PROJECT 3.2 DESIGN DOCUMENT

Angel Batista, Christian Londoño, Lucy Yu

Features for the MVP

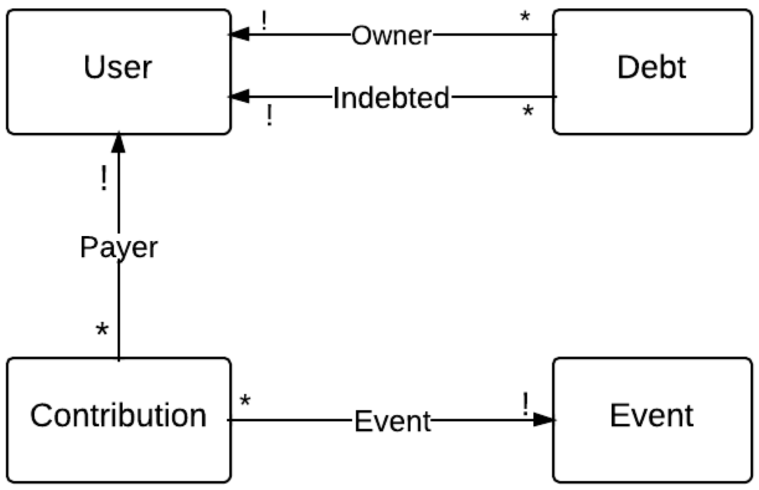
- **Create account**
  Users can create an account and log in. Security concerns are taken into account with checks and validations, such as minimum password length.

- **Create events**
  Users can create an "event" which is defined as many users splitting a cost together. In our MVP, one user creates the event for many people: that is to say, one person lists the total cost and lists the contributions of everyone (by typing their email), as well as how much each person was supposed to pay. (So they can calculate an even split or a weighted split).

- **Keep track of debts**
  When events are saved debts between users are calculated to efficiently split the cost discrepancies among users. Each user can see how much they owe a user and how much other people owe them. With repeated events the debts are automatically updated. (Ie. user 1 overpays at one event so user 2 is indebted, but then user2 overpays at another event and the debt is lessened). When a user pays back their debt in real life they can update their debt on FriendSplitter to see the changed records.

Reduced Data Model for the MVP

*** Reductions include elimination of the Groups model for now, and elimination of the Creator relationship between User and Event ***



Design Challenges

1. In order to create pending events, do we create a "Pending Event" and "Event" object or simply just an "Event" object?

As we reduced the original data model for the MVP, we realized that the notion of a "Pending Event" wasn't expressed in the data model. It was suggested to add another object named "Pending Event," which would turn into an "Event" once every user had accepted a contribution assigned to them. This then introduced the notion of a "Pending Contribution," where a pending event, would consist of pending contributions, which would then turn into an event and contributions, respectively. In an attempt to avoid redundancy, we decided that we would simply implement pending events with a binary value stored in the Event and Contribution objects. As a pro, we avoid creating excessive objects. On the con side, we would have to iterate through all events in order to figure out which were pending and which were confirmed. Ultimately, the functionality remains the same.

2. How exactly do you create an event with multiple users?

Creating a form for a single object with rails is extremely easy, especially with the use of form helpers. However, creating a form to create multiple objects was very difficult. An initial idea was to simply find a way to pass a hash containing attributes for each contribution, which would then be created after the event was created. This could have been accomplished using the rails accepts_nested_attributes_for method, which would allow for a form to create multiple objects if there was a one to many relationship (i.e. one event to many contributions). However, due to rails 4 strong parameters, it wasn't possible to simply pass a user's email address for the hash containing the hash attributes. Therefore, we created a small hack where contributions have an email field. This is used to allow us to pass user emails through the form to the create action of the event controller. From there, we check to see whether or not the emails are valid. This prevents us from using rails validations for the contributions, but it allows us to create the contributions for an event while creating the event itself.

3. Should an event have a creator?

In our initial data model, we had decided to track the creator of the event. However, after talking through the design we realized that there wouldn't be much gain from doing so, because a creator and another participant have the same role as contributors of the total bill. Therefore, we decided to get rid off it.

4. With the current data model, there is no correlation between debts and events. Should we change that?

Since we are viewing the debt between two users to be a simple number that tracks the accumulation of debts across many events, we did not include a relationship between debts and events. One benefit of creating such a relationship would be to allow the user to see a history of debt between himself/herself and another user. However, in an attempt to keep the system simple, we decided to keep the debt independent of the events.

5. When creating debts from an event, how exactly should we go about doing so?

Our main objection was to abstract the small details of splitting a bill for the users. Therefore, given any number of paid amounts and owed amounts, we would have to determine who owes who, and how much they would owe. Currently, our approach is to have those who owe the most pay those who are owed the least. This is a greedy approach in an attempt to minimize the number of debts that we create. For the final release, we will have to figure out a better algorithm that can further minimize the number of debts created. For example, we can start out by matching users who owe and are owed the same amounts, and then continue with the greedy approach used in the MVP.

6. Should we allow users to delete events?

We briefly implemented the ability for any participate in an event to delete events from the database/listing; but we decided to disable it because there may be contributors to the event who do not want the event deleted. For a final implementation, perhaps we could implement a delete option if all users agree to delete.