Design Notification Service

Notification services are widely used in various products today to alert users of important updates or changes. For example, you might receive a notification about a price change for a product you're interested in or a new job specification that matches your search criteria. With the widespread use of mobile applications, receiving push notifications on our mobile devices and traditional email notifications has become more convenient.

In this blog, we will explore the design of a notification service, a critical component of many products and a popular topic in interviews. Let's delve into the process of building such a service.

Key Requirements

Before designing a notification service, it is important to clearly define the key requirements that it should meet. These requirements may vary depending on the specific use case, but some common ones include:

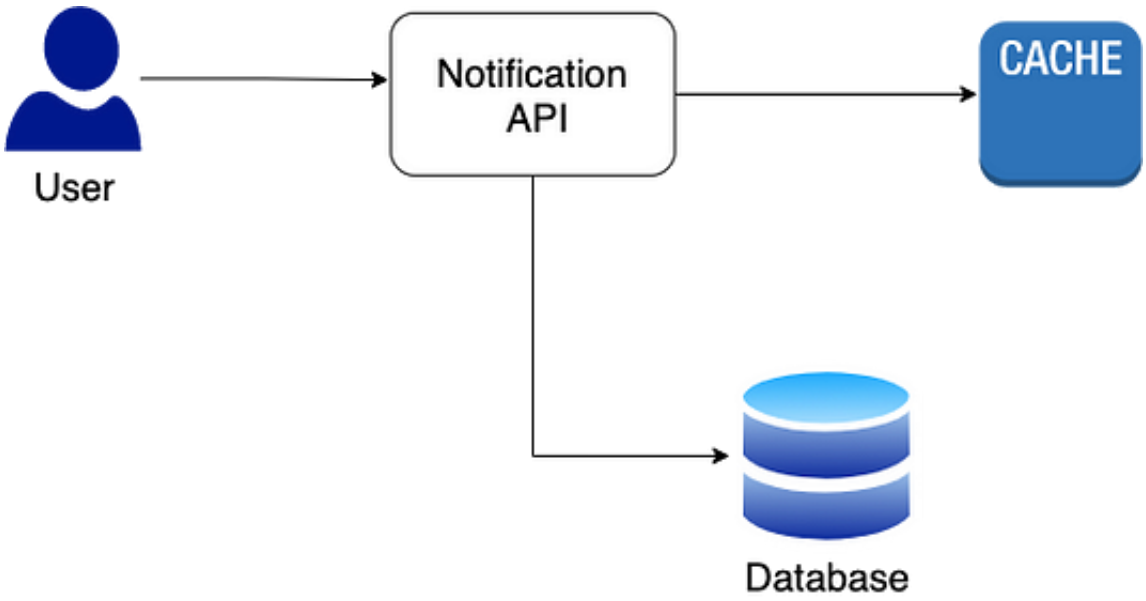
- Users should be able to "watch" or "follow" a product to receive notifications about it.
- The service should be able to send users email or push notifications when relevant events occur.
- The notification service should be able to operate seamlessly with third-party integrations.

It is also worth considering whether the notification service should be integrated into an existing service or separated into its own set of components.

High Level Design

Having well-designed APIs is an essential part of the notification service. These APIs should allow us to create and delete notification entries as

needed.



In order to determine when a user should be notified, the notification service needs to save criteria for each product or set of search parameters selected by the user. These criteria can be stored in a database. The following APIs would be important for managing and accessing these notification criteria:

- **create*notification(key, product*id, user_id)**: This API would create a new notification event based on the given product and user.
- **delete*notification(key, notification*id)**: This API would delete an existing notification event with the given id.
- **get*notifications(key, user*id)**: This API would retrieve the notifications for a particular user.

APIs for the notification service can be implemented using RESTful or gRPC architectures. Depending on the specific requirements of the service, one of these approaches may be more suitable. For example, if the service allows consumers to view a list of generated notifications, it may be expected to have more read operations. On the other hand, if the service frequently create and delete notifications, the write performance of the API may be more important.

Detailed Components Design

We may use a variety of ways to create our notification service. We have the option of using batch jobs or the API internal response logs. However, in this article, we'll construct our service utilizing the batch jobs technique.

A simple method for alerting would be to run a batch process regularly to see if any user changes are "watched." Let's suppose users want to be notified if the price changes; thus, the batch job has to call an internal API that returns the product's pricing to verify the lowest price.

This method will provide adequate coverage and will cover all of the database's search criteria. However, as the number of notification items grows, so does the number of calls to the internal API. Scheduler, Data Pipeline, Message Queue, Batch Job, and Communications service is required for the batch jobs-based strategy.

Scheduler

This component may be used to perform our data pipeline, batch processing, and other services regularly, such as daily or weekly.

The following factors influence the start time of the data pipeline and batch processing:

- Off-peak hours: When there is less traffic, we want to check the database for the Notification service.
- User's time zone: The scheduler should be set to receive the notice when the user is awake. This will undoubtedly result in a higher probability of receiving the notice, especially if it is a push notification.

Data pipeline

The data pipeline is told when to start by the scheduler. It searches the database for alerts, organizes them according to criteria, and adds them to a queue. This component reads the Notification API Database. When reading from the Notification API Database, consider the following

strategies:

- Full Database scan: The batch job will read all the rows in the Notification API database.
- Partial database scan grouped by regions: The batch job reads the entries based on their region.

The data pipeline will publish the notification items to a message queue after reading them from the database.

Batch Jobs

Batch jobs are subscribed to the message queue's notification entries subject. These jobs will pick up the messages from the notification queue. The batch job will use the Internal Pricing or Availability API to retrieve the most up-to-date pricing depending on the database's notification criteria.

- Compare the most current API pricing to what was previously stored in the database.
- Decide whether or not to alert the user based on the price difference or whether or not the product is available. Sending a request to the Communications component to notify the user.
- We'll update the notification record in the database when we've finished the request to the Communication component. At this time, the last update and Prices fields will need to be changed.

Notification Service Use Case

This part will discuss how notification systems and real-time feeds are implemented using message queues using the Facebook newsfeed notification system.

Assume we're creating a social network similar to Facebook. To add asynchronous behavior to our program, we'd utilize a message queue. A user will have a large number of followers and friends. One person can have a lot of friends and be friends with a lot of people. We must store a post created by a user in the database. As a result, we'll need a User table

and a Post table. Because a single user can create many posts, the user and posts database will have a one-to-many connection. We must display the user's post on his friends' home page while saving the post in the database. This new post necessitates sending a notification to our friends.

Now the question arises, how to send notifications? Let's find out :)

We can utilize a notification system that is based on push notifications. When data becomes available, the available server will deliver it to the client. As a result, instead of polling the database regularly, we'll use a message queue to transmit data to the clients.

When a user makes a new post in this method, the system does two distributed activities. One action is to update the database to save the post, which is necessary; otherwise, the data may be lost. Another option is to submit the post's content to a message queue.

As a result, after the data has been received, the message queue will asynchronously push the post to the user's connections. As a result, there's no need to query the database regularly to see if a buddy has made a post. In this approach, we may avoid polling procedures. We can utilize message queue storage with an expiry period to wait for offline users' connections to come online in the event of offline users. Then send the changes to those individuals.

As a result, combining a push-based technique with message queue implementation may improve application performance while lowering resource consumption.

The database storage procedure must be handled with caution. We can't push the message to the message queue if it fails since it would generate a system discrepancy. Before sending the message to the message queue, we need to ensure that the database persistence was successful. Otherwise, we'll have to inform the user that the post could not be created due to server issues.

Conclusion

In this blog, we tried to explain the internal working and design of the notification service. They are used in almost every product, and as a result, it is essential to know about them from the perspective of any system design interview. I hope you liked this blog. Please do share your views.

Thanks to Suyash for his contribution in creating the first version of this content. If you have any queries/doubts/feedback, please write us at contact@enjoyalgorithms.com. Enjoy learning, Enjoy system design, Enjoy algorithms!