Messaging Queues & Async Systems

I. Synchronous vs. Asynchronous

In a system design interview, this is about **User Experience (UX)** and **Resource Management.**

- **Synchronous (Sync):** The "Wait-and-See" approach. The client sends a request and stays on the line until the server finishes *everything*.
- **Asynchronous (Async):** The "I'll get back to you" approach. The client sends a request, the server acknowledges it immediately, and the heavy lifting happens in the background.

Interview Talk Track:

"I prefer Asynchronous communication for non-critical, time-consuming tasks. For example, if a user uploads a profile picture, they shouldn't wait for us to resize it, generate thumbnails, and update the CDN. We should save the image, return a 'Success' message immediately, and let a background worker handle the processing. This keeps the app feeling snappy and responsive."

II. What is a Message Queue (MQ)?

A Message Queue is a form of **asynchronous service-to-service communication**. It acts as a buffer between the service producing the data and the service processing it.

Core Components:

1. **Producer:** The service that creates the message (e.g., the Checkout service).
2. **Broker/Queue:** The storage layer that holds the message (e.g., RabbitMQ or Kafka).
3. **Consumer:** The service that pulls the message and does the work (e.g., the Email service).

Interview Talk Track:

"The primary benefit of a Message Queue is Decoupling. Without a queue, if my 'Order Service' needs the 'Email Service' to be online to work, they are tightly coupled. If the Email Service goes down, the Order Service fails. With a queue, the Order Service just drops a message and moves on. The Email Service can process it whenever it's back online."

III. Advanced Queue Concepts

1. Delivery Guarantees

- **At-most-once:** Message might be lost, but never duplicated. (Fastest)
- **At-least-once:** Message is never lost, but might be delivered twice. (Most common)
- **Exactly-once:** Every message is delivered exactly one time. (Most difficult/expensive to implement).

2. Dead Letter Queue (DLQ)

If a message fails to be processed after multiple retries (perhaps due to a bug or bad data), it is moved to a **DLQ**.

3. Messaging Patterns

- **Point-to-Point:** One producer sends to one specific consumer.
- **Publish-Subscribe (Pub-Sub):** One producer "broadcasts" a message to a **Topic**, and multiple different services (Email, SMS, Analytics) can subscribe to that topic to get a copy.

IV. When NOT to use a Queue

Senior engineers know when *not* to use a tool.

- **Low Latency Requirement:** If you need a result *now* (like checking a password), a queue adds too much delay.
- **Simple Systems:** Don't add a queue if your app only has 10 users; it adds unnecessary complexity.

Practice Interview Questions (Day 9–10)

1. "How do you handle a scenario where the Consumer service crashes while processing a message?"

- **Answer:** We use **Acknowledgements (ACKs)**. The queue won't delete the message until the consumer sends a 'Success' signal. If the consumer crashes, the 'un-acked' message stays in the queue and is eventually handed to another healthy consumer.

2. "What is the difference between RabbitMQ and Kafka?"

- **Answer: RabbitMQ** is a traditional message broker; it deletes messages once they are consumed. **Kafka** is a distributed streaming platform; it keeps messages on a disk for a

set period, allowing you to "replay" them if needed. (Kafka is better for huge data like logs/analytics).

3. "How do you prevent a Queue from growing too large (Backpressure)?"

- **Answer:** We can use **Auto-scaling** to add more consumers when the queue size hits a certain threshold. We can also implement **Rate Limiting** on the producer side to slow down the incoming messages.

4. "If a consumer processes a message twice (At-least-once delivery), how do we prevent duplicate orders?"

- **Answer:** We make the consumer **Idempotent**. We check a unique ID (like `order_id`) in the database before processing. If we see that `order_id` has already been processed, we simply ignore the duplicate message.

5. "What is a 'Poison Pill' message and how does a DLQ help?"

- **Answer:** A 'Poison Pill' is a message that causes the consumer to crash every time it tries to read it (maybe due to a null value). A **DLQ** helps by taking that "poison" message out of the main loop after x no. of failed attempts, so it doesn't block the rest of the queue.