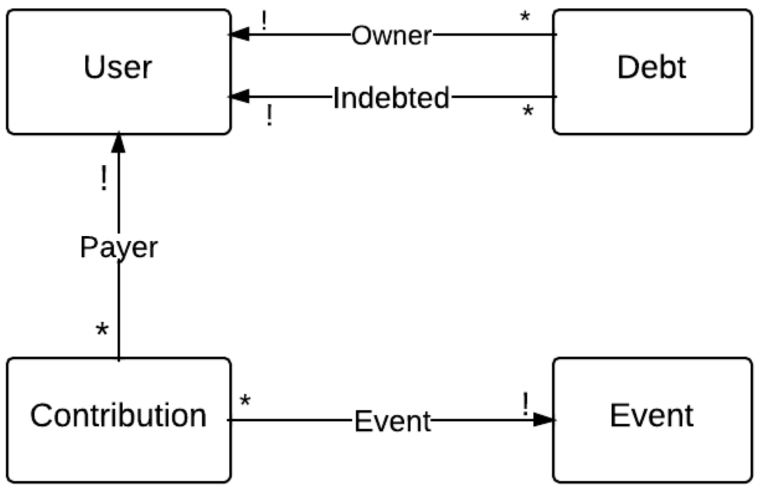
PROJECT 3.2 DESIGN DOCUMENT

Angel Batista, Christian Londoño, Lucy Yu

2.1 Heroku URL:
http://whispering-hamlet-9262.herokuapp.com/

2.2 MVP Identification

Our main goal for the MVP was to create a website with a basic user interface where people could **sign up**, **create an event with another user**, and then have **debts appropriately distributed and updated through accumulated events**. These are all simplified functionalities that we hope will become more robust in the future. But by implementing these simplified features, we believe we are making real progress toward our goal. Splitting costs and keeping track of the debt that constantly changes throughout time is the key idea of our app. Our app strives to make it easy for people to record “events” such as eating out at a restaurant or taking a cab, and not have to think about how much they owe a friend after the latest interaction together. Therefore it was important to have at least rudimentary capability to do this in our MVP. With that in mind, we reduced our data model as follow:



Changes from the initial design documentation include the elimination of the Groups model and the elimination of the Creator relationship between Event and User.

2.3 Subset of Features for MVP

- **Easy and secure account creation and login**
  Users can create an account and login. Since we are using the gem "Devise" to handle this process, we are also defaulting security concerns, such as authorization, to the gem's protocols.

- **Record events that have happened**
  Users can create an "event" which is defined now as two users splitting a cost together. In our MVP, one user creates the event for two people: that is to say, one person lists the total cost and lists the contributions of each person (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 split the cost of 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 (i.e. user 1 overpays at one event so user 2 is indebted, but then user 2 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.

2.4 Issues Postponed

- **Implementing Groups**
  "Groups" are intended to help users create an event quicker. Since this feature is solely intended to improve user experience, we decided to postpone its implementation.

- **Events with more than two people**
  Events with more than two people require a slightly more complicated calculation for distribution of debt, so we will code this for the final product.

- **Non-integer representations of money**
  Currently we represent money as integers, enforced in our front end and also in our back end (the data type in the table is an integer). There are several options to

represent currency (ie. postgres has a currency datatype, as well as a “numeric” and floating point) so we will consider our options.

- **Pending Events/Security**
  Pending events will be crucial to our final product. For the MVP, we wanted to focus on implementing the functionality of the system in terms of making sure that events and debts could be created appropriately. Therefore, we postponed defaulting events into pending events until the final implementation. We acknowledge that this means that in the MVP, any user can create a false event with any other users that they please, thus creating false debts, so we are “trusting” users to be faithful/truthful. However, in the final product, we will eliminate that dependency and security concern with the implementation of pending events.

2.5 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.