



**Argentina  
programa  
4.0**



Ministerio de Economía  
**Argentina**

Secretaría de  
Economía del Conocimiento

***primero  
la gente***

# Clase 28: Sequelize y Node JS

## Utilizar bases de datos MySQL

## Agenda de hoy

- A. Sequelize
  - a. Fundamentos de Sequelize
  - b. Ventajas de uso
- B. Instalar Sequelize
- C. Configurar Sequelize en Node.js
- D. Los métodos más importantes
- E. Definir un CRUD con MySQL
- F. Crear los endpoint

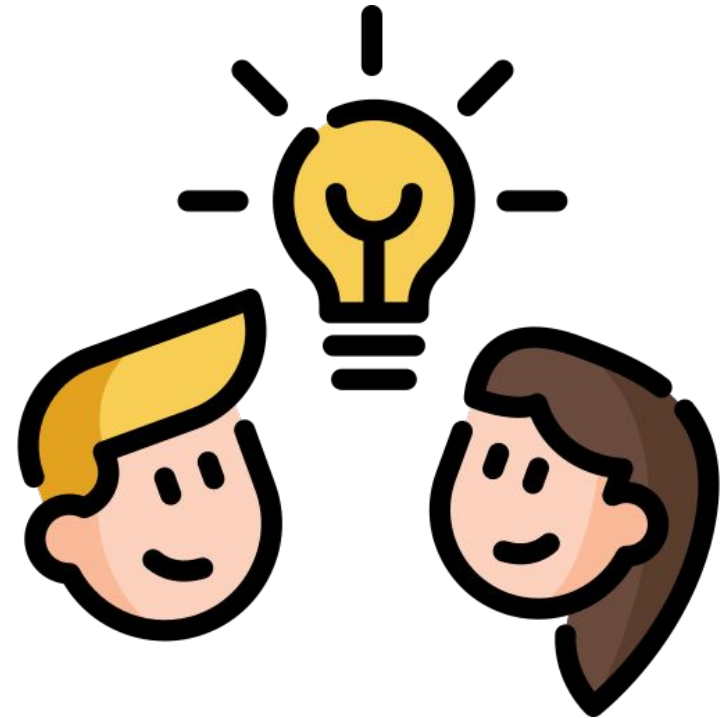


# Welcome back!

**Nos alejaremos por una clase de MySQL, para volver a trabajar con Node.js.**

**La idea es poder conocer herramientas Node que nos permitan integrar bases de datos SQL al entorno de desarrollo de aplicaciones backend.**

**Hoy veremos qué es Sequelize y cómo podemos interactuar desde este con MySQL, de cara a la segunda etapa de elaboración de nuestra pre-entrega 3.**

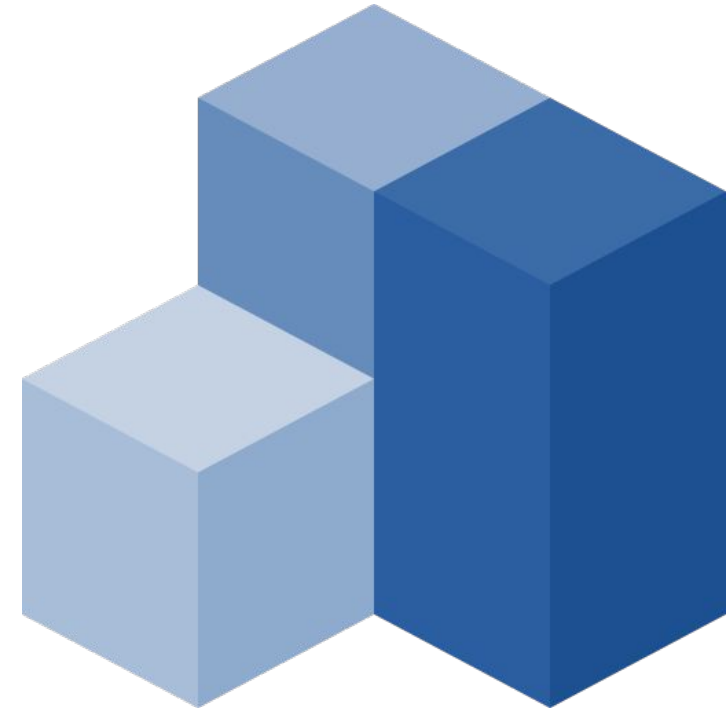


# Sequelize

# Sequelize

Sequelize es una librería JS que actúa como una herramienta de abstracción de base de datos. Simplifica y agiliza la forma en que interactuamos con bases de datos en nuestras aplicaciones.

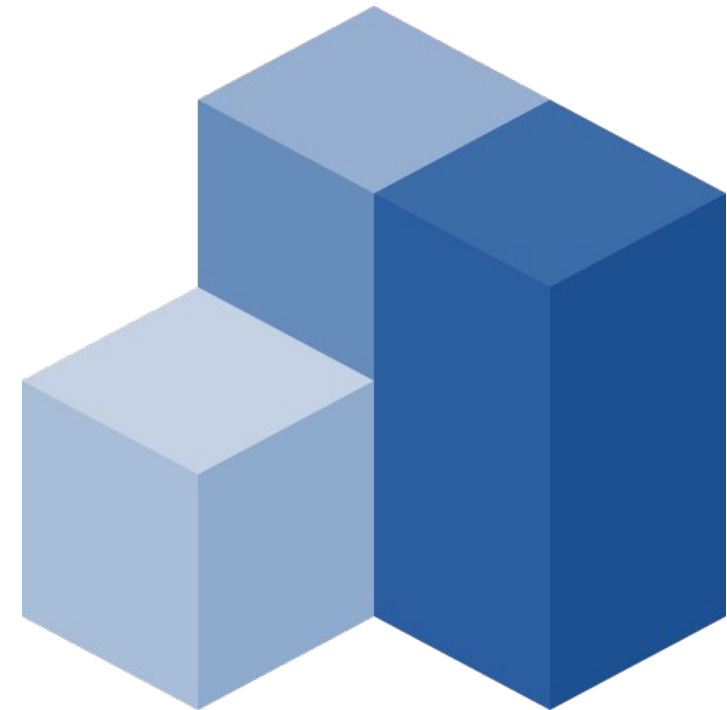
Para evitar escribir consultas SQL complejas y manejar manualmente la conexión e intercambio de datos con la bb.dd, Sequelize proporciona una capa de abstracción que nos permite realizar estas tareas más sencillamente.



# Sequelize

Con Sequelize, podemos interactuar con la base de datos utilizando un lenguaje de programación familiar como JavaScript, lo que facilita el proceso de desarrollo.

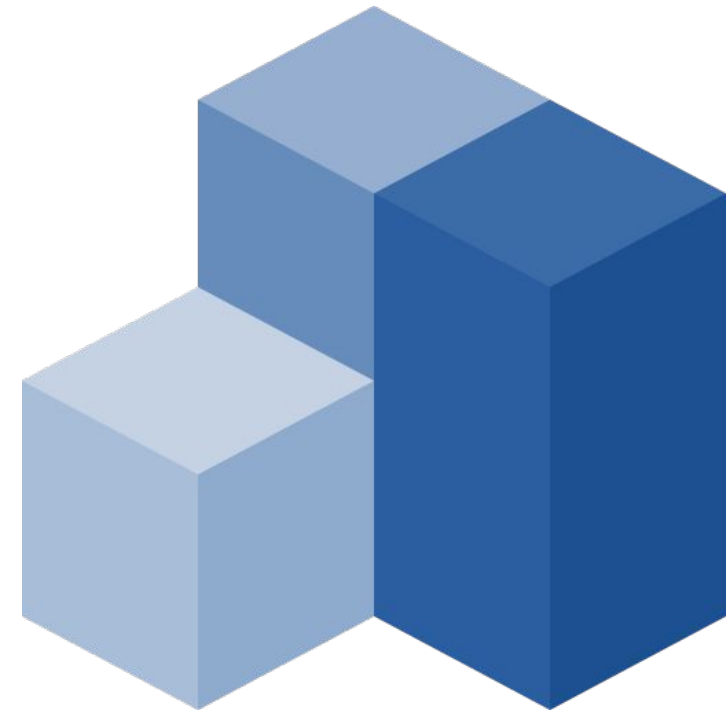
En lugar de tener que aprender y escribir consultas en SQL, podemos utilizar métodos y funciones de Sequelize para realizar operaciones de creación, lectura, actualización y eliminación de datos en la base de datos.



# Sequelize

Además, proporciona un conjunto de herramientas que nos permiten modelar los datos de nuestra aplicación de una manera más intuitiva. Podemos definir modelos que representan las tablas de la base de datos, y luego trabajar con los datos de forma similar a cómo lo haríamos con objetos en JavaScript.

Esto simplifica aún más la interacción con la base de datos y nos ayuda a organizar y estructurar la información de manera más eficiente.



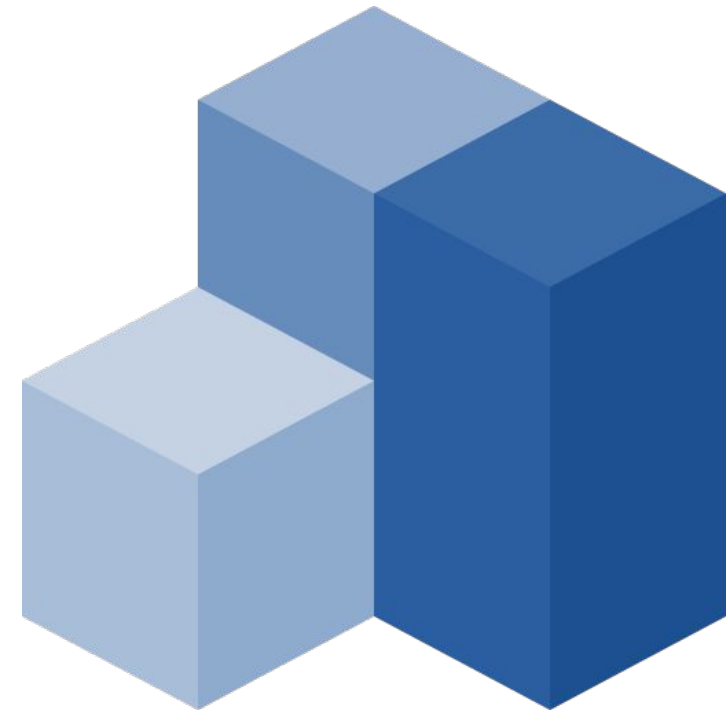


# Sequelize

## Migraciones y control de versiones

Dentro de los beneficios de utilizar Sequelize, encontramos el poder realizar migraciones, o cambios estructurales en la bb.dd, de forma controlada y organizada.

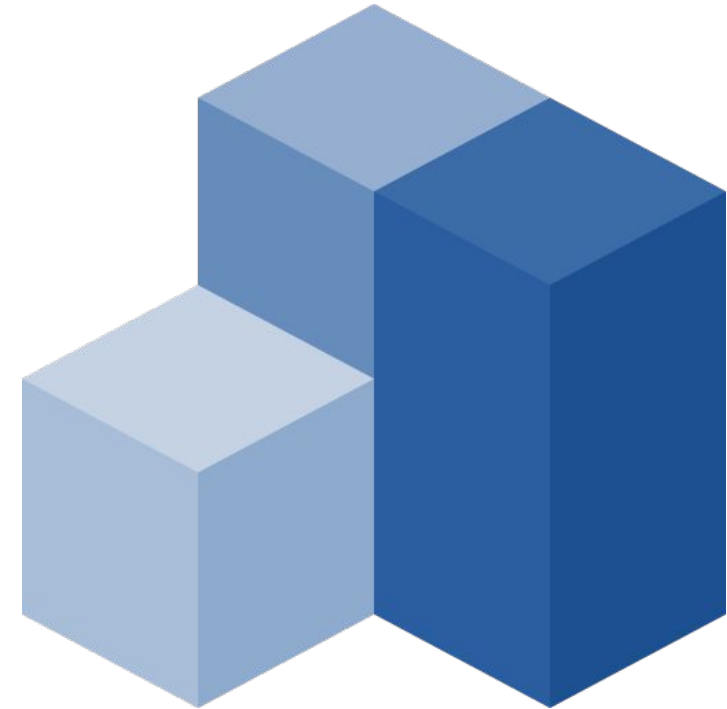
Podremos crear y aplicar las migraciones de manera incremental, para facilitar el control de versiones y el trabajo con otras compañeras del equipo de desarrollo.



# Sequelize

## Ventajas de implementar Sequelize

- Facilita la interacción con la bb.dd, simplificando las consultas y el modelado de datos
- Proporciona un control de versiones para los cambios en la estructura de la base de datos
- Es compatible con MySQL, MariaDB, PostgreSQL SQLite, SQL Server, lo que nos brinda flexibilidad en nuestras aplicaciones



# Instalar Sequelize

# Instalar Sequelize

En la web oficial: [www.sequelize.org](http://www.sequelize.org)  
encontraremos la documentación oficial  
de esta librería y la información  
referente a cómo instalar la misma  
dentro de un proyecto de Node.js.



# Instalar Sequelize

El comando para su instalación, se cumplimenta tal cual venimos trabajando hasta ahora dentro de los proyectos Node.js. NPM será nuestra vía de acceso a descargar Sequelize.

```
npm  
  
npm install --save sequelize
```

# Instalar Sequelize

Como Sequelize permite trabajar con múltiples bases de datos del tipo SQL, necesitaremos instalar aparte el soporte correspondiente de acuerdo al “*sabor*” de SQL que necesitamos trabajar.



# Instalar Sequelize

Sequelize llama a esto “*dialectos*”, tal como sucede con el idioma chino en sus diferentes regiones, el ruso, o en la mismísima Italia.

Los dialectos son los que nos permitirán comunicarnos de una forma efectiva con SQL Server, o MySQL, o Postgresql, etc.



# Instalar Sequelize

Una vez instalado Sequelize, debemos instalar manualmente el driver correspondiente a la bb.dd con la que trabajaremos. Este driver cuenta con el “dialecto” necesario para que Sequelize se comunique de manera efectiva con el motor de MySQL o cualquier otra bb.dd.

```
npm

# One of the following:
$ npm install --save pg pg-hstore # Postgres
$ npm install --save mysql2
$ npm install --save mariadb
$ npm install --save sqlite3
$ npm install --save tedious # Microsoft SQL Server
$ npm install --save oracledb # Oracle Database
```



# Instalar Sequelize

Aquí tenemos representado el comando correspondiente según la bb.dd que utilizemos.

**En nuestro caso, instalaremos el que nos permite trabajar con MySQL.**

```
npm

# One of the following:
$ npm install --save pg pg-hstore # Postgres
$ npm install --save mysql2
$ npm install --save mariadb
$ npm install --save sqlite3
$ npm install --save tedious # Microsoft SQL Server
$ npm install --save oracledb # Oracle Database
```

# Configurar Sequelize en Node.js

# Configurar Sequelize en Node.js

Ya teniendo instalada la librería Sequelize y el mecanismo de dialecto correspondiente a nuestra base de datos, ya podemos integrar Sequelize a Node.js.

Para ello, importamos en principio la librería, tal como venimos trabajando con otras librerías anteriormente.



# Configurar Sequelize en Node.js

Definimos entonces una constante que nos permite comenzar a manipular Sequelize en nuestros proyectos Node.js.

Luego de referenciarla, debemos pensar en establecer el mecanismo que nos conecte a la bb.dd.

```
Node.js  
  
const Sequelize = require('sequelize');
```

# Configurar Sequelize en Node.js

La conexión a la base de datos debemos pensar que será tratada tal como lo hicimos anteriormente con MongoDB.

- Nos conectamos
- Operamos
- Nos desconectamos

Esto nos ayudará a no sobrecargar con transacciones y múltiples conexiones al motor de MySQL.



# Configurar Sequelize en Node.js

Otro de los puntos que debemos tener en cuenta para conectarnos a la bb.dd, es volcar la información de la conexión en un archivo **.env**.

De esta forma agregamos una capa más de seguridad, y podemos manejar por separado lo que será la configuración de nuestro ambiente de desarrollo y el futuro ambiente de testing, el de producción, etcétera.



# Configurar Sequelize en Node.js

Debemos instanciar la librería **Sequelize**, informando en esta el nombre de la bb.dd, el nombre de usuario, y la contraseña, a través de parámetros separados.

Luego, en un objeto literal que también viaja como parámetro, definimos la URL del servidor de bb.dd, su dialecto, y cualquier otro parámetro adicional requerido, como ser la encriptación si es que manejamos.

```
Node.js

const sequelize = new Sequelize('database', 'username', 'password', {
  host: 'localhost',
  dialect: 'mssql',
  dialectOptions: { options: { encrypt: true } },
  define: { timestamps: false }
});
```



# Configurar Sequelize en Node.js

También debemos crear una función para autenticar la conexión a la base de datos. Para ello utilizamos el método **sequelize.authenticate()**.

Si ocurre algún error durante la autenticación, se captura el mismo mediante el bloque catch.

Este proceso se debe ejecutar previo a realizar operaciones CRUD.

```
Node.js

async function authenticate() {
  try {
    await sequelize.authenticate();
    console.log('Conexión a la base de datos establecida correctamente.');
```

```
  } catch (error) {
    console.error('Error al conectar a la base de datos:', error);
  }
}
```



# Configurar Sequelize en Node.js

También debemos crear una función para autenticar la conexión a la base de datos. Para ello utilizamos el método **sequelize.authenticate()**.

Si ocurre algún error durante la autenticación, se captura el mismo mediante el bloque **catch**.

Este proceso se debe ejecutar previo a realizar operaciones CRUD.

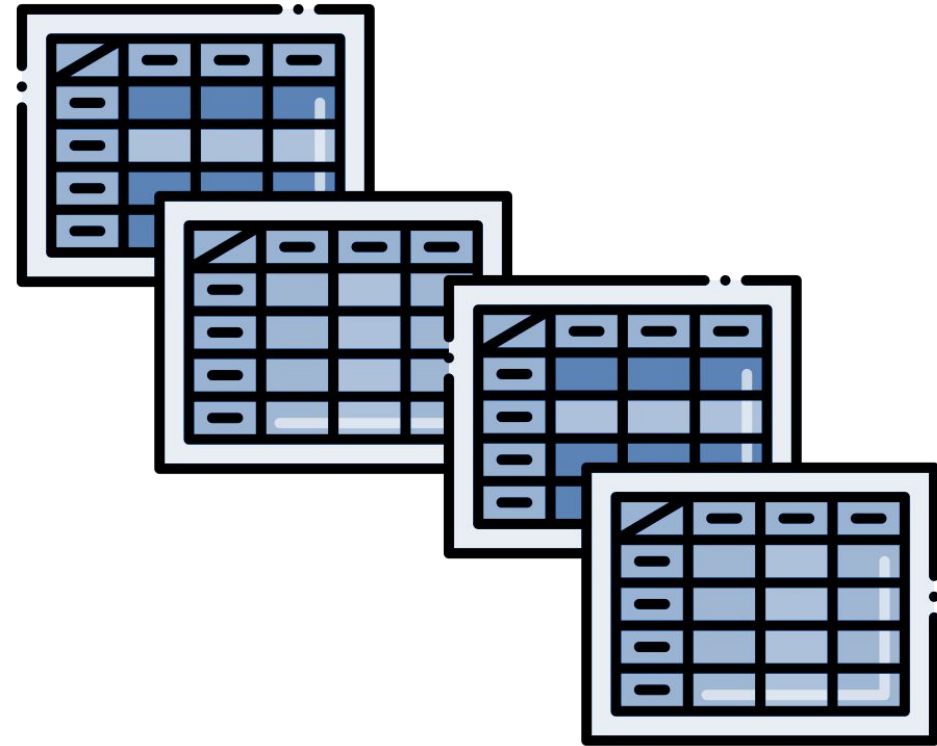
```
Node.js

async function closeConnection() {
  try {
    await sequelize.close();
    console.log('Conexión cerrada correctamente.');
```

# Definir los modelos para nuestras tablas

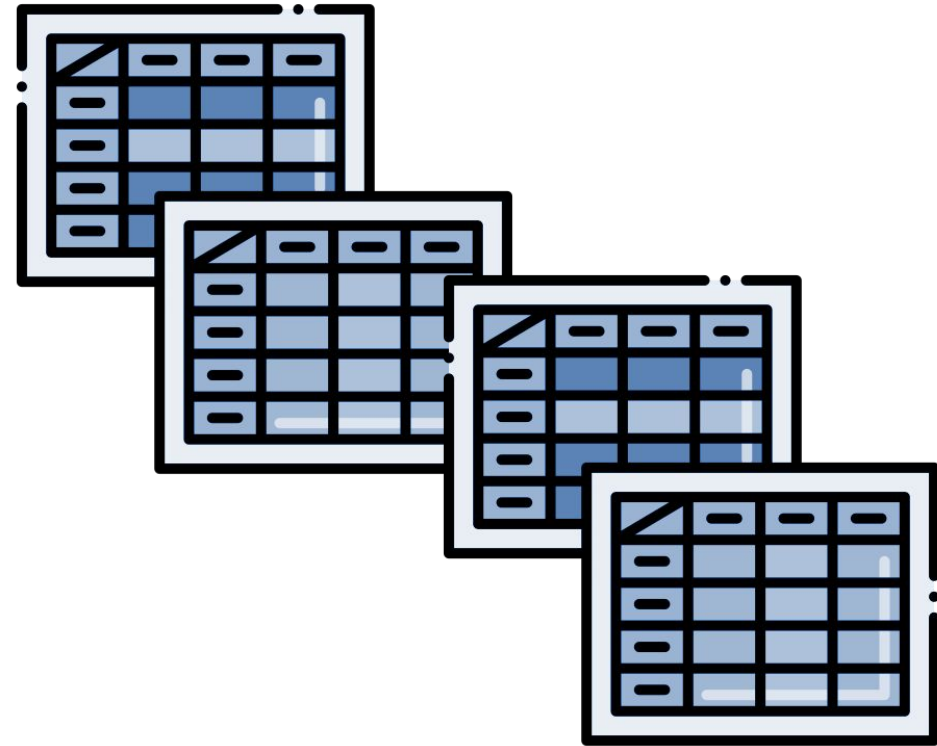
# Definir los modelos para nuestras tablas

Una vez conectados a MySQL desde Node.js y Sequelize, para comenzar a interactuar con las tablas de la bb.dd, debemos definir un modelo de datos que nos permita generar esta interacción desde Node.js.



# Definir los modelos para nuestras tablas

Este modelo de datos a generar será la estructura intermedia que nos permitirá trabajar con los registros de una Tabla SQL, desde Node.js, tal como si tuviésemos los registros de la tabla en un array de objetos JS.



# Definir los modelos para nuestras tablas

Mediante el método **define()**, sequelize nos permite definir el nombre de la estructura de datos que nos conectará con, por ejemplo, una tabla SQL llamada **contactos**. Esta tiene un **ID**, un campo **nombreCompleto**, un campo **Email**, otro campo **Telefono**, y tal vez algunos otros más.

Además de definir el modelo de datos, indicamos cuál será el tipo de dato que almacenará cada uno, entre otros datos técnicos, propios de una bb.dd relacional.

```
Node.js

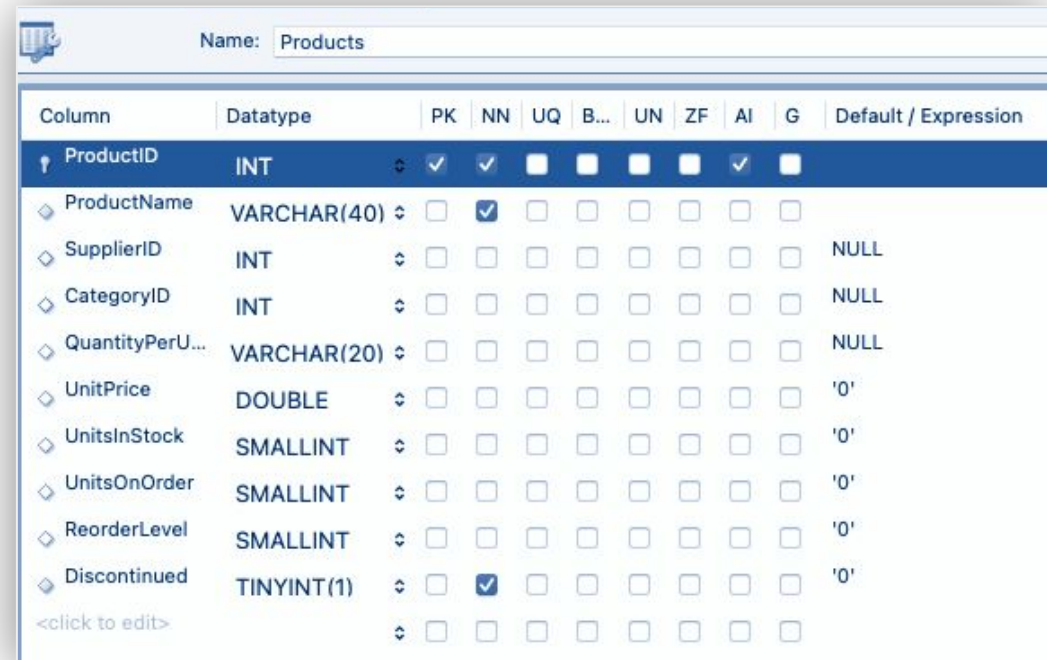
const Contactos = sequelize.define('Contactos', {
  id: {
    type: Sequelize.INTEGER,
    autoIncrement: true,
    primaryKey: true,
  },
  nombreCompleto: {
    type: Sequelize.STRING,
  },
  Email: {
    type: Sequelize.STRING,
    unique: false,
  },
  Telefono: {
    type: Sequelize.STRING,
    unique: false,
  },
})
```

# Definir los modelos para nuestras tablas

Volvemos a **MySQL Workbench** y editamos la tabla **Products**, de nuestra base de datos de ejemplo llamada **Northwind**.

Aquí tenemos la estructura de datos de la tabla **Products** con sus columnas, los tipos de datos definidos, y algunas propiedades adicionales más.

Veamos entonces cómo podemos crear un modelo con Sequelize que sea aplicable a esta tabla.



The screenshot shows the MySQL Workbench interface for editing the 'Products' table. The table structure is as follows:

Column	Datatype	PK	NN	UQ	B...	UN	ZF	AI	G	Default / Expression
ProductID	INT	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
ProductName	VARCHAR(40)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
SupplierID	INT	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	NULL
CategoryID	INT	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	NULL
QuantityPerU...	VARCHAR(20)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	NULL
UnitPrice	DOUBLE	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	'0'
UnitsInStock	SMALLINT	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	'0'
UnitsOnOrder	SMALLINT	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	'0'
ReorderLevel	SMALLINT	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	'0'
Discontinued	TINYINT(1)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	'0'
<click to edit>		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	

# Definir los modelos para nuestras tablas

Aquí tenemos una representación de la tabla **Products** convertida a un modelo de datos de Sequelize. Cada campo del modelo se define como una propiedad en el objeto pasado a **define()**, donde la clave representa el nombre del campo y el valor representa el tipo de datos y las opciones adicionales.

Por ejemplo, **ProductID** se define como una **clave primaria** (`primaryKey: true`), **autoincrementable** (`autoIncrement: true`), y **no nula** (`allowNull: false`).

```
Sequelize Model

const Product = sequelize.define('Product', {
  ProductID: {
    type: DataTypes.INTEGER, primaryKey: true, autoIncrement: true,
    allowNull: false
  },
  ProductName: {
    type: DataTypes.STRING(40), allowNull: false
  },
  SupplierID: {
    type: DataTypes.INTEGER, defaultValue: null
  },
  CategoryID: {
    type: DataTypes.INTEGER, defaultValue: null
  },
  QuantityPerUnit: {
    type: DataTypes.STRING(20), defaultValue: null
  },
  UnitPrice: {
    type: DataTypes.DOUBLE, defaultValue: 0
  },
  UnitsInStock: {
    type: DataTypes.SMALLINT, defaultValue: 0
  },
  UnitsOnOrder: {
    type: DataTypes.SMALLINT, defaultValue: 0
  },
  ReorderLevel: {
    type: DataTypes.SMALLINT, defaultValue: 0
  },
  Discontinued: {
    type: DataTypes.BOOLEAN, allowNull: false, defaultValue: false
  },
}, {
  tableName: 'Products',
  timestamps: false,
})

module.exports = Product
```

# Definir los modelos para nuestras tablas

La opción **tableName** se establece en **Products** para indicar que el modelo está asociado a la tabla "*Products*" en la base de datos.

La opción **timestamps** se establece en **false** para desactivar la creación automática de las columnas **createdAt** y **updatedAt** en el modelo.

Por último, se exporta el modelo **Product** para que pueda ser utilizado en otras partes de la aplicación.

```
Sequelize Model

const Product = sequelize.define('Product', {
  ProductID: {
    type: DataTypes.INTEGER, primaryKey: true, autoIncrement: true,
    allowNull: false
  },
  ProductName: {
    type: DataTypes.STRING(40), allowNull: false
  },
  SupplierID: {
    type: DataTypes.INTEGER, defaultValue: null
  },
  CategoryID: {
    type: DataTypes.INTEGER, defaultValue: null
  },
  QuantityPerUnit: {
    type: DataTypes.STRING(20), defaultValue: null
  },
  UnitPrice: {
    type: DataTypes.DOUBLE, defaultValue: 0
  },
  UnitsInStock: {
    type: DataTypes.SMALLINT, defaultValue: 0
  },
  UnitsOnOrder: {
    type: DataTypes.SMALLINT, defaultValue: 0
  },
  ReorderLevel: {
    type: DataTypes.SMALLINT, defaultValue: 0
  },
  Discontinued: {
    type: DataTypes.BOOLEAN, allowNull: false, defaultValue: false
  },
  tableName: 'Products',
  timestamps: false,
});

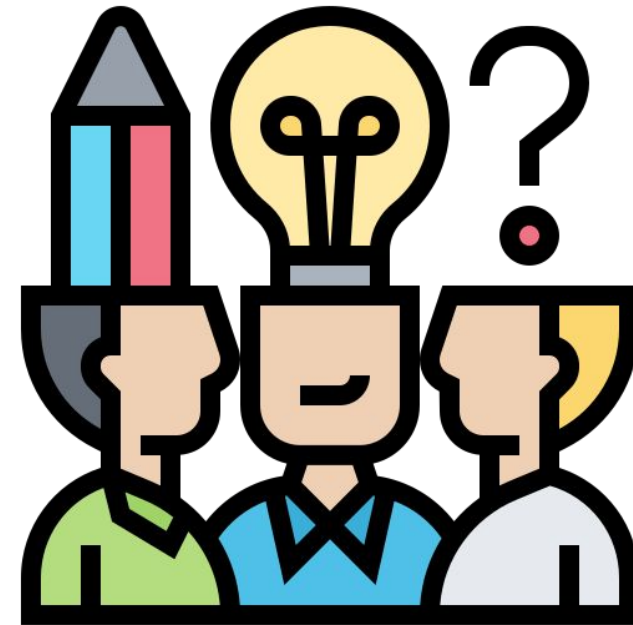
module.exports = Product
```



# **Los métodos más importantes**

# Los métodos más importantes

Sequelize ofrece varios métodos para realizar operaciones en MySQL desde Node.js.  
Veamos a continuación, una tabla con la referencia a los métodos más comunes:



# Los métodos más importantes

Métodos Sequelize	
Método	Descripción
<b>create</b>	Crea un nuevo registro en la tabla.
<b>findAll</b>	Busca todos los registros que cumplan con uno o más criterios
<b>findOne</b>	Busca un registro específico en la tabla que cumpla con ciertos criterios
<b>update</b>	Actualiza uno o varios registros en la tabla
<b>destroy</b>	Elimina uno o varios registros en la tabla

Aquí tenemos las operaciones básicas que podemos realizar con Sequelize utilizando Node.js y trabajando de forma conectada a MySQL.



# Los métodos más importantes

Métodos Sequeize	
Método	Descripción
<b>count</b>	Cuenta el número de registros en una tabla.
<b>sum</b>	Calcula la suma de un campo específico en los registros de la tabla.
<b>max</b>	Obtiene el valor máximo de un campo específico en los registros de una tabla.
<b>min</b>	Obtiene el valor mínimo de un campo específico en los registros de una tabla.

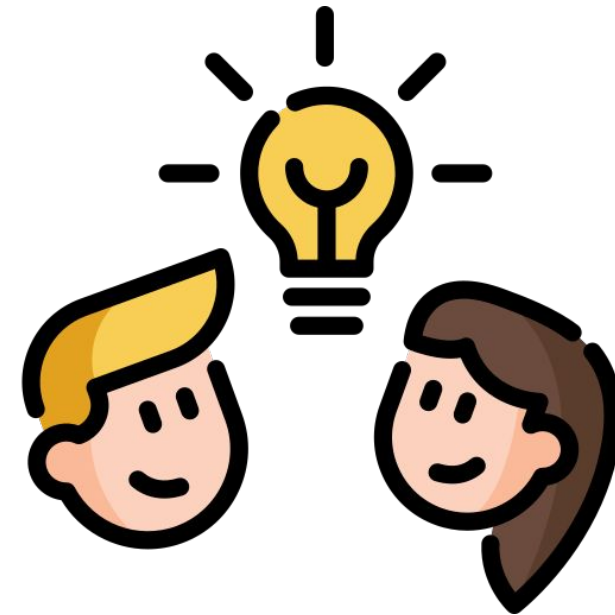
En este caso, podemos ver que estamos frente a métodos que responden a las funciones de agregación que vimos oportunamente en MySQL.

## Los métodos más importantes

**Estos son solo algunos de los métodos disponibles en Sequelize para realizar operaciones en MySQL.**

**Cada uno de ellos tiene su propia función y acepta diferentes parámetros para personalizar la consulta y los resultados obtenidos.**

**Te recomendaría consultar la documentación oficial de Sequelize para obtener más información sobre cada método y cómo utilizarlos de manera efectiva.**



## Los métodos más importantes

Aquí tenemos un ejemplo de cómo crear un nuevo registro en una hipotética tabla **Users**. Informamos el **nombre** y **Email** en un objeto, y enviamos dicho objeto como parámetro del método **create()**.

Luego, controlamos el proceso de alta con una promesa, y de igual manera cualquier error que surja.

```
Sequelize INSERT

User.create({ nombre: 'Donna Clark', email: 'donna.clark@mutiny.com' })
  .then((usuario) => {
    console.log('Usuario creado:', usuario);
  })
  .catch((error) => {
    console.error('Error al crear el usuario:', error);
  })
```

## Los métodos más importantes

En este otro ejemplo de código, utilizamos el método **findOne()** para buscar un registro específico en la tabla **Users**, a partir de su nombre.

Notemos aquí que, el objeto que enviamos como parámetro a **findOne**, se encierra en una propiedad **where**.

```
Sequelize findOne

User.findOne({ where: { nombre: 'Cameron Howe' } })
  .then((usuario) => {
    if (usuario) {
      console.log('Usuario encontrado:', usuario);
    } else {
      console.log('Usuario no encontrado');
    }
  })
  .catch((error) => {
    console.error('Error al buscar el usuario:', error);
  })
```

# Los métodos más importantes

En este otro ejemplo de código utilizamos el método **findAll()** para obtener todos los registros de la tabla **Users**.

Notemos aquí que el método no recibe ningún parámetro. Por ello, retornará un array con todos los registros encontrados.

```
Sequelize findAll

User.findAll()
  .then((usuarios) => {
    console.table(usuarios);
  })
  .catch((error) => {
    console.error('Error al buscar los usuarios:', error);
  })
```



## Los métodos más importantes

En este otro ejemplo de código utilizamos el método **findAll()** y pasamos un objeto con una propiedad **where** la cual contiene la condición de búsqueda.

La condición es definida utilizando el operador **Op.like** para realizar una comparación con el filtro **LIKE** ...

```
Sequelize findAll

User.findAll({
  where: { email: { [Op.like]: '%@mutiny.com' } }
})
.then((usuarios) => {
  console.table(usuarios)
})
.catch((error) => {
  console.error('Error al buscar los usuarios:', error)
})
```

## Los métodos más importantes

... la expresión ***%@mutiny.com*** especifica que estamos buscando registros en la columna **email** que contengan la cadena "***@mutiny.com***" en cualquier posición.

`findAll` devuelve un array de objetos que representan los usuarios encontrados que cumplan con la condición de búsqueda.

```
Sequelize findAll

User.findAll({
  where: { email: { [Op.like]: '%@mutiny.com' } }
})
.then((usuarios) => {
  console.table(usuarios)
})
.catch((error) => {
  console.error('Error al buscar los usuarios:', error)
})
```

# **Definir un CRUD con MySQL**

# Definir un CRUD con MySQL

Tomemos el modelo de ejemplo de la tabla **Northwind.Products**, y veamos cómo debemos realizar un CRUD sobre esta tabla, combinando **Express** y **Sequelize**.

En principio, tenemos un proyecto con toda la base necesaria de Express y la conexión a MySQL resuelta.

```
Sequelize INSERT

User.create({ nombre: 'Donna Clark', email: 'donna.clark@mutiny.com' })
  .then((usuario) => {
    console.log('Usuario creado:', usuario);
  })
  .catch((error) => {
    console.error('Error al crear el usuario:', error);
  })
```

# Definir un CRUD con MySQL

Tanto el modelo de datos realizado para la tabla **Products**, como el resto de los modelos de datos, deben ser almacenados en una subcarpeta del proyecto dedicada llamada **/models**.

Luego, debemos importar el archivo del modelo a nuestra aplicación Node.js para poder utilizarlo.

```
Node.js

// Importar el modelo de Product
const Product = require('./models/Product');
```

CRUD

# Definir un CRUD con MySQL

## Create

Definimos el endpoint correspondiente junto al método **post()** el cual nos permite realizar las operaciones de creación de un nuevo recurso. Este método debe ser asincrónico, y deberá contener el manejo de la operación bajo la estructura **try - catch**, para que nos garantice poder controlar cualquier tipo de error que ocurra durante el proceso de alta.

```
Node.js

app.post('/products', async (req, res) => {
  try {

  } catch (error) {
    console.error('Error al crear el producto:', error);
    res.status(500).json({ error: 'Error al crear el producto' });
  }
})
```

# Definir un CRUD con MySQL

## Create

Dentro del bloque **try {}**, desestructuramos el parámetro recibido del cliente a través de **req.body**. Luego, invocamos el método **create()**, enviando a este un objeto completo con los valores de la desestructuración realizada por cada uno de los campos.

Por último, informamos la operación exitosa mediante el **código de estado 201**.

```
Node.js

const { ProductName, SupplierID, CategoryID,
      QuantityPerUnit, UnitPrice, UnitsInStock,
      UnitsOnOrder, ReorderLevel, Discontinued } = req.body;

const nuevoProducto = await Product.create({
  ProductName, SupplierID, CategoryID, QuantityPerUnit,
  UnitPrice, UnitsInStock, UnitsOnOrder, ReorderLevel,
  Discontinued });

res.status(201).json(nuevoProducto);
```





# Definir un CRUD con MySQL

## Read

El método más simple de todos: **GET**.  
En este ejemplo retornamos todos los registros de la tabla **Products**. Si ocurre algún error en el proceso, respondemos con un **código de estado 500**.

Al igual que el resto de los endpoints, este también debe ser asincrónico.

```
Node.js

try {
  const productos = await Product.findAll();
  res.json(productos);
} catch (error) {
  console.error('Error al consultar los productos:', error);
  res.status(500).json({ error: 'Error al consultar los productos' });
}
```

# Definir un CRUD con MySQL

## Read One

También podemos utilizar **GET** para ubicar un solo registro. Para ello, obtenemos el ID del producto desde **req.params.productId**. Utilizamos **Product.findByPk()** para buscar el producto por su **ID**, y si lo encuentra lo retornamos, sino, retornamos el error con el **código de estado 404**.

Al igual que el resto de los endpoints, este también debe ser asincrónico.

```
Node.js

try {
  const productId = req.params.productId;
  const producto = await Product.findByPk(productId);

  !producto ? res.status(404).json({ error: 'Producto no encontrado' })
    : res.json(producto);

} catch (error) {
  console.error('Error al buscar el producto:', error);
  res.status(500).json({ error: 'Error al buscar el producto' });
}
```

# Definir un CRUD con MySQL

## Update

En este ejemplo modificamos un registro existente. Para ello, obtenemos los datos actualizados desde **req.body**. Usamos el método **Product.findByPk()**, para buscar el producto por su **ID**.

Si no existe, enviamos un **error 404** como respuesta. Si existe, usamos el método **Product.update()** junto a **updatedData** para que actualice el registro.

```
Node.js

try {
  const productId = req.params.productId;
  const updatedData = req.body;
  const producto = await Product.findByPk(productId);
  !producto ? res.status(404).json({ error: 'Producto no encontrado' })
    : await producto.update(updatedData);

  res.json(producto);
} catch (error) {
  console.error('Error al actualizar el producto:', error);
  res.status(500).json({ error: 'Error al actualizar el producto' });
}
```

Ten presente que en este ejemplo, el endpoint también debe ser asincrónico y utilizar el método **PUT**.



# Definir un CRUD con MySQL

## Delete

En este ejemplo eliminamos un registro existente obteniendo los datos del mismo desde **req.params.productId**. Usamos el método **Product.findByPk()**, para buscar el producto por su **ID**.

Si no existe, enviamos un **error 404** como respuesta. Si existe, eliminamos el mismo mediante el método **Product.destroy()**.

```
Node.js

try {
  const productId = req.params.productId;
  const producto = await Product.findByPk(productId);

  !producto ? res.status(404).json({ error: 'Producto no encontrado' });
              : await producto.destroy();

  res.json({ message: 'Producto eliminado correctamente' });
} catch (error) {
  console.error('Error al eliminar el producto:', error);
  res.status(500).json({ error: 'Error al eliminar el producto' });
}
```

Ten presente que en este ejemplo, el endpoint también debe ser asíncronico y utilizar el método **DELETE**.



**Crear los endpoint**

# Crear los endpoint

Para este modelo de ejemplo debemos crear los siguientes endpoint funcionales:

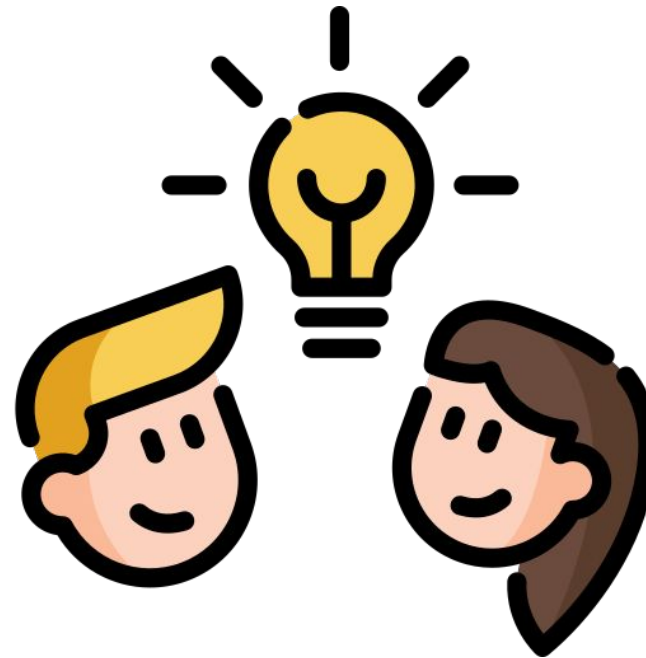
Endpoint	Método	Descripción
<b>/products</b>	<b>GET</b>	Obtienes el listado total de los registros, utilizando <b>modelo.findAll()</b> .
<b>/products:id</b>	<b>GET</b>	Obtienes un registro específico, utilizando <b>modelo.findAll(Id)</b> .
<b>/products</b>	<b>POST</b>	Creas un nuevo registro en la tabla, utilizando <b>modelo.create(param1, param2, etc)</b> .
<b>/products</b>	<b>PUT</b>	Creas un nuevo registro en la tabla, utilizando <b>modelo.update(updatedData)</b> .
<b>/products/:id</b>	<b>DELETE</b>	Eliminas un registro de la tabla, utilizando <b>modelo.destroy()</b> .



## Crear los endpoint

**Ya lo analizamos en los ejemplos de código, pero igual lo reforzamos:**

**Todos los métodos deberán ser asincrónicos, y las operaciones (CRUD) a realizar, deben estar controladas por un bloque try - catch.**



# **Pre-Entrega 3**



# Pre-Entrega 3

Aprovecha el contenido visto en la clase de hoy, para seguir diseñando otra parte del tramo del trabajo final. En esta oportunidad, define la estructura base del proyecto creando el código base de Node.js, con Express.

Integra la librería Sequelize, genera la conexión, y crea los modelos de datos para las tablas del proyecto Traileflix.



## Pre-entrega 3

Al crear la conexión desde Node.js a MySQL, recuerda **definir el host**, el **usuario** y la **contraseña**, dentro de un archivo **.env**.

Prueba la conexión a MySQL, enviando un mensaje a la consola (Terminal) de Node.js.

Crea la carpeta **/models**, donde almacenarás los modelos de las tablas SQL que hayan surgido de nuestro encuentro anterior.



## Pre-entrega 3

Define el modelo que tendrán todos los endpoints de tu proyecto, a través del código JS.

Ten presente que, el endpoint principal de trailerflix, deberá retornar el set de datos tal como tenemos en el modelo de archivo JSON.

Para resolver esto, debes crear una vista SQL respetando la estructura del archivo JSON.



# Muchas gracias.



Ministerio de Economía  
**Argentina**

Secretaría de  
Economía del Conocimiento

*primero  
la gente*