Validaciones con Sequelize

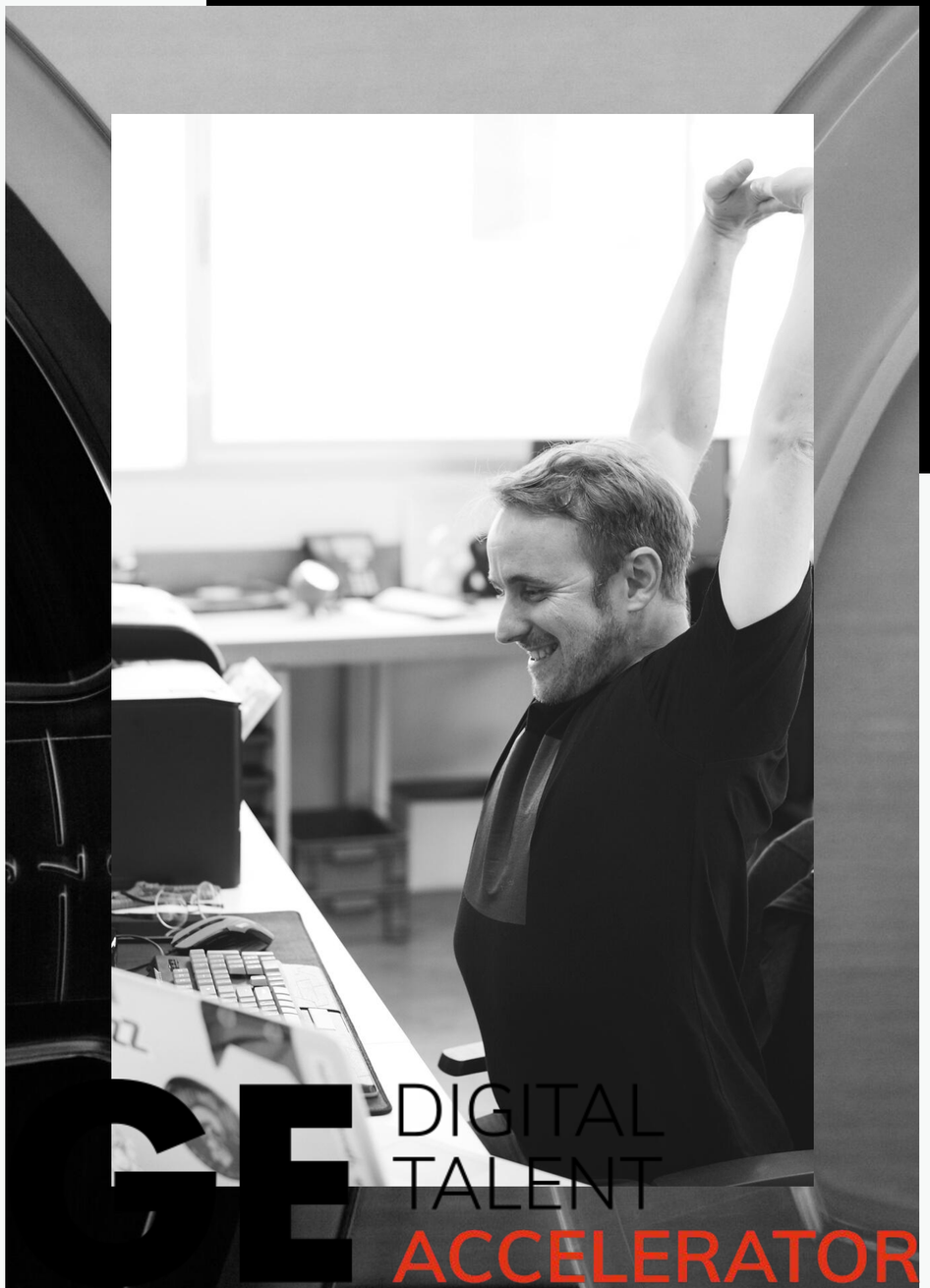
THE BRIDGE
DIGITAL TALENT ACCELERATOR

Índice

¿Qué son las validaciones?

Validaciones en los modelos

Middleware para el manejo de errores



THE BRIDGE DIGITAL TALENT ACCELERATOR

¿Qué son las validaciones?

La validación de datos es un proceso que asegura la entrega de datos limpios y claros a los programas, aplicaciones y servicios que lo utilizan. **Comprueba la integridad y validez de los datos que se están introduciendo en diferentes software y sus componentes**. La validación de los datos garantiza que los datos cumplen con los requisitos y los parámetros de calidad.

La validación de datos ayuda principalmente a garantizar que los datos enviados a las aplicaciones conectadas sean completos, precisos, seguros y consistentes. Esto se logra a través de controles de validación de datos y reglas que rutinariamente comprueban la validez de los datos.

¿Dónde hacer las validaciones?

Antes de hacer las validaciones lo primero que debes pensar es dónde debe realizarse esa validación, ¿en el **cliente o en el servidor**?

¿Dónde hacer las validaciones?

La respuesta corta primero: Idealmente las dos, y si sólo puedo elegir una, definitivamente del lado servidor.

Validando campos

Lo primero que debemos **valorar es si el campo es nullable o no**, en el caso de un usuario querremos su nombre y correo si o si, por lo que le decimos que no es nullable, **y en el correo además queremos validar si** de verdad el correo que se nos envía **es un correo y no cualquier otra cosa**, así que también validamos eso:

- Recursos: [Validaciones Sequelize]

```
User.init(
  {
    name: {
      type: DataTypes.STRING,
      allowNull: false,
      validate: {
        notNull: {
          msg: "Por favor introduce tu nombre",
        },
      },
    },
    email: {
      type: DataTypes.STRING,
      allowNull: false,
      validate: {
        notNull: {
          msg: "Por favor introduce tu correo",
        },
        isEmail: {
          msg: "Por favor introduce un correo valido",
        },
      },
    },
  password: DataTypes.STRING,
. . .
```

le decimos que el campo no puede ser nulo

Validamos que el correo que se rellena es un correo

Cambiar una columna de una Tabla

Creamos una nueva migración para cambiar una columna de la tabla users:

```
$ sequelize migration:generate --name changeUserColumn
```

Cambiar una columna de una Tabla

Ahora en el nuevo archivo que hemos creado le diremos que el campo email de la tabla users sea de tipo string y único:

```
async up(queryInterface, Sequelize) {
    return queryInterface.changeColumn("Users", "email",  {
      type:Sequelize.STRING,
      unique:true
    });
  },
```

definimos el campo como único

Middleware errors

Ahora vamos a crear un middleware que nos va a permitir controlar los errores a la hora de crear un elemento nuevo.

Actualizamos el controlador

Vamos a introducir cambios en la parte del manejo del error en el controlador UserController (catch), para ello necesitamos **utilizar el parámetro “next” y pasarle el error** que tenemos capturado. También **vamos a modificar el error añadiendo una nueva propiedad “origin”.**

```
async create(req, res,next) {
    try {
        req.body.role = "user";
        const password = await bcrypt.hash(req.body.password, 10);
        const user = await User.create({ ...req.body, password });
        res.send(user);
    } catch (error) {
        console.error(error)
        next(error)
    }
},
```

Incluimos el parámetro next

Le pasamos el error al next

Middleware Errors

En la **carpeta “middlewares”** creamos un archivo llamado **“errors.js”**.

};

Este middleware lo utilizamos para manejar errores de validación.

En la función typeError, se comprueba si el error **err** es de tipo **SequelizeValidationError** o **SequelizeUniqueConstraintError**. Estos son tipos de errores específicos que se producen cuando hay problemas de validación en los datos proporcionados por el cliente. Si se produce alguno de estos errores, se llama a la **función handleValidationError para manejar el error y devolver una respuesta adecuada al cliente.**

```
const handleValidationError = (err, res) => {
  let errors = err.errors.map((el) => el.message);
  if (errors.length > 1) {
    const msgErr = errors.join(" || ");
    res.status(400).send({ messages: msgErr });
  } else {
    res.status(400).send({ messages: errors });
  }
};

const typeError = (err, req, res, next) => {
  if (
    err.name === "SequelizeValidationError" ||
    err.name === "SequelizeUniqueConstraintError"
  ) {
    handleValidationError(err, res);
  } else {
    res.status(500).send({ msg: "Hubo un problema",err });
  }
};

module.exports = { typeError };
```

Filtramos por el tipo de error

Middleware Errors

En el caso de que el **error no pertenezca a ninguna validación del modelo**, **mostraremos un mensaje de que algo ha ido mal** en esa parte del controlador, con un estado 500 y devolveremos el error.

```
const typeError = (err, req, res, next) => {
  if (
    err.name === "SequelizeValidationError" ||
    err.name === "SequelizeUniqueConstraintError"
  ) {
      handleValidationError(err, res);
  } else {
    res.status(500).send({ msg: "Hubo un problema",err
});
  }
};
```

Mensaje 500 en caso de fallo en los controladores

Nuestro hilo principal ha cambiado

En **nuestro archivo principal** (index.js, main.js, ...) **vamos a incorporar el nuevo middleware**, y además lo vamos a llamar justo después de cuando se terminan de ejecutar las rutas.

```
const express = require('express');
const app = express();
const { typeError } = require('./middlewares/errors');

app.use(express.json())

app.use('/books',require('./routes/books'))
app.use('/genres',require('./routes/genres'))
app.use('/users',require('./routes/users'))

app.use(typeError)

app.listen(8000,()=>  console.log('servidor levantado en el puerto 8000'))
```

Importamos nuestro middleware

Ejecutamos el middleware