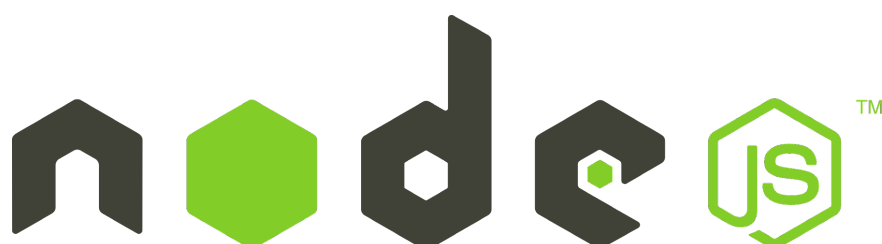


DESARROLLO DE SERVICIOS WEB CON NODEJS



Email: *julietalanciotti@gmail.com*

Módulo 5 - Migraciones

Migraciones

Ejecutar Migraciones

Seeders

Ejecutar Seeders

Validación de Datos

Paso 1 - Instalar dependencia JOI

Paso 2 - Crear directorio middlewares y schemes

Paso 3 - Crear esquema del usuario

Paso 4 - Archivo Validate

Paso 5 - Llamar al validate

Manejo de Errores

Paso 1 - Crear middleware

Paso 2 - Ejecutar middleware

Paso 3 - Errores Personalizados

Paso 4 - Error de validación

ANEXO

Migraciones

Una migración en Sequelize es un archivo javascript que exporta dos funciones: Up y Down. Estas dicen cómo realizar la migración y cómo deshacerla.

Estas funciones deben definirse manualmente. Se debe escribir las consultas necesarias utilizando las funciones que brinda Sequelize.

Para mantener el orden del proyecto, utilizaremos el directorio llamado **migrations**. En esta carpeta estarán todos los archivos que se tengan que ejecutar.

Por ejemplo: supongamos que queremos agregar una columna a la tabla paciente. Como no queremos eliminar nuestra tabla con datos ya cargados, debemos realizar una migración para realizar la modificación al modelo.

```
'use strict';

module.exports = {

  up: (queryInterface, Sequelize) => {

    return Promise.all([ //NOS PERMITE HACER MAS DE UN CAMBIO

      // A LA TABLA PACIENTE LE AGREGAREMOS UN CAMPO LLAMADO SINTOMA QUE ES DE TIPO STRING Y
      ACEPTA EL VALOR NULL
      queryInterface.addColumn('paciente', 'sintoma', {
        type: Sequelize.STRING,
        allowNull: true,
      }),

    ]);
  },

  down: (queryInterface, Sequelize) => {
    // SI NO FUNCIONA LA MIGRACION, PODRÍAMOS REVERTIR CAMBIOS, O ELIMINAR LA TABLA POR EJEMPLO.
    //DEPENDE DE CADA UNO LA FUNCIONALIDAD QUE SE LE QUIERA DAR. NO HACE FALTA IMPLEMENTARLO
  }
};
```

1. Con la instrucción **addColumn** agregamos una columna a una tabla ya existente
2. Con la instrucción **changeColumn** editamos las propiedades de una columna. Por ejemplo el tipo, valor por defecto, si acepta o no el valor null, etc

 Para más información sobre migraciones:

<https://sequelize.org/docs/v6/other-topics/migrations/>

Ejecutar Migraciones

`npm run sequelize db:migrate` → de esta manera se ejecutarán todas las migraciones que se encuentren en la carpeta /database/migrations.

Importante!

La ejecución de las migraciones quedan guardadas en un historial, por lo tanto, tener en cuenta que si no se ejecuta algún cambio, probablemente sea porque quedó registrado como que ya se ejecutó.

Seeders

Supongamos que queremos insertar cierta información por default en alguna tabla.

Por ejemplo, información de prueba, valores constantes en tablas, etc.

Para esto existen los **seeders**. Son un tipo de migración que se ejecutarán, insertando datos en una tabla.

Para esto, utilizaremos el **directorio seeders**. Allí estarán todos los archivos que se ejecutarán.

*Por ejemplo, supongamos que queremos agregar usuarios de prueba. Lo que deberíamos hacer será “simular” un **insert into** de SQL, pero en formato Sequelize:*

```
'use strict';

const models = require("../models/index");

module.exports = {
  up: function (queryInterface, Sequelize) {
    return Promise.all([
      models.usuario.findOrCreate({
        where: {
          id: "1"
        },
        defaults: {
          nombre: "Juan",
          apellido: "Gonzalez",
          email: "emailJuan@email.com",
          edad: 25
        }
      }),
      models.usuario.findOrCreate({
        where: {
          id: "2"
        },
        defaults: {
          nombre: "Pedro",
          apellido: "Gomez",
          email: "emailPedro@email.com",
          edad: 67
        }
      })
    ])
  },
};
```



A diferencia de las migraciones, la ejecución de los seeders no se guardan en ningún historial.

Ejecutar Seeders

`npm run sequelize db:seed:all` → de esta manera se ejecutarán todos los seeders que se encuentren en la carpeta `/database/seeders`.

Validación de Datos

Existen muchas maneras de manejar los errores que se presentan en nuestro código. En este curso hablaremos de una manera de cómo manejar errores sencillos y validar algunos datos.

Veamos primero cómo podemos validar datos. En nuestro caso nos enfocaremos en validar los datos que vienen por POST a través del body.

Por ejemplo, supongamos que tenemos el siguiente endpoint: **POST /api/usuarios/**

```
crear: async (req, res, next) => {  
  
  const user = await models.usuario.create(req.body)  
  
  res.json({  
    success: true,  
    data: {  
      id: user.id  
    }  
  })  
},
```

Esa función se encargará de crear un usuario en nuestro sistema. Supongamos también que nuestro usuario está conformado por estos campos:

```
nombre: {  
  type: DataTypes.STRING,  
  allowNull: false  
},  
  
apellido: {  
  type: DataTypes.STRING,  
  allowNull: false  
},  
  
email: {  
  type: DataTypes.STRING,  
  allowNull: true  
},  
  
edad: {  
  type: DataTypes.INTEGER,  
  allowNull: true  
},
```

Como vemos, tiene 4 campos. Por lo tanto, cuando ejecutamos el endpoint de crear, deberíamos mandarle por el body esa información.

¿Pero qué pasa si le enviamos otra información? ¿Qué pasa si no le mandamos el nombre o el apellido, siendo estos obligatorios?

Para esto sirve validar los datos. Es importante hacerlo para evitar errores, y lograr una pequeña “seguridad”.

Veamos paso a paso cómo poder configurar esto.

 Te recomiendo ver este video donde muestro cómo hacerlo:

<https://youtu.be/5hRO2AXe4bY>

Paso 1 - Instalar dependencia JOI

Joi (object schema validation) es una librería que nos va a ayudar a validar los esquemas.

Para instalarla, debemos ejecutar el siguiente comando → `npm install joi`

 Para más información, te dejo la documentación:

<https://www.npmjs.com/package/joi>

Paso 2 - Crear directorio middlewares y schemes

Dentro de la carpeta **/src** crearemos una nueva carpeta llamada **middlewares**.

Un *middleware* es una función que se puede ejecutar antes o después del manejo de una ruta. Esta función tiene acceso al objeto Request, Response y la función next().

Suelen ser utilizadas como mecanismo para verificar niveles de acceso antes de entrar en una ruta, manejo de errores, validación de datos, etc.

Dentro de esta carpeta **/middlewares**, crearemos una segunda carpeta llamada **schemes**. Esta carpeta contendrá todos los archivos que indiquen el esquema que deberán cumplir las peticiones de cada ruta. Es decir, por ejemplo, indicar los tipos de datos y nombres de los campos que debe mandarse por el body cuando queramos crear un nuevo usuario.

Paso 3 - Crear esquema del usuario

Dentro de la carpeta /scheme crearemos un archivo que se llamará **usuario.scheme.js**

Este archivo utilizará la dependencia de Joi anteriormente instalada, la cual nos permitirá indicar los datos que se esperan recibir.

```
const Joi = require('joi') // importar Joi para validar los datos de entrada

let crearUsuario = Joi.object({
  nombre: Joi.string().required(),
  apellido: Joi.string().required(),
  email: Joi.string().email().optional(),
  edad: Joi.number().optional()
})

module.exports = {
  crearUsuario
}
```

Veamos línea por línea qué significa este código:

- **const Joi = require('joi')** → importamos la librería para poder utilizar las funciones.

A continuación, crearemos un objeto con Joi (como nos indica la documentación), para poder posteriormente validar la información.

- **nombre: Joi.string().required()** → está indicando que por el body se espera recibir un parámetro llamado **nombre**, que deberá ser de tipo **string**, y que es **obligatorio** que exista en el body.

- `email: Joi.string().email().optional()` → el campo email será **opcional**, es decir, si no se envía por el body, está permitido. Deberá ser de tipo **string** y deberá cumplir con el **formato de un email**.
- `edad: Joi.number().optional()` → el campo edad será **opcional** enviarlo y deberá de ser un **número**.

En caso de que algunas de estas “condiciones” no se cumplan, la dependencia nos brindará un mensaje de error que indique al usuario que ingresó mal los datos.

Paso 4 - Archivo Validate

Como esto no funcionará mágicamente solo, debemos hacer ciertas configuraciones.

Crearemos un *middleware*. Dentro de la carpeta `/middlewares` crearemos un archivo llamado **validate.js**

Este archivo se ejecutará “en medio” del llamado al controller. De esta manera, antes que comience a ejecutarse nuestro controlador con la función correspondiente, primero se validará el esquema de datos que vienen por el body.

```
const Joi = require('joi') // importar Joi para validar los datos de entrada

module.exports = (scheme) => {

  return (req, res, next) => {
    let result = scheme.validate(req.body) // valida los datos de entrada con el esquema

    if (result.error) { // si hay error
      next(result.error) // envia el error al controlador
    } else { // si no hay error
      next() // continua con la ejecucion del controlador
    }
  }
}
```

Con este código, lo que le estaremos diciendo es que recibirá un esquema, y lo verificará. Si todo está bien, seguirá la ejecución normal del endpoint (**función next()**), pero si hay algún error, entonces lo informará.

Paso 5 - Llamar al validate

Una vez configurado, solo resta llamar a la ejecución de todo esto. Para hacerlo, como dijimos anteriormente, el *middleware* se ejecutará antes del controlador. Es por eso, que dentro del archivo de rutas del usuario (en este ejemplo), deberemos llamar a la **función validate**.

```
// Archivo usuario.routes.js

const router = require("express").Router();
const usuarioController = require('../controllers/usuario.controller')

// importar el middleware de validacion de datos
const validate = require('../middlewares/validate')

// importar el scheme de validacion de datos
const usuarioScheme = require('../middlewares/schemes/usuario.scheme')

router.post('/', validate(usuarioScheme.crearUsuario), usuarioController.crear)

module.exports = router;
```

Como se puede observar, para realizar esto debemos traernos la función `validate` de nuestro middleware:

```
const validate = require('../middlewares/validate')
```


Luego, deberemos traernos el esquema del usuario. En nuestro caso, solo hicimos un esquema para la función `crear`, pero podríamos tener más. Por ejemplo para cuando se quiera editar un usuario, o para algún filtro, etc.

```
const usuarioScheme =
require('../middlewares/schemes/usuario.scheme')
```

Solo falta llamar a la función `validate`, enviando el esquema. Es por eso, que dentro de la “declaración” de la ruta, escribimos:

```
validate(usuarioScheme.crearUsuario)
```

Manejo de Errores

 **Recomiendo ver el siguiente video dónde ponemos en práctica todo lo que tiene que ver con manejo de errores:** <https://youtu.be/Ua7khncnIQY>

Respecto al manejo de errores, en este curso aprenderemos una manera de mostrar de mejor manera ciertos errores que pueden aparecer en nuestra API. (incluso los generados por la validación de datos)

Por ejemplo, además de validar los datos que vienen por el body cuando queremos crear un usuario, podríamos informar también que un usuario específico no existe si se pide mostrar su información y no se encuentra en la BD.

Por ejemplo, supongamos que tengo la siguiente ruta: **GET /api/usuarios/1**

Como ya sabemos, este endpoint se encargará de listar la información del usuario con ID=1. **Pero, ¿qué pasaría si ese usuario no existe?**

Veamos cómo es nuestro funcionamiento del endpoint con lo que aprendimos hasta ahora:

```
listarInfo: async (req, res, next) => {  
  
  const user = await models.usuario.findOne({  
    where: {  
      id: req.params.idUsuario  
    }  
  })  
  
  res.json({  
    success: true,  
    data: {  
      usuario: user  
    }  
  })  
  
},
```

Hasta ahora teníamos una función **listarInfo** que se encargaba de buscar un usuario con la condición de que coincidiera el parámetro que venía, con el campo ID de la tabla usuario.

Si el usuario no existe, el JSON que devolveremos será **null**. Esto está bien, pero no del todo. Lo ideal sería poder indicar qué tipo de error hubo, o mostrar algún tipo de mensaje para indicar que este usuario no existe.

Con la forma que veremos de manejar esto, nos servirá para manejar diferentes tipos de errores, por ejemplo: not found, errores de sequelize, errores de CORS, errores de conexión a la BD, etc.

Paso 1 - Crear middleware

Obviamente, para poder manejar este tipo de errores, necesitamos crear un middleware. Es por eso, que dentro de la **carpeta /middlewares** crearemos un archivo llamado **error.js**. Este archivo se encargará de todo lo que es el funcionamiento de los errores y qué mensaje mostrar.

```
module.exports = function (err, req, res, next) {

  let response = {
    success: false,
    error: {
      code: err.code || 500, // si no hay un codigo de error, se asigna 500
      message: err.message || 'Internal server error'
      // si no hay un mensaje de error, se asigna 'Internal server error'
    }
  }

  // si el error es de NotFound
  if (err.message === 'Not Found') {
    response.error.code = 404
    response.error.message = 'Not Found'
  }

  res.status(200).json(response) // envia la respuesta al cliente
}
```

Veamos el siguiente código:

- **module.exports = function** → este archivo exportará una función que es la que se encargará de manejar los errores.
- **let response** → vamos a crear una variable llamada *response*. Esta variable será el mensaje que devolverá nuestro endpoint.

- El código será el código del error. Si no tiene, se le asigna el código 500 por defecto.
- El mensaje será lo que se muestre.
- `if(err.message === 'Not Found')` → Si el error que se recibe tiene como mensaje “Not Found”, entonces devolveremos el código 404 con el texto correspondiente. (se lo asignamos a la respuesta **response**)
- `res.status(200).json(response)` → por último, haremos que nuestra api devuelva la respuesta en formato JSON.

Paso 2 - Ejecutar middleware

Una vez creado nuestro código del middleware, al igual que con el validate, necesitamos decir cuándo se ejecutará.

Este tipo de middleware, se llamará luego de configurar las rutas. Recordemos la estructura del archivo **index.js** que configuraba toda nuestra API.

Teníamos una función llamada **configuracionApi** y otra función llamada **configuracionRouter**. Estas dos funciones, las llamábamos en nuestra función **init()** que era la que se ejecutaba al principio de todo para levantar nuestro servidor Express.

Luego de configurar el router, es necesario llamar a nuestro manejador de errores. Esto es porque, gracias a la función **next()** y el asincronismo, al producirse un error, podremos “recibirlo” con nuestro middleware.

1. Deberemos traernos la función que creamos en nuestro middleware:

```
let errorHandler = require('./middlewares/error')
```

2. Además, utilizaremos una dependencia ya instalada llamada **http-errors**

```
let createError = require('http-errors')
```

Esta dependencia nos permitirá crear nuestros propios errores bajo el protocolo http.

3. Dentro de la función **configuracionRouter**, debajo de la configuración de las rutas, deberemos escribir el siguiente código:

```
const configuracionRouter = (app) => { // configurar las rutas
  app.use('/api/', routerConfig.rutas_init())

  app.use(function (req, res, next) {
    next(createError(404))
  })

  app.use(errorHandler)
};
```

Simplificando la explicación de estas líneas de código, lo que hacen es simplemente verificar si la ruta no se encuentra. Si pasa esto, se crea un error 404, y luego se ejecuta el manejador de errores.

De esta manera, cuando se ejecuta el middleware, va a ejecutar la función recibiendo un **err**. Al asignarle el **Not Found** se ejecutará el if que programamos, y podremos mostrar el mensaje de error.

Paso 3 - Errores Personalizados

Veamos cómo con esto ya configurado, podemos mostrar nuestros propios errores. Volviendo al ejemplo que vimos del usuario inexistente, mostraremos un mensaje que indique esto en vez de solo devolver **null**.

Dentro de la carpeta **/const** crearemos un archivo llamado **errors.js**. Este archivo guardará todos los nombre de los errores que queramos crear, junto con su código de error y un mensaje.

Por ejemplo, para poder mostrar el mensaje de *“El usuario no existe”*, debemos escribir el siguiente código en este nuevo archivo:

```
module.exports = {  
  
  'UsuarioInexistente': {  
    code: 1002,  
    message: 'El usuario no existe'  
  }  
  
}
```

Una vez creado el formato de nuestro error, deberemos “cortar” la ejecución de nuestro controlador, para que el error llegue al manejador de errores.

Por lo tanto, en nuestro controller de usuario, luego de buscar el usuario, deberemos verificar si existe o no. Si no existe, devolveremos el error de la siguiente manera:

```
listarInfo: async (req, res, next) => {
  try {
    const user = await models.usuario.findOne({
      where: {
        id: req.params.idUsuario
      }
    })
    if(!user) return next(errors.UsuarioInexistente)

    res.json({
      success: true,
      data: {
        usuario: user
      }
    })
  } catch (err) {
    return next(err)
  }
},
```

- `if(!user) return next(errors.UsuarioInexistente)` → recordar la función de `next()`. Esta función hace que se siga ejecutando lo que sigue de nuestro código. Es por eso que el error lo puede recibir el manejador. El `return` hará que corte la ejecución de nuestra función `listarInfo`.
- `errors.UsuarioInexistente` → será el mensaje de error que estaremos enviando al manejador. Para esto deberemos importar el archivo `errors.js` creado anteriormente. Por lo tanto, al principio de nuestro controller, escribiremos: `const errors = require("../const/errors")`

Entonces nuestro controlador se verá así:

```
const models = require("../database/models/index")
const errors = require("../const/errors")

module.exports = {

  listarInfo: async (req, res, next) => {
    try {
      const user = await models.usuario.findOne({
        where: {
          id: req.params.idUsuario
        }
      })
      if(!user) return next(errors.UsuarioInexistente)

      res.json({
        success: true,
        data: {
          usuario: user
        }
      })
    } catch (err) {
      return next(err)
    }
  }
}
```

Paso 4 - Error de validación

Si queremos mostrar con un mensaje de error personalizado lo que tenga que ver con la validación de datos, podemos hacer lo siguiente:

Dentro del archivo de **/const/errors.js** podemos crear dos tipos de errores personalizados:

```
'ValidationError': {  
  code: 1000,  
  message: 'Error de validacion'  
},  
'FaltanCampos': {  
  code: 1001,  
  message: 'Faltan parámetros necesarios'  
},
```

1. Uno será un error de validación (si no se cumplen los tipos de datos), y otro error será si el usuario se olvidó de enviar campos obligatorios.
2. Dentro del archivo **/middlewares/error.js** deberemos verificar si el error es de validación de datos. Para esto, escribiremos el siguiente código:

```
const errors = require('../const/errors')
...

if (err.isJoi) {
  let validationErrorType = err.details[0].type
  let errorKey = 'ValidationError'
  if (validationErrorType === 'any.required') {
    errorKey = 'FaltanCampos'
  }
  response.error.code = errors[errorKey].code
  response.error.message = errors[errorKey].message
}
```

Este código preguntará si el error surgió a causa de nuestra dependencia JOI. Si es así, guardaremos el tipo de error en una variable llamada **validationErrorType**, y consideraremos que nuestro error, en principio será de validación.

Si nuestro error fue porque faltaban parámetros, entonces a nuestra respuesta le asignaremos el código y mensaje que generamos en el archivo **/const/errors.js**.

De esta manera, dependiendo el error que sea, es el mensaje que devolverá nuestra API.

ANEXO

Como un extra, les dejo algunas validaciones más para el archivo
/middlewares/error.js

```
/**
 * Sequelize validation error (from the model)
 */
if (err.name === 'SequelizeValidationError') {
  let validationError = err.errors[0]
  response.error.code = errors['ValidationError'].code
  response.error.message = validationError.message
}

/**
 * CORS not allowed error
 */
if (err.message === 'Not allowed by CORS') {
  response.error.code = 403
}

/**
 * Sequelize error produced when a value is out of the range of the defined type
 */
if (err.name === 'SequelizeDatabaseError' && err.message.indexOf('out of range') >= 0) {
  response.error.code = errors['ValidationError'].code
  response.error.message = errors['ValidationError'].message
}

/**
 * Sequelize connection error with the database
 */
if (err.name === "SequelizeConnectionError") {
  response.error.code = 500
  response.error.message = 'Internal server error'
}
```