数据库系统原理实验：餐厅点餐系统设计

22336104 蒋乐璇 22336106 晋一卓

一. 引言

本次实验从“餐厅点餐场景”为主体对用户“顾客(customer)”“服务员(waiter)”进行确认，通过分析“餐厅点餐”和“后台管理”的具体流程触发时所产生的具体需求对实体对象进行确认。

【餐厅点餐】针对用户：顾客

1. **顾客进入餐厅**:顾客进入餐厅后需要询问餐厅中餐桌情况，即“还有座位吗？”。此时顾客可以查看餐厅中每张餐桌当前的状态，并可选择空闲的餐桌进行点餐。随后选择的餐桌被占用。
   - 因此需要建立餐桌（Table）实体(座位数，桌号，餐桌状态)
   - 使用状态包括：用餐中，空闲，以及预定。顾客无法选择处于“用餐中”以及“预订”状态的餐桌。
   - 若为顾客提前对餐桌进行预约，在顾客到店后将餐桌状态由“预订”转换为“用餐中”。
2. **顾客进行点餐**：顾客选定餐桌后开始点餐，此时涉及三个实体：餐桌、订单(Order)、餐品(Dish)。
   - 通过查看餐品的名称、价格等信息，顾客将选择的餐品加入订单中，此时订单中同时包括顾客所坐的餐桌桌号信息。
3. **顾客提交订单**：顾客提交的订单被同步传送给服务员，服务员根据用户的订单需求为顾客上菜，并计算订单总金额（这也将在顾客点餐时被顾客所知晓）。
4. **顾客用餐结束**，结账：根据订单计算的总金额，服务员协助顾客进行结账。订单完成后，服务员，餐桌的状态将从“用餐中”转换为“空闲”，为接待下一轮顾客进行准备

【后台管理】针对用户：服务员

1. 可对餐桌的状态进行主动修改
2. 可对餐品信息（包括价格、名称）进行调整
3. 可对菜单中餐品进行添加与删除
4. 可查询顾客的订单，并对订单进行修改

- 为实现对餐品信息的调整，额外引出第四个实体：餐品信息(OrderDetail)。
- 为区别点餐系统所服务的不同对象，在系统的初始界面需要进行用户登录：顾客可以直接进入点餐页面，服务员则需要输入正确的密码而进入后台管理。需要额外创建实体：用户(Users)：（用户名，扮演角色，密码）。

【小组成员分工】

蒋乐璇：设计数据库、搭建 Node.js + React + PostgreSQL 整体框架、编写并调试 Login、CustomerPage 页面及相关后端控制文件和路由文件

晋一卓：设计数据库、设计实验报告、编写并调试 WaiterPage 页面及相关后端控制文件和路由文件

二．系统功能需求

1. 服务员：

1. 菜品信息添加、修改、查询
2. 订单信息修改、查询
3. 餐桌状态修改（空闲→忙碌；预定→忙碌；忙碌→空闲）、查询

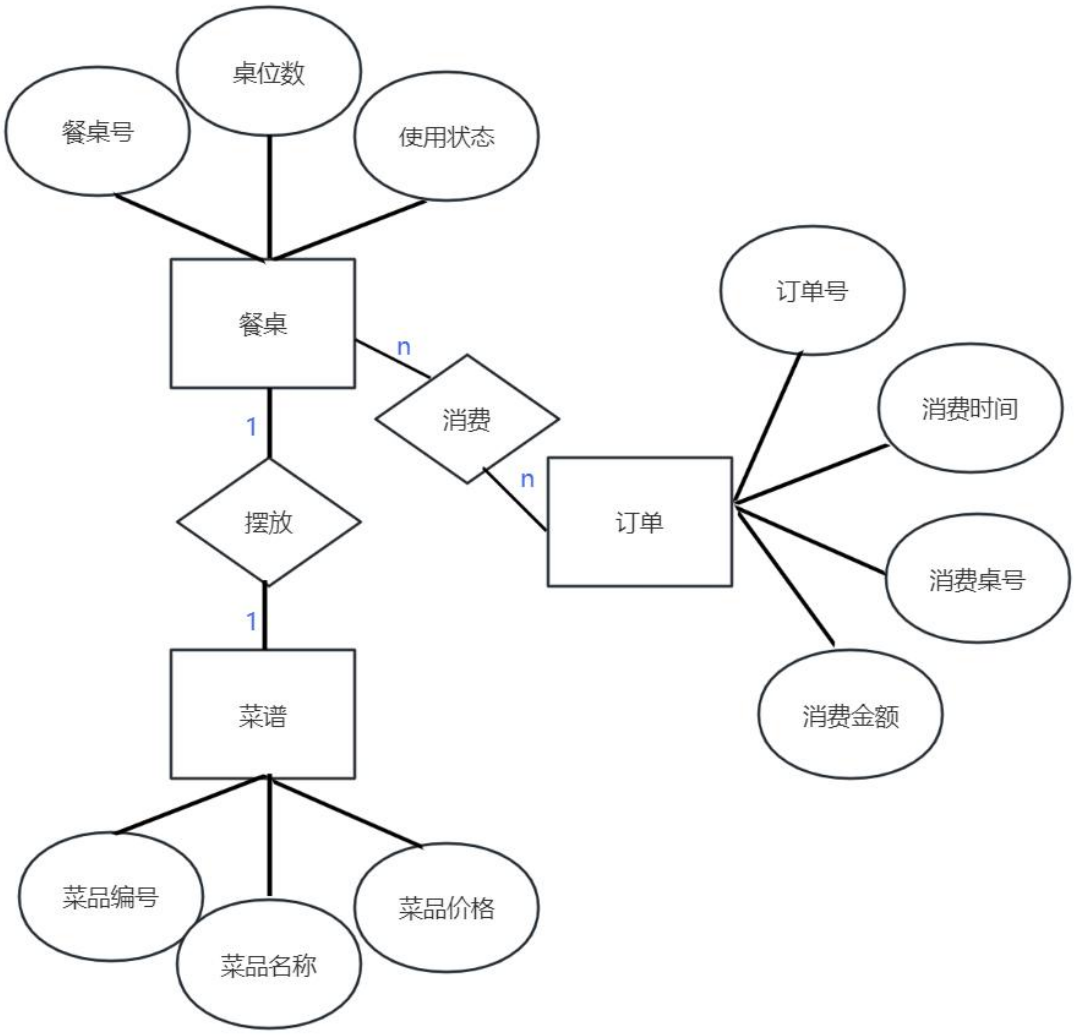
2. 顾客：

1. 菜品查询；
2. 订单录入；
3. 餐桌状态修改（空闲→忙碌），查询

3. 用户管理：用户登录

三．数据库设计（E-R 图，创建表，索引设计）

1. E-R 图



2. 数据库设计

1. 用户表

```sql
CREATE TABLE users (
  username VARCHAR(50) PRIMARY KEY,
  role VARCHAR(50),
  password VARCHAR(255)
);
```

2. 餐桌表

```sql
CREATE TABLE "Table" (
    SeatCount INT,
    TableID INT PRIMARY KEY,
    Status VARCHAR(50)
);
```

3. 餐品表

```sql
CREATE TABLE Dish (
    DishID INT PRIMARY KEY,
    DishName VARCHAR(100),
    Price DECIMAL(10,2)
);
```

4. 餐品信息表

询 查询历史

```sql
-- Table: public.OrderDetail

-- DROP TABLE IF EXISTS public."OrderDetail";

CREATE TABLE IF NOT EXISTS public."OrderDetail"
(
    detailid integer NOT NULL DEFAULT nextval('"OrderDetail_detailid_seq"'::regclass),
    orderid integer,
    dishid integer,
    quantity integer,
    CONSTRAINT "OrderDetail_pkey" PRIMARY KEY (detailid),
    CONSTRAINT "OrderDetail_DishID_fkey" FOREIGN KEY (dishid)
        REFERENCES public.dish (dishid) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION,
    CONSTRAINT "OrderDetail_OrderID_fkey" FOREIGN KEY (orderid)
        REFERENCES public."Order" (orderid) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
)

TABLESPACE pg_default;

ALTER TABLE IF EXISTS public."OrderDetail"
    OWNER to postgres;
```

5. 订单表

```sql
-- DROP TABLE IF EXISTS public."Order";

CREATE TABLE IF NOT EXISTS public."Order"
(
    orderid integer NOT NULL,
    tableid integer,
    consumetime timestamp without time zone,
    totalamount numeric(10,2),
    CONSTRAINT "Order_pkey" PRIMARY KEY (orderid),
    CONSTRAINT "Order_tableid_fkey" FOREIGN KEY (tableid)
        REFERENCES public."Table" (tableid) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
)

TABLESPACE pg_default;

ALTER TABLE IF EXISTS public."Order"
    OWNER to postgres;
```

四．技术实现（技术选型，系统架构，后端实现，前端实现，数据库实现）

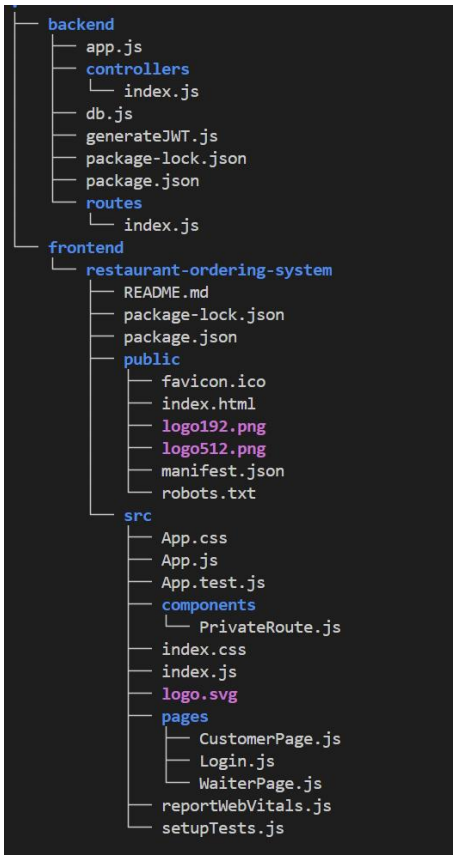
1. 技术选型

- 后端：Node.js Express
- 前端：React
- 数据库：PostgreSQL

2. 系统架构

采用前后端分离设计，包括前端界面、后端服务器与数据库。

文件目录树：



注意：该目录树是在 WSL 子系统下运行命令 tree -I 'node_modules'生成，忽略了已安装好的 node_modules。

3. 后端实现

以下内容均位于./backend 中。

./：

【.env】

环境变量，后端的环境变量主要用于存储敏感信息，即指定数据库名称、密码、用户名、端口、JWT 秘钥等信息，以及定义监听服务器的 api。

【db.js】

配置并导出一个 PostgreSQL 数据库连接池，使得应用程序能够与 PostgreSQL 数据库交互。其中，pg 是 PostgreSQL 的官方 Node.js 客户端库，提供与数据库通信的能力，{ Pool }是从 pg 中导入的连接池类，用于管理多个数据库连接。

```
const { Pool } = require('pg');
require('dotenv').config();

const pool = new Pool({
  host: process.env.DB_HOST,
  port: process.env.DB_PORT,
  user: process.env.DB_USER,
  password: process.env.DB_PASSWORD,
  database: process.env.DB_NAME,
});

module.exports = pool;
```

【generateJWT.js】

用于生成 JWT 秘钥。通过在命令行中执行命令 node generateJWT.js，即可获得 JWT 秘钥，需要配置在前面提到的.env 文件中.

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

// Generate a random secret key
const jwtSecret = crypto.randomBytes(64).toString('hex');
console.log(`Generated JWT Secret: ${jwtSecret}`);
```

【app.js】

该代码实现了一个 Express.js 的后端服务，提供了 API 路由、数据库连接、登录接口处理等功能。

1. 引入所需模块：

```
const express = require('express');
const bodyParser = require('body-parser');
const cors = require('cors');
const { Pool } = require('pg');
const routes = require('./routes');
require('dotenv').config();

const { getUser } = require('./controllers');
```

express: 创建一个 Web 服务器，处理 HTTP 请求和响应。

body-parser: 解析请求体（如 JSON 数据）。

cors: 启用跨域资源共享，允许前端和后端之间的跨域通信。

pg: PostgreSQL 客户端，用于与数据库交互。

routes: 导入自定义的路由模块，用于管理 API 的路由逻辑。

dotenv: 加载环境变量文件 .env 中的变量到 process.env。

从控制器模块（./controllers，后文会详细介绍）中导入 getUser 函数用以处理/login 的登录逻辑。

2. 创建数据库连接池：

```
const pool = new Pool({
  host: process.env.DB_HOST,
  port: process.env.DB_PORT,
  user: process.env.DB_USER,
  password: process.env.DB_PASSWORD,
  database: process.env.DB_NAME,
});
```

3. 配置中间件：这一步很重要，因为与跨域问题息息相关，处理不好会导致跨域问题相关报错。

路由设置：即 app.use（'/api', routes）和 app.post 这两个函数。

```
app.use(bodyParser.json());
// app.use(cors());
app.use(cors({
  origin: 'http://localhost:3000',
  credentials: true,
  methods: ['GET', 'POST', 'PUT', 'DELETE', 'OPTIONS'],
  allowedHeaders: ['Content-Type', 'Authorization']
}));
app.use('/api', routes);
app.post('/login', getUser);
app.options('*', cors()); // 预检请求
```

其中，bodyParser.json(): 用于解析 application/json 类型的请求体，将其转换为 JavaScript 对象。

配置 CORS（跨域资源共享），允许前端（运行在 http://localhost:3000）与后端通信：

origin: 允许的来源。

credentials: 允许发送跨域请求时包含凭据。

methods: 允许的 HTTP 方法，这里为了满足后续的需求，允许 GET、POST、PUT、DELETE、OPTIONS 方法。

allowedHeaders: 允许的请求头。

app.use（'/api', routes）用于路由设置，确保所有以 /api 开头的请求会被转发到 routes 模块处理。

app.post 语句则定义 /login 路由，处理用户登录逻辑，调用 getUser 函数进行处理。

app.options 函数用于处理预检请求，确保复杂的跨域请求（如 Authoriztion）能够成功。

4. 服务器启动：

设置服务器监听的端口号为 5000。

```
const PORT = process.env.PORT || 5000;
app.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}`);
});
```

./controllers：

【index.js】

实现餐厅点餐系统后端功能：

- 用户角色分配与验证
- 餐桌管理：获取与状态更新

• 菜品管理：查询、更新、新增
• 订单管理：查询、更新
（以更新菜品功能的实现为例：）
获取请求函数：从请求路径参数 rep.parmas 中获取菜品的唯一标识符 dishid，同时从请求体 rep.body 中获取需要更新的信息 dishname、price。
使用 pool.connect()获取数据库连接

```
const updateDish = async (req, res) => {
  const { id } = req.params;
  const { dishname, price } = req.body;
  const client = await pool.connect();
```

定义与执行 SQL 语句，若查询成功返回成功响应：状态码 200 与 JSON 消息

```
  try {
    const query = 'UPDATE dish SET dishname = $1, price = $2 WHERE dishid = $3';
    const values = [dishname, price, id];
    await client.query(query, values);
    res.status(200).json({ message: 'Dish updated successfully' });
```

错误处理：若发生错误，打印错误到服务器控制台便于调试，并返回 HTTP 服务器端错误状态码 500 与 JSON 消息给客户端，响应错误。

```
  } catch (error) {
    console.error(error);
    res.status(500).json({ message: 'Internal server error' });
```

最后，释放数据库连接，确保链接被归还到连接池中。

```
  } finally {
    client.release();
  }
};
```

./routes：
【index.js】
通过 express 模块创建 router，定义路由，router 对象注册不同 API 路由机器之间的处理逻辑，同时引入外部 controllers 文件处理路由逻辑。

```
const express = require('express');
const router = express.Router();
const {
  getOrderDetails,
  getUser,
  createOrder,
  updateTableStatus,
  updateDish,
  getTables,
  getDishes,
  addDish,
  getOrders,
  updateOrder,
  queryTableStatus,
  getDishById
} = require('../controllers');
```

对路由及其功能进行定义

```
router.post('/login', getUser);
router.post('/order', createOrder);
router.put('/table/:id', updateTableStatus);
router.put('/dish/:id', updateDish);
router.get('/tables', getTables);
router.get('/dishes', getDishes);
router.post('/dish', addDish);
router.get('/orders', getOrders);
router.put('/order/:id', updateOrder);
router.get('/tables/status', queryTableStatus);
router.get('/order/:orderId', getOrderDetails);
router.get('/dish/:id', async (req, res) => {
```

将 router 模块进行导出

```
module.exports = router;
```

4. 前端实现

【./frontend/.env】

.env 为环境变量，位于项目文件夹根目录，前端的环境变量主要用于指定与后端服务器交互的 URL。

```
frontend > restaurant-ordering-system > .env
1   REACT_APP_API_URL=http://localhost:5000
```

以下内容均位于./frontend/restaurant-ordering-system/src。

./components：

【PrivateRoute.js】

该代码实现了一个 React 路由守卫组件 PrivateRoute，用于控制用户是否有权限访问特定页面。

1. 引入依赖：

```
import React from 'react';
import { Route, Redirect } from 'react-router-dom';
import { jwtDecode } from "jwt-decode";
```

Route 和 Redirect:Route: 定义路由规则，与页面路径匹配后渲染相应组件。Redirect: 如果用户没有访问权限，可以将其重定向到其他页面。

jwtDecode:用于解码 JWT（JSON Web Token）。从存储的令牌中提取用户信息（如角色）

2. 定义相关组件：

```
const PrivateRoute = ({ component: Component, roles, ...rest }) => {
  const token = localStorage.getItem('token');
  const role = token ? jwtDecode(token).role : null;
```

localStorage.getItem('token'): 从本地存储中获取 JWT（用户令牌）。

jwtDecode(token).role：如果令牌存在，使用 jwtDecode 解析其中的用户角色。

3. 渲染逻辑：

```
  return (
    <Route
      {...rest}
      render={(props) =>
        roles.includes(role) ? (
          <Component {...props} />
        ) : (
          <Redirect to="/" />
        )
      }
    />
  );
};
```

<Route>:定义实际的路由规则，并通过 render 属性指定渲染逻辑。roles.includes(role):检查解码出的用户角色是否在允许访问的角色列表 roles 中。如果匹配成功，渲染对应的页面组件 Component，并传递所有路由参数 props。如果匹配失败，使用 <Redirect> 将用户重定向到 /。

./：

【App.js】

该代码实现了一个基于 React 的路由结构。

1. 引入依赖

```
import React from 'react';
import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';
import Login from './pages/Login';
import CustomerPage from './pages/CustomerPage';
import WaiterPage from './pages/WaiterPage';
import PrivateRoute from './components/PrivateRoute';
```

react-router-dom: 提供 React 应用的路由功能：Router 路由器，用于包裹整个应用程序，管理 URL 与组件的对应关系；Route 定义路径与组件的映射规则；Switch:确保一次只渲染一个路由（匹配到的第一个路由）。

页面组件: Login: 登录页面组件。CustomerPage: 客户页面组件。WaiterPage: 服务员页面组件。PrivateRoute:用于实现路由访问控制，只有特定角色的用户才能访问指定的页面。

2. App 组件结构：Router 包裹整个应用，其中定义了路由规则，使用 exact 确保只有路径完全匹配 / 时，才渲染 Login 组件。

```
const App = () => {
  return (
    <Router>
      <Switch>
        <Route path="/" exact component={Login} />
        <PrivateRoute path="/customer_user" component={CustomerPage} roles={['customer_user']}
        <PrivateRoute path="/waiter_user" component={WaiterPage} roles={['waiter_user']} />
      </Switch>
    </Router>
  );
};
```

./pages：

【Login】

Login.js 函数主要负责处理用户登录相关逻辑。在该页面，用户选择角色并（可能）输入密码。提交表单后，与后端交互完成身份验证。根据用户角色，解码令牌并跳转到对应页面。

1. 引入依赖：

   React 和 useState：用于创建 React 组件和定义组件内部的状态。useState 用于管理动态数据，如角色选择和密码。

   Axios：用于发送 HTTP 请求，特别是在这里处理登录的 POST 请求。

   jwtDecode：用于解码从服务器返回的 JSON Web Token (JWT)，以提取用户信息（如角色）。

   useHistory：React Router 提供的钩子，用于在用户登录成功后实现页面跳转。

```
import React, { useState } from 'react';
import axios from 'axios';
import { jwtDecode } from "jwt-decode";
import { useHistory } from 'react-router-dom';
```

2. 定义常量与状态：

   定义 apiUrl，定义应用程序与后端服务器交互的基础 URL 为 http://localhost:5000（即后端服务器的监听地址），构建 API 请求的路径。此处虽然在.env 中定义过 REACT_APP_API_URL，但是为了进一步防止前后端无法连接，在页面中需要再定义一次。

```
5   const apiUrl = process.env.REACT_APP_API_URL || 'http://localhost:5000';
```

   定义组件状态变量：useHistory 用于页面跳转，在登录成功后跳转到对应角色的主页。

```
const Login = () => {
  const [role, setRole] = useState('customer_user');
  const [password, setPassword] = useState('');
  const history = useHistory();
```

3. 提交表单处理函数

   阻止默认行为:e.preventDefault() 防止表单提交时页面刷新。

   发送 POST 请求:使用 axios.post 向后端发送登录请求，提交的数据包括：role: 用户角色、password（仅当角色为 waiter_user 时才提交）。

   解码 JWT:如果登录成功，后端返回一个 JWT。使用 jwtDecode 解码令牌，提取用户信息。

   存储数据:将令牌和用户角色存储到浏览器的本地存储（localStorage），以便在后续请求中使用。

   页面跳转:根据用户角色跳转到不同的页面。

   如果请求失败（例如角色或密码错误），会显示错误消息。

```
const handleSubmit = async (e) => {
  e.preventDefault();
  try {
    const response = await axios.post(`${apiUrl}/login`, { role, ...(role === 'waiter_user' && { password }) });
    const decoded = jwtDecode(response.data.token);
    localStorage.setItem('token', response.data.token);
    localStorage.setItem('role', decoded.role);
    history.push(`/${decoded.role.toLowerCase()}`);
  } catch (error) {
    console.error(error.response ? error.response.data : error.message);
    alert('Login failed. Please check your credentials.');
  }
};
```

4. JSX 渲染逻辑：

   标题和表单：显示标题 "Login"。onSubmit={handleSubmit}: 当表单提交时，调用 handleSubmit 函数

角色选择：提供一个下拉框供用户选择角色，默认为 customer_user。当用户选择角色时，更新 role 状态。
密码输入框（仅服务员可见）：如果用户选择了 waiter_user，显示密码输入框。输入的密码会实时更新 password 状态。
提交按钮：用户点击按钮时触发表单提交。

```
return (
  <div>
    <h1>Login</h1>
    <form onSubmit={handleSubmit}>
      <div>
        <label>Role:</label>
        <select value={role} onChange={(e) => setRole(e.target.value)}>
          <option value="customer_user">Customer</option>
          <option value="waiter_user">Waiter</option>
        </select>
      </div>
      {role === 'waiter_user' && (
        <div>
          <label>Password:</label>
          <input type="password" value={password} onChange={(e) => setPassword(e.target.value)} />
        </div>
      )}
      <button type="submit">Login</button>
    </form>
  </div>
);
};
```

【Customer】
CustomerPage.js 通过 React 状态管理实现了动态页面，能够选择餐桌、添加菜单到订单并提交，同时支持动态展示订单详情。它通过 axios 和后端通信，确保数据从后端获取并实时更新到页面。

1. 导入模块：导入 React 库；导入 axios 库用于发送 HTTP 请求以实现前后端连接。
2. 定义常量与状态：
   定义 apiUrl，定义应用程序与后端服务器交互的基础 URL 为 http://localhost:5000。

```
const apiUrl = process.env.REACT_APP_API_URL || 'http://localhost:5000';
```

定义其它状态变量：

```
const CustomerPage = () => {
  const [tables, setTables] = useState([]); // 餐桌信息
  const [dishes, setDishes] = useState([]); // 菜单信息
  const [selectedTable, setSelectedTable] = useState(null); // 选中的餐桌
  const [order, setOrder] = useState(null); // 当前订单
  const [loading, setLoading] = useState(true); // 加载状态
  const [error, setError] = useState(null); // 错误信息
  const [selectedTableId, setSelectedTableId] = useState(null); // 选中餐桌ID
  const [orderDetails, setOrderDetails] = useState([]); // 当前订单详情
  const [currentOrderId, setCurrentOrderId] = useState(null); // 当前订单ID
```

定义 token 和 config：token 从浏览器的本地存储（localStorage）中读取用户的身份验证令牌（token），用于与后端 API 通信时进行身份验证。定义 config 中配置了请求头 headers，其中包括身份验证字段 Authorization，其作用为在请求时附加 Bearer 令牌，以便后端验证用户身份，如请求需要授权的 API 时（提交订单、获取订单详情等），使用此配置。

```
const token = localStorage.getItem('token');
const config = {
  headers: {
    Authorization: `Bearer ${token}`,
  },
};
```

函数部分

3. 数据获取函数（以 fetchOrderDetails 为例）：
   调用 axios.get 向 API 请求订单详情（路径为 /api/order/:orderId）。请求成功后，将订单详情存储到 orderDetails 状态。请求失败时，记录错误信息到 error。

```
const fetchOrderDetails = async (orderId) => {
  try {
    console.log(`Fetching order details for order ID: ${orderId}`);
    const response = await axios.get(`${apiUrl}/api/order/${orderId}`, config)
    console.log('Order Details:', response.data);
    setOrderDetails(response.data || []);
  } catch (err) {
    console.error("Error fetching order details:", err.response || err);
    setError(`Failed to fetch order details for order ID: ${orderId}`);
  }
};
```

4. 生成函数（generateOrderID）：
   该函数的作用是生成一个基于当前时间戳的唯一订单 ID。使用 Math.floor(Date.now() / 1000) 获取当前时间的时间戳。返回生成的整数作为订单 ID。

```
const generateOrderId = () => {
  return Math.floor(Date.now() / 1000);
};
```

5. 事件处理函数（以 handleSubmitOrder 为例）：
   handleSubmitOrder 函数的作用是提交订单到后端。检查是否选中餐桌，以及订单中是否有菜品。生成唯一订单 ID。调用 axios.post，发送订单数据到后端 /api/order 接口。如果提交成功，保存订单 ID 并调用 fetchOrderDetails 获取新订单的详细信息。提交失败时，将错误信息存储到 error 状态。

```
const handleSubmitOrder = async () => {
  if (!selectedTable || !order.dishes.length) {
    alert('Please select a table and add dishes to your order.');
    return;
  }

  const generatedOrderId = generateOrderId();
  try {
    const response = await axios.post(
      `${apiUrl}/api/order`,
      {
        orderId: generatedOrderId,
        tableId: selectedTable.tableid,
        dishes: order.dishes.map((dish) => ({ dishId: dish.dishId, quantity: dish.quantity })),
      },
      config
    );

    if (response.data) {
      setCurrentOrderId(generatedOrderId); // 保存当前订单ID
      fetchOrderDetails(generatedOrderId); // 提交订单后获取订单详情
    }
  } catch (err) {
    setError('Failed to submit order');
  }
};
```

6. useEffect：
   其作用是在组件挂载时初始化数据。useEffect 在组件首次加载时运行。
   调用 fetchTables 和 fetchDishes 分别获取餐桌和菜单信息。
7. JSX 渲染逻辑：用于构建页面 UI。
   以“选择餐桌”部分为例，这部分会使得餐桌显示状态，如果状态为“已预订”或“有人”，则选择餐桌按钮会变灰且无法点击；若“空”则可以选择且颜色正常；顾客选中桌子时按钮则会变绿，并显示 You have selected Table { tableid}。其它部分同理。

```
{selectedTable && (
  <div style={{ color: 'green', marginBottom: '10px' }}>
    You have selected Table {selectedTable.tableid}.
  </div>
)}

<div>
  <h2>Tables</h2>
  <ul>
    {tables.map((table) => (
      <li
        key={table.tableid}
        onClick={table.status === '空' ? () => handleTableSelect(table) : null}
        style={{
          cursor: table.status === '空' ? 'pointer' : 'not-allowed',
          padding: '8px',
          margin: '4px',
          border: '1px solid #ccc',
          backgroundColor: selectedTableId === table.tableid ? '#d4edda' : 'white',
          fontWeight: selectedTableId === table.tableid ? 'bold' : 'normal',
          opacity: table.status !== '空' ? 0.5 : 1,
        }}
      >
        Table {table.tableid} - {table.status}
      </li>
    ))}
  </ul>
</div>
```

【Waiter】

React 组件用于实现餐厅服务员后台操作界面。通过调用后端 API 来获取与更新餐厅的餐桌、

菜品和订单信息。

1. 导入模块：导入 React 库并解构出 useState 与 useEffect 钩子；

导入 axios 库用于发送 HTTP 请求。

2. 定义常量与状态

定义 API 基础 URL

```
const apiUrl = process.env.REACT_APP_API_URL || 'http://localhost:5000';
```

使用 useState 钩子定义表状态，初始都设为空（null）

```
const WaiterPage = () => {
  const [dishes, setDishes] = useState([]); // 菜品信息
  const [orders, setOrders] = useState([]); // 订单信息
  const [tables, setTables] = useState([]); // 餐桌信息
  const [loading, setLoading] = useState(true); // 加载状态
  const [error, setError] = useState(null); // 错误信息
  const [selectedDish, setSelectedDish] = useState(null); // 选中的菜品
  const [newDish, setNewDish] = useState({ dishname: '', price: '' }); // 新菜品信息
  const [orderDetails, setOrderDetails] = useState([]); // 订单详情
  const [selectedOrder, setSelectedOrder] = useState(null); // 选中的订单
  const [newStatus, setNewStatus] = useState(''); // 新订单状态

  const token = localStorage.getItem('token');
  const config = {
    headers: {
      Authorization: `Bearer ${token}`,
    },
  };
```

3. 生命周期钩子：使用 useEffect 钩子在组件挂载时调用相关 fetch 函数，获取初始数据（通过 fetch 函数获取初始数据：事先在数据库中添加的数据）

```
useEffect(() => {
```

函数部分：以实现“菜品管理”的 Dish 相关函数为例，实现其余功能的函数均在代码标注

4. 函数部分：数据获取函数

从 API 获取餐品数据，并更新 dishes 状态

```
const fetchDishes = async () => {
    const response = await axios.get(`${apiUrl}/api/dishes`);
    setDishes(response.data);
  };
```

5. 函数部分：数据更新函数

【修改餐品信息】更新餐品信息，在更新成功后重新获取餐品数据

```
  const fetchDishes = async () => {
    try {
      const response = await axios.get(`${apiUrl}/api/dishes`, config);
      setDishes(response.data);
    } catch (err) {
      setError('Failed to fetch dishes');
    }
  };
```

【添加餐品信息】添加餐品信息，添加成功后重新获取餐品数据，同时将输入栏清空以避免

出现重复操作。

```
// 处理添加菜品
const handleAddDish = async () => {
  if (!newDish.dishname || !newDish.price) {
    alert('Dish name and price are required.');
    return;
  }

  try {
    await axios.post(`${apiUrl}/api/dish`, newDish, config);
    fetchDishes();
    setNewDish({ dishname: '', price: '' }); // 清空表单
  } catch (err) {
    console.error('Error adding dish:', err);
    setError('Failed to add dish');
  }
};
```

6. 函数部分：事件处理函数

【餐品选择】选择餐品，并更新 selectedDish 状态，更新 newDishInput 状态

解析餐品添加输入框中的内容，更新 newDish 状态

```
// 处理修改菜品
const handleUpdateDish = async () => {
  if (!selectedDish || !selectedDish.dishname || !selectedDish.price) {
    alert('Please select a dish to update.');
    return;
  }
  try {
    await axios.put(`${apiUrl}/api/dish/${selectedDish.dishid}`, selectedDish, conf
    fetchDishes();
    setSelectedDish(null); // 清空选择
  } catch (err) {
    setError('Failed to update dish');
  }
};
```

7. 渲染函数：定义组件渲染逻辑，包括标题、以及各种列表的信息

展示菜品管理相关：

```
{/* 菜品管理 */}
<div>
  <h2>莱欧斯每日新鲜先打迷宫八级大厨森西进行制作亚米亚米</h2>
  <div>
    <h3>森西上新</h3>
    <input
      type="text"
      placeholder="Dish name"
      value={newDish.dishname}
      onChange={(e) => setNewDish({ ...newDish, dishname: e.target.value })}
    />
    <input
      type="number"
      placeholder="Price"
      value={newDish.price}
      onChange={(e) => setNewDish({ ...newDish, price: e.target.value })}
    />
    <button onClick={handleAddDish}>添加！</button>
  </div>
```

```
<div>
  <h3>迷宫饭の修改</h3>
  <select onChange={(e) => setSelectedDish(dishes.find(d => d.dishid === parseInt(e.target.value)))}>
    <option value="">选择</option>
    {dishes.map((dish) => (
      <option key={dish.dishid} value={dish.dishid}>
        {dish.dishname}
      </option>
    ))}
  </select>

  {selectedDish && (
    <div>
      <input
        type="text"
        value={selectedDish.dishname}
        onChange={(e) => setSelectedDish({ ...selectedDish, dishname: e.target.value })}
      />
      <input
        type="number"
        value={selectedDish.price}
        onChange={(e) => setSelectedDish({ ...selectedDish, price: e.target.value })}
      />
      <button onClick={handleUpdateDish}>修改！ </button>
    </div>
  )}
 </div>
</div>
```

该段渲染函数实现以下功能：

- 用户可选择餐品并查看餐品信息
- 允许对选中的餐品信息进行修改
- 允许添加新餐品，将其加入菜单中，同时可对新餐品进行修改

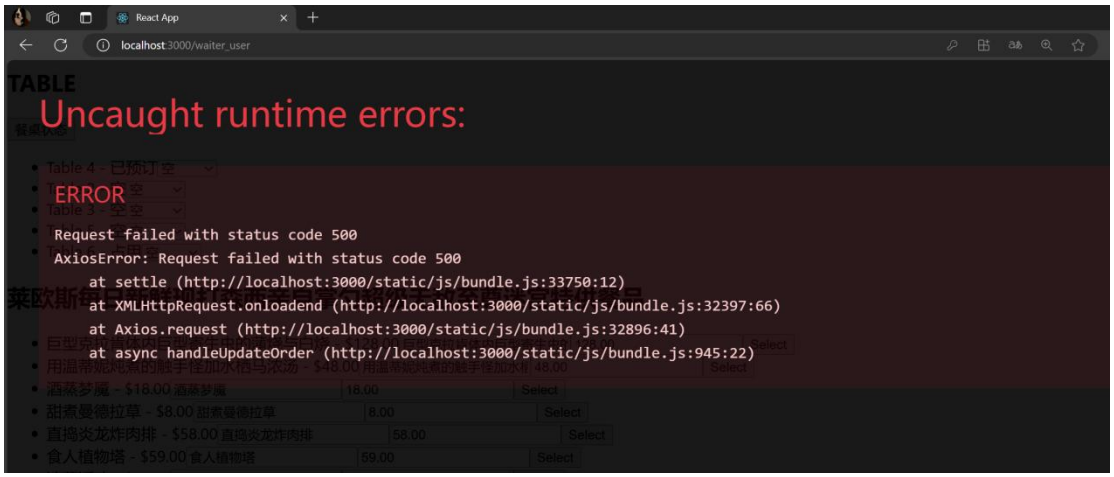
8. 导出组件：对 WaiterPage 组件进行导出

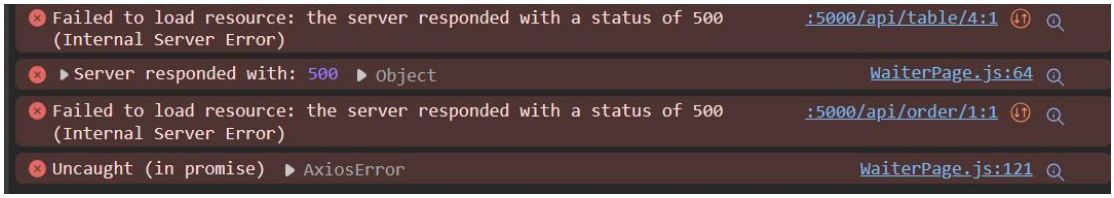
```
export default WaiterPage;
```

五．问题解决

【跨域问题】

在完成系统构建，尝试在 web 页面对系统实现功能进行测试时，出现跨域问题：AxiosError





WEB 是构建在同源策略基础之上的，同源策略会阻止一个域的 js 脚本与另一个域的内容进行交互。在餐厅点餐系统中由于缺少了同源策略，因此前端中某些“行为”无法与后端进行连接，导致 web 页面产生跨域问题。

为解决跨域问题，我们尝试了以下方法：

在服务器端配置 CORS（跨源资源共享）：首先完成 cors 中间件的安装，在后端 express 应用中添加 cors 中间件。

```
//解决跨域问题
const express = require('express');
const cors=require('cors');
const app = express();
app.use(cors());
```

以上只是最基本的代码演示。通过此方法，浏览器在进行跨域请求时会在请求中添加头部发送给服务器，服务器接收请求并进行响应，浏览器接收到服务器的“允许访问”响应后便可进行跨域请求。

了解以上基本解决方法后，我们在 app.js(express.js)中设置服务器的中间件与路由：

```
app.use(bodyParser.json());
// app.use(cors());
app.use(cors({
  origin: 'http://localhost:3000',
  credentials: true,
  methods: ['GET', 'POST','PUT', 'DELETE', 'OPTIONS'],
  allowedHeaders: ['Content-Type', 'Authorization']
}));
```

对位于 express.js 的 cors 中间件进行配置，指定可访问服务器资源、HTTP 方法与请求头。我们在配置 cors 中间件时考虑前端代码中添加的具体需求以及浏览器页面的跨域请求问题。通过以上配置，客户端可以从 http://localhost:3000 源发送跨域请求到服务器，并访问服务器的 API 资源。至此解决跨域问题。

【前后端数据交互问题】

在为前端 WaiterPage 与 CustomerPage 添加函数时后，终端进入后端尝试启动 node 时出现报错：

```
C:\Users\Administrator\Desktop\数据库原理\ros\frontend>npm start
npm error code ENOENT
npm error syscall open
npm error path C:\Users\Administrator\Desktop\数据库原理\ros\frontend\package.json
npm error errno -4058
npm error enoent Could not read package.json: Error: ENOENT: no such file or directory, open 'C:\Users\Administrator\Des
ktop\数据库原理\ros\frontend\package.json'
npm error enoent This is related to npm not being able to find a file.
npm error enoent
npm error A complete log of this run can be found in: D:\nodes.js数据库app安装\node_cache\_logs\2024-12-23T02_10_48_157Z
-debug-0.log
```

分析原因得知，在前端添加函数时未在后端 index.js(连接前后端作用)对新增函数进行定义，导致前后端分离，因此需在后端 index 中对函数进行定义，至此问题解决。

【控制器函数参数与数据库表信息一致性问题】

控制器函数中大部分功能函数对前端功能的实现有很大的影响，在引入 SQL 语句时需要额外注意数据库表的名称以及其包含的属性的完整性与正确性。否则会出现无法查询连接到数据库相关表的情况，导致服务器端错误 500 并返回 fault 错误反馈。

在编写实现“添加餐品”功能代码时，起初并未考虑 dishid 的添加，导致数据库 dish 表无法获得一个完整的、正确的元组，导致操作失败。
为解决该问题，尝试将餐品编号作为一个系统能够自行添加的属性值，以避免报错以及进行添加操作时产生混淆。
首先确定数据库在定义 dish 时并未为 dishid 添加递增特性，故从数据库修改入手：
创建一个承载递增序列的顺序串，并为 dishid 添加添加递增特性。

```
CREATE SEQUENCE dish_dishid_seq
  START WITH 1
  INCREMENT BY 1
  NO MINVALUE
  NO MAXVALUE
  CACHE 1;
```

```
ALTER TABLE dish
ALTER COLUMN dishid SET DEFAULT nextval('dish_dishid_seq');
```

并在 controllers/index 中添加 SQL 语句，实现 dishid 自动递增生成功能。

```
try {
  // 插入新的菜品到数据库，dishid 会自动生成
  const result = await pool.query(
    'INSERT INTO dish (dishname, price) VALUES ($1, $2) RETURNING *',
    [dishname, price]
  );
```

六. 系统测试

1. 代码测试

终端操作启动 node，后端代码成功运行：

```
PS D:\ros> cd backend
PS D:\ros\backend> node app.js
Server is running on port 5000
```

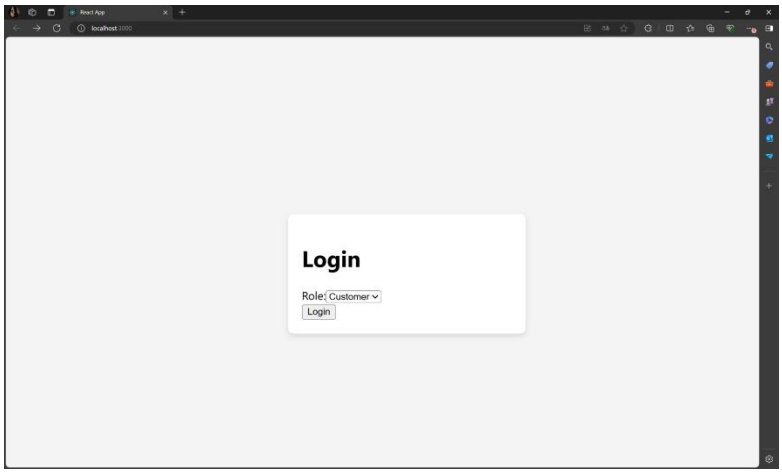
终端操作进入网页，前端代码无逻辑错误，正常运行

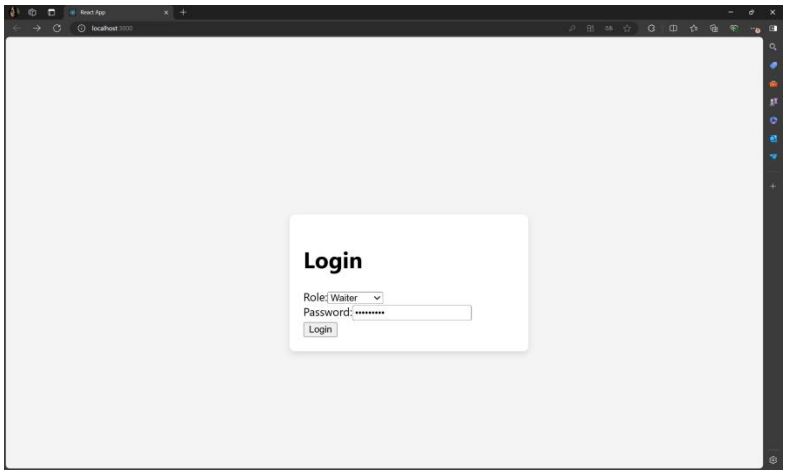
```
PS D:\ros> cd frontend/restaurant-ordering-system
PS D:\ros\frontend\restaurant-ordering-system> npm start

> restaurant-ordering-system@0.1.0 start
> react-scripts start
```

```
Compiled successfully!

You can now view restaurant-ordering-system in the browser.

  Local:            http://localhost:3000
  On Your Network:  http://192.168.56.1:3000

Note that the development build is not optimized.
To create a production build, use npm run build.

webpack compiled successfully
```

2. 功能测试

【用户管理】进入网页可以看见 login 登陆页面，通过点击 role 下标切换用户

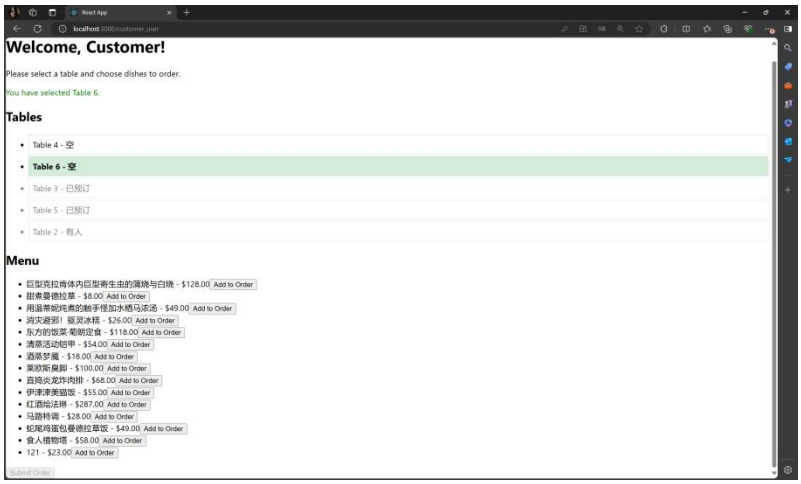




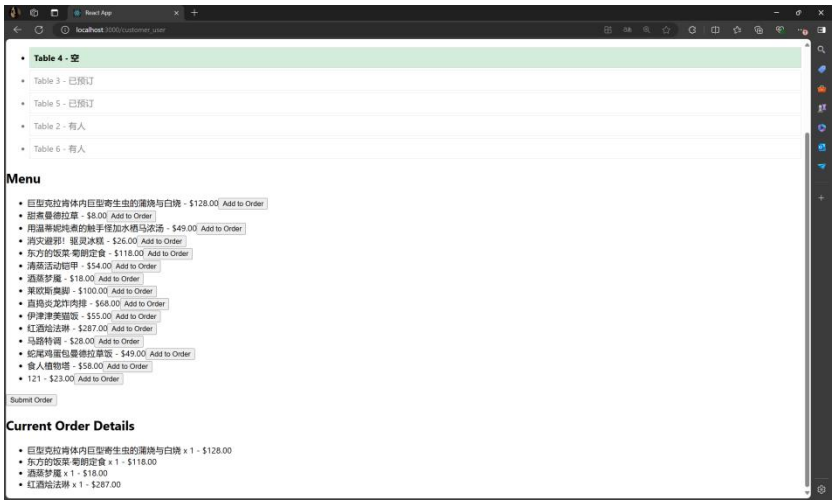
- 顾客（customer）身份可以直接进入点餐页面
- 服务员（waiter）身份在通过密码验证后可进入后台管理页面

【customer】

进入顾客页面，可看到餐桌编号以及餐桌状态信息，顾客只能选择点击状态为“空”的餐桌进行点餐操作。选择餐桌后，顾客通过菜单查看餐品信息。



选择餐品并下单后，顾客可查询当前订单信息，同时选择的餐桌状态变为“有人”



【waiter】

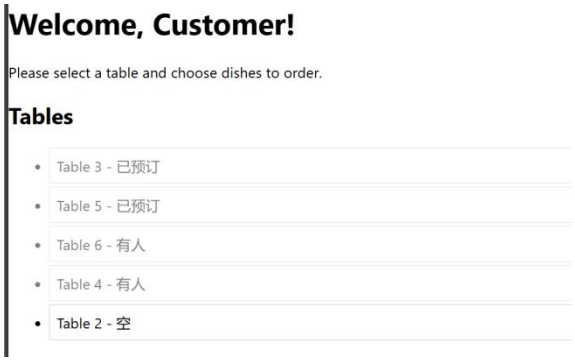
<餐桌状态管理>

后台可查看当前餐桌状态，并对餐桌状态进行“已预订”“有人”“空”之间的修改，修改后餐桌的新状态同步到顾客页面。

例如：将原本状态为“有人”的 2 号餐桌设为“空”，可在顾客页面发现 2 号桌的状态更新

（服务员页面：修改餐桌状态：有人→空）

（顾客页面：对应餐桌状态修改）



<订单管理>

在订单管理栏目，服务员可通过选择期望编号的订单对订单信息进行查询。查询内容包括订单中餐品的名称、价格以及具体数量。

（选择订单编号）

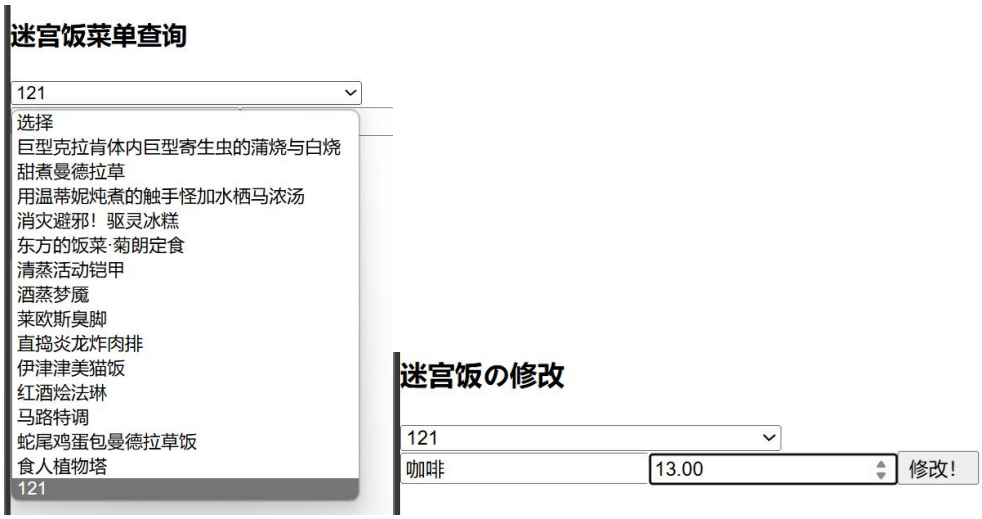
（查看订单详情）

同时可通过修改订单窗口对期望修改的订单进行信息修改

<餐品管理>

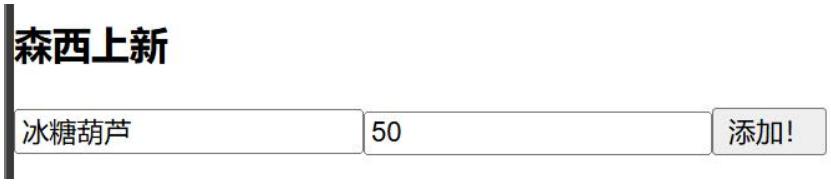
在修改窗口，通过下拉选择可以查询所有餐品的信息，并对期望修改的餐品进行餐品名称、价格的修改：

（餐品查看与选择）



（餐品信息修改：这里将餐品“121-$23”修改为“咖啡-$13”）
修改完成后可以查询到修改后的餐品信息：

餐品管理同时实现添加新餐品的功能，在餐品添加窗口，输入“dish name”与“price”后系统将把新添加的餐品信息添加到菜单中，服务员可在菜单中查看新餐品，并对餐品信息进行修改。（例如，添加餐品“冰糖葫芦-$50”）



添加后添加窗口内容清空，可在菜单中查看“冰糖葫芦”的餐品信息

七．实验总结

餐厅点餐系统通过数据库的合理设计与功能模块的有效实现，为用户提供相对直观、便捷的点餐与后台维护体验，帮助餐厅提高了管理效率。本次实验综合了数据库设计、后端开发与前端展示多方面知识，深化实验者对数据库系统原理理论知识的掌握与实践，并了解数据库系统原理的未来发展方向与研究领域。

优点：在该系统中，用户可以较为直观且便捷地进行操作以满足自己的需求，同时在设计数据库的过程中我们做到了保护数据库（如使用外键约束保证参照完整性等）。

可改进：页面的美化程度较低，进一步优化时可以增添图片信息并调整页面配色以实现更优良的用户体验；功能可以进一步完善，例如，进一步优化时可以使得顾客在点餐时实现客制化需求（如少盐、少糖）。