Functional Documentation

Nikhil Ramesh Hegde, Pavani Bokam

November 15, 2024

Contents

1 Scope

This document explains the structure and design of a system built using microservices. The goal of the project is to create a platform that can grow easily, is easy to manage, and keeps data safe. The platform will support key business tasks like managing users, handling vendors, displaying products, processing orders. Each task is separated into its own microservice, so they work independently. These microservices talk to each other through RESTful APIs to make the system work smoothly together.

2 Functional Requirements

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

- **User Management**: User should be able to sign up, log in, log out and update 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 vendors to place orders, view order history.

3 Non-Functional Requirements

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

- **Performance**: The system should handle up to 1,000 vendors at once, and responses should be quick (under 200 milliseconds).
- **Scalability**: The design should make it easy to add more servers to handle extra vendors.
- **Security**: Data must be encrypted when sent or stored. Use JWT for authentication and authorization.
- **Availability**: The system should work reliably with 99.9% uptime and have backup options if parts fail.
- **Maintainability**: Code should follow SOLID principles, with high modularity and loose 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 interface.

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 vendor and vendor authentication using JWT.
- **User Service**: Handles user-related operations such as registration, login, change pass-word, 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.
- **Notification Service**: Sends email notifications to vendors 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 log 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 Order Service updates the order status.
- The Notification Service sends a confirmation message to the vendor.

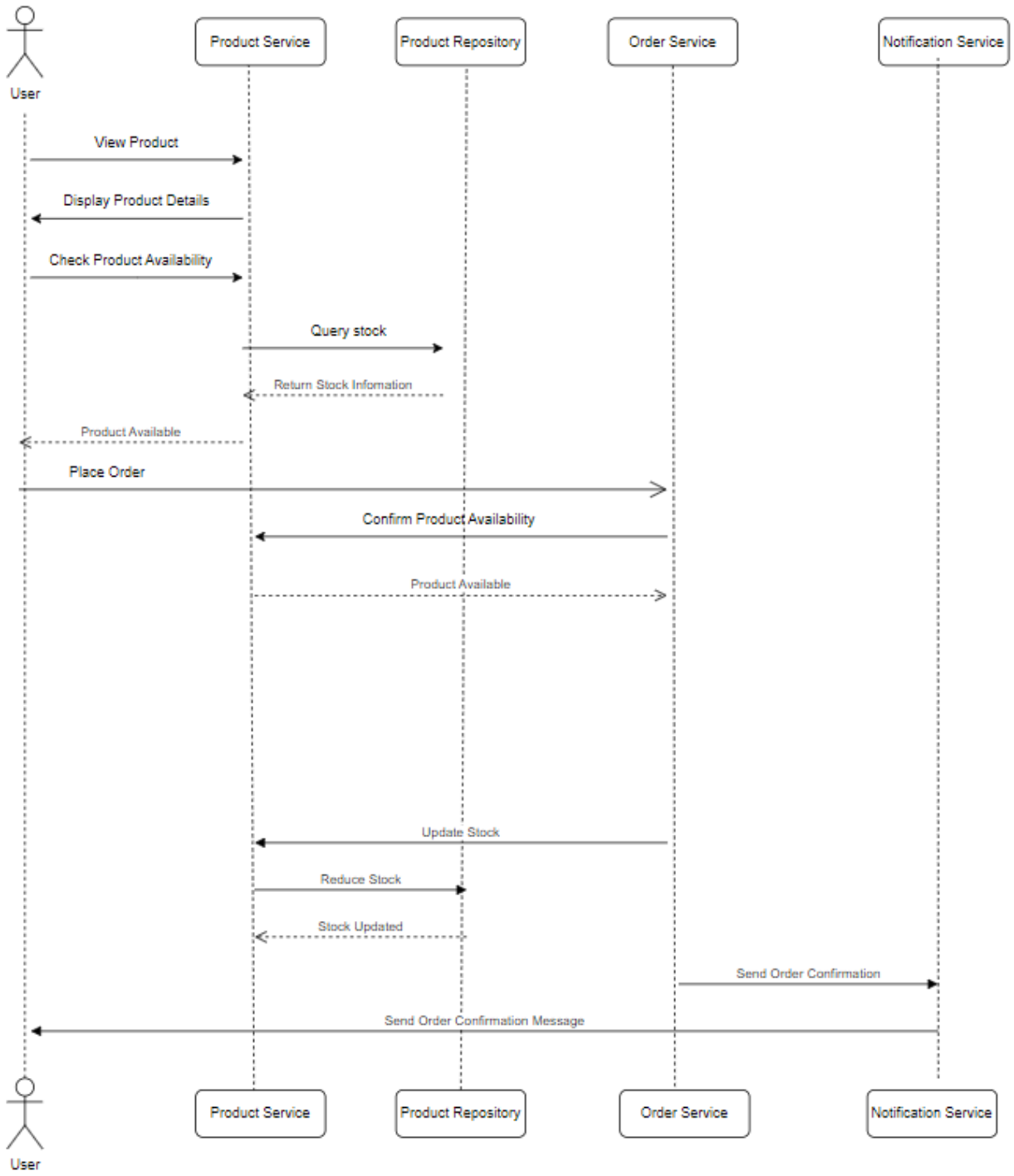

Figure 1: Sequence Diagram: Placing an Order

Sequence Diagram: vendor/user Login

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

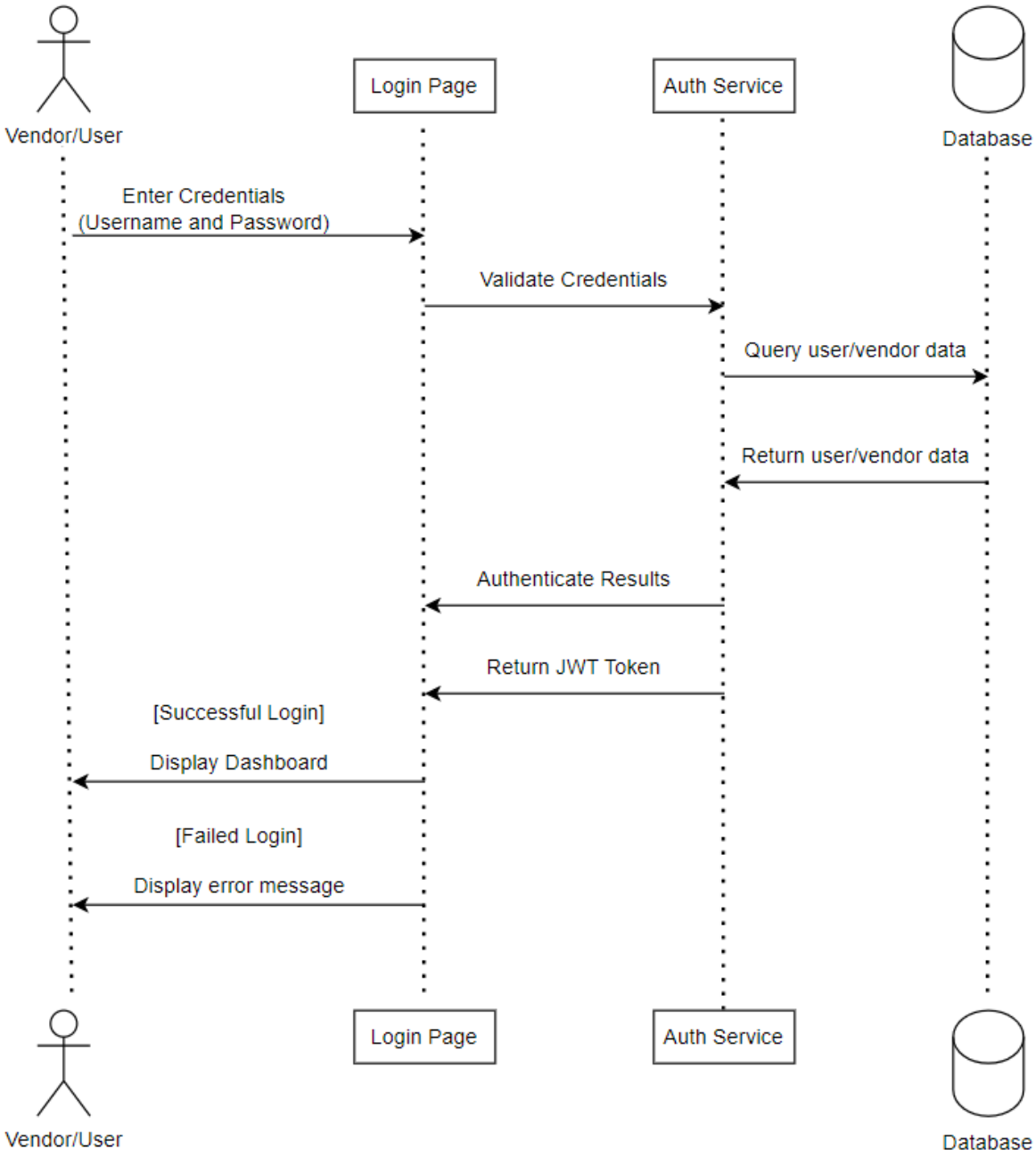

Figure 2: Sequence Diagram: vendor/user Login

5.3 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.

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

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 vendors and vendors only access permit- ted resources.
- **Data Encryption**: Sensitive data is encrypted both in transit and at rest.
- **Input Validation**: All vendor 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', 'vendor', 'Order', 'vendor', and 'Payment'.

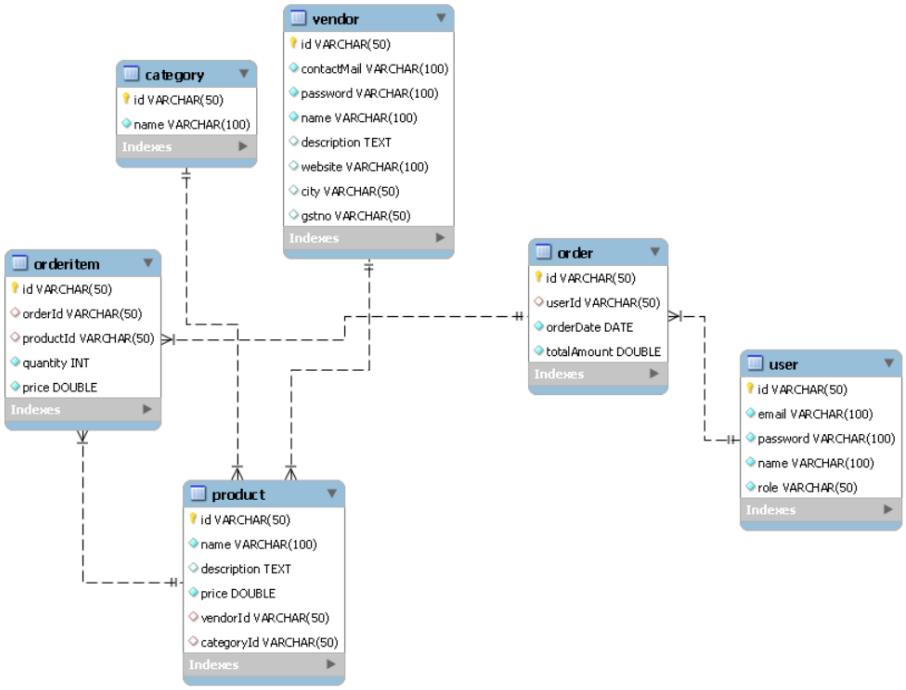

Figure 3: ER Diagram

7.2 Class Diagram

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

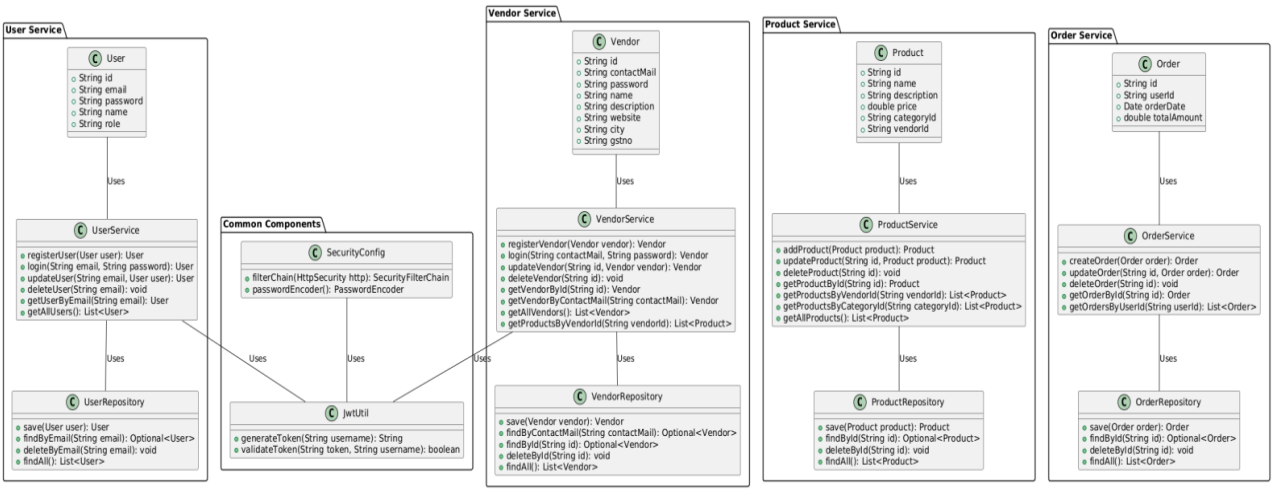

Figure 4: Class Diagram

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.

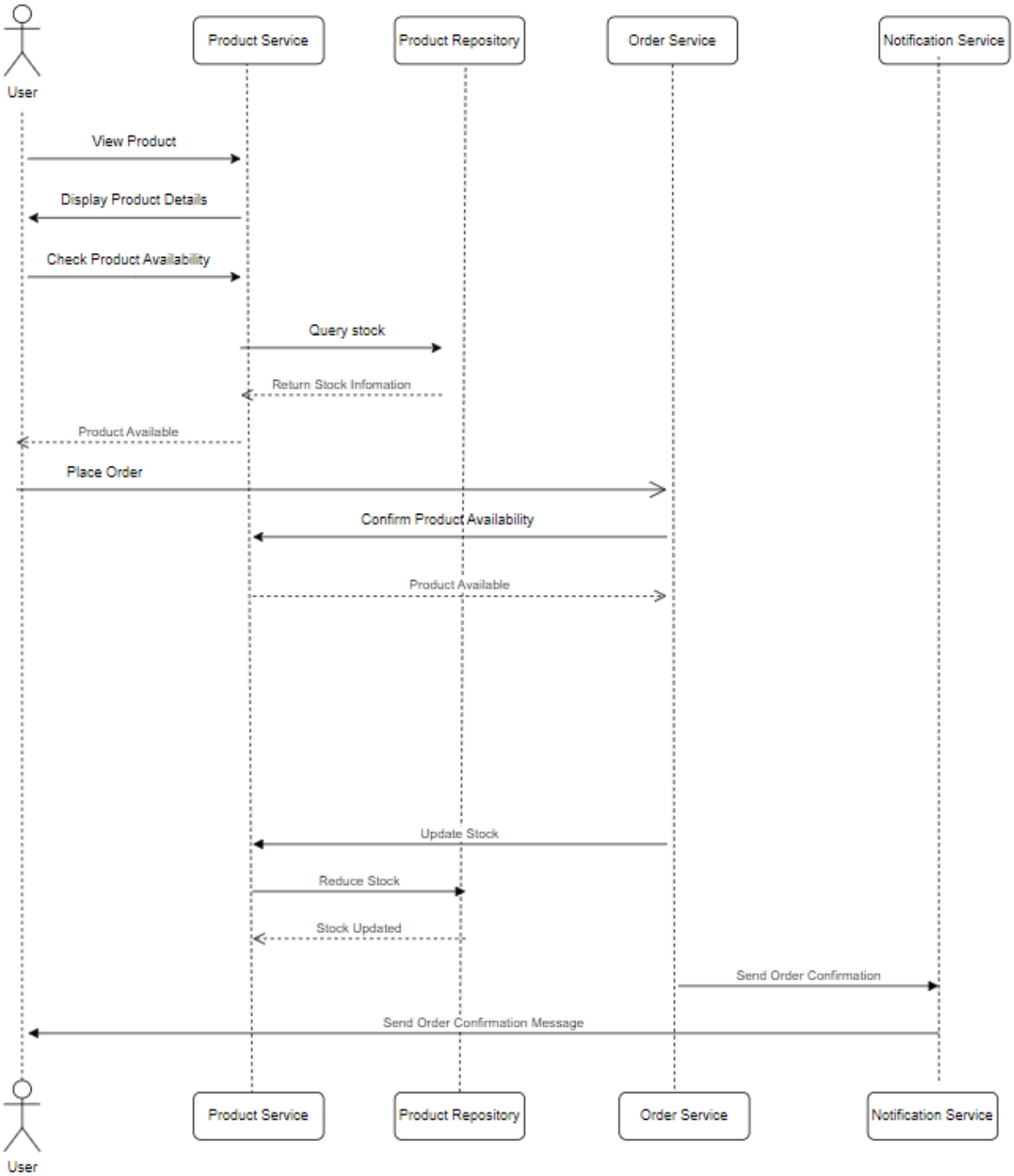

Figure 5: Sequence Diagram: Placing an Order

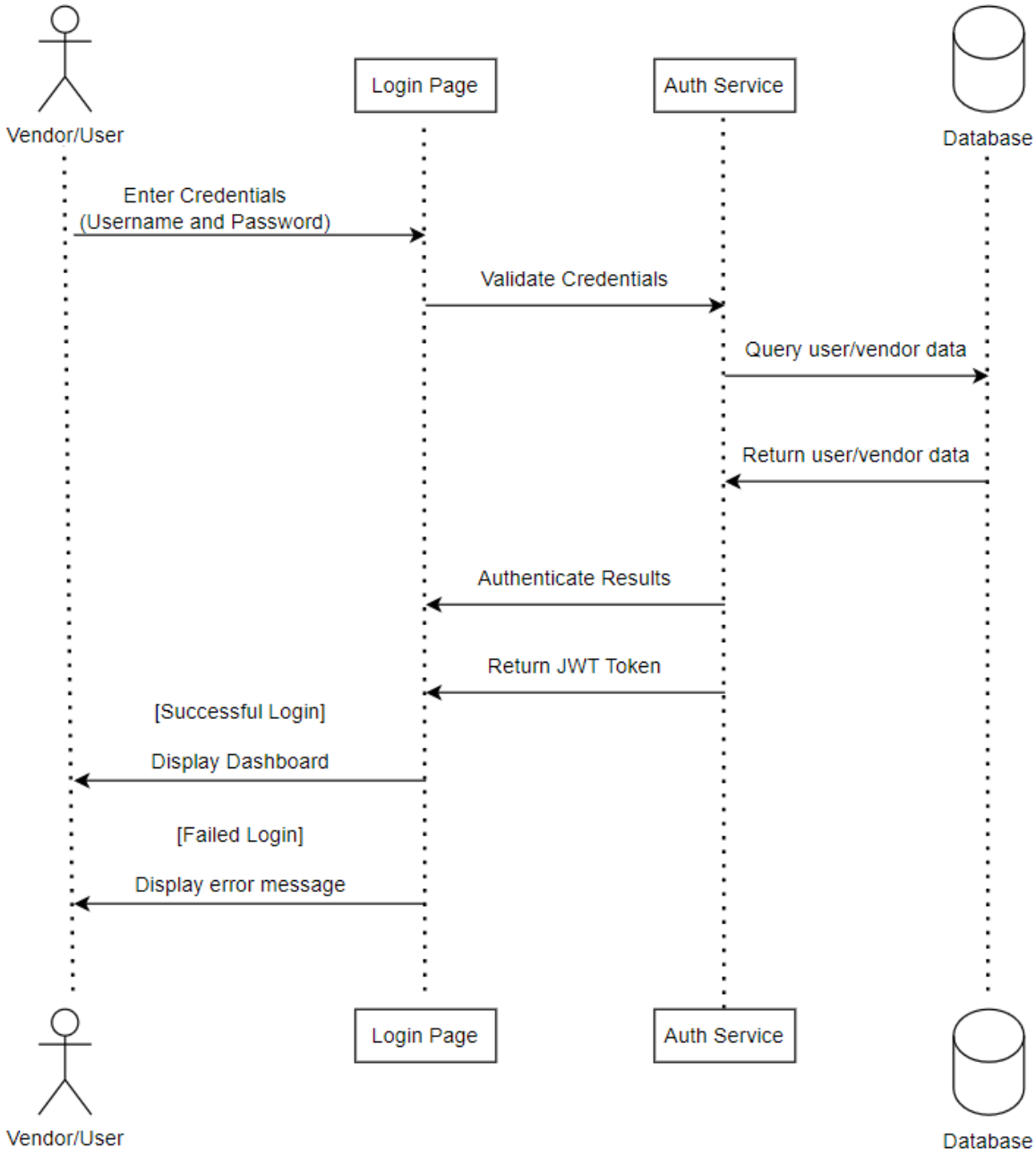

Figure 6: Sequence Diagram: vendor/user Login

7.4 Deployment Diagram

A deployment diagram shows how a system's parts work together in a real environment. It includes the setup of hardware and software components, like servers and applications. This helps us understand how everything is connected and runs together once deployed.

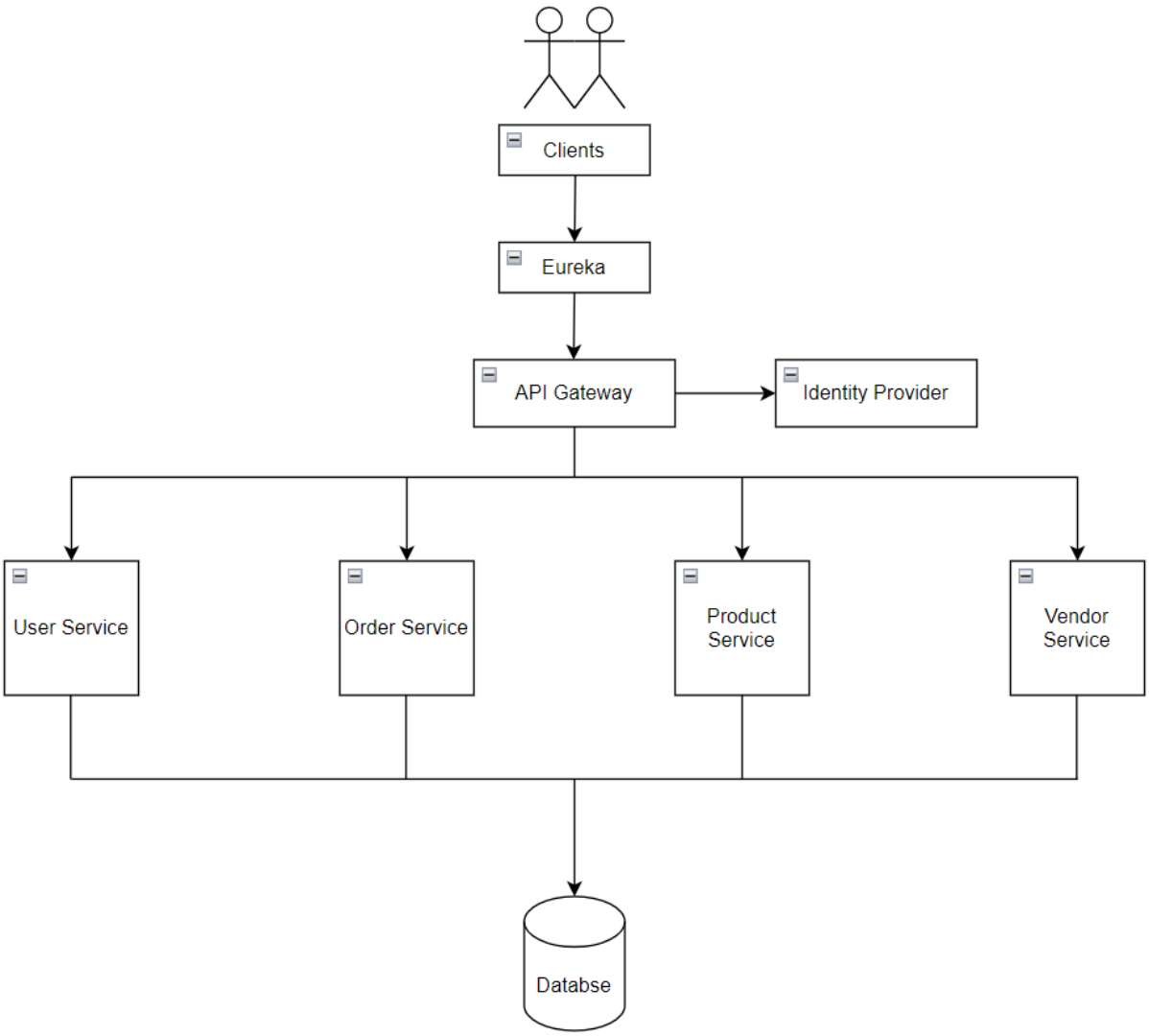

Figure 7: Deployment Diagram

7.5 Use Case Diagram

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

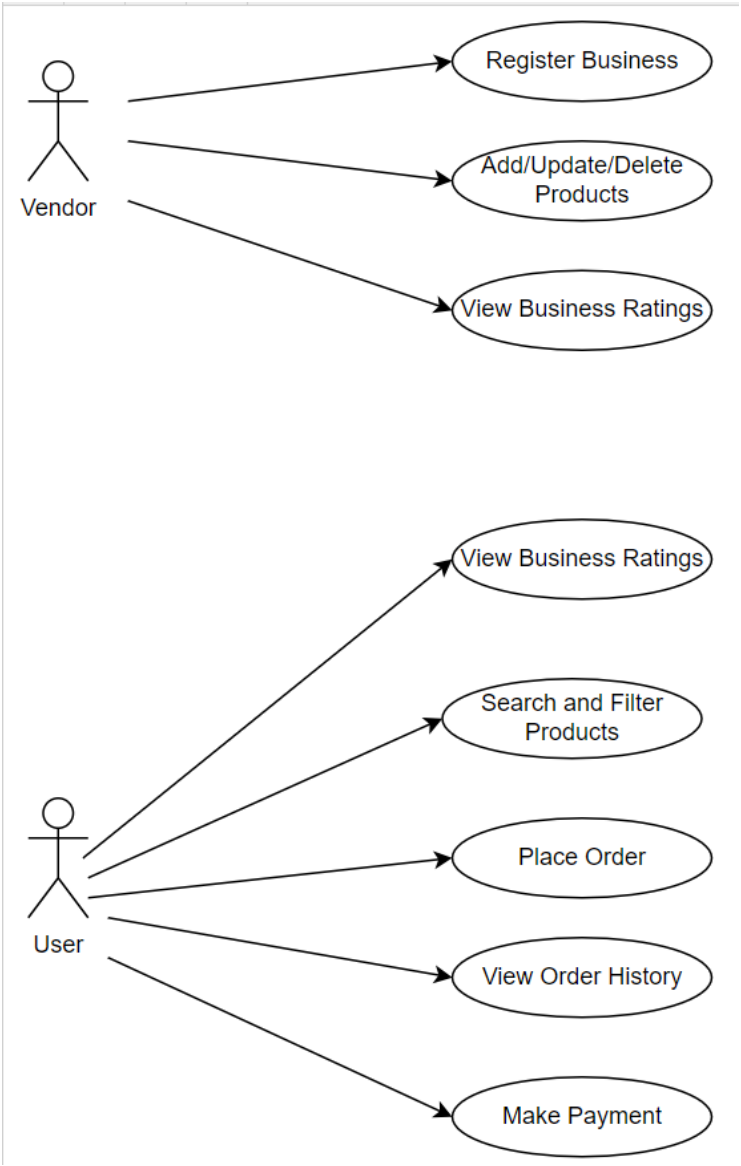

Figure 8: Use Case Diagram: System Interactions

7.6 Package Diagram

A package diagram is a simple way to organize different parts of a system into groups, called "packages." It shows how these groups are connected and depend on each other. This makes it easier to understand big projects by showing which parts work together.

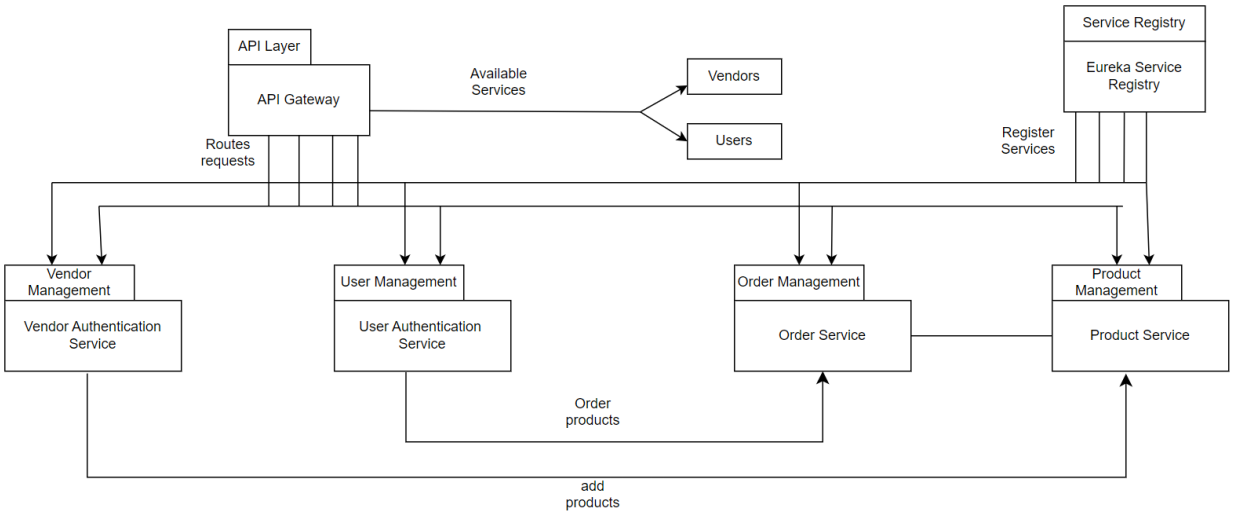

Figure 9: Package Diagram

8 Conclusion

This microservices architecture is designed to be flexible, easy to maintain, and secure, covering all key needs like vendor and vendor management, product cataloging, order processing, and payments. Each part of the system is split into smaller services that focus on one task, making development, testing, and deployment easier. Spring Boot and MongoDB create a solid backend, while Angular provides a responsive vendor experience. Security and scalability are strengthened with JWT for authentication and Docker for containerized deployment. The system can grow by adding more servers to handle increased usage smoothly. This design also supports future updates and changes, ensuring it remains efficient and effective over time.