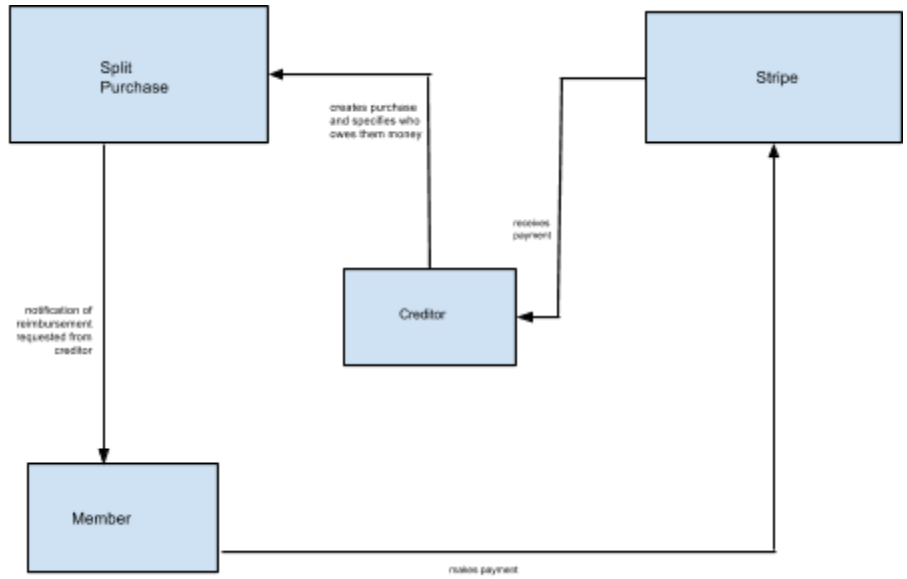
Final Design Documentation

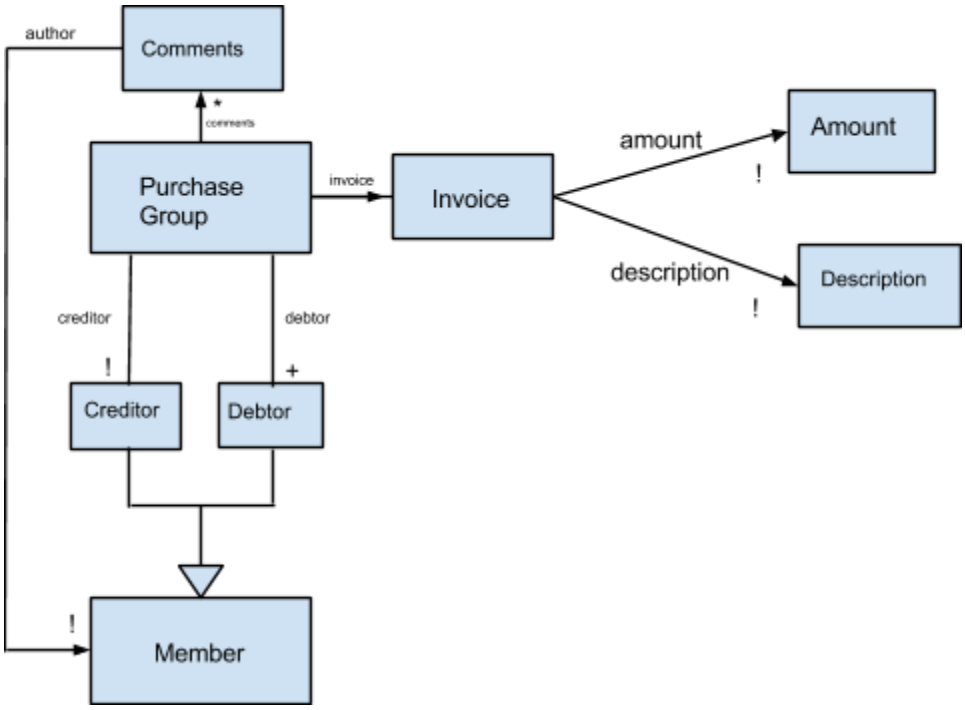
Primary Author: Isabella Tromba

Context Diagram



Primary Author: Isabella Tromba

Data model



Primary Authors: Isabella Tromba and Isra Shabir

Features

- **Purchase Group**- Displays an amount of a purchase and the invoices(attached to members) that are a part of that purchase
- **Payment Dues Visibility:** debitors can see the amount they have to pay to a certain group purchase
- **Group Creator:** One user creates the group that needs to split a payment - this user (creditor) invites others members (debitors) to the Purchase Group
- **Safe Payment System:** Once the payments are allocated, users will use STRIPE and enter their credit card information to make a payment.

Primary Author: Isabella Tromba

Security Concerns

1. **Security Requirement:** Adhere to PCI guidelines when taking credit card info/dealing with e-commerce on website
   Addressed: Being PCI compliant is very costly so doing this for our app is not going to be feasible. Instead, we will use a third party payment system (Stripe) that through cross-domain AJAX is able to send credit card info directly to their server. We will not be responsible for storing users' credit card information. Stripe is a well established third party API for such transactions. Since we used stripe we did not have to personally think about this security issue.

2. **Security Requirement:** Secure data transfer. All user information (password/address/etc) must be protected.
   Addressed: Data will always travel on secure channel: https. All password data will be encrypted and authenticated according to IPsec (Internet Protocol Security). Rails takes care of this if we set the ssl checking so that all requests happen over https.

3. **Security Requirement:** Secure storage of data on the server.
   Addressed: All data stored on the server (user passwords/ user information) will be encrypted. We used devise for authentication and devise automatically stores only encrypted passwords.

4. **Security Requirement:** Payments that are made are valid and come from valid credit card numbers/ real people
   Addressed: User Stripe for credit card validation and to track successful payments. Stripe provides all of the validation and bears all of the risk for this aspect of the

application.

5. **Security Requirement:** User Access Control.
   Addressed: Only users who have permissions(must be authenticated and authorized) to view a groups bills can do so. We required people to be signed in to view pages and added before_filters to only let members who were allowed to view a page to do so. This was a huge security hole that we did not previously realize that we were not addressing. It was mentioned to us by Carolyn and we were able to fix the issue and only allow users who had access to a certain page to view that page.

6. **Security Requirement**: Cross-Site Request Forgery
   Addressed: We need to make sure that sessions expire and we need to include security tokens in non-GET requests that check on the server side. If the security token does not match what was expected, the session will be reset.

7. **Security Requirement:** Session Hijacking
   Addressed: Session hijacking would allow an attacker to use a web application in someone elses name. In an insecure network this could be accomplished by sniffing the cookies being passed. In order to deal with this issue, we decided to always force an SSL connection. Also, we display prominently a log-out button so that users will be encouraged to log out of their sessions in the case that they leave a public terminal.

8. **Security Requirement:** Session Fixation
   Addressed: Session fixation is when an attacker sets someone else's session id to an id that they set themselves. In order to deal with this issue, on successful login the session needs to be reset. This means that the fixed session id that the attacker created is no longer valid and they cannot use it to co-use the application. Rails takes care of this resetting and so does devise (what we used for authentication).

9. **Security Requirement:** SQL injection attacks
   Addressed: The main problems here include attackers trying to bypass authentication and trying to read or create/delete objects in the database. We do not need to worry about SQL injection attacks because we are using active record to access/create objects in the database and activerecord handles these potential injection attacks.

Design Challenges

Primary Author: Isabella Tromba

1. How creating invoices works

Option 1: Do not allow to create invoice as part of group purchase for same person multiple times. Do not allow multiple invoices to be added for the same person. Do not allow group creator to add anyone who is not a member of the group.

Pros: Displaying who is part of a group is simpler because there should be no debtors that are duplicates as part of the group purchase.
Cons: We do not allow anyone to be added that is not already signed up with the account. This means that you have to send a personal email to the person informing them that they need to sign up in order for you to send them a group purchase invite request. If you want to charge a particular person for each individual purchase as part of the group you cannot do this. For instance, suppose 3 people go to a baseball game together. You may want to create a separate invoice for the food that the person ate at the game and the price of the ticket. Under the way we designed it, you would have to add these values together yourself and make only a single invoice for this person.

Option 2: Allow multiple invoices for the same person. Allow group creator to add someone who is currently not a member of the group.

Pros: In the scenario described in the cons of the previous option, this design would be better suited to tackle that issue. However, in most cases we do not believe people would need this functionality. Allowing users to send email invites to friends who are not yet users of the application would be a great feature. It would make sending invites a lot easier and more people would use the application.
Cons: Since the case listed above about the food and ticket purchases being separate seems to be a fringe case to us we believe our implementation is in fact more common and suits the needs of more users. The only con with the email invitations is the lack of time to implement this additional feature :(

We chose: We decided to go with option 1 because we believe that most of our users would be using our application with this kind of functionality and use case. The cons of option 2 outweigh the pros. The tokenization in javascript just checks the conditions of whether to display certain users and you can only create an invoice for one person at a time.

Primary Author: Isabella Tromba

2. What to display to different kinds of users (creditors/debtors)

We wanted to only allow group creators to add invoices if the unallocated balance is nonzero. We also wanted to allow group creators to see how much balance is left to allocate so that they cannot overallocate invoices to debtors. The unallocated balance is however not so important for debtors to know.

Option 1: Do not allow debtors to see the group purchase information (i.e. who else is invited)

Pros: Perhaps it is superfluous to show debtors who else is a part of the group purchase. Also, perhaps the creator of the group purchase wants to charge a group each a different amount but does not want each group member to see that they are paying a different amount (more/less) than other members
Cons: Exactly opposite of the pros. Most likely a debtor wants to see who else owes what to verify that they actually want to pay the amount allocated to them in the invoice. Perhaps they feel that what they owe is not fair because someone else owes less. So adding this visibility would make debtors more willing to pay and not allowing them to see

is hiding valuable information from them.

Option 2: Allow debtors to see the group purchase information but limit the functionality they have and the information they see(i.e. do not let them destroy a group purchase if they do not have the proper permissions).

Pros: Basically a reiteration of the cons in the previous option.The debtors would like to see who else owes money and if they are being charged a fair amount. Also, if we do not let the debtors see the group purchase they will not be able to see who they are paying/ who invited them to the purchase. This would not be great because then you are paying someone but you do not know who!

Cons: The main con is that perhaps you do not want the debtors to know how much or who you are charging as part of a purchase. We believe that this is ultimately a fringe case. We do not display the unallocated balance of the group purchase - Perhaps debtors would like to know how much money is still left to be unallocated. But in general we believe this is unnecessary because they could calculate it themselves and because they are not adding others to the group purchase, they wouldn't need to see this.

We chose: We decided to go with Option 2 because we believe that in a majority of the use cases for this application that is the functionality the users would want.

Primary Author: Isabella Tromba

3. Calculating Balances - How balances are split among debtors

Option 1: Allow the group creator to evenly split the balance among all members of the group

Pros: If you know exactly how many people were part of a purchase and that each one of them owes an equal amount of the total then you would want to be able to click one button and evenly divide the group purchase. You do not have to manually enter the same amount for each person (this implementation removes this repetitive task)

Cons: What if you create a group purchase and you think you only have 5 people that owe you money but then another person shows up at the party and you need to re-allocate all of the funds. Right now, you would have to destroy the purchase and re-allocated all the money evenly. You need to know exactly how many people are part of a group purchase when you create it.

Option 2: Make the group creator manually enter the invoice balance for each person they add to the group purchase

Pros: Allow the creator of the purchase to specify individually how much each person owes. That way if you want to add new people to the group purchase the creator can easily specify how much to make the new person pay based on how much has been allocated already (how much money they still need to collect). If we had implemented with an even split than a debtor would get a notification that they owe a certain amount (lets say there are only 2 people and the balance of the group purchase is $50 so each would owe $25). However, suppose now that the group creator forgot to add another person before sending out the invoices. This would mean that we would have to re-split the check (making each person owe $16.67) but that would require new invoices to be

sent out and the old ones destroyed. We decided that this was a messy implementation because you always have to know exactly how many people you are going to add to a group purchase when you create it.

Cons: The major con is the inefficiency introduced if the purchase is in fact divided evenly among all members. Suppose that you go Go-karting with your friends (20 of them) and they each owe you $50. Well now it seems like a big nuisance to go through and add each of them and specify individually that they owe $50. In this use case, our implementation is not as good. We designed our application with the first use case as our primary use case so we decided to go with that implementation.

We chose: We decided to allow the group creator to specify individually who owes what in creating the invoices. We believe that this makes the most sense for our type of application and adding the other option of splitting evenly would pollute the UI and our application would suffer in UX (unnecessary confusion/feature)

Primary Author: Isabella Tromba

5. How to display purchases that you created/owe money to

Option 1: Display groups you have been added to in the order that you were added so most recent groups appear on the bottom and old groups appear on the top

Pros: The most recent purchases are displayed on top so you take notice to them when you are added to a new one.

Cons: You may be more interested in seeing that you have been added to a new group purchase than interested in paying your old ones.

Option 2: Display the group purchases in reverse order that you added/were added to so that the group was the most recently added is displayed first and the older ones are displayed last

Pros: You will be incentivized by the ordering to pay the old ones and bring the newer ones higher up on the queue. Displaying it like a queue seems to make sense because thats the order of most importance that you make the payments.

Cons: You may miss that you have been added to a new payment because it displays on the bottom of the queue and not on the top.

We chose: We decided to go with Option 1 where the most recent ones are displayed at the end. We believe that this is the best option because the oldest added ones are the most pressing. Since they were created first you should pay these first because they are the most overdue.
This is the way that rails naturally displays them on the page.

Primary Author: Kevin White

5. How to make a payment

Option 1: Use a button next to the balance that enables the user to make payments via Stripe

Pros: There is clarity for the end user in how to make a payment for an invoice they received. There is no ambiguity.

Cons: It makes the design less clean and appealing. As more invoices appear on a

member's home page, more pay buttons appear on the screen, creating clutter

Option 2: Have the balance as a hyperlink that enables the user to make payments via Stripe by clicking on it.

Pros: It provides for a more minimalistic design that eliminates the need for a superfluous button on the screen.

Cons: It may be unclear to the end user that by clicking on the hyperlink, they are able to make a payment. As a result, the user might not know how to make a payment for their invoice.

We chose: In our design, we chose to integrate stripe using a payment button. This gave the user clarity in how to use our application while trading off a more minimalistic interface. At the end of the day, we felt the intuitiveness of our app was more important than minimalism.

Primary author: Isra Shabir

Payments solution:

Option 1: Paypal

Pros:

- Well established brand name for online transfers
- many people already have accounts

Cons: From a technical standpoint, documentation wasn't developer friendly to allow smoother integration into a rails app

Option 2: Venmo

Pros: Handles transfers between people more smoothly as it is their primary service

Cons:

- we weren't too sure about documentation on the web as it is a relatively newer product.
- users need to create a separate Venmo account to use their service which defeats the purpose of our application - a black box to allow payments once a bank account has been setup

Option 3: Stripe

Pros:

- promote their product as being the third party API which is developer friendly and meant for smooth integration into any application that requires payments i.e. better service to go live and have a better on boarding service
- serves the black box desirability for users. Users only need to setup a payment account once and subsequently only hit a "pay" button to make payments to their creditors - no need for separate accounts

Cons: Because there's more functionality, there's heavier documentation. despite smoother integration, setup takes long.

We chose: Stripe. Our app design required a third party API that would allow payments in the

smoothest and most efficient way possible. Once integrated, users only need to setup a payments account once before starting to make payments. It serves our need for a black box payments system well.

Note:
Integrating Payments was a relatively harder component of the application as it took time to read up on the existing documentation online to 1) choose the best possible solution, 2) implement a solution by reading the documentation, 3) test the payments solution