Scalable Notification System Design

1. Requirements

Functional Requirements:

- **Notification Types:** Support various notification types (Email, SMS, Push notifications).
- **Retry Mechanism:** Retry failed notifications.
- **Rate Limiting:** Handle rate limiting to avoid overwhelming external services.

Non-Functional Requirements:

- **Scalability:** Handle a large number of notifications.
- **Low Latency:** Notifications should be delivered with minimal delay.
- **Reliability:** Ensure notifications are delivered even in case of partial system failures.
- **Extensibility:** Easy to add new types of notifications.

2. System Components

1. **API Gateway:**
    - Provides RESTful endpoints for creating and managing notifications.
    - Authentication and authorization for clients.
2. **Notification Service:**
    - Core service responsible for sending notifications.
    - Interacts with external services (Email, SMS, Push) via providers.
    - Implements retry logic and rate limiting
3. **Worker Queue:**
    - Queues notifications to be processed by the Notification Service.
    - Ensures that notifications are processed asynchronously.
4. **Notification Providers:**
    - External services responsible for delivering notifications (e.g., SendGrid for email, Twilio for SMS).
    - Each provider will have its own interface and implementation.

5. **Monitoring & Logging:**
    - Monitor the health of the notification system.
    - Log failed notifications for auditing and retry purposes.

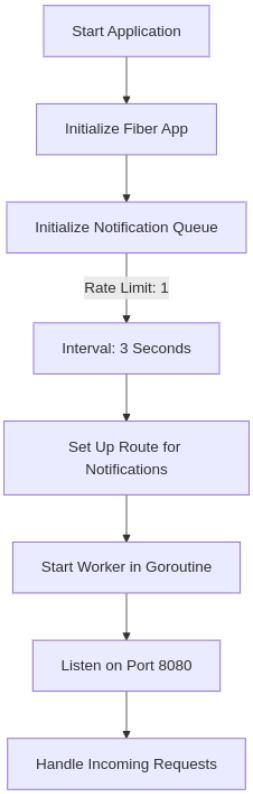
3. Data Flow

1. **Client Interaction:**
    - Clients interact with the API Gateway to create notifications.
    - API Gateway forwards the requests to the Notification Service.
2. **Notification Processing:**
    - Notification Service validates the notification.
    - If valid, the notification is placed on a Worker Queue.
3. **Worker Queue:**
    - Notifications are processed asynchronously from the queue.
    - Notification Service retrieves the notification and sends it via the appropriate provider.
    - If the notification fails, it is either retried or logged for future retry.
4. **Rate Limiting & Throttling:**
    - Rate limiting logic ensures that the Notification Service doesn't exceed the rate limits of the external providers.
5. **Logging & Monitoring:**
    - Logs the notification status and monitors the performance of the system.

4. Technology Stack

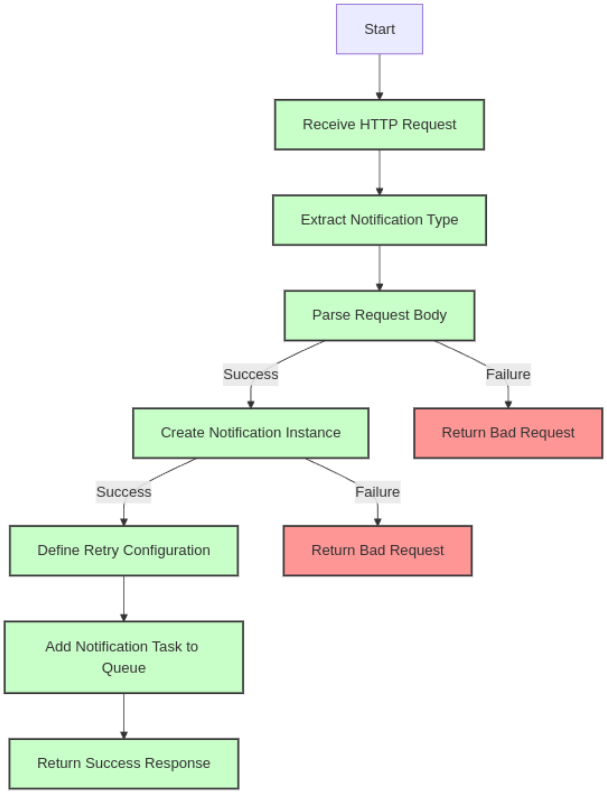
- **Programming Language:** Golang
- **Web Framework:** Fiber
- **Logging & Monitoring:** Prometheus, Grafana for monitoring, and structured logging using Logrus or Zerolog.
- **Notification Providers:** Mock implementations for email, SMS, push notifications, which can be replaced by actual providers (like SendGrid, Twilio).

Flowchart 1: Main Application Flow



- Start Application: The application begins execution.
- Initialize Fiber App: A Fiber web application is initialized.
- Initialize Notification Queue: The notification queue is created with a rate limit of 1 task every 3 seconds.
- Set Up Route for Notifications: A route is set up to handle incoming notification requests.
- Start Worker in Goroutine: The worker for processing notifications is started in a separate goroutine.
- Listen on Port 8080: The application listens for incoming requests on port 8080.
- Handle Incoming Requests: The application handles incoming HTTP requests.

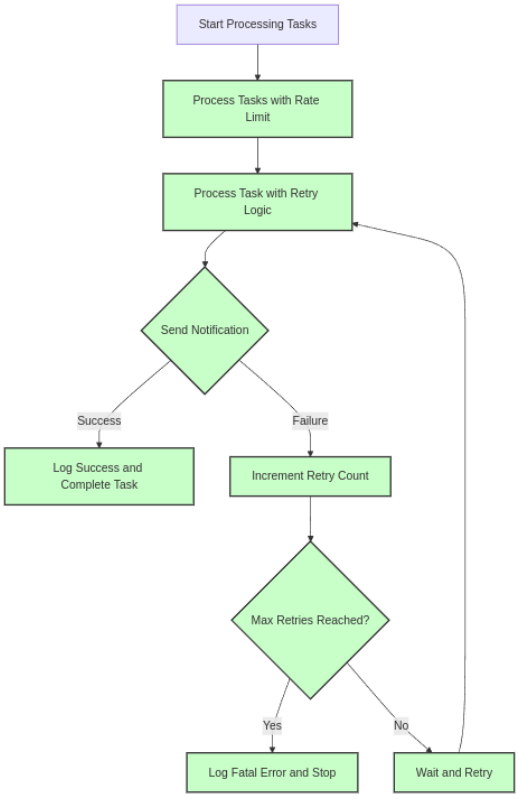
Flowchart 2: Notification Handling Flow



- Start: The process begins when an HTTP request is received.
- Receive HTTP Request: The handler receives the request containing the notification type and payload.
- Extract Notification Type: The notification type is extracted from the URL parameters.
- Parse Request Body: The body of the request is parsed to extract the notification payload.
    - Success: If parsing is successful, the flow proceeds to create a notification instance.
    - Failure: If parsing fails, a bad request response is returned, and the error is logged.
- Create Notification Instance: A factory is used to create a notification instance based on the extracted type.
    - Success: If the instance is created successfully, the retry configuration is defined.
    - Failure: If there is an error creating the notification, a bad request response is returned, and the error is logged.
- Define Retry Configuration: The maximum number of retries and the delay between retries are defined.

- Add Notification Task to Queue: The notification task is added to the queue with the defined retry logic.
- Return Success Response: A success response is sent back to the client indicating that the notification has been queued.

Flowchart 3: Notification Queue Processing Flow



- Start Processing Tasks: The queue begins processing tasks.
- Process Tasks with Rate Limit: The queue processes up to the defined rate limit of tasks.
- Process Task with Retry Logic: Each task is processed with retry logic.
- Send Notification: The notification is attempted to be sent.
    - Success: If the notification is sent successfully, a success message is logged, and the task is completed.
    - Failure: If there is an error sending the notification, the retry count is incremented.
- Max Retries Reached?: A check is performed to see if the maximum number of retries has been reached.

    - Yes: If the maximum retries are reached, a fatal error is logged, and the process stops for that task.
    - No: If retries are still available, the process waits for the specified delay and retries sending the notification.
- These flowcharts provide a comprehensive view of the architecture and flow of the notification service application.