M3 - Requirements and Design

1. Change History

1. Separated user authentication (via Google OAuth) from Manage Accounts. (2.12.2024)
    - Authentication was entangled as a smaller use case in Manage Accounts. Now, delineate authentication so users seperately log in. After, allow edits to username, profile picture.etc.
2. Specify new User - Location matching algorithm: identify elements in photo and generate "tags" for users. Then, recommend users to users or location based on tag similarity, proximity, and preference. (2.12.2024)
    - Original technical highlight was on banning/unbanning users, a secondary functionality compared to our app's purpose.
    - Therefore, the recommendation become our main complexity of our project
3. Elaborated NFR 1 and changed NFR2 to be specific and measurable (2.12.2024)
    - NFR1 lacked a concrete justification to the upload speed.
    - NFR2 was updated to a more measurable problem.
4. Modify the interface of component, remove some interfaces that are actually implement in frontend, and add the reasonable input and output. Also, add the new interface. (2.17.2024)
    - Also, need to modify the sequencal diagram, will modify them one by one to fit our new logic
5. Due to our TA suggestion. We put our main effort on recommendation logic, therefore the ban algorithm is not implement yet. Therefore, the main actor of our current program is only user. The actor Administrator does not have a special power right now. (Some administrator interfaces have been implemented, but they are currently mainly used as testing features.) (2.20.2024)
6. Modify the use case and scenario
    - Remove the supervision use case, since it is not implement yet. Instead, we add a new use case: View Gallery. (2.22.2024)
    - Modify share gallery feature, remove one sub usecase. Also adjust upload photo use case, add one sub usecase (2.22.2024)
    - These changes to use case allow us to focus on addressing our primary users more, ensuring our key features match the target audience.
7. Add new framework: AWS Route 53, and External Modules (2.20.2024)
8. Clarified and modified use cases: View Map Info, Upload Photos, Receive Recommendation.
    - Past descriptors misaligned with our vision during development, and we incorporated concrete tests to test for specific behavior. (3.18)
9. Renamed usecase "Recommend Locations" to "Explore Locations of Interest" (3.30)

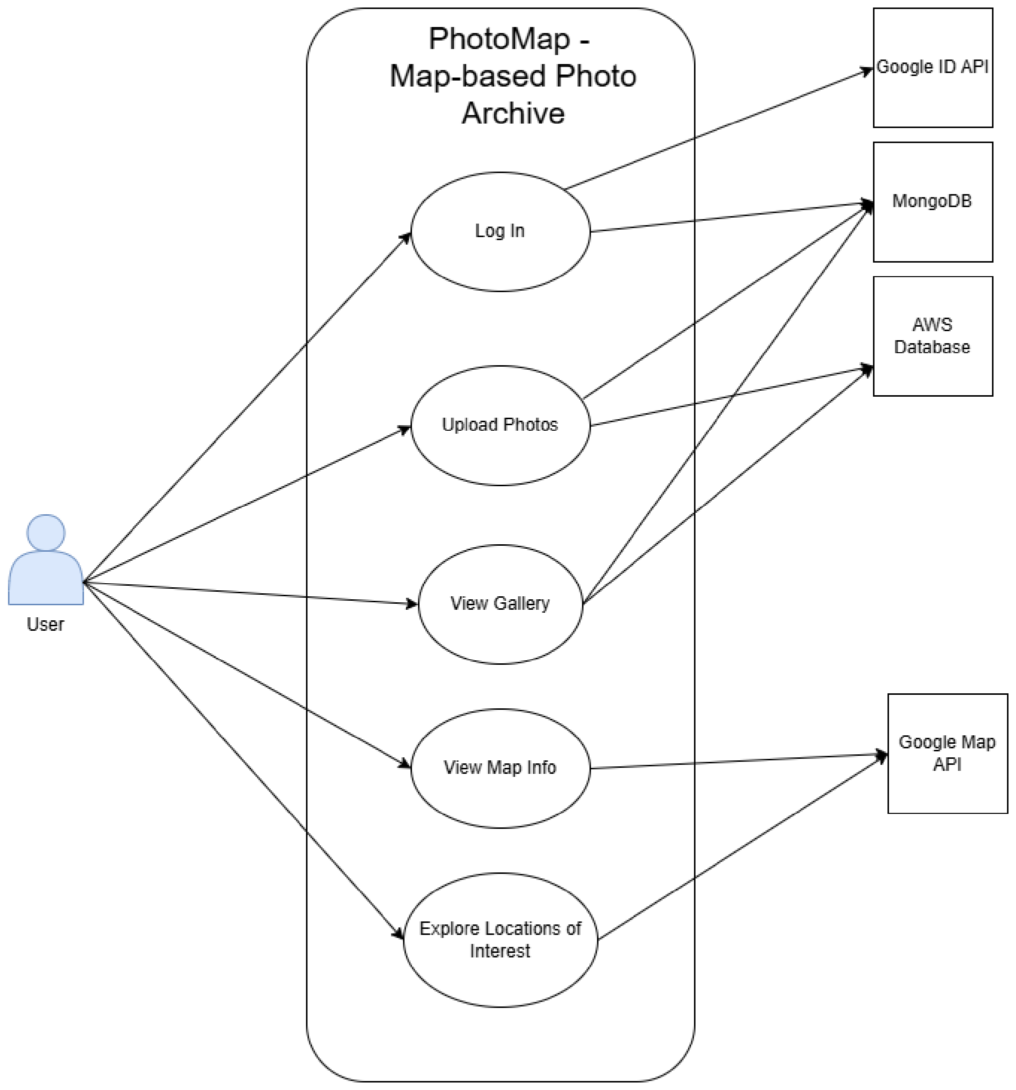
- New name better reflects the functionality of use case

2. Project Description

PhotoMap: Personalized map-based photography assistant and archive. Users can upload photos and have them displayed on the map, at the same location where the photo was taken. Uses can view their uploaded photos and navigate between their location markers on the map. PhotoMap will also provide recommendation information about potential locations of interest, based on the geographical data of location-markers and user's photo history.

3. Requirements Specification

3.1. Use-Case Diagram



3.2. Actors Description

1. **Indie photographers (User)**: take photos and update locations with photos; receive recommendation of places of interests, as well as sharing their photos for other photos.

3.3. Functional Requirements

1. **Log In**

- **Overview**:

    1. Create Account

- **Detailed Flow for Each Independent Scenario**:

    1. **Create an Account**:
        - **Description**: The user creates a new account in the system by Google sign-in. The Google Sign-In API handles existing user log-in and creates new google users.
        - **Primary actor(s)**: User
        - **Main success scenario**:
            - 1.The user navigates to the "Sign Up" page.
            - 2.The user is taken to the Google Sign-in API.
            - 3.The system confirms successful account creation and provides login prompts.
        - **Failure scenario(s)**:
            - 2a. Google Sign-in API returns Error
                - 2a1. The system displays an error message prompting the user to try sign-in again.
            - 3a. The system cannot store the user data due to a server error or network issue.
                - 3a1. The system displays an error message and prompts the user to try again later.

2. **Upload Photos**

- **Overview**:

    1. Create Marker
    2. Upload Photos

- **Detailed Flow for Each Independent Scenario**:

    1. **Create Marker**:

        - **Description**: The user creates a marker, pinpointing a location on the map. They can use this location marker to upload and view their photos
        - **Primary actor(s)**: User
        - **Main success scenario**:
            - 1.The user clicks on any non-marker-occupied location on the map. A form appears prompting the user to fill in marker info.
            - 2.The user fills in the marker title and selects a marker color from the form, and clicks "Add" to confirm.
            - 3.The marker is created and saved to the database. A message appears notifying the user that the marker is successfully added.
            - 4.The camera moves to center on the newly created marker.
        - **Failure scenario(s)**:
            - 3a. The user changes their mind and clicks cancel to abort marker creation
                - 3a1. The bottomsheet disappears and no marker is shown on the map.

2. **Upload Photos**:
    - **Description**: The user uploads photos from their local device to the system by clicking "Upload Photos".
    - **Primary actor(s)**: User
    - **Main success scenario**:
        - 1.The user selects a marker and the upload photo button appears. It should not appear if a user does not have a button selected.
        - 2.The user clicks the upload button, and a bottomsheet appears prompting the user to select a photo from their device.
        - 3.The user selects a photo from their device and clicks submit.
        - 4.The system displays a success message and shows the newly uploaded photos in a recyclerview update above the marker.
    - **Failure scenario(s)**:
        - 3a. The user tries to upload a null image.
            - 3a1. The system outputs a message: No image selected.
        - 4a. Network or server error: the upload fails due to a connection issue.
            - 4a1. The system notifies the user and offers the option to retry or cancel.

3. **View Map Info**
    - **Overview**:
        1. View Map Info
    - **Detailed Flow for Each Independent Scenario**:
        1. **View Map Info**:
            - **Description**: The user can pan around the map to see their markers and click on markers to see previously uploaded photos.
            - **Primary actor(s)**: User
            - **Main success scenario**:
                - 1.The user navigates to the “Map” section.
                - 2.The system loads the map interface with the user’s current location and saved locations.
                - 3.The user can zoom in/out or pan around the map.
                - 4.The user selects a specific location or marker to view more details (e.g., marker name, photo preview, option to delete marker).
            - **Failure scenario(s)**:
                - 2a. The system fails to retrieve marker or photo data from the backend.
                    - 2a1. An error message is displayed indicating that marker and photo data fetching has failed.

4. **Explore Locations of Interest**
    - **Overview**:
        1. Receive Location Recommendation

- **Detailed Flow for Each Independent Scenario**:

    1. **Receive Location Recommendation**:
        - **Description**: The system provides location recommendations based on previously uploaded photos and marker density. Once the "Recommend" button is clicked, the user receives a summary of their major photo themes and most active locations. The recommendation also includes a list of locations matching the user's preferences.
        - **Primary actor(s)**: User
        - **Main success scenario**:
            - 1.The user clicks the button "Popular Location Notification"
            - 2.The system displays a bottomsheet with the user’s most active location and relevant tags. The bottomsheet also contains a list of recommended locations.
            - 3.The user selects a recommended location and the system creates a marker at the selected location.
        - **Failure scenario(s)**:
            - 2a1. User uploads no images or creates no markers before clicking "Recommend".
                - 2a1.1. The system displays a message prompting the user to create more markers and upload more images.
            - 2a2. No relevant recommendations are available
                - 2a2.1. The system notifies the user that there are no relevant locations that match the user’s preferences.
            - 4a. The system encounters a server error or network issue and cannot fetch recommendations.
                - 4a1. The system displays an error message and prompts the user to retry later.

5. **View Gallery**

- **Overview**:

    1. View and Share Images inside Gallery

- **Detailed Flow for Each Independent Scenario**:

    1. **Share a Photo from Gallery**:
        - **Description**: The user goes to the photo gallery and chooses to share photos (one photo at a time) to the other user of the app.
        - **Primary actor(s)**: User
        - **Main success scenario**:
            - 1.The user navigates to “Gallery” and selects a picture for sharing.
            - 2.The user clicks the image that they want to share and goes to full screen view mode.
            - 3.The user click the share button on the bottom right corner below the image.
            - 4.The user can either manually enter the user email of the people they want to share to or click the dropdown menu to select the person if he/she is

already the user's friend.
        - 5.Users click the button on the pop up window with the text “share” to share the image.
        - 6.System confirms the image has been shared.
    - **Failure scenario(s)**:
        - 3a. If the image is shared by someone else to you, then you don’t have permission to share this image and thus the share button will not appear.
        - 6a. The system notifies the email that the user wants to share the image to is either invalid or not a user of this app.
        - 6b. The system encounters a server error or network issue and cannot process the sharing request from the user.

3.4. Screen Mockups

3.5. Non-Functional Requirements

1. **Photo Gallery Upload Speed**
    - **Description**: The application should ensure that photo uploads to the gallery are achieved at a minimum of 6s
    - **Justification**: According to a study of user preference on upload speed, 10mb/s is sufficient to handle photo and video upload. Regular photos are usually 4-5 mB in size, rendering 10mB/s an appropriate speed for photo upload. A slow upload experience can frustrate users and discourage them from archiving their photos efficiently. Therefore, for uploading more than one image to app should not take more than 6s
2. **Recommendation Delivery Speed**
    - **Description**: When retrieving the recommendation locations, the buffer time should be no more than 2 seconds.
    - **Justification**: Companies such as Google recommend server response thresholds to be around 800 milliseconds. Hence, we determine a suitable requirement of a response time not exceeding 2 seconds, accounting for both backend server response and Places API response.

4. Designs Specification

4.1. Main Components

1. **Map**

    - **Purpose**: The map component contains all the interactions with the map in our project.
    - **Interfaces**:
        1. getRecommendation
            - **Purpose**: Get the location recommendation based on user data. This will return the information that frontend can use it to search a place.
            - **Parameter**: googleEmail
            - **Return Value**: JSON format, with recommend position (latitude and longitude), and tags

2. **Image Services**

- **Purpose**: Our project is a photomap, obviously photos are a very important part of our project. All the photo related functionalities will be counted in this component
- **Interfaces**:
  1. imageUpload
     - **Purpose**: Users can upload the image to let the image become a part of the photomap.
     - **Parameter**: photo (file) , uploadby (googleEmail), location (JSON object)
     - **Return Value**: image data the store in the database or error message
  2. deleteImage
     - **Purpose**: Users can delete any image they have uploaded.
     - **Parameter**: image filename
     - **Return Value**: Success Message or error
  3. getImage
     - **Purpose**: Get one image and its information from the cloud
     - **Parameter**: image filename in database
     - **Return Value**: all image information in database and a presign URL that can display the image
  4. getImagesByUploader
     - **Purpose**: Get all image and their information from the cloud for the specific user
     - **Parameter**: googleEmail
     - **Return Value**: image information in database and presign URL that can display the image for all the image
  5. getAllImages
     - **Purpose**: Get all image and their information from the cloud
     - **Parameter**: none
     - **Return Value**: image information in database and presign URL that can display the image for all the image
  6. shareImage
     - **Purpose**: Share the images selected to a specific user
     - **Parameter**: googleEmail (current user), googleEmail(share to user), image filename
     - **Return Value**: Success Message or error
  7. analyzeImageLabels
     - **Purpose**: Call AWS Rekognition to indicate the tags of the image
     - **Parameter**: image filename
     - **Return Value**: tags of image
  8. analyzeImageModeration
     - **Purpose**: Call AWS Rekognition to indicate the moderation of the image
     - **Parameter**: image filename
     - **Return Value**: moderation tags of image
  9. processImage
     - **Purpose**: Convert input image to .jpg or .png format. Makesure the MIMETYPE is correct when uploading to S3
     - **Parameter**: input file from imageUpload method
     - **Return Value**: image with available type
  10. updateImageDescription
      - **Purpose**: Update the metadata to the photo in database

        - **Parameter**: filename name, user email, description, location
        - **Return Value**: JSON with updated full image metadata
    11. deleteAllImagesByUser
        - **Purpose**: Delete all images that uploaded by a user
        - **Parameter**: user email
        - **Return Value**: action result
    12. getSharedImages
        - **Purpose**: Get all images that shared by or with user
        - **Parameter**: user email
        - **Return Value**: JSON that contain a lists of image metadata
    13. cancelShare
        - **Purpose**: Remove the shared to user of an image
        - **Parameter**: user email, image filename
        - **Return Value**: action result
    14. cancelShareForUser
        - **Purpose**: Remove one of the shared to email from image metadata, cancel share one by one
        - **Parameter**: user email, image filename, receiver email
        - **Return Value**: action result

3. **User**

    - **Purpose**: User component is mandatory as we will store user information. This component will include all interactions directly related to the user. Also, some functions for Administrator use only.
    - **Interfaces**:
        1. postUser
            - **Purpose**: Upload a new user or update if user exist.
            - **Parameter**: googleEmail
            - **Return Value**: Success Message
        2. getProfileInfo
            - **Purpose**: Users are able to view their personal profile, additional ban, warning history information for supervisor
            - **Parameter**: googleEmail
            - **Return Value**: JSON format of user's profile information
        3. updateProfile
            - **Purpose**: Request to update the personal profile.
            - **Parameter**: googleEmail, new profile information
            - **Return Value**: Success Message or error
        4. deleteUser
            - **Purpose**: Delete a user and it's information in database
            - **Parameter**: googleEmail
            - **Return Value**: successed message
        5. getUserList
            - **Purpose**: get a list of all users from data
            - **Parameter**: none
            - **Return Value**: JSON that contains all users with their data in database

6. removeLocation
    - **Purpose**: remove one location from the user
    - **Parameter**: location, user email
    - **Return Value**: action result
7. addFriend
    - **Purpose**: add a friend email to the user's friend list
    - **Parameter**: user email, receiver email
    - **Return Value**: action result
8. deleteFriend
    - **Purpose**: remove a friend email to the user's friend list
    - **Parameter**: user email, receiver email
    - **Return Value**: action result
9. getFriends
    - **Purpose**: remove one location from the user
    - **Parameter**: user gmail
    - **Return Value**: list of email

4.2. Databases

1. **MongoDB**
    - **Purpose**: We will be using MongoDB to store and track a list of properties of users. We will store the user’s login token, status, etc.

4.3. External Modules

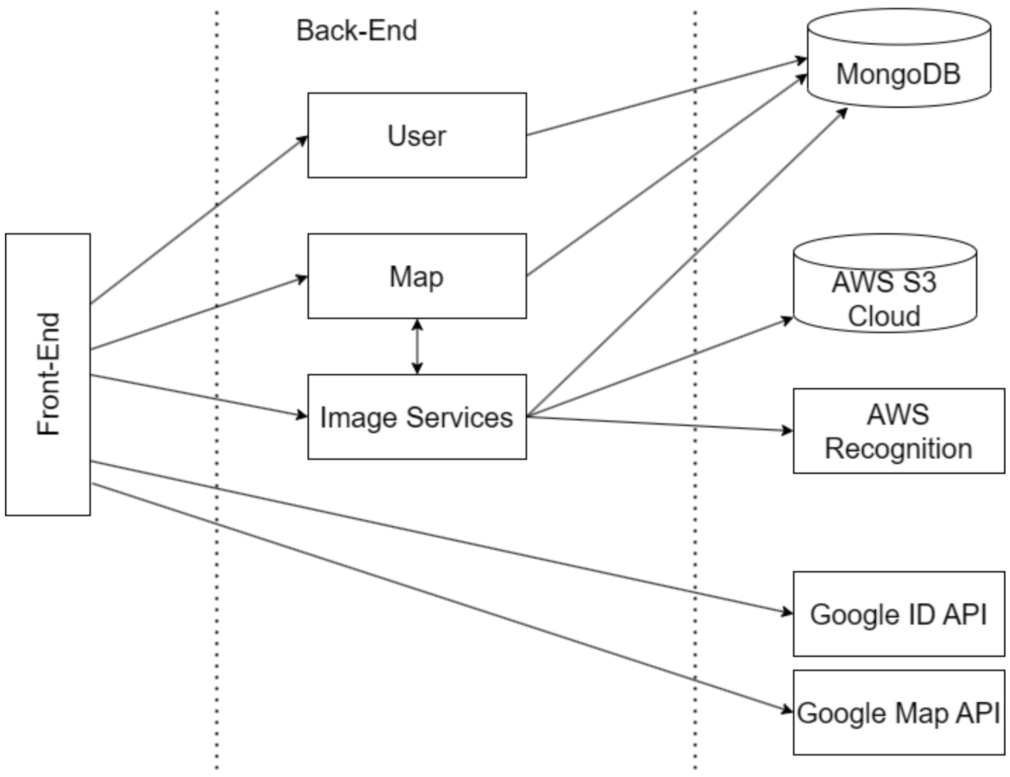
1. **Google Map API**
    - **Purpose**: The graphical user interface of our project is a map, and all functionalities of the app are all around this map. Hence Google Map API is mandatory.
2. **Google ID API**
    - **Purpose**: We are planning to make users login/create accounts based on their Google account. Hence Google ID API is required.
3. **AWS Rekognition**
    - **Purpose**: We want to check if images uploaded by a user are appropriate or not, and the tags of this photo. AWS Rekognition can help us indicate that.

4.4. Frameworks

1. **AWS EC2 Instance**
    - **Purpose**: Provide a online cloud for database connection between backend and frontend
    - **Reason**: It is a powerful cloud provider that can provide a stable connection cloud server
2. **AWS S3 Cloud Storage**
    - **Purpose**: Provide an online cloud data storage for image storage and is mandatory in order to use some external modules
    - **Reason**: MongoDB and SQL-like databases are not ideal for storing images. Alternatively, we will use AWS Rekognition as our model to detect uploaded images is appropriate, and it is required to use AWS S3
3. **AWS Route 53**
    - **Purpose**: Provide an domain to our EC2 instance

- **Reason**: In case to use SSL certification. We need to have a stable domain and record our aws public DNS. Otherwise, we cannot use https to access our routes.

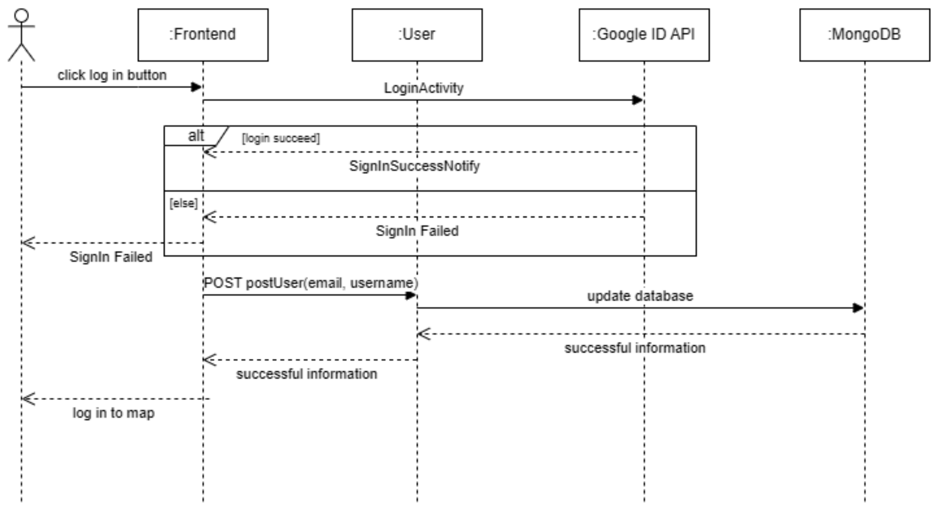
4.5. Dependencies Diagram



4.6. Functional Requirements Sequence Diagram

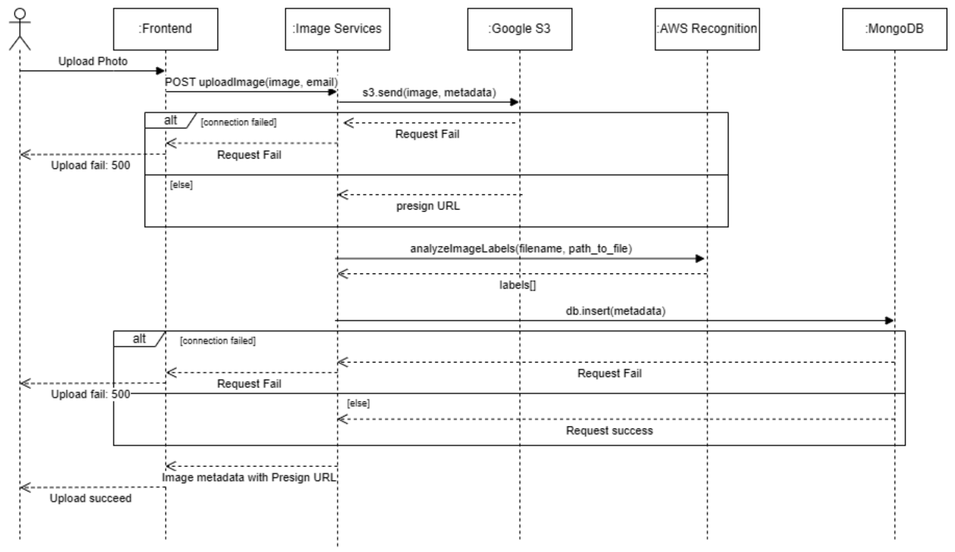
1. **Manage Account**

Log In



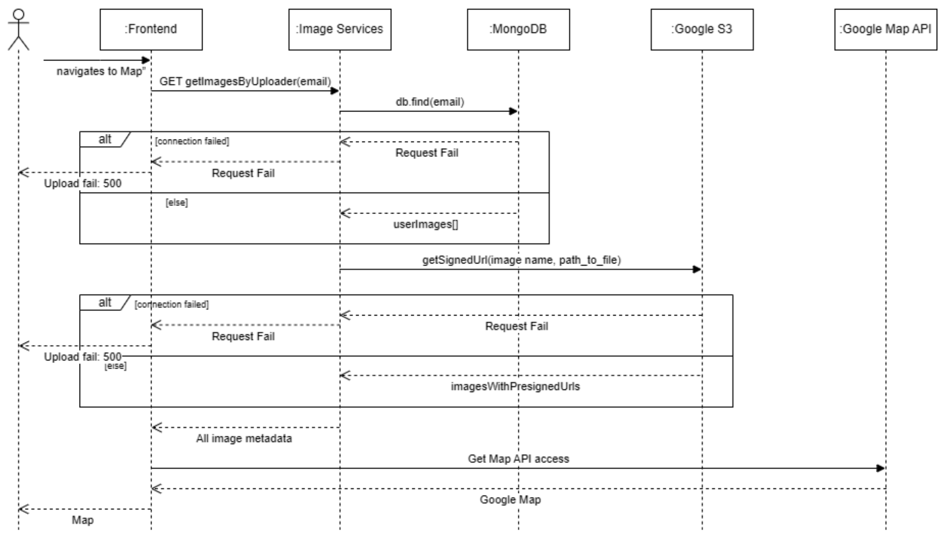
2. **Upload Photos**

Upload Photo



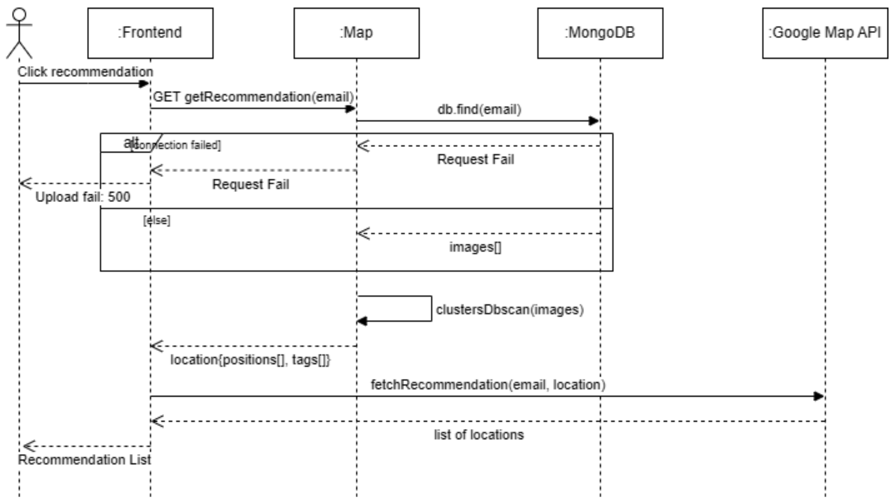
3. View Map Info

View Map Info

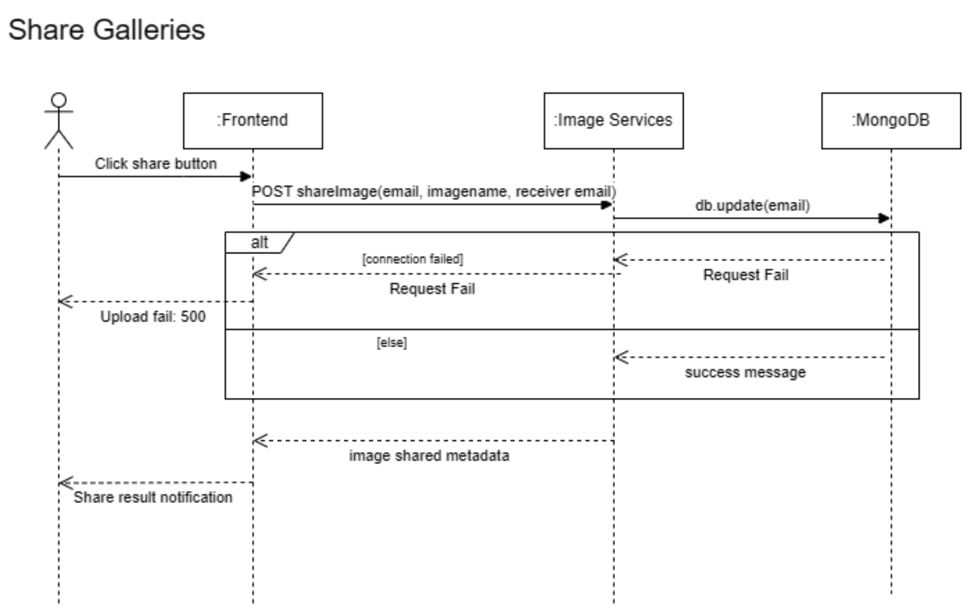


4. Receive Location Recommendation

Receive Location Recommendation



5. Share Galleries



4.7. Non-Functional Requirements Design

1. **Photo Gallery Upload Speed**
    - **Validation**: We will be using AWS S3, a cloud storage that enables multiparts uploading, optimizing file transfer by splitting large files into smaller chunks. Additionally, we will introduce an option to compress images to reduce file size.
2. **Recommendation Delivery Speed**
    - **Validation**: We will store the user locations histories and images content tags into database and associate with correct user. Then, when we try to get hte recommendation, it will use the exist knowledge to process. This will make the entire process faster.

4.8. Main Project Complexity Design

Location Recommendation

- **Description**: A smart algorithm that uses the user's data to provide a new location that user may interested in.
- **Why complex?**: This algorithm uses the user's data history, photo locations, tags of photos. It also uses Density-Based Spatial Clustering of Applications with Noise (DBSCAN) algorithm to analyze the photo locations to provide locations that the user can easily reach.
- **Design**:
    - **Input**: User email, user photo location information, photo tags information
    - **Output**: A position include latitude and longitude, and a array of tags.
    - **Main computational logic**: Use DBSCAN to clustering the user location history. Then pick one (or more in future improve) most famous location cluster. Then, based on the position in this cluster, calculate a new position in this region. Next, use the photo information that in this cluster, to figure our the top three famous tags.
    - **Pseudo-code**:

```
async getRecommendation(req: Request, res: Response, next:
NextFunction)

    // Fetch only images uploaded by this user
    const db = clinet.db("images");
    const images = await db.collection("metadata").find().toArray();

    // Filter out invalid lat/lng values
    const points = images.filter().map()

    // Fit to DBSCAN model and predict the cluster
    const clustered = DBSCAN(points)

    // Track the largest cluster
    const clusterData = clustered.features.forEach(cluster => {})

    // Get the largest cluster's data and tags
    const largestCluster = clusterData[largestIndex]
    const allTagsInLargestCluster = largestCluster.tags[]

    // Compute average lat/lng for the largest cluster
    const avgPosition: [number, number] = largestCluster.reduce().map()
as [number, number];

    // Count tag frequencies & get the top 3
    const tagCounts = allTagsInLargestCluster.reduce();

    const topTags = Object.entries(tagCounts).sort().slicS().map();


    // Send the location and tags
    res.status(200).send({
        popularLocation: {
            position: { lat: avgPosition[1], lng: avgPosition[0] },
            tags: topTags
        }
    });
```

5. Contributions

- Alex Cheng: Generating Sequence diagram and dependencies diagram. Modifying component's interface. Generating complexity
- Ray Yu: Generating components, and database, external modules, and frameworks
- Jiashu Long: Creating main actors, functional requirements and non-functional requirements. Also generating use-case diagram