Ticketing System Architecture Documentation

Table of Contents

1. Architecture Overview

This Ticketing System follows a Domain-Driven Design (DDD) approach, organizing the codebase into specific layers, each responsible for a particular part of the application logic.

- Domain Layer: Represents core business concepts and rules.
- Application Layer: Defines application behavior and orchestrates domain logic.
- Infrastructure Layer: Implements technical services such as data persistence and event handling.
- Interfaces Layer: Exposes API endpoints to external clients.

2. Core Components and Workflows

This section describes the main workflows and use cases:

2.1 Creating a Ticket:

- API Request -> CommandHandler -> TicketService -> Repository (SQL Server) -> Event Publisher (RabbitMQ) -> Event Consumer (Elasticsearch).

2.2 Retrieving Tickets:

- API Request -> QueryHandler -> TicketService -> Repository (Elasticsearch) -> API Response.

3. Infrastructure and Deployment

The application is containerized using Docker, with SQL Server, Elasticsearch, and RabbitMQ managed via Docker Compose.

- docker-compose.yml: Defines multi-container setup and service dependencies.
- Environment Variables: Configuration for external services is managed via environment variables.

4. Error Handling and Logging

Each handler and repository method includes error handling to provide meaningful responses. The code includes logs for key events to aid in debugging and monitoring.

- Error Handling: Ensures meaningful API responses on failure.
- Logging: Logs critical steps such as database connections, event publishing, and query execution.

5. Swagger Documentation

The API is documented with Swagger annotations in the code, providing an interactive API documentation interface for developers to test each endpoint easily.

- Swagger UI: Provides interactive API documentation.
- Annotations: Each endpoint is documented within the code, specifying parameters and responses.

6. Scalability and Extensibility

The system's architecture allows for scalability and easy extension:

- Scalability: Using Docker, each service can be scaled independently.
- Extensibility: The DDD structure makes it easy to add new features, entities, or services with minimal changes.