Dating Matchmaking App – Project Report

1. Overview

The Dating Matchmaking App is a backend application developed using FastAPI, SQLAlchemy, and SQLite. Its primary purpose is to facilitate finding the right match for users based on their profile information. The system leverages key parameters such as age, interests, gender, and city to calculate a compatibility rating and suggest the top 3 potential matches.

2. System Architecture

- **FastAPI is used as the web framework, ensuring high performance and ease of development with automatic API documentation.**
- **SQLAlchemy serves as the ORM, enabling smooth interaction with the SQLite database.**
- **SQLite is used as the underlying database for simplicity and quick prototyping.**
- **Pydantic models are employed for request validation and response serialization.**
- **Logging is integrated using Python's logging module for traceability and easier debugging.**
- **Exception Handling is built into each endpoint to manage both expected and unexpected errors gracefully.**

3. API Endpoints

a. Create User – POST /create-user

- **Purpose: Registers a new user into the system.**
- **Parameters:**
  - **name, age, gender, email, city, interests (the interests are stored as a list).**
- **Processing:**
  - **The endpoint accepts a JSON payload containing user details.**
  - **It creates a new user record in the database.**
  - **If an IntegrityError occurs (e.g., duplicate email), the endpoint catches it and returns a 409 Conflict error.**
- **Response: Returns the created user record with an auto-generated ID.**
- **Exception Handling:**
  - **IntegrityError is caught for unique constraint violations.**
  - **A general exception block catches other errors, logs them, and returns a 500 error.**

b. Read All Users – GET /find-all-users

- **Purpose: Retrieves a list of all registered users.**
- **Processing:**
  - **Executes a query to fetch all users from the database.**
  - **Logs the number of users fetched.**
- **Response: A list of user records.**

- **Exception Handling:**
  - **Wrapped in a try/except block to catch and log any unexpected errors, returning a 500 error if needed.**

c. Read Specific User – GET /users/{user_id}

- **Purpose: Retrieves details of a specific user by ID.**
- **Processing:**
  - **Queries the database for the user with the specified ID.**
  - **If the user is not found, a 404 error is returned.**
- **Response: The user's record.**
- **Exception Handling:**
  - **Includes logging and error handling to capture any issues during database access.**

d. Calculate Match – GET /match

- **Purpose: Provides match suggestions for a user based on key matching parameters.**
- **Parameters:**
  - **UserIdentifier: A schema that requires only the user's name and email.**
- **Processing:**
  1. **User Identification:**
     - **The endpoint retrieves the full user profile from the database using the provided name and email.**
  2. **Candidate Selection:**
     - **Filters out candidates of the opposite gender.**
  3. **Matching Algorithm:**
     - **Interests Similarity: Uses Jaccard similarity between user and candidate interests.**
     - **Age Compatibility: Computes a score based on the absolute age difference (with a threshold of 5 years).**
     - **City Match: Awards a bonus if both users reside in the same city.**
     - **Composite Score: Combines the scores with predefined weights: 50% for interests, 30% for age, and 20% for city.**
  4. **Suggestion Generation:**
     - **Calculates a match percentage for each candidate.**
     - **Returns the top 3 suggestions based on the match percentage.**
- **Response: A list of candidate suggestions with a computed compatibility rating.**
- **Exception Handling:**
  - **Uses try/except blocks to handle and log database access errors and other unexpected errors.**
  - **Specific HTTP exceptions are raised for cases like "User not found" or "No matches found."**

4. Exception Handling and Logging

- **Exception Handling:**
  - **Each API endpoint is wrapped in try/except blocks to manage errors gracefully.**

    - Specific exceptions (e.g., IntegrityError) are caught to provide detailed feedback (such as conflict errors for duplicate users).
    - A general exception block ensures that any unanticipated error is logged and a 500 Internal Server Error is returned.
- Logging:
    - The logging configuration is set to INFO level with a detailed format, including timestamps and log levels.
    - Informative log statements are present at the start and end of each endpoint, as well as upon exceptions, ensuring that developers can trace operations and diagnose issues effectively.

5. Matching Parameters and Algorithm

- Interests:
    - Uses the Jaccard similarity index to determine the overlap between user interests.
    - A higher overlap yields a higher interest score.
- Age:
    - Calculates an age compatibility score based on the absolute difference in ages.
    - A smaller difference results in a higher score; scores decrease linearly with an age difference up to 5 years.
- City:
    - Provides a binary bonus (1 for a match, 0 otherwise) if both users are from the same city.
- Composite Score:
    - The overall match percentage is calculated using the formula:
      Total Score = (Interest Score * 0.5) + (Age Score * 0.3) + (City Score * 0.2)
    - This composite score is then converted into a percentage, which helps rank the potential matches.

6. Future Improvements

- Security Enhancements:
    - Integrate robust authentication (e.g., JWT-based authentication).
    - Secure endpoints with HTTPS and implement rate limiting.
    - Enhance input validation to prevent injection attacks.
- SOLID Principles & Code Organization:
    - Refactor the codebase to adhere to SOLID principles, ensuring separation of concerns.
    - Implement design patterns (e.g., Repository Pattern) for database interactions.
    - Write unit tests to validate each component.
- System Design and Scalability:
    - Migrate from SQLite to a more scalable relational database like PostgreSQL.
    - Introduce asynchronous endpoints and use an async database driver (e.g., asyncpg) for handling higher loads.
    - Consider using microservices architecture for different functionalities of the application.

- **Additional Features:**
  - **Enrich the matching algorithm by incorporating additional user attributes (e.g., education, hobbies, lifestyle).**
  - **Introduce real-time chat and messaging features.**
  - **Implement profile verification and reputation systems.**
  - **Add machine learning components to improve match accuracy over time.**

7. Conclusion

This Dating Matchmaking App is a robust starting point for a matchmaking system. It demonstrates key backend development practices with FastAPI, SQLAlchemy, and proper exception handling and logging. While the current implementation focuses on core functionality such as user creation, fetching, and match calculations based on age, interests, and city, future improvements will enhance security, scalability, and overall user experience.