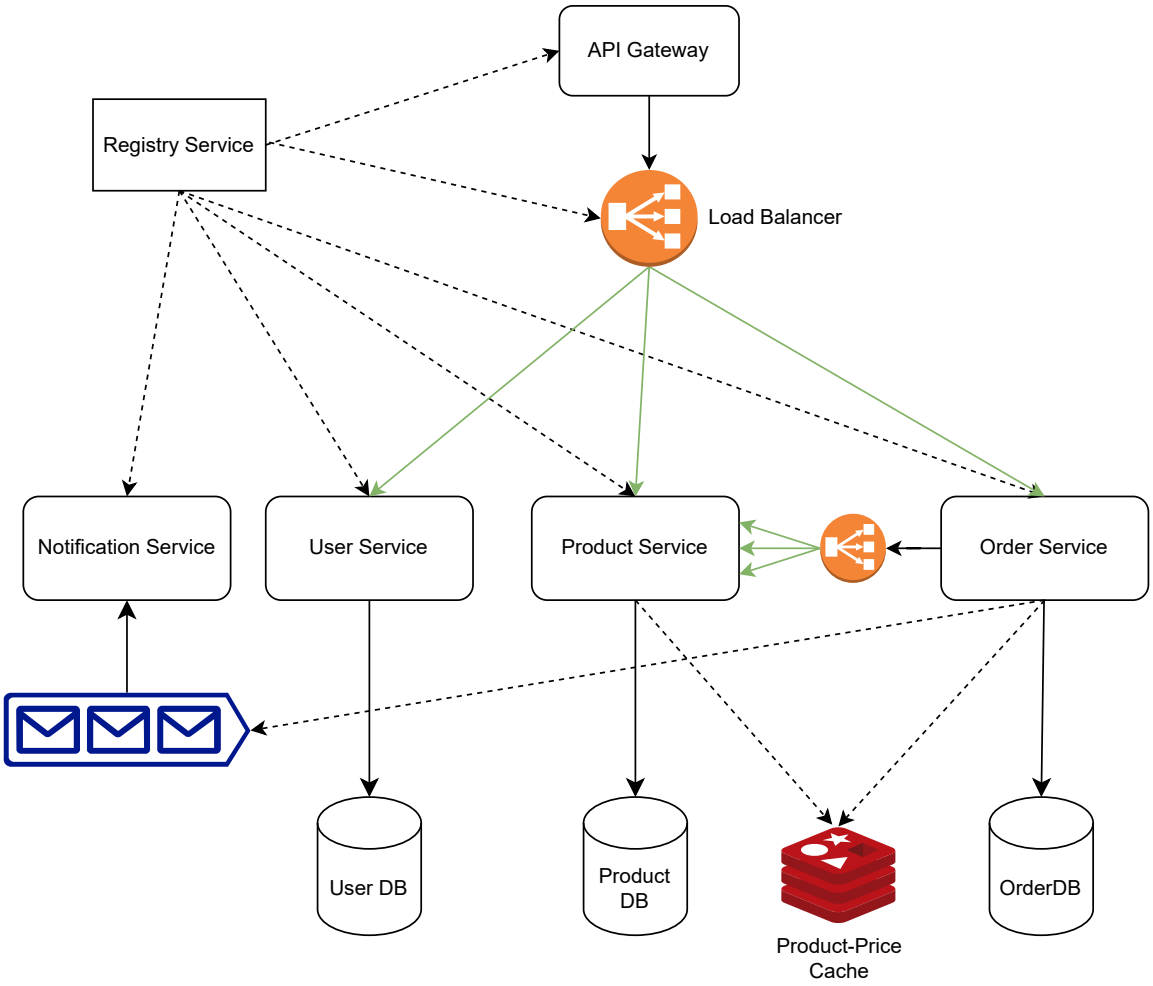
Microservice Based E-commerce System Architecture

Ahmet Karapinar
ahmetkarapinarr00@gmail.com
January 14, 2025


API Gateway
Registry Service
Load Balancer
Notification Service
User Service
Product Service
Order Service
User DB
Product DB
Product-Price Cache
OrderDB

1 Why a Microservice-Based Approach for E-Commerce?

A microservice-based approach is particularly well-suited for e-commerce platforms due to its ability to handle the diverse and dynamic nature of e-commerce operations. E-commerce systems often consist of distinct functionalities like user management, product cataloging, order processing, payment handling, and notifications. By breaking these functionalities into independent microservices, each service can focus on a specific domain, ensuring better organization and maintainability.

This architecture enables scalability, allowing high-demand services like the Product or Order Service to scale independently during peak times, such as flash sales or holiday events. It also enhances fault isolation, ensuring that issues in one service, such as a payment failure, don't disrupt other services like product browsing or order management.

Microservices align naturally with the distributed and modular requirements of e-commerce, as they allow each service to use the most suitable technology stack.

2 Rationale Behind the Design Choices

2.1 API Gateway Implementation

The Gateway is implemented using Spring Cloud Gateway to provide a **centralized entry point** for all incoming client requests. It ensures **centralized authentication and authorization**, enhancing security and simplifying token validation across multiple services. Additionally, the Gateway plays a critical role in **logging and observability**, capturing detailed metrics about incoming traffic, request latency, and response statuses, which helps in monitoring and debugging the system. Spring Cloud Gateway inherently supports **load balancing**, which evenly distributes incoming requests among available service instances.

The use of an API Gateway significantly enhances the security, performance, and availability quality attributes. This design choice ensures centralized control over traffic, streamlines authentication and authorization, optimizes request handling, and improves system resilience through load balancing

2.2 Registery Service Implementation

The Registry Service is implemented using Netflix Eureka to enable dynamic service discovery, allowing microservices to locate and communicate with each other seamlessly. Eureka's lightweight and highly available architecture eliminates hardcoded dependencies between services, improving the system's flexibility and scalability as new instances are registered or removed dynamically. This design choice ensures real-time awareness of service availability, enhances fault tolerance, and supports effective load distribution across the architecture.

2.3 Role-based JWT Authentication

JWT authentication is chosen over cookie-based authentication due to its performance and scalability in a stateless microservices architecture. Unlike cookies, which rely on server-side storage and session management, JWTs are self-contained tokens that don't require maintaining session state, making authentication faster and reducing overhead.

Authentication and authorization are managed by Spring Security, with role-based JWT tokens used to embed user claims directly in the token. This design minimizes extra calls to the User Service, significantly improving performance, especially in high-traffic scenarios. However, this approach

involves a trade-off between security and performance, as including roles in tokens could pose risks if the tokens are not properly secured. Overall, JWT authentication aligns well with the decentralized and stateless nature of the system.

2.4 Redis Cache Implementation

A caching mechanism using Redis is implemented to store product-price pairs, which is an excellent fit for caching due to the predictable access patterns and frequent read operations associated with pricing data. In e-commerce systems, product prices are requested repeatedly during operations like order creation, price calculations, and cart updates. These requests often outnumber updates to the prices themselves, making product-price data highly suitable for a cache-first approach.

Caching product-price pairs improves the system's performance and scalability by reducing the dependency on the Product Service and its underlying database for every pricing query. Prices tend to change less frequently compared to the volume of read requests, making it efficient to store this data in a high-speed in-memory data store like Redis. This ensures rapid retrieval of prices, particularly in high-traffic scenarios, such as sales events or checkout processes. Additionally, caching helps offload a significant amount of traffic from the Product Service, freeing it to handle critical tasks like managing stock levels or product updates.

2.5 Load Balancing Between Order Service and Product Service

The communication between the Order Service and Product Service is facilitated through an OpenFeign proxy, which seamlessly integrates with the Registry Server to enable dynamic service discovery and built-in load balancing. OpenFeign ensures that requests are distributed evenly across multiple instances of the Product Service, preventing any single instance from being overloaded. This design not only enhances the performance and reliability of inter-service communication but also ensures high availability, even during periods of high traffic.

2.6 RabbitMQ Message Broker Utilization

The system uses RabbitMQ to enable asynchronous communication between services. Each service produces messages to the queue, while the Notification Service consumes these messages to handle tasks like sending email notifications or alerts.

This design decouples producers (e.g., Order Service) from consumers, improving scalability and fault tolerance. RabbitMQ ensures reliable message delivery through features like persistence and acknowledgments, allowing producers to offload tasks and continue operations without delays. This approach enhances performance and supports a modular, event-driven architecture. The current implementation focuses on order status notifications, where the Notification Service processes messages related to order confirmations or payment updates. However, this architectural design is highly extensible and capable of supporting additional mechanisms in the future.

3 Functional Requirements

API Gateway

The API Gateway acts as a single entry point for all client requests. It routes incoming requests to appropriate backend services such as the User Service, Product Service, Order Service, and Notification Service. Additionally, it provides load balancing to evenly distribute requests across multiple instances of services. The API Gateway implements JWT authentication to validate tokens, extract user roles, and forward them to downstream services, while rejecting unauthorized or invalid tokens.

Registry Service

The Registry Service maintains a dynamic registry of all service instances, enabling efficient service discovery. It supports scalability by registering new instances and deregistering unavailable ones to ensure system health and reliability.

User Service

The User Service provides user management functionalities such as registering new users with unique email addresses, authenticating users with secure credentials, and managing user roles. It also allows users to view and update their profiles, including name, email, and address. Security is ensured through JWT-based authentication, which includes issuing tokens with user claims like user ID, role, and email. Additionally, role-based access control is implemented to restrict access to specific endpoints.

Product Service

The Product Service handles product management tasks, such as adding, updating, and deleting products (restricted to admins), retrieving product details by ID, and fetching products by category. It also supports product search using filters like price range and availability. The service manages product pricing by maintaining and updating prices and provides cached product prices to improve performance. Furthermore, it handles stock management by updating stock levels after successful order processing and checking stock availability for requested quantities.

Order Service

The Order Service manages the user's cart by allowing products to be added or removed with specific quantities and retrieving the current cart for the authenticated user. During order processing, the service verifies stock availability, processes payment, and creates an order if the payment is successful. It then deducts purchased quantities from the product stock and sends an order confirmation notification to the user. The service also provides functionalities to retrieve order details by order ID and update the status of an order (e.g., IN_PROGRESS, COMPLETED).

Notification Service

The Notification Service handles email notifications by sending order confirmations to users. It integrates with the message queue to process notifications asynchronously and ensures reliable delivery of messages through efficient queue management.