## 1. System Architecture Description

The communication flow implemented in this lab decouples the order placement process from the notification process:

1. **User Action:** A customer places an order via the Order Service.
2. **Producer (Order Service):** Instead of sending an email directly, the service serializes the order details into a JSON object and publishes it to a RabbitMQ queue named order_queue. The service then immediately responds to the user.
3. **Message Queue:** RabbitMQ acts as a buffer, storing the message persistently until a consumer is available.
4. **Consumer (Notification Service):** A separate background process continuously listens to order_queue. Upon receiving a message, it deserializes the data and executes the email sending logic (simulated with a time delay).

## 2. Implementation Details

### 2.1. Order Service (Producer)

The Producer script (producer.php) is responsible for connecting to the RabbitMQ server and dispatching events. Key implementation details include:

- Establishing a connection to localhost on port 5672.
- Declaring a durable queue named order_queue to ensure message persistence.
- Encoding order data (Order ID, Customer Email, Price) into JSON format.
- Publishing the message using basic_publish and immediately closing the connection to free up resources.

### 2.2. Notification Service (Consumer)

The Consumer script (consumer.php) runs as a persistent background worker. Its logic includes:

- maintaining a live connection to RabbitMQ.

- Defining a **callback function** to handle incoming messages.
- Within the callback, a sleep(2) function is used to simulate the latency of communicating with an external email server (e.g., SMTP).
- Acknowledging the message processing to remove it from the queue.

### 3. Test Scenarios and Results

To validate the architecture, we conducted two primary test scenarios using the Command Line Interface (CLI).

#### 3.1. Scenario A: Asynchronous Performance Test

**Objective:** Prove that the User (Producer) does not wait for the Email (Consumer) to finish.

**Execution:**

1. We started the Consumer script in Terminal 1.
2. We ran the Producer script in Terminal 2.

**Result:**

- **Terminal 2 (Producer):** The script finished execution instantly (less than 0.1 seconds), confirming that the order placement is non-blocking.
- **Terminal 1 (Consumer):** Simultaneously, the consumer detected the message. It displayed "Received...", paused for 2 seconds (simulating work), and then displayed "Email sent".
- **Conclusion:** The system successfully demonstrated asynchronous processing. The heavy lifting of sending emails was offloaded from the main request thread.

#### 3.2. Scenario B: Fault Tolerance and Decoupling Test

**Objective:** Ensure that orders are not lost if the Notification Service crashes (Service Unavailability).

**Execution:**

1. We manually terminated the Consumer script (simulating a server crash).

2. While the Consumer was down, we executed the Producer script three times consecutively.
3. We then restarted the Consumer script.

**Result:**

- During the downtime, the Producer script completed successfully without errors. The user experience was not affected by the backend failure.
- Upon restarting the Consumer, the script immediately retrieved all three "backlogged" messages from RabbitMQ and processed them sequentially.
- **Conclusion:** The system demonstrated high reliability. The order_queue successfully buffered the events, preventing data loss during service outages.