Project Title: Travel Buddy! - Trip Planning Assistant

Group Members: Sam James, Erica Lee, Brianna Ly, Joshua Constine, Sean Pitman, Theodore Bongolan

Project Description:

This travel planning application helps users plan, manage, and organize trips. The user can log in securely to store and access their trip information, complete with flight details, accommodations, activities, expenses, and personalized notes/descriptions. The application will be able to present the user's trip details in an interactive calendar view. Additionally, the application will also include an AI recommendation feature, which provides travel suggestions tailored to the user's inputted travel destination.

Part 2: Define the API Endpoints
(Our yaml file is submitted along with this doc)

Endpoint	HTTP Method	Description	Request Parameters	Response Structure
/users/{id}	GET	Gets the current users information		User JSON
/users/{id}	DELETE	Removes a user	User id	JSON Confirmation msg

/users	POST	Creates a user	{
			Email:
			Password:
			confirmaPassword
			}
				User JSON

/login	POST	Attempts a login		
/logout	PUT	Logs out		
/trips	GET	Gets a list of trips		
/trips	POST	Creates a new trip	-Name -start date -end date	
/trips/{id}	GET	Gets a trip by ID returns all information	ID path param	Trip info JSON

/trips/{id}	DELETE	Removes a trip, Cascade removal	Trip ID	JSON Confirmation msg
/items	GET	Gets a list of Items (hotel, flight, activity etc)		List if activities
/items	POST	Creates an item	-Name -typeID (hotel, activity etc) -link -date	
/items/{id}	DELETE	Deletes an item	Item id	JSON Confirmation msg
/items/{id}	PATCH	Update an item	Item id	JSON Update Object

Part 3 Data Models:

User Model:

- **Attributes:**
    - id: int, primary key, auto-increment
    - name: str, required
    - email: str, required, unique
    - phone: str, optional, unique
    - password: str, required (hashed)
    - created_at: datetime, default to current time
- **Relationships:** A user can have many trips.

Trip Model:

- **Attributes:**
    - trip_id: int, primary key, auto-increment
    - user_id: int, foreign key to User(id)
    - title: str, required
    - start_date: date, required
    - end_date: date, required
    - description: str, required

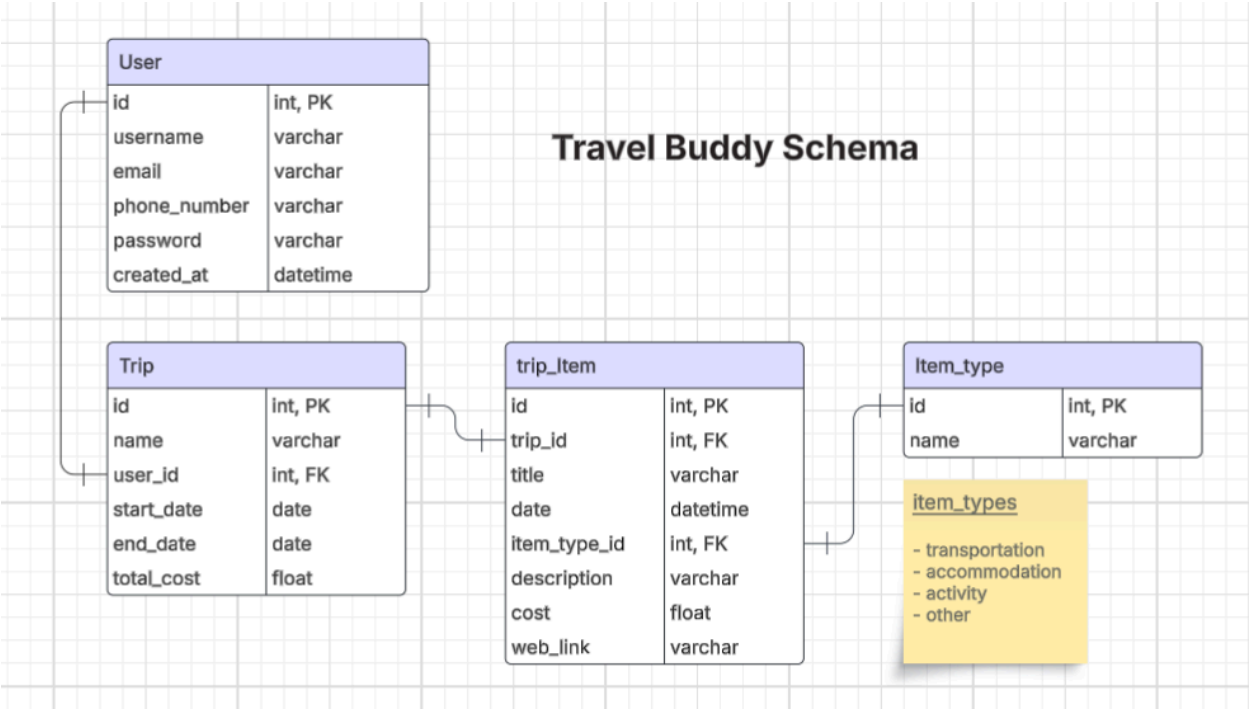
    - total_cost: float {%.2f}, required, internally calculated from item(s) cost
- **Relationships:** A trip can have many items. A trip can be owned by only 1 user.

Item Model:

- **Attributes:**
    - item_id: int, primary key, auto-increment
    - trip_id: int, foreign key to Trip(trip_id)
    - title: str, required
    - date: datetime, required
    - item_type: str, required, from {accommodation, transportation, activity, other}
    - description: str, optional
    - cost: float {%.2f}, required
    - web_link: str, optional
- **Relationships:** An item can be within only one trip. An item can be owned by only one user.

Part 4: Database Schema

https://lucid.app/lucidchart/6de3148c-91cf-4c6a-b66b-7c81321e020e/edit?viewport_loc=681%2C71%2C1286%2C854%2C0_0&invitationId=inv_bd882e9c-2ec1-4625-80fc-6a212213d31d



Part 5: Additional Considerations

1. Authentication: Describe the authentication method you will use (e.g., JWT, OAuth).

- We are using JWT (authentication to securely verify users and control access to protected resources.
- Users will log in with their email and password, and if successful, they will receive a JWT access token.
- This token is digitally signed using the HS256 algorithm and includes user identification data (e.g., sub: user@gmail.com).
- The JWT token has a default expiration time of 1 hour, after which the user must log in again to obtain a new token.
- Passwords are hashed using bcrypt before being stored in the database to enhance security.
- When a user makes a request to a protected route (e.g., /trips), they must include the JWT token in the Authorization header (Authorization: Bearer <token>).
- The backend will decode and validate the JWT to confirm the user's identity before granting access.
- If the token is invalid, expired, or missing, the API will return a 401 Unauthorized error.

2. Middleware: Outline any middleware you plan to implement (e.g., CORS, logging).

We plan to implement middleware for validating authentication, authorization of privileges, logging, and CORS. We need to validate authentication to ensure user privacy and protection, we need to authorize privileges to ensure that users don't have any permissions that could affect the website itself, we need logging to help us when fixing a potential error or bug, and we need CORS to allow users to access our resources from a different domain.

- CORS Middleware: This enables secure cross-origin resource sharing; this will allow the frontend to communicate with the backend while preventing unauthorized access.
- Authentication Middleware: This ensures that only authenticated users can access protected routes by validating JWT tokens or session-based authentication.
- Authorization Middleware: This restricts access to specific actions based on user roles, preventing unauthorized users from modifying, deleting, or accessing restricted data.
- Logging Middleware: This will track and log all API requests, including method, URL, and timestamps.
- Rate Limiting Middleware: This will protect the API from excessive requests.

3. Error Handling: Provide a brief overview of how you will handle errors and what standard responses will look like.

Each of our endpoints will require error handling. The response will be dependent on the nature of our end point method such as post, delete, get, or put. The returned responses will be tailored to the category of failure and inform the user where the failure occurred using the

returned error code. For example, if a scenario where the user tries to post a trip item with invalid or missing required data, the returned response will be an error from the 400 series. Then around the area of interest on screen, there will be a highlight to indicate the invalid input. If our users come across an error, we would also utilize toast notifications to inform them that an error has occurred. Depending on the error, we can either undo whatever action they were trying to do, redirect to another page, or highlight the client error.

4. Testing: Mention how you plan to test the API (e.g., using Postman, automated tests).

We will be testing the API using Postman. From there, we'll be creating automated tests to verify that each component of our API is working correctly. With Postman, we can test multiple requests using different http methods, such as post, get, put, delete, and more. The test will be set up with valid and invalid data bodies to then send a request. The responses will be recorded and verified to enable our backend to be as strong as it can be to better the user experience.

Extra backend/database diagram: