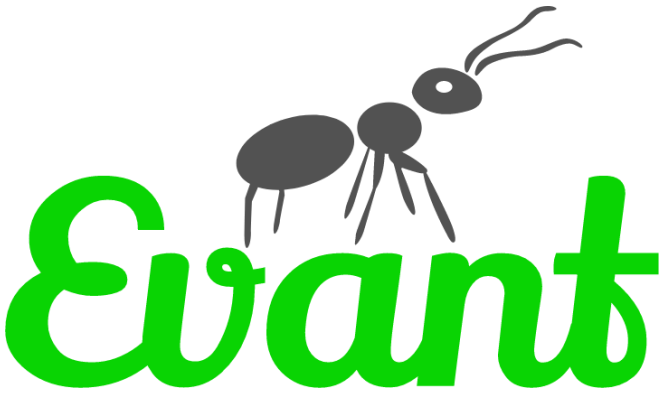


Team 24 CS307 Design Document

Avi Agarwal, Adam McDonald, Soun Kim, Abigail Urnes, Naomi Urnes, Soumil Uppal

Table of Contents

Purpose

People often want to take part in events or activities specifically pertaining to their interests. However, they are discouraged because of the challenge of finding well-organized local events and enough people who are also interested in the same activity. Although current popular methods, such as Facebook and Foursquare, are successful in coordinating events, these products limit users to their current social circle. The aim of the project is to solve this problem through a mobile Android app which would allow users to connect with other people on the bases of common interests and proximity. The app will show public events hosted by other users sorted by distance, time, and specific interests. Events created by users can cover a wide range of activities such as study groups for specific classes, recreational sports, game nights, and social events.

Design Outline

High-level overview:

Evant will use a client-server model to connect users with events hosted by other users. The client will communicate with the database through the server in order to populate the frontend with the requested information. Below is a figure that demonstrates the high-level overview of the system.

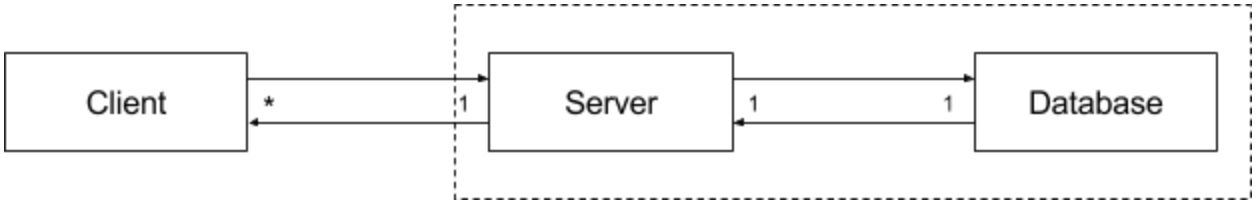


Figure 1. High-level overview.

1. **Client:** Android application created with Android Studio.
    a. Client sends SQL command request to the server
        i. Possible Requests: user creates event, user wants list of events, user joins events, user rates event, user logs in
    b. Client receives a SQL response from server
        i. Possible Responses: app receives list of events, login validation, user information data (settings, rating, etc.)
2. **Server:** Apache server self-hosted on Raspberry Pi.

   a. Handle all traffic to and from database.
   b. Authenticate users via Facebook and database.

3. **Database:** MySQL database self-hosted on Raspberry Pi.
   a. The database will store all non-volatile data needed for the app
      i. Metadata about users
      ii. Metadata about events
   b. There is an API of communicating with a mySQL database across an internet connection for Java. We will use this to query the DB for information.

Application Flow

Figure 2 illustrates the progression of the activity of the app. First, a user opens the app on their Android device. The login request is sent to the database, and is either validated or denied. Once a login request has been validated, the client requests information to fill the main page such as upcoming events and user information. Once the request has been validated, the database sends the the information to the client, and the client fills the main page. From then on, whenever the user accesses a page on the app, the client requests the necessary information from the database to fill that page. The database receives the request and if it is valid, the database sends the client the requested information. If the request is invalid, it throws an error to the client. Once the client receives the necessary information, the client populates the page and then displays it to the user.

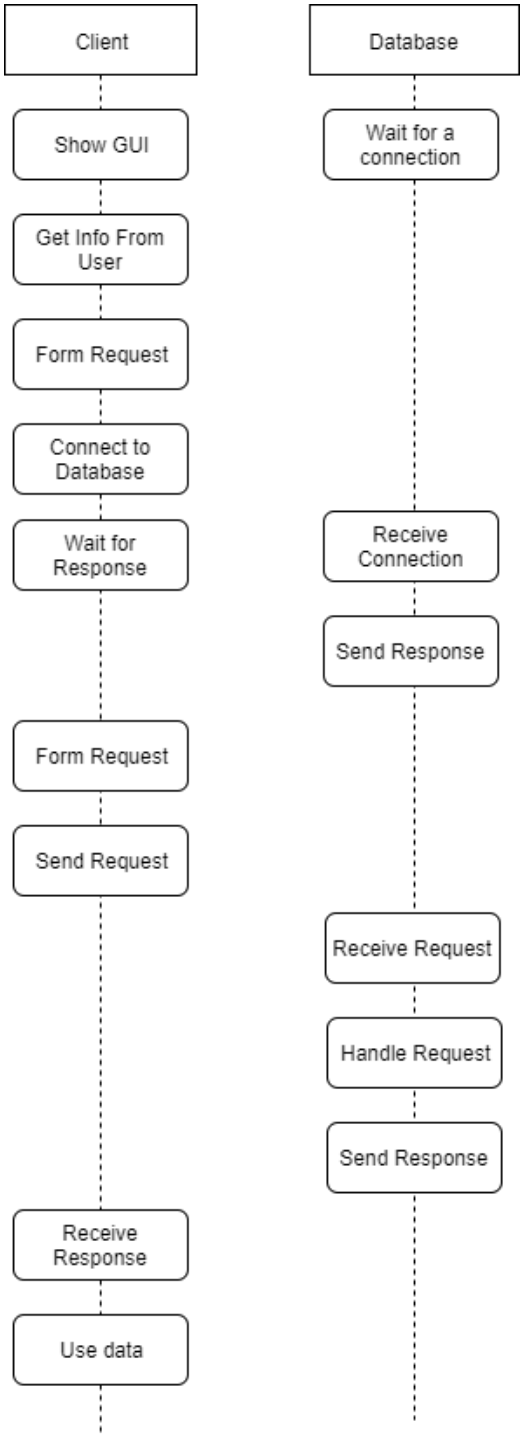


Figure 2. Application flow.

Design Issues

Functional Issues

1. How should events be displayed to users?
    a. Map view with pins.
    b. List view with descriptions.
    c. Both, toggle between map and list view.

    Decision:

    Users should be able to toggle between a map and list view of events. Map view allows users to make selections primarily based on location, while list view allows users to see information from several events in one view.

2. How should the list of previously attended events be displayed to users?
    a. Openly listed on the user's profile.
    b. A button/tab on the profile to bring up the list.
    c. Only displayed privately on the user's device. A button on the navigation bar brings up the list.

    Decision:

    The list of past events a user has attended should be displayed privately. The list should be accessed from a button on the navigation menu. The only reason for this not to be private is social stalking, which is not the

focus of our app.

3. How should the app determine when an event has ended?
    a. Only the event organizer can end event.
    b. The event will be deactivated after certain time.
    c. The event will be deactivated after a certain time, but the organizer can end it early or cancel it.

Decision:

The host of an event should be able to cancel it if necessary, but it should automatically expire after it ends. Only events that come to fruition will go into the database; deleting an event early will also remove its record in the database.

4. How will the users rate event hosts?
    a. Rate out of 5
    b. Thumbs up/ thumbs down
    c. “Will attend again”
    d. Mood rating (Happy, neutral, unhappy)

Decision:

Thumbs up/thumbs down to avoid ambiguity. Thumbs up will equal 1 and thumbs down will equal 0, and the user’s overall rating will be the average.

5. Should users be allowed to see the ratings of other hosts?
    a. **Yes, users should be allowed to see event organizer's rating and decide whether to attend**
    b. No, rating should only be used as a parameter for hosting events and not public

Decision:

Yes. The purpose of the rating is so that prospective guests can be aware of the details of events.

6. What should be the very first screen a user should see (after login)?
    a. A screen with two options: host an event or join an event
    b. The categories list view
    c. **The map with event pins on it**
    d. Profile with upcoming events, recommended events

Decision:

A map with event pins with a floating button to initiate hosting an event.

Non-Functional Issues

7. What backend framework should we use?
    a. Python
    b. Java
    c. PHP

    Decision:

    Java. We will be using Android Studio, a Java-exclusive development environment, to create the app.

8. How are we going to host our backend services?
    a. Using IaaS Services
        i. Amazon Web Services
        ii. Microsoft Azure
    b. Set up our own server using Raspberry Pi

    Decision:

    We have decided to use a Raspberry Pi to host the database because that gives us more control as developers, it leaves room to expand, and gives us more space without having to spend money.

9. What database software should we use?
    a. SQL
        i. MySQL

ii. SQL

iii. SQLite

b. NoSQL

i. MongoDB

ii. Firebase

Decision:

We have decided to use MySQL because it is very efficient, allows us to design our own tables, and is what our developers are most experienced using.

10. How will we handle user authentication?

a. OAuth

b. Verification with database

c. Firebase API

Decision:

Verification with database. Since we have our information stored in our own custom database, it will be most efficient and robust to authenticate the user from the same location.

11. How will we obtain app error and crash reports?

a. BugSense

b. Log API

c. Google Play Developers Console

Decision:

We will use BugSense to obtain reports as it provides an easy to use method using only a few lines of code. The app also provides clearly structured reports.

12. Should we have a seperate table for past events?

a. Have a table just for events that have already happened

b. Have one table to for all events and not remove old events

Decision:

We will keep them in one table. It will be easier to implement and efficient, barring a substantial increase in traffic.

13. What color/design scheme will we follow?

a. Green and white

b. Blue and white

c. Grey and white

d. Green and gray

Decision:

The color scheme will be white with green and gray accents. The green evokes a grassy picnic environment for the ants. See logo.

Design Details

Classes

The Event class is a major class which will serve as the basis for the rendering of event information cards in many different contexts throughout the app. Its variables will include location, name of event, time/date, host, future/past Boolean, category, text description, list of users that have RSVP'd, rating of the host, maximum attendees, and a private/public Boolean. It will get information mainly from its own entry in the database, but it will interact with the pins and the list view of events when an entry is clicked.

The User class will have private variables for username, password, event history, radius, and location. It will have public variables for display name, host rating, and subscribed interests. The User class will interact with the Map class when it sets location and radius, but it will mostly interact with the database when it retrieves the list of event history from the user's Events column. Also, the method calls for Facebook linking will possibly be here.

The Map class will primarily be composed of computations and functions from the Google Maps API. It will have the user's location and the current radius setting, which will be obtained by interacting with the User class. It will also get information from the backend table of events, and maintain a list of events within the current radius. Using all of this information, the Map shows pins via some mechanism of Google Maps. Clicking the pins will result in a method call to the Event class, which will render an Event object.

There will also be a Client class that will be used to talk to the database. It will have public functions that allow it to get specific info from the database. For example, it will have a function validateLogin that takes in a username and a password and then checks that that user exists in the users table in the database. It will return a User object of the user attempting to log in. It will also have functions to get user by name, add user to the database, modify a user in the database. get event by name, get events by location/keyword/category, add a new event, and modify an event. These functions will also account for modifying the tables correctly when adding keywords and categories to events.

Class Interactions

The first class created when a user signs up for Evant will be the User class. Upon opening the app, the user will either be automatically logged in or need to sign up. If the user is automatically logged in, the mainActivity script will use the Client class to request the information of the user from the database. Once the information is passed to the app, the script will create an instance of the User class that represents the current user of the app. If the user needs to sign up, they will be directed to the sign-up page, where they will give the necessary information. Once they finish, their information will be sent to the database using a command from the Client class, and the database will return the user information to the app and internally store the new user. With the new information, the mainActivity script will create an instance of the User class that represents the new user.

If the user wants to look at a list of events, the app will use the Client class to request information from the database about events filtered by the user's specific preferences, which are stored in local variables. The database will return a list of events that pass the filter to the mainActivity script, which will create instances of the Event class for each event in the list. Each Event object will then be placed into an array of Event objects. Using a RecyclerView and card scripting in Android Studio, the events will be displayed to the user as a list of cards. If the user clicks on one of these cards, the app will redirect to a page that displays the information of that specific event. The information will have been already stored in the corresponding Event object. If a user decides to join an event, the Client class will notify the database, which will then update the information about the event and the user. In addition, the client will notify the host that the user has joined their event. If a user clicks on the Create New Event button, the app will redirect them to an Event Creation page, where the user will fill in the necessary information (time, description, location, et cetera). Then, the Client class will pass that information to the database, where it will internally create and store the event and update the list of events that specific user has hosted. From that point on, whenever another user joins the event, the user will be notified.

When the user accesses the map of events, the Client class will request a list of events whose location variable places them within the user's specified radius (facilitated using the Google Maps API). The mainActivity script will then create an instance of the Event class for each event in the returned list. Using the information from each event with Google Maps API, the user will see events as pins on a map. The user can select each pin

and go to a page with information about the event. As discussed above, this information will already be stored in the corresponding Event object, which will be retrieved from the database.

Class Diagram

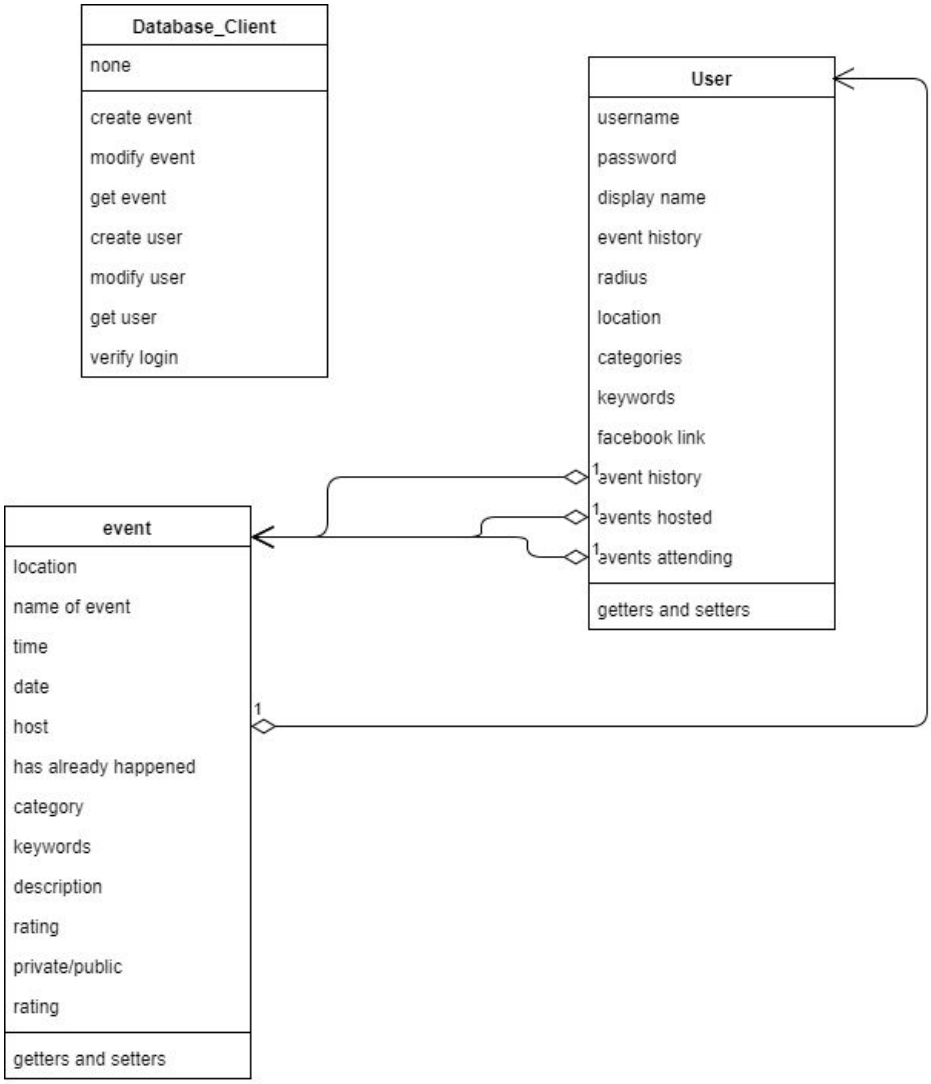


Figure 3. Class diagram.

Activity / State Diagram

Figure 4 illustrates various states of the application. When the user opens the application, it will launch into the Log In UI state. After the Authentication state, the application goes into the Main UI state. From the Main UI state, user can navigate to a specific state according to the user's state.

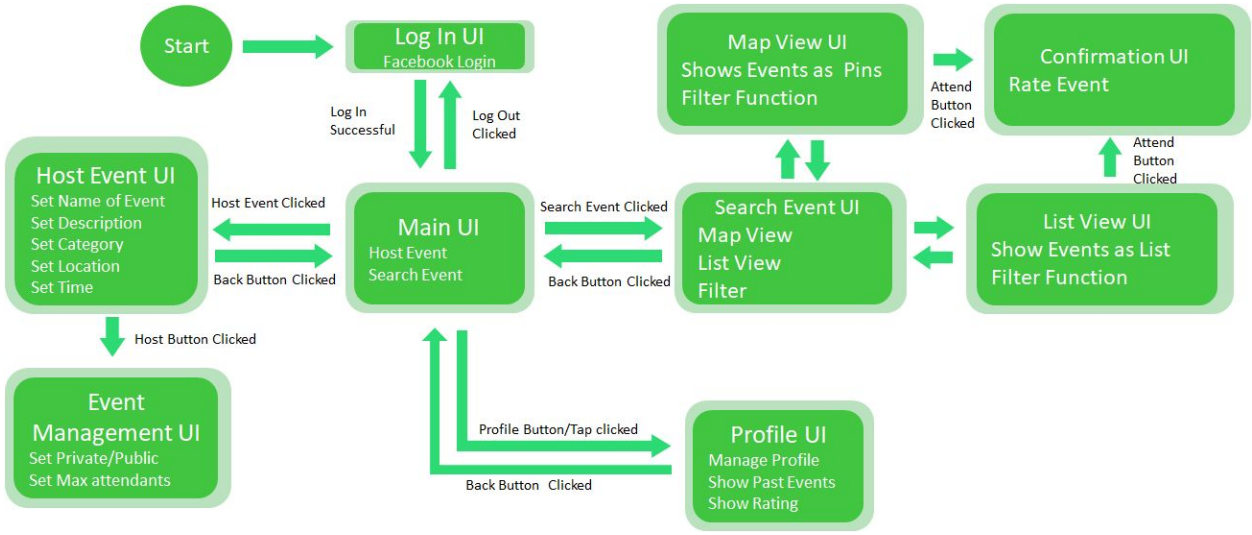


Figure 4. Activity diagram.

App Maintenance Cycle (Bug, Crash and Error reports)

Figure 5 describes the maintenance cycle we will follow to keep the app running. In case of a crash/error, the Bugsense SDK will collect the data from the user's app and upload it to the cloud server. This data will then be available on the Splunk Mint app which can be viewed and analyzed by the developers. The app developers will then update the app accordingly.

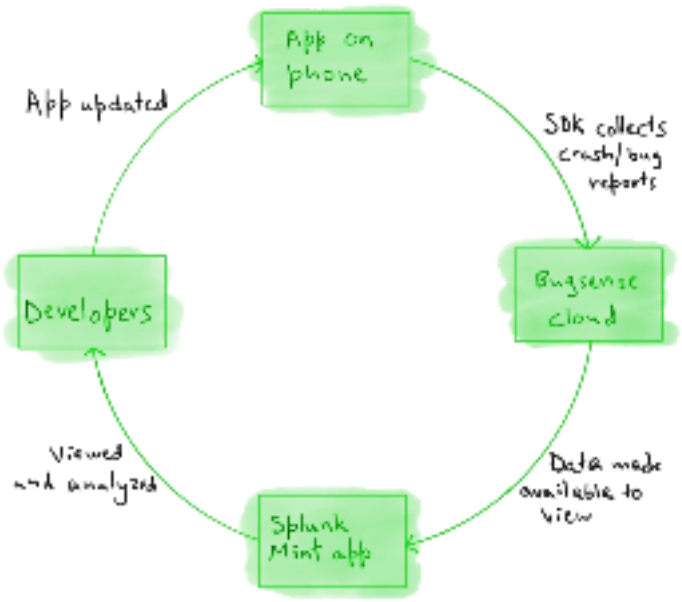


Figure 5. App Maintenance Cycle.

The Database

We will be creating a mySQL database to store information about the events and the users. The database will be set up with nine tables: four for entities and five for relations. The four types of entities we are working with are users, events, keywords, and categories.

Users are the people who use our app. Users both create and search for events. They will have a username and password, as well as a display name that will be displayed online to other users. They also have saved preferences of location and radius, which will determine which events each user is shown. A user also has a profile picture and a link to their Facebook account. For the users that host events the database will also store a value that represents their average host rating.

Events are the entities about which information must be stored. We will be storing each event's name, time, date, location, description, associated photo, maximum attendees, host, and whether it is a private or a public event. We also have a variable that stores what the event is rated because we will continue to store events even after they have ended. This will allow us to give users a list of their past events. Also, if we want to expand this project it will allow us to get heuristics about the users' preferences.

When a user searches for events, two entities are utilized: categories and keywords. Categories are a list of general classifications that we have created for the user. The database will only store their ID's and names. For more user customization, we also will store keywords. When a user makes an event, they will select categories from a dropdown list. After the category is selected, the user can then type in as many original keywords as they want. As with categories, the database will only store the names and ID's of the keywords.

In addition, we have 5 tables that represent the relations between entities. Users interact with events in 3 ways. They can attend an event, they can have attended an event, or they can host an event. Attending events is a many-to-many relationship, as is having attended events previously. Hosting, on the other hand, is a one-to-many relationship since an event can only have one host, but one user can host many events. As such, events and users can be related using a single attribute of an event.

Another type of relationship is a single event having several keywords and several categories. Through this relationship, people can find an event from keywords and filter events by category. These are both many-to-many relationships.

The last type of relationship is a connection between a user and the categories they, personally, are interested in. This connection will facilitate the automatic suggestion of events tailored to user interests. However, keywords are too numerous to be stored on a user-by-user basis.

The advantages of this design are that it will be efficient and that it will have room to grow. The design is in third normal form, which gives it adequate efficiency. If we find that we are experiencing long load times, we can also separate the events table into “upcoming” and “past.” An entity relationship diagram and a relational model are listed below.

We will be hosting the mySQL database on a Raspberry Pi. We will do this because it will give us a lot of control. It will also allow us to store images in a folder and store links in the database. We will use an Apache Web server as an FTP server to access the images. The Pi allows us to host all of these things in one place and gives us the ability to be flexible in the design of the backend.

Database Diagrams

Entity Relation Diagram for Database

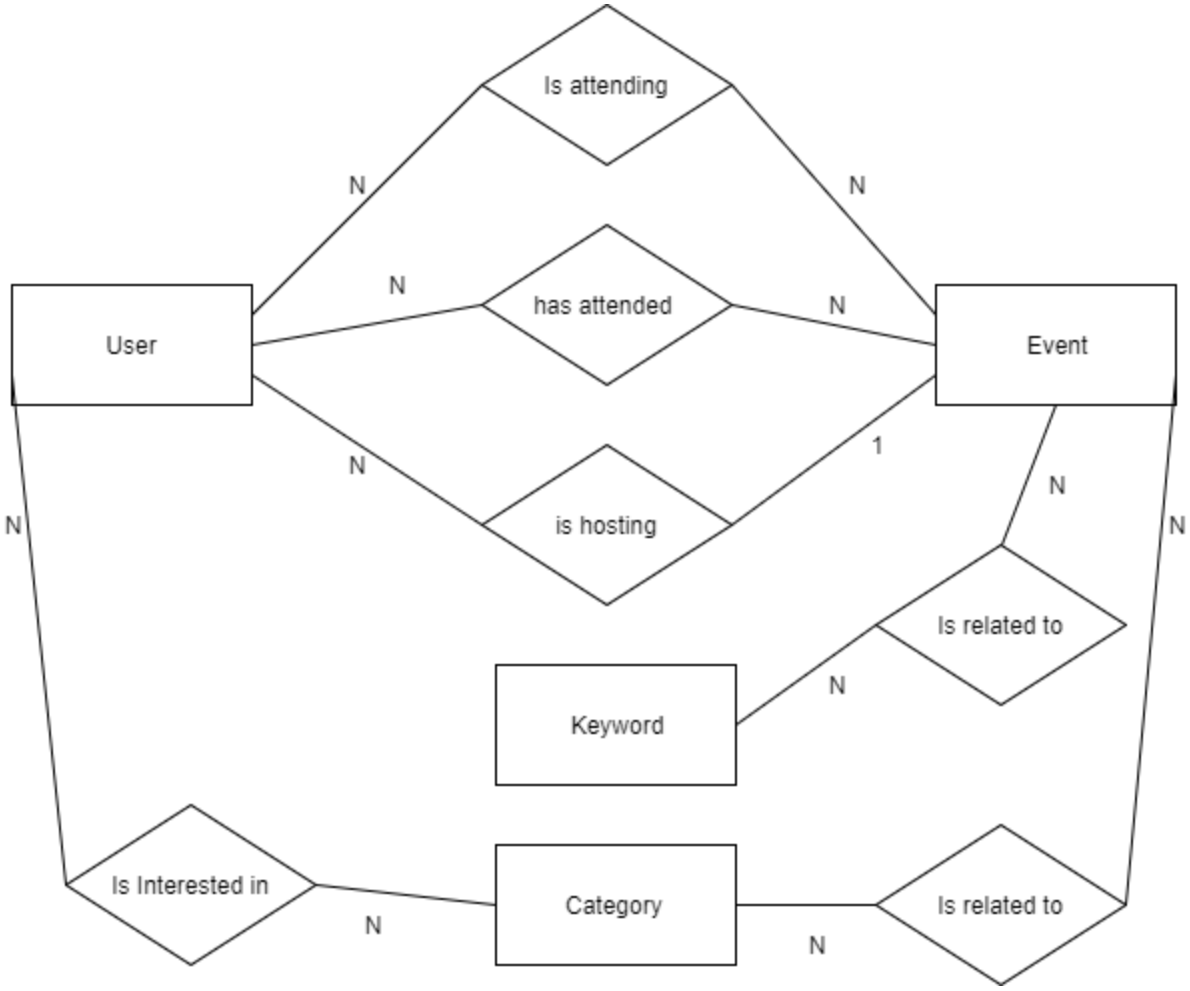


Figure 6. Entity Relation Diagram.

Relational Model of the Database

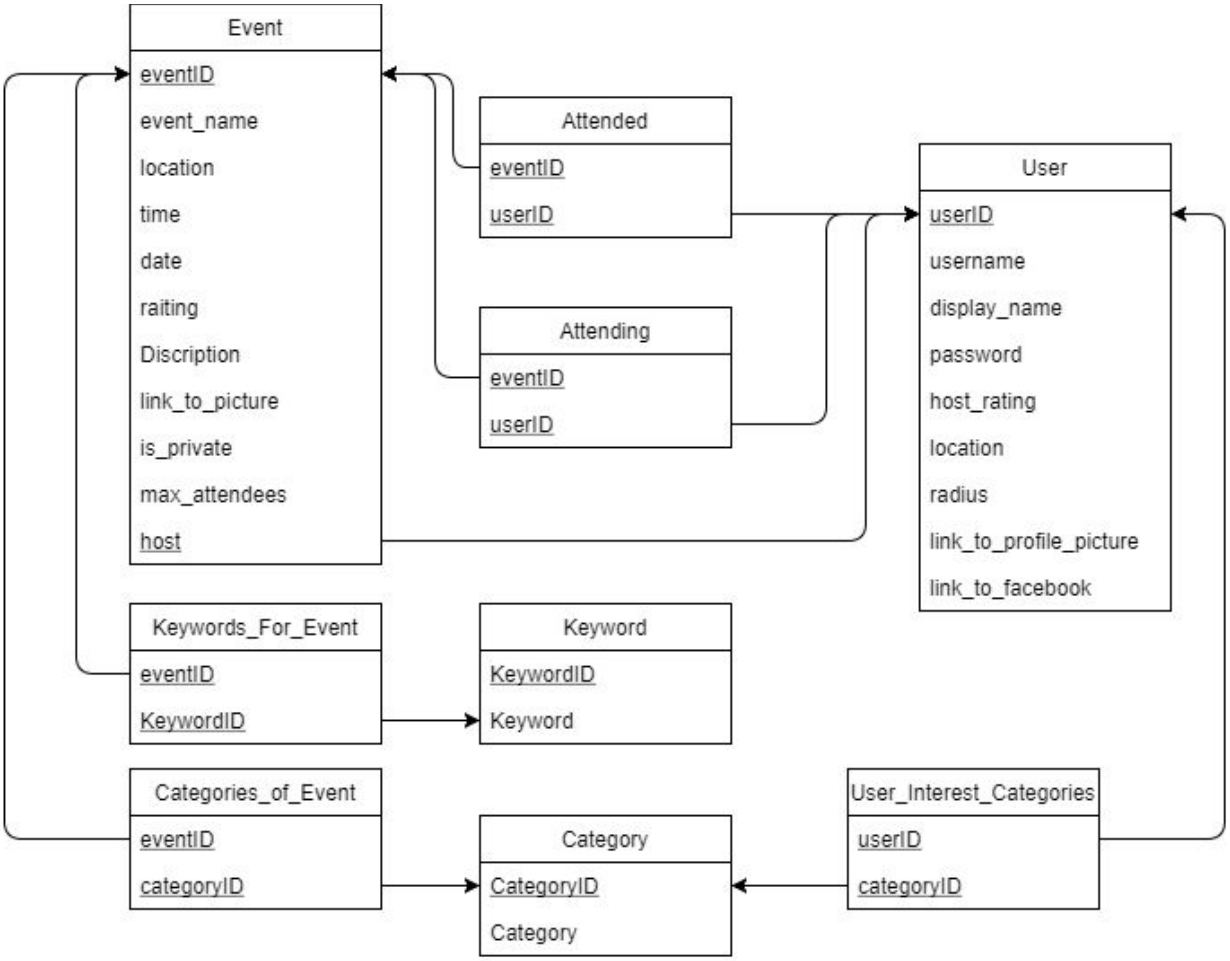
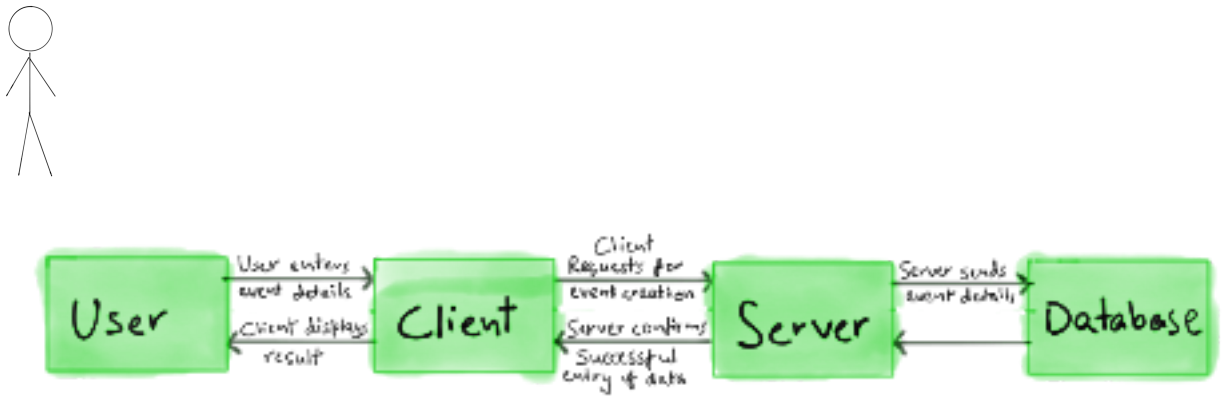


Figure 7. Relational Model.

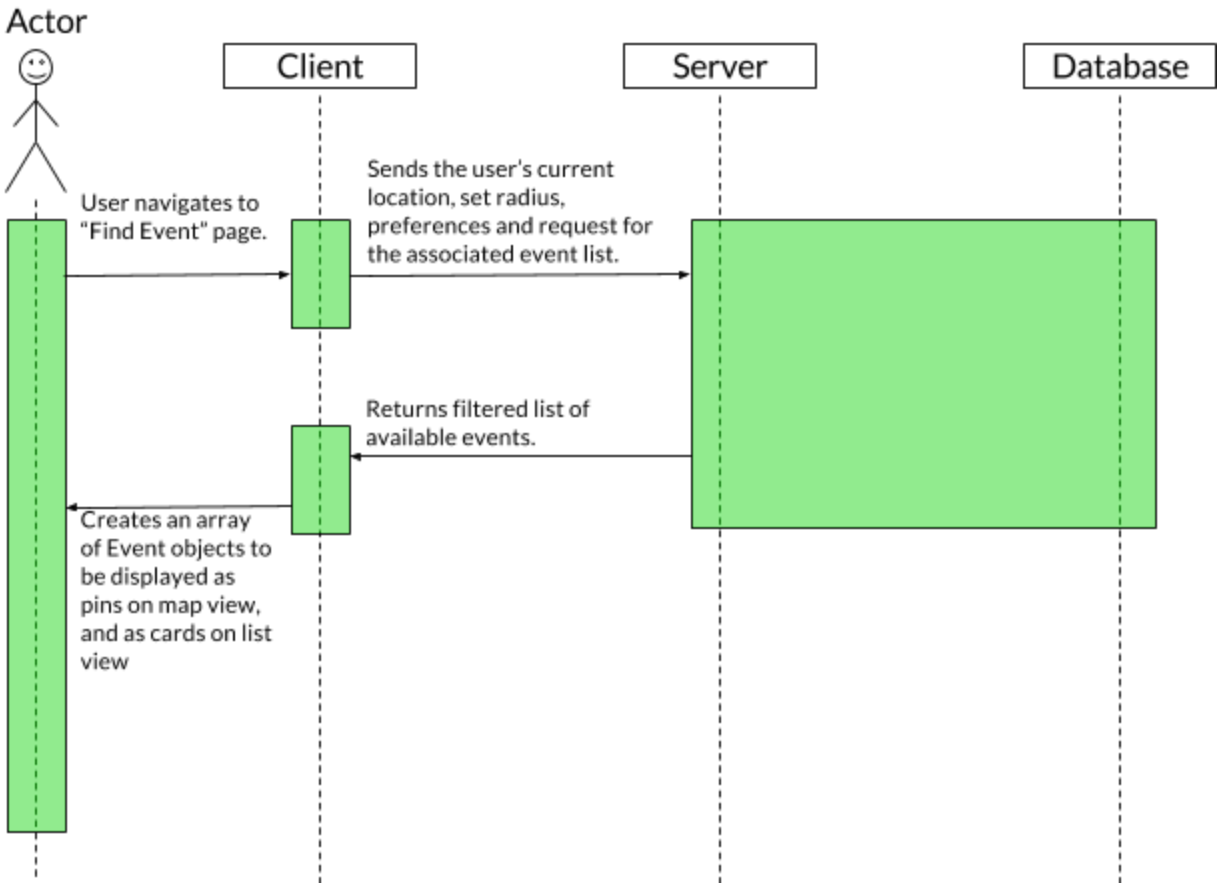
Sequence Diagrams

1. Figure 8. Event Creation



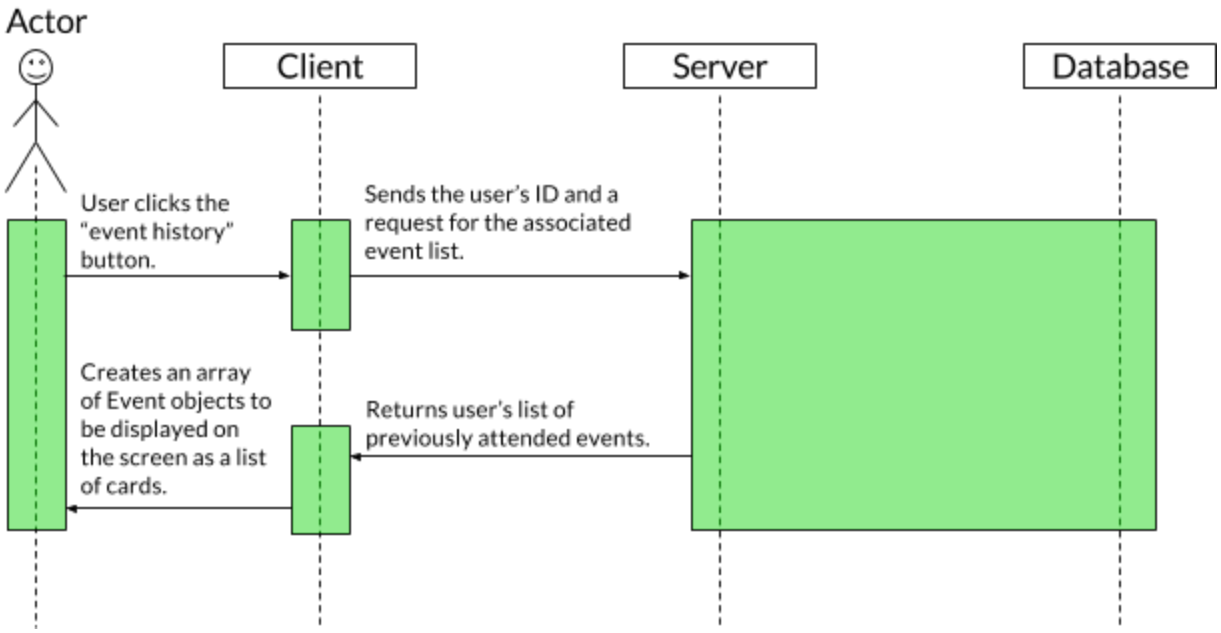
When the user clicks on “Create Event”, they will be directed to another page, where they can enter details of the event (name, location, time, et cetera). Once entered, the information will be sent to the server with a Request for the addition of the new event to the database. In addition to the event, the server will also enter the associated data into the database. The server confirms the storage of the data and Replies to the client, which then displays the result to the user.

2. Figure 9. Event Search



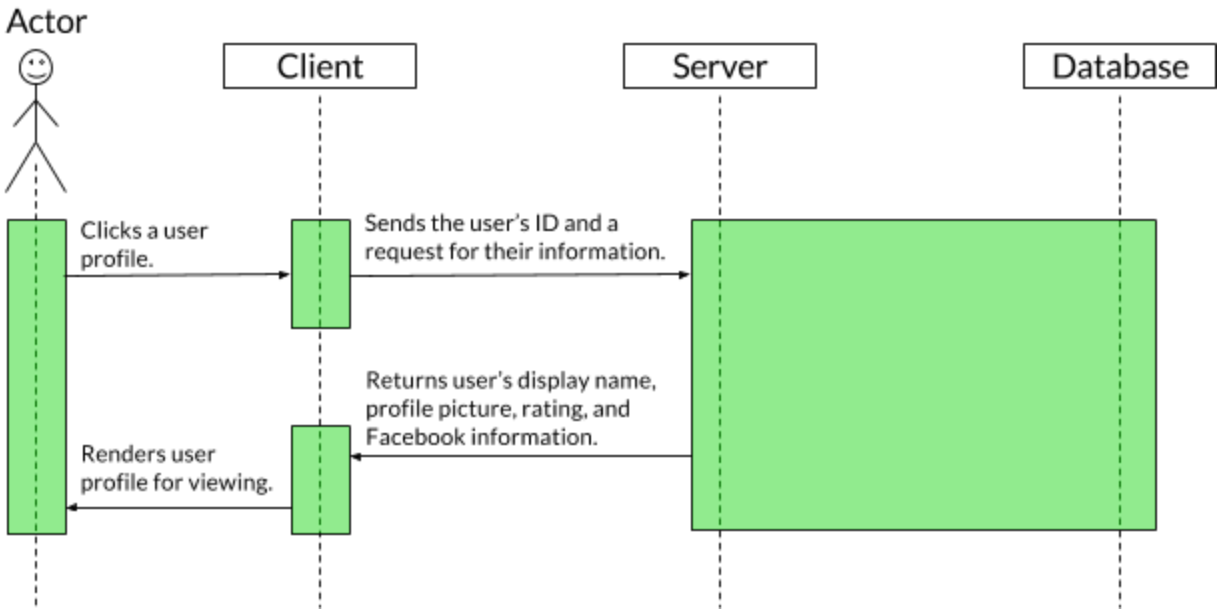
When a user is searching for an event, they will navigate the map view or the list view as their current location updates in real time. This activity will trigger a call from the client to the database. The query contains the current location of the user, the selected radius of user, and preference settings. Then, the server sends back the filtered list of events, which the client will render as cards for list view and as pins for map view.

3. Figure 10. Event History



When a user wishes to see a list of events they attended previously, they can navigate to their event history through the sidebar menu. Clicking "event history" triggers a call from the client to the database. The query contains the user's identification number and returns the corresponding event list. Then, the client renders the events as a list of cards that link to the pages of the events.

4. Figure 11. User Profiles



A user may navigate to a user profile by clicking on the name of an event host, an event guest, or by clicking the button to access their own profile. When this happens, the client sends a request to the server along with the user ID of the profile to load. The server will retrieve the user's display name, profile picture, rating, and Facebook information from the database and send it back to the client. Then, the client will populate the fields of the profile page and render it for viewing.

UI Mockups

Figure 12. Login page

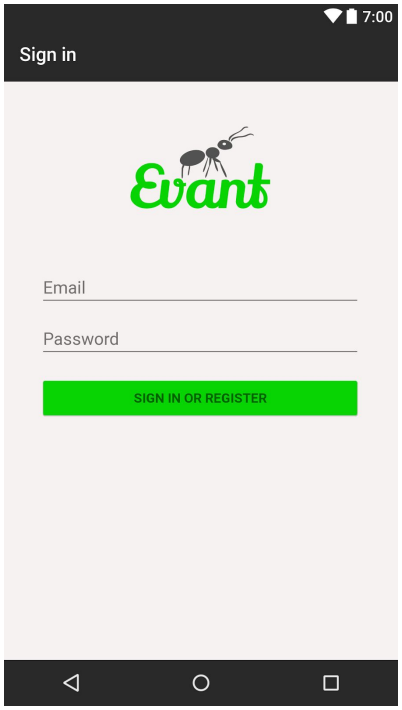


Figure 13. Map/First landing page

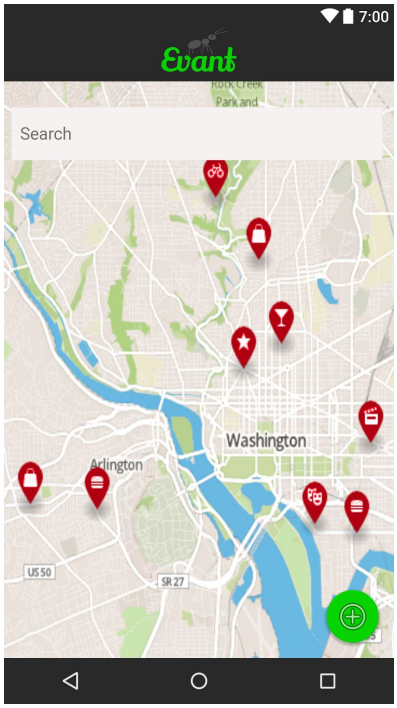


Figure 14. User profile page