Section 1 - Project Description

1.1 Project

Meet In The Middle

1.2 Description

Meet In The Middle (MITM) is a web app designed for families and friend groups, and is intended to simplify event planning and social coordination. To make planning easier, the app suggests meeting places and provides a way to organize *events*. Additional features of the MITM app include real-time traffic updates, event notifications, and privacy controls to ensure a seamless and enjoyable social planning experience.

Note to Reader: Throughout this document the terms “event” and “meeting” are used interchangeably. An event is simply a meeting that contains one or more users.
*In sections 4 and 6 the term meeting is used when referring to the database.
*In sections 5 and 7 an event is used to describe the logical flow of the program.
Despite the differences in terminology, event and meeting are synonymous in this document.

1.3 Revision History

Date	Comment	Author
10/8/2023	Version 0.1 – Initial draft	Craig Walkup Catherine Laserna Jonathan Wheeler Jason Whitlow Elizabeth Bergshoeff

11/13/2023	Version 1.1 – Draft 2	Craig Walkup Catherine Laserna Jonathan Wheeler Jason Whitlow Elizabeth Bergshoeff
	Version 2.0 – Final Draft	Craig Walkup Catherine Laserna Jonathan Wheeler Jason Whitlow Elizabeth Bergshoeff

Contents

Section 2 - Overview

2.1 Purpose

This document is intended to be used by the MITM development team and by project maintainers.

2.2 Scope

This document describes the components of the Meet in the Middle app.

The primary benefit of the Meet in the Middle (MITM) app is to facilitate easy social planning and event coordination among family, friends, and groups. We aim to streamline the process of selecting meeting destinations based on user preferences, distance, and traffic, while the goals include enhancing social connectivity and making event planning more efficient and enjoyable.

2.3 Requirements

Your mileage may vary -- we typically break down the requirements to provide a ballpark estimate.

2.3.1 Estimates

#	Description	Weeks Est.
1	UX/UI	3
2	Frontend Development	4
3	Backend Development	4
	TOTAL:	11

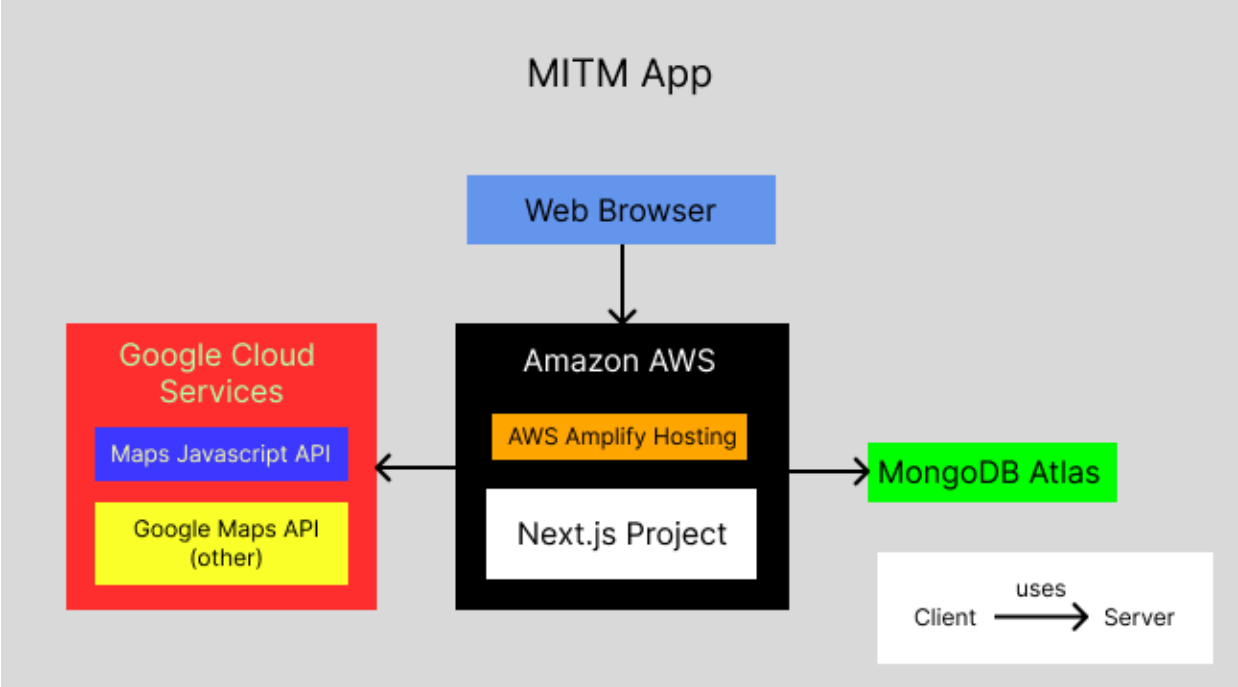
2.3.2 Traceability Matrix

Cross reference this document with your requirements document and link where you satisfy each requirement

SRS Requirement	SDD Module
Req 2	2
Req 3	3.1, 3.2, 3.3

Section 3 - System Architecture

The MITM app uses a Next.js server, secured and hosted with Amazon AWS and powered by MongoDB Atlas. MITM App uses Places API and Maps Javascript API, two Google Cloud services.



System Architecture Diagram: The user accesses the web server, deployed through Amazon AWS, from a web browser. In some cases, the web server will use Google Cloud Services and/or MongoDB Atlas to fulfill requests.

Summary

The MITM app is a web app, using node.js. The MITM app uses the Next.js web framework, MongoDB Atlas for data storage and retrieval, and the Google Maps API for map rendering, location data (Places API), and directions. The MITM app is powered by Amazon AWS services.

System Requirements

This application supports the most recent versions of Firefox, Safari, Microsoft Edge, and Google Chrome, and should work on most commercially available laptops, mobile devices, and desktops. However, some graphics cards do *not* support Google Maps functionality.The Javascript interpreter may download and additional runtimes, as needed. All of the web pages are prepared and delivered by the MITM web application.

For additional details about the user interface, see section 7. For more information about the services offered by the Meet in the Middle app, see section 5.

Amazon AWS Deployment

The MITM app is built on AWS microservices (FaaS, serverless). This includes processors, a public web address, CI/CD tools, and additional features. When the user fetches a page or sends a query, a lambda function instance is created to fulfill the request.

By using Amazon AWS, the MITM team assumes the responsibility of ensuring that the application is compliant with all applicable regulations. The AWS Artifact service contains resources to help customers i.e. the MITM group with compliance.

Project Management - Security

The root account is used only to monitor billing. Instead, IAM users are created for different tasks e.g. deployment and connecting the app with backend resources.

NextJS

Next.js is a React web framework. React is a user interface framework for the web. Web frameworks facilitate web development by providing tools and structure for web applications.

Next.js is an opinionated framework. In a Next.js project, each directory inside of the `src` top level directory defines a route, and special file names are used to create UI.

Next.js is available as an npm package. MongoDB Atlas to store and retrieve data for the user and Google Cloud services to obtain and display geographic information. A Next.js project also contains templates, images, and react components. While Next.js provides a structure and constraints to facilitate good design, the MITM team must be trained to use and deploy Next.js.

Environment Variables: `/application/.env.local`

Backend Routes: `/application/src/api`

Mongoose Models: `/application/src/models`

AWS Amplify

AWS Amplify Pricing

Amplify Hosting automates the build process. The MITM app will be deployed through AWS Amplify from a GitHub repository.

The web application is a consumer of web services, itself. It uses the MongoDB Atlas cluster to store, modify, and retrieve data for the end user. Google Cloud services are used by the web application to retrieve geographic information, to find routes, and to display geographic information.

Google Cloud

The Google Maps API will be used to find locations and routes, and the Maps Javascript API will be used for rendering.

Google Cloud location services can be used in our application at no cost to the development team. In addition to the Google Cloud trial credit, the Google Cloud free tier offers $200 in credit, monthly. Google Cloud services are billed according to different factors, including region. The “Places API (New)” charges a rate of $20 per one thousand API calls. To find more precise costs, from the “APIs and Services -> Library” tab, navigate to the “Places API” or the “Places API (New)” product details. Both of these pages contain links to detailed pricing schemes.

MongoDB Atlas

The web application will use the MongoDB npm package to query a MongoDB Atlas cluster for data storage and retrieval. While MongoDB is a robust NoSQL database, it is not native to AWS. AWS Amplify must whitelist MongoDB to use it. The MongoDB atlas credentials will be stored in the Next.js project in the top level directory. From the repository:

Credentials: `/application/.env.local`

Mongoose

Mongoose is an “object modeling tool;” It provides methods to define *model* objects.

Section 4 - Data Dictionary

Provides a description of all objects and variables.

User		
Field	**Notes**	**Type**
_id	Unique Identifier for each User	BINARY(16)
userName	User name can be full name or alias	VARCHAR
email	User email (UNIQUE)	VARCHAR
password	Encrypted Password	VARCHAR
image	A picture of the User	VARCHAR
bio	A short description of the User	LONG TEXT

defaultLocationId	Reference to the User's default location	BINARY(16)
createdAt	Time User was created	DATETIME
updatedAt	Last time User was updated	DATETIME

User stores email, password, and other personal information. Also contains the addresses to all other data objects. When a new User is created a default location object is also created. The default location can be found by querying the default location id.

Friend Relation

Field	Notes	Type
_id	Unique identifier for each friend relation	BINARY(16)
userId	A user that has a relation with a friend.	BINARY(16)
friendId	A friend that has a relation with a user.	BINARY(16)
createdAt	Time Friend List was created	DATETIME

When a userA becomes friends with userB, two friend relation objects are created. The first relation object stores userA in the userId field and userB in the friendId field. The second object does the opposite, storing userB in the userId field and userA in the friendId field. The partition key is set as the userId so by creating two entries the choice for better time complexity is made over storage space.

Friend Request

Field	Notes	Type
_id	Unique identifier for each friend requests object	BINARY(16)
senderId	The sender of the friend request	BINARY(16)
recipientId	The recipient of the friend request	BINARY(16)
message	Message sent to recipient from sender	LONG TEXT
createdAt	Time friend request was created	DATETIME

Every time a friend request is made, a new object is created in the database. The partition key is set to recipient id for friend request. This allows a user to quickly query their friend requests. If a user wishes to see outgoing friend requests they can query the sender id.

Meeting

Field	Notes	Type
_id	Unique identifier for each meeting location	BINARY(16)
creatorId : userId	The Id of the user that created the meeting	BINARY(16)
meetingLocationId	The Id of the location where the meeting will take place	VARCHAR
pending [] : userId	List of users that have been invited to the meeting	BINARY(16)

denied [ ] : userId	List of users that have rejected the invitation	BINARY(16)
accepted [] : userId	List of users that have accepted the invitation	BINARY(16)
createdAt	Time meeting was created	DATETIME
updatedAt	Last time meeting was updated	DATETIME

Every time a meeting is created a new meeting object is stored in the database. Each meeting has a creator that can modify the meeting, and 3 fields that determine the status users that have been invited to, rejected, or plan to attend the meeting. When a user is invited to the meeting, they are added to the pending field. When a user rejects the invitation they are removed from pending and added to denied. When a user accepts the invitation they are removed from pending and added to accepted.

In order to find meetings a user is associated with the pending and accepted fields are queried. This causes poor time complexity for querying meetings.

Default Location

Field	Notes	Type
_id	Unique identifier for each location	BINARY(16)
name	The name associated with the location (optional)	VARCHAR
addressStreet	Street address of the location	VARCHAR
addressCity	City of the location	VARCHAR
addressState	State of the location	VARCHAR
addressZip	Zip code of the location	VARCHAR
coordinates	Longitude and Latitude of the location	VARCHAR
createdAt	Time location was created	DATETIME
updatedAt	Last time meeting location was updated	DATETIME

Location is used in two ways. First it is used to store the User's default home address. Second it is used to store the location of where meetings will take place.

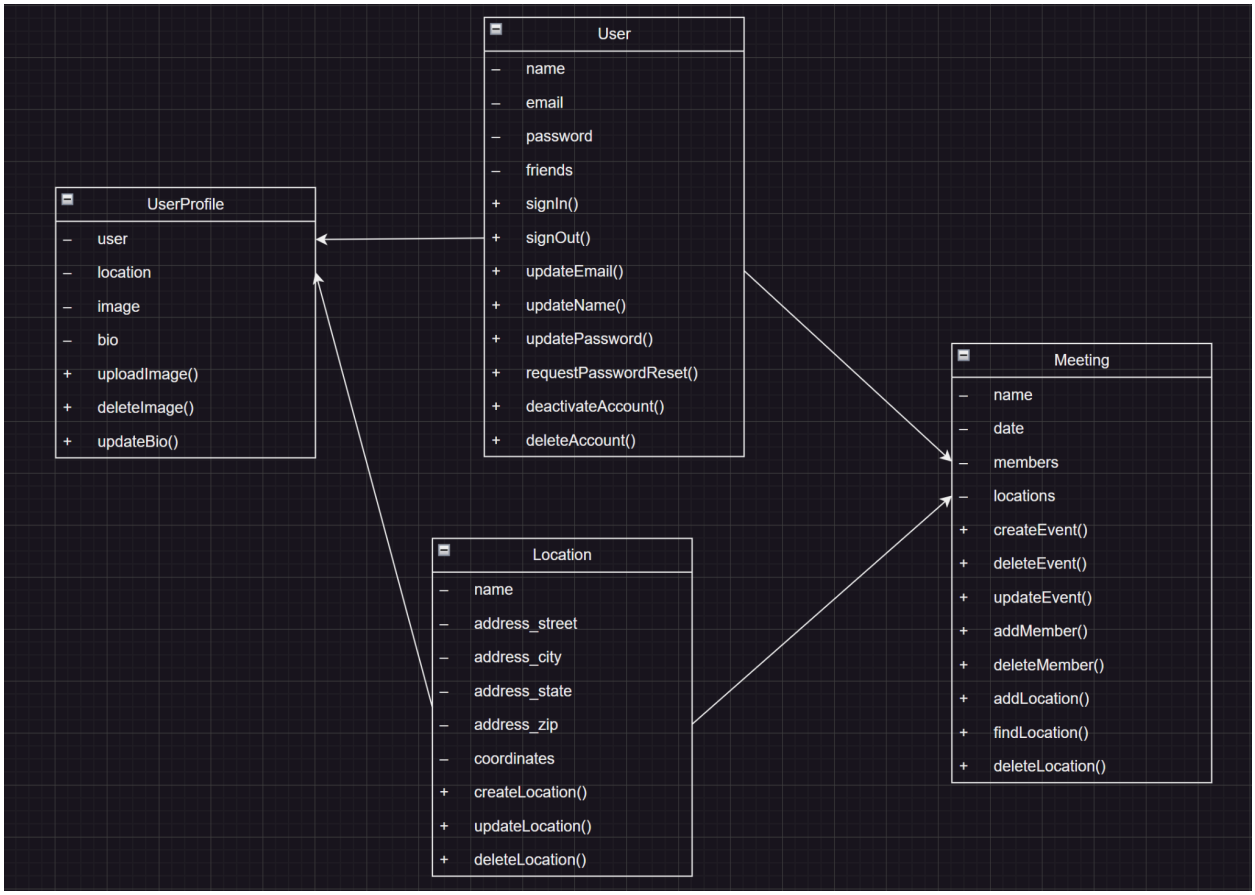
Notification

Field	Notes	Type
_id	Unique identifier for each notification object	BINARY(16)
userId	The User that is connected to the notifications	BINARY(16)
message	The message the notification will hold	LONG TEXT
createdAt	Time notification object was created	DATETIME

Notifications are used to notify the user of friend requests and meeting invitations. In both cases a new notification will appear on the notification page. Each notification is stored as its own object and userId is set as the partition key to facilitate queries.

Section 5 - Software Domain Design

5.1 Software Application Domain Chart



5.2 Software Application Domain

A comprehensive high-level description of each object and the associated methods.

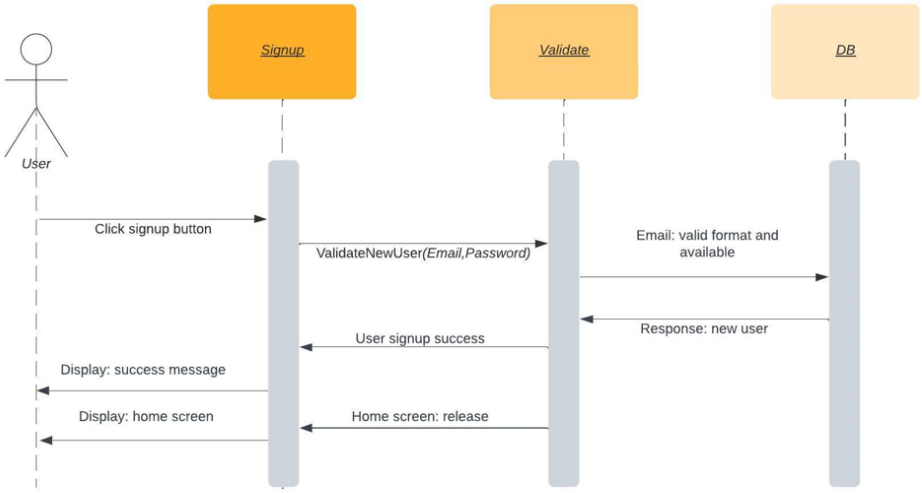
5.2.1 Login, Registration, and Authentication

The front end registration form requires the user to input their email, password, username, and home location. This form data is sent to the backend via a POST request to the `/api/signup` endpoint.

1. The user registration data is validated on the backend before account creation.
   a. The backend checks for required fields and duplicate emails, since emails must be unique.

b. If any validation errors occur, the backend returns a 400 error with a message. The front end displays this error message to prompt the user to fix their input and retry registration.

2. The sign-in form requires their password. Their credentials and information are then sent through the Next Auth package for authentication and validation.
    a. The Next Auth Package simply sends a post request to the appropriate sign-in route in our Rest API backend.
3. If valid, it generates a JSON Web Token (JWT) containing the user ID, email, and other claims.
    a. This JWT is used for session management and stored in a cookie.
    b. Relevant User information (ID) is also stored in session storage



5.2.2 General User Information

1. The user profile data is stored in the database and can be retrieved or updated via API routes that require authentication.
    a. The user ID is generated when a new account is created and is included in the JWT after logging in.
    b. Related data like meetings, friends, locations, and notifications are stored in separate database collections.
        i. These are associated with the user ID to track ownership and relationships
        ii. The front end can retrieve this related data by making an authenticated API request including the user ID.

Notifications *

1. The current implementation relies on pull notifications - the front end requests any new notifications on page load.

a. To enable push notifications, a web socket must be established to allow the server to proactively send notification data to the client.

5.2.3 Friend / Friend Requests

1. Friend requests are stored in the friend_requests database collection. Upon login, the API checks for pending requests and notifies the user.
    a. The front end displays a notification dot and a list of open requests.
    b. Friend requests are accessible by `/api/user/friend-requests/:userId`.
2. When a friend request is sent a notification is added to the notification inbox of the potential friend.

Searching for Friends

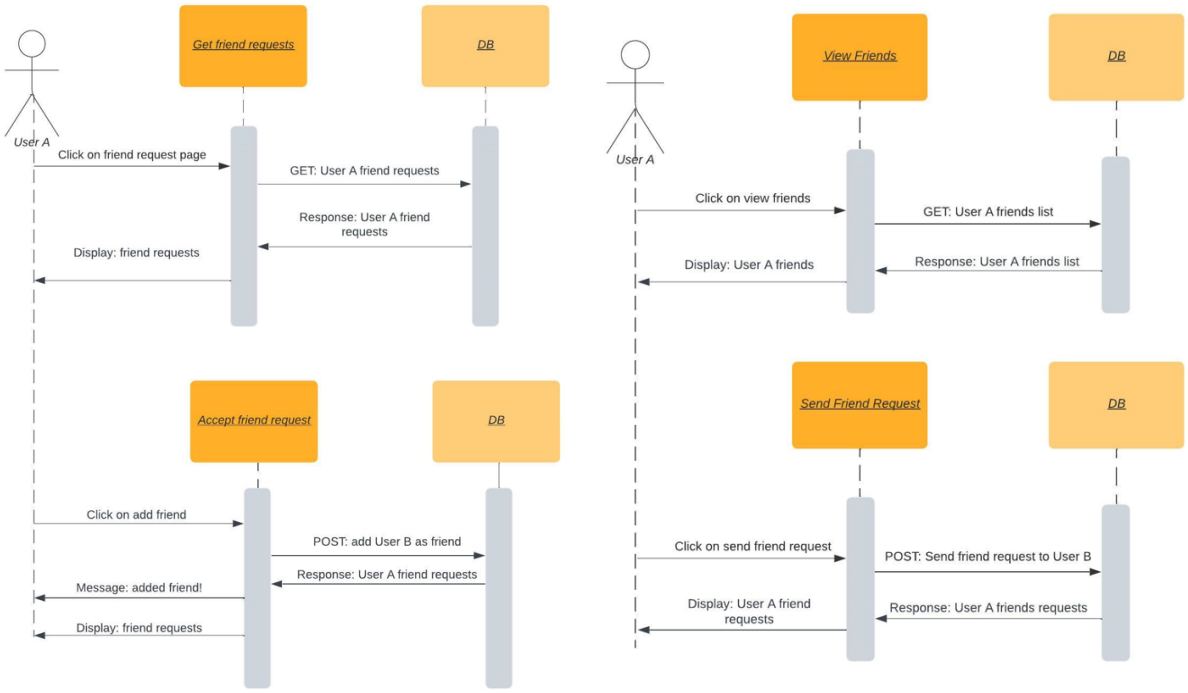
1. Users can search for friends through the “Search for Friends” page.
2. When the page loads, the front end makes a GET request to `/api/users/friends` with the query parameters.
    a. The backend searches the user's collection for matching email or username
    b. Matched user documents are returned containing public fields like name, avatar, bio.

Sending / Accepting Friend Requests

1. To send a friend request, the user clicks the "Add Friend" button on the search results.
    a. Adding them sends a POST to `/api/user/send-friend-request/:userId` with the recipient's ID.
2. The backend validates the request and inserts a document into friend_requests. Contains the sender ID, recipient ID, and timestamp
3. If the recipient accepts, a request is made to `/api/user/accept-friend-requests/:userId`
    a. The sender ID is added to the recipient's friends array.
    b. The recipient ID is added to the sender's friends array.
4. To revoke access, a user can reject by clicking on the reject button
    a. Friends are then removed via `/api/user/reject-friend-requests/:userId`

Security notes:

- Default locations are not exposed during search
- Friend APIs require authentication via JWT



Viewing Friends

The user can retrieve a list of their friends through the “Friends” page. This list is fetched from the backend through the user’s stored User ID stored in that session (which would’ve been stored when they signed in).

Unfriend

1. The user can unfriend another user through the “Friends” page.
    a. When a user unfriends another, both User IDs and information are deleted from both lists. This logic is handled by the backend.
    b. On the front end, a simple request is made to the backend containing both User IDs to begin this process.

5.2.4 Meetings

Each meeting has a unique ID, a user that created the meeting, and a list of members that are attending the meeting. Every time a new meeting is created, a new meeting object is added to the database.

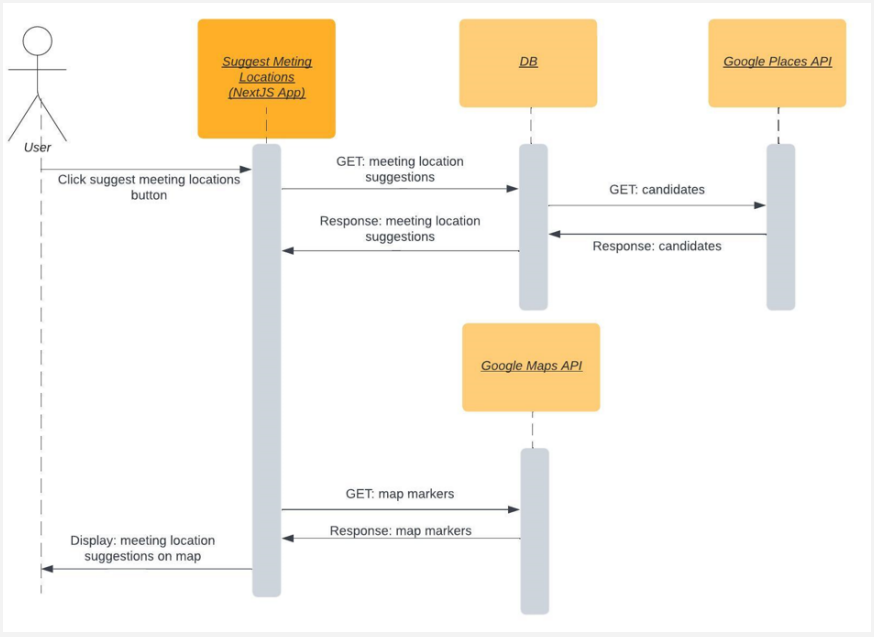
Meeting Management

The creator will add items to the meeting’s itinerary, send invitations, and perform a search from the meeting’s management page. Upon submission of

the “add an activity” form, a new activity will be added to the meeting by sending a request to the backend Rest API.

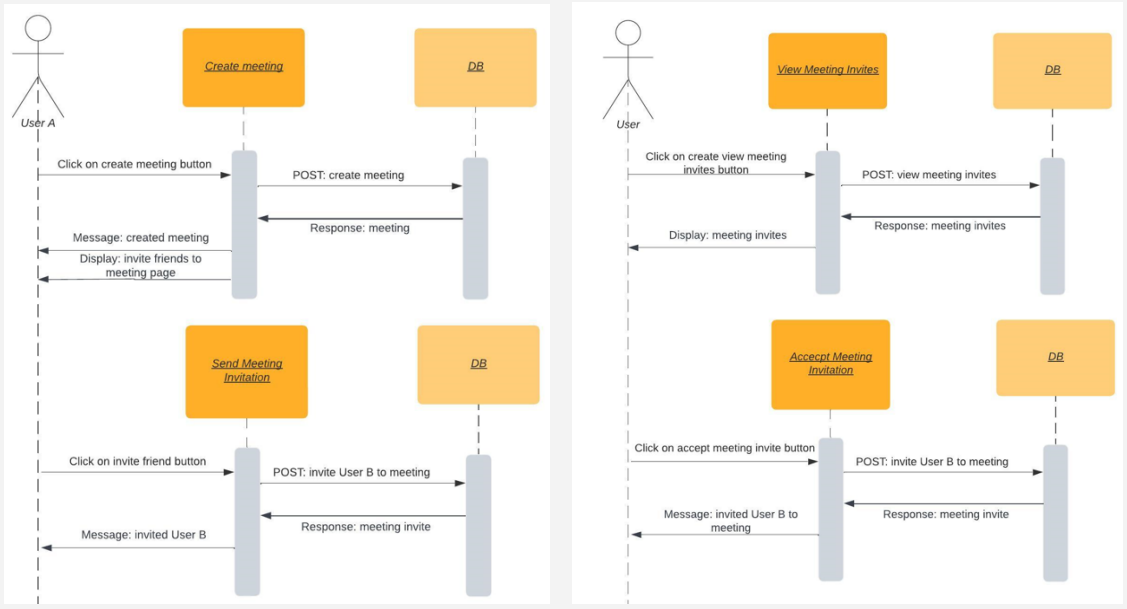
New Meetings

1. Creators will be able to fill out a new meeting form to create a meeting page. The meeting form will include a selection of friends.
    a. The front end will create a request to add the meeting information to the database and send the meeting IDs/invitations to the recipient ID by making a POST request to `/api/user/meeting/:userId`



Meeting Invites

Immediately after creation of the meeting, the user is redirected to a page that gives the User the option to send invites to their friends. With every new friend that is added a new meeting invite is added to the database that contains a sender ID, receiver ID, and meeting ID. Meeting Invites are created through a POST request to `/api/user/meeting-invite/:userId`



Meeting Retrieval

Attendees will be able to retrieve an itinerary for the meeting. Each item in the itinerary represents a group activity and will provide a time, location, and description. The creator will be able to add and remove meetings from the itinerary by making a GET request to `/api/user/meeting/:userId`

Itinerary Map (Optional Feature)

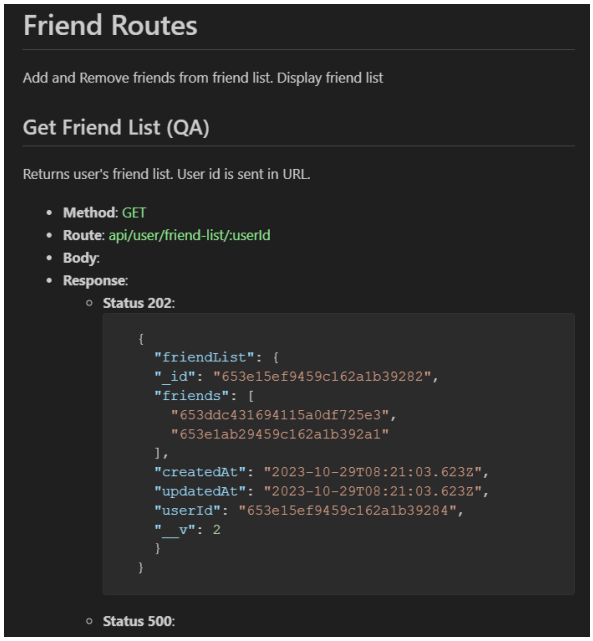
1. The itinerary map is an optional feature that is not currently implemented in the database. The map will visualize the travel paths for the attendees by retrieving each attendant's default location (longitude and latitude coordinates)
    a. This location is displayed through the Google Map API as a marker. A path is drawn between each marker to the respective meeting location.

5.2.5 Location Recommendations

1. Our function, getMidPoint, calculates the geographical midpoint of multiple location points represented by longitude and latitude tuples.
    a. The function converts these coordinates from degrees to radians.
    b. Then, it computes the average of the Cartesian coordinates in a 3D space before converting the result back to longitude and latitude in degrees.
    c. This returns the midpoint for a given set of location points and is used for the main purpose of finding a meeting location that works for everyone in the meeting.

5.3 Routes

A comprehensive list of routes to access all database objects and functions is available at the following link. Each route includes a GET/POST example and the expected json response from the database.



Friend Routes

Add and Remove friends from friend list. Display friend list

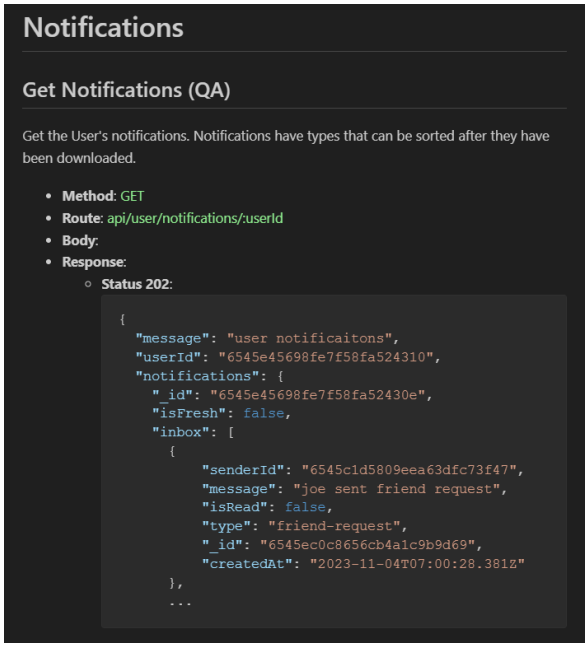
Get Friend List (QA)

Returns user's friend list. User id is sent in URL.

- **Method**: GET
- **Route**: api/user/friend-list/:userId
- **Body**:
- **Response**:
  - **Status 202**:

```
{
  "friendList": {
  "_id": "653e15ef9459c162a1b39282",
  "friends": [
    "653ddc431694115a0df725e3",
    "653e1ab29459c162a1b392a1"
  ],
  "createdAt": "2023-10-29T08:21:03.623Z",
  "updatedAt": "2023-10-29T08:21:03.623Z",
  "userId": "653e15ef9459c162a1b39284",
  "__v": 2
  }
}
```

  - **Status 500**:



Notifications

Get Notifications (QA)

Get the User's notifications. Notifications have types that can be sorted after they have been downloaded.

- **Method**: GET
- **Route**: api/user/notifications/:userId
- **Body**:
- **Response**:
  - **Status 202**:

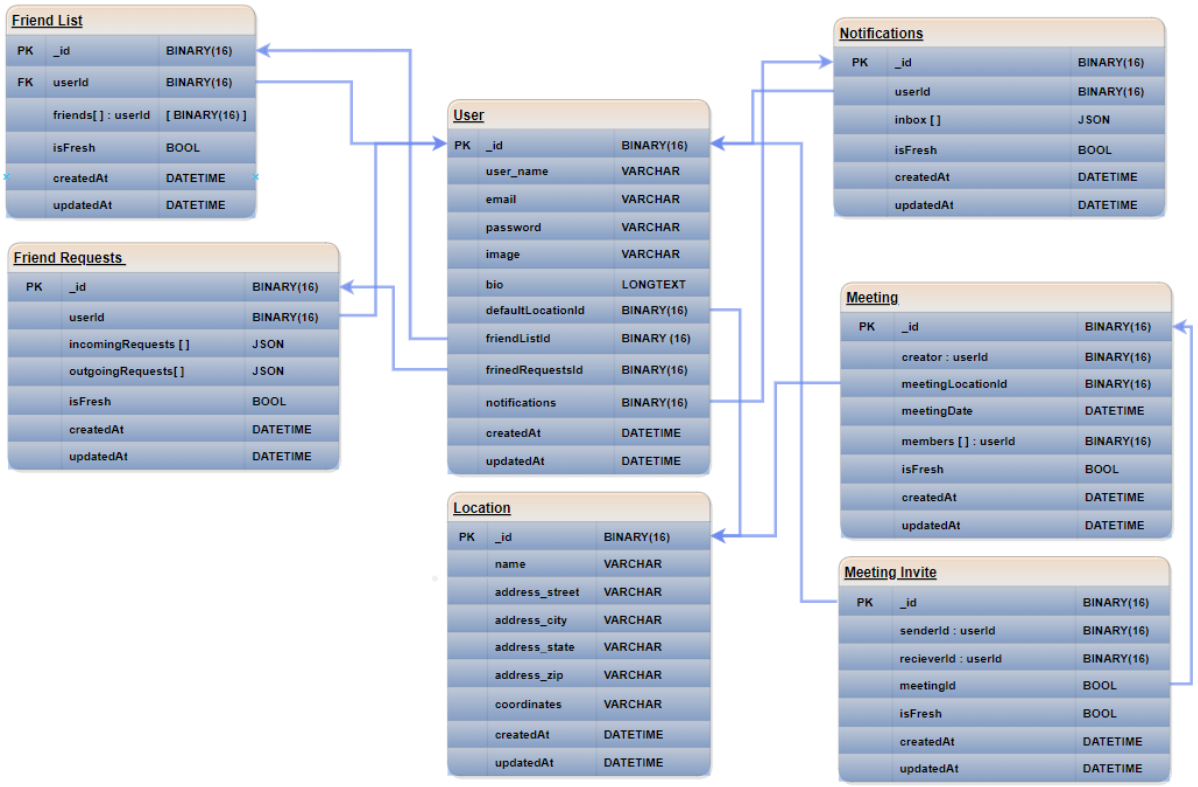
```
{
  "message": "user notificaitons",
  "userId": "6545e45698fe7f58fa524310",
  "notifications": {
    "_id": "6545e45698fe7f58fa52430e",
    "isFresh": false,
    "inbox": [
      {
          "senderId": "6545c1d5809eea63dfc73f47",
          "message": "joe sent friend request",
          "isRead": false,
          "type": "friend-request",
          "_id": "6545ec0c8656cb4a1c9b9d69",
          "createdAt": "2023-11-04T07:00:28.381Z"
      },
      ...
```

https://github.com/CSCI150-LAB01/Meet_in_the_Middle/tree/working/application/rotue_info

Section 6 – Data Design

6.1 Persistent/Static Data

6.1.1 Database



6.1.2 Static Data

User Model, Friend List Model, Notifications Model, Location Model, Meeting Model, Meeting Invite Model and Friend Requests Model.

Mongoose is used to manage the database models and these models are not expected to change

6.1.3 Persisted data

Users, Friend List, Friend Request, Notifications

These objects are all initialized during User signup and they will not be removed from the database unless a User is deleted. When a User is deleted all objects

are removed from the database. Any *Meetings, Locations,* or *Meeting Invites* associated with the objects are also deleted.

6.2 Transient/Dynamic Data

Meeting and *Location* objects are created when the User decides to create a meeting. When a meeting is created a location is linked to the meeting by addition of a location ID to the meeting object(as previously discussed). Any *Meeting* and *Location* object may be deleted at any time..

Google Places API returns a list of *locations*. These locations will not be stored in the database and are immediately sent to the front end. If a place is selected as a destination it will be stored as a part of the meeting, otherwise the data is discarded.

6.3 External Interface Data

Google Maps API accepts a coordinate pair and returns an interactive map and locations to the front end.

*Google Places AP*I accepts a coordinate pair and returns a list of places near the coordinate pair.

6.4 Transformation of Data

User email and *picture* are obtained from Google OAuth and then inserted into the user object before being stored in the database.

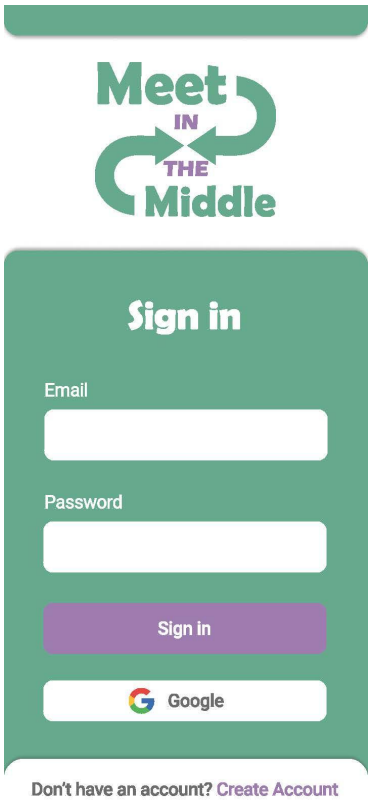
Google Maps API and *Google Places API* return *locations* that may be modified to fit the format of the database. It is also possible to modify the database to fit the format of the Maps and Places API depending on the return types.

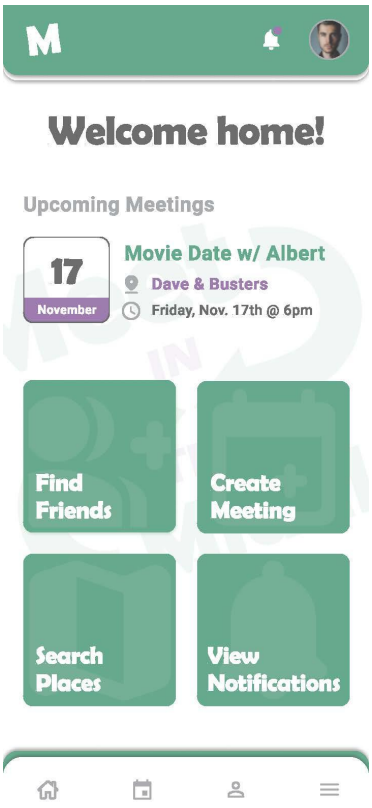
Section 7 - User Interface Design

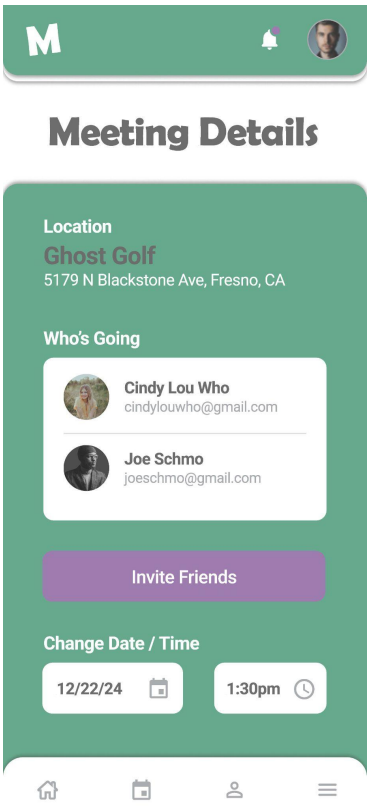
7.1 User Interface Design Overview

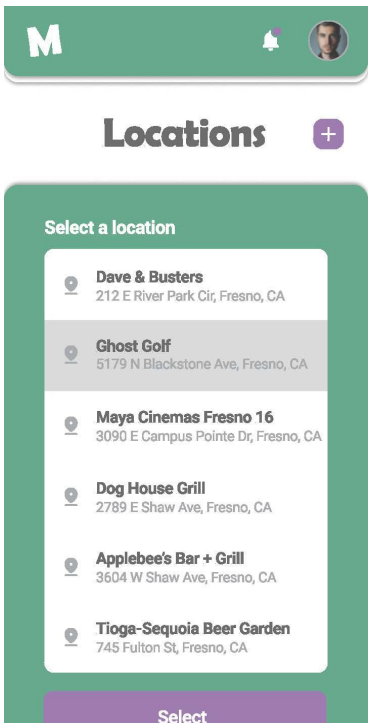
The user interface design consists of a simple color scheme with three primary colors following the 60-30-10 rule to provide a stylistic and clean experience. The main design is centered around the logo with soft accents and a rounded appearance. The main pages will utilize a menu bar at the bottom of the application for ease of access and control. The links at the bottom navigation will consist of a home link, a maps link, a person search link, and a more link that will provide a menu for main navigation. This will include the ability to view the map, handle meetings, handle friends, and handle

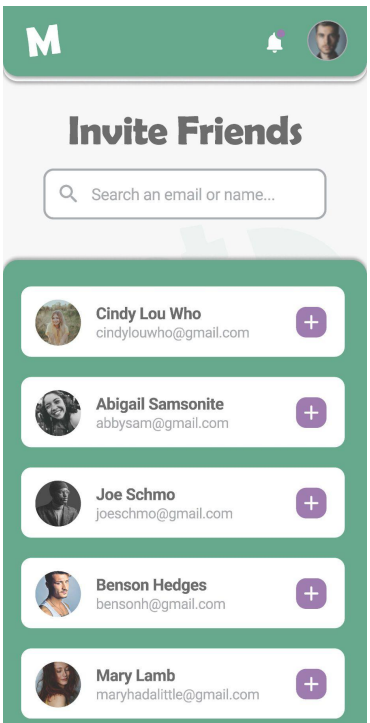
their location. There will be a header at the top with the profile image and logo. The profile image will also serve as a secondary navigation menu with the profile controls and logout link.

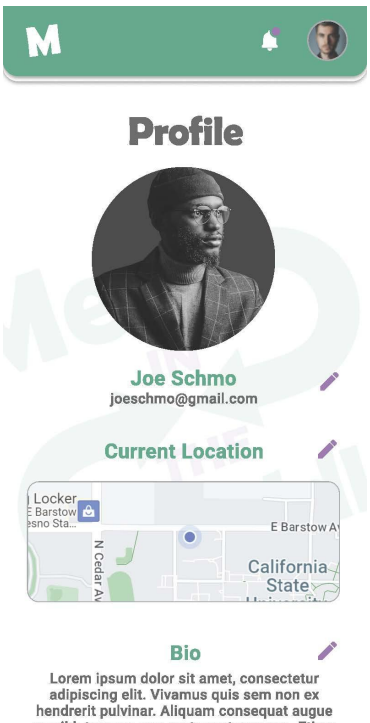






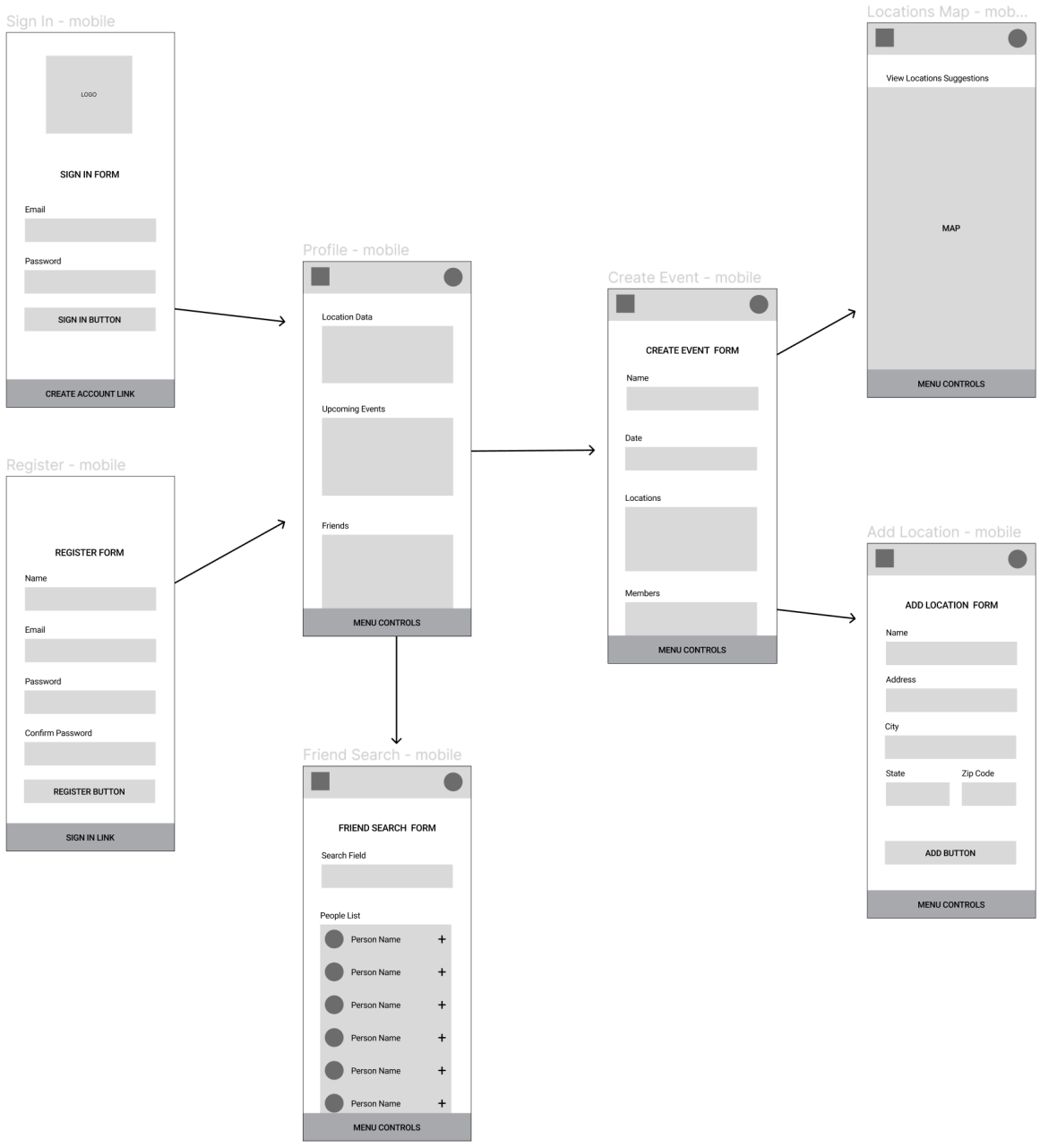






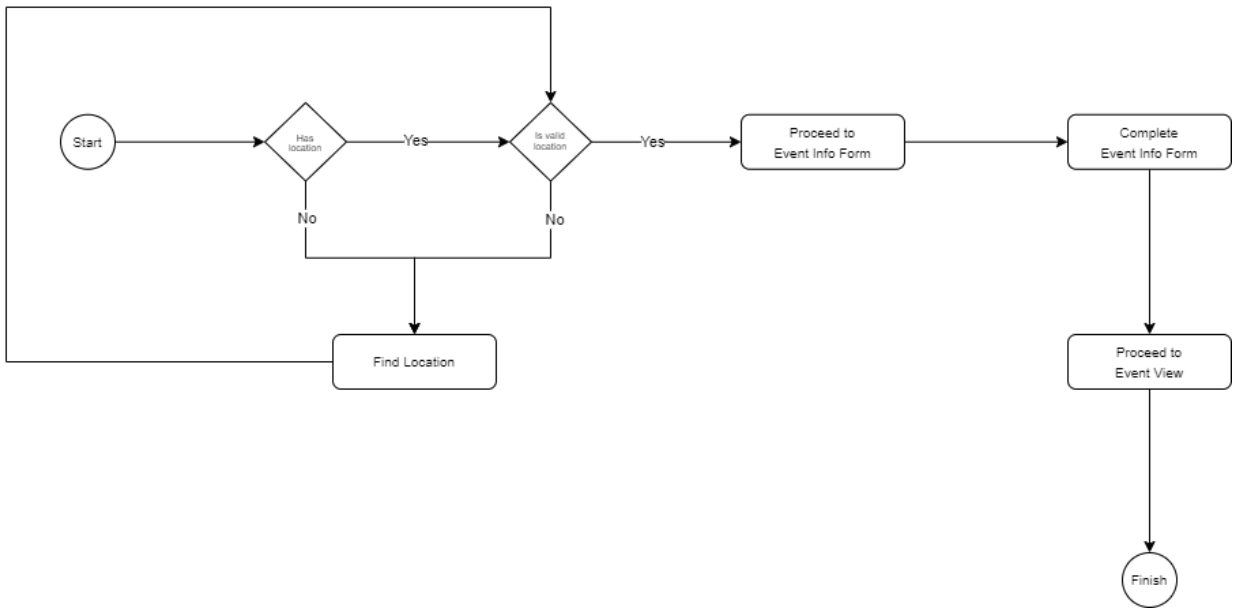
7.2 User Interface Navigation Flow

The user interface navigation flow is displayed here using a portion of the wireframe that was created to help guide the design process and ensure the main functionality and interactions were added to the application. Since the application is intended for mobile use the navigation is kept primarily on the bottom to provide ease-of-access and also includes a profile navigation at the top as well.

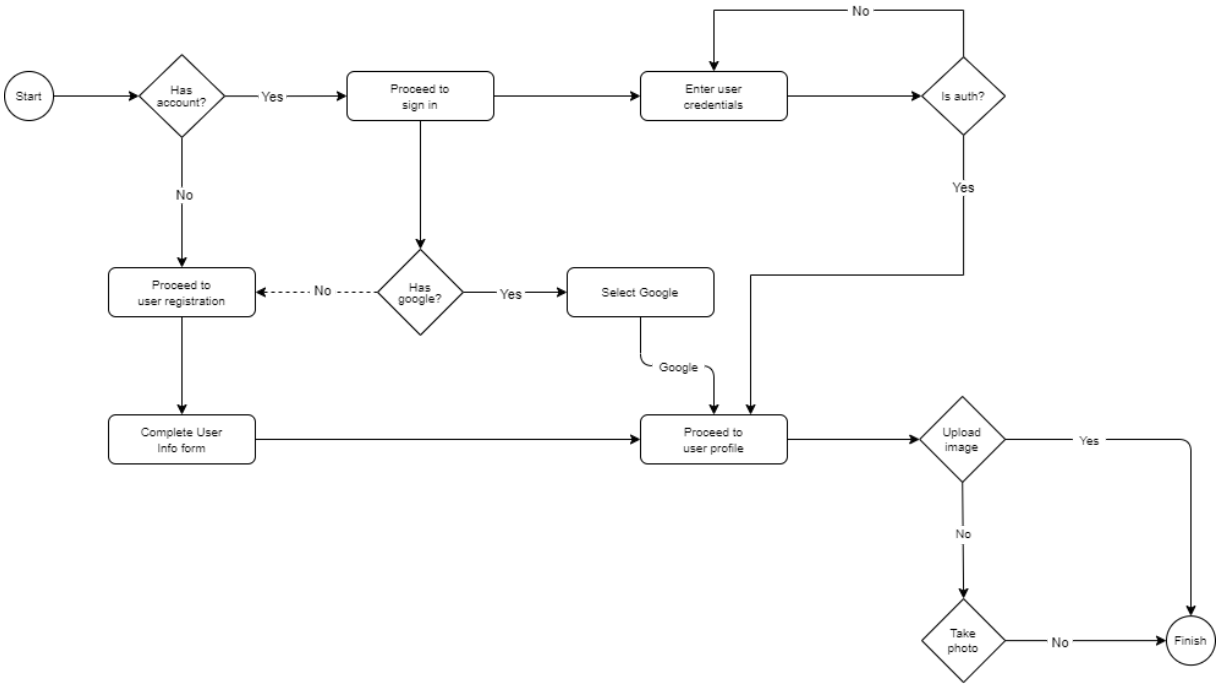


The state diagrams provide the expected user state while performing the following interactions. Here the intention is to map out the potential paths that a user may engage in and the resulting state.

Event Creation Diagram

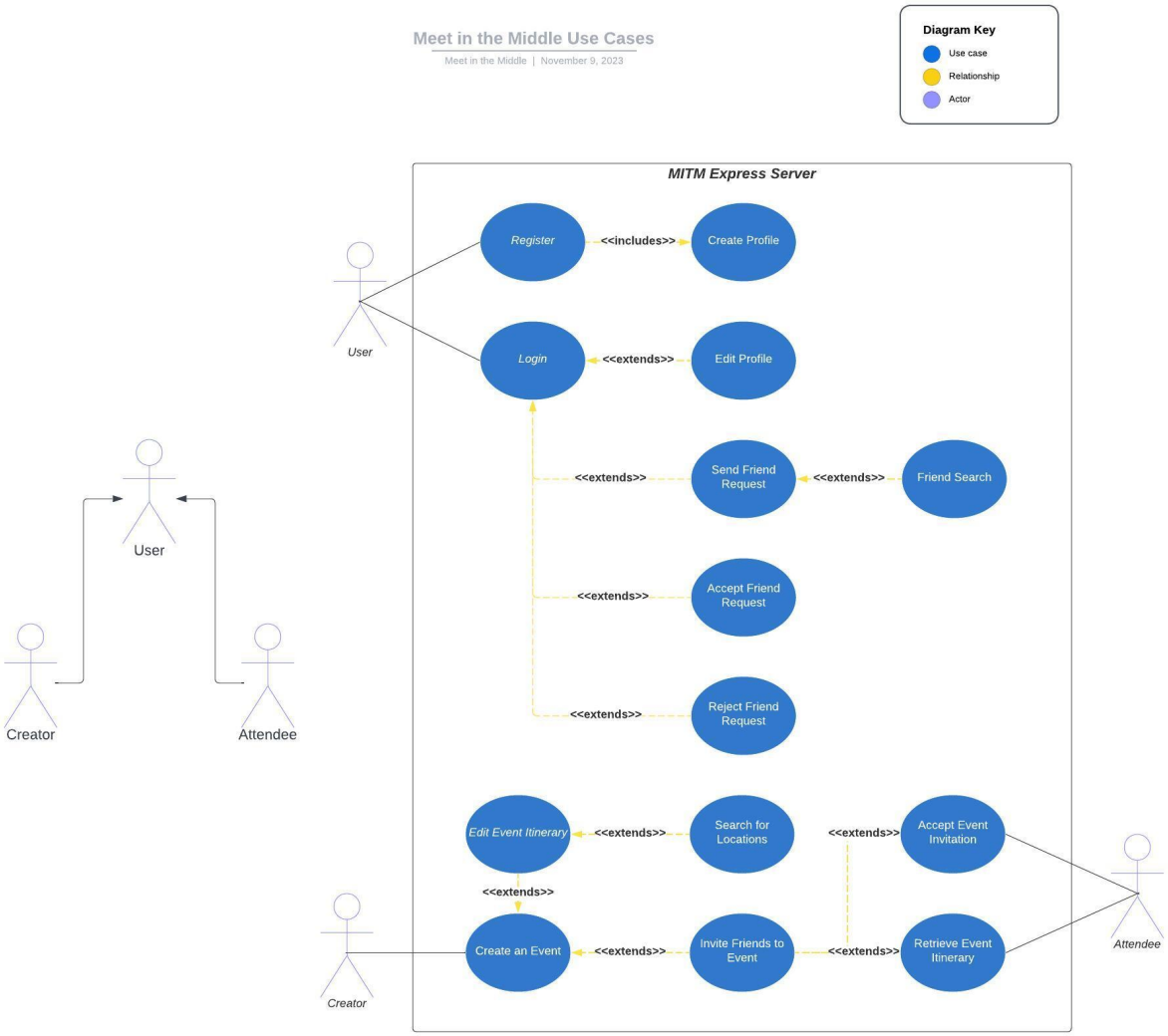


User Sign In and Sign Up Diagram



7.3 Use Cases / User Function Description

After the initial authorization the user will be taken to their main profile screen to get a quick overview of what is immediately available. The bottom navigation allows the user to go back to the profile overview via the home link, view a map of locations via the maps link, view friends and add friends via the people link, and get access to other navigation opens via the menu link. The main menu will allow users to navigate to the meetings management where they will be able to add meetings, modify meetings, and delete meetings. User management will allow for users to be searched by name or email and added. Users must be friends to add individuals to meetings. Forms will be designed and implemented to control the flow of data.



Section 8 - Other Interfaces

The following is a list of external interfaces.

8.1 External Interfaces

Google Places API

Google Places API is contacted by the NextJS backend server. The Places API is sent a coordinate pair and optional metadata and returns a list of places to NextJS.

Google Maps API

The API provides a way for us to access maps, geocoding, and places. A coordinate pair passed as an argument to a Google Maps API marker from the NextJS frontend to the Google Maps API. An interactive map is then displayed with this marker as a child, to the user.

Google OAuth

NextJS contacts Google OAuth for User authentication.

Section 9 - Extra Design Features / Outstanding Issues

9.1 Extra Features

Web Socket

Currently notifications are only presented to the user when the page is refreshed. This is a pull notification structure. If a web socket were implemented the notifications would be instant and this would eliminate any need for polling from the front end.

The web socket would also provide a messaging feature where users could contact each other directly in real time.

9.2 Outstanding Issues

Testing

Trivial cases have been tested with Postman. However, automated tests have not been prepared and more cases should be tested. Either Postman or Cypress will be used to develop the automated tests.

Google Authentication

The system is designed to work with Google OAuth but it is not currently implemented.

Error handling/Security Issues

The current implementation does not take into consideration how to properly return errors and data to the frontend to avoid security and privacy issues.

Experimental Routes

Users should only be allowed to access the services which are intended for use. The prescribed routes are the only routes which the deployed product should have available, and all experimental routes should be excluded from the final product. Presently, the API contains several routes for debugging and experimental purposes.

Messaging

There are many different IM applications available. However, in-app messaging would support the other functions of the app. The MITM team would have more control over the messaging application and users may prefer the messaging feature over 3rd party messaging applications. Users would have more flexibility in specifying their preferences, and may use the messaging feature instead of switching contexts. The in-app messaging could be used to send global messages from the MITM team to users. A messaging system could be used as a base for additional functionality, including prompts, message suggestions, games, and custom text.

Section 10 – References

Github MITM Project:
https://github.com/CSCI150-LAB01/Meet_in_the_Middle

Routes:
https://github.com/CSCI150-LAB01/Meet_in_the_Middle/tree/working/application/rotue_info

Figma:
https://www.figma.com/file/nFn3YriyGmE9nnfBMgbhP2/MeetInTheMiddle-team-library?type=design&node-id=0%3A1&mode=design&t=dbD4rbzD7fE1v1rT-1

Diagrams:
https://github.com/CSCI150-LAB01/Meet_in_the_Middle/tree/main/ui_ux/diagrams

Compliance:
https://cloudsecurityalliance.org/star/registry/amazon/services/amazon-web-services/

Tests:

The following routes have been manually tested with Postman. The GET, DELETE, and POST requests were manually drafted and sent. Automated tests and more rigorous tests are in development.

/api/signup
/api/user/[id]
/api/user/accept-friend-request
/api/user/default-location
/api/user/friend-list
/api/user/friend-requests
/api/user/get-notifications
/api/user/notifications
/api/user/reject-friend-request
/api/user/remove-friend
/api/user/send-friend-request
/api/user-list

Section 11 – Glossary

Glossary of Terms / Acronyms

MITM - Meet In The Middle
JWT - JSON Web Token