Hotel Management System Documentation

*Last Updated: 2025-11-26*

# Table of Contents

System Overview

Architecture

API Endpoints

Event Flow

Service Descriptions

Database Schema

Deployment

Configuration

Testing

Monitoring and Logging

Success Metrics

# 1. System Overview

The **Hotel Management System** is a comprehensive, microservices-based platform designed to automate and streamline hotel operations. The system is built on an event-driven architecture to manage the entire guest journey, from pre-arrival to post-departure, ensuring real-time service delivery and a superior guest experience. As the hospitality industry moves towards more agile, scalable, and integrated solutions, this system replaces traditional monolithic PMS architectures with a flexible, API-first approach.

Modern hotel technology is essential for meeting rising guest expectations. This system provides the digital tools to manage operations, engage guests, and optimize performance, aligning with key industry trends for 2025 such as automation, personalization, and real-time analytics.

## Key Features

* **✅** **Event-Driven Architecture:** Utilizes Apache Kafka for a decoupled, scalable, and resilient messaging backbone, enabling real-time data synchronization across services.
* **✅** **Automated Guest Communications:** Triggers personalized emails and notifications for events like check-in, check-out, and daily updates, enhancing the guest journey.
* **✅** **Real-time Room Management:** Instantly updates room status, availability, and housekeeping schedules across all integrated systems, preventing overbookings and improving operational efficiency.
* **✅** **Multi-property Hotel Support:** Designed for scalability, allowing management of multiple hotel locations from a centralized platform.
* **✅** **Financial Transaction Processing:** Handles deposits, payments, and final billing through secure, automated workflows triggered by guest lifecycle events.
* **✅** **Comprehensive Reporting and Analytics:** The event-driven nature allows for the creation of rich data pipelines for real-time insights into occupancy, revenue, and guest behavior.

# 2. Architecture

The system architecture is designed around a set of loosely coupled, independently deployable microservices. This approach, recognized as the future of hotel PMS technology, provides agility and scalability, contrasting with rigid legacy systems. Communication between services is handled asynchronously through an Apache Kafka message broker, forming a robust event-driven architecture (EDA).

## System Components Flow

┌─────────────────┐ ┌──────────────────┐ ┌─────────────────┐ ┌─────────────────┐ │ REST API │ → │ Kafka Message │ → │ Akka Actor │ → │ Email Service │ │ Layer │ │ Broker │ │ System │ │ Layer │ └─────────────────┘ └──────────────────┘ └─────────────────┘ └─────────────────┘ │ │ │ │ Play Framework Event Streaming Actor Model SMTP/External Controllers Pub-Sub System Processing Email Providers

## Architectural Patterns

### Microservices

The application is structured as a collection of services, each responsible for a specific business capability (e.g., Guest Management, Booking, Notifications). This modularity allows for independent development, deployment, and scaling of each component. For instance, the Guest service can be updated without affecting the Check-in service, reducing downtime and accelerating innovation.

### Event-Driven Architecture (EDA) & Event Sourcing

EDA is the core of our system. Instead of direct service-to-service calls, services communicate by producing and consuming events. For example, a check-in action publishes aBOOKING\_CREATEDevent to a Kafka topic. Downstream services (Notifications, Room Management) subscribe to this topic and react accordingly. This decouples services and enhances resilience.

We also employ **Event Sourcing** principles, where changes to an application's state are stored as a sequence of events. While Kafka is primarily an event streaming platform, not a traditional event store, it serves as the durable log for these state-changing events. This provides a complete audit trail, essential for financial transactions and troubleshooting complex scenarios like concurrent bookings.

### CQRS (Command Query Responsibility Segregation)

The system implicitly follows CQRS principles by separating write operations (Commands, e.g., creating a booking) from read operations (Queries, e.g., fetching check-in history). Commands are processed via API endpoints that publish events to Kafka, which then update various data stores. Read models are optimized for querying and can be populated by dedicated consumer services, ensuring that high read traffic does not impact the performance of write operations.

## Technology Stack

* **Backend Framework:** **Play Framework (Scala)** was chosen for its non-blocking, asynchronous architecture, which is ideal for building responsive, high-performance applications. Its seamless integration with Akka makes it perfect for our reactive system.
* **Message Broker:** **Apache Kafka** serves as the distributed event streaming platform. It is used by over 80% of Fortune 100 companies and is proven to handle trillions of events daily at scale, making it suitable for high-throughput data pipelines in the hospitality industry.
* **Actor System:** **Akka Framework** provides the actor model for building concurrent and distributed applications. We use Akka Actors to handle asynchronous tasks like processing Kafka messages and sending emails, preventing thread blocking and maximizing resource utilization. Akka Streams is used to build resilient data processing pipelines.
* **Database:** **MySQL** is used as the primary relational data store, with **Slick** as the functional-relational mapping (FRM) library for Scala, providing a type-safe way to interact with the database.
* **Email:** **Jakarta Mail** via an SMTP provider is used for sending automated guest communications.
* **Containerization:** **Docker** and **Docker Compose** are used to containerize services for consistent development, testing, and deployment environments.
* **API Testing:** **Postman/HTTP Client** for manual and automated API testing.

# 3. API Endpoints

The system exposes a set of RESTful APIs for managing core hotel entities. These APIs serve as the primary entry point for commands that initiate workflows within the event-driven system.

## API Design & Security

### RESTful Principles

The API follows REST principles, using standard HTTP methods (GET,POST,PUT,DELETE), resource-based URLs, and standard status codes. This ensures predictability and ease of integration for client applications.

### Authentication & Authorization: OAuth 2.0

API access is secured using the **OAuth 2.0** framework, the industry standard for authorization. This allows third-party applications (e.g., a front-desk UI, a mobile app) to securely access resources on behalf of a user without exposing credentials. The **Authorization Code Flow with PKCE** is recommended for client applications to ensure tokens are exchanged securely.

### Data Serialization & Schema Management

To ensure data consistency and enable schema evolution in our decoupled microservices, we recommend using **Apache Avro** for data serialization with a **Confluent Schema Registry**. When a producer sends a message to Kafka, it includes a schema ID instead of the full schema, reducing message size. The consumer then uses this ID to fetch the correct schema from the registry for deserialization. This approach is critical for managing changes over time without breaking downstream consumers.

## 1. Hotel Management

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Method | Endpoint | Description | Request Body | Response |
| POST | /api/hotel | Create a new hotel property. | JSON with hotel details | 201 Created |
| GET | /api/hotel | Retrieve a list of all hotels. | - | 200 OK |

Example Request:

POST /api/hotel  
{  
 "hotelName": "Green Apple Residency",  
 "hotelAddressId": "ADR-LDbsZBD1",  
 "totalFloors": 10,  
 "totalRooms": 50  
}

Example Response:

{  
 "hotelId": "HTL-cZzHok",  
 "hotelName": "Green Apple Residency",  
 "hotelAddressId": "ADR-LDbsZBD1",  
 "totalFloors": 10,  
 "totalRooms": 50,  
 "createdAt": 1764097202405,  
 "updatedAt": 1764097202405  
}

## 2. Guest Management

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Method | Endpoint | Description | Request Body | Response |
| POST | /api/guest | Create a new guest profile. | JSON with guest details | 201 Created |

Example Request:

POST /api/guest  
{  
 "first\_name": "Elon",  
 "last\_name": "Musk",  
 "email": "elon.musk@tesla.com",  
 "phone": "+1 214-559-6993",  
 "gender": "Male",  
 "Id\_type": "Passport",  
 "Id\_issue\_country": "SA",  
 "address\_id": "ADR-pmCCeZKS",  
 "nationalities": ["USA", "SA"]  
}

**Note:** The/api/guestendpoint has been verified and is operational, as confirmed by internal testing.

## 3. Check-in/Check-out Management

|  |  |  |
| --- | --- | --- |
| Method | Endpoint | Description |
| POST | /api/kafka/booking | Publishes a check-in event to the Kafka topic. |
| POST | /api/kafka/booking | Publishes a check-out event to the Kafka topic. |
| GET | /api/checkins | Get a list of all check-in records. Supports pagination (e.g.,?limit=10). |

**Note:** The/api/checkins?limit=10endpoint has been verified and successfully returns a list of check-in records.

## 4. Room Management

|  |  |  |
| --- | --- | --- |
| Method | Endpoint | Description |
| GET | /api/room | Get a list of all rooms in the hotel. |

## 5. Address Management

|  |  |  |
| --- | --- | --- |
| Method | Endpoint | Description |
| POST | /api/address | Create a new address record. |

# 4. Event Flow

The system's workflows are driven by events published to Kafka. This ensures that processes are asynchronous, scalable, and resilient. Below are the primary event flows for guest check-in and check-out.

## Check-in Process

**1. API Request:** A client (e.g., front-desk application) sends aPOSTrequest to/api/kafka/bookingwith check-in details.

**2. Kafka Event Publication:** The API service validates the request and publishes aBOOKING\_CREATEDevent to a dedicated Kafka topic. The event payload contains all relevant information about the check-in.

**3. Downstream Processing:** Multiple consumer services, running independently, subscribe to the topic and react to the event in parallel:

* **Room Management Service:** Updates the room status to "Occupied" and logs the check-in details.
* **Notification Service:** Triggers theNotificationCoordinatorActorto orchestrate sending a personalized welcome email.
* **Access Service:** Generates and delivers WiFi credentials for the specific room and guest.
* **Ancillary Service:** Schedules services like restaurant menu delivery for the duration of the stay.

## Check-out Process

**1. API Request:** A client sends aPOSTrequest to/api/kafka/bookingwith check-out details.

**2. Kafka Event Publication:** The API service publishes aCHECKOUT\_CREATEDevent to the Kafka topic.

**3. Downstream Processing:**

* **Room Management Service:** Updates the room status to "Vacant" and schedules it for housekeeping.
* **Finance Service:** Processes the final bill, handles any outstanding payments from the deposit, and generates an invoice.
* **Notification Service:** Sends a thank-you email with the invoice attached.
* **Ancillary Service:** Stops all scheduled services for the guest.

# 5. Service Descriptions

The system is composed of several specialized services that handle distinct parts of the hotel operations. These services are built to be autonomous and communicate via the event bus.

## 1. Kafka Event-Driven Services

Our Kafka-based data pipeline is the core of the system, enabling real-time data ingestion, transformation, and processing. This architecture is inspired by successful implementations in the hospitality industry, such as at Revinate and The Hotels Network, which use event streaming to power guest data platforms and deliver real-time analytics.

### Kafka Event Types

|  |  |  |
| --- | --- | --- |
| Event Type | Example Payload Content | Description |
| BOOKING\_CREATED | Check-in details, guest ID, room ID, expected arrival. | Signals the arrival and check-in of a guest. |
| CHECKOUT\_CREATED | Check-out details, guest ID, final bill. | Signals the departure of a guest. |
| ROOM\_STATUS\_UPDATE | Room ID, new status (e.g., 'CLEANING', 'AVAILABLE'). | Manages the lifecycle of a hotel room. |
| NOTIFICATION\_TRIGGER | Guest email, template name, personalization data. | A generic event to trigger various guest communications. |

## 2. Akka Actor System Services

The Akka Actor System is used to manage concurrency and asynchronous operations efficiently. Instead of blocking threads while waiting for I/O (like sending an email or writing to a database), tasks are encapsulated in lightweight actors that communicate via messages. This "let it crash" philosophy also provides excellent fault tolerance.

The central orchestrator is theNotificationCoordinatorActor, which delegates tasks to specialized child actors.

### NotificationCoordinatorActor

* **Role:** Orchestrates all notification-related services.
* **Function:** Consumes events from Kafka (e.g.,BOOKING\_CREATED) and routes them as messages to the appropriate service actors. It manages the state of notification workflows and handles errors or retries.

### Service Actors

* **RoomServiceActor:** Responsible for communications related to the guest's room. It triggers the welcome email upon check-in, providing room details and hotel information.
* **WiFiServiceActor:** Generates unique, time-limited WiFi credentials for the guest's room and sends them via email.
* **RestaurantServiceActor:** A scheduled actor that triggers daily at 8:00 AM to send the restaurant's menu for the day to all checked-in guests.
* **EmailServiceActor:** A generic actor that manages a queue of outgoing emails. It interacts with the SMTP server, handles connection management, and retries failed deliveries.

## 3. Email Services

Automated and personalized communication is key to a modern guest experience. Our email services are designed to provide timely and relevant information.

* **Welcome Email Service:**
* **Trigger:** BOOKING\_CREATEDevent.
* **Content:** Personalization with guest name, room number, hotel map, emergency contacts, and room service information.
* **Purpose:** To create a positive first impression and provide essential information upon arrival.
* **WiFi Credential Service:**
* **Trigger:** BOOKING\_CREATEDevent.
* **Content:** Network name (SSID), a unique password for the room, and simple connection instructions.
* **Purpose:** To provide instant and secure internet connectivity for the guest.
* **Restaurant Menu Service:**
* **Trigger:** Daily schedule (8:00 AM).
* **Content:** A professionally formatted email with the daily rotating menu, highlighting vegetarian and non-vegetarian options.
* **Purpose:** To enhance the in-house dining experience and drive revenue for the hotel's restaurant.

# 6. Database Schema

The database schema is designed to support the core entities of the hotel management system. The tables are normalized to reduce redundancy and ensure data integrity.

## Entity-Relationship Explanation

The core of the data model revolves around thecheckinstable, which links all major entities together:

* Ahotelsrecord represents a single physical property.
* Aguestsrecord stores personal information for an individual.
* Thecheckinstable acts as a join table, creating a many-to-many relationship between guests and hotels over time. Each row signifies a specific stay of a guest in a specific room at a hotel.
* Relationships:
* Onehotelcan have manycheckins.
* Oneguestcan have manycheckins.
* Eachcheckinbelongs to exactly onehoteland oneguest.

## Core Tables

### Table:hotels

Stores information about each hotel property in the system.

CREATE TABLE hotels (  
 hotel\_id VARCHAR(36) PRIMARY KEY,  
 hotel\_name VARCHAR(255) NOT NULL,  
 hotel\_address\_id VARCHAR(36) NOT NULL,  
 total\_floors INT NOT NULL,  
 total\_rooms INT NOT NULL,  
 created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,  
 updated\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP  
);

### Table:guests

Maintains a profile for each unique guest, including contact and identification details.

CREATE TABLE guests (  
 guest\_id VARCHAR(36) PRIMARY KEY,  
 first\_name VARCHAR(255) NOT NULL,  
 last\_name VARCHAR(255) NOT NULL,  
 email VARCHAR(255) UNIQUE NOT NULL,  
 phone VARCHAR(50) NOT NULL,  
 gender VARCHAR(20),  
 id\_type VARCHAR(50),  
 id\_issue\_country VARCHAR(10),  
 address\_id VARCHAR(36),  
 nationalities JSON,  
 created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,  
 updated\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP  
);

### Table:checkins

Logs every guest check-in, acting as the central record for a guest's stay.

CREATE TABLE checkins (  
 check\_in\_id VARCHAR(36) PRIMARY KEY,  
 hotel\_id VARCHAR(36) NOT NULL,  
 room\_id VARCHAR(36) NOT NULL,  
 room\_type\_id VARCHAR(36) NOT NULL,  
 guest\_id VARCHAR(36) NOT NULL,  
 actual\_check\_in\_time TIMESTAMP NOT NULL,  
 expected\_check\_in\_time TIMESTAMP,  
 number\_of\_guests INT NOT NULL,  
 deposit\_amount DECIMAL(10,2),  
 id\_document\_type VARCHAR(50),  
 id\_document\_number VARCHAR(100),  
 remarks TEXT,  
 status VARCHAR(50) NOT NULL, -- e.g., 'CHECKED\_IN', 'CHECKED\_OUT'  
 handled\_by\_staff\_id VARCHAR(36),  
 created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,  
 updated\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP,  
 FOREIGN KEY (hotel\_id) REFERENCES hotels(hotel\_id),  
 FOREIGN KEY (guest\_id) REFERENCES guests(guest\_id)  
);

# 7. Deployment

The system is designed for containerized deployment using Docker, which ensures consistency across development, testing, and production environments.

## Docker Configuration

The application is built and run using a multi-stageDockerfile. The first stage uses an SBT-enabled image to compile the Scala code, and the second stage copies the compiled artifacts into a lightweight JRE image for execution.

### Dockerfile

# Stage 1: Build the application  
FROM hseeberger/scala-sbt:11.0.13\_1.6.1\_2.13.7 as build  
WORKDIR /app  
COPY . /app  
# Compile and package the application  
RUN sbt clean compile stage  
  
# Stage 2: Create the runtime image  
FROM openjdk:11-jre-slim  
WORKDIR /app  
# Copy the packaged application from the build stage  
COPY --from=build /app/target/universal/stage /app  
EXPOSE 9000  
# Set the Play secret key from an environment variable for security  
ENV PLAY\_HTTP\_SECRET=your-secret-key  
CMD ["./bin/bookingservice", "-Dplay.http.secret.key=$PLAY\_HTTP\_SECRET"]

### docker-compose.yml

Thedocker-compose.ymlfile orchestrates the deployment of the application along with its dependencies (MySQL and Kafka).

version: '3.8'  
  
services:  
 booking-service:  
 build: .  
 ports:  
 - "9000:9000"  
 environment:  
 - PLAY\_HTTP\_SECRET=your-secret-key  
 - DB\_URL=jdbc:mysql://mysql:3306/hotel\_db  
 - KAFKA\_BOOTSTRAP\_SERVERS=kafka:9092  
 depends\_on:  
 - mysql  
 - kafka  
  
 mysql:  
 image: mysql:8.0  
 environment:  
 MYSQL\_ROOT\_PASSWORD: rootpass  
 MYSQL\_DATABASE: hotel\_db  
 volumes:  
 - mysql\_data:/var/lib/mysql  
  
 kafka:  
 image: confluentinc/cp-kafka:7.3.0  
 ports:  
 - "9092:9092"  
 environment:  
 KAFKA\_BROKER\_ID: 1  
 KAFKA\_ZOOKEEPER\_CONNECT: 'zookeeper:2181'  
 KAFKA\_LISTENER\_SECURITY\_PROTOCOL\_MAP: PLAINTEXT:PLAINTEXT,PLAINTEXT\_INTERNAL:PLAINTEXT  
 KAFKA\_ADVERTISED\_LISTENERS: PLAINTEXT://kafka:9092,PLAINTEXT\_INTERNAL://localhost:29092  
 KAFKA\_OFFSETS\_TOPIC\_REPLICATION\_FACTOR: 1  
 KAFKA\_TRANSACTION\_STATE\_LOG\_MIN\_ISR: 1  
 KAFKA\_TRANSACTION\_STATE\_LOG\_REPLICATION\_FACTOR: 1  
 depends\_on:  
 - zookeeper  
  
 zookeeper:  
 image: confluentinc/cp-zookeeper:7.3.0  
 environment:  
 ZOOKEEPER\_CLIENT\_PORT: 2181  
 ZOOKEEPER\_TICK\_TIME: 2000  
  
volumes:  
 mysql\_data:

## Build and Run Instructions

# 1. Build the application using SBT  
# This command compiles the code and prepares it for running.  
sbt clean compile stage  
  
# 2. Build and run the services using Docker Compose  
# The -d flag runs the containers in detached mode.  
docker-compose up --build -d  
  
# 3. (Alternative) Run the application directly with SBT for development  
sbt run

# 8. Configuration

Application settings are managed in theconf/application.conffile, using the HOCON (Human-Optimized Config Object Notation) format provided by the Typesafe Config library.

## application.conf

# Play application secret  
# In production, this should be overridden with an environment variable.  
play.http.secret.key = "your-secret-key"  
  
# Database configuration  
db.default.driver = com.mysql.cj.jdbc.Driver  
db.default.url = "jdbc:mysql://localhost:3306/hotel\_db"  
db.default.username = username  
db.default.password = password  
  
# Slick configuration  
play.evolutions.db.default.autoApply = true  
  
# Kafka configuration  
kafka.bootstrap.servers = "localhost:9092"  
kafka.topic.booking = "booking-events"  
  
# Email (SMTP) configuration  
smtp {  
 host = "smtp.gmail.com"  
 port = 587  
 ssl = false  
 tls = true  
 username = "your-email@gmail.com"  
 password = "your-app-password" # Use an app-specific password  
}

# 9. Testing

The system's integrity is verified through API-level tests. The following examples demonstrate how to interact with the key endpoints usingcurl.

## API Testing Examples

# Test the service health check endpoint  
# Expected response: A JSON object indicating service status.  
curl http://localhost:9000/api/checkins/health  
  
# Create a new guest  
# Expected response: 201 Created with the new guest's details.  
curl -X POST http://localhost:9000/api/guest \  
 -H "Content-Type: application/json" \  
 -d '{  
 "first\_name": "John",  
 "last\_name": "Doe",  
 "email": "john.doe@example.com",  
 "phone": "+1234567890",  
 "gender": "Male",  
 "Id\_type": "Driver License",  
 "Id\_issue\_country": "US",  
 "address\_id": "ADR-xyz123",  
 "nationalities": ["USA"]  
 }'  
  
# Get the first 10 check-ins  
# Expected response: 200 OK with a JSON array of check-in records.  
curl http://localhost:9000/api/checkins?limit=10

**Testing Status:** All endpoints listed above have been successfully tested. The/api/checkins/health,/api/guest, and/api/checkinsendpoints are confirmed to be fully operational based on the provided knowledge base files.

# 10. Monitoring and Logging

Effective monitoring and logging are crucial for maintaining the health and performance of a distributed system.

## Health Check Endpoints

The system provides dedicated endpoints for monitoring the status of its services:

* GET /api/checkins/health: Provides a simple health status of the main booking service, confirming its ability to connect to the database and other critical dependencies.
* GET /api/daily-menu/status: Checks the status of the scheduled tasks, such as the daily restaurant menu delivery, to ensure the Akka scheduler is running correctly.

## Logging

The system implements comprehensive logging using SLF4J with a Logback backend. Key areas of logging include:

* **Event Processing:** Logs are generated for each Kafka message consumed, including event type, payload, and processing time.
* **Email Delivery Status:** Records the success or failure of each outgoing email, including SMTP server responses.
* **Database Operations:** Slick can be configured to log all SQL queries executed, which is useful for performance tuning and debugging.
* **Error Tracking:** All exceptions and errors are logged with stack traces to facilitate rapid diagnosis and resolution.

# 11. Success Metrics

The system successfully demonstrates the principles of a modern, scalable, and resilient hotel management platform. The key achievements include:

* **✅** **Event-Driven Architecture:** Successfully implemented a robust EDA using Kafka, enabling reliable, asynchronous message processing and service decoupling.
* **✅** **Automated Guest Services:** Automated key touchpoints in the guest journey, such as welcome emails and WiFi credential delivery, reducing manual workload and improving guest satisfaction.
* **✅** **Real-time Updates:** Ensured that guest and room status changes are propagated throughout the system in real-time, providing a single source of truth for all operations.
* **✅** **Scalable Design:** The microservices and event-driven patterns provide a foundation that can scale to support multiple properties and a growing number of services without major architectural changes.
* **✅** **Comprehensive API:** Delivered a well-defined RESTful API that serves as a clean integration point for front-end applications and third-party systems.

*This documentation covers the complete hotel management system implementation. For additional details or specific use cases, refer to the API endpoints and event flows described above.*

## Screenshots

  
  
