Functional Documentation

Ebthisam Moideen Koya, Jeevan Babu G

September 3, 2024

Contents

1 Scope

This document outlines the architecture and design of a microservices-based system. The project aims to create a scalable, maintainable, and secure platform that supports various business functionalities such as user management, vendor management, product catalog, order processing, and payments. Each functionality is encapsulated within its microservice, which communicates with others via RESTful APIs.

2 Functional Requirements

The functional requirements define what the system should do. They include:

- **User Management**: The system must allow users to register, log in, log out, change password, and manage their profiles.
- **Vendor Management**: Vendors should be able to register, log in, manage their profiles, and view products associated with them.
- **Product Management**: The system must support adding, updating, deleting, and retrieving product information, along with managing reviews.
- **Order Management**: The system must allow users to place orders, view order history, and track order status.
- **Payment Processing**: The system must handle payment transactions securely and update order statuses accordingly.

3 Non-Functional Requirements

Non-functional requirements define how the system performs its functions:

- **Performance**: The system should handle at least 1000 concurrent users with response times under 200ms.
- **Scalability**: The architecture should allow horizontal scaling to support increased load.
- **Security**: Data must be encrypted in transit and at rest. Use JWT for authentication and authorization.
- **Availability**: The system should have an uptime of 99.9%, with failover mechanisms in place.
- **Maintainability**: Code should follow SOLID principles, with high modularity and low coupling.

4 Technologies Used

The project employs the following technologies:

- **Programming Language**: Java 17 for backend services.
- **Frameworks**:
  - Spring Boot for building microservices.
  - Spring Security for authentication and authorization.

  - Spring Cloud Netflix Eureka for service discovery.
  - Spring Cloud Gateway for API routing.

- **Database**: MongoDB for storing persistent data.
- **API Communication**: RESTful APIs using Spring Web.
- **Frontend**: Angular for building the user interface.
- **Containerization**: Docker for containerizing microservices.

5 High-Level Design

5.1 Architectural Overview

The architecture is based on a microservices pattern, where each service is responsible for a specific business capability. Each microservice runs independently, communicating with others via HTTP REST APIs. The key components include:

- **API Gateway**: Acts as a single entry point for all client requests, routing them to the appropriate microservice.
- **Authentication Service**: Manages user and vendor authentication using JWT.
- **User Service**: Handles user-related operations such as registration, login, change password, and profile management.
- **Vendor Service**: Manages vendor-related operations, including registration, login, and profile management.
- **Product Service**: Manages the product catalog, including CRUD operations, category management, and review functionality.
- **Order Service**: Processes orders, manages order items, and tracks order status.
- **Payment Service**: Processes payments and updates order statuses.
- **Notification Service**: Sends email and SMS notifications to users regarding their orders.

5.2 Functional Specifications

This section details the functionality and behavior of each component. Below is an example sequence diagram that illustrates the process of placing an order:

Sequence Diagram: Placing an Order

- The user logs in and browses products.
- The user selects a product and adds it to their cart.
- Upon checkout, the Order Service verifies product availability with the Product Service.
- The Payment Service processes the payment.
- Upon successful payment, the Order Service updates the order status.
- The Notification Service sends a confirmation message to the user.

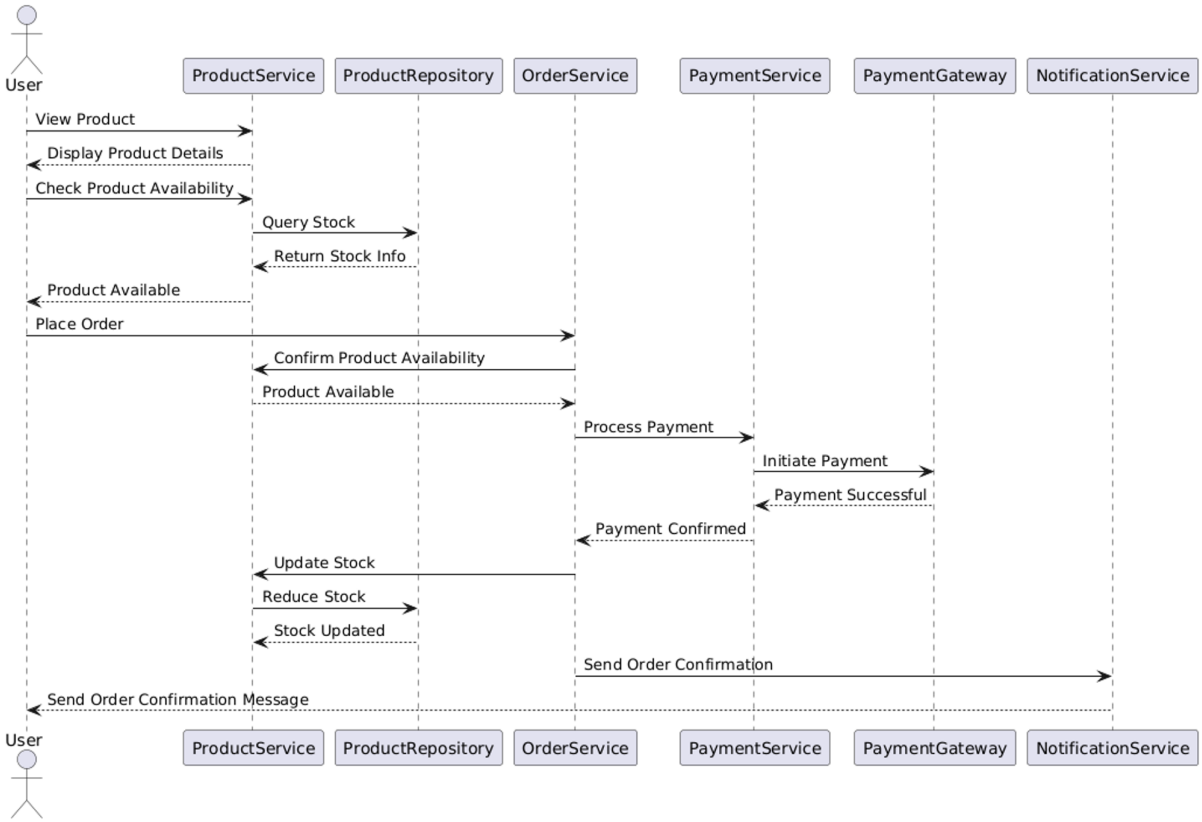

Figure 1: Sequence Diagram: Placing an Order

Sequence Diagram: Vendor Login

- The vendor submits login credentials.
- The Vendor Service validates the credentials.
- If valid, the service generates a JWT token and returns it.
- The vendor uses the token for subsequent requests.

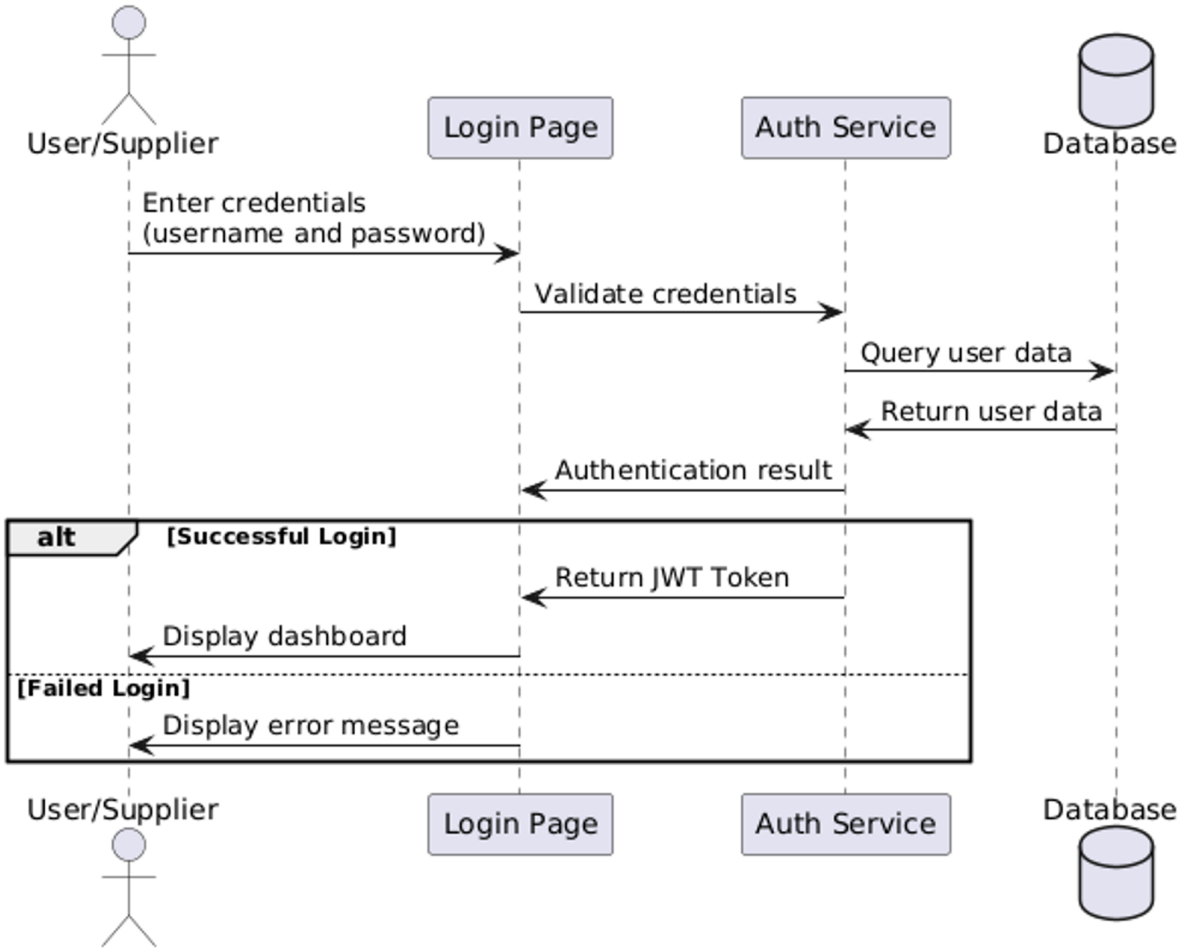

Figure 2: Sequence Diagram: Vendor Login

5.3 System Interactions

This section describes how different components interact with each other. The interactions are illustrated using a wireframe diagram that shows the flow between different pages and components of the system. For example:

- The homepage allows users to log in or browse products by category.
- After logging in, users can access their profile, view order history, and manage their cart.
- Vendors can view and manage products they have listed on the platform.
- The checkout process involves interaction between the Order, Payment, and Notification services.

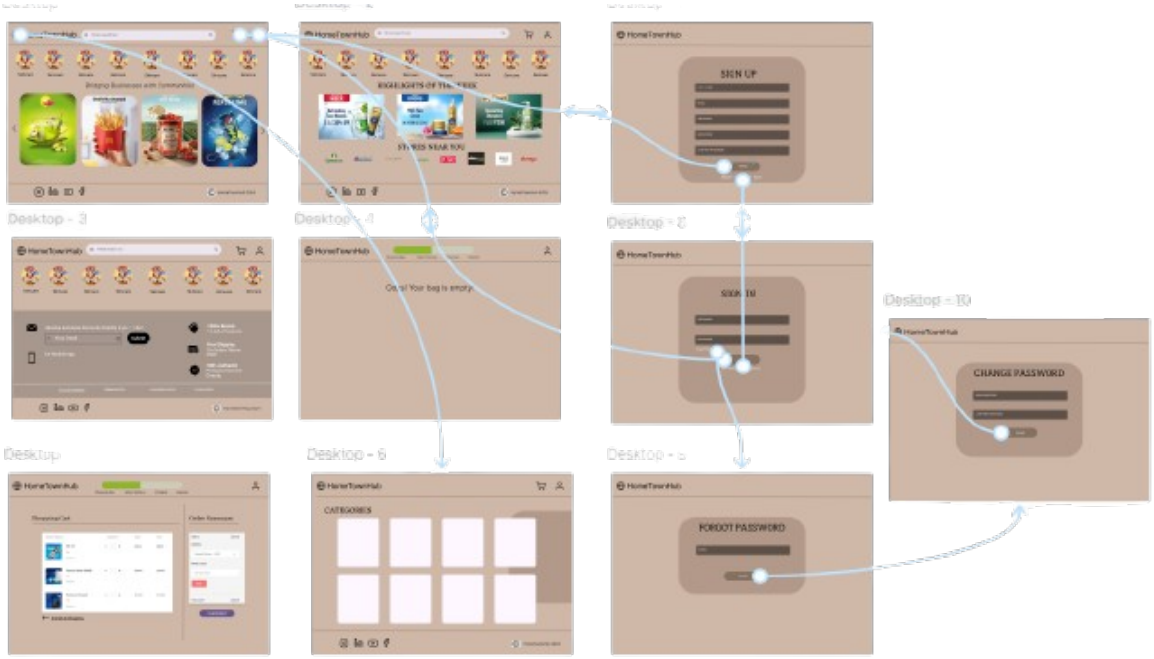

Figure 3: Wireframe Diagram: System Interactions

5.4 Performance Considerations

The system is designed to be highly performant. Key considerations include:

- **Caching**: Frequently accessed data, such as product listings, is cached to reduce load on the database.
- **Load Balancing**: Incoming requests are distributed across multiple instances of each microservice to ensure even load distribution.
- **Asynchronous Processing**: Non-critical tasks, such as sending notifications, are processed asynchronously using message queues to improve response times.
- **Database Optimization**: Indexing is used on frequently queried fields to speed up data retrieval.

5.5 Technology Stack

The technology stack includes:

- **Backend**: Java, Spring Boot, Spring Security.
- **Frontend**: Angular, Bootstrap, HTML, CSS
- **Database**: MongoDB
- **Containerization**: Docker
- **Service Discovery**: Spring Cloud Netflix Eureka for managing microservices discovery.
- **API Gateway**: Spring Cloud Gateway for routing requests to the appropriate services.
- **CI/CD**: Jenkins, GitHub Actions for continuous integration and deployment

6 Low-Level Design

6.1 Detailed Component Specification

Each microservice is composed of several components, including controllers, services, and repositories:

- **Controller**: Exposes RESTful endpoints and handles HTTP requests.
- **Service**: Contains the business logic and interacts with repositories.
- **Repository**: Manages data persistence and retrieval from the MongoDB database.

For example, the Vendor Service includes:

- **VendorController**: Handles vendor-related API requests.
- **VendorService**: Contains the business logic for managing vendors, including registration, login, and profile management.
- **VendorRepository**: Interfaces with the database to manage vendor data.

6.2 Interface Definitions

Interfaces define the contracts between different services and components. Each service exposes a set of RESTful APIs that can be consumed by other services or the frontend. The endpoints are as follows:

- **User Service API**:
  - ‘POST /api/users/register‘: Registers a new user.
  - ‘POST /api/users/login‘: Authenticates a user and returns a JWT token.
  - ‘PUT /api/users/email‘: Updates a user’s details by email.
  - ‘DELETE /api/users/email‘: Deletes a user by email.
  - ‘GET /api/users/email‘: Retrieves a user by email.
  - ‘GET /api/users‘: Lists all users.
  - ‘GET /api/users/userId/orders‘: Retrieves all orders associated with a specific user.
- **Vendor Service API**:
  - ‘POST /api/vendors/register‘: Registers a new vendor.
  - ‘POST /api/vendors/login‘: Authenticates a vendor and returns a JWT token.
  - ‘PUT /api/vendors/id‘: Updates a vendor’s details by ID.
  - ‘DELETE /api/vendors/id‘: Deletes a vendor by ID.
  - ‘GET /api/vendors/id‘: Retrieves a vendor by ID.
  - ‘GET /api/vendors/contact/contactMail‘: Retrieves a vendor by contact email.
  - ‘DELETE /api/vendors/contact/contactMail‘: Deletes a vendor by contact email.
  - ‘GET /api/vendors‘: Lists all vendors.
  - ‘GET /api/vendors/id/products‘: Retrieves all products associated with a specific vendor.
- **Product Service API**:

  - ‘POST /api/products‘: Adds a new product.
  - ‘GET /api/products‘: Retrieves a list of all products.
  - ‘GET /api/products/id‘: Retrieves details of a specific product, including reviews.
  - ‘GET /api/products/vendor/vendorId‘: Retrieves all products associated with a specific vendor.
  - ‘GET /api/products/category/categoryId‘: Retrieves all products under a specific category.
  - ‘PUT /api/products/id‘: Updates an existing product.
  - ‘DELETE /api/products/id‘: Deletes a product.
  - ‘GET /api/products/id/reviews‘: Retrieves all reviews associated with a specific product.

- **Order Service API**:
  - ‘POST /api/orders‘: Creates a new order.
  - ‘GET /api/orders‘: Lists all orders.
  - ‘GET /api/orders/id‘: Retrieves an order by ID.
  - ‘PUT /api/orders/id‘: Updates an existing order by ID.
  - ‘DELETE /api/orders/id‘: Deletes an order by ID.
  - ‘GET /api/orders/user/userId‘: Retrieves all orders associated with a specific user.
  - ‘GET /api/orders/order-items/orderId‘: Retrieves all order items associated with a specific order.

6.3 Resource Allocation

Resource allocation involves distributing CPU, memory, and storage among the microservices. Each service is allocated resources based on its expected load:

- **Product Service**: Requires more memory for caching product data.
- **Order Service**: Needs higher CPU allocation for processing large numbers of transactions.
- **Payment Service**: Must have secure and fast processing, so it is allocated both CPU and memory resources.

6.4 Error Handling and Recovery

Error handling is crucial to maintain system stability. The following strategies are implemented:

- **Try-Catch Blocks**: Used to handle exceptions in code.
- **Fallback Mechanisms**: In case of service failure, fallback methods are invoked to ensure continuity.
- **Logging**: Errors are logged with details for debugging and analysis.
- **Circuit Breaker Pattern**: Prevents a chain of failures when one service is down.

6.5 Security Considerations

Security is a top priority in the system design. Key considerations include:

- **Authentication**: JWT is used for secure user and vendor authentication.
- **Authorization**: Role-based access control ensures users and vendors only access permitted resources.
- **Data Encryption**: Sensitive data is encrypted both in transit and at rest.
- **Input Validation**: All user inputs are validated to prevent injection attacks.

6.6 Code Structure

The codebase is structured according to the principles of modularity and separation of concerns. Each microservice follows a similar structure:

- **Controller Layer**: Manages HTTP requests.
- **Service Layer**: Contains business logic.
- **Repository Layer**: Manages database operations.
- **DTOs**: Data Transfer Objects are used to encapsulate data for transfer between layers.
- **Utils**: Utility classes and methods that are commonly used across the service.

7 Diagrams

7.1 ER Diagrams

The Entity-Relationship (ER) diagram provides a visual representation of the database schema. It shows the relationships between entities such as ‘Product‘, ‘User‘, ‘Order‘, ‘Vendor‘, and ‘Payment‘.

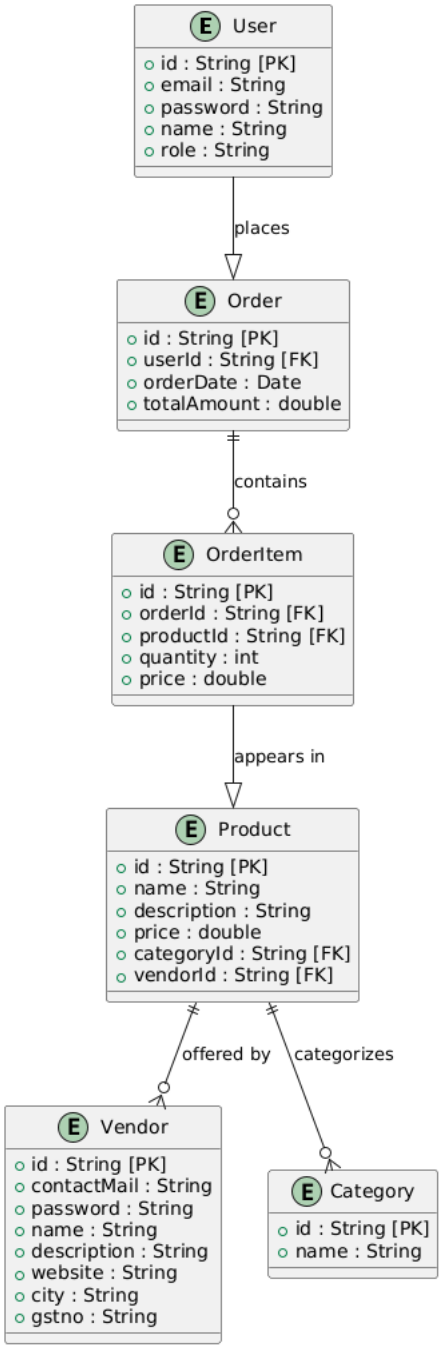

Figure 4: ER Diagram

7.2 Class Diagrams

The class diagram illustrates the structure of the microservices, showing classes, attributes, methods, and the relationships between them.

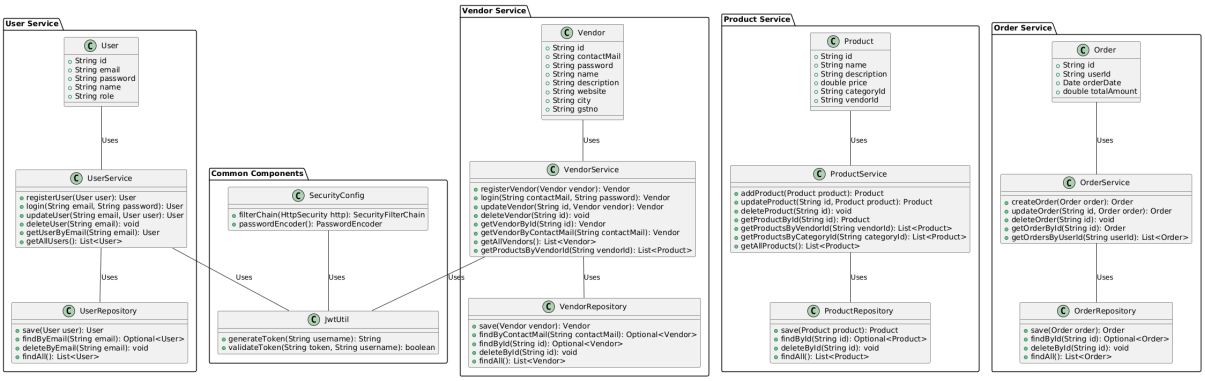

Figure 5: Class Diagram

7.3 Sequence Diagrams

Sequence diagrams depict the interaction between objects or components in a specific order, illustrating how processes or workflows are carried out.

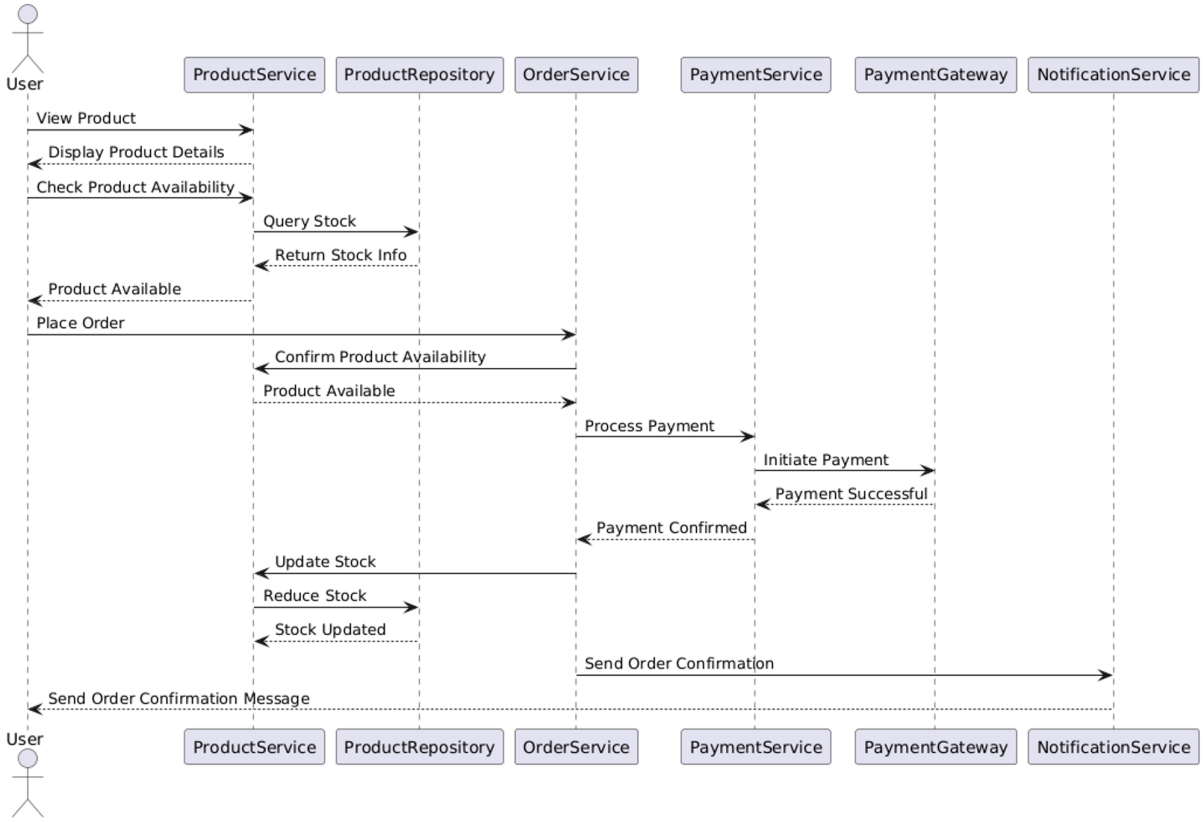

Figure 6: Sequence Diagram: Placing an Order

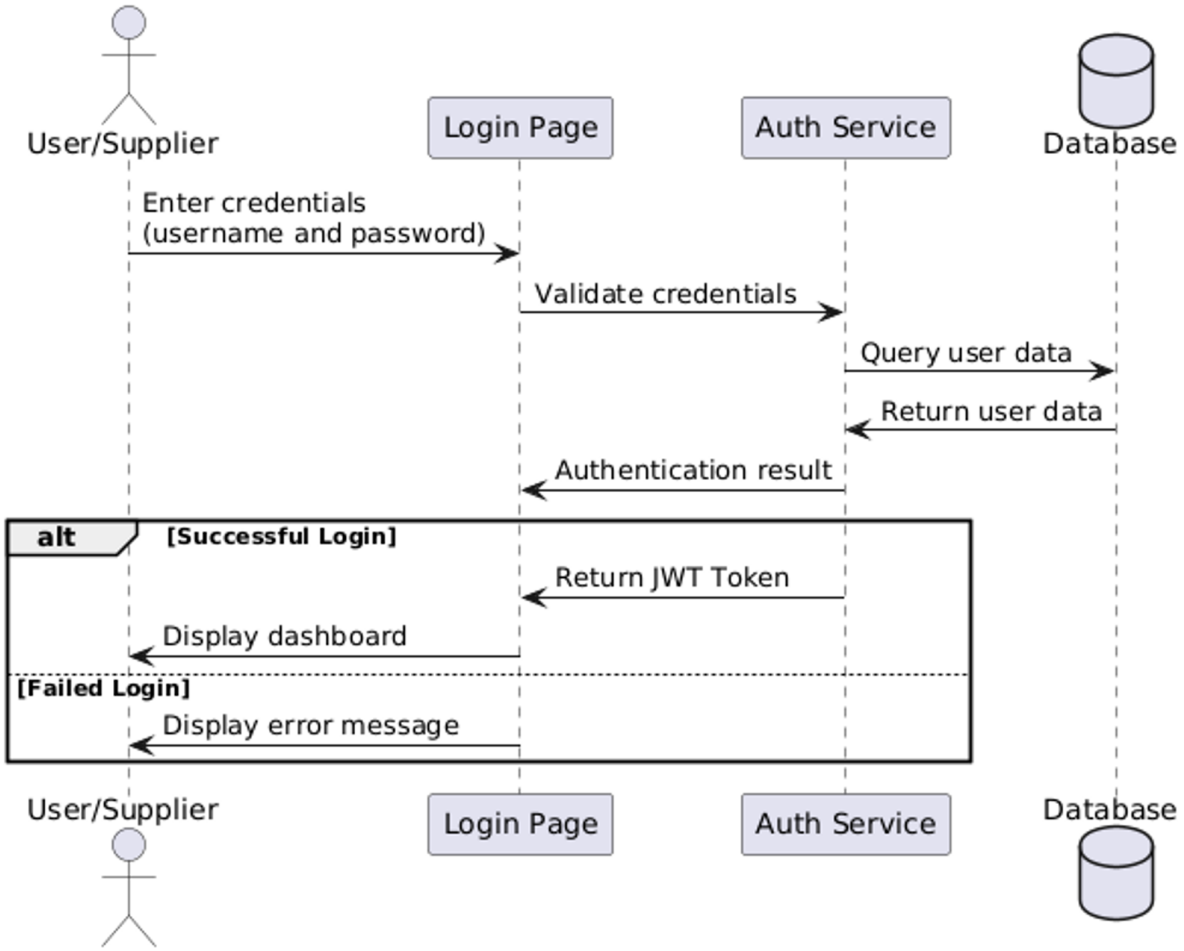

Figure 7: Sequence Diagram: Vendor Login

7.4 Activity Diagrams

Activity diagrams represent the flow of control or data through a sequence of actions or steps, showing the dynamic aspects of the system.

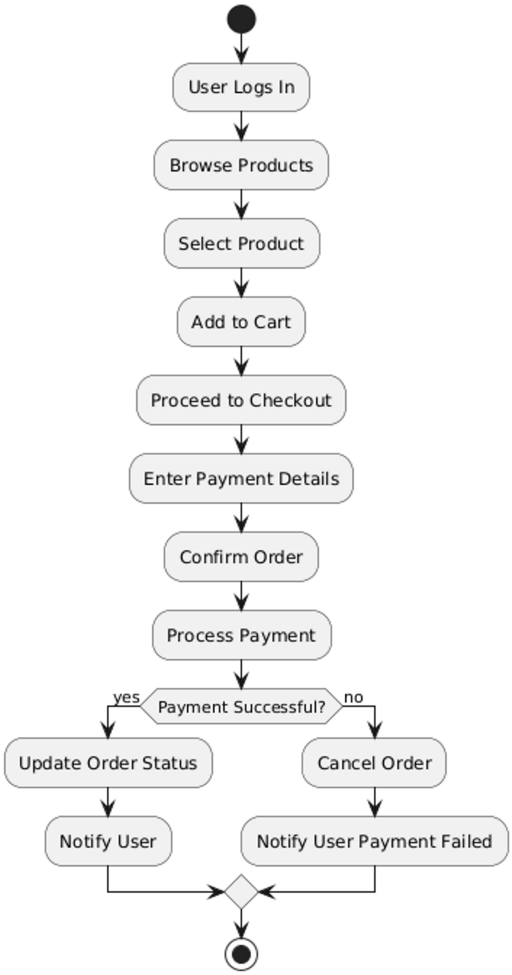

Figure 8: Activity Diagram: Order Processing Workflow

7.5 Use Case Diagrams

Use Case diagrams describe the interactions between users (actors) and the system, highlighting the functionalities provided by the system.

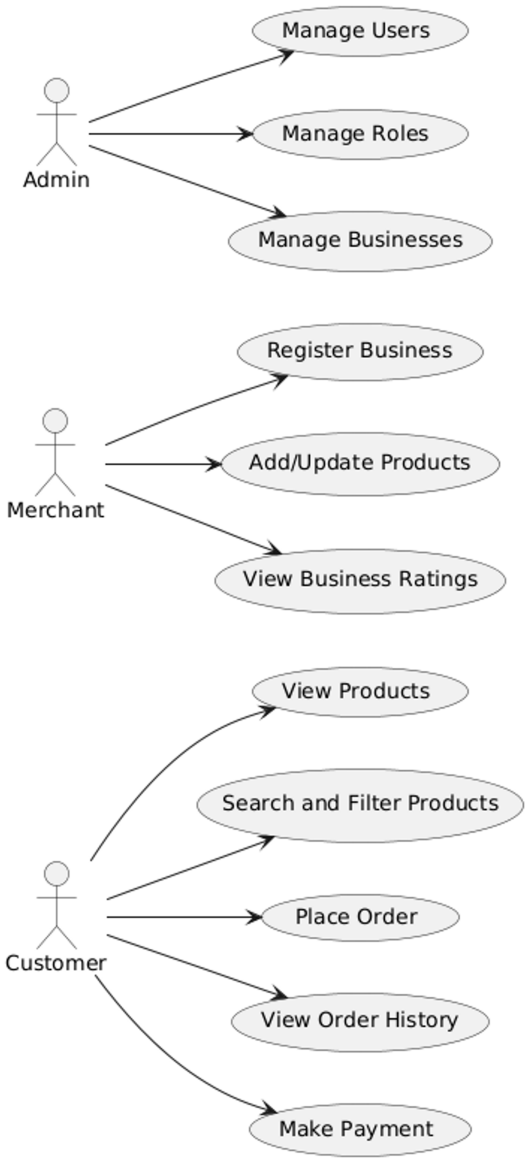

Figure 9: Use Case Diagram: System Interactions

8 Conclusion

In conclusion, the microservices architecture outlined in this document offers a scalable, maintainable, and secure solution for the system's diverse requirements, including user management, vendor management, product cataloging, order processing, and payment handling. By breaking down the system into distinct services, each responsible for a specific business capability, the architecture promotes modularity and separation of concerns, allowing for easier development, testing, and deployment.

The use of Spring Boot, coupled with MongoDB, ensures a robust backend, while Angular provides a responsive and dynamic user interface. The incorporation of JWT for authentication and Docker for containerization further enhances the security and scalability of the system. Additionally, the architecture supports horizontal scaling, ensuring that the system can handle increased loads without compromising performance.

By adhering to the principles of microservices, this system is well-equipped to adapt to future changes and expansions, making it a sustainable solution for the long term. The diagrams and design decisions documented herein provide a clear roadmap for implementation, ensuring that

the development process is both efficient and aligned with best practices.

Overall, this architecture is poised to meet the project's goals and requirements, providing a strong foundation for the system's ongoing success.