# Detailed Design Document: Intelligent Hotel Management Application

## 1. Introduction and Justification

### 1.1. Project Overview

This document outlines the detailed design for an intelligent hotel management application. The goal is to develop a modern web application that leverages a microservices architecture and contemporary full-stack and DevOps technologies. This solution aims to provide hotel establishments with an efficient system for managing reservations, customers, rooms, and ancillary services. It will feature an intuitive user interface and a robust, scalable backend.

### 1.2. Justification: Digitalization in the Hospitality Sector

The hospitality sector is undergoing a significant digital transformation. Traditional, monolithic hotel management systems often struggle with scalability, flexibility, and integration with new technologies. The increasing demand for online booking, personalized guest experiences, and efficient operational management necessitates a more agile and resilient technological infrastructure.

Our proposed microservices-based application addresses these challenges by offering a modular, scalable, and independently deployable system. This approach allows for easier maintenance, faster development cycles, and better fault isolation. Furthermore, the use of modern technologies like NestJS, Angular, Kafka, Docker, and Kubernetes ensures the application is future-proof, highly performant, and capable of handling a growing user base and evolving business requirements. The integration of a message broker like Kafka facilitates seamless communication between different services, enabling real-time updates and notifications, which are crucial for a dynamic environment like hotel management. This project aligns with the current trends in digitalization, providing a competitive edge to hotels adopting such advanced solutions.

## 2. Architecture Overview

The application is designed around a microservices architecture, where each core business capability is encapsulated within an independent service. This approach

promotes modularity, scalability, and resilience. Communication between services is primarily asynchronous via a message broker (Kafka) and synchronous via GraphQL APIs.

## 2.1. High-Level Architecture Diagram

(Refer to `uml_architecture_diagram.png` for a visual representation of the system architecture.)

## 2.2. Microservices Breakdown

### 2.2.1. Authentication Service

- **Purpose:** Handles user registration, login, session management, and role-based access control (admin/client).
- **Technologies:** NestJS, MongoDB (for user data), Redis (for session caching), JWT for token generation and validation.
- **API:** Exposes GraphQL endpoints for authentication operations.
- **Interactions:** Communicates with other services via Kafka for user-related events (e.g., new user registration).

### 2.2.2. Reservation Service

- **Purpose:** Manages the lifecycle of room reservations, including creation, modification, cancellation, and simulated payment processing.
- **Technologies:** NestJS, MongoDB (for reservation data).
- **API:** Exposes GraphQL endpoints for reservation operations.
- **Interactions:**
  - Communicates with the Room Service to check room availability.
  - Interacts with a mock payment gateway for simulated payment processing.
  - Publishes reservation-related events to Kafka (e.g., `reservation_created`, `reservation_cancelled`).

### 2.2.3. Room Management Service

- **Purpose:** Manages hotel room inventory, including adding/modifying rooms, setting prices, and tracking availability.
- **Technologies:** NestJS, MongoDB (for room data).
- **API:** Exposes GraphQL endpoints for room management operations.
- **Interactions:**
  - Subscribes to Kafka topics for events that might affect room availability (e.g., `reservation_created`, `reservation_cancelled`).

    - Provides room availability information to the Reservation Service.

#### 2.2.4. Other Potential Services (Future Expansion)

- **Notification Service:** Handles sending notifications to clients and administrators (e.g., reservation confirmations, check-in reminders) via email, SMS, or in-app messages.
- **Billing Service:** Manages invoicing and actual payment processing (if integrated with real payment gateways).
- **Client Management Service:** Dedicated service for comprehensive client profile management beyond basic authentication.

### 2.3. Data Management

- **Database:** MongoDB is chosen as the primary database for its flexibility and scalability, suitable for handling diverse data structures across different microservices. Each microservice will have its own dedicated database or collection within MongoDB to maintain data independence.
- **Caching:** Redis is utilized for caching frequently accessed data, such as user sessions and room availability, to improve application performance and reduce database load.

### 2.4. Message Broker (Kafka)

Kafka serves as the central nervous system for asynchronous communication between microservices. It enables:

- **Event-Driven Architecture:** Services can publish events (e.g., a new reservation) and other services can subscribe to these events to react accordingly (e.g., updating room availability).
- **Decoupling:** Services are loosely coupled, meaning they don't need direct knowledge of each other's existence, enhancing system resilience and maintainability.
- **Scalability:** Kafka's distributed nature allows for high-throughput and fault-tolerant message processing.

### 2.5. API Gateway (GraphQL)

Each microservice will expose its own GraphQL endpoint. This provides flexibility and allows for efficient data fetching by clients, as they can request exactly what they need. The frontend application will interact directly with these individual GraphQL endpoints.

# 3. Technical Stack

## 3.1. Backend Technologies

- **Framework:** NestJS is a progressive Node.js framework for building efficient, reliable, and scalable server-side applications. It uses TypeScript and combines elements of OOP, FP, and FRP, providing a robust architecture out of the box.
- **API:** GraphQL is chosen for its efficiency in data loading, allowing clients to request exactly the data they need, reducing over-fetching and under-fetching. Each microservice will expose its own GraphQL endpoint.
- **Database:** MongoDB, a NoSQL database, offers high performance, high availability, and automatic scaling. Its document-oriented model is well-suited for the flexible data structures often found in microservices.
- **Cache:** Redis is an in-memory data structure store, used as a database, cache, and message broker. It will be used for user sessions and frequently accessed data like room availability to ensure low-latency responses.
- **Message Broker:** Apache Kafka is a distributed streaming platform capable of handling trillions of events a day. It will facilitate asynchronous, real-time communication between microservices, ensuring high throughput and fault tolerance.
- **Authentication:** JSON Web Tokens (JWT) will be used for secure authentication and authorization. Tokens will be transmitted via cookies, and roles (admin/client) will be embedded within the tokens to manage access control.

## 3.2. Frontend Technologies

- **Framework:** Angular is a platform and framework for building single-page client applications using HTML and TypeScript. Its comprehensive features and strong community support make it ideal for building complex, enterprise-grade applications.
- **UI Kit:** Angular Material is a UI component library that implements Material Design in Angular. It provides a rich set of pre-built, high-quality UI components that ensure a consistent and modern look and feel across the application, while also being responsive.
- **Functionalities:** The frontend will include a responsive interface, login/signup pages, room consultation and reservation functionalities, an administrator dashboard, and robust user state management (sessions, feedbacks, loaders).

### 3.3. DevOps & Deployment

- **Versioning:** GitHub will be used for source code management, with a clear branching strategy (e.g., Git Flow or GitHub Flow) to manage development, features, and releases.
- **CI/CD:** GitHub Actions will automate the Continuous Integration and Continuous Deployment pipeline. This includes linting, running tests, building Docker images, and deploying services to the Kubernetes cluster.
- **Containerization:** Docker will be used to containerize all microservices and the frontend application. This ensures consistency across different environments (development, testing, production) and simplifies deployment.
- **Local Testing:** Docker Compose will be used for local development and testing, allowing developers to spin up all services and their dependencies with a single command.
- **Orchestration:** Kubernetes (K8s) will be the container orchestration platform, managing the deployment, scaling, and management of containerized applications. This will involve deploying pods, services, and ingress controllers for external access.
- **Monitoring:** Prometheus and Grafana will be used for monitoring the application's health and performance. Prometheus will collect metrics from services, and Grafana will visualize these metrics through dashboards, providing insights into system behavior and potential issues.
- **Hosting:** AWS (Amazon Web Services) will be the cloud provider. Depending on the scale and specific requirements, either EC2 instances (for simpler deployments) or EKS (Elastic Kubernetes Service) for managed Kubernetes clusters will be utilized.

## 4. UML Diagrams

### 4.1. Use Case Diagram

The Use Case Diagram illustrates the main functionalities of the system and the interactions between the actors (Client and Administrator) and the system. It provides a high-level view of what the system does.

(Refer to `uml_use_case_diagram.png` for the visual representation.)

### 4.2. Class Diagram

The Class Diagram depicts the static structure of the system, showing the classes, their attributes, methods, and relationships. This diagram focuses on the core entities within the hotel management system.

(Refer to `uml_class_diagram.png` for the visual representation.)

## 4.3. Sequence Diagrams

Sequence Diagrams illustrate the interactions between objects in a sequential order, focusing on the flow of messages between them. Key scenarios such as reservation, cancellation, and authentication are detailed.

### 4.3.1. Reservation Sequence Diagram

(Refer to `uml_sequence_reservation.png` for the visual representation.)

### 4.3.2. Cancellation Sequence Diagram

(Refer to `uml_sequence_cancellation.png` for the visual representation.)

### 4.3.3. Authentication Sequence Diagram

(Refer to `uml_sequence_authentication.png` for the visual representation.)