Notification Systems and Real-Time Feeds With Message Queues

In this lesson, we will discuss how notification systems and real-time feeds are implemented using message queues.

We'll cover the following

This is the part where you will get an insight into how notification systems and real-time feeds are designed with the help of message queues. However, these modules are really complex in today's modern Web 2.0 applications. They involve machine learning, understanding user behavior, recommending new and relevant information, integrating other modules with them, and so on. We won't get into this level of complexity, simply because it's not required.

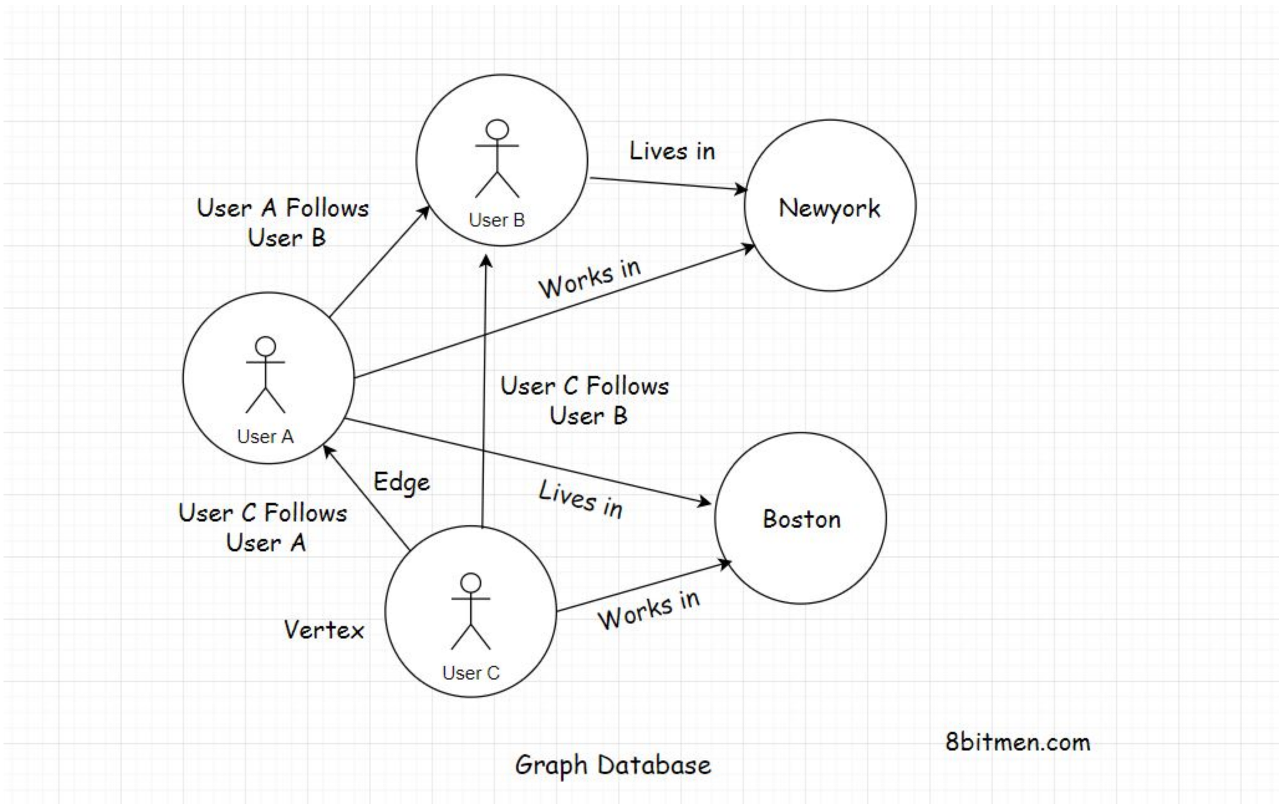
I present a very simple use case so we can wrap our heads around it.

Also, as we discuss this use case, think from your own perspective. Imagine how you would implement such a notification system from the bare bones. This will help you understand the concept better.

Real-world use case#

Alright!! So, imagine we are writing a social network like Facebook using a relational database, and we are using a message queue to add the asynchronous behavior to our application.

In the application, a user will have many friends and followers. This is a *many-to-many* relationship, like a social graph. One user has many friends, and they would be friends of many users. This is like we discussed in the graph database lesson. Remember?



So, when a user creates a post on the website, we would persist it in the database. There will be one *User* table and another *Post* table. Since one user will create many posts, it will be a *one-to-many* relationship between the user and their posts.

At the same time, as we persist the post in the database, we have to show the post created by the user on the home page of their friends and followers, even sending notifications if needed.

How would you implement this? Pause and think, before you read further.

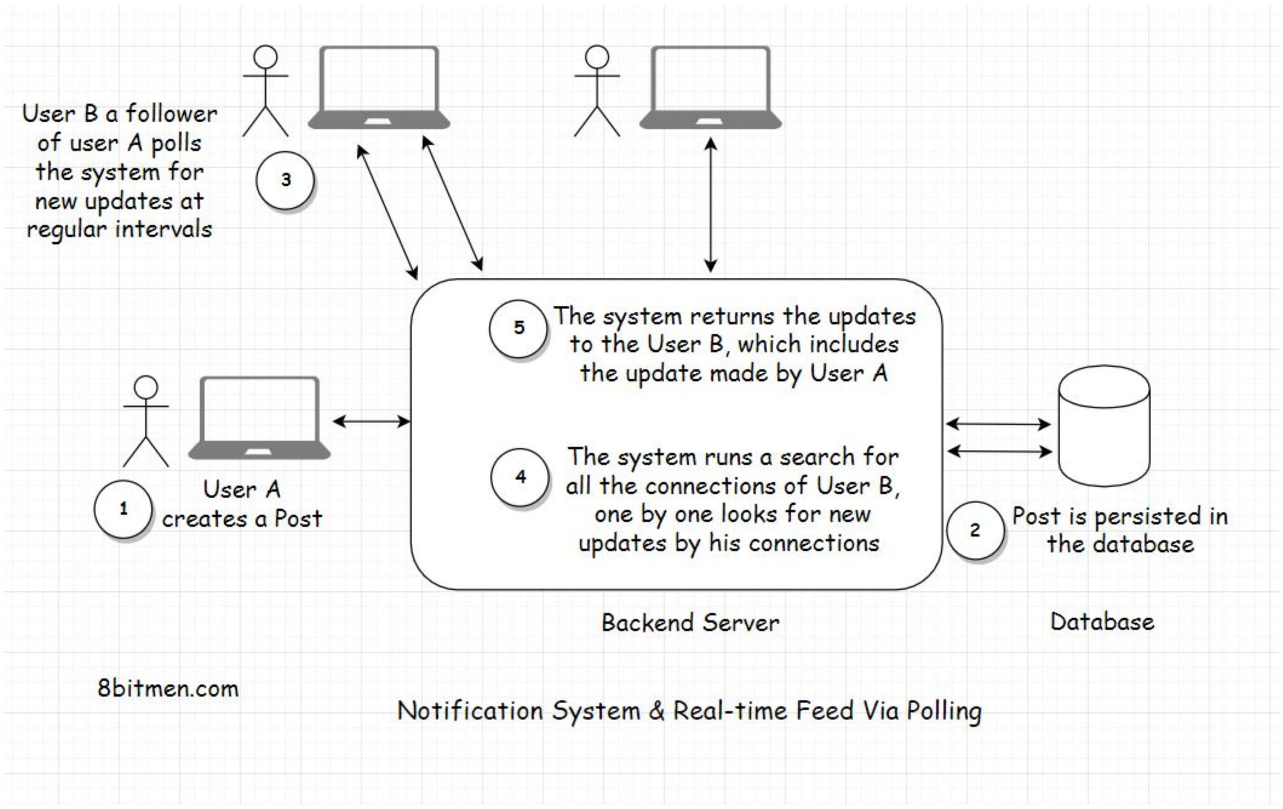
Pull-based approach#

Alright!!

One simple way to implement this without the message queue would be for every user on the website to poll the database if any of their connections have a new update at regular short intervals.

For this, we will find all the user's connections and run a check on every connection one by one for a new post created by them.

If there are new posts created by the user's connections, the query will pull them all and display the posts on their home page. We can also send the notifications to the user about new posts tracking the notification count using a notification counter column in the *User* table and adding an extra *AJAX* poll query from the client for new notifications.


User B a follower of user A polls the system for new updates at regular intervals
3
5
The system returns the updates to the User B, which includes the update made by User A
4
The system runs a search for all the connections of User B, one by one looks for new updates by his connections
1
User A creates a Post
2
Post is persisted in the database
Backend Server
Database
8bitmen.com
Notification System & Real-time Feed Via Polling

What do you think of this approach? It's pretty simple and straightforward right?

There are two major downsides to this approach.

First, we are polling the database very often, and this is expensive. It will consume a lot of bandwidth and will also put a lot of unnecessary load on the database.

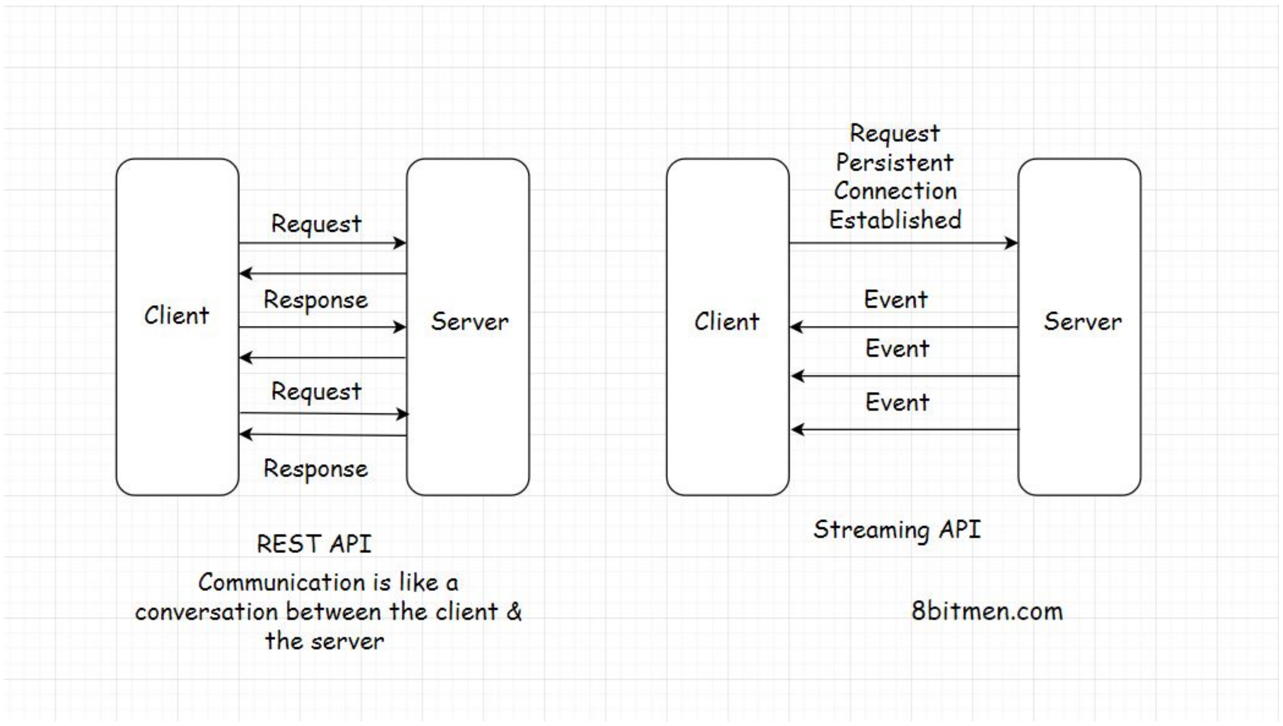
The second downside is that the user's post displayed on the home page of their connection's homepage will not be in real-time. The posts won't display until the database is polled. We may consider this real-time, but it is not really real-time.

Push-based approach#

Let's make our system more performant. Instead of polling the database every now and then, we will take the help of a message queue.

This time, when a user creates a new post, it will have a *distributed transaction*. One transaction will update the database, and the other transaction will send the post payload to the message queue. *Payload* means the content of the message posted by the user.

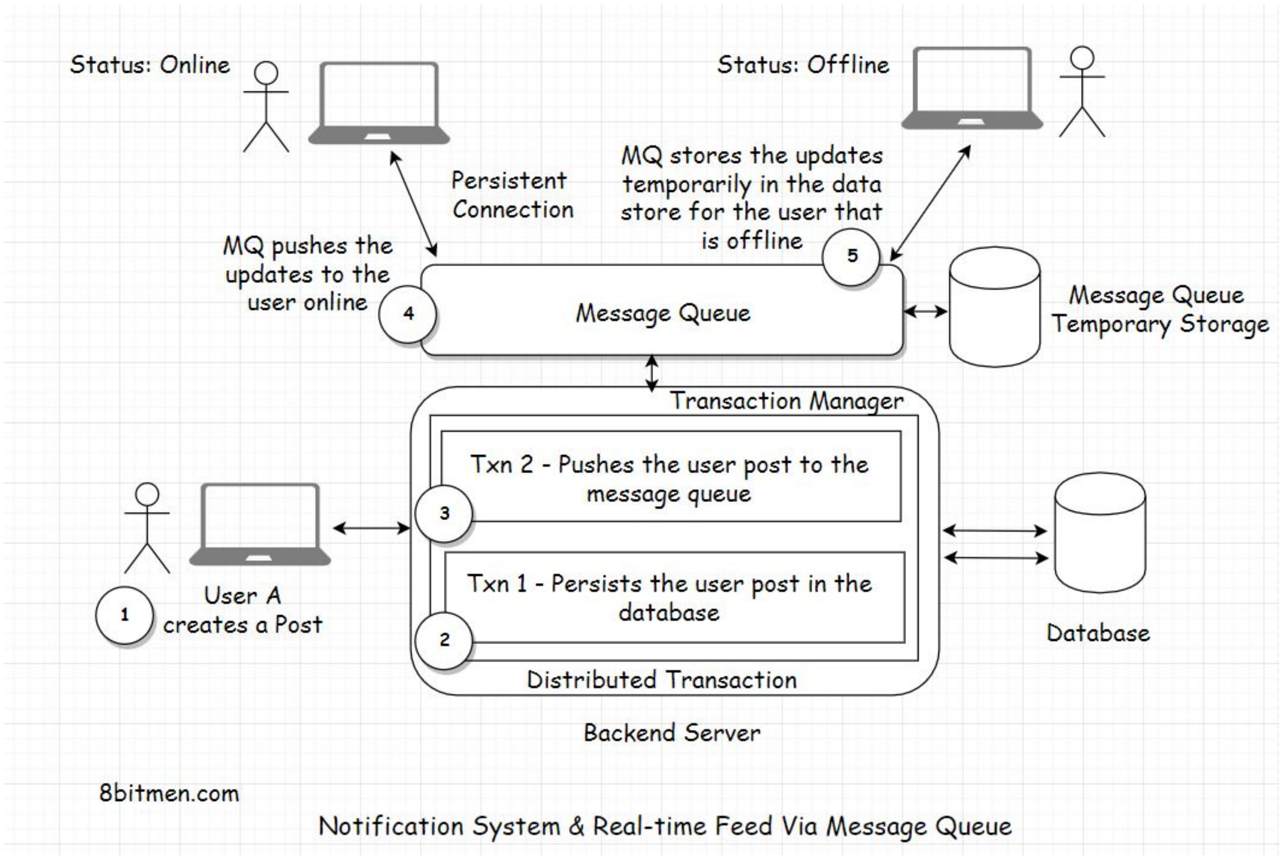
Notification systems and real-time feeds establish a *persistent connection* with the database to facilitate streaming data in real-time. We have already been through this.


Client
Request
Response
Request
Response
Server
REST API
Communication is like a conversation between the client & the server
Client
Request Persistent Connection Established
Event
Event
Event
Server
Streaming API
8bitmen.com

On receiving the message, the message queue will asynchronously and immediately push the post to the user's connections that are online. There is no need for them to poll the database at regular intervals to check if the user has created a post.

We can also use the message queue temporary storage with a *TTL* for the user's connections of the user to come online and then push the updates to them. We can also have a separate *Key-value* database to store the details of the user required to push the notifications to their connections, like their connection ID's and stuff. This would avert the need to even poll the database to get the user's connection.

So, did you see how we transitioned from a *pull-based* mechanism to a *push-based* mechanism with the help of message queues? This would certainly spike the performance of the application and cut down a lot of resource consumption, thus saving truckloads of our hard-earned money.


Status: Online
Status: Offline
Persistent Connection
MQ stores the updates temporarily in the data store for the user that is offline
5
MQ pushes the updates to the user online
4
Message Queue
Message Queue Temporary Storage
Transaction Manager
Txn 2 - Pushes the user post to the message queue
3
Txn 1 - Persists the user post in the database
2
1
User A creates a Post
Database
Distributed Transaction
Backend Server
8bitmen.com
Notification System & Real-time Feed Via Message Queue

There are several ways we can handle the distributed transactions. Although the transactions are distributed, they can still work as a single unit.

If the database persistence fails, naturally, we will roll back the entire transaction. There won't be any message push to the message queue either.

What if the message queue push fails? Do you want to roll-back the transaction, or do you want to proceed? The decision depends entirely on you and how you want your system to behave.

Even if the message queue push fails, the message isn't lost. It can still be persisted in the database.

When the user refreshes their homepage, you can write a query where they can poll the database for new updates. Take the polling approach we discussed initially as a backup.

Otherwise, you can totally rollback the transaction, even if the database persistence transaction is a success but the message queue push transaction fails. The post still hasn't gone into the database yet as it is generally a two-phase commit. We can always write custom distributed transaction code or leverage the distributed transaction managers that the frameworks offer.

I can go on and on about this, into the minutest of the details, but that would just make the lesson unnecessarily lengthy. For now, I have just touched the surface, to help you understand how notification systems and real-time feeds work.

Just like the post creation process, or the flow repeats itself when the users trigger events like visiting a public place, eating at a restaurant, etc. The message queues push the events to the connections of the user.

When designing scalable systems, I want to emphasize that there is no perfect or best solution. The solution should serve us well and fulfill our business requirements. Optimization is an evolutionary process, so don't sweat about it in the initial development cycles.

First, get the skeleton in place and then start optimizing notch by notch.

Recommended read - How Does LinkedIn Identify Its Users Online? (https://www.8bitmen.com/linkedin-real-time-architecture-how-does-linkedin-identify-its-users-online/)

← Back

Point-to-Point Model

Next →

Handling Concurrent Requests With ...

Mark as Completed

Report an Issue