

BACKEND

ESTRUCTURA DEL BACKEND

- **`package.json`**: contiene información sobre el proyecto, como las dependencias que necesita para funcionar, así como algunos scripts útiles para iniciar el servidor
- **`package-lock.json`**: Asegura que cuando instalas las dependencias del proyecto, siempre obtienes las mismas versiones para evitar problemas de compatibilidad en el futuro.
- **`.env`**: incluye las credenciales de nuestra base de datos
- **`src/backend.js`**: Este archivo es como el corazón de tu aplicación. Es donde se inicia el servidor, se configuran las conexiones a la base de datos y se inician otros componentes importantes.
- **`src/models`** donde definimos cómo se ven y se comportan los datos. Es como el diseño de tu base de datos
- **`src/database`** Aquí es donde se encuentra toda la lógica relacionada con la base de datos. Se divide en:
 - **`src/database/migrations`** aquí se define el esquema de la base de datos
 - **`src/database/seeds`** aquí se agrega datos de ejemplo a tu base de datos para probar tu aplicación.
- **`src/routes`** carpeta donde se definen los URI y se hace referencia a middlewares y controladores
- **`src/controllers`** carpeta donde se implementa la lógica empresarial, incluidas las operaciones en la base de datos.
 - **`src/controllers/validation`** Aquí es donde puedes asegurarte de que los datos que recibes de los usuarios sean válidos y seguros. Un archivo de validación para cada entidad
- **`src/middlewares`** son como filtros que se ejecutan antes o después de que se ejecuten los controladores.
- **`src/config`** carpeta: donde se almacenan algunos archivos de configuración globales (para ejecutar migraciones y seeders desde cli)
- **`src/test`** Carpeta: almacenará las solicitudes de prueba unitaria en nuestra API Rest, utilizando el módulo [SuperTest](#) .

MIGRATIONS

Las migraciones en el contexto de desarrollo backend son como "registros de cambios" para tu base de datos. Es un archivo especial que contiene instrucciones sobre cómo hacer cambios en la estructura de la base de datos, como crear o modificar tablas, añadir o eliminar columnas, etc.

Nosotros los usaremos para crear nuestro esquema de base de datos. Tendremos una migración para cada entidad.

Cada migración tiene 2 métodos:

-**up**: incluye como debemos crear cada tabla en la BD y sus campos. Especificamos entre otras cosas PK, FK, atributos.

-**down**: incluye como deshacer la creación de las tablas y campos que había en up. Es como tu plan para deshacer lo que hiciste si algo sale mal o si necesitas revertir los cambios

Las migraciones tienen el siguiente esquema y pueden tener los tipos :

```
module.exports = {
  up: async(queryInterface, Sequelize) => {
    // Logic for transforming into the new state },
  down: async(queryInterface, Sequelize) => {
    // Logic for reverting the changes}}
```

<u>TIPO</u>	<u>CORRESPONDENCIA</u>
Cadena	Sequelize.STRING
Entero	Sequelize.INTEGER o Sequelize.BIGINT
Decimal	Sequelize.FLOAT o Sequelize.DOUBLE
Fecha	Sequelize.DATE
Booleano	Sequelize.BOOLEAN
Enumerado	Sequelize.ENUM
Array	Sequelize.ARRAY
Rango	Sequelize.RANGE

<u>ATRIBUTO</u>	<u>EXPLICACION</u>
allowNull: <code>false</code>	Define si la columna puede contener valores NULL. Con false decimos que no
defaultValue:	Especifica un valor por defecto para la columna
primaryKey: <code>true</code>	Define si la columna es una clave primaria. Con true decimos que si
autoIncrement: <code>true</code>	Define si la columna tiene auto-incremento. Con true decimos que si
unique: <code>true</code>	Define si los valores de la columna deben ser únicos.. Con true decimos que si
references: { model: 'OtraTabla', key: 'id' }	Define una foreing key a otra tabla.
onDelete: ' <code>CASCADE</code> ' o ' <code>SET NULL</code> '	Define el comportamiento al borrar una foreing key
onUpdate: ' <code>CASCADE</code> ' o ' <code>SET NULL</code> '	Define el comportamiento al actualizar una foreing key

Veamos algunos ejemplos básicos para ver cómo funcionan:

```
module.exports = {
  up: async(queryInterface, Sequelize) => {
    await queryInterface.createTable('Person', {
      name: Sequelize.STRING,
      isBetaMember: {
        type: Sequelize.BOOLEAN,
        defaultValue: false,
        allowNull: false
      }
    })
  },
  down: async(queryInterface, Sequelize) => {
    await queryInterface.dropTable('Person');
  }
}
```

- **up**: define cómo se creará la tabla 'Person' en la base de datos. La tabla tendrá dos columnas: 'name', que será de tipo STRING, y 'isBetaMember', que será de tipo BOOLEAN (booleano). También se especifica que 'isBetaMember' por defecto será falso y que no se permitirá que sea nulo
- **down**: describe cómo deshacer la creación de la tabla 'Person'. Elimina la tabla 'Person' de la base de datos

```
module.exports = {
  up: async (queryInterface, Sequelize) => {
    await queryInterface.createTable('Person', {
      name: Sequelize.STRING,
      isBetaMember: {
        type: Sequelize.BOOLEAN,
        defaultValue: false,
        allowNull: false },
      userId: {
        type: Sequelize.INTEGER,
        references: {
          model: {
            tableName: 'users',
            schema: 'schema' },
          key: 'id' },
        allowNull: false },
    })
  },
  down: async (queryInterface, Sequelize) => {
    await queryInterface.dropTable('Person');
  }
}
```

- **up**: define cómo se creará la tabla 'Person' en la base de datos. La tabla tendrá dos columnas: 'name', que será de tipo STRING, y 'isBetaMember', que será de tipo BOOLEAN (booleano). También se especifica que 'isBetaMember' por defecto será falso y que no se permitirá que sea nulo. En este caso, también tiene una FK, la columna 'userId' de tipo INTEGER. Hace referencia a la tabla 'users'
- **down**: describe cómo deshacer la creación de la tabla 'Person'. Elimina la tabla 'Person' de la base de datos

Product
name: String
description: String [0..1]
price: Double
image: String [0..1]
order: Integer [0..1]
availability: Boolean

```
module.exports = {

  up: async (queryInterface, Sequelize) => {
    await queryInterface.createTable('Products', {
      id: {
        allowNull: false,
        autoIncrement: true,
        primaryKey: true,
        type: Sequelize.INTEGER },
      name: {
        allowNull: false,
        type: Sequelize.STRING},
      description: {
        type: Sequelize.TEXT },
      price: {
        allowNull: false,
        type: Sequelize.DOUBLE },
      image: {
        type: Sequelize.STRING },
      order: {
        type: Sequelize.INTEGER },
      availability: {
        type: Sequelize.BOOLEAN },
    })
  },

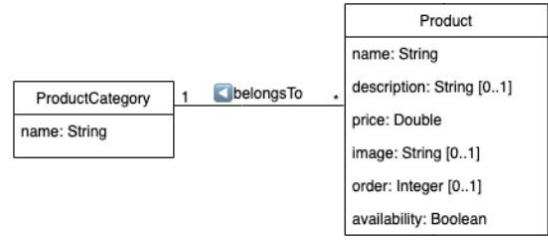
  down: async (queryInterface, Sequelize) => {
    await queryInterface.dropTable('Products')
  }
}
```

HAY QUE AÑADIR LOS ATRIBUTOS DE LOS QUE LA TABLA QUE ESTAMOS CREANDO ES FK O TIENE ALGUNA RELACION CON OTRA TABLA

```

productCategoryId: {
  type: Sequelize.INTEGER,
  allowNull: false,
  references: {
    model: {
      tableName: 'ProductCategories'},
      key: 'id' }}}

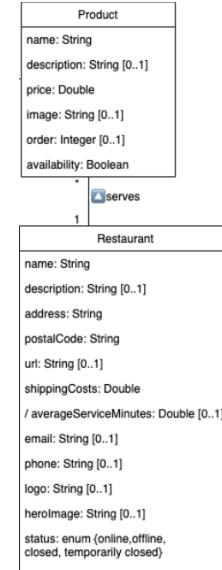
```



```

restaurantId: {
  type: Sequelize.INTEGER,
  allowNull: false,
  references: {
    model: {
      tableName: 'Restaurants'},
      key: 'id' },
      onDelete: 'cascade'}}}

```



```

down: async (queryInterface, Sequelize) => {
  await queryInterface.dropTable('Products')
}

```

- **up**: crea una tabla llamada 'Products' en la base de datos con columnas para almacenar información sobre productos, como su id, nombre, descripción, precio, imagen, disponibilidad, y a qué restaurante y categoría pertenecen. Además, se establecen FK para referenciar las tablas 'Restaurants' y 'ProductCategories'.
- **down**: elimina la tabla 'Products' de la base de datos

TODO: Migration de Restaurant

```

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')
  }
}

```

SOLUCIÓN:

```
module.exports = {
  up: async (queryInterface, Sequelize) => {
    await queryInterface.createTable('Restaurants', {
      id: {
        allowNull: false,
        autoIncrement: true,
        primaryKey: true,
        type: Sequelize.INTEGER
      },
      name: {
        allowNull: false,
        type: Sequelize.STRING
      },
      description: {
        type: Sequelize.TEXT
      },
      address: {
        allowNull: false,
        type: Sequelize.STRING
      },
      postalCode: {
        allowNull: false,
        type: Sequelize.STRING
      },
      url: {
        type: Sequelize.STRING
      },
      shippingCosts: {
        allowNull: false,
        type: Sequelize.DOUBLE
      },
      averageServiceMinutes: {
        allowNull: true,
        type: Sequelize.DOUBLE
      },
      email: {
        type: Sequelize.STRING
      },
      phone: {
        type: Sequelize.STRING
      },
      logo: {
        type: Sequelize.STRING
      },
      heroImage: {
        type: Sequelize.STRING
      }
    })
  }
}
```

```

status: {
  type: Sequelize.ENUM,
  values: [
    'online',
    'offline',
    'closed',
    'temporarily closed'
  ],
  createdAt: {

    allowNull: false,
    type: Sequelize.DATE,
    defaultValue: new Date()
  },
  updatedAt: {
    allowNull: false,
    type: Sequelize.DATE,
    defaultValue: new Date()
  },
}

restaurantCategoryId: {
  type: Sequelize.INTEGER,
  allowNull: false,
  references: {
    model: {
      tableName: 'RestaurantCategories'
    },
    key: 'id'
  }
},
userId: {
  allowNull: false,
  type: Sequelize.INTEGER,
  onDelete: 'CASCADE',
  references: {
    model: {
      tableName: 'Users'
    },
    key: 'id'
  }
}
}),
down: async (queryInterface, Sequelize) => {
  await queryInterface.dropTable('Restaurants')}

```

Vemos que efectivamente se ha creado esta tabla con los atributos en la BD:

```

cd DeliverUS-Backend
npx sequelize-cli db:migrate

```

		name	description	price		order	availability	restaurantId		productCategoryId		createdAt	updatedAt
--	--	------	-------------	-------	--	-------	--------------	--------------	--	-------------------	--	-----------	-----------

MODELS

Los modelos son estructuras de datos que utilizamos para interactuar con la base de datos. Nos permiten realizar operaciones en la base de datos, como crear, leer, actualizar y eliminar datos(CRUD), de una manera más estructurada y controlada.

Un modelo es una clase que extiende [Model](#). El modelo le dice a Sequelize varias cosas sobre la entidad que representa, como el nombre de la tabla en la base de datos y qué columnas tiene (y sus tipos de datos).

Un modelo en Sequelize tiene un nombre. Este nombre no tiene que ser el mismo nombre de la tabla que representa en la base de datos. Por lo general, los modelos tienen nombres en singular (como User), mientras que las tablas tienen nombres en plural (como Users).

Por otro lado, debemos tener en cuenta las [asociaciones](#) entre las entidades. Para ello, Sequelize proporciona 4 tipos de asociaciones que conviene combinar para crearlas:

- ❖ Para crear una relación **uno a uno**, las asociaciones `hasOne` y `belongsTo` se utilizan juntas: A. `hasOne(B)`
B. `belongsTo(A)`
- ❖ Para crear una relación **uno a muchos**, las asociaciones `hasMany` y `belongsTo` se utilizan juntas; A. `hasMany(B)`
B. `belongsTo(A)`
- ❖ Para crear una relación **de muchos a muchos**, `belongsToMany` se utilizan dos llamadas juntas.

Veamos cómo se crean los modelos con un ejemplo:

Consideraremos que queremos crear un [modelo](#) para representar a los usuarios, que tienen a `firstName` y `lastName`. Queremos que se llame a nuestro modelo `User` y que se llame a la tabla que representa `Users` en la base de datos(migración).

```
import { Model } from 'sequelize'  
  
const loadModel = (sequelize, DataTypes) => {  
  
  class User extends Model {}  
  
  User.init({  
    firstName: {  
      type: DataTypes.STRING,  
      allowNull: false},  
    lastName: {  
      type: DataTypes.STRING }}),  
  
  {sequelize,  
   modelName: 'User' })  
  
  return User  
}  
export default loadModel
```

AQUÍ DEFINIMOS LAS ASOCIACIONES

AQUÍ DEFINIMOS LOS
ATRIBUTOS

Veamos el modelo de Product:

```
import { Model } from 'sequelize'
const loadModel = (sequelize, DataTypes) => {
  class Product extends Model {
    static associate (models) {
      // ASOCIACIONES !!!!
      const OrderProducts = sequelize.define('OrderProducts',
        {quantity: DataTypes.INTEGER,
         unityPrice: DataTypes.DOUBLE
      })
      Product.belongsToMany(models.Order, { as: 'orders', through:
      OrderProducts })

      Product.belongsTo(models.Restaurant,
      {foreignKey: 'restaurantId', as: 'restaurant', onDelete: 'cascade' })

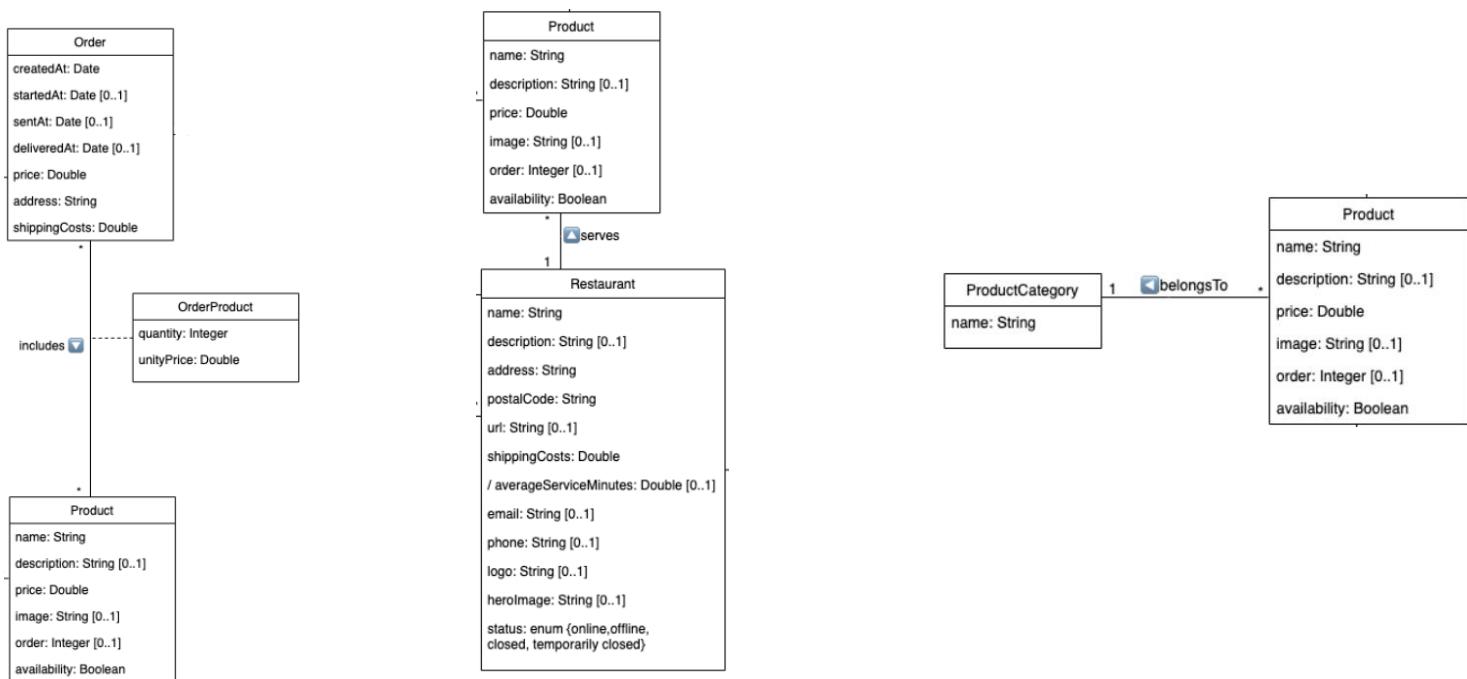
      Product.belongsTo(models.ProductCategory, { foreignKey:
      'productCategoryId', as: 'productCategory' })
    }
  }
}
```

Definimos el metodo `associate` que nos va a permitir definir las asociaciones entre los diferentes modelos.

1.RELACION MUCHOS-MUCHOS: sabemos que esta asociacion crea necesariamente otro modelo intermedio. En este caso tenemos una asociacion de tipo muchos-muchos entre `Product` y `Order` lo que da lugar a `OrderProducts`

2.RELACION MUCHOS-UNO: establece que cada producto pertenece a un Restaurante es decir un restaurante tiene muchos producto pero un producto solo puede pertenecer a un unico restaurante.

3.RELACION MUCHOS-UNO: establece que cada producto pertenece a un ProductCategory.



```

    // ATRIBUTOS !!!!
Product.init({
  name: DataTypes.STRING,
  description: DataTypes.STRING,
  price: DataTypes.DOUBLE,
  image: DataTypes.STRING,
  order: DataTypes.INTEGER,
  availability: DataTypes.BOOLEAN,
  restaurantId: DataTypes.INTEGER,
  productCategoryId: DataTypes.INTEGER}){
  sequelize,
  modelName: 'Product'
})
return Product}
export default loadModel

```

Product
name: String
description: String [0..1]
price: Double
image: String [0..1]
order: Integer [0..1]
availability: Boolean

TODO: Acabar modelo de Restaurant

```

const loadModel = (sequelize, DataTypes) => {
  class Restaurant extends Model {
    static associate (models) {
      // define association here
      Restaurant.belongsTo(models.RestaurantCategory, { foreignKey:
'restaurantCategoryId', as: 'restaurantCategory' })
      Restaurant.belongsTo(models.User, { foreignKey: 'userId', as:
'user' })
      RestauranthasMany(models.Product, { foreignKey: 'restaurantId',
as: 'products' })
      RestauranthasMany(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}
export default loadModel

```

SOLUCIÓN:

```
Restaurant.init({
    // TODO: Include the rest of the properties of the Restaurant model
    name: {
        allowNull: false,
        type: DataTypes.STRING
    },
    description: {
        type: DataTypes.TEXT
    },
    address: {
        allowNull: false,
        type: DataTypes.STRING
    },
    postalCode: {
        allowNull: false,
        type: DataTypes.STRING
    },
    url: {
        type: DataTypes.STRING
    },
    shippingCosts: {
        allowNull: false,
        type: DataTypes.DOUBLE
    },
    averageServiceMinutes: {
        allowNull: true,
        type: DataTypes.DOUBLE
    },
    email: {
        type: DataTypes.STRING
    },
    phone: {
        type: DataTypes.STRING
    },
    logo: {
        type: DataTypes.STRING
    },
    heroImage: {
        type: DataTypes.STRING
    },
    status: {
        type: DataTypes.ENUM,
        values: [
            'online',
            'offline',
            'closed',
            'temporarily closed'
        ]
    }
})
```

```

},
restaurantCategoryId: {
  type: DataTypes.INTEGER,
  allowNull: false
},
userId: {
  allowNull: false,
  type: DataTypes.INTEGER,
  onDelete: 'CASCADE'
},
createdAt: {
  allowNull: false,
  type: DataTypes.DATE,
  defaultValue: new Date()
},
updatedAt: {
  allowNull: false,
  type: DataTypes.DATE,
  defaultValue: new Date()
},
{
  sequelize,
  modelName: 'Restaurant'
})

return Restaurant
}
export default loadModel

```

(Copiar y pegar el migration remplazando “Sequelize” por “DataTypes” con Ctr+H y de las FK borramos lo de references)

CONTROLLERS

Los controladores son los componentes que se encargan de implementar toda la lógica de negocio, es decir operaciones sobre la BD. Cada método de controlador recibe una solicitud **req** y un **res** objeto de respuesta. El objeto de solicitud **req** representa la **solicitud HTTP** y tiene las siguientes propiedades:

- **req.body** representa los datos que provienen del cliente
- **req.params** representa los parámetros de ruta. . Por ejemplo, si en la ruta de una solicitud tenemos /:restaurantId, entonces cualquier valor que se ponga en lugar de restaurantId será accesible a través de `req.params.restaurantId`
- **req.query** representa parámetros de consulta. . Por ejemplo, si la URL de una solicitud incluye `?status=activo`, entonces podrías acceder al valor `activo` usando `req.query.status`.
- **req.user** representa el usuario que inició sesión y realizó la solicitud. Por lo tanto, si necesitas información sobre el usuario que está realizando la solicitud, puedes encontrarla aquí

Por otro lado, el objeto de respuesta **res** representa la **respuesta HTTP** que envía una aplicación Express cuando recibe una solicitud HTTP. Tiene los siguientes métodos:

- **res.json(entityObject)** Este método envía al cliente un objeto `entityObject` como un documento JSON con un código de estado HTTP 200. Por ejemplo, si queremos enviar información sobre un restaurante al cliente, usaremos `res.json(restaurante)` y el cliente recibirá los datos del restaurante en formato JSON.
- **res.json(message)** devuelve una cadena `message` al cliente como un documento JSON con código de estado HTTP 200.
- **res.status(500).send(err)** devuelve el `err` objeto (que normalmente incluye algún tipo de mensaje de error) y un código de estado HTTP 500 al cliente

Los estados del código HTTP utilizados en este proyecto son:

- **200.** Solicitud atendidas exitosamente.
- **401.** Credenciales incorrectas.
- **403.** Solicitud prohibida (no hay suficientes privilegios).
- **404.** No se encontró el recurso solicitado.
- **422.** Error de validación.
- **500.** Error general.

Veamos las operaciones CRUD de Restaurat:

TODO: Controller de Restaurant

CREAR UNA ENTIDAD:

```
const create = async function (req, res) {  
    //SINCRONO:: se crea directamente  
    const newRestaurant = Restaurant.build(req.body)  
    // creamos un nuevo restaurante con la informacion del cliente  
    newRestaurant.userId = req.user.id // usuario actualmente autenticado  
    try{  
        //ASINCRONO: se necesita algo de tiempo en realizarse  
        const restaurant = await newRestaurant.save()  
        // guardamos el restaurante nuevo en la bd  
        res.json(restaurant) // devolvemos en formato json el restaurante  
    }  
    catch(err){  
        res.status(500).send(err)  
    }  
}
```

La función create recibe 2 parametros, req que representa la solicitud HTTP y res que representa la respuesta HTTP que enviara la aplicación.

Luego, se intenta guardar este nuevo producto en la base de datos utilizando el método save () proporcionado por Sequelize. Como este proceso puede generar un error, se utiliza un bloque

Si el producto se guarda correctamente en la base de datos, se responde al cliente con el objeto del producto recién creado en formato JSON utilizando res.json (newProduct)

Si ocurre algún error durante el proceso de guardado, se envía al cliente un mensaje de error junto con un código de estado HTTP 500

LEER UNA ENTIDAD-LISTADO

Así seria incluyendo todo y sin ninguna peculiaridad:

```
const index = async function (req, res) {  
    try{  
        const restaurants = Restaurant.findAll()  
        res.json(restaurants)  
    }  
    catch(err){  
        res.status(500).send(err)  
    }  
}
```

```

const index = async function (req, res) {
  try {
    const restaurants = await Restaurant.findAll(
      {
        attributes: { exclude: ['userId'] },
        include: [
          { model: RestaurantCategory, as: 'restaurantCategory' },
          { order: [[{ model: RestaurantCategory, as: 'restaurantCategory' }, 'name', 'ASC']] }
        ]
      }
    )
    res.json(restaurants)
  } catch (err) {
    res.status(500).send(err)
  }
}

```

La función index recibe 2 parámetros, req que representa la solicitud HTTP y res que representa la respuesta HTTP que enviará la aplicación.

Dentro del bloque try se utiliza el método findAll () proporcionado por Sequelize para buscar todos los restaurantes en la base de datos. Este método devuelve una lista de objetos de restaurante.

En esta llamada al método, personalizamos la consulta:

- Excluye el atributo userId de los resultados para proteger la privacidad del usuario propietario del restaurante
- Incluye la información de la tabla relacionada RestaurantCategory bajo el alias restaurantCategory
- Ordena los resultados por el nombre de la categoría del restaurante en orden ascendente.

Si ocurre algún error durante el proceso de guardado, se envía al cliente un mensaje de error junto con un código de estado HTTP 500

LEER UNA ENTIDAD ESPECIFICA + DETALLES

Así sería incluyendo todo y sin ninguna peculiaridad:

```

const show = async function (req, res) {
  try{
    const restaurant = await Restaurant.findByPk(req.params.restaurantId)
    res.json(restaurant)
  }
  catch(err){
    res.status(500).send(err)
  }
}

```

Queremos consultar los detalles de los restaurantes y los productos que se ofrecen. Teniendo en cuenta que debemos incluir los productos y la categoría del restaurante. Recuerdamos que los productos deben ordenarse según el valor del campo de pedido:

```
const show = async function (req, res) {
  try{
    const restaurant = await Restaurant.findByPk(req.params.restaurantId,
      //el id del restaurante a consultar esta en la ruta, y se pasa como
      parametro
    {
      attributes: { exclude: ['userId'] },
      include: [
        {
          model: Product,
          as: 'products',
          include : { model: ProductCategory, as: 'productCategory' },
        },
        {
          model: RestaurantCategory,
          as: 'restaurantCategory'
        }
      ],
      order: [[{model:Product, as: 'products'}, 'order', 'ASC']],
    }
  )
  res.json(restaurant)
}
catch(err){
  res.status(500).send(err)
}
}
```

Product a su vez tiene una asociación luego tenemos q incluirla tb!!

La función show recibe 2 parámetros, req que representa la solicitud HTTP y res que representa la respuesta HTTP que enviará la aplicación.

utiliza el método Restaurant.findByPk() de Sequelize para buscar un restaurante específico en la BD. Se utiliza req.params.restaurantId para obtener el ID del restaurante de los parámetros de la solicitud.

En esta llamada al método, personalizamos la consulta:

- Excluye el atributo userId de los resultados para proteger la privacidad del usuario propietario del restaurante
- Incluye la información de las asociaciones
- Ordena los resultados por orden de los productos en orden ascendente.

Si ocurre algún error durante el proceso de guardado, se envía al cliente un mensaje de error junto con un código de estado HTTP 500

ACTUALIZAR UNA ENTIDAD

```
const update = async function (req, res) {
  try {
    await Restaurant.update(req.body,
    { where: { id: req.params.restaurantId } })
    const updatedRestaurant = await
    Restaurant.findByPk(req.params.restaurantId)
    res.json(updatedRestaurant)
  } catch (err) {
    res.status(500).send(err)
  }
}
```

La función update recibe 2 parametros, req que representa la solicitud HTTP y res que representa la respuesta HTTP que enviara la aplicación.

Se utiliza el método `Restaurant.update()` de Sequelize para actualizar la información del restaurante en la base de datos. Se pasa `req.body`, que contiene los datos actualizados del restaurante, y se utiliza `where: { id: req.params.restaurantId }` para especificar qué restaurante debe ser actualizado, identificándolo por su ID.

Después de la actualización, se utiliza `Restaurant.findByPk()` para encontrar el restaurante actualizado utilizando su id y devolverlo como respuesta en formato json

Si ocurre algún error durante el proceso de guardado, se envía al cliente un mensaje de error junto con un código de estado HTTP 500

ELIMINAR UNA ENTIDAD

```
const destroy = async function (req, res) {
  try {
    const result = await Restaurant.destroy({ where: { id:
    req.params.restaurantId } })
    let message = ''
    if (result === 1) {
      message = 'Successfully deleted restaurant id.' +
      req.params.restaurantId}
    else {
      message = 'Could not delete restaurant.'}
    res.json(message)
  } catch (err) {
    res.status(500).send(err) }}
```

La función destroy recibe 2 parametros, req que representa la solicitud HTTP y res que representa la respuesta HTTP que enviara la aplicación.

Se utiliza el método `Restaurant.destroy()` de Sequelize para eliminar el restaurante de la base de datos. Se utiliza `where: { id: req.params.restaurantId }` para especificar qué restaurante debe ser eliminado, identificándolo por su id. El método destroy devuelve **el número de elementos que fueron destruidos** en la base de datos. En este caso, se almacena este resultado en la variable result.

Si result es igual a 1, significa que se eliminó exitosamente un restaurante, por lo que se asigna un mensaje que indica el éxito de la eliminación. Si no, no se puede eliminar el restaurante indicado.

Si ocurre algún error durante el proceso de guardado, se envía al cliente un mensaje de error junto con un código de estado HTTP 500

ROUTES

Las rutas tienen 2 funciones principales:

1. Definen las operaciones que ofrece la API
2. Redirigen las peticiones al controlador correspondiente.

Las rutas reciben 2 parámetros de entrada (verbo HTTP y el recurso) y como salida devuelve el controlador que da servicio a la petición. El recurso lo vamos a identificar a través de una ruta dentro la URL. ¿Cómo se le pasa información del frontend al backend, es decir del cliente al servidor? De 3 maneras:

- ❖ Parámetros: dentro de la ruta como parte de los recursos: restaurants/1 = restaurant/:restaurantId
- ❖ Query: dentro de la ruta, todos los pares clave-valor que cumplan lo indicado después de ? → ?date=2024/01/01
- ❖ Body: en el cuerpo de la petición se manda en formato .json los pares clave-valor

Las rutas tienen el siguiente esquema:

```
app.route('/path')
  .get(
    EntityController.index)
  .post(
    EntityController.create)
```

TODO: Routes de Restaurant

```
import RestaurantController from '../controllers/RestaurantController.js'

const loadFileRoutes = function (app) {
  //FR1: Listado de restaurantes: los clientes podrán consultar todos los restaurantes.
  app.route('/restaurants')
    .get(RestaurantController.index)
    .post(RestaurantController.create)

  //FR2: Detalles y menú de restaurantes: Los clientes podrán consultar los detalles de los restaurantes y los productos que ofrecen + CRUD
  app.route('/restaurants/:restaurantId')
    .get(RestaurantController.show)
    .put(RestaurantController.update)
    .delete(RestaurantController.destroy)

}

export default loadFileRoutes
```

Para implementar ciertas funcionalidades del Backend, es necesario conocer la estructura del body dentro de la req q nos manda el Frontend.

En DeliverUS-Backend\tests\e2e\utils\order.js, encontramos un ejemplo del body que va a contener una petición cuando llegue al Backend:

```
return {
  address: 'Calle falsa 123',
  restaurantId: restaurant.id,
  products: [
    { productId: createdPaellaProduct.id, quantity: paellaQuantity },
    { productId: createdCervezaProduct.id, quantity: cervezaQuantity }
  ]
}
```

La estructura que tendrá el body de una req cuando llegue al Backend será un objeto con 3 propiedades:

- 1.La dirección a la que quiero mandar el pedido
- 2.El id del restaurante al que estoy realizando el pedido
- 3.Listado de productos que he solicitado en mi pedido. Cada producto que se ha pedido tiene a su vez 2 propiedades: el id del producto solicitado y la cantidad solicitada.

MIDDLEWARES

Los middlewares son funciones que se ejecutan entre la solicitud de un usuario y la respuesta del servidor es decir entre que llega la req del usuario y ANTES de que llegue al controlador. De esta manera, cuando llegue info/req al controlador se verifica que se va validado/verificado dicha información.

Tenemos un middlewares para cada entidad, uno para verificar si una identificación determinada identifica un registro de una entidad determinada en la base de datos y otro para autenticación/autorización.

Por ejemplo, cuando un usuario envía una solicitud para crear un nuevo producto, utilizaremos componentes de middleware para:

- Comprobar que el usuario ha iniciado sesión
- Verificar que el usuario tenga el rol de propietario (ya que los clientes no pueden crear productos)
- Gestionar la subida de la imagen del producto
- Comprobar que el producto pertenece a un restaurante de su propiedad (los datos incluyen un restaurantId que pertenece al propietario que realiza la solicitud)
- Verificar que los datos del producto incluyan valores válidos para cada propiedad para que se creen de acuerdo con nuestros requisitos de información.

A continuación se explican funciones de middlewares compartidos por varias entidades:

AUTENTICACION/IDENTIFICACION

```
const hasRole = (...roles) => (req, res, next) => {
  if (!req.user) {
    return res.status(403).send({ error: 'Not logged in' })
  }
  if (!roles.includes(req.user.userType)) {
    return res.status(403).send({ error: 'Not enough privileges' })
  }
  return next()}
```

La función hasRole recibe una serie de nombres de roles y verifica si el usuario que inició sesión tiene el rol necesario. Devuelve una función que toma tres parámetros `req, res, next`.

Si NO esta loggeado es decir si NO ha iniciado sesión `!req.user` se lanza error 403.

Por otro lado, si esta loggeado comprobamos que el rol del usuario que ha iniciado sesión pertenece al rol/es pasado/s por parámetros. Si no lanzamos error 403.

En otro caso, continua comprobando middlewares

```
const isLoggedIn = (req, res, next) => {
  passport.authenticate('bearer', { session: false })(req, res, next)
}
```

La función isLoggedIn devuelve una función que toma tres parámetros `req, res, next`. Comprueba que se tiene el token de autenticación que se genera cuando un usuario se loggea

IDENTIFICACION EN BD

```
const checkEntityExists = (model, idPathParamName) => async (req, res, next) => {
  try {
    const entity = await model.findByPk(req.params[idPathParamName])
    if (!entity) { return res.status(404).send('Not found') }
    return next()
  } catch (err) {
    return res.status(500).send(err)
  }
}
```

La función checkEntityExists recibe como parámetro el modelo que queremos comprobar que exista y el nombre del parámetro establecido en la ruta que contiene el id que se quiere buscar. Devuelve una función que toma tres parámetros `req, res, next`

Se intenta buscar la entidad en la base de datos atraves del id que hemos especificado en nuestra ruta. Si no se encuentra significa que no existe y por lo tanto se lanza error 404. Si se encuentra se sigue comprobando middlewares con `next()`.

Si ocurre algún error durante la búsqueda de la entidad en la base de datos, captura ese error en `catch`

GENERAL

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

La función handleValidation recoleta los errores (de validación solo)detectados en las validaciones (`validationResult(req)`) y si ha detectado algún error lanza error mientras que si no se detecta ninguno, continua la siguiente middleware.

¡¡ESTE MIDDLEWARE DESPUES DE VALIDAR SOLO PQ SI NO NO TIENE SENTIDO PQ NO VA A RECOLETAR NINGUN ERROR!!

IMÁGENES/ARCHIVOS

```
const handleFilesUpload = (fieldNames, folder) => (req, res, next) => {
  const multerInstance = createMulter(fieldNames, folder)
  multerInstance(req, res, (err) => {
    if (err) {
      res.status(500).send({ error: err.message })
    } else {
      addFilenameToBody(req, fieldNames)
      next()
    }
  })
}
```

Cuando el usuario nos mande un fichero, que suelen ser imágenes , estas imágenes la guardaremos una carpeta específica del servidor para trabajar con la ruta de dichas imágenes. Con `createMulter` recibe un array de archivos y una carpeta. Se crea la carpeta, se guarda las imágenes y se devuelve la ruta de las imágenes.

Recibe una matriz de nombres de campos, correspondientes a atributos de tipo de archivo para una entidad, así como la ruta base del servidor donde se deben almacenar estos archivos. Se encarga de gestionar la carga de los archivos al servidor, incluyendo la generación de un nombre único para cada archivo y el almacenamiento de las rutas completas en los campos correspondientes de la entidad.

Para comprobar todos estos requisitos tenemos que incluir cada método de middleware en la ruta correspondiente:

```
app.route('/products')
  .post(
    isLoggedIn,
    hasRole('owner'),
    handleFilesUpload(['image'], process.env.PRODUCTS_FOLDER),
    ProductValidation.create,
    handleValidation,
    ProductMiddleware.checkProductRestaurantOwnership,
    ProductController.create)
```

TODO: Middlewares de Restaurant

Para cada ruta debemos determinar si:

- ¿Es necesario que un usuario haya iniciado sesión?
- ¿Es necesario que el usuario tenga un rol particular?
- ¿Los datos pueden incluir archivos?
- ¿Es necesario que el restaurante pertenezca al usuario que ha iniciado sesión? (Los datos del restaurante deben incluir un ID de usuario que pertenezca al propietario de ese restaurante)
- ¿Es necesario que los datos del restaurante incluyan valores válidos para cada propiedad para poder ser creados de acuerdo con nuestros requisitos de información?

```
app.route('/restaurants')
  .get(
    RestaurantController.index)
  .post(
    isLoggedIn,
    hasRole('owner'),
    handleFilesUpload(['logo', 'heroImage'],
process.env.RESTAURANTS_FOLDER),
    RestaurantValidation.create,
    handleValidation,
    RestaurantController.create)
```

```
app.route('/restaurants/:restaurantId')
  .get(RestaurantController.show)

  .put(
    isLoggedIn,
    hasRole('owner'),
    checkEntityExists(Restaurant, 'restaurantId'),
    RestaurantMiddleware.checkRestaurantOwnership,
    handleFilesUpload(['logo', 'heroImage'],
process.env.RESTAURANTS_FOLDER),
    RestaurantValidation.update,
    handleValidation,
    RestaurantController.update)

  .delete(
    isLoggedIn,
    hasRole('owner'),
    checkEntityExists(Restaurant, 'restaurantId'),
    RestaurantMiddleware.checkRestaurantOwnership,
    RestaurantMiddleware.restaurantHasNoOrders,
    RestaurantController.destroy)

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

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

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

VALIDATIONS

Los middlewares de validación están destinados a comprobar si los datos que llegan en una solicitud cumplen con los requisitos de información. La mayoría de estos requisitos se definen a nivel de base de datos y se incluyeron en las migrations. Algunos otros requisitos se verifican en la capa de aplicación.

VALIDATOR	DESCRIPCION
contains(str, seed)	Verifica si la cadena contiene la semilla.
equals(str, comparison)	Verifica si la cadena coincide con la comparación.
isEmail(str)	Verifica si la cadena es un correo electrónico válido
isEmpty(str)	Verifica si la cadena tiene una longitud de cero
isLength(str, { min, max })	Verifica la longitud de la cadena.
isURL(str)	Verifica si la cadena es una URL válida
isInt(str)	Verifica si la cadena es un entero
isFloat(str)	Verifica si la cadena es un número decimal
isBoolean(str)	Verifica si la cadena es un booleano
isDate(str)	Verifica si la cadena es una fecha válida.
exists()	

SANITIZADOR	DESCRIPCION
toBoolean(input [, strict])	Convierte el input a un booleano. Todo excepto '0', 'false' y retorna true. En modo estricto, solo '1' y 'true' retornan `true` .
toDate(input)`	Convierte el input a una fecha, o `null` si el input no es una fecha.
toFloat(input)	Convierte el input a un número decimal, o `NaN` si el input no es un número decimal.
toInt(input [, radix])	Convierte el input a un número entero, o `NaN` si el input no es un número entero.
trim(input [, chars])	Recorta caracteres (espacios en blanco por defecto) de ambos lados del input.

ENTENDER EL PROYECTO

1.MODELO

2.CONTROLADOR→ Quien manejará la lógica para obtener la información requerida. Aquí es donde interactúas con la base de datos y procesas los datos según sea necesario

3.MIDDLEWARE o VALIDATION→ Si necesitamos realizar alguna verificación o procesamiento antes de llegar al controlador, como autenticación o validación de datos aquí se hace

4.RUTA→ Establecemos una ruta en tu servidor para manejar la solicitud específica

5.TEST

1.RESTAURANTES

RF Clientes relacionados con Restaurantes

FR1: listado de restaurantes

Los clientes podrán consultar todos los restaurantes.

2.CONTROLADOR

Necesitamos implementar una función en el controlador que nos permita obtener todos los restaurantes:

```
// devuelve todos los restaurantes
const index = async function (req, res) {
  try {
    const restaurants = await Restaurant.findAll(
      {
        attributes: { exclude: ['userId'] },
        include: [
          {
            model: RestaurantCategory,
            as: 'restaurantCategory'
          },
          {
            order: [[{ model: RestaurantCategory, as: 'restaurantCategory' }, 'name', 'ASC']]
          }
        ]
      }
    )
    res.json(restaurants)
  } catch (err) {
    res.status(500).send(err)
  }
}
```

3.RUTA

Necesitamos implementar una ruta para poder obtenerlo, utilizando el método del controlador:

```
// FR1:Los clientes podrán consultar todos los restaurantes.
app.route('/restaurants')
  .get(
    RestaurantController.index)
```

4. MIDDLEWARE O VALIDATION

¿Necesitamos realizar alguna verificación o procesamiento antes de llegar al controlador?

Como es de lectura, se verifica en el propio test que para cada restaurante en `checkRestaurantProperties`

5. TEST

Esta parte es importante para ver como tiene que superar las pruebas lo que hemos hecho (y ver que nos puede faltar).

```
describe('Get all restaurants', () => {
  let restaurants, app
  beforeAll(async () => {
    app = await getApp()
  })
  it('There must be more than one restaurant', async () => {
    const response = await request(app).get('/restaurants').send()
    expect(response.status).toBe(200)
    expect(Array.isArray(response.body)).toBeTruthy()
    expect(response.body).nottoHaveLength(0)
    restaurants = response.body
  })
  it('All restaurants must have an id', async () => {
    expect(restaurants.every(restaurant => restaurant.id !== undefined)).toBe(true)
  })
  it('All restaurants must have a restaurant category', async () => {
    expect(restaurants.every(restaurant => restaurant.restaurantCategory !== undefined)).toBe(true)
  })
  // eslint-disable-next-line jest/expect-expect
  it('All restaurants properties must be correctly defined', async () => {
    restaurants.forEach(restaurant => _checkRestaurantProperties(restaurant))
  })
  afterAll(async () => {
    await shutdownApp()
  })
})
```

Se definen dos variables : `restaurants`, `app` donde la primera almacena la lista de restaurantes que se obtendrá y la segunda la app que se está probando.

Solicitud → GET /restaurants

1. Respuesta 200 + body sea un array + body no vacío
2. Todos los restaurantes tienen id
3. Todos los restaurantes tienen una categoría
4. Todos los restaurantes tienen las propiedades bien definidas

```
> restaurants: [31] [Restaurant, Restaurant, Restaurant, Restaurant, Restaurant, Restaurant, Restaurant, Restaurant, Restaurant]
> 0: Restaurant {dataValues: {}, _previousDataValues: {}, unique: 1, _changed: Set(0), _options: {}, ...}
> 1: Restaurant {dataValues: {}, _previousDataValues: {}, unique: 1, _changed: Set(0), _options: {}, ...}
> 2: Restaurant {dataValues: {}, _previousDataValues: {}, unique: 1, _changed: Set(0), _options: {}, ...}
> 3: Restaurant {dataValues: {}, _previousDataValues: {}, unique: 1, _changed: Set(0), _options: {}, ...}
> 4: Restaurant {dataValues: {}, _previousDataValues: {}, unique: 1, _changed: Set(0), _options: {}, ...}
> 5: Restaurant {dataValues: {}, _previousDataValues: {}, unique: 1, _changed: Set(0), _options: {}, ...}
> 6: Restaurant {dataValues: {}, _previousDataValues: {}, unique: 1, _changed: Set(0), _options: {}, ...}
> 7: Restaurant {dataValues: {}, _previousDataValues: {}, unique: 1, _changed: Set(0), _options: {}, ...}
> 8: Restaurant {dataValues: {}, _previousDataValues: {}, unique: 1, _changed: Set(0), _options: {}, ...}
> 9: Restaurant {dataValues: {}, _previousDataValues: {}, unique: 1, _changed: Set(0), _options: {}, ...}
> 10: Restaurant {dataValues: {}, _previousDataValues: {}, unique: 1, _changed: Set(0), _options: {}, ...}
> 11: Restaurant {dataValues: {}, _previousDataValues: {}, unique: 1, _changed: Set(0), _options: {}, ...}
> 12: Restaurant {dataValues: {}, _previousDataValues: {}, unique: 1, _changed: Set(0), _options: {}, ...}
> 13: Restaurant {dataValues: {}, _previousDataValues: {}, unique: 1, _changed: Set(0), _options: {}, ...}
> 14: Restaurant {dataValues: {}, _previousDataValues: {}, unique: 1, _changed: Set(0), _options: {}, ...}
> 15: Restaurant {dataValues: {}, _previousDataValues: {}, unique: 1, _changed: Set(0), _options: {}, ...}
> 16: Restaurant {dataValues: {}, _previousDataValues: {}, unique: 1, _changed: Set(0), _options: {}, ...}
> 17: Restaurant {dataValues: {}, _previousDataValues: {}, unique: 1, _changed: Set(0), _options: {}, ...}
> 18: Restaurant {dataValues: {}, _previousDataValues: {}, unique: 1, _changed: Set(0), _options: {}, ...}
```

FR2: Detalles y menú de restaurantes

Los clientes podrán consultar los detalles de los restaurantes y los productos que ofrecen.

2. CONTROLADOR

Necesitamos implementar una función en el controlador que nos permita obtener todos los detalles de restaurantes y productos dado un restaurante concreto:

```
// muestra los datos de un restaurante, incluyendo productos + su asociaciones
const show = async function (req, res) {
  try {
    const restaurant = await Restaurant.findById(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'] }
      ]
    })
    res.json(restaurant)
  } catch (err) {
    res.status(500).send(err)
  }
}
```

:

```
> Block:show
< restaurant: Restaurant {dataValues: {}, _previousDataValues: {}, uniqno: 1, _changed: Set(), _options: {}}, ...
| > _changed: Set(0) {size: 0}
| > _options: {isNewRecord: false, _schema: null, _schemaDelimiter: ''}, include: Array(2), includeNames: Array(2), ...
| > _previousDataValues: {id: 1, name: 'Valid Restaurant', description: 'Cocina Tradicional', address: '123 Valid Street', postalCode: '12345', ...}
| > address: & f () {\n    return this.get(attribute);\n  }
| > averageServiceMinutes: & f () {\n    return this.get(attribute);\n  }
| > createdAt: & f () {\n    return this.get(attribute);\n  }
| > dataValues: {id: 1, name: 'Valid Restaurant', description: 'Cocina Tradicional', address: '123 Valid Street', postalCode: '12345', ...}
| > description: & f () {\n    return this.get(attribute);\n  }
| > email: & f () {\n    return this.get(attribute);\n  }
| > heroImage: & f () {\n    return this.get(attribute);\n  }
| > id: & f () {\n    return this.get(attribute);\n  }
| > isNewRecord: false
| > logo: & f () {\n    return this.get(attribute);\n  }
| > name: & f () {\n    return this.get(attribute);\n  }
| > phone: & f () {\n    return this.get(attribute);\n  }
| > postalCode: & f () {\n    return this.get(attribute);\n  }
| > products: (7) [Product, Product, Product]
| > restaurantCategory: RestaurantCategory {dataValues: {}, _previousDataValues: {}, uniqno: 1, _changed: Set(), _options: {}}, ...
| > restaurantCategoryId: & f () {\n    return this.get(attribute);\n  }
| > sequelize: & f sequelize() {\n    return this.constructor.sequelize;\n  }
| > shippingCosts: & f () {\n    return this.get(attribute);\n  }
| > status: & f () {\n    return this.get(attribute);\n  }
| > uniqno: 1
| > updatedAt: & f () {\n    return this.get(attribute);\n  }
| > url: & f () {\n    return this.get(attribute);\n  }
| > userId: & f () {\n    return this.get(attribute);\n  }
> [[Prototype]]: Model
this: undefined
```

3.RUTA

Necesitamos implementar una ruta para poder obtenerlo. **Siempre que queramos ver una entidad concreta vamos a preguntar si existe: checkEntityExists**

```
app.route('/restaurants/:restaurantId')
  .get(
    checkEntityExists(Restaurant, 'restaurantId'),
    RestaurantController.show
  )
```

4. MIDDLEWARE O VALIDATION

¿Necesitamos realizar alguna verificación o procesamiento antes de llegar al controlador? Como es de lectura, se verifica en el propio test

5. TEST

```
describe('Get restaurant details', () => {
  let restaurantIdToBeShown, app, returnedRestaurant
  beforeAll(async () => {
    app = await getApp()
    restaurantIdToBeShown = 1
  })
  it('Should return 404 with incorrect id', async () => {
    const response = await request(app).get('/restaurants/incorrectId').send()
    expect(response.status).toBe(404)
  })
  it('Should return 200 with correct id', async () => {
    const response = await request(app).get(`/restaurants/${restaurantIdToBeShown}`).send()
    expect(response.status).toBe(200)
    returnedRestaurant = response.body
  })
  it('The restaurant must not include its userId', async () => {
    expect(returnedRestaurant.userId === undefined).toBe(true)
  })
  // eslint-disable-next-line jest/expect-expect
  it('All restaurants properties must be correctly defined', async () => {
    _checkRestaurantProperties(returnedRestaurant)
  })
  it('The restaurant must include products', async () => {
    expect(returnedRestaurant.products !== undefined).toBe(true)
  })
  it('The products of the restaurant must not be empty', async () => {
    expect(returnedRestaurant.products.length > 0).toBe(true)
  })
  it('The restaurant must have a restaurant category', async () => {
    expect(returnedRestaurant.restaurantCategory !== undefined).toBe(true)
  })
  it('The products the restaurant must have a product category', async () => {
    expect(returnedRestaurant.products.every(product => product.productCategory !== undefined)).toBe(true)
  })
  afterAll(async () => {
    await shutdownApp()
  })
})
```

Se definen tres variables `restaurantIdToBeShown`, `app`, `returnedRestaurant`, el id del restaurante con el que vamos a comprobar cosas, app y el restaurante en si:

1. 404 si el id es incorrecto
2. 200 si el id es correcto
3. El restaurante no debe de añadir el user Id
4. Las propiedades del restaurante tiene que estar bien definidas
5. El restaurante debe incluir productos
6. Los productos no pueden estar vacíos
7. El restaurante tiene una categoría
8. El restaurante tiene una categoría de producto

RF Propietario relacionados con Restaurantes

Como propietario de un restaurante, el sistema debe proporcionar las siguientes funcionalidades:

FR1: agregar, enumerar, editar y eliminar restaurantes

Los restaurantes están relacionados con un propietario, por lo que los propietarios pueden realizar estas operaciones a los restaurantes de su propiedad. Si un propietario crea un Restaurante, automáticamente quedará relacionado (de propiedad) con él. Si se elimina un restaurante, también se deben eliminar todos sus productos.

1. CONTROLADOR Tenemos que implementar las operaciones CRUD en el controlador:

LEER:

```
const indexOwner = async function (req, res) {
  try {
    const restaurants = await Restaurant.findAll(
      {
        attributes: { exclude: ['userId'] },
        where: { userId: req.user.id },
        include: [
          {
            model: RestaurantCategory,
            as: 'restaurantCategory'
          }
        ]
      }
    )
    res.json(restaurants)
  } catch (err) {
    res.status(500).send(err)
  }
}
```

CREAR:

```
const create = async function (req, res) {
  const newRestaurant = Restaurant.build(req.body)
  newRestaurant.userId = req.user.id // usuario actualmente autenticado
  try {
    const restaurant = await newRestaurant.save()
    res.json(restaurant)
  } catch (err) {
    res.status(500).send(err)
  }
}
```

ACTUALIZAR:

```
//actualiza un restaurante
const update = async function (req, res) {
  try {
    await Restaurant.update(req.body, { where: { id: req.params.restaurantId } })
    const updatedRestaurant = await Restaurant.findByPk(req.params.restaurantId)
    res.json(updatedRestaurant)
  } catch (err) {
    res.status(500).send(err)
  }
}
```

BORRAR:

```
const destroy = async function (req, res) {
  try {
    const result = await Restaurant.destroy({ where: { id: req.params.restaurantId } })
    let message = ''
    if (result === 1) {
      message = 'Successfully deleted restaurant id.' + req.params.restaurantId
    } else {
      message = 'Could not delete restaurant.'
    }
    res.json(message)
  } catch (err) {
    res.status(500).send(err)
  }
}
```

3.RUTA

```
app.route('/restaurants')
  .post(
    isLoggedIn,
    hasRole('owner'),
    handleFilesUpload(['logo', 'heroImage'], process.env.RESTAURANTS_FOLDER),
    RestaurantValidation.create,
    handleValidation,
    RestaurantController.create)
```

4.MIDDLEWARE O VALIDATION

¿Necesitamos realizar alguna verificación o procesamiento antes de llegar al controlador? Se ve claramente viendo el test

LEER: No, solo que se verifiquen las propiedades

- CREAR:**Si**:
- Usuario tiene que estar loggeado
 - Tiene que ser un owner es decir un rol específico
 - Tiene que tener validado los valores

- ACTUALIZAR:**Si**
- Tiene que existir el restaurante que queremos actualizar
 - Usuario tiene que estar loggeado
 - Tiene que ser un owner y tiene que ser su restaurante
 - Tiene que tener validado los valores

- BORRAR: **Si**
- Tiene que existir el restaurante que queremos actualizar
 - Usuario tiene que estar loggeado
 - Tiene que ser un owner y tiene que ser su restaurante
 - No se puede borrar un restaurante con pedidos
 - No se puede borrar un restaurante ya borrado = tiene que existir

MIDDLEWARE

Tenemos que implementar funciones que nos permitan comprobar esto:

“Tiene que existir el restaurante que queremos actualizar”:

```
const checkEntityExists = (model, idPathParamName) => async (req, res, next) => {
  try {
    const entity = await model.findByPk(req.params[idPathParamName])
    if (!entity) { return res.status(404).send('Not found') }
    return next()
  } catch (err) {
    return res.status(500).send(err)
  }
}
```

“Usuario tiene que estar loggeado”

```
const isLoggedIn = (req, res, next) => {
  | passport.authenticate('bearer', { session: false })(req, res, next)
}
```

“Tiene que ser un owner es decir un rol especifico”

```
const hasRole = (...roles) => (req, res, next) => {
  | if (!req.user) {
  |   return res.status(403).send({ error: 'Not logged in' })
  }
  | if (!roles.includes(req.user.userType)) {
  |   return res.status(403).send({ error: 'Not enough privileges' })
  }
  | return next()
}
```

“Tiene que ser su restaurante”

```
const checkRestaurantOwnership = async (req, res, next) => {
  try {
    | const restaurant = await Restaurant.findByPk(req.params.restaurantId)
    | if (req.user.id === restaurant.userId) {
    |   return next()
    }
    | return res.status(403).send('Not enough privileges. This entity does not belong to you')
  } catch (err) {
    | return res.status(500).send(err)
  }
}
```

“No se puede borrar un restaurante con pedidos”

```
const restaurantHasNoOrders = async (req, res, next) => {
  try {
    | const numberOfRestaurantOrders = await Order.count({
    |   where: { restaurantId: req.params.restaurantId }
    })
    | if (numberOfRestaurantOrders === 0) {
    |   return next()
    }
    | return res.status(409).send('Some orders belong to this restaurant.')
  } catch (err) {
    | return res.status(500).send(err.message)
  }
}
```

VALIDATION

Hay que comprobar el formato correcto:

```
const 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')
]
```

```

const 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')
]

```

5.TEST

```

describe('Get owner restaurants', () => {
  let owner, ownerRestaurants, app
  const fakeToken = 'fakeToken'
  beforeAll(async () => {
    app = await getApp()
    owner = await getLoggedInOwner()
  })
  it('should not be able to retrieve restaurants with a fake token', async () => {
    const response = await request(app).get('/users/myRestaurants').set('Authorization', `Bearer ${fakeToken}`).send()
    expect(response.status).toBe(401)
  })
  it('should be able to retrieve restaurants with the real token', async () => {
    const response = await request(app).get('/users/myRestaurants').set('Authorization', `Bearer ${owner.token}`).send()
    expect(response.status).toBe(200)
    ownerRestaurants = response.body
  })
  it('The owner must have more than one restaurant', async () => {
    expect(Array.isArray(ownerRestaurants)).toBeTruthy()
    expect(ownerRestaurants).not.toHaveLength(0)
  })
  it('All owner restaurants must have an id', async () => {
    expect(ownerRestaurants.every(restaurant => restaurant.id !== undefined)).toBe(true)
  })
  it('All owner restaurants must have a restaurant category', async () => {
    expect(ownerRestaurants.every(restaurant => restaurant.restaurantCategory !== undefined)).toBe(true)
  })
  afterAll(async () => {
    await shutdownApp()
  })
})

```

```

describe('Create restaurant', () => {
  let owner, customer, app
  beforeAll(async () => {
    app = await getApp()
    owner = await getLoggedInOwner()
    customer = await getLoggedInCustomer()
  })
  it('Should return 401 if not logged in', async () => {
    const response = await request(app).post('/restaurants').send(invalidRestaurant)
    expect(response.status).toBe(401)
  })
  it('Should return 403 when logged in as a customer', async () => {
    const response = await request(app).post('/restaurants').set('Authorization', `Bearer ${customer.token}`).send(invalidRestaurant)
    expect(response.status).toBe(403)
  })
  it('Should return 200 when valid restaurant', async () => {
    const validRestaurant = { ...bodeguitaRestaurant }
    validRestaurant.restaurantCategoryId = (await request(app).get('/restaurantCategories').send()).body[0].id
    const response = await request(app).post('/restaurants').set('Authorization', `Bearer ${owner.token}`).send(validRestaurant)
    expect(response.status).toBe(200)
  })
  afterAll(async () => {
    await shutdownApp()
  })
})

```

```

describe('Edit restaurant', () => {
  let owner, customer, newRestaurant, app
  beforeEach(async () => {
    app = await getApp()
    owner = await getLoggedInOwner()
    customer = await getLoggedInCustomer()
    const validRestaurant = { ...bodeguitaRestaurant }
    validRestaurant.restaurantCategoryId = (await request(app).get('/restaurantCategories').send()).body[0].id
    newRestaurant = (await request(app).post('/restaurants').set('Authorization', `Bearer ${owner.token}`).send(validRestaurant)).body
  })
  it('Should return 401 if not logged in', async () => {
    const response = await request(app).put('/restaurants/${newRestaurant.id}`).send(invalidRestaurant)
    expect(response.status).toBe(401)
  })
  it('Should return 403 when logged in as a customer', async () => {
    const response = await request(app).put('/restaurants/${newRestaurant.id}`).set('Authorization', `Bearer ${customer.token}`).send(invalidRestaurant)
    expect(response.status).toBe(403)
  })
  it('Should return 403 when trying to edit a restaurant that is not yours', async () => {
    const restaurantNotOwned = await createRestaurant()
    const { userId, ...editedRestaurantNotOwned } = restaurantNotOwned
    editedRestaurantNotOwned.name = `${editedRestaurantNotOwned.name} updated`
    const response = await request(app).put('/restaurants/${restaurantNotOwned.id}`).set('Authorization', `Bearer ${owner.token}`).send(editedRestaurantNotOwned)
    expect(response.status).toBe(403)
  })
  it('Should return 200 when valid restaurant', async () => {
    const { userId, ...editedRestaurant } = newRestaurant
    editedRestaurant.name = `${newRestaurant.name} updated`
    const response = await request(app).put('/restaurants/${newRestaurant.id}`).set('Authorization', `Bearer ${owner.token}`).send(editedRestaurant)
    expect(response.status).toBe(200)
    expect(response.body.name).toBe(`${newRestaurant.name} updated`)
  })
  afterAll(async () => {
    await shutdownApp()
  })
})

describe('Remove restaurant', () => {
  let owner, customer, newRestaurant, app
  beforeEach(async () => {
    app = await getApp()
    owner = await getLoggedInOwner()
    customer = await getLoggedInCustomer()
    const validRestaurant = { ...bodeguitaRestaurant }
    validRestaurant.restaurantCategoryId = (await request(app).get('/restaurantCategories').send()).body[0].id
    newRestaurant = (await request(app).post('/restaurants').set('Authorization', `Bearer ${owner.token}`).send(validRestaurant)).body
  })
  it('Should return 401 if not logged in', async () => {
    const response = await request(app).delete('/restaurants/${newRestaurant.id}`).send()
    expect(response.status).toBe(401)
  })
  it('Should return 403 when logged in as a customer', async () => {
    const response = await request(app).delete('/restaurants/${newRestaurant.id}`).set('Authorization', `Bearer ${customer.token}`).send()
    expect(response.status).toBe(403)
  })
  it('Should return 403 when trying to delete a restaurant that is not yours', async () => {
    const restaurantNotOwned = await createRestaurant()
    const response = await request(app).delete('/restaurants/${restaurantNotOwned.id}`).set('Authorization', `Bearer ${owner.token}`).send()
    expect(response.status).toBe(403)
  })
  it('Should return 409 when removing a restaurant with orders', async () => {
    const restaurantWithOrders = (await request(app).get('/users/myRestaurants').set('Authorization', `Bearer ${owner.token}`)).body[0]
    const response = await request(app).delete('/restaurants/${restaurantWithOrders.id}`).set('Authorization', `Bearer ${owner.token}`).send()
    expect(response.status).toBe(409)
  })
  it('Should return 200 when valid restaurant', async () => {
    const response = await request(app).delete('/restaurants/${newRestaurant.id}`).set('Authorization', `Bearer ${owner.token}`).send()
    expect(response.status).toBe(200)
  })
  it('Should return 404 when trying to delete a restaurant already deleted', async () => {
    const response = await request(app).delete('/restaurants/${newRestaurant.id}`).set('Authorization', `Bearer ${owner.token}`).send()
    expect(response.status).toBe(404)
  })
  afterAll(async () => {
    await shutdownApp()
  })
})

```

PROYECTO-PARTE BACKEND

FR3: Agregar, editar y eliminar productos a un nuevo pedido.

Un cliente puede agregar varios productos y varias unidades de un producto a un nuevo pedido. Antes de confirmar, el cliente puede editar y eliminar productos.

- // 1. Si el precio es superior a 10€ los gastos de envío tienen que ser 0.
- // 2. Si el precio es menor o igual a 10€, los gastos de envío deben ser los gastos de envío predeterminados del restaurante y deben sumarse al precio total del pedido.
- // 3. Para guardar el pedido y los productos relacionados, inicie una transacción, almacene el pedido, almacene cada línea de producto y confirme la transacción.
- // 4. Si se genera una excepción, capturarla y revertir la transacciónA tener en cuenta:
- // 1. Si el precio es superior a 10€ los gastos de envío tienen que ser 0.
- // 2. Si el precio es menor o igual a 10€, los gastos de envío deben ser los gastos de envío predeterminados del restaurante y deben sumarse al precio total del pedido.
- // 3. Para guardar el pedido y los productos relacionados, inicie una transacción, almacene el pedido, almacene cada línea de producto y confirme la transacción.
- // 4. Si se genera una excepción, capturarla y revertir la transacción

FR5: Listar mis pedidos confirmados.

Un Cliente podrá consultar sus pedidos confirmados, ordenados del más reciente al más antiguo.

FR8: Editar/eliminar orden.

Si el pedido está en estado 'pendiente', el cliente puede editar o eliminar los productos incluidos o eliminar todo el pedido.

La dirección de entrega también se puede modificar en el estado 'pendiente'.

Si el pedido está en el estado 'enviado' o 'entregado' no se permite edición ni eliminación

2.CONTROLADOR

3.MIDDLEWARE o VALIDATION

4.RUTA

5.TEST

FR5: Listar mis pedidos confirmados.

Un Cliente podrá consultar sus pedidos confirmados, ordenados del más reciente al más antiguo.

2.CONTROLADOR

Tenemos que implementar una función en el controlador que nos permita LEER los pedidos confirmados ordenador del pedido mas reciente al pedido más antiguo. Yo por ejemplo aquí no había excluido userId y me daba error y no sabía por que era:

```
const indexCustomer = async function (req, res) {
  try {
    const orders = await Order.findAll({
      where: {
        userId: req.user.id
      },
      include: [
        {
          model: Product,
          as: 'products'
        },
        {
          model: Restaurant,
          as: 'restaurant',
          attributes: { exclude: ['userId'] }
        },
        order: [['createdAt', 'DESC']]
      }
    })
    res.json(orders)
  } catch (err) {
    res.status(500).send(err)
  }
}
```

IMPORTANTE

Siempre que tengamos que añadir info y este el atributo userId, tenemos que excluirlo pq si no nos va a dar fallo

3.TEST

```
describe('Get customer orders', () => {
  let customer, owner, customerOrders, app
  beforeAll(async () => {
    customer = await getLoggedInCustomer()
    owner = await getLoggedInOwner()
    app = await getApp()
  })
  it('Should return 401 if not logged in', async () => {
    const response = await request(app).get('/orders').send()
    expect(response.status).toBe(401)
  })
  it('Should return 403 when logged in as an owner', async () => {
    const response = await request(app).get('/orders').set('Authorization', `Bearer ${owner.token}`).send({})
    expect(response.status).toBe(403)
  })
  it('Should return 200 when logged in as a customer', async () => {
    const response = await request(app).get('/orders').set('Authorization', `Bearer ${customer.token}`).send()
    customerOrders = response.body
    expect(response.status).toBe(200)
  })
  it('All orders must have an id', async () => {
    expect(customerOrders.every(order => order.id !== undefined)).toBe(true)
  })
  it('All orders must have products', async () => {
    expect(customerOrders.every(order => order.products !== undefined)).toBe(true)
  })
  it('All orders must belong to the customer', async () => {
    expect(customerOrders.every(order => order.userId === customer.id)).toBe(true)
  })
  afterAll(async () => {
    await shutdownApp()
  })
})
```

4. MIDDLEWARE o VALIDATION

Analizamos el test:

- Tiene que estar loggeado
- Tiene que tener el rol de customer
- Tiene que tener id y productos los pedidos
- Todas las ordenes pertenecen al cliente

IMPORTANTE

Yo por ejemplo aquí he confundido la ultima comprobacion del test con el middleware `checkOrderCustomer`

TEST: verifica que todas las order obtenidas pertenezcan al cliente

MIDDLEWARE: la orden concreta pertenece al usuario autenticado

5. RUTA

```
app.route('/orders')
  .get(
    isLoggedIn,
    hasRole('customer'),
    OrderController.indexCustomer)
```

6. RESULTADO

Vemos el formato de la **petición** para entender todo un poco mejor. La petición tiene un montón de cosas nos centramos en las mas importantes:

- **req.user** representa el usuario que inició sesión y realizó la solicitud.

```
> user: User {dataValues: {}, _previousDataValues: {}, uniqno: 1, _changed: Set(), _options: {}}
  > _changed: Set() {size: 0}
  > _options: {isNewRecord: false, _schema: null, _schemaDelimiter: '', raw: true, attributes: Array(14)}
  > _previousDataValues: {id: 1, firstName: 'Customer 1', lastName: 'Fake 1', email: 'customer1@customer.com', password: '$2a$05$LD523YvMgleSUD6inLmYy.Q4LN6iJx.YXC38sM7B1U/HYxG4H/11', ...}
    address: 'Fake street 123'
    avatar: 'public/avatars/maleAvatar.png'
    createdAt: Mon Jun 17 2024 12:10:21 GMT+0200 (hora de verano de Europa central)
    email: 'customer1@customer.com'
    firstName: 'Customer 1'
    id: 1
    lastName: 'Fake 1'
    password: '$2a$05$LD523YvMgleSUD6inLmYy.Q4LN6iJx.YXC38sM7B1U/HYxG4H/11'
    phone: '+3466677888'
    postalCode: '41010'
    token: '0a537a3e442b9f7e8b48ed49a48c4d3e77278f8b'
    tokenExpiration: Mon Jun 17 2024 17:53:47 GMT+0200 (hora de verano de Europa central)
    updatedAt: Mon Jun 17 2024 16:53:46 GMT+0200 (hora de verano de Europa central)
    userType: 'customer'
```

```
  method: 'GET'
> next: f next(err)
  originalUrl: '/orders'
```

Vemos el formato de la **respuesta**, que son los pedidos:

```
> orders: (25) [Order, Order, Order, Order, Order, Order, Order, Order, Order, Order, ...
  > 0: Order {dataValues: {}, _previousDataValues: {}, uniqno: 1, _changed: Set(), _options: {}}
  > 1: Order {dataValues: {}, _previousDataValues: {}, uniqno: 1, _changed: Set(), _options: {}}
  > 2: Order {dataValues: {}, _previousDataValues: {}, uniqno: 1, _changed: Set(), _options: {}}
  > 3: Order {dataValues: {}, _previousDataValues: {}, uniqno: 1, _changed: Set(), _options: {}}
  > 4: Order {dataValues: {}, _previousDataValues: {}, uniqno: 1, _changed: Set(), _options: {}}
  > 5: Order {dataValues: {}, _previousDataValues: {}, uniqno: 1, _changed: Set(), _options: {}}
  > 6: Order {dataValues: {}, _previousDataValues: {}, uniqno: 1, _changed: Set(), _options: {}}
  > 7: Order {dataValues: {}, _previousDataValues: {}, uniqno: 1, _changed: Set(), _options: {}}
  > 8: Order {dataValues: {}, _previousDataValues: {}, uniqno: 1, _changed: Set(), _options: {}}
  > 9: Order {dataValues: {}, _previousDataValues: {}, uniqno: 1, _changed: Set(), _options: {}}
  > 10: Order {dataValues: {}, _previousDataValues: {}, uniqno: 1, _changed: Set(), _options: {}}
  > 11: Order {dataValues: {}, _previousDataValues: {}, uniqno: 1, _changed: Set(), _options: {}}
  > 12: Order {dataValues: {}, _previousDataValues: {}, uniqno: 1, _changed: Set(), _options: {}}
  > 13: Order {dataValues: {}, _previousDataValues: {}, uniqno: 1, _changed: Set(), _options: {}}
  > 14: Order {dataValues: {}, _previousDataValues: {}, uniqno: 1, _changed: Set(), _options: {}}
  > 15: Order {dataValues: {}, _previousDataValues: {}, uniqno: 1, _changed: Set(), _options: {}}
  > 16: Order {dataValues: {}, _previousDataValues: {}, uniqno: 1, _changed: Set(), _options: {}}
  > 17: Order {dataValues: {}, _previousDataValues: {}, uniqno: 1, _changed: Set(), _options: {}}
  > 18: Order {dataValues: {}, _previousDataValues: {}, uniqno: 1, _changed: Set(), _options: {}}
```

FR3: Agregar, editar y eliminar productos a un nuevo pedido.

Un cliente puede agregar varios productos y varias unidades de un producto a un nuevo pedido. Antes de confirmar, el cliente puede editar y eliminar productos.

- // 1. Si el precio es superior a 10€ los gastos de envío tienen que ser 0.
- // 2. Si el precio es menor o igual a 10€, los gastos de envío deben ser los gastos de envío predeterminados del restaurante y deben sumarse al precio total del pedido.
- // 3. Para guardar el pedido y los productos relacionados, inicie una transacción, almacene el pedido, almacene cada línea de producto y confírmelo la transacción.
- // 4. Si se genera una excepción, capturarla y revertir la transacción
- // 1. Si el precio es superior a 10€ los gastos de envío tienen que ser 0.
- // 2. Si el precio es menor o igual a 10€, los gastos de envío deben ser los gastos de envío predeterminados del restaurante y deben sumarse al precio total del pedido.
- // 3. Para guardar el pedido y los productos relacionados, inicie una transacción, almacene el pedido, almacene cada línea de producto y confírmelo la transacción.
- // 4. Si se genera una excepción, capturarla y revertir la transacción

Vemos el formato de la **petición** para entender todo un poco mejor. La petición tiene un montón de cosas nos centramos en las mas importantes:

- **req.body** representa los datos que provienen del cliente:

```
▼ body: {address: 'Calle falsa 123', restaurantId: 1, products: Array(2)}  
  address: 'Calle falsa 123'  
  ▼ products: (2) [ { - }, { - } ]  
    > 0: {productId: 3431, quantity: 1}  
    > 1: {productId: 3432, quantity: 8}  
    length: 2  
    > [[Prototype]]: Array(0)  
    | > [[Prototype]]: Object  
    restaurantId: 1
```

2. CONTROLADOR

Cuando estamos creando un pedido (order) que incluye otro modelo (products) en una BD guardamos ambos modelos dentro de una transacción para asegurar la consistencia de los datos. Es decir, todo hay que guardar.

Vemos el modelo de Orders y vemos que no hay un modelo específico para OrderProducts si no que esta definido en el propio modelo de Orders. Vemos que:

```
static associate (models) {  
  const OrderProducts = sequelize.define('OrderProducts', {  
    quantity: DataTypes.INTEGER,  
    unityPrice: DataTypes.DOUBLE  
  })  
  
  Order.belongsTo(models.Restaurant, { foreignKey: 'restaurantId', as: 'restaurant' })  
  Order.belongsTo(models.User, { foreignKey: 'userId', as: 'user' })  
  Order.belongsToMany(models.Product, { as: 'products', through: OrderProducts }, { onDelete: 'cascade' })  
}
```

```

const create = async (req, res) => {
  // Iniciamos transaccion
  const transaction = await sequelizeSession.transaction()
  // Creamos el order con la info(sin productos)
  const newOrder = Order.build(req.body)
  try {
    // Calculamos el precio del pedido. El precio del pedido es la suma de todos los productos x cantidad
    let price = 0
    for (const product of req.body.products) {
      const productoBD = await Product.findByPk(product.productId)
      price += productoBD.price * product.quantity
    }
    newOrder.price = price

    // Obtenemos el restaurante del pedido
    const restaurant = await Restaurant.findByPk(newOrder.restaurantId)
    newOrder.shippingCosts = price > 10 ? 0 : restaurant.shippingCosts + price

    newOrder.userId = req.user.id
    newOrder.shippingCosts = price > 10 ? 0 : restaurant.shippingCosts

    // Guardamos el pedido en la BD
    const order = await newOrder.save({ transaction })

    // Asociamos los productos al pedido y guardamos en BD
    for (const product of req.body.products) {
      const prodBD = await Product.findByPk(product.productId)
      await order.addProduct(prodBD, {
        through: {
          quantity: product.quantity,
          unityPrice: prodBD.price
        },
        transaction
      })
    }
    await transaction.commit()

    const finalOrder = await Order.findByPk(order.id, {
      include: ['products']
    })
    res.json(finalOrder.dataValues)
  } catch (err) {
    await transaction.rollback()
    res.status(500).send('ERROR EN EL CONTROLLER')
  }
}

```

1. Aunque la tabla Orders no tiene una columna products, la relación entre Orders y Products está definida a través de la tabla intermedia OrderProducts. Cuando incluimos products en la consulta, estás diciendo a Sequelize que también obtenga los productos relacionados a través de la tabla intermedia.

2. El método addProduct es una función generada automáticamente por los métodos de asociación en Sequelize cuando defines una relación "muchos a muchos" dos modelos utilizando una tabla intermedia:

```

await order.addProduct(prodBD, {
  through: {
    quantity: product.quantity,
    unityPrice: prodBD.price
  },
  transaction
})

```

productDB: Es la instancia del producto que se encontró en la base de datos.

through: Especifica los atributos adicionales que se deben almacenar en la tabla intermedia, en este caso, la cantidad del producto y el precio unitario.

3. MIDDLEWARE o VALIDATION

VALIDATION

1. Verificar que restaurantId esté presente en el cuerpo y corresponda a un restaurante existente
2. Verificar que productos sea una matriz no vacía compuesta por objetos con productId y cantidad mayor que 0
3. Verificar que los productos estén disponibles-
4. Comprobar que todos los productos pertenecen al mismo restaurante

```
const checkCreationOrder2 = async (value, { req }) => {
  try {
    // 1
    if (req.body.products.length < 1) {
      return Promise.reject(new Error(' El array de productos no puede estar vacio'))
    }
    // 2
    for (const product of value) {
      if (product.productId < 1) {
        return Promise.reject(new Error(' El id no puede ser 0'))
      }
      const productBD = await Product.findByPk(product.productId)
      // 3
      if (!productBD.availability) { return Promise.reject(new Error(' Todos los productos tienen que estar disponibles')) }

      // 4
      if (productBD.restaurantId !== req.body.restaurantId) { return Promise.reject(new Error(' Todos los productos tienen que estar disponibles')) }
    }
    return Promise.resolve()
  } catch (err) {
    return Promise.reject(new Error(err))
  }
}

const create = [
  // 1. Check that restaurantId is present in the body and corresponds to an existing restaurant
  check('restaurantId').exists(),
  check('price').default(null).optional({ nullable: true }).isFloat().toFloat(),
  check('address').exists().isString().isLength({ min: 1, max: 255 }).trim(),
  check('products').custom(checkCreationOrder2),
  // 2- and quantity greater than 0
  check('products.*.quantity').isInt({ min: 1 }).toInt()
]
```

4. RUTA

```
app.route('/orders')
  .post(
    isLoggedIn,
    hasRole('customer'),
    OrderMiddleware.checkRestaurantExists,
    OrderValidation.create,
    handleValidation,
    OrderController.create)
```

5.TEST

- Loggeado
- Customer
 - Tiene que existir el restaurante
 - Tiene que tener productos, no 0, no cantidad negativa...
 - Tiene que pedir productos disponibles
 - Solo pedidos de un restaurantes

```
describe('Create order', () => {
  let customer, restaurant, anotherRestaurant, orderData, newOrderData, createdOrderId, app, owner
  beforeAll(async () => {
    app = await getApp()
    customer = await getLoggedInCustomer()
    owner = await getLoggedInOwner()
    restaurant = await getFirstRestaurantOfOwner(owner)
    anotherRestaurant = await createRestaurant(owner)
    orderData = await getNewOrderData(restaurant)
  })
  it('Should return 401 if not logged in', async () => {
    const response = await request(app).post('/orders').send(orderData)
    expect(response.status).toBe(401)
  })
  it('Should return 403 when logged in as an owner', async () => {
    const response = await request(app).post('/orders').set('Authorization', `Bearer ${owner.token}`).send(orderData)
    expect(response.status).toBe(403)
  })
  it('Should return 409 when trying to order from nonexistent restaurant', async () => {
    const invalidOrder = { ...orderData }
    invalidOrder.restaurantId = 'invalidId'
    const response = await request(app).post('/orders').set('Authorization', `Bearer ${customer.token}`).send(invalidOrder)
    expect(response.status).toBe(409)
  })
  it('Should return 422 when trying to order without products', async () => {
    const invalidOrder = { ...orderData }
    delete invalidOrder.products
    const response = await request(app).post('/orders').set('Authorization', `Bearer ${customer.token}`).send(invalidOrder)
    const errorFields = response.body.errors.map(error => error.param)
    expect(['products'].every(field => errorFields.includes(field))).toBe(true)
    expect(response.status).toBe(422)
  })
  it('Should return 422 when trying to order with a non-array of products', async () => {
    const invalidOrder = { ...orderData }
    invalidOrder.products = 2
    const response = await request(app).post('/orders').set('Authorization', `Bearer ${customer.token}`).send(invalidOrder)
    const errorFields = response.body.errors.map(error => error.param)
    expect(['products'].every(field => errorFields.includes(field))).toBe(true)
    expect(response.status).toBe(422)
  })
  it('Should return 422 when trying to order products with quantity 0', async () => {
    const invalidOrder = { ...orderData }
    invalidOrder.products = invalidOrder.products.map(product => ({ ...product, quantity: 0 }))
    const response = await request(app).post('/orders').set('Authorization', `Bearer ${customer.token}`).send(invalidOrder)
    const errorFields = response.body.errors.map(error => error.param)
    // Comprobar que errorFields contiene una cadena con formato 'products[*].quantity', donde * puede ser cualquier caracter
    expect(errorFields.some(field => field.match(/^products\[.*\].quantity$/))).toBe(true)
    expect(response.status).toBe(422)
  })
})
```

```

it('Should return 422 when trying to order products with negative quantity', async () => {
  const invalidOrder = { ...orderData }
  invalidOrder.products = invalidOrder.products.map(product => ({ ...product, quantity: -1 }))
  const response = await request(app).post('/orders').set('Authorization', `Bearer ${customer.token}`).send(invalidOrder)
  const errorFields = response.body.errors.map(error => error.param)
  // Comprobar que errorFields contiene una cadena con formato 'products[*].quantity, donde * puede ser cualquier caracter'
  expect(errorFields.some(field => field.match(/\^products\[.*\]\.quantity$/))).toBe(true)
  expect(response.status).toBe(422)
})
it('Should return 422 when trying to order unavailable products', async () => {
  const unavailableOrderData = await getNewOrderDataWithUnavailableProduct(restaurant)
  const invalidOrder = { ...unavailableOrderData }
  const response = await request(app).post('/orders').set('Authorization', `Bearer ${customer.token}`).send(invalidOrder)
  const errorFields = response.body.errors.map(error => error.param)
  // Comprobar que errorFields contiene una cadena con formato 'products[*].quantity, donde * puede ser cualquier caracter'
  expect(['products'].every(field => errorFields.includes(field))).toBe(true)
  expect(response.status).toBe(422)
})
it('Should return 422 when trying to order from different restaurants', async () => {
  const invalidOrder = await getNewOrderData(restaurant)
  const productFromAnotherRestaurant = await createProduct(owner, anotherRestaurant)
  invalidOrder.products.push({ productId: productFromAnotherRestaurant.id, quantity: 1 })
  const response = await request(app).post('/orders').set('Authorization', `Bearer ${customer.token}`).send(invalidOrder)
  const errorFields = response.body.errors.map(error => error.param)
  // Comprobar que errorFields contiene una cadena con formato 'products[*].quantity, donde * puede ser cualquier caracter'
  expect(['products'].every(field => errorFields.includes(field))).toBe(true)
  expect(response.status).toBe(422)
})
it('Should return 422 when invalid order data', async () => {
  const invalidOrder = await getNewOrderData(restaurant)
  delete invalidOrder.address
  invalidOrder.products[0].productId = 'invalidId'
  const response = await request(app).post('/orders').set('Authorization', `Bearer ${customer.token}`).send(invalidOrder)
  expect(response.status).toBe(422)
  const errorFields = response.body.errors.map(error => error.param)
  expect(['products', 'address'].every(field => errorFields.includes(field))).toBe(true)
})
it('Should return 200 and the correct created order when valid order', async () => {
  newOrderData = await getNewOrderData(restaurant, 15)
  const response = await request(app).post('/orders').set('Authorization', `Bearer ${customer.token}`).send(newOrderData)
  expect(response.status).toBe(200)
  expect(response.body.id).toBeDefined()
  expect(response.body.price).toBe(await computeOrderPrice(response.body))
  await checkOrderEqualsOrderData(response.body, newOrderData)
  createdOrderId = response.body.id
})
it('Should return 200 and the correct created order when requesting order details', async () => {
  const response = await request(app).get(`/orders/${createdOrderId}`).set('Authorization', `Bearer ${customer.token}`).send()
  expect(response.status).toBe(200)
  expect(response.body.id).toBeDefined()
  expect(response.body.price).toBe(await computeOrderPrice(response.body))
  await checkOrderEqualsOrderData(response.body, newOrderData)
})
it('Create order <10€ that should add shipping costs. Should return 200 and the correct created order adding shipping costs', async () => {
  const newOrderData = await getNewOrderData(restaurant, 3)
  const response = await request(app).post('/orders').set('Authorization', `Bearer ${customer.token}`).send(newOrderData)
  expect(response.status).toBe(200)
  expect(response.body.id).toBeDefined()
  expect(response.body.price).toBe(await computeOrderPrice(response.body))
  await checkOrderEqualsOrderData(response.body, newOrderData)
  createdOrderId = response.body.id
})
it('Should return 200 and the correct created order with added shipping costs when requesting order details', async () => {
  const response = await request(app).get(`/orders/${createdOrderId}`).set('Authorization', `Bearer ${customer.token}`).send()
  expect(response.status).toBe(200)
  expect(response.body.id).toBeDefined()
  expect(response.body.price).toBe(await computeOrderPrice(response.body))
  await checkOrderEqualsOrderData(response.body, newOrderData)
})
afterAll(async () => {
  await shutdownApp()
})
})

```

FR8: Editar/eliminar orden.

Si el pedido está en estado 'pendiente', el cliente puede editar o eliminar los productos incluidos o eliminar todo el pedido.

La dirección de entrega también se puede modificar en el estado 'pendiente'.

Si el pedido está en el estado 'enviado' o 'entregado' no se permite edición ni eliminación

// 1. Si el precio es superior a 10€ los gastos de envío tienen que ser 0.

// 2. Si el precio es menor o igual a 10€, los gastos de envío deben ser los gastos de envío predeterminados del restaurante y deben sumarse al precio total del pedido.

// 3. Para guardar el pedido y los productos relacionados, inicie una transacción, almacene el pedido, almacene cada línea de producto y confirme la transacción.

// 4. Si se genera una excepción, capturarla y revertir la transacciónA tener en cuenta:

// 1. Si el precio es superior a 10€ los gastos de envío tienen que ser 0.

// 2. Si el precio es menor o igual a 10€, los gastos de envío deben ser los gastos de envío predeterminados del restaurante y deben sumarse al precio total del pedido.

// 3. Para guardar el pedido y los productos relacionados, inicie una transacción, almacene el pedido, almacene cada línea de producto y confirme la transacción.

// 4. Si se genera una excepción, capturarla y revertir la transacción

EDITAR

REQ

```
▼ body: {address: 'Calle falsa 123', products: Array(2)}
  address: 'Calle falsa 123'
  ▼ products: (2) [{}]
    ▶ 0: {productId: 6963, quantity: 1}
    ▶ 1: {productId: 6964, quantity: 13}

url: '/orders/714'
▼ user: User {dataValues: {}, _previousDataValues: {}, uniqno: 1, _changed: Set(0), _options: {}, ...}
  ▶ _changed: Set(0) {size: 0}
  ▶ _options: {isNewRecord: false, _schema: null, _schemaDelimiter: '', raw: true, attributes: Array(14)}
  ▼ _previousDataValues: {id: 1, firstName: 'Customer 1', lastName: 'Fake 1', email: 'customer1@customer.
    address: 'Fake street 123'
    avatar: 'public/avatars/maleAvatar.png'
    createdAt: Mon Jun 17 2024 12:10:21 GMT+0200 (hora de verano de Europa central)
    email: 'customer1@customer.com'
    firstName: 'Customer 1'
    id: 1
    lastName: 'Fake 1'
    password: '$2a$05$LD523YvMgleSUD6inLmYY.Q4LWN6iJx.YXC38sM7BlU/HYXG4M/1i'
    phone: '+3466677888'
    postalcode: '41010'
    token: '9a7b32a646ca18a434e38c83f8d406a3ad1e49b6'
    tokenExpiration: Wed Jun 19 2024 19:17:56 GMT+0200 (hora de verano de Europa central)
    updatedAt: Wed Jun 19 2024 18:17:52 GMT+0200 (hora de verano de Europa central)
    userType: 'customer'
```

2. CONTROLADOR

```
const update = async function (req, res) {
  // Obtenemos la orden que queremos editar
  const original = await Order.findByPk(req.params.orderId)
  const transaction = await sequelizeSession.transaction()
  try {
    let price = 0
    // Calculamos el precio del pedido
    for (const product of req.body.products) {
      const productDB = await Product.findByPk(product.productId)
      price += product.quantity * productDB.price
    }
    // Actualizamos body de la solicitud para luego usar ya el precio para actualizar la orden
    req.body.price = price

    const restaurant = await Restaurant.findByPk(original.restaurantId)
    // Calculamos gastos de envío
    req.body.shippingCosts = price > 10 ? 0 : restaurant.shippingCosts + price
    // Actualizamos en BD
    await Order.update(req.body, { where: { id: req.params.orderId } }, transaction)

    const order = await Order.findByPk(req.params.orderId)
    // Eliminamos todas las asociaciones de productos existentes, pasandole un array vacío nos aseguramos que borra todas las asociaciones
    // Sería el método "contrario" al addProducts(), pues este crea asociaciones
    await order.setProducts([], transaction)

    // Guardamos las asociaciones de los productos
    for (const product of req.body.products) {
      const productDB = await Product.findByPk(product.productId)
      await order.addProduct(productDB, {
        through: {
          quantity: product.quantity,
          unityPrice: productDB.price
        },
        transaction
      })
    }

    await transaction.commit()

    // Añadimos la asociación de los productos a la orden
    const finalOrder = await Order.findByPk(req.params.orderId, {
      include: 'products'
    })
    res.json(finalOrder.dataValues)
  } catch (err) {
    await transaction.rollback()
    res.status(500).send(err)
  }
}
```

3. MIDDLEWARE o VALIDATION

```
const checkUpdatedOrder = async (value, { req }) => {
  try {
    if (req.body.products.length < 1) {
      return Promise.reject(new Error('The array of products is empty'))
    }
    for (const p of value) {
      if (p.productId < 1) {
        return Promise.reject(new Error('The product has not valid Id'))
      }
      const product = await Product.findByPk(p.productId)
      if (!product.availability) {
        return Promise.reject(new Error('The product is not available'))
      }
      const original = await Order.findByPk(req.params.orderId)
      if (product.restaurantId !== original.restaurantId) {
        return Promise.reject(new Error('The product does not belong to the original restaurant'))
      }
    }
    return Promise.resolve()
  } catch (err) {
    return Promise.reject(new Error(err))
  }
}
```

```

// TODO: Include validation rules for update that should:
// 1. Check that restaurantId is NOT present in the body.
// 2. Check that products is a non-empty array composed of objects with productId and quantity greater than 0
// 3. Check that products are available
// 4. Check that all the products belong to the same restaurant of the originally saved order that is being edited.
// 5. Check that the order is in the 'pending' state.

const update = [
  check('userId').not().exists(),
  check('restaurantId').not().exists(),
  check('address').exists().isString().isLength({ min: 1, max: 255 }).trim(),
  check('products').custom(checkUpdatedOrder),
  check('products.*.quantity').isInt({ min: 1 }).toInt()
]

```

4.RUTA

```

app.route('/orders/:orderId')
  .put(
    isLoggedIn,
    hasRole('customer'),
    checkEntityExists(Order, 'orderId'),
    OrderMiddleware.checkOrderCustomer,
    OrderMiddleware.checkOrderIsPending,
    OrderValidation.update,
    handleValidation,
    OrderController.update
  )

```

5.TEST

- Loggeado
- No owner
- Customer que hizo el pedido
- Order que tiene que existir
- Pedido confirmado
- Pedido enviado
- Pedido delivered
- El restaurante no se puede modificar
- Tiene que tener los valores correctos
- No se puede modificar un order sin producctos
- Tiene que tener productos, no 0, no cantidad negativa...
- Tiene que pedir productos disponibles
- Solo pedidos de un restaurantes

```

describe('Edit order', () => {
  let customer, owner, restaurant, anotherRestaurant, orderData, order, editedOrderData, app
  beforeAll(async () => {
    app = await getApp()
    customer = await getLoggedInCustomer()
    owner = await getLoggedInOwner()
    restaurant = await getFirstRestaurantOfOwner(owner)
    anotherRestaurant = await createRestaurant(owner)
    order = await createOrder(customer, restaurant, await getNewOrderData(restaurant))
    orderData = await getNewOrderData(restaurant)
    delete orderData.restaurantId
  })
  it('Should return 401 if not logged in', async () => {
    const response = await request(app).put('/orders/${order.id}`).send(orderData)
    expect(response.status).toBe(401)
  })
  it('Should return 403 when logged in as an owner', async () => {
    const response = await request(app).put('/orders/${order.id}').set('Authorization', `Bearer ${owner.token}`).send(orderData)
    expect(response.status).toBe(403)
  })
  it('Should return 403 when logged in as another customer', async () => {
    const response = await request(app).put('/orders/${order.id}').set('Authorization', `Bearer ${await getNewLoggedInCustomer().token}`).send(orderData)
    expect(response.status).toBe(403)
  })
  it('Should return 404 when trying to modify a nonexistent order', async () => {
    const response = await request(app).put('/orders/invalidOrderId').set('Authorization', `Bearer ${customer.token}`).send(orderData)
    expect(response.status).toBe(404)
  })
  it('Should return 409 when order is confirmed', async () => {
    let confirmedOrder = await getNewOrderData(restaurant)
    confirmedOrder.startedAt = new Date()
    confirmedOrder = await createOrder(customer, restaurant, confirmedOrder)
    const response = await request(app).put('/orders/${confirmedOrder.id}').set('Authorization', `Bearer ${customer.token}`).send(orderData)
    expect(response.status).toBe(409)
  })
  it('Should return 409 when order is sent', async () => {
    let sentOrder = await getNewOrderData(restaurant)
    sentOrder.startedAt = new Date()
    sentOrder.sentAt = new Date()
    sentOrder = await createOrder(customer, restaurant, sentOrder)
    const response = await request(app).put('/orders/${sentOrder.id}').set('Authorization', `Bearer ${customer.token}`).send(orderData)
    expect(response.status).toBe(409)
  })
  it('Should return 409 when order is delivered', async () => {
    let deliveredOrder = await getNewOrderData(restaurant)
    deliveredOrder.startedAt = new Date()
    deliveredOrder.sentAt = new Date()
    deliveredOrder.deliveredAt = moment().add(5, 'minutes').toDate()
    deliveredOrder = await createOrder(customer, restaurant, deliveredOrder)
    const response = await request(app).put('/orders/${deliveredOrder.id}').set('Authorization', `Bearer ${customer.token}`).send(orderData)
    expect(response.status).toBe(409)
  })
  it('Should return 422 when trying to modify the restaurant', async () => {
    const invalidOrder = { ...orderData }
    invalidOrder.restaurantId = (await getRandomRestaurant(restaurant)).id
    const response = await request(app).put('/orders/${order.id}').set('Authorization', `Bearer ${customer.token}`).send(invalidOrder)
    expect(response.status).toBe(422)
  })
  it('Should return 422 when trying to modify an order without products', async () => {
    const invalidOrder = { ...orderData }
    delete invalidOrder.products
    const response = await request(app).put('/orders/${order.id}').set('Authorization', `Bearer ${customer.token}`).send(invalidOrder)
    const errorFields = response.body.errors.map(error => error.param)
    expect(['products'].every(field => errorFields.includes(field))).toBe(true)
    expect(response.status).toBe(422)
  })
  it('Should return 422 when trying to order with a non-array of products', async () => {
    const invalidOrder = { ...orderData }
    invalidOrder.products = 2
    const response = await request(app).put('/orders/${order.id}').set('Authorization', `Bearer ${customer.token}`).send(invalidOrder)
    const errorFields = response.body.errors.map(error => error.param)
    expect(['products'].every(field => errorFields.includes(field))).toBe(true)
    expect(response.status).toBe(422)
  })
  it('Should return 422 when trying to order products with quantity 0', async () => {
    const invalidOrder = { ...orderData }
    invalidOrder.products = invalidOrder.products.map(product => ({ ...product, quantity: 0 }))
    const response = await request(app).put('/orders/${order.id}').set('Authorization', `Bearer ${customer.token}`).send(invalidOrder)
    const errorFields = response.body.errors.map(error => error.param)
    // Comprobar que errorFields contiene una cadena con formato 'products[*].quantity', donde * puede ser cualquier caracter
    expect(errorFields.some(field => field.match(/products\[.\d+\]\.quantity$/))).toBe(true)
    expect(response.status).toBe(422)
  })
  it('Should return 422 when trying to order products with negative quantity', async () => {
    const invalidOrder = { ...orderData }
    invalidOrder.products = invalidOrder.products.map(product => ({ ...product, quantity: -1 }))
    const response = await request(app).put('/orders/${order.id}').set('Authorization', `Bearer ${customer.token}`).send(invalidOrder)
    const errorFields = response.body.errors.map(error => error.param)
    // Comprobar que errorFields contiene una cadena con formato 'products[*].quantity', donde * puede ser cualquier caracter
    expect(errorFields.some(field => field.match(/products\[.\d+\]\.quantity$/))).toBe(true)
    expect(response.status).toBe(422)
  })
})

```

```

it('Should return 422 when trying to order unavailable products', async () => {
  const unavailableOrderData = await getNewOrderDataWithUnavailableProduct(restaurant)
  delete unavailableOrderData.restaurantId
  const invalidOrder = { ...unavailableOrderData }
  const response = await request(app).put('/orders/${order.id}`).set('Authorization', `Bearer ${customer.token}`).send(invalidOrder)
  const errorFields = response.body.errors.map(error => error.param)
  // Comprobar que errorFields contiene una cadena con formato 'products[*].quantity', donde * puede ser cualquier caracter
  expect(['products'].every(field => errorFields.includes(field))).toBe(true)
  expect(response.status).toBe(422)
})
it('Should return 422 when trying to order from different restaurants', async () => {
  const invalidOrder = await getNewOrderData(restaurant)
  delete invalidOrder.restaurantId
  const productFromAnotherRestaurant = await createProduct(owner, anotherRestaurant)
  invalidOrder.products.push({ productId: productFromAnotherRestaurant.id, quantity: 1 })
  const response = await request(app).put('/orders/${order.id}`).set('Authorization', `Bearer ${customer.token}`).send(invalidOrder)
  const errorFields = response.body.errors.map(error => error.param)
  // Comprobar que errorFields contiene una cadena con formato 'products[*].quantity', donde * puede ser cualquier caracter
  expect(['products'].every(field => errorFields.includes(field))).toBe(true)
  expect(response.status).toBe(422)
})
it('Should return 422 when invalid order data', async () => {
  const invalidOrder = await getNewOrderData(restaurant)
  delete invalidOrder.restaurantId
  delete invalidOrder.address
  invalidOrder.products[0].productId = 'invalidId'
  const response = await request(app).put('/orders/${order.id}`).set('Authorization', `Bearer ${customer.token}`).send(invalidOrder)
  expect(response.status).toBe(422)
  const errorFields = response.body.errors.map(error => error.param)
  expect(['products', 'address'].every(field => errorFields.includes(field))).toBe(true)
})
it('Should return 200 and the edited order when valid order', async () => {
  editedOrderData = await getNewOrderData(restaurant, 15)
  const editedOrderDataWithoutRestaurantId = { ...editedOrderData }
  delete editedOrderDataWithoutRestaurantId.restaurantId
  const response = await request(app).put('/orders/${order.id}`).set('Authorization', `Bearer ${customer.token}`).send(editedOrderDataWithoutRestaurantId)
  expect(response.status).toBe(200)
  expect(response.body.id).toBeDefined()
  expect(response.body.price).toBe(await computeOrderPrice(response.body))
  await checkOrderEqualsOrderData(response.body, editedOrderData)
})
it('Should return 200 and the correct created order when requesting order details', async () => {
  const response = await request(app).get('/orders/${order.id}`).set('Authorization', `Bearer ${customer.token}`).send()
  expect(response.status).toBe(200)
  expect(response.body.id).toBeDefined()
  expect(response.body.price).toBe(await computeOrderPrice(response.body))
  await checkOrderEqualsOrderData(response.body, editedOrderData)
})
it('Create order <10€ that should add shipping costs. Should return 200 and the correct edited order adding shipping costs', async () => {
  editedOrderData = await getNewOrderData(restaurant, 3)
  const editedOrderDataWithoutRestaurantId = { ...editedOrderData }
  delete editedOrderDataWithoutRestaurantId.restaurantId
  const response = await request(app).put('/orders/${order.id}`).set('Authorization', `Bearer ${customer.token}`).send(editedOrderDataWithoutRestaurantId)
  expect(response.status).toBe(200)
  expect(response.body.id).toBeDefined()
  expect(response.body.price).toBe(await computeOrderPrice(response.body))
  await checkOrderEqualsOrderData(response.body, editedOrderData)
})
it('Should return 200 and the correct edited order with added shipping costs when requesting order details', async () => {
  const response = await request(app).get('/orders/${order.id}`).set('Authorization', `Bearer ${customer.token}`).send()
  expect(response.status).toBe(200)
  expect(response.body.id).toBeDefined()
  expect(response.body.price).toBe(await computeOrderPrice(response.body))
  await checkOrderEqualsOrderData(response.body, editedOrderData)
})
afterAll(async () => {
  await shutdownApp()
})
}

```

ELIMINAR

2.CONTROLADOR

```
const destroy = async function (req, res) {
  const transaction = await sequelizeSession.transaction()
  try {
    const result = await Order.destroy({ where: { id: req.params.orderId } })
    let message = ''
    if (result === 1) {
      message = 'Successfully deleted order id.' + req.params.orderId
    } else {
      message = 'Could not delete order.'
    }
    res.json(message)
  } catch (err) {
    await transaction.rollback()
    res.status(500).send(err)
  }
}
```

4.RUTA

5.TEST

```
describe('Delete order', () => {
  let customer, restaurant, order, app, owner
  beforeAll(async () => {
    app = await getApp()
    customer = await getLoggedInCustomer()
    owner = await getLoggedInOwner()
    restaurant = await getFirstRestaurantOfOwner(owner)
    order = await createOrder(customer, restaurant, await getNewOrderData(restaurant))
  })
  it('Should return 401 if not logged in', async () => {
    const response = await request(app).delete('/orders/${order.id}`).send()
    expect(response.status).toBe(401)
  })
  it('Should return 403 when logged in as an owner', async () => {
    const response = await request(app).delete('/orders/${order.id}')
      .set('Authorization', `Bearer ${owner.token}`)
      .send()
    expect(response.status).toBe(403)
  })
  it('Should return 403 when logged in as another customer', async () => {
    const response = await request(app).delete('/orders/${order.id}')
      .set('Authorization', `Bearer ${await getNewLoggedInCustomer().token}`)
      .send()
    expect(response.status).toBe(403)
  })
  it('Should return 404 when trying to delete a nonexistent order', async () => {
    const response = await request(app).delete('/orders/invalidOrderId')
      .set('Authorization', `Bearer ${customer.token}`)
      .send()
    expect(response.status).toBe(404)
  })
  it('Should return 409 when order is confirmed', async () => {
    let confirmedOrder = await getNewOrderData(restaurant)
    confirmedOrder.startedAt = new Date()
    confirmedOrder = await createOrder(customer, restaurant, confirmedOrder)
    const response = await request(app).delete('/orders/${confirmedOrder.id}')
      .set('Authorization', `Bearer ${customer.token}`)
      .send()
    expect(response.status).toBe(409)
  })
  it('Should return 409 when order is sent', async () => {
    let sentOrder = await getNewOrderData(restaurant)
    sentOrder.startedAt = new Date()
    sentOrder.sentAt = new Date()
    sentOrder = await createOrder(customer, restaurant, sentOrder)
    const response = await request(app).delete('/orders/${sentOrder.id}')
      .set('Authorization', `Bearer ${customer.token}`)
      .send()
    expect(response.status).toBe(409)
  })
  it('Should return 409 when order is delivered', async () => {
    let deliveredOrder = await getNewOrderData(restaurant)
    deliveredOrder.startedAt = new Date()
    deliveredOrder.sentAt = new Date()
    deliveredOrder.deliveredAt = moment().add(5, 'minutes').toDate()
    deliveredOrder = await createOrder(customer, restaurant, deliveredOrder)
    const response = await request(app).delete('/orders/${deliveredOrder.id}')
      .set('Authorization', `Bearer ${customer.token}`)
      .send()
    expect(response.status).toBe(409)
  })
  it('Should return 200 when removing valid order', async () => {
    const response = await request(app).delete('/orders/${order.id}')
      .set('Authorization', `Bearer ${customer.token}`)
      .send()
    expect(response.status).toBe(200)
  })

  it('Should return 404 when the order has been removed', async () => {
    const response = await request(app).delete('/orders/${order.id}')
      .set('Authorization', `Bearer ${customer.token}`)
      .send()
    expect(response.status).toBe(404)
  })
  afterAll(async () => {
    await shutdownApp()
  })
})
```

- Loggeado
- No owner
- Customer que hizo el pedido
- Order que tiene que existir para poder borrarse
- Pedido confirmado no se puede borrar
- Pedido enviado no se puede borrar
- Pedido delivered no se puede borrar

FRONTEND

INTRODUCCIÓN

React Native es un framework de código abierto desarrollado por Facebook que se utiliza para crear aplicaciones móviles multiplataforma.

React Native es un framework de código abierto desarrollado por Facebook que se utiliza para crear aplicaciones móviles multiplataforma. En el desarrollo del Frontend se tienen 2 herramientas básicas:

1.Expo: Librería de JavaScript que se integra con ReactNative para extender su funcionalidad. Nos ayudara a trabajar con React y a desplegar la aplicación

2.React: Framework de desarrollo de interfaces de usuario

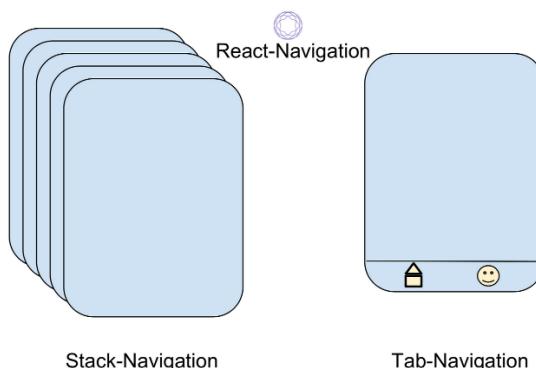
TAB NAVIGATION

La navegación por pestañas es el estilo de navegación más común en las aplicaciones móviles



Lo desarrollaremos para crear 3 vistas principales, 3 “pantallitas”: Restaurante, Perfil de usuario y Controlador. A tener en cuenta que los componentes serian como las “funciones” de los elementos del programa. Un conjunto de etiquetas <>, <>, <>, <>, <>, se pueden agrupar en una etiqueta <> que es un componente. Por lo tanto, en el contexto de React etiqueta, componente y función estamos hablando de lo mismo

STACK NAVIGATION NESTED IN THE TAB NAVIGATOR



COMPONENTS

En general, los componentes de software son una especie de artefactos que encapsulan un conjunto de funciones relacionadas para que puedan reutilizarse. Los componentes de React son los bloques de construcción reutilizables que podemos definir para crear las interfaces de usuario de nuestras aplicaciones.

Nosotros usaremos las *Function Components*, una forma de definir componentes en React y React Native utilizando funciones en lugar de clases. El componente de función definido toma como entrada algunos parámetros que se denominan **props** y devuelve un elemento React.

Un ejemplo de componente seria:

```
export default function RestaurantsScreen({ navigation }) {
  return (
    <View style={styles.container}>
      <TextRegular style={{ fontSize: 16, alignSelf: 'center', margin: 20 }}>Random Restaurant</TextRegular>
      <Pressable
        onPress={() => {
          navigation.navigate('RestaurantDetailScreen', { id: Math.floor(Math.random() * 100) })
        }}
        style={({ pressed }) => [
          {
            backgroundColor: pressed
              ? GlobalStyles.brandBlueTap
              : GlobalStyles.brandBlue
          },
          styles.actionButton
        ]}
      >
        <TextRegular textStyle={styles.text}>
          Go to Random Restaurant Details
        </TextRegular>
      </Pressable>
    </View>
  )
}
```

Exportamos la función `RestaurantsScreen` que recibe como parámetro `navigation` para manejar la navegación entre pantallas.

Se devuelve una tupla con el componente `View` como contenedor principal y se le aplica un estilo de `styles.container`. Dentro de la vista, hay un componente `TextRegular` que muestra el texto "Random Restaurant".

Se utiliza el componente `Pressable` para crear un área presionable en la pantalla. Cuando se presiona, se navegará a la pantalla 'RestaurantDetailScreen' pasando un objeto con una propiedad `id` que es un número aleatorio entre 0 y 99.

STATES

Los componentes a menudo necesitan mantener cierta información en la memoria para recordar cosas o manejar cambios dinámicos en la interfaz de usuario. Esto se logra utilizando el concepto de *estado*. El estado en React se refiere a cualquier dato que el componente necesita mantener y que puede cambiar a lo largo del tiempo.

Nosotros usaremos el hook `useState`, con el cual estamos creando una variable que almacenará el estado de nuestro componente y una función que nos permitirá actualizar ese estado

```
const [state, setState] = useState(initialState)
```

Cuando realizamos una solicitud al backend para recuperar la lista de restaurantes, los datos devueltos deben mantenerse en el estado del RestaurantsScreen componente.

Entonces, en este componente, necesitamos definir un state que contendrá la matriz de restaurantes (inicialmente será una matriz vacía []) como:

```
const [restaurants, setRestaurants] = useState([])
```

PROPS

Los componentes de React utilizan *props* para comunicarse entre sí. Nos permiten pensar en los componentes padre e hijo de forma independiente. Por ejemplo en este contexto Profile es el componente principal y Avatar un componente secundario:

```
export default function Profile() {
  return (
    <Avatar
      person={{ name: 'Lin Lanying', imageUrl: '1bX5QH6' }}
      size={100}
    />
  );
}
```

Le hemos pasado 2 *props* a Avatar `person` que es un objeto y `size` que es un entero. Podemos leer estos props enumerando sus nombres `person`, `size` separados por comas dentro {} y directamente después function Avatar. Esto nos permite usarlos dentro del Avatar:

```
function Avatar({ person, size }) {
```

Otro ejemplo seria:

```
export default function RestaurantDetailScreen ({ route }) {
  const { id } = route.params
  return (
    <View style={styles.container}>
      <TextRegular style={{ fontSize: 16, alignSelf: 'center',
margin: 20 }}>Restaurant details. Id: {id}</TextRegular>
    </View>
  )
}
```

En este caso el componente RestaurantDetailScreen utiliza *props* para recibir datos de navegación (en este caso, el id del restaurante)

HOOKS

Son funciones especialmente implementadas que nos permiten agregar funcionalidad a los componentes de React más allá de simplemente crear y devolver elementos de React.

- **useState**: permite añadir estado o datos a componentes funcionales. Con este hook, puedes declarar variables de estado en tus componentes y también obtener métodos para actualizar ese estado.
- **useEffect**: permite ejecutar efectos secundarios en componentes funcionales. Podemos usarlo para realizar tareas como hacer llamadas a APIs, suscribirnos a eventos ...
- **useContext**: Compartir datos entre componentes definiendo un contexto y recuperar ese contexto

useEffect

Los efectos permiten que un componente se conecte y sincronice con sistemas externos. Tiene 2 argumentos: la función que se ejecutará cuando se active el hook y una matriz opcional que contiene los valores de dependencia que activarán el hook cuando sus valores hayan cambiado.

```
useEffect(() => {
  //code to be executed
}, [object1, object2, ...])
```

Por ejemplo:

```
function ChatRoom({ roomId }) {
  useEffect(() => {
    const connection = createConnection(roomId)
    connection.connect()
    return () => connection.disconnect()
  }, [roomId])
```

En este componente ChatRoom se garantiza que cada vez que el id de la sala cambie, se cree una nueva conexión utilizando ese id y se establezca la conexión. Además, cuando el componente se desmonte o cuando el id de la sala cambie, se desconectará la conexión para evitar fugas de memoria o problemas de rendimiento.

RESTAURANTsSCREEN

*R*estaurantsScreen es una pantalla que debería mostrar una lista de restaurantes que pertenecen al propietario. Cada elemento debe representar al menos el nombre del restaurante y, si se hace clic o se toca un elemento, se debe navegar a la pantalla de detalles del restaurante de ese restaurante.

*R*estaurantDetailScreen es una pantalla que debería mostrar los detalles del restaurante seleccionado previamente en RestaurantsScreen, incluyendo el menú de dicho restaurante.

RESTAURANTsSCREEN

1. Definimos el objeto estado del componente, que será un restaurants array de objetos donde almacenaremos la lista de restaurantes. En la primera renderización , el objeto restaurants tendrá como estado una matriz vacía [].

```
const [restaurants, setRestaurants] = useState([])
```

2. Cargamos los restaurantes en el estado usando el gancho useEffect que tomará por defecto 2 argumentos: la función que se activará y una matriz opcional de objetos que activa la función cuando se cambian sus valores. Establecemos un tiempo de espera de dos segundos para que los restaurantes carguen después de estos dos segundos de retraso.

Recordamos que el hook *useEffect* define un efecto secundario tras la primera renderización del componente, que *setTimeout* establece un temporizador de 2 segundos y tras este, se llama a la función *getAll*, que obtiene los datos de los restaurantes. Por último con *setRestaurants*, establecemos el estado de restaurants.

```
useEffect(() => {
  console.log('Loading restaurants, please wait 2 seconds')
  setTimeout(() => {
    setRestaurants(getAll())
    console.log('Restaurants loaded')
  }, 2000)
}, [])
```

RENDERIZAR

Es el proceso de construir y actualizar la interfaz de usuario (UI) basada en el estado actual del componente y sus props. En el contexto de renderizar, **UI** (Interfaz de Usuario, por sus siglas en inglés "User Interface") se refiere a todo aquello con lo que el usuario interactúa directamente en una aplicación. Esto incluye elementos visuales como botones, textos, imágenes, formularios, menús, y cualquier otro componente visible o interactivo en la pantalla.

- **Renderización Inicial:** Se realiza cuando el componente se monta por primera vez.
- Renderización Posterior:** Se realiza cada vez que el estado o las props cambian

Por lo tanto, cuando `restaurants` cambia, el componente se vuelve a renderizar, mostrando la lista actualizada de restaurantes

FlatList

Es una interfaz eficaz para representar listas planas y básicas. Para utilizar este componente necesitamos:

`data`: especifica la matriz de elementos que se van a representar en la lista

`renderItem`: función que se llama para renderizar cada elemento en la lista. Recibe un objeto que contiene información sobre el elemento a renderizar y devuelve un componente que representa ese elemento.

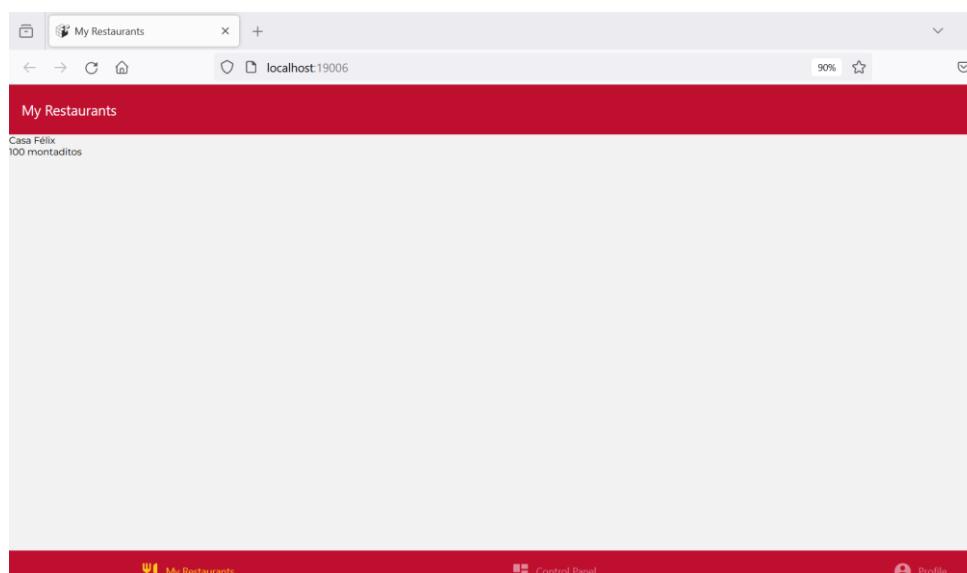
`keyExtractor`: función que extrae una clave única para cada elemento en la lista. Para cada elemento en el array `data` toma el elemento y devuelve una cadena que sirve como clave única.

3. Definimos el renderItem. Recibe un item que contiene la información sobre el restaurante a renderizar y devuelve un componente que representa un Restaurante. Se devuelve un componente `Pressable` (componente de React Native que se puede presionar) el cual tiene un estilo definido en `style`. El prop `onPress` está configurado para navegar en la pantalla `RestaurantDetailScreen` cuando se presiona el restaurante de id del item dado (`id: item.id`). `TextRegular` muestra el nombre del restaurante

```
const renderRestaurant = ({ item }) => {
  return (
    <Pressable
      style={styles.row}
      onPress={() => {
        navigation.navigate('RestaurantDetailScreen', { id: item.id })
      }}
      <TextRegular>
        {item.name}
      </TextRegular>
    </Pressable>)
}
```

4. Representamos el componente FlatList Devuelve un componente que renderiza la lista de restaurantes. Se indica el estilo, el array de datos a renderizar, la función a renderizar cada uno de los elementos de data y la función para obtener la clave única.

```
return (
  <FlatList
    style={styles.container}
    data={restaurants}
    renderItem={renderRestaurant}
    keyExtractor={item => item.id.toString()}/>
```



RESTAURANTDETAILSCREEN

1.Definimos el objeto estado del componente , que será un restaurants array de objetos donde almacenaremos la lista de restaurantes. En la primera renderización , el objeto `restaurants` tendrá como estado una matriz vacía [].

```
const [restaurant, setRestaurant] = useState([])
```

2.Cargamos los detalles de los restaurantes en el estado usando el gancho

`useEffect` .Establecemos 1 segundo de tiempo de espera para que los detalles de los restaurantes se carguen. Cuando se cumple el tiempo establecido esta línea de código llama a la función `getDetail()` con el id del restaurante obtenido de `route.params.id`. Esta función es responsable de obtener los detalles del restaurante según su id. Luego, los detalles del restaurante se establecen en el estado `restaurant` utilizando `setRestaurant`

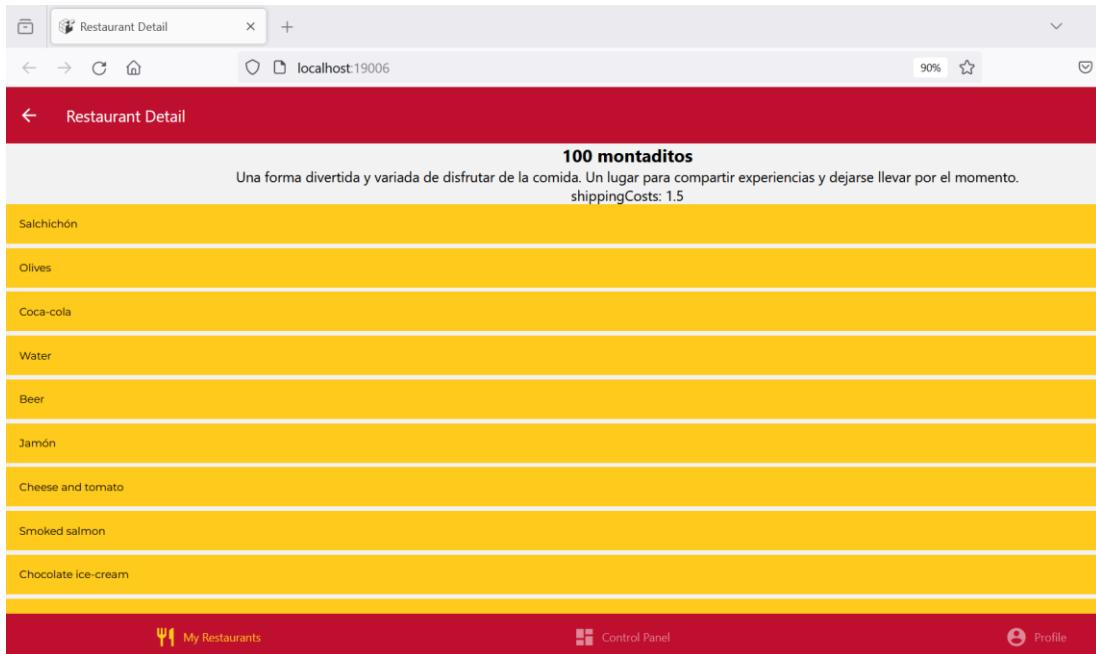
```
useEffect(() => {
  console.log('Loading restaurant details, please wait 1 second')
  setTimeout(() => {
    setRestaurant(getDetail(route.params.id))
    console.log('Restaurant details loaded')
  }, 1000)
}, [])
```

3.Definimos el renderItem. Recibe un item que contiene la info sobre el producto que se está renderizando en data. Se devuelve un componente `Pressable` (componente de React Native que se puede presionar) el cual tiene un estilo definido en `style`. Con `TextRegular` mostramos el nombre del producto

```
const renderItem = ({ item }) => {
  return (
    <Pressable
      style={styles.row}
      onPress={() => { }}
      >
      <TextRegular>
        {item.name}
      </TextRegular>
    </Pressable>
  )
}
```

4. Representamos el componente FlatList

```
return (
  <View style={styles.container}>
    <TextRegular
      style={styles.textTitle}>{restaurant.name}</TextRegular>
    <TextRegular
      style={styles.text}>{restaurant.description}</TextRegular>
    <TextRegular style={styles.text}>shippingCosts:
      {restaurant.shippingCosts}</TextRegular>
    <FlatList
      style={styles.container}
      data={restaurant.products}
      renderItem={renderProduct}
      keyExtractor={item => item.id.toString()}
    />
  </View>)
```



RESTAURANTENDPOINTS IMPLEMENTATION

En este archivo están las definiciones de las rutas URL y los controladores asociados para diversas funciones relacionadas con la gestión de restaurantes.

```
function getAll () {
  return get('/users/myrestaurants')
}

function getDetail (id) {
  return get(`restaurants/${id}`)
}
```

RESTAURANTsSCREEN IMPLEMENTATION

Para obtener los restaurantes del propietario es necesario iniciar sesión luego antes de ejecutar la solicitud, preguntamos si hay un usuario que haya iniciado sesión.

Utilizamos el hook `useContext` de React para acceder al contexto llamado `AuthorizationContext` y extraer la variable `loggedInUser`:

```
const { loggedInUser } = useContext(AuthorizationContext)
```

Si `loggedInUser` es null entonces ningún usuario ha iniciado sesión. Por el contrario, si un usuario ha iniciado sesión, en `loggedInUser` se almacenará el objeto del usuario, incluidas sus propiedades.

```
const [restaurants, setRestaurants] = useState([])
```

Recordamos que `useState` tiene 2 parámetros: una variable que almacenará el estado de nuestro componente `Restaurant` (`restaurants`) y una función que nos permitirá actualizar ese estado (`setRestaurants`). También definimos un state que contendrá la matriz de restaurantes (`[]`).

Recordamos que `useEffect` tiene 2 argumentos: la función que se ejecutará cuando se active el hook:

```
async () => {
  const fetchedRestaurants = await getAll()
  setRestaurants(fetchedRestaurants)},
```

y una matriz opcional que contiene los valores de dependencia que activarán el hook cuando sus valores hayan cambiado: `[]`

En este caso, la función almacena en la constante `fetchedRestaurants` el resultado de ejecutar `getAll()` y luego actualiza el estado del componente `Restaurant` con la lista de restaurantes obtenida en `setRestaurants`.

```
useEffect(async () => {
  const fetchedRestaurants = await getAll()
  setRestaurants(fetchedRestaurants)}, [])
```

Sin embargo, no estamos teniendo en cuenta 3 problemas:

1.Uso de async/await dentro de useEffect: useEffect no permite que la función pasada como argumento sea async. Esto es porque useEffect espera que la función devuelva nada, no un Promise.

2.Dependencia de loggedInUser no incluida: loggedInUser debería ser incluido en el array de dependencias de useEffect

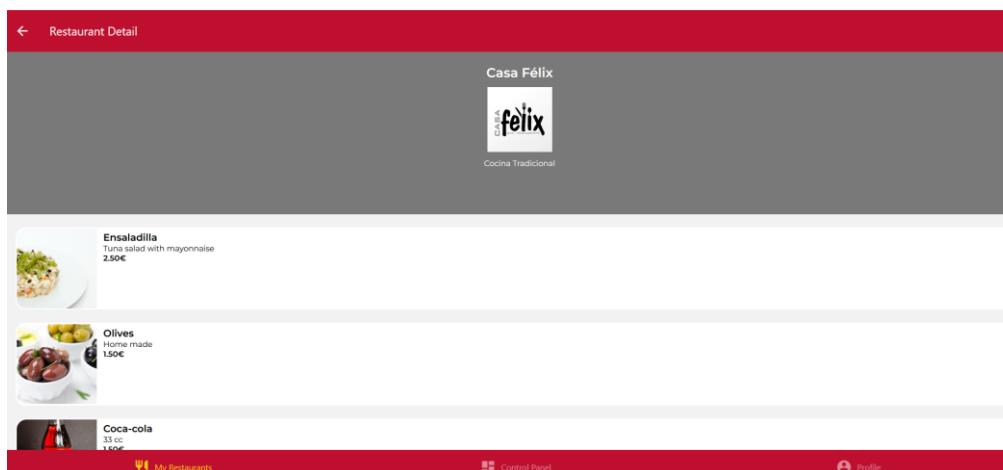
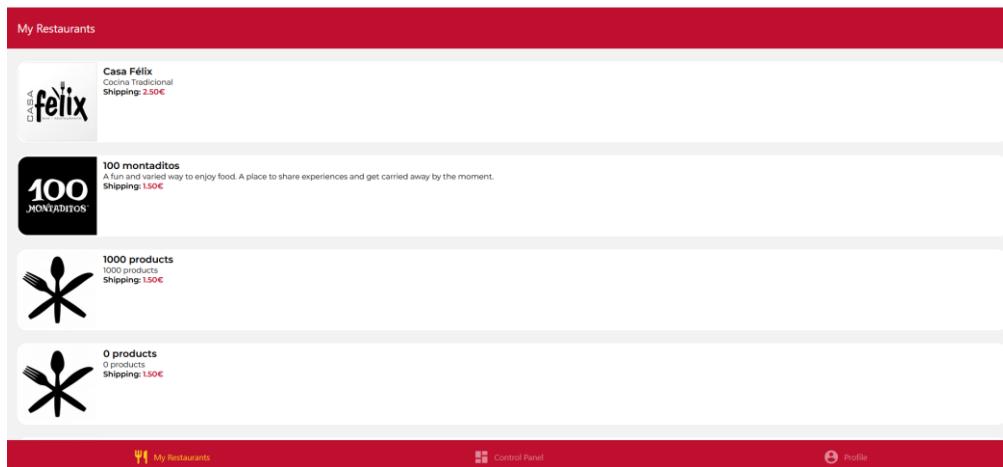
3. No hay manejo de errores

```
useEffect(() => {
    async function fetchRestaurants () { // SOLUCION PROBLEMA 1
        try {
            const fetchedRestaurants = await getAll()
            setRestaurants(fetchedRestaurants)
        } catch (error) { // SOLUCION PROBLEMA 3
            showMessage({
                message: `There was an error while retrieving restaurants.
${error}`,
                type: 'error',
                style: GlobalStyles.flashStyle,
                titleStyle: GlobalStyles.flashTextStyle
            })
        }
    }
    if (loggedInUser) { //SOLUCION PROBLEMA 2
        fetchRestaurants()
    } else {
        setRestaurants(null)
    }
}, [loggedInUser])
```

En primer lugar definimos fetchRestaurants como una función asíncrona : `async function fetchRestaurants ()` En esta función, intentamos obtener la lista de restaurantes de la API con la función `getAll()` `const fetchedRestaurants = await getAll()` y actualizar el estado del componente Restaurant con la lista obtenida: `setRestaurants(fetchedRestaurants)` Manejamos los posibles errores con el bloque try, catch. Si hay algún usuario que ha iniciado sesión (`if (loggedInUser)`), entonces ejecutamos esta función creada, si no, establecemos el estado del componente Restaurante a null. Cada vez que loggedInUser cambie de valor, es decir cada vez que un usuario diferente inicie sesión, se activará y/o ejecutará el useEffect

RESTAURANTDETAILSCREEN IMPLEMENTATION

```
useEffect(() => {
  //Funcion que intentará obtener los detalles de un restaurante
  //concreto
  async function fetchRestaurantDetail () {
    try {
      const fetchedRestaurant = await getDetail(route.params.id)
      setRestaurant(fetchedRestaurant)
    } catch (error) {
      showMessage({
        message: `There was an error while retrieving restaurant
details (id ${route.params.id}). ${error}`,
        type: 'error',
        style: GlobalStyles.flashStyle,
        titleStyle: GlobalStyles.flashTextStyle
      })
    }
  }
  fetchRestaurantDetail()
}, [])
```



FLEX BOX

Los componentes nativos de React utilizan el algoritmo Flexbox para definir el diseño de sus hijos. Las propiedades más comunes son:

- **flexDirection** que puede tomar dos valores: `column` (predeterminado) si queremos que sus hijos se representen verticalmente o `row` si queremos que se representen horizontalmente.
- **justifyContent** que puede tomar los siguientes valores:
 - **flex-start** (por defecto). El contenido se distribuye al inicio del eje primario (la dirección de flexión determina el eje primario y secundario)
 - **center**. Los contenidos se distribuyen en el centro.
 - **flex-end**. Los contenidos se distribuyen al final.
 - **space-around** y **space-between** así los contenidos se distribuyen uniformemente.
- **alignItems** definir cómo se alinearán los hijos a lo largo del eje secundario (dependiendo del flexDirection)
 - **flex-start**,
 - **center**,
 - **flex-end**,
 - **stretch**(predeterminado) el contenido se ampliará para llenar el espacio disponible

FLEX DIRECTION - COLUMN	
FLEX DIRECTION - ROW	
JUSTIFY CONTENT- FLEX START	

JUSTIFY CONTENT- FLEX END



JUSTIFY CONTENT- SPACE AROUND



JUSTIFY CONTENT- SPACE BETWEEN

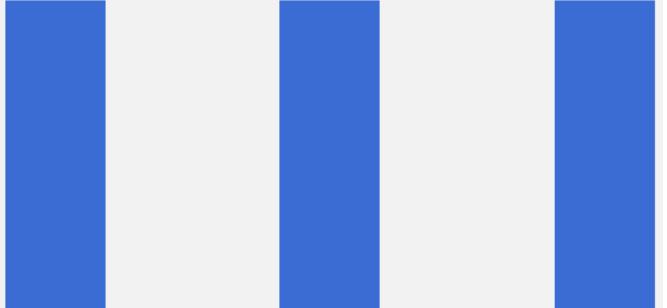


ALIGNITEMS- FLEX START



ALIGNITEMS- CENTER



ALIGNITEMS-FLEX END	
ALIGNITEMS-STRETCH	

VIEW AS CONTAINER

```
export default function CreateRestaurantScreen () {
  return (
    // CONTENIDO DEL COMPONENTE
    <View style={{ alignItems: 'center' }}> //centramos elementos hijos horizontalmente
      <View style={{ width: '60%' }}> //otro View dentro del primero con un estilo que le da
        un ancho del 60% del contenedor padre.
    // CAMPOS DE ENTRADA
    <InputItem
      name='sampleInput'
      label='Sample input'
    />
    <InputItem
      name='sampleInput'
      label='Sample input'
    />
  )
}
```

```
        />
      <InputItem
        name='sampleInput'
        label='Sample input'
      />
      <InputItem
        name='sampleInput'
        label='Sample input'
      />
      <InputItem
        name='sampleInput'
        label='Sample input'
      />
    // BOTON
    <Pressable
      onPress={() => console.log('Button pressed')}
      style={({ pressed }) => [ //define el estilo del boton cambiando si esta presionado
        {
          backgroundColor: pressed
          ? GlobalStyles.brandPrimaryTap
          : GlobalStyles.brandPrimary
        },
        styles.button
      ]}
      <TextRegular textStyle={styles.text}> //define el texto del boton
        Create restaurant
      </TextRegular>
    </Pressable>
  </View>
</View>
)
}
```

FORMULARIOS

Los formularios son la forma de permitir a los usuarios enviar datos desde la GUI del frontend al backend. Esto es necesario para crear nuevos elementos de nuestras entidades. Para crear y mantener el estado del formulario, utilizaremos un componente de terceros: <Formik>.

Formik gestiona el estado de las entradas dentro del formulario y puede aplicarles reglas de validación. El componente Formik debe inicializarse con los nombres y valores iniciales de las entradas del formulario.

Los formularios presentan al usuario varios campos de entrada. Los más populares son:

- **Text inputs:** donde el usuario introduce algún tipo de texto. Suele ser el input más general, podemos utilizarlo para que los usuarios puedan incluir información como: nombres, apellidos, correos electrónicos, descripciones, urls, direcciones, precios, códigos postales o teléfonos.
- **Image/File pickers:** donde el usuario puede seleccionar una imagen/archivo de su galería o sistema de archivos para cargarlos.
- **Select/Dropdown:** donde los usuarios pueden seleccionar un valor para un campo de un conjunto determinado de opciones. Los casos de uso típicos incluyen: seleccionar alguna categoría de las que existen, seleccionar algún valor de estado de un conjunto determinado de valores posibles.
- **Switches:** donde se le pregunta al usuario entre dos opciones que normalmente se envían como booleanas.

FORMULARIO DE CREATeRESTAURANT

InputItem

- **name:** el nombre del campo. **Tiene que coincidir con el nombre del campo esperado en el backend.**
- **label:** el texto presentado al usuario para que se represente entre la entrada de texto.

name, description, address, postalCode,
url, shippingCosts, email, phone

← Create Restaurant

Name

Description

Address

PostalCode

Url

Email

Phone

Logo:

```
<InputItem  
  name='name'  
  label='Name'  
/>  
<InputItem  
  name='description'  
  label='Description'  
/>  
<InputItem  
  name='address'  
  label='Address'  
/>  
<InputItem  
  name='postalCode'  
  label='PostalCode'  
/>  
<InputItem  
  name='url'  
  label='Url'  
/>  
<InputItem  
  name='email'  
  label='Email'  
/>  
<InputItem  
  name='phone'  
  label='Phone'  
/>
```

Image pickers

El código en conjunto permite al usuario seleccionar una imagen de la biblioteca del dispositivo, actualizar el campo 'logo' del formulario con la imagen seleccionada y mostrarla en la interfaz de usuario.

1.Uso del hook useEffect para obtener permisos del dispositivo para acceder a la galería multimedia

```
useEffect(() => {
  (async () => {
    if (Platform.OS !== 'web') {
      const { status } = await
ExpoImagePicker.requestMediaLibraryPermissionsAsync()
      if (status !== 'granted') {
        alert('Sorry, we need camera roll permissions to make this
work!')
      }
    }
  })()
}, [])
```

2.Incluimos un botón que incluya un texto para la etiqueta y una imagen para visualizar la imagen seleccionada. Una vez que presionamos y seleccionamos una imagen, almacenaremos su contenido en la variable de estado usando la setLogo

```
  <Pressable onPress={() =>
    pickImage(
      async result => {
        await setFieldValue('logo', result)
      }
    )
    style={styles.imagePicker}
  >
    <TextRegular>Logo: </TextRegular>
    <Image style={styles.image} source={values.logo ? { uri: values.logo.assets[0].uri } : restaurantLogo} />
  </Pressable>
</View>
</View>
</ScrollView>
)}
```

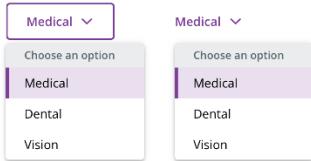
3.Implementamos la función pickImage

```
const pickImage = async (onSuccess) => {

  const result = await ExpoImagePicker.launchImageLibraryAsync({
    mediaTypes: ExpoImagePicker.MediaTypeOptions.Images,
    allowsEditing: true,
    aspect: [1, 1],
    quality: 1
  })
  if (!result.canceled) {
    if (onSuccess) {
      onSuccess(result)
    }
  }
}
```

Select/Drop

Los restaurantes y productos pueden pertenecer a algunas categorías. Incluya una entrada de selección para permitir al usuario seleccionar entre los valores disponibles de RestaurantCategories y entre estados válidos al crear un nuevo restaurante.



1. Estado de las categorías

```
const [restaurantCategories, setRestaurantCategories] = useState([])
```

2. Función asíncrona para obtener las categorías del backend + estado

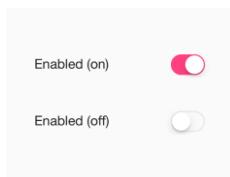
```
useEffect(() => {
  //Función sincrona para obtener las categorías del backend
  async function fetchRestaurantCategories () {
    try {
      const fetchedRestaurantCategories = await getRestaurantCategories() // petición get para obtener las categorías
      const fetchedRestaurantCategoriesReshaped = fetchedRestaurantCategories.map((e) => {
        return {
          label: e.createdAt, // label: lo que sale en la web
          value: e.id // lo que identifica
        }
      })
      setRestaurantCategories(fetchedRestaurantCategoriesReshaped)
    } catch (error) {
      showMessage({
        message: `There was an error while retrieving restaurant categories. ${error} `,
        type: 'error',
        style: GlobalStyles.flashStyle,
        titleStyle: GlobalStyles.flashTextStyle
      })
    }
  }

  fetchRestaurantCategories()
}, [])
```

3. Componente DropDownPicker

```
<DropDownPicker
  open={open}
  value={values.restaurantCategoryId}
  items={restaurantCategories}
  setOpen={setOpen}
  onSelectItem={ item => {
    setFieldValue('restaurantCategoryId', item.value)
  }}
  setItems={setRestaurantCategories}
  placeholder="Select the restaurant category"
  containerStyle={{ height: 40, marginTop: 20 }}
  style={{ backgroundColor: GlobalStyles.brandBackground }}
  dropDownStyle={{ backgroundColor: '#fafafa' }}
/>
```

Switch



1.Componente

<Switch

```
    trackColor={{ false: GlobalStyles.brandSecondary, true:  
GlobalStyles.brandPrimary }}  
    thumbColor={values.availability ?  
GlobalStyles.brandSecondary : '#f4f3f4'}  
    value={values.availability}  
    style={styles.switch}  
    onValueChange={value =>  
      setFieldValue('availability', value)  
    }  
  />
```



FORMIK

Los formularios del Fronted deben validarse antes de enviarlos al Frontend. Por ejemplo: un input para correo electrónico debe contener un correo electrónico válido, o la contraseña debe tener un tamaño mínimo.

Podemos escribir estas validaciones o usando Yup para crear un **schema validation** (required, email, strings, numbers, dates or default values):

STRING (yup.string())	METODOS DE VALIDACIÓN
<code>required()</code>	Requiere que el campo no esté vacío.
<code>max(length)</code>	Requiere que el campo tenga una longitud máxima especificada.
<code>min(length)</code>	Requiere que el campo tenga al menos una longitud mínima especificada.
<code>email()</code>	Requiere que el campo tenga un formato de correo electrónico válido.
<code>url()</code>	Requiere que el campo tenga un formato de URL válido.
<code>matches(regex, message)</code>	Requiere que el campo coincida con una expresión regular.

NUMBER (yup.number())	METODOS DE VALIDACIÓN
<code>required()</code>	Requiere que el campo no esté vacío.
<code>max(number)</code>	Requiere que el campo tenga una longitud máxima especificada.
<code>min(number)</code>	Requiere que el campo tenga al menos una longitud mínima especificada.
<code>positive()</code>	Requiere que el campo tenga un formato de correo electrónico válido.
<code>negative()</code>	Requiere que el valor del campo sea un número negativo.
<code>integer()</code>	Requiere que el valor del campo sea un número entero.

DATE (yup.date())	METODOS DE VALIDACIÓN
<code>required()</code>	Requiere que el campo no esté vacío.
<code>max(date)</code>	Requiere que la fecha sea posterior o igual a una fecha mínima.
<code>min(date)</code>	Requiere que la fecha sea anterior o igual a una fecha máxima.

BOOLEAN (yup.boolean())	METODOS DE VALIDACIÓN
<code>required()</code>	Requiere que el campo no esté vacío.

OBJECT (<code>yup.object()</code>)	METODOS DE VALIDACIÓN
<code>required()</code>	Requiere que el campo no esté vacío.
<code>shape(fields)</code>	Define el esquema de validación para las propiedades del objeto.

ARRAY (<code>yup.array()</code>)	METODOS DE VALIDACIÓN
<code>required()</code>	Requiere que el campo no esté vacío.
<code>max(length)</code>	Requiere que el array tenga una longitud máxima especificada.
<code>min(length)</code>	Requiere que el array tenga al menos una longitud mínima.
<code>of(schema)</code>	Define el esquema de validación para los elementos del array.

Además, Formik tiene las siguientes **propiedades**:

- **validationSchema**: las reglas de validación,
- **initialValues** : los valores iniciales dados a cada una de las entradas del formulario.
- **onSubmit** :
- la función que se llamará cuando los valores del formulario insertados pasen la validación
- **handleSubmit** : es la función que desencadena la validación. Debe llamarse cuando el usuario presiona el botón de envío.
- **values** q es el conjunto de elementos que representa el estado del formulario.
- **SetFieldValue**: A veces tendremos que manejar manualmente el almacenamiento de los valores de los campos. Esta es una función que recibe como primer parámetro el nombre del campo y como segundo parámetro el valor de ese campo. Será necesario para controles de entrada no estándar, InputItems como por ejemplo Imagepickers, Dropdown/select

1. Valores iniciales de los inputs

Tiene que coincidir con los del Backend

```
const initialRestaurantValues = { name: null, description: null, address: null, postalCode: null, url: null, shippingCosts: null, email: null, phone: null, restaurantCategoryId: null }
```

2.Creamos un objeto validationSchema:

```
const validationSchema = yup.object().shape({  
  name: yup  
    .string()  
    .max(255, 'Name too long')  
    .required('Name is required'),  
  address: yup  
    .string()  
    .max(255, 'Address too long')  
    .required('Address is required'),  
  postalCode: yup  
    .string()  
    .max(255, 'Postal code too long')  
    .required('Postal code is required'),  
  url: yup  
    .string()  
    .nullable()  
    .url('Please enter a valid url'),  
  shippingCosts: yup  
    .number()  
    .positive('Please provide a valid shipping cost value')  
    .required('Shipping costs value is required'),  
  email: yup  
    .string()  
    .nullable()  
    .email('Please enter a valid email'),  
  phone: yup  
    .string()  
    .nullable()  
    .max(255, 'Phone too long'),  
  restaurantCategoryId: yup  
    .number()  
    .positive()  
    .integer()  
    .required('Restaurant category is required')  
})
```

3.Añadimos la validación al Formik

```
return (  
  <Formik  
    validationSchema={validationSchema}  
    initialValues={initialRestaurantValues}  
  ...
```

4.Por ejemplo, los DropDownPicker:

```
<DropDownPicker
    open={open}
    value={values.restaurantCategoryId}
    items={restaurantCategories}
    setOpen={setOpen}
    onSelectItem={ item => {
        setFieldValue('restaurantCategoryId', item.value)
    }}
    setItems={setRestaurantCategories}
    placeholder="Select the restaurant category"
    containerStyle={{ height: 40, marginTop: 20 }}
    style={{ backgroundColor: GlobalStyles.brandBackground }}
    dropDownStyle={{ backgroundColor: '#fafafa' }}
/>
<ErrorMessage name={'restaurantCategoryId'} render={msg =>
<TextError>{msg}</TextError> }/>
```

4. Por ejemplo las imagePickers:

```
<Pressable onPress={() =>
    pickImage(
        async result => {
            await setFieldValue('logo', result)
        }
    )
}
style={styles.imagePicker}
>
<TextRegular>Logo: </TextRegular>
<Image style={styles.image} source={values.logo ? { uri:
values.logo.assets[0].uri } : restaurantLogo} />
</Pressable>
```

Hay que añadir al componente Pressable final:

```
<Pressable
    onPress={handleSubmit}
    style={({ pressed }) => [
        {
            backgroundColor: pressed
            ? GlobalStyles.brandSuccessTap
            : GlobalStyles.brandSuccess
        },
        styles.button]}>
```

Name:
 Name is required

Description:

Address:
 address must be a `string` type, but the final value was: `null`. If "null" is intended as an empty value be sure to mark the schema as `.nullable()`

Postal code:
 postalCode must be a `string` type, but the final value was: `null`. If "null" is intended as an empty value be sure to mark the schema as `.nullable()`

Url:
 Please enter a valid url

Shipping costs:
 shippingCosts must be a `number` type, but the final value was: `NaN`.

Email:
 Please enter a valid email

Phone:

Select the restaurant category
 restaurantCategoryId must be a `number` type, but the final value was: `NaN`.

POST de RESTAURANT desde FRONTEND

Si queremos crear un Restaurante desde el Frontend debemos hacer una petición POST al backend

- END POINT

```
function createRestaurant (datosRestaurante) {
  return post('restaurants', datosRestaurante)
}
```

-FUNCION AUX en SCREEN

Aquí pueden pasar varias cosas:

- ❖ Errores en el Backend
- ❖ Operaciones asíncronas

```

const createRestaurant = async (values) => {
  setBackendErrors([])
  try {
    const createdRestaurant = await createRestaurantEnd(values)
    showMessage({
      message: `Restaurant ${createdRestaurant.name} successfully
created`,
      type: 'success',
      style: GlobalStyles.flashStyle,
      titleStyle: GlobalStyles.flashTextStyle
    })
    navigation.navigate('RestaurantsScreen', { dirty: true })
  } catch (error) {
    console.log(error)
    setBackendErrors(error.errors)
  }
}

return (
  <Formik
    validationSchema={validationSchema}
    initialValues={initialRestaurantValues}
    onSubmit={createRestaurant} >

```

POST de PRODUCT desde FRONTEND

1.

```

const initialProductValues = { name: null, description: null, price:
null, order: null, productCategoryId: null, availability: true,
restaurantId: route.params.id

}

```

2.

```

const validationSchema = yup.object().shape({
  name: yup
    .string()
    .max(255, 'Name too long')
    .required('Name is required'),
  description: yup
    .string()
    .max(255, 'Description too long')

```

```

    .required('Description is required'),
  price: yup
    .number()
    .max(255, 'Price code too long')
    .required('Price code is required'),
  order: yup
    .number()
    .nullable()
    .positive('Please provide a positive order value')
    .integer('Please provide an integer order value'),
  availability: yup
    .boolean(),
  productCategoryId: yup
    .number()
    .positive()
    .integer()
    .required('productCategory is required')
)

```

3.

```

const createProduct = async (values) => {
  setBackendErrors([])
  try {
    const createdProduct = await createProductEnd(values)
    showMessage({
      message: `Product ${createdProduct.name} successfully created`,
      type: 'success',
      style: GlobalStyles.flashStyle,
      titleStyle: GlobalStyles.flashTextStyle
    })
    //Pantalla destino:RestaurantDetailScreen
    // id del Restaurante concreto que queremos crear el producto
    navigation.navigate('RestaurantDetailScreen', { id:
      route.params.id, dirty: true })
  } catch (error) {
    console.log(error)
    setBackendErrors(error.errors)
  }
}

```

4.

```

function createProductEnd (datosProducto) {
  return post('products', datosProducto)
}

```

5.

```
return (
  <Formik
    validationSchema={validationSchema}
    initialValues={initialProductValues}
    onSubmit={createProduct}
  ...
)
```

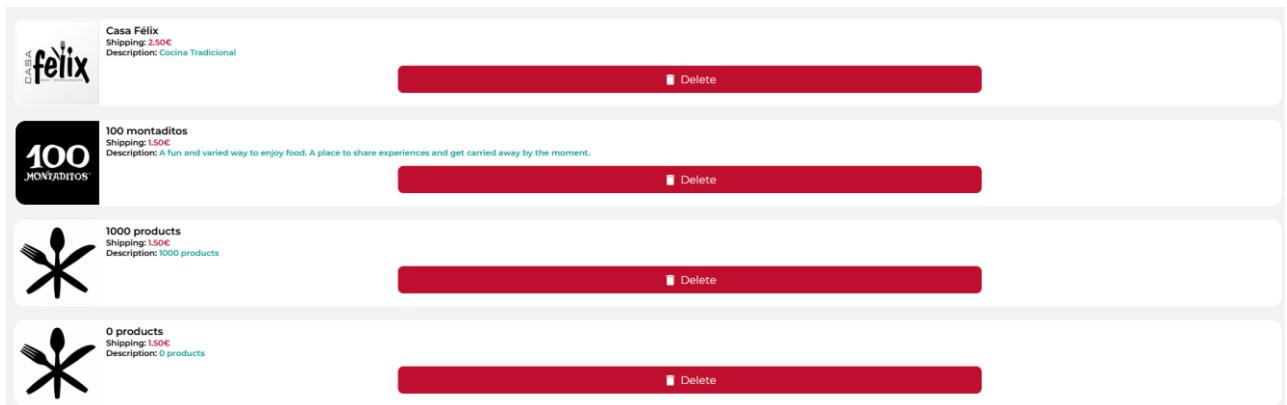
DELETE de RESTAURANT desde FRONTEND

- END POINT

```
function deleteRestaurantEnd (id) {
  return destroy(`restaurants/${id}`)
}
```

1.Boton para eliminar

```
<Pressable
  onPress={() => console.log(`Delete pressed for restaurantId =
${item.id}`)}
  style={({ pressed }) => [
    {
      backgroundColor: pressed
        ? GlobalStyles.brandPrimaryTap
        : GlobalStyles.brandPrimary
    },
    styles.actionButton
  ]}>
  <View style={[{ flex: 1, flexDirection: 'row', justifyContent: 'center' }]}>
    <MaterialCommunityIcons name='delete' color={'white'} size={20}/>
    <TextRegular textStyle={styles.text}>
      Delete
    </TextRegular>
  </View>
</Pressable>
```

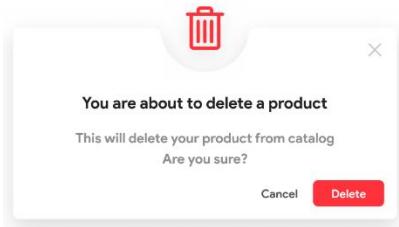


2.Fucion asincrona

```
const [restaurantToBeDeleted, setRestaurantToBeDeleted] = useState(null)

const removeRestaurant = async (restaurant) => {
    try {
        await deleteRestaurantEnd(restaurant.id)
        await fetchRestaurants()
        setRestaurantToBeDeleted(null)
        showMessage({
            message: `Restaurant ${restaurant.name} successfully removed`,
            type: 'success',
            style: GlobalStyles.flashStyle,
            titleStyle: GlobalStyles.flashTextStyle})
    } catch (error) {
        console.log(error)
        setRestaurantToBeDeleted(null)
        showMessage({
            message: `Restaurant ${restaurant.name} could not be removed.`,
            type: 'error',
            style: GlobalStyles.flashStyle,
            titleStyle: GlobalStyles.flashTextStyle
        })
    }
}
```

Delete Modal



-**isVisible**: una expresión booleana que se evalúa para mostrar u ocultar la ventana modal.

-**onCancel**: la función que se ejecutará cuando el usuario presione el botón cancelar.

-**onConfirm**: la función que se ejecutará cuando el usuario presione el botón de confirmación.

3. Añadimos el componente DeleteModal

```
<DeleteModal
  isVisible={restaurantToDelete !== null}
  onCancel={() => setRestaurantToDelete(null)}
  onConfirm={() => removeRestaurant(restaurantToDelete)}>
  <TextRegular>The products of this restaurant will be deleted as
  well</TextRegular>
  <TextRegular>If the restaurant has orders, it cannot be
  deleted.</TextRegular>
</DeleteModal>
```