LABS-RESUELTOS-DE-NOOB-A-PRO-EN-...

PesteInfinita

Introducción a la Ingeniería del Software y los Sistemas de Información II

2º Grado en Ingeniería Informática - Ingeniería del Software

Escuela Técnica Superior de Ingeniería Informática
Universidad de Sevilla

DE NOOB A PRO EN LAB IISSI II

Soy Gigi 🚨 Y te traigo este perfecto manual para lograr entender esta asignatura que se explica tan mal en la US. El fallo es que no dan los apuntes de una forma clara y para toda la familia. No te preocupes, vamos a ver qué es lo que hay que saber y vas a acabar siendo todo un experto en esto. MONDONGO.

CONTACTA CONMIGO PARA MANDARME NUDES, SER MI AMIGO (ESTOY MÁS SOLA QUE LA UNA) O DONARME DINERITO POR PAYPAL: **gigienlacasa@gmail.com**

0. PASOS PREVIOS

Necesitarás el siguiente software:

MariaDB: https://mariadb.org/download/

MacOS:https://mariadb.com/kb/en/installing-mariadb-on-macos-using-homebrew/

Node.js: https://nodejs.org/en/download/

Nodemon: npm update & npm install -g nodemon

Visual Studio Code: https://code.visualstudio.com/

Extensions:

- **ThunderClient** (rangav.vscode-thunder-client)
- **ES7+ React/Redux/React-Native Snippets** (dsznajder.es7-react-js-snippets)
- **ESLint** (dbaeumer.vscode-eslint)

Git: https://git-scm.com/

Expo: npm install --global expo-cli

Configuración inicial del primer proyecto:

• Únete a una asignación en GitHub Classroom

• Tu profesor de laboratorio compartirá un enlace como este: https://classroom.github.com/a/xxxxxx

• Debes iniciar sesión en github.com (ten en cuenta que NO es github.eii.us.es)

• La primera vez que aceptes una asignación, se te pedirá que vincules a tu usuario de GitHub a un miembro específico de tu aula de laboratorio.

• Se creará una copia de un repositorio en tu cuenta de GitHub.

• Clona el repositorio en Visual Studio Code

• Instala las dependencias del proyecto: npm install

• Crea una tabla de datos "deliverus" y un usuario en MariaDB

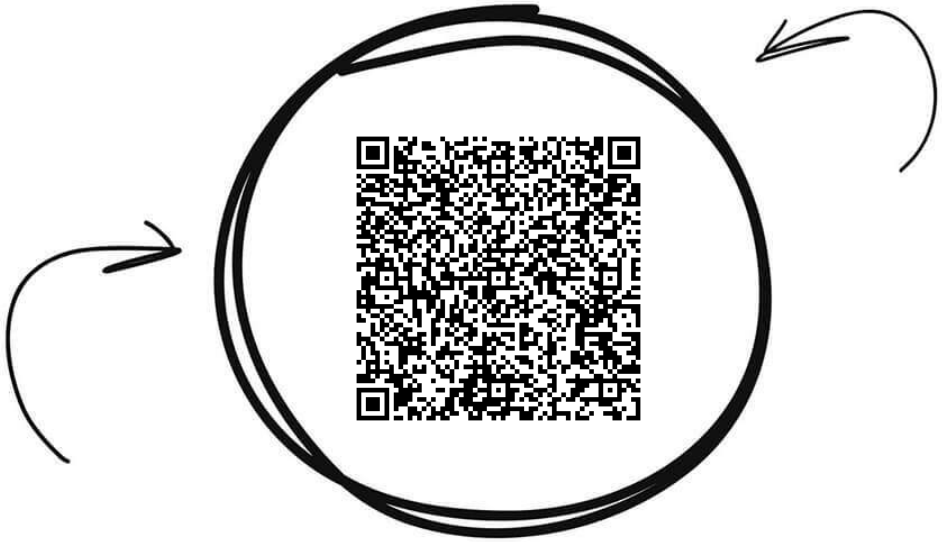
• Establece todos los permisos para el usuario en "deliverus"

• Copia .env.example a .env, y configura el usuario de la base de datos, la contraseña y el nombre.

• Inicia el servidor: nodemon


McDonald's
“SOY CREW DE McDONALD’S,
Y POR SUPUESTO QUE
MI TRABAJO Y MI PASIÓN
SON COMPATIBLES”
¿TE VIENES?
My CREW
Mi trabajo. Mi pasión. Mi gente.

Introducción a la Ingeniería...



Banco de apuntes de la

WUOLAH

Comparte estos flyers en tu clase y consigue más dinero y recompensas

1. Imprime esta hoja
2. Recorta por la mitad
3. Coloca en un lugar visible para que tus compis puedan escanar y acceder a apuntes
4. Llévate dinero por cada descarga de los documentos descargados a través de tu QR

PRIMERA PARTE: BACKEND

1. LAB 1 - Backend model / Modelo backend

Introducción

Aprenderemos cómo ejecutar un servidor HTTP básico de *Node.js* y configurar nuestra estructura de proyecto usando el marco de trabajo *Express*. En segundo lugar, comprenderemos cómo crear el modelo de nuestro proyecto (a partir del patrón MVC) y aprenderemos cómo el paquete *Sequelize* nos ayudará a crear el esquema de la base de datos relacional y realizar operaciones en la base de datos Maria.

BACKEND, ¿QUÉ ES?

Imaginemos que estás cocinando una pizza REALFOOD alta en proteína para las gains. La pizza tiene dos partes principales: la masa y los ingredientes. La masa es la base de la pizza, es lo que sostiene los ingredientes y da forma a la pizza. Los ingredientes son lo que hacen que la pizza esté más buena y están sobre la masa.

Ahora, si pensamos en una aplicación o sitio web, el frontend sería como los ingredientes de la pizza. Es lo que los usuarios ven y con lo que interactúan. Pueden hacer clic en botones, escribir en formularios y ver el contenido en la pantalla.

Por otro lado, el backend sería como la masa de la pizza. Es la parte que no se ve, pero que es esencial para que todo funcione correctamente. Es donde se guarda la información de la aplicación, se procesan las solicitudes de los usuarios y se comunican con otras aplicaciones y servicios.

Así como la masa de la pizza puede ser de diferentes tipos y espesores, el backend también puede ser de diferentes tipos y complejidades. Pero lo importante es que esté ahí, funcionando correctamente, para que todo el sitio o la aplicación puedan ser disfrutados por los usuarios de manera suave y sin problemas.

Requisitos previos

- Tenga en cuenta que estamos desarrollando el software de backend necesario para el proyecto DeliverUS. Por favor, lea los requisitos del proyecto que se encuentran en: https://github.com/IISSI2-IS-2022-2023/DeliverUS-Backend-2022-2023/blob/main/README.md
    - El proyecto de plantilla incluye la configuración de EsLint para que solucione automáticamente los problemas de formato tan pronto como se guarda un archivo.

Ejercicios

1. Aceptar la tarea de GitHub Classroom y clonar

Acepta la tarea de GitHub Classroom para crear tu propio repositorio basado en esta plantilla. A continuación, clona tu propio repositorio abriendo VScode y clona el repositorio base del laboratorio abriendo la paleta de comandos (Ctrl+Shift+P o F1) y clonando este repositorio de Git, o usando la terminal y ejecutando `git clone <url>`

Alternativamente, puedes usar el botón *Control de origen* en la barra lateral izquierda y hacer clic en el botón *Clonar repositorio*.

En caso de que se te pregunte si confías en el autor, por favor selecciona sí.

Puede ser necesario configurar tu nombre de usuario de git ejecutando los siguientes comandos en tu terminal para poder confirmar y enviar:

```
git config --global user.name "FIRST_NAME LAST_NAME" git config --global
user.email "MY_NAME@example.com"
```

2. Inspeccionar la estructura del proyecto

Encontrarás los siguientes elementos (algunos de ellos aparecerán en los siguientes laboratorios):

- **`package.json`:** scripts para ejecutar el servidor y las dependencias de paquetes, incluyendo express, sequelize y otros. Este archivo se suele crear con `npm init`, pero ya lo encontrarás en tu proyecto clonado.
    - Para añadir más dependencias de paquetes, debes ejecutar `npm install nombrePaquete --save` o `npm install nombrePaquete --save-dev` para las dependencias necesarias sólo para el entorno de desarrollo (por ejemplo, nodemon). Para obtener más información sobre npm, consulta su documentación.
- **`package-lock.json`:** instala exactamente las mismas dependencias en futuros despliegues. Ten en cuenta que las versiones de las dependencias pueden cambiar, por lo que este archivo garantiza descargar y desplegar el mismo árbol de dependencias exactamente.
- **`backend.js`:** ejecuta el servidor http, configura las conexiones a Mariadb e inicializa varios componentes.
- **`.env.example`:** ejemplo de variables de entorno.
- **Carpeta `models`:** donde se definen las entidades del modelo.
- Carpeta **`database`:** donde se encuentra toda la lógica para crear y poblar la base de datos
    - Carpeta **`database/migrations`:** donde se define el esquema de la base de datos.
    - Carpeta **`database/seeders`:** donde se define la muestra de datos de la base de datos.
- Carpeta **`routes`:** donde se definen las URIs y se hace referencia a middlewares y controladores.
- Carpeta **`controllers`:** donde se implementa la lógica empresarial, incluidas las operaciones en la base de datos.
    - Carpeta **`controllers/validation`:** validación de los datos incluidos en las solicitudes de los clientes. Un archivo de validación para cada entidad.
- Carpeta **`middlewares`:** varios controles necesarios, como autorización, permisos y propiedad.

- Carpeta **`config:`** donde se almacenan algunos archivos de configuración global (para ejecutar migraciones y semillas desde la línea de comandos).
- Carpeta **`example_api_client:`** almacenará solicitudes de prueba a nuestra API Rest.
- Carpeta **`.vscode:`** configuración de VSCode para este proyecto.

3. Inspeccionar y ejecutar backend.js

3.1. Valores de entorno

Necesitamos un archivo de entorno que incluya las credenciales de nuestra base de datos. Para ello, haga una copia de .env.example y llame al nuevo archivo como **.env** en la carpeta raíz del proyecto.

Reemplace los valores de conexión de la base de datos para que coincidan con sus credenciales de base de datos.
Es importante tener en cuenta que el archivo .env contiene credenciales para acceder a su base de datos, por lo que no debe ser enviado a su repositorio (como se especifica en .gitignore).

NOTA: necesita un usuario de base de datos y un esquema de base de datos llamado deliverus *(VER 0.PASOS PREVIOS)*

3.2. Ejecutar el servidor HTTP y conectarse a la base de datos

Ejecutaremos nuestra primera versión del servidor backend. Primero podemos inspeccionar las operaciones que se necesitan:

- Importación de módulos:

```
const express = require('express')
const cors = require('cors')
const dotenv = require('dotenv')
dotenv.config()
const helmet = require('helmet')
const { Sequelize } = require('sequelize')
```

- `express`: un marco de aplicaciones web que permite crear rutas y manejar solicitudes HTTP.
- `cors`: un paquete que proporciona middleware para permitir solicitudes cruzadas entre dominios.
- `dotenv`: un paquete que permite cargar variables de entorno desde un archivo `.env`.
- `helmet`: un paquete que ayuda a proteger la aplicación mediante la configuración de encabezados HTTP.
- `Sequelize`: un ORM (Object-Relational Mapping) para trabajar con bases de datos relacionales, como MySQL o PostgreSQL.

La importación de estos módulos es necesaria para utilizar sus funciones y características en la aplicación.

- Configurar la aplicación Express.js y algunos middlewares para analizar las solicitudes como objetos JSON, habilitar Cross-Origin Resource Sharing (cors) o seguridad (helmet)

```
const app = express()
app.use(express.json())
app.use(cors())
app.use(helmet({
  crossOriginResourcePolicy: false // to allow image loading from public folder
}))
```

- Configurar la conexión a la base de datos

```
// config/sequelize.js
const databaseHost = process.env.DATABASE_HOST
const databasePort = process.env.DATABASE_PORT
const databaseUsername = process.env.DATABASE_USERNAME
const databasePassword = process.env.DATABASE_PASSWORD
const databaseName = process.env.DATABASE_NAME

const sequelize = new Sequelize(databaseName, databaseUsername, databasePassword, {
  host: databaseHost,
  port: databasePort,
  dialect: 'mariadb'
})
```

- Conexión a la base de datos y si se realiza correctamente, inicie la aplicación Express (servidor http):

```
sequelize.authenticate()
  .then(() => {
    console.info('INFO - Database connected.')
    const port = process.env.APP_PORT
    return app.listen(port)
  })
  .then((server) => {
    console.log('Deliverus listening at http://localhost:' + server.address().port)
  })
  .catch(err => {
    console.error('ERROR - Unable to connect to the database:', err)
  })
```

- Ejecuta backend.js abriendo una terminal (Ctrl+Shift+`) y ejecutando `npm install` (si no lo ha hecho anteriormente) y `npm start`, y verifique el registro de la terminal. Este comando lanzará `nodemon backend.js`, según se define en `package.json`. Al usar nodemon, cada vez que cambie y guarde algún archivo de su proyecto, lo detendrá y lo ejecutará de nuevo, por lo que es muy adecuado para fines de desarrollo.

Aparecerá algo como:

```
[nodemon] starting `node backend.js` Executing (default): SELECT 1+1 AS result
INFO - Database connected. Deliverus escuchando en http://localhost:3000
```

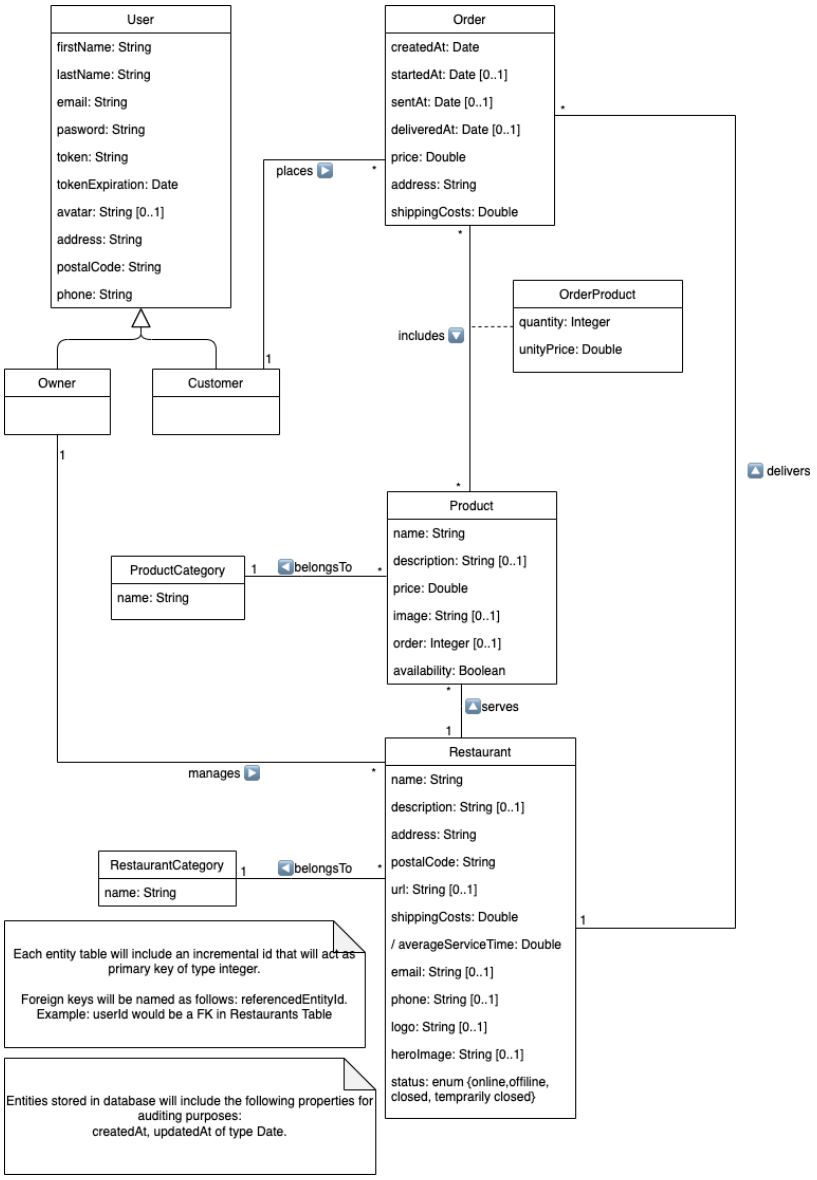
Alternativamente, puede ejecutar y depurar su proyecto utilizando la herramienta *Ejecutar y depurar* de VSCode. Se puede encontrar en la barra lateral izquierda o escribiendo `shift+ctrl+D`, y seleccionando `Run Script: start` en la lista desplegable. Agregue un punto de interrupción en las líneas 33 y 36 de backend.js, y haga clic en el icono de reproducción en la herramienta *Ejecutar y depurar* para depurar este archivo. Inspeccione las variables `server` y `error` respectivamente.

4. Migraciones

Tenga en cuenta los requisitos descritos en: https://github.com/IISSI2-IS-2022-2023/DeliverUS-Backend-2022-2023/blob/main/README.md

> Las migraciones son como las instrucciones que seguimos para construir una casa. Imagina que quieres construir una casa nueva. Primero, necesitas un plano para la casa, con todas las habitaciones que quieres y cómo quieres que se vea. Ese plano es como nuestro esquema de base de datos.
>
> Luego, cuando empezamos a construir la casa, necesitamos seguir un orden para construir cada parte de la casa, desde los cimientos hasta el techo. En el caso de la base de datos, las migraciones son como los pasos que seguimos para crear las tablas y los campos necesarios para almacenar información en la base de datos.
>
> Cada migración tiene dos partes: el método "up" y el método "down". El método "up" nos dice cómo crear la tabla o el campo nuevo en la base de datos. Y el método "down" nos dice cómo deshacer ese cambio si lo necesitamos.
>
> En este caso, tenemos migraciones para cada parte importante de nuestro sistema, como Usuarios, Restaurantes, Productos y Pedidos. Pero, aún nos falta crear la migración para la entidad Restaurante.

Y este es el diagrama de entidad propuesto:



Las migraciones son una herramienta poderosa para realizar un seguimiento de su esquema de base de datos y sus estados. Durante este tema, los usaremos para crear nuestro esquema de base de datos. Tenga en cuenta que puede encontrar una

migración para cada entidad: Usuario, Restaurante, Producto, Pedido (y Categoría de Producto + Categoría de Restaurante).

Cada migración tiene dos métodos: `up` y `down`, que dictan cómo realizar la migración y deshacerla.

Para nuestros propósitos, el método `up` incluirá la creación de cada tabla y sus campos, definiendo PrimaryKey y ForeignKeys.

Encontrará archivos de migración completados para todas las entidades, excepto Restaurante.

4.1. Completa la migración Create Restaurant

Completa el código del archivo `migrations\create_restaurant.js` para incluir las propiedades de la entidad Restaurante (es obligatorio nombrarlas como se muestra en el diagrama de entidad, específicamente: name, description, address, postalCode, url, restaurantCategoryId, shippingCosts, email, logo, phone, createdAt, updatedAt, userId, status). Consulte la documentación de Sequelize para Migraciones Skeleton y DataTypes; alternativamente, puede verificar la migración de Producto para ver ejemplos.

Tenga en cuenta que las relaciones se implementan mediante el uso de claves externas. Verifique las relaciones del Restaurante y defina las propiedades de la clave externa y cómo se están referenciando las tablas relacionadas. Por ejemplo, un Restaurante está relacionado con RestarantCategory, por lo que es posible que tenga que definir la siguiente clave externa:

```
restaurantCategoryId: {
  type: Sequelize.INTEGER,
  references: {
    model: {
      tableName: 'RestaurantCategories'
    },
    key: 'id'
  }
}
```

SOLUCIÓN:
Comenzamos con este código:

```
'use strict'
module.exports = {
  up: async (queryInterface, Sequelize) => {
    await queryInterface.createTable('Restaurants', {
      id: {
        allowNull: false,
        autoIncrement: true,
        primaryKey: true,
        type: Sequelize.INTEGER
      },
      //TODO: Include the rest of the fields of the Restaurants table
    })
  },
  down: async (queryInterface, Sequelize) => {
    await queryInterface.dropTable('Restaurants')
  }
}
```

Y el //TODO se haría de la siguiente forma:

```
'use strict'
module.exports = {
  // Método up para crear la tabla Restaurants y definir sus columnas
  up: async (queryInterface, Sequelize) => {
    await queryInterface.createTable('Restaurants', {
      // Columna de identificación de restaurantes
      id: {
        allowNull: false,
        autoIncrement: true,
        primaryKey: true,
        type: Sequelize.INTEGER
      },
      // Columna de nombre de restaurante (requerida)
      name: {
        allowNull: false,
        type: Sequelize.STRING
      },
```

```
// Columna de descripción de restaurante
description: {
  type: Sequelize.TEXT
},
// Columna de dirección de restaurante (requerida)
address: {
  allowNull: false,
  type: Sequelize.STRING
},
// Columna de código postal de restaurante (requerida)
postalCode: {
  allowNull: false,
  type: Sequelize.STRING
},
// Columna de URL de restaurante
url: {
  type: Sequelize.STRING
},
// Columna de costo de envío de restaurante (requerida)
shippingCosts: {
  allowNull: false,
  defaultValue: 0.0,
  type: Sequelize.DOUBLE
},
```

```
// Columna de tiempo promedio de servicio de restaurante
averageServiceMinutes: {
  allowNull: true,
  type: Sequelize.DOUBLE
},
// Columna de correo electrónico de restaurante
email: {
  type: Sequelize.STRING
},
```

```
// Columna de correo electrónico de restaurante
email: {
  type: Sequelize.STRING
},
// Columna de número de teléfono de restaurante
phone: {
  type: Sequelize.STRING
},
// Columna de logo de restaurante
logo: {
  type: Sequelize.STRING
},
// Columna de imagen principal de restaurante
heroImage: {
  type: Sequelize.STRING
},
// Columna de estado de restaurante con valores predefinidos (por defecto 'offline')
status: {
  type: Sequelize.ENUM,
  values: [
    'online',
    'offline',
    'closed',
    'temporarily closed'
  ],
  defaultValue: 'offline'
},
```

```
  // Columna de fecha de última actualización de restaurante (por defecto la fecha actual)
  updatedAt: {
    allowNull: false,
    type: Sequelize.DATE,
    defaultValue: new Date()
  },
```

```
    // Columna de identificación de usuario asociado al restaurante (requerida)
    userId: {
      allowNull: false,
      type: Sequelize.INTEGER,
      onDelete: 'CASCADE',
      references: {
        model: {
          tableName: 'Users'
        },
        key: 'id'
      }
    },
    // Columna de identificación de categoría de restaurante asociada al restaurante (requeri
    restaurantCategoryId: {
      allowNull: false,
      type: Sequelize.INTEGER,
      references: {
        model: {
          tableName: 'RestaurantCategories'
        },
        key: 'id'
      }
    }
  })
},
```

```
  // Método down para eliminar la tabla Restaurants
  down: async (queryInterface, Sequelize) => {
    await queryInterface.dropTable('Restaurants')
  }
}
```

Una vez que haya completado la migración de la tabla de Restaurantes, debe ejecutar las migraciones. Para este fin, hay un binario de la interfaz de línea de comandos (CLI) disponible (llamado `sequelize-cli`). Utiliza los detalles de conexión de la base de datos que se encuentran en `config\config.js`.

Para ejecutar migraciones, ejecútelos usando npx (herramienta para ejecutar binarios de paquetes npm) en la terminal:

```
npx sequelize-cli db:migrate
```

Después de hacer esto, debería encontrar las tablas creadas en su mariadb.

Para deshacer las migraciones, puede ejecutar:

```
npx sequelize-cli db:migrate:undo:all
```

Más información sobre migraciones se puede encontrar en:
https://sequelize.org/master/manual/migrations.html

5. Seeders

Los archivos Seed se utilizan para poblar las tablas de la base de datos con datos de muestra o de prueba. Puedes encontrarlos en la carpeta "seeders". Ten en cuenta que "restaurants_seeder.js" supone un nombre determinado para los campos de la tabla de restaurantes.

Puedes ejecutar tus seeders para poblar la base de datos ejecutando:

```
npx sequelize-cli db:seed:all
```

Y puedes deshacerlos ejecutando:

```
npx sequelize-cli db:seed:undo:all
```

Puedes encontrar más información sobre los seeders en:
https://sequelize.org/master/manual/migrations.html#creating-the-first-seed

Si realizas algún cambio en las migraciones o en los seeders, puedes actualizar la base de datos ejecutando los comandos "**undo migrations**", "**run migrations**" y "**run seeders**" todos juntos utilizando la tarea "**Rebuild database**". Para ejecutar esta tarea, ejecuta el comando "**Run Task**" y selecciona "**Rebuild database**". Puedes ver la definición del comando asociado con esta tarea en el archivo "**.vscode/tasks.json".**

6. Modelos

El mapeo objeto-relacional (ORM) es una técnica de programación de software para vincular objetos de lógica empresarial a fuentes de datos, de modo que los programadores puedan trabajar directamente con objetos de alto nivel para realizar operaciones de base de datos sin problemas. Por lo general, los objetos relacionados con las entidades de la base de datos se llaman **"modelos"** y trabajamos con ellos para interactuar con sus entidades de base de datos correspondientes para realizar operaciones CRUD (crear, leer, actualizar y eliminar) estándar. Al usar herramientas ORM, se proporcionan las siguientes operaciones: crear, findAll, actualizar y eliminar (entre otras).

Sequelize es una herramienta de mapeo objeto-relacional de Node.js que proporciona todas las herramientas necesarias para establecer conexiones con la base de datos (como se explica en la sección 3), ejecutar migraciones y seeders (secciones 4 y 5), definir modelos y realizar operaciones.

Puedes encontrar las definiciones de los modelos para todas las entidades en la carpeta "models". Cada modelo es una clase llamada como su tabla correspondiente (pero en singular) y extiende la clase Model de Sequelize.

6.1. Modelo completo de Restaurante

Completa el código del archivo "models\restaurant.js" para incluir todas las propiedades que coincidan con los campos correspondientes de la tabla de Restaurantes.

Ten en cuenta que también hemos definido las relaciones entre los modelos. Por ejemplo, el modelo de Restaurante está relacionado con RestaurantCategory, User, Product y Order. Para definir estas relaciones, debemos incluir el siguiente método:

```
static associate (models) {
    // define association here
    Restaurant.belongsTo(models.RestaurantCategory, { foreignKey: 'restaurantCategoryId', as: 'restaurantCategory' })
    Restaurant.belongsTo(models.User, { foreignKey: 'userId', as: 'user' })
    Restaurant.hasMany(models.Product, { foreignKey: 'restaurantId', as: 'products' })
    Restaurant.hasMany(models.Order, { foreignKey: 'restaurantId', as: 'orders' })
  }
```

En el otro lado de la relación, tienes que incluir la relación opuesta. Por ejemplo, puedes encontrar que un Producto *pertenece a* un Restaurante, o que una Categoría de Restaurante *tiene muchos* Restaurantes.

Finalmente, puedes definir métodos que realicen cálculos sobre el modelo. Por ejemplo, en el modelo de Restaurante, puedes encontrar un método que calcule y devuelva el tiempo de servicio promedio de un restaurante.

Lo que me dan inicialmente es:

```
'use strict'
const moment = require('moment')
const {
  Model
} = require('sequelize')

module.exports = (sequelize, DataTypes) => {
  class Restaurant extends Model {
    /**
     * Helper method for defining associations.
     * This method is not a part of Sequelize lifecycle.
     * The `models/index` file will call this method automatically.
     */
    static associate (models) {
      // define association here
      Restaurant.belongsTo(models.RestaurantCategory, { foreignKey: 'restaurantCategoryId', as: 'restaurantCategory' })
      Restaurant.belongsTo(models.User, { foreignKey: 'userId', as: 'user' })
      Restaurant.hasMany(models.Product, { foreignKey: 'restaurantId', as: 'products' })
      Restaurant.hasMany(models.Order, { foreignKey: 'restaurantId', as: 'orders' })
    }

    async getAverageServiceTime () {
      try {
        const orders = await this.getOrders()
        const serviceTimes = orders.filter(o => o.deliveredAt).map(o => moment(o.deliveredAt).diff(moment(o.createdAt), 'minutes'))
        return serviceTimes.reduce((acc, serviceTime) => acc + serviceTime, 0) / serviceTimes.length
      } catch (err) {
        return err
      }
    }
  }
```

```
  Restaurant.init({
  //TODO: Include the rest of the properties of the Restaurant model

  }, {
    sequelize,
    modelName: 'Restaurant'
  })
  return Restaurant
}
```

7. Prueba de migraciones, sembradoras y modelo de Restaurante

Para hacer una prueba mínima, hemos incluido el siguiente código en `controllers/RestaurantController.js` y `routes/RestaurantRoutes.js` (abordaremos los detalles de la implementación de rutas y controladores en el próximo laboratorio, *se ve más adelante en este documento*)

```
// RestaurantController.js
const models = require('../models')
const Restaurant = models.Restaurant
const RestaurantCategory = models.RestaurantCategory

exports.index = async function (req, res) {
  try {
    const restaurants = await Restaurant.findAll(
      {
        attributes: ['id', 'name', 'description', 'address', 'postalCode', 'url', 'shippingCosts', 'averageServiceMinutes', 'email',
        include:
      {
        model: RestaurantCategory,
        as: 'restaurantCategory'
      },
        order: [[{ model: RestaurantCategory, as: 'restaurantCategory' }, 'name', 'ASC']]
      }
    )
    res.json(restaurants)
  } catch (err) {
    res.status(500).send(err)
  }
}
```

Observa que la función `indexRestaurants` realiza una consulta al modelo para recuperar todos los restaurantes de la base de datos, ordenados por Categoría de

Restaurante, y los devuelve como un documento JSON. A continuación, definimos el punto final `/restaurants` que responde a las solicitudes utilizando la función `indexRestaurants`.

Para cargar todas las rutas definidas en la carpeta `routes`, hemos incluido este código en `backend.js`:

```
const requireOptions = { app } require('./routes/')(requireOptions)
```

Abre la extensión ThunderClient (https://www.thunderclient.io/), y recarga las colecciones haciendo clic en Colecciones → menú ≡ → recargar. Las colecciones se almacenan en `example_api_client`.

Haz clic en la carpeta de Restaurantes y encontrarás una simple solicitud GET ALL. Ejecuta la solicitud, debería devolver un *Código de estado HTTP 200 OK* y un JSON con la información de los Restaurantes.

2. LAB 2 - Backend routing controllers / rutas y controladores

Introducción

Aprenderemos cómo definir e implementar los Endpoints de nuestro backend y cómo los controladores (del patrón MVC) manejan los endpoints. Los controladores ejecutan algunas partes de la lógica de negocio de nuestra aplicación.

En segundo lugar, aprenderemos cómo una herramienta de software ORM (el paquete *Sequelize*) nos ayudará a realizar operaciones en la base de datos MariaDB desde estos controladores.

Ejercicios

1. Aceptar la tarea de GitHub Classroom y clonar

Seguimos trabajando sobre el mismo proyecto de antes. Si queremos clonar, lo que habría que hacer es repetir los pasos del lab 1.

Recuerda hacer las migraciones que ya fueron explicadas.

2. Rutas

El software backend puede publicar sus funcionalidades a través de servicios RESTFul. Estos servicios siguen los patrones arquitectónicos del protocolo HTTP. Las funcionalidades de DeliverUS se explican en https://github.com/IISSI2-IS-2022-2023/DeliverUS-Backend-2022-2023/blob/main/README.md#functional-requirements

Como ejemplo, si el sistema proporciona operaciones CRUD sobre una entidad, debe haber un punto final para cada operación. El punto final HTTP POST para Crear, HTTP GET para Leer, HTTP PUT|PATCH para Actualizar y HTTP DELETE para Eliminar.

Las rutas generalmente se crean siguiendo algunos patrones comunes y buenas prácticas. Por ejemplo, para las operaciones CRUD en la entidad Restaurante:

- `HTTP POST /restaurantes` para Crear un restaurante. El método del controlador generalmente se llama `create`
- `HTTP GET /restaurantes` para Leer todos los restaurantes. El método del controlador generalmente se llama `index`
- `HTTP GET /restaurantes/{restaurantId}` para Leer detalles del restaurante con id=restaurantId (un parámetro de ruta). El método del controlador generalmente se llama `show`
- `HTTP PUT /restaurantes/{restaurantId}` para Actualizar detalles del restaurante con id=restaurantId (un parámetro de ruta). El método del controlador generalmente se llama `update`
- `HTTP DELETE /restaurantes/{restaurantId}` para Eliminar el restaurante con id=restaurantId (un parámetro de ruta). El método del controlador generalmente se llama `destroy`

Además, un punto final puede definir algunos parámetros de consulta. Estos generalmente se destinan a incluir algunos parámetros opcionales en la solicitud, como implementar una búsqueda en la entidad. Por ejemplo, si queremos consultar los pedidos filtrados por estado, se debe definir un parámetro de consulta `estado`.

3.1. Definición de rutas de Restaurant

Para definir rutas en una aplicación *Express Node.js*, debemos seguir la siguiente plantilla:

```
app.route('/path') //the endpoint path
    .get( //the http verb that we want to be available at the previous path
      EntityController.index) // the function that will attend requests for that http verb and that path
    .post( //we can chain more http verbs for the same endpoint
      EntityController.create) // the function that will attend requests for that http verb and that path
```

El proyecto DeliverUS organiza sus rutas en la carpeta `routes`. Definimos rutas para cada entidad en su propio archivo. Por ejemplo, las rutas de los restaurantes se definirán en el archivo `RestaurantRoutes.js`.

Complete el archivo `RestaurantRoutes.js` para definir los puntos finales para los siguientes requisitos funcionales:

- Requisitos funcionales de cliente:
  - FR1: Listado de restaurantes: Los clientes podrán consultar todos los restaurantes.
  - FR2: Detalles de restaurantes y menú: Los clientes podrán consultar los detalles de los restaurantes y los productos que ofrecen.
- Requisitos funcionales del propietario:
  - FR1: Crear, Leer, Actualizar y Eliminar (CRUD) restaurantes: los restaurantes están relacionados
  - FR3: Listar los pedidos de un restaurante. Un propietario podrá inspeccionar los pedidos de cualquiera de los restaurantes que posee. El pedido debe incluir los productos relacionados.
  - FR5: Mostrar un panel de control que incluya algunos análisis de negocios: #PedidosDeAyer, #PedidosPendientes, #PedidosAtendidosHoy, #FacturadoHoy (€). Tenga en cuenta que la función controladora que atiende esta solicitud es `OrderController.analytics`).

En un principio, lo que tengo es lo siguiente:

```
'use strict'
const RestaurantValidation = require('../controllers/validation/RestaurantValidation')
const RestaurantController = require('../controllers/RestaurantController')
const OrderController = require('../controllers/OrderController')
const ProductController = require('../controllers/ProductController')
const multer = require('multer')
const fs = require('fs')
const Restaurant = require('../models').Restaurant

const storage = multer.diskStorage({
  destination: function (req, file, cb) {
    fs.mkdirSync(process.env.RESTAURANTS_FOLDER, { recursive: true })
    cb(null, process.env.RESTAURANTS_FOLDER)
  },
  filename: function (req, file, cb) {
    cb(null, Math.random().toString(36).substring(7) + '-' + Date.now() + '.' + file.originalname.split('.').pop())
  }
})

const upload = multer({ storage }).fields([
  { name: 'logo', maxCount: 1 },
  { name: 'heroImage', maxCount: 1 }
])

module.exports = (options) => {
  const app = options.app

  // TODO: Include routes for restaurant described in the lab session wiki page.

}
```

Se completará parcialmente, a falta de los middleware, los cuales se explicarán en el siguiente lab.

Y la solución queda de la siguiente manera:

```
'use strict'
const RestaurantValidation = require('../controllers/validation/RestaurantValidation')
const RestaurantController = require('../controllers/RestaurantController')
const OrderController = require('../controllers/OrderController')
const ProductController = require('../controllers/ProductController')
const multer = require('multer')
const fs = require('fs')
const Restaurant = require('../models').Restaurant

const storage = multer.diskStorage({
  destination: function (req, file, cb) {
    fs.mkdirSync(process.env.RESTAURANTS_FOLDER, { recursive: true })
    cb(null, process.env.RESTAURANTS_FOLDER)
  },
  filename: function (req, file, cb) {
    cb(null, Math.random().toString(36).substring(7) + '-' + Date.now() + '.' + file.originalname.split('.').pop())
  }
})
```

```
const upload = multer({ storage }).fields([
  { name: 'logo', maxCount: 1 },
  { name: 'heroImage', maxCount: 1 }
])

module.exports = (options) => {
  const app = options.app
  const middlewares = options.middlewares

  app.route('/restaurants')
    .get(
      RestaurantController.index)
    .post(
      // TODO: Add needed middlewares
      RestaurantController.create)

  app.route('/restaurants/:restaurantId')
    .get(RestaurantController.show)
    .put(
      // TODO: Add needed middlewares
      RestaurantController.update)
    .delete(
      // TODO: Add needed middlewares
      RestaurantController.destroy)

  app.route('/restaurants/:restaurantId/orders')
    .get(
      // TODO: Add needed middlewares
      OrderController.indexRestaurant)

  app.route('/restaurants/:restaurantId/products')
    .get(
      // TODO: Add needed middlewares
      ProductController.indexRestaurant)

  app.route('/restaurants/:restaurantId/analytics')
    .get(
      // TODO: Add needed middlewares
      OrderController.analytics)
}
```

4. Controladores.

Los controladores son los componentes principales de la capa de lógica empresarial. Las funcionalidades y reglas de negocio pueden implementarse en los controladores especialmente según el patrón arquitectónico MVC. El proyecto DeliverUS organiza sus controladores en la carpeta `controllers`. Definimos controladores para la lógica empresarial relacionada con cada entidad en su propio archivo. Por ejemplo, el controlador de restaurante se definirá en el archivo `RestaurantController.js`.

Cada método de controlador recibe un objeto de solicitud `req` y un objeto de respuesta `res`. El objeto de solicitud *representa la solicitud HTTP y tiene propiedades para la cadena de consulta de la solicitud, los parámetros, el cuerpo, las cabeceras HTTP, y así sucesivamente* (consulte https://expressjs.com/en/4x/api.html#req para obtener más detalles).

En nuestro proyecto necesitaremos los siguientes atributos de la solicitud:

- `req.body` representa los datos que provienen del cliente (generalmente un documento JSON o un formulario enviado como multipart/form-data cuando se necesitan archivos).
- `req.params` representa los parámetros de la ruta. Por ejemplo, si definimos un parámetro `:restaurantId`, tendremos acceso a él mediante `req.params.restaurantId`.
- `req.file` o `req.files` representa los archivos adjuntos a una solicitud multipart/form-data. Por ejemplo, `req.files.logo` daría acceso a un parámetro de archivo llamado `logo`.
- `req.query` representa los parámetros de consulta. Por ejemplo, si una solicitud incluye un parámetro de consulta `estado`, se accederá mediante `req.query.estado`.
- `req.user` representa el usuario que inició sesión que hizo la solicitud. Aprenderemos más sobre esto en el laboratorio 3.

El objeto de respuesta *representa la respuesta HTTP que envía una aplicación Express cuando recibe una solicitud HTTP* (consulte https://expressjs.com/en/4x/api.html#res.

En nuestro proyecto necesitaremos los siguientes métodos del objeto `res`:

- `res.json(entityObject)` devuelve el objeto `entityObject` al cliente como un documento JSON con el código de estado HTTP 200. Por ejemplo, `res.json(restaurant)` devolverá el objeto de restaurante como documento json.
- `res.json(message)` devuelve una cadena de caracteres `message` al cliente como un documento JSON con el código de estado HTTP 200.
- `res.status(500).send(err)` devuelve el objeto `err` (que típicamente incluye algún tipo de mensaje de error) y un código de estado HTTP 500 al cliente.

Los códigos de estado HTTP que se utilizarán en este proyecto son:

- `200`. La solicitud se atendió correctamente.
- `401`. Credenciales incorrectas.
- `403`. Solicitud prohibida (no hay suficientes privilegios).
- `404`. El recurso solicitado no se encontró.
- `422`. Error de validación.
- `500`. Error general.

Para obtener más información sobre los códigos de estado HTTP, consulte: https://developer.mozilla.org/es/docs/Web/HTTP/Status

En un principio, tenemos lo siguiente:

```
1  'use strict'
2  const models = require('../models')
3  const Restaurant = models.Restaurant
4  const Product = models.Product
5  const RestaurantCategory = models.RestaurantCategory
6  const ProductCategory = models.ProductCategory
7
```

```
8   //TODO: Complete the following functions
9   exports.index = async function (req, res) {
10
11  }
12
13  exports.indexOwner = async function (req, res) {
14
15  }
16
17  exports.create = async function (req, res) {
18
19  }
20
21  exports.show = async function (req, res) {
22
23  }
24
25  exports.update = async function (req, res) {
26  }
27
28  exports.destroy = async function (req, res) {
29
30  }
```

Vamos a ver, paso a paso cómo completar los //TODO y el motivo de cada cosa.

4.1. Métodos del controlador de Restaurantes.

4.1.1 Método Create para crear una entidad.

Típicamente, esperamos que el cuerpo de la solicitud incluya un documento JSON con toda la información necesaria para crear un nuevo elemento de la entidad. Para acceder a este documento, usamos el atributo `req.body`.

Sequelize ofrece una forma de crear nuevos elementos, el método `Model.build` que recibe un objeto JSON que incluye los campos necesarios para construir un nuevo elemento y luego un método `Model.save` para almacenarlo en la tabla de base de datos correspondiente.

Para el controlador de Restaurantes podemos hacer esto usando los siguientes fragmentos de código:

```
const newRestaurant = Restaurant.build(req.body)
```

Luego, tenemos que incluir al dueño que inició sesión como el userId del restaurante. Tenga en cuenta que hasta el próximo laboratorio, el sistema no implementará la autenticación, por lo que la siguiente línea NO se incluirá durante este laboratorio, por lo que el restaurante no estará relacionado con ningún dueño. Arreglaremos esto en el próximo laboratorio.

```
// newRestaurant.userId = req.user.id // usuario autenticado
```

A continuación, debemos comprobar si los archivos están presentes. Cuando creamos un restaurante, podemos incluir una imagen de portada (heroImage) y un logotipo (logoImage). Para ello, comprobaremos si cada archivo está presente y, si es así, almacenaremos la ruta del archivo en el campo correspondiente de la tabla de la base de datos.

```
if (typeof req.files?.heroImage !== 'undefined') {
  newRestaurant.heroImage = req.files.heroImage[0].destination + '/' + req.files.heroImage[0].filename
}
if (typeof req.files?.logo !== 'undefined') {
  newRestaurant.logo = req.files.logo[0].destination + '/' + req.files.logo[0].filename
}
```

Finalmente, podemos guardar el restaurante creado. Nuestra variable `newRestaurant` está construida, pero no se ha guardado. Para ello, Sequelize ofrece un método `save` para cada modelo. Dado que esta es una operación de entrada/salida, no queremos bloquear el sistema, por lo que el método `save` devuelve una promesa. Utilizaremos la sintaxis await/async para hacer que nuestro código sea más legible. Podemos utilizar el siguiente fragmento de código:

```
try {
  const restaurant = await newRestaurant.save()
  res.json(restaurant)
} catch (err) {
    res.status(500).send(err)
}
```

Finalmente, la función queda de la siguiente manera:

```
exports.create = async function (req, res) {
  // Crear un nuevo objeto "Restaurant" con los datos recibidos en el cuerpo de la petición
  const newRestaurant = Restaurant.build(req.body)
  // Asignar al atributo "userId" del objeto creado, el id del usuario actualmente autenticado
  newRestaurant.userId = req.user.id

  // Si se ha enviado una imagen para el campo "heroImage", se guarda la ruta completa en el atributo correspondiente del objeto "newRestaurant"
  if (typeof req.files?.heroImage !== 'undefined') {
    newRestaurant.heroImage = req.files.heroImage[0].destination + '/' + req.files.heroImage[0].filename
  }
  // Si se ha enviado una imagen para el campo "logo", se guarda la ruta completa en el atributo correspondiente del objeto "newRestaurant"
  if (typeof req.files?.logo !== 'undefined') {
    newRestaurant.logo = req.files.logo[0].destination + '/' + req.files.logo[0].filename
  }
  try {
    // Guardar el objeto "newRestaurant" en la base de datos
    const restaurant = await newRestaurant.save()
    // Devolver como respuesta el objeto "restaurant" creado
    res.json(restaurant)
  } catch (err) {
      // Si se produce un error al guardar el objeto en la base de datos, devolver un mensaje de error con el código de estado HTTP 500
      res.status(500).send(err)
  }
}
```

4.1.2 Métodos de índice para leer entidades.

Implementar el FR1: Listado de restaurantes: Los clientes podrán consultar todos los restaurantes. Para ello, se puede utilizar el método `Model.findAll` de Sequelize:

```
exports.index = async function (req, res) {
  try {
    // Se utiliza el método `findAll` de Sequelize en el modelo `Restaurant`
    //para buscar todos los restaurantes almacenados en la base de datos.
    //La consulta se realiza de forma asíncrona (`await`) y se guarda el
    //resultado en una constante `restaurants`.
    const restaurants = await Restaurant.findAll(
      {
        // Se especifican los atributos de los objetos `Restaurant` que se
        //quieren devolver en la respuesta, así como la relación.
        //`RestaurantCategory` que se incluirá para cada objeto `Restaurant`
        // Además, se ordenan los resultados por el nombre de la categoría
        //de restaurante de manera ascendente.
        attributes: ['id', 'name', 'description', 'address', 'postalCode', 'url', 'shippingCosts', 'averageServiceMinutes',
                     'email', 'phone', 'logo', 'heroImage', 'status', 'restaurantCategoryId'],
        include:
        {
          model: RestaurantCategory, // Se especifica el modelo de la relación
          //que se quiere incluir (`RestaurantCategory`).
          as: 'restaurantCategory' // Se especifica el alias que se usará para acceder
                                   //a los objetos `RestaurantCategory` desde los objetos `Restaurant`.
        },
        order: [[{ model: RestaurantCategory, as: 'restaurantCategory' }, 'name', 'ASC']]
        // Se especifica el ordenamiento de los resultados en base a la categoría de restaurante de manera ascendente.
      }
    );
    // Se devuelve una respuesta HTTP con el resultado de la consulta en formato JSON utilizando el método
    //`json()` del objeto `res`.
    res.json(restaurants);
  } catch (err) {
    // En caso de que se produzca un error durante la ejecución del bloque `try`, se captura el error y
    //se envía una respuesta HTTP con el estado 500 y un mensaje de error utilizando el método `status()` y `send()` del objeto `res`.
    res.status(500).send(err);
  }
};
```

Implementar el FR3: Lista de pedidos de un restaurante. Un propietario podrá inspeccionar los pedidos de cualquiera de los restaurantes de su propiedad. El pedido debe incluir los productos relacionados. Para ello, se puede utilizar el método `Model.findAll` de Sequelize, incluyendo una cláusula where.

```
exports.indexOwner = async function (req, res) {
  try {
    // Se utiliza el método `findAll` de Sequelize en el modelo `Restaurant` para buscar todos los restaurantes almacenados
    //en la base de datos que pertenezcan al usuario que hizo la solicitud. La consulta se realiza de forma asíncrona
    //(`await`) y se guarda el resultado en una constante `restaurants`.
    const restaurants = await Restaurant.findAll(
      {
        // Se especifican los atributos de los objetos `Restaurant` que se quieren devolver en la respuesta y se agrega
        //una condición para filtrar los restaurantes que pertenecen al usuario que hizo la solicitud.
        attributes: ['id', 'name', 'description', 'address', 'postalCode', 'url', 'shippingCosts', 'averageServiceMinutes',
                     'email', 'phone', 'logo', 'heroImage', 'status', 'restaurantCategoryId'],
        where: { userId: req.user.id }
      });
    // Se devuelve una respuesta HTTP con el resultado de la consulta en formato JSON utilizando el método `json()` del objeto `res`.
    res.json(restaurants);
  } catch (err) {
    // En caso de que se produzca un error durante la ejecución del bloque `try`, se captura el error y se envía una respuesta
    //HTTP con el estado 500 y un mensaje de error utilizando el método `status()` y `send()` del objeto `res`.
    res.status(500).send(err);
  }
};
```

4.1.3 Métodos de visualización para devolver detalles de la entidad.

Implementar el FR2: Detalles del restaurante y menú: Los clientes podrán consultar los detalles de los restaurantes y los productos que ofrecen.

Para ello, se recibirá un `req.params.restaurantId` que identificará el restaurante. Se puede utilizar el método `Model.findByPk` de Sequelize.

Tenga en cuenta que se necesitará incluir sus productos y su categoría de restaurante. Recuerde que los productos deben ordenarse según el valor del campo de orden. Se puede utilizar el siguiente fragmento de código para realizar la consulta:

```
const restaurant = await Restaurant.findByPk(req.params.restaurantId, {
  attributes: { exclude: ['userId'] },
  include: [{
    model: Product,
    as: 'products',
    include: { model: ProductCategory, as: 'productCategory' }
  },
  {
    model: RestaurantCategory,
    as: 'restaurantCategory'
  }],
  order: [[{model:Product, as: 'products'}, 'order', 'ASC']],
}
)
```

A continuación, devolver el restaurante mediante el método `res.json()` que recibe el objeto que se va a devolver. Rodear este código con el try y catch correspondiente. En caso de que se produzca una excepción, se debe devolver el código de estado HTTP 500 en el bloque catch utilizando los métodos `res.status(httpCode).send(error)`.

Finalmente, el código queda:

```
exports.show = async function (req, res) {
  // Se devuelve únicamente la información pública del restaurante.
  try {
    // Se utiliza el método `findByPk` de Sequelize en el modelo `Restaurant` para buscar un restaurante en
    //particular en la base de datos a partir del valor del parámetro `restaurantId` de la solicitud.
    const restaurant = await Restaurant.findByPk(req.params.restaurantId, {
      // Se especifican los atributos que se quieren devolver en la respuesta. En este caso, se excluye el
      //atributo `userId` del objeto `Restaurant`.
      attributes: { exclude: ['userId'] },
      // Se incluyen los modelos `Product` y `RestaurantCategory` como objetos anidados para que también se
      //devuelvan sus atributos.
      include: [
        {
          model: Product,
          as: 'products',
          // Se incluye el modelo `ProductCategory` como objeto anidado de `Product` para que también se
          //devuelvan sus atributos.
          include: { model: ProductCategory, as: 'productCategory' }
        },
        {
          model: RestaurantCategory,
          as: 'restaurantCategory'
        }
      ],
      // La consulta se ordena por el atributo `order` del modelo `Product` de forma ascendente.
      order: [[{model:Product, as: 'products'}, 'order', 'ASC']]
    });
    // Se devuelve una respuesta HTTP con el resultado de la consulta en formato JSON utilizando el método `json()` del objeto `res`.
    res.json(restaurant);
  } catch (err) {
    // En caso de que se produzca un error durante la ejecución del bloque `try`, se captura el error y se envía una respuesta HTTP
    //con el estado 500 y un mensaje de error utilizando el método `status()` y `send()` del objeto `res`.
    res.status(500).send(err);
  }
};
```

4.1.4 Método de actualización para modificar la entidad.

Al igual que en la creación, comprobar si hay archivos presentes. Luego, utilizar el método `Model.update`. En caso de éxito, se debe devolver el elemento de restaurante actualizado consultando la base de datos (utilizando el método `findByPk`) después de la actualización.

Este método sigue los mismos pasos que al crear un restaurante, de manera que ueda lo siguiente:

```
exports.update = async function (req, res) {
  // Si existe la propiedad "heroImage" en el objeto "files" de la solicitud, se asigna la ruta de la
  //imagen al campo "heroImage" de la base de datos
  if (typeof req.files?.heroImage !== 'undefined') {
    req.body.heroImage = req.files.heroImage[0].destination + '/' + req.files.heroImage[0].filename
  }
  // Si existe la propiedad "logo" en el objeto "files" de la solicitud, se asigna la ruta de la
  //imagen al campo "logo" de la base de datos
  if (typeof req.files?.logo !== 'undefined') {
    req.body.logo = req.files.logo[0].destination + '/' + req.files.logo[0].filename
  }
  try {
    // Se actualiza el registro del restaurante correspondiente a través de la función "update"
    //de Sequelize, utilizando el objeto "req.body" que contiene los datos actualizados del restaurante
    await Restaurant.update(req.body, { where: { id: req.params.restaurantId } })
    // Se recupera el registro actualizado del restaurante para enviarlo en la respuesta
    const updatedRestaurant = await Restaurant.findByPk(req.params.restaurantId)
    res.json(updatedRestaurant)
  } catch (err) {
    // Si ocurre un error, se envía una respuesta con un estado HTTP 500 y el mensaje de error
    res.status(500).send(err)
  }
}
```

4.1.5 Método de eliminación para eliminar la entidad.

Utilice el método `Model.destroy`. Se debe especificar una cláusula where para eliminar sólo el restaurante identificado por `req.params.restaurantId`. Destroy devuelve el número de elementos destruidos. Devolver un mensaje informativo.

Este sería el código:

```
exports.destroy = async function (req, res) {
  try {
    // Se utiliza el método destroy de Sequelize para eliminar un restaurante de la base de datos
    const result = await Restaurant.destroy({ where: { id: req.params.restaurantId } })
    let message = ''
    if (result === 1) {
      // Si result es igual a 1, significa que se eliminó correctamente un restaurante con el id especificado
      message = 'Sucessfuly deleted restaurant id.' + req.params.restaurantId
    } else {
      // Si result no es igual a 1, significa que no se pudo eliminar el restaurante
      message = 'Could not delete restaurant.'
    }
    // Devuelve un mensaje de éxito o de error según el valor de result
    res.json(message)
  } catch (err) {
    // En caso de haber algún error, se devuelve un mensaje de error con un código de estado 500
    res.status(500).send(err)
  }
}
```

3. LAB 3 - Backend validation middleware

Introducción

Aprenderemos cómo definir e implementar el middleware de validación y otros en nuestras aplicaciones de backend Node.js. Los middlewares están destinados a ejecutar algunas partes específicas de nuestra lógica empresarial, como:

- Validación de datos de los clientes (el software que realiza operaciones contra el backend).
- Verificación de autorización.
- Verificación de permisos.
- Verificación de propiedad de recursos.

En segundo lugar, aprenderemos cómo un paquete de validación nos ayudará a realizar la validación de datos que vienen de los clientes.

Seguimos con el proyecto del laboratorio del apartado anterior.

1. Middlewares y middleware de validación.

Encontrarás middlewares en la carpeta middlewares. Uno para cada entidad, uno para verificar si un dato identifica un registro de una entidad dada en la base de datos, y otro para autenticación/autorización.

En el archivo **AuthMiddleware.js** encontrarás dos métodos:

- **isLoggedIn** verifica si el usuario ha iniciado sesión (la solicitud incluye un token de portador válido).
- **hasRole** recibe un array de nombres de roles y verifica si el usuario que ha iniciado sesión tiene el rol necesario.

En el archivo **EntityMiddleware.js** encontrarás un método:

- **checkEntityExists** verifica, para un id y una entidad dados, si existe un registro en la tabla correspondiente en la base de datos que coincida con dicho id, y en caso de que no exista, devuelve el código de estado HTTP 404.

En el archivo **ValidationHandlingMiddleware.js** encontrarás un método:

- **handleValidation** verifica el resultado de express-validator y si se encuentra un error, devuelve 422 (Error de validación) y detiene el procedimiento de validación.

A continuación, encontrarás un archivo de middleware para cada entidad. Dependiendo de la entidad y los requisitos funcionales, necesitaremos verificar si el usuario actualmente autenticado tiene suficientes privilegios para realizar la operación solicitada.

Por ejemplo, cuando un usuario envía una solicitud para crear un nuevo producto, necesitaremos verificar que:

- ➔ el usuario ha iniciado sesión
- ➔ el usuario tiene el rol de propietario (ya que los clientes no pueden crear productos)
- ➔ el producto pertenece a un restaurante que él/ella posee (los datos incluyen un restaurantId que pertenece al propietario que hace la solicitud)
- ➔ los datos del producto incluyen valores válidos para cada propiedad para que pueda ser creado de acuerdo con nuestros requisitos de información.

Además, si los datos pueden incluir archivos, encontrarás un middleware de carga que manejará esto.

Para verificar todos estos requisitos, debemos incluir cada método de middleware en la ruta correspondiente.

```
app.route('/products')
    .post(
      middlewares.isLoggedIn,
      middlewares.hasRole('owner'),
      upload,
      ProductValidation.create,
      middlewares.handleValidation,
      middlewares.checkProductRestaurantOwnership,
      ProductController.create
    )
```

1.1. Middlewares de validación

Los **middlewares de validación** están destinados a verificar si los datos que vienen en una solicitud cumplen con los requisitos de información. La mayoría de estos requisitos están definidos a nivel de base de datos y se incluyeron al crear el esquema en los archivos de migración. Algunos otros requisitos se verifican en la capa de aplicación.

Por ejemplo, si desea crear un nuevo restaurante, se pueden proporcionar algunas imágenes: imagen de logo e imagen de héroe. Estos archivos deben ser archivos de imagen y su tamaño debe ser inferior a 10 MB. Para verificar estos otros requisitos, utilizaremos el paquete express-validator. Es una buena práctica hacer una validación completa usando express-validator independientemente de si dicha validación está parcialmente incluida en la base de datos o no.

Observa que crearemos una matriz de reglas para cada punto final que requiera validación, por lo general, una matriz de creación de reglas para crear nuevos datos y una matriz de actualización de reglas para actualizar datos.

Se puede encontrar más información sobre cómo usar y escribir middlewares en la documentación de Express: https://expressjs.com/en/guide/using-middleware.html

1.2. Definición de middlewares y middlewares de validación para las rutas de Restaurant

Abre el archivo routes/RestaurantRoutes.js. Encontrarás que se definen las rutas, pero es necesario definir qué middlewares se llamarán para cada ruta.

Incluye los middlewares necesarios para las rutas de Restaurant según los requisitos del proyecto Deliverus. Para cada ruta, deberá determinar si:

- ¿Es necesario que un usuario haya iniciado sesión?
- ¿Es necesario que el usuario tenga un rol particular?
- ¿Es necesario que el restaurante pertenezca al usuario que ha iniciado sesión? (los datos del restaurante deben incluir un userId que pertenezca al propietario de ese restaurante)
- ¿Es necesario que los datos del restaurante incluyan valores válidos para cada propiedad para poder crearse según nuestros requisitos de información?

Solución:

```
app.route('/restaurants')
  .get(
    RestaurantController.index)
  .post(
    middlewares.isLoggedIn,
    middlewares.hasRole('owner'),
    upload,
    RestaurantValidation.create,
    middlewares.handleValidation,
    RestaurantController.create)

app.route('/restaurants/:restaurantId')
  .get(RestaurantController.show)
  .put(
    middlewares.isLoggedIn,
    middlewares.hasRole('owner'),
    middlewares.checkEntityExists(Restaurant, 'restaurantId'),
    middlewares.checkRestaurantOwnership,
    upload,
    RestaurantValidation.update,
    middlewares.handleValidation,
    RestaurantController.update)
  .delete(
    middlewares.isLoggedIn,
    middlewares.hasRole('owner'),
    middlewares.checkEntityExists(Restaurant, 'restaurantId'),
    middlewares.checkRestaurantOwnership,
    RestaurantController.destroy)
```

```
  app.route('/restaurants/:restaurantId/orders')
    .get(
      middlewares.isLoggedIn,
      middlewares.hasRole('owner'),
      middlewares.checkEntityExists(Restaurant, 'restaurantId'),
      middlewares.checkRestaurantOwnership,
      OrderController.indexRestaurant)

  app.route('/restaurants/:restaurantId/products')
    .get(
      middlewares.checkEntityExists(Restaurant, 'restaurantId'),
      ProductController.indexRestaurant)

  app.route('/restaurants/:restaurantId/analytics')
    .get(
      middlewares.isLoggedIn,
      middlewares.hasRole('owner'),
      middlewares.checkEntityExists(Restaurant, 'restaurantId'),
      middlewares.checkRestaurantOwnership,
      OrderController.analytics)
}
```

1.3. Implementar middleware de validación para Restaurant create()

Abrae el archivo controllers/validation/RestaurantValidation.js. Encontrarás las matrices de reglas para validar los datos al crear create y al actualizar update. Las propiedades del restaurante se definen a nivel de base de datos. Puedes verificar la migración correspondiente. Algunas validaciones se realizan a nivel de aplicación, por ejemplo, incluiremos validaciones para verificar que los datos de correo electrónico son un correo electrónico válido.

Además, es común aplicar alguna limpieza de datos. Por ejemplo, para eliminar espacios en blanco al principio y al final de los valores de cadena, podemos usar el método **trim ().**

Para agregar validaciones, sigue este fragmento de código:

```
create: [
    check('name').exists().isString().isLength({ min: 1, max: 255 }).trim(),
    check('description').optional({ nullable: true, checkFalsy: true }).isString().trim(),
    check('address').exists().isString().isLength({ min: 1, max: 255 }).trim(),
    check('postalCode').exists().isString().isLength({ min: 1, max: 255 }),
    check('url').optional({ nullable: true, checkFalsy: true }).isString().isURL().trim(),
    check('shippingCosts').exists().isFloat({ min: 0 }).toFloat(),
    check('email').optional({ nullable: true, checkFalsy: true }).isString().isEmail().trim(),
    check('phone').optional({ nullable: true, checkFalsy: true }).isString().isLength({ min: 1, max: 255 }).trim(),
    check('restaurantCategoryId').exists({ checkNull: true }).isInt({ min: 1 }).toInt(),
    check('userId').not().exists(),
    check('heroImage').custom((value, { req }) => {
      return checkFileIsImage(req, 'heroImage')
    }).withMessage('Please upload an image with format (jpeg, png).'),
    check('heroImage').custom((value, { req }) => {
      return checkFileMaxSize(req, 'heroImage', maxFileSize)
    }).withMessage('Maximum file size of ' + maxFileSize / 1000000 + 'MB'),
    check('logo').custom((value, { req }) => {
      return checkFileIsImage(req, 'logo')
    }).withMessage('Please upload an image with format (jpeg, png).'),
    check('logo').custom((value, { req }) => {
      return checkFileMaxSize(req, 'logo', maxFileSize)
    }).withMessage('Maximum file size of ' + maxFileSize / 1000000 + 'MB'),
  ], update:[
    ...
  ]
```

Así es cómo se ve en un principio:

```
const { check } = require('express-validator')
const { checkFileIsImage, checkFileMaxSize } = require('./FileValidationHelper')
const maxFileSize = 2000000 // around 2Mb

module.exports = {
  create: [
    // TODO Check that the body includes valid values for its properties when creating a restaurant

    check('heroImage').custom((value, { req }) => {
      return checkFileIsImage(req, 'heroImage')
    }).withMessage('Please upload an image with format (jpeg, png).'),
    check('heroImage').custom((value, { req }) => {
      return checkFileMaxSize(req, 'heroImage', maxFileSize)
    }).withMessage('Maximum file size of ' + maxFileSize / 1000000 + 'MB'),
    check('logo').custom((value, { req }) => {
      return checkFileIsImage(req, 'logo')
    }).withMessage('Please upload an image with format (jpeg, png).'),
    check('logo').custom((value, { req }) => {
      return checkFileMaxSize(req, 'logo', maxFileSize)
    }).withMessage('Maximum file size of ' + maxFileSize / 1000000 + 'MB'),
  ],
```

```
    update: [
      // TODO Check that the body includes valid values for its properties when updating a restaurant
      check('heroImage').custom((value, { req }) => {
        return checkFileIsImage(req, 'heroImage')
      }).withMessage('Please upload an image with format (jpeg, png).'),
      check('heroImage').custom((value, { req }) => {
        return checkFileMaxSize(req, 'heroImage', maxFileSize)
      }).withMessage('Maximum file size of ' + maxFileSize / 1000000 + 'MB'),
      check('logo').custom((value, { req }) => {
        return checkFileIsImage(req, 'logo')
      }).withMessage('Please upload an image with format (jpeg, png).'),
      check('logo').custom((value, { req }) => {
        return checkFileMaxSize(req, 'logo', maxFileSize)
      }).withMessage('Maximum file size of ' + maxFileSize / 1000000 + 'MB'),
    ]
  }
```

Solución:

```
module.exports = {
  create: [
    check('name').exists().isString().isLength({ min: 1, max: 255 }).trim(),
    check('description').optional({ nullable: true, checkFalsy: true }).isString().trim(),
    check('address').exists().isString().isLength({ min: 1, max: 255 }).trim(),
    check('postalCode').exists().isString().isLength({ min: 1, max: 255 }),
    check('url').optional({ nullable: true, checkFalsy: true }).isString().isURL().trim(),
    check('shippingCosts').exists().isFloat({ min: 0 }).toFloat(),
    check('email').optional({ nullable: true, checkFalsy: true }).isString().isEmail().trim(),
    check('phone').optional({ nullable: true, checkFalsy: true }).isString().isLength({ min: 1, max: 255 }).trim(),
    check('restaurantCategoryId').exists({ checkNull: true }).isInt({ min: 1 }).toInt(),
    check('userId').not().exists(),
    check('heroImage').custom((value, { req }) => {
      return checkFileIsImage(req, 'heroImage')
    }).withMessage('Please upload an image with format (jpeg, png).'),
    check('heroImage').custom((value, { req }) => {
      return checkFileMaxSize(req, 'heroImage', maxFileSize)
    }).withMessage('Maximum file size of ' + maxFileSize / 1000000 + 'MB'),
    check('logo').custom((value, { req }) => {
      return checkFileIsImage(req, 'logo')
    }).withMessage('Please upload an image with format (jpeg, png).'),
    check('logo').custom((value, { req }) => {
      return checkFileMaxSize(req, 'logo', maxFileSize)
    }).withMessage('Maximum file size of ' + maxFileSize / 1000000 + 'MB'),
  ],
```

```
update: [
  check('name').exists().isString().isLength({ min: 1, max: 255 }).trim(),
  check('description').optional({ nullable: true, checkFalsy: true }).isString().trim(),
  check('address').exists().isString().isLength({ min: 1, max: 255 }).trim(),
  check('postalCode').exists().isString().isLength({ min: 1, max: 255 }),
  check('url').optional({ nullable: true, checkFalsy: true }).isString().isURL().trim(),
  check('shippingCosts').exists().isFloat({ min: 0 }).toFloat(),
  check('email').optional({ nullable: true, checkFalsy: true }).isString().isEmail().trim(),
  check('phone').optional({ nullable: true, checkFalsy: true }).isString().isLength({ min: 1, max: 255 }).trim(),
  check('restaurantCategoryId').exists({ checkNull: true }).isInt({ min: 1 }).toInt(),
  check('userId').not().exists(),
  check('heroImage').custom((value, { req }) => {
    return checkFileIsImage(req, 'heroImage')
  }).withMessage('Please upload an image with format (jpeg, png).'),
  check('heroImage').custom((value, { req }) => {
    return checkFileMaxSize(req, 'heroImage', maxFileSize)
  }).withMessage('Maximum file size of ' + maxFileSize / 1000000 + 'MB'),
  check('logo').custom((value, { req }) => {
    return checkFileIsImage(req, 'logo')
  }).withMessage('Please upload an image with format (jpeg, png).'),
  check('logo').custom((value, { req }) => {
    return checkFileMaxSize(req, 'logo', maxFileSize)
  }).withMessage('Maximum file size of ' + maxFileSize / 1000000 + 'MB'),
]
}
```

Para una lista completa de métodos de validación, consulte https://github.com/validatorjs/validator.js#validators, y para una lista completa de métodos de saneamiento, consulte https://github.com/validatorjs/validator.js#sanitizers

1.4. Verificar la validación en los controladores

Cuando la validación falla, se pasa al siguiente método de middleware en la cadena de middleware. En este caso, el siguiente método debe ser el método de manejo de validación que siempre se ejecutará después del middleware de validación.

Dentro del método **handleValidation**, podemos verificar si se ha violado alguna regla de validación y devolver la respuesta apropiada. Con este fin, el método handleValidation incluye el siguiente código:

```
const handleValidation = async (req, res, next) => {
  const err = validationResult(req)
  if (err.errors.length > 0) {
    res.status(422).send(err)
  } else {
    next()
  }
}
```

2. Prueba de rutas, controladores y middlewares de Restaurantes

Abre la extensión **ThunderClient** ('https://www.thunderclient.io/') y recarga las colecciones si aún no se han cargado haciendo clic en Collections→ ≡ menú → recargar. Estas colecciones se almacenan en **example_api_client/thunder-tests.**

Haz clic en la carpeta Collections y encontrarás un conjunto de solicitudes con pruebas para todos los puntos finales. Ejecuta toda la colección y verás en el lado derecho si una prueba es exitosa o no. Algunas solicitudes realizan más de una prueba.

* Anexo sobre transacciones

Algunas operaciones de la base de datos deben estar incluidas en una transacción de base de datos. Esto es útil cuando se necesita asegurar que varias operaciones se ejecuten con éxito en la base de datos y, en caso de que alguna de ellas genere una excepción, deshacer todas las operaciones anteriores. Por ejemplo, para crear un pedido, es necesario insertar un registro en la tabla Pedidos e insertar varios registros en la tabla **OrderProducts**. Esto debe hacerse en una transacción.

Para crear una transacción utilizando Sequelize, puedes seguir este ejemplo de código:

```
const models = require('../models')
const EntityName = models.EntityName

const someFunctionThatNeedsTransaction = async (req, res) => {
  let newEntity = EntityName.build(req.body)
  const transaction = await models.sequelize.transaction() //creates a transaction
  try {
    newEntity = await newEntity.save({transaction}) //use the transaction in every operation
    await newEntity.addRelatedEntity(relatedEntityId,
                                     { through: { associatedAttribute1: value1, associatedAttributeN: valueN }, transaction })
    //adding associated element through an association table
    await transaction.commit() //confirm all operations
    res.json(newEntity)
  } catch (err) {
    await transaction.rollback() //in case of error, rollback all the operations executed in the context of the transaction
    res.status(500).send(err)
  }
```