Backend Logic for Handling Notifications without a Queue

Overview

This document outlines the backend logic for handling notifications in the Neighborly app using Firebase for push notifications. The queue system will be eliminated to simplify the process, reduce latency, and improve the user experience. Notifications will be directly pushed to users and stored in a database for record-keeping and retrieval purposes.

Objectives

1. **Real-Time Push Notifications**: Deliver notifications to users instantly using Firebase Cloud Messaging (FCM).
2. **Persistent Storage**: Store all notifications in the database for audit, retrieval, and display within the app.
3. **Endpoint Management**: Provide endpoints to manage user notifications, including fetching, updating, and marking notifications as read.

Components

1. **Firebase Cloud Messaging (FCM)**: Used for delivering push notifications to the Neighborly app.
2. **Database**: MongoDB will be used to store notification details persistently.

Notification Workflow

1. **Notification Creation**:
    - When a new event triggers a notification (e.g., new message, event reminder), the backend service will immediately prepare the notification payload.
    - The payload includes details such as the notification title, message, target user(s), and any additional data (e.g., post ID, event ID) required for the app to handle the notification.
2. **Push Notification Delivery**:
    - The backend sends the notification payload to Firebase, which handles the delivery to the specified user(s) based on their Firebase registration tokens.
    - The payload can include a `click_action` that specifies the activity or page in the app that should be opened when the user interacts with the notification.
3. **Notification Storage**:
    - Simultaneously, the backend stores a copy of the notification in the MongoDB database. The notification schema should include:
        - `notificationId`: Unique identifier for the notification.
        - `userId`: Identifier for the user receiving the notification.
        - `title`: Title of the notification.
        - `message`: Message content of the notification.
        - `data`: Any additional data included in the notification (e.g., post ID, event ID).
        - `timestamp`: Time when the notification was created.
        - `status`: Status of the notification (e.g., unread, read).
        - `firebaseMessageId`: The Firebase message ID (optional, for tracking purposes).
4. **Handling User Interaction**:
    - When the user interacts with the notification, the app will navigate to the relevant screen (e.g., post detail, event page) and may perform additional REST API calls to load the required data.
    - The app can also mark the notification as read by calling an endpoint on the backend.
5. **Notification Retrieval and Management**:
    - The backend will provide endpoints to:
        - Fetch all notifications for a user.
        - Update the status of a notification (e.g., marking as read).

        - Delete notifications (optional, based on retention policy).

6. **Endpoint Specifications**:
    - **GET /notifications**: Retrieve all notifications for the authenticated user.
    - **PATCH /notifications/{notificationId}**: Update the status of a specific notification (e.g., mark as read).
    - **DELETE /notifications/{notificationId}**: Delete a specific notification (if required).
7. **Error Handling and Logging**:
    - Implement error handling to manage potential issues such as failed notification delivery or database errors.
    - Log all notification events, including successful deliveries and failures, for monitoring and debugging purposes.

Database Schema for Notifications (MongoDB)

```
{
  "_id": "ObjectId",
  "notificationId": "String",
  "userId": "String",
  "title": "String",
  "message": "String",
  "data": {
    "postId": "String",
    "eventId": "String",
    "messageId": "String",
    "commentId": "String",
  },
  "timestamp": "ISODate",
  "status": "String",
  //e.g.,
  "unread",
  "read""firebaseMessageId": "String"//Optional
}
```

Considerations

1. **Scalability**: Ensure the system can handle a large volume of notifications, particularly during peak times.
2. **Data Retention**: Define a retention policy for notifications in the database. You might choose to delete older notifications after a certain period.
3. **Security**: Secure the endpoints to ensure only authenticated users can access and manage their notifications.

Future Enhancements

- **Batch Processing**: Consider adding batch processing for notifications in the future if the volume grows.
- **User Preferences**: Allow users to customize their notification preferences (e.g., opting out of certain types of notifications).