API Design Decisions, Assumptions, and Reasoning

1. Overview

This document outlines the design decisions, assumptions, and reasoning followed in implementing a real-time employee recognition GraphQL API for Nest Solutions. The API allows employees to send kudos with messages and emojis, supports visibility rules, and enables real-time notifications via subscriptions. It is implemented using Apollo Server v5, Express, TypeScript, and WebSocket (graphql-ws).

2. API Schema Design

The GraphQL schema is modular, extensible, and supports the core requirements defined in the assessment. Each type is documented with meaningful descriptions.

- `User`: Represents an employee with `id`, `name`, `role`, and `team`.

- `Recognition`: Represents a kudos sent from one user to another with fields for message, emoji, visibility, sender, recipient, and timestamp.

- `Role`: Enum to distinguish access rights between EMPLOYEE, MANAGER, HR, and LEAD.

- `Visibility`: Enum to control message visibility (PUBLIC, PRIVATE, ANONYMOUS).

- Queries: `users`, `recognitions(userId)`, `myRecognitions(userId)`

- Mutations: `sendRecognition(...)`

- Subscriptions: `newRecognition` for real-time updates

3. Data Model

The data model includes two in-memory entities:

- • User
    - - id: string
    - - name: string
    - - role: enum (EMPLOYEE | MANAGER | HR | LEAD)
    - - team: string
- • Recognition
    - - id: string
    - - senderId: string
    - - recipientId: string
    - - message: string
    - - emoji: string

- - visibility: enum (PUBLIC | PRIVATE | ANONYMOUS)
- - createdAt: timestamp

4. Assumptions

- Authentication is simulated using `userId` passed as a query/mutation argument.

- Real-time updates are implemented using GraphQL subscriptions; fallback polling (every 10 minutes) is documented but not implemented.

- HR and MANAGER roles can view all recognitions, while EMPLOYEE and LEAD roles have restricted views.

- Recognition visibility logic is enforced at resolver level, not via field-level permissions.

- System is designed to be compatible with future Slack/MS Teams integration via webhooks.

5. Reasoning

The design favors simplicity, modularity, and extensibility. Using TypeScript with in-memory data allows rapid iteration without DB setup, aligning with the assessment's focus on architecture and critical thinking. GraphQL Subscriptions were prioritized to meet the real-time requirement. Role-based access control is enforced at query/mutation level using a simulated `getCurrentUser()` function. The schema was built to support future analytics (e.g., recognitions by team or keyword) and extensibility (e.g., badges, likes, comments) with minimal changes.

Kudos Project - Design Decisions, Assumptions, and Reasoning

Github URL - https://github.com/vishruth1997/nest-kudos-api

1. GraphQL Schema Design

- Defined types: User, Recognition, along with Role and Visibility enums.
- Queries: `users`, `recognitions`, and `myRecognitions`.
- Mutations: `sendRecognition` allows employees to send kudos.
- Subscriptions: `newRecognition` supports real-time delivery.
- Every field and operation includes descriptions for clarity and maintainability.
- Schema is modular and designed to be extensible for future fields like badges, likes, and comments.

2. Role-Based Access Control (RBAC)

- Roles defined: EMPLOYEE, MANAGER, HR, LEAD.
- Access rules enforced in resolvers using a utility function `getCurrentUser()`.
- HR and MANAGER can access all recognitions.
- EMPLOYEEs can only view public recognitions and their own private/anonymous ones.
- Simulated authentication is used by passing `userId` in the query/mutation.

3. Real-Time Subscription with Fallback

- Implemented real-time subscriptions using `graphql-ws` and WebSocketServer.
- Subscriptions are triggered when `sendRecognition` publishes a new event.
- If real-time fails, the system can fall back to polling every 10 minutes.
- WebSockets are integrated into Apollo Server v5 using `useServer`.

4. Mock Data Strategy

- In-memory arrays used for users and recognitions (`mockData.ts`).
- Enables quick testing and submission without external dependencies.
- Users are assigned realistic roles and teams for RBAC testing.

5. Authentication Assumptions

- No full authentication system (e.g., JWT, sessions) is implemented.
- Assumed that user identity is passed as `userId` argument for simplicity.
- This approach satisfies the requirement of demonstrating RBAC behavior.

6. Server Architecture

- Apollo Server v5 used with Express and WebSocket integration.
- Shared GraphQL schema used for both HTTP and WebSocket transport.
- Express middleware includes CORS and JSON body parsing.
- Graceful shutdown of WebSocket server handled via Apollo plugin.

7. Analytics and Future Extensibility

- Data model supports analytics by team and message keywords.
- Schema can be extended to support:
  - Badges
  - Likes and reactions
  - Comments
- Design is modular to support integration with Slack or Microsoft Teams.
- `team` field on User and `message` field on Recognition enable analytical queries.

8. Final Notes

- The project structure is modular and well-commented.
- README includes setup instructions, usage examples, and clear documentation.