Epic: Real Time Backend Project Milestone 1

Notification Delivery Service

Definition: The Notification Delivery Service is a system responsible for delivering notifications to users across various channels such as SMS, email, and mobile push notifications.

Functional Requirements

- Support three types of notification formats: mobile push notifications, SMS messages, and emails.
- Allow users to opt out of receiving notifications.
- Enable client services to create different types of notifications (e.g., promotions, newsletters, profile status updates).
- Allow notification recipients to unsubscribe from client service notifications and set preferences to receive only specific types of notifications.
- Handle notification delivery failures and retry when necessary.
- Provide basic delivery analytics (e.g., success/failure rates).
- **Massive Notification Pushes:** The system should support large-scale broadcast messages(eg, sending the same message, to multiple end users).
-

Non-Functional Requirements

- Scalability to handle millions of users and services.
- High availability and reliability.
- Low latency in message delivery.
- Fault tolerance and graceful degradation.

What It Does Not Support

- **Notification Priority:** Introducing priority levels could complicate the system. If allowed, most services might label their messages as high priority. Handling this fairly might require a paid prioritization system. Currently, the system is intended to remain free.
- **In-depth Analytics Services:** Only basic delivery metrics are supported.
- No Notification scheduling

Audience

- Medium to large software companies integrating this service into their applications to notify users.
- End users receiving notifications (customers, clients, or users of those applications).

Metrics and System Capacity

Scale and Usage Assumptions

- The system is designed to support **1 million client services**.
- Daily active client services = 20% of total services. This aligns with SaaS industry benchmarks where 15–30% of services are active daily (e.g., Twilio, SendGrid).
    - **Daily Active Services (DAC)** = 1,000,000 × 0.2 = **200,000 DAC**.

Notification Volume Estimation

- Service usage pattern:
    - Low-volume services (80%) send ~50 notifications/day.
    - High-volume services (20%) send 1,000–10,000 notifications/day.
- Weighted average:
    - (0.8 × 50) + (0.2 × 1,000) = 40 + 200 = **200 notifications/service/day**.
- **Total Daily Notifications** = 200 notifications/service/day × 200,000 DAC = **40 million notifications/day**.

Daily Active Notification Recipients

- Assuming users receive ~5 notifications per day:
    - **8 million unique users** = 40 million notifications / 5 notifications/user.

Request Throughput (RPS)

- Total notifications per day = 40,000,000.
- Total seconds/day = 24 × 60 × 60 = 86,400 seconds.
- Average RPS = 40,000,000 / 86,400 ≈ **463 RPS**.
- Applying a **Peak Factor** of 10 to account for burst load:
    - **Peak RPS ≈ 4,630**.
    - Anticipated extreme traffic peaks: **~10,000 RPS**.

User Retention

- Targeting **80% 7-day retention**. This is typical for critical infrastructure services like notification systems, where switching costs are high and reliability is essential.

Storage Requirements

Client Service Metadata

- **Each service stores:**

    - **Service name (64B)**
    - **API token (256B)**
    - **Notification types (up to 10)**
        - **Each type name: 64B**
        - **Each type description: 256B**
- **Total storage per service:**
    - **Metadata: ~64B (name) + 256B (token) = 320B**
    - **Notification types: 10 × (64 + 256) = 3,200B**
    - **Total ≈ 3.5KB/service**
- **For 1 million services: 1,000,000 × 3.5KB = ~3.5GB**

End-User Preferences

- **Metadata + channel preferences + type preferences ≈ 1KB/user**
    - **8 million users × 1KB = 8GB**

Notification Data

- **40 million notifications/day × 1KB = 40GB/day**
- **Annual storage = 40GB × 365 = 14.6TB/year**

Network Bandwidth Requirements

- **Inbound Bandwidth** = 1KB/request × 4,630 RPS = **4.63 MB/s** → ~12TB/month.
- **Outbound Bandwidth** = 500B/response × 4,630 RPS = **2.31 MB/s** → ~6TB/month.

Architecture: High-Level Design

1. Client Service Registration Flow

When a client service wants to use the Notification Delivery Service:

- **The client sends a registration request with:**
    - **Service name**
    - **Supported notification types (maximum 10), each including:**
        - **A name (max 64 characters)**
        - **A description (max 256 characters)**

- **The system:**
    - **Generates a unique API token**
    - **Stores client service metadata in the database**
    - **Returns the client ID and API token to the client**

---

2. Notification Service

The Notification Service is the main entry point for:

- **Service registration**
- **Notification requests**

Clients send a "Send Notification" API request containing:

- **Content**
- **Recipient ID**
- **Notification type**
- **Delivery channel (e.g., email, SMS, push)**

Responsibilities of the Notification Service:

- **Authenticate requests using API tokens**
- **Enforce rate limits**
- **Validate payloads (e.g., valid phone numbers or email formats)**
- **Query the User Preference Service to ensure:**
    - **The end-user has opted in for the specified notification type and channel**

- **If allowed, enqueue the notification into the appropriate message queue for asynchronous processing**

---

3. User Preference Service

The User Preference Service is responsible for managing and enforcing user-specific preferences for receiving notifications.

Key Features:

- **Stores per-client user preferences, including:**
  - **Preferred channels (email, SMS, push)**
  - **Allowed or blocked notification types**
- **Provides APIs for:**
  - **Querying preferences (used during notification processing)**
  - **Updating preferences (triggered when users follow a "manage preferences" or "unsubscribe" link)**

---

4. Message Queues

We use separate message queues for each delivery channel (Email, SMS, Push).

Reasons:

- **Decouples system components for scalability and reliability**
- **Buffers load spikes, preventing bottlenecks**
- **Isolates failures a disruption in one channel does not affect others**

Technology:
Use Kafka, a highly reliable, distributed event streaming platform with strong delivery guarantees and message durability.

---

5. Workers

Workers are stateless services that:

- **Poll channel-specific queues**
- **Transform messages if necessary**
- **Retry mechanism upon failure.**
- **Send them to the appropriate third-party delivery provider**

Workers can be scaled independently to handle traffic surges per channel.

---

6. Third-Party Delivery Services

These services are responsible for the final delivery of notifications to the end-users.

a. Push Notifications

- **iOS Devices:**
  Use Apple Push Notification Service (APNs), Apple's native push system for secure, efficient delivery.

- **Android Devices:**
  Use Firebase Cloud Messaging (FCM), the standard Google service for push notifications to Android apps.

b. SMS

- **Provider: Twilio**
  - **Reason: Offers global coverage, robust APIs, delivery status callbacks, and high reliability**
  - **Alternatives: Nexmo (Vonage), Plivo**

c. Email

- **Provider: SendGrid**

    - **Reason: Trusted platform with high deliverability, rich APIs, and built-in analytics (open rate, bounce rate, etc.)**

    - **Alternatives: Mailchimp, Amazon SES**

---

7. End-User Preferences

- **All notifications include a “Manage Preferences” or “Unsubscribe” link.**

- **When users click the link, they are directed to a UI backed by the User Preference Service.**

- **They can:**

    - **Opt out of specific notification types**

    - **Choose a preferred delivery channel.**
- **Changes are persisted and used to enforce filtering in future notification deliveries.**

Data Layer Architecture

---

1. Client Service Metadata

- **Purpose**
  Stores metadata for each client service using the notification system. This includes service name, API token, and supported notification types.
- **Database:** MongoDB
- **Reason:** Flexible schema fits well since notification types can vary by client. No strong relational joins are required, good for fast lookups and scalability, and the document-based structure aligns with client-based grouping.
- **Schema Structure:**

```JSON
{
  "_id": "client_id",
  "service_name": "AcmeCorp",
  "api_token": "abcd1234...",
  "notification_types": [
    {
      "name": "order_update",
      "description": "Order status change"
    },
    {
      "name": "promo",
      "description": "Promotional offers"
    }
  ]
}
```

- **Cache (between Notification Service and DB):**
  - **Caching:** Redis
  - Cache client service metadata (ID, token, notification types)
  - Speeds up frequent validation/auth checks

---

2. End User + Preferences

- **Purpose**
  Stores notification preferences for each user per client service. Specifies allowed channels and notification types.
- **Database:** MongoDB
- **Reason:** Preferences vary per user and per service. The document model is ideal for nesting preferences and fast reads/writes. No heavy relational joins required.
- **Schema Structure:**

```JSON
{
  "_id": "user_id",
  "preferences": {
    "client_id_1": {
      "channels": ["email", "sms"],
      "allowed_types": ["promo", "order_update"]
    },
    "client_id_2": {
      "channels": ["push"],
      "allowed_types": ["security_alert"]
    }
  }
}
```

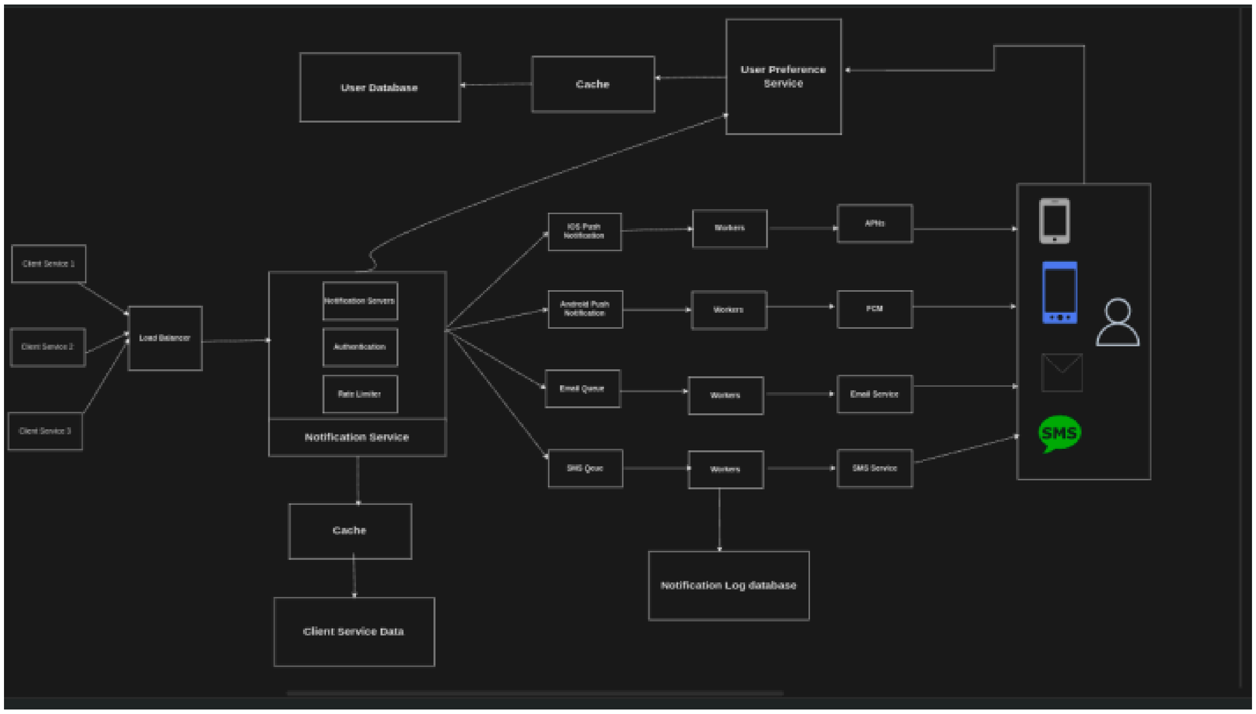
- **Cache (between User Preference Service and DB):**
  - **Caching:** Redis
  - User preference by (user_id, client_id)

---

3. Notification Logs

- **Purpose**
  Ensure delivery guarantees by persisting every notification event (including content, status, and timestamps). This ensures notifications can be retried in case of failure and are never lost. Retry logic uses this table to track failures and reschedule delivery attempts. Notifications are queued using Kafka, but logged in PostgreSQL for durability.
- **Database:** PostgreSQL
- **Reason:** Strong durability guarantees, ACID compliance, and structured querying support make PostgreSQL suitable for persistent logs and retries.
- **Schema Structure:**

JSON
```
Cnotification_logs (
  id UUID PRIMARY KEY,
  client_id UUID,
  user_id UUID,
  notification_type TEXT,
  channel TEXT,
  content JSONB,
  status TEXT, -- sent, failed, retry_pending, etc.
  retry_count INT,
  created_at TIMESTAMP,
  updated_at TIMESTAMP
);
```


User Database
Cache
User Preference Service
Client Service 1
Client Service 2
Client Service 3
Load Balancer
Notification Servers
Authentication
Rate Limiter
Notification Service
IOS Push Notification
Android Push Notification
Email Queue
SMS Queue
Workers
Workers
Workers
Workers
APNs
FCM
Email Service
SMS Service
SMS
Cache
Client Service Data
Notification Log database