6.170 4.2 Design Analysis

Key Design Challenges

1. Deciding what types of accounts are necessary and how to represent them
2. How to process payments from donors to the organization
3. How to represent donations to a person participating in a specific event rather than to the event as a whole or the person as a whole.

Data Representation

volunteers

- id: Server-generated ID for the volunteer
- name: Volunteer's name
- email: Volunteer's email (unique across both volunteers and organizations)
- password_digest: Hashed and salted password

organizations

- id: Server-generated ID for the volunteer
- name: Organization's name
- email: Organization's email (unique across both volunteers and organizations)
- password_digest: Hashed and salted password
- stripe_token: Stripe Connect token for the organization
- bio: Free-form bio of organization

events

- id: Server-generated ID for the event
- organization_id: ID of organization that's running the event
- description: Free-form description of the event

participations

- id: Server-generated ID for the participation
- volunteer_id: The ID of the volunteer who's participating
- event_id: The ID of the event that the volunteer is participating in
  the tuple (volunteer_id, event_id) is enforced to be unique across participations
- note: Free-form note provided by the volunteer to display on the participation page
- goal: The amount of money the volunteer hopes to raise for the event

donations

- id: Server-generated ID for the donation
- participation_id: The participation that was donated to
- amount: The amount of money donated

note that we do not store billing information: the one-use stripe token generated by stripe.js and sent to the server is used once to bill the donor and then discarded.

Key Design Decisions

1. Type of accounts:

Ways to implement:

Our system has three different types of users — organization, volunteer, and donor — who each have separate events. We could represent each type of user with a separate class, represent them all as one class with a field denoting special features of each class, or something in between.

What we did:

We chose to create separate classes for organizations and volunteers, both of which did not inherit from a parent class. We also chose not to create donor as a separate object for our MVP for simplicity. Also, we didn't want to require donors to have to create an account just to donate. We decided to separate organizations from volunteers since their permissions and events on the site are almost completely disjoint.

2. Payments:

Ways to implement:

In the simplest case we could have made payments just numbers in a database that don't actually pertain to real money (as many people did in the shopping cart project). We wanted to have a functional payment system that could actually deal with credit card information so that left PayPal and Stripe as being viable options.

What we did:

We decided to implement payment/donation processing through Stripe Connect so we wouldn't have to store a version of our users' payment information in our own databases. Security is key, and using Stripe means that the only information we have to store is a token, rather than credit card information, PIN numbers, etc... With Stripe, we don't have to handle any money — money flow directly from a donor to the organization. We chose not to use PayPal because we would have to store more information in our databases, and we have to give users a choice to be redirected to PayPal. PayPal also doesn't allow for a middle person to handle payments (accept payments from one entity and forward it to another).

3. Where a donation goes:

Ways to implement:

Our system has volunteers participating in events hosted by organizations. When a donor donates money the end result is that their money should go to the organization but we needed to represent in some way that the money is for a particular volunteer's participation in the event. This is important to keep track of since each volunteer wants to know how much money they have raised since oftentimes walks and events have a fundraising minimum. This can be implemented by having a donation object that belongs to a user and to an event and the user can find out how much money they have raised for each event by looking through all of their

donations associated events. This can also be done by having a separate participation object that belongs to an event and to a user that represents the user's involvement in a specific event so that donations can then be associated with just the participation. In this model the donation belongs to the user and the event through the participation.

What we did:
We decided to implement the second option where a user has multiple participation objects that belong to different events. We thought this extra layer of abstraction simplified the process of organizing which donations belonged to what event. In the first option in order to calculate how much money each user has raised for an event you have to go through all of their donations and group them by event each time which. This sorting and summing in our opinion is much more complicated than just having a separate participation object that contains all of the donations already grouped by event and by user.