

Explicación simulacro IISSI2

estoy haciendo el backend de una pagina web en javascript con eslint, el backend general ya está creado pero tengo que añadirle una funcionalidad, ahora voy a pasarte las pautas y tu me vas a explicar que habría que hacer en los diferentes puntos del backend para lograr lo que quiero



Claro, estaré encantado de ayudarte con tu proyecto de backend en JavaScript utilizando ESLint. Por favor, pásame las pautas específicas de la funcionalidad que desees añadir, y te explicaré los pasos necesarios para implementarla en tu backend.

solo quiero realizar por ahora la parte del backend. Si tienes alguna pregunta adicional de algo del proyecto realizado con anterioridad (DeliverUS), hazmela y te la responderé pudiendo adjuntar lineas de código o clases enteras del proyecto



Perfecto, estaré listo para ayudarte con la implementación de la nueva funcionalidad en tu backend. Para empezar, ¿podrías proporcionarme las pautas específicas de la funcionalidad que desees añadir? Esto podría incluir:

- 1 **Descripción general de la funcionalidad:** ¿Qué nueva capacidad o característica estás intentando agregar al backend?
- 2 **Endpoints requeridos:** ¿Qué endpoints nuevos necesitas?
- 3 **Modelos de datos:** ¿Qué cambios o adiciones necesitas hacer a tus modelos de datos?
- 4 **Lógica de negocio:** ¿Hay alguna lógica de negocio específica que deba ser implementada?
- 5 **Autenticación y autorización:** ¿Hay requisitos específicos de autenticación o autorización?
- 6 **Validación y manejo de errores:** ¿Qué validaciones y manejo de errores son necesarios?

Con esta información, podré darte una guía detallada de los cambios y adiciones que necesitas hacer en tu backend para lograr la funcionalidad deseada.

Una vez se ha puesto en marcha la primera versión de DeliverUS, los inversores han solicitado la inclusión de una nueva funcionalidad que consiste en ofrecer a los propietarios la posibilidad de promocionar sus restaurantes. Cada propietario sólo podrá promocionar uno de sus restaurantes.

Un propietario podrá promocionar un restaurante de dos maneras distintas:

En el formulario de creación de restaurante. Por defecto, se seleccionará la opción de no promocionado. Si el propietario indica que el nuevo restaurante debe estar promocionado, pero ya existían restaurantes promocionados del mismo propietario, al pulsar el botón Save se mostrará un error y no se creará el restaurante.

En la pantalla de "Mis restaurantes", mediante un botón mostrado junto a cada restaurante, que permitirá mediante su pulsación promocionar el restaurante en cuestión. Si el propietario pulsa el botón para promocionar un nuevo restaurante y ya existían otros restaurantes promocionados del mismo dueño, se procederá a promocionar el restaurante indicado y se marcará como "no promocionado" el restaurante que lo fuese anteriormente. La aplicación debe pedir confirmación al propietario cuando se pulse el botón; utilice para ello el componente suministrado ConfirmationModal, similar al componente DeleteModal utilizado en clase.

Además, los restaurantes promocionados aparecerán siempre al principio de los listados de restaurantes que se le presentan tanto a los propietarios como a los clientes. Además de presentarse al principio, los restaurantes promocionados deben destacarse visualmente, por lo que aparecerá una etiqueta de texto ¡En promoción! con el color principal de la marca.



Para implementar esta nueva funcionalidad en tu backend, sigue estos pasos:

- 1 **Modificación del Modelo de Datos:**
- 2 Añadir un campo `promocionado` en el modelo de `Restaurante`.

3 Actualización del Controlador de Restaurantes:

- 4 Modificar la lógica en la creación y actualización de restaurantes para gestionar la promoción.
- 5 Añadir lógica para promocionar un restaurante desde la pantalla "Mis restaurantes".

6 Endpoints Nuevos o Modificados:

- 7 Modificar el endpoint de creación de restaurante para manejar la promoción.
- 8 Crear o modificar el endpoint para actualizar el estado de promoción de un restaurante.

9 Validación y Manejo de Errores:

- 10 Añadir validaciones para asegurar que solo un restaurante puede estar promocionado por propietario.
- 11 Manejar errores adecuadamente en la creación y promoción de restaurantes.

12 Modificación de Listados:

- 13 Modificar los endpoints que devuelven listados de restaurantes para que los promocionados aparezcan primero.

1. Modificación del Modelo de Datos

Suponiendo que estás usando Mongoose para manejar tus modelos, añade un campo **promocionado** al esquema del restaurante.

javascript

 Copiar código

```
const mongoose = require('mongoose');

const restauranteSchema = new mongoose.Schema({
  nombre: String,
  direccion: String,
  propietario: { type: mongoose.Schema.Types.ObjectId, ref: 'Usuario' },
  promocionado: { type: Boolean, default: false },
  // otros campos...
});

const Restaurante = mongoose.model('Restaurante', restauranteSchema);


module.exports = Restaurante;
```

2. Actualización del Controlador de Restaurantes

Crear Restaurante

Modifica el controlador para manejar la promoción al crear un restaurante.

javascript

 Copiar código

```
const Restaurante = require('../models/Restaurante');

const crearRestaurante = async (req, res) => {
  try {
    const { nombre, direccion, propietario, promocionado } = req.body;

    if (promocionado) {
      const restaurantesPromocionados = await Restaurante.find({ propietario, promocionado });
      if (restaurantesPromocionados.length > 0) {
        return res.status(400).json({ error: 'Ya existe un restaurante promocionado' });
      }
    }
  }
}
```

```

const nuevoRestaurante = new Restaurante({
  nombre,
  direccion,
  propietario,
  promocionado,
});

await nuevoRestaurante.save();
res.status(201).json(nuevoRestaurante);
} catch (error) {
  res.status(500).json({ error: 'Error al crear el restaurante.' });
}
};

module.exports = {
  crearRestaurante,
  // otros métodos...
};

```

Promocionar Restaurante desde "Mis Restaurantes"

Añadir una lógica para promocionar un restaurante desde la pantalla de "Mis restaurantes".

javascript



```

const promocionarRestaurante = async (req, res) => {
  try {
    const { restauranteId } = req.params;
    const propietarioId = req.user.id;

    const restaurante = await Restaurante.findById(restauranteId);

    if (!restaurante || restaurante.propietario.toString() !== propietarioId) {
      return res.status(404).json({ error: 'Restaurante no encontrado.' });
    }

    const restaurantesPromocionados = await Restaurante.find({ propietario: propietarioId });

    for (const resto of restaurantesPromocionados) {
      resto.promocionado = false;
      await resto.save();
    }

    restaurante.promocionado = true;
    await restaurante.save();

    res.status(200).json(restaurante);
  } catch (error) {
    res.status(500).json({ error: 'Error al promocionar el restaurante.' });
  }
};

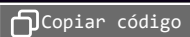
module.exports = {
  promocionarRestaurante,
  // otros métodos...
};

```

3. Endpoints Nuevos o Modificados

Ruta para crear restaurante

javascript



```
const express = require('express');
const router = express.Router();
const { crearRestaurante, promocionarRestaurante } = require('../controllers/restaurante');

router.post('/restaurantes', crearRestaurante);
router.put('/restaurantes/:restauranteId/promocionar', promocionarRestaurante);

module.exports = router;
```


4. Validación y Manejo de Errores

Ya hemos añadido las validaciones necesarias en los controladores para asegurar que solo un restaurante puede estar promocionado por propietario y manejamos los errores apropiadamente.

5. Modificación de Listados

Modificar los listados de restaurantes para que los promocionados aparezcan primero.

javascript

 Copiar código

```
const listarRestaurantes = async (req, res) => {
  try {
    const restaurantes = await Restaurante.find().sort({ promocionado: -1, nombre: 1 });
    res.status(200).json(restaurantes);
  } catch (error) {
    res.status(500).json({ error: 'Error al listar los restaurantes.' });
  }
};

module.exports = {
  listarRestaurantes,
  // otros métodos...
};
```

Resumen

Con estos cambios, has añadido la funcionalidad de promoción de restaurantes de manera segura y eficiente. Asegúrate de probar todos los endpoints y casos de uso para verificar que todo funcione según lo esperado. Si tienes alguna pregunta adicional o necesitas más detalles sobre alguna parte, no dudes en preguntar.

de ahora en adelante cuando te pregunte usa funciones parecidas a esta, si necesitas alguna más avisame y te la envío:

```
const index = async function (req, res) {
  try {
    const restaurants = await Restaurant.findAll(
      {
        attributes: { exclude: ['userId'] },
        include:
          {
            model: RestaurantCategory,
            as: 'restaurantCategory'
          }
      },
```


```
// Solution
order: [['promoted', 'DESC'], [{ model: RestaurantCategory, as:
                                'restaurantCategory' }, 'name', 'ASC']]
    }
    )
    res.json(restaurants)
  } catch (err) {
    res.status(500).send(err)
  }
}
```



Perfecto, adaptaré mi estilo a la función que me has proporcionado. A continuación, te muestro cómo podrías implementar las funciones mencionadas para manejar la creación y promoción de restaurantes:

Crear Restaurante

javascript

 Copiar código

```
const createRestaurant = async function (req, res) {
  try {
    const { name, address, userId, promoted } = req.body;


    if (promoted) {
      const promotedRestaurants = await Restaurant.findAll({ where: { userId, promoted } });
      if (promotedRestaurants.length > 0) {
        return res.status(400).json({ error: 'Ya existe un restaurante promocionado' });
      }
    }

    const newRestaurant = await Restaurant.create({
      name,
      address,
      userId,
      promoted,
    });

    res.status(201).json(newRestaurant);
  } catch (err) {
    res.status(500).send(err);
  }
}
```

Promocionar Restaurante desde "Mis Restaurantes"

javascript

 Copiar código

```
const promoteRestaurant = async function (req, res) {
  try {
    const { restaurantId } = req.params;
    const userId = req.user.id;

    const restaurant = await Restaurant.findOne({ where: { id: restaurantId, userId } });

    if (!restaurant) {
      return res.status(404).json({ error: 'Restaurante no encontrado' });
    }

    restaurant.promoted = true;
    await restaurant.save();

    res.status(200).json(restaurant);
  } catch (err) {
    res.status(500).send(err);
  }
}
```

```
    return res.status(404).json({ error: 'Restaurante no encontrado.' });
  }

  const promotedRestaurants = await Restaurant.findAll({ where: { userId, promoted } });

  for (const resto of promotedRestaurants) {
    resto.promoted = false;
    await resto.save();
  }

  restaurant.promoted = true;
  await restaurant.save();

  res.status(200).json(restaurant);
} catch (err) {
  res.status(500).send(err);
}
}
```

Listar Restaurantes

javascript

 Copiar código

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

Rutas

javascript

 Copiar código

```
const express = require('express');
const router = express.Router();
const { createRestaurant, promoteRestaurant, listRestaurants } = require('../controllers');

router.post('/restaurants', createRestaurant);
router.put('/restaurants/:restaurantId/promote', promoteRestaurant);
router.get('/restaurants', listRestaurants);

module.exports = router;
```

Validaciones y Manejo de Errores

La lógica de validación y manejo de errores ya está incluida en los controladores para garantizar que solo un restaurante puede estar promocionado por propietario y que se manejan adecuadamente los errores.

Resumen

Con estas funciones, tu backend ahora soporta la creación y promoción de restaurantes siguiendo las pautas proporcionadas. Recuerda probar todos los casos posibles para asegurarte de que la lógica funciona correctamente y que los errores se manejan adecuadamente. Si necesitas más funciones o algún detalle adicional, avísame.

explicame detalladamente que hace esta funcion:

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



Claro, vamos a desglosar la función `index` paso a paso para entender completamente su funcionamiento.

Descripción General

La función `index` es un controlador de una ruta en un servidor Express que maneja una solicitud para listar todos los restaurantes de la base de datos. Utiliza Sequelize, un ORM (Object-Relational Mapping) para interactuar con la base de datos. La función realiza lo siguiente:

- 1 Consulta la base de datos para obtener una lista de todos los restaurantes.
- 2 Excluye el campo `userId` de los resultados.
- 3 Incluye información relacionada del modelo `RestaurantCategory`.
- 4 Ordena los restaurantes, primero por el campo `promoted` en orden descendente, y luego por el nombre de la categoría del restaurante en orden ascendente.
- 5 Devuelve los resultados en formato JSON.

- 6 Maneja cualquier error que ocurra durante el proceso y devuelve una respuesta con estado 500 en caso de error.

Análisis Detallado

javascript

 Copiar código

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

1. Inicio de la Función y Manejo de Errores

javascript


 Copiar código

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

La función comienza con una estructura **try-catch** para manejar errores. Si ocurre un error durante la ejecución del bloque **try**, el bloque **catch** captura el error y envía una respuesta con el código de estado HTTP 500, indicando un error del servidor, junto con el error mismo.

2. Consulta a la Base de Datos

javascript

 Copiar código


```
const restaurants = await Restaurant.findAll({
  attributes: { exclude: ['userId'] },
  include: {
    model: RestaurantCategory,
    as: 'restaurantCategory'
  },
  order: [['promoted', 'DESC'], [{ model: RestaurantCategory, as: 'restaurantCategory' }]];
});
```

Aquí es donde se realiza la consulta a la base de datos utilizando Sequelize:

- `Restaurant.findAll({ ... })`: `findAll` es un método de Sequelize que se usa para obtener todas las entradas de un modelo (`Restaurant` en este caso).

3. Exclusión de Atributos

javascript

 Copiar código

```
attributes: { exclude: ['userId'] },
```

Esta línea indica que se deben excluir los campos `userId` de los resultados. Esto significa que, aunque el campo `userId` esté presente en el modelo `Restaurant`, no se incluirá en los resultados devueltos.

4. Inclusión de Relaciones

```
javascript
```

 Copiar código

```
include: {  
  model: RestaurantCategory,  
  as: 'restaurantCategory'  
},
```

Esta parte especifica que se deben incluir datos del modelo relacionado `RestaurantCategory`:

- `model: RestaurantCategory`: Indica el modelo relacionado que se debe incluir en los resultados.
- `as: 'restaurantCategory'`: Especifica el alias utilizado para la relación. Esto debe coincidir con la definición en las asociaciones de Sequelize.

5. Ordenamiento de Resultados

```
javascript
```

 Copiar código

```
order: [['promoted', 'DESC'], [{ model: RestaurantCategory, as: 'restaurantCategory' }],
```

Esta línea define el orden en que se deben devolver los resultados:

- `['promoted', 'DESC']`: Ordena los restaurantes primero por el campo `promoted` en orden descendente (`DESC`). Los restaurantes promocionados aparecerán primero.
- `[{ model: RestaurantCategory, as: 'restaurantCategory' }, 'name', 'ASC']`: Luego, dentro de cada grupo (promocionado/no promocionado), ordena los restaurantes por el nombre de su categoría (`restaurantCategory.name`) en orden ascendente (`ASC`).

6. Envío de la Respuesta

```
javascript
```

 Copiar código

```
res.json(restaurants);
```

Después de obtener los resultados de la base de datos y ordenarlos según lo especificado, se envían de vuelta al cliente en formato JSON.

Resumen

En resumen, la función `index` realiza una consulta a la base de datos para obtener todos los restaurantes, excluye el campo `userId`, incluye información de la categoría de cada restaurante, ordena los resultados para que los restaurantes promocionados aparezcan primero y dentro de cada grupo, los restaurantes se ordenen por el nombre de su categoría. Finalmente, devuelve los resultados en formato JSON y maneja cualquier error que pueda ocurrir durante el proceso.

ahora lo msimo con esta funcion:

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



Vamos a desglosar la función `indexOwner` paso a paso para entender completamente su funcionamiento.


Descripción General

La función `indexOwner` es un controlador de una ruta en un servidor Express que maneja una solicitud para listar todos los restaurantes de un propietario específico (el usuario autenticado). Utiliza Sequelize para interactuar con la base de datos. La función realiza lo siguiente:

- 1 Consulta la base de datos para obtener una lista de todos los restaurantes pertenecientes al usuario autenticado.
- 2 Excluye el campo `userId` de los resultados.
- 3 Incluye información relacionada del modelo `RestaurantCategory`.
- 4 Ordena los restaurantes por el campo `promoted` en orden descendente.
- 5 Devuelve los resultados en formato JSON.
- 6 Maneja cualquier error que ocurra durante el proceso y devuelve una respuesta con estado 500 en caso de error.

Análisis Detallado

javascript

 Copiar código

```
const indexOwner = async function (req, res) {
  try {
    const restaurants = await Restaurant.findAll({
      attributes: { exclude: ['userId'] },
```


```

    where: { userId: req.user.id },
    order: [['promoted', 'DESC']],
    include: [{
      model: RestaurantCategory,
      as: 'restaurantCategory'
    }]
  });
  res.json(restaurants);
} catch (err) {
  res.status(500).send(err);
}
}

```

1. Inicio de la Función y Manejo de Errores

javascript

 Copiar código

```


const indexOwner = async function (req, res) {
  try {
    ...
  } catch (err) {
    res.status(500).send(err);
  }
}

```

La función comienza con una estructura `try-catch` para manejar errores. Si ocurre un error durante la ejecución del bloque `try`, el bloque `catch` captura el error y envía una respuesta con el código de estado HTTP 500, indicando un error del servidor, junto con el error mismo.

2. Consulta a la Base de Datos

javascript

 Copiar código

```

const restaurants = await Restaurant.findAll({
  attributes: { exclude: ['userId'] },
  where: { userId: req.user.id },
  order: [['promoted', 'DESC']],
  include: [{
    model: RestaurantCategory,
    as: 'restaurantCategory'
  }]
});


```

Aquí es donde se realiza la consulta a la base de datos utilizando Sequelize:

- `Restaurant.findAll({ ... }): findAll` es un método de Sequelize que se usa para obtener todas las entradas de un modelo (`Restaurant` en este caso).

3. Exclusión de Atributos

javascript

 Copiar código

```


attributes: { exclude: ['userId'] },

```

Esta línea indica que se deben excluir los campos `userId` de los resultados. Esto significa que, aunque el campo `userId` esté presente en el modelo `Restaurant`, no se incluirá en los resultados devueltos.

4. Filtrado por Propietario

javascript

 Copiar código

```
where: { userId: req.user.id },
```

Esta línea añade un filtro a la consulta para que solo se devuelvan los restaurantes que pertenecen al usuario autenticado (`req.user.id`). `req.user.id` se asume que está disponible a través de middleware de autenticación que añade el objeto `user` al objeto `req`.

5. Ordenamiento de Resultados

javascript

 Copiar código

```
order: [['promoted', 'DESC']],
```

Esta línea define el orden en que se deben devolver los resultados:

- `[['promoted', 'DESC']]`: Ordena los restaurantes por el campo `promoted` en orden descendente (`DESC`). Los restaurantes promocionados aparecerán primero.

6. Inclusión de Relaciones

javascript

 Copiar código

```
include: [{  
  model: RestaurantCategory,  
  as: 'restaurantCategory'  
}]
```

Esta parte especifica que se deben incluir datos del modelo relacionado `RestaurantCategory`:

- `model: RestaurantCategory`: Indica el modelo relacionado que se debe incluir en los resultados.
- `as: 'restaurantCategory'`: Especifica el alias utilizado para la relación. Esto debe coincidir con la definición en las asociaciones de Sequelize.

7. Envío de la Respuesta

javascript

 Copiar código

```
res.json(restaurants);
```

Después de obtener los resultados de la base de datos y ordenarlos según lo especificado, se envían de vuelta al cliente en formato JSON.

Resumen

En resumen, la función `indexOwner` realiza una consulta a la base de datos para obtener todos los restaurantes pertenecientes al usuario autenticado, excluye el campo `userId`, incluye información de la categoría de cada restaurante, ordena los resultados para que los restaurantes promocionados aparezcan primero, y devuelve los resultados en formato JSON. Además, maneja cualquier error que pueda ocurrir durante el proceso y devuelve una respuesta con el código de estado HTTP 500 y el error mismo en caso de que algo salga mal.

ahora con esta funcion:

```
const promote = async function (req, res) {  
  const t = await sequelizeSession.transaction()  
  try {
```

```
const existingPromotedRestaurant = await
Restaurant.findOne({where: {userId: req.user.id, promoted:
true}})
if (existingPromotedRestaurant) {
await Restaurant.update(
{promoted: false},
{where: {id: existingPromotedRestaurant.id}},
{transaction: t}
)
}
await Restaurant.update(
{promoted: true},
{where: {id: req.params.restaurantId}},
{transaction: t}
)
await t.commit();
const updatedRestaurant = await
Restaurant.findByPk(req.params.restaurantId)
res.json(updatedRestaurant)
}
catch(err) {
await t.rollback()
res.status(500).send(err)
}
}
```



Vamos a desglosar la función `promote` paso a paso para entender completamente su funcionamiento.

Descripción General

La función `promote` es un controlador de una ruta en un servidor Express que maneja la promoción de un restaurante específico para el usuario autenticado. Utiliza Sequelize para interactuar con la base de datos y gestiona transacciones para asegurar que las actualizaciones sean atómicas. La función realiza lo siguiente:

- 1 Inicia una transacción de base de datos.
- 2 Busca un restaurante promocionado existente del usuario autenticado.
- 3 Si existe un restaurante promocionado, lo despromociona.
- 4 Promociona el restaurante especificado en la solicitud.
- 5 Confirma la transacción si todas las operaciones son exitosas.
- 6 Devuelve el restaurante actualizado en formato JSON.
- 7 Maneja cualquier error que ocurra durante el proceso, revierte la transacción y devuelve una respuesta con estado 500 en caso de error.

Análisis Detallado


javascript

 Copiar código

```
const promote = async function (req, res) {
  const t = await sequelizeSession.transaction();
  try {
    const existingPromotedRestaurant = await Restaurant.findOne({ where: { userId:
    if (existingPromotedRestaurant) {
      await Restaurant.update(
        { promoted: false },
        { where: { id: existingPromotedRestaurant.id } },
        { transaction: t }
      );
    }
    await Restaurant.update(
      { promoted: true },
      { where: { id: req.params.restaurantId } },
      { transaction: t }
    );
    await t.commit();
    const updatedRestaurant = await Restaurant.findByPk(req.params.restaurantId);
    res.json(updatedRestaurant);
  } catch (err) {
    await t.rollback();
    res.status(500).send(err);
  }
};
```

1. Inicio de la Función y Manejo de Transacción

javascript

 Copiar código

```
const t = await sequelizeSession.transaction();
```

La función comienza iniciando una transacción utilizando `sequelizeSession.transaction()`. Esto asegura que todas las operaciones de actualización dentro de la transacción sean atómicas, es decir, todas se completan con éxito o ninguna se aplica.

2. Búsqueda de Restaurante Promocionado Existente

javascript

 Copiar código

```
const existingPromotedRestaurant = await Restaurant.findOne({ where: { userId: req
```

Aquí se realiza una consulta a la base de datos para encontrar si hay un restaurante ya promocionado por el usuario autenticado (`req.user.id`).

3. Despromocionar Restaurante Existente (si aplica)

javascript

 Copiar código

```
if (existingPromotedRestaurant) {
  await Restaurant.update(
    { promoted: false },
    { where: { id: existingPromotedRestaurant.id } },
    { transaction: t }
  );
}
```

Si existe un restaurante promocionado, se actualiza su campo `promoted` a `false` dentro de la transacción. Esto asegura que solo un restaurante esté promocionado por usuario en cualquier momento.

4. Promocionar el Restaurante Especificado

javascript

 Copiar código

```
await Restaurant.update(  
  { promoted: true },  
  { where: { id: req.params.restaurantId } },  
  { transaction: t }  
);
```

Se promociona el restaurante especificado en los parámetros de la solicitud (`req.params.restaurantId`) actualizando su campo `promoted` a `true` dentro de la misma transacción.

5. Confirmación de la Transacción

javascript

 Copiar código

```
await t.commit();
```

Si todas las operaciones anteriores se completan con éxito, se confirma la transacción con `t.commit()`, aplicando todos los cambios a la base de datos.

6. Envío de la Respuesta

javascript

 Copiar código

```
const updatedRestaurant = await Restaurant.findByPk(req.params.restaurantId);  
res.json(updatedRestaurant);
```

Después de confirmar la transacción, se consulta el restaurante actualizado por su clave primaria (`req.params.restaurantId`) y se envía de vuelta al cliente en formato JSON.

7. Manejo de Errores

javascript

 Copiar código

```
catch (err) {  
  await t.rollback();  
  res.status(500).send(err);  
}
```

Si ocurre algún error durante el proceso, se revierte la transacción con `t.rollback()` para asegurar que ninguna de las operaciones se aplique parcialmente, y se envía una respuesta con el código de estado HTTP 500 junto con el error.

Resumen

En resumen, la función `promote` maneja la promoción de un restaurante para el usuario autenticado de forma segura utilizando transacciones para asegurar que las actualizaciones sean atómicas. La función busca un restaurante ya promocionado y lo despromociona si existe, luego promociona el restaurante especificado en la solicitud, confirma la transacción si todas las operaciones son exitosas y devuelve el restaurante actualizado en formato JSON. Si ocurre un error, la transacción se revierte y se maneja el error apropiadamente.

ahora con esta funcion:

```
const checkBussinessRuleOneRestaurantPromotedByOwner =  
  async (ownerId, promotedValue) => {  
    if (promotedValue) {  
      try {  
        const promotedRestaurants = await Restaurant.findAll({where:  
          {userId: ownerId, promoted: true}})  
        if (promotedRestaurants.length !== 0) {  
          return Promise.reject(new Error("You can only promote one  
            restaurant at a time"))  
        }  
      }  
      catch (err){  
        return Promise.reject(new Error(err))  
      }  
    }  
  
    return Promise.resolve('ok')  
  }
```



Vamos a desglosar la función `checkBussinessRuleOneRestaurantPromotedByOwner` paso a paso para entender completamente su funcionamiento.

Descripción General

La función `checkBussinessRuleOneRestaurantPromotedByOwner` verifica una regla de negocio que asegura que un propietario (`ownerId`) solo puede tener un restaurante promocionado a la vez. Si se intenta promocionar un nuevo restaurante cuando ya existe uno promocionado, la función rechaza la operación con un error. Utiliza Sequelize para interactuar con la base de datos.

Análisis Detallado

javascript

Copiar código

```
const checkBussinessRuleOneRestaurantPromotedByOwner = async (ownerId, promotedValue) => {  
  if (promotedValue) {  
    try {  
      const promotedRestaurants = await Restaurant.findAll({ where: { userId: ownerId, promoted: true }})  
      if (promotedRestaurants.length !== 0) {  
        return Promise.reject(new Error("You can only promote one restaurant at a time"))  
      }  
    } catch (err) {  
      return Promise.reject(new Error(err));  
    }  
  }  
  
  return Promise.resolve('ok');  
}
```

1. Definición de la Función

javascript

 Copiar código

```
const checkBussinessRuleOneRestaurantPromotedByOwner = async (ownerId, promotedValue)
```

La función es una función asíncrona que toma dos argumentos:

- `ownerId`: El ID del propietario del restaurante.
- `promotedValue`: Un valor booleano que indica si se desea promocionar el restaurante.

2. Verificación de `promotedValue`

javascript

 Copiar código

```
if (promotedValue) {
```

La función solo realiza la verificación si `promotedValue` es verdadero. Si `promotedValue` es falso, la función no hace nada especial y simplemente resuelve con `'ok'`.

3. Consulta a la Base de Datos

javascript

 Copiar código

```
try {  
  const promotedRestaurants = await Restaurant.findAll({ where: { userId: ownerId,
```

Dentro de un bloque `try`, la función realiza una consulta a la base de datos utilizando Sequelize para encontrar todos los restaurantes que pertenecen al propietario (`ownerId`) y que están promocionados (`promoted: true`).

4. Verificación de la Regla de Negocio

javascript

 Copiar código

```
if (promotedRestaurants.length !== 0) {  
  return Promise.reject(new Error("You can only promote one restaurant at a time"));  
}
```

Si la consulta devuelve uno o más restaurantes (es decir, `promotedRestaurants.length !== 0`), la función rechaza la operación devolviendo una `Promise.reject` con un mensaje de error. Esto indica que ya hay un restaurante promocionado y no se puede promocionar otro.

5. Manejo de Errores de la Base de Datos

javascript


 Copiar código

```
} catch (err) {  
  return Promise.reject(new Error(err));  
}
```

Si ocurre algún error durante la consulta a la base de datos, se captura en el bloque `catch` y la función rechaza la operación devolviendo una `Promise.reject` con el error capturado.

6. Resolución Exitosa

javascript

 Copiar código

```
return Promise.resolve('ok');
```

Si `promotedValue` es falso o no hay restaurantes promocionados existentes, la función simplemente resuelve con `'ok'`, indicando que la operación puede proceder sin problemas.

Resumen

En resumen, la función `checkBussinessRuleOneRestaurantPromotedByOwner` verifica si un propietario ya tiene un restaurante promocionado antes de permitir la promoción de otro. Si se intenta promocionar un nuevo restaurante mientras ya hay uno promocionado, la función rechaza la operación con un error. Utiliza Sequelize para realizar la consulta a la base de datos y maneja cualquier error que pueda ocurrir durante el proceso. Si no hay problemas, la función resuelve con `'ok'`, permitiendo que la operación continúe.

ahora con esta funcion:

```
app.route('/restaurants/:restaurantId/promote')
  .patch(
    isLoggedIn,
    hasRole('owner'),
    checkEntityExists(Restaurant,'restaurantId'),
    RestaurantMiddleware.checkRestaurantOwnership,
    RestaurantController.promote)
```



Vamos a desglosar la configuración de la ruta `/restaurants/:restaurantId/promote` paso a paso para entender completamente su funcionamiento.

Descripción General

La configuración de la ruta define cómo se maneja una solicitud `PATCH` a la URL `/restaurants/:restaurantId/promote`. Esta ruta se utiliza para promocionar un restaurante específico, identificado por `restaurantId`. La configuración incluye varios middleware que verifican diferentes condiciones antes de llegar al controlador que maneja la promoción del restaurante.

Análisis Detallado

javascript

 Copiar código

```
app.route('/restaurants/:restaurantId/promote')
  .patch(
    isLoggedIn,
    hasRole('owner'),
    checkEntityExists(Restaurant, 'restaurantId'),
    RestaurantMiddleware.checkRestaurantOwnership,
    RestaurantController.promote
  );
```

1. Definición de la Ruta y Método HTTP

javascript


 Copiar código

```
app.route('/restaurants/:restaurantId/promote').patch(...)
```

La ruta se define utilizando `app.route`, que es un método de Express para manejar rutas. Se especifica que esta configuración es para el método HTTP `PATCH` a la URL `/restaurants/:restaurantId/promote`.

2. Middleware: `isLoggedIn`

javascript

 Copiar código

```
isLoggedIn,
```

Este middleware verifica que el usuario que realiza la solicitud esté autenticado. Si el usuario no está autenticado, este middleware probablemente responderá con un error y no permitirá que la solicitud proceda a los siguientes middleware.

3. Middleware: `hasRole('owner')`

javascript

 Copiar código

```
hasRole('owner'),
```

Este middleware verifica que el usuario autenticado tenga el rol de `owner`. Si el usuario no tiene este rol, este middleware responderá con un error y no permitirá que la solicitud proceda a los siguientes middleware.

4. Middleware: `checkEntityExists(Restaurant, 'restaurantId')`

javascript

 Copiar código

```
checkEntityExists(Restaurant, 'restaurantId'),
```

Este middleware verifica que la entidad `Restaurant` con el `restaurantId` proporcionado en la URL existe en la base de datos. Si el restaurante no existe, este middleware responderá con un error y no permitirá que la solicitud proceda a los siguientes middleware.

5. Middleware: `RestaurantMiddleware.checkRestaurantOwnership`

javascript

 Copiar código

```
RestaurantMiddleware.checkRestaurantOwnership,
```

Este middleware verifica que el usuario autenticado es el propietario del restaurante especificado por `restaurantId`. Si el usuario no es el propietario del restaurante, este middleware responderá con un error y no permitirá que la solicitud proceda al controlador.

6. Controlador: `RestaurantController.promote`

javascript

 Copiar código

```
RestaurantController.promote
```

Finalmente, si todos los middleware anteriores pasan con éxito, se llama al controlador `promote` de `RestaurantController`. Este controlador maneja la lógica para promocionar el restaurante, tal como lo desglosamos anteriormente.

Resumen

En resumen, la configuración de la ruta `/restaurants/:restaurantId/promote` define una serie de middleware y un controlador para manejar una solicitud `PATCH` para promocionar un restaurante. La solicitud pasa por los siguientes pasos:

- 1 Verifica que el usuario esté autenticado (`isLoggedIn`).
- 2 Verifica que el usuario tenga el rol de `owner` (`hasRole('owner')`).
- 3 Verifica que el restaurante especificado por `restaurantId` exista (`checkEntityExists(Restaurant, 'restaurantId')`).
- 4 Verifica que el usuario autenticado sea el propietario del restaurante (`RestaurantMiddleware.checkRestaurantOwnership`).
- 5 Finalmente, si todas las verificaciones pasan, se llama al controlador `RestaurantController.promote` para manejar la promoción del restaurante.

Este flujo asegura que solo los propietarios autenticados puedan promocionar sus propios restaurantes y que todas las condiciones necesarias se cumplan antes de proceder con la promoción.

```

                                ahora de esta funcion:
export default function ConfirmationModal (props) {
  return (
    <Modal
      presentationStyle='overFullScreen'
      animationType='slide'
      transparent={true}
      visible={props.isVisible}
      onRequestClose={props.onCancel}>
      <View style={styles.centeredView}>
        <View style={styles.modalView}>
          <TextSemiBold textStyle={{ fontSize: 15 }}>Please confirm
            action</TextSemiBold>
            {props.children}
            <Pressable
              onPress={props.onCancel}
              style={({ pressed }) => [
                {
                  backgroundColor: pressed
                    ? GlobalStyles.brandBlueTap
                    : GlobalStyles.brandBlue
                },
                styles.actionButton
              ]}>
          <View style={[[{ flex: 1, flexDirection: 'row', justifyContent:
            'center' }]]}>
            <MaterialCommunityIcons name='close' color={'white'} size=
              {20}/>
            <TextRegular textStyle={styles.text}>
              Cancel

```

```
        </TextRegular>
      </View>
    </Pressable>
    <Pressable
      onPress={props.onConfirm}
      style={({ pressed }) => [
        {
          backgroundColor: pressed
            ? GlobalStyles.brandPrimaryTap
            : GlobalStyles.brandPrimary
        },
        styles.actionButton
      ]>
    <View style={{ flex: 1, flexDirection: 'row', justifyContent:
      'center' }}>
      <MaterialCommunityIcons name='check-outline' color=
        {'white'} size={20}/>
      <TextRegular textStyle={styles.text}>
        Confirm
      </TextRegular>
    </View>
  </Pressable>
</View>
</View>
</Modal>
)
}
```



Vamos a desglosar el componente `ConfirmationModal` paso a paso para entender completamente su funcionamiento.

Descripción General

El componente `ConfirmationModal` es un componente funcional de React que representa un modal de confirmación. Este modal es usado para pedir al usuario que confirme una acción. Utiliza varios componentes y estilos de React Native para crear la interfaz del modal.

Análisis Detallado

javascript

Copiar código

```
export default function ConfirmationModal(props) {
  return (
    <Modal
      presentationStyle='overFullScreen'
      animationType='slide'
      transparent={true}
      visible={props.isVisible}
      onRequestClose={props.onCancel}
    >
      <View style={styles.centeredView}>
```

```
<View style={styles.modalView}>
  <TextSemiBold textStyle={{ fontSize: 15 }}>Please confirm action</TextSemiBold>
  {props.children}
  <Pressable
    onPress={props.onCancel}
    style={({ pressed }) => [
      {
        backgroundColor: pressed ? GlobalStyles.brandBlueTap : GlobalStyles.brandPrimaryTap,
      },
      styles.actionButton
    ]}
  >
    <View style={{ flex: 1, flexDirection: 'row', justifyContent: 'center' }}>
      <MaterialCommunityIcons name='close' color='white' size={20} />
      <TextRegular textStyle={styles.text}>Cancel</TextRegular>
    </View>
  </Pressable>
  <Pressable
    onPress={props.onConfirm}
    style={({ pressed }) => [
      {
        backgroundColor: pressed ? GlobalStyles.brandPrimaryTap : GlobalStyles.brandBlueTap,
      },
      styles.actionButton
    ]}
  >
    <View style={{ flex: 1, flexDirection: 'row', justifyContent: 'center' }}>
      <MaterialCommunityIcons name='check-outline' color='white' size={20} />
      <TextRegular textStyle={styles.text}>Confirm</TextRegular>
    </View>
  </Pressable>
</View>
</Modal>
);
}
```

Explicación Detallada

1. Definición del Componente

```
javascript
Copiar código
```

```
export default function ConfirmationModal(props) {
```

El componente se define como una función que recibe `props` como argumento. Las propiedades (`props`) permiten personalizar el comportamiento y la apariencia del modal.

2. Modal

```
javascript
Copiar código
```

```
<Modal
  presentationStyle='overFullScreen'
  animationType='slide'
  transparent={true}
  visible={props.isVisible}
  onRequestClose={props.onCancel}
/>
```

El componente `Modal` de React Native se usa para crear el modal de confirmación:

- `presentationStyle='overFullScreen'`: Define que el modal se presenta sobre la pantalla completa.

- `animationType='slide'`: Especifica que el modal se desliza cuando aparece.
- `transparent={true}`: Hace que el fondo del modal sea transparente.
- `visible={props.isVisible}`: Controla si el modal es visible basado en la propiedad `isVisible` pasada desde `props`.
- `onRequestClose={props.onCancel}`: Define la función que se llama cuando se intenta cerrar el modal (por ejemplo, al presionar el botón de atrás en Android).

3. Contenedor del Modal

javascript


 Copiar código

```
<View style={styles.centeredView}>
  <View style={styles.modalView}>
```

Estos `View` componentes definen la estructura y los estilos del contenido del modal. `styles.centeredView` y `styles.modalView` son estilos personalizados que posicionan y estilizan el modal.

4. Título del Modal

javascript

 Copiar código

```
<TextSemiBold textStyle={{ fontSize: 15 }}>Please confirm action</TextSemiBold>
```

Se muestra un texto de título en negrita utilizando el componente `TextSemiBold`, indicando al usuario que debe confirmar la acción. Se aplica un estilo de texto con un tamaño de fuente de 15.

5. Contenido del Modal

javascript


 Copiar código

```
{props.children}
```

Esta línea permite incluir contenido adicional dentro del modal, utilizando la propiedad `children` de React.

6. Botón de Cancelar

javascript

 Copiar código

```
<Pressable
  onPress={props.onCancel}
  style={({ pressed }) => [
    {
      backgroundColor: pressed ? GlobalStyles.brandBlueTap : GlobalStyles.brandBlue,
    },
    styles.actionButton
  ]}
>
  <View style={{ flex: 1, flexDirection: 'row', justifyContent: 'center' }}>
    <MaterialCommunityIcons name='close' color='white' size={20} />
    <TextRegular textStyle={styles.text}>Cancel</TextRegular>
  </View>
</Pressable>
```

Este botón permite cancelar la acción. Utiliza el componente `Pressable` que cambia de color cuando se presiona:

- `onPress={props.onCancel}`: Llama a la función `onCancel` pasada desde `props` cuando el botón es presionado.
- `style={({ pressed }) => [...]}`: Aplica un estilo dinámico basado en si el botón está presionado o no.

7. Botón de Confirmar

javascript

 Copiar código

```
<Pressable
  onPress={props.onConfirm}
  style={({ pressed }) => [
    {
      backgroundColor: pressed ? GlobalStyles.brandPrimaryTap : GlobalStyles.brandPrimary,
    },
    styles.actionButton
  ]}
>
  <View style={{ flex: 1, flexDirection: 'row', justifyContent: 'center' }}>
    <MaterialCommunityIcons name='check-outline' color='white' size={20} />
    <TextRegular textStyle={styles.text}>Confirm</TextRegular>
  </View>
</Pressable>
```

Este botón permite confirmar la acción. Similar al botón de cancelar, utiliza `Pressable` para cambiar de color cuando se presiona:

- `onPress={props.onConfirm}`: Llama a la función `onConfirm` pasada desde `props` cuando el botón es presionado.
- `style={({ pressed }) => [...]}`: Aplica un estilo dinámico basado en si el botón está presionado o no.

Resumen

En resumen, el componente `ConfirmationModal` es un modal de confirmación que utiliza componentes y estilos de React Native para proporcionar una interfaz de usuario atractiva y funcional. El modal incluye:

- Un título que pide al usuario confirmar la acción.
- Contenido adicional pasado como `children`.
- Un botón de cancelar que llama a `props.onCancel` cuando se presiona.
- Un botón de confirmar que llama a `props.onConfirm` cuando se presiona.

El modal es visible o no basado en la propiedad `props.isVisible` y maneja el cierre mediante `props.onCancel`. Los estilos y la disposición se personalizan utilizando los estilos definidos en `styles` y `GlobalStyles`.

ahora con esta parte de la función:

```
<TextRegular>Is it promoted?</TextRegular>
<Switch
  trackColor={{false: GlobalStyles.brandSecondary, true:
    GlobalStyles.brandPrimary}}
  thumbColor={values.promoted? GlobalStyles.brandSecondary:
    '#f4f3f4'}
```

```
value={values.promoted}
style={styles.switch}
onValueChange={value => setFieldValue('promoted', value)}
/>
```



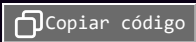
Vamos a desglosar esta parte de la función para entender completamente su funcionamiento.

Descripción General

Este fragmento de código muestra una interfaz de usuario que permite al propietario de un restaurante decidir si un restaurante debe estar promocionado o no. Utiliza componentes de React Native para mostrar un texto de etiqueta y un interruptor (Switch). El estado del interruptor refleja si el restaurante está promocionado y permite al usuario cambiar este estado.

Análisis Detallado

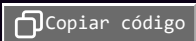
javascript



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

1. Texto de Etiqueta

javascript

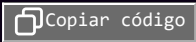


```
<TextRegular>Is it promoted?</TextRegular>
```

El componente `TextRegular` muestra una etiqueta con el texto "Is it promoted?". Este componente probablemente aplica un estilo regular al texto.

2. Interruptor (Switch)

javascript



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

El componente `Switch` de React Native se utiliza para permitir al usuario alternar entre dos estados (promocionado o no promocionado).

Atributos del Switch

- **trackColor:** Define los colores de la pista del interruptor. Utiliza un objeto con dos propiedades:
 - **false:** El color de la pista cuando el interruptor está en la posición "apagado" (`GlobalStyles.brandSecondary`).
 - **true:** El color de la pista cuando el interruptor está en la posición "encendido" (`GlobalStyles.brandPrimary`).
- **thumbColor:** Define el color del "pulgar" del interruptor. Utiliza un operador ternario para determinar el color:
 - `values.promoted ? GlobalStyles.brandSecondary : '#f4f3f4'`: Si `values.promoted` es verdadero, el color es `GlobalStyles.brandSecondary`. Si es falso, el color es `#f4f3f4`.
- **value:** Define el valor del interruptor, que determina si está en la posición "encendido" o "apagado". Se establece en `values.promoted`, lo que refleja si el restaurante está promocionado.
- **style:** Aplica estilos personalizados al interruptor usando `styles.switch`.
- **onValueChange:** Define una función que se llama cuando el valor del interruptor cambia:
 - `value => setFieldValue('promoted', value)`: Esta función actualiza el campo `promoted` en el formulario con el nuevo valor del interruptor. `setFieldValue` es una función de Formik que se utiliza para actualizar el valor de un campo específico en el formulario.

Resumen

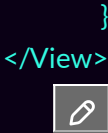
En resumen, esta parte de la función proporciona una interfaz de usuario que permite al propietario de un restaurante decidir si el restaurante debe estar promocionado. Incluye:

- Un texto de etiqueta que pregunta "Is it promoted?".
- Un interruptor (`Switch`) que permite al usuario alternar entre los estados "promocionado" y "no promocionado".

El `Switch` utiliza colores específicos para la pista y el "pulgar" del interruptor, reflejando el estado actual del restaurante. Cuando el usuario cambia el estado del interruptor, la función `onValueChange` actualiza el campo `promoted` en el formulario con el nuevo valor. Esto asegura que el estado del interruptor y el valor del campo `promoted` estén siempre sincronizados.

ahora esta parte de la funcion:

```
<View style={{flexDirection: 'row', justifyContent: 'space-between', alignItems: 'flex-end'}} >
  <TextSemiBold>Shipping: <TextSemiBold textStyle={{ color: GlobalStyles.brandPrimary }}>{item.shippingCosts.toFixed(2)} €</TextSemiBold></TextSemiBold>
  { item.promoted &&
    <TextRegular textStyle={{styles.badge, {color: GlobalStyles.brandPrimary, borderColor: GlobalStyles.brandSuccess}}}>
      ¡En promoción!
    </TextRegular>
  }
```



Vamos a desglosar esta parte de la función para entender su funcionamiento completo.

Descripción General

Este fragmento de código muestra información sobre los costos de envío de un restaurante y una etiqueta de promoción si el restaurante está promocionado. Utiliza componentes de React Native para estructurar y estilizar la interfaz.

Análisis Detallado

javascript

 Copiar código

```
<View style={{ flexDirection: 'row', justifyContent: 'space-between', alignItems:
  <TextSemiBold>Shipping: <TextSemiBold textStyle={{ color: GlobalStyles.brandPrim
    {item.promoted &&
      <TextRegular textStyle={[styles.badge, { color: GlobalStyles.brandPrimary, bor
        ¡En promoción!
      </TextRegular>
    }
  </View>
```

1. Contenedor Principal (View)

javascript

 Copiar código

```
<View style={{ flexDirection: 'row', justifyContent: 'space-between', alignItems:
```

El contenedor principal es un **View** de React Native que organiza su contenido en una fila:

- **flexDirection: 'row':** Organiza los elementos hijos en una fila.
- **justifyContent: 'space-between':** Distribuye el espacio entre los elementos hijos de manera uniforme, colocando el primer hijo a la izquierda y el segundo a la derecha.
- **alignItems: 'flex-end':** Alinea los elementos hijos al final del contenedor (inferior).

2. Costos de Envío (TextSemiBold)

javascript

 Copiar código

```
<TextSemiBold>
  Shipping:
  <TextSemiBold textStyle={{ color: GlobalStyles.brandPrimary }}>
    {item.shippingCosts.toFixed(2)}€
  </TextSemiBold>
</TextSemiBold>
```

Este componente muestra los costos de envío en un texto con estilo en negrita:

- **TextSemiBold:** Un componente de texto que aplica un estilo de negrita.
- **Shipping: ...:** Muestra el texto "Shipping: ".

- `{item.shippingCosts.toFixed(2)}€`: Muestra los costos de envío formateados a dos decimales seguido del símbolo de euro (€). El texto se estiliza con el color definido en `GlobalStyles.brandPrimary`.

3. Etiqueta de Promoción (`TextRegular`)

javascript

 Copiar código

```
{item.promoted &&
  <TextRegular textStyle={[styles.badge, { color: GlobalStyles.brandPrimary, border
    ¡En promoción!
  </TextRegular>
}
```

Esta parte del código muestra una etiqueta de promoción si el restaurante está promocionado (`item.promoted`):

- `item.promoted && ...`: Condición que verifica si el restaurante está promocionado. Si es verdadero, se muestra la etiqueta de promoción.
- `TextRegular`: Un componente de texto que aplica un estilo regular.
- `textStyle`: Aplica múltiples estilos al texto:
 - `styles.badge`: Estilo personalizado para la etiqueta.
 - `color: GlobalStyles.brandPrimary`: Color del texto definido en `GlobalStyles.brandPrimary`.
 - `borderColor: GlobalStyles.brandSuccess`: Color del borde definido en `GlobalStyles.brandSuccess`.
- `¡En promoción!`: El texto que se muestra si el restaurante está promocionado.

Resumen

En resumen, esta parte de la función proporciona una interfaz de usuario que muestra información sobre los costos de envío y si un restaurante está promocionado. Incluye:

- Un contenedor principal (`View`) que organiza su contenido en una fila y distribuye el espacio entre los elementos hijos.
- Un componente de texto en negrita (`TextSemiBold`) que muestra los costos de envío, formateados a dos decimales y estilizados con un color primario.
- Una etiqueta de promoción (`TextRegular`) que se muestra solo si el restaurante está promocionado. La etiqueta se estiliza con colores definidos en `GlobalStyles` y utiliza un estilo personalizado para la etiqueta.

Esto asegura que los costos de envío se muestren de manera prominente y que los restaurantes promocionados se destaquen visualmente con una etiqueta especial.

ahora con esta parte de codigo:

```
<Pressable
  onPress={() => { setRestaurantToBePromoted(item) }}
  style={({ pressed }) => [
    {
      backgroundColor: pressed
        ? GlobalStyles.brandSuccessTap
```

```
      : GlobalStyles.brandSuccess
    },
    styles.actionButton
  ]])>
  <View style={[[{ flex: 1, flexDirection: 'row', justifyContent:
    'center' }]]}>
    <MaterialCommunityIcons name='octagram' color={'white'}
      size={20}/>
    <TextRegular textStyle={styles.text}>
      Promote
    </TextRegular>
  </View>
</Pressable>
```



Vamos a desglosar esta parte del código para entender completamente su funcionamiento.

Descripción General

Este fragmento de código crea un botón usando el componente **Pressable** de React Native. Este botón permite al usuario promocionar un restaurante. Cuando se presiona, se establece el restaurante a ser promocionado utilizando una función de actualización de estado.

Análisis Detallado

javascript



```
<Pressable
  onPress={() => { setRestaurantToBePromoted(item) }}
  style={({ pressed }) => [
    {
      backgroundColor: pressed ? GlobalStyles.brandSuccessTap : GlobalStyles.brandPrimary,
    },
    styles.actionButton
  ]}
>
  <View style={{ flex: 1, flexDirection: 'row', justifyContent: 'center' }}>
    <MaterialCommunityIcons name='octagram' color={'white'} size={20} />
    <TextRegular textStyle={styles.text}>
      Promote
    </TextRegular>
  </View>
</Pressable>
```

1. Botón (Pressable)

javascript



```
<Pressable
  onPress={() => { setRestaurantToBePromoted(item) }}
  style={({ pressed }) => [
    {
      backgroundColor: pressed ? GlobalStyles.brandSuccessTap : GlobalStyles.brandPrimary,
    },
    styles.actionButton
  ]}
>
```


```
  ]}
  >
```

El componente `Pressable` de React Native se utiliza para crear un botón que responde a la presión del usuario:

- **onPress:** Define la función que se ejecuta cuando el botón es presionado. En este caso, se llama a la función `setRestaurantToBePromoted(item)`, que establece el restaurante a ser promocionado (`item`) en el estado.
- **style:** Aplica estilos dinámicos al botón basados en si está presionado o no:
 - **backgroundColor:** `pressed ? GlobalStyles.brandSuccessTap : GlobalStyles.brandSuccess`: Cambia el color de fondo del botón dependiendo de si está presionado (`pressed`) o no. Si está presionado, se usa `GlobalStyles.brandSuccessTap`; de lo contrario, se usa `GlobalStyles.brandSuccess`.
 - **styles.actionButton:** Aplica estilos adicionales definidos en `styles.actionButton`.

2. Contenedor Interno (`View`)

```
javascript
```

 Copiar código


```
<View style={{ flex: 1, flexDirection: 'row', justifyContent: 'center' }}>
```

Este contenedor interno organiza los elementos dentro del botón:

- **flex: 1:** Hace que el contenedor ocupe todo el espacio disponible.
- **flexDirection: 'row':** Organiza los elementos hijos en una fila.
- **justifyContent: 'center':** Centra los elementos hijos horizontalmente.

3. Icono (`MaterialCommunityIcons`)

```
javascript
```

 Copiar código


```
<MaterialCommunityIcons name='octagram' color={'white'} size={20} />
```

Este componente muestra un icono usando la biblioteca `MaterialCommunityIcons`:

- **name='octagram':** Especifica el nombre del icono (en este caso, un octagrama).
- **color={'white'}:** Define el color del icono como blanco.
- **size={20}:** Define el tamaño del icono como 20.

4. Texto del Botón (`TextRegular`)

```
javascript
```

 Copiar código

```
<TextRegular textStyle={styles.text}>
  Promote
</TextRegular>
```

Este componente muestra el texto del botón:

- **TextRegular:** Un componente de texto que aplica un estilo regular.
- **textStyle={styles.text}:** Aplica estilos adicionales definidos en `styles.text`.
- **Promote:** El texto que se muestra en el botón.

Resumen

En resumen, esta parte del código crea un botón interactivo que permite al usuario promocionar un restaurante. Incluye:

- Un componente `Pressable` que responde a la presión del usuario, llamando a una función (`setRestaurantToBePromoted(item)`) cuando se presiona.
- Estilos dinámicos que cambian el color de fondo del botón dependiendo de si está presionado o no.
- Un contenedor interno (`View`) que organiza un ícono (`MaterialCommunityIcons`) y un texto (`TextRegular`) en una fila y los centra horizontalmente.

Este botón permite al usuario seleccionar un restaurante para promocionarlo, proporcionando una interfaz de usuario clara y fácil de usar para esta acción.


ahora de esta función:

```
const promoteRestaurant = async (restaurant) => {
  try {
    await promote(restaurant.id)
    await fetchRestaurants()
    setRestaurantToBePromoted(null)
    showMessage({
      message: `Restaurant ${restaurant.name} succesfully
      promoted`,
      type: 'success',
      style: GlobalStyles.flashStyle,
      titleStyle: GlobalStyles.flashTextStyle
    })
  } catch (error) {
    console.log(error)
    setRestaurantToBePromoted(null)
    showMessage({
      message: `Restaurant ${restaurant.name} could not be
      promoted.`,
      type: 'error',
      style: GlobalStyles.flashStyle,
      titleStyle: GlobalStyles.flashTextStyle
    })
  }
}
```



Esta función `promoteRestaurant` se encarga de promocionar un restaurante. Vamos a desglosarla paso a paso:

javascript

 Copiar código

```
const promoteRestaurant = async (restaurant) => {
  try {
    await promote(restaurant.id)
    await fetchRestaurants()
    setRestaurantToBePromoted(null)
    showMessage({
      message: `Restaurant ${restaurant.name} succesfully promoted`,
      type: 'success',
      style: GlobalStyles.flashStyle,
      titleStyle: GlobalStyles.flashTextStyle
    })
  } catch (error) {
    console.log(error)
    setRestaurantToBePromoted(null)
    showMessage({
      message: `Restaurant ${restaurant.name} could not be promoted.`,
      type: 'error',
      style: GlobalStyles.flashStyle,
      titleStyle: GlobalStyles.flashTextStyle
    })
  }
}
```

Desglose de la función:

1 Bloque try-catch:

- 2 Se utiliza un bloque try-catch para manejar cualquier error que ocurra durante el proceso de promoción del restaurante.

3 Promoción del restaurante:

- 4 `await promote(restaurant.id)`: Llama a la función `promote` para promocionar el restaurante con el ID proporcionado. Este paso espera a que la promoción se complete antes de continuar.

5 Actualización de la lista de restaurantes:

- 6 `await fetchRestaurants()`: Llama a la función `fetchRestaurants` para actualizar la lista de restaurantes después de que se haya promocionado el restaurante.

7 Limpieza del estado:

- 8 `setRestaurantToBePromoted(null)`: Restablece el estado del restaurante a ser promocionado a `null`, lo que indica que no hay restaurante pendiente de promoción.

9 Mostrar mensaje de éxito:

- 10 `showMessage({...})`: Utiliza una función para mostrar un mensaje de éxito mediante una notificación. Esto informa al usuario que la promoción se realizó con éxito. El mensaje incluye el nombre del restaurante promocionado y utiliza estilos globales definidos en `GlobalStyles`.

11 Manejo de errores:

- 12 Si hay algún error durante la promoción del restaurante, se captura en el bloque `catch`.
- 13 `console.log(error)`: Imprime el error en la consola para su registro y seguimiento.
- 14 `setRestaurantToBePromoted(null)`: Restablece el estado del restaurante a ser promocionado a `null`.
- 15 `showMessage({...})`: Muestra un mensaje de error al usuario, indicando que la promoción no se pudo realizar y utilizando estilos globales definidos en `GlobalStyles`.

En resumen, esta función `promoteRestaurant` gestiona el proceso de promoción de un restaurante, actualiza la lista de restaurantes y muestra mensajes de éxito o error al usuario según corresponda. Esto proporciona una experiencia de usuario completa y ayuda a mantener al usuario informado sobre el estado de la promoción del restaurante.

ahora de esta parte del codigo:

```

        return (
          <>
            <FlatList
              style={styles.container}
              data={restaurants}
              renderItem={renderRestaurant}
              keyExtractor={item => item.id.toString()}
              ListHeaderComponent={renderHeader}
              ListEmptyComponent={renderEmptyRestaurantsList}
            />
            <DeleteModal
              isVisible={restaurantToBeDeleted !== null}
              onCancel={() => setRestaurantToBeDeleted(null)}
              onConfirm={() =>
                removeRestaurant(restaurantToBeDeleted)}>
              <TextRegular>The products of this restaurant will be deleted as
                well</TextRegular>
              <TextRegular>If the restaurant has orders, it cannot be deleted.
            </TextRegular>
            </DeleteModal>

            { /* Solution */ }
            <ConfirmationModal
              isVisible={restaurantToBePromoted !== null}
              onCancel={() => setRestaurantToBePromoted(null)}
              onConfirm={() =>
                promoteRestaurant(restaurantToBePromoted)}>
              <TextRegular>Other promoted restaurant, if any, will be
                unpromoted.</TextRegular>
            </ConfirmationModal>
          </>
        )
      }

```



Este fragmento de código muestra la interfaz principal de la página de administración de restaurantes. Describiré cómo funciona esta parte específica del código:

jsx

 Copiar código

```

return (
  <>
    <FlatList
      style={styles.container}
      data={restaurants}
      renderItem={renderRestaurant}
      keyExtractor={item => item.id.toString()}
      ListHeaderComponent={renderHeader}
      ListEmptyComponent={renderEmptyRestaurantsList}
    />
    <DeleteModal
      isVisible={restaurantToBeDeleted !== null}
      onCancel={() => setRestaurantToBeDeleted(null)}
      onConfirm={() => removeRestaurant(restaurantToBeDeleted)}>
      <TextRegular>The products of this restaurant will be deleted as well</TextRegular>
      <TextRegular>If the restaurant has orders, it cannot be deleted.</TextRegular>
    </DeleteModal>

    { /* Solution */ }
    <ConfirmationModal
      isVisible={restaurantToBePromoted !== null}
      onCancel={() => setRestaurantToBePromoted(null)}
      onConfirm={() => promoteRestaurant(restaurantToBePromoted)}>
      <TextRegular>Other promoted restaurant, if any, will be unpromoted.</TextRegular>
    </ConfirmationModal>
  </>
)

```

Desglose:

- 1 FlatList:
 - 2 Se utiliza un componente `FlatList` para mostrar la lista de restaurantes.
 - 3 `data={restaurants}`: La lista de restaurantes que se mostrará.
 - 4 `renderItem={renderRestaurant}`: Función para renderizar cada elemento de la lista.
 - 5 `keyExtractor={item => item.id.toString()}`: Función para extraer claves únicas para cada elemento de la lista.
 - 6 `ListHeaderComponent={renderHeader}`: Componente que se renderiza encima de la lista (por ejemplo, un encabezado).
 - 7 `ListEmptyComponent={renderEmptyRestaurantsList}`: Componente que se renderiza si la lista está vacía.
- 8 DeleteModal:
 - 9 Se utiliza un modal (`DeleteModal`) para confirmar la eliminación de un restaurante.
 - 10 `isVisible={restaurantToBeDeleted !== null}`: Controla la visibilidad del modal.
 - 11 `onCancel={() => setRestaurantToBeDeleted(null)}`: Función para cancelar la eliminación del restaurante.
 - 12 `onConfirm={() => removeRestaurant(restaurantToBeDeleted)}`: Función para confirmar y eliminar el restaurante seleccionado.
 - 13 Contiene texto explicativo sobre las consecuencias de la eliminación del restaurante.
- 14 ConfirmationModal (Solución):

- 15 Se utiliza otro modal (`ConfirmationModal`) para confirmar la promoción de un restaurante.
- 16 `isVisible={restaurantToBePromoted !== null}`: Controla la visibilidad del modal.
- 17 `onCancel={() => setRestaurantToBePromoted(null)}`: Función para cancelar la promoción del restaurante.
- 18 `onConfirm={() => promoteRestaurant(restaurantToBePromoted)}`: Función para confirmar y promocionar el restaurante seleccionado.
- 19 Contiene texto explicativo sobre las consecuencias de promover un restaurante (es decir, que otros restaurantes promocionados pueden ser despromocionados).

En resumen, esta parte del código muestra una lista de restaurantes, permite al usuario eliminar un restaurante con un modal de confirmación y permite al usuario promocionar un restaurante con otro modal de confirmación. Cada modal proporciona una confirmación visual y texto explicativo sobre la acción que se realizará.