LAB 7 REPORT: EVENT-DRIVEN ARCHITECTURE (EDA)

Project Name: Movie Ticket Booking System

Member Name: Nguyễn Đình Quyền - 23010485

Trần Hữu Kiên - 23010258

---

1. OBJECTIVES

The goal of this lab is to implement asynchronous communication between microservices using **RabbitMQ**. Specifically, we decouple the **Booking Service** (Producer) from the **Notification Service** (Consumer).

- **Producer:** Handles booking logic and publishes a `TicketBooked` event.
- **Consumer:** Listens for events and simulates sending a confirmation email.

2. TECHNOLOGY STACK

- **Language:** Node.js (JavaScript)
- **Message Broker:** RabbitMQ
- **Library:** `amqplib` (Node.js client for RabbitMQ)
- **Data Storage:** JSON File System (`bookings.json`)

---

3. IMPLEMENTATION DETAILS

3.1. Producer Implementation (Booking Service)

The Booking Service first persists the booking data to the file system (using the provided `createBooking` logic), and **immediately afterwards** publishes an event to RabbitMQ.

Code Snippet: Booking Logic & Event Publishing *(Based on provided `bookingModel.js` and integrated with RabbitMQ)*

JavaScript
```
const fs = require('fs');
const path = require('path');
const amqp = require('amqplib'); // Library for RabbitMQ
const dbPath = path.join(__dirname, '../../data/bookings.json');
const QUEUE_NAME = 'booking_events';

// 1. Core Business Logic (Your Code)
const createBooking = async (movieId, newSeats) => {
    // ... [Existing logic to read file and check seats] ...
    let allBookings = JSON.parse(fs.readFileSync(dbPath, 'utf8'));

    const newBooking = {
        id: Date.now(),
        movieId: movieId,
        seats: newSeats,
        date: new Date().toISOString()
    };

    allBookings.push(newBooking);
    fs.writeFileSync(dbPath, JSON.stringify(allBookings, null, 2));

    // 2. EDA Integration: Publish Event after successful save
    await publishToQueue(newBooking);

    return newBooking;
};

// 3. RabbitMQ Publisher Logic
async function publishToQueue(bookingData) {
    try {
        const connection = await amqp.connect('amqp://localhost');
        const channel = await connection.createChannel();

        await channel.assertQueue(QUEUE_NAME, { durable: false });

        const message = JSON.stringify(bookingData);
        channel.sendToQueue(QUEUE_NAME, Buffer.from(message));

        console.log(` [x] Event Published: Booking ID ${bookingData.id}`);

        setTimeout(() => { connection.close(); }, 500);
    } catch (error) {
        console.error("RabbitMQ Error:", error);
    }
}

module.exports = { createBooking };
```

3.2. Consumer Implementation (Notification Service)

The Notification Service runs independently. It listens to the `booking_events` queue. When a new booking is received, it simulates a time-consuming email sending process without blocking the Booking Service.

Code Snippet: Notification Worker

JavaScript
```
const amqp = require('amqplib');

const QUEUE_NAME = 'booking_events';

async function startConsumer() {
    try {
        const connection = await amqp.connect('amqp://localhost');
        const channel = await connection.createChannel();

        await channel.assertQueue(QUEUE_NAME, { durable: false });

        console.log(" [*] Notification Service waiting for messages...");

        channel.consume(QUEUE_NAME, (msg) => {
            const booking = JSON.parse(msg.content.toString());
            console.log(` [x] Received Booking ID: ${booking.id}`);

            // Simulate Email Processing Delay (3 seconds)
            setTimeout(() => {
                console.log(` [v] EMAIL SENT to user for Movie ID: 
${booking.movieId}`);
            }, 3000);

        }, { noAck: true });

    } catch (error) {
        console.error("Consumer Error:", error);
    }
}

startConsumer();
```

---

4. EXECUTION AND RESULTS

Decoupling Verification

We executed both the Producer (Booking App) and Consumer (Notification Worker) simultaneously.

Observation:

1. **Booking Service:** When `createBooking` was called, it saved the data and printed `[x] Event Published` immediately. It **did not wait** for 3 seconds. The API response was instant.
2. **Notification Service:** Received the message, waited for the simulated 3-second delay, and then printed `[v] EMAIL SENT`.

---

5. CONCLUSION

By integrating `amqplib` with the existing Node.js booking logic, we successfully demonstrated **Event-Driven Architecture**. The critical path (booking tickets) is now decoupled from the non-critical path (sending emails), ensuring high performance and fault isolation for the Movie Ticket Booking System.