MariGold

Team 5 - Design Document

Andrew Bass, Will Borland, Pravin Sivabalan, Devin Sova

Purpose

Medication is a core part of everyday life that comes with many challenges and risks. If the wrong medications are taken together, the results can be disastrous for patients. Our project goal is to organize, inform, and simplify a user's life with respect to the medications they take throughout the week. We aim to prevent medical tragedies by telling users when their medications conflict and when to take their medication. User's can also find medications to address symptoms and can determine which of their symptoms may be side effects.

Requirements for our app include:

1. Managing an Account
    a. Any user that has downloaded the app may create an account.
    b. This information is registered to the database including their medication list.
    c. Users will be able to delete their account from our system or reset their password.
    d. Users data will be secure and personal. Only accessible to them.
2. Medications
    a. Users may manually input or scan in a medication into the app which will register it into the system.
    b. Medications will be in an editable listview that users may tap to get more information on a specific medication.
3. Conflicts
    a. When a user adds a new medication, the app will use the previously stored information to see if it conflicts with medication the user has already entered.
    b. The app will also check if the new medication is a part of any banned substances in a league that the user is a part of.
4. Side Effects
    a. Users will be able to quickly view a list of possible symptoms or side effects due to the medications they are taking and possible conflicts between them.
5. Search
    a. Users can quickly search and get detailed information on symptoms they may be experiencing and information about medications to resolve those symptoms.

    b. The search will also inform the user if the symptom they have searched is a result of any of the medication they are already taking

6. Dashboard
    a. Upon opening the app, users will be greeted with an overview of their medications and schedule.
7. Notifications
    a. Users will be able to opt into notifications reminding them to take medications at their appropriate times during the week.
    b. In addition, users may opt into being reminded when to get a medication refilled.

Design Outline

Marigold is an application that allows users to track their medication and get information regarding their medication. Our application will use a many-to-one client-server architecture style where many clients request data from one server. Our server will interact with a database, our Optical Character Recognition (OCR) module, and the FDAs openDrug API.

Client

Our client will be an iOS app that users interact with. The app will handle all interactions with the user.

Server

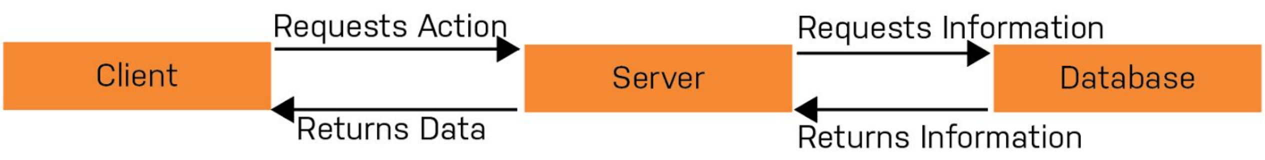
Our server will control all interactions with the client, database, OCR module and FDA API. The server will supply the client with the requested data, make and receive queries from the database, run our OCR module and talk to the FDA API.

Database

The database will store all information about users and their medications. Anytime the server requests data the database will serve that data to the server.

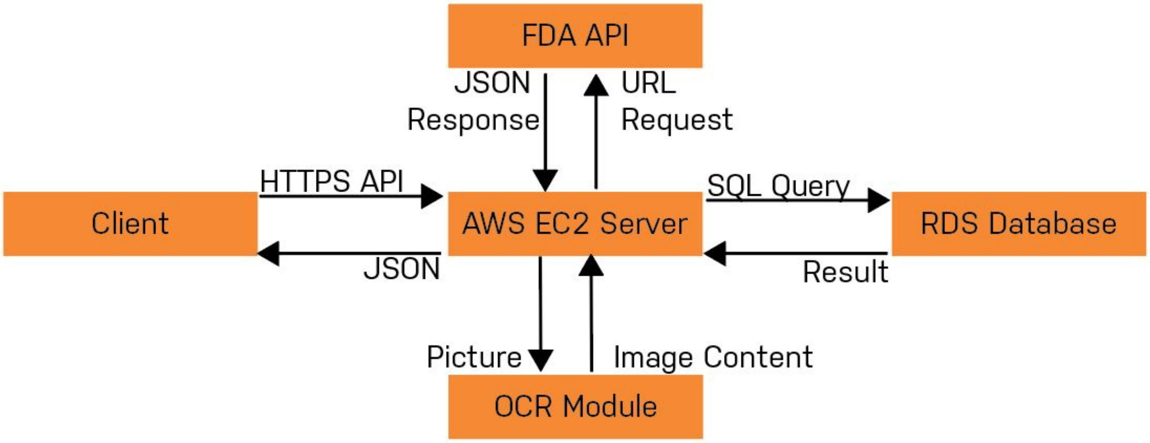
High Level Overview

Marigold will use the many-to-one client-server architecture model. The client will request data from the server when they need user medications, medication conflicts or symptom information. The server will query data from the database and compile the information into the correct format for the client.



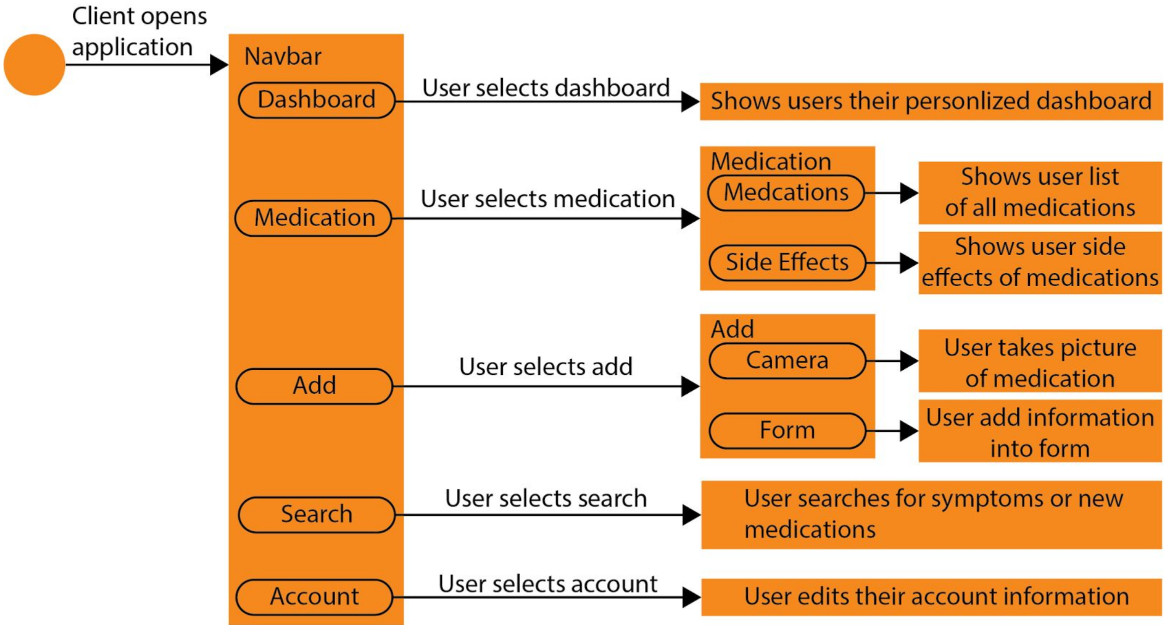
Architecture

Our iOS app will request data from the server through a HTTPS request. Our server then will interpret the request and perform the necessary function to respond to the client. When needed, our server will request the FDA API whenever it needs information about an unknown medication or side effect. When a user takes a picture of their medication label, our server's OCR Module will pare the picture and return its information to the server.



Activity/State Diagram

The diagram below shows the different states that our app can be in. The user can select any of the options in the navbar and the app takes the user to that page.


Client opens application
Navbar
Dashboard
User selects dashboard
Shows users their personlized dashboard
Medication
User selects medication
Medication
Medcations
Shows user list of all medications
Side Effects
Shows user side effects of medications
Add
User selects add
Add
Camera
User takes picture of medication
Form
User add information into form
Search
User selects search
User searches for symptoms or new medications
Account
User selects account
User edits their account information

Design Issues

Non-Functional Issues

Issue: What framework should we use for our backend?

- Option 1: Laravel (PHP)
- **Option 2: Flask (Python)**
- Option 3: Django (Python)
- Option 4: Node (Javascript)

We decided to go with Flask and Python for our backend. We chose Python for its simplicity and for the wide range of third-party libraries that are easy to install. It tied into our goal of keeping our backend lightweight and simple. Since we are developing a mobile app, we wanted a framework that excelled at communicating raw data to and from clients, with no need for complex HTML features. Flask's clean design fit our desire perfectly.

Issue: What mobile platform shall we target?

- Option 1: React Native
- **Option 2: Native iOS**
- Option 3: Android

After considering the strengths and weaknesses of each platform we decided on Native iOS. There are performance issues with React Native and no one on the team is familiar with the platform. We ruled out Android due to its poorly performing emulator. We chose iOS because of its superior camera API and the tech stack that comes with it. iOS's superior camera API makes the scanning portion of our app easier to implement. We also prefer the tech stack on iOS over the approach Android takes. Native iOS allows us to achieve great performance, battery consumption and design.

Issue: Where will our backend be hosted?

- **Option 1: Amazon Web Services**
- Option 2: Digital Ocean

We chose Amazon Web Services to deploy our backend because of the flexibility and control we have over the linux server. AWS allows us to easily attach an SSL certificate to our web API. Scalability also was a big factor in choosing a platform and AWS beats out Digital Ocean in this respect. In addition, our team has more experience working with AWS.

Issue: What database platform should we use?

- Option 1: Postgres
- Option 2: SQLite
- **Option 3: MySQL**

After comparing each database system we chose to use MySQL. SQLite is a simple solution, but we wanted the application to have multiple clients use the same database. SQLite is commonly used for local storage, but doesn't scale well enough to handle large-scale interactions. Postgres was considered, but we decided to go with MySQL because of its popularity and our groups familiarity.

Issue: What authentication system do we use to authenticate the user?

- **Option 1: JSON Web Token**
- Option 2: Sessions Token

In order to authenticate the user's requests we needed to choose a system to validate that the user making the request is actually the user. We had the choice between sessions and JWT. We decided to go with JWT because of the fact that they are stateless and are able to carry a small payload, whereas sessions aren't stateless and this could cause issues when scaling.

Issue: How do we store information from the FDA API?

- Option 1: Keep generic information on each medication.
- Option 2: Call API on every information request.
- **Option 3: Use a combination of Option 1 and Option 2**

In order to get information about users medications, we are going to use the FDAs openDrug API. In our design process we debated on calling the API every time we needed information about a certain medicine or storing generic information about each medication. We decided to go with a mixture of both because we realized that the medication could be updated on the API, and we didn't want to risk giving our users outdated information. Yet, we still wanted to optimize performance by reducing the number of calls made to an external API. This is why we decided to store information that can't be changed on the backend, and when the frontend application needs more specific information on a medication a call can be made to the API to get the most recent information.

Issue: What language should the iOS app be?

- Option 1: Objective-C
- **Option 2: Swift**

While Objective-C used to be the standard for developing iOS apps, today the recommended language for new apps to use by Apple is Swift. Compared to Objective-C, Swift has a much lighter syntax that is easier to remember and read. The languages also supports more features than Objective-C, such as enum cases with specialized values. It's much easier to find documentation for modern iOS features that uses Swift as the example language. In contrast, Objective-C's documentation is outdated and oriented towards older iOS versions. To make development faster and to keep our code cleaner, Swift is the clear choice.

Issue: What type of architecture should be used?

- **Option 1: Client-Server Architecture**
- Option 2: Multi-Layer

We decided to go with the client-server architecture style. We wanted to easily change the iOS app look and feel whenever we wanted and not have to change the backend. With client-server we will have multiple clients connecting to the same server to request personal and shared data.

Issue: How do we query the database?

- Option 1: ORM
- **Option 2: Plain SQL**

We chose to communicate with the database by using plain SQL. Using an ORM would allow us to have other software handle SQL statement validation and table mapping. Our database is not that expensive and we thought it would be better to use built in SQL verification and our own queries.

Functional Issues

Issue: How will the user navigate the app?

- **Option 1: Navbar**
- Option 2: Main page
- Option 3: Hamburger Menu

We considered the amount of core functions that our application will have. When considering this we came to the conclusion that out application will have around 5 core functions. This fact played a major factor in us deciding which method to navigate our app. The benefit of the main page is that it allows you give the user multiple links to display, but it prevents you from navigating using any page, instead you have to always navigate back to the main page. The benefit of the hamburger menu is that it can be accessed from anywhere in the app, and it can handle a longer list of links. The issue with the hamburger menu is that it requires two taps to navigate. We decided to use the navbar because it allows navigation from anywhere and only requires one tap.

Issue: How will users login?

- Option 1: Allow users to login with Facebook, Google, Amazon etc..
- Option 2: No logins
- **Option 3: Custom login for our app**

Our app must associate unique data with each user. Therefore, not using logins and allowing people to use the app anonymously is not a valid option. We considered allowing people to use social media accounts as a login option. There is definitely a convenience factor, but there is an important caveat. We store medical data about users, and must therefore uphold their privacy. We don’t want any chance of a third-party accessing individuals medical information. Therefore, we will implement our own login system that allows users to quickly register and login.

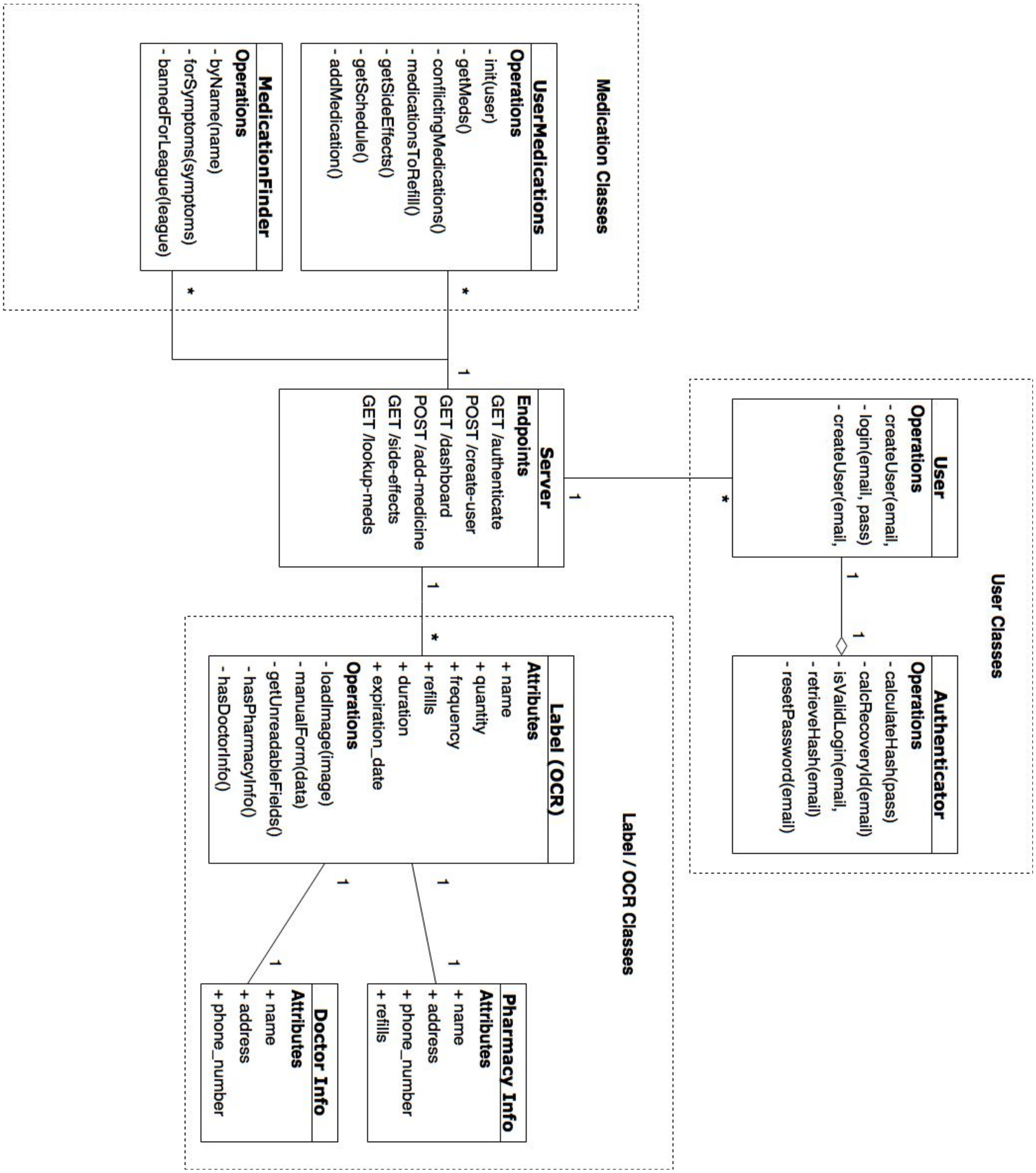
Issue: How will users receive notifications?

- **Option 1: iOS Notifications**
- Option 2: Email
- Option 3: SMS

The notifications that users will be getting from our application will include reminders to take medication and reminders to refill their medications. We chose not to do emails since they are not really appropriate for such reminders. Emails are for important account messages only. SMS was also considered but isn't as easy to setup and manage. Local iOS native notifications provide quick and customized notifications to users.

Design Details

Backend Request Handler Class Diagram



Classes and Models

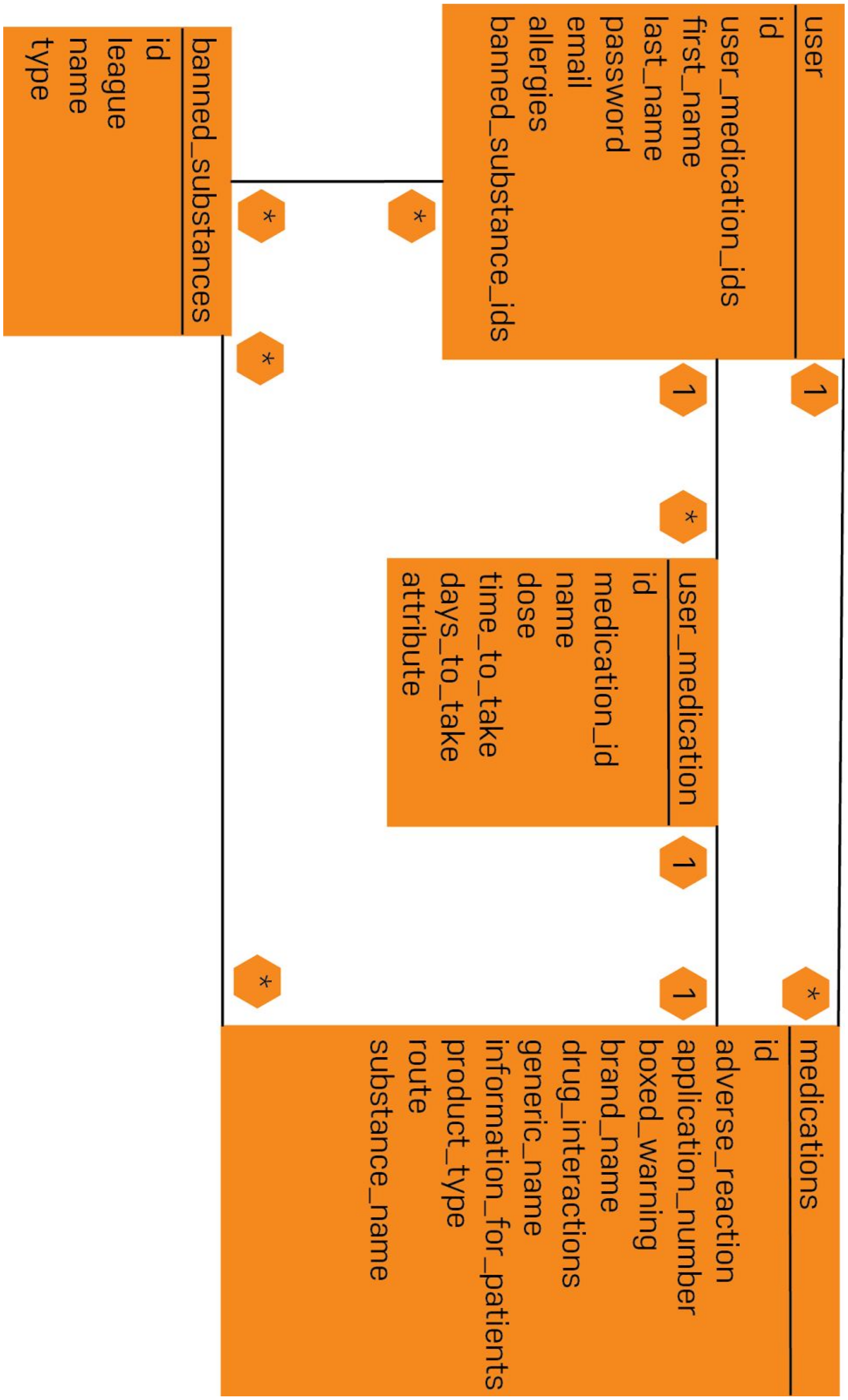
Our Backend Request Handler Class DIagram details the relationships between the classes that carry out user requests. We've divided our backend classes into 3 main sections.

The User classes are wrappers around the user information from the database. Our User class uses the Authenticator class to handle the details of the user subsystem, such as hashing passwords and creating recovery IDs for password recovery.

The Medication Class group includes classes that manage a user's medication and find medication from the openDrug api. The MediciationFinder class provides a simple interface for finding medication based on the criteria required by the app. The UserMedications class acts as a middleman to add, modify, and query a given user's medicine from our database.

Lastly, the Label classes encapsulates the functionality that is used by our OCR subsystem to parse information from medication labels. An abstract Label object is used as the interface to this subsystem. An important feature of this subsystem is that the backend can discover which fields were unparsable from the label. The user can then manually fill these fields in.

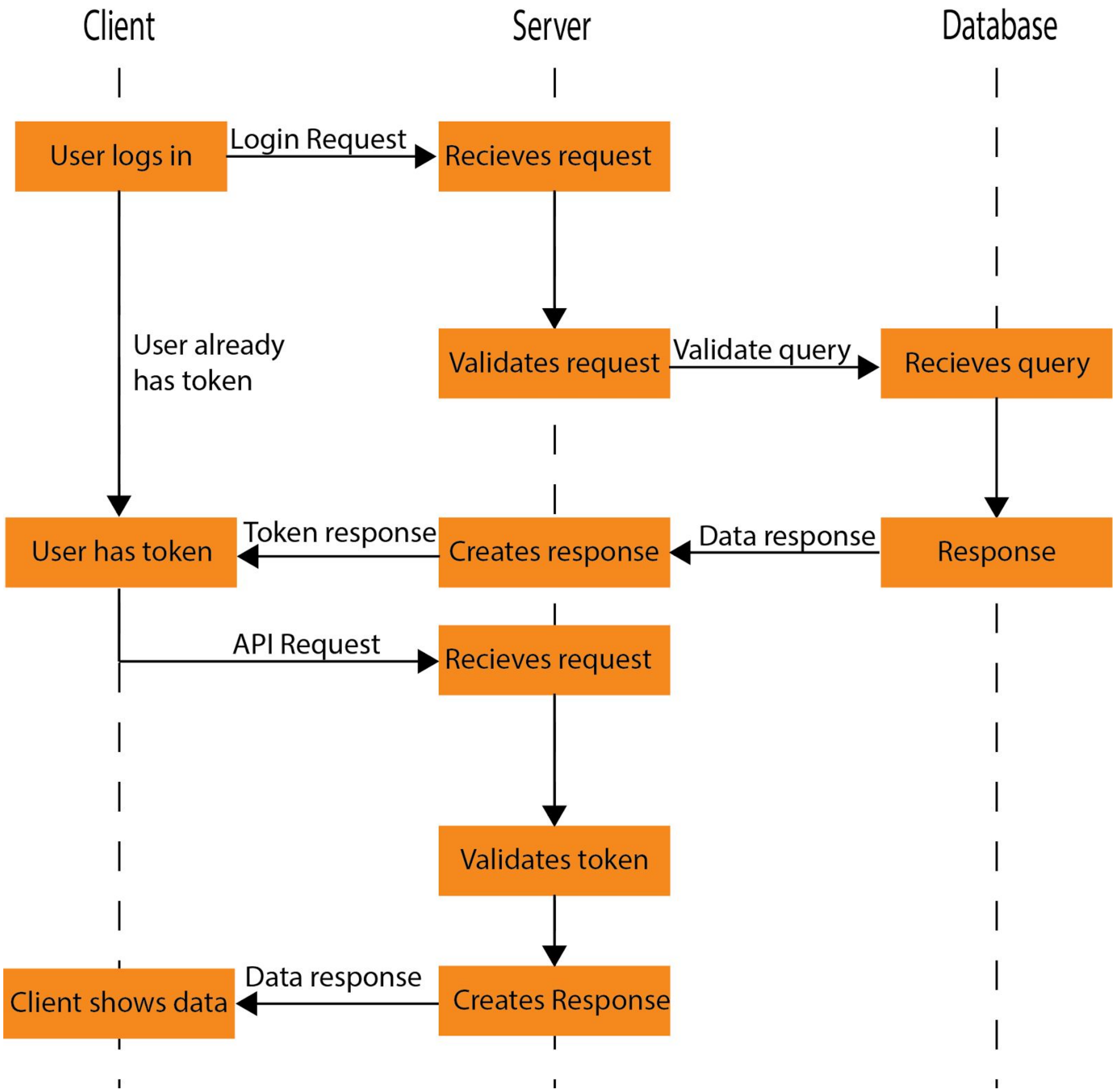
Database Scheme



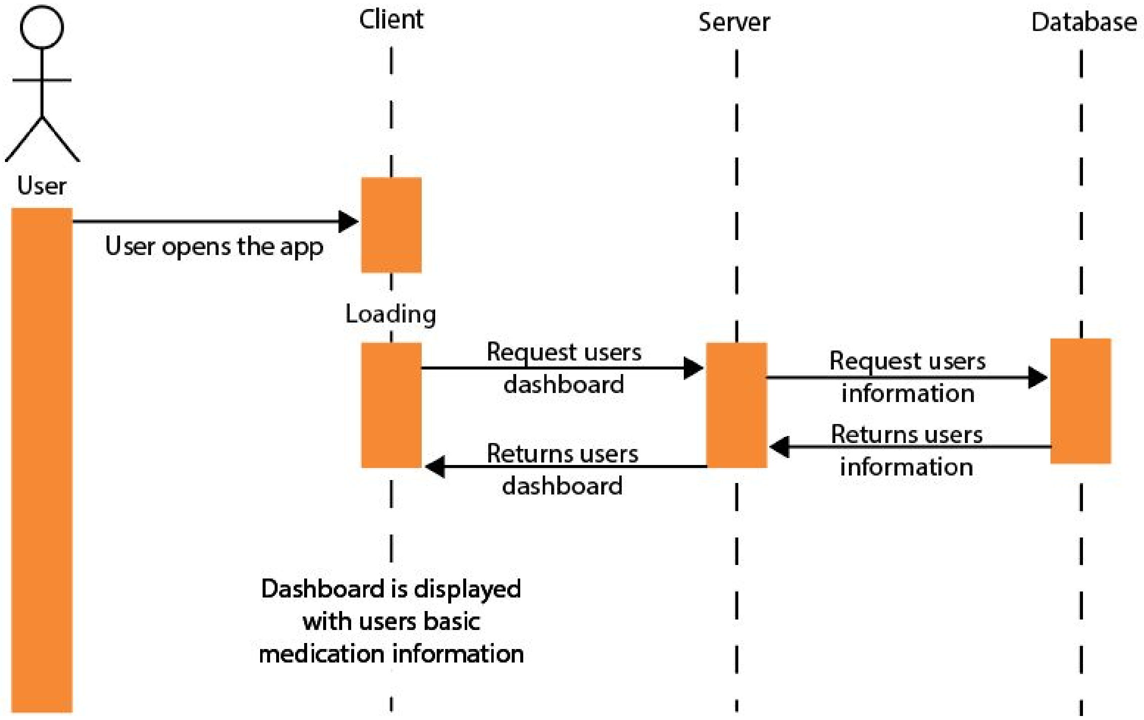
JSON Web Token Authentication

We are going to use JSON web tokens (JWT) to authenticate API requests.

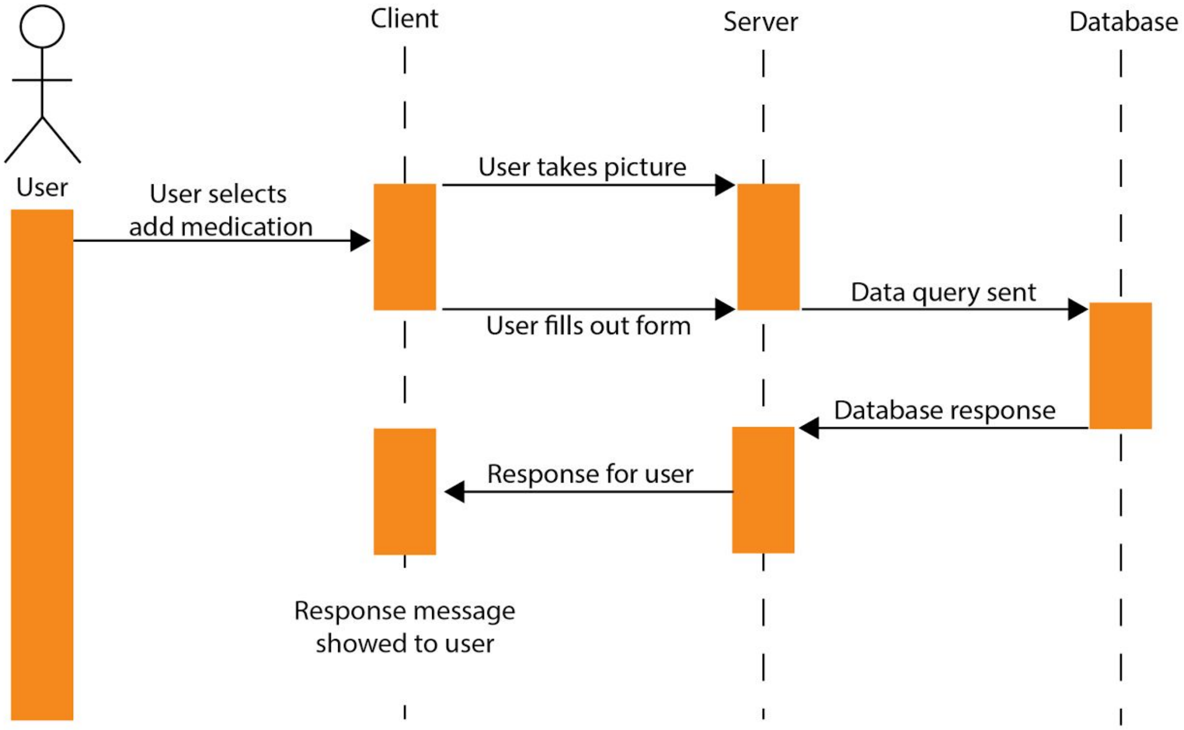
Our JWT authentication will work with first having the user login and then have the server and database validate the users information. The server will then craft a response. An invalid login will send an error to the user and a valid login will create a JWT and be sent to the user. After the user has their JWT every API request will require this token. When a request happens the server will decode the token and either send the request data or an error message.



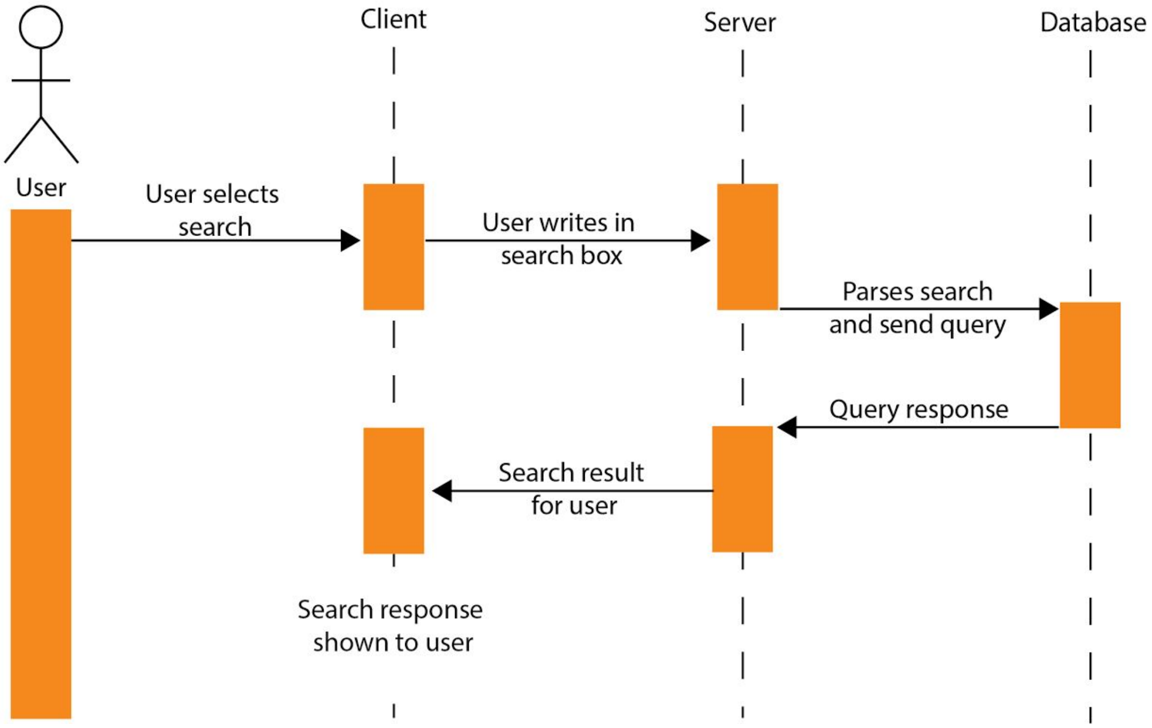
Sequence of Events When User Opens App


Client
Server
Database
User
User opens the app
Loading
Request users dashboard
Request users information
Returns users information
Returns users dashboard
Dashboard is displayed with users basic medication information

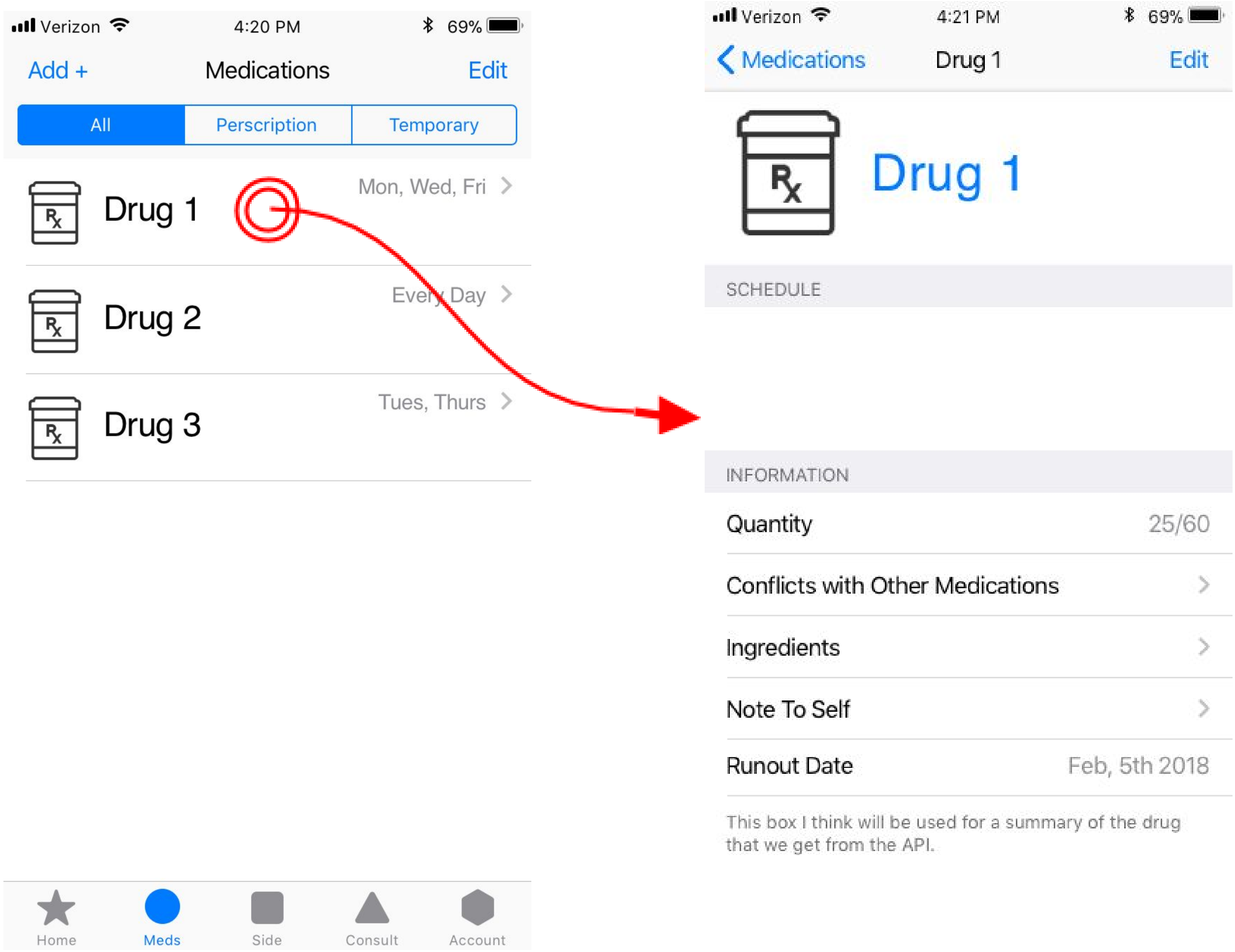
Sequence of Events When User Add Medication



Sequence of Events When User Searches


Client
Server
Database
User
User selects search
User writes in search box
Parses search and send query
Query response
Search result for user
Search response shown to user

UI Mockups



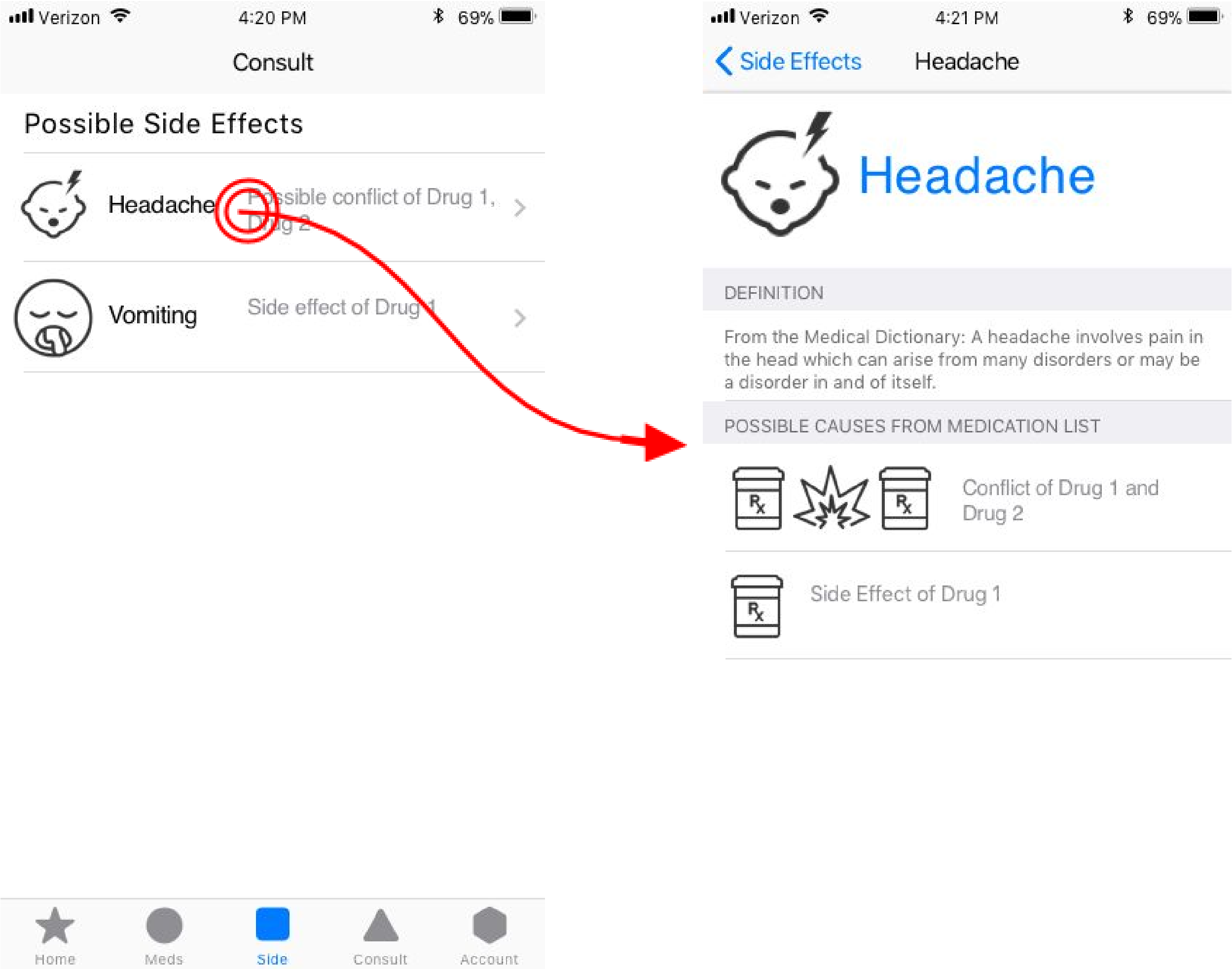
Quantity 25/60

Runout Date Feb, 5th 2018

This box I think will be used for a summary of the drug that we get from the API.

The left UI mockup show how a users medication tab will look. The right UI mockup shows what a single detailed user medication will look like.

UI Mockups - Continued



Verizon 4:20 PM 69%

Consult

Possible Side Effects

Headache Possible conflict of Drug 1, Drug 2

Vomiting Side effect of Drug 1

Verizon 4:21 PM 69%

< Side Effects Headache

Headache

DEFINITION

From the Medical Dictionary: A headache involves pain in the head which can arise from many disorders or may be a disorder in and of itself.

POSSIBLE CAUSES FROM MEDICATION LIST

Conflict of Drug 1 and Drug 2

Side Effect of Drug 1

The left UI mockup shows what the side effects tab will look like. This tab shows the user what possible side effects the user could be experiencing due to the medications they are using. The right UI mockup shows an individual symptom that the user may be experiencing and what medication or conflicts may be causing it.