Event Notification System

🎯 Objective

Build a **Java-based Event Notification System** that can:

- Accept three types of events via REST API: **EMAIL, SMS, and PUSH notifications.**
- Process each event type **in its own separate queue.**
- Process events asynchronously and in the **order they arrive** (FIFO).
- Notify the client of the final status via a **callback URL.**

✅ Problem Description

Your system must handle **three types of notification events:**

📩 1. Email Event

- **Purpose:** Send an email to a user.
- **Processing Time:** 5 seconds per event.
- **Payload Example:**

```
{
  "recipient": "user@example.com",
  "message": "Welcome to our service!",
  "callbackUrl": "http://client-system.com/api/event-status"
}
```

📱 2. SMS Event

- **Purpose:** Send an SMS to a user.
- **Processing Time:** 3 seconds per event.
- **Payload Example:**

```
{
  "phoneNumber": "+911234567890",
  "message": "Your OTP is 123456",
  "callbackUrl": "http://client-system.com/api/event-status"
```

🔔 3. Push Notification Event

- **Purpose:** Send a push notification to a mobile or web app.
- **Processing Time:** 2 second per event.
- **Payload Example:**

```
{
  "deviceId": "abc-123-xyz",
  "message": "Your order has been shipped!",
```

```
  "callbackUrl": "http://client-system.com/api/event-status"
}
```

🛠️ System Requirement

1. Add New Event API

- **Endpoint:** `POST /api/events`
- **Request Body:**

```
{
  "eventType": "EMAIL",
  "payload": { ... },    // Depends on the event type
  "callbackUrl": "http://client-system.com/api/event-status"
}
```

- **Response Example:**

```
{
  "eventId": "e123",
  "message": "Event accepted for processing."
}
```

---

2. Event Processing Rules

- Each event type should have:
  - **A separate queue.**
  - **A separate processing thread.**
- Events must be processed in the **order they are received** (FIFO) for each queue.
- Processing time per event type:
  - EMAIL: 5 seconds
  - SMS: 3 seconds
  - PUSH: 2 second
- The system should **simulate random failures** (10% of events can fail randomly).

---

3. Callback Notification

- When the event processing is **completed or failed**, the system must automatically send a **POST request to the provided callback URL.**
- Callback request body (Success Example):

```
{
  "eventId": "e123",
  "status": "COMPLETED",
  "eventType": "EMAIL",
  "processedAt": "2025-07-01T12:34:56Z"
}
```

- Callback request body (Failure Example):

```
{
  "eventId": "e123",
  "status": "FAILED",
  "eventType": "EMAIL",
  "errorMessage": "Simulated processing failure",
  "processedAt": "2025-07-01T12:34:56Z"
}
```

---

4. Graceful Shutdown

- On system shutdown (like pressing `Ctrl+C` or sending a kill signal):
  - The system should **stop accepting new events.**
  - All events currently in the queues must finish processing.
  - All threads must shut down cleanly.

5. Dockerization Requirements

- The candidate must provide a **Dockerfile** to build and run the Java application.
- The candidate must provide a **docker-compose.yml** file to start the application.
- The API must be exposed on **port** `8080`**.**
- As an evaluator, I should be able to start the system with:

```
docker compose up
```

- Once started, the API should be available at:

```
http://localhost:8080/api/events
```

🧪 6. Unit Testing Requirements

The project **must include JUnit-based unit tests** to ensure the system is working correctly and can handle failure scenarios.

✅ Minimum Unit Test Coverage:

1. **Core Business Logic:**
   - Event creation: Verify that events are correctly created and added to the correct queue based on the event type.
   - Queue handling: Ensure that each event type is added to the correct queue and is processed in FIFO order.
   - Event processing: Verify that the event is properly processed with the correct simulated delay (3s, 2s, 1s based on type).
2. **Failure Handling:**

- Simulate random failure logic: Test that the system can correctly mark an event as failed and still complete the process.
- Test graceful shutdown: Validate that the system stops accepting new events during shutdown and allows in-progress events to finish.
- Test proper thread termination on shutdown.

3. **API Layer Testing:**
   - Validate that a `POST /api/events` request correctly accepts and parses incoming event payloads.
   - Validate that incorrect requests (missing fields, wrong eventType, invalid payloads) return appropriate HTTP error responses.
   - You may **mock the callback URL HTTP request** to test whether the system is attempting to send the callback correctly.

✅ Specific Testing Scenarios to Cover:

Test Case	Expected Outcome
Valid Event Submission	Event is added to the correct queue, response returns eventId
Invalid Event Type	Return 400 Bad Request
Missing Payload Fields	Return 400 Bad Request
Random Failure Simulation	Event processing occasionally fails as expected
Graceful Shutdown	System stops accepting new events and completes remaining queue
Callback Trigger	System correctly POSTs callback with final status
Thread Shutdown	All processing threads terminate cleanly on shutdown

✅ Evaluation Criteria

Skill	What to Evaluate
Java Basics	Clean class design, enums, interfaces
REST APIs	API contract, input validation, error handling
Concurrency	Multi-threading, thread-safe queues
Async Processing	FIFO logic, separate queue processors
Callback Handling	HTTP request building, robust failure handling
Graceful Shutdown	Clean resource and thread management
Dockerization	Dockerfile correctness, Docker Compose setup
Unit Testing	JUnit coverage for key business logic