Project Report: E-commerce API Gateway

Candidate: Budi Santoso

Date: 30 September 2025

1. PROJECT OBJECTIVE

The main objective of this project was to develop a secure and scalable API Gateway as a single entry point for a microservices-based e-commerce platform. The gateway is responsible for routing requests to the appropriate downstream services (e.g., User Service, Product Service, Order Service), handling user authentication, and enforcing security policies.

2. TECHNOLOGIES USED

- **Language/Runtime:** Node.js with TypeScript
- **Framework:** Express.js
- **Authentication:** JSON Web Tokens (JWT)
- **Database:** (For user data) PostgreSQL
- **Containerization:** Docker

3. ARCHITECTURE OVERVIEW

The API Gateway sits between the client applications (web/mobile) and the backend microservices. This approach decouples the client from the internal service architecture.

1. Client sends an HTTP request to the gateway.

2. A middleware checks for a valid JWT in the Authorization header for protected routes.

3. Based on the request path (e.g., '/api/products'), the gateway proxies the request to the corresponding internal microservice (e.g., http://product-service:3001).

4. The response from the microservice is then returned to the client through the gateway.

4. KEY FEATURES IMPLEMENTED

- **User Authentication:** Implemented login and registration endpoints that generate JWTs upon successful authentication.
- **Protected Routes:** Created a middleware to verify JWTs and protect routes that require authentication.
- **Request Routing:** Set up a dynamic routing mechanism to forward requests to different microservices.
- **Input Validation:** Used 'express-validator' library to validate and sanitize incoming request bodies to prevent common vulnerabilities like injection attacks.

5. CODE QUALITY & BEST PRACTICES

The project was developed with a focus on maintainability. I used ESLint and Prettier for consistent code formatting and followed the Airbnb style guide. The code is structured by features (e.g., auth, routes, services) to ensure a clear separation of concerns. All configurations are managed through environment variables.

6. ERROR HANDLING

A centralized error handling middleware was implemented. It catches all errors thrown within the application, logs them, and sends a standardized JSON error response to the client with an appropriate HTTP status code. This prevents stack traces from leaking to the end-user and provides a consistent error format.

7. SETUP & INSTALLATION

The project is fully containerized with Docker for easy setup.

1. Create a `.env` file from the provided `.env.example`.
2. Run `docker-compose up --build`.
3. The gateway will be available at `http://localhost:3000`.