Notification Service – System Design (Part 1)

1. Problem Understanding

The goal is to design a centralized Notification Service capable of sending and managing notifications across multiple channels (Email, SMS, Push). The service must handle requests from other systems, support multiple delivery channels, prevent duplicates (idempotency), track status, and allow future extensibility.

2. Design Goals

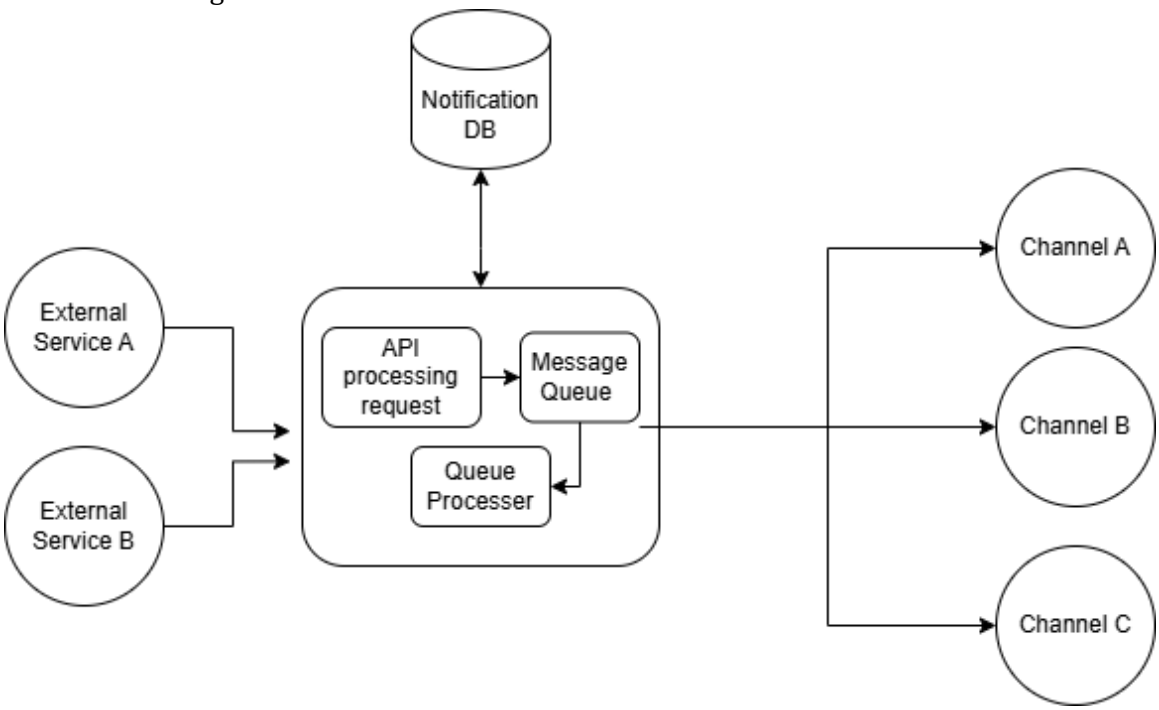
The design aims to achieve scalability, reliability, extensibility, and maintainability. It should ensure that notifications are delivered efficiently, failures are retried gracefully, and new communication channels can be added without affecting existing components.

3. High-Level Architecture

The system follows an event-driven microservice architecture with queue-based processing. This approach decouples the API layer from background processing, enabling asynchronous delivery, scalability, and resilience against service failures.

Key components include an API Gateway, PostgreSQL database, Redis cache, message queue (BullMQ), background workers, and pluggable channel handlers (Email, SMS, Push).

Architecture Diagram:



4. Components and Responsibilities

Component	Purpose	Why This Choice
API Gateway	Validates input, checks idempotency via Redis, stores notifications, enqueues jobs.	Separates client requests from business logic, improves scalability and maintainability.
Redis	Caches notification requests for idempotency.	In-memory speed ensures duplicate prevention.
PostgreSQL	Stores notifications, templates, and logs.	Reliable ACID database with JSONB for flexibility.
Message Queue (BullMQ)	Decouples processing from requests, handles retries.	Improves fault tolerance and throughput.
Workers	Consume jobs, render templates, send via channels.	Allows horizontal scaling and fault isolation.
Channel Handlers	Define Email, SMS, and Push interfaces.	Enables easy extension to new channels like WhatsApp or Slack.

5. Data Model

The main tables include:
- Notifications: Stores every notification request received from client systems. Tracks status, retries, and ensures idempotency to prevent duplicate sends.

stores user_id, channel, template_id, data, status, idempotency_key, error_message, retry_count, created_at, updated_at.

- Templates: Defines reusable message templates per channel and notification type. Supports both subject/body and placeholder variables for dynamic rendering.

stores id, channel, name, subject/body placeholders, created_at, updated_at.

- Logs: Stores logs for each delivery attempt, enabling auditing and retry tracking. Each log links to a single notification and tracks provider response.

stores id, notification_id, attempt, status, error_message, provider_response, created_at.

This schema ensures tracking and auditing capabilities while maintaining flexibility for new notification types.

6. Notification Flow

The flow is split into synchronous and asynchronous stages:
1. Client sends a notification request to the API.
2. API validates input, checks idempotency in Redis, saves to DB as 'Pending', and enqueues the job.
3. Background workers consume the job, fetch the template, render it with dynamic data, and send through the proper channel.
4. Worker updates notification status in the database to 'Sent' or 'Failed'.
This separation ensures responsiveness and reliability even under heavy load.

7. Technology Stack & Rationale

The selected technology stack balances **development speed**, **performance**, **scalability**, and **ease of maintenance**.
Each choice is aligned with the problem requirements — handling asynchronous workloads, ensuring reliability, and supporting extensibility for future channels.

Technology	Purpose	Why This Choice
Node.js + TypeScript	Core backend framework	Node.js offers non-blocking, event-driven I/O that is ideal for real-time and high-throughput workloads like notifications. TypeScript adds static typing, reducing runtime errors and improving maintainability.
Express.js	API layer	Lightweight and minimalist framework for building RESTful APIs. Its middleware pattern provides flexibility for validation, error handling, and security.
PostgreSQL	Main database	Chosen for its reliability, ACID compliance, and JSONB support, allowing structured and semi-structured data storage (for templates and dynamic variables). It scales well vertically and horizontally.
Redis	Caching and idempotency store	In-memory speed makes it ideal for preventing duplicate sends and managing short-lived data such as idempotency keys or rate limits.
BullMQ (Redis-based queue)	Message queue	Provides asynchronous job handling, retry logic, and persistence with minimal setup. Tight integration with

Technology	Purpose	Why This Choice
		Redis simplifies deployment and monitoring.
Handlebars.js	Template rendering engine	Allows defining flexible, human-readable templates with variable placeholders. Makes it easy for non-engineers (like marketing teams) to modify templates safely.
Jest	Testing framework	Ensures code reliability with unit and integration tests. Offers good TypeScript support and fast test execution.
Docker	Containerization	Enables consistent runtime environments, simplifies local development, and facilitates CI/CD pipelines and cloud deployment.

8. Scalability & Fault Tolerance

Scalability is achieved via stateless API servers, multiple worker instances, database read replicas, and a Redis cluster. Reliability is ensured through durable queues, transactional database writes, and retry mechanisms with exponential backoff. Monitoring tools like Prometheus and alerts on failure rates support proactive maintenance.

9. Future Extensibility

New channels can be added easily through the strategy/factory pattern. To add a channel (e.g., WhatsApp): implement the INotificationChannel interface, register it in the channel factory, and create new templates, no core code changes required.