6.170 Project 3: Food Hunter
Phase 3
Team Design Doc
Members: Rebekah Cha, Yachi (Judy) Chang, Dana Mukusheva

Programming

Modularity

Our code demonstrates modularity by separating model, view, and controller:

- View
    - Our one-page application currently has 4 components: a map, a list of events the user created, an add event form, and a subscription list.
    - These components takes input from the user and send it to the Controller via AJAX calls, which returns a JSON object containing the result data. At no point does the UI interacts with the database, but retrieves all the data it needs via API calls.
    - The client HTML and JavaScript files are also the only files that manipulates the UI in any way. No routers or database methods contribute any information on how the UI should look; they only send data.
- Model
    - Our database is called solely by the Controller, or the router methods.
- Controller
    - The Controller takes input from the View, and retrieves data from the Model database accordingly. The Controller then takes the returned data from the database and formats it into the JSON object the View requires.
    - All JSON objects passed back to the View is of form: {statusCode: code, content: something }, where the statusCode indicates whether the data retrieval was successful. If successful, the field would then contain the data for View to manipulate.

Verification

All mongoose functions enables us to catch and handle errors while connecting and querying the database. Also, we run additional checks and validations on the user input data, to avoid nonsense or repeating input.

- Runtime assertions:
    - We verify the correctness of the server-client data exchange by sending the returned JSON object with a status code. We send 200 when the data is successfully retrieved, 404 when the object the user requested is not found, 400 if the user sends in invalid parameters, and 500 for any errors in

mongoose. With any status code other than 200, we also include a message parameter detailing the error.

- Whenever client-side receives a non-200 status code, which identifies server interaction failures, we send an alert and redirect to the main page.
- Whenever a new event instance is created, we verify that the inputted date and time have valid value, e.g. date is not in the past or end time is smaller than the start time.
- Whenever a client is trying to subscribe to the same type of events, his or her action is ignored. Thus we avoid storing duplicates in the Subscriptions model.

- Unit tests:

  We provided QUnit test suites for all API blocks, such as Events, Subscriptions, Locations and Users. For each block there are ajax calls for all available methods ('POST', "GET', "PUT", "DELETE"). We form some dummy data and send it to the server; if the attempt is successful, the server sends back either retrieved or newly created object from the database. Then, in the QUnit assert method we compare the details of the received object and what we expect to see. Since all the methods require authenticated user, whenever in the test routes, we create a dummy user with kerberos "test". **You can find test suites in the main page right after *login* input box.**

Design Changes

(Transition from phase 1 to phase 2)

- **Models**
  1. The new design redefined the schema for Location. Unlike the previous version, the newer one does not contain room number (which is now considered as a part of the event description), but stores the name of the building if it's available.
  2. The new design includes a model for users. Collection "Users" contains information about each user, such as list of events the user hosts and list of subscriptions the user has chosen.
- **Use of external web agents**
  1. Mapbox: marks locations of the ongoing events on the campus map
  2. SendGrid: emails to the list of chosen users
  3. MIT Map API: provides with data about MIT buildings, their names and building numbers
  4. MIT People API: provides list of MIT affiliates, including their kerberos

- **Authentication challenge**
  In the initial design, we decided to grant access only to the MIT affiliates. We originally wanted to authenticate using MIT certificates, but wasn't able to implement it correctly. Hence, we switch over to having the user enter his or her kerberos, and we would search for it via the MIT People database. If the kerberos is valid, the user is logged in.

Design Overview

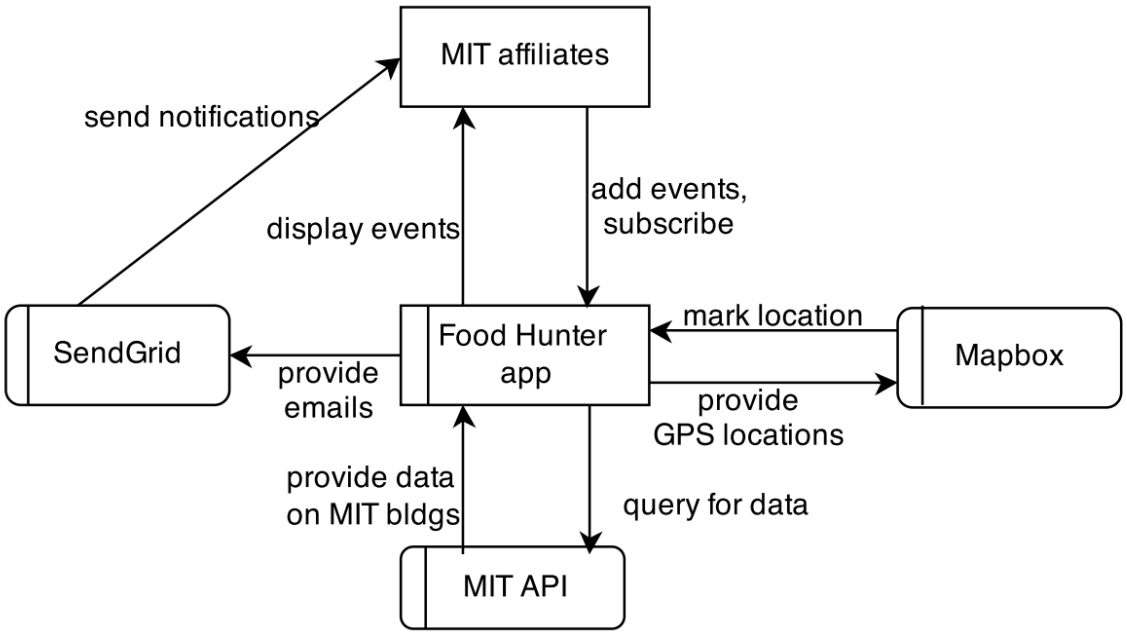
Food Hunter is a web application that organizes and visualizes free food events taking place on the MIT campus. MIT affiliates can post events to the app, and our app will list these events on a master calendar and also provide a real-time map of where they are for all MIT affiliates to see. Users can also subscribe to email notifications about events in their preferred time ranges and/or locations.

Our big idea is to organize MIT's free food events data. We feel the current email spamming approach that many groups adopt is not effective. It creates a vicious cycle of users getting spammed, creating a filter to prevent spam, groups needing to spam more to publicize, and so on. The free food mailing list is also hectic at best. Our app aims to visualize these events in a clean and meaningful way, so hungry students can easily see where food is available, without having their inboxes explode or having to dig through the spam.

Some specific purposes of our app includes:

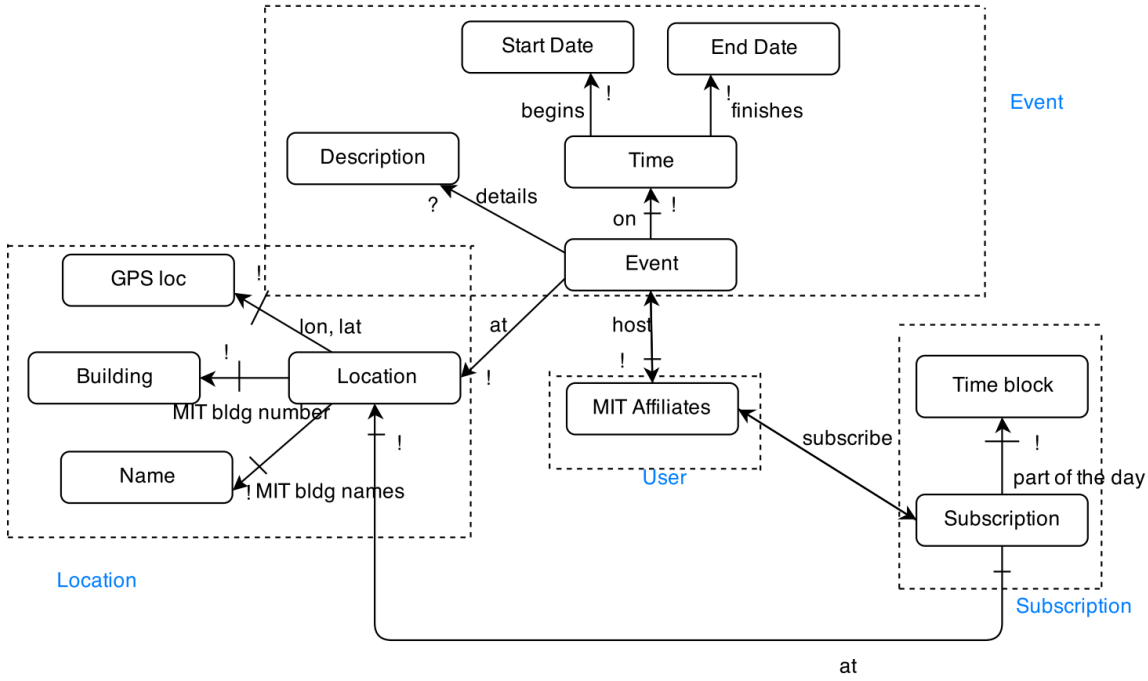
- Allowing MIT organizations to notify interested parties about events with free food. This can aid with their event publicizing.
- Reducing the amount of wasted leftover food.
- Allowing all MIT affiliates to easily see when/where free food will be available and/or when it's gone

Context Diagram



The above diagram demonstrates the context our app works in. Only MIT affiliates are allowed to access the information of our app, because we want to prevent non-affiliates from taking advantage of MIT's resources (e.g. attending events that target the MIT community). Our application also uses the MIT Map API to retrieve all locations on campus, the Mapbox API to create the map on the UI, and the SendGrid service to send emails to our users.

Data Model



The above demonstrates our data model design. Each Event contains information like the time, location, and status of the particular event, and each Event is linked to a host MIT affiliate, who posted the event to Food Hunter. Each MIT affiliate can also subscribe to email notifications about events at specific locations and/or times.
In the actual implementation we define four database collections, that are denoted with contour lines: Event, Location, User and Subscription. In the diagram, boxes with the corresponding names (for users, it would be MIT Affiliates box) represent object IDs within each collection. All other boxes represent collection fields. If there is an arrow from one collection to the other, then, there is a reference from one collection to another.

Concepts:

- **Event:** Representation of any free food event ongoing or upcoming on campus. Each Event and its details is visualized on the online real-time map. The Event can be hosted only by one single user. The Event can be viewed and updated by any MIT affiliate. The Event is time and location specific, but does not necessarily has a detailed description. This fulfills our purposes of allowing MIT organizations to notify others about their free food and other affiliates to see these events.
- **Subscription**: Each MIT affiliate can subscribe to food event notifications based on time and location preferences. Each Subscription is associated with one time and one location, and each user can have multiple Subscriptions. When an event is

added or updated, a notification is sent out to the users who are subscribed to the related time and location. For example, one can subscribe to Building 32 mornings and W20 evenings. This helps allow MIT affiliates know right away when free food is available with less spam in their inbox.
- **Location:** Each building (e.g. 32 is Stata Center) is linked to a GPS location, which we retrieve from the MIT Campus Map API. This helps us create a map of the events.

API

Our API comprised of four main blocks: Events, Locations, Subscriptions and Locations.

- Events API contain GET, PUT, POST and DELETE method. All authenticated users can see events on the main page map. Each authenticated user can also see a list of his/her individual subscriptions and events he/she hosts. Only creators can edit event details or delete events they host.
- Locations API contains only GET methods since all locations objects are predefined and already entered into the database. No one is permitted to change/delete location objects.
- Subscriptions API contains POST, DELETE and GET method. When a method POST called, if there is no such subscription object yet, it will be created and saved in the database. No one is permitted to delete Subscription objects form the database. However, when a method DELETE is called, the user deletes reference to the subscription object in his/her list and a reference to the user is deleted from the Subscription object users list.
- Users API contains only GET methods. Once a User object is created and saved in the database, no one is enabled to delete it. User objects can be changed indirectly via POST/PUT/DELETE calls on the Subscription and Event objects.

Below you can see the table with all HTTP verbs we used and corresponding URLs. We also include the format of JSON objects returned if the action were successful. If the action failed, all JSON objects returned are of format: {statusCode: 500, message: String}, where message details the type of error that occurred. There is a uniform errorRederict method that would first alert the user of the error that had occured and then redirects the user to the login page.

Method	URL	Returned JSON	Behavior
EVENTS			
GET	/events/	**Regular:** {statusCode: 200, events: [Event] } **Error:** {statusCode:500, message: "mongoose error getting events"}	**Regular:** Get all event objects from database **Exceptional:** Returns error if there is a problem with the database
GET	/events/:eventID	**Regular:** {statusCode: 200, event: Event } **Error:** {statusCode:500, message: "Error finding an event"}	**Regular:** Get specified event from database **Exceptional:** Returns error if there is a problem with the database
POST	/events/user/:userID	**Regular:** {statusCode: 200, event: Event } **Error:** {statusCode: 500, message: "Error creating a new event instance"} {statusCode: 500, message: "Error adding an event to user"} {statusCode: 500, message: "Error finding user in database when adding event"} {statusCode: 500, message: "Error adding an event to the User.events”} {statusCode: 400, message:"Event happens in the	**Regular:** Creates new event object in database. Emails out to users who are subscribed to the event’s location and time **Exceptional:** If user is not authenticated, returns error and goes back to the login page

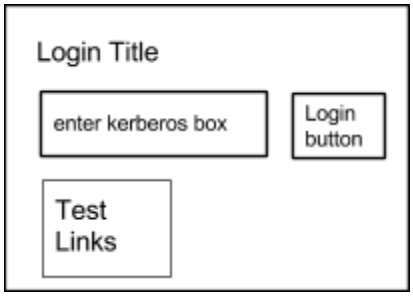
		past"}	
PUT	/events/:eventID/user/:userID	**Regular:** {statusCode: 200, event: Event } **Error:** {statusCode:500, message: "Error while updating the event: eventID"}	**Regular:** Updates specified event in database. **Exceptional:** If user is not authenticated, returns error and goes back to login page
DELETE	/events/:eventID/user/:userID	**Regular:** { statusCode: 200 } **Error:** {statusCode:500, message: "Error while deleting the event: eventID"}	**Regular**: Removes specified event from database **Exceptional:** If event does not exist previously, returns error.
USERS			
GET	/users/:userID/	**Regular:** { statusCode: 200, user: User } **Error:** {statusCode:404, message:"User not found"} {statusCode:500, message:"mongoose get user error"}	**Regular:** Returns specified user from database **Exceptional:** Returns error if there is a problem with the database or no such user exists
POST	/users/login	**Regular:** { statusCode: 200, user: User } **Error:** {statusCode: 500, message: "mongoose create user error"} {statusCode: 500, message: "MIT server error or	**Regular:** Creates new user object in database **Exceptional:** If userID is not valid, returns error and goes back to the login page

		invalid kerberos"}	
SUBSCRIP-TIONS			
POST	/subscriptions/subscribe/user/:userID	**Regular:** { statusCode: 200, subscription: Subscription} **Error:** {statusCode: 500, message: "Error finding the user who wants to subscribe"} {statusCode: 404, message:"Error finding the user who wants to subscribe, no such user"} {statusCode: 500, message:"Error while fingding sub"} {statusCode: 500, message:"Error adding a newly creating sub to user list"} {statusCode: 409, message : "User already subscribes to this."} {statusCode : 500, message : "Error adding an existing sub to user list"}	**Regular:** Updates the specified user's subscriptions and also the specified subscription's userlist. **Exceptional:** Returns error if there is a problem with the database. Gives a popup message if the user is already subscribed to a certain location and time.
DELETE	/subscriptions/subscribe/user/:userID	**Regular:** { statusCode: 200} **Error:** {statusCode: 404, message: "Subscription does not exist anyways"} {statusCode: 500, message:"Error	**Regular:** Removes the subscription from the user's subscription list and also removes the user from the subscription's userlist **Error:**

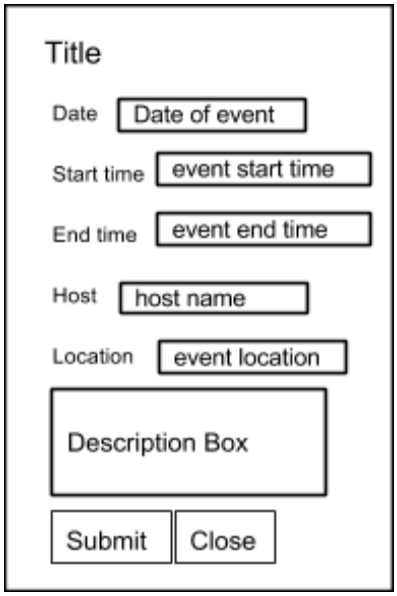
		deleting user from subscription list"} {statusCode: 500, message: "Error deleting sub from user's list: kerberos"}	Returns an error if there is a problem with the database
GET	/subscriptions/:user_id	**Regular:** { statusCode: 200, subscriptions: [Subscription]} **Error:** {statusCode:500, message:"mongoose find user error"} {statusCode:404, message: "mongoose find user error"}	**Regular:** Returns the subscriptions of the specified user **Exceptional:** Returns error if there is a problem with the database
LOCATIONS			
GET	/locations/	**Regular:** { statusCode: 200, locations: [Location]} **Error:** {statusCode: 500, message: "mongoose error retrieving locations"}	**Regular:** Gets all of the locations from the database **Exceptional:** Returns error if there is a problem with the database
GET	/locations/:locID	**Regular:** {statusCode: 200, location: Location} **Error:** {statusCode: 404, message: "location not found"} {statusCode: 500, message: "mongoose error retrieving location"}	**Regular:** Returns specified location **Exceptional:** Returns an error if there is a problem with the database

User Interface

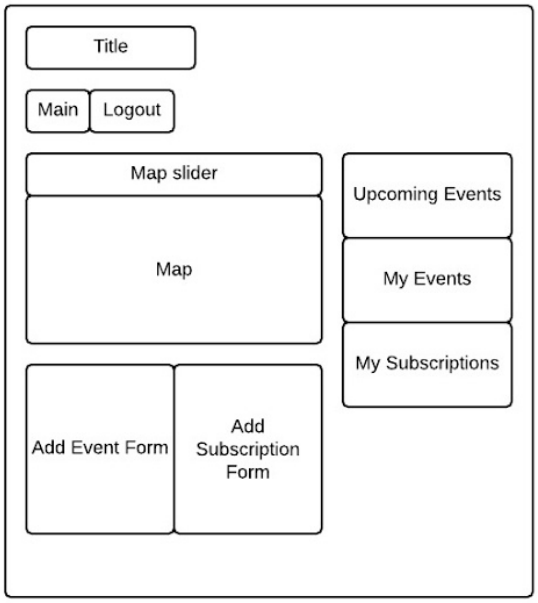
Login Page



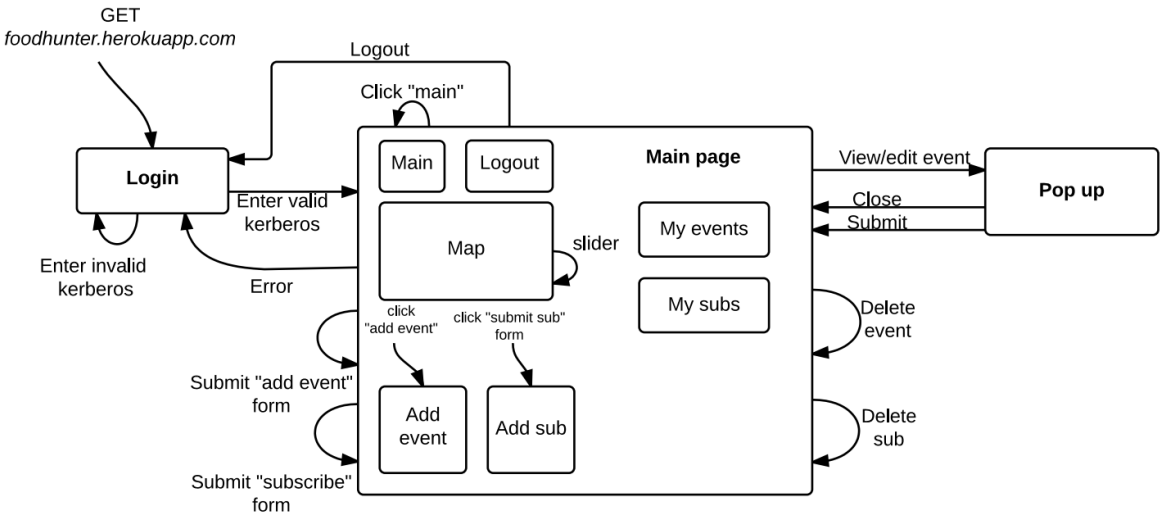
Main Page



View/Edit Event page (popup)



Action flow



The above diagram illustrates the action flow of the application. The application includes three pages: login page, main page and the pop-up window. The only actions allowed on the login page are user authentication routines. Once succeeded, the user is redirected to the main page , otherwise, he/she stays on the same page until valid kerberos is entered. Most of the web page functionality is encapsulated in the actions on the main page, such as add/delete events, create/delete subscriptions, update map. In order to view single event details and edit them, a user calls an action on a main page but modifies the entries in a popup window.

Phase 1 Design Challenges

1. Audience control. The are several options for choosing the audience:
    - Everyone can use the app
    - Everyone can view events, but only MIT affiliates can host events
    - Only MIT affiliates can view and host events
    - **Solution:** We decided that only MIT affiliates can access the web app in order to avoid unwanted public and strangers. Therefore, we will authenticate users by requiring their MIT certificates.
2. Subscription representation. Users may want to be notified of certain free food events depending on location and time. A few possible options we came up with were:
    - One subscription for each user. Each subscription would have a list of locations and a mutable time preference. However, this is difficult to use when querying once an event is added or updated.

    - Multiple subscriptions for each user, with one location and one time preference for each subscription.
    - **Solution:** We decided to have multiple subscriptions for each user for easier querying purposes
3. Event status. Free food can disappear quickly and to save people a trip, there should be an indicator letting users know whether there is still free food or not.
    - Having comments that are attached to the event
    - Having a status options that users would make use of to indicate that there is no more food. This could create issues where users would edit the status that is different from the actual status of the free food
    - **Solution:** We decided to use the status option as it is very straightforward with regards to our purpose of letting users have easy access to where free food is.
4. User representation.
    - Create a separate representation for viewers and hosts. This option requires two separate designs for a general user (viewer) and an event creator (host).
    - Create one single representation for all MIT community members.
    - **Solution:** Keep all users in the same data collection. The simplicity of this option allows any MIT affiliate to be a host and a viewer at the same time. Thus, notification routine will require scanning through only one single collection.

Phase 2 Design Challenges

1. Deployment Choice:
    - Use OpenShift
    - Use Heroku
    - **Solution:** OpenShift supports all the external agents, but the server freezes after a few seconds of use. Heroku, although it requires credit card information to use Mongo, is much easier to deploy.

2. Enable the user to edit his/her existing subscriptions
    - User can add, edit, and delete subscription
    - User can only add and delete, any updates require re-add
    - **Solution:** Since the Subscription objects remain the same once created, in case of enabled editing, the back-end would have been handling two requests: one removing the existing subscription from the user's list and another adding a new subscription to the user's list. Therefore, in order to simplify subscription list maintenance, the user can only delete and add subscriptions.

3. **Users**
   - Simply save the user's kerberos string to refer to the user
   - Create a separate User collection, that stores the user's events and subscriptions
   - **Solution:** Even though the app's nature is centered on subscriptions and events and the only real user information we need is his or her kerberos, we find that having a separate User collection is still much handier, since the UI will be built around the user, and retrieving his or her events and subscriptions would be easier if we just have one collection to search, rather than multiple.

4. **Locations**
   - Store all room numbers with GPS locations
   - Store only building numbers, and make users enter room numbers in descriptions
   - **Solution:** We chose the second because our feature of Subscriptions allow users to subscribe to events of a specific area. This quickly becomes problematic if we had store all room numbers. For example, if the user wants to subscribe to building 5 and building 5 has 50 rooms, then we would need to append the user to 50 different subscriptions. This may become a problem when emailing out and also in reality, sometimes free food is left not in a room, but in a hallway, or just randomly on a table. By allowing users to describe where the food actually is, we also make the app more flexible.

5. **User Authentication**
   - Use MIT Certificates
   - User accounts/password scheme
   - Verify valid kerberos (combined with some sort of security scheme)?
   - **Solution:** We chose the last, although we haven't implemented the security scheme. MIT Certificates look like the most secure option, but the implementation over-complicated our code and we couldn't implement it correctly under our timeline. Our website is designed to be for MIT affiliates only, so we feel the second choice did not make any sense since we know each MIT affiliate already has an unique Kerberos.
   - Our choice posts the threat that technically anyone can access the website as long as they can find someone's kerberos. This can be remedied using passwords, which we are considering implementing in phase 3.

Phase 3 Design Challenges

1. **How to let users edit events**
   - Popup window with event details users can change and then submit
   - Add an edit event form onto main UI page
   - **Solutions:** We chose the first because this makes our main UI page cleaner, and also since only hosts can edit the event; it would not be as frequently used as the other features, and making a separate UI component on the main UI just for the few edits did not make sense. However, this poses the problem that whenever the user edits an event, we must reload the parent page to reflect changes. We didn't have time to investigate how to pass information from the popup back to the parent.

Teamwork

1. **Code on local branch and then merge with master branch instead of directly pushing into master branch.** We found it difficult to revert the master branch back to old commits once we pushed code, and throughout the project, it was sometimes frustrating when we accidentally pushed incorrect code and broke the app. It also wasted time with merge conflicts, especially when we're all working on different versions of the code and pushing around the same time.
2. **Team communication.** It's easier and faster to meet in person and code together than to work remotely. Questions can be immediately resolved, and it's also more motivating when the other people are ~~suffering~~ working with you. It's also essential to go through the design phase together, because people often have different approaches to the same problem, and we feel that our design was a lot more complete when we bounce ideas off each other.