Deborah Chen, Cosmin Gheorghe, Louis DeScioli
November 3, 2013
Project 3.3 Deliverable

URL

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

Goal

Split-Pay 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. [1]

Security concerns

Security requirements

- Only the creator/organizer of an event should be able to edit, close, or delete an event.
- A user may only edit or delete purchases they have made, and not those of other users.
- A user may never delete payments.
- Only members in an event should be able add purchases or make payments to the event.
- Once an event is closed, none of the its members should be able to make new payments or purchases.

Potential risks

Fraudulent purchases

A person will submit fake purchases and will receive payments from the rest of the group for the fraudulent purchase.

Mitigation: First, since an organizer must know the username of who to add to an event, there must be some sort of existing relationship between them, which implies some level of trust. Thus, an organizer would be unlikely to add someone untrustworthy. Second, our app sets a cap of $250 for a purchase amount, so even if someone did submit a false purchase, its damage would be limited. Finally, a user must actively make a payment, so if one did suspect fraudulent activity, he or she could simply decline to pay.

[1] For the purposes of this project, we chose to mock out the Venmo integration. We keep a record of what the money transfers would have been, so that if we wanted, we could call the Venmo API with the information in this table. We also don't deal with properly tying up a venmo account with our app account.

Freeloader problem:

A person could decline to pay their share of their event.

Mitigation: On the Events page, a user can see the history of every payment and purchase made to the event, as well as all the other members in the event. If someone does not pay, this will be clear to other members (their name will not show up under payments), creating a form of social pressure. We did consider including a feature that stated explicitly who did not pay, but we decided that it would create too much pressure for someone to pay quickly and would be too uncomfortable to the end user. Having the information about who hasn't paid present, but not explicit is a reasonable compromise.

Fake events:

A malicious user could create and add users to a fake event, e.g. "Buying a gift for a mutual friend." These users then make their payments to the event. Then, the malicious user absconds with the money without spending it on the supposed gift.

Mitigation: A malicious user is disincentivized from creating fake events, because the users they defraud can followup with them in real life. Furthermore, users can always decline to make payments for events they believe are fake.

Cross-site Request Forgery Attack (CSRF):

When a user is logged into our website and enters into a new, malicious one, the owner of the malicious website could hijack the existing credentials for malicious activity. In the worst case, the attacker could create a separate event, and use the credentials to make you pay for their fraudulent event.

Mitigation: We use standard Rails protection against CSRF. In the Application controller, we include the `protect_from_forgery` helper so a security token is sent with each request; thus, the request from the attacker would be rejected.

SQL injection:

The malicious attacker could enter SQL into the forms or the URL, which would be executed by the application. In the worst case, they could delete purchases to decrease their debt, or even drop entire tables.

Mitigation: We use Rails ActiveRecord for all queries, and never generate SQL queries on our own. When we do need to parse a parameter from the URL, which only happens once with the `event_id`, we sanitize it by converting it to an integer.

Threat model

- For small events, assume that users know each other well in real life, so there is little chance of fraud.
- For larger events, assume the users may not know each other very well, but at least have some mutual connection. Assume the organizer (creator) of the event is then responsible for adding people he/she trusts.
    - Assume that if a friend of the organizer wants someone the organizer doesn't know to be added, then the friend will vouch for the new person offline, before the organizer can add it.
- Assume the process of getting everyone signed up for accounts occurs outside the application using trusted means (such as an email list). This reduces the chance of fraud succeeding.

Design challenges

Paying to events versus people

One major design challenge was conveying the idea that in Split-Pay, 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 Split-Pay distributes their money behind the scenes to the right people. We considered the alternative where a person in an event would choose who they wanted to pay, but we decided it would be simpler and more fair if the event handled the distribution of money.

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 or payment to an event, the debt and credit field of each `user_event_balance` is recalculated :

1) new_debt =

$$\frac{\Sigma(\textit{event purchases})}{(\#\ \textit{event members})} - \Sigma\,(\text{user's purchases}) - \Sigma\,(\text{user's payments})$$

2) new_credit =

$$\Sigma\,(\text{user's purchases}) + \Sigma\,(\text{user's payments}) - \Sigma\,(\text{payments to user}) - \frac{\Sigma(\textit{event purchases})}{(\#\ \textit{event members})}$$

To distribute a payment, we cycle through all users who are owed money, and give them money from the payment, until the money has been fully distributed. The person who is owed the most money is paid first.

We made the decision to calculate the new debt and credit for each user based on all their previous history, instead of dynamically subtracting/updating based on each new entry, as we thought it would be cleaner to have one definitive formula. Though these calculations may take more time, as it requires more lookups, this tradeoff is acceptable, because we assume the number of purchases/payments for an event to be small.

Editing events, payments, and purchases

For the final design, we decided to allow users to edit their purchases but not their payments. Purchases are editable, because one may need to change a description or adjust the amount of the item. The trade-off of making it editable, is that a user could go back and increase the purchase amount to get more money back from the event; however, we decided this risk is acceptable because the close group of friends has little incentive to defraud the other members (see fraudulent purchases under Security concerns).

Payments, however, are non-editable, because they represent actual money transfers. In real life, if one pays for something and money is exchanged, it is difficult to “undo” the payment and take your money back. Furthermore, it would make users uncomfortable to see that they received a deposit from our application, but it was deducted later.
Because of this, we decided to only allow a user to pay at most the amount he/she owes to an event. This way, users would not need to edit/delete their payments as they can’t accidentally overpay.

In addition, users can not be added or removed after an event was created. We believe that users entering and leaving an event after payments or purchases have made it confusing to other users, as the amounts they owe or are owed would fluctuate unintuitively. (For example, if there were 3 people in an event, and everyone had settled their debt, a 4th person joining and adding a purchase would suddenly cause the other 3 members to go into debt again). Though having this option would have given more power and flexibility to the organizers, we decided that it was more important for the users expectations to not be violated, especially when it comes to the subject of money.

Closing events

We also decided to give the organizer the option of closing an event. At that time, no more purchases and payments can be made to that event. We considered the option of leaving events open indefinitely, so that people can add purchases they may have forgotten at a more convenient time, but we decided that the close option had distinct advantages. First, the fact that a “Close” option exists incentivizes people to submit their purchases earlier so they do not miss the cutoff set by the organizer. Second, it reduces the chance of fraud in the case that months later, members receive a malicious email asking them to pay, as they will not actually be able to make payments. Finally, this allows the organizer to make it such that once an event has been settled, it will remain that way forever.

Stubbing out Venmo

Finally, a design challenge was to stub out Venmo integration. 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, payment_split records are created with a receiver, amount, and original sender. Though this is invisible to the user, it provides a mock-up of the functionality of the app with venmo. If we were to actually use venmo we would have called the venmo api for every record in payment_split.

Concepts

Event: An event is a gathering of friends or acquaintances in which people split the costs of the items involved in the event. For example, an event could be a surprise birthday party, in which the items would be the supplies involved; or it could be a group of people who split the cost of a gift for their professor.

Closed event: An event where no more purchases or payments can be made.

Purchase: A purchase is a record of an item or list of items that a user has bought for an event, with an associated cost in dollars. Items can be physical (e.g. pizza) or represent a service (e.g. catering service).

Payments: A transfer of money from a user to *an event*. The payment is made directly to the event, and the app computes the individual transfers (between users) that have to be made so that all users receive what they owe.

Debt: The amount the user owes to the event

Credit: The amount the event owes to the user.

Organizer: A person that is responsible for creating an event by naming it and choosing the participants. An organizer is also a participant in the event who can add purchases and shares the costs of others' purchases. The organizer, however, has extra permissions for the event: they may edit the event description/name and close the event.

Testing

We have written tests for all of the models, except `payment_split` which is only used for mocking Venmo transactions (It is not actively used in the app, just populated). Each model has tests for the validations to make sure unwanted data can not be saved in the app. \

We also wrote unit tests for the logic methods in each model, paying special attention to the `payments` and `user_event_balances` models. These contain the core logic of the application, so we wrote comprehensive tests to ensure that debt and credit amounts are correctly computed after various combinations of purchases and payments.