CS 307
Software Engineering

Team 17

Bryan Duffy, Michael Reed, Simon Smith
Derek Schrag, Hari Mantena, Akshit Gudoor

Roomedy

Design Document

Table of Contents

Purpose

Roommates, even the good ones, struggle with communication. Until now, there has been no singular solution to help roommates gently remind each other and keep track of everything in their house. We plan to provide the one-stop solution for roommates to check on each other and keep track of everything from their debts to the contents of their fridge. Other solutions to roommate management exist, however ours is different in that it allows you to take all actions, such as paying your share of the bills, without leaving the app through integration with services most people already use. Other solutions simply provide reminders, we provide you resources to act.

This design document thoroughly describes our functionality requirements.
Account Management: The Roommates and the tenants must be able to manage their own personal account and other accounts.

1. As a user, I want
    a. to be able to create a personal account.
    b. to be able to create a house account and invite other users.
    c. to be able to transfer the house administrative right to other roommates.
    d. to be able to recover my account if I forget my password.
    e. to be able to manage my personal account.
    f. to be able to edit my personal information.
    g. to be able to remove users that do not live with me anymore.
    h. to be able to access the website on my phone without trouble (if time allows).

Financial: Roommates will be able to manage their own financials. They will be able to perform multiple financial activities making it much easier for all the users.

1. As a user, I want
    a. to be able to pay or request money from my roommates
    b. to be able to keep track of how much my roommates owe me
    c. to be notified when my bills are due
    d. to be able to keep track of my monthly expenses
    e. to be able to see statistics about my financials
    f. to be able to integrate services such as Paypal or Google Wallet
2. As a user group, we want
    a. to be able to pay our bills by splitting up the total cost
    b. to be able to pool money together for group expenses

Inventory:

1. As a user group, we want
    a. to be able to have a common shopping list.
    b. to be able to maintain a list of our current food and supplies.
    c. to know how much our inventory is worth, based off of average prices of our items or the price we enter.
2. As a user, I want
    a. to be reminded of items that I commonly purchase.
    b. to be suggested cheaper alternatives to items on my shopping list. (if time allows)

Reminders and Events

1. As a user, I want
    a. all of my events synced to my calendar.
    b. to receive text, email, and push notifications of events on my calendar.
    c. to be able to send global notifications to my group.
    d. to receive text, email, and push notifications when a financial event happens.
    e. to be able to opt out of notifications.
    f. to be able to send messages to one of the house members.
2. As a user group, we want
    a. to have a common calendar with all of our schedules synced to it.
    b. to be able to receive common notifications about new events made to the common calendar.
    c. to be able to have a rotating schedule of chores.
    d. to be able to pin messages to a message board.
    e. to be able to set up a voting system to make group decisions.

General Priorities

The General Priorities of this project will mainly focus on ensuring the UI is user-friendly and a successful application by prioritizing the user experience. The main priorities of this application are usability, maintainability, performance and security.

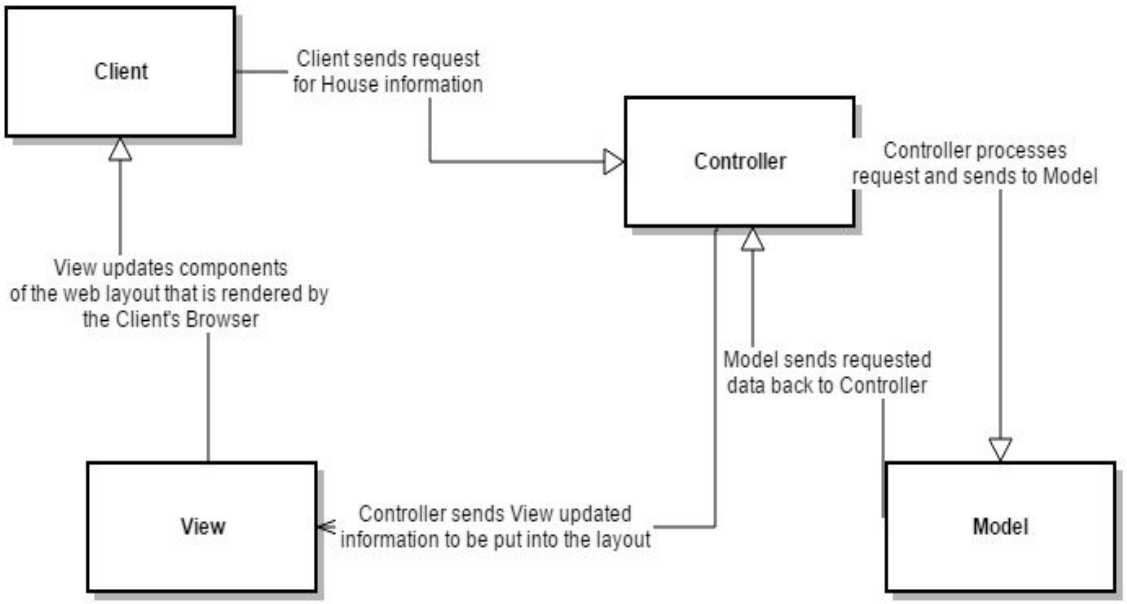
Usability
Our main priority is usability because until now, there has been no singular solution to help roommates gently remind each other and keep track of everything in their house. Our application provides a one-stop solution for roommates to check on each other and keep track of everything. The application will provide the user will all the necessary option that a proper roommate management system has to provide. The users will be able to perform all the necessary actions that will help them manage their house properly. The application will be utilizing HTML/Javascript/CSS in order to make it as interactive as possible.

Maintainability
Maintainability is also an important priority in our project. We will be mainly focusing on embracing the Model-View-Controller (MVC) pattern in our project. The majority of the website will be constructed using the Ruby on Rails Framework. Ruby on Rails itself operates using MVC principles, which we believe is the design pattern that aligns most with our needs. We want most of the processing and storage of data to be done server side, with a thin client that is only displaying the data pushed from the server.

Performance
Performance is important for any system to be functioning at its best. Our website ensures that all the information that has been updated by the user is being processed at a very fast rate. Whenever a user requests the server, the server retrieves the information from the database at a very fast rate. Our website refreshes instantly at a certain interval of time updating all the information and the changes made by the user.

Security
Security is one of the major priorities of the system because the consequences of security breaching is relatively high and it also contains information about banking and financial services. We will be also implementing a system that ensures the security of the personal information about the user. Account data such as passwords, contact and financial information will be encrypted using standard password-encryption methods.

Design Outline



Overview and Components

Our system will implement the Model-View-Controller design pattern, and implement the following components:

- Model
    - Our Model stores data modeled after our specific classes.
        - E.g. User data (Username, Password, Attributes), House data (Lists, Rooms, etc.)
    - Our Model allows creation and deletion of instances of these classes.
    - Data stored in the Model is updated via requests from our Controller.

- View
    - View interacts between Browser and Controller.
    - View contains the layout that is sent and rendered by the browser
    - View updates components of its layout via requests from the Controller.
- Controller
    - Controller receives requests from Clients.
    - Controller handles these requests to invoke procedures that retrieve data from the Model and modifies the View accordingly
- Client
    - Client renders data provided by the View.
    - Client provides interactivity of the View to the user.
    - Client sends requests to the Controller.

Design Issues

Issue I - Server Framework

While frameworks are not completely necessary in developing a web applications, they allow for an easier, cleaner, development processes and more structurally sound product.

1. Ruby on Rails
2. Python Web Framework (Flask or similar)
3. PHP Framework (Laravel)
4. Java (Spring, Vaadin)

We chose Ruby on Rails for our framework of choice for this project. While our group had various experiences with most of these technologies, we felt that Ruby on Rails provided the over all the best functionality and ease of learning for our group. Ruby on Rails also follows a MVC centric design which fits well with our design outline and architecture.

Issue II - On-Site Hardware

1. Use on site hardware for appliance integration
2. Do not use on site hardware and focus on online services

We chose to not use on site hardware. Due to our inexperience with hardware solutions and our limited capital, we decided it was not worth pursuing at this time. We do recognize using on site hardware would provide a more unique service and would provide a product that is better differentiated from competing services. While we choose not to follow this now, we would love to in the future with improved circumstances.

Issue III - Database Software/Architecture

1. SQLite
2. MySQL
3. PostgreSQL

We chose to use PostgreSQL for our project. These are all Relational Database Management Systems and for this project any of these solutions would work. SQLite is designed more for small services and embedded software. While SQLite can handle small websites, it does not have the concurrency support required for larger websites. Now MySQL and PostgreSQL are even more similar to each other than SQLite, both are designed to be scalable for web services and both integrate fine with Ruby. There are various small differences between MySQL and PostgreSQL with read and write speed, backups, ease of learning, and support of the SQL standard, but those difference are largely irrelevant for this project. We decided on using PostgreSQL for this project as it used with our current choice of webhost, Heroku. MySQL may easily be a better choice for a different webhost and we have not put it out of consideration.

Issue IV Whiteboard/Discussion/Note Integration Technology

1. Develop a service for sharing notes/discussions between tenants
2. Integrate with an external service, such as GroupMe

We chose to develop our service for sharing notes between tenants to keep our messaging system cleaner. By not relying on another service, we can develop our system to have level of functionality that we might not be able to achieve using a service like GroupMe. For example, if we were to use GroupMe, we would not be able to allow users to be able to edit notes, and erase previous versions of them. If a note contained an embarrassing typo, the message would be stuck on GroupMe's servers, readable by other members of the group forever.

Issue V How will expenses of the house, be divided between Tenants?

1. Expenses are always split evenly between every tenant.
2. Tenants specify the splits with every expense.
3. By default, the split is even, but with an option to specify which tenants are involved in the expense, and how much each tenant pays.

We chose to have the most customizable bill option. Bills will be able to be setup to be associated with different users. Each user can be set to pay a different percentage of the total expense. By default, the total will be split evenly for each user associated with it, however there will be functionality to allow users to set custom percentages to accommodate all payment plans. Using this method will provide users the most utility and usability.

Issue VI Securing User Data

1. Plaintext
2. Simple Hashing (MD5, SHA-1)
3. Cryptographic hashing algorithm (SHA-256, PBKDF2)

This wasn't as much as an issue of *should* we secure user's data, but how and how to do it properly. Plaintext has the advantage of being the fastest and easiest to implement, but it is also the least secure and the most embarassing as Sony has found out multiple times[1]. Simply hashing the passwords is also ineffective if the database is compromised. Hashes can be precomputed into lookup tables or Rainbow tables that can make simple cracking of your stored passwords if the hashing is known. Luckily, this is a well studied problem in the industry and there are many resources for learning that cryptographic hashing algorithms are the standard in the industry, and would be very feasible for implementation in this project.

[1] http://arstechnica.com/tech-policy/2011/06/sony-hacked-yet-again-plaintext-passwords-posted/

Issue VII How will user's data and house data be viewed?

1. Have separate pages for each aspect the user can view.
2. One central hub for the user to view everything at.

We decided for a mixture of these two extremities. While we do want a dashboard to view most of the details at a glance, we don't want it to be over cluttered. Our current plan to have a dashboard view with the important information displayed, while smaller details, such as an user's account details, to be accessed on a different page.

Issue VIII How will the House be administered?

1. House will be administered, democratically. A vote is required to remove members, delete the house, etc.
2. House will be administered by a single administrator of the house. By default this is the creator of the house.

We decided to have the house be administered by a single administrator of the house. In addition to its simpler implementation, it will not allow a single user to cause problems among the house setting. We also plan to have the administrator title be able to transferred at the current administrators discretion.

Issue IX Financial Service Functionality

1. Dwolla
2. Synapse Pay
3. Venmo
4. Paypal
5. Amazon Payments
6. Square Connect

This design issue hasn't been completely resolved, this is to do with our inexperience in this area. We can narrow down our field of selection though. Paypal, Amazon Payments, and Square Connect, are designed for businesses with direct products and not designed for sending payments to people besides the vendor. Dwolla and Synapse Pay are directly competing services and offer a similar suite of services, both are designed for bank account and debit card payments. Venmo is a person to person payment service and support most payments choices, including bank accounts, debit cards, and credit cards. Dwolla and Synapse Pay use flat costs on their transactions instead of a percentage based that Venmo provides. This is important as we have planned to allow users to pay rent through our service, which a percentage cost is much more costly than a flat cost. A common trend is to is use both Dwolla and Venmo for services, which covers most if not all of the functionality we want in our application. This is probably the route we will take.

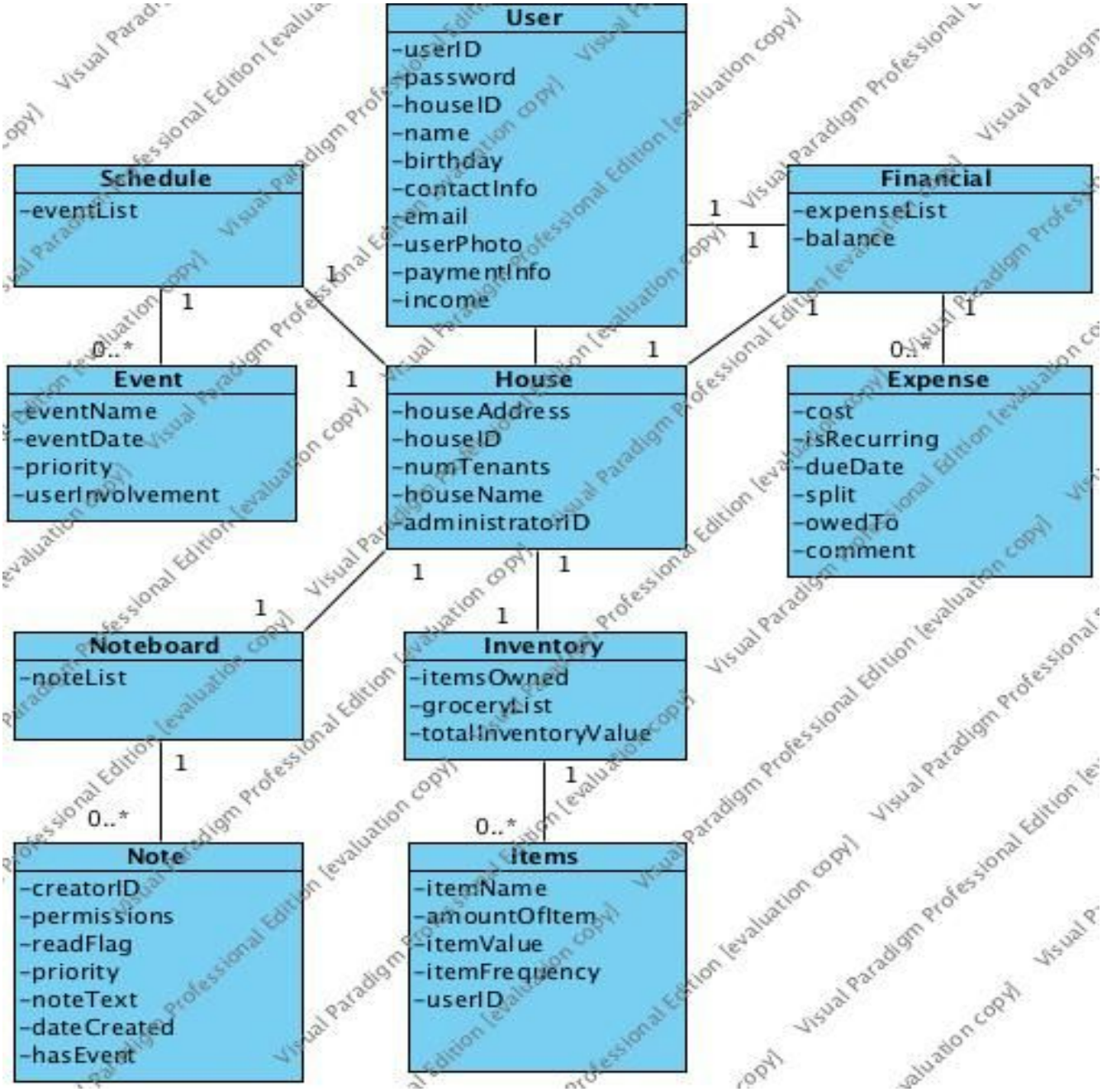
Issue X How will reminders be pushed to the users?

1. GroupMe
2. Text Message (Twilio)
3. Email Reminders
4. Browser Notifications

We chose to have a combination of browser notifications and emails.. As accompanied with issue IV, we are concerned with the viability of using GroupMe in the project, compared to when we considered it. Text messaging is very appealing to use at the moment, no external apps are required, like GroupMe) and accessible to older phones as well. Browser notifications are our default option for this. Real time updates pushed are almost standard for these types of applications, and we decided early on that having these would be important. Email reminders is also an option, but too many emails can be considered spammy. So we having a mixture of the two is for the best, and having a the user being able to customize their notification settings is a strong possibility. Adding text messaging in the future is a strong possibility too, due to its universality.

Design Details

Class Diagram



Description of Classes:

User:

- The user class encapsulates all data needed to classify a user such as unique ID, username, password, and other contact/personal info.
- User stores pertinent information to the app like payment info and income.
- Users will belong to a House object.
- Users will have a Financial object to help monitor their expenses.

House:

- The House class will have all of the information about a current household like ID, address, number of tenants, house name, and adminID.
- A House will have User objects based on the number of tenants living in the house.
- The House will have a schedule object for keeping track of the common house schedule.
- The House will have an Inventory object to monitor the collective house inventory.
- The House will have a Financial object to manage Household expenses (i.e. utilities)
- The House will have a Noteboard object to allow users to leave notes for the whole house.

Financial:

- A Financial object is meant to be used by Users and Houses to keep a list of expenses as well as the owner's current balance.
- A Financial object will own a list of Expense objects.

Expense:

- An Expense object contains information that is common for different types of expenses like cost, whether or not is it a recurring expense, the due date, if the expense is split & how it is split, who the expense is owed to, and a comment about the expense.

Schedule:

- A Schedule object serves as a manager for a list of Events.
- A Schedule will be owned by a House to allow all the Users to know the events of the House.

Event:

- An Event object will encapsulate information pertaining to an event like name, date, priority, and who will be involved.
- Event priority will determine whether or not we integrate it into specific users calendars and send them email alerts about the event.

Noteboard:

- A Noteboard will belong to a House and will serve as the manager for a number of Note objects.
- The Noteboard will display the Notes prominently on the House dashboard.

Note:

- A Note is an object that can be created by a User and added to the House Noteboard.
- The Note will store information about the note in the form of a creator ID, permissions on who can view the note, a flag to signal if it has been read, the priority of the note, the message of the note, the date it was created, and whether the note has an event associated with it.

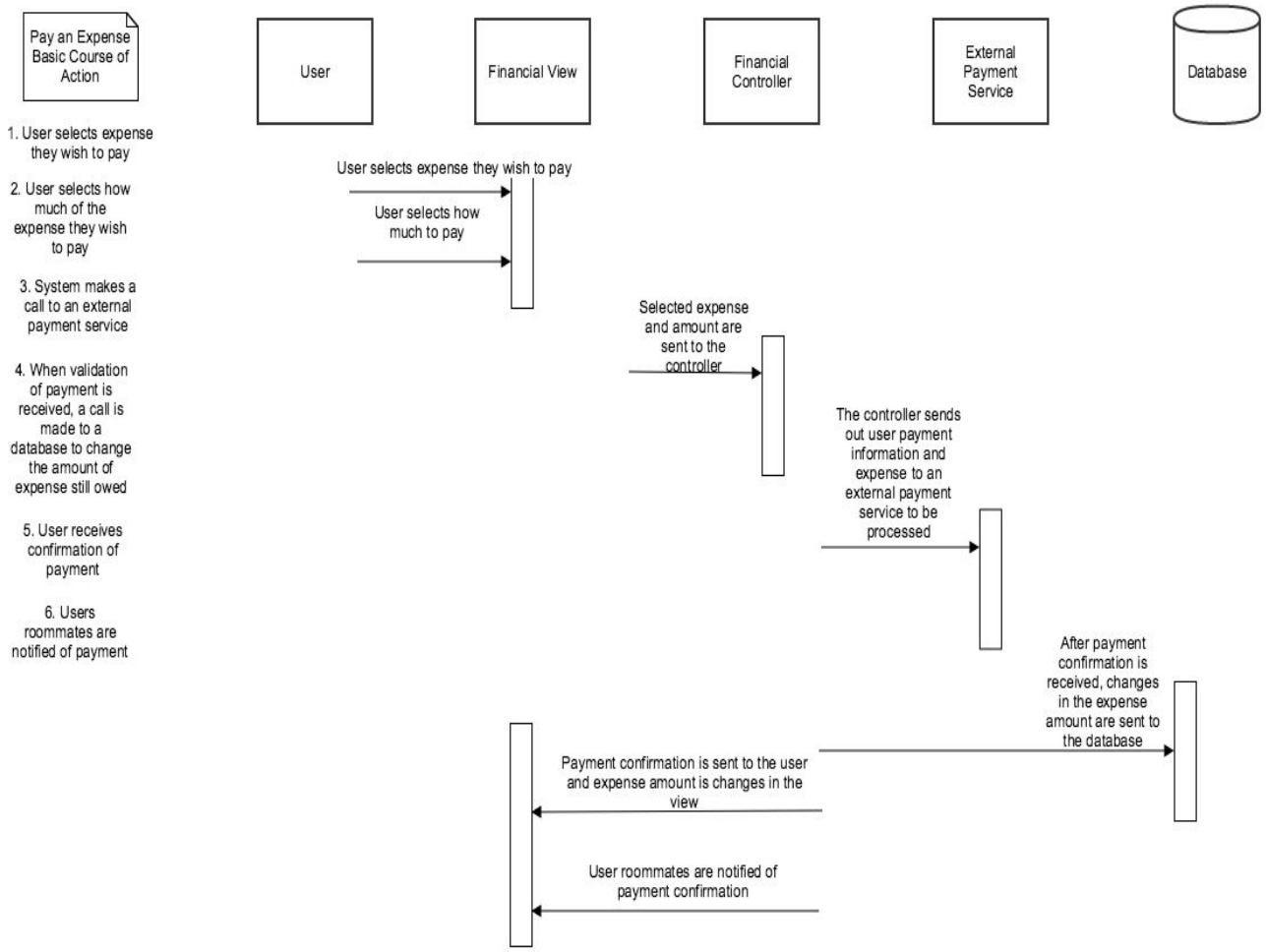
Inventory:

- The Inventory object will serve as a manager for the House inventory by maintaining a list of Item objects.
- Inventory stores basic information such as the items owned, the grocery list (items needed), and the total value of the inventory.

Item:

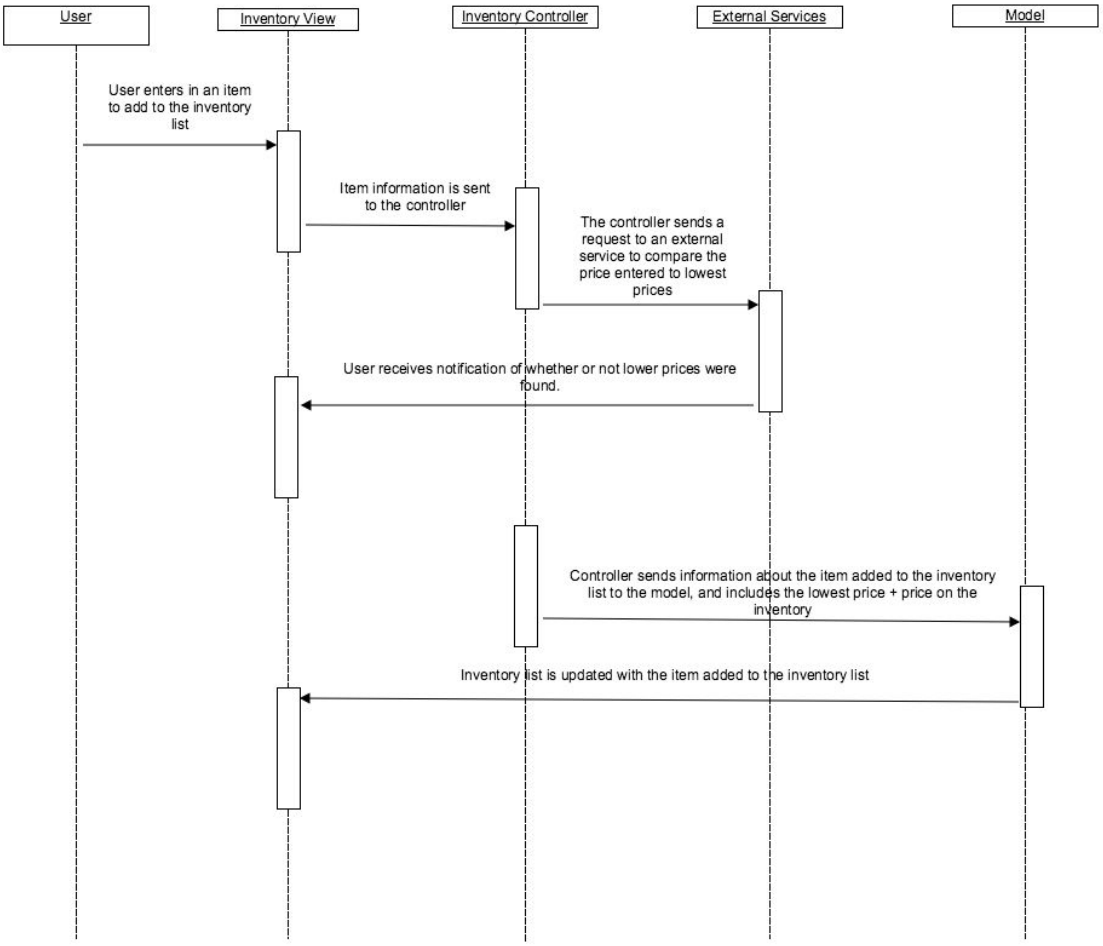
- An Item object stores information about an item that is either owned or needed by the House.
- Items include information like name, amount owned/needed, cost, frequency of purchase, and the ID of the user who owns/needs it.

Paying an Expense Sequence Diagram



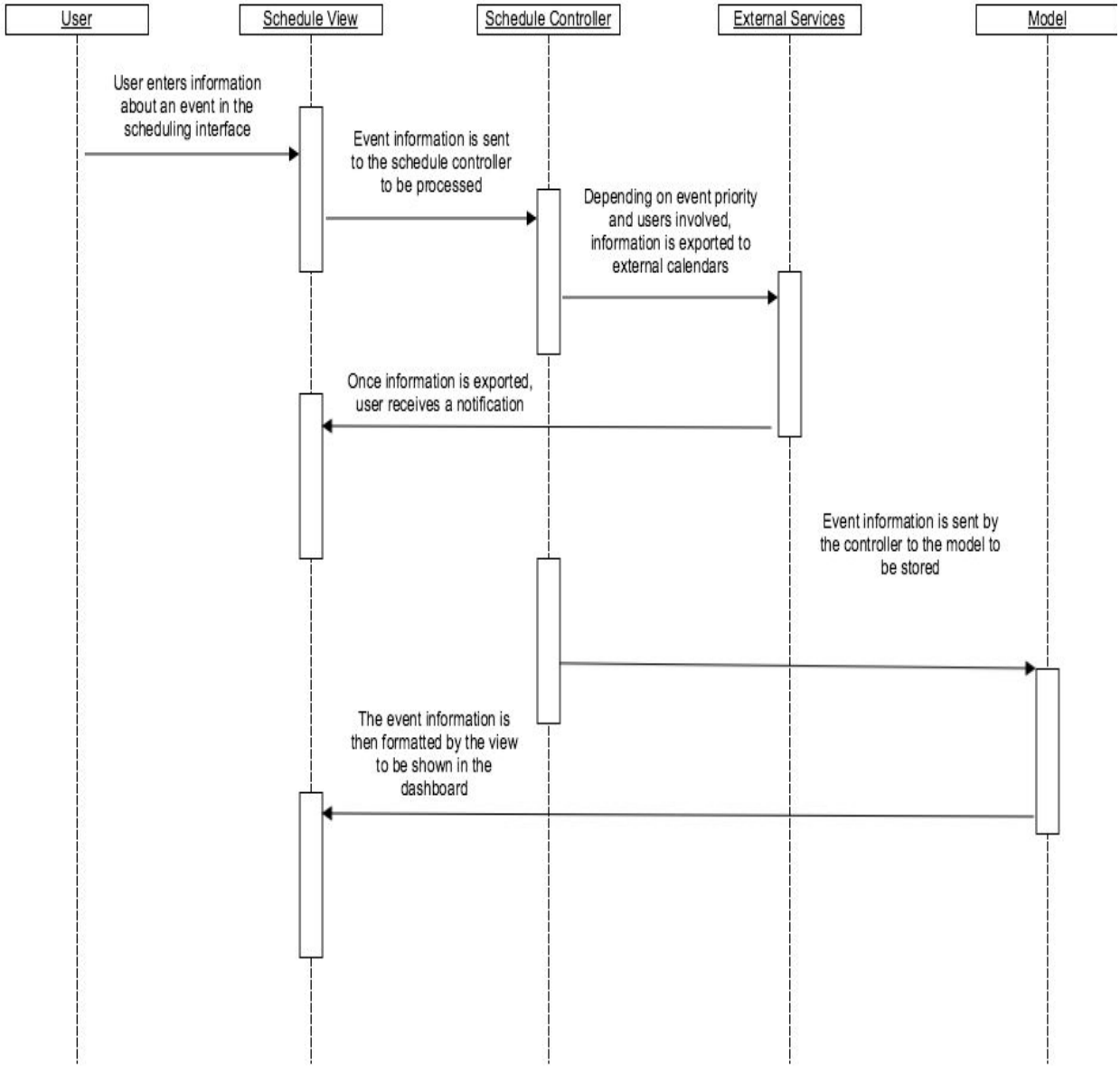
One of the major ways that our solution to roommate management competes with existing solutions is that we allow users to take all pertinent actions within the web application itself. Roomedy will be the only solution that allows users to pay their expenses in the dashboard instead of just organizing how payments are made. To make a payment a user will chose the existing expense to be paid from the view and then how much of what they owe they want to pay at the time. This information is sent to the controller, which then sends out a request to an external payment service with the user's payment information and the information about the expense to be paid and waits for confirmation of the payment. Once confirmation is received the changes to the expense are sent out to the controller which then updates the expense information in the model. After the model is updated, both the user and their roommates receive a notification of an expense payment and the view is updated to show the new expense total.

Adding an Item to Inventory List Sequence Diagram



Our app will allow users to keep both a list of current items stocked and a grocery list of items to buy. To add an item to the inventory list, users will enter the name, amount, and price of the item in the inventory view. This information is then sent to the controller, which sends a request out to an external service to compare the price entered to check and see if cheaper options are available. If a cheaper option is found, users are notified of this and then the controller sends the information about the item added to the inventory list to the model. This information that is sent includes whether or not a cheaper option was found, and this information is used to give users a weekly report on how to more efficiently spend their money. Once the model is updated, the inventory view is then updated to correctly show the user entered item.

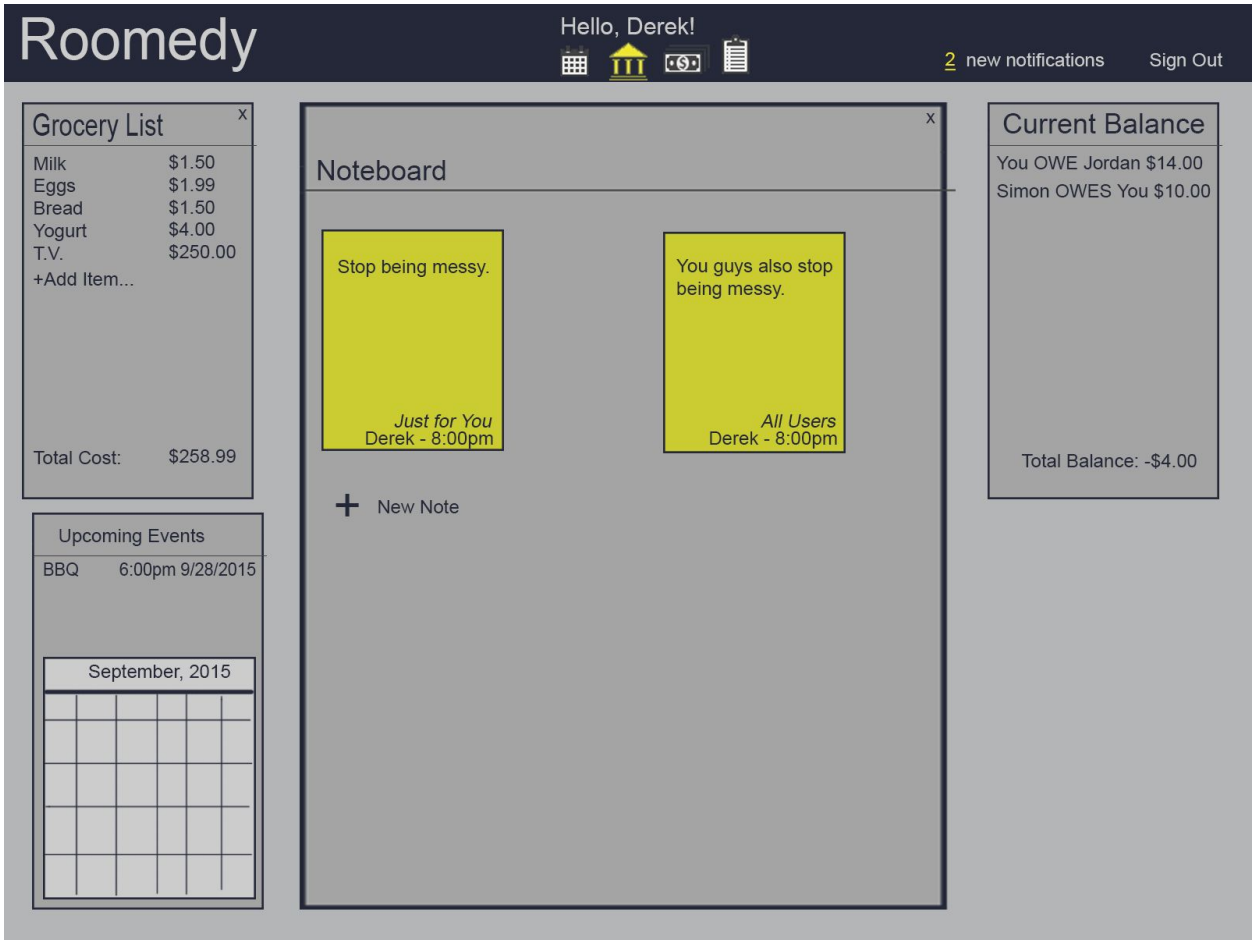
Scheduling an Event Sequence Diagram



We include a scheduling and calendar service within our app to keep roommates aware of each other's schedules and to help them coordinate house related events with each other. The process to schedule an event in the tool starts with the user entering the information for the event in the Schedule View. They enter the data, time, description of the event, a priority of the event, and the other users involved. This information is then sent to the controller, which processes the data entered. If the event has a priority that requires the attention of all other users or if other users are involved, it then exports the data to the calendars of the other users involved. This will allow the

users to automatically have priority events scheduled into their Outlook or Google Calendars. The user scheduling the event will then receive a notification in the dashboard confirming the event has been exported to other's calendars. After that, the event information is sent to the model by the controller, which allows the Schedule View in the dashboard to be updated accordingly.

UI Mockup



This UI Mockup gives a rough idea of what the User Experience using the Dashboard will be like. The final version will be a modified Bootstrap Carousel-style page. Users will be able to accomplish the most common tasks from their home page. When they require more specific details or options for a certain category, they will use the Carousel-style navigation at the top of the page, to choose, for example, the page dedicated to all of their financial information.

The Dashboard itself is created by the View object, and is populated by the data given to it from the Controller. The Dashboard is then rendered in the Client's web browser. The data the Dashboard renders will be based off the information that is available to the user. The Dashboard also provides the buttons that will allow Users to send/receive updated information.