Overview:

ListJam is a web service for selling and buying items. This service is for people who want to create their own shops online with emphasis on their shop rather than on the individual items they are selling. There is also emphasis on lists (carts) as guides for what to buy for specific "ideas" (eg. a list of items for going on a picnic, throwing a party, etc). Essentially I wish to create a more social experience for finding groups of things tailored to an "idea" for a user to buy. There is less emphasis on finding individual items and more on finding these groups of items, however, users can build these lists by finding individual items at different shops.

Context Diagram:

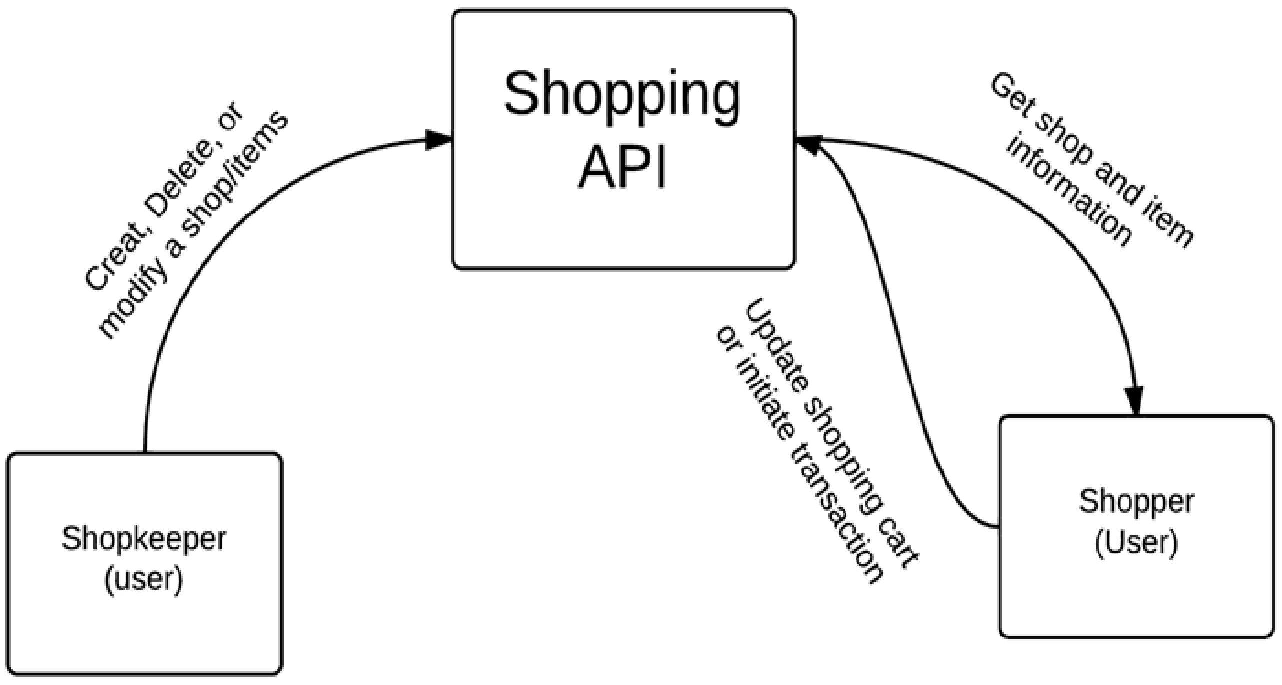
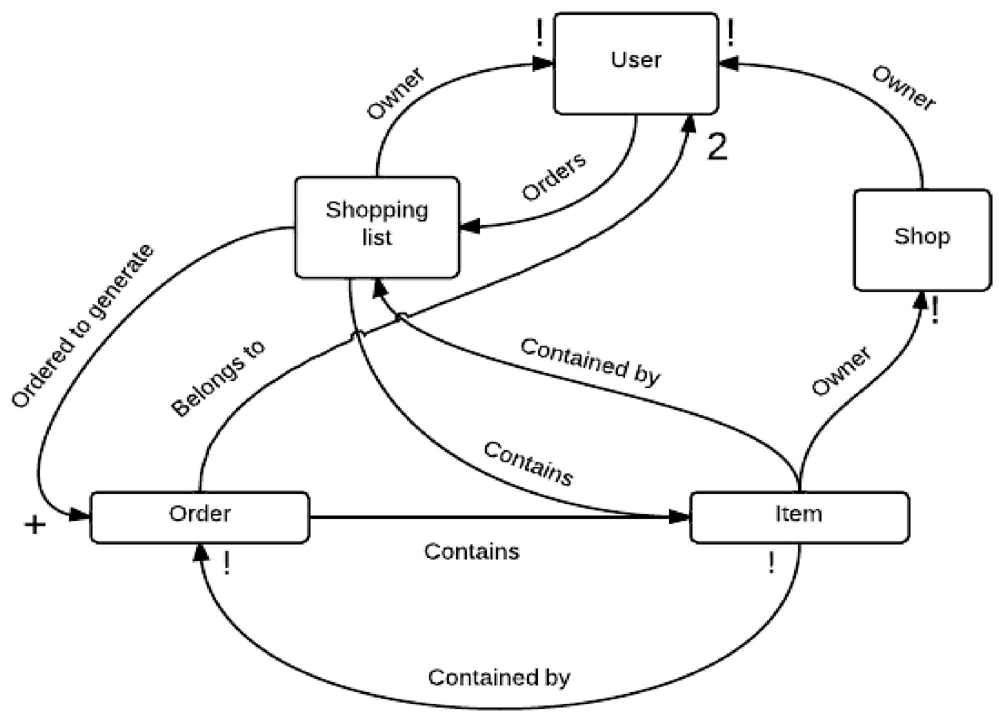

Shopping API
Creat, Delete, or modify a shop/items
Get shop and item information
Update shopping cart or initiate transaction
Shopkeeper (user)
Shopper (User)

Figure 1: Context Diagram for ListJam. The edges represent interactions a user can have with the service.

Concepts:

User—Owns many shopping carts AKA lists (I am using carts and lists interchangeably to explain it but I used "list" throughout the service.
Shopping cart(list)—holds items.
Item—is sold by a shop.
Shop—has a selection of items.
Shopkeeper—owns and manages shops.
Order—is placed by a User and given to a Shop.

Object Model:



Behavior

Feature Descriptions:

- **Shop creation**: A registered user can create shops and **Items** which are added to the shop.
- **List creation**: A user can create a **list** which holds items that can be ordered at a later time. These lists cannot currently be deleted and are visible for anyone to see via the "lists" index for registered users, or via direct link for temporary users.
- **User creation:** All people who visit the site automatically get a temporary user assigned to them. They can choose to become a registered user by using the "sign up" feature. Registered users can create more lists and can create shops; registered user's lists also show up under the "list" index while a temporary user's lists are not publicized (but a temporary user's list can still be shared via direct-link using the list's id). Additionally, all accounts can be viewed via direct link so anyone can see anyone else's shops and lists.
- **Ordering:** Users can order their list. This actually creates an order for every item in their list similar to Amazon. The user can then see these transactions under "Orders" if they are the user ordering, or "Sales" if they are the user whose item was ordered.

Security concerns:

Problem: Storing passwords

Option 1: Store passwords in plain text. This option was not chosen because if a hacker were to somehow break into my database then they'd be able to steal password information

Option 2: Hash the passwords. This option was chosen because even if a hacker were to somehow break into my database then they would not be able to steal any password information.

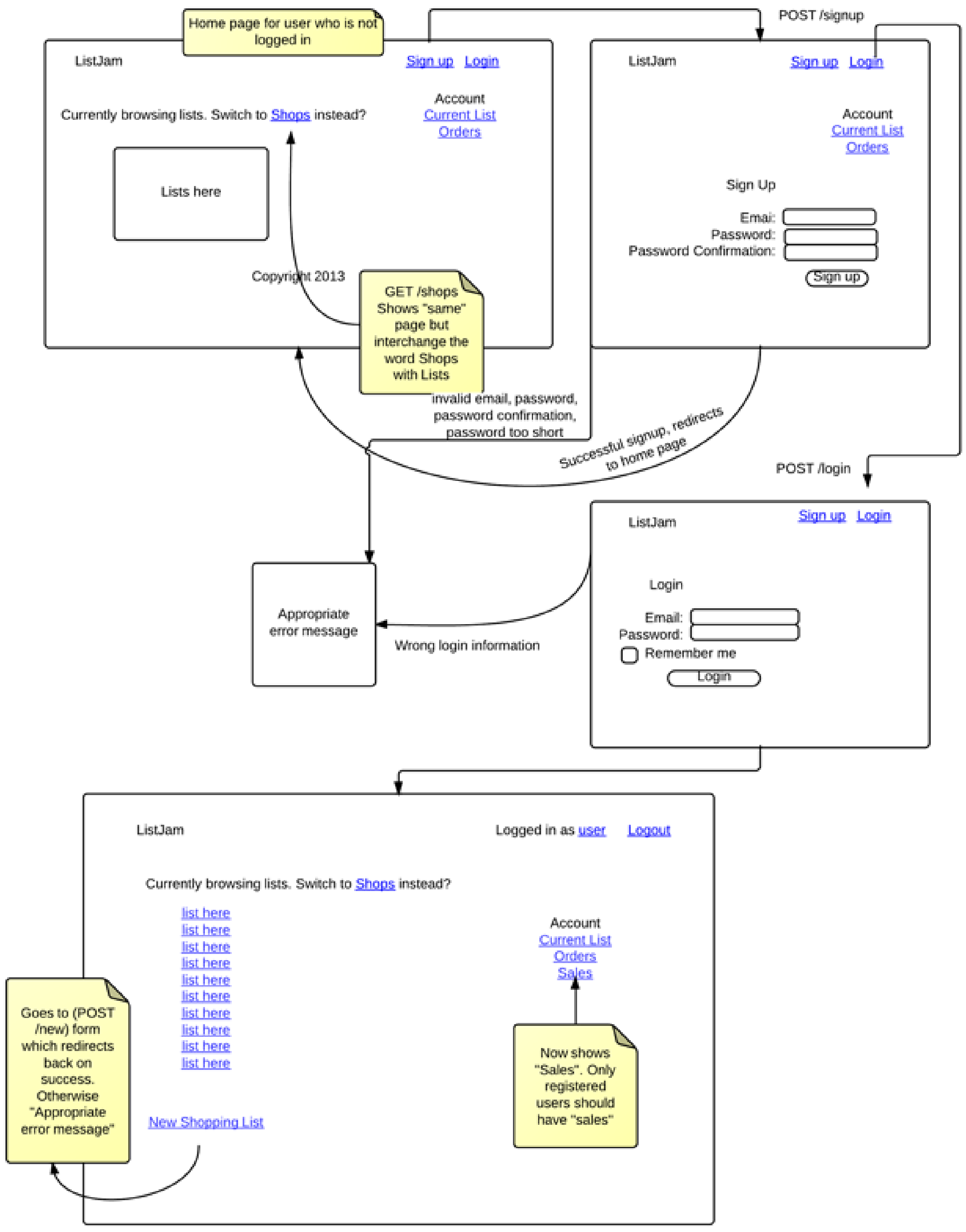
The only con is that we have to hash a user's password every time they login which is not a big performance concern.

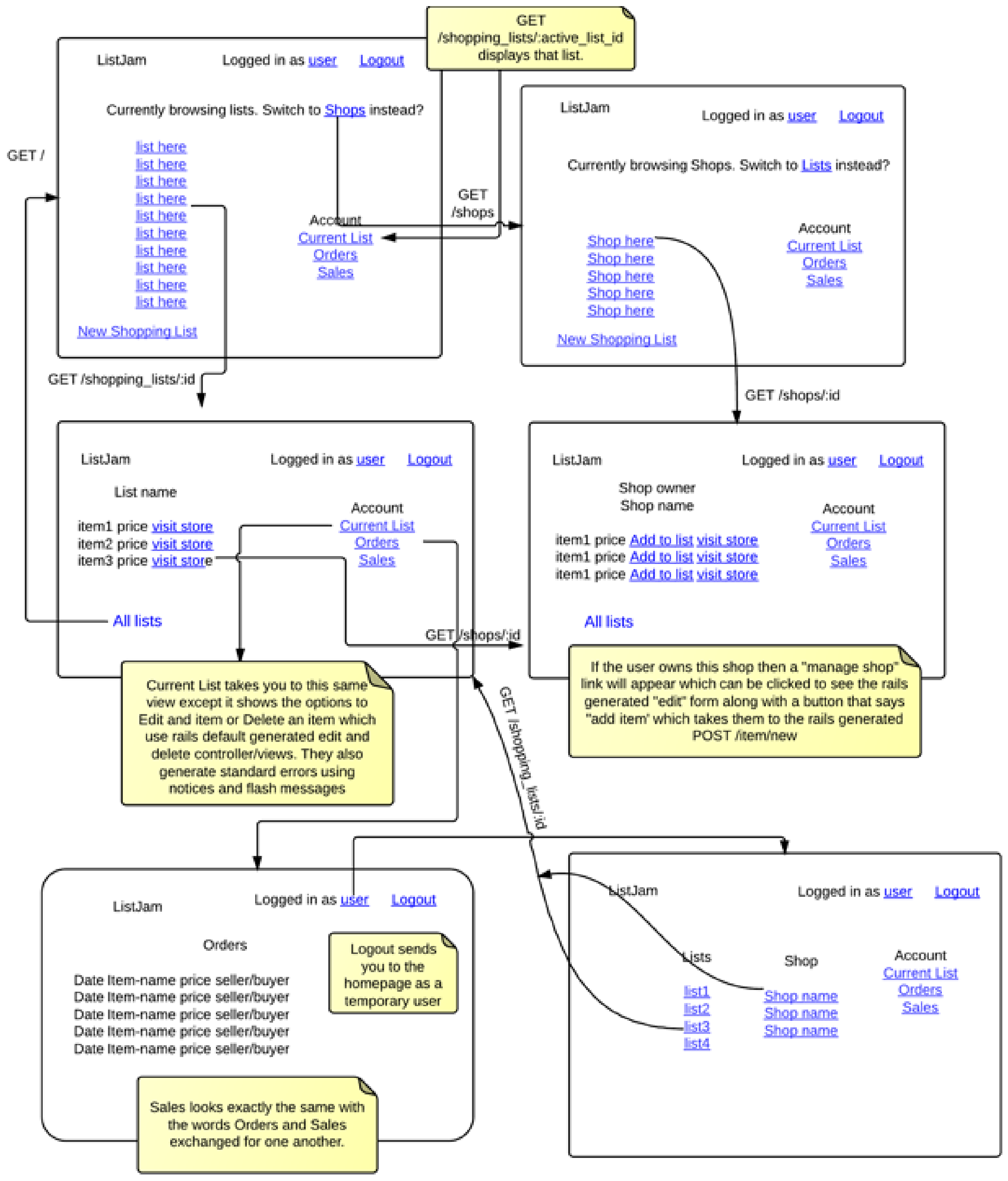
Problem: Accessing resources specific to a user (Shop management, item management, and list management)

Option 1: Use a URL with a special hash to the resource (making it impossible to guess). This option was not chosen because a hacker could steal browsing information from a user and use this to modify the resource.

Option 2: Use a normal route with no special URL that uses the user's session to determine what information the user can access. If the user is not logged in then they will be redirected to the home page and told they do not have permission to do the action they were attempting to do

Wireframes:


Home page for user who is not logged in
ListJam
Sign up
Login
Account
Current List
Orders
Currently browsing lists. Switch to Shops instead?
Lists here
Copyright 2013
GET /shops Shows "same" page but interchange the word Shops with Lists
POST /signup
ListJam
Sign up
Login
Account
Current List
Orders
Sign Up
Email:
Password:
Password Confirmation:
Sign up
invalid email, password, password confirmation, password too short
Successful signup, redirects to home page
POST /login
ListJam
Sign up
Login
Login
Email:
Password:
Remember me
Login
Appropriate error message
Wrong login information
ListJam
Logged in as user
Logout
Currently browsing lists. Switch to Shops instead?
list here
list here
list here
list here
list here
list here
list here
list here
list here
list here
New Shopping List
Account
Current List
Orders
Sales
Now shows "Sales". Only registered users should have "sales"
Goes to (POST /new) form which redirects back on success. Otherwise "Appropriate error message"



If the user owns this shop then a "manage shop" link will appear which can be clicked to see the rails generated "edit" form along with a button that says "add item' which takes them to the rails generated POST /item/new

Current List takes you to this same view except it shows the options to Edit and item or Delete an item which use rails default generated edit and delete controller/views. They also generate standard errors using notices and flash messages

Logout sends you to the homepage as a temporary user

Sales looks exactly the same with the words Orders and Sales exchanged for one another.

Challenges

Deciding between having a separate buyer user and a separate seller user.

I decided not to distinguish between the two. I have one user who can do both selling and buying. This makes it easier for the user to do both without need two accounts which is a good pro. On the other hand this means I had to set up trickier relationships for orders, specifically I needed to specify different foreign keys to distinguish between the seller and buyer of an order. This leaves a lot of room on my part for error and possibly harder to maintain code; it also leads to a much larger user class. One alternative would have been to subclass a base user class and have one for seller and one for a buyer which might have been preferable.

Letting un-registered users have carts and order.

I decided to let un-registered users have carts. To do this I created a user that was flagged as temporary. This user would be saved to the database and would have a shopping list that is also flagged as temp. However, If the temporary user attempts to order his list he will be told that only registered users can order, but this is no problem because as soon as the temporary user logs in his old cart will be transferred to the account he logs in to (if the list is not empty, otherwise the list is destroyed), at which point the temporary user is destroyed. One pro to doing this is that, if a user gets to the point of ordering after taking the time to build a list, he/she will probably do the last step of registering and logging in to finish that order. One con to doing this is that if a user doesn't register then his user will never be erased from the database; nor will the shopping list that is automatically created for it, this can lead to a database cluttered with users and shopping lists not actually needed. One solution to this might be to auto-prune models with a temporary flag that have not seen any activity in the past 7 days. An alternative solution would have to be just not let unregistered users have lists; this would have mitigated the problem.

Giving users multiple lists.

I decided to let users have multiple lists. A consequence of this is that I have an extra attribute "active_list_id" that I use to keep track of the list the user has as "active" where "active" defines which list is updated when the user adds an item to his list, that is, whenever a user adds an item to a list he is adding that item to his "active" list. The user can visit any of his other lists and click on "make active list" to make that list active. This is good for the user because it lets him/her have multiple predefined lists that can order whenever he/she needs. The alternative to this would have been to only have 1 list which would mean I don't need to track an active_list because a user only has one list as opposed to many.

Note on code design:

I decided to have only one user model which means I have to use a flag to tell the difference between a temporary user and a normal user. I also decided to have only 1 item class which meant I also had to flag my items to check if they were "ordered". The consequence of my latter decision was that I have duplicates of an item equal to the number of times that the item was ordered. The reason that I duplicate the items is because the owner of the item might change the item's price, hence we clone the item to create a snapshot of the item at the time of purchase.