Sabyasachi Nayak Follow
Jul 13 · 10 min read · Listen

Save

Design a Notification System

Why do we need a notification system?

Financial orgs: It is very important to tell people about their upcoming payments.

ECommerce orgs: When subscribed to an e-commerce outlet, customers may be notified of the availability of a product, order placement, or order shipment.

LinkedIn: When subscribed to a recruiting platform, users may be notified of the latest jobs appearing for the job criteria that they have set.

Facebook NewsFeed: The notifications that pop up on your mobile phone for a new post on your Facebook newsfeed also come to you via the notification service.

Many applications use a notification system to send OTPs via email or SMS to

- The system should be able to send notifications to subscribed consumers.
- The system should be able to prioritize notifications. OTPs are high-priority messages while news feed updates may be a lower priority.
- The system should be able to support Email, SMS, and push notifications on mobile and web browsers.
- Single/simple and bulk notification messages
- No same notification twice
- Log every notification dispatched, delivered, opened, seen, unsuccessful, canceled

Non Functional Requirements

- The system should be highly available.
- Latency should be low. OTP messages are time-critical.
- It should be scalable, to handle a growing number of subscriptions.
- Scalable for higher load on-prem (VMware Tanzu) and on public cloud services like AWS, GCP, or Azure etc.

Capacity Estimation:

The notification system is not just expected to send notifications but also handle the scale of the organization.

Any large size organization sends about **1 million** notifications every day.

How many Notifications do we need to send?

And organizations like Facebook, with their 2B user base, let's say 25% of them are active users and 25% of those active users have opted in for push notifications, that would come out to be `2 * 10^6 * 10^3 * .25 * .25 = 125M` every day.



Let's break it down a little more to get hold of it.

`125M / 24 / 3600 = 1446 approx.` notifications being sent **every second**.

How many machines do we need to send those notifications to?

Let's say one synchronous thread takes around **100 milliseconds** to process and send one notification, that would mean 1 thread could send 10 notifications every second. If it ran one single synchronous thread per machine, then FB would need around `1446 / 10 = 145 approx` servers spread across the globe to process the assumed load.

Design a High-Level View of the System

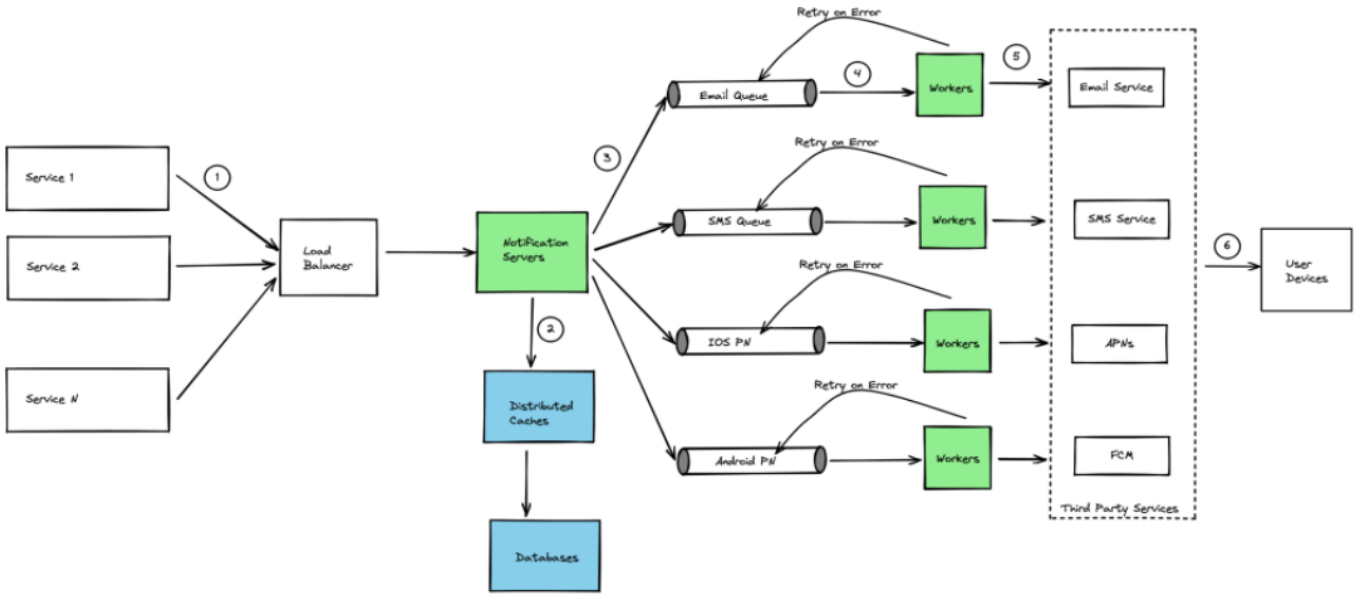
Always start from what you know. And we know that:

- IOS notification delivery is handled by **APNs**. You trigger their API with the required payload and they deliver the notification to the user.
- Similarly, GCM (**Google Cloud Messaging**) by Google sends the notification to android devices.
- **Mailchimp** for sending email notifications.
- **Twilio** for SMS.

So What we are building is the **Notification System** that would make use of these services to deliver the notification to our users.

1. There will be services that would be sending data to the notification service. A service could be a micro-service, cron-job etc. For example — a billing system will send emails and push notifications to remind its customers of the due date for the payment.
2. **Notification System**: This is the heart of it all that orchestrates sending/receiving of notifications. This notification system will fetch the user preferences from the database, construct the payload and send it to the third-party services for delivering notifications. This will expose REST APIs to clients and interact with the clients

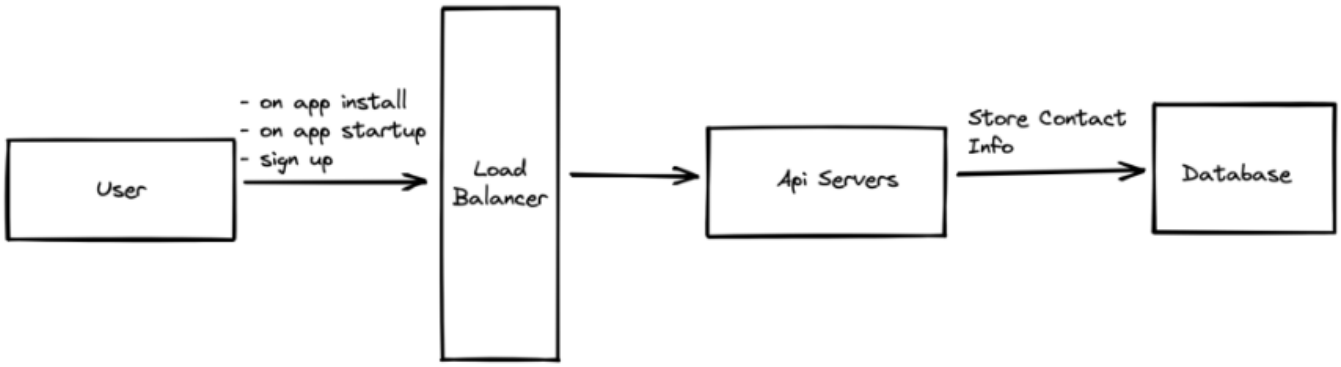


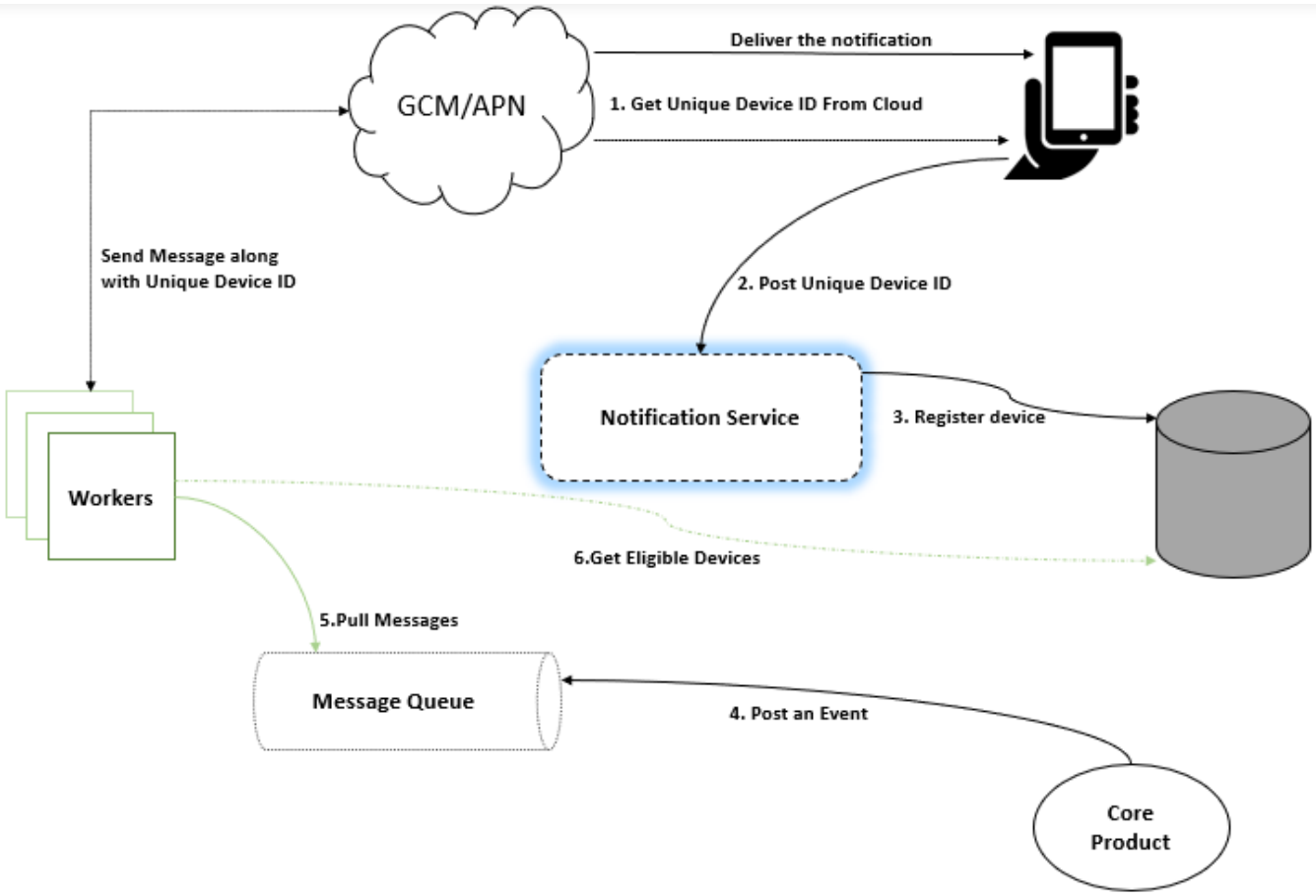
Before discussing all this how the notification system will gather user information?

To send notifications we need to gather email, mobile numbers, and device tokens.

Different information could be accessed and stored during different times like;

- App Install — device token
- App Startup — device token
- Sign Up — contact info





How the device ID is generated and how it is used by the notification system:

1. The first step is to get the unique device ID from the cloud messaging service. The mobile application should make a call to the cloud messaging service every time it is launched. It is important to make this call every time as it may so happen that the unique device ID assigned earlier has now expired.

2. The second step is to call the **Notification web service** from the mobile application and include the unique device ID (received from cloud messaging service) in the payload. You may want to pass some additional information like the device type, device operating system, etc in the payload.

3. The Notification web service will store the registration ID in a local database for later use.

4. Whenever the "**Notification service**" needs to send the notification, it has to call the

5. [sourcecode language=”javascript”] DevicePayload:{ deviceId:’APA9sdsdA’, deviceType:’mobile/tablet’, deviceOS:’android/apple’, userID:’xyz@gmail.com’, } [/sourcecode]

Design Deep Dive

From **figure1** the Notification service is horizontally scalable (by adding more servers) and it is decoupled from the other components by introducing a message queue.

Notification Service:

This exposes REST APIs to clients and interacts with the clients. They are responsible to build notification messages by consuming **Template Service.**

It can be of two types:

- **Simple Notification Service:** It’s the main service, which will handle simple notification requests.
- **Bulk Notification Service:** It’s the main service, which will handle bulk notification requests. A bulk notification is a notification that is sent to a group of people.

We can define Notification service into three components:

👉**Validation Service:** This service is solely responsible for validating notification messages against business rules and expected format

👉**Scheduling Service:** This service will provide APIs to schedule notifications immediately or at any given time. It could be any of the followings:

- Second
- Minute
- Hourly
- Daily



- Yearly
- Custom frequency etc.

There could be other services also, which can be auto-triggered messages based on the scheduled times.

👉**Prioritization Service:** It will also prioritize notification based on high, medium, and low priorities. OTP notification messages have a higher priority with a time-bound expiry time, they will always be sent in higher priority.

NotificationTemplate Service:

A large notification system would send millions of notifications for thousands of different reasons. Therefore, every notification system contains some sort of templating mechanism that provides the structure for every type of notification.

So This service manages all ready-to-use templates for OTP, SMS, Email, chat, and other notification messages.

User Selection Service:

This service will provide services to choose target users and various application modules. There could be use cases to send bulk messages to a specific group of users or different application modules. It could be also AD/IAM/eDirectory/user database/ user groups based on customers' preferences.

User Profile Service:

This service will provide various features including managing users' profile and their preferences. It will also provide a feature to unsubscribe for notifications and also notification receiving frequency etc. **Notification Service** will be dependent on this service.

Notification Database Service:

Store all notification messages with their delivery time and status. Metadata contains the

👉Answer: **Casandra** or **MongoDB**

The reasons are:

Schema Flexibility: If we take a social networking app like Facebook, then the various entities are posts, friend requests, groups etc. Often application features are extended and new entities will be introduced, so new notifications related to those entities will also have to be added. The database schema should be flexible enough to support notifications for new entities easily.

Write-heavy System: One of the major issues is the rate at which our data grows. For example, If we take a social networking app like Facebook, there are various entities like posts, friend requests, and group activities etc, each one would contribute to events that will eventually trigger notifications.

Highly available System: Our system should be highly available to capture every event.

Event Hub or Message Queue Service:

Why do we need a message queue here?

Module Decoupling:

The notification system handles data storage and retrieval, information processing, and sending notifications to third-party.

Processing and sending notifications can be resource-intensive. For example, generating an HTML document with the correct user data with all validations in place. Sending that payload to service and waiting for the acknowledgment. All this combined could take signification time in processing. Handling everything in one place could result in a system overload. So we need to decouple the information processing and notification sending services.

Event Prioritizing:



For example, if use Kafka as a message queue. Kafka is a fast, scalable, distributed in nature by its design, partitioned, and replicated commit log service. So there is no priority on topic or message.

Create topics in kafka queue,Let say:

1. high_priority_queue
2. medium_priority_queue
3. low_priority_queue

Publish high priority message in high_priority_queue and medium priority message in medium_priority_queue.

You get a stream of each topic. Now you can first read the high_priority topic if the topic does not have any message then fall back on the medium_priority_queue topic. if medium_priority_queue is empty then read low_priority queue.

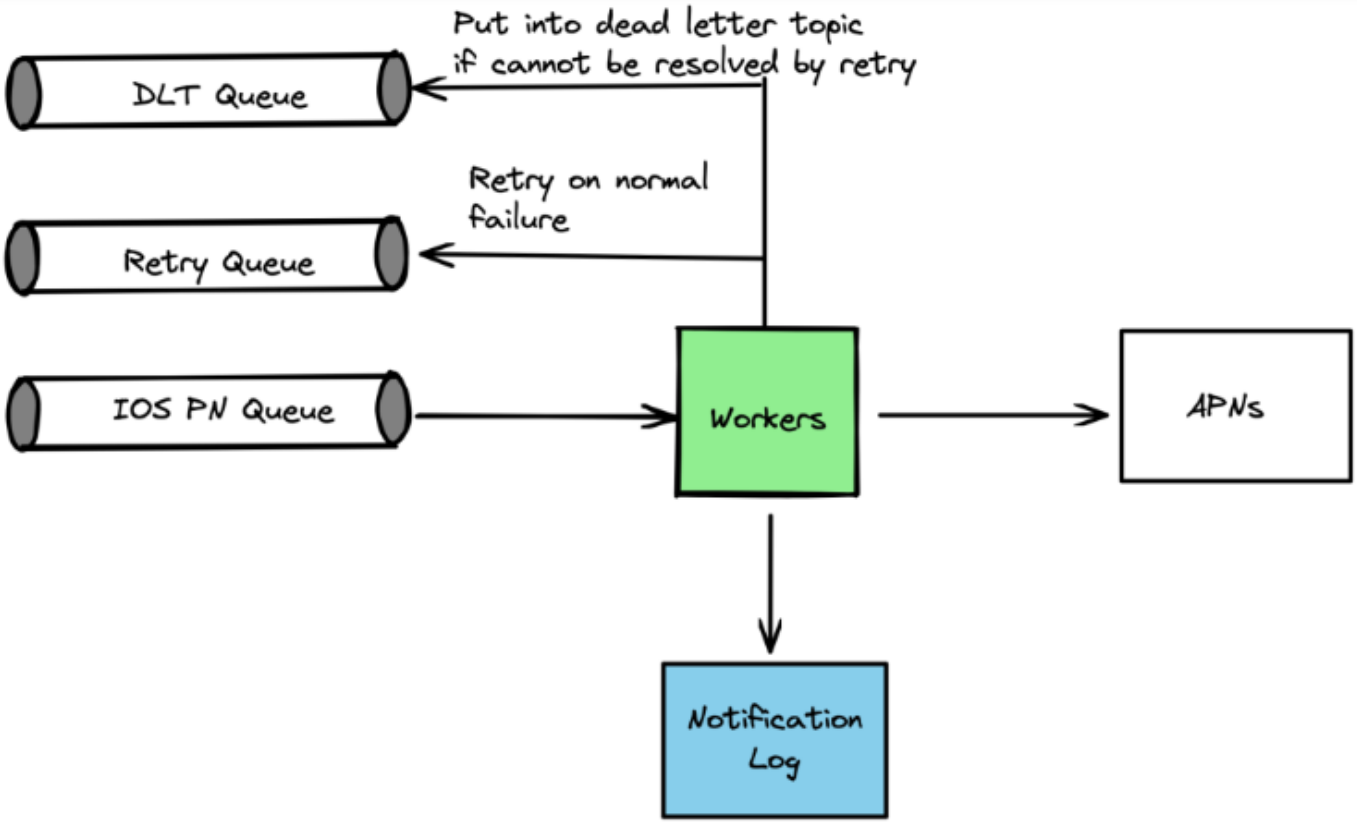
Reliability:

The notifications could be delayed or out of order but they cannot be dropped. Notifications must be sent to the customer. And reliability becomes important. For this, it is better to maintain different queues that could be used for retry and other stuff. So, for every queue, we will create the respective retry and DLT (Dead Letter Topic) queue.

Another thing that we need is the notification log. A notification system should keep track of every notification that is processed. This could be used for auditing and debugging as well. All and all it seems to be a good idea to keep a notification log database.

The component design for the retry mechanism is below:





Retry mechanism

With this, the notification system is capable of handling failed notifications. Keep track of every notification. And it's much more reliable. This idea can be expanded further. Like you could create another worker that will read data from the DLT queue and process the known failures. So, that the manual work could be minimized.

Rate-Limiter

The rate limiter checks the messages for two things:

- If the publisher is allowed to send that many requests.
- If the subscriber can receive as many notifications.

The rate-limiting information is stored in the cache and is checked and updated against the rate-limiting data that comes with the message. If the rate limit isn't exceeded, the request is forwarded. If the rate limit has exceeded, the request is throttled (dropped)

These are adapters that will transform incoming messages from the event hub (Kafka) and send them to external vendors according to their supported format. These are a few adapters, we can add more based on use case requirements:

- OTP Adapter Service
- SMS Adapter Service
- Email Adapter Service
- In-App Notification Adapter Service
- WhatsApp Chat Notification Adapter Service
- Telegram Notification Adapter Service

Notification Vendors:

These are the external SAAS (on cloud/on-prem) vendors, which provide actual notification transmission using their infrastructure and technologies. They may be paid enterprise services like AWS SNS, MailChimp, etc.

- SMS Vendor Integration Service
- Email Vendor Integration Service
- App Push Notification Vendor Integration Service
- WhatsApp Vendor Integration Service
- Telegram Vendor Integration Service

Notification Analytical Service

- Analyze and identify notification usage, trends
- Using Cassandra as an analytical DB



who often send or communicates with the user, and many more

How to ensure At Most Once Delivery?

Is it possible to receive a notification exactly once?

The answer is no. It is not possible.

The distributed nature of the entire system could result in a duplicate notification. For instance — let's say service 1 pushed the 10 events in the queue and before committing the status it fails. Next time when it comes up, it checks the state and retries 10 events again. That way those 10 events will be duplicated. In short, duplication is not possible at the producer level. However, duplication could be minimized.

For minimizing duplicates, we could implement a de-duplication mechanism before sending a notification to the customer.

De-Deuplication Mechanism

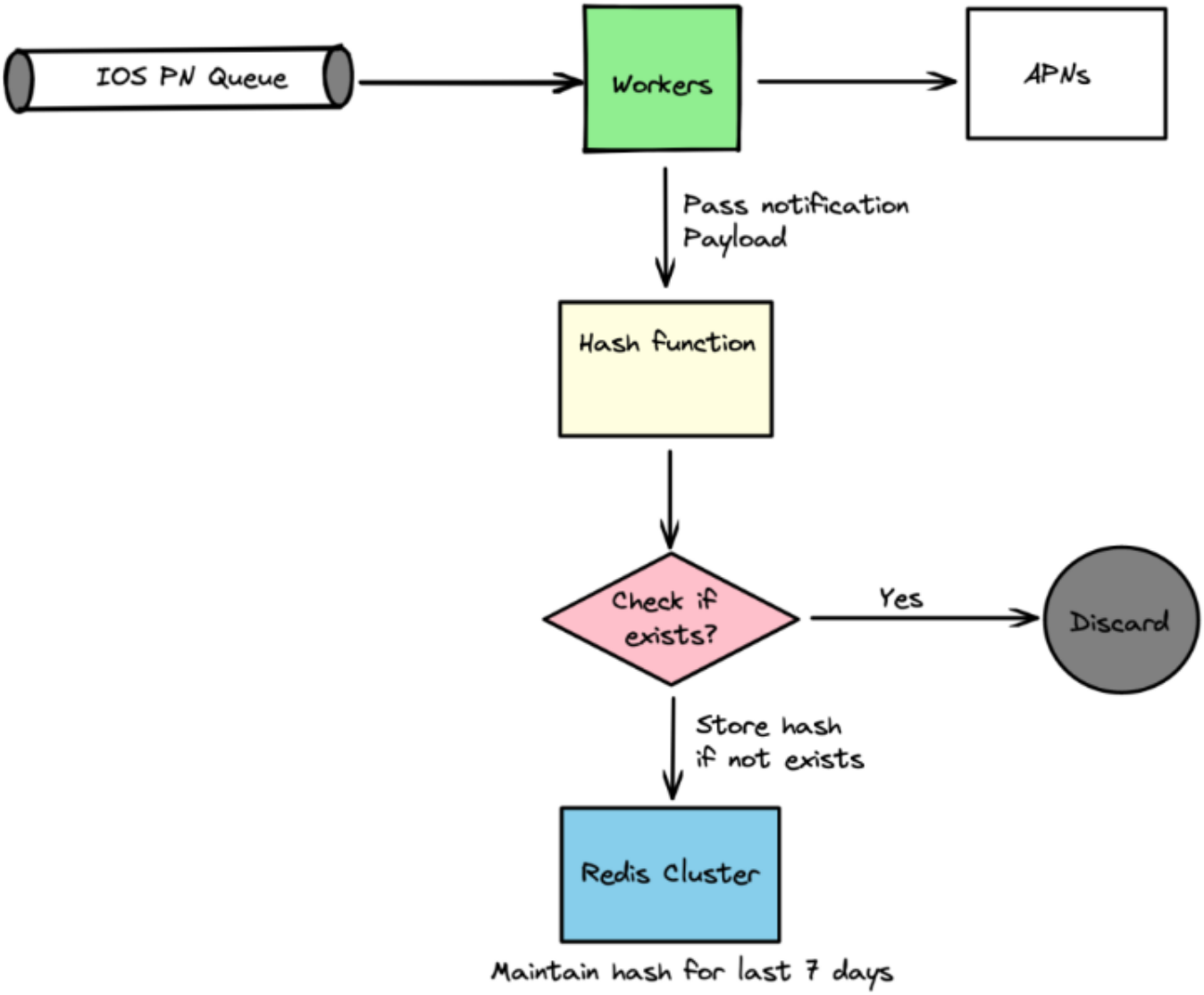
De-duplication logic could be implemented in many ways.

Hash the payload

The easiest way to identify if the payload is duplicate or not is to hash the entire payload and compare it in the database. If the hash is found then the notification should be discarded.

Check the below diagram:



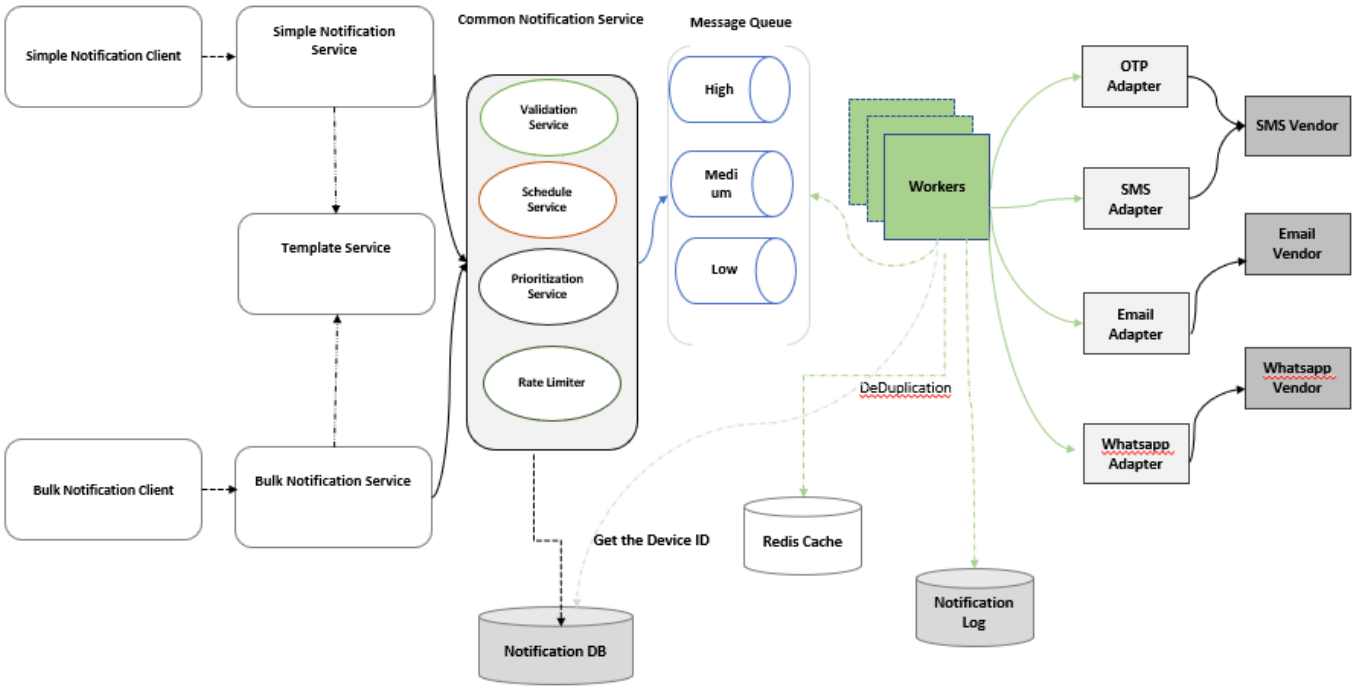


The component will work in the following way:

1. Hash the incoming notification payload
2. Query cached database to check and see if it exists or not
3. If the hash doesn't exist then store the hash and process the event (send the notification)
4. If hash exists then skip sending the notification

In this case, we could limit the size of the cache to the last 7 days. Actually, 7 days is also

Let's put everything together and see what the entire notification system will look like:



References:

Notification System Architecture

Let's design a notification service.

medium.com

Design A Notification System - BeMyAficionado

The notification system is a very important part of any organization, especially financial orgs. For ex: It is very…

www.bemyaficionado.com