HUrricane

**Architecture Notebook**

# **Purpose**

[Always address Sections 2 through 6 of this template. Other sections are recommended, depending on the amount of novel architecture, the amount of expected maintenance, the skills of the development team, and the importance of other architectural concerns.]

This document describes the philosophy, decisions, constraints, justifications, significant elements, and any other overarching aspects of the system that shape the design and implementation. This document focuses on the core sections of the basic architecture of the software system.

# **Architectural goals and philosophy**

[Describe the philosophy of the architecture. Identify issues that will drive the philosophy, such as: Will the system be driven by complex deployment concerns, adapting to legacy systems, or performance issues? Does it need to be robust for long-term maintenance?

Formulate a set of goals that the architecture needs to meet in its structure and behavior. Identify critical issues that must be addressed by the architecture, such as: Are there hardware dependencies that should be isolated from the rest of the system? Does the system need to function efficiently under unusual conditions?]

**Architectural Philosophy**

The architecture is designed to ensure (scalability, maintainability, performance, etc)] while addressing the specific needs of the system. The guiding philosophy centers around:

**Modularity,** The system is designed with modularity in mind, as it allows partial operations to be isolated, making error handling safer and implementation easier for developers.

**Adaptability,** The interfaces should be adaptable to different platforms also, the database architecture should be replicable to other database management systems.

**Deployment Simplicit,** Deployment should be easy to understand for other developers and also, have a simple interface for end-users.

**Compatibility,** Software architecture is prepared to be suitable with the deployment itself. In a high percentage, implementation should sort together with the architecture.

**Resilience**, for handling failures and minimizing system downtime with backup architecture.

**Security**: Given the integration with banking services, security is a priority, especially in handling sensitive user data and financial transactions.

**Possible Driven Issues**

**Bank Integration:** The architecture must address the challenges of securely integrating with external banking APIs, including handling authentication, transaction validation, and potential service downtimes. **Frontend-Backend Communication:**  Ensuring smooth communication between the frontend and backend is critical. The architecture must detect and handle potential API latency, partial failures, and maintain consistent state between the UI and the backend.  
**Database Management:** High-frequency transactions, such as simultaneous food orders and payments, raise issues of data consistency and concurrency. The architecture must ensure atomicity and avoid race conditions or data loss.

**Scalability under Traffic Peaks**: The system must cope with high loads during peak hours (e.g., lunch and dinner), requiring a scalable backend and efficient database management.

**Performance Problems**: Legacy code and inefficient implementations with poor algorithmic performance may lead to failures and latency issues. These can result in critical problems such as data leaks and website crashes.

**Architectural Goals**

The goals have a clearer scope in light of the architectural philosophy. Our software architecture goals are defined based on core software principles and with precautions for potential risks we have identified.

**Reliability**: Ensure the system is fault-tolerant, with proper error handling and failover mechanisms to prevent downtime.

**Scalability**: Ensure that the backend and database can scale independently, especially during peak usage times.  
**Security**: Implement strong encryption, access controls, and security protocols for handling sensitive financial data, following industry standards (e.g., PCI-DSS for payment processing).

**Maintainability**: The system should be easy to update and extend, with clear modularization that supports adding new features (like promotional offers, user reviews, etc.).

**Adaptability:** The system should have a portable foundation to accommodate potential platform changes or future transformation into a mobile application.

**Usability & Deployment Clarity**: Ensure that deployment processes are developer-friendly and that the user interface is simple and intuitive.

**System Compatibility**: Maintain consistency between architecture and implementation, ensuring seamless integration across modules and environments.

# **Assumptions and dependencies**

[List the assumptions and dependencies that drive architectural decisions. This could include sensitive or critical areas, dependencies on legacy interfaces, the skill and experience of the team, the availability of important resources, and so forth]

### **Assumptions**

**Bank API is stable and secure**: We assume the bank provides a well-documented, reliable payment API that supports real-time transactions.

**Intermediate developer team**: The architecture assumes a development team with moderate experience in full-stack web development (e.g., Java, React, Spring Boot, SQL).

**Limited simultaneous users**: We assume a medium load of users; architecture is not initially designed for high-concurrency environments (e.g., thousands of users per second).

**Project will not integrate legacy systems**: We assume no old or legacy software needs to be connected to this system.

**Deployment target is a cloud-based environment**: We assume the system will be deployed using tools like Docker or containerized services on cloud platforms like AWS, or Azure.

**User-Friendly Interface**: We assume that the UI design will provide an intuitive, accessible, and user-centric interface that enables users to easily understand.

**Order Management Support for Restaurants**: It is assumed that the platform will provide restaurants with a functional management panel that enables real-time viewing and management of customer orders, supporting operational efficiency aligned with their business objectives.

**System accessed through web browsers only**: We assume all users interact through a desktop or mobile browser, no mobile app is in scope..

### **Dependencies**

**Database (e.g., PostgreSQL/MySQL)**: Required for storing and retrieving data like orders, user profiles, and payment records.

**Frontend (React or similar)**: Interface users interact with — depends on proper backend APIs being available.

**Backend Server:** It is assumed that backend server would function properly.

**Authentication library**: Depends on tools like JWT or OAuth for secure user login and session handling.

**Network infrastructure**: Relies on stable internet and server-to-server communication (backend ↔ database ↔ bank API).

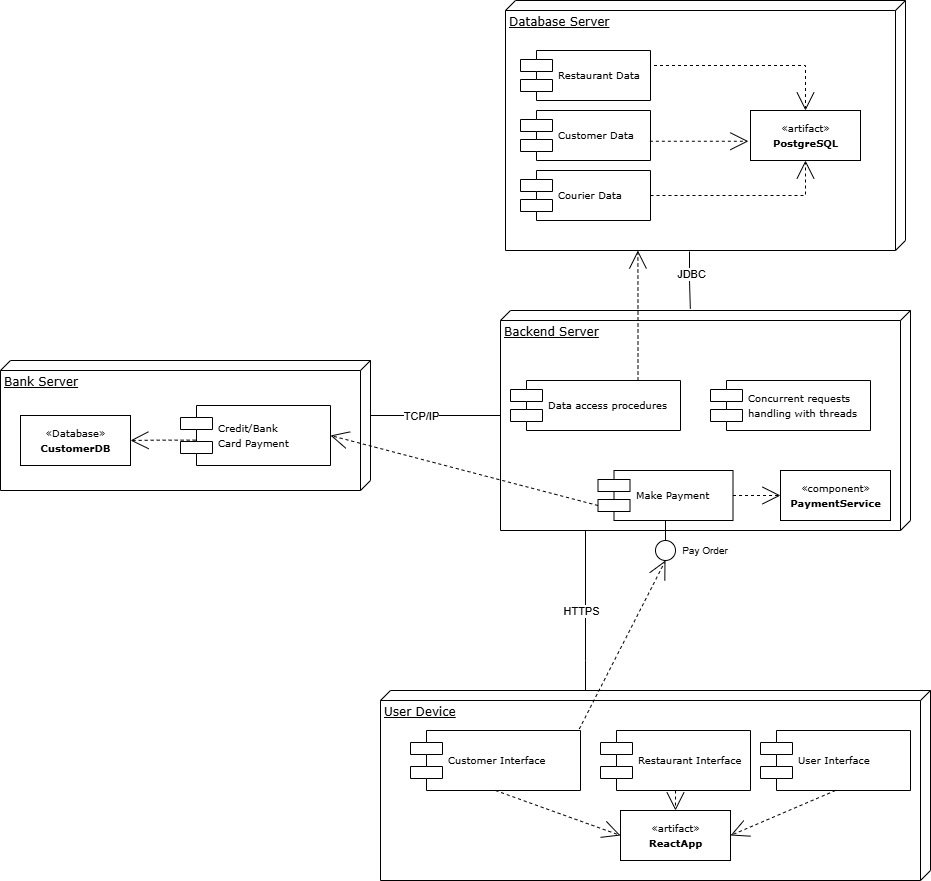
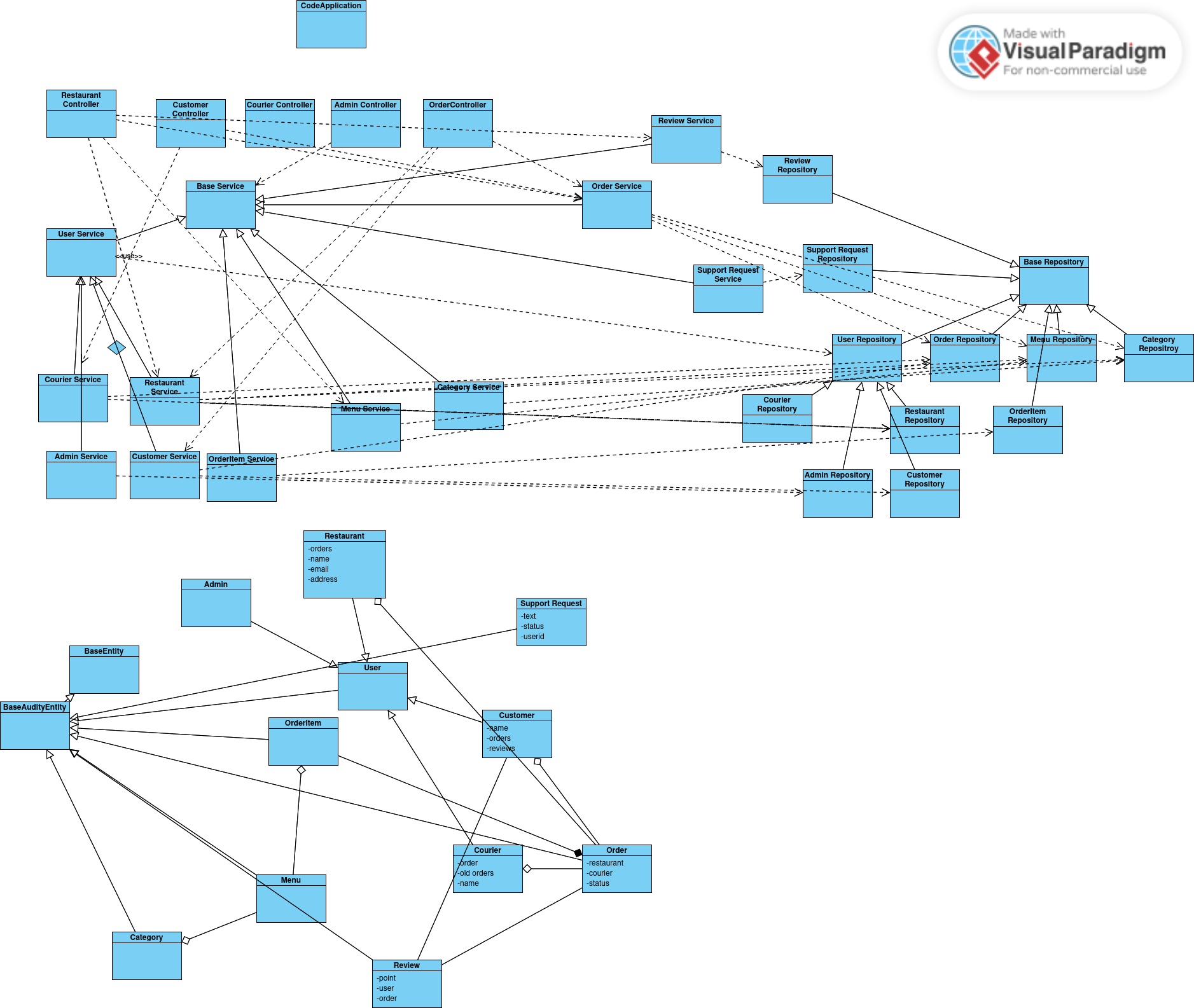
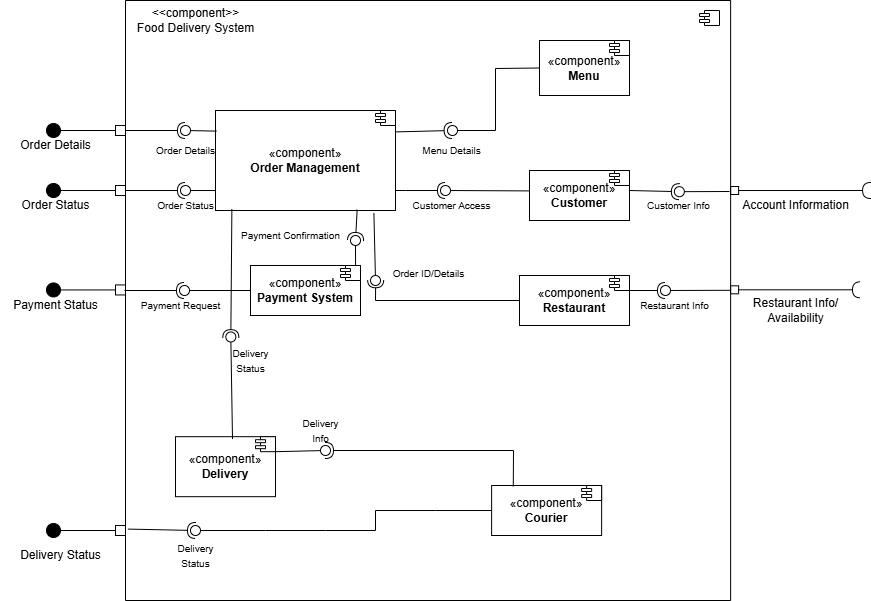
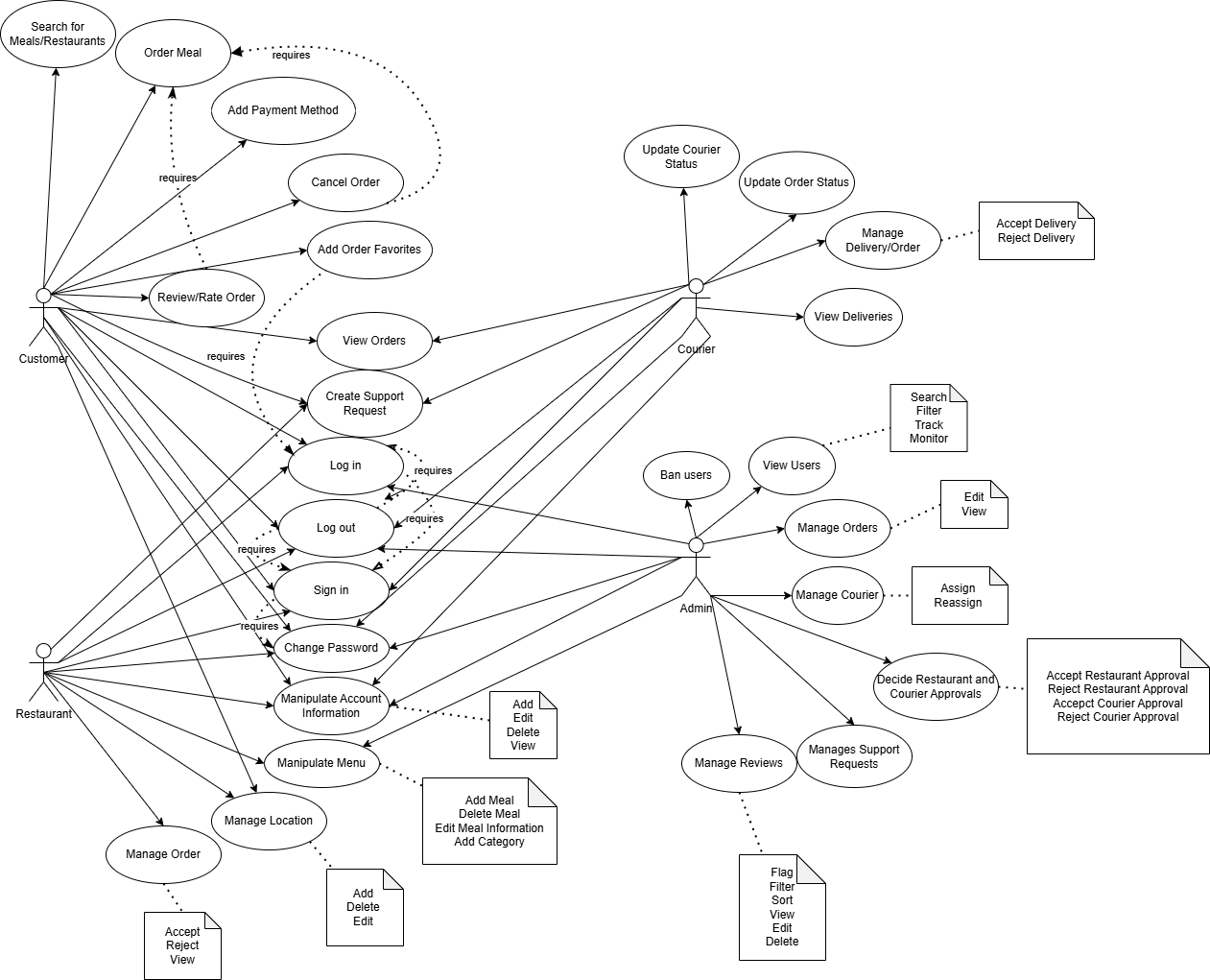
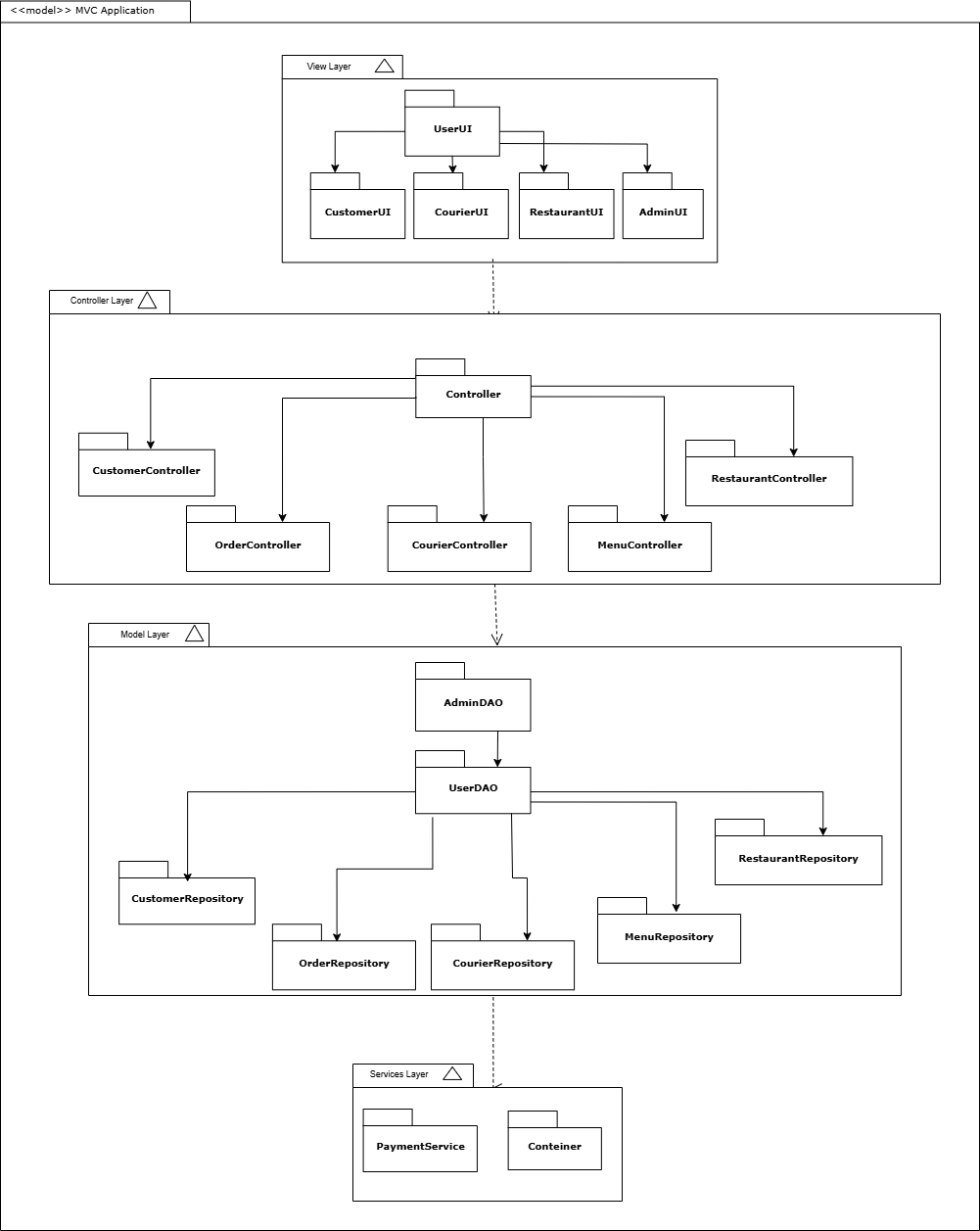
**Version control (e.g., GitLab)**: Essential for collaborated work and correct deployment flow.

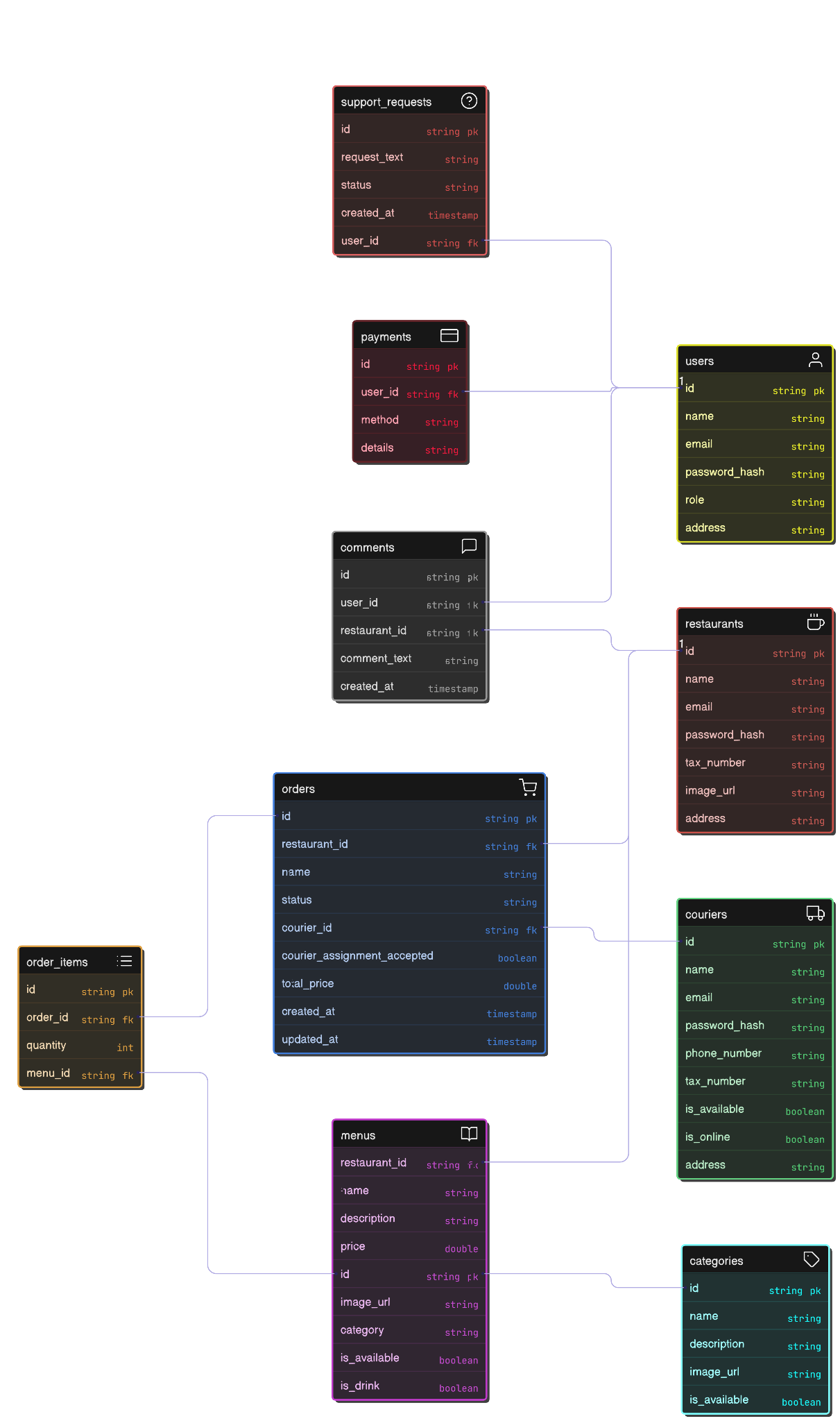
**Third-Party Services**: The security of user data depends on the Data Transfer Object (DTO) functionality within the Spring Boot framework. This mechanism is critical for safeguarding user information during data transfers.

**Database Availability**: It is anticipated that user data will be stored in a PostgreSQL database. Consequently, the database infrastructure must be prepared, configured, and optimized for availability of the system.

# **Architecturally significant requirements**

[Insert a reference or link to the requirements that must be implemented to realize the architecture.]





# **Decisions, constraints, and justifications** [**[1]**](https://chatgpt.com/c/67fac6ce-76c0-8011-aa78-b035ae688686)

[List the decisions that have been made regarding architectural approaches and the constraints being placed on the way that the developers build the system. These will serve as guidelines for defining architecturally significant parts of the system. Justify each decision or constraint so that developers understand the importance of building the system according to the context created by those decisions and constraints. This may include a list of DOs and DON’Ts to guide the developers in building the system.]

#### **a. Minimizing Runtime Errors**

**Decision:** The application must be developed in a way that ensures minimal runtime errors during operation.  
 **Justification:** A stable and reliable user experience is a core goal of the system. Runtime errors not only harm usability but also reduce user trust and increase maintenance costs.

#### **b. Optimizing Page Load Speed**

**Decision:** Frontend components must be optimized for performance, with fast loading times across all devices.  
 **Justification:** Fast performance is critical for user satisfaction and retention. Techniques such as lazy loading, code splitting, and caching will be applied to improve speed.

#### **c. Consistent and Intuitive User Interface**

**Constraint:** The user interface must maintain visual and functional consistency across all pages and components.  
 **Justification:** A consistent UI enhances usability and helps users navigate the system with ease. It also supports accessibility and minimizes the learning curve for new users.

#### **d. Clean and Readable Backend Code**

**Decision:** Backend code must follow clean code principles and be written in a modular, well-documented, and easy-to-understand manner.  
 **Justification:** Clean code improves maintainability, facilitates onboarding of new developers, and reduces the risk of bugs. It also supports long-term scalability and easier debugging.

# **Architectural Mechanisms** [**[2]**](https://chatgpt.com/c/67fac8d0-ffc4-8011-bb11-48bc5b40df1a)

[List the architectural mechanisms and describe the current state of each one. Initially, each mechanism may be only name and a brief description. They will evolve until the mechanism is a collaboration or pattern that can be directly applied to some aspect of the design.]

#### **Architectural Mechanism 1: Consistent Component-Based UI Design**

**Purpose:** To maintain a consistent and intuitive user interface throughout the application.

**Attributes:**

* Reusable UI components
* Centralized styling and theming
* Predictable and uniform user experience

**Function:** The frontend is built using a component-based architecture in React. UI elements like buttons, forms, and modals are created as reusable components with consistent styling, often managed through a design system or CSS-in-JS libraries (e.g., styled-components, Tailwind). This approach ensures visual harmony and reduces UI-related bugs or inconsistencies.

#### **Architectural Mechanism 2: Clean Code Practices in Backend Development**

**Purpose:** To ensure backend code is easy to understand, maintain, and extend over time.

**Attributes:**

* Readable and modular code structure
* Clear separation of concerns
* Comprehensive documentation and naming conventions

**Function:** Backend services are structured following clean code principles such as single-responsibility, clear function names, and proper abstraction layers (e.g., controller, service, repository). Code reviews, linters, and formatters are used to enforce consistency and improve code quality. This leads to better team collaboration and easier long-term maintenance.

#### **Architectural Mechanism 3: Global Error Handling**

**Purpose:** To ensure that runtime errors are centrally managed and do not disrupt the user experience.

**Attributes:**

* Captures and logs all unexpected errors
* Provides consistent and user-friendly error messages
* Prevents system crashes by gracefully handling exceptions

**Function:** A global error-handling mechanism is implemented in the backend (e.g., using ControllerAdvice in Spring Boot) to catch and manage all exceptions uniformly. On the frontend, dedicated error components display helpful messages to the user without breaking the application flow. This improves overall system reliability and debugging efficiency.

#### **Architectural Mechanism 4: Lazy Loading and Caching for Performance Optimization**

**Purpose:** To reduce initial load times and improve performance by only loading necessary data and components when needed.

**Attributes:**

* Improves perceived speed for users
* Reduces server load and network traffic
* Enhances scalability of the system

**Function:** Frontend components are loaded lazily using tools like ReactLazy and Suspense, meaning they are only fetched when required. Backend responses for frequently accessed data are cached using mechanisms such as Redis or in-memory caches to reduce database hits. This ensures smooth and fast page transitions across the application.

# **Key abstractions**

[List and briefly describe the key abstractions of the system. This should be a relatively short list of the critical concepts that define the system. The key abstractions will usually translate to the initial analysis classes and important patterns.]

In a food delivery application, there are several key abstractions that define the core structure and behavior of the system. One of the most essential abstractions is the **User**. This includes customers who place orders, restaurant owners who manage menus, and couriers who deliver food and system admin. Each user has specific attributes such as name, contact details, delivery addresses, and a record of past activities. Authentication and user roles are also important parts of this abstraction.

Another critical abstraction is the **Restaurant**. This represents the businesses offering food on the platform. Each restaurant has its own **Menu**, which includes a list of **Items** such as meals, drinks, and desserts. These items have properties like name, description, price, and sometimes images. There is also the **Cart** abstraction, which allows users to collect and review their selected items before placing an order. The cart supports actions like adding or removing items and calculating the total cost.

Additionally, the system includes abstractions for **Order** and **Delivery**, which manage the process from placing an order to delivering it to the customer. The **Order** tracks details like which items were ordered, the total amount, and the order status (e.g., preparing, on the way, delivered). The **Delivery** abstraction includes information about the assigned **Courier**, estimated delivery time, and delivery tracking. Together, these abstractions allow the system to operate efficiently and provide a smooth experience for all users involved.

# **Layers or architectural framework**[**[3]**](#bookmark=id.fjb1p76620tg)

[Describe the architectural pattern that you will use or how the architecture will be consistent and uniform. This could be a simple reference to an existing or well-known architectural pattern, such as the Layer framework, a reference to a high-level model of the framework, or a description of how the major system components should be put together.]

### **Architectural Pattern: Client-Server**

This project follows the **Client-Server architectural pattern**, which provides a consistent and uniform structure for system development. The architecture separates the application into two main roles: the **Client**, which handles the user interface and interactions, and the **Server**, which handles business logic, data processing, and communication with the database.

#### **Client (Frontend)**

The **React frontend** serves as the **Client** in the system. It is responsible for:

* Rendering the user interface (UI)
* Capturing and responding to user input
* Sending HTTP requests (such as GET and POST) to the server
* Handling the server's responses and updating the UI accordingly

The client operates independently from the server and communicates with it via RESTful API calls, usually over HTTP/HTTPS.

#### **Server (Backend)**

The **Java Spring backend** serves as the **Server**. It is responsible for:

* Receiving and processing client requests
* Executing the appropriate business logic
* Validating input and handling errors
* Communicating with the database to fetch or store data
* Returning structured responses (e.g., JSON) to the client

The server acts as the central authority in terms of data and application logic.

#### **Database**

The system includes a **relational database** (e.g., PostgreSQL, MySQL, or another supported by Spring) that stores persistent application data. The backend interacts with the database through Java-based data access layers, managing:

* User data
* Application records
* Business-related entities

The server ensures that all data operations are secure, validated, and abstracted away from the client.

#### **Overall Communication Flow**

The overall communication follows a **request-response model**:

1. The client sends an HTTP request to the server (e.g., to fetch or update data).
2. The server processes the request, performs any needed database operations, and formulates a response.
3. The server returns the response (typically in JSON) to the client.
4. The client updates the UI based on the response.

#### **Benefits of this Architecture**

* **Separation of concerns** between UI, logic, and data
* **Independent development** of frontend and backend
* **Scalability** in terms of handling multiple clients
* **Maintainability** and ease of updates
* **Security** through controlled server-side logic and database access

# **Architectural views**[**[4]**](#bookmark=id.fjb1p76620tg)

[Describe the architectural views that you will use to describe the software architecture. This illustrates the different perspectives that you will make available to review and to document architectural decisions.]

To design and document the system architecture effectively, we adopt the **4+1 View Model of Software Architecture**, which captures the software from multiple perspectives. Each view addresses specific stakeholder concerns and collectively provides a comprehensive understanding of the system.

#### **1. Logical View (Conceptual)**

This view focuses on the **functional requirements** of the system, representing how the system is structured in terms of key abstractions and their relationships. We use:

* **Class Diagram**: To model the main classes and their relationships in the system.
* **Updated ER Diagram**: To illustrate the entities, relationships, and data model used in the backend.

This view is most useful for developers and end-users to understand the system’s design at a high level.

#### **2. Process View**

This view addresses the **runtime behavior** of the system, particularly how system components interact and communicate during execution. It emphasizes performance, scalability, and concurrency. In our project:

* The system follows a **Client-Server model**, where the frontend (React) interacts with the backend (Spring) via HTTP requests.
* Use of asynchronous requests, API endpoints, and data handling mechanisms is key to the process view.
* This can be complemented by **sequence diagrams** or **communication diagrams** (if needed) to visualize request/response flows.

#### **3. Development View (Implementation)**

This view, also called the **Implementation View**, shows how the software is organized in the development environment. It is useful for programmers and software managers. We have used:

* **Package Diagram**: To represent the modular breakdown of the backend into controllers, services, repositories, and models.
* **Component Diagram**: To show how various frontend and backend components are wired together.

This helps in understanding the project structure and managing codebase complexity.

#### **4. Physical View (Deployment)**

This view describes the **physical deployment** of software artifacts on the hardware. It includes the infrastructure, servers, and nodes. We have created:

* **Deployment Diagram**: Illustrating how the client (browser) communicates with the server (Java Spring backend) which in turn connects to a relational database.
* Shows the mapping of software components to hardware infrastructure (e.g., web server, application server, database server).

#### **5. Scenario View (Use-Case)**

The **central "Scenario" view** represents typical use-cases and user interactions with the system. It ties all the other views together and validates architectural design decisions. We have:

* **Updated Use-Case Diagram**: Representing various user roles and their interactions with the system.
* **User Stories**: To describe real-world usage and expectations from the end-user's perspective.

# **Traceability Table**

|  | Notebook text | Class Diagram | Use Case Diagram | Package Diagram | Component Diagram | Deployment Diagram | Summation |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Yusuf Küçüköner | 4h | 2h | 1h | 2h | 3h | 2h | 14 |
| Salih Eren Yüzbaşıoğlu | 3h | 2h | 2h | 2h | 1h | 2h | 14 |
| Şükriye Öztürk | 4h | 2h | 1h | 2h | 2h | 4h | 15 |
| Mustafa Furkan Ateş | 1h | 2h | 2h | 3h | 4h | 2h | 14 |
| Bedirhan Gençaslan | 2h | 2h | 2h | 3h | 3h | 2h | 14 |

# **Prompts**

* ChatGPT: <https://chatgpt.com/c/67fac6ce-76c0-8011-aa78-b035ae688686>
* ChatGPT: <https://chatgpt.com/c/67fac8d0-ffc4-8011-bb11-48bc5b40df1a>

* ChatGPT: https://chatgpt.com/share/67faed09-dc20-8000-81e1-5703f8b28c14