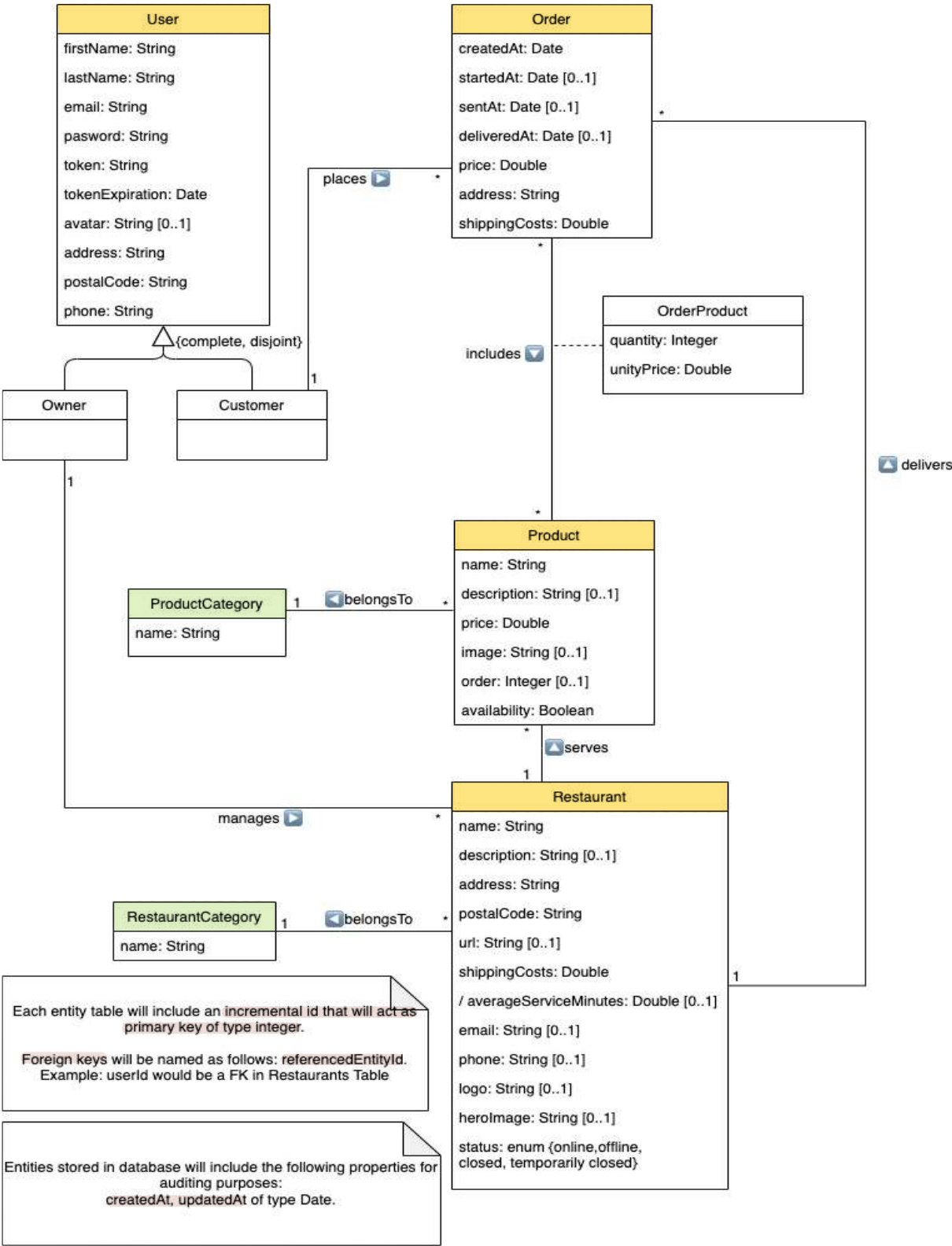


Backend

1. Diagrama de clase
2. Models
3. Migrations
4. Seeders
5. Controllers
6. Validation y middleware
7. Routes

Diagrama de clases

El modelo de restaurante sigue el siguiente diagrama de clases:



En total tendremos 7 tablas:

OrderProducts

Orders ← 'products' ~ de OrderProducts
 'restaurant' ~ con RestaurantId

	#	Nombre	Tipo de datos	Longitud/Con...	Sin signo	Permitir NULL	Rellen...	Predeterminado
🔑	1	id	INT	11	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	AUTO_INCREME...
	2	startedAt	DATETIME		<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	NULL
	3	sentAt	DATETIME		<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	NULL
	4	deliveredAt	DATETIME		<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	NULL
	5	price	DOUBLE		<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	NULL
	6	address	VARCHAR	255	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Sin valor predeter...
	7	shippingCosts	DOUBLE		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Sin valor predeter...
🔑	8	restaurantId	INT	11	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Sin valor predeter...
🔑	9	userId	INT	11	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Sin valor predeter...
	10	createdAt	DATETIME		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	'2024-06-10 15:43...
	11	updatedAt	DATETIME		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	'2024-06-10 15:43...


Products



	#	Nombre	Tipo de datos	Longitud/Con...	Sin signo	Permitir NULL	Rellen...	Predeterminado
🔑	1	id	INT	11	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	AUTO_INCREME...
	2	name	VARCHAR	255	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Sin valor predeter...
	3	description	TEXT		<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	NULL
	4	price	DOUBLE		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Sin valor predeter...
	5	image	VARCHAR	255	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	NULL
	6	order	INT	11	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	NULL
	7	availability	TINYINT	1	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	NULL
🔑	8	restaurantId	INT	11	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Sin valor predeter...
🔑	9	productCateg...	INT	11	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Sin valor predeter...
	10	createdAt	DATETIME		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	'2024-06-10 15:43...
	11	updatedAt	DATETIME		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	'2024-06-10 15:43...




ProductCategories

	#	Nombre	Tipo de datos	Longitud/Con...	Sin signo	Permitir NULL	Rellen...	Predeterminado
🔑	1	id	INT	11	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	AUTO_INCREME...
	2	name	VARCHAR	255	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Sin valor predeter...
	3	createdAt	DATETIME		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	'2024-06-10 15:43...
	4	updatedAt	DATETIME		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	'2024-06-10 15:43...




RestaurantCategories

#	Nombre	Tipo de datos	Longitud/Con...	Sin signo	Permitir NULL	Rellen...	Predeterminado
 1	id	INT	11	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	AUTO_INCREME...
2	name	VARCHAR	255	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Sin valor predeter...
3	createdAt	DATETIME		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	'2024-06-10 15:43...
4	updatedAt	DATETIME		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	'2024-06-10 15:43...

Restaurant  'Products' ~ con restaurantId de product
 'Orders' ~ con restaurantId de Order

#	Nombre	Tipo de datos	Longitud/Con...	Sin signo	Permitir NULL	Rellen...	Predeterminado
 1	id	INT	11	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	AUTO_INCREME...
2	name	VARCHAR	255	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Sin valor predeter...
3	description	TEXT		<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	NULL
4	address	VARCHAR	255	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Sin valor predeter...
5	postalCode	VARCHAR	255	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Sin valor predeter...
6	url	VARCHAR	255	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	NULL
7	shippingCosts	DOUBLE		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	'0'
8	averageServic...	DOUBLE		<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	NULL
9	email	VARCHAR	255	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	NULL
10	phone	VARCHAR	255	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	NULL
11	logo	VARCHAR	255	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	NULL
12	herolImage	VARCHAR	255	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	NULL
13	status	ENUM	'online','offli...	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	'offline'
14	createdAt	DATETIME		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	'2024-06-10 15:43...
15	updatedAt	DATETIME		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	'2024-06-10 15:43...
 16	userId	INT	11	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Sin valor predeter...
 17	restaurantCate...	INT	11	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Sin valor predeter...

Users

#	Nombre	Tipo de datos	Longitud/Con...	Sin signo	Permitir NULL	Rellen...	Predeterminado
 1	id	INT	11	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	AUTO_INCREME...
2	firstName	VARCHAR	255	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Sin valor predeter...
3	lastName	VARCHAR	255	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Sin valor predeter...
 4	email	VARCHAR	255	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Sin valor predeter...
5	password	VARCHAR	255	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Sin valor predeter...
 6	token	VARCHAR	255	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	NULL
7	tokenExpiration	DATETIME		<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	NULL
8	phone	VARCHAR	255	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Sin valor predeter...
9	avatar	VARCHAR	255	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	NULL
10	address	VARCHAR	255	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Sin valor predeter...
11	postalCode	VARCHAR	255	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Sin valor predeter...
12	userType	ENUM	'customer','o...	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	'customer'
13	createdAt	DATETIME		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	'2024-06-10 15:43...
14	updatedAt	DATETIME		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	'2024-06-10 15:43...

1. Migrations

Las migraciones crean el esquema de la base de datos, creando una migración por cada entidad.

Tienen **dos métodos principales**:

- **up**: se utiliza para aplicar cambios, como crear una nueva tabla o añadir un campo a una tabla existente.
- **down**: se utiliza para revertir los cambios aplicados por el método up, permitiendo un regreso seguro al estado anterior de la base de datos.

En el caso de Restaurantes, se crearán migraciones para Usuario, Restaurante, Pedido, Producto (además de Categoría de Restaurante y Categoría de producto).

Para hacer las migraciones, tenemos que fijarnos en la entidad del diagrama de clases, por ejemplo, la migración de restaurant será:

```
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,
        defaultValue: 0.0,
        type: Sequelize.DOUBLE
      },
      averageServiceMinutes: {
        allowNull: true,
        type: Sequelize.DOUBLE
      },
      email: {
        type: Sequelize.STRING
      },
      phone: {
        type: Sequelize.STRING
      },
      logo: {
        type: Sequelize.STRING
      },
    })
  },
```

```

    logo: {
      type: Sequelize.STRING
    },
    heroImage: {
      type: Sequelize.STRING
    },
    status: {
      type: Sequelize.ENUM,
      values: [
        'online',
        'offline',
        'closed',
        'temporarily closed'
      ],
      defaultValue: 'offline'
    },
    createdAt: {
      allowNull: false,
      type: Sequelize.DATE,
      defaultValue: new Date()
    },
    updatedAt: {
      allowNull: false,
      type: Sequelize.DATE,
      defaultValue: new Date()
    },
    userId: {
      allowNull: false,
      type: Sequelize.INTEGER,
      onDelete: 'CASCADE',
      references: {
        model: {
          tableName: 'Users'
        },
        key: 'id'
      }
    },
    restaurantCategoryId: {
      allowNull: false,
      type: Sequelize.INTEGER,
      references: {
        model: {
          tableName: 'RestaurantCategories'
        },
        key: 'id'
      }
    }
  })
},
down: async (queryInterface, Sequelize) => {
  await queryInterface.dropTable('Restaurants')
}
}

```

2. Seeders

Los seeders (inyecciones) nos permiten popular la base de datos (inyectan los datos):

Para popular la base de datos: `npm run sequelize-cli db:seed:all`, y actualizar: `npm run migrate:backend`

```
module.exports = {
  up: async (queryInterface, Sequelize) => {
    module.exports.copyFiles()
    await queryInterface.bulkInsert('Restaurants',
      [
        { name: 'Casa Félix', description: 'Cocina Tradicional', address: 'Av. Reina Mercedes 51, Sevilla', postalCode: '41012', url: 'https://goo.gl/maps/GZEfzge4zXz6ySLR8', restaurantCategoryId: 1, shippingCosts: 2.5, email: 'casafelix@restaurant.com', logo: process.env.RESTAURANTS_FOLDER + '/casaFelixLogo.jpeg', phone: 954123123, createdAt: new Date(), updatedAt: new Date(), userId: 2, status: 'closed' },
        { name: '100 montaditos', description: 'A fun and varied way to enjoy food. A place to share experiences and get carried away by the moment.', address: 'JÄ³zefa 34, KrakÄ³w, Poland', postalCode: '123-4567', logo: 'public/restaurants/100MontaditosLogo.jpeg', heroImage: 'public/restaurants/100MontaditosHero.jpeg', url: 'http://spain.100montaditos.com/', restaurantCategoryId: 2, shippingCosts: 1.5, email: 'info@restaurant.com', phone: '+48123456789', createdAt: new Date(), updatedAt: new Date(), userId: 2, status: 'online' },
        { name: '1000 products', description: '1000 products', address: 'JÄ³zefa 34, KrakÄ³w, Poland', postalCode: '123-4567', url: 'http://1000products.com/', restaurantCategoryId: 2, shippingCosts: 1.5, email: 'info@restaurant.com', phone: '+48123456789', createdAt: new Date(), updatedAt: new Date(), userId: 2, status: 'online' } ], {}))
  },

  down: async (queryInterface, Sequelize) => {
    const { sequelize } = queryInterface
    try {
      await sequelize.transaction(async (transaction) => {
        const options = { transaction }
        await sequelize.query('SET FOREIGN_KEY_CHECKS = 0', options)
        await sequelize.query('TRUNCATE TABLE Restaurants', options)
        await sequelize.query('SET FOREIGN_KEY_CHECKS = 1', options)
      })
    } catch (error) {
      console.error(error)
    }
  },

  copyFiles: () => {
    const originDir = 'public/example_assets/'
    const destinationDir = process.env.RESTAURANTS_FOLDER + '/'
    console.error(destinationDir)
    if (!fs.existsSync(destinationDir)) {
      fs.mkdirSync(destinationDir, { recursive: true })
    }
    const restaurantsFileNames = ['casaFelixLogo.jpeg', 'casaFelixHero.jpeg', '100MontaditosHero.jpeg', '100MontaditosLogo.jpeg']
    restaurantsFileNames.forEach(restaurantFilename => {
      fs.copyFile(originDir + restaurantFilename, destinationDir + restaurantFilename, (err) => {
        if (err) throw err
      })
    })
  }
}
```


3. Models

Para vincular los objetos de lógica empresarial a los objetos de la base de datos se utilizan los modelos, gracias a a estos podemos hacer operaciones de bases de datos sin problemas. Por lo general, los objetos relacionados con las entidades de la base de datos se denominan Modelos, y los usamos para interactuar con sus entidades de las bases de datos, mediante las operaciones CRUD.

Utilizamos la herramienta sequelize para el mapeo relacional de objetos de Node.js que proporciona todas las herramientas necesarias para establecer conexiones a la base de datos, ejecutar migraciones y seeders, definir modelos y realizar operaciones.

Para el modelo de Restaurant sería:

```
import { Model } from 'sequelize' → clase para crear un modelo
import moment from 'moment'

const loadModel = (sequelize, DataTypes) => {
  class Restaurant extends Model {
    /**
     * Helper method for defining associations.
     * This method is not a part of Sequelize lifecycle.
     * The `models/index` file will call this method automatically.
     */
    static associate (models) { → para definir las asociaciones con otros modelos
      // define association here
      Restaurant.belongsTo(models.RestaurantCategory, { foreignKey:
'restaurantCategoryId', as: 'restaurantCategory' }) → 1:1
      Restaurant.belongsTo(models.User, { foreignKey: 'userId', as: 'user' }) → 1:1
      Restaurant.hasMany(models.Product, { foreignKey: 'restaurantId', as:
'products' }) 1:N
      Restaurant.hasMany(models.Order, { foreignKey: 'restaurantId', as: 'orders' })
    }
    async getAverageServiceTime () {
      try {
        const orders = await this.getOrders()
        const serviceTimes = orders.filter(o => o.deliveredAt).map(o =>
moment(o.deliveredAt).diff(moment(o.createdAt), 'minutes'))
        return serviceTimes.reduce((acc, serviceTime) => acc + serviceTime, 0) /
serviceTimes.length
      } catch (err) {
        return err
      }
    }
  }
}
```



```

Restaurant.init({ → Inicializa el modelo
  name: {
    allowNull: false, → No permite nulos
    type: DataTypes.STRING
  },
  description: DataTypes.TEXT, → Si permite nulos
  address: {
    allowNull: false,
    type: DataTypes.STRING
  },
  postalCode: {
    allowNull: false,
    type: DataTypes.STRING
  },
  url: DataTypes.STRING,
  shippingCosts: {
    allowNull: false,
    type: DataTypes.DOUBLE
  },
  averageServiceMinutes: DataTypes.DOUBLE,
  email: DataTypes.STRING,
  phone: DataTypes.STRING,
  logo: DataTypes.STRING,
  heroImage: DataTypes.STRING,
  status: {
    type: DataTypes.ENUM,
    values: [
      'online',
      'offline',
      'closed',
      'temporarily closed'
    ]
  },
  restaurantCategoryId: {
    allowNull: false,
    type: DataTypes.INTEGER
  },
  userId: {
    allowNull: false,
    type: DataTypes.INTEGER
  },
  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

```

→ nombre del modelo.

4. Controladores

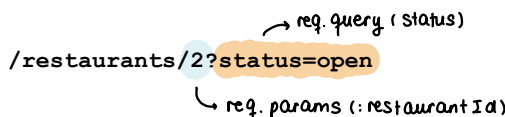
Los controladores son los componentes principales de la capa lógica, es dónde podemos implementar funcionalidades o lógica de negocio por cada entidad.

Cada método del controlador recibe un objeto de solicitud `req` y un objeto de respuesta `res`.

req: El objeto de solicitud representa la solicitud HTTP y tiene propiedades para la cadena de consulta de la solicitud, parámetros, cuerpo, encabezados HTTP...

Los atributos de la solicitud que vamos a ver son:

- **req.body:** representa los datos que provienen del cliente (documento JSON), contiene pares clave-valor de los datos enviados en el cuerpo de la solicitud.
- **req.params:** Esta propiedad es un objeto que contiene propiedades asignadas a la ruta denominados "params" (parámetros), es decir, representa los parámetros de la ruta.
Por ejemplo si tenemos una ruta `/restaurants/:restaurantId` tendremos acceso al parámetro "`restaurantId`" mediante `req.params.restaurantId`,
- **req.query:** representa parámetros de consulta de la ruta, por ejemplo si tenemos una ruta `/restaurants/status=open` tendremos acceso al parámetro "status" mediante `req.query.status`.
- **req.user:** representa el usuario loggeado que hace la petición, por ejemplo: `req.user.id`.


El diagrama muestra la URL `/restaurants/2?status=open`. Una etiqueta `req.query (status)` con una flecha apunta al parámetro de consulta `status=open`. Otra etiqueta `req.params (: restaurantId)` con una flecha apunta al identificador de ruta `2`.

El objeto de respuesta representa la respuesta HTTP que una Express app envía cuando recibe una petición HTTP.

- Los métodos del objeto `res` que necesitamos son:

- **res.json(entityObject):** devuelve el objeto `entityObject` al cliente como un documento JSON con el código 200 de HTTP. Por ejemplo, `res.json(restaurant)` devolverá el objeto `Restaurante` como un documento JSON.
- **res.json(message):** devuelve un mensaje de texto al cliente como un documento JSON con el código de estado HTTP 200.
- **res.status(500).send(err):** devuelve el objeto `err` (que suele incluir un mensaje) y un código de estado HTTP 500.

Los estados del código HTTP utilizados en este proyecto son:

- 200. Solicitudes 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.

¿Qué tipo de `res` tenemos que utilizar en las operaciones CRUD?

GET: operaciones de lectura de datos sin modificarlos.

- **req.params:** Para obtener un recurso específico, por ejemplo: `/restaurants/:restaurantId`
- **req.query:** leer datos con filtro específicos, por ejemplo: `/restaurants/:restaurantId?status=open`
- **req.user:** leer los datos de un usuario en específico.

POST: operaciones de creación completa:

- **req.body:** crea un nuevo recurso, por ejemplo, enviar datos de un nuevo restaurante en `req.body` al hacer `post` a `/restaurants`.

PUT Y PATCH: actualizar un recurso completo (PUT) o parcialmente (PATCH):

- **req.params:** Para actualizar un recurso existente, por ejemplo: `/restaurants/:restaurantId`
- **req.body:** modificar un recurso existente, por ejemplo enviar datos de un nuevo restaurante en `req.body` al hacer `PUT` o `PATCH` a `/restaurants/:restaurantId`
- **req.user:** actualizar los datos de un usuario en específico.

DELETE: para eliminar un recurso específico.

- **req.params:** para borrar un recurso existente, por ejemplo: `/restaurants/:restaurantId`
- **req.user:** eliminar los datos de un usuario en específico.

Recuerda: Cada método del controlador recibe un objeto de solicitud **req** y un objeto de respuesta **res**.

Controlador para leer todos los restaurantes (GET):

- Se encargará de listar todos los restaurantes registrado en la base de datos (para que el cliente pueda consultarlos) utilizando el método `Model.findAll` de Sequelize.

```
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)  
  }  
}
```

Parámetro de solicitud (petición) → **req**

Parámetro de respuesta de la solicitud → **res**

Método de sequelize que recupera todos los registros de la tabla 'Restaurant' de la BD.

la ejecución del código debe esperar hasta que la promesa de `findAll` se resuelva/rechaze

Attributes específica que campos deben ser incluidos o excluidos en los resultados de la consulta.

Excluimos del resultado devuelto el atributo 'userId', todos los demás se incluyen.

Incluye específica la inclusión de relaciones asociadas al modelo principal que se consulta.

Incluye el modelo `RestaurantCategory`

Alias para leer en el resultado

Order toma un Array de arrays:

1er elemento del sub-array: describe el modelo y el alias del atributo por el que se va a ordenar.

Atributo

Orden Ascendente

Si la consulta es exitosa, envía los resultados al cliente en formato JSON.

Si la consulta no es exitosa

Código 500 HTTP (error del servidor)

Controlador para crear un restaurante (POST):

- El documento JSON que contendrá toda la información necesaria para crear un nuevo elemento de la entidad vendrá dado por la petición del cliente, en este caso, para acceder a el tendríamos que usar el atributo: `req.body`
- Utilizaremos el método de sequelize **`Model.build()`** que recibe:
 - Un **objeto JSON** que incluye los campos necesarios para construir un nuevo elemento.
- Y el método **`Model.save()`** para almacenar el nuevo elemento en la tabla de base de datos correspondiente.

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

documento JSON con los datos de creación.

no guarda el objeto, sólo crea la instancia en memoria

Id del usuario actualmente autenticado

intenta guardar el nuevo restaurante en la BD.

Si la operación es exitosa se envía al cliente el objeto Restaurant nuevo en JSON.

Si la op. es exitosa, el método `save()` devuelve el objeto del restaurante guardado

Código 500 HTTP (error del servidor)

Controlador para leer los detalles de un restaurante (GET /{filtroParaUnRestaurante}):

Si queremos obtener los detalles de un restaurante en concreto, el cliente deberá de haber incluido en la petición algún parámetro que filtra ese restaurante, para acceder a este, utilizaremos `req.params.parametroDeFiltro`. Utilizaremos el método de sequelize `Model.findByIdPk` ya que el parámetro de filtro es un id de un restaurante, además nos piden que se deben mostrar los productos de ese Restaurante (otra entidad) y la categoría de ese Restaurante.

```
const show = async function (req, res) {
  // Only returns PUBLIC information of restaurants
  try {
    const restaurant = await Restaurant.findByIdPk( 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)
  }
}
```

Busca en la Base de Datos un registro por su clave primaria, en este caso el parámetro ha sido proporcionado en la ruta.

Parámetros de la ruta

Excluye de la respuesta el id del usuario.

Productos asociados al restaurante

Cada producto incluye su categoría.

incluye información sobre la categoría del restaurante

Orden ascendente

Entidad que tiene el atributo que vamos a utilizar para ordenar.

Atributo de orden

Si la consulta es exitosa, envía el objeto del restaurante como respuesta en formato JSON.

Código 500 HTTP (error del servidor)

relaciones que incluye en el resultado de la consulta

Controlador para actualizar un restaurante (PUT):

Utilizaremos el método `Model.update()` que recibirá el cuerpo de la petición del cliente (en formato JSON) con los datos a actualizar (`req.body`) y el restaurante en concreto a actualizar lo obtendrá de la ruta (`req.params`)

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

Contiene los datos que el cliente envía para actualizar el restaurante.

Método para actualizar

Actualizará los registros de la tabla 'Restaurant' en la que el id coincida con el restaurantId de la ruta.

Busca el restaurante actualizado por su Id.

id de la ruta.

envía el restaurante actualizado.

Controlador para eliminar un restaurante (DELETE):

Utilizaremos el método `Model.destroy()` que recibirá el restaurante en concreto a eliminar en la ruta (`req.params`)

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

Método para eliminar

Condición: se elimina el restaurante cuyo id coincida con el de la ruta.

— Si se ha eliminado debe devolver 1.

— Envía el mensaje

Controlador para obtener un restaurante que pertenecen a un usuario (GET):

Se encargará de listar todos los restaurantes registrados en la base de datos de los que el propietario sea dueño, utilizando el método `Model.findAll` de Sequelize.

```
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)
  }
}
```

usuario loggeado y autenticado

— incluye la categoría

Recuerda: Cada método del controlador recibe un objeto de solicitud `req` y un objeto de respuesta `res`.

Controlador para leer todos los orders como cliente (GET):

- Se encargará de listar todos los orders registrados en la base de datos (para que el cliente pueda consultarlos) utilizando el método `Model.findAll` de Sequelize.

```
/// TODO: Implement the indexCustomer function that queries orders from current
logged-in customer and send them back.
// Orders have to include products that belongs to each order and restaurant details
// sort them by createdAt date, desc.

const indexCustomer = async function (req, res) {
  try {
    const orders = await Order.findAll({

      //from current logged-in customer
      where: {
        userId: req.user.id → id del cliente que hace la petición
      },

      //Orders have to include products that belongs to each order
      include: [ → Include especifica la inclusión de relaciones asociadas al modelo principal que se consulta.
        {
          model: Product, → Incluye el modelo Product.
          as: 'products' → Alias para leer en el resultado
        },

        // and restaurant details
        {
          model: Restaurant,
          as: 'restaurant',
          attributes: → Attributes especifica que campos deben ser incluidos o excluidos en los resultados de la consulta.
            ['name', 'description', 'address', 'postalCode', 'url',
            'shippingCosts', 'averageServiceMinutes', 'email', 'phone', 'logo',
            'heroImage', 'status', 'restaurantCategoryId']
        }
      ],

      //sort them by createdAt date, desc.
      order: [ → Order como un Array de arrays:
        ['createdAt', 'DESC']
      ]
      // Atributo de Order      Orden descendente
    })

    res.json(orders)
  } catch (err) {
    res.status(500).send(err)
  }
}
```

Controlador para crear un order (POST):

- El documento JSON que contendrá toda la información necesaria para crear un nuevo elemento de la entidad vendrá dado por la petición del cliente, en este caso, para acceder a el tendríamos que usar el atributo: `req.body`
- Utilizaremos el método de sequelize **Model.build()** que recibe:
 - Un **objeto JSON** que incluye los campos necesarios para construir un nuevo elemento.
- Y el método **Model.save()** para almacenar el nuevo elemento en la tabla de base de datos correspondiente.

```
// TODO: Implement the create function that receives a new order and stores it in the database.
// Take into account that:
// 1. If price is greater than 10€, shipping costs have to be 0.
// 2. If price is less or equals to 10€, shipping costs have to be restaurant default shipping costs and
have to be added to the order total price
// 3. In order to save the order and related products, start a transaction, store the order, store each
product linea and commit the transaction
// 4. If an exception is raised, catch it and rollback the transaction

const create = async (req, res) => {
  // Use sequelizeSession to start a transaction
  const transaction = await sequelizeSession.transaction()
  const newOrder = Order.build(req.body)

  try {
    let price = 0
    // Compute the shipping costs and the price
    for (const product of req.body.products) {
      const productDB = await Product.findByPk(product.productId) // We look for that product in the DB
      price += product.quantity * productDB.price
    }
    newOrder.price = price // We update the price of the order to the sum of the products price

    // Check the cost limit
    if (newOrder.price < 10) {
      const restaurantOrder = await Restaurant.findByPk(newOrder.restaurantId)
      newOrder.shippingCosts = restaurantOrder.shippingCosts
    } else {
      newOrder.shippingCosts = 0
    }

    newOrder.userId = req.user.id
    newOrder.price += newOrder.shippingCosts
    newOrder.createdAt = new Date()

    // Save all operations performed to the order
    const order = await newOrder.save({ transaction })

    // Save each product in the order
    for (const product of req.body.products) {
      const productDB = await Product.findByPk(product.productId) // Fetch the product in the DB
      await order.addProduct(productDB, {
        through: {
          quantity: product.quantity,
          unityPrice: productDB.price
        },
        transaction
      })
    }

    await transaction.commit() // Commit the transaction since all properties have been fulfilled

    const finalOrder = await Order.findByPk(order.id, {
      include: 'products'
    })

    res.json(finalOrder.dataValues)
  } catch (err) {
    res.status(500).send(err)
    await transaction.rollback()
  }
}
```


Controlador para actualizar un order (PUT):

Utilizaremos el método `Model.update()` que recibirá el cuerpo de la petición del cliente (en formato JSON) con los datos a actualizar (`req.body`) y el order en concreto a actualizar lo obtendrá de la ruta (`req.params`)

```
// Take into account that:
// 1. If price is greater than 10€, shipping costs have to be 0.
// 2. If price is less or equals to 10€, shipping costs have to be restaurant default shipping costs
// and have to be added to the order total price
// 3. In order to save the updated order and updated products, start a transaction, update the
// order, remove the old related OrderProducts and store the new product lines, and commit the
// transaction
// 4. If an exception is raised, catch it and rollback the transaction

const update = async function (req, res) {
  const original = await Order.findByPk(req.params.orderId)
  const transaction = await sequelizeSession.transaction()
  try {
    let price = 0
    // Compute the shipping costs and the price
    for (const product of req.body.products) {
      const productDB = await Product.findByPk(product.productId) // We look for that product in the
      // DB so that we can properly fetch its price
      price += product.quantity * productDB.price
    }
    req.body.price = price

    // Check the cost limit
    if (req.body.price <= 10) {
      const restaurant = await Restaurant.findByPk(original.restaurantId)
      req.body.shippingCosts = restaurant.shippingCosts
    } else {
      req.body.shippingCosts = 0
    }
    req.body.price += req.body.shippingCosts

    await Order.update(req.body, { where: { id: req.params.orderId } }, transaction)

    // Save all operations performed to the order
    const order = await Order.findByPk(req.params.orderId)
    await order.setProducts([], transaction)

    // Save each product in the order
    for (const product of req.body.products) {
      const productDB = await Product.findByPk(product.productId) // Fetch the product in the DB
      await order.addProduct(productDB, {
        through: {
          quantity: product.quantity,
          unityPrice: productDB.price
        },
        transaction
      })
    }

    await transaction.commit() // Commit the transaction since all properties have been fulfilled

    const finalOrder = await Order.findByPk(req.params.orderId, {
      include: 'products'
    })

    res.json(finalOrder.dataValues)
  } catch (err) {
    // Use sequelizeSession to start a transaction
    await transaction.rollback()
    res.status(500).send(err)
  }
}
```

5. Validation y middlewares

Tanto las validation como los middlewares sirven para ver si los datos que llegan en una petición cumplen con los requisitos de nuestra aplicación, para ser más claros, cada uno servirá para:

- Validation: verifica los tipos de datos (a la hora de obtener esos datos o de actualizarlos)
- Middleware: verifica requisitos más concretos (que seas el propietario, que el restaurante tenga pedidos ...)

Empecemos por los validations:

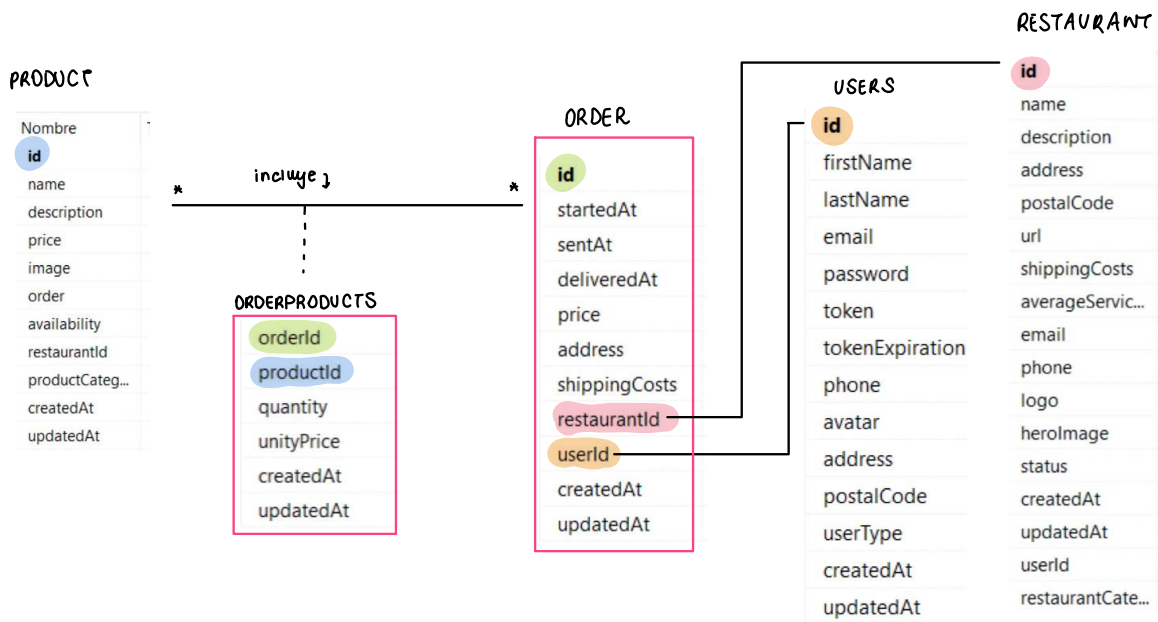
Vemos el modelo de Restaurant en el diagrama de clases, veremos que tipo de datos son los atributos del restaurante, y a la hora de crear uno, tendremos que verificar que son el tipo de dato adecuado.

```
const create = [ Validación para crear un restaurante
  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 = [ Para actualizar
  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')
]
```

Restaurant	
name:	String
description:	String [0..1]
address:	String
postalCode:	String
url:	String [0..1]
shippingCosts:	Double
/ averageServiceMinutes:	Double [0..1]
email:	String [0..1]
phone:	String [0..1]
logo:	String [0..1]
heroImage:	String [0..1]
status:	enum (online, offline, closed, temporarily closed)

Aunque también podemos verificar los datos de entrada mediante funciones más complejas, por ejemplo, las validations de la entidad Order son, además de las normales que hemos visto antes, unas más complejas:



Para crear un pedido, nos piden:

1. Check that restaurantId is present in the body and corresponds to an existing restaurant.
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

Es crear (POST) luego el cliente introduce los datos en el cuerpo de la petición (req), luego para comprobar estos datos accedemos a ellos mediante: `req.body`, que son los que están rodeados en rosa.

```
// req.body = {orderId, startedArt,..., updatedAt, [[orderId, productId0, unityprice0, quantity0], ... [orderId, productIdN, unityPriceN, quantityN]]}

const checkCreationOrder = async (value, { req }) => {
  try {
    // 2. Check that products is a non-empty array
    if (req.body.products.length < 1) {
      return Promise.reject(new Error('The array of products is empty'))
    }
    for (const p of value) {
      // composed of objects with productId
      if (p.productId < 1) {
        return Promise.reject(new Error('The product has not valid Id'))
      }
      // 3. Check that products are available
      const product = await Product.findByPk(p.productId)
      if (!product.availability) {
        return Promise.reject(new Error('The product is not available'))
      }
      // 4. Check that all the products belong to the same restaurant
      if (product.restaurantId !== req.body.restaurantId) {
        return Promise.reject(new Error('The product does not belong to the restaurant'))
      }
    }
    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(checkCreationOrder),
  // 2. and quantity greater than 0
  check('products.*.quantity').isInt({ min: 1 }).toInt()
]
```

Para actualizar un pedido, nos piden:

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.

Es actualizar (PUT) luego el cliente introduce los datos en el cuerpo de la petición (req), luego para comprobar estos datos accedemos a ellos mediante: `req.body` y el order a actualizar estará en la uri, accedemos a el mediante: `req.params.orderId`

```
// req.body = {orderId, startedAt,..., updatedAt, [[orderId, productId0, unityprice0,
quantity0], ... [orderId, productIdN, unityPriceN, quantityN]]}

const checkUpdatedOrder = async (value, { req }) => {
  try {
    // 2. Check that products is a non-empty array
    if (req.body.products.length < 1) {
      return Promise.reject(new Error('The array of products is empty'))
    }
    for (const p of value) {
      // composed of objects with productId
      if (p.productId < 1) {
        return Promise.reject(new Error('The product has not valid Id'))
      }
      // 3. Check that products are available
      const product = await Product.findByPk(p.productId)
      if (!product.availability) {
        return Promise.reject(new Error('The product is not available'))
      }
      // 4. Check that all the products belong to the same restaurant
      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))
  }
}
```

```

const update = [
  // 1. Check that restaurantId is NOT present in the body.
  check('userId').not().exists(),
  check('restaurantId').not().exists(),
  check('address').exists().isString().isLength({ min: 1, max: 255 }).trim(),
  check('products').custom(checkUpdatedOrder),
  // 2. and quantity greater than 0
  check('products.*.quantity').isInt({ min: 1 }).toInt()
]

```

En cuento a los **middlewares**, en Order por ejemplo, nos pedían:

Implement the following function to check if the order belongs to current loggedIn customer (order.userId equals or not to req.user.id)

```

const checkOrderCustomer = async (req, res, next) => {
  try {
    const order = await Order.findByPk(req.params.orderId) → Sacamos el order de la petición.
    if (order.userId === req.user.id) {
      // id del usuario del body === id del usuario que hace la petición.
      return next()
    } else {
      return res.status(403).send('Not enough privileges. This entity does not belong to you')
    }
  } catch (error) {
    return res.status(500).send(error)
  }
}

```

Implement the following function to check if the restaurant of the order exists

```

const checkRestaurantExists = async (req, res, next) => {
  try {
    const restaurant = await Order.findByPk(req.body.restaurantId) → Sacamos el restaurante del order del cliente que hace la petición.
    if (!restaurant) {
      return res.status(409).send('Not found')
    }
    return next()
  } catch (err) {
    return res.status(500).send(err)
  }
}

```

6. Rutas

El backend puede publicar sus funcionalidades a través de servicios RESTFUI, que siguen los patrones arquitectónicos del protocolo HTTP. Por ejemplo, si el sistema proporciona las operaciones CRUD sobre una entidad, debería de hacer un endpoint para cada operación:

- HTTP POST endpoint para CREAR
- HTTP GET endpoint para LEER
- HTTP PUT / PATCH endpoint para ACTUALIZAR
- HTTP DELETE endpoint para ELIMINAR

Las rutas se crean siguiendo buenas prácticas y patrones, por ejemplo:

- HTTP POST `/restaurants` para Crear un restaurante, el método del controlador se llamará `create`.
- HTTP GET `/restaurants` para Leer un restaurante, el método del controlador se llamará `index`.
- HTTP GET `/restaurants/{restaurantId}` para Leer los detalles de un restaurante con `id = restaurantId`. El método del controlador se llamará `show`.
- HTTP PUT `/restaurants/{restaurantId}` para Actualizar los detalles del restaurante con `id = restaurantId`. El método del controlador se llamará `update`.
- HTTP DELETE `/restaurants/{restaurantId}` para Eliminar un restaurante con `id = restaurantId`. El método del controlador se llamará `destroy`.

Por ejemplo, las rutas de restaurante:

```
const loadFileRoutes = function (app) {
  app.route('/restaurants')
    .get(
      RestaurantController.index) —————> LOS Controller ∞ al final
    .post(
      isLoggedIn,
      hasRole('owner'),
      handleFilesUpload(['logo', 'heroImage'], process.env.RESTAURANTS_FOLDER),
      RestaurantValidation.create,
      handleValidation, —————> Despues de las validaciones
      RestaurantController.create)

  app.route('/restaurants/:restaurantId')
    .get(
      checkEntityExists(Restaurant, 'restaurantId'),
      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.restaurantHasNoOrders,
      RestaurantMiddleware.checkRestaurantOwnership,
      RestaurantController.destroy)
```

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

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

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


Frontend

1. Components
2. States
3. Props
4. Hooks
5. Screens
6. Validation y middleware
7. Routes

1. 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.

La opción preferida para crear componentes en React y React-native son los llamados **Function Components**. El componente de función definido toma como entrada algunos parámetros que se denominan **props** y devuelve un elemento React.

- **States:** los componentes normalmente deben mantener datos en memoria para recordar cosas, lo que es llamado state. Para crear y actualizar el estado usamos el hook `useState`:
 - `const [state, useState] = useState(initialValue)`, definimos un array de elementos incluyendo el objeto state y el método para cambiar el estado, además podemos definir el valor inicial de state con `initialValue`.
- Por ejemplo, cuando queramos hacer una petición al backend para recibir una lista de los restaurantes, los datos devueltos deberán quedarse en el state del componente `RestaurantScreen`. En ese componente, necesitaremos definir un state que contenga el array de restaurantes (inicialmente será un array vacío) como:
`const [restaurants, setRestaurants] = useState([])`
- **Props:** podemos usar los props para pasar datos entre componentes (route)

1. Hooks

Los hooks son funciones especialmente implementadas que nos permiten añadir funcionalidad a los componentes a parte de solo crear y devolver elementos React. Los usamos para:

- Mantener el estado de un componente: `useState`
- Actualizar nuestra interfaz cuando los datos son actualizados o recuperan: `useEffect`
- Para comparar datos entre componentes definiendo un contexto y recupere ese contexto usando el `useContext` hook.
- **useState:** devuelve un array que contiene:
 - El objeto state
 - Una función para actualizar el objeto state, esta función admite un nuevo state como parámetro, y provoca la renderización del componente.
- **useEffect:** toma dos argumentos:
 - La función que se ejecuta cuando el hook es activado
 - un array opcional que contiene los valores de dependencia que activarán el hook cuando estos cambien, si el array está vacío, será ejecutado una vez el componente este montado, si no está presente, se ejecutará cuando el componente se monte y cada vez que se actualice.

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

1. Screens

Queremos desarrollar dos pantallas:

- RestaurantsScreen deberá 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.
- RestaurantDetailScreen debe mostrar los detalles del restaurante seleccionado en la pantalla anterior, incluida la descripción y los productos (menú) de ese restaurante.

RestaurantScreen:

PARTE 1: DECLARAR LOS STATES

Restaurants State:

- restaurants: estado que almacenará el objeto de restaurantes obtenidos del backend, inicialmente vacío.
- setRestaurants: para actualizar el estado cuando los datos se obtengan del backend.

PopularProducts state:

- popularProducts: estado que almacenará el objeto de productos obtenidos del backend, inicialmente vacío.
- setRestaurants: para actualizar el estado cuando los datos se obtengan del backend.

```
export default function RestaurantsScreen ({ navigation, route }) {  
  // PROPS: navigation(for navigating between screens) AND route(for accessing  
  route parameters)  
  
  const [restaurants, setRestaurants] = useState([])  
  const [popularProducts, setPopularProducts] = useState([])
```

Una vez definimos los estados, podemos usarlos para la lógica de, renderizado y en los efectos (lógica de obtención de datos)

PARTE 2: DEFINIR LOS MÉTODOS A USAR EN LOS ENDPOINTS.

Antes de hacer los efectos, tendremos que definir en el endpoint de restaurant cual es la acción que queremos hacer:

```
function getAllRestaurant () {  
  return get('restaurants')  
}  
  
function getProductsWithPopularity () {  
  return get(`products/popular`)  
}
```

PARTE 3: DEFINIR LOS EFECTOS (lógica para obtener los datos):

Los estados definidos anteriormente se actualizan dentro de funciones asíncronas que se llaman dentro de los hooks 'useEffect':

```
useEffect(() => { — Dentro vamos a definir una función que se ejecutará cada vez que el componente se monte
                    o cada vez que el prop 'route' cambie.
  async function fetchRestaurants () { — f.asíncrona que obtendrá los datos del backend

    try {
      const fetchedRestaurants = await getAllRestaurant() — llamada a la API, esperamos a
                                                              que la función del endpoint se
                                                              ejecute (GET restaurants)
      // Lista de los restaurantes

      setRestaurants(fetchedRestaurants) — Actualización del estado: si la llamada a la API es
                                          exitosa, utilizamos setRestaurants para actualizar
                                          el estado restaurants con los nuevos datos.

    } catch (error) {
      showMessage({
        message: `There was an error while retrieving restaurants. ${error} `,
        type: 'error',
        style: GlobalStyles.flashStyle,
        titleStyle: GlobalStyles.flashTextStyle
      })
    }
  }
  // 1º parámetro de useEffect()
  fetchRestaurants() — llamamos a la función dentro del useEffect, para que se ejecute cuando
                      el componente se monte o el prop 'route' cambie.
}, [route]) — 2º parámetro de useEffect()
              — Array de dependencias de useEffect, con esto, cada vez que 'route' cambie, el effect se vuelve a ejecutar.
```

```
useEffect(() => {
  async function fetchPopular () {

    try {
      const fetchedPopular = await getProductsWithPopularity() — llamada a la API, esperamos a
                                                                que la función del endpoint se
                                                                ejecute (GET restaurants)
      // Lista de los restaurantes

      setPopularProducts(fetchedPopular)

    } catch (error) {
      showMessage({
        message: `There was an error while retrieving popular products. ${error} `,
        type: 'error',
        style: GlobalStyles.flashStyle,
        titleStyle: GlobalStyles.flashStyle
      })
    }
  }
  fetchPopular()
}, [route])
```

Una vez obtenido los datos, tendremos que mostrarlos por pantallas, para ello, los renderizamos:

Primero hacemos la función que irá adentro del flatList que mostrará los datos:

```
const renderPopular = ({ item }) => {
  return (
    <View style={styles.cardBody}>
      <ImageCard
        imageUri={item.image ? { uri: process.env.API_BASE_URL + '/' + item.image } :
          defaultProductImage}
        onPress={() => {
          navigation.navigate('RestaurantDetailScreen', { id: item.restaurantId })
        }}
      />
      <TextRegular style={styles.cardText}>{item.name}</TextRegular>
      <TextRegular style={{ marginRight: 1 }} numberOfLines={1}>{item.description}</TextRegular>
      <TextSemiBold textStyle={styles.price}>{item.price.toFixed(2)}€</TextSemiBold>
    </View>
  )
}
```

mostrará lo que tengamos en esa clase.

Nombre del restaurante

Descripción

Shipping costs



100 montaditos

A fun and varied way to enjoy food. A place to share experiences and get carried away by the moment.

Shipping: 1.50€

```
const renderPopular = ({ item }) => {
  return (
    <View style={styles.cardBody}>
      <ImageCard
        imageUri={item.image ? { uri: process.env.API_BASE_URL + '/' + item.image } :
          defaultProductImage}
        onPress={() => {
          navigation.navigate('RestaurantDetailScreen', { id: item.restaurantId })
        }}
      />
      <TextRegular style={styles.cardText}>{item.name}</TextRegular>
      <TextRegular style={{ marginRight: 1 }} numberOfLines={1}>{item.description}</TextRegular>
      <TextSemiBold textStyle={styles.price}>{item.price.toFixed(2)}€</TextSemiBold>
    </View>
  )
}
```



Cheap product 1

De marisco

0.00€



Cheap product 2

De marisco

0.00€



Cheap product 3

De marisco

0.00€

Tendremos que tener en cuenta que si hay fallo, mostremos una lista vacía

```
const renderEmptyRestaurantsList = () => {
  return (
    <TextRegular textStyle={styles.emptyList}>
      There is no restaurants yet.
    </TextRegular>
  )
}

const renderEmptyProductsList = () => {
  return (
    <TextRegular textStyle={styles.emptyList}>
      There is no products yet.
    </TextRegular>
  )
}
```

Y por último, la función return que renderizara los datos y los mostrará:

```
return (
  <View style={styles.container}>
    <View style={styles.centeredContent}>
      <TextRegular style={styles.title}>TOP 3 PRODUCTS OF THE WEEK</TextRegular>
      <FlatList
        horizontal={true}
        data={popularProducts}
        renderItem={renderPopular}
        keyExtractor={item => item.id.toString()}
        ListEmptyComponent={renderEmptyProductsList} —————> si no hay datos
      />
    </View>
    <TextRegular style={styles.title}> </TextRegular>
    <TextRegular style={styles.title}>LOOKING FOR SOMETHING TO EAT?</TextRegular>
    <FlatList
      style={styles.restaurantList}
      data={restaurants}
      renderItem={renderRestaurant}
      keyExtractor={item => item.id.toString()}
      ListEmptyComponent={renderEmptyRestaurantsList}
    />
  </View>
)
```