饱了么

——饱了么外卖平台

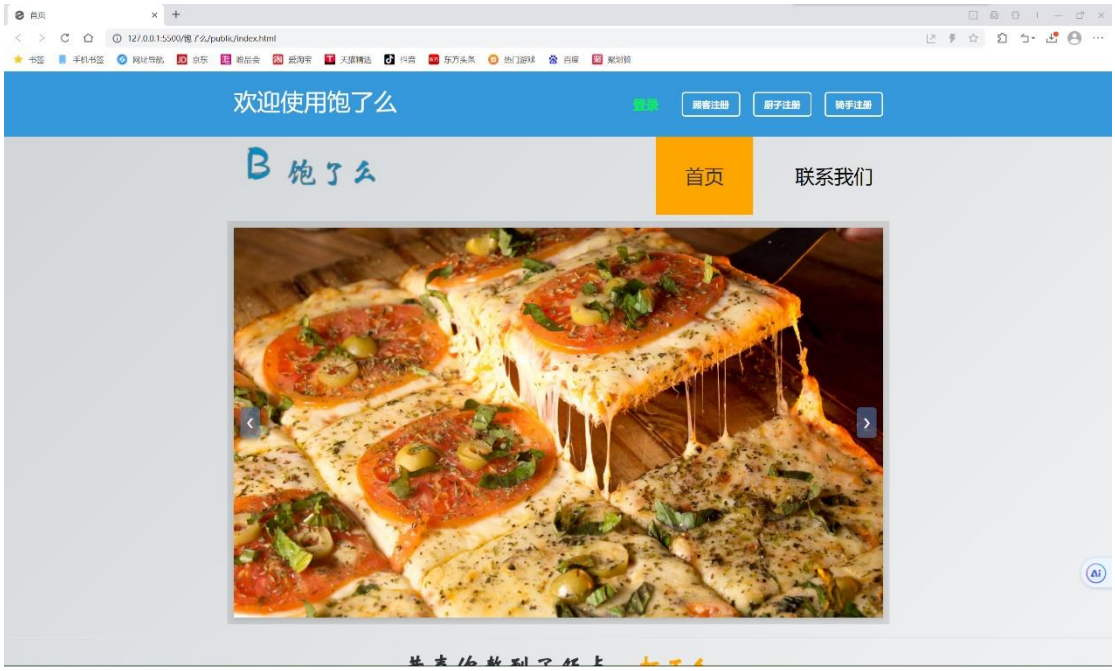
2023-12-17

前端页面设计

饱了么项目的前端页面使用 HTML、CSS 和 JAVASCRIPT 构建。主页版心为 1000 像素，具有顶部导航条、顶部菜单、页面主要内容、底部菜单等板块，并具有交互功能。

一、首页及登录、注册界面

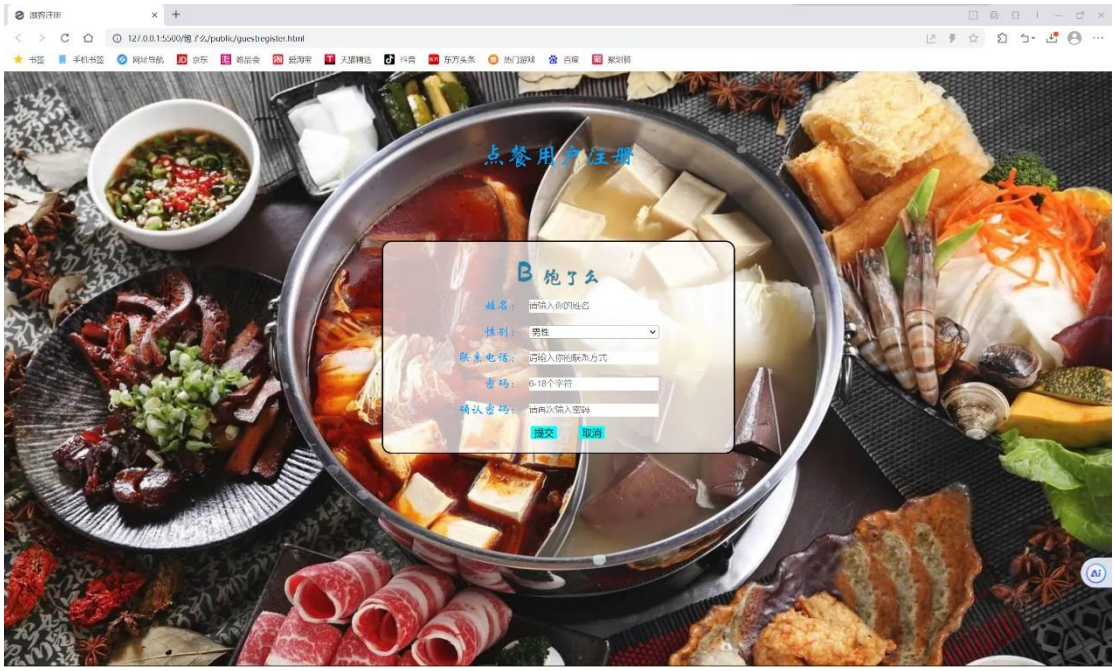
1. 首页：



首页的主要内容是中间的美食展示图，该展示图采用了轮播图的形式，将图片对象存在了数组里，并设置两个按钮，分别添加不同的点击事件来完成数组内图片的循环渲染。此外，还添加了定时器，保证图片能够每秒钟自动切换到下一张。

```
next.addEventListener('click', function () {
    i++;
    if (i >= roundPictures.length) {
        i = 0;
    }
    img.src = roundPictures[i];
})
prev.addEventListener('click', function () {
    i--;
    if (i < 0) {
        i = roundPictures.length - 1;
    }
    img.src = roundPictures[i];
})
let autoPlay = setInterval(function () {
    next.click();
}, 1000)
pic.addEventListener('mouseenter', function () {
    clearInterval(autoPlay);
})
pic.addEventListener('mouseleave', function () {
    clearInterval(autoPlay);
```

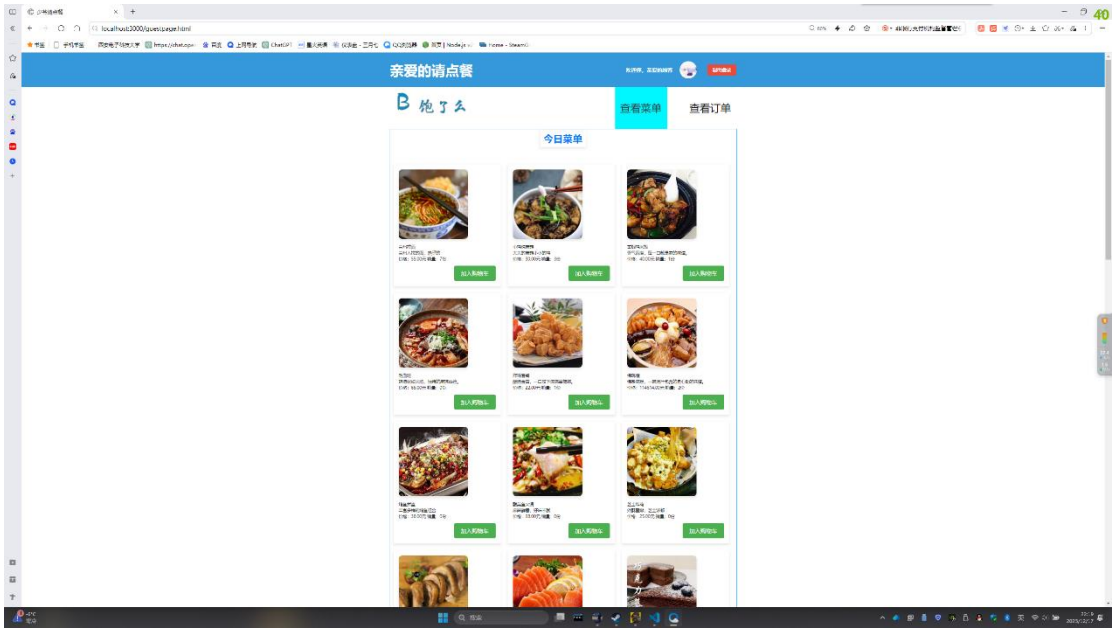
2. 注册和登录界面：



注册界面和登录界面的风格类似，均采用了背景图和一个半透明的 form 表单，表单负责获取用户的输入信息，通过输入合法性检验后，使用 post 方法将获取到的数据发往后端，由后端再进行数据库的比对或者入库。

二、顾客操作界面

1. 顾客点餐界面：



点餐界面是顾客页面的主要功能，该页面的主要内容是菜单和购物车。菜单和购物车的显示采用了动态渲染的方式，前端通过 get 方法将菜单数据从服务器获取到一个 js 数组里，再循环遍历这个数组，每循环一次就产生一个新的 HTML 元素作为菜单框架的子元素显示在页面上，从而能够动态显示菜单数据。

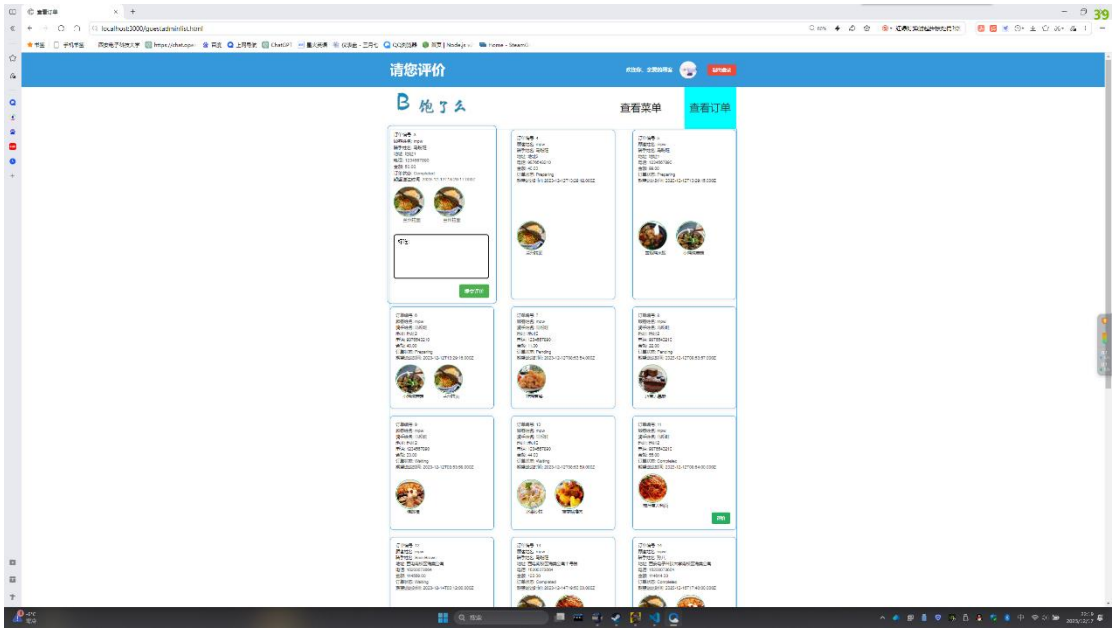
```
async function render() {
  const response = await fetch('/getDishes');
  data = await response.json();
  for (let i = 0; i < data.dishes.length; i++) {
    const li = document.createElement("li");
    li.innerHTML = `
        <img class="dish-picture" src="${data.dishes[i].dish_image_path}" alt="菜品">
        <h3>${data.dishes[i].dish_name}</h3>
        <p>${data.dishes[i].dish_description}</p>
        <div>价格：${data.dishes[i].dish_price}元 销量：${data.dishes[i].dish_sales}份
          <br>
          <p class="rightfix add-to-cart-btn" href="javascript:" data-num=${i}>加入购物车</p>
        </div>
      `;
    li.classList.add("leftfix");
    li.classList.add("dish-block");
    ul.appendChild(li);
  }
}
```

购物车的显示逻辑和菜单相似，我们为菜单菜品上的“加入购物车”按钮添加了点击事件，点击该按钮后会将该菜品对象的信息 push 入购物车数组里，并对购物车板块进行一次循环渲染。当顾客填好有关信息，提交购物车，完成下单时，会生成一个订单对象，该对象会 post 向后端，来由餐厅员工用户和外卖骑手用户查看和完成相应的操作。

```
function renderGogocar() {
  ul2.innerHTML = ""; // 清空ul2内容
  for (let i = 0; i < myGoGoCar.length; i++) {
    const li = document.createElement("li");
    li.innerHTML = `
        <img class="dish-picture" src="${myGoGoCar[i].dish_image_path}" alt="菜品">
        <h3>${myGoGoCar[i].dish_name}</h3>
        <p>${myGoGoCar[i].dish_description}</p>
        <div>价格：${myGoGoCar[i].dish_price}元 销量：${myGoGoCar[i].dish_sales}份
        <br>
        <p class="rightfix cancel-button" href="javascript:" data-num=${i}>取消</p>
        `;
    li.classList.add("leftfix");
    li.classList.add("dish-block");
    ul2.appendChild(li);
  }
}

ul.addEventListener("click", function (e) {
  if (e.target.tagName === "P") {
    myGoGoCar.push(data.dishes[e.target.dataset.num]);
    renderGogocar();
  }
});
```

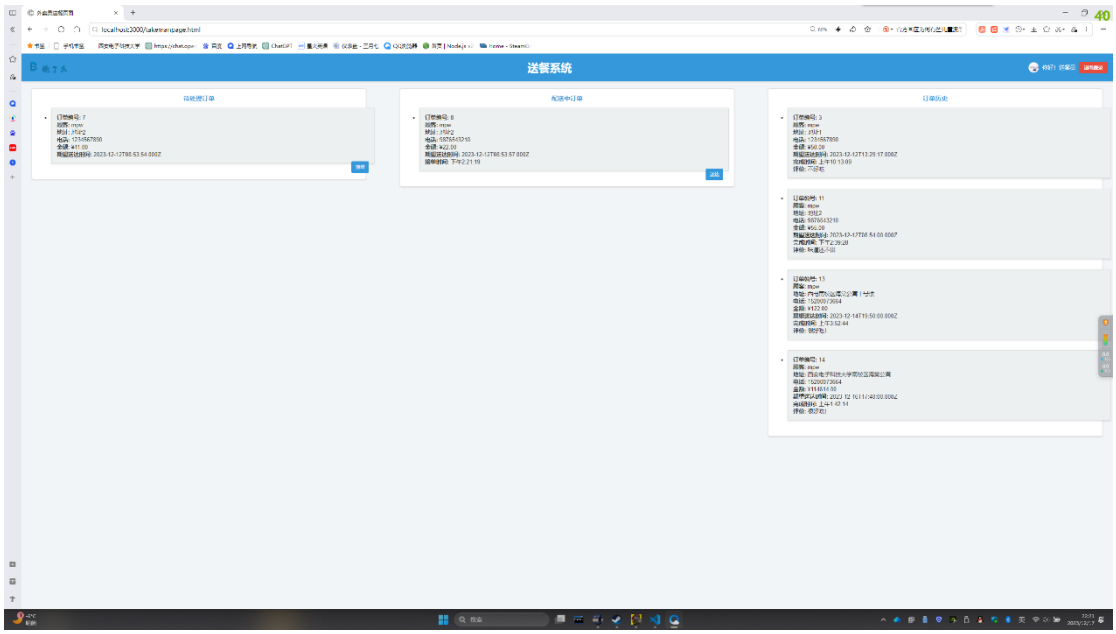
2. 顾客订单页面：



订单也采用动态渲染的方式生成在页面上，当点餐顾客的订单状态为“已送达”时，顾客可以点击评价按钮，此后会弹出文本框，供顾客书写和提交评价。

三、送餐员操作界面

1. 送餐员界面：



送餐员只有一个页面，该页面分为三列，分别用来存储和显示未接单的订单、正在配送的订单和已经送达的订单。当点击接单或已送达按钮的同时，会改变订单的状态属性，并且该订单会从父元素盒子中消失，而被添加到对应状态的盒子中。

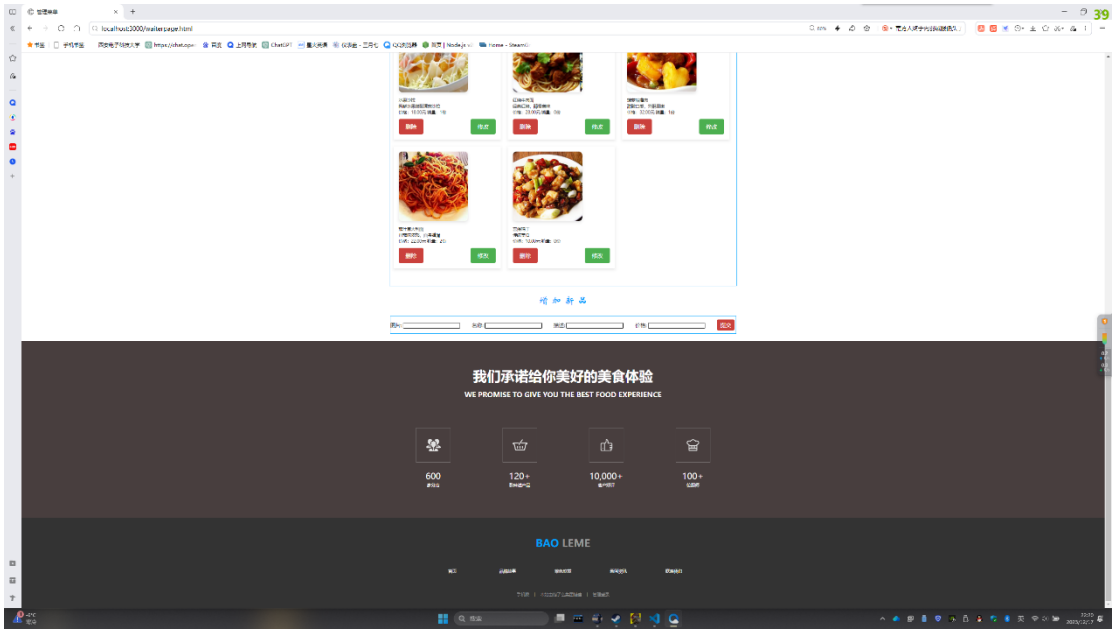
```
if (orders.length > 0) {
    orders.forEach(order => {
        console.log('Processing order:', order);
        const li = document.createElement('li');
        li.className = 'order-item';
        li.innerHTML = `
            <strong>订单编号:</strong> ${order.order_id} <br>
            <strong>顾客:</strong> ${order.customer_name} <br>
            <strong>地址:</strong> ${order.address} <br>
            <strong>电话:</strong> ${order.contact_number} <br>
            <strong>金额:</strong> ¥${order.amount} <br>
            <strong>期望送达时间:</strong> ${order.expectation_time} <br>
        `;

        const status = order.order_status.charAt(0).toUpperCase() + order.order_status.slice(1);
        console.log('Order status:', status);

        if (status === 'Pending') {
            li.innerHTML += `<button onclick="acceptOrder(${order.order_id})">接单</button>`;
            orderListElement.appendChild(li);
        } else if (status === 'Delivering') {
            li.innerHTML += `<strong>接单时间:</strong> ${formatTime(order.acceptance_time)} <br>`;
            li.innerHTML += `<button onclick="completeOrder(${order.order_id})">送达</button>`;
            currentOrderListElement.appendChild(li);
        } else if (status === 'Completed') {
            li.innerHTML += `<strong>完成时间:</strong> ${formatTime(order.completion_time)} <br>`;
            li.innerHTML += `<strong>评价:</strong> ${order.review} <br>`;
            orderHistoryElement.appendChild(li);
        }
    });
} else {
    console.log('No orders available.');
```

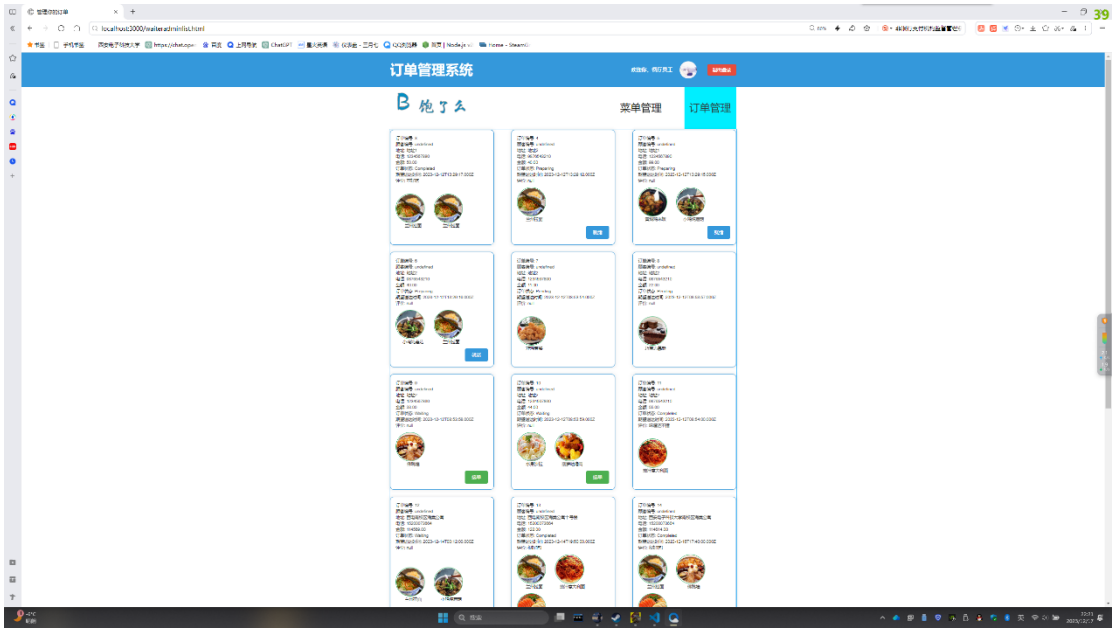
四、餐厅员工界面

1. 餐厅员工菜单管理页面：



餐厅员工的主要需求是对菜单和订单的管理，菜单管理部分我们主要实现餐厅员工对菜单的增加、删除和修改。我们为增加菜品设计了用于提交的表单，用户可在表单中输入菜品的图片、价格、名称等信息，提交表单后，数据被发送到后端，数据库做出了相应的添加操作，并且会在页面上再进行一次渲染，使得员工能够看到自己修改之后的菜单。我们给每个被渲染到页面上的菜品都添加了一个删除按钮，用于完成菜品的删除操作，当点击该按钮后，向后端发送请求去删除数据库里的这一条数据，删除目标菜品的新菜单也会重新渲染在页面上。修改操作通过我们为菜品对象添加的修改按钮完成，该按钮添加了点击事件，触发该事件后会弹出一个修改菜品的表单，并且会产生一个半透明的黑色背景遮罩，在表单中重新输入菜品的信息提交后，会向后端发出请求，修改掉数据库中的这条数据，并会进行新的渲染。

2. 餐厅员工订单管理页面



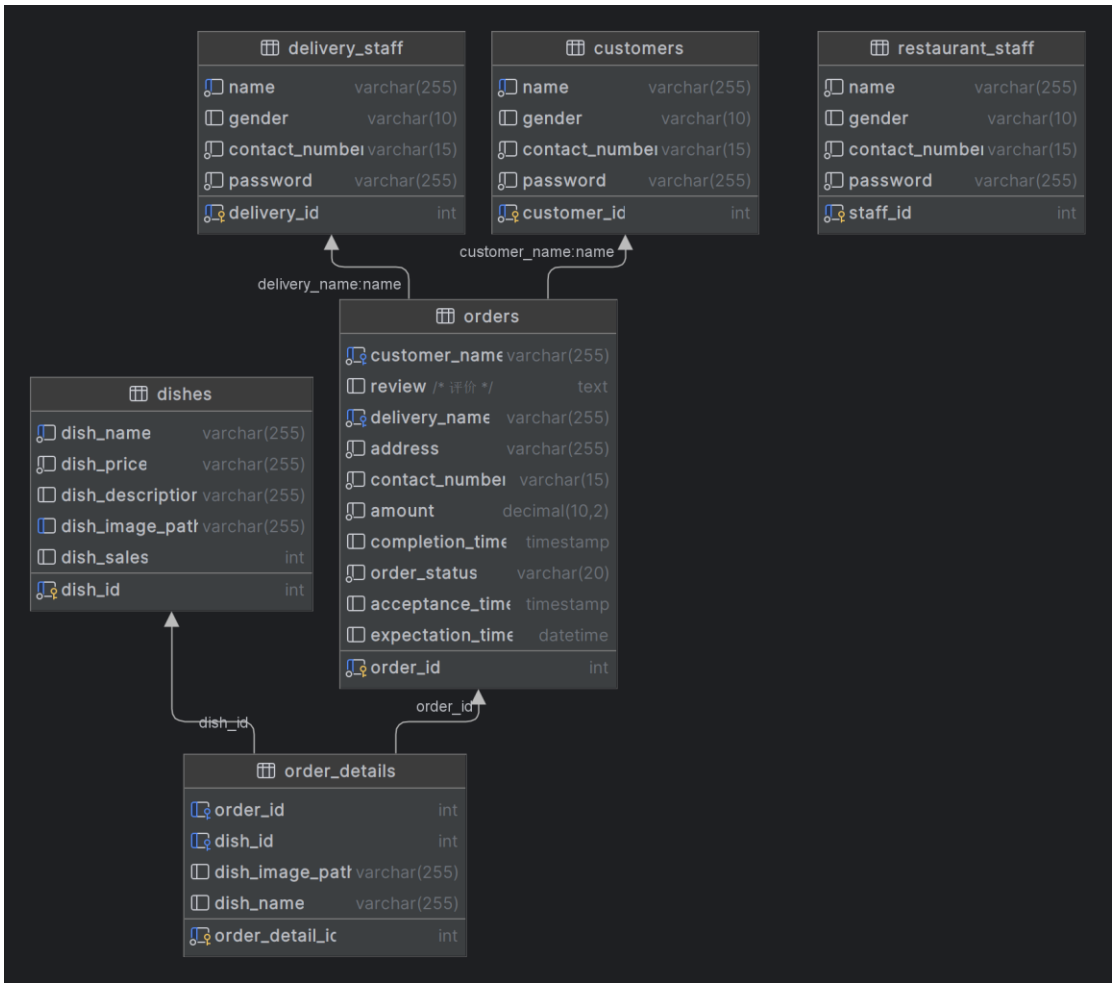
该页面主要完成餐厅员工对订单状态的管理，通过 get 方法会将订单数据从后端获取过来，通过渲染函数使其呈现在页面上。再通过点击按钮调用函数来改变订单状态，并将修改后状态的订单传送给后端。

数据库设计及说明

在本项目中，所采用的数据库为 Mysql

一、图示

主要设计了 6 个表



二、详细说明

1. CUSTOMERS 表

存储顾客的基本信息，如姓名、联系方式等。

字段	类型	约束	索引	备注
customer_id	INT	主键，自增		顾客信息
name	VARCHAR(255)	不为空	idx_name	
gender	VARCHAR(10)	可为空		
contact_number	VARCHAR(15)	不为空		
password	VARCHAR(255)	不为空		

2. RESTAURANT STAFF 表

存储餐厅员工的基本信息，如姓名、联系方式等。

字段	类型	约束	备注
staff_id	INT	主键，自增	餐厅员工 ID
name	VARCHAR(255)	不为空	姓名
gender	VARCHAR(10)	可为空	性别
contact_number	VARCHAR(15)	不为空	联系电话
password	VARCHAR(255)	不为空	密码

3. DELIVERY STAFF 表

存储配送员的基本信息，如姓名、联系方式等。

字段	类型	约束	索引	备注
delivery_id	INT	主键，自增		配送员 ID

name	VARCHAR(255)	不为空	idx_name_delivery_staff	配送员姓名
gender	VARCHAR(10)	可为空		性别
contact_number	VARCHAR(15)	不为空		联系电话
password	VARCHAR(255)	不为空		密码

4. DISHES 表

存储菜品的相关信息，包括菜品名称、价格、描述等。

字段	类型	约束	索引	备注
dish_id	INT	主键，自增	idx_dish_id	菜品 ID
dish_name	VARCHAR(255)	不为空	idx_dish_name	菜品名
dish_price	VARCHAR(255)	不为空		菜品价格
dish_description	VARCHAR(255)	可为空		菜品描述
dish_image_path	VARCHAR(255)	可为空	idx_dish_image_path	菜品图片路径
dish_sales	INT	默认 0，可为空		菜品销量

索引：

idx_dish_id：对 dish_id 列的索引，用于提高根据菜品 ID 进行查询的效率。

idx_dish_image_path：对 dish_image_path 列的索引，用于提高根据菜品图片路径进行查询的效率。

idx_dish_name：对 dish_name 列的索引，用于提高根据菜品名进行查询的效率。

5. ORDERS 表

存储订单的详细信息，包括顾客信息、配送信息、订单状态等。

字段	类型	约束	外键	备注
order_id	INT	主键，自增		订单 ID
customer_name	VARCHAR(255)	不为空	fk_orders_customer	顾客姓名
review	TEXT	可为空		评价
delivery_name	VARCHAR(255)	不为空	fk_orders_delivery_staff	配送员姓名
address	VARCHAR(255)	不为空		配送地址
contact_number	VARCHAR(15)	不为空		联系电话
amount	DECIMAL(10, 2)	不为空		订单金额
completion_time	TIMESTAMP	可为空		订单完成时间
order_status	VARCHAR(20)	默认 'Waiting'，不为空		订单状态
acceptance_time	TIMESTAMP	可为空		订单接受时间
expectation_time	DATETIME	可为空		订单预计送达时间

外键:

fk_orders_customer：关联到 customer_name 列至 customers 表的 name 列，确保顾客信息的一致性。

fk_orders_delivery_staff：关联到 delivery_name 列至 delivery_staff 表的 name 列，确保配送员信息的一致性。

6. ORDER DETAILS 表

存储订单中每个菜品的详细信息，包括菜品 ID、订单 ID 等。

字段	类型	约束	外键	索引	备注
order_detail_id	INT	主键，自增			订单详情 ID
order_id	INT	可为空	order_details_ibfk_1	order_id	订单 ID
dish_id	INT	可为空	fk_dish_id		菜品 ID
dish_image_path	VARCHAR(255)	可为空			菜品图片路径
dish_name	VARCHAR(255)	可为空			菜品名

外键：

fk_dish_id：关联到 dish_id 列至 dishes 表的 dish_id 列，级联删除，确保与菜品表的关联关系。

order_details_ibfk_1：关联到 order_id 列至 orders 表的 order_id 列，确保与订单表的关联关系。

索引：

order_id：对 order_id 列的索引，用于提高根据订单 ID 进行查询的效率。

三、触发器说明

1、BEFORE_INSERT_UPDATE_ALL_TABLES 触发器

```
create definer = root@localhost trigger before_insert_update_all_tables
    before insert
    on customers
    for each row
BEGIN
    IF EXISTS (
        SELECT 1 FROM (
            SELECT contact_number FROM restaurant_staff
            UNION
            SELECT contact_number FROM delivery_staff
            UNION
            SELECT contact_number FROM customers
        ) AS all_contacts
        WHERE all_contacts.contact_number = NEW.contact_number
    ) THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Duplicate contact number in customers, restaurant_staff, or delivery_staff table';
    END IF;
END;
```

在插入 Customers 表数据之前，检查联系电话是否在 restaurant_staff、delivery_staff 或 customers 表中存在重复。如果存在重复，触发 SQLSTATE '45000' 错误，阻止插入操作。

2、UPDATE_DISH_SALES 触发器

```
create definer = root@localhost trigger update_dish_sales
    after insert
    on order_details
    for each row
BEGIN
    -- 菜品ID
    DECLARE dish_id_val INT;

    -- 获取新插入的菜品ID
    SET dish_id_val = NEW.dish_id;

    -- 更新菜品表的销量字段（假设你有一个菜品表叫做 dishes，包含字段 dish_id 和 sales）
    UPDATE dishes
    SET dish_sales = dish_sales + 1
    WHERE dish_id = dish_id_val;
END;
```

在插入 Order Details 表数据之后，更新相应菜品的销量信息。获取新插入的菜品 ID，然后在 Dishes 表中更新对应菜品的销量字段。

后端设计

框架：整个后端项目基于 Node.js 平台，使用 Express 框架搭建。Express 是一个轻量级、灵活的 Web 应用框架，它简化了 Node.js 应用的开发流程，提供了强大的路由、中间件和模板引擎等功能，适用于构建各种类型的 Web 和移动应用。

连接数据库

使用 mysql2 库建立与 MySQL 数据库的连接。在模块的开头导入 mysql2 库，然后使用 createConnection 方法创建连接，在每个控制器文件的头部引用该文件以连接数据库。

```
const mysql = require('mysql2');

const connection = mysql.createConnection({
  host: 'localhost',
  user: 'root',
  password: '123456',
  database: 'baoleme',
  waitForConnections: true,
  connectionLimit: 10,
  queueLimit: 0
});
```

通过调用 **connect** 方法建立与数据库的连接。如果在连接过程中出现错误，将记录错误信息；否则，将打印连接成功的消息。

```
connection.connect((err) => {
  if (err) {
    console.error('数据库连接失败: ' + err.stack);
    return;
  }
  console.log('已连接到数据库');
});
```

整个后端项目中主要用到的控制器文件和路由如下图所示，下面将分别从每个控制器文件着笔，介绍整个后端的设计。

```
// routes.js

const express = require('express');
const router = express.Router();
const loginController = require('../controllers/loginController');
const registerController = require('../controllers/registerController');
const orderController = require('../controllers/orderController');
const dishController = require('../controllers/dishController'); // 引入菜品控制器
const reviewController = require('../controllers/reviewController');
const deliveryController = require('../controllers/deliveryController'); // 引入送餐员控制器

// 登录路由
router.post('/login', loginController.loginHandler);

// 注册路由
router.post('/register', registerController.registerHandler);

// 获取订单路由
router.get('/getOrders', orderController.getOrders);

// 更新订单状态路由
router.post('/updateOrderStatus/:orderId/:status', orderController.updateOrderStatus);

// 获取订单详情路由
router.get('/getOrderDetails/:orderId', orderController.getOrderDetails);

// 获取菜品路由，使用 getDishes 函数
router.get('/getDishes', dishController.getDishes);

// 删除菜品路由，使用 deleteDish 函数
router.delete('/deleteDish/:id', dishController.deleteDish);

// 更新菜品路由
router.post('/updateDish/:id', dishController.updateDish);

// 添加菜品路由
router.post('/addDish', dishController.addDish);

// 添加评价路由
router.post('/addReview', reviewController.addReview);

// 获取随机送餐员路由
router.get('/getRandomDeliveryStaff', deliveryController.getRandomDeliveryStaff);

// 提交订单路由
router.post('/submitOrder', orderController.submitOrder);

module.exports = router;
```

一、LOGINCONTROLLER.JS

LOGINCONTROLLER.JS 负责处理用户登录请求，根据用户类型选择不同的表进行查询。通过使用 db.query 进行数据库查询，实现用户身份验证。

1. 处理用户登录请求

在 loginHandler 函数中，通过 Express 中间件解析请求体，获取用户类型（userType）、账户（account）和密码（password）。

```
const loginHandler = (req, res) => {
    const { userType, account, password } = req.body;
```

2. 根据用户类型选择相应的用户表

通过 switch 语句根据用户类型确定要查询的数据库表，分别为顾客表（customers）、餐厅员工表（restaurant_staff）和送餐员表（delivery_staff）。

```
let tableName, accountColumnName;
switch (userType) {
    case 'guest':
        tableName = 'customers';
        accountColumnName = 'contact_number';
        break;
    case 'waiter':
        tableName = 'restaurant_staff';
        accountColumnName = 'contact_number';
        break;
    case 'takeman':
        tableName = 'delivery_staff';
        accountColumnName = 'contact_number';
        break;
    default:
        console.log(userType);
        return res.status(400).json({ error: 'Invalid user type' });
}
```

3. 使用 DB.QUERY 进行数据库查询

构建 SQL 查询语句，使用 db.query 函数执行查询。通过参数化查询，传递用户提供的账户和密码，避免 SQL 注入攻击。

```
const query = `SELECT * FROM ${tableName} WHERE ${accountColumnName} = ? AND password = ?`;
    db.query(query, [account, password], (err, results) => {
```

4. 处理异常情况并返回相应的 HTTP 状态码和错误信息

在数据库查询过程中，对错误进行处理，确保提供适当的响应。如果查询成功，根据查询结果返回成功响应，否则返回身份验证失败的错误响应。

```
        if (err) {
            console.error('Database query error: ' + err.stack);
            return res.status(500).json({ error: 'Database error' });
        }

        if (results.length === 1) {
            const user = results[0];
            // 用户名和密码正确，可以跳转到首页或者返回用户信息
            res.json({ success: true, user });
        } else {
            // 用户名或密码错误，返回错误信息
            res.status(401).json({ error: 'Invalid credentials' });
        }
    });
};
```

二、REGISTERCONTROLLER.JS

REGISTERCONTROLLER.JS 处理用户注册请求，根据用户选择的身份插入新用户记录。通过使用 db.query 进行数据库插入操作。

1. 注册处理函数

在 registerHandler 函数中，用户向后端发起注册请求。通过 Express 中间件解析请求体，获取用户提供的姓名、性别、联系方式、密码和身份。

```
// 注册处理函数
const registerHandler = async (req, res) => {
    const { name, gender, contact_number, password, role } = req.body;
```

2. 根据身份信息选择要插入的表

通过 switch 语句根据用户选择的身份（role）确定要插入的数据库表，分别为顾客表（customers）、餐厅员工表（restaurant_staff）和送餐员表（delivery_persons）。

```
// 根据身份信息选择要插入的表
switch (role) {
    case 'guest':
        tableName = 'customers';
        break;
    case 'waiter':
        tableName = 'restaurant_staff';
        break;
    case 'takeman':
        tableName = 'delivery_persons';
        break;
    default:
        return res.status(400).json({ error: '无效的用户身份。' });
}
```

3. 插入新用户

使用 db.query 执行插入操作，将用户提供的信息插入相应的数据库表。通过参数化查询，避免 SQL 注入攻击。

```
// 插入新用户
db.query(`INSERT INTO ${tableName} (name, gender, contact_number, password) VALUES (?, ?, ?, ?)`, [name, gender, contact_number, password], (err, results) => {
  if (err) {
    console.error(err);

    if (err.code === 'ER_DUP_ENTRY') {
      // 错误码 'ER_DUP_ENTRY' 表示唯一键冲突，即手机号重复
      return res.status(400).json({ error: '该手机号已经注册过了。' });
    }

    return res.status(500).json({ error: '注册失败，请稍后再试。' });
  }

  // 注册成功，返回成功消息或其他信息
  res.status(200).json({ message: '注册成功！' });
});
```

4. 异常处理

使用 try-catch 块捕获可能发生的异常情况，确保在发生错误时提供适当的响应。如果出现异常，将错误信息记录在控制台并返回适当的错误响应给前端。

```
    try {···
    } catch (error) {
        console.error(error);
        res.status(500).json({ error: '注册失败，请稍后再试。' });
    }
};
```

三、ORDERCONTROLLER.JS

ORDERCONTROLLER.JS 的主要作用是实现订单相关操作的业务逻辑，通过与数据库的交互，提供了获取订单列表、更新订单状态、获取订单详情和提交订单等功能。

1. 获取所有订单

接收前端的获取所有订单的请求。

构造 SQL 查询语句，从数据库中选择所有订单的信息。

通过 db.query 执行查询操作，处理查询结果。

将订单信息以 JSON 格式返回给客户端，用于在前端展示所有订单的概要信息。

```
const getOrders = (req, res) => {
    // console.log('Request received for getOrders');
    const query = `
        SELECT
            *
        FROM
            orders
    `;

    db.query(query, (err, results) => {
        if (err) {
            console.error('数据库查询错误:', err);
            return res.status(500).json({ error: '内部服务器错误', errorMessage: err.message });
        }

        // console.log('Orders fetched successfully:', results);
        res.json({ orders: results });
    });
};
```

2. 更新订单状态

接收前端的更新订单状态的请求，提取请求参数中的订单 ID 和新的状态信息。

构造 SQL 更新语句，包括订单状态、接受时间和完成时间（如果提供）。

通过 db.query 执行更新操作，然后返回成功的响应。

```
// 更新订单状态、接受时间和完成时间的函数
const updateOrderStatus = (req, res) => {
    const { orderId, status } = req.params;
    const { acceptanceTime, completionTime } = req.body;

    // 确保提供了orderId和status
    if (!orderId || !status) {
        return res.status(400).json({ error: '无效的参数' });
    }

    // 构造更新查询
    let query = `UPDATE orders SET order_status = ?`;

    const params = [status];

    // 如果提供了接受时间，则将其添加到查询中
    if (acceptanceTime) {···
    }
    // 如果提供了完成时间，则将其添加到查询中
    if (completionTime) {···
    }
    // 添加过滤条件
    query += ` WHERE order_id = ?`;
    params.push(orderId);

    // 执行更新查询
    db.query(query, params, (err, result) => {···
    });
};
```

3. 获取订单详情

通过提供的订单 ID 调用异步函数 db.getOrderDetailsFromDatabase 从数据库中获取订单详情。

等待数据库查询完成后，将订单详情以 JSON 格式返回给客户端。

允许前端展示特定订单的详细信息，包括每个菜品的信息。

```
async function getOrderDetails(req, res) {
    const orderId = req.params.orderId;

    try {
        const orderDetails = await db.getOrderDetailsFromDatabase(orderId);
        res.json({ dishes: orderDetails });
    } catch (error) {
        console.error('Error fetching order details:', error);
        res.status(500).json({ error: 'Internal Server Error' });
    }
}
```

```
async function getOrderDetailsFromDatabase(orderId) {
  const query = 'SELECT * FROM order_details WHERE order_id = ?';
  const [orderDetails] = await connection.promise().query(query, [orderId]);
  return orderDetails;
}
```

4. 提交订单

处理提交订单的请求，包括检查购物车是否为空、验证顾客是否存在、计算订单金额、获取随机的送餐员信息等。

通过多个数据库查询和插入操作，将订单相关信息存储到数据库中。

如果所有步骤都成功，则返回订单提交成功的消息。

```
async function submitOrder(req, res) {
  try {
    const {…
    } = req.body;

    // 检查购物车是否为空
    if (order_details.length === 0) {…
    }

    // 检查顾客是否存在
    db.query('SELECT 1 FROM customers WHERE name = ? AND contact_number = ?', [customer_name, contact_number], function (error, customerExists) {
      if (error) {…
      }

      if (!customerExists.length) {…
      }

      // 计算订单金额
      const orderAmount = order_details.reduce((total, dish) => total + parseFloat(dish.dish_price), 0);
      console.log("orderAmount", orderAmount);

      // 获取随机的送餐员信息
      db.query('SELECT name FROM delivery_staff ORDER BY RAND() LIMIT 1', function (error, deliveryStaff) {
        if (error) {…
        }

        const deliveryName = deliveryStaff[0].name;
        console.log("deliveryName:",deliveryName);

        // 插入订单数据
        db.query('INSERT INTO orders SET ?',
        {…
        }, function (error, orderResult) {
          if (error) {…
          }

          const orderId = orderResult.insertId;
          console.log("orderId:",orderId);

          // 为购物车中的每个菜品插入订单详情
          order_details.forEach(function (dish) {
            // 修改插入订单详情的查询，使用上面生成的 orderId
            db.query('INSERT INTO order_details SET ?', {…
            });
          });

          res.status(200).json({ success: true, message: '订单提交成功！' });
        });
      });
    });
  } catch (error) {…
  }
}
```

四、DISHCONTROLLER.JS

DISHCONTROLLER.JS 充当了与菜品相关的业务逻辑处理模块，负责处理与菜品有关的前端请求，通过与数据库进行交互，实现获取、删除、更新和添加菜品等功能。通过这些功能，前端可以与菜品数据进行交互，使用户能够查看、修改和操作菜品信息。

1 .获取所有菜品

构建 SQL 查询语句，选择所有菜品的相关信息。

通过数据库连接执行查询，并获取结果。

处理查询结果，将菜品列表以 JSON 格式返回给客户端，包括菜品的 ID、名称、描述、价格、销量和图片路径。

```
// 获取所有菜品的函数
const getDishes = (req, res) => {
  const query = `
      SELECT
          dish_id,
          dish_name,
          dish_description,
          dish_price,
          dish_sales,
          dish_image_path
      FROM
          dishes
  `;

  db.query(query, (err, results, fields) => {
    if (err) {···
    }

    res.json({ dishes: results });
  });
};
```

2. 删除菜品

从请求参数中获取要删除的菜品 ID。

使用数据库连接执行删除操作，删除指定 ID 的菜品。

根据删除结果返回相应的响应，如果删除成功，返回成功消息；否则，返回菜品未找到或删除失败的错误信息。

```
const deleteDish = (req, res) => {
  const dishId = req.params.id;

  // 在数据库中删除菜品
  db.query('DELETE FROM dishes WHERE dish_id = ?', [dishId], (err, result) => {
    if (err) {···
    }

    if (result.affectedRows === 1) {
      console.log(`菜品ID为${dishId}的菜品删除成功`);
      res.status(200).json({ message: '菜品删除成功' });
    } else {···
    }
  });
};
```

3. 更新菜品信息

从请求参数中获取要更新的菜品 ID。

从请求体中获取新的菜品信息，包括图片路径、名称、描述和价格。

使用数据库连接执行更新操作，将指定 ID 的菜品信息更新为新的值。

根据更新结果返回相应的响应，如果更新成功，返回成功消息；否则，返回菜品未找到或更新失败的错误信息。

```
const updateDish = async (req, res) => {

  const dishId = req.params.id;
  const { picture, name, desc, price } = req.body;
  try {
    // 更新单个菜品信息
    const updateResult = await db.query(
      'UPDATE dishes SET dish_image_path = ?, dish_name = ?, dish_description = ?, dish_price = ? WHERE dish_id = ?',
      [picture, name, desc, price, dishId]
    );

    if (updateResult.affectedRows === 1) {
      res.json({ success: true, message: '菜品更新成功' });
    } else {…
    }
  } catch (error) {…
  }
};
```

4. 添加新菜品

从请求体中获取新菜品的信息，包括图片路径、名称、描述和价格。

使用数据库连接执行插入操作，将新的菜品信息添加到数据库中。

根据插入结果返回相应的响应，如果插入成功，返回成功消息；否则，返回添加失败的错误信息。

```
const addDish = (req, res) => {
  const { picture, name, desc, price } = req.body;

  // 在数据库中插入新菜品
  db.query(
    'INSERT INTO dishes (dish_image_path, dish_name, dish_description, dish_price) VALUES (?, ?, ?, ?)',
    [picture, name, desc, price],
    (error, results) => {
      if (error) {…
      }

      // 检查是否成功插入
      if (results.affectedRows === 1) {
        // 添加成功，返回成功消息或其他信息
        res.status(200).json({ message: '添加成功！' });
      } else {…
      }
    }
  );
};
```

五、REVIEWCONTROLLER.JS

REVIEWCONTROLLER.JS 主要包含一个函数 addReview，负责处理添加订单评价的请求和相应的业务逻辑。该控制器的主要作用是确保每个订单只能被评价一次，并将评价信息存储到数据库中。评价字段通常用于记录用户对订单的满意度或提供其他相关反馈。通过这个功能，系统能够记录用户的评价并在必要时进行分析或展示。

ADDREVIEW 函数

- 处理添加评价的请求。

- 从请求体中提取订单 ID（**orderId**）和评价内容（**review**）。
- 查询数据库，检查订单是否已经评价过，如果已评价则返回错误信息。
- 如果订单尚未评价，则更新订单的评价字段。
- 返回相应的成功消息或错误信息。

```
// 添加评价
function addReview(req, res) {
  const { orderId, review } = req.body;

  // 检查订单是否已经评价过
  db.query('SELECT * FROM orders WHERE order_id = ?', [orderId], (error, order) => {
    if (error) {···
    }

    if (order && order.length > 0 && order[0].review) {
      return res.status(400).json({ error: '该订单已经评价过了。' });
    }

    // 更新订单的评价字段
    db.query('UPDATE orders SET review = ? WHERE order_id = ?', [review, orderId], (updateError) => {
      if (updateError) {···
      }

      // 返回成功的响应
      res.status(200).json({ message: '评价成功。' });
    });
  });
}
```

六、DELIVERYCONTROLLER.JS

DELIVERYCONTROLLER.JS 主要包含一个函数 getRandomDeliveryStaff，该控制器的主要作用是提供一个端点，用于随机选择一名送餐员，以确定为特定订单分配的送餐员。

GETRANDOMDELIVERYSTAFF 函数

- 处理获取随机送餐员的请求。
- 构造 SQL 查询语句，从数据库中选择一名随机送餐员的信息。
- 使用异步函数等待数据库查询完成，获取查询结果。
- 返回相应的成功消息（包含随机送餐员信息）或错误信息。

```
// 获取随机送餐员
async function getRandomDeliveryStaff(req, res) {
  try {
    const query = 'SELECT * FROM delivery_staff ORDER BY RAND() LIMIT 1';
    const [randomStaff] = await yourDatabaseQueryFunction(query);
    res.status(200).json(randomStaff);
  } catch (error) {
    console.error('Error getting random delivery staff:', error);
    res.status(500).json({ error: 'Failed to get random delivery staff.' });
  }
}
```