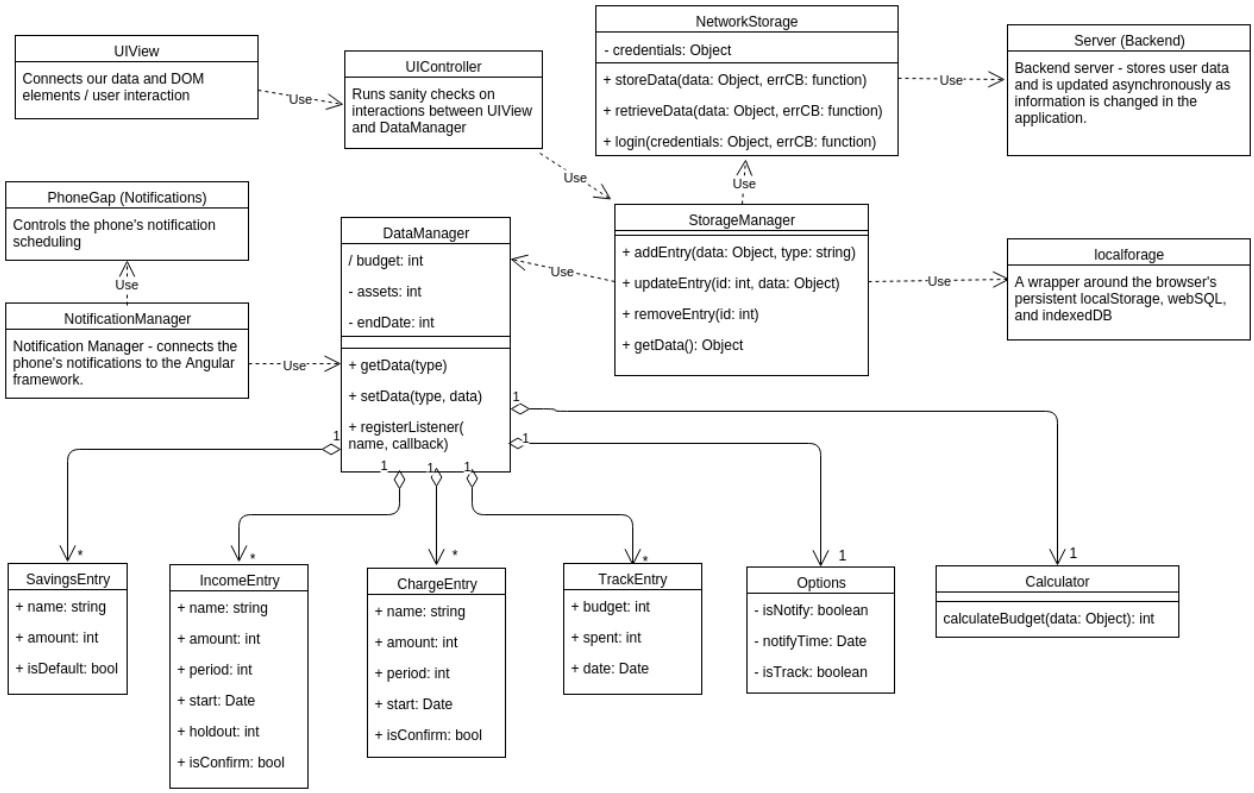
Budding Budget

owsenk	Kyle Owsen
lizs3434	Elizabeth Schibig
ischaaf	Isaac Schaaf
jbktsang	Jessica Tsang
hstefan	Stefan Holdener
mjsc	Maxton Scott Coulson

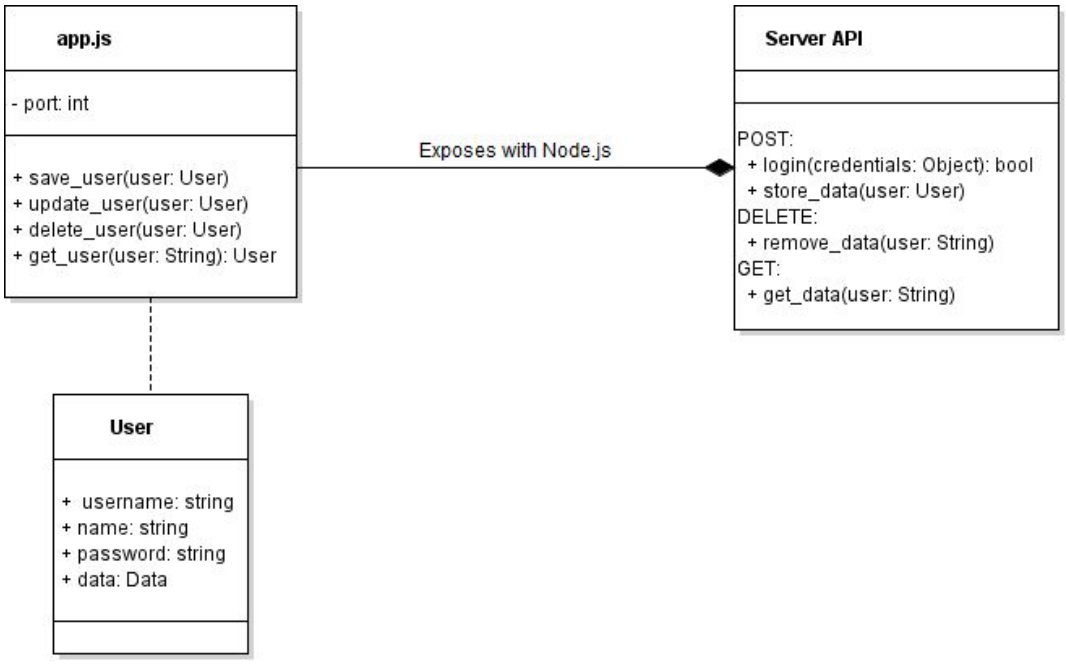
System Architecture

UML Class Diagram - Application



This diagram shows the major modules and interfaces contained within the Budding Budget phone application. It connects to the server implementation, with the server below represented by the “Server (Backend)” box in the above diagram. Right now, the UIView and NotificationManager modules have the ability to get data from our data cache, but only StorageManager has the ability to set it, based on data it receives from localforage and/or NetworkStorage.

UML Class Diagram - Server



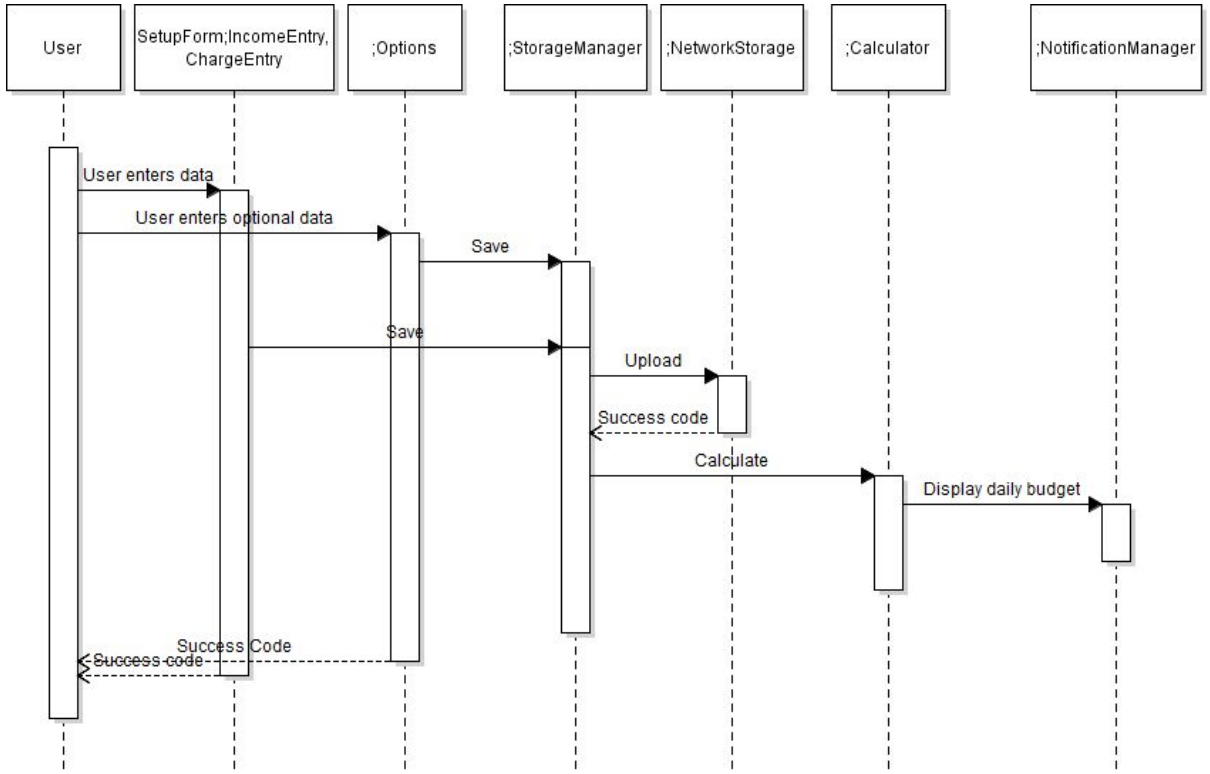
Database Schema

For the application, the database schema will resemble the “Data” object in the UML Class Diagram for the phone application. There will be separate tables for Savings, Income, Charges, and Tracking. Each of these tables will have a column for each field in their Entry objects in the diagram. Alongside that, we will store the assets, endDate, and options using PhoneGap’s LocalStorage as simple values.

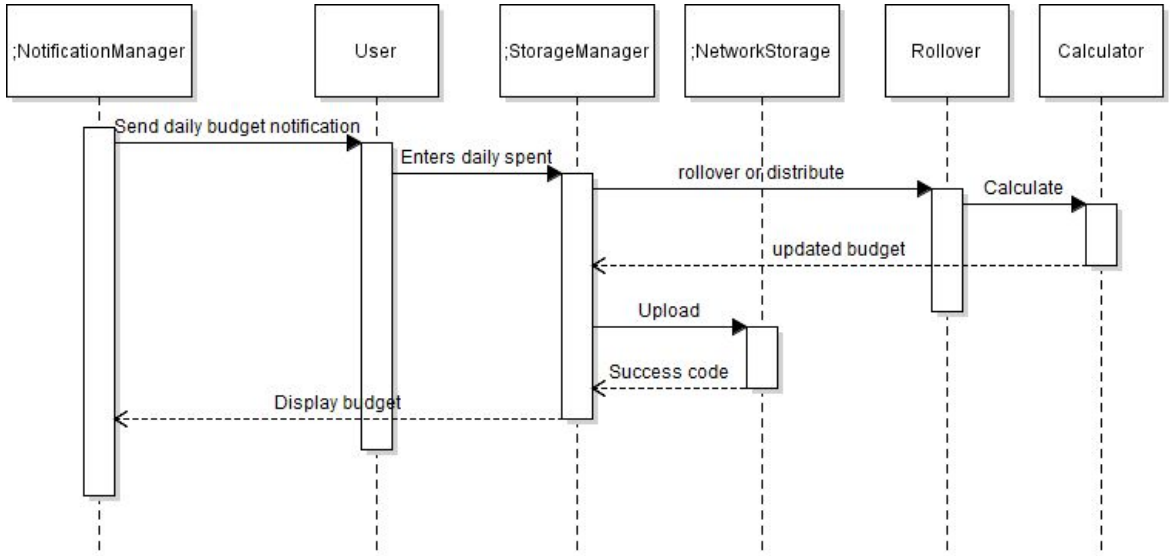
On the server, the data will be stored using MongoDB with the User object serving as the schema. One alternative we considered was using MySQL instead of MongoDB. We decided against this because while the client is locked into SQLite (with PhoneGap) the server is not. Therefore we can take advantage of MongoDB’s use of JSON to form database schema and simplify the server database interaction greatly.

UML Sequence Diagrams

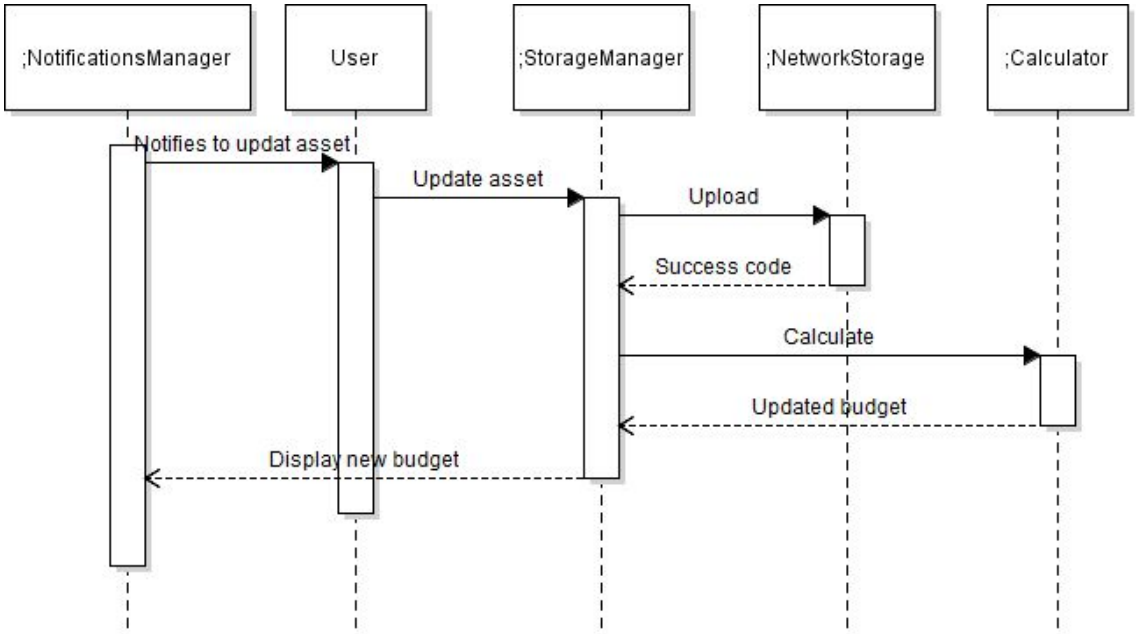
Initial setup



Spending over/under



Update assets



Process

Risk Assessment

1. The app is not compelling

Medium likelihood, High impact

Accurate understanding of likelihood will occur after final implementation of features. For impact it is high impact as the product's use will potentially be void if it is not compelling. With most apps, if the user does not use them for days or weeks at a time, they can come back to the app later with little or no issue. If they skip budget tracking for just one day, the information given by the app is no longer valid.

To reduce the likelihood of this issue occurring we will keep this issue in mind as well as have multiple stages of testing with potential users. This testing will also allow us to keep aware of the likelihood of this issue occurring.

We plan on mitigating this risk by forcing the user to either update their assets, or confirm that the budgeting information is still correct if they miss a day of tracking.We plan to do extensive user testing so the tracking is easy and appealing to the user. If we are unable to overcome this risk, our app will not be useful, which is why it's the most important risk.

Plan for detecting the problem: Test with users and evaluate their level of use

2. Issues with the algorithm to track spending

Low likelihood, High impact

This issue seems unlikely for us to run into as we have multiple team members with strong mathematical backgrounds, and the algorithm's required appear straightforward to establish. It is high impact if we have issues with the formula as it is the foundation for the idea/use of the product.

As far as reducing the likelihood of this issue occurring, as a team we simply need to clearly establish what our requirements are for the algorithms and research into any issues that we may come across in formulating such an algorithm.

We can manage this risk by creating the algorithm early in development, so we can adjust the direction our development goes based on any shortcomings in the process we create. If we cannot make a satisfactory algorithm, we'll have to cut back on some of the more ambitious elements of the app, like being able to set an indefinite end date, or considering recurring charges.

However, if the algorithm provides too much of an issue we will have to rework what we are requiring of the app design.

Plan for detecting the problem: Test the algorithm and compare against the desired results, also test the algorithm in a test build of the app to see if the algorithm continues to work.

3. Issues with the cloud sync component

Medium likelihood, Low impact

There is a medium likelihood of encountering this issue as we are generally inexperienced with cloud software, therefore we will probably run into issues but should be able to overcome them with research and testing. It is also low impact as it is not a crucial component to the product, but very desirable.

To try and reduce the likelihood of this issue occurring all we need to do is research into the limitations and potential issues with cloud sync technology, and keep those in mind when coding. Thus, we would be less likely to run into issues.

Our plan for this is to settle conflicts based on whatever syncs to the database first, and to force the other device to pull data from this sync. Since we see the user tracking a budget on multiple devices at the same time as an edge case, this solution being less than elegant from a user perspective is fine so long as the database remains consistent.

If we cannot make this work, we'll need to either only allow one device to be associated with an account at any one time, or cut the online feature entirely, but these options should only be seen as a last resort, doomsday scenario.

Plan for detecting the problem: Test the syncing and see if the results meet our expectations and requirements.

4. The Phonegap API not being adequate

Medium likelihood, High impact

Because we are unfamiliar with Phonegap there is potential to run into issues with the API, but because we aren't requiring anything particularly demanding of Phonegap it is only a medium likelihood of risk. There is a high impact as it would require us to use alternative software or even worse case scenario have to design our code entirely.

To mitigate this issue (as well as reduce the likelihood of issues occurring) we will research into the Phonegap API and see what its limitations are, then when we are coding we shall be able to take these limitations into account.

Or, worse case scenario, which is that it is not adequate, we have to either code directly as a phone app or we use another software similar to Phonegap.

Plan for detecting the problem: Testing Phonegap, researching into what the limitations for Phonegap are.

5. Database schema information not being sufficient

Low likelihood, Medium impact

There should be a low likelihood of issues occurring with database usage but it should be acknowledged and taken into account as we are storing information. The impact of issues with this database is potentially high but more probably just requires a small reworking of the database or of the code.

To try and avoid this issue we intend to extensively test our database structure to ensure integrity, as well as have the entire team review the database.

If this problem were to occur we would have to rework our database and have the team weigh in on these decisions to make sure it doesn't threaten the overall code integrity.

Plan for detecting the problem: Testing Databases

Since the last SRS we have identified more risks and evaluated each risk to a more in depth level. We are also now aware of how much research is required to limit the potential risks of affecting other aspects of the project. Furthermore it is apparent how much testing will be required during the construction stage of our app in terms of both coding as well as user testing.

Project Schedule

<table>
<tr><td>Week - Group Goals</td><td>Front End</td><td>Back End</td></tr>
<tr><td>Th : 1/28- Finish Software Design Spec
Due Monday 2/1</td><td colspan="2">Divided up sections of the SDS document between members of the whole team, review final document together.</td></tr>
<tr><td rowspan="2">T : 2/2- Finish Design Presentation
Due Tuesday 2/2

Th: 2/4- Work on documentation for users and developers, start zero-feature release</td><td colspan="2">Split requirements, design, and planning slides for presentation.
Assign half of team to practice presenting.</td></tr>
<tr><td>Starting coding, designing the UI. Learning the toolchain.

Have version control, bug tracking, other tools setup beforehand.</td><td>No back end work.
Zero feature release is UI only.

Start working on product website and user/developer documentation instead.</td></tr>
<tr><td rowspan="2">M: 2/8- Zero-feature release due

T: 2/9- CODE

Th: 2/11- Update SRS, SDS and other documentation, establish unit tests</td><td rowspan="2">Skeleton UI should be done.</td><td>Get cloud database setup.
Get local storage features integrated with cloud, etc.

Start working on internal calculator</td></tr>
<tr><td colspan="2">Begin updating documentation.
Each member of team responsible for their own unit/system tests.</td></tr>
<tr><td rowspan="2">T: 2/16- Major features implemented

Th: 2/18- Prep for demo.
If time allowed: Started stretch features</td><td colspan="2">Get core features to the app (front end and back end) integrated</td></tr>
<tr><td>Debug/clean existing code.
Finish most of UI</td><td>Debug/clean existing code.
More back-end implementation (calculator, database, etc)</td></tr>
</table>

F: 2/19- Beta Release, beta demos due	All prep for demo
T: 2/23- Work on stretch features Th: 2/25- Nearly all relevant bugs resolved. Update all documentation. F: 2/26- Feature-complete release due	Complete all remaining main features (notifications, network, remaining UI Components) Keep documentation up to date. Start work on fixing bugs and issues, all relevant edge cases, etc.
T: 3/1- Perform/finish user testing Th: 3/3- Work on code reviews, all bugs resolved. F: 3/4- Release Candidate due	Finish implementing account login Create more test cases to find bugs Recovery Week: Fix any issues, catch up Polish UI
T: 3/8- Final Release Due W: 3/9- Final Project Presentations	All work on wrap-up and prep for presentations

Team Structure

Our team will be dynamic following an agile development model. We do not want to impose a fixed team structure and project schedule and will update it each week to stay flexible.

We will divide the members in both front end and back end, with specific tasks specified in the project schedule, as well as make each person an expert on a certain topic to ensure we have enough knowledge for successful development.

The team members will focus on following topics:
Front end
UI: Elizabeth, Jessica
algorithm: Stefan
PhoneGap: Kyle

Back end
server: Isaac
network & testing: Maxton

We meet twice a week on Tuesdays, 9:30-12:00 and on Thursdays, 9:30-12:00 to be able to communicate in real time and discuss the most important issues. We are using the bi-weekly reports to fix tasks we try to tackle for the coming week. We use the GitHub wiki for basic information about the project for both our self, as well as others interested about the project. We will have discussion on issues and features on the GitHub Issue Tracker and use ZenHub to plan sprints and our workflow directly on GitHub with the tracked Issues. For other communication we use a Google Groups mailing list.

Test Plan

We will use Jasmine as our main testing suite for javascript code. We use Travis CI for continuous integration on GitHub. Karma will be used to manage the Unit tests written in Jasmine. Every developer writes white-box tests for all of his classes. At peer reviewing other developers will review them and also add additional black-box unit tests. We use continuous integration and do a nightly build with all the features.

Our system tests will cover the main use cases we have. Jasmine can handle some basic system tests, and Protractor can be used to drive Selenium tests. Ideally everybody helps writing system tests as early as connected components are able to be tested. We do not think it is needed to automate this testing but are still flexible. They will be run ideally each few days and after each larger change to the system.

The usability tests will cover the basic workflow in the app and make sure it is intuitive and easy to use. We are confident that we can use our team members to test out the basic usability of the app in the beginning. In later stages we will test the usability on other people to see how intuitive it is for newcomers. These tests will be run in frequent but not fixed intervals.

We believe these tests are enough to mitigate major errors and maximize the productive time. We can monitor if these tests do not meet our expectations and still adjust the frequency of them to better adjust them.

We will use the GitHub integrated Issue Tracker to track our bugs and problems that arise, as well as discuss and solve them.

Documentation Plan

Our app will have a introduction tutorial to explain new user how to use the app that can also be accessed later on. We will have a guide on our project website as well that will have similar information there. On our GitHub we will have a manual for developers on the structure of the project and how to build it.

Coding Style Guidelines

Each team member will read through the guidelines for Javascript coding as discussed in http://www.w3schools.com/js/js_conventions.asp and take these points into concideration when they do their own coding. Furthermore, team members will peer review code to see if it is in line with the guidelines (as well as general comments on the code structure/quality). During the coding process we will also use automatic code checking to ensure that more immediate issues in coding style are addressed.

Design Changes and Rationale

Due to a restructuring, and us abandoning Angular, our UML Class diagram for the application is significantly different. The reasons for the change from Angular are detailed in the process description, and many of the other changes followed naturally from that.

We removed the allotted time for working on stretch features from our calendar because we no longer believe we'll be feature complete for the app's main functionality with enough time to spare to work on those.