General requirements

- Use Docker containerization
- Use RabbitMQ (or similar) messaging queue

1.) Water Quality Monitoring System

Develop an application that simulates monitoring and controlling water quality levels in a reservoir or river system using message queues. This system will involve three separate clients/components for generating, processing, and reporting water quality data.

Component 1: Water Quality Data Generation Client

- Client 1: Connects to the **waterQualityQueue** using a point-to-point connection.
- Function: Sends messages with randomly assigned water quality readings, focusing on parameters such as **pH levels**, every 5 seconds.

Component 2: Water Quality Alert Processor

- Message Processor: Exclusively receives water quality readings from **waterQualityQueue**.
- Function: Monitors for critical water quality issues, such as extremely high turbidity or unsafe pH levels.
- Notification: After receiving 3 consecutive readings indicating critical conditions (e.g., pH below 5.0 or above 9.0), the processor sends a notification to the **waterQualityAlertQueue**, indicating that critical water quality levels have been detected.

Create tests to validate:

- The functionality of sending and receiving messages correctly.
- The correct identification of critical water quality readings.
- The accurate accumulation and dispatch of alerts after detecting critical water quality conditions.

2.) RGB-Colors System

flowchart LR subgraph Client1 [Client 1: Color Generator] direction TB start1([Start]) -->|Every 2s| sendColors{Send Colors} sendColors -->|RED, GREEN, BLUE| ColorQueue([colorQueue]) end subgraph Processor [Message Processor] direction TB ColorQueue --> receiveColors[Receive Colors] receiveColors -->|Count messages per color| checkCount{Check Count} checkCount -->|10 messages of same color?| sendStat checkCount -->|Less than 10| receiveColors sendStat[Send '10 color processed'] --> ColorStatsQueue([colorStatistics Queue]) end subgraph Client3 [Client 3: Statistics Reporter] direction TB ColorStatsQueue --> readStats[Read Statistics] readStats --> displayConsole[Display on Console] end classDef startend fill:#f9f,stroke:#333,stroke-width:2px; class start1 startend; classDef operation fill:#bbf,stroke:#fff,stroke-width:2px; class sendColors,receiveColors,displayConsole operation; classDef decision fill:#fbf,stroke:#333,stroke-width:2px; class checkCount decision; classDef process fill:#ccf,stroke:#333,stroke-width:2px; class readStats,sendStat process;

Create an application consisting of three clients. The first client connects to the 'colorQueue' message queue using point-to-point connection and sends messages with randomly assigned parameters (RED, GREEN, and BLUE), every 2 seconds. Create a message processor that receive messages with the 'RED', 'GREEN', and 'BLUE' parameters exclusively. After receiving every 10 messages of the same color, the processor sends a message to the 'colorStatistics' queue indicating that they have processed 10 messages of a given color.

Create a third client that reads the statistics from the 'colorStatistics' queue and outputs to the console, for example, '10 'RED' messages have been processed'.

Create tests to validate the original functionality

3.) Temperature Alert System

Create an application that simulates temperature monitoring using message queues. This system involves three separate clients for generating, processing, and reporting temperature data.

Component 1: Temperature Generation Client

- Client 1 connects to the **temperatureQueue** using a point-to-point connection.
- Function: Sends messages with randomly assigned temperature readings (in degrees Celsius) within a normal range (-10 to 40 degrees Celsius) every 3 seconds.

Component 2: Temperature Monitoring Processor

- Message Processor: Exclusively receives temperature readings from temperatureQueue.
- Function: Monitors for abnormal temperature readings. A reading is considered abnormal if it is below -5 or above 35 degrees Celsius.
- Notification: After receiving 5 abnormal readings, the processor sends a notification to the **alertQueue**, indicating that 5 abnormal temperature readings have been detected.

Component 3: Alert Reporting Client

- Reads from the **alertQueue**.
- Output: Prints a message to the console, e.g., “5 abnormal temperature readings have been detected.”

Tests to validate:

- The functionality of sending and receiving messages correctly.
- The correct identification of abnormal temperature readings.
- The accurate accumulation and dispatch of alerts after every 5 abnormal readings.

4.) Humidity Control System

Create an application that simulates monitoring and controlling humidity levels within an environment using message queues. This system will involve three separate clients for generating, processing, and reporting humidity data.

Component 1: Humidity Generation Client

- Client 1: Connects to the **humidityQueue** using a point-to-point connection.
- Function: Sends messages with randomly assigned humidity readings (expressed as percentages) within a range of 30% to 90% every 4 seconds.

Component 2: Humidity Monitoring Processor

- Message Processor: Receives humidity readings from **humidityQueue**.
- Function: Monitors for high humidity readings, considered high if they exceed 70%.
- Notification: After receiving 3 consecutive high humidity readings, the processor sends a notification to the **alertQueue**, indicating that high humidity levels have been detected.

Component 3: Alert Reporting Client

- Client 3: Reads from the **alertQueue**.
- Output: Prints a message to the console, e.g., “High humidity alert: 3 consecutive readings over 70%.”

Create tests to validate:

- The functionality of sending and receiving messages correctly.
- The correct identification of high humidity readings.
- The accurate accumulation and dispatch of alerts after detecting high humidity conditions.

5.) Air Quality Monitoring System

Create an application that simulates monitoring and controlling air quality levels within an indoor environment using message queues. This system will involve three separate clients for generating, processing, and reporting air quality data.

Component 1: Air Quality Data Generation Client

- Client 1: Connects to the **airQualityQueue** using a point-to-point connection.
- Function: Sends messages with randomly assigned air quality readings, quantified as Air Quality Index (AQI) values, within a range of 0 (Good) to 300 (Hazardous) every 3 seconds.

Component 2: Air Quality Alert Processor Message Processor: Exclusively receives AQI readings from **airQualityQueue**. Function: Monitors for poor air quality readings, considered poor if they exceed an AQI of 150. Notification: After receiving 2 consecutive readings above the threshold, the processor sends a notification to the **airQualityAlertQueue**, indicating that poor air quality levels have been detected.

Component 3: Alert Reporting Client

- Client 3: Reads from the **airQualityAlertQueue**.
- Output: Prints a message to the console, e.g., “Air quality alert: 2 consecutive readings above 150 AQI.”

Create tests to validate:

- The functionality of sending and receiving messages correctly.
- The correct identification of poor air quality readings.
- The accurate accumulation and dispatch of alerts after detecting poor air quality conditions.

6.) Light Intensity Monitoring System

Create an application that monitors indoor light levels (measured in lux). The system is composed of three separate clients: one for generating data, one for processing it, and one for alert reporting.

Component 1: Light Intensity Generation Client

1. Connection: This client connects to the lightIntensityQueue using a point-to-point messaging model.
2. Function: Every 3 seconds, it sends random light intensity (lux) readings (e.g., between 0 and 2000 lux).

Component 2: Light Intensity Processor

1. Message Processing: This processor exclusively receives messages from the *lightIntensityQueue*.
2. Evaluation: Determines whether the light level is too low. For example, consider values below 100 lux as “dark.”
3. Alert Trigger: If it encounters 3 consecutive readings below 100 lux, it sends an alert message to the lightAlertQueue stating, for example, “Low light alert: 3 consecutive readings below 100 lux.”

Component 3: Alert Reporting Client

1. Consumption: This client listens to the *lightAlertQueue* for alerts.
2. Output: Prints any received alert to the console, for instance: “Low light alert: 3 consecutive readings below 100 lux.”

Create tests to validate:

- Message sending and receiving: Verify that light intensity values are correctly sent and received by the respective clients.
- Low light detection: Check that the processor correctly identifies 3 consecutive values below 100 lux.
- Alert mechanism: Ensure that the alert message is added to the lightAlertQueue and that the reporting client displays it on the console.

7.) Sound Level Alert System

Create an application that monitors sound levels (measured in decibels) in a given environment. It has three clients: one for generating noise level data, one for processing it, and one for reporting alerts.

Component 1: Sound Level Generation Client

1. Connection: This client connects to the soundLevelQueue using a point-to-point messaging model.
2. Function: Every 2 seconds, it sends random decibel readings (e.g., between 30 dB and 120 dB).

Component 2: Sound Level Processor

1. Message Processing: Exclusively receives messages from the *soundLevelQueue*
2. Evaluation: Determines if the sound level is above the threshold (e.g., 80 dB) considered “too loud.”
3. Alert Trigger: After collecting 5 “too loud” readings (either consecutively or within a certain timeframe—depending on your design), the processor sends a message to the *soundAlertQueue*: “High noise alert: 5 high decibel readings detected.”

Component 3: Alert Reporting Client

1. Consumption: Subscribes to the soundAlertQueue
2. Output: Prints to the console, for example: “High noise alert: 5 high decibel readings detected.”

Create tests to validate:

- Message sending and receiving: Ensure decibel readings flow correctly from the generation client to the processor.
- High noise identification: Check that readings above 80 dB are recognized properly.
- Alert message: Verify that an alert is sent to the soundAlertQueue and that the reporting client prints it to the console.

8.) Pressure Monitoring System

Develop an application that monitors fluid or gas pressure in a system. The task involves three clients: one that generates the pressure readings, one that processes them, and one that handles the alerts.

Component 1: Pressure Generation Client

1. Connection: Connects to the *pressureQueue* using a point-to-point messaging model.
2. Function: Every 4 seconds, sends random pressure readings, for example, between 0 and 10 bar.

Component 2: Pressure Alert Processor

1. Message Processing: Exclusively receives messages from the *pressureQueue*.
2. Evaluation: Determines if the pressure reading is dangerously high (e.g., above 8 bar).
3. Alert Trigger: If it detects 2 consecutive readings above 8 bar, the processor sends an alert to the *pressureAlertQueue* stating: “High pressure alert: 2 consecutive readings above 8 bar detected.”

Component 3: Alert Reporting Client

1. Consumption: Reads messages from the *pressureAlertQueue*
2. Output: Prints the alert message to the console, for example: “High pressure alert: 2

consecutive readings above 8 bar detected.”

Create tests to validate:

- Message flow: Verify that pressure readings are correctly sent from the generation client to the processor.
- High pressure detection: Check that the processor accurately identifies readings above 8 bar.
- Consecutive reading logic: Ensure alerts are triggered only when there are 2 consecutive high readings.