3 Interview Questions on Event-Driven Patterns

Also useful for projects

SAURABH DASHORA

SEP 10, 2024

System Design interviews these days often test a candidate's knowledge of event-driven systems, particularly if you've already been working on them and have mentioned them on your resume.

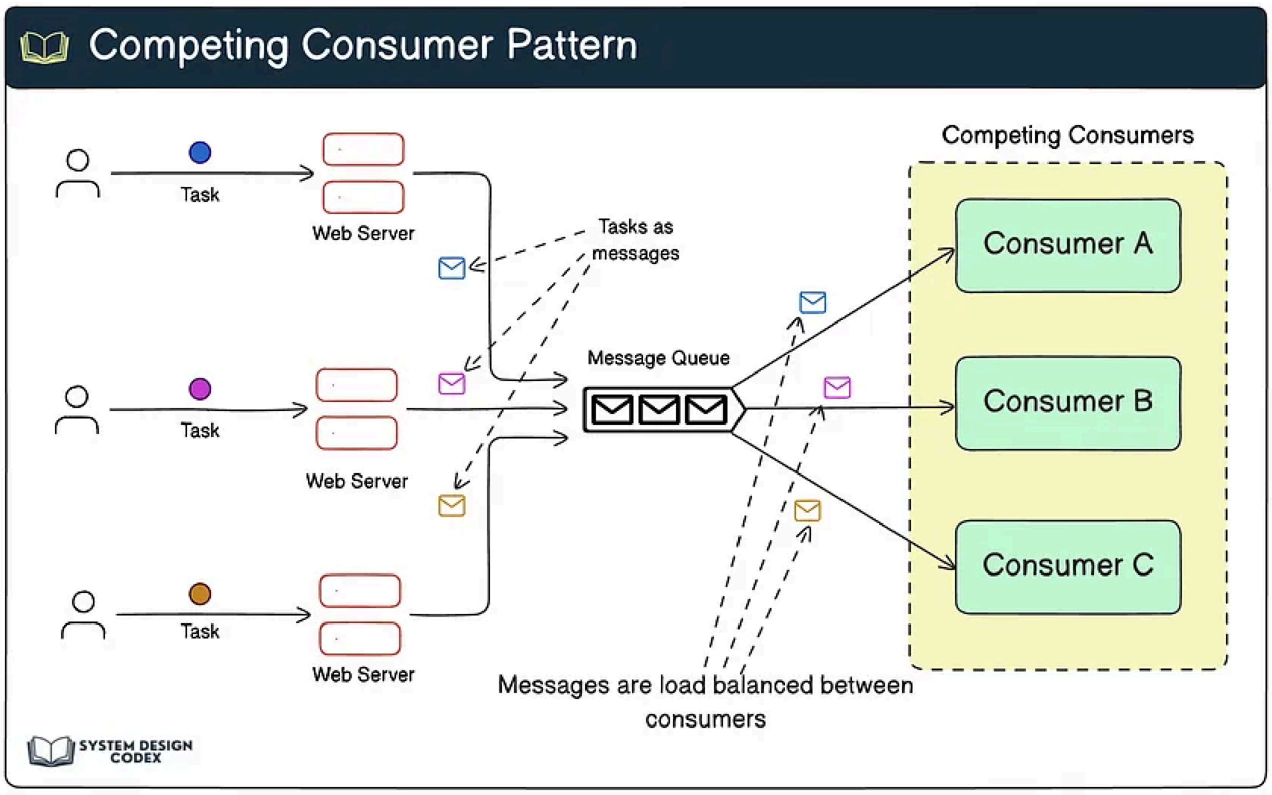
In today's post, we'll look at three event-driven patterns that you must be aware of while attending an interview. Plus, they can also help you in your project work.

1 - Competing Consumer Pattern

The question that can come up around this is as follows:

> **"How can you load balance a huge number of asynchronous messages across consumers?"**

The simplest approach is to let the consumers compete against each other. This is also known as the Competing Consumers Pattern.



You can play around with the diagram using Eraser.io

How does it work?

- One or more producers add messages to a queue. These messages are like tasks that need to be performed.
- Multiple consumer instances are set up to process messages or tasks from this queue.
- Each consumer competes to retrieve and process messages.
- Once a consumer successfully claims a message, it becomes unavailable to other consumers.
- After processing, the consumer acknowledges the message and removes it from the queue.

As you can see, the process is straightforward.

An important bit, however, is ensuring that a message is processed by only one consumer. In other words, how is the message claimed by a consumer made unavailable to other consumers?

Different platforms handle it in different ways.

RabbitMQ uses a prefetch count

- Consumers set a prefetch count, limiting the number of unacknowledged messages they can have.
- When a consumer receives a message, it's considered "in flight" and won't be delivered to other consumers.

Azure Service Bus uses a peek-lock mechanism

- A consumer receives a message in peek-lock mode, which locks the message.
- The message remains in the queue but is invisible to other consumers.
- After processing, the consumer marks it as complete.
- Once the lock expires, the message becomes visible again.

AWS SQS sets a visibility timeout

- When a consumer receives a message, SQS sets a visibility timeout.
- During this timeout, the message is hidden from other consumers. After processing, the consumer deletes the message.
- If the timeout expires before the message is deleted, it becomes visible again for other consumers.

2 - Retry Messages Pattern

The question around this pattern usually goes like this:

> **"How do you retry failed transactions using message queues?"**

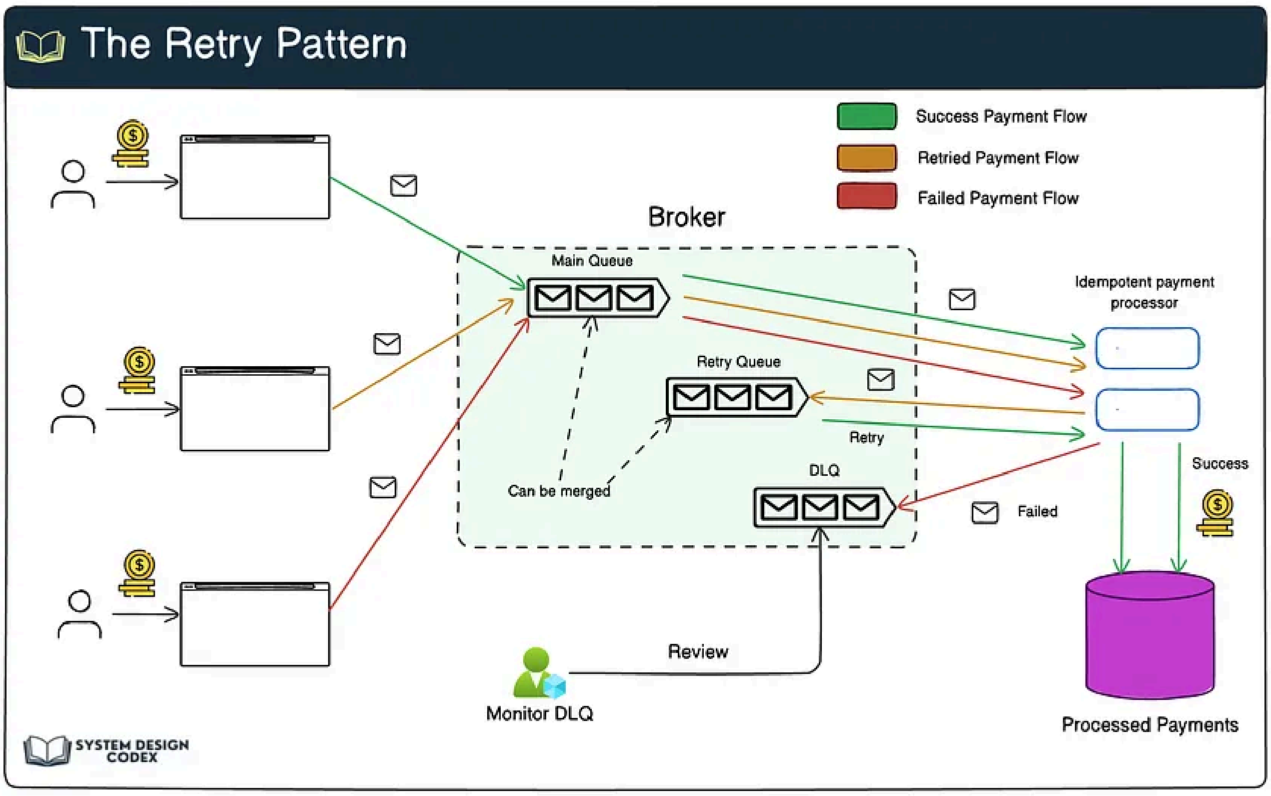
This is a common pattern to handle transient errors. Let's understand with the help of payment processing as an example.

The general approach to implementing a retry mechanism using message queues has 3 main parts:

- **Main Queue:** This is where new payment transactions are queued.
- **Dead Letter Queue:** A separate queue for messages that failed processing multiple times.

- **Retry Queue:** This is where retries are scheduled with delays. This queue is optional as you can also use the main queue for it.

See the diagram below:



You can play around with the diagram on Eraser.io

Here's how the process works:

1. The consumer or payment processor picks up a message from the main queue. It attempts to process the payment transaction.
2. If processing fails, it checks the retry count that's often stored in the message metadata.
3. If retry count < max retries, increment count and re-queue the message.
4. If retry count ≥ max retries, move the message to the DLQ.
5. For retries, you can either re-queue directly to the main queue with a delay or use a separate retry queue with a time-based trigger.
6. Lastly, monitor the DLQ for messages that have exhausted retry attempts. Implement a process for dealing with them.

Some best practices to keep in mind while following this pattern:

👉 **Exponential Backoff:** Increase the delay between retries exponentially to avoid overwhelming the system.

👉 **Idempotency**: Ensure that the payment processor can safely retry payment without crashing the economy

👉 **Message TTL:** Set an overall TTL for messages to stop very old transactions from being processed.

👉 **Retry Limits:** Set a value for max number of retries

👉 **Error Types:** Distinguish between transient errors (can be retried) and permanent errors (direct to DLQ)

3 - Async Request Response Pattern

The question goes like this:

> **"How to handle request-response communication with message queues?"**

First of all, we need to understand why this is needed.

With a message queue, you can let two services talk to each other (request-response) asynchronously. This approach is quite important for long-running tasks. Also, no synchronous calls means less chance of a single service bringing down the application.

But there is a problem.

What if there are multiple requester and responder instances?

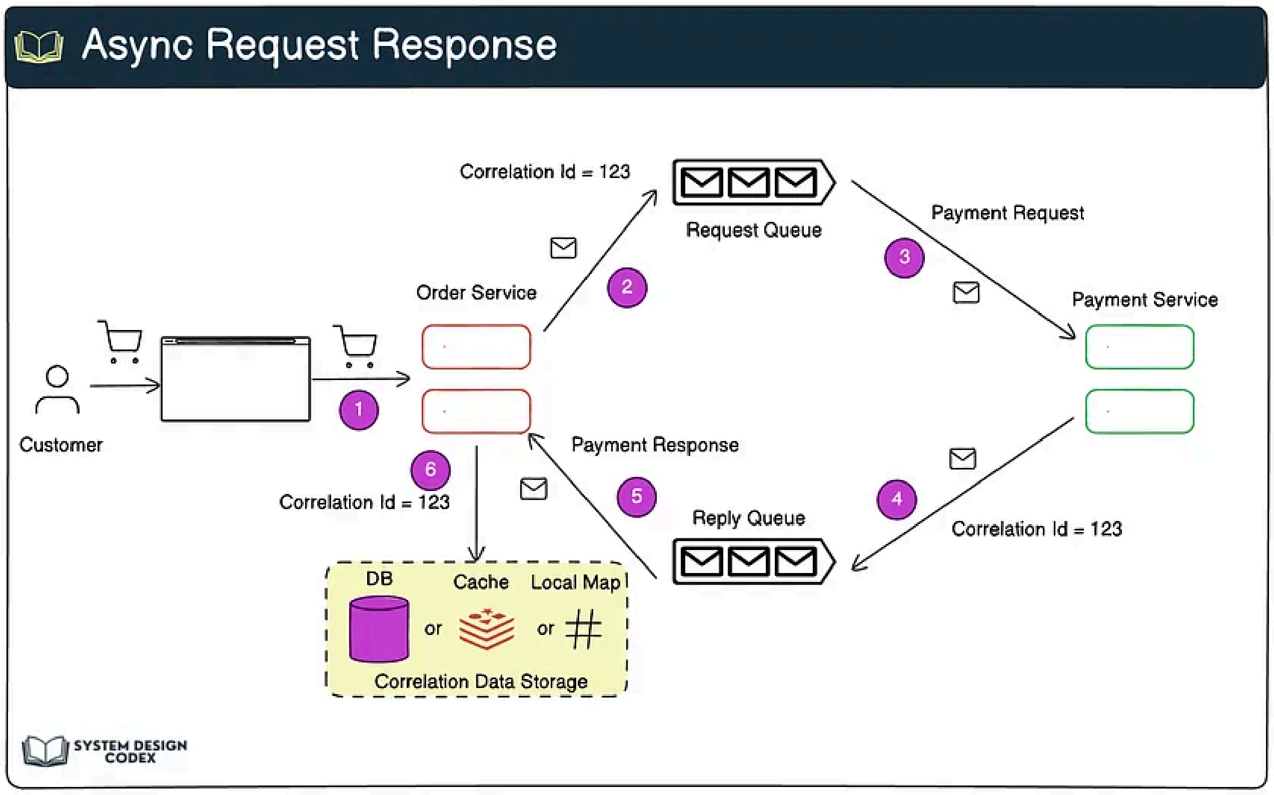
This doesn't matter much in a synchronous REST API call because one requester instance always connects to just one responder instance. There is a temporal coupling.

Not the case with async request-response.

The requester instance that made the request may not be the one that ultimately receives the response. It may be down or unavailable by the time the response comes back.

So - how do you relate a request and response across multiple ephemeral instances?

See the diagram below:



You can play around with the diagram on Eraser.io

As you can see, one common way is to use a correlation ID. Here's how it works:

- Suppose there is an Order Service and Payment Service that interact in an async request-reply model.
- A customer places an order through one instance of the Order Service. It generates a unique correlation ID for this order, storing the corresponding data in a database, distributed cache, or even a local instance-level HashMap.
- Next, it sends a payment request message to the Payment Service along with the correlation ID.
- The Payment Service (a particular instance) processes the payment and sends the response to the reply queue. The response message contains the same correlation ID.
- The Order Service (same or another instance) picks up the response message. It uses the correlation ID in the message to match the response to the original order request and takes the necessary action.

At this point, you may ask question the need for correlation ID. The same can be accomplished with the order ID.

True, but there are some benefits:

- Multiple payment requests can be sent for the same order (retries, partial payments, etc). A unique correlation ID/request helps match them properly.
- The correlation ID separates the routing logic from the business context (like Order ID).
- Correlation IDs make it easy to trace the flow of the request across multiple services.

👉 **So - which other event-driven patterns have you seen or used?**

Shoutout

Here are some interesting articles I've read recently:

- In the real world, you might need more than a simple Work Queue by Raul Junco
- Is Coding Now Pay To Win? by Akos
- Hexagonal Architecture with TDD by Daniel Moka

That's it for today! ☀️

Enjoyed this issue of the newsletter?

Share with your friends and colleagues.

See you later with another edition — Saurabh

---

56 Likes · 2 Restacks

Discussion about this post

Comments Restacks

Write a comment...

Dennis Onyango Sep 20 ♥ Liked by Saurabh Dashora

I must have been blessed to have realised your newsletter channel. Honestly I have learnt a lot from this single letter. Looking forward to learn more.

Thank you Sir.

LIKE (2) REPLY SHARE

Hitesh Garg Hitesh Garg Sep 11 ♥ Liked by Saurabh Dashora

What an amazing Article Saurabh!

I have one doubt that when we say we can save correlation Id in local hashmap as well, then one instance's local hashmap is accessible to other instances?

LIKE (1) REPLY SHARE

1 reply by Saurabh Dashora

9 more comments...