Handling Concurrent Requests With Message Queues

In this lesson, we will explore how concurrent requests are handled with a message queue.

We'll cover the following

- Using a message queue to handle the traffic surge
- How Facebook handles concurrent requests on its live video streaming service with a message queue?

Using a message queue to handle the traffic surge#

In the distributed *NoSQL* databases lesson, you learned about eventual consistency (https://www.educative.io/collection/page/6064040858091520/6411938009448448/6373547041619968) and strong consistency (https://www.educative.io/collection/page/6064040858091520/6411938009448448/5677430486335488). We discussed how both the consistency models come into effect when incrementing the value of a "*Like*" counter.

Here is a quick insight into how we can use a message queue to manage a high number of concurrent requests to update an entity.

When millions of users around the world update an entity concurrently, we can queue all the update requests in a high throughput message

queue. Then, we can process them one by one in a *First in First Out (FIFO)* approach sequentially.

This would enable the system to be highly available and open to updates while remaining consistent at the same time.

Though implementing this approach is not as simple as it sounds, implementing anything in a distributed, real-time environment is not so trivial. I thought I should just bring this approach up so you people could reflect upon this.

How Facebook handles concurrent requests on its live video streaming service with a message queue?#

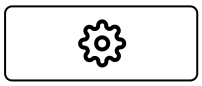
Facebook's approach of handling concurrent user requests on its LIVE video streaming service is another good example of how queues can be used to efficiently handle the traffic surge.

On the platform, when a popular person goes LIVE, there is a surge of user requests on the LIVE streaming server. To avert the incoming load on the server, Facebook uses cache to intercept the traffic.

However, since the data is streamed LIVE, the cache often is not populated with real-time data before the requests arrive. Now, this would naturally result in a *cache-miss*, and the requests would move on to hit the streaming server.

To avert this, Facebook queues all the user requests, requesting the same data. It fetches the data from the streaming server, populates the cache, and then serves the queued requests from the cache.

and then serve the queued requests from the cache.



This is a recommended read on Facebook's Live Streaming architecture (https://engineering.fb.com/ios/under-the-hood-broadcasting-live-video-to-millions/).

Alright, moving on!! In the next chapter we will take a deep dive into stream processing.

← Back

Notification Systems and Real-Time F...

Next →

Message Queue Quiz

Mark as Completed

Report an Issue