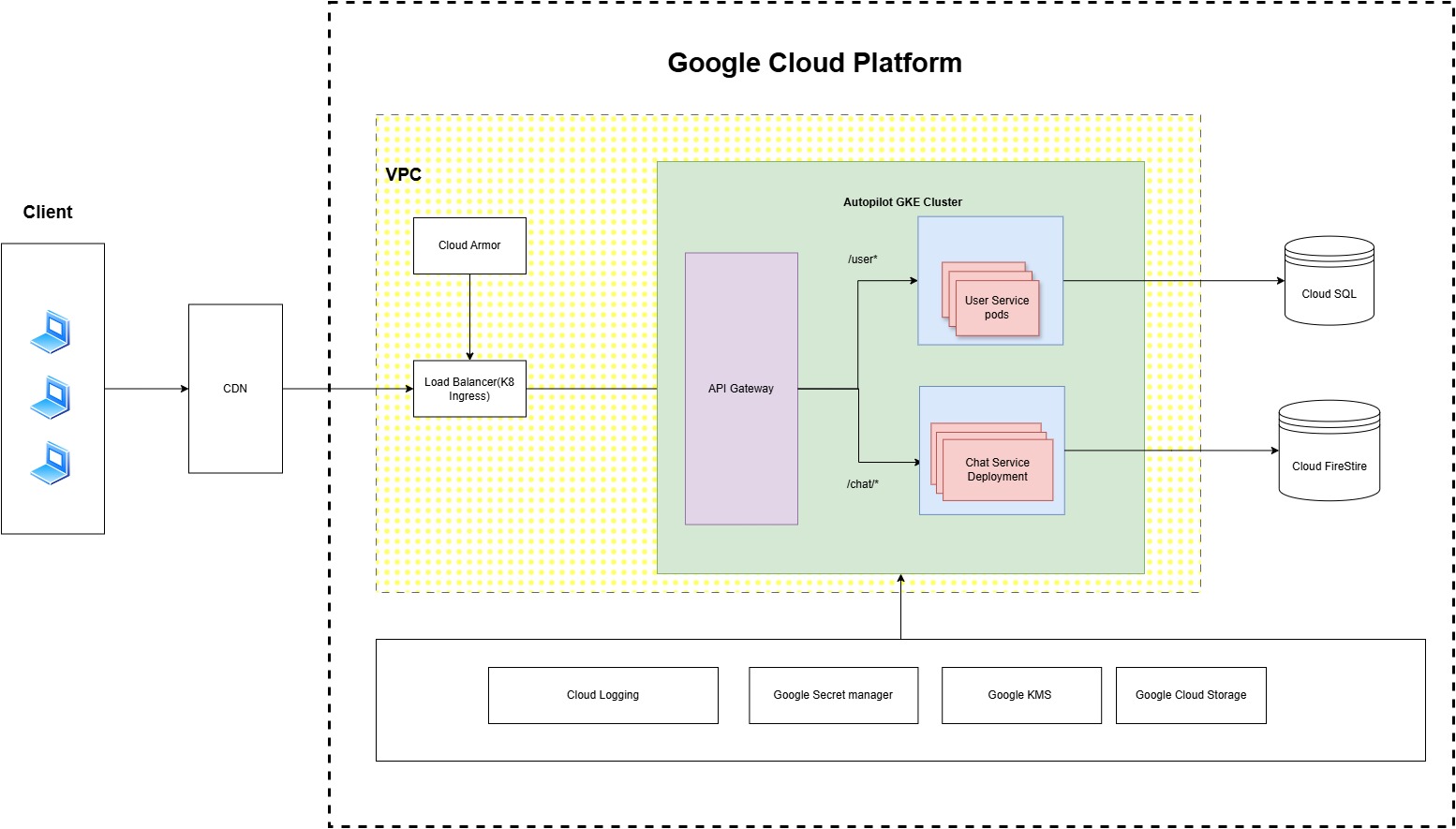
**Multi-Tenant SaaS Application Architecture**

**1. Overview**

This solution is based on two Node.js microservices, one for user, project management, authentication (User Service) and one for real-time chat functionality (Chat Service). These microservices will be hosted on Google Kubernetes Engine (GKE) to handle scaling, availability, and security requirements. The React front-end communicates with these APIs to provide a seamless user experience for project management and collaboration.

**2. Microservices API Architecture**



**3. Microservices Overview**

**3.1 User and project management Service**

This Service is responsible for user registration, authentication, and role-based access control (RBAC). It will include the following features:

* **User Registration:** Allows users to sign up for the platform.
* **Authentication:** Verifies the identity of users using JWT tokens.
* **Authorization (RBAC):** Defines and enforces roles for users (admin, member, etc.) within each organization.
* **User Management:** Admin users can manage user permissions and roles within their organization.
* **Project and Organization Management**

**3.2 Chat Service**

The Chat Service provides real-time communication capabilities, allowing users to send and receive messages instantly. This service will:

* **Real-time Chat:** Implements WebSocket-based communication to allow users to join chat rooms and send/receive messages in real time.
* **Message Persistence:** Stores messages in a database for history and retrieval.
* **Organization-based Chat Rooms:** Each organization can have separate chat rooms to ensure data isolation.

**4. Database Schema Design**

To support multi-tenancy, the database schema needs to isolate data between organizations while allowing for efficient querying and data management.

**4.1) Database Schema User service (SQL)**

**4.1.1) Users Table:** This table stores information about users, including their authentication data, role, and association with organizations.

CREATE TABLE IF NOT EXISTS Users (

id VARCHAR(255) PRIMARY KEY, -- Unique user ID

email VARCHAR(255) UNIQUE NOT NULL, -- User's email (must be unique)

passwordHash VARCHAR(255) NOT NULL, -- Hashed password

role VARCHAR(50) NOT NULL, -- Role of the user (admin, member, etc.)

organizationId VARCHAR(255) NOT NULL, -- ID of the organization the user belongs to

createdAt TIMESTAMP DEFAULT CURRENT\_TIMESTAMP, -- Timestamp of user creation

updatedAt TIMESTAMP DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP, -- Timestamp of last update

FOREIGN KEY (organizationId) REFERENCES Organizations(id) -- Link to organization

);

**4.1.2) Organizations Table:** To manage users within different organizations, this table stores organization details.

CREATE TABLE IF NOT EXISTS Organizations ( id VARCHAR(255) PRIMARY KEY, -- Unique organization ID name VARCHAR(255) NOT NULL, -- Organization name createdAt TIMESTAMP DEFAULT CURRENT\_TIMESTAMP -- Timestamp of organization creation );

**4.1.3) Projects Table:** The Projects Table stores the details of projects within an organization. Each project will be linked to a specific organization.

CREATE TABLE IF NOT EXISTS Projects (

id VARCHAR(255) PRIMARY KEY, -- Unique project ID

name VARCHAR(255) NOT NULL, -- Name of the project

description TEXT, -- Description of the project

organizationId VARCHAR(255) NOT NULL, -- ID of the organization the project belongs to

createdBy VARCHAR(255) NOT NULL, -- User ID of the person who created the project

createdAt TIMESTAMP DEFAULT CURRENT\_TIMESTAMP, -- Timestamp of project creation

updatedAt TIMESTAMP DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP, -- Timestamp of last update

FOREIGN KEY (organizationId) REFERENCES Organizations(id), -- Link to organization

FOREIGN KEY (createdBy) REFERENCES Users(id) -- Link to the user who created the project

);

**4.2) Database Schema Chat Service (NOSQL)**

* + 1. **Rooms Collection:** The rooms collection stores all chat rooms for each organization. Each document represents a specific room.

**Document ID**: roomId (unique ID for the room)

**Fields**:

1. name: string (name of the chat room)
2. organizationId: string (ID of the organization the room belongs to)
3. createdBy: string (user ID of the person who created the room)
4. description: string (description of the room)
5. createdAt: timestamp (the date and time the room was created)

{

"name": "Project Discussion",

"organizationId": "12345",

"createdBy": "user\_id\_1",

"description": "Discussion room for the XYZ project",

"createdAt": "2024-12-15T00:00:00Z"

}

* + 1. **Messages Collection:** The messages collection stores messages within each room. Each document represents an individual message in a specific room.

**Document ID:** messageId (unique ID for the message)

**Fields**:

* + - roomId: string (the ID of the room where the message was sent)
    - senderId: string (user ID of the sender)
    - message: string (content of the message)
    - timestamp: timestamp (the date and time the message was sent)

{

"roomId": "room\_123",

"senderId": "user\_id\_1",

"message": "Hello everyone!",

"timestamp": "2024-12-15T00:01:00Z"

}

**4.3) Data Isolation**

* + Use **organizationId** in every table to isolate data between organizations.
  + Ensure that every query checks for the correct organizationId to prevent cross-tenant data access.
  + Each user and project is associated with a specific organizationId, ensuring complete data isolation between organizations.

**5. Authentication and Role-based Access Control (RBAC)**

The User Service will handle authentication and authorization using JWT (JSON Web Tokens).

**5.1 Authentication Flow**

1. **User Registration:** A user can register with an email and password.
2. **Login:** Upon successful login, a JWT token is issued with claims for the user’s roles and organizationId.
3. **Token Verification:** For protected routes (like accessing chat rooms or projects), the token is verified for authenticity and organizationId matching.

**5.2 Role-based Access Control (RBAC)**

Roles can be assigned to users when they are created or modified. Common roles include:

* **Admin:** Full access to all resources Across all organizations.
* **Org\_Admin:** Full access to all resources within organization.
* **Member:** Limited access, typically to view and interact with projects or chat rooms.

**6. Real-Time Chat Using WebSocket**

The Chat Service uses WebSocket for real-time communication.

**6.1 WebSocket Server Setup**

* A WebSocket server will be set up within the Node.js service to handle real-time communication.
* Upon connecting, users are authenticated using their JWT token, which is passed as part of the WebSocket connection request.
* Users will join rooms based on projects or teams, and messages will be broadcast to all connected users in the room.

**6.2 Message Persistence**

* Messages will be stored in the database with a reference to the organizationId, roomId and senderId.
* This ensures that messages can be retrieved later while maintaining multi-tenant isolation.

**7. API Endpoints**

**7.1 User Service API Endpoints**

* + **POST /users/register**: User registration endpoint. Allows a new user to sign up by providing an email, password, role, and organization ID.
  + **POST /users/login**: User login endpoint. Authenticates the user using their email and password, returning a JWT token on success.
  + **GET /users/{id}**: Get user details by ID. Fetches information about a specific user using their unique ID.
  + **PUT /users/{id}**: Update user details. Allows modification of user details such as email and role by providing the user ID.
  + **DELETE /users/{id}**: Delete user by ID. Removes a user from the system using their unique ID.
  + **PUT /users/{id}/role**: Update user role. Changes the role (e.g., admin, member) of a user identified by their ID.
  + **POST /projects**: Create a new project for an organization. Allows creation of a project by providing project details such as name, organization ID, created by user ID, and description.
  + **GET /projects/{id}**: Get a project’s details by ID. Retrieves detailed information about a specific project identified by its unique ID.
  + **PUT /projects/{id}**: Update a project’s details. Allows modification of project details like name and description.
  + **DELETE /projects/{id}**: Delete a project by ID. Removes a project from the system using its unique ID.
  + **POST /organizations**: Create a new organization. Allows the creation of an organization by providing the organization name.
  + **GET /organizations/{id}**: Get an organization’s details by ID. Fetches information about a specific organization using its unique ID.
  + **PUT /organizations/{id}**: Update an organization’s details. Allows modification of an organization’s name.
  + **DELETE /organizations/{id}**: Delete an organization by ID. Removes an organization from the system using its unique ID.

Attached is the Open Api document for all the endpoints.



**7.2 Chat Service API Endpoints**

* **POST /chat/rooms**: Create a new chat room for an organization.
* **GET /chat/rooms/:id**: Retrieve messages for a specific chat room.
* WebSocket connection for real-time messaging: ws://<server\_url>/chat/:roomId .

Attached is the Open Api document for all the endpoints.



**8. Scalability Considerations**

Hosting on **Google Kubernetes Engine (GKE)** ensures scalability for both microservices. GKE automatically handles horizontal scaling, where additional pods can be spun up or down based on the demand.

**8.1 Horizontal Scaling**

* Both services can be scaled horizontally to handle more traffic, especially as the number of organizations and users grows.
* Each microservice can be scaled independently based on its workload (e.g., more replicas for the chat service during peak communication periods).

**8.2 Load Balancing**

* GKE provides built-in load balancing to distribute traffic evenly across available pods, ensuring high availability.

**8.3 Auto-Scaling**

* Kubernetes can automatically scale services based on CPU/memory usage or custom metrics. Autoscaling related configuration can be seen in shared code. Refer the K8-Mnaifest folder.

**9. Security Measures**

Security is a top priority, and the following measures should be implemented:

**9.1 Authentication Security**

* Use **JWT** tokens for secure authentication and include an expiration time for tokens.
* **Token Revocation:** Implement token revocation mechanisms for logging out users or invalidating tokens after a password change.

**9.2 Data Security**

* **Encryption:** Ensure that sensitive data such as passwords are encrypted before storing in the database. Use bcrypt for hashing passwords.
* **HTTPS:** Use HTTPS for all API calls to ensure secure communication between the front-end and back-end services.

**9.3 Role-based Permissions**

* Ensure that users only have access to resources they are authorized to view, based on their roles (admin, member).

**9.4 WebSocket Security**

* Secure WebSocket connections with JWT tokens to authenticate users and prevent unauthorized access to chat rooms.

**10. Running the Application On local**

To run the application, follow these steps:

* 1. From root of the code shared. Navigate to services\user-api folder
  2. Run commands npm install and Node index
  3. This will start the first microservice. Verify the service at url <http://localhost:8080/user/api-docs/#/>
  4. From root of the code shared. Navigate to services\chat-api folder
  5. Run commands npm install and Node index
  6. This will start the Second microservice. Verify the service at url http://localhost:8081/chat/api-docs/#/
  7. From root of the code shared. Navigate to react-app folder
  8. Run commands npm install and npm start
  9. Access the react app at <http://localhost:3000/>
  10. Use username: [test@gmail.com](mailto:test@gmail.com) and password: test123
  11. To see the admin view ese username: [admin@gmail.com](mailto:admin@gmail.com) and password: test123