

# Design Documentation for Grand Exchange

Ami Suzuki, Eunice Lin, George Du, Jeffrey Sun

## Overview

### Motivation

### Description

The Grand Exchange is an item-centric centralized online marketplace for the MIT community. Users can post auction-style sell or buy offers on the application for a particular item and will be notified by the app when a match is made between a buyer and a seller. When a match is made, the offers will be taken off the marketplace and the users can meet in person to do the actual transaction.

### Key purposes

1. Provide an item-centric rather than offer-centric online marketplace
  - a. Consolidates all offers for the same item so that users do not have to scroll through lots of posting of the same item
  - b. ~~Provides the fairest price determined by the market~~ Allow the supply, demand, and pricing for an item to be determined by the users.
2. Centralization of buy and sell
  - a. Consolidates all buy and sell offers in the same place, rather than dispersed over emails forums, Facebook groups, etc.
3. Takes the burden off of buyers and sellers by facilitating the time burden of transactions
  - a. Does the match-making for the users so that buyers and sellers do not need to negotiate the offers or respond to specific offers themselves, which saves time and effort

### Deficiencies of existing solutions

- eBay and Craigslist
  - Offer-centric; users have to scroll through long lists of postings searching for an item they want.
    - We want to create an item-centric model where user can just name the item and price they want without having to search through a bunch of different postings.
- APO Book Exchange
  - Only for textbooks and clickers.
    - We want to expand our application to accommodate items in addition to textbooks and clickers.
  - Not online; people have to bring books to the APO Exchange and if they are not sold they will have to bring them back.

- Users of our application will only have to meet in person to do the transaction if they were guaranteed a match. This will save people the time and effort of bringing their items to sell in case no one buys it.
- Facebook groups and email forums
  - Not centralized and difficult to keep track of; users will have to post in the group individually and their post are not searchable so it might get buried by other posts. Users also have to periodically check the site for new offers and privately message the other person if they want to arrange a transaction. Often times, people would message the seller, and the item would have already been claimed.
  - We want to create a centralized platform so that the user only has to name the price and item and they will be notified by the application when a match as been made. Therefore, the user does not have to worry about their post being buried by other postings and do not have to check the website frequently.

## Context Diagram



## Design Model

### Concepts

- Item -- The object being bought/sold and its buy and sell offers  
Motivated by:
  - Purpose 1: Each Item can have multiple offers. Users will look at the many offers of an item, rather than the single item of an offer, so we are providing an item-centric rather than offer-centric marketplace.
  - Purpose 2: Each Item has both buy and sell offers, which centralizes buy and sell
- Offer -- A user's intent to buy or sell at a particular price, [and with other users at a particular reputation or above](#). Items may have multiple buy and sell Offers  
Motivated by:
  - Purpose 1: Offers exist under Items and Items may have multiple buy and sell offers, making the marketplace item-centric

- Purpose 2: There are both buy and sell offers for each Item so buying and selling is centralized
  - Purpose 3: Offers allow the system to make a match so users can easily buy or sell
- Transaction -- Has exactly one buy offer and one sell offer that have been matched. The system matches the highest price buy offer with the lowest price sell offer if the sell price is less than the buy price. Also keeps track of whether a review has been set for buyer and seller.

Motivated by:

- Purpose 1: The system creates a Transaction when there is a match from the collection of buy and sell offers, so buy and sell are centralized
  - Purpose 2: Transactions are created by the system, so buyers and sellers do not have the burden of finding matches themselves.
- User -- MIT community members (with MIT email addresses) that can create buy/sell offers

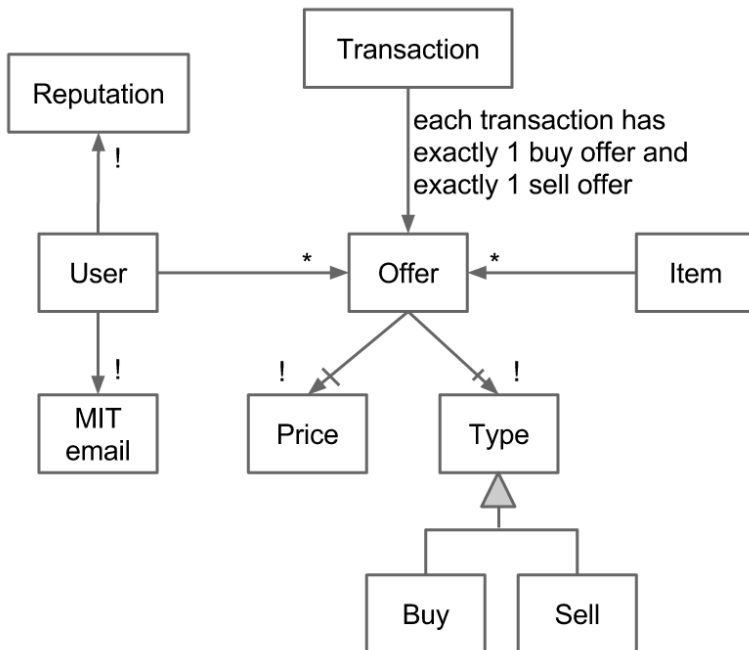
Motivated by:

- Purpose 1: Users can create offers under Items so each Item may have multiple offers, making the marketplace item-centric
  - Purpose 2: Users are able to create both buy and sell offers, which centralizes buy and sell
- Review -- Consists of text and a score from one to five.

Motivated by:

- Purpose 1: Users can rate transactions and leave feedback
  - Purpose 2: Allow users to limit matching to people with >certain reputation

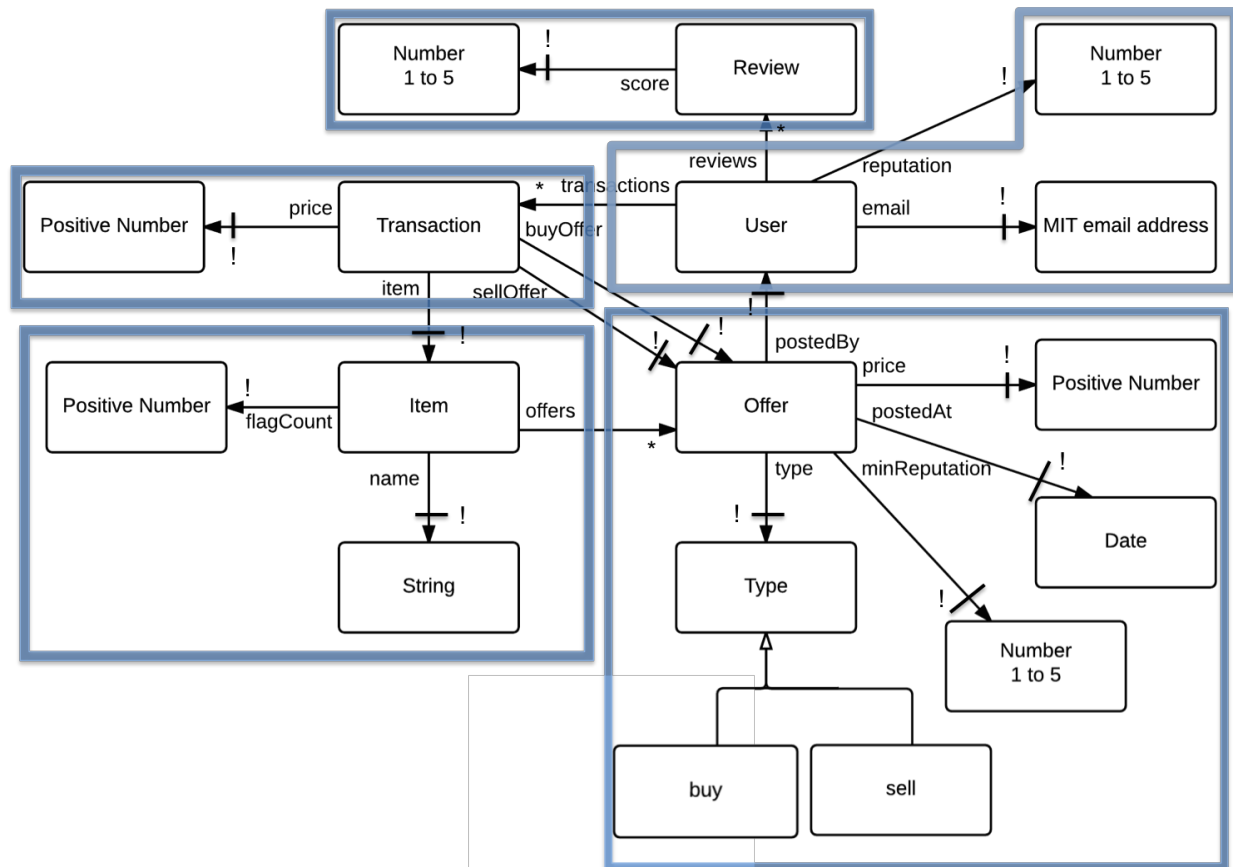
## Data Model



Users of the application must have a valid MIT email address. Users can post offers on the application. Each offer has a fixed price and a fixed type, either buy or sell. An item can have zero or more offers. A transaction is created when a match is made between a buy offer and a sell offer, [where each user is willing to trade given the other's reputation](#). Each transaction consists of exactly one buy offer and one sell offer, which must be for the same item. The price of the transaction is the minimum of the two offer prices, and the sell offer price must be greater than or equal to the buy offer price. Each user has a reputation, which is a rating based on other users' feedback on them. Users can rate each other [on a scale from one to five](#).

## Data Design

### Transformed Data Model



- A transaction is created (automatically) when, for a particular item, there is a buy offer priced for more than the lowest sell offer's price.
- A user's reputation is the average of his/her review scores.
- A review is generated by a POST through a transaction, ensuring that a user only gets reviewed once per transaction.

## Behavior

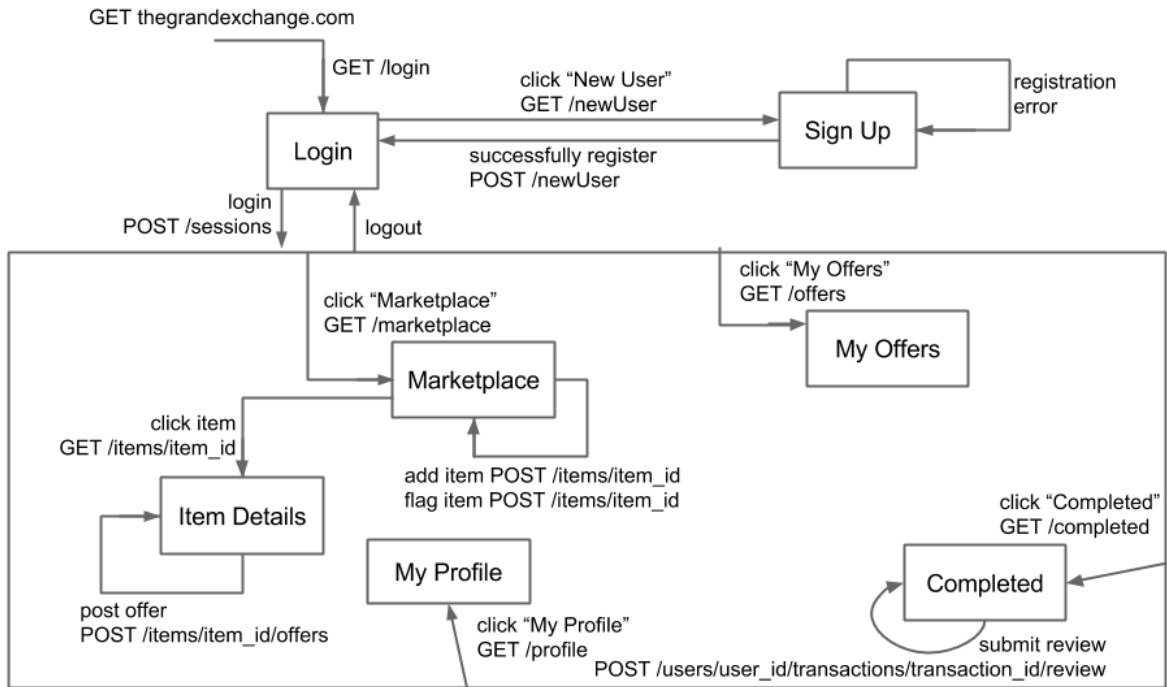
### Security Concerns

Our basic security goal is that users should be limited to performing only allowed actions and viewing allowed data. The allowed actions are: buy/sell offers, rate users, and add items. The allowed data is: the available items, offers on those items, information about the user's own offers, and other users' reputation. This will be accomplished by using a REST API which acts

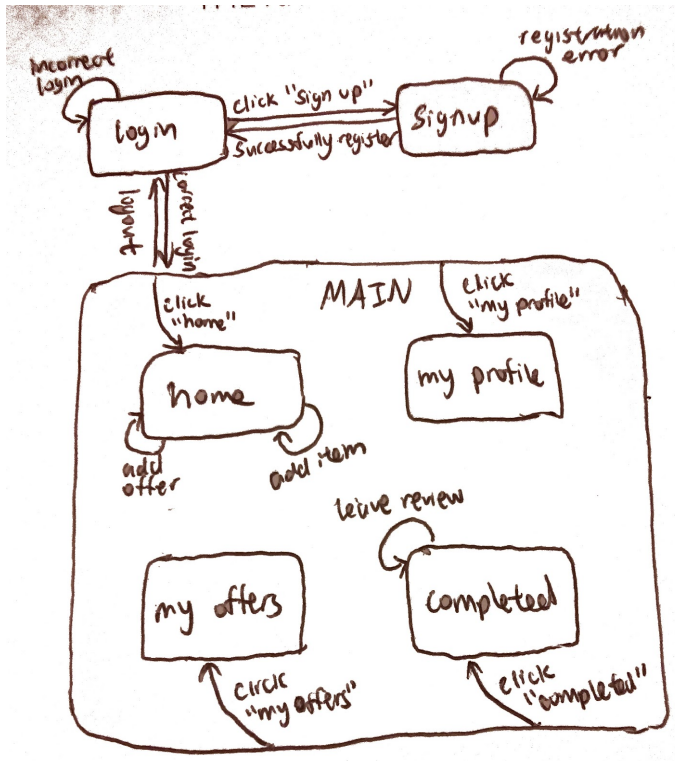
as an RPC interface limiting the actions a browser can perform, even using custom-built requests. The standard attacks such as those listed in the OWASP top ten are mostly taken care of by our framework AngularJS (injection, XSS, and CSRF in particular.)

We model the attacker as an entity which can send arbitrary requests w/ arbitrary cookies such as those created in Postman. Our API should be designed so that any malicious requests (those not normally allowed for a user) will fail.

## User workflow diagram [Revised]

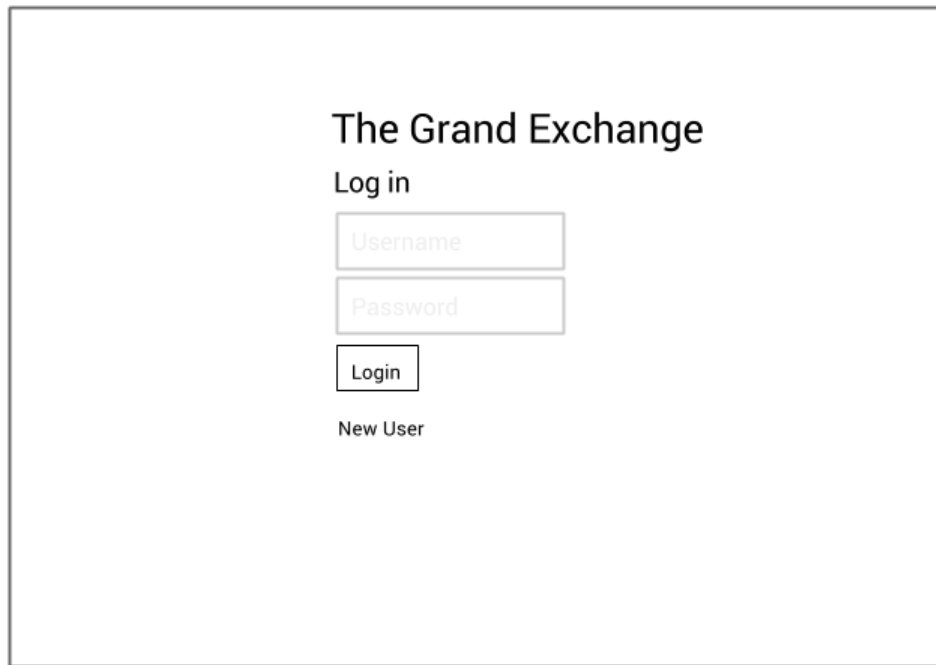


## User workflow diagram [Previous]



## Page Wireframes [Revised]

### Login Page



The Grand Exchange

Log in

Username

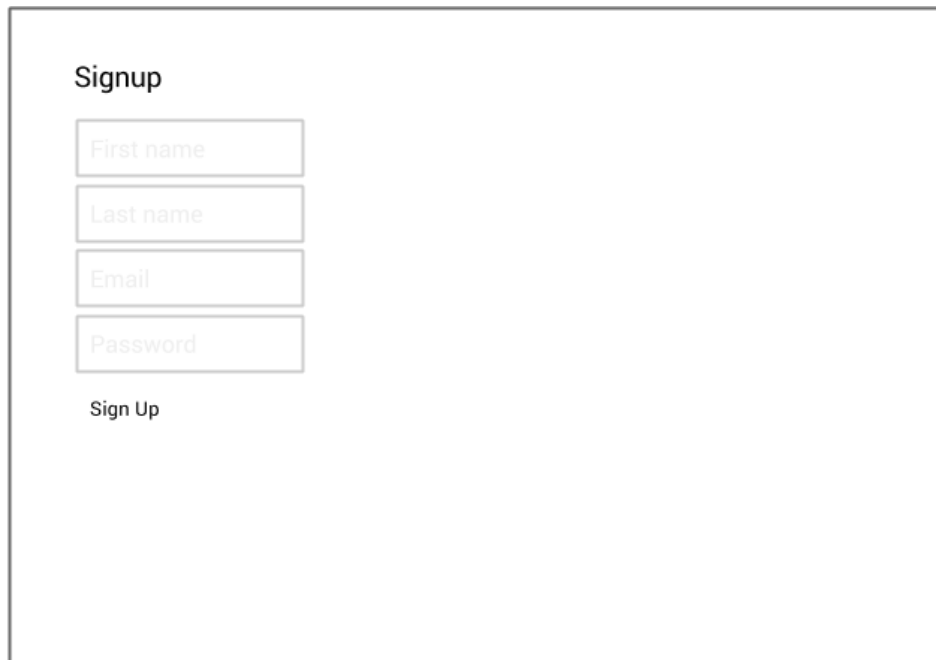
Password

Login

New User

This wireframe for the login page features a central heading 'The Grand Exchange' in a large, bold font. Below it is the text 'Log in'. There are two input fields: 'Username' and 'Password', each with a light gray border and placeholder text. Below the password field is a 'Login' button with a black border. At the bottom of the login section is a link labeled 'New User'.

### Signup Page



Signup

First name

Last name

Email







Password

Sign Up








This wireframe for the signup page features a heading 'Signup' in a bold font. Below it are four input fields stacked vertically: 'First name', 'Last name', 'Email', and 'Password', each with a light gray border and placeholder text. Below the password field is a 'Sign Up' button with a black border.



## Marketplace

Marketplace	<h3>Marketplace</h3> <p>The Grand Exchange is an auction-style, item-centric centralized online marketplace for the MIT community.</p> <div><div><div>Sell: xx.xx</div><div>Image</div><div>Buy: xx.xx </div></div><div><div>Sell: xx.xx</div><div>Image</div><div>Buy: xx.xx </div></div><div><div>Sell: xx.xx</div><div>Image</div><div>Buy: xx.xx </div></div><div><div>Sell: xx.xx</div><div>Image</div><div>Buy: xx.xx </div></div></div> <div><div><div>Sell: xx.xx</div><div>Image</div><div>Buy: xx.xx </div></div><div><div>Sell: xx.xx</div><div>Image</div><div>Buy: xx.xx </div></div><div><div>+</div></div></div>
-------------	---

## Adding an Item

Marketplace	<h3>Marketplace</h3> <p>The Grand Exchange is an auction-style, item-centric centralized online marketplace for the MIT community.</p> <div><div><div>Sell: xx.xx</div><div>Image</div><div>Buy: xx.xx </div></div><div><div>Sell: xx.xx</div><div>Image</div><div>Buy: xx.xx </div></div><div><div>Sell: xx.xx</div><div>Image</div><div>Buy: xx.xx </div></div><div><div>Sell: xx.xx</div><div>Image</div><div>Buy: xx.xx </div></div></div> <div><div><div>Sell: xx.xx</div><div>Image</div><div>Buy: xx.xx </div></div><div><div>Sell: xx.xx</div><div>Image</div><div>Buy: xx.xx </div></div><div><div><h4>New Item</h4><div>Name</div><div>Description</div><div>Upload Photo </div></div></div></div>
My Offers	
Completed	
My Profile	
Log Out	

## My Profile

Marketplace	<div>Your Name Here</div> <div>Reputation: x</div> <div>Reviews</div> <div>Reviews</div> <div>Reviews</div>
My Offers	
Completed	
My Profile	
Log Out	

## Item Detail

Marketplace	Sell: xx.xx with reputation y	
My Offers	Sell: xx.xx with reputation y	
Completed	Sell: xx.xx with reputation y	
My Profile	<div>Image</div>	Description: yada yada yada yada yada yada yada yada yada yada
Log Out		I'm willing to <div>Buy</div> for <div>xx.xx</div> <div>Sell</div>
	Buy: xx.xx with reputation y Buy: xx.xx with reputation y Buy: xx.xx with reputation y Buy: xx.xx with reputation y Buy: xx.xx with reputation y	to someone with reputation at least: <div>y</div>

My Offers

Marketplace			
My Offers	Price	B/S	Item
Completed	xx.xx	B	yada1
My Profile	xx.xx	S	yada2
Log Out	xx.xx	B	yada3
	xx.xx	S	yada4
	xx.xx	B	yada5
	xx.xx	S	yada6
Show 25 50 99 on page			

Completed Transactions

Marketplace			
My Offers			
Completed	Your Completed Transactions		
My Profile	<div><div>Bought x from y for \$z</div><div>Rate your transaction: ★★★★★</div><div><div>Write a review...</div><div>Submit</div></div></div>		
Log Out	<div><div>Sold x to y for \$z</div><div>Rate your transaction: ★★★★★</div><div><div></div><div>Submit</div></div></div>		

## Design Challenges

### *Preset items vs. dynamic items*

Since our application is item-centric rather than offer-centric, one of the limitations is that we can only offer homogenous goods, since users cannot distinguish between different types or quality of the same item. Therefore, we thought about having pre-set items that our site could offer. This prevents the users from adding inappropriate or unreasonable items, such as “my old laptop” or “a dinosaur.” The other option is to allow users to add new items when they want to buy or sell something that is not already offered on the site. This allows our site to be more flexible in accommodating everything that people want to buy or sell. We chose to allow users to add any item to the site and regulate inappropriate or fake items with reviews or the reputation system, such that users who make inappropriate or fake offers will receive low ratings and negative comments from other users and will eventually be banned from the site.

### *Transaction objects*

Transactions are matched automatically when buy and sell offers intersect. In an ideal world, this would correspond to a transaction happening. Realistically, however, we have to take into account things such as fake transactions, transaction spamming, and transactions that fail to occur due to low item quality, etc. We choose to have transaction objects which can reference the offers that spawned them, thus providing a way to recreate those offers if the transaction fails. Although this feature may not be present in our MVP, our data model anticipates later challenges related to transaction verification and provides a way to mitigate them.

### *User Verification*

Since we need users to be able to trust one another, we decided to make our app open only to MIT community members. We could either have users with MIT certificates, or users with MIT email addresses. Since there are a variety of MIT email address types (such as @mit.edu and @csail.mit.edu), using MIT email addresses would require us to use libraries that would do email verification. On the other hand, using MIT certificates has its own challenges. Since scripts.mit.edu does not work with node.js, in order to use certificates, we would need to generate and sign a fake certificate. However, this would cause the browser to question every time whether our site should be trusted. We have decided to use MIT email addresses rather than deal with the difficulties that may arise from trying to use certificates.

### *User Workflows*

One important issue is how to structure the pages in the front-end of the application. Specifically, there are multiple ways to allow users to view an item’s details and put in offers. One solution is to display item details and offer inputs on the same page that shows all the items, and have the item expand on click. While this approach speeds up the offer process and motivates the user to spend more time buying and selling, it doesn’t give individual items an identity. On the other hand, putting the page detail and offer form on its own page allows individual items to be linked to from a user’s offer list. This was the chosen approach, because it was decided that granularity at the item level is worth a sacrifice in speed.

### *Dealing with Inappropriate Items*

Since we decided that users should be able to add any items they want to the site, we want to have a way of maintaining the qualities of the items added, so that the marketplace does not become crowded with inappropriate items. An option is to have an admin who would have the ability to remove items whereas other users cannot. This option is good because we can be the ones to decide what items are appropriate for our site or not. However, this means we have to constantly browse the marketplace and check for inappropriate items, and if we don't check frequently, the inappropriate items will not get removed. Another option is to allow users to be able to delete items. This way we will not need an admin, but there is a problem where users may have different opinions on what an appropriate item is, so a user's item which may be useful to some people may actually get deleted. Another option is to have a flagging system where users can flag an item as inappropriate, and when three users have flagged an item, the item will be removed from the marketplace. This method ensures that any item that is deleted is something that many people agree is inappropriate while not requiring an admin who needs to constantly check the site. Therefore, we decided to implement the flagging system to deal with inappropriate items.

### *User Reputation*

Although it was relatively straightforward to allow users to rate one another on transactions and to generate a reputation from these reviews, it was less clear what we should actually do with user reputation. Some possibilities are: allowing people to cancel transactions with users of low reputation, banning users whose reputations fall below a certain level, and allowing users to see others' reputations when making offers, allowing them to avoid those offers with bad reputations by pricing their offers accordingly. However, each of these solutions has its own downsides. Cancelling transactions defeats the idea of a stock market, since an offer should be a commitment to buy or sell. Banning users is a last-resort solution. And the last solution seems too passive, putting the onus on the user to avoid low-reputation users. Finally, we settled on allowing users to set a minimum reputation for each offer, so that the offer would only match offers made by users with that reputation or higher.

## Implementation Challenges

### *Client-side MVC*

One of our goals is to maintain client-side code that is easy to understand and modify. We determined that it was better to use an MVC framework than, for example, jQuery with handlebars, because it allows us to better separate logic used in the view from the logic used to transmit data to and from the server. To that end, we also modularized the client side, putting each controller, each service, and each view in a separate file, and joining them at the end using a build system.

### *Transaction Rating*

Our initial data design was not sufficient to allow for rating transactions, as it did not provide any way for the system to know whether a user had already been rated for a particular transaction. Specifically, we originally intended to have “rating” be a POST on a “user/:userid/review” route. We solved this by moving the action to be a POST on “user/:userid/transaction/:transactionid”, and added two boolean fields to transactions “buyerRated” and “sellerRated”, which ensured that a user could not rate a transaction multiple times.

### *Testing*

We wish to be able to test our server-side methods, especially those touching the database, without making any unwanted changes to actual user data. To solve this, we are considering using a library such as mocha-mongoose, which will allow us to use a dummy database with sample data for testing.

### *Offer Matching*

The matching of offers and creation of Transactions must occur in the createOffer method so that we check for possible matches immediately when a user creates an offer. Offer matching requires prices of various offers of opposite types (buy or sell) to be compared, so we needed a way to access all of a given item’s offers’ prices and types. When createOffer is called, we are passed the data for the offer and the ID of the item to which the offer will belong. In order to get all of the item’s offers’ type and price information, we realized the need to populate the item’s list of offer IDs. We did this by calling findOne to get the item by its ID, then calling populate. To reduce the number of prices to compare later, we also found that calling populate with a match parameter would allow us to get only the “buy” or “sell” offers of the item. In this way, our app is able to check each offer of the opposite type belonging to the item to find the best matching offer.

### *Deep Populate*

In order to reduce the number of calls required on the client-side, we realized the need to populate the objects returned by our back-end. We initially had our GET method for all transactions of a user return transactions with only ObjectId references to their buyOffers and sellOffers. However, in cases such as when displaying a transaction, the transaction’s buy offer, and information about the user who posted that buy offer, having the transaction object with populated offers, which have populated users would provide convenience on the front-end. We searched for methods to deep populate with Mongoose, and were able to find a way that requires models to rely on other models. For example, populating a transaction with offers, and then populating those offers with their users relied on the Transaction, Offer and User models. By implementing this method, we were able to deep populate most of our objects returned by GET calls.

However, there was another challenge we encountered when we tried to populate in the GET method for offers of a user. We wanted to be able to access a specific user’s offers, and then access the item information of those offers. Using the above method, we would need to use the Item model in order to deep populate the items of the offers. However, this GET call was

in the User and Transaction file, which did not have access to the Item model. We tried to import the Item model by using a require statement at the top of our User file, but this did not work because the Item file was also importing the User model. We found that we could not allow both files to import each other, or a loop would form. We decided on a balance to allow the back-end to do as much populating as possible without the import loop, and compensate in other parts of our code by making extra function calls there to reach the required data.