Business Requirements Document

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, product catalog, order processing, and payments. Each functionality is encapsulated within its microservice, which communicates with others via RESTful APIs. The system is designed to be highly modular, allowing independent development, deployment, and scaling of each service.

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, and manage their profiles. This includes the ability to change passwords, update personal information, and manage security settings.
- **Product Management**: The system must support adding, updating, deleting, and retrieving product information. Products should be organized by categories, and vendors should be able to manage their product listings.
- **Order Management**: The system must allow users to place orders, view order history, and track order status. Orders should be linked to user profiles and include details like order items, total amount, and shipping status.
- **Payment Processing**: The system must handle payment transactions securely and update order statuses accordingly. This includes integrating with third-party payment gateways and ensuring PCI compliance.
- **Business Management**: Merchants should be able to register their businesses, list products, view ratings, and manage their profiles. This includes handling inventory management and viewing sales reports.
- **Review and Rating Management**: The system must allow users to rate and review products they have purchased. Vendors should be able to view and respond to reviews.
- **Search and Filtering**: The system must provide robust search functionality with filters based on categories, price range, ratings, and other product attributes.

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. Performance testing must be conducted to ensure the system meets this requirement under peak load conditions.
- **Scalability**: The architecture should allow horizontal scaling to support increased load. Microservices should be stateless where possible to facilitate scaling.
- **Security**: Data must be encrypted in transit and at rest. Use JWT for authentication and authorization. Implement role-based access control (RBAC) to ensure users only access authorized resources.

- **Availability**: The system should have an uptime of 99.9%, with failover mechanisms in place. Implement load balancing and database replication to ensure high availability.
- **Maintainability**: Code should follow SOLID principles, with high modularity and low coupling. The system should support continuous integration and continuous deployment (CI/CD) pipelines to facilitate regular updates and maintenance.
- **Logging and Monitoring**: Implement comprehensive logging and monitoring for all services to track system health, performance, and security incidents. Logs should be centralized and searchable.
- **Backup and Disaster Recovery**: Regular backups must be taken to prevent data loss. Implement a disaster recovery plan to ensure business continuity in case of a major failure.

4 Technologies Used

The project employs the following technologies:

- **Programming Language**: Java 17 for backend services, providing a robust, object-oriented programming environment suitable for enterprise-level applications.
- **Frameworks**:
  - Spring Boot for building microservices, providing a streamlined approach to creating and deploying Java-based microservices.
  - Spring Security for authentication and authorization, supporting JWT and OAuth2 for securing the system.
  - Spring Cloud for handling service discovery, configuration management, and fault tolerance in a microservices architecture.
- **Database**: PostgreSQL for storing persistent data. PostgreSQL is chosen for its robustness, scalability, and support for advanced features such as JSONB storage and full-text search.
- **API Communication**: RESTful APIs using Spring Web, ensuring that services can interact with each other and with external clients in a standardized manner.
- **Message Broker**: (Optional) RabbitMQ for inter-service communication in asynchronous workflows, enabling decoupled communication between services for better scalability and reliability.
- **Frontend**: Angular for building the user interface, offering a dynamic, responsive, and feature-rich frontend experience.
- **Containerization**: Docker for containerizing microservices, ensuring consistent environments across development, testing, and production.
- **CI/CD**: Jenkins and GitHub Actions for continuous integration and deployment, automating the testing and deployment process to ensure quick and reliable updates.
- **Version Control**: Git for version control, facilitating collaborative development and version tracking.
- **Testing Frameworks**: JUnit and Mockito for unit testing, ensuring that the code is reliable and meets the required standards.
- **Monitoring and Logging**: Prometheus and Grafana for monitoring, and ELK Stack (Elasticsearch, Logstash, Kibana) for centralized logging and analytics.

5 Architecture Overview

The system follows a microservices architecture, where each service is independently deployable and responsible for a specific business capability. The key components of the architecture include:

- **API Gateway**: Acts as a single entry point for all client requests, routing them to the appropriate microservice. It also handles security, rate limiting, and load balancing.
- **Service Discovery**: Spring Cloud Netflix Eureka for dynamically discovering microservices in the architecture, allowing for scalable service registration and discovery.
- **Configuration Management**: Spring Cloud Config for managing external configurations across all environments.
- **Security**: Implemented using Spring Security with JWT for authentication, role-based access control, and secure API communication.
- **Database Management**: Each microservice manages its own database, ensuring loose coupling and allowing for scalability and resilience.
- **Inter-Service Communication**: RESTful APIs and RabbitMQ for synchronous and asynchronous communication between microservices.
- **Monitoring and Logging**: Prometheus and Grafana for real-time monitoring, and the ELK Stack for centralized logging and analytics.

6 Conclusion

This Business Requirements Document outlines the key functional and non-functional requirements of the project, along with the technologies and architecture used to implement a scalable, secure, and maintainable microservices-based system. The system is designed to support future growth and changes, ensuring long-term viability and success.