Project Design: Tripmates

Team: JuiCy buNS

Section Authors:

- Overview -- Sophia Kwon
- Conceptual Design -- Cynthia Zhou
- Data Model -- Janice Lee
- Schema Design -- Janice Lee
- Security Concerns -- Sophia Kwon
- Wireframes -- Nancy Luong
- Design Commentary -- everyone
- Beyond CRUD -- everyone

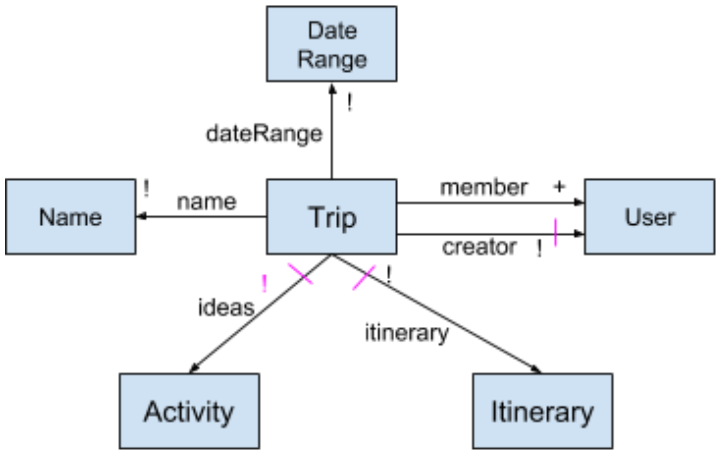
Overview

Tripmates is an online tool to collaboratively plan trips with other people. It provides an organized way for everyone involved in a trip to list activities they're interested in, as well as an interface to collaboratively construct an itinerary for the trip. Tripmates offers a solution to the problem of trying to find an organized way to plan trips with other people. Often, it's difficult to find a way for everyone to record their ideas and plan the logistics of the trip in one place, without things getting extremely cluttered and difficult to read.

Tripmates aims to improve upon the existing solutions to this problem. Some people have attempted to use Google Docs to plan trips with others, as it is a popular collaboration tool. They would list links, as bullet points or something similar, in the doc. However, it's difficult to format trip ideas neatly in this environment, as users would have to scroll through multiple pages of potentially irregularly formatted ideas. Google Docs also lacks a nice interface for making an itinerary. Another existing solution is Google Trips, a tool for planning trips specifically. The main drawback to Google Trips is that it does not support collaborative planning--only one user can plan/edit a trip. It also has the restriction that it is only available as a downloadable app. There is also another tool, called Roadtrippers, which allows people to plan road trips together. However, it's really only intended for road trips, and it can only be used for trips in certain supported locations. Tripmates aims to provide an organized interface for collaboration that is flexible enough to be used for any trip.

Conceptual Design

Name: **Trip**
Purpose: Collaboratively plan for a trip with multiple people, so each person can share their ideas with the group and create schedules together.
Structure:



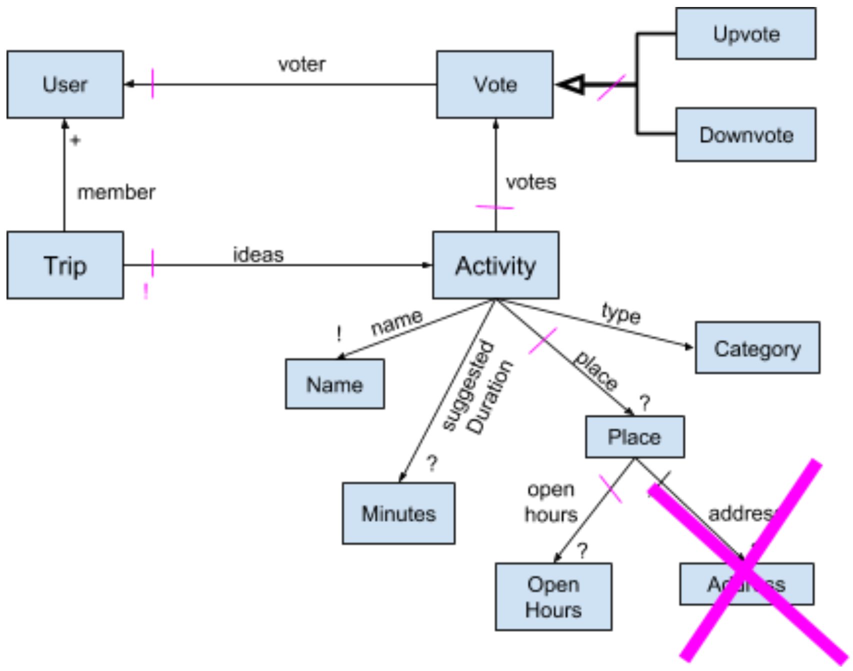
Behavior:

```
createTrip(u: User, n: Name, dr: DateRange): Trip
        effects         result.name := n
                        result.dateRange := dr
                        result.creator := u
                        result.members := Set(u)
join(t: Trip, u: User):
        requires        not t.members.contains(u)
        effects         t.members.add(invitee)
createActivity(t: Trip, … ): Activity
        [look at Activity concept]
createItinerary(t:Trip, …): Itinerary
        [look at Itinerary concept]
search(n: Name): Trip
        effects         result where result.name == n
editName(t: Trip, n: Name, u: User):
        requires        t.members.contains(u)
        effects         t.name = n
editDateRange(t: Trip, dr: DateRange, u: User):
        requires        t.members.contains(u)
        effects         t.dateRange = dr
```

delete(t: Trip, u: User):
requires t.members.contains(u)
effects delete t

Operational Principle:
User u wants to plan a trip with other people.
u creates a Trip t with name n and daterange dr.
u invites Users u' and u" to join t.
u, u', and u" can createActivity and createItinerary for t.
~~Sarah wants to create a trip with other people for Jan 10 - Jan 23.~~
~~She invites other people to "join" the trip (by sending them the trip's unique code) so everyone can plan the trip together. They can add their own ideas for activities they can do on the trip, and collaboratively create schedules with those Activities.~~

Name: **Activity**
Purpose: Share an idea of something that can be done on a trip with other members of the trip, and gauge how interested everyone is for that idea.
Structure:



Behavior:
createActivity(t: Trip, u: User, n: Name, m: Minutes, addr: Address, oh: OpenHours, c: Set(Category)): Activity
requires t.members.contains(u)
effects result.name := n
result.suggestedDuration := m
p := new Place; p.openHours := oh; p.address := addr
result.place := p
result.types := c

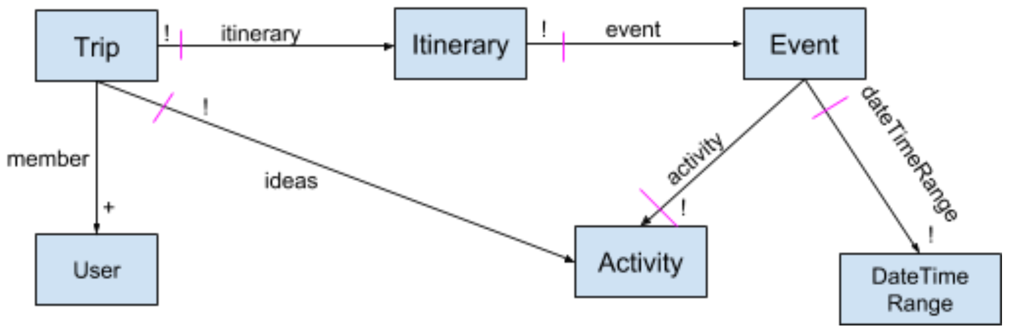
```
                    t.ideas += result
upvote(u: User, a: Activity):
        requires    a.trip.members.contains(u)
        effects     if not a.votes.upvotes.contains(upvote where upvote.voter == u):
                            a.votes.upvotes.add(new Upvote(voter=u))
                    if a.votes.downvotes.contains(downvote where downvote.voter == u):
                            a.votes.downvotes.remove(downvote)
downvote(u: User, a: Activity):
        requires    a.trip.members.contains(u)
        effects     if not a.votes.downvotes.contains(downvote where downvote.voter == u):
                            a.votes.downvotes.add(new Downvote(voter=u))
                    if a.votes.upvotes.contains(upvote where upvote.voter == u):
                            a.votes.upvotes.remove(upvote)
getVotes(a: Activity):  Int
        effects     result := count(activity.votes.upvotes) - count(activity.votes.downvotes)
filter(c: Category): Set(Activity)
        effects     return all activities in this trip with this category
editName(a: Activity, n: Name, u: User):
        requires    a.trip.members.contains(u)
        effects     a.name = n
editDuration(a: Activity, m: Minutes, u: User):
        requires    a.trip.members.contains(u)
        effects     a.suggestedDuration = m
editOpenHours(a: Activity, oh: OpenHours, u: User):
        requires    a.trip.members.contains(u)
        effects     a.place.openHours = oh
editCategory(a: Activity, c: Category, u: User):
        requires    a.trip.members.contains(u)
        effects     a.category = c
delete(a: Activity, u: User):
        requires    a.trip.members.contains(u)
        effects     delete a
```

Operational Principle:

User u creates Activity a for Trip t.
Any User in t.members can filter through t.activities where activity.category = c.
Any User in t.members can upvote, downvote, editName, editDuration, editPlace, editCategory, and delete a.
~~A user named Bob is planning a Trip with other people. He has an idea of an activity he wants to do.~~
~~Bob creates an Activity with a name, suggested duration (optional), address (optional), open hours for that address (optional), and a set of categories (optional).~~
~~Bob adds the Activity to the Trip.~~

~~People who are planning the Trip with Bob can now see the newly created Activity and upvote or downvote the Activity, based on whether they like the idea or not.~~

Name: **Event**
Purpose: Decide to do an activity at a certain time on a trip.
Structure:

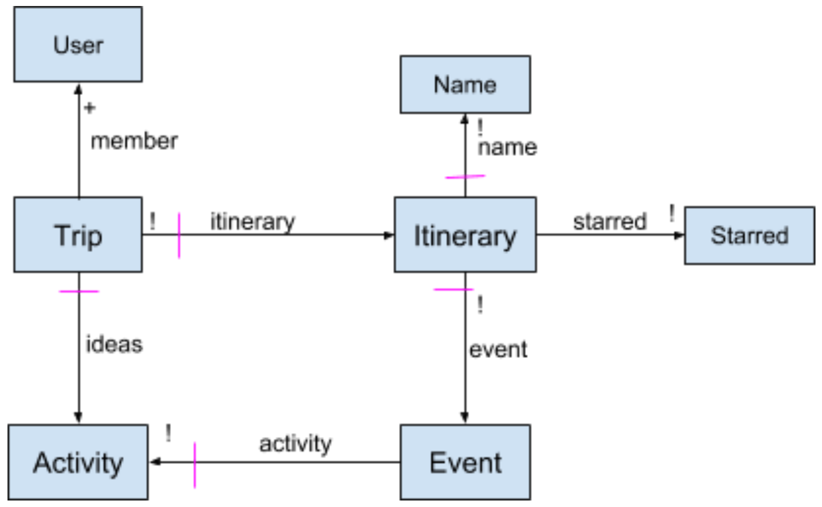


Behavior:

```
createEvent(i: Itinerary, a: Activity, dtr: DateTimeRange, u: User): Event
        requires    i.trip.ideas.contains(a)
                    dtr falls within i.trip.dateRange
                    i.trip.members.contains(u)
                    dtr falls within a.place.openHours
        effects     result.dateTimeRange := dtr
                    result.activity := a
                    i.events += result
editDateTimeRange(e: Event, dtr: DateTimeRange, u: User):
        requires    dtr falls within i.trip.dateRange
                    i.trip.members.contains(u)
        effects     e.dateTimeRange := dtr
delete(e: Event, u: User):
        requires    a.trip.members.contains(u)
        effects     delete e
```

Operational Principle:
User u creates an event e for Itinerary i with Activity a and DateTimeRange dtr (but the event has to fall within a.openHours).
Any User in t.members can editDateTimeRange of e to be a new DateTimeRange dtr2.
Any User in t.members can delete e.
~~Bob wants to add an Activity to an Itinerary for a Trip.~~
~~Bob adds the Activity to the Itinerary, and sets the Activity to happen from a start DateTime to an end DateTime, resulting in an Event on the Itinerary.~~
~~Other members of the trip can change the dateTimeRange for the Event.~~

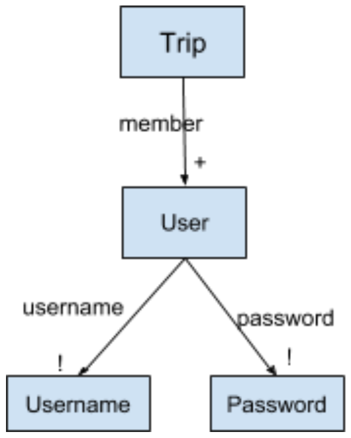
Name: **Itinerary**

Purpose: Collaboratively make drafts of plans for what activities members of a trip will do and when they'll do them.
Structure:


Behavior:
createItinerary(t: Trip, n: Name, u: User): Itinerary
        requires   t.members.contains(u)
        effects   result.name := n
                        t.itineraries += result
starItinerary(i: Itinerary):
        effects i.starred := true
unstarItinerary(i: Itinerary):
        effects i.starred := false
createEvent(i: Itinerary, …): Event
        [look at **Event** concept]
editName(I: itinerary, n: Name, u: User):
        requires   a.trip.members.contains(u)
        effects   i.name = n
delete(i: Itinerary, u: User):
        requires   a.trip.members.contains(u)
        effects   delete i
Operational Principle:
User u creates an Itinerary i for Trip t.
Any member in t.members can star i (so now all t.members can see that i is starred) or unstar i.
Any member in t.members can edit the name of i, delete i, and create an event for i.
~~Sarah wants to create a schedule for a Trip using some of the Activities that everyone has listed in the Trip.~~
~~She creates an Itinerary for the Trip and named it "Draft 1". She adds Events to the "Draft 1". Other members of the Trip can add Events / edit Events in "Draft 1".~~
~~The group is satisfied with "Draft 1", so they star the itinerary to signify that it's a preferred itinerary for now.~~

~~Bob creates another Itinerary named “Bob and Fred Get Away” for the Trip because he and Fred will separate from the group on Jan 15 and they want to schedule what they’ll do for that day.~~

~~Name: **User**~~
~~Purpose: Give identity to people using Tripmates so users can choose who to plan a trip with.~~
~~Structure:~~

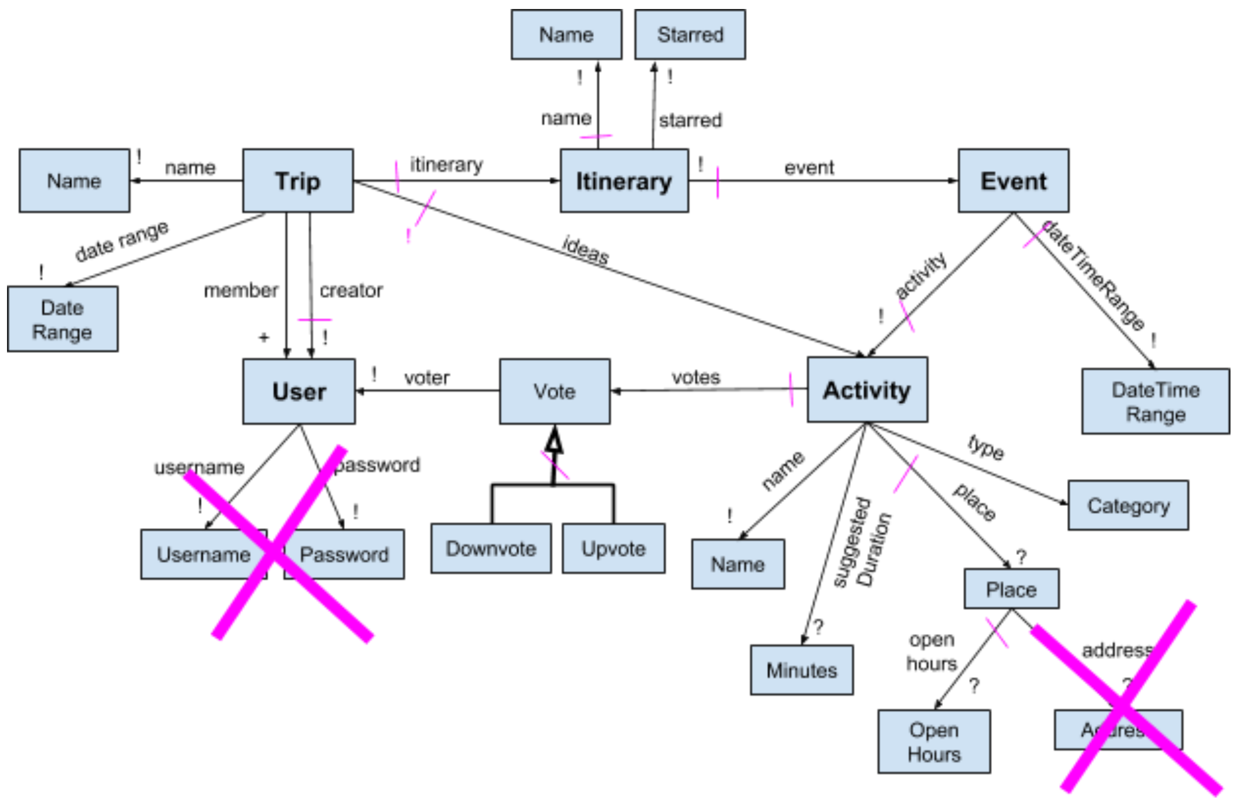


~~Behavior:~~

```
~~createUser(u: Username, p: Password): User~~
    ~~requires~~      ~~username is unique~~
    ~~effects~~       ~~result.username := u~~
                  ~~result.password := p~~
~~invite(t: Trip, inviter: User, invitee: User):~~
    ~~[look at **Trip** concept]~~
```

~~Operational Principle:~~
~~Bob wants to plan a trip with Sarah, Fred, and Amy.~~
~~Bob creates an account with a username he selected, and his own password.~~
~~He invites Sarah, Fred, and Amy to also create accounts if they haven’t already, and invites them to plan the trip with him.~~

Data Model

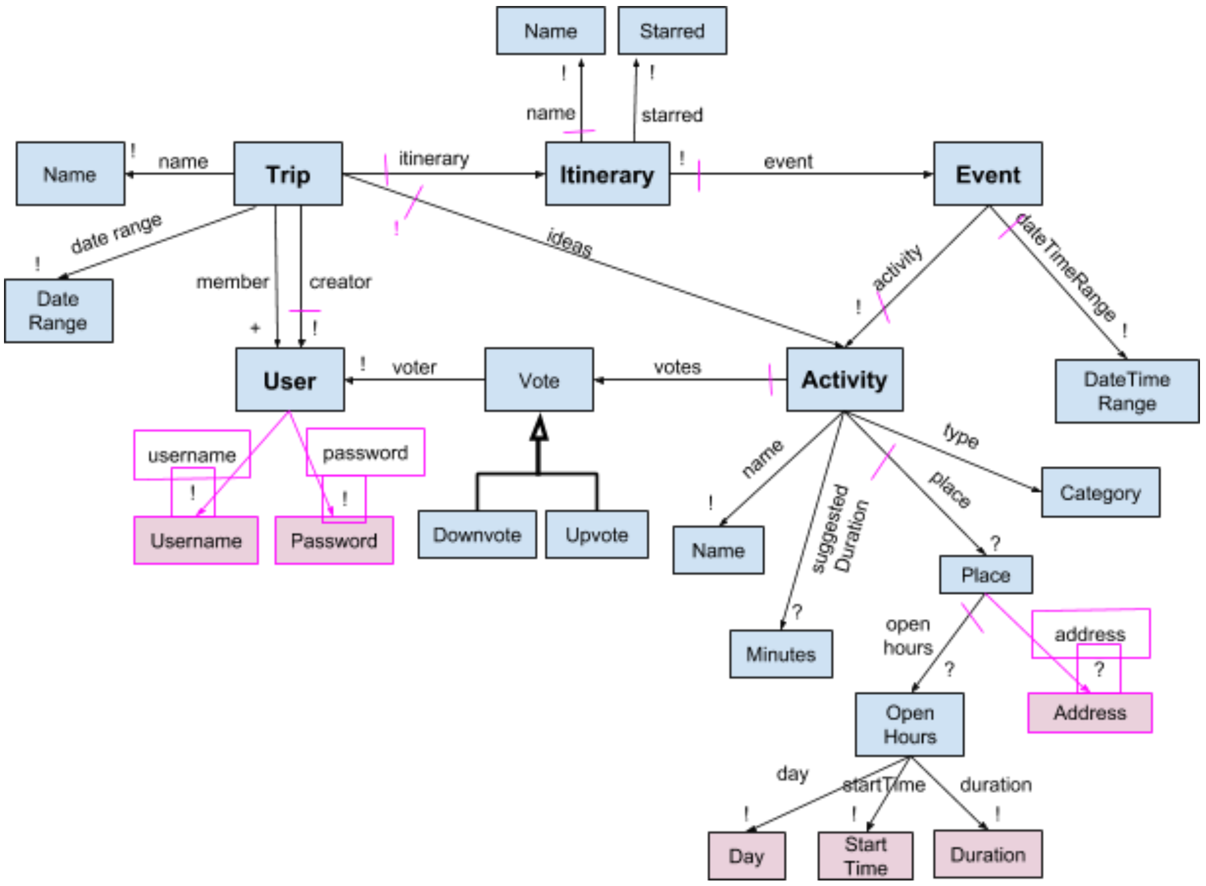


<u>Textual Constraints:</u>
Itinerary i can only contain an Event e where e.activity belongs in i.trip.activities

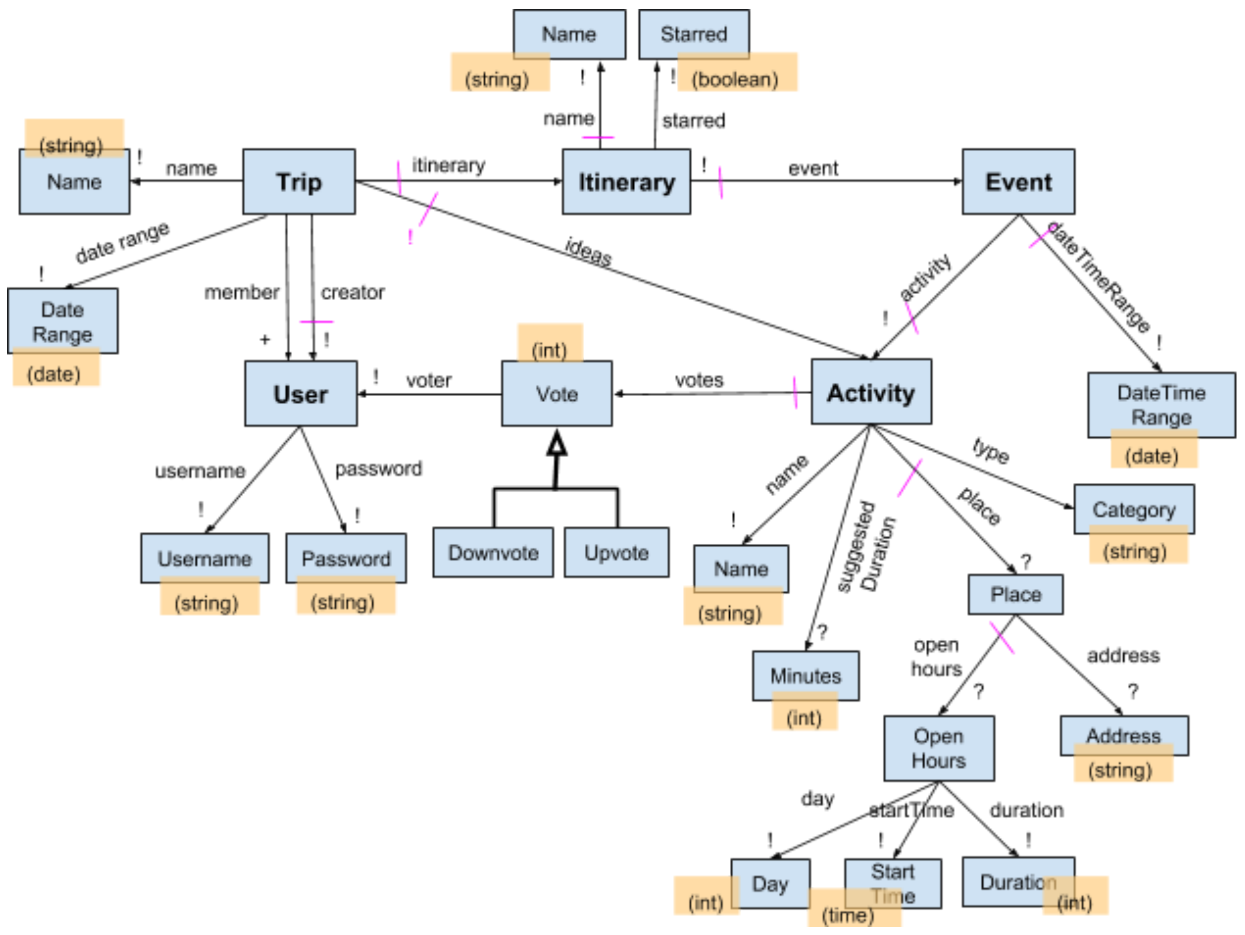
Schema Design

How to obtain schema:

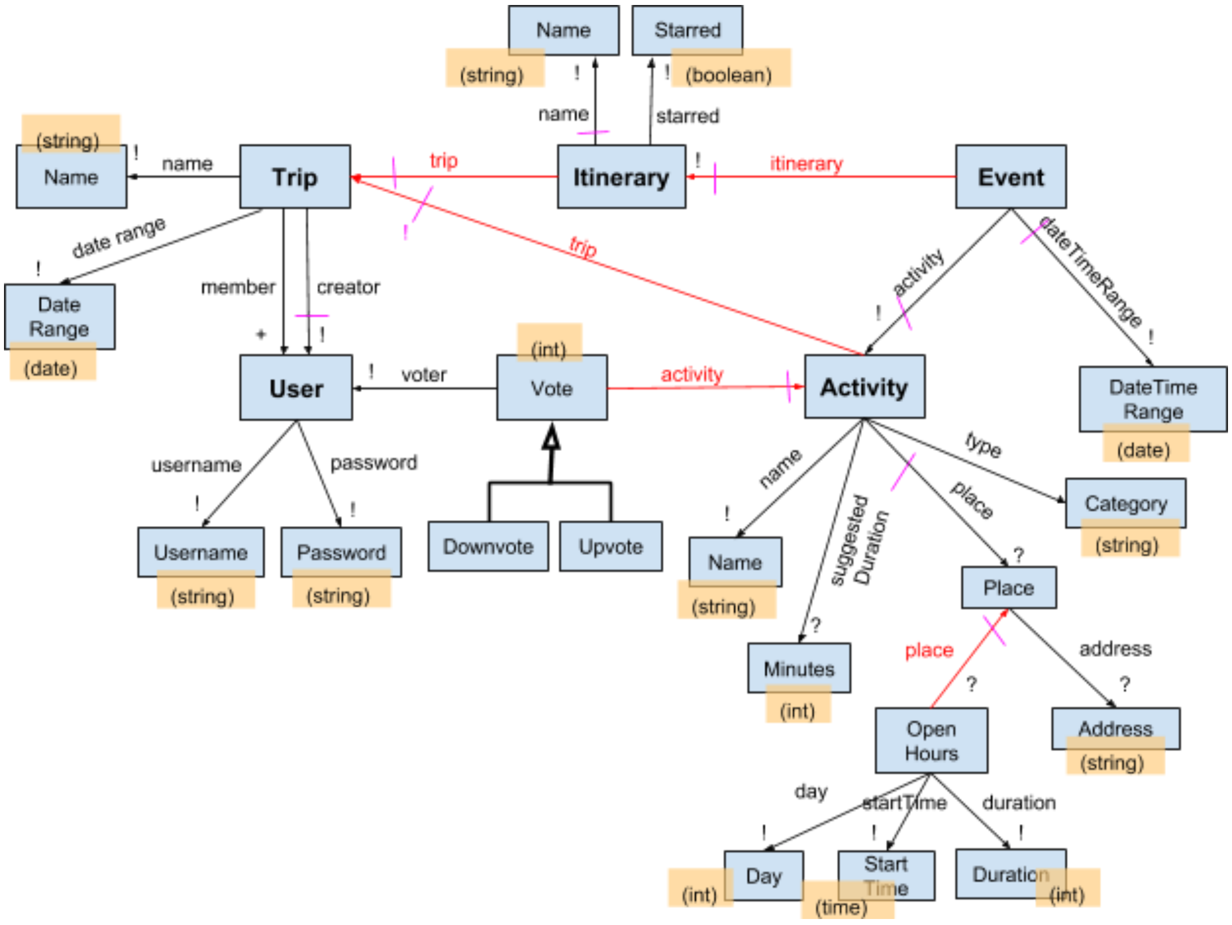
1. Add attributes



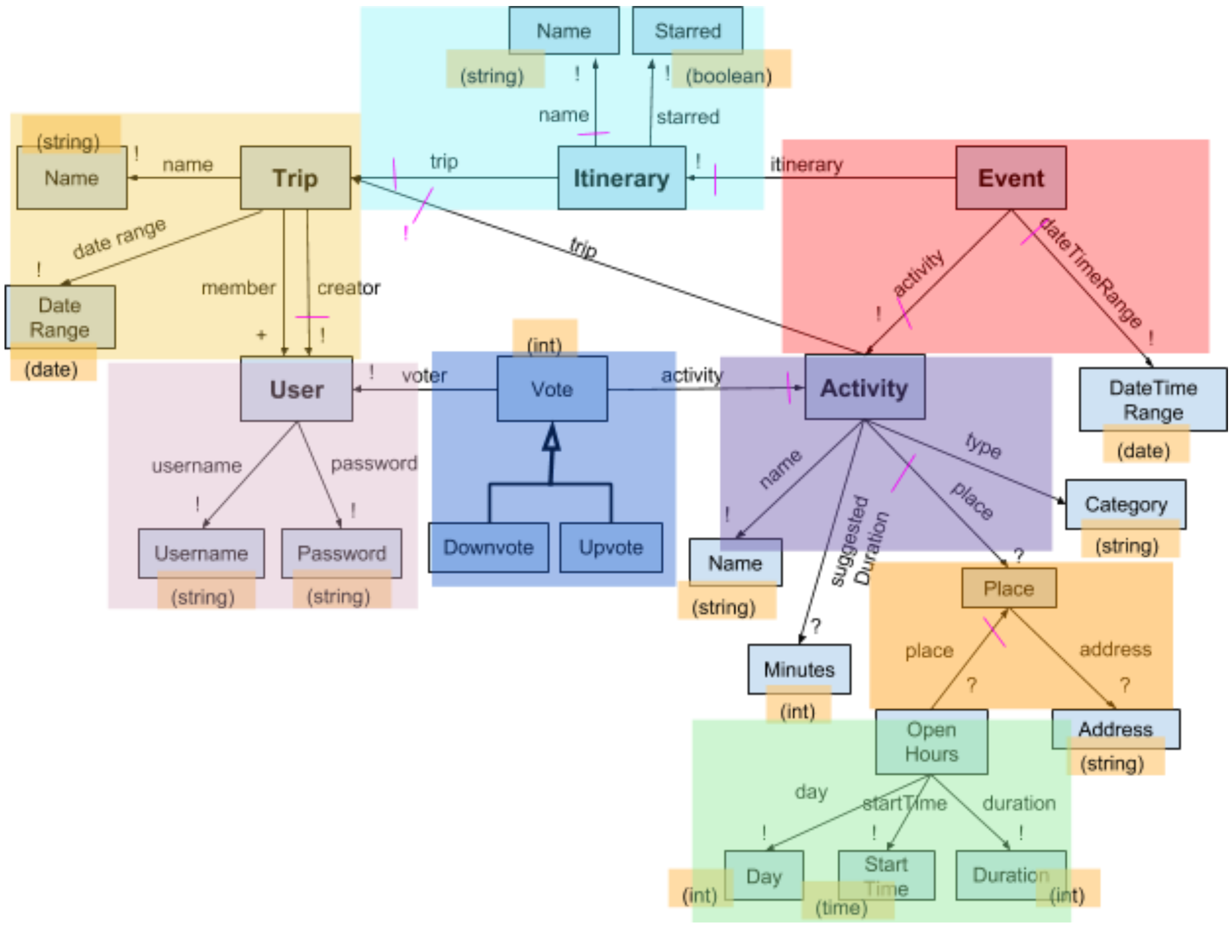
2. Pick primitive types



3. Reverse relations



4. Group into tables



5. Write out tables (below!)

Schema:

```
CREATE TABLE user (
	id INT PRIMARY KEY AUTOINCREMENT,
	username VARCHAR(20) NOT NULL UNIQUE,
	password VARCHAR(60) NOT NULL,
)

CREATE TABLE trip (
	id INT PRIMARY KEY AUTOINCREMENT,
	name VARCHAR(~~20~~40) NOT NULL,
	FOREIGN KEY(creatorId) REFERENCES user(id) NOT NULL,
	startdate ~~DATE~~ VARCHAR(10) NOT NULL,
	enddate ~~DATE~~ VARCHAR(10) NOT NULL
)

CREATE TABLE tripmembership (
	FOREIGN KEY(userId) REFERENCES user(id) NOT NULL,
	FOREIGN KEY(tripId) REFERENCES trip(id) NOT NULL,
	PRIMARY KEY (userId, tripId) UNIQUE
)

CREATE TABLE activity (
	id INT PRIMARY KEY AUTOINCREMENT,
	name VARCHAR(~~20~~40) NOT NULL,
	suggestedDuration INT,
	FOREIGN KEY (placeID) REFERENCES place(id),
	FOREIGN KEY (tripId) REFERENCES trip(id) NOT NULL,
	category VARCHAR(20)
)

CREATE TABLE place (
	id INT PRIMARY KEY AUTOINCREMENT,
	name VARCHAR(40) NOT NULL,
	address VARCHAR(100)
)

CREATE TABLE openHours (
	id INT PRIMARY KEY AUTOINCREMENT
	FOREIGN KEY(placeId) REFERENCES place(id),
	day INT NOT NULL,
	startTime ~~TIME~~ VARCHAR(5) NOT NULL,
	duration INT NOT NULL
)

CREATE TABLE itinerary (
      id INT PRIMARY KEY AUTOINCREMENT,
      name VARCHAR(~~20~~40) NOT NULL,
      FOREIGN KEY(tripId) REFERENCES trip(id) NOT NULL,
      starred BOOLEAN NOT NULL
)

CREATE TABLE event (
      id INT PRIMARY KEY AUTOINCREMENT,
      FOREIGN KEY(activityId) REFERENCES activity(id) NOT NULL,
      startDateTime ~~DATETIME~~ VARCHAR(16) NOT NULL,
      endDateTime ~~DATETIME~~ VARCHAR(16) NOT NULL,
      FOREIGN KEY (itineraryID) REFERENCES itinerary(id) NOT NULL
)

CREATE TABLE activityVotes (
      FOREIGN KEY(activityId) REFERENCES activity(id) NOT NULL,
      FOREIGN KEY(userId) REFERENCES user(id) NOT NULL,
      value INT NOT NULL,
      PRIMARY KEY (activityId, userId) UNIQUE
)
```

Security Concerns

We will implement Tripmates to protect against some common security vulnerabilities, as discussed in this class. We will escape query values in order to protect against SQL injection. We will sanitize all inputs in order to prevent cross-site scripting attacks. The users database table will store hashed passwords, instead of plaintext passwords, in order to prevent attackers from being able to log into accounts using passwords stolen from the table. In order to prevent integrity violations, everything in the database that references a user will reference the user id, which is unique for each created user; this way, no user will be able to take over another account using a clever username or something similar. We will protect against cross-site request forgery attacks by requiring requests to be accompanied by a token, which the user would have received upon logging in to Tripmates.

One additional security concern that pertains to Tripmates is making sure that malicious users are unable to join trips they have not been invited to. The process of adding users will go as follows. Once a user creates a trip, he will receive a code to share with other users whom he wants to add to this trip. Each user who wishes to join will input this code into Tripmates, which sends a “join request” to the creator of the trip. The creator will look at the username (and avatar/profile picture) of the user asking to join the trip, and will accept or reject the request. That way, the creator can prevent people they didn’t invite from joining the trip.

Wireframes

Found at:
https://www.figma.com/file/RaDI7m6tmrBuMXVjwgjvPkq6/Tripmates?node-id=0%3A1

Design Commentary

How to store opening hours: We considered different ways to store opening hours for places. For instance, we considered storing the entire set of opening hours for all days of the week as a string, but that would be error-prone. We also considered storing each set of opening hours as: day of the week, start time, and end time; but that wouldn't work for places that open past midnight, since for most places, if a store is open till 2am, it's still considered the previous day. We also considered storing as: start day, start time, end day, end time; but we realized we're adding an "end day" column for only a few exceptions, and we would need to check if end day + end time is after start day + start time. We ended up associating places with a set of: day, start time, duration. This way, we can calculate the end time using the duration, it solves the problem of having hours going past midnight and onto another day, and we remove the need to check for valid start times and end times. We store the open hours in a separate table in our database because we don't know how many open hours we need (for example some places could have multiple open hour windows each day e.g restaurants with lunch and dinner hours, etc).

Whether places should be usable by different trips: We were considering whether places should be usable by more than one trip (for instance, the same place but inputted by different users). We decided that they should be usable, since it makes sense to use the same place. For instance, if multiple users of completely different trips want to go to the Eiffel Tower, this place is the same, and there's no point in storing the Eiffel Tower twice. This decision is reflected in the way places are stored. We make it so that activities reference a place ID, instead of activities all having their own places.

Voting: A feature we decided to add was added in part to address feedback on how sometimes it's hard to allow all collaborators of a trip have their say in chosen activities. We decided to allow users to vote on activities (upvote or downvote). This involved several decisions. For instance, how to store these votes. One possible design was having two tables, an upvote and downvote table, but we decided that one table was sufficient since we could just tally up counts for upvotes / downvotes, represented by the ints 1 or -1. In addition, we were deciding whether to show the total vote account, or the count each for upvotes and downvotes. We decided on the latter, because we thought it's important to know how many people like and don't like an event (since it can allow people to possibly split up and create multiple itineraries, seeing who likes what activity).

Which fields are required/optional in data model: We also discussed which fields should be required or optional in the data model. For instance, we made it so that a place is optional for an activity. Perhaps users want to do something general, like "stroll in Manhattan", which may not consist of one place. We also decided to make open hours optional, since some places do not have set open hour. Another design decision was making the suggested duration of an activity optional, since users may not choose when or how long they want to be doing an activity until they start creating an itinerary.

Calendar View: We chose to allow for the user to toggle between day, week, and month views for their itineraries. This gives the user more flexibility (as opposed to having only one view) when they are planning their itineraries so that can easily navigate to and edit different parts of the trip as well as get a better sense of how much they have planned for the duration of the trip.

Joining trips: One possible design we considered for inviting other users to join a trip was to generate an invite link for the creator of the trip to share with friends they want to invite to the trip. They can then follow the link which will first verify that the user is logged in (and prompt them to sign up/sign in if they are not) before adding them to the trip. However, this is more complicated to implement and there isn't a large benefit in having an easily shareable link when most users are only likely going to want to invite a few other users to collaborate on the trip.

An alternative solution was to have the trip creator add users by inputting their usernames. This method, however, is more susceptible to human errors such as mistyping a username and adding the wrong person to the trip. Although we will support the function to remove a user from the trip, this is extra work that the user would have to perform.

Our ultimate design for inviting users is to generate a unique 4-digit code for each trip created. The trip creator then shares this code with the users they want to invite. Then each of these users just simply inputs this code to join trips. The flow of this process is short and easy to follow for all parties since it only involves copying and pasting a short code once.

Beyond CRUD

We display the itineraries as a calendar and allow the user to interact with the calendar to create and edit events. We implement various checks on the data, to see whether date/time ranges are valid, etc. We also check events for conflicts, with either other events in the same itinerary, or with the open hours of the place at which the event takes place. Since Tripmates supports collaborative editing, we also have to make sure there aren't concurrency issues when multiple users try to work with the same data. We will also display a map with events' locations.