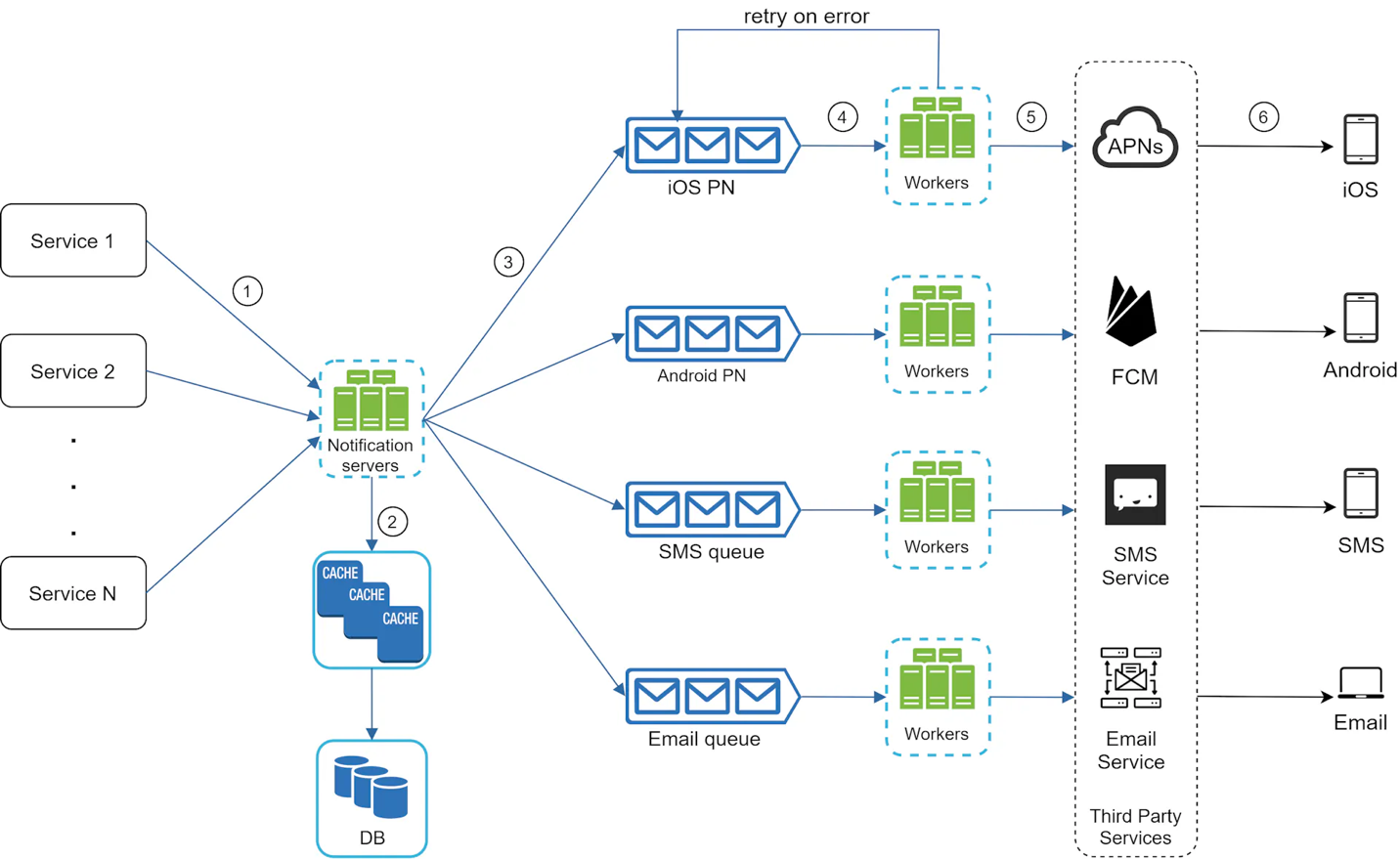
Byte byte go solution



Notifiacton servers is important

1. So not a single point of failure here

How would we scale when there is a spike in notification?

1. Every time you see a spike, just scale up the # of workers you currenlty have and then scale them so now it can handle all the requests coming in here

What

This is what we have here

1. Move the database and cache out of the notification server.
2. Add more notification servers and set up automatic horizontal scaling.
3. Introduce message queues to decouple the system components.

Using message queue here

**Service 1 to N**: They represent different services that send notifications via APIs provided by notification servers.

**Notification servers**: They provide the following functionalities:

* Provide APIs for services to send notifications. Those APIs are only accessible internally or by verified clients to prevent spams.
* Carry out basic validations to verify emails, phone numbers, etc.
* Query the database or cache to fetch data needed to render a notification.
* Put notification data to message queues for parallel processing.

And then next here

**Workers**: Workers are a list of servers that pull notification events from message queues and send them to the corresponding third-party services.

**Why is there a separate queue for each notification type?**

For each notification type a separate queue has been introduced along with dedicated workers to process the messages for that queue. This was necessary and preferred over a single queue for all notification message types which aims to avoid issues when a third-party notification service (e.g. Twilio) becomes no longer available for a short time. This might flood the single queue with numerous SMS notification messages that would result in affecting other types of notifications waiting on the same queue but never reaching their workers due to Twilio being down.

Different queues should be used for different notification servies.

**What json structure for send notification?**

Although ByteByteGo’s original solution provides a notification JSON message with a specific structure, I put forward below a different JSON message structure holding a template\_name and substitution arguments for sending a templated message:

# Message will send to a specific user\_id and specific template\_name and  
# substitute the variables defined in the template.  
{  
 "user\_id": 123456,  
 "type": "EMAIL", # Not important since endpoints define type  
 "template\_name": "WELCOME\_NEW\_USER",  
 "substitute": {  
 "NEW\_USER\_FULLNAME": "First Last",  
 "FROM\_EMAIL": "noreply@xyz.com",  
 "VERIFY\_URL": "https://xyz.com/verify?username"  
 }  
}

**How is each notification queue protected against dedupe logic considering they provide at-least-once delivery?**

Dedupe logic here

1. And why this is important here,

**Option #3.** If avoiding de-duplicate messages is a must, we could also consider using a key-value database like DDB or Redis. We should also add unique message ids for each message on the queue. A transactional database query will check if the message ID exists, then write it if not, before processing (i.e., sending the notification).. If the same message gets duplicated, the second time the application code will try to check if it exists in DDB and notice a record for this message id already exists and thus skip processing.

Also be sure to enable the TTL for datbase as well here

**What should be monitored in this system?**

Monitoring the queue sizes is the most important since it gives an indicator whether the configured number of workers are catching up or not. If the queue sizes start growing then most probably more workers might need to be added.

**How to prevent data loss?**

To satisfy this requirement, the notification system persists notification data in a database and implements a retry mechanism. The notification log database is included for data persistence, as shown in Figure 11.

We can have a database table to store all the db here

A screenshot of a notification table

Description automatically generated

**How do we know if a message has been processed?**

A screenshot of a computer

Description automatically generated

Once a msg is processed we will then let the worker write back to to the user information table here, and then update it to either failed or success as said

**Why would we choose a nosql over mysql?**

Wrirting notifications can be very high volumne, so writign very fast in mysql not very perfomant here.

**How do we deal with Failed notifications?**

1. It gets lost in the q first since it has been already removed?

This is what this table will do in general

what it's gonna do is pull all the rows from the my sql table so the notification table that has the status set to failed so these are the failed notifications and it's going to re-enqueue them in the message queue okay so all this is doing is reading the field notification from the my sequel table and re-enqueuing them in the message queue

so that the workers can attempt to send it again if it fails for the second time the worker is going to update the status of that notification to failed again and the notification scheduled job is gonna pick up that field message again and then re-enqueue it for further processing you can have an upper bound so of course you see over here we have a retry count so you can tell this job that if a row has been or a notification has been retried four or five times forget about it you don't have to enqueue it again so you can have logic like that up to make it more to make it less busy

How to monitor queued notifications?

A key metric to monitor is the total number of queued notifications. If the number is large, the notification events are not processed fast enough by workers. To avoid delay in the notification delivery, more workers are needed. Figure 12 (credit to [7]) shows an example of queued messages to be processed.