Project Documentation

Project Title	Hotel Reception — Visitor Management System
Author	Onkar Vallal
Document Version	1.0
Date	2025-11-25

1. Project Overview

1.1. Problem Statement

Modern hotel and office reception environments require a reliable, efficient, and auditable system for managing visitors. Traditional manual processes, such as paper logbooks, are prone to errors, cause delays during check-in, result in missed or delayed host notifications, and offer poor traceability for security and compliance purposes. This leads to a suboptimal experience for visitors and operational inefficiencies for the organization.

> *This project aims to solve these challenges by providing a digital solution that facilitates:*
>
> - *Fast visitor check-in and check-out with persistent, auditable logging.*
> - *Automated, real-time host notifications upon visitor arrival.*
> - *Orchestration of optional workflows for providing temporary services (e.g., Wi-Fi access) and alerting security personnel.*
> - *A modular, event-driven microservice architecture that ensures loose coupling, scalability, and resilience.*

1.2. Solution and Approach

The solution is a microservice-based system built on a reactive technology stack. It decouples the primary visitor registration process from subsequent notification and logging tasks using an event-driven architecture.

- A **Play Framework backend** (`visitor-play-backend`) serves as the core API gateway. It exposes REST endpoints for managing visitors, employees, and visit logs. It handles data persistence using the Slick ORM and, upon a successful check-in, produces a JSON event to an Apache Kafka topic.
- An **Akka-based service** (`visitor-akka-service`) acts as an asynchronous event processor. It consumes events from the Kafka topic and uses the Akka actor model to delegate tasks to specialized actors responsible for host notification, IT service provisioning, and security logging.

This approach highlights several key design principles:

- **Decoupling:** Apache Kafka serves as a durable event bus, completely decoupling the front-facing API from the backend processing of side-effects like sending emails or writing security logs. This means the API can respond quickly to the user while the notifications are handled asynchronously.
- **Reactivity and Concurrency:** The use of Akka Typed actors allows for concurrent, isolated, and fault-tolerant handling of different tasks. Each actor (e.g., for host notifications, IT support) operates independently.
- **Scalability:** Both the Play backend and the Akka consumer service can be scaled independently to handle varying loads on the API and event processing pipeline.

2. System Architecture

2.1. High-Level Architecture

The system is composed of two primary services communicating asynchronously via a message broker. The database provides persistent storage for core entities, while the actor system handles business logic triggered by events.

```
[Client/UI]
    ↓
[Play HTTP API] ⟷ [Database (Postgres/MySQL)]
    ↓ (Kafka Producer)
[Kafka Topic: visitor_notifications]
```

```
    ↓
[Akka Consumer Service]
    ↓
[HostActor] [ITActor] [SecurityActor]
    ↓         ↓         ↓
  Email     Email    Log/DB
```

2.2. Component Diagram

The following diagram illustrates the interaction between the main components of the system.

```
+----------------+       +--------------------+       +------------------+
| Reception UI / | ----> | visitor-play-backend| ----> |   Kafka Broker   |
| cURL / Client  |       |   (Play + Slick)   |       | (visitor_notif.) |
+----------------+       +--------------------+       +------------------+
                                                            |
                                                            |
                                                            v
+-----------------------+       +------------------+
|  visitor-akka-service | <---- |  Other Consumers |
|       (Akka)          |       |     (future)     |
+-----------------------+       +------------------+
           |
           v
+-----------------------+
| HostActor, IT,        |
| Security, Coordinator |
+-----------------------+
```

2.3. Component Breakdown

1. **Play Framework Backend (visitor-play-backend):** This service is the synchronous entry point for all client interactions. Its responsibilities include:
    - Exposing a REST API for CRUD operations on visitors and related entities.
    - Validating incoming requests and authenticating users via JWT.
    - Persisting data to a relational database (Postgres/MySQL) using the Slick functional-relational mapping library.
    - Producing events to a Kafka topic upon significant actions, such as a visitor check-in.

2. **Akka Consumer Service (visitor-akka-service):** This is an event-driven, asynchronous service responsible for processing the side-effects of visitor events.
    - Consuming messages from the `visitor_notifications` Kafka topic using Akka Streams.
    - Coordinating different workflows using a type-safe actor system (Akka Typed).
    - Delegating specific tasks like sending email notifications and logging security events to dedicated actors.
3. **Apache Kafka:** The central nervous system of the architecture, acting as a durable and scalable event bus.
    - Provides loose coupling between the API backend and the consumer service.
    - Guarantees reliable, at-least-once message delivery to consumers.
    - Enables future expansion by allowing new, independent consumer services to subscribe to the same event stream.

3. Event Flow Sequence

A typical visitor check-in event flows through the system as follows:

```
User (Reception UI / cURL)
 |
 | POST /api/visitors (create visitor + check-in)
 |------------------------------------------->
Play Backend (visitor-play-backend)
 | 1. Persist Visitor entity to the database.
 | 2. Create a new VisitLog entry with a 'check-in' status.
 | 3. Produce a Kafka message to the `visitor_notifications` topic.
 |------------------------------------------->
Kafka (topic: visitor_notifications)
 | (Message is routed and stored)
 |------------------------------------------->
Akka Consumer Service (visitor-akka-service)
 | 1. Consume the message from the topic via Akka Streams.
 | 2. The ServiceCoordinatorActor parses the event.
 | 3. Dispatch tasks to specialized child actors:
 |    - HostEmployeeActor -> Sends an email notification to the host.
 |    - ITSupportActor    -> Provisions and emails Wi-Fi credentials to the visitor.
 |    - SecurityActor     -> Logs the visitor entry to the console (or a security
database).
```

```
 V
(Side-effects executed: Emails sent, Logs written)
```

4. Technology Stack

Category	Technology	Purpose
Backend Technologies	Scala	Primary programming language, enabling functional and reactive patterns.
	Play Framework	High-performance web framework for building the REST API.
	Akka Typed	Toolkit for building concurrent and distributed applications using the actor model.
	Apache Kafka	Distributed event streaming platform for asynchronous communication.
	Slick ORM	Modern database query and access library for Scala.
	JWT (JSON Web Tokens)	Standard for securing API endpoints.
Data Processing & Integration	Play JSON / Spray JSON	Libraries for JSON serialization and deserialization.
	Akka Streams + Alpakka	Reactive streams implementation for building the Kafka consumer pipeline.
	Google Guice	Dependency injection framework used by Play.

5. Codebase Structure

5.1. Play Backend (visitor-play-backend/)

The backend service follows the standard Play application structure.

```
visitor-play-backend/
├── app/
│   ├── controllers/
│   │   └── VisitorController.scala    # Defines REST endpoints for visitor management.
│   ├── dao/
│   │   └── VisitorDAO.scala           # Data Access Object for database operations.
│   ├── models/
│   │   ├── domain/Visitor.scala       # Case class representing the Visitor domain
model.
│   │   └── json/ModelFormats.scala    # Implicit JSON formatters for serialization.
│   ├── security/
│   │   ├── JwtUtil.scala              # Utilities for creating and validating JWTs.
│   │   └── JwtAuthFilter.scala        # Action filter to protect routes.
│   └── services/
│       └── KafkaProducerService.scala # Service responsible for publishing events to
Kafka.
└── conf/
    ├── application.conf               # Main configuration file.
    └── routes                         # HTTP route definitions.
```

5.2. Akka Service (visitor-akka-service/)

The consumer service is structured around actors and the Kafka consumption pipeline.

```
visitor-akka-service/
└── src/main/scala/
    ├── actors/
    │   ├── ServiceCoordinatorActor.scala # Supervising actor; dispatches events.
    │   ├── HostEmployeeActor.scala       # Handles host email notifications.
    │   ├── ITSupportActor.scala          # Manages IT support workflows (e.g., Wi-Fi).
    │   └── SecurityActor.scala           # Responsible for security logging.
    ├── kafka/
    │   └── KafkaVisitorConsumer.scala    # Implements the Akka Streams Kafka consumer.
    ├── utils/
    │   └── EmailHelper.scala             # Utility for sending emails via SMTP.
```

```
    └── Main.scala                          # Application entry point to bootstrap the
ActorSystem.
```

6. Key Features Implemented

6.1. Scala Language Features

- **Case Classes:** Used extensively for immutable domain models (e.g., `Visitor`, `Employee`, `VisitLog`), providing boilerplate-free data carriers.
- **Futures:** Leveraged for all non-blocking I/O operations, such as database queries with Slick and message production to Kafka, ensuring the application remains responsive under load.
- **Pattern Matching:** Employed for safe data extraction from `Option` types and for routing logic within actors based on message types.
- **Option Types:** Used throughout the codebase to handle the potential absence of values gracefully, avoiding `NullPointerException`s.

6.2. Play Framework Features

- **Asynchronous REST Controllers:** All API endpoints are implemented asynchronously to maximize throughput.
- **JSON Formats:** Automatic JSON serialization and deserialization is configured using Play JSON's typeclass-based approach.
- **Dependency Injection:** The application is wired together using Google Guice, Play's default DI container.
- **Slick Database Integration:** Provides a type-safe, functional API for database interactions.

6.3. Akka Features

- **Typed Actors:** The entire actor system is built with Akka Typed, providing compile-time safety for actor interactions.

- **Actor Hierarchy & Supervision:** The `ServiceCoordinatorActor` acts as a supervisor, creating and managing child worker actors, which isolates failures.
- **Akka Streams:** The Kafka consumer is built as a reactive stream, providing back-pressure and resilient stream processing.
- **Alpakka Kafka Connector:** A specialized Akka Streams connector used for seamless and reliable integration with Apache Kafka.

7. API Specification

7.1. Endpoints

The core API for visitor management includes the following endpoints:

- `POST /api/visitors`: Creates a new visitor and logs their check-in.
- `GET /api/visitors`: Retrieves a list of all registered visitors.
- `GET /api/visitors/:id`: Fetches the details of a specific visitor by their ID.
- `DELETE /api/visitors/:id`: Removes a visitor from the system.

7.2. Sample Request & Response

Below is an example of creating a new visitor via a `curl` command and the expected JSON response from the server.

7.2.1. HTTP Request

```
curl -X POST 'http://localhost:9000/api/visitors' \
-H 'Content-Type: application/json' \
-d '{
  "fullName": "John Doe",
  "email": "john@example.com",
  "phone": "9876543210",
  "purpose": "Meeting",
```

```
  "hostEmployeeId": 1
}'
```

7.2.2. Expected JSON Response

```
{
  "message": "Visitor created & check-in logged",
  "data": {
    "id": 10,
    "fullName": "John Doe",
    "email": "john@example.com",
    "phone": "9876543210",
    "purpose": "Meeting",
    "hostEmployeeId": 1
  }
}
```

7.2.3. Akka Consumer Console Output

Simultaneously, the `visitor-akka-service` console will display logs indicating that the event was consumed and processed by the actors.

```
Kafka Message Received: {"eventType":"VISITOR_CHECK_IN","visitor":{...},"host":{...}}
[INFO] [ServiceCoordinatorActor] - Dispatching VISITOR_CHECK_IN event for visitor: John Doe
[INFO] [HostEmployeeActor] - Sending email to host@example.com about visitor John Doe
[INFO] [ITSupportActor] - Sending WiFi credentials to john@example.com
[INFO] [SecurityActor] - Visitor ENTERED: John Doe (ID: 10)
```

8. Actor System Design

The Akka service employs a supervisor-worker pattern where a central coordinator dispatches tasks to specialized actors.

8.1. ServiceCoordinatorActor

- **Role:** Central event dispatcher and supervisor.
- **Responsibilities:** It is the primary consumer of Kafka messages. It parses the incoming JSON event, determines the event type (e.g., `VISITOR_CHECK_IN`), and spawns or routes the task to the appropriate child actor.

8.2. HostEmployeeActor

- **Role:** Host notification handler.
- **Responsibilities:** Receives a notification task from the coordinator, fetches host details if necessary, and uses the `EmailHelper` utility to send an email alert to the host employee.

8.3. ITSupportActor

- **Role:** IT service provisioning handler.
- **Responsibilities:** Manages automated IT-related workflows. For a new visitor, this typically involves generating temporary Wi-Fi credentials and emailing them directly to the visitor.

8.4. SecurityActor

- **Role:** Security logging and auditing.
- **Responsibilities:** Logs all significant visitor events, such as check-ins and check-outs. In the current implementation, it logs to the console, but it is designed to be extended to persist these logs to a dedicated security database or a log aggregation system.

9. Deployment Plan

9.1. Prerequisites

- A running **Apache Kafka** broker (e.g., at `localhost:9092`).
- A running **Relational Database** (PostgreSQL or MySQL) with connection details configured in the Play application.
- Valid **SMTP server credentials** configured in the Akka service for email notifications to function.

9.2. Deployment Steps

1. **Setup Kafka:** Start Kafka and create the required topic.

   ```
   # Example command to create the topic
   bin/kafka-topics.sh --create --topic visitor_notifications \
     --bootstrap-server localhost:9092 --replication-factor 1 --partitions 1
   ```

2. **Prepare Database:** Ensure the database schema is created. If using a migration tool with Slick, run the migration scripts. Otherwise, manually create the tables required by the DAOs.
3. **Build and Run Play Backend:**

   ```
   cd visitor-play-backend

   # Run in development mode
   sbt run

   # Or, for production, create a distributable package
   sbt dist
   ```

4. **Build and Run Akka Service:**

   ```
   cd visitor-akka-service
   sbt run
   ```

9.3. Production Considerations

- **Managed Services:** Use a managed Kafka service (e.g., Confluent Cloud, AWS MSK) and a managed database service (e.g., AWS RDS) for reliability and scalability.

- **Configuration Management:** Externalize all configurations (database URLs, Kafka brokers, API keys, SMTP credentials) using environment variables or a configuration service. Avoid hardcoding secrets.
- **Security:** Deploy the Play backend behind a reverse proxy like NGINX, enforce HTTPS, and implement rate limiting.
- **Containerization:** Package both services as Docker containers and manage them with an orchestrator like Kubernetes for automated deployment, scaling, and management.
- **Monitoring and Logging:** Integrate with Prometheus/Grafana for metrics collection and the ELK stack (Elasticsearch, Logstash, Kibana) for centralized, structured logging.

10. Testing and Observability

10.1. Testing Strategy

- **Unit Tests (ScalaTest):** Test individual functions and actor behaviors in isolation.
- **Integration Tests (Akka TestKit, Embedded Kafka):** Verify the complete Kafka producer/consumer flow and interactions between actors.
- **API Tests (Play Test):** Write tests to validate the behavior, responses, and status codes of the REST endpoints.
- **End-to-End Tests:** Simulate a full user workflow, from making an API call to verifying that all resulting side-effects (e.g., database entries, logs) occurred correctly.

10.2. Monitoring & Observability

- **Structured Logging:** Implement structured (JSON) logging across both services to enable easy parsing and searching in a centralized logging system.
- **Metrics Collection:** Expose key metrics from the applications, including JVM health, database connection pool status, Kafka consumer lag, and business-level metrics (e.g., number of check-ins per hour).
- **Distributed Tracing:** Implement tracing to follow a single request from the client API call through the Kafka event to the final processing by the Akka actors.

11. Future Enhancements

11.1. Short-term Improvements

- **Externalized Configuration:** Move all environment-specific settings out of the codebase into configuration files or environment variables.
- **Dead Letter Queue (DLQ):** Implement a DLQ for the Kafka consumer to handle messages that repeatedly fail processing, preventing the consumer from getting stuck.
- **Comprehensive Test Coverage:** Increase unit and integration test coverage to improve code quality and reliability.
- **Persistent Security Logs:** Modify the `SecurityActor` to write logs to a dedicated database table or a secure log file instead of just the console.

11.2. Long-term Vision

- **Real-time Dashboard:** Develop a web-based UI to display the status of current visitors in real-time.
- **Mobile Application Integration:** Create a mobile app for hosts to receive push notifications and manage their visitors.
- **Advanced Security:** Integrate with physical access control systems, QR code generation for visitor badges, or biometric verification.
- **Analytics and Reporting:** Build a data pipeline to analyze visitor traffic, peak hours, and other trends for business intelligence.
- **Multi-tenancy:** Adapt the system to support multiple independent organizations or office locations within a single deployment.