6.170 Project 3: Food Hunter
Phase 2
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: {success: 0, content: something }, where the success parameter is either 0 or 1, indicating 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 object with attribute *success.* Whenever there is a database retrieval error, we set the *success* to be zero and provide the client-side with detailed explanation of the error.

- Whenever client-side receives zero valued *success* message, 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

- **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 API: provides with data about MIT buildings, their names and building numbers

- **Authentication challenge**

In the initial design, we decided to grant access only to the MIT affiliates. While it is still the case, the kerberos authentication routine is still under construction. Possible solutions include working with SSL Certificates or usage of npm packages such as passport-kerberos.

Design Overview

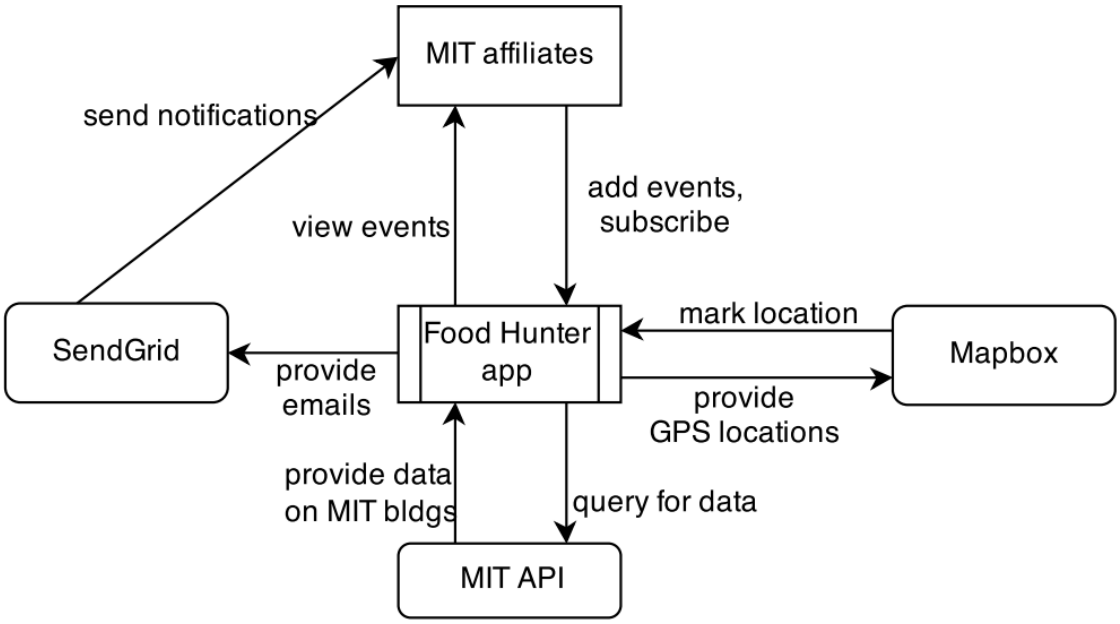
Food Hunter is a web application that organizes and visualizes free food events taking place on the MIT campus. MIT affiliates can post events to the app, and our app will list these events on a master calendar and also provide a real-time map of where they are for all MIT affiliates to see. Users can 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:

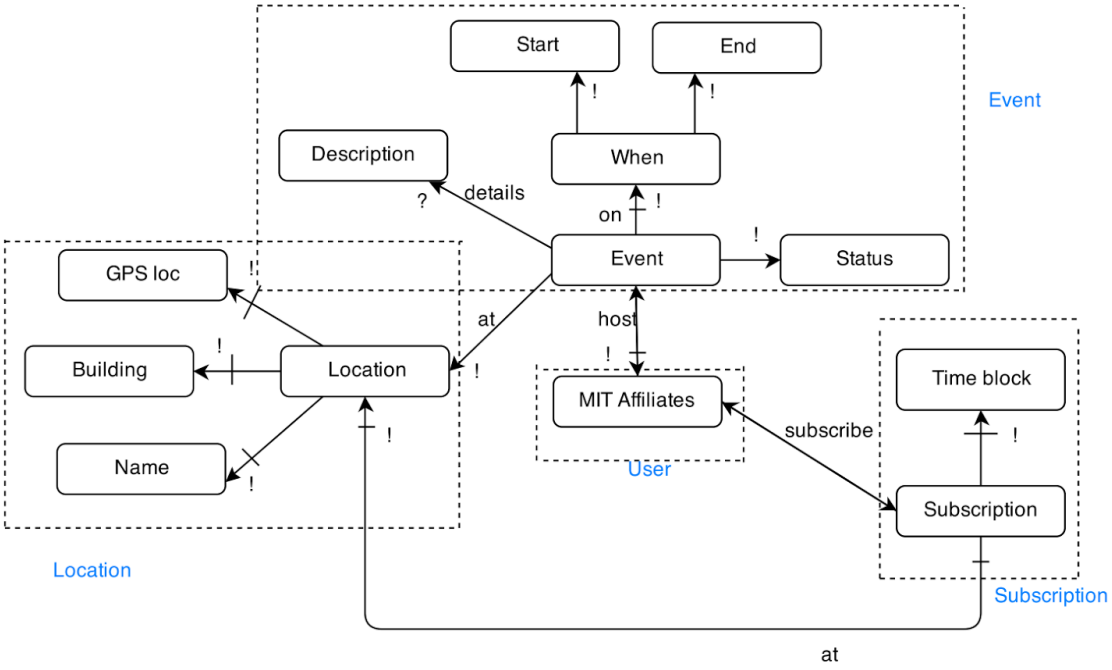
- Allowing MIT organizations to notify interested parties about events with free food. This can aid with their event publicizing.
- Reducing the amount of wasted leftover food.
- Allowing all MIT affiliates to easily see when/where free food will be available and/or when it's gone

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.

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.
- **Status:** Lets users know the current state of the event, such as “Food” and “No food”. The Status is set when the event is created, but it can be modified at any

instance of time by any app user. This would help our purpose of allowing all MIT affiliates easily know if there is still free food or not.

- **Location:** Each room number (e.g. 32-123) is linked to a GPS location, which we retrieve from the MIT Campus Map web service. This helps us create a map of events and also lets users see where the event exactly is.

API

Method	**URL**	**Action**
GET	/events/	Show all events on campus
POST	/events/	Add new event
PUT	/events/:eventID/	Update event
DELETE	/events/:eventID/	Remove event
GET	/users/:userID/	Shows user-specific page
POST	/subscriptions/subscribe/	User subscribes to a bunch of subscriptions
DELETE	/subscriptions/subscribe/	Remove subscription from the current user's list
GET	/subscriptions/	Show all user-related subscriptions
GET	/locations/	Retrieve all location objects in the database
GET	/locations/:locID	Retrieve location-specific info

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 doesn't support SendGrid package

2. Enable the user to edit his/her existing subscriptions
   - user can edit some subscription
   - user can only delete some subscription
   - **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 some current subscription.
3. Collection Users
   - have a separate collection, one doc per user
   - do not have a separate collection, store it in Events and Subscriptions collections
   - **Solution:** For simple event and subscription editing and email sendout, it is very handy and time-saving to store events and subscriptions only related to one single user. Thus, we decided to diverge from the original design and initialize Users collection.