

Dispatchr: Project Requirements Document

By: Team Savage

Members:

Alok Gupta (Lead)
Jordan Nguyen (Scribe)
Sal Olivares
Spencer Prescott
Brian Yan

Mentors:

Lu Jin
Daniel Vicory
Instructors:
Chandra Krintz
Ankita Singh

Introduction	2
Innovation and Science	2
Project Specifics	2
Team Goals	3
Background	3
Assumptions	3
System Architecture Overview	4
High Level Diagram	4
User Interaction and Design	6
Glossary	13
Requirements	13
User Stories	13
Prototyping code, tests, metrics	17
System Models	17
UML Class Diagrams	17
Sequence Diagrams	19
Fetch Requests From API	19
Add New Items to Request	20
Create New Request	21
Appendices	22
Technologies Employed	22
Technical Definitions	22

Introduction

For many college students who do not have cars, it can be challenging to go the store to get groceries, supplies, toiletries, etc. For others, it can be a struggle to make time to go shopping. Dispatchr is the solution to both these problems and more. Dispatchr is a community-centered platform that allows users to help each other when it comes to shopping. Users can post requests for what items they want to buy from what store, and other users can accept these requests and buy those items. Users can also post which store they will be shopping at, and other users can request products from that store they'd like picked up.

Innovation and Science

While there are various applications and services already existent that provide similar functionality to Dispatchr, there are several components that set it apart from the competition. Dispatchr's first competitive advantage is the ability to get any item picked up, regardless of whether it's groceries or school supplies. In contrast, other services only provide items of a single category, most commonly food from local restaurants, to be picked up.

The second competitive advantage of Dispatchr is the community aspect, particularly within college communities. In other services, products are picked up by random "contractors." In Dispatchr, items are picked up by local students that reside in the same community as those requesting items for pick up. Through this, students are able to help serve others within their community and do social good.

Project Specifics

This project is for primarily college students. In particular, it's a major pain point today for many to retrieve groceries or other items from local stores, especially since many college students don't have cars. On the other hand, other college students that have a car and are able to visit stores regularly might be willing to pick up a few extra items for their peers, especially if they are compensated. As college students who can empathize with this dire issue in our community, we wanted to build a solution.

Dispatchr provides a two-sided platform for college students. The students who want items can post the item(s) they need, as well as any further information such as when they would like it by, if they have a store preference, as well as distance to actually pick up the item. On the other side, the "dispatcher" can view items from stores they are considering visiting and can opt to pick it up for their peer. In addition, when they visit stores that others are requesting items from,

they will receive a push notification alerting them that they can pick up certain Items for their peers since they are there, and that they will be compensated accordingly. We will also include gamification in the form of trophies that are awarded to users as they continue to help their community peers, further incentivizing them to pick up items for others when they are shopping.

Team Goals

By the end of this project we plan to learn the scrum and software engineering process by building and deploying our application from scratch. Our main goal is to solve a major problem for college students around the world by continuously iterating and improving the product we are building. More specifically, we want to better learn and understand the software development lifecycle. We will be learning how to build an application from scratch, which includes building the client side mobile applications in React-Native, an API with Ruby on Rails using PostgreSQL for the database, as well as Redis for caching requests. In addition, we will learn deployment processes using Heroku. Throughout our software development lifecycle, we will be practicing test-driven development in order to build a robust and polished application as well as learn about the best software development practices.

Background

Dispatchr is an iOS and Android application geared primarily for college students. In particular, it's an inconvenience for students to retrieve groceries or other items from local stores, especially since many college students don't have cars to do so. On the other hand, other college students that have a car and are able to visit stores regularly might be willing to pick up a few extra items for their peers, especially if they are compensated. Dispatchr resolves both of these issues for college students across the world for both iOS and Android.

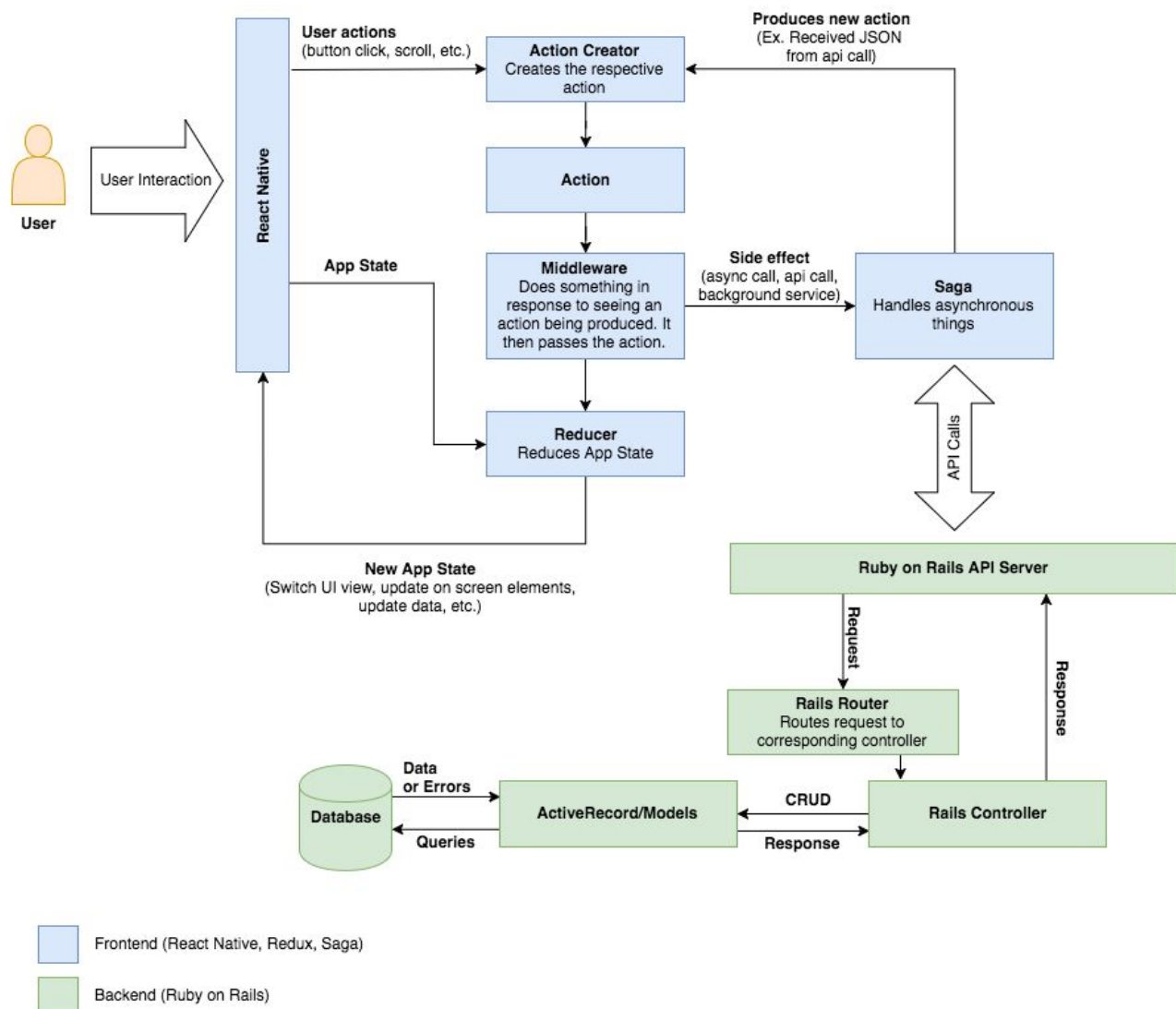
Assumptions

There are various assumptions that we are making while building this product. First, we're assuming that college students are willing to pay \$5-\$10 to have their items picked up, and that the students who are picking the items up are willing to accept this amount as well. We need to conduct further validation from students in Isla Vista to determine the final price for this by potentially creating a supply and demand curve. In addition, we are assuming that students will be willing to pick up items for others in the first place, although we're hoping that the compensation will incentivize them to do so for their peers, as well as the gamification.

System Architecture Overview

High Level Diagram

Our mobile application consists of our frontend built with React-Native and server-side code consisting of Ruby on Rails that serves as our backend for this project. The following is the diagram overviewing this architecture:



The frontend mobile application is built with React.js, React-Native, Redux, and Redux-Saga. This client-side application serves as the interface for the user to interact with the system. When the user takes any action in the app (button click, scroll, field submission), it is routed through the action creator which creates the respective action. The action is then executed, through the

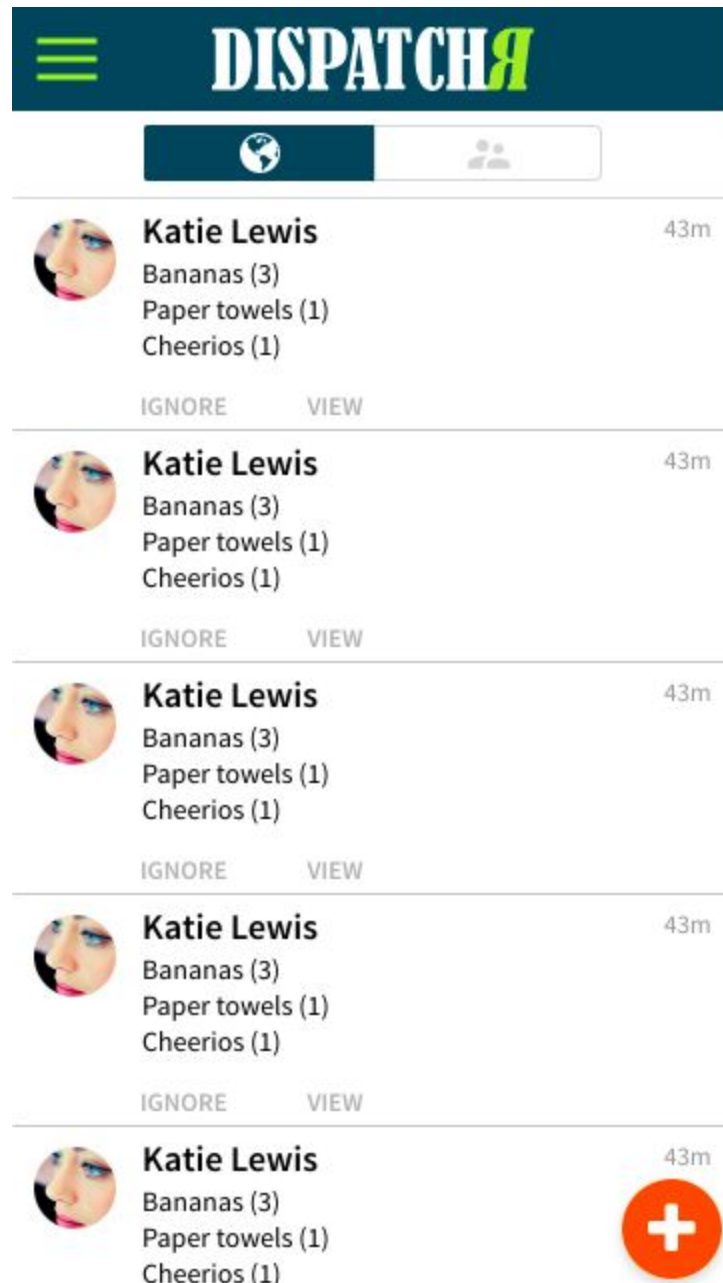
help of any middleware required during this process. The middleware, through Redux and Redux-Saga, also helps manage any side effects, such as asynchronous calls, API calls, background service, or cron job, with Saga. Once the middleware processes are completed, the Reducer helps manage and reduce the app state. The new app state is then pushed to the React-Native view for the user. The advantage of React is that it automatically calculates the fastest Document Object Model (DOM) transactions required to update the DOM tree for the user. This results in high performance for the user and ultimately, high user satisfaction/experience.

As of now, we are not making any third party API calls, and only use our Rails built API server. When API calls are required, the middleware passes the task off to Saga which then makes the appropriate API calls. The custom Dispatchr API is accessible via the cloud due to Heroku Platform-As-A-Service. It's built using Ruby on Rails along with PostgreSQL and Redis. Rails is a framework for Ruby that enables rapid, full-stack web development. PostgreSQL serves as a relational, persistent data-store used to help serve data to the client. When a request is made, Rails router helps determine which controller to serve the request to. Based on the context of the request, it's passed to the appropriate controller, which then usually makes certain database transactions in order to retrieve the required data. The ActiveRecord Object Relational Model (ORM) makes it seamless to interface with the PostgreSQL database, which then returns the desired data. Generally, some logic is applied to this data before it is successfully returned as a JSON response to the client-side application, which is what the consumer is interfacing with.



This is a high-level overview of our system architecture. We will be building a client applications for both iOS and Android, using React-Native. These clients will be powered with our Ruby on Rails API which will serve to persist data and conduct our backend logic. This RESTful API will use PostgreSQL for persisting the data, as well as Redis for caching particular requests, and return responses in JSON format that our client applications will interface with for use by our end users.

User Interaction and Design

Our wireframe contains a mockup of our application with regards to user interaction and design. The entire wireframe can be viewed [here](#). The following are the images of our wireframe:




Screen 1: Global Request View

New Request

CANCEL

Date neededNov. 18, 2016X>

 Enter your item

How many? - 1 +

Date neededNov. 18, 2016 >



SAVE

SAVE + ADD

Screen 2: Submitting a new Request

Screen 3: Calendar Picker for New Request

Screen 3: Calendar Picker for New Request









 

My Request

CANCEL


Date needed

Nov. 18, 2016 >



3 bananas		
2 apples		
1 paper towel roll		
1 tomato soup		


Publish my request


Screen 4: Viewing current request



DISPATCHЯ




**Your request has been published!**

**Katie Lewis**

43m

Bananas (3)
Paper towels (1)
Cheerios (1)


IGNOREVIEW

**Katie Lewis**

43m

Bananas (3)
Paper towels (1)
Cheerios (1)


IGNOREVIEW

**Katie Lewis**

43m

Bananas (3)
Paper towels (1)
Cheerios (1)


IGNOREVIEW

**Katie Lewis**

43m

Bananas (3)
Paper towels (1)
Cheerios (1)


IGNOREVIEW

**Katie Lewis**


43m

Bananas (3)
Paper towels (1)
Cheerios (1)

IGNOREVIEW




Screen 5: Request successfully submitted



Open Request


CANCEL




Katie Lewis


Date needed: Nov. 18, 2016


43m







3 bananas







2 apples







1 paper towel roll

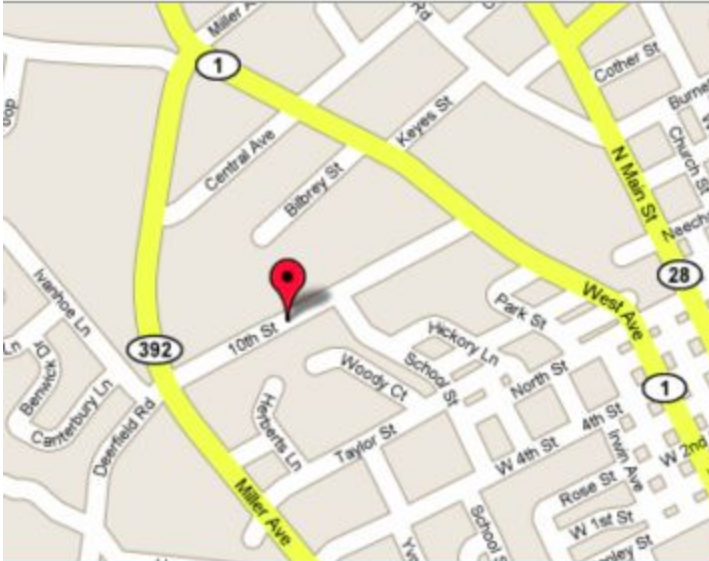




1 tomato soup

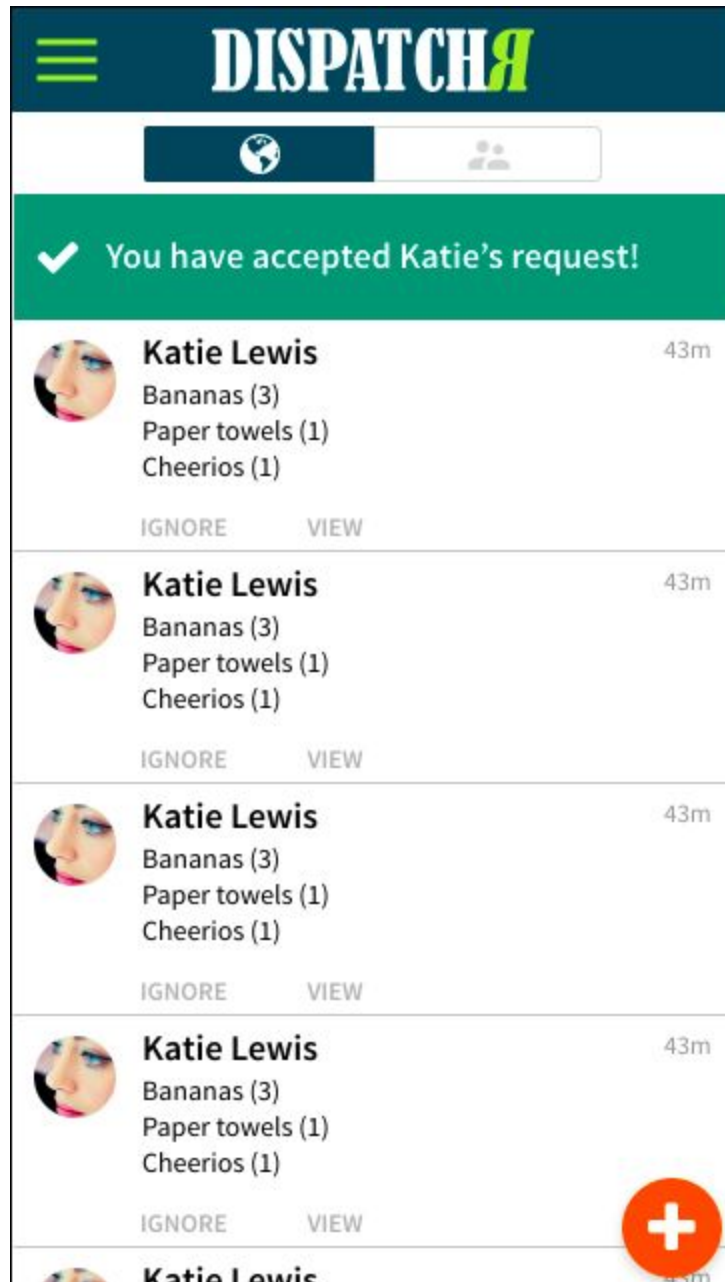






Accept request

Screen 6: Request Detail View



Screen 7: Accepting an existing request

In our wireframe, you can see our user interaction by pressing the available actions on each page. First, you are presented to a GlobalRequest View (Screen 1) which shows the available requests that users have submitted. Here, you have two options. You may press “View”, and you are brought to a more in-depth view (Screen 6) of a single request, where requests can be accepted (Screen 7). Otherwise, you can press the “plus button” in order to add a request. In this case, you are brought to the add request view (Screen 2). Here, you can select the date needed by, or select Save. By pressing date, you are brought to a date picker (Screen 3). Otherwise, you can select Save and you are presented with a confirmation view of your request

(Screen 4). At this view, you can select “publish request” and you will be brought back to the original GlobalRequest View with a confirmation stating that your request has been published (Screen 5).

Glossary

1. Post - a submission from a user that comes in two forms
 - a. Request - users post what items they want picked up from which store
 - b. Buying - users post they are at the store and are available to buy and deliver items
2. User - anyone using the application
 - a. Requester - a user who posts requests
 - b. Buyer - a user who buys and delivers items for requesters
3. Item - something that can be bought and delivered at a store
4. Feed - a list of posts in reverse chronological order that can be filtered. There is a feed for Requesters and Buyers
5. Profile - user profile linked to basic information
6. Rating - user rating based on previous user interactions
7. Dispatcher - buyers who purchase items for requesters

Requirements

User Stories

Note: Prototyping code URLs link to our Trello user stories, which have our corresponding GitHub commits directly linked to them.

1. As a requester, I can input the items I want so that I can submit a request. See prototyping code [here](#), [here](#), and [here](#).
 - a. Precondition: Requester wants to submit a request of items to be picked up.
 - b. Event: Requester can input an item they would like.
 - c. Postcondition: Requester is able to select exactly the items they would like.
2. As a requester, I can submit a request of multiple items that I want picked up so other users can fulfill my request. See prototyping code [here](#), [here](#), [here](#), and [here](#).
 - a. Precondition: Requestor has an idea of what they need to get from the grocery store.
 - b. Event: Requestor can enter all the items they need on an easy to use form in the app.
 - c. Postcondition: Requestor is shown their list with all the items they added before saving it.

3. As a requester, I can select where I want my items to be delivered so that I can pick up my request. See prototyping code [here](#), [here](#), and [here](#).
 - a. Precondition: Requestor has a list of items that he or she needs to be delivered.
 - b. Event: Requestor enters a location where they will be to pick up their items.
 - c. Postcondition: The requester's items have a delivery location so buyers know where to drop of the items.
4. As a requester, I can specify a price range for my request so a buyer does not buy an item I cannot afford. See prototyping code [here](#) and [here](#).
 - a. Precondition: Requester has a list of items they need to be picked up.
 - b. Event: Requester specifies for each item, the price range they are willing to pay for that item.
 - c. Postcondition: Each item has a max price so the buyer does not buy an item the requester cannot afford.
5. As a requester, I can select when I want to pick up my items for convenience. See prototyping code [here](#) and [here](#).
 - a. Precondition: Requester has a list of items they need to be picked up.
 - b. Event: Requester specifies when they can pick up their items.
 - c. Postcondition: Each item has a delivery location so the buyer knows where to drop of the item.
6. As a user, I want to be able to see a list of requested items so that I can see my orders as well as those of my community. See prototyping code [here](#), [here](#), and [here](#).
 - a. Precondition: Requester wants to see the global list of items that are requested by people.
 - b. Event: Requester can view the list of items people would like.
 - c. Postcondition: Requester can see the items his community has ordered as well as his request amongst theirs.
7. As a user, I want to easily switch feeds between global requested items and my requested items so that I can choose what role to participate as. See prototyping code [here](#).
 - a. Precondition: User wants to change which view they are looking at.
 - b. Event: User selects the feed they wish to see at the top of the screen.
 - c. Postcondition: The feed the user selected will be displayed on their screen.
8. As a buyer, I want to be able to see a more in-depth view of a request so I see information such as location, items, price, and quantity. See prototyping code [here](#).
 - a. Precondition: Buyer sees a list of requests and wants to select one request for more information
 - b. Event: Buyer selects the request he wants to see more information about.
 - c. Postcondition: Buyer is brought to a screen including a more in-depth view of a single request
9. As a user, I want to be able to sign up/ log into my account so that I can see my current requests. See prototyping code [here](#) and [here](#).
 - a. Precondition: User enters the application and is provided a signup/login screen.
 - b. Event: User enters in credentials such as username, password, email

- c. Postcondition: User's account will either be created or logged in.
10. As a requester, I would like to select the items I want from a list of potential items, so that I don't have to manually enter them. See prototyping code [here](#) and [here](#)
- a. Precondition: Requester wants to request several items.
 - b. Event: When creating a new request, there's a lot of selection of items to choose from.
 - c. Postcondition: The requester can quickly select the items they need.
11. As a buyer, I want to be able to easily filter the posts, to be able to more easily purchase items for others.
- a. Precondition: Buyer is at the grocery store looking for items to buy.
 - b. Event: Buyer filters for items he or she is alright with buying.
 - c. Postcondition: Buyer is shown items that requesters have added for pickup that also match the filters supplied.
12. As a requester, I will be able to cancel my request before someone accepts it so that I do not receive items I no longer need.
- a. Precondition: Requester decides they no longer need the items they requested.
 - b. Event: Requester goes to their posting and deletes it.
 - c. Postcondition: The requester's post has been deleted from the feed and is no longer accessible.
13. As a requester, I want to be able to receive notification updates regarding my order
- a. Precondition: Requester has an order that has been accepted by a buyer
 - b. Event: Buyer selects a notification he wants to send to his requester
 - c. Postcondition: Requester receives notification update regarding the order
14. As a buyer, I want to be able to receive notifications displaying related requests I can accept when I am near a store
- a. Precondition: Buyer is at a store and is shopping for items
 - b. Event: Buyer receives notifications displaying requests he could possibly fulfill
 - c. Postcondition: Buyer will be able to select which requests he wants to accept
15. As a buyer, I want to be able to see a pin showing the location of where to deliver the item
- a. Precondition: Buyer wants to see the location of where he needs to deliver the items to.
 - b. Event: Buyer selects a single request and is brought to a map-view of the single request
 - c. Postcondition: Buyer will be able to see a pin displaying the location of where to deliver the items to
16. As a user, I want to give a rating to those I interact with so that others will know the quality of people they are dealing with.
- a. Precondition: Two users have finished their interaction with each other, and now have the option to give each other ratings through the application.
 - b. Event: Either user gives a rating for the other user.
 - c. Postcondition: User ratings are updated on their profile and can be viewed by anyone using the application.

17. As a user, I can upload a picture of myself so others know what I look like when I deliver or pick up items.
 - a. Precondition: User is on their profile settings and wants to upload their picture.
 - b. Event: User selects to upload a picture via camera roll or through taking a new picture.
 - c. Postcondition: User's profile has a new picture associated with it so other users can see it.
18. As a user, I want to be able to chat with other users that I have an active request with
 - a. Precondition: A buyer has accepted a requester's request
 - b. Event: Buyer or Requester wants to be able to chat with one another
 - c. Postcondition: Buyer or Requester will be able to send a message to the other.
19. As a user, I want to be able to report other posts to take down illegal or inappropriate material
 - a. Precondition: A user sees an inappropriate usage of the application
 - b. Event: The user reports the request as inappropriate
 - c. Postcondition: User reported request will be flagged and undergo further review.
20. As a requester, I want to see the average price of an item I am requesting
 - a. Precondition: Requester wants to input a price of the item he wished to request
 - b. Event: Requester selects "See average price" for the price of an item
 - c. Postcondition: The average price of the item is displayed to the requester
21. As a requester, I want to see the approximate cost of delivery for my item(s).
 - a. Precondition: Requester has finished selecting the items for his/her request
 - b. Event: Requester requests for the approximate cost of delivery for his items.
 - c. Postcondition: Requester will be able to see the approximate cost of delivery for his request
22. As a buyer, I want to see which stores I will need to visit in order to pick up the items in a request.
 - a. Precondition: Buyer sees a list of items that needs to be picked up
 - b. Event: Buyer will be able to click on an item to see which store it can be picked up from
 - c. Postcondition: Buyer will be able to see which items can be picked up certain stores
23. As a buyer, I want to be able to receive money before fulfilling a request so that I will not be scammed.
 - a. Precondition: Buyer agrees to pick up items for a requester.
 - b. Event: The requester is presented with a list of payment options available by the buyer, and selects one.
 - c. Postcondition: Buyer receives a notification and confirmation of payment.

Prototyping code, tests, metrics

We attached this section in the above section. Please visit the links that are attached to the user stories.

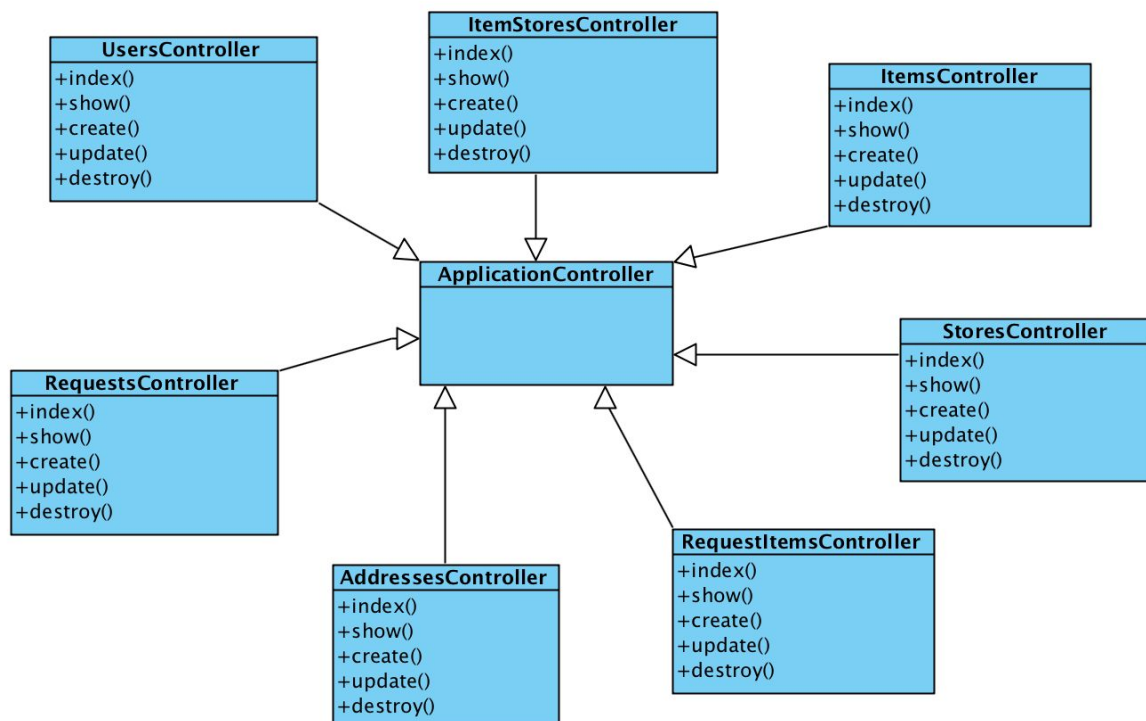
Trello contains our prototyping code, tests, and metrics for the user stories we have implemented thus far. In addition, we have included documentation for the front end components that do not have tests [here](https://trello.com/b/ByUTw9tb/dispatchr). You can see our commits for each story are displayed in the Trello Cards.

<https://trello.com/b/ByUTw9tb/dispatchr>

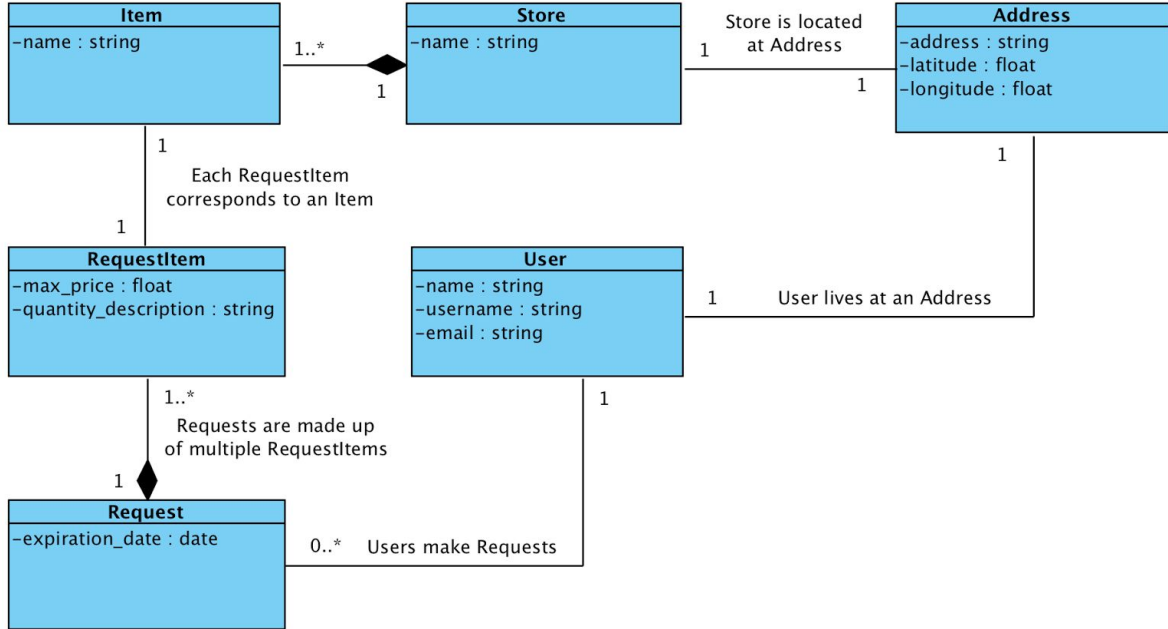
<https://github.com/brianyan/Dispatchr>

System Models

UML Class Diagrams



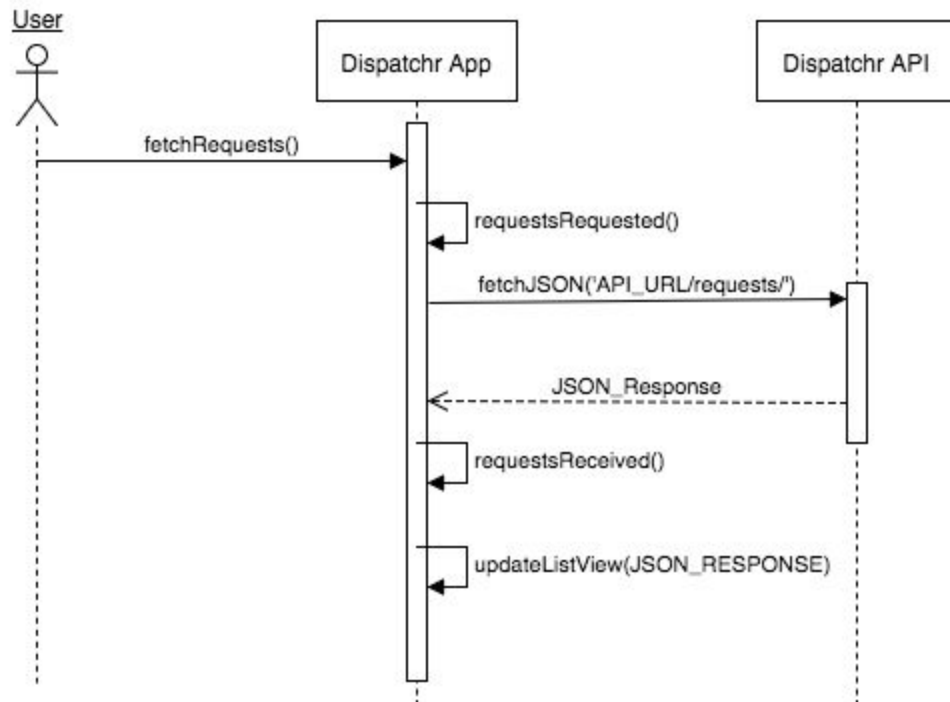
API Controller Classes all inherit from ApplicationController and implement CRUD functionality.



This diagram shows the relationships between all API model classes.

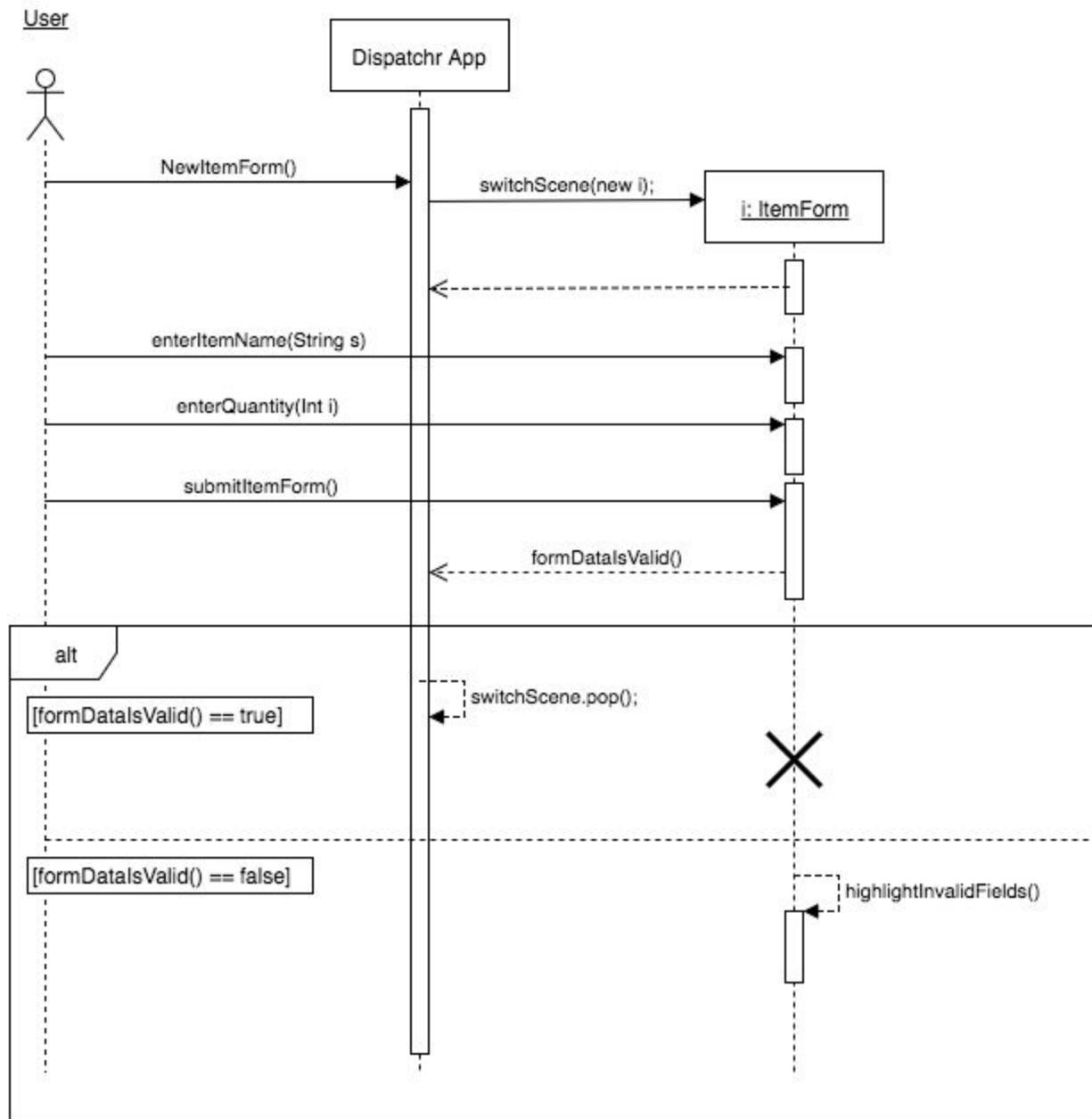
Sequence Diagrams

Fetch Requests From API



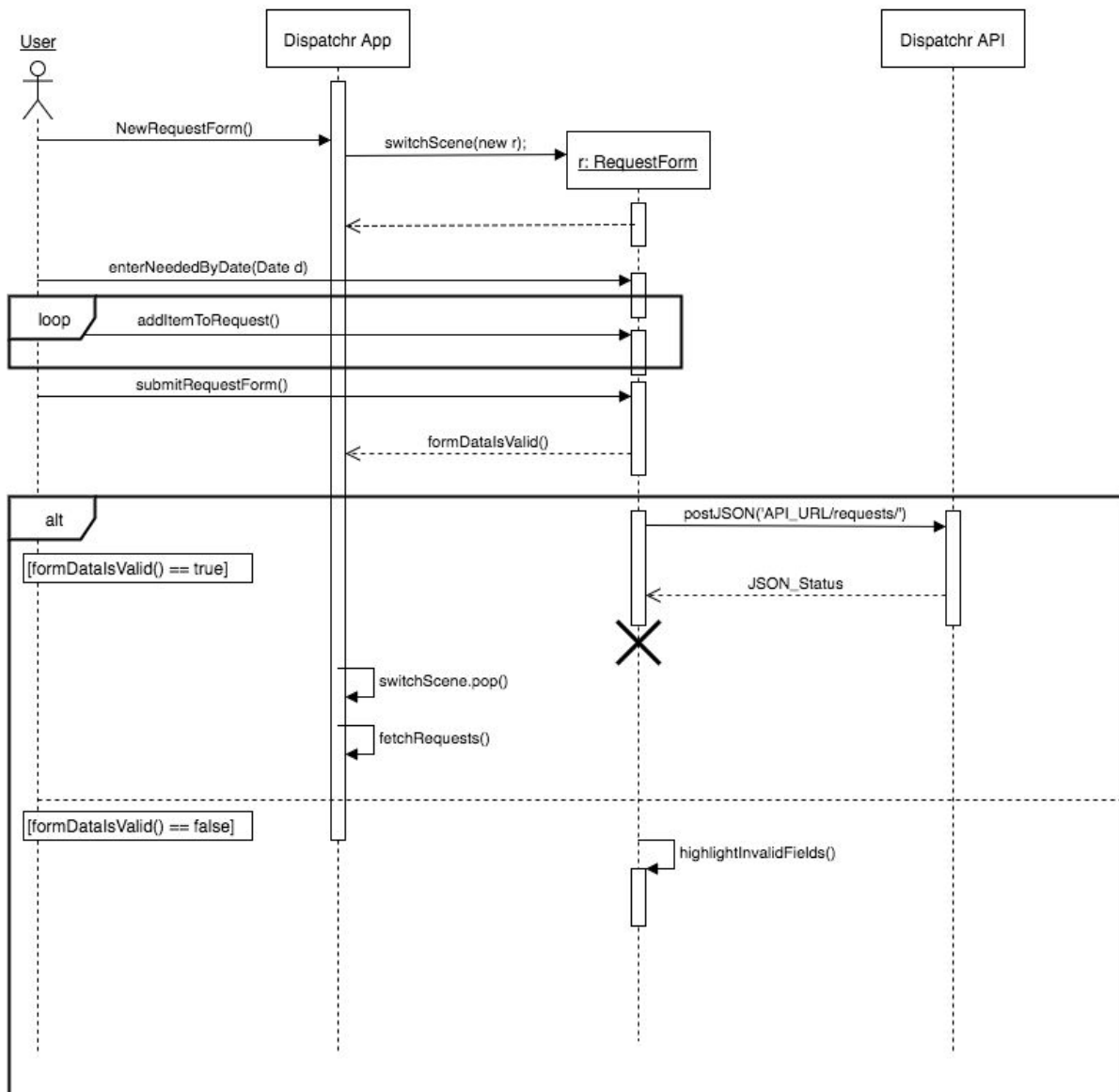
Fetch Requests From API: When the user pulls on the list of requests in order to refresh, the function `fetchRequests()` is called. This will abstract `fetchRequests()` in the following diagrams

Add New Items to Request



Add New Items To Request: When the user is creating a new request, he/she needs to add the items he/she wants picked up and delivered. This will abstract `addItemToRequest()` in following diagrams.

Create New Request



Create new request: Allows user the ability to create a new request, and have it appear in the global request list.

Appendices

Technologies Employed

- Heroku
- Ruby on Rails
 - Rspec
- PostgreSQL
- React-Native
- Redux
- Redux-Saga
- Trello
- Github

Technical Definitions

Web Service: Comprised of HTML and JavaScript, a web service is what runs on the consumer's app or web browser and talks to other services directly.

RESTful API: REST stands for Representational Stateless Transfer. It's a type of architecture used to build web services such as APIs. There are six primary constraints: uniform interface, stateless, cacheable, client-server separation, layered system, and code on demand (optional). A RESTful API is powering the Dispatchr servers.

Heroku: Heroku is a Platform-As-A-Service that enables developers to seamlessly deploy their applications to the cloud for easy access in any context. In Dispatchr, Heroku is used to deploy our API and ensure that the mobile frontend is able to access the endpoints.

Ruby on Rails: Ruby is a scripting programming language. Rails is a framework that runs on top of Ruby, enabling developers to rapidly build robust, full-stack web applications. The Dispatchr API is developed using Ruby on Rails.

Rspec: Rspec is a testing library for Ruby on Rails that allows developers to write unit tests to ensure the proper functionality of their code. In the context of Dispatchr, it is used to write the unit tests before any code is written, a practice known as test-driven development.

PostgreSQL: PostgreSQL is a relational database system. Dispatchr uses PostgreSQL in order to persist data in the backend of the application.

React-Native: React-Native is an extension of React.js, a popular JavaScript framework used to build high-performance websites quickly. React-Native enables developers to build applications for iOS and Android at once using its libraries. See project documentation for more details [here](#).

Redux: Redux is used for managing data and state throughout the application. It allows developers to write apps that behave consistently and across multiple environments and platforms (client, server, and native). Redux also provides live code editing combined with a time traveling debugger. In Dispatchr, it's used in combination with React.js to build the mobile frontend. See project documentation for more details [here](#).

Redux-Saga: Redux-Saga is a library that simplifies the side-effects of JavaScript for developers and makes it easier to develop with. These side-effects include asynchronous tasks such as data API calls, or long computations. Redux-Saga is constantly running in the background waiting to be called. See project documentation for more details [here](#).

Trello: Trello is a project management tool used to keep track the state of the project. It's used to monitor the status of user stories for Dispatchr, whether a story is to be completed within the current sprint, assigned to someone, in-progress, in review, etc. It allows our team to monitor our progress and assures accountability as well.

Github: Github is the version control software used to manage the Dispatchr codebase.