



**Argentina  
programa  
4.0**



Ministerio de Economía  
**Argentina**

Secretaría de  
Economía del Conocimiento

***primero  
la gente***

# Clase: Node.js y Express

## Búsqueda, filtrado y ordenamiento de datos con Sequelize

## Agenda de hoy

- A. Los métodos más importantes
  - a. Buscar por clave primaria
  - b. Buscar por parámetro estricto
  - c. Filtrar con un parámetro laxo
  - d. Otros operadores de comparación
- B. Ordenamiento de datos
  - a. (ascendente y descendente)
  - b. (simple y múltiple)
- C. Integrar Vistas SQL
- D. Información adicional en los set de datos



# 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 de acceso a la información, más comunes:



# Los métodos más importantes

Métodos Sequelize	
Método	Descripción
<code>.findAll()</code>	Busca todos los registros que cumplan con uno o más criterios
<code>.findOne()</code>	Busca un registro específico en la tabla que cumpla con ciertos criterios
<code>.findByPk()</code>	Busca un registro específico en la tabla utilizando su clave primaria (PK)

Aquí tenemos las operaciones básicas que podemos realizar con Sequelize utilizando Node.js y trabajando de forma conectada a MySQL. En nuestro encuentro anterior, ya tuvimos el honor de interactuar con el método `.findAll()` para obtener todos los registros de una tabla.

# Los métodos más importantes

**Recuperaremos nuestro proyecto anterior, para seguir trabajando sobre la tabla Products, en la cual definiremos diferentes tipos de búsqueda, aprovechando los diferentes métodos que propone Sequelize.**

**Pero antes, acomodaremos el código, para evitar ser repetitivo y no conectarnos con la base de datos en cada endpoint que ejecutemos.**



# Los métodos más importantes

En nuestro archivo de proyecto **server.js**, creamos contiguo al Middleware que invoca a **express.json()**, un nuevo Middleware.

En este agregaremos la conexión a MySQL, definiendo un método asincrónico y en sus parámetros agregamos **req**, **res** y **next**.

```
app.use(express.json());
app.use(async (req, res, next) => {
  try {
    await sequelize.authenticate();
    await Product.sync();
    await Employee.sync();
    next();
  } catch (error) {
    res.status(500).json({ error: 'Error en el servidor',
      description: error.message });
  }
});
```



## Los métodos más importantes

Con esto, nos aseguramos de que, al ejecutarse nuestra aplicación de servidor, ante una nueva petición de datos, la misma ejecute la conexión a la base de datos MySQL, y luego la función **next()**, libere la tarea para que siga ejecutándose el resto de las funciones de esta aplicación web.

```
app.use(express.json());
app.use(async (req, res, next) => {
  try {
    await sequelize.authenticate();
    await Product.sync();
    await Employee.sync();
    next();
  } catch (error) {
    res.status(500).json({ error: 'Error en el servidor',
      description: error.message });
  }
});
```

# Los métodos más importantes

Agregado este Middleware, ya podemos eliminar el método **.authenticate()** y el método **.sync()** definido en los endpoints creados anteriormente.

El endpoint **/productos** debe quedar como el siguiente ejemplo:

```
app.get('/productos', async (req, res) => {  
  try {  
    const allProducts = await Product.findAll();  
  
    allProducts.length !== 0 ? res.status(200).json(allProducts)  
      : res.status(404).json({ error: "No se encontraron productos." });  
  
  } catch (error) {  
    res.status(500).json({ error: 'Error en el servidor',  
      description: error.message });  
  }  
});
```

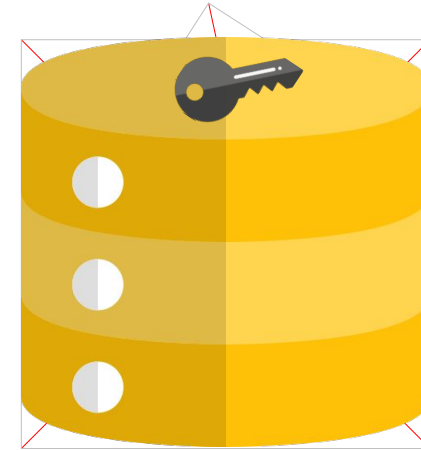


# Búsqueda por clave primaria

# Búsqueda por clave primaria

El método **.findByPk()** es el método que debemos utilizar para buscar algún registro específico por su clave primaria.

Para ello definimos un endpoint que reciba un parámetro, tal como hicimos oportunamente en el CRUD con MongoDB, desestructuramos el mismo desde `req.params`, y luego lo enviamos como parámetro al método **.findByPk()**.



# Búsqueda por clave primaria

Un punto clave que debemos tener en cuenta es que, el parámetro del endpoint, conviene definirlo con el mismo nombre de la clave primaria.

De esta forma, cuando le pasamos el parámetro al método **findByPk()**, nos ahorramos tener que repetir el nombre del campo y el nombre de la variable o constante desestructurada.

```
app.get('/productos/:productID', async (req, res) => {  
  try {  
    const { productID } = req.params;  
    const product = await Product.findByPk(productID);  
    ...  
  }  
  ...  
})
```

# Búsqueda por clave primaria

Si se encuentra un registro coincidente con dicha clave primaria, se almacenará en la constante definida.

Luego controlamos con un condicional `if - else`, si la constante posee un valor almacenado o no, y retornamos el código de estado más el mensaje apropiado, según el resultado de la condición evaluada.

```
app.get('/productos/:productID', async (req, res) => {  
  try {  
    const { productID } = req.params;  
    const product = await Product.findByPk(productID);  
  
    !product ? res.status(404).json({ error: "Producto no encontrado." })  
      : res.status(200).json(product);  
  } catch (error) {  
    res.status(500).json({ error: 'Error en el servidor',  
      description: error.message });  
  }  
});
```

**Búsqueda por parámetro estricto**

# Búsqueda por parámetro estricto

Cuando realizamos una búsqueda por un parámetro estricto, por ejemplo **Employees.FirstName = "Andrew"**, debemos aplicar utilizar el método **findOne()**, el cual retornará el primer valor coincidente.

Si existe más de uno, solo retornará el primero de los valores, tal como lo hace el método de arrays **.find()**.

```
app.get('/productos/nombre/:productName', async (req, res) => {  
  try {  
    const { productName } = req.params;  
    const product = await Product.findOne({ where: { productName } });  
    ...  
  }  
  ...  
});
```



# Búsqueda por parámetro estricto

Junto al método **findOne()**, debemos aplicar el uso de WHERE, tal como lo hacemos en MySQL.

A diferencia de la base de datos en cuestión, en Sequelize la cláusula **WHERE** debe estar constituida por un objeto literal (*encerrada entre llaves de bloque*).

De igual forma, el parámetro a buscar, también debe encerrarse entre llaves del bloque, contiguo a la propiedad WHERE.

```
app.get('/productos/nombre/:productName', async (req, res) => {  
  try {  
    const { productName } = req.params;  
    const product = await Product.findOne({ where: { productName } });  
    ...  
  }  
  ...  
});
```

# Búsqueda por parámetro estricto

```
const product = await Product.findOne({ where: { productName } });
```

También podemos sacar partido en la búsqueda, utilizando como variable que almacena el parámetro a buscar, el mismo nombre del campo de la tabla, así nos ahorramos tener que definir ambos datos por separado.

# Búsqueda por parámetro estricto

En el caso de querer realizar una búsqueda estricta para que, por ejemplo, obtengamos de una tabla de datos múltiples registros que cumplan con una condición (*Todos los productos de una categoría*), allí recurrimos al método **findAll()** combinando con la cláusula WHERE y el parámetro a filtrar.

```
app.get('/productos/categoria/:categoryID', async (req, res) => {  
  try {  
    const { categoryID } = req.params;  
    const products = await Product.findAll({ where: { categoryID } });  
  
    !products ? res.status(404).json({ error: 'Producto no encontrado' })  
      : res.status(200).json(products);  
  } catch (error) {  
    res.status(500).json({ error: 'Error en el servidor',  
      description: error.message });  
  }  
});
```

**Filtrar con un parámetro laxo**

# Filtrar con un parámetro laxo

Reflotando un poco lo que vimos oportunamente a través de los métodos JS de orden superior, y durante la cursada de MySQL con la cláusula **LIKE**, también podemos aplicar este tipo de búsquedas laxas utilizando Sequelize.

El método que utilizaremos será **findAll()**, junto al objeto **WHERE**, pero debemos sumar el objeto **Op**, integrado a Sequelize.

```
app.get('/productos/buscar/:query', async (req, res) => {  
    // búsqueda de uno o más resultados con un valor parcial  
})
```

# Filtrar con un parámetro laxo

El primer paso, será obtener el objeto **Op**, integrado a la librería **Sequelize**, desestructurando el mismo mediante el uso de la función **require()**.

Esto lo definiremos en el encabezado de nuestra aplicación **Node.js** (*archivo server.js*).

Con esto ya resuelto, podremos aplicarlo dentro del método **findAll()**. Veamos cómo, a continuación:

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

const sequelize = require('./src/conexion/connection');
const Product = require('./src/modelos/product');
const Employee = require('./src/modelos/employee');
const { Op } = require('sequelize');
```



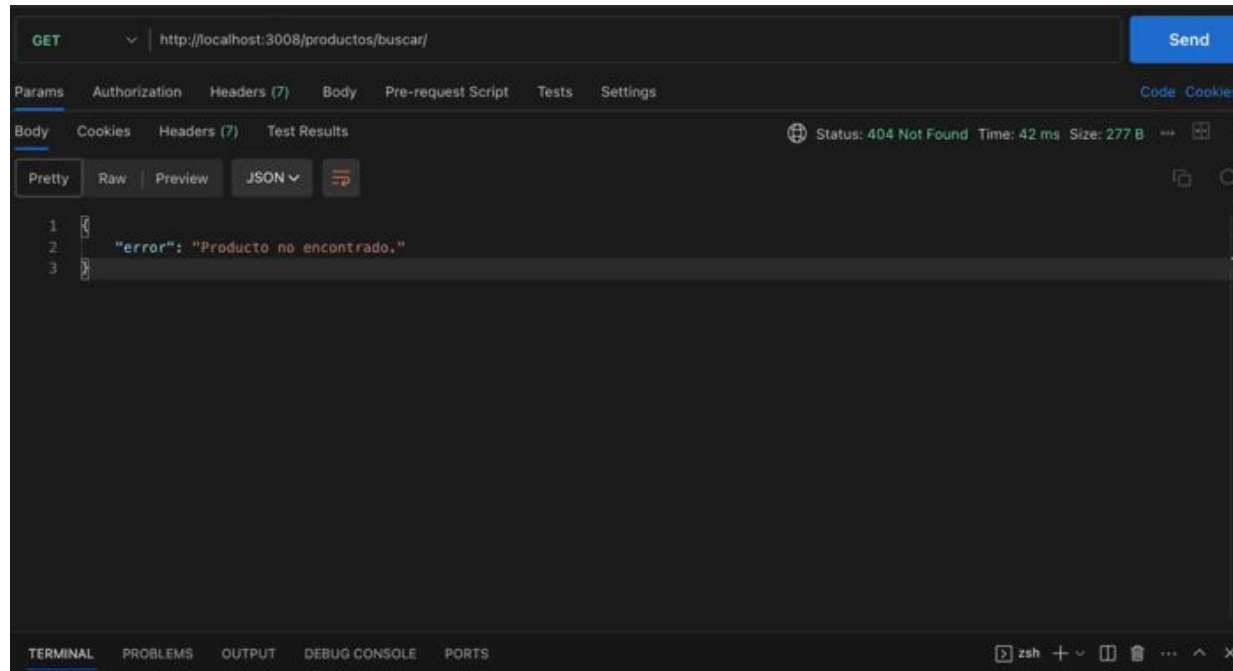
# Filtrar con un parámetro laxo

En el endpoint en cuestión, declaramos el objeto literal que oficia de cláusula **WHERE**, y dentro de este, ahora sí, debemos definir el nombre del campo donde deseamos aplicar el filtro.

Luego definimos el objeto **Op** y su propiedad **like**, y utilizando Template String + Template Literals, agregamos el parámetro de consulta.

```
app.get('/productos/buscar/:query', async (req, res) => {  
  try {  
    const { query } = req.params;  
    const products = await Product.findAll({  
      where: { productName: {  
        [Op.like]: `%${ query }%`,  
      },  
    },  
  });  
  ...  
}
```

# Filtrar con un parámetro laxo



En este ejemplo podemos apreciar el comportamiento de una búsqueda utilizando el operador **LIKE**.

Como vemos, la estructura de este nos permite buscar uno o más resultados, indistintamente de mayúsculas y minúsculas, tal como funciona LIKE en el motor MySQL.



# Otros operadores de comparación

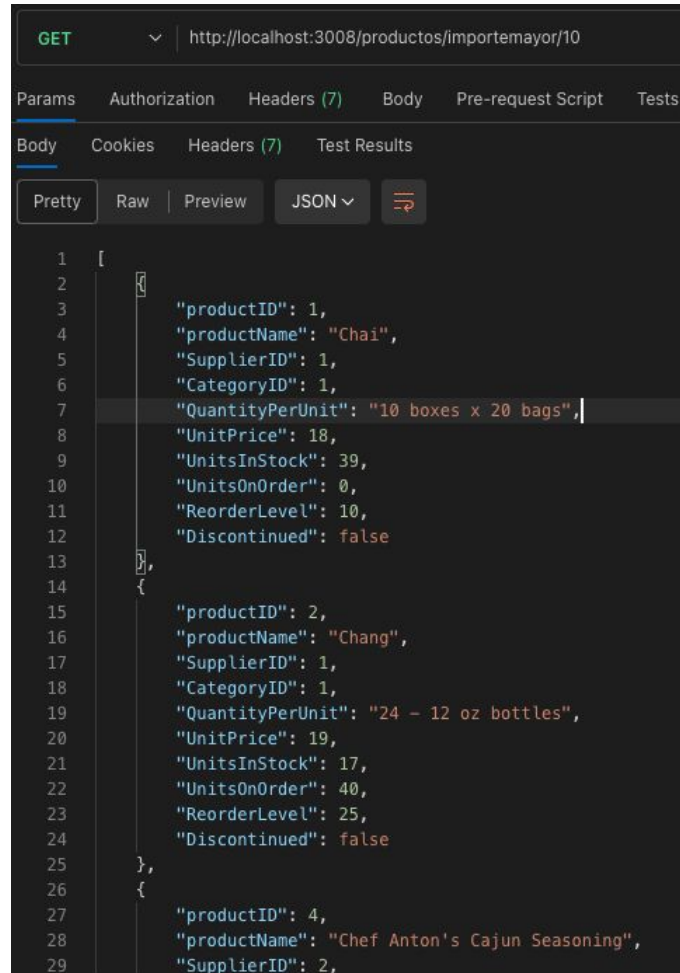
## Otros operadores de comparación

El mismo objeto **Op**, nos da acceso al resto de los operadores de comparación que podemos aplicar en una consulta con MySQL.

Aquí vemos un ejemplo donde realizamos una consulta para obtener todos los productos que tengan un importe mayor a un valor determinado. Por ejemplo: **12**.

```
app.get('/productos/importemayor/:unitPrice', async (req, res) => {  
  try {  
    const { unitPrice } = req.params;  
    const products = await Product.findAll({  
      where: { UnitPrice: {  
        [Op.gt]: unitPrice,  
      },  
    });  
    ...  
  });  
});
```

# Otros operadores de comparación



```
GET http://localhost:3008/productos/importemayor/10

Params Authorization Headers (7) Body Pre-request Script Tests
Body Cookies Headers (7) Test Results
Pretty Raw Preview JSON

1 [
2   {
3     "productID": 1,
4     "productName": "Chai",
5     "supplierID": 1,
6     "categoryID": 1,
7     "quantityPerUnit": "10 boxes x 20 bags",
8     "unitPrice": 18,
9     "unitsInStock": 39,
10    "unitsOnOrder": 0,
11    "reorderLevel": 10,
12    "discontinued": false
13  },
14  {
15    "productID": 2,
16    "productName": "Chang",
17    "supplierID": 1,
18    "categoryID": 1,
19    "quantityPerUnit": "24 - 12 oz bottles",
20    "unitPrice": 19,
21    "unitsInStock": 17,
22    "unitsOnOrder": 40,
23    "reorderLevel": 25,
24    "discontinued": false
25  },
26  {
27    "productID": 4,
28    "productName": "Chef Anton's Cajun Seasoning",
29    "supplierID": 2,
```

Aquí tenemos el resultado que otorga el endpoint, utilizando el parámetro de búsqueda **mayor a**.

Como podemos ver, el nombre de la propiedad del objeto **Op** que responde a este operador de comparación, es muy similar a lo que vimos oportunamente con los operadores de comparación de MongoDB.



# Otros operadores de comparación

Aquí tenemos una lista de casi todos los operadores de comparación alternativos.

Todos estos operadores conforman una propiedad dentro del objeto **Op**.

Como vimos en los primeros ejemplos, cuando la búsqueda es igual a determinado parámetro, podemos obviar el uso del operador **.eq**.

Operadores de comparación integrados al objeto Op	
Operador	Descripción
.eq	Igual a
.ne	No es igual a
.gt	Mayor a
.gte	Mayor o igual a
.lt	Menor que
.lte	Menor o igual que
.between	Entre
.notBetween	No esté entre
.in	Incluido en
.notIn	No incluido en
.like	Como
.notLike	No sea como



## Otros operadores de comparación

```
try {  
  const products = await Product.findAll({  
    where: {  
      UnitPrice: {  
        [Op.between]: [10, 20],  
      },  
    },  
  });  
}
```

Aquí un ejemplo de cómo implementar el uso del operador **BETWEEN**. Como podemos apreciar, no se utiliza el operador lógico **AND** como conector de los valores de rango.

```
try {  
  const categorias = [1, 4, 5];  
  
  const products = await Product.findAll({  
    where: {  
      CategoryID: {  
        [Op.in]: categorias,  
      },  
    },  
  });  
}
```

Aquí otro ejemplo, pero implementando el uso del operador **IN**, para filtrar las categorías de productos **1, 4 y 5**.

# Ordenamiento de datos

# Ordenamiento de datos

También tenemos la posibilidad de ordenar datos de la consulta SQL ejecutada.

Para ello, utilizamos el objeto literal con la propiedad **order**, y encerramos entre corchetes el nombre del campo y la cláusula correspondiente al tipo de ordenamiento a aplicar.

```
const products = await Product.findAll({  
  order: [['productName', 'ASC']]  
});
```

# Ordenamiento de datos

Si debemos ordenar por múltiples campos y/o múltiples tipos de ordenamiento, el mismo objeto nos permite definir por separado, cada campo y el tipo de orden a aplicar.

Como vemos en el ejemplo, cada campo va definido dentro de un bloque de corchetes.

```
const products = await Product.findAll({  
  order: [  
    ['category', 'ASC'],  
    ['productName', 'DESC']  
  ]  
});
```



# Ordenamiento de datos

Y de esta forma, podemos combinar una consulta de selección, definiendo un parámetro por el cual debemos filtrar, y luego ordenando la consulta resultante por un campo específico.

En este ejemplo, traemos todos los productos de la **Categoría** no. **5**, y ordenamos la consulta por el campo **Precio Unitario** de forma descendente.

```
const categoryId = 5;

const products = await Product.findAll({
  where: {
    CategoryID: categoryId,
  },
  order: [['UnitPrice', 'DESC']],
});
```



# Integrar Vistas SQL

# Integrar Vistas SQL

Si necesitamos estructurar los datos de una tabla SQL que sirvan información precisa a través de un endpoint, podemos recurrir a la construcción de una Vista SQL.

De esta forma lograremos mostrar la información que solo queremos que viaje ante cada petición.

Aquí un ejemplo de cómo podemos visualizar determinados campos de la tabla **Products**, estableciendo una relación con la tabla **Categories**, para poder mostrar los productos y la descripción de sus categorías.

```
SELECT P.ProductID,  
       P.ProductName,  
       P.CategoryID,  
       C.CategoryName,  
       P.QuantityPerUnit,  
       P.UnitPrice,  
       P.UnitsInStock  
FROM Northwind.Products P  
RIGHT JOIN Northwind.Categories C  
ON C.CategoryID = P.CategoryID  
ORDER BY P.productID;
```

# Integrar Vistas SQL

Result Grid							
Filter Rows: <input type="text" value="Search"/>							Export:
ProductID	ProductName	CategoryID	CategoryName	QuantityPerUnit	UnitPrice	UnitsInStock	
1	Chai	1	Beverages	10 boxes x 20 bags	18	39	
2	Chang	1	Beverages	24 - 12 oz bottles	19	17	
3	Aniseed Syrup	2	Condiments	12 - 550 ml bottles	10	13	
4	Chef Anton's Cajun Seasoning	2	Condiments	48 - 6 oz jars	22	53	
5	Chef Anton's Gumbo Mix	2	Condiments	36 boxes	21.35	0	
6	Grandma's Boysenberry Spread	2	Condiments	12 - 8 oz jars	25	120	
7	Uncle Bob's Organic Dried Pears	7	Produce	12 - 1 lb pkgs.	30	15	
8	Northwoods Cranberry Sauce	2	Condiments	12 - 12 oz jars	40	6	
9	Mishi Kobe Niku	6	Meat/Poultry	18 - 500 g pkgs.	97	29	
10	Ikura	8	Seafood	12 - 200 ml jars	31	31	
11	Queso Cabrales	4	Dairy Products	1 kg pkg.	21	22	
12	Queso Manchego La Pastora	4	Dairy Products	10 - 500 g pkgs.	38	86	
13	Konbu	8	Seafood	2 kg box	6	24	
14	Tofu	7	Produce	40 - 100 g pkgs.	23.25	35	

Esta consulta SQL que utiliza **JOIN**, se puede convertir en una Vista SQL.

Luego, dicha vista, podrá ser utilizada específicamente por un endpoint, tal como si estuviésemos utilizando una tabla SQL.

# Integrar Vistas SQL

```
1 • CREATE
2     ALGORITHM = UNDEFINED
3     DEFINER = `root`@`localhost`
4     SQL SECURITY DEFINER
5     VIEW `productsandcategories` AS
6     SELECT
7         `P`.`ProductID` AS `ProductID`,
8         `P`.`ProductName` AS `ProductName`,
9         `P`.`CategoryID` AS `CategoryID`,
10        `C`.`CategoryName` AS `CategoryName`,
11        `P`.`QuantityPerUnit` AS `QuantityPerUnit`,
12        `P`.`UnitPrice` AS `UnitPrice`,
13        `P`.`UnitsInStock` AS `UnitsInStock`
14    FROM
15        (`categories` `C`
16     LEFT JOIN `products` `P` ON ((`C`.`CategoryID` = `P`.`CategoryID`)))
17    ORDER BY `P`.`ProductID`
```

Por ejemplo, creamos una vista SQL con el nombre **ProductsAndCategories**.

Guardamos la misma, y ahora nos quedará pendiente la construcción del Modelo de datos en Node.js, de acuerdo a los campos seleccionados dentro de esta vista.

# Integrar Vistas SQL

```
const ProductCategoryView = sequelize.define('ProductCategoryView', {
  ProductID: {
    type: DataTypes.INTEGER,
    primaryKey: true,
  },
  ProductName: {
    type: DataTypes.STRING,
  },
  CategoryID: {
    type: DataTypes.INTEGER,
  },
  CategoryName: {
    type: DataTypes.STRING,
  },
  QuantityPerUnit: {
    type: DataTypes.STRING,
  },
  UnitPrice: {
    type: DataTypes.DOUBLE,
  },
  UnitsInStock: {
    type: DataTypes.SMALLINT,
  },
}, {
  tableName: 'productsandcategories',
  timestamps: false,
});

module.exports = ProductCategoryView;
```

Definimos los tipos de datos correspondientes para cada uno de los campos integrados en esta vista SQL.

El nombre del Modelo de datos lo definimos en inglés, respetando la estructura utilizada para los anteriores Modelos.

También, dentro de su nombre aclaramos que los datos provienen de una Vista SQL. **Este último punto es clave para que cualquier otro desarrollador que trabaje sobre este endpoint, sepa que es una Vista, y que no podrá desarrollarse un endpoint utilizando POST o PUT.**



# Integrar Vistas SQL

```
const ProductCategoryView = require('./src/modelos/productsandcategories');

app.get('/productosycategorias', async (req, res) => {
  try {
    const allProducts = await ProductCategoryView.findAll();
    allProducts.length !== 0 ? res.status(200).json(allProducts)
      : res.status(404).json({ error: "Sin productos." });

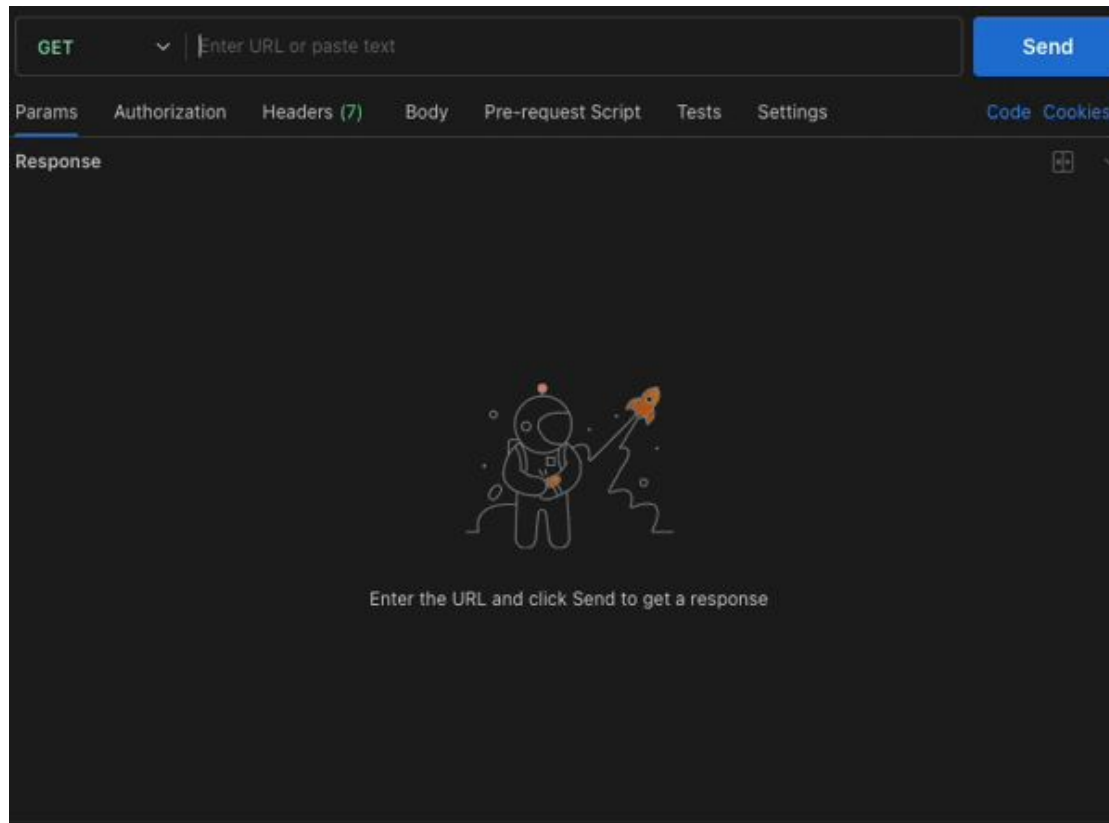
    } catch (error) {
      res.status(500).json({ error: 'Error en el servidor',
        description: error.message });
    }
  });
```

Luego nos resta declarar el Modelo de datos para la vista creada, al inicio de nuestro archivo **server.js**.

Por último, definimos el endpoint que nos permitirá obtener todos los datos de la Vista, tal como si se tratara de una tabla SQL más.



# Integrar Vistas SQL



Nos resta probar el funcionamiento del endpoint, utilizando POSTMAN o Thunder Client.

Si todo fue bien, ya tenemos el mismo funcionando al 100%. 💪



# **Información adicional en los set de datos**

# Información adicional en los set de datos

En determinadas situaciones, es probable que se requiera enviar información adicional dentro de la respuesta de un endpoint, que aporte valor agregado en los datos que comparte una aplicación backend con una aplicación frontend.

Por ejemplo, si el endpoint funciona con información paginada, debemos indicar en el response enviado en qué número de página se encuentran los datos que se están visualizando.



# Información adicional en los set de datos

```
GET https://api.randomuser.me/?results=60
Query Headers 2 Auth Body Tests Pre Run
Status: 200 OK Size: 63.25 KB Time: 307 ms
3352 {
3353   "name": "NINO",
3354   "value": "HS 55 90 56 N"
3355 },
3356   "picture": {
3357     "large": "https://randomuser.me/api/portraits/men/87.jpg",
3358     "medium": "https://randomuser.me/api/portraits/med/men/87.jpg",
3359     "thumbnail": "https://randomuser.me/api/portraits/thumb/men/87.jpg"
3360   },
3361   "nat": "GB"
3362 }
3363 },
3364   "info": {
3365     "seed": "3fec12818cf19ae0",
3366     "results": 60,
3367     "page": 1,
3368     "version": "1.4"
3369   }
3370 }
```

También se puede informar qué versión de endpoint se está utilizando, la fecha y hora del set de datos entregados, el total de registros enviados, entre otros tantos datos.

Todo esto, se puede construir y acoplar a los datos obtenidos a partir de Sequelize, previo a enviar la información al cliente que la solicitó.

# Información adicional en los set de datos

Para lograrlo, simplemente armamos un objeto literal al cual le agregamos la propiedad **results** y, como valor, le definimos una estructura de array más **Spread Operator** para que disemine todos los objetos JSON correspondientes a la consulta SQL realizada.

```
const response = {  
  results: [...allProducts],  
  info: {  
    dateOfQuery: new Date(),  
    totalRecords: allProducts.length || 0,  
    database: sequelize.getDatabaseName()  
  }  
}
```

# Información adicional en los set de datos

Luego nos apoyamos en herramientas JS como ser la clase **Date()** para obtener la fecha y hora, la constante **allProducts**, conformada por un array con todos los registros desde donde podemos leer el total de objetos, y así con el resto de los datos que necesitemos enviar.

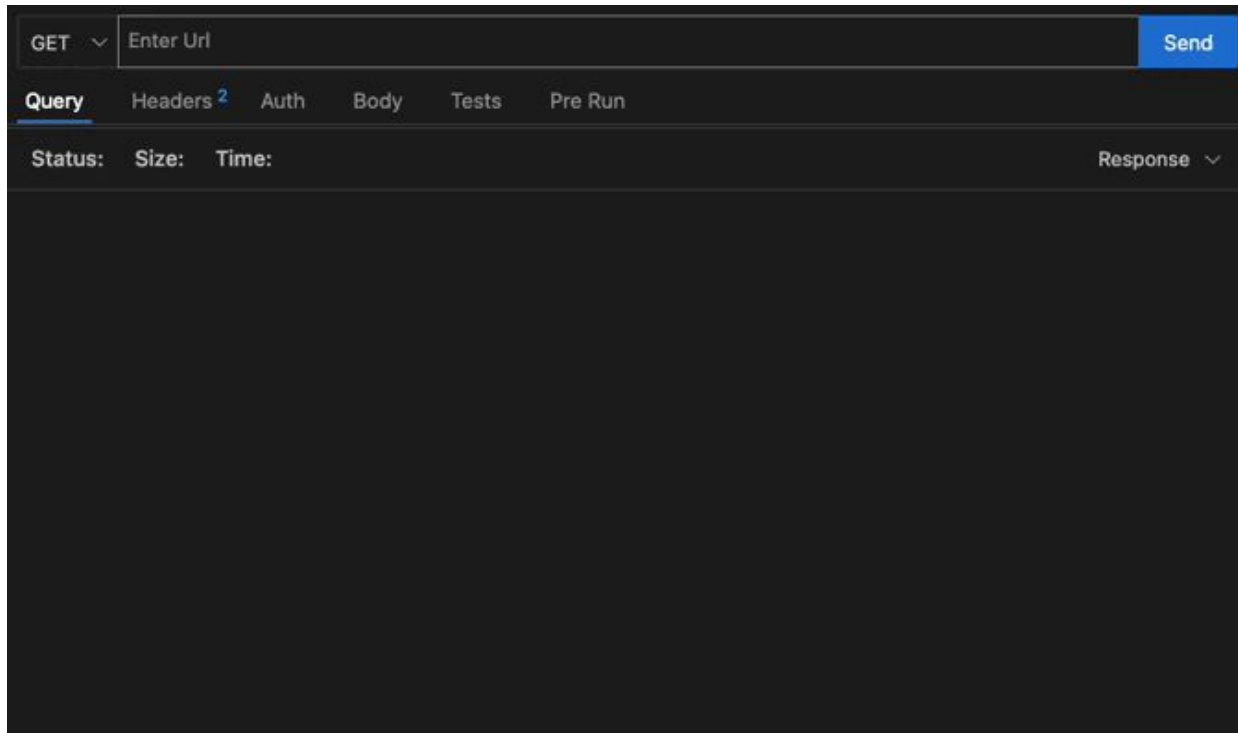
```
const response = {  
  results: [...allProducts],  
  info: {  
    dateOfQuery: new Date(),  
    totalRecords: allProducts.length || 0,  
    database: sequelize.getDatabaseName()  
  }  
}
```

# Información adicional en los set de datos

Finalmente, lo que enviamos como respuesta de la petición será este nuevo objeto, y por supuesto que deberemos aclarar todo esto en la documentación pertinente de nuestro proyecto.

```
app.get('/productosycategorias', async (req, res) => {  
  try {  
    const allProducts = await ProductCategoryView.findAll();  
  
    const response = {  
      results: [...allProducts],  
      info: {  
        dateOfQuery: new Date(),  
        totalRecords: allProducts.length || 0,  
        database: sequelize.getDatabaseName()  
      }  
    }  
  
    allProducts.length !== 0 ? res.status(200).json(response)  
      : res.status(404).json({ error: "No se encontraron productos." });  
  }  
  ...  
}
```

# Información adicional en los set de datos



The image shows a screenshot of a REST client application. At the top, there is a dropdown menu set to 'GET' and a text input field labeled 'Enter Url'. To the right of the input field is a blue 'Send' button. Below this, there is a horizontal tab bar with five tabs: 'Query' (which is selected and underlined), 'Headers' (with a small blue '2' next to it), 'Auth', 'Body', and 'Pre Run'. Below the tabs, there is a header row with 'Status:', 'Size:', 'Time:', and 'Response' (with a dropdown arrow). The main area below this header is currently empty.

Ya podemos validar en un cliente HTTP, que todo funcione tal como lo esperamos y que nuestro endpoint entregue toda la información que sea necesaria aportar, junto a cada petición HTTP que este reciba.

# Información adicional en los set de datos

Como podemos ver, Sequelize es una herramienta muy flexible y completa la cual nos permite lograr prácticamente todo lo mismo que aprendimos trabajando con MySQL, pero combinando objetos, arrays, y frameworks JS.

Sequelize nos da un montón de herramientas adicionales más, para poder trabajar en profundidad con los diferentes objetos que conforman una base de datos.

Te invitamos a seguir explorando esta herramienta, en pos de conocer más allá de estas clases, cuántas otras fabulosas opciones nos pone a disposición.





# Sección Práctica

# Sección Práctica

Pongamos a trabajar todo lo nuevo que aprendimos en este encuentro.

El objetivo de esta sección práctica es construir una Vista SQL y un nuevo Modelo de datos Sequelize, basado en dicha vista.

Veamos la consigna, a continuación:



# Sección Práctica

Tiempo estimado: 20 minutos

1. Crea una Vista SQL combinando las tablas OrderDetails, Products.
  - a. la Vista deberá tener los campos OrderID, ProductID, ProductName, UnitPrice, Quantity de la tabla OrderDetails
  - b. deberás combinar ProductID con la tabla Products para traer el nombre del producto
  - c. la Vista deberá llamarse VistaOrderDetails
  
2. Define un nuevo Modelo de datos para la Vista anterior, y crea un único endpoint al cual le puedas peticionar todas las órdenes de esta Vista SQL, o una sola orden específica
  - a. si el endpoint no recibe un número de orden, retornará todas Órdenes de compra de la Vista SQL
  - b. si recibe un número de orden, deberá aplicar un filtro a la Vista SQL para entregar solamente esa O.C.
  - c. Los registros deberán estar ordenados por el campo ProductID de forma descendente

# Muchas gracias.



Ministerio de Economía  
**Argentina**

Secretaría de  
Economía del Conocimiento

*primero  
la gente*