***Activities and deliverables**. In this phase, you'll define a subset of the features that comprise a minimum viable product (MVP), and you'll implement it. The relevant deliverables are those in the MVP subsection of the teamwork section of the checklist. As well as reducing the feature set, you may also reduce the data model; if you do so, you should construct a new data model and highlight the key differences. In addition to the MVP, you should also make an initial cut at the design challenges section of the checklist, so that your MVP is constructed in the context of a better understanding of the array of design issues that you face.*

Deborah Chen, Cosmin Gheorghe, Louis DeScioli
October 27, 2013
Project 3.2 Deliverable

URL
http://mysterious-springs-1514.herokuapp.com

Goal

PaymentSplit allows friends or acquaintances to easily settle shared expenses for events. It handles record keeping for groups and leverages Venmo, an online payment solution, to handle the actual monetary transactions. For each event, users no longer have to worry about paying individuals, but instead merely pays to the group. Behind the scenes, the application automatically distributes this money to ensure everyone is paid.

Minimum Viable Product (MVP)
Identification of minimum viable product for first release

The MVP provides basic functionality to achieve the goal stated above. To do this, it is supported with the following features:

- Handle Events: Create events with multiple persons involved. Record purchases and payments for any event in which you are participating.
- No hassle payments: Pay directly to events without worrying about who exactly has to be paid. The application will compute what you owe.

Issues postponed

Integration with Venmo

Currently, once a user makes a payment, a record of the transaction required is saved in a table, to stub out Venmo. In the final round, we will use the Venmo API to actually transfer the money.

UI issues

We wish to clean up the basic UI so that it better matches the wireframes and improves overall usability. Overall flow between pages will also be updated and improved.

Event flexibility

Currently, an event only accepts purchases while it is open. When it is closed, it can take only payments. Furthermore, users cannot join an event after it has been created. Ideally, people can join after the fact and make payments and add purchases in any order. Our app will support this user behavior in the final product.

Testing and security

Basic login security exists for the application. We are using standard procedures for password hashes and salts to safely store user's credentials. In addition, we are also utilizing a built-in Rails function to protect against Cross-Site Forgery Attacks. This can be found in the application controller.

Test cases are still needed for each model. Validations for payments, purchases, and events exists, and new ones will be added to support additional functionality, as needed.

Design challenges

Paying to events versus people

One major design challenge was conveying the idea that in PaymentSplit, users pay to events and not to individual people. This means that when users settle their debts to an event, they simply pay a certain amount to the pot, and PaymentSplit distributes their money behind the scenes to the right people.

To do this, we created the `user_event_balance` model. Each user_event_balance tied to each user and event. For each event, each user has a `user_event_balance`, which has a field for **credit** and **debt**. Debt is how much the user owes to the event, and credit is how much the event owes them.

When a user submits a purchase to an event, the debt and credit field of each `user_event_balance` is recalculated :

1) new_debt = $\frac{\Sigma(\textit{event purchases})}{(\#\ \textit{event members})}$ - Σ(user's purchases) - Σ(user's payments)

2) new_credit = $\frac{(\#\ \textit{event members} - 1)}{(\#\ \textit{event members})}$ * Σ(user's purchases)

When a user makes a payment to an event, the debt and credit field of each user_event_balance is recalculated:

3). **new_debt** = (users share of total event purchases) - (event payments already made by user) - (share covered by own purchases) [Same as 1]

4). if user is owed most amount of money:
new_credit = old credit - payment;
else new_credit = old credit

In PaymentSplit, the person who is owed the most money by the event is paid first. We made this decision, because making a large amount of purchases for an event suggests a high committment, which should be rewarded.

Editing events, payments, and purchases

Another design challenge was in deciding how flexible to be with editing events, payments and purchases. For the MVP, we decided that an organizer could not add more users after the event was created. This allowed us to handle the payment and purchase logic described above without worrying about adjusting for new members. We will support this for the final version however; thus, we can handle the case in which if everyone settles their debts when there are 3 people in

an event, and another person joins, then those 3 people have overpaid and require a refund.

With respect to editing payments and purchases, we decided to be more conservative about allowing users to do so, as our app deals with the transfer of money. We only allow a user to pay exactly what they owe to the event, to ensure that PaymentSplit can redistribute their payment in using the fewest number of transactions. (For example, if a person owes and pays $10, PaymentSplit can give all of the $10 to another user in one transaction, as opposed to the using paying 5 installments of $2, which results in 5 transactions). Thus, a user may not edit their payment. A user may not edit existing purchases, but can add new ones, as long as the event is open (i.e. accepting purchases). In the final product, we will remove the requirement that an event must be open to accept purchases and closed to accept payments, and allow users to pay and add purchases anytime.

Stubbing out Venmo

Finally, a design challenge was to stub out Venmo integration, to be added in for the final product. To do this, we used a *payment_split* model to record every transaction required to direct a user's payment to the correct user. After a person makes a payment, a payment_split record is created with a receiver, amount, and original sender. Though this is invisible to the user, in the final round, we will use this information to make an API call to Venmo to actually transfer the money.