



**Argentina
programa
4.0**



Ministerio de Economía
Argentina

Secretaría de
Economía del Conocimiento

*primero
la gente*

Clase 28: Sequelize y Node.js

Agenda de hoy

- Sequelize
 - Instalación
 - Configuración
- Separando la lógica y la capa de negocios
- Definir los modelos de las tablas
- Volver a acceder a los datos
- API REST con Express y MySQL

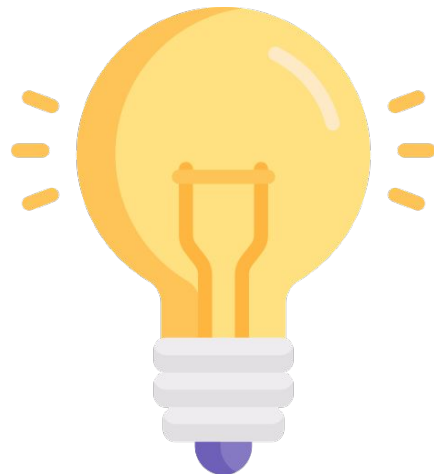


Welcome back!

Volvemos al ruedo para continuar trabajando con Node.js.

La idea es poder conocer herramientas de Node.js 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 del próximo trabajo integrador.



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

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 consultas en SQL, podemos utilizar métodos y funciones de Sequelize para realizar operaciones (CRUD) de creación, lectura, actualización y eliminación de datos en la base de datos.



Sequelize

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

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

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



Sequelize

Instalar Sequelize

Instalar Sequelize

- 1) Iniciamos un proyecto desde cero con **Node.js**.
 - a) Abre luego la ventana **Terminal** para instalar las dependencias
- 2) El siguiente paso es instalar **sequelize** y **mysql2**. Ambas son las dependencias que necesitamos para establecer la conexión a la bb.dd. MySQL que tenemos localmente.

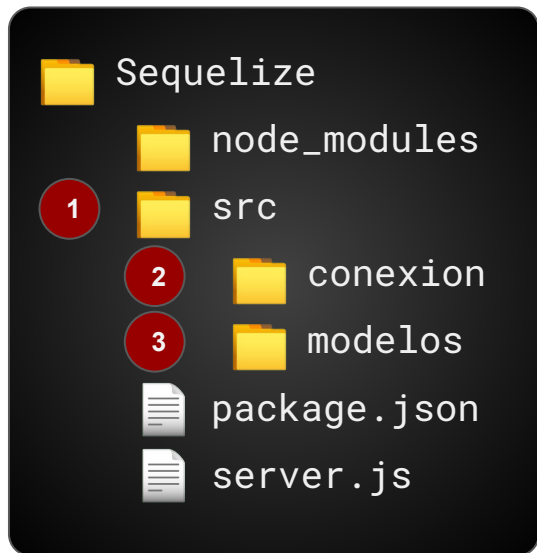
*(Recuerda anteponer el comando **sudo** en el punto (2), si utilizas **Linux** o **MacOS**)*

```
1- npm init -y
```

```
2- npm install sequelize mysql2
```

Instalar Sequelize

Finalizado el paso anterior, definiremos la estructura de carpetas de nuestro proyecto, a partir del siguiente modelo:



1

src será la subcarpeta que almacene todo archivo adicional con lógica de conexión u operaciones generales de nuestra aplicación.

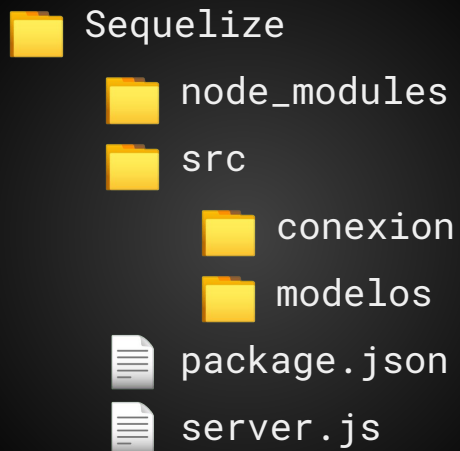
2

La subcarpeta **conexion**, contendrá el código base de Sequelize que nos conecta a la bb.dd. MySQL

3

La subcarpeta **modelos**, almacenará la información correspondiente a cada tabla de la bb.dd. a la cual necesitemos acceder.

Instalar Sequelize



Si bien este será el modelo definitivo de la estructura de nuestro proyecto web, primero **haremos una conexión a la base de datos MySQL, la definición de un modelo de datos** para una tabla, y la **obtención de todos los registros** de dicha tabla, en un único archivo **.js**.

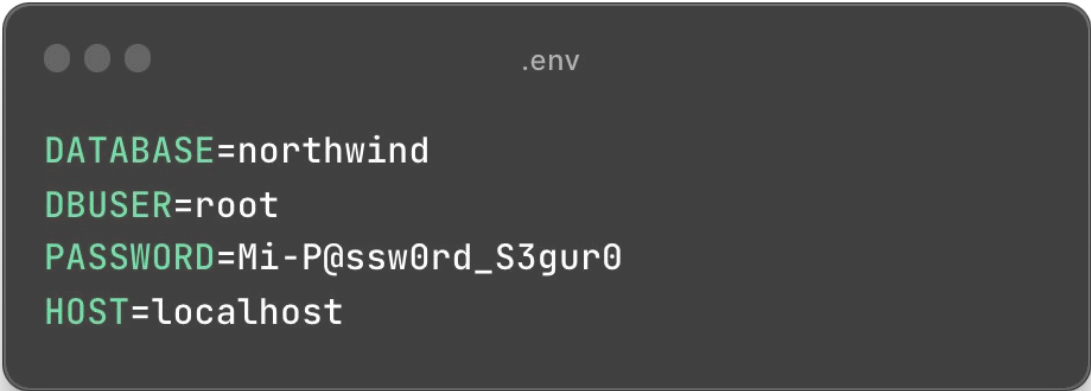
Luego nos ocuparemos de separar en diferentes archivos JS toda esta lógica y capa de negocios.

Configurar Sequelize

Configurar Sequelize

Creamos un archivo **.ENV** para almacenar una variable de entorno para la base de datos, otra para el usuario de MySQL, otra variable más para el Password de MySQL, y la última variable con la información del Host de la base de datos.

Será **localhost** si la tenemos instalada de forma local.

A screenshot of a dark-themed code editor window titled ".env". The window contains four lines of text in a light green monospace font: "DATABASE=northwind", "DBUSER=root", "PASSWORD=M!-P@ssw0rd_S3gur0", and "HOST=localhost".

```
.env  
  
DATABASE=northwind  
DBUSER=root  
PASSWORD=M!-P@ssw0rd_S3gur0  
HOST=localhost
```


Configurar Sequelize

Luego importamos **sequelize** desestructurando el objeto homónimo y el objeto **DataTypes**.

Seguido a ello, armamos la cadena de conexión, utilizando para los datos sensibles, las variables de entorno.

Recordemos instalar la dependencia **dotenv** para aprovechar las variables de entorno.

```
const { Sequelize, DataTypes } = require('sequelize');
const dotenv = require('dotenv');
dotenv.config();

const sequelize = new Sequelize(process.env.DATABASE, process.env.DBUSER,
process.env.PASSWORD, {
  host: process.env.HOST,
  dialect: 'mysql',
})
```

```
/> sudo npm install dotenv
```

Configurar Sequelize

A continuación, definiremos la estructura de la tabla **Products** correspondiente a la base de datos **Northwind**.

Sequelize requiere que armemos un Modelo de datos, a través de su método **define()**. En este debemos reflejar los tipos de datos de cada campo de la tabla **Products**, tal como si estuviésemos creándola en MySQL.

Al final declaramos **tableName**, con el nombre de la tabla, y definimos que la misma no posee datos del tipo **timestamp**.

```
const Product = sequelize.define('Product', {
  productID: {
    type: DataTypes.INTEGER,
    primaryKey: true,
    autoIncrement: true,
  },
  productName: {
    type: DataTypes.STRING,
    allowNull: false,
  },
  supplierID: {
    type: DataTypes.INTEGER,
    allowNull: false,
    default: 1,
  },
  categoryID: {
    type: DataTypes.INTEGER,
    allowNull: false,
    default: 1,
  },
  quantityPerUnit: {
    type: DataTypes.STRING,
    allowNull: false,
    default: "N/A",
  },
  ...
});
```

```
UnitPrice: {
  type: DataTypes.DOUBLE,
  allowNull: false,
  default: 0.00,
},
UnitsInStock: {
  type: DataTypes.SMALLINT,
  allowNull: false,
  default: 0,
},
UnitsOnOrder: {
  type: DataTypes.SMALLINT,
  allowNull: false,
  default: 0,
},
ReorderLevel: {
  type: DataTypes.SMALLINT,
  allowNull: false,
  default: 1,
},
Discontinued: {
  type: DataTypes.BOOLEAN,
  allowNull: false,
  default: 0,
},
}, {
  tableName: 'Products',
  timestamps: false,
});
```

Configurar Sequelize

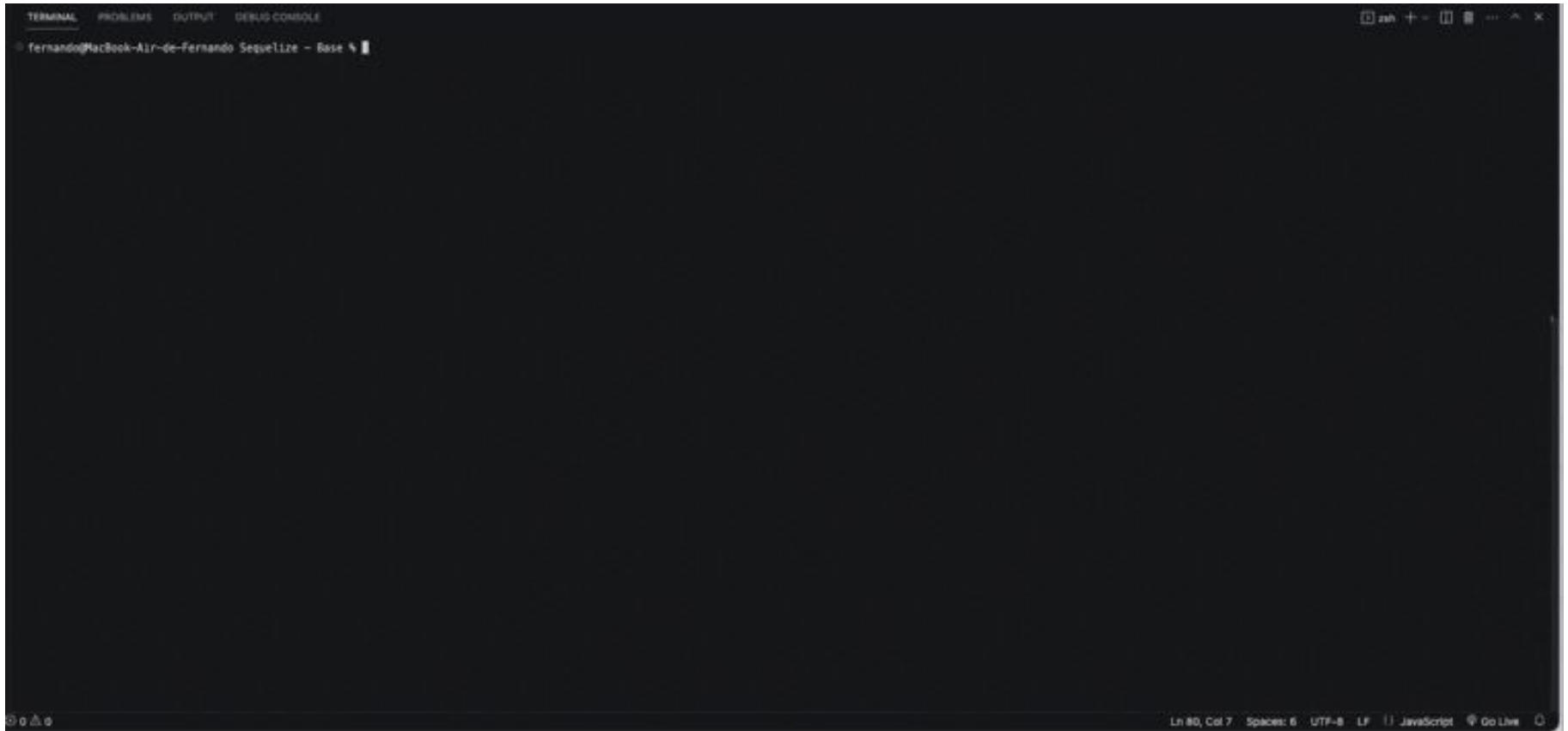
Por último, declaramos una función asíncronica la cual engloba todos los pasos necesarios para conectarnos con la base de datos Northwind del motor MySQL, petitionar la tabla **Products**, obtener todos los datos, y convertirlos a un array de objetos para que puedan finalmente visualizarse en la ventana **Terminal**, mediante **console.table()**.

También nos ocupamos de manejar cualquier error que surja en alguna de estas operaciones.

```
async function main() {
  try {
    await sequelize.authenticate()
    console.log('Conexión exitosa a la base de datos.')
    await Product.sync()
    const allProducts = await Product.findAll()
    const allProductsData = allProducts.map(product => product.dataValues)
    console.table(allProductsData)

  } catch (error) {
    console.error('Error de acceso a la BB.DD:', error)
  } finally {
    sequelize.close()
  }
}

main()
```



Si todo fue bien configurado, ya podemos probar en la Terminal, el acceso a los registros de la tabla **Northwind.Products**.

Separando la lógica y la capa de negocios

Separando la lógica y la capa de negocios

Ahora sí, vamos a separar todo el código en diferentes archivos JS:

Dentro de la subcarpeta
/src/conexion, creamos un
archivo llamado **connection.js**.

En este, agregamos el código
correspondiente a la
dependencia Sequelize y el
enlace a la bb.dd. MySQL.

```
const { Sequelize } = require('sequelize');
const dotenv = require('dotenv');
dotenv.config();

const sequelize = new Sequelize(process.env.DATABASE,
process.env.DBUSER, process.env.PASSWORD, {
  host: process.env.HOST,
  dialect: 'mysql',
});

module.exports = sequelize;
```

Separando la lógica y la capa de negocios

importamos la dependencia **sequelize**.

indicamos la bb.dd. a conectarnos. Por ejemplo: **northwind**.

el usuario MySQL definido en la instalación del motor; usualmente **root**.

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

```
const sequelize = new Sequelize('bb_dd', 'dbuser', 'password', {  
  host: 'localhost',  
  dialect: 'mysql',  
})
```

La contraseña correspondiente a MySQL.

definimos el host de la bb.dd. Si es local, directamente ponemos **localhost**.

le indicamos el dialecto a utilizar. En nuestro caso; **mysql**.

 *Parli italiano?*

Separando la lógica y la capa de negocios

En nuestro caso, aprovechamos las variables de entorno para una correcta configuración de los parámetros de conexión.

Finalmente lo exportamos como un módulo JS, para utilizarlo desde el archivo principal de nuestro proyecto.

```
const { Sequelize } = require('sequelize');
const dotenv = require('dotenv');
dotenv.config();

const sequelize = new Sequelize(process.env.DATABASE,
process.env.DBUSER, process.env.PASSWORD, {
  host: process.env.HOST,
  dialect: 'mysql',
});

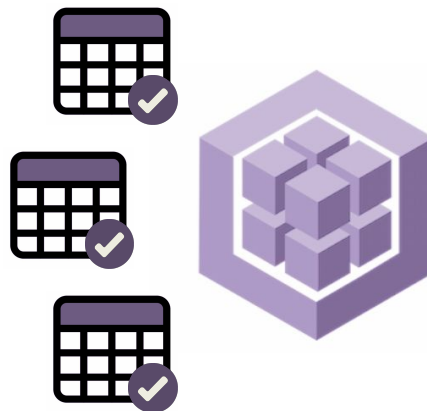
module.exports = sequelize;
```


Definir los modelos de las tablas

Definir los modelos de las tablas

Como vimos anteriormente, en Sequelize debemos definir un modelo de datos que nos permita operar con las diferentes tablas de una base de datos.

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 de las tablas

```
const Product = sequelize.define('Product', {  
  productID: {  
    type: DataTypes.INTEGER,  
    primaryKey: true,  
    autoIncrement: true,  
  },  
  productName: {  
    type: DataTypes.STRING,  
    allowNull: false,  
  },  
  SupplierID: {  
    type: DataTypes.INTEGER,  
    allowNull: false,  
    default: 1,  
  },  
  CategoryID: {  
    type: DataTypes.INTEGER,  
    allowNull: false,  
    default: 1,  
  },  
  QuantityPerUnit: {  
    type: DataTypes.STRING,  
    allowNull: false,  
    default: "N/A",  
  },  
},
```

En el archivo **./src/modelos/product.js** desestructuramos el objeto **DataTypes** integrado en **Sequelize**.

Este será quien permita definir cada tipo de datos necesario, por cada una de las propiedades que definimos en el objeto **Product** que estamos creando.

Definir los modelos de las tablas

```
, {  
  tableName: 'Products',  
  timestamps: false,  
};
```

```
module.exports = Product;
```

Los parámetros finales, correspondiente a **tableName**, le indican a nuestro modelo con qué tabla MySQL estamos enlazándonos.

timeStamps es una serie de métricas que pueden tener las tablas MySQL. En este caso no queremos tener las mismas de este lado, así que las obviamos con el valor **false**.

Por último, exportamos el módulo.

Volver a acceder a los datos

Volver a acceder a los datos

```
const sequelize = require('./src/conexion/connection');
const Product = require('./src/modelos/product');

async function main() {
  try {
    await sequelize.authenticate();
    console.log('Conexión exitosa a la base de datos.');
    await Product.sync();
    const allProducts = await Product.findAll();
    const allProductsData = allProducts.map(product => product.dataValues);
    console.table(allProductsData);
  } catch (error) {
    console.error('Error al conectar o consultar la base de datos:', error);
  } finally {
    sequelize.close();
  }
}
```

Importamos los archivos **connection.js** y **product.js** para nuestro archivo principal.

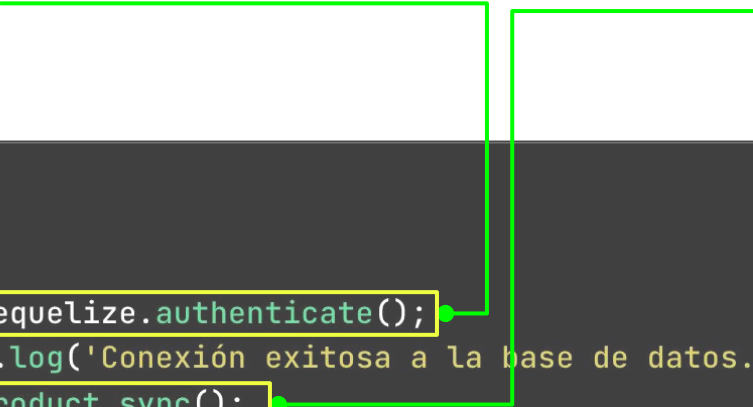
Luego, definimos una función que se ocupará de realizar los pasos correspondientes para llegar a los datos y poder visualizarlos en la **Terminal**.

Volver a acceder a los datos

sequelize.authenticate() abre la conexión con la base de datos.

Product.sync() se ocupa de sincronizar la tabla **Products** con nuestro modelo de datos **Product**.

```
try {  
  await sequelize.authenticate();  
  console.log('Conexión exitosa a la base de datos.');
```

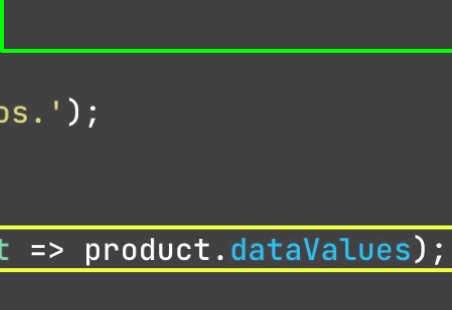


```
  await Product.sync();  
  const allProducts = await Product.findAll();  
  const allProductsData = allProducts.map(product => product.dataValues);  
  console.table(allProductsData);  
  ...  
}
```

Volver a acceder a los datos

Mapeamos el objeto **allProducts** para obtener de cada **producto**, la información de la propiedad **dataValues**.

```
try {  
  await sequelize.authenticate();  
  console.log('Conexión exitosa a la base de datos.');
```



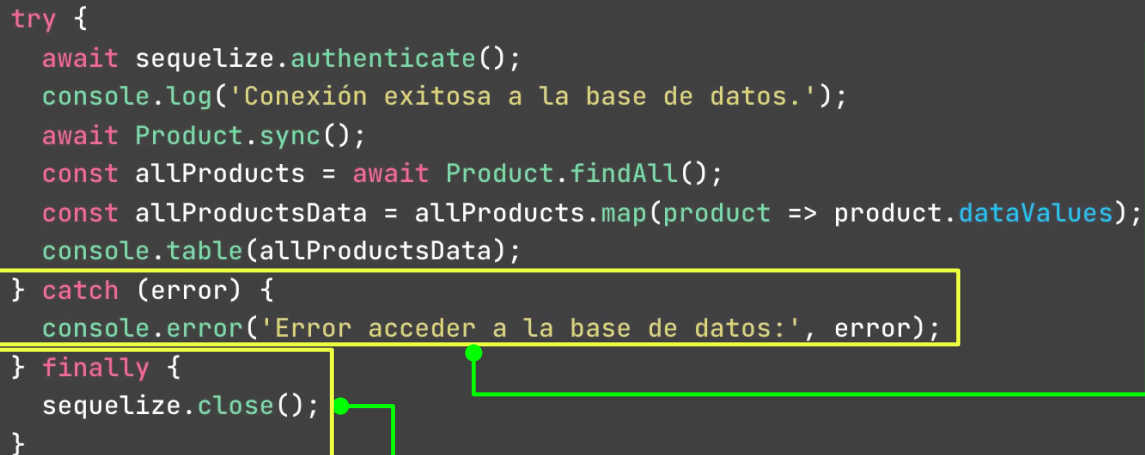
```
  await Product.sync();  
  const allProducts = await Product.findAll();  
  const allProductsData = allProducts.map(product => product.dataValues);  
  console.table(allProductsData);  
  ...  
}
```

The diagram illustrates the mapping process. A green line originates from the text 'Mapeamos el objeto allProducts...' and points to the `allProducts` argument in the `map` function. Another green line points from the `product` parameter in the arrow function to the `product.dataValues` property access. A third green line points from the `console.table(allProductsData);` line to the text 'Finalmente, vemos la información en console.table().'

Finalmente, vemos la información en **console.table()**.

Volver a acceder a los datos

```
try {  
  await sequelize.authenticate();  
  console.log('Conexión exitosa a la base de datos.');
```



```
  await Product.sync();  
  const allProducts = await Product.findAll();  
  const allProductsData = allProducts.map(product => product.dataValues);  
  console.table(allProductsData);  
} catch (error) {  
  console.error('Error acceder a la base de datos:', error);  
}  
finally {  
  sequelize.close();  
}
```

Siempre debemos recordar manejar cualquier error que acontezca durante este tipo de operaciones.

Y aquí, aprovechamos **finally{}** para cerrar la conexión a la base de datos, una vez que interactuamos con la tabla.

API REST con Express y MySQL

API REST con Express y MySQL

```
> npm install express
```

Instalemos la dependencia **express JS**.

```
server.js

const express = require('express');
const app = express();

const sequelize = require('./src/conexion/connection');
const Product = require('./src/modelos/product');

const port = process.env.PORT || 3000;

app.use(express.json());
```

Luego, la importamos a nuestro proyecto, declaramos la constante **app**, definimos el puerto del servidor web, y utilizamos el Middleware **express.json()** para exportar los datos a este formato de transporte.

API REST con Express y MySQL

```
server.js

app.get('/productos', async (req, res) => {
  try {
    // autenticación y obtención de datos de la tabla Products.
  } catch (error) {
    // Aquí dejamos el manejo de error.
  }
});
```

Creamos ahora un endpoint **/productos**, donde serviremos todos los productos de la tabla **Northwind.Products**. Este endpoint debe ser asíncrono y manejar la petición de datos mediante **try-catch**. El bloque **finally{}** lo eliminaremos.

API REST con Express y MySQL

```
server.js

app.get('/productos', async (req, res) => {
  try {
    await sequelize.authenticate()
    await Product.sync();
    const allProducts = await Product.findAll();
    const allProductsData = allProducts.map(product => product.dataValues);
    res.status(200).json(allProductsData);
  } catch (error) {
    // Aquí dejamos el manejo de error.
  }
});
```

En el bloque **try{}** agregamos la petición de datos a la tabla **Products**, mapeamos los mismos y los enviamos como respuesta a la petición, convertidos al formato JSON, y con el código de estado correspondiente.

API REST con Express y MySQL

```
server.js

    } catch (error) {
      res.status(500).json({ error: 'Error en el servidor',
                             description: error.message });
    }
  });
```

En el bloque **catch{}** respondemos con el código de Error 500, por si ocurre algún tipo de error al intentar acceder a la información. Describimos también el tipo de error en el servidor que aconteció.

API REST con Express y MySQL

```
connection.js

const sequelize = new Sequelize(process.env.DATABASE,
                                process.env.DBUSER,
                                process.env.PASSWORD, {
  host: process.env.HOST,
  dialect: 'mysql',
  pool: {
    max: 5, // Máximo de conexiones en el grupo
    min: 0, // Mínimo de conexiones en el grupo
    acquire: 30000, // Tiempo máximo, para liberar conexiones inactivas
    idle: 10000, // Tiempo máximo para cerrar conexiones inactivas
  },
});
```

Y volvemos al archivo **connection.js** para modificar la petición de la conexión a la base de datos. Agregamos en el mismo el parámetro **pool**, el cual nos permite manejar conexiones inactivas que debemos liberar y/o cerrar.

API REST con Express y MySQL

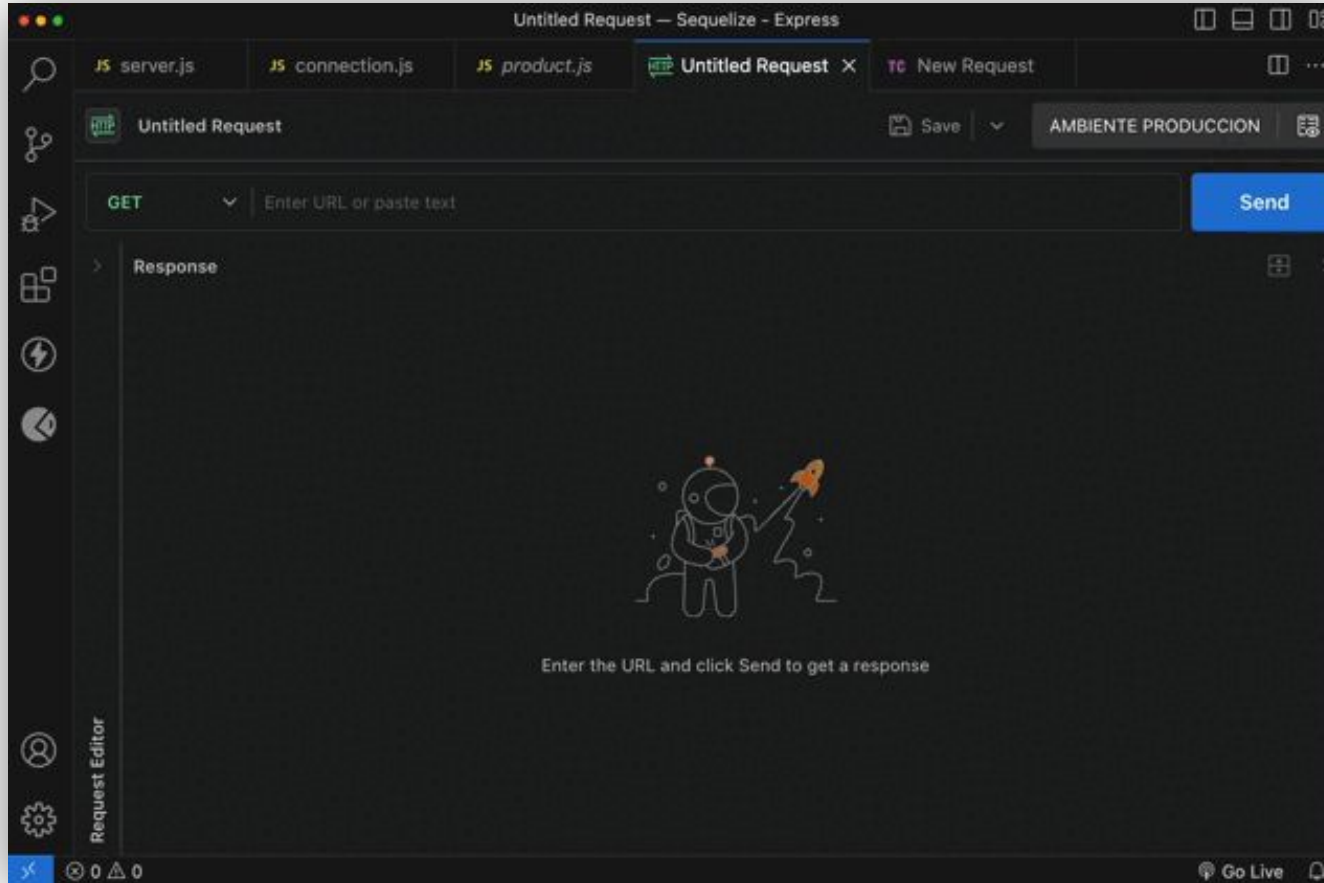
```
connection.js

const sequelize = new Sequelize(process.env.DATABASE,
                                process.env.DBUSER,
                                process.env.PASSWORD, {
  host: process.env.HOST,
  dialect: 'mysql',
  pool: {
    max: 5,
    min: 0,
    acquire: 30000,
    idle: 10000,
  },
});
```

Estos parámetros son configurados con valores numéricos, que pueden modificarse incrementando o decrementando sus valores de acuerdo a la carga de peticiones que podamos llegar a tener en el **endpoint**.

De esta forma, deslindamos a Express la responsabilidad de conectarnos y desconectarnos de la bb.dd.

API REST con Express y MySQL



Ejecutemos el endpoint y probemos las peticiones simultáneas con **POSTMAN** y/o **Thunder Client**.

API REST con Express y MySQL

Cuando trabajamos una API REST con Express y MongoDB, utilizamos el driver **mongodb**, oficial de esta empresa, para conectarnos y manipular los datos de forma fácil y práctica.

Existe también otro driver, mayormente utilizado en el mercado, que se llama **mongoose**. Este se asimila a **Sequelize** dado que también requiere que creamos Modelos de datos de cada **Colección**, para luego acceder a los mismos.

Te alentamos a investigarlo para conocer otra opción de driver que te permita utilizar MongoDB y Express.



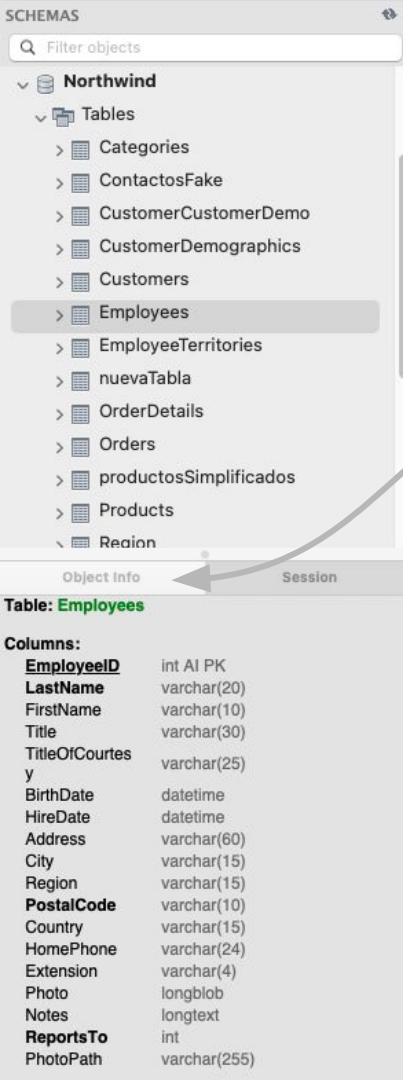
Sección Práctica

Sección Práctica

Es momento de poner en práctica el uso de Sequelize, Express y MySQL.

Te proponemos a continuación, que realices un nuevo endpoint que permita acceder a otra tabla de la base de datos Northwind, a partir de este modelo de ejemplo que construimos en la clase.





Sección Práctica

Tiempo estimado: **20 minutos**

1. Define un nuevo modelo de datos, esta vez para acceder a la información de los Empleados registrados en la tabla **Northwind.Employees**.
 - a. apóyate en **MySQL Workbench** para obtener un resumen de los tipos de datos y campos para crear el modelo en cuestión. Esto lo encuentras ejecutando la tabla **Employees**, en el panel inferior izquierdo denominado **Object Info**.
2. Una vez que tengas el modelo definido, importa el mismo al archivo **server.js**.
3. Crea a continuación un endpoint llamado **/empleados**. Luego, define el acceso a los datos de los empleados, teniendo la premisa de definir el endpoint de forma asincrónica, y retornando toda la información de estos.

Muchas gracias.



Ministerio de Economía
Argentina

Secretaría de
Economía del Conocimiento

*primero
la gente*