# Argentina programa 4.0

Ministerio de Economía
Argentina

Secretaría de
Economía del Conocimiento

primero la gente

UNTREF
UNIVERSIDAD NACIONAL
DE TRES DE FEBRERO

# Clase: Node.js y Express

## Operaciones CRUD con Sequelize

Ministerio de
Desarrollo Productivo
Argentina

# Agenda de hoy

A. Operaciones CRUD
   a. Agregar un registro
   b. Modificar un registro
   c. Eliminar un registro.

B. Otros posibles errores

C. Ejercitación

# Operaciones CRUD

# Operaciones CRUD

Como vimos oportunamente con MongoDB, las operaciones CRUD son un punto importante dentro del desarrollo de software, dado que manejan el alta, baja y modificación de datos en una base de datos.

Sequelize

# Operaciones CRUD

Sequelize integra una serie de métodos que podemos invocar para realizar las operaciones CRUD sobre una base de datos, de una forma fácil y práctica.

Veamos a continuación cuáles son dichos métodos.

# Operaciones CRUD

| Métodos CRUD Sequelize | |
|---|---|
| **Método** | **Descripción** |
| `.create()` | Crea un nuevo registro con los datos enviados por POST. |
| `.update()` | Actualiza un registro existente con la información enviada por PUT. |
| `.destroy()` | Elimina un registro existente con el parámetro enviado por DELETE. |

Como vemos, podemos fácilmente integrar operaciones CRUD en Sequelize. Estas se deben combinar con el uso de operaciones de búsqueda y filtrado, para lograr así llegar a los datos primero, y luego eliminarlos.

# Agregar un registro

# Agregar un registro

El método .create() nos permite crear un nuevo registro.

El mismo está ubicado dentro del objeto - Modelo que representa a la tabla donde deseamos insertar dicho registro.

```
const table = await Table.create({
                    nombreCampo,
                    categoryaCampo,
                    precioCampo,
                    unidadesCampo,
                    discontinuadoCampo
              });
```

# Agregar un registro

```
app.post('/table/', async (req, res) => {

const { nombreCampo, categoryCampo,
      precioCampo, unidadesCampo,
      discontinuadoCampo} = req.body;
      ...
```

Para implementarlo, debemos cumplir con todos los pasos previos, vistos en las clases anteriores:

- definir el modelo de datos de la tabla en Sequelize
- tener un usuario con permisos de inserción
- crear un enpoint utilizando el método POST

# Agregar un registro

```
app.post('/table/', async (req, res) => {

const { nombreCampo, categoryCampo,
      precioCampo, unidadesCampo,
      discontinuadoCampo} = req.body;
      ...
```

Como siempre, el endpoint debe utilizar el método POST, y nosotros debemos recibir los datos de éste y desestructurar los mismos.

Debemos también tener presente que, los nombres de las propiedades enviadas por POST, cumplimenten con el nombre de los campos de la tabla donde insertaremos el nuevo registro.

# Agregar un registro

```
app.post('/table/', async (req, res) => {

        const { nombreCampo, categoryCampo,
              precioCampo, unidadesCampo,
              discontinuadoCampo} = req.body;

              const table = await Table.create({
                  nombreCampo,
                  categoryCampo,
                  precioCampo,
                  unidadesCampo,
                  discontinuadoCampo
              });

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

Con todo esto resuelto, invocamos al método **.create()** utilizando el modelo de la tabla donde deseamos insertar el registro.

Este método realizará la inserción y retornará un objeto con la información del registro insertado. Por ello, debemos capturar el proceso del método **.create()** en una constante.

Finalmente retornamos el código de estado 201, y el registro insertado en formato JSON. El mismo incluirá en este caso, el ID que le haya generado la tabla SQL.



Clasificación YPF: No Confidencial

# Agregar un registro

```
app.post('/productos/', async (req, res) => {
      try {
            const { productName, CategoryID,
                  SupplierID, QuantityPerUnit,
                  UnitPrice, UnitsInStock, ReorderLevel,
                  UnitsOnOrder, Discontinued
            } = req.body;

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

            res.status(201).json(product);
      } catch (error) {
            res.status(500).json({
                  error: 'Error en el servidor',
                  description: error.message
            });
      }
})
```

Aquí vemos un ejemplo completo de cómo insertar un nuevo registro en la tabla **Northwind.Products.**

Como siempre, debemos realizar todo este tipo de operaciones, de forma asincrónica, y encerrando los módulos de cada proceso en los bloques try - catch, para poder capturar y responder ante cualquier error inesperado en este proceso.



Clasificación YPF: No Confidencial

# Modificar un registro

# Modificar un registro

```
app.put('/productos/:ProductID', async (req, res) => {
        const { ProductID } = req.params;

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

    }
});
```

En cuanto a modificar un registro existente, debemos tener una serie de pasos en cuenta, para que el proceso se pueda llevar a cabo correctamente y a su vez informar también, correctamente, al usuario en el caso que no se pueda haber ejecutado esta operación.

# Modificar un registro

```
app.put('/productos/:ProductID', async (req, res) => {
        const { ProductID } = req.params;

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

    }
});
```

El método Express.put() debe ser el que debemos utilizar, recibiendo por parámetro el Identificador del registro a modificar.

Lo primero que hacemos es desestructurar y validar que sea un ID válido (de acuerdo a su tipo de dato). Este punto es clave porque será el valor que utilicemos en la condición WHERE para aplicar UPDATE sobre el registro.

# Modificar un registro

```
app.put('/productos/:ProductID', async (req, res) => {
        const { ProductID } = req.params;

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

    }
});
```

Luego, desestructuramos el resto de la información que llega en el cuerpo de la petición PUT, obteniendo así cada campo correspondiente al registro que vamos a modificar.

Con este punto realizado, ya estamos listos para modificar el registro con el método **.update()**.

# Modificar un registro

```
await Product.update(
      {
            ProductName,
            SupplierID,
            CategoryID,
            QuantityPerUnit,
            UnitPrice,
            UnitsInStock,
            UnitsOnOrder,
            ReorderLevel,
            Discontinued,
      },
      {
            where: { ProductID },
      }
);
```

🟢 Invocamos al método **.update()** sobre el Modelo de datos de la tabla donde insertaremos el registro. Le enviamos como un objeto el parámetro de todos los campos que modificaremos.

🟡 Debajo, en otro objeto, le indicamos la cláusula **WHERE** y el **ID** o condición del registro que deseamos modificar.



Clasificación YPF: No Confidencial

# Modificar un registro

```
const [productToUpdate] = await Product.update(
        {
                ProductName,
                SupplierID,
                CategoryID,
                QuantityPerUnit,
                UnitPrice,
                UnitsInStock,
                UnitsOnOrder,
                ReorderLevel,
                Discontinued,
        },
        {
                where: { ProductID },
        }
);
```

🟢 El método **.update()** es un proceso asincrónico, al igual que los otros, por lo tanto debemos declarar la cláusula **await.**

🟡 Y como retorna un resultado, al igual que **.create(),** debemos declarar una constante para capturar el o los registros que retorne.



Clasificación YPF: No Confidencial

# Modificar un registro

```
const [productToUpdate] = await Product.update(
        {
                ProductName,
                SupplierID,
                CategoryID,
                QuantityPerUnit,
                UnitPrice,
                UnitsInStock,
                UnitsOnOrder,
                ReorderLevel,
                Discontinued,
        },
        {
                where: { ProductID },
        }
);
```

🟡 Como vemos, la constante está definida como un array, y no como una constante simple.

Esto se debe a que, la cláusula **WHERE** puede llegar a ser aplicada sobre una condición que haga que la actualización de datos impacte en más de un registro, por lo cual lo que retorne este método podría ser un conjunto de registros, y no uno solo.



Clasificación YPF: No Confidencial

Clasificación YPF: No Confidencial

# Modificar un registro

Finalmente, validamos si la constante **productToUpdate** tiene un valor igual a 0, es porque no se modificó ningún registro. En ese caso, retornamos el código de estado **404.**

Sino, buscamos el registro con el método **.findByPk()** y lo retornamos con el código de estado **200.**

```
if (productToUpdate === 0) {
    return res.status(404).json({ error: 'Producto no encontrado.' });
}

const updatedProduct = await Product.findByPk(ProductID);
res.status(200).json(updatedProduct);
```



Clasificación YPF: No Confidencial

# Modificar un registro

**Tal como vimos oportunamente en el aprendizaje de MySQL, los métodos update y delete requieren mucha atención y asegurarse siempre si existe una condición a aplicar sobre la operación, de que la misma sea válida.**

**Sino, podremos terminar modificando o eliminando accidentalmente un conjunto de registros, cuando este no era el objetivo principal.**



Clasificación YPF: No Confidencial

# Modificar un registro

```
app.put('/productos/:ProductID', async (req, res) => {
      try {
            const { ProductID } = req.params;

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

            const [productToUpdate] = await Product.update(
                  {
                      ProductName, SupplierID, CategoryID,
                      QuantityPerUnit, UnitPrice, UnitsInStock,
                      UnitsOnOrder, ReorderLevel, Discontinued,
                  },
                  {
                      where: { ProductID },
                  }
            );
            if (productToUpdate === 0) {
                return res.status(404).json({ error: 'Producto no encontrado.' });
            }

            const updatedProduct = await Product.findByPk(ProductID);
            res.status(200).json(updatedProduct);

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

El código completo del endpoint de modificación de un producto sobre la tabla Northwind.Products.

Siempre realizamos estas operaciones sobre un bloque try - catch, y capturamos cualquier error que esté más allá de las operaciones que realicemos con Sequelize.



Clasificación YPF: No Confidencial

# Eliminar un registro

# Eliminar un registro

Vamos con el último de los métodos correspondiente a las operaciones CRUD.

Para eliminar un registro de MySQL, primero desestructuramos el mismo, quien debe llegar como parámetro a este endpoint.

Una vez que lo tenemos, buscamos dicho registro sobre el Modelo, utilizando el método .findByPk(). Capturamos el resultado de la búsqueda en una constante.

```
app.delete('/productos/:ProductID', async (req, res) => {
      const { ProductID } = req.params;
      const productToDelete = await Product.findByPk(ProductID);
      ...

});
```

# Eliminar un registro

Validamos el valor asignado a nuestra constante, luego de buscar el registro. 🟡
Si la misma no posee un dato válido, retornaremos un código de estado 404 con el correspondiente mensaje de que el registro no fue encontrado.

🟢 Si la constante tiene el valor esperado, ejecutamos el método .destroy() para eliminar el registro en cuestión.

```
const { ProductID } = req.params;
const productToDelete = await Product.findByPk(ProductID);
if (!productToDelete) {
    return res.status(404).json({ error: 'Producto no encontrado.' });
}

await productToDelete.destroy(); // Elimina el registro
```

# Eliminar un registro

🟢 El código de estado a enviar, como respuesta, es el 204: “Sin contenido”.

En algunos casos se suele enviar el código de estado 404, pero el mismo corresponde a un error del cliente, por lo tanto es más conveniente enviar el código 204, el cual se encuentra dentro de los códigos de estado correspondientes a operaciones realizadas exitosamente.

# Eliminar un registro

**En el caso de utilizar el código de estado 204, no es necesario enviar un mensaje como respuesta a la petición en cuestión, dado que dicho código de estado ya expresa:**

1) **una operación exitosa (204)**
2) **su significado es justamente que no hay contenido**

**Por ello, aunque agreguemos un mensaje como respuesta dentro del método `.send()`, este no llegará a destino. Más allá de esto, siempre convendrá aclararlo en la documentación para aquellos usuarios no técnicos, o con poco conocimiento de Express y Sequelize.**

# Eliminar un registro

```
app.delete('/productos/:ProductID', async (req, res) => {
      const { ProductID } = req.params;

      try {
            const productToDelete = await Product.findByPk(ProductID);

            if (!productToDelete) {
                  return res.status(404).json({ error: 'Producto no encontrado.' });
            }

            await productToDelete.destroy();
            res.status(204).send();

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

El código completo del endpoint para eliminar un producto de la tabla **Northwind.Products.**



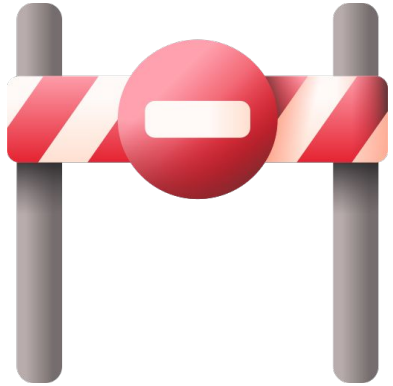
Clasificación YPF: No Confidencial

# Otros posibles errores

# Otros posibles errores

Si formamos parte de un equipo de desarrollo y no tuvimos ningún tipo de injerencia en la construcción de la base de datos SQL, y en la definición de su estructura, es importante que conozcamos de esta algunos aspectos técnicos más allá del acceso a la documentación de sus Tablas, Relaciones, ID’s, Vistas y/o Procedimientos y/o Funciones Almacenadas.

En algunos casos, las operaciones CRUD pueden recurrir eliminar registro o registros de una tabla, los cuales están bloqueados por integridad referencial.



```
const { DataTypes } = require('sequelize');
const sequelize = require('../conexion/connection');

const Category = sequelize.define('Category', {
    CategoryID: {
      type: DataTypes.INTEGER,
      primaryKey: true,
      autoIncrement: true,
    },
    CategoryName: {
      type: DataTypes.STRING,
      allowNull: false,
    },
    Description: {
      type: DataTypes.INTEGER,
      allowNull: true,
      default: null,
    },
}, {
  tableName: 'Categories',
  timestamps: false,
});

module.exports = Category;
```

# Otros posibles errores

Construyamos un ejemplo de eliminación de registros sobre una tabla donde se aplica la Integridad Referencial que restringe la eliminación de datos.

Trabajaremos con la Tabla Categories.

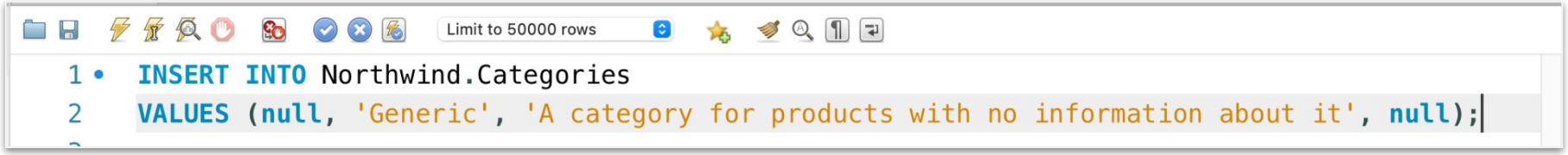
Verifiquemos tener su Modelo construido.

# Otros posibles errores

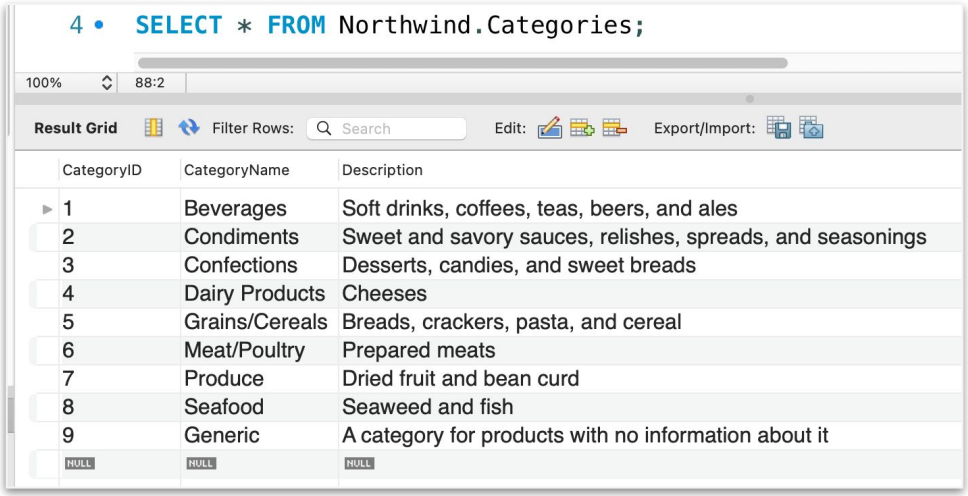
```
const Category = require('./src/modelos/category');

app.use(async (req, res, next) => {
      try {
            await sequelize.authenticate();
            await Product.sync();
            await Category.sync();
            next();
            ...
});
```

También de tenerla referenciada 🟡 en nuestro archivo **server.js** e integrada en la inicialización de la base de datos y 🟢 sincronización de tablas, a través del correspondiente **MiddleWare**.

# Otros posibles errores




Agregamos una categoría genérica a través de **MySQL Workbench**, para intentar eliminarla.




| CategoryID | CategoryName | Description |
|---|---|---|
| 1 | Beverages | Soft drinks, coffees, teas, beers, and ales |
| 2 | Condiments | Sweet and savory sauces, relishes, spreads, and seasonings |
| 3 | Confections | Desserts, candies, and sweet breads |
| 4 | Dairy Products | Cheeses |
| 5 | Grains/Cereals | Breads, crackers, pasta, and cereal |
| 6 | Meat/Poultry | Prepared meats |
| 7 | Produce | Dried fruit and bean curd |
| 8 | Seafood | Seaweed and fish |
| 9 | Generic | A category for products with no information about it |
| NULL | NULL | NULL |

Finalmente validamos el ID que le corresponde, dado que lo necesitaremos para utilizarlo con el cliente HTTP.

# Otros posibles errores

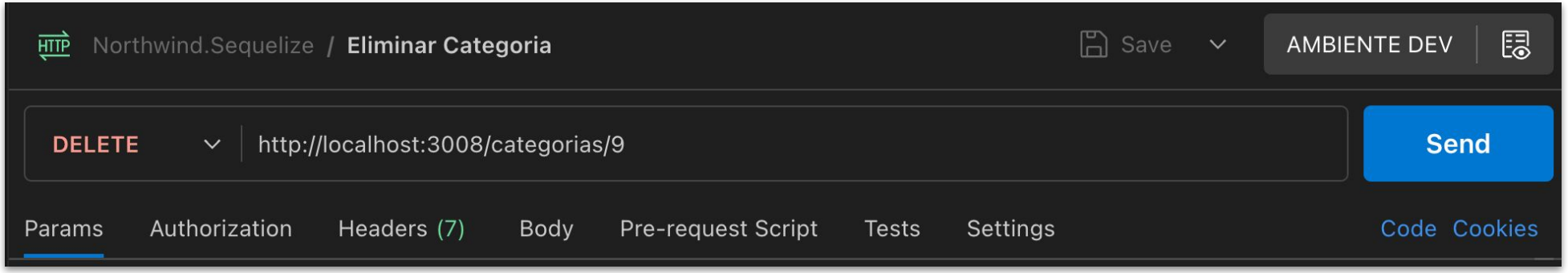
Luego definimos el endpoint para eliminar una categoría, luego de haber creado previamente a ésta, de forma genérica, para llevar adelante este propósito.

```
app.delete('/categorias/:CategoryID', async (req, res) => {
      const { CategoryID } = req.params;

      try {
            const categoryToDelete = await Category.findByPk(CategoryID);
            if (!categoryToDelete) {
                  return res.status(404).json({ error: 'Categoría no encontrada.' });
            }
            await categoryToDelete.destroy();
            res.status(204).send();
      } catch (error) {
            res.status(500).json({
                  error: 'Error en el servidor',
                  description: error.message
            });
      }
});
```

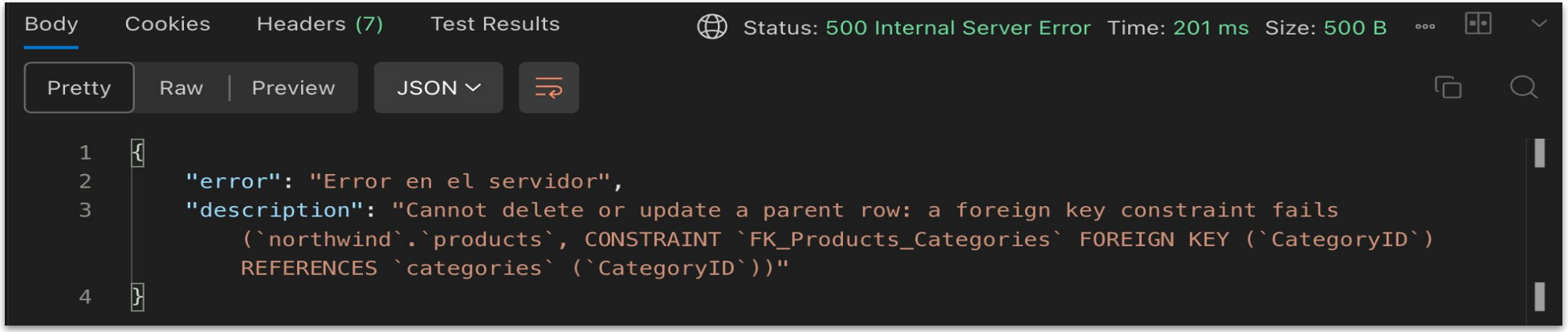


Clasificación YPF: No Confidencial

# Otros posibles errores




Luego probamos de eliminar la categoría creada en MySQL, definiendo en el endpoint el ID correcto que le fue asignada.

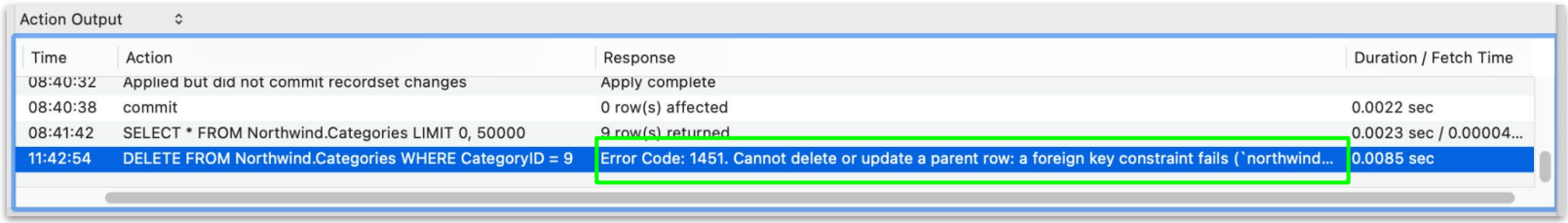


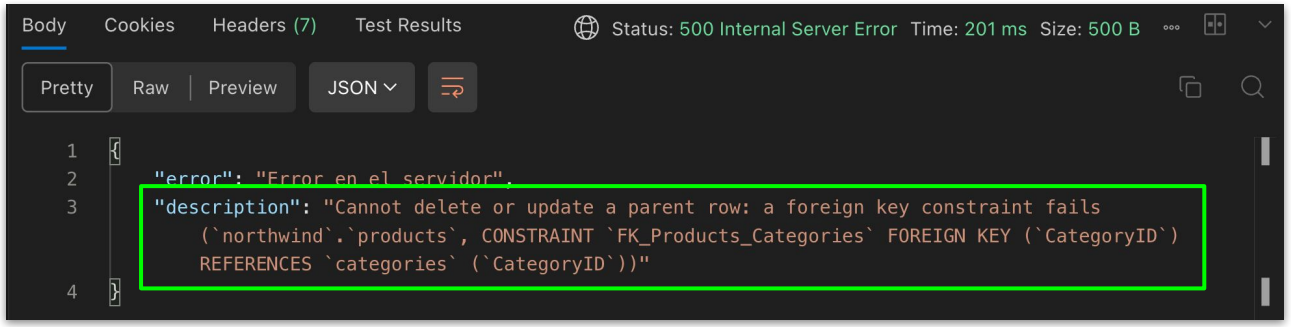

En la descripción del error ocurrido, veremos que se representa el mismo error que obtendremos en MySQL al intentar eliminar el registro.



Clasificación YPF: No Confidencial

# Otros posibles errores




Si bien esto debe prevenirse antes de desarrollar el endpoint, durante la etapa de Análisis, puede darse la situación de que se pase este tipo de detalles. Por ello, es clave definir todas las reuniones previas posibles, además de contar con toda la documentación necesaria, leer y entender la misma.



Clasificación YPF: No Confidencial

# Sección Práctica

# Sección Práctica

Pongamos en práctica el uso de CRUD con Sequelize, a través de los ejercicios prácticos que te proponemos a continuación:

# Sección Práctica

Tiempo estimado: **25 minutos**

1. El profe/tutor@s te compartirán un archivo JSON con una serie de productos nuevos para insertar en Northwind.Products
   a. Valida la estructura de dicho contenido para que coincida con los campos de la tabla Products
   b. Si falta información, valida que el/los campo(s) de la tabla Products permitan la inserción de nulos
   c. Valida también que el Modelo product.js acepte también nulos en dichos campos

2. Si no creaste el endpoint de inserción de Productos, créalo
   a. Define un nuevo Request con THUNDER CLIENT o POSTMAN para inserción de datos, utilizando el endpoint correspondiente
   b. Agrega al menos tres productos del archivo JSON compartido
   c. Verifica luego con el Cliente HTTP que los productos agregados, aparezcan listados

# Muchas gracias.

Ministerio de Economía
Argentina

Secretaría de
Economía del Conocimiento

primero
la gente