Content for the Document:

1. **GitHub Repository URL:**
    - https://github.com/hanifmasy/dating-app-backend
2. **Functional Requirements:**
    - User Registration (Sign up)
    - User Authentication (Login)
    - Profile Management (View, Edit)
    - Swiping Functionality (Like, Pass)
    - Premium Features (Purchase, Unlock)
    - User Settings (Update)
3. **Non-Functional Requirements:**
    - Scalability for potential growth in user base
    - High availability to ensure the system is accessible
    - Security measures for user data protection
    - Performance optimization for quick response times
4. **Tech Stacks and Reasoning:**
    - **Node.js**: JavaScript runtime for building scalable network applications.
    - **Express.js**: Web application framework for Node.js, providing robust routing and middleware.
    - **MongoDB**: NoSQL database for flexibility and scalability.
    - **Mongoose**: MongoDB object modeling for Node.js.
    - **TypeScript**: Adds static typing to JavaScript for better development experience.
5. **System Design:**
    - ## ERD Diagram:

User Entity

- **Attributes:**
  - `userId` (Primary Key): Unique identifier for each user.
  - `username` : User's chosen username.
  - `email` : User's email address.
  - `password` : Hashed password for user authentication.
  - `isPremium` : Boolean indicating whether the user has a premium account.
  - `createdAt` : Timestamp for user registration.
- **Relationships:**
  - One-to-Many Relationship with Swipe Entity:
    - One user can have multiple swipes (likes and passes).
  - Many-to-Many Relationship with PremiumFeature Entity:
    - Many users can have many premium features.

Swipe Entity

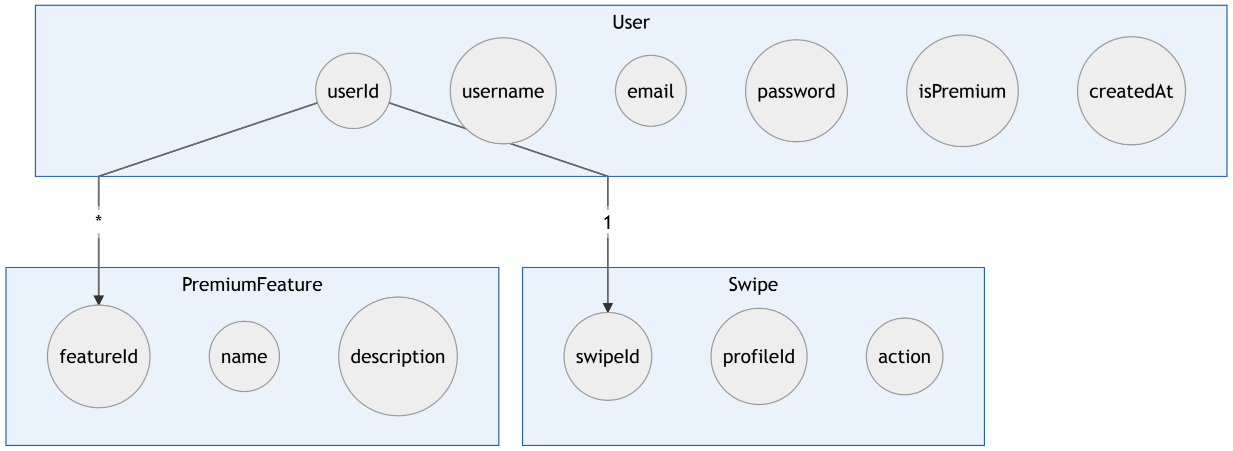
- **Attributes:**
  - `swipeId` (Primary Key): Unique identifier for each swipe action.
  - `userId` (Foreign Key): References the User entity.
  - `profileId` : Identifier for the profile being swiped.
  - `action` : Indicates the action (like or pass).
  - `createdAt` : Timestamp for when the swipe occurred.
- **Relationships:**
  - Many-to-One Relationship with User Entity:
    - Many swipes belong to one user.

PremiumFeature Entity

- **Attributes:**

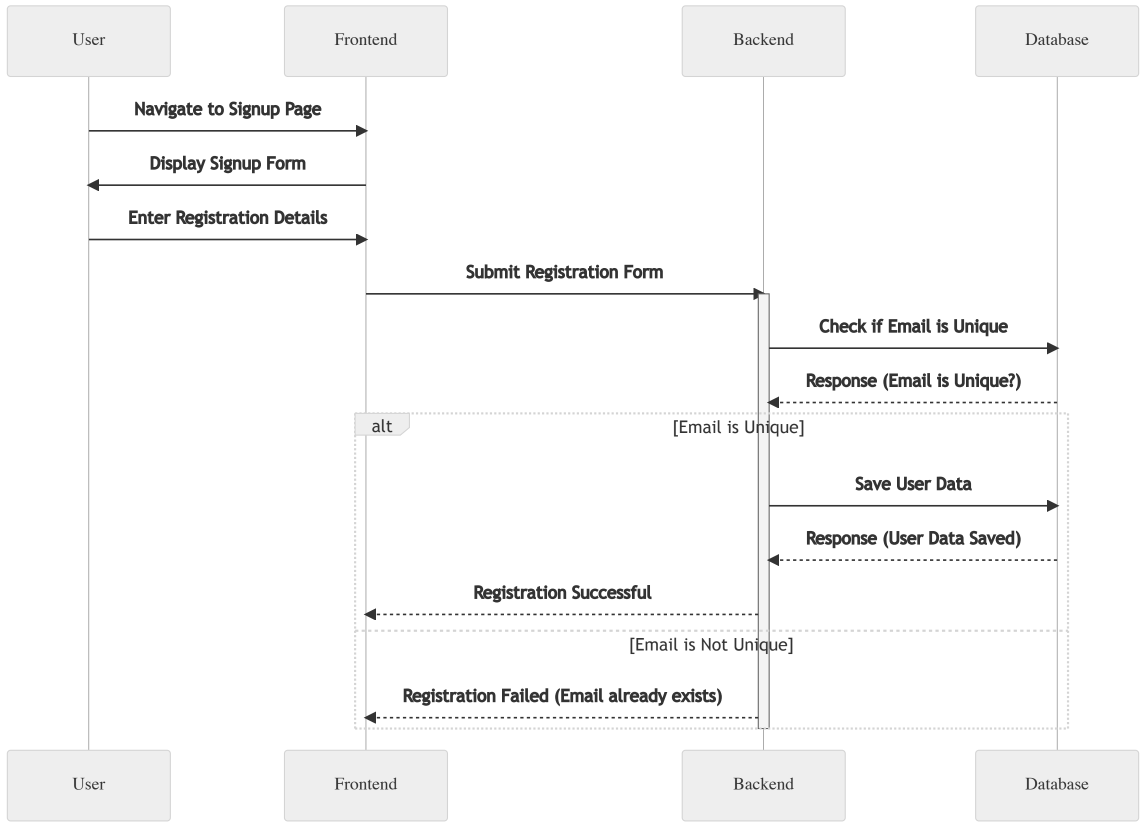
- `featureId` (Primary Key): Unique identifier for each premium feature.
- `name` : Name of the premium feature.
- `description` : Description of the premium feature.

- **Relationships:**
  - Many-to-Many Relationship with User Entity:
    - Many users can have many premium features.



• Sequence Diagram:

Creating a comprehensive sequence diagram for a Dating App backend can be complex due to the various interactions between components. However, I'll provide a simplified example focusing on the user registration process as an illustration. In this example, I'll assume that the registration involves the frontend (client), backend server, and the database.



This sequence diagram outlines the flow of events during the user registration process:

1. The user navigates to the signup page.

2. The frontend displays the signup form.

3. The user enters registration details.

4. The frontend submits the form to the backend.
5. The backend checks if the email is unique in the database.
6. If the email is unique, the backend saves the user data.
7. The frontend receives a successful registration response.
8. If the email is not unique, the frontend receives a failure response.

- ## Test Cases:

  Consider the signup endpoint in your Dating App backend. I'll provide practical unit tests using Jest and Supertest to test the user registration process. The backend uses a MongoDB database through Mongoose.

```
// Assuming your backend app is defined in 'app.ts'
import request from 'supertest';
import app from '../src/app';
import mongoose from 'mongoose';

// Define a sample user for testing
const sampleUser = {
  username: 'testuser',
  email: 'test@example.com',
  password: 'password123',
};

// A utility function to clear the User collection in the database
const clearUserData = async () => {
  await mongoose.connection.collection('users').deleteMany({});
};

describe('User Registration (Signup) Endpoint', () => {
  // Clear the user collection before each test
  beforeEach(async () => {
    await clearUserData();
  });
  // Close the MongoDB connection after all tests are done
  afterAll(async () => {
    await mongoose.connection.close();
  });

  it('should register a new user when valid details are provided', async ()
    const response = await request(app)
      .post('/auth/signup')
      .send(sampleUser);

    // Assert the response status and message
    expect(response.status).toBe(201);
    expect(response.body).toHaveProperty('message', 'User registered succes

    // Assert that the user is actually saved in the database
    const userInDatabase = await mongoose.model('User').findOne({ email: sa
    expect(userInDatabase).toBeTruthy();
    expect(userInDatabase.username).toBe(sampleUser.username);
  });

  it('should handle duplicate email registrations', async () => {
    // Register the same user twice
    await request(app).post('/auth/signup').send(sampleUser);
    const duplicateResponse = await request(app).post('/auth/signup').send(

    // Assert the response status and message
    expect(duplicateResponse.status).toBe(400);
    expect(duplicateResponse.body).toHaveProperty('message', 'Email already

    // Assert that only one user is saved in the database
    const usersInDatabase = await mongoose.model('User').find({ email: samp
    expect(usersInDatabase.length).toBe(1);
  });
});
```

Explanation:

- **BeforeEach Hook:**
  - The `beforeEach` hook ensures that the User collection in the database is cleared before each test to maintain a clean state.

- **AfterAll Hook:**
  - The `afterAll` hook closes the MongoDB connection after all tests are done.
- **First Test (Valid Registration):**
  - Sends a valid registration request and checks if the response status is 201 (Created).
  - Verifies that the response body contains the expected success message.
  - Checks if the user is actually saved in the MongoDB database.
- **Second Test (Duplicate Email Registration):**
  - Registers the same user twice to simulate a duplicate email scenario.
  - Ensures that the response status is 400 (Bad Request) due to a duplicate email.
  - Verifies that the response body contains the expected error message.
  - Checks that only one user with the given email is saved in the database.