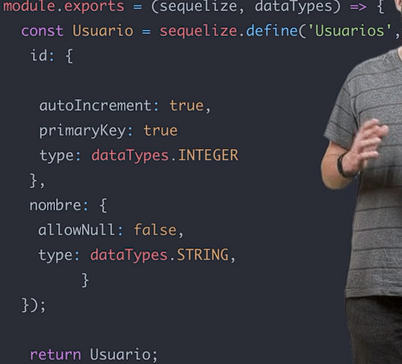
El **tercero config** con dos parámetros **tableName** con el nombre de la tabla si es necesario. Sequelize infiere que el nombre de la tabla es el nombre del modelo en plural. Si necesario aclarar el nombre de la tabla usamos tableName. Y como segundo parámetro el atributo **timestaps**: sequelize asume que por defecto las tablas tienen dos columnas llamadas “**createaAt”** y **“updateAt”** (no son obligatorias, pero es parte del estándar de las DB) indicando la fecha que se creó u la fecha que se actualizó. Si tenemos estas columnas, pero con otros nombres, se lo aclaramos así:

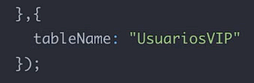


En el caso de que no los tenga ponemos como valor false.



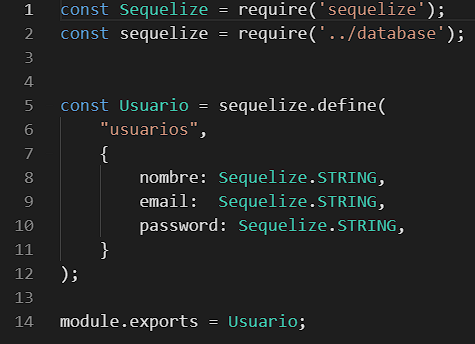
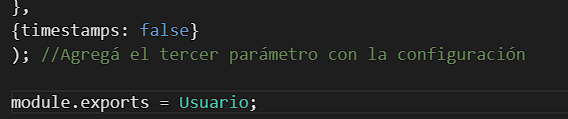
Otro ejemplo:





El campo id tiene que ser incremental (autoIncrement: true) y como es una clave primaria ponemos primaryKey: true, y algo que no debe faltar nunca es el tipo de dato, aquí usamos dataType que nos da el tipo de dato.

Otro ejemplo del playground:

5.

findAll, findByPk, findOne

todo el código que aprendimos en MySQL worckbench se puede hacer en JS más fácil

**FindeAll()** es buscar todo o sea select \* FROM tabla.

SELECT = find.

Para especificar de que tabla vamos estamos hablando la vamos a tener que importar y almacenarla en una variable



Para ver todas las películas el código sería

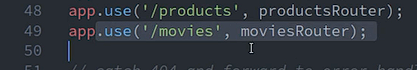


Sabe que esa es la tabla porque lo definimos en el medelo.

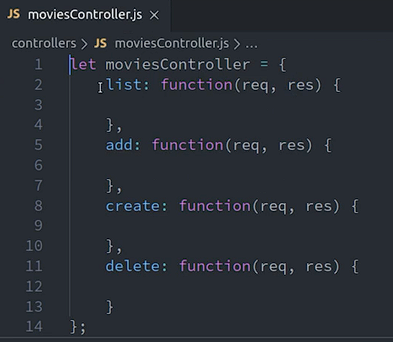
Como es código asinconico tenemos que utilizar el .then y dentro todo lo que queramos hacer con esa tabla.

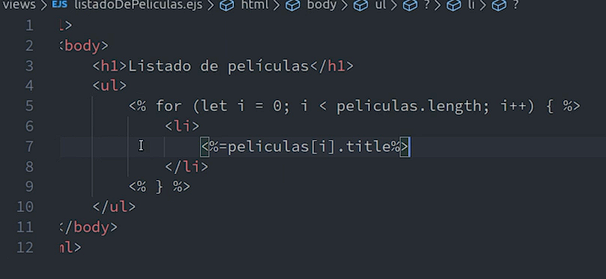
Ejemplo practico:

Tenemos en app.js un enrutados que responde a /movies dentro del archivos de rutas tenemos un /list, y en el control tenemos un controlador con varias funciones una de ellas list que enlistara las películas y una vista que recorrerá una array películas y mostrara el titulo de cada una.

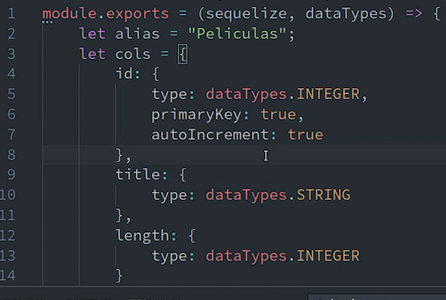








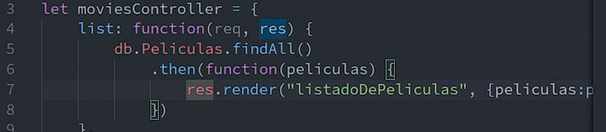
En modelo ya esta armado el modelo de películas (la tabla de películas)



En el controlles requerimos la variable db que nos da sequelize

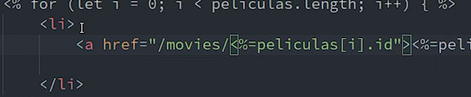


Usamos db de la tabla peliculas (que sequelize ya sabe como es y todo) entonces usamos el .findAll() que nos dara todo de esa tabla. Como es asincrónico usamos .then donde renderiza la vista deseada con las peliculas. En la vista la usamos como queramos.



**findByPk()** busca algo por su clave primaria

En la vista ponemos los títulos como hipervínculos que nos llevara a /movies/id de la película.

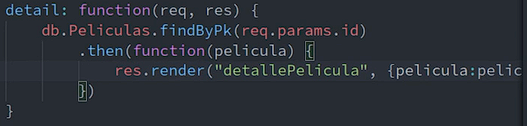


En el router hacemos el get a /:id y usando el método detail



En el controler hacemos el método detail.

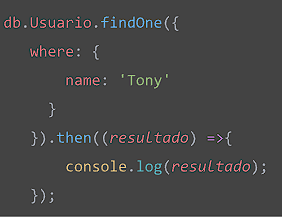
Comienza igual. Usamos el findByPk y le pasamos el req.params.id que es lo que nos lleva desde la URL (el id es porque en la ruta pusimos “:id”). Then porque todo pedido a la DB es asincrónico.



**dindOne()** nos permite buscar resultado que coincidan con el atributos indicado en el objeto literal que recibe el método, o sea el

{Where: {Name: “tony”}

}



6.

Where y operadores: condiciones a la consulta, filtros, etc.

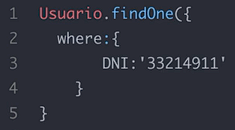
Todos los operadores de sequelize:

<https://sequelize.org/docs/v6/core-concepts/model-querying-basics/#operators>

El **método de búsqueda (findAll, findByPk, findUno)** recibe un objeto literal con el atributo where

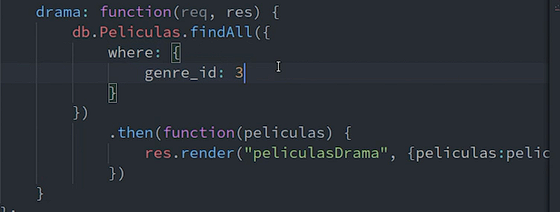


En este caso nos traerá el primer usuario que machee con el where

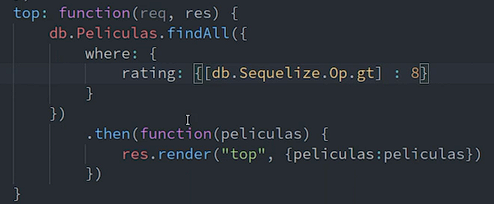


Ejemplos prácticos:

Una vista que muestra solo las películas de genero “drama”. En el controler del método que utilizaremos en la ruta /drama. Dentro del findAll hacemos un obj literal con where y la consecuencia. Luego le pasamos la película a la vista.



Cuando trabajamos con operadores que no son de igualdad (como el ejemplo anterior), y queremos hacer operación tipo películas con raiting mayor a x, por ejemplo, del lado derecho de la propiedad vamos a poner un obj literal a la izq dentro de [], llamamos a la DB, usamos el método Sequelize y ponemos Op. Dentro de Op vamso a tener todos lo operadores en este caso usamos **gt** (mayor o igual a), como valor el numero el cual estemos comparando.



7.

Order y Limit

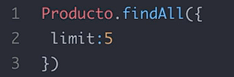
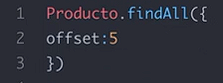
Ordenar listas y limitar la cantidad de cosas queremos.

**Order**

al finadAll le pasamos un obj literal donde pondremos el método order. Como valor pondremos un array de arrays, donde cada una de ellas será el condicionante, primero el nombre de la columna y luedo que sea desendente todo entre comillas.

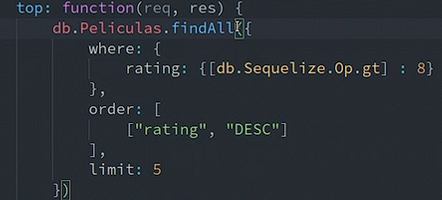
****

**Limit y ofsett** funcionan parecido

Convinar tod esto seria algo asi:

Consecuencia de que el rating sea mayor que 8, ordenado por raiting desendiente y un limute de 5 nos en la array



**Diferencia entre findAll con limit: 1, y findOne:**

findAll siempre nos dará una array, ya sea con 3 20 o 1 objeto. En cambio findOne nos dará sola mente un objeto literal.