Implementation Report

Project Overview

This report outlines the implementation of a FastAPI user profile management system with matching capabilities. The application provides CRUD operations for user profiles and includes a matching algorithm to find potential matches based on user attributes.

Implemented Features

I've implemented all the required endpoints with robust error handling and validation:

1. User Create Endpoint

Implemented a POST endpoint at `/users/` that creates a new user profile. The implementation includes:

- Email format validation using regex
- Duplicate email checking to ensure uniqueness
- Comprehensive error handling with try/except blocks
- Proper HTTP status codes for different error conditions

2. User Update Endpoint

Implemented a PUT endpoint at `/users/{user_id}` that allows updating user information with partial data. The implementation includes:

- User existence validation
- Partial updates using Pydantic's optional fields
- Email format and uniqueness validation when email is updated
- Efficient database operations that only modify provided fields

2. User Deletion Endpoint

Implemented a DELETE endpoint at `/users/{user_id}` that removes a user from the database and returns the deleted user object for confirmation.

3. Find Matches Endpoint

Implemented a flexible GET endpoint at `/users/{user_id}/matches` that finds potential matches based on configurable criteria including:

- Optional city filtering (via `filter_by_city` parameter)
- Opposite gender matching (always applied)

- Optional interest-based filtering (via `filter_by_interests` parameter)

The matching algorithm applies filters progressively:

1. First filters by opposite gender (always applied)
2. Then applies city filtering if requested
3. Finally, performs interest-based matching if requested

This approach allows for flexible matching scenarios:

- With both filters disabled: matches based only on opposite gender
- With both filters enabled: matches users in the same city who share interests
- With various combinations of filters: customized matching based on user preferences

Example URL patterns:

- `/users/1/matches` - Basic matching (only opposite gender)
- `/users/1/matches?filter_by_city=true` - Match by city and opposite gender
- `/users/1/matches?filter_by_interests=true` - Match by interests and opposite gender
- `/users/1/matches?filter_by_city=true&filter_by_interests=true` - Match by city, interests, and opposite gender

4. Email Validation

Implemented email validation using a combination of:

- A regex pattern validation function in the main application
- Schema validation with Pydantic

Technical Approach

Database Design

- Used SQLite for simplicity and portability
- Created a custom JsonList type to store arrays in SQLite (which doesn't natively support array types)
- Implemented proper indexing for frequently queried fields

API Design

- Followed RESTful principles for endpoint design
- Used query parameters for flexible matching criteria instead of creating multiple specialized endpoints

- Implemented partial updates to minimize data transfer
- Added proper error handling and status codes

Matching Algorithm

- Prioritized database-level filtering for performance
- Sorted results by relevance (number of shared interests)
- Made the matching criteria configurable to support different use cases

Assumptions Made

1. **Gender Matching**: Assumed the application matches users with opposite genders (male/female). A production system would likely need more sophisticated gender preferences.

2. **Interest Representation**: Assumed interests are represented as simple string arrays. In a production system, interests might have categories, weights, or hierarchies.

3. **Email Uniqueness**: Assumed emails must be unique across all users.

4. **City Matching**: Assumed exact city name matching (case-insensitive). A production system might use geographic proximity.

Conclusion

The implemented solution provides a solid foundation for a user profile management system with matching capabilities. The design prioritizes efficiency, flexibility, and maintainability while meeting all the specified requirements. The code is structured to allow for easy extension and enhancement in the future.