UNIVERSITY OF SCIENCE – VNU-HCM
FACULTY OF INFORMATION TECHNOLOGY
----------□•□----------

SUBJECT: CSC13106 – SOFTWARE ARCHITECTURE
REPORT:

Lab 01 - Micro-Services

Class: 22KTPM2

Group members:

- Nguyễn Gia Phúc 22127482
- Võ Trung Tín 22127417
- Nguyễn Trọng Phúc 22127335

Professors:

- Ngô Huy Biên
- Ngô Ngọc Đăng Khoa
- Hồ Tuấn Thanh

Ho Chi Minh city, June 17th 2024

Table of Contents

Contribution Table

Member	Student ID	Assigned Tasks
Nguyễn Gia Phúc	22127482	User Service
Võ Trung Tín	22127417	Order Service
Nguyễn Trọng Phúc	22127335	Menu Service

I. OVERVIEW OF MICROSERVICES ARCHITECTURE

What is Microservices?

Microservices is an architectural style that structures an application as a **collection of loosely coupled services**, each responsible for a specific business capability. These services are independently deployable and communicate with each other via APIs.

Key Characteristics

- **Independent services**: Each service owns its codebase and database.
- **Domain-based separation**: Services are split according to business responsibilities.
- **API communication**: Services interact through HTTP/REST or messaging.
- **Scalability**: Each service can scale independently.
- **Autonomy**: Each team can work on a service without affecting others.

When to Use Microservices

- Microservices are ideal when:
  - The system is complex and has multiple domains.
  - Development teams work in parallel.
  - You want independent deployment and scalability.
  - You need better fault isolation and maintainability.
- It may not be suitable for very small systems due to added complexity.

II. ABOUT OUR PRODUCT

Our product simulates a **simple online food ordering platform** with the following features:

- **User Service**: Register and login with email and password.
- **Menu Service**: Show available menu items.
- **Order Service**: Place an order by selecting an item and quantity.

III. WHY WE CHOSE MICROSERVICES OVER MONOLITHIC

Aspect	Monolithic Approach	Microservices Approach
Codebase	Single, unified	Split by domain, multiple repos
Deployment	Whole system redeployed together	Individual services deployed separately

Scalability	Scale entire app	Scale only the required service
Team Collaboration	Harder to work in parallel	Easier to divide responsibilities
Maintainability	Code becomes harder to manage over time	Each service is small, clean, and easier to test
Flexibility	Technology lock-in	Each service can use different tech stack if needed

In our system, we separated the application into three clear domains:

- **User Service**: Handles registration and login.
- **Menu Service**: Manages food items.
- **Order Service**: Handles food orders.

This separation allows for **independent development, testing, and scaling**, which wouldn't be possible in a monolithic structure. We also found **meaningful justifications** to divide it into **three microservices**, each with clear responsibilities:

1. User Service (Authentication & Identity)

- **Reason to separate**:
  - Authentication and user account handling is a **universal concern** that can be reused across different systems (e.g., admin portal, mobile app).
  - Requires its own security logic (e.g., password hashing, JWT tokens).
  - If developed in monolith, user logic becomes tightly coupled with business logic like ordering and menu display.
- **Microservices Benefit**:
  - Future-ready for **Single Sign-On (SSO)** or **OAuth**.
  - Authentication logic can evolve without affecting business services.
  - Easy to replace with an external identity provider in the future.

2. Menu Service (Menu & Product Catalog)

- **Reason to separate**:
  - Menu management (CRUD of food items) is mostly **read-heavy** and rarely changes once deployed.

- This service can be used by both customers (for viewing) and admins (for managing the menu).
- Does not depend on user or order logic to function.

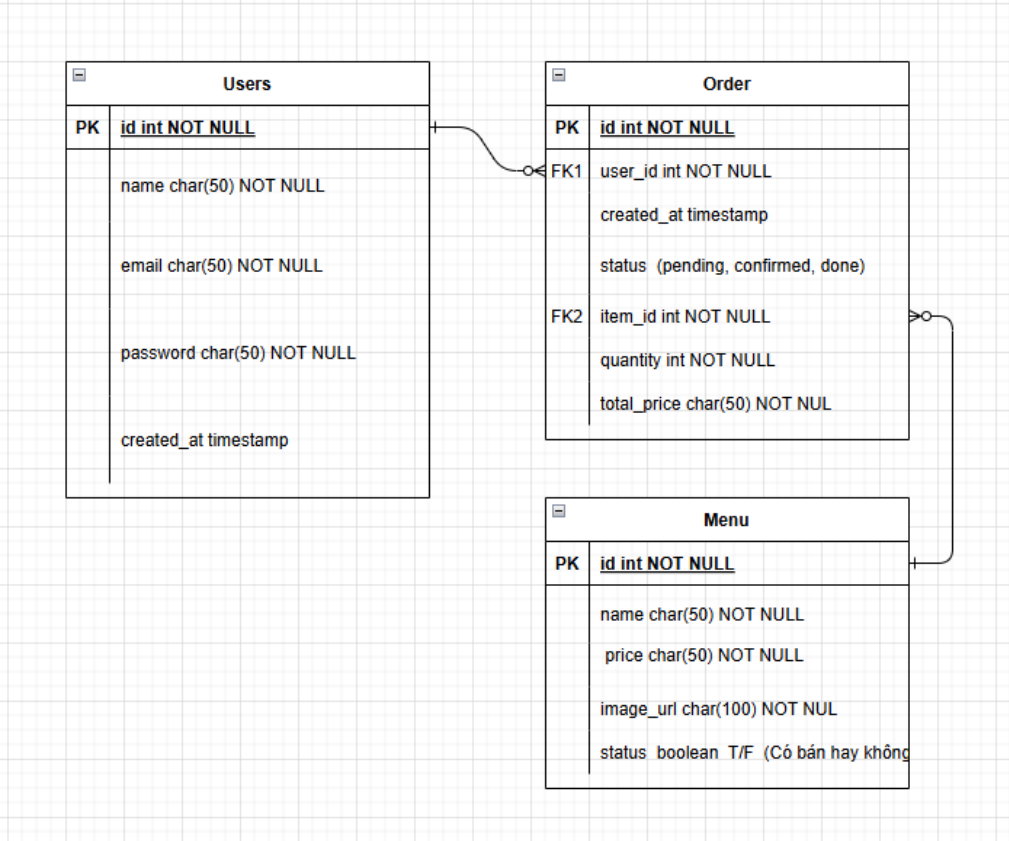
- **Microservices Benefit**:
  - Can be **cached** or scaled separately for performance (read-heavy optimization).
  - Easily extended with a **recommendation engine** in the future without affecting orders or users.
  - If menu service crashes or is being updated, order service can still operate with cached data.

3. Order Service (Order Processing)

- **Reason to separate**:
  - Order logic involves different concerns: inventory, pricing, transaction status, etc.
  - Needs to interact with **user service** (who placed the order) and **menu service** (what items were ordered) — ideal case for service communication.
  - High potential for scaling under real-world traffic (peak hours, flash sales).
- **Microservices Benefit**:
  - Enables **isolated testing of order rules** (discounts, availability, validation).
  - Future integration with **payment gateway**, **delivery tracking**, or **invoice generation**.
  - Order service can be scaled independently under high load.

IV. DATABASE DESIGN

We used **3 tables**, one for each service to maintain **data ownership and independence**:



Users

- **Purpose**: Stores information about registered users of the system.
- **Primary Key (PK)**: id (Integer, NOT NULL)
    - Uniquely identifies each user.
- **Attributes**:
    - name (Character(50), NOT NULL)
        - The user's full name.
    - email (Character(50), NOT NULL)
        - The user's email address, likely used for login or communication.
    - password (Character(50), NOT NULL)
        - The user's encrypted password.
    - created_at (Timestamp)
        - The date and time when the user account was created.

Order

- **Purpose**: Stores details about customer orders.
- **Primary Key (PK)**: id (Integer, NOT NULL)
    - Uniquely identifies each order.

- **Foreign Keys (FK)**:
  - user_id (Integer, NOT NULL)
    - References the id column in the Users table, linking an order to the user who placed it.
  - item_id (Integer, NOT NULL)
    - References the id column in the Menu table, linking an order to a specific menu item.
- **Attributes**:
  - created_at (Timestamp)
    - The date and time when the order was created.
  - status (Enum: 'pending', 'confirmed', 'done')
    - The current status of the order, indicating its progress (pending, confirmed, or completed).
  - quantity (Integer, NOT NULL)
    - The number of items ordered.
  - total_price (Character(50), NOT NULL)
    - The total cost of the order, stored as a string (possibly for formatting reasons, e.g., "$12.99").

Menu

- **Purpose**: Stores the available menu items that can be ordered.
- **Primary Key (PK)**: id (Integer, NOT NULL)
  - Uniquely identifies each menu item.
- **Attributes**:
  - name (Character(50), NOT NULL)
    - The name of the menu item (e.g., "Spaghetti Carbonara").
  - price (Character(50), NOT NULL)
    - The price of the menu item, stored as a string (possibly for formatting, e.g., "$9.99").
  - image_url (Character(100), NOT NULL)
    - The URL of an image representing the menu item.
  - status (Boolean, Default: True)
    - Indicates whether the menu item is available (True) or unavailable (False). The diagram notes "TF (Có bán hay không)" which translates to "TF (Available or not)" in Vietnamese, confirming this is a boolean field.

Relationships

Users to Order:

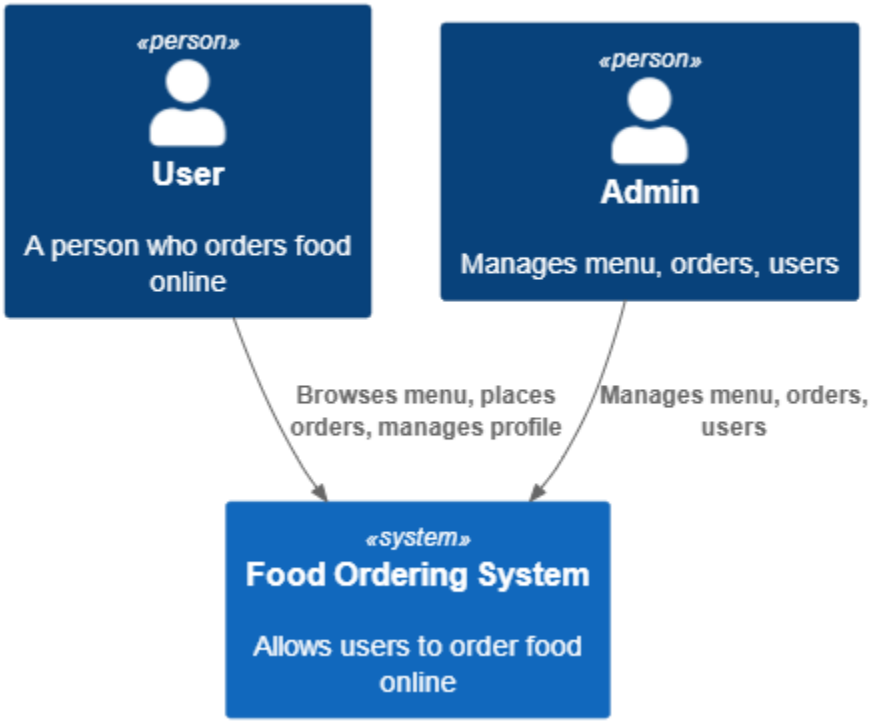
- A one-to-many relationship exists between Users and Order.
- The user_id foreign key in the Order table links to the id primary key in the Users table.
- This means one user can place multiple orders, but each order is associated with exactly one user.

Menu to Order:

- A one-to-many relationship exists between Menu and Order.
- The item_id foreign key in the Order table links to the id primary key in the Menu table.
- This means one menu item can be included in multiple orders, but each order line item references a single menu item.

V. ARCHITECTURE

a. System Context Diagram

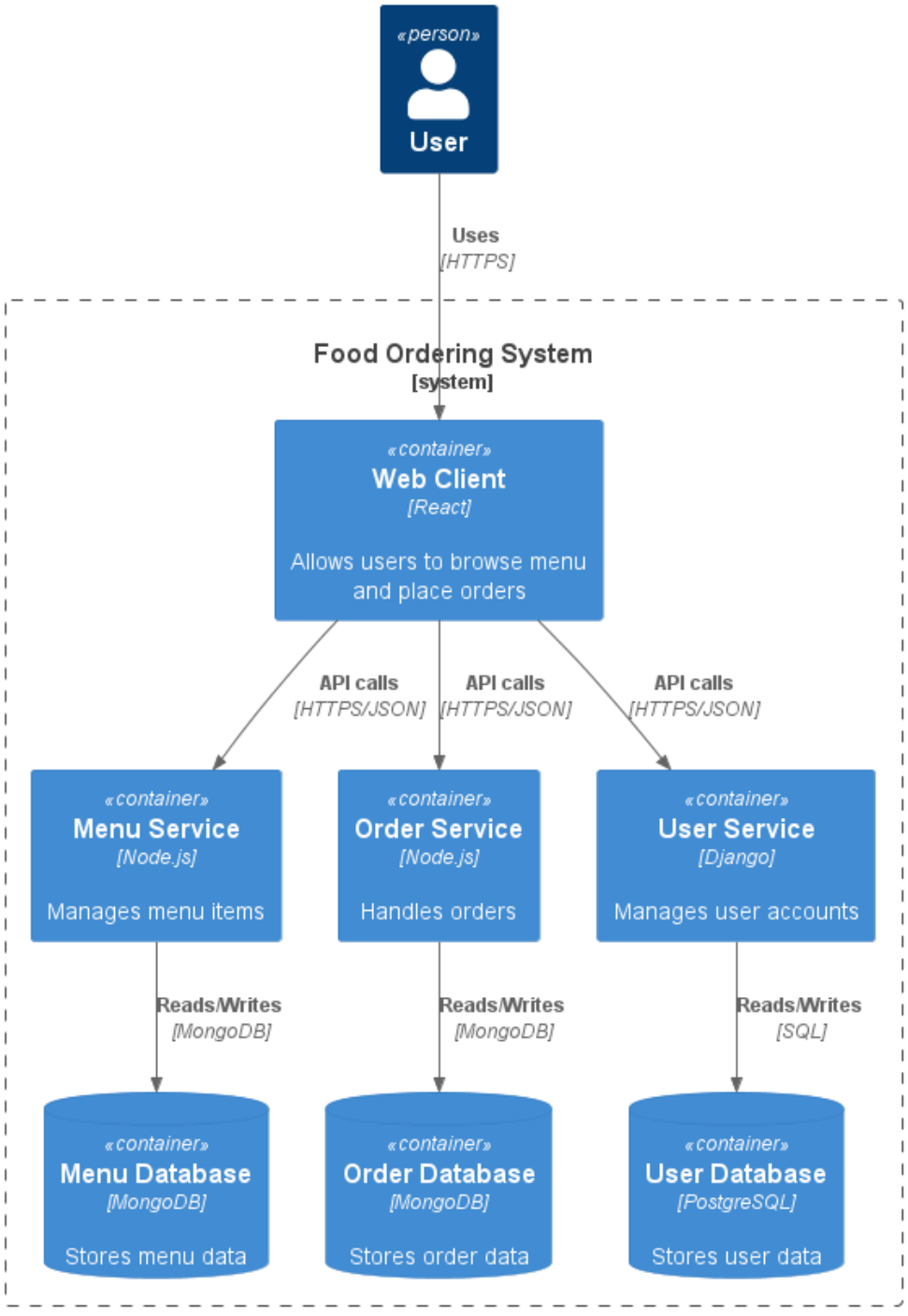


The system is designed to facilitate online food ordering and is centered around two primary user roles: **User** and **Admin**, both of which interact with the system to perform distinct functions.

- **User**: Represented as a person who orders food online, the User engages with the Food Ordering System by browsing the menu, placing orders, and managing their profile. This role is intended for customers who use the platform to explore available food items, make purchases, and maintain their account details.
- **Admin**: Also depicted as a person, the Admin has elevated privileges and is responsible for managing the system's core components. This includes overseeing the menu (e.g., adding, updating, or removing items), managing orders (e.g., tracking and updating order statuses), and handling user accounts (e.g., adding or modifying user permissions). The Admin ensures the system operates smoothly and meets the needs of both the business and its customers.
- **Food Ordering System**: Positioned at the center, this system serves as the platform that enables the described interactions. It allows Users to order food online by providing a menu interface and order processing capabilities, while also supporting Admin functions to maintain and manage the system's data and operations. The system acts as the intermediary, connecting the User and Admin roles to deliver a seamless online food ordering experience.

The relationships are visually represented with arrows indicating the flow of actions: Users browse the menu, place orders, and manage their profiles, while Admins have broader control, managing menu items, orders, and users. This structure highlights the system's dual-purpose design, catering to customer convenience and administrative oversight.

b. Container Diagram



User

- **Role**: The primary actor, represented as a person interacting with the system.

- **Interaction**: Uses the system via HTTPS requests, typically through a web browser or mobile application.
- **Purpose**: Engages with the Food Ordering System to perform actions such as browsing the menu and placing orders.

Food Ordering System (System Boundary)

- **Overview**: Enclosed within a blue boundary labeled "Food Ordering System (system)", this represents the scope of the software system, containing all relevant containers.
- **Containers**: The system comprises four main containers, each deployed as a separate, independently executable unit (e.g., Docker containers).

Containers and Their Responsibilities

- **Web Client (React)**:
  - **Responsibility**: Serves as the front-end user interface, allowing users to browse the menu and place orders.
  - **Technology**: Built with React, a JavaScript library for building single-page applications (SPAs), ensuring a dynamic and responsive user experience.
  - **Interaction**: Communicates with backend services via API calls using HTTP/JSON.
  - **Deployment**: Deployed as a container, indicating it can be scaled and managed independently.
- **Menu Service (Node.js)**:
  - **Responsibility**: Manages menu items, including creation, retrieval, updates, and deletions.
  - **Technology**: Implemented using Node.js, leveraging its event-driven, non-blocking I/O model for efficient handling of menu-related requests.
  - **Interaction**: Receives API calls from the Web Client and interacts with the MenuDatabase.
  - **Deployment**: Containerized, allowing isolation and scalability.
- **Order Service (Node.js)**:
  - **Responsibility**: Handles order processing, including creation, tracking, and status updates.
  - **Technology**: Also implemented in Node.js, ensuring consistency with the Menu Service for a unified development approach.
  - **Interaction**: Receives API calls from the Web Client and interacts with the OrderDatabase.
  - **Deployment**: Containerized for independent deployment and management.
- **User Service (Django)**:

- **Responsibility**: Manages user accounts, including registration, authentication, profile management, and authorization.
- **Technology**: Built with Django, a high-level Python web framework, suitable for complex user management logic.
- **Interaction**: Receives API calls from the Web Client and interacts with the UserDatabase.
- **Deployment**: Containerized, enabling separate scaling and maintenance.

Databases (Supporting Containers)

- **Menu Database (MongoDB)**:
  - **Responsibility**: Stores menu-related data, such as item names, prices, images, and availability status.
  - **Technology**: MongoDB, a NoSQL database, chosen for its flexibility in handling semi-structured menu data.
  - **Interaction**: Read from and written to by the Menu Service.
  - **Deployment**: Containerized, ensuring portability and consistency with the service.
- **Order Database (MongoDB)**:
  - **Responsibility**: Stores order-related data, including order IDs, user IDs, items, quantities, and statuses.
  - **Technology**: MongoDB, consistent with the Menu Database for a unified NoSQL approach in this domain.
  - **Interaction**: Read from and written to by the Order Service.
  - **Deployment**: Containerized for independent management.
- **User Database (PostgreSQL)**:
  - **Responsibility**: Stores user-related data, such as names, emails, passwords, and creation timestamps.
  - **Technology**: PostgreSQL, a relational database, selected for its support of complex queries and transactions, ideal for user account management.
  - **Interaction**: Read from and written to by the User Service.
  - **Deployment**: Containerized, aligning with the microservices architecture.

Relationships and Interactions

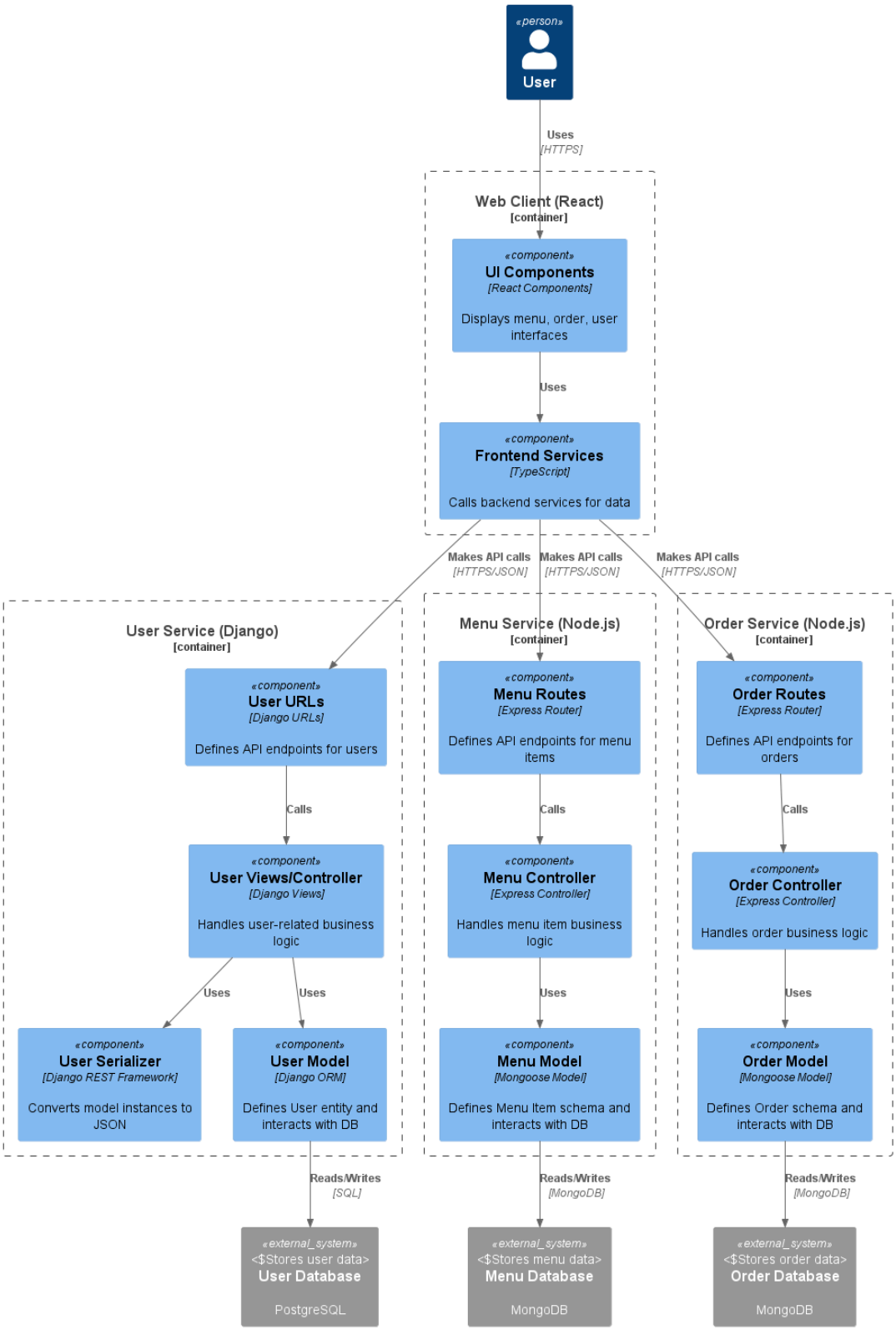
- **User to Web Client**: The User accesses the Web Client via HTTPS, initiating actions like browsing or ordering.
- **Web Client to Services**: The Web Client makes API calls (HTTP/JSON) to the Menu Service, Order Service, and User Service to fetch data or perform operations.
- **Services to Databases**: Each service interacts with its respective database:
  - Menu Service reads/writes to Menu Database.

    - Order Service reads/writes to Order Database.
    - User Service reads/writes to User Database.
- **Communication**: All interactions between the Web Client and services use HTTP/JSON, indicating a RESTful API architecture.

Containerization and Deployment

- All components (Web Client, Menu Service, Order Service, User Service, and databases) are depicted as containerized entities (e.g., Docker containers), denoted by the <container> notation. This suggests the use of a container orchestration system (e.g., Docker Compose or Kubernetes) for deployment, enabling scalability, isolation, and portability.

c. Component Diagram



Containers and Their Components

- **Web Client (React) [container]**
  - **Responsibility**: Serves as the front-end user interface.
  - **Component**:
    - **UI Components**:
      - **Function**: Displays menu, order, and user interfaces.
      - **Technology**: Built with React, ensuring a dynamic and responsive user experience.
      - **Interaction**: Renders the user interface and sends user inputs to the backend.
  - **Interaction**: The Web Client makes API calls (HTTP/JSON) to the backend services.
- **Front-end Services [container]**
  - **Responsibility**: Acts as an intermediary layer that calls backend services for data.
  - **Component**:
    - **Front-end Services**:
      - **Function**: Handles API requests to backend services and processes responses.
      - **Technology**: Likely implemented in JavaScript/TypeScript, coordinating with the Web Client.
      - **Interaction**: Makes API calls to the Menu Service, Order Service, and User Service.
- **User Service (Django) [container]**
  - **Responsibility**: Manages user-related functionalities, such as authentication and profile management.
  - **Components**:
    - **User URLs**:
      - **Function**: Defines API endpoints for user-related operations (e.g., login, registration).
      - **Technology**: Part of the Django framework.
    - **User Views/Controller**:
      - **Function**: Handles user-related business logic, such as processing requests and responses.
      - **Technology**: Django View/Controller, leveraging Django's request-handling capabilities.
    - **User Model**:
      - **Function**: Defines the entity and schema for user data, interacting with the database.
      - **Technology**: Django ORM (Object-Relational Mapping).

    - **Interaction**: The User Views/Controller uses the User Model to read/write data to the UserDatabase (PostgreSQL) via SQL queries.
    - **External System**: The UserDatabase (PostgreSQL) stores user data and is accessed through the Django ORM.
- **Menu Service (Node.js) [container]**
    - **Responsibility**: Manages menu-related functionalities, such as creating and retrieving menu items.
    - **Components**:
        - **Menu Routes**:
            - **Function**: Defines API endpoints for menu-related operations (e.g., get menu, add item).
            - **Technology**: Express.js routes.
        - **Menu Controller**:
            - **Function**: Handles menu-related business logic, processing requests and responses.
            - **Technology**: Express.js Controller.
        - **Menu Model**:
            - **Function**: Defines the menu schema and interacts with the database.
            - **Technology**: Mongoose, a MongoDB ORM for Node.js.
    - **Interaction**: The Menu Controller uses the Menu Model to read/write data to the MenuDatabase (MongoDB) via MongoDB queries.
    - **External System**: The MenuDatabase (MongoDB) stores menu data and is accessed through Mongoose.
- **Order Service (Node.js) [container]**
    - **Responsibility**: Manages order-related functionalities, such as processing and tracking orders.
    - **Components**:
        - **Order Routes**:
            - **Function**: Defines API endpoints for order-related operations (e.g., place order, get order status).
            - **Technology**: Express.js routes.
        - **Order Controller**:
            - **Function**: Handles order-related business logic, processing requests and responses.
            - **Technology**: Express.js Controller.
        - **Order Model**:
            - **Function**: Defines the order schema and interacts with the database.
            - **Technology**: Mongoose, a MongoDB ORM for Node.js.

  - **Interaction**: The Order Controller uses the Order Model to read/write data to the OrderDatabase (MongoDB) via MongoDB queries.
  - **External System**: The OrderDatabase (MongoDB) stores order data and is accessed through Mongoose.

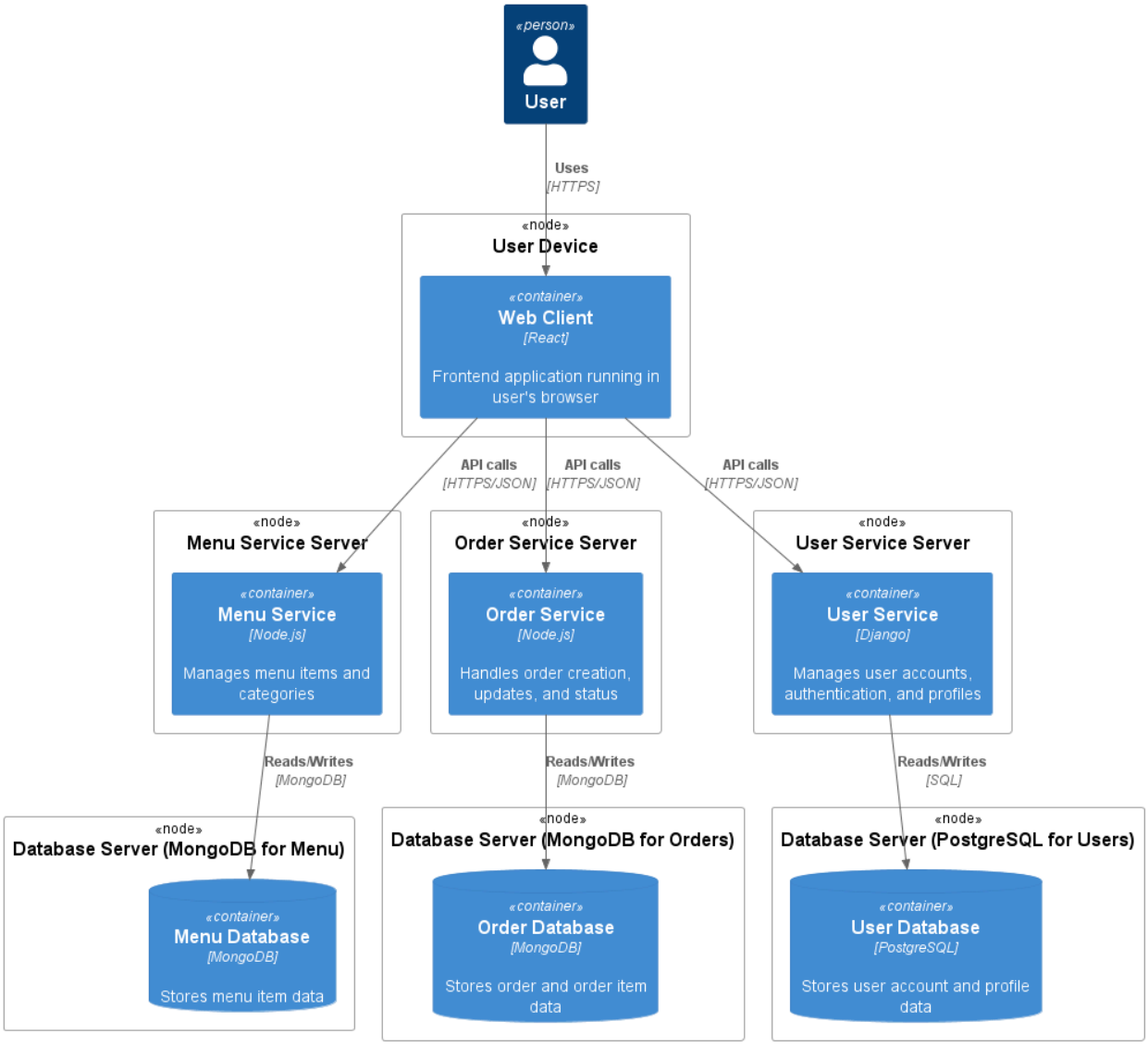
External Systems (Databases)

- **User Database (PostgreSQL)**:
  - **Responsibility**: Stores user-related data (e.g., names, emails, passwords).
  - **Technology**: PostgreSQL, a relational database supporting complex queries and transactions.
  - **Interaction**: Accessed by the User Service via the Django ORM.
- **Menu Database (MongoDB)**:
  - **Responsibility**: Stores menu-related data (e.g., item names, prices, images).
  - **Technology**: MongoDB, a NoSQL database for flexible, document-based storage.
  - **Interaction**: Accessed by the Menu Service via Mongoose.
- **Order Database (MongoDB)**:
  - **Responsibility**: Stores order-related data (e.g., order IDs, user IDs, items).
  - **Technology**: MongoDB, consistent with the Menu Database for a unified NoSQL approach.
  - **Interaction**: Accessed by the Order Service via Mongoose.

Relationships and Interactions

- **User to Web Client**: The User interacts with the Web Client via HTTPS, initiating actions like browsing or ordering.
- **Web Client to Front-end Services**: The UI Components send user inputs to the Front-end Services, which coordinate API calls.
- **Front-end Services to Backend Services**: The Front-end Services make API calls (HTTP/JSON) to the User Service, Menu Service, and Order Service.
- **Backend Services to Databases**: Each service's Controller uses its respective Model to interact with the corresponding database:
  - User Service → UserDatabase (PostgreSQL).
  - Menu Service → MenuDatabase (MongoDB).
  - Order Service → OrderDatabase (MongoDB).

d. Deployment Diagram



User Device

- **Node**: Represented as a <node> labeled "User Device," hosting the Web Client.
- **Container**:
  - **Web Client**:
    - **Function**: A frontend application running in the user's browser, built with React.
    - **Responsibility**: Provides the user interface for browsing menus, placing orders, and managing profiles.
    - **Interaction**: Makes API calls (HTTP/JSON) to the backend servers (Menu Service Server, Order Service Server, and User Service Server).
- **Deployment**: Runs on the user's device (e.g., a personal computer, tablet, or smartphone), indicating a client-side application.

Backend Servers

The system is deployed across three separate server nodes, each hosting a specific service and interacting with its respective database server.

- **Menu Service Server**
  - **Node**: Represented as a <node> labeled "Menu Service Server," running Node.js.
  - **Container**:
    - **Menu Service (Node.js)**:
      - **Function**: Manages menu items and categories.
      - **Responsibility**: Handles CRUD operations for menu data.
      - **Interaction**: Makes API calls to the Web Client and reads/writes to the MongoDB for Menu database server.
  - **Deployment**: Hosted on a dedicated server or virtual machine, containerized for scalability.
- **Order Service Server**
  - **Node**: Represented as a <node> labeled "Order Service Server," running Node.js.
  - **Container**:
    - **Order Service (Node.js)**:
      - **Function**: Handles order creation, updates, and status management.
      - **Responsibility**: Processes order-related requests and updates.
      - **Interaction**: Makes API calls to the Web Client and reads/writes to the MongoDB for Orders database server.

    - **Deployment**: Hosted on a dedicated server or virtual machine, containerized for independent management.
- **User Service Server**
    - **Node**: Represented as a <node> labeled "User Service Server," running Django.
    - **Container**:
        - **User Service (Django)**:
            - **Function**: Manages user accounts, authentication, and profiles.
            - **Responsibility**: Handles user registration, login, and profile updates.
            - **Interaction**: Makes API calls to the Web Client and reads/writes to the PostgreSQL for Users database server.
    - **Deployment**: Hosted on a dedicated server or virtual machine, containerized for isolation.

Database Servers

The system utilizes three external database servers, each supporting a specific service and deployed as separate nodes.

- **Database Server (MongoDB for Menu)**
    - **Node**: Represented as a <node> labeled "Database Server (MongoDB for Menu)."
    - **Container**:
        - **Menu Database (MongoDB)**:
            - **Function**: Stores menu item data, such as names, prices, and categories.
            - **Responsibility**: Provides persistent storage for menu-related information.
            - **Interaction**: Read from and written to by the Menu Service Server.
    - **Deployment**: Hosted on a dedicated server or virtual machine, containerized for scalability.
- **Database Server (MongoDB for Orders)**
    - **Node**: Represented as a <node> labeled "Database Server (MongoDB for Orders)."
    - **Container**:
        - **Order Database (MongoDB)**:
            - **Function**: Stores order and order item data, such as order IDs, statuses, and quantities.

        - **Responsibility**: Provides persistent storage for order-related information.
        - **Interaction**: Read from and written to by the Order Service Server.
    - **Deployment**: Hosted on a dedicated server or virtual machine, containerized for independent management.
- **Database Server (PostgreSQL for Users)**
    - **Node**: Represented as a <node> labeled "Database Server (PostgreSQL for Users)."
    - **Container**:
        - **User Database (PostgreSQL)**:
            - **Function**: Stores user account and profile data, such as names, emails, and passwords.
            - **Responsibility**: Provides persistent storage for user-related information.
            - **Interaction**: Read from and written to by the User Service Server.
    - **Deployment**: Hosted on a dedicated server or virtual machine, containerized for robust data management.

Relationships and Interactions

- **User to User Device**: The User interacts with the Web Client on their device via HTTPS.
- **User Device to Backend Servers**: The Web Client makes API calls (HTTP/JSON) to the Menu Service Server, Order Service Server, and User Service Server.
- **Backend Servers to Database Servers**: Each service server reads from and writes to its respective database server:
    - Menu Service Server → MongoDB for Menu.
    - Order Service Server → MongoDB for Orders.
    - User Service Server → PostgreSQL for Users.
- **Communication**: All interactions between the User Device and backend servers use HTTP/JSON, indicating a RESTful API architecture.

VI. HOW TO RUN

Step 1: Set Up the Python user-service (Django + PostgreSQL)

1. Navigate to the directory

```
cd backend/user-service
```

2. Create a virtual environment

```
virtualenv venv
```

=> This creates a `venv/` folder for the virtual environment.

3. Activate the environment

- On **Windows**:

```
venv\Scripts\activate
```

- On **macOS/Linux**:

```
source venv/bin/activate
```

You should see (venv) at the beginning of your terminal prompt.

4. Install dependencies

```
pip install -r requirements.txt
```

5. Configure environment variables

Create a `.env` file or set environment variables manually. Example:

```
DB_NAME=postgres
DB_USER=postgres
DB_PASSWORD=your_password
DB_HOST=localhost
DB_PORT=5432
```

6. Apply database migrations and run the server

```
python manage.py migrate
python manage.py runserver 8000
```

Step 2: Set Up the Node.js menu-service

```
cd backend/menu-service
npm install
```

Run the server:

```
npm run dev
```

Step 3: Set Up the Node.js order-service

```
cd backend/order-service
npm install
```

Run the server:

```
npm run dev
```

Step 4: Set Up the Frontend restaurant (React + Vite + Tailwind)

```
cd frontend/restaurant
npm install
```

Run the development server:

```
npm run dev
```

Note: Deactivate Python Virtual Environment (When Done)

```
deactivate
```