IOU

SHARE AND TRACK EXPENSES WITH YOUR FACEBOOK FRIENDS

Jose B. Gomes - 12500741

Sunday, 3rd, May, 2015

word count: 13336

word count excluding headers, appendices and sample code: 13336

BSc Computing Project Report,
Birkbeck College,
University of London, 2015

Abstract

When shopping with friends, it can be rather cumbersome to keep track of all shared expenses. What about having a mean to swiftly store what you spend with a view to settling it at a more convenient time? IOU allows you to create log books, that can be shared with your Facebook friends and keeps track of those things you buy together. From grocery lists to household goods, party makers to house shares, IOU has you covered, simply create a new log book, invite your friends and enter what you spend as you spend. IOU will calculate what you owe and what your friends owe you, with a clever change optimised algorithm. You can even create items you intend to buy, so no need for shopping lists anymore. And what is best, it is all updated in real time.

Table of Contents

Chapter 1

Introduction

The idea for IOU came up from a need raised long before the current technology allowed its' existence.

Being a student living in London for over a decade, almost by definition, meant that I had to share my house expenses with other housemates. This is not constrained to myself and my housemates, most of my friends also share their houses or flats. In fact this problem is not even constrained to a students, most young citizens also find it much easier to share their houses, easing down not only their expenses, but also making friends and having due company in the process.

IOU is a simple application, that allows users to share and track their expenses with their Facebook friends, in fact, IOU has been designed so that one is not constrained to only use it for such purposes, one may wish to use it in order to track his own expenses, or who owes who in the last Barbecue party or even a family using it to know what groceries to buy.

The concept aims to be as simple as it can get, users create or are added to lists of products, each user can only view and edit lists that he/she belongs to. Each list has two sublists, a list of users and a list of products. The list of products is then subdivided into two lists, products that are due to be bought and products that have been already bought.

The sum of all products bought in a list, divided by the number of users, is the ideal amount each list member should had spent, however, in practice a member will rarely be in this situation, IOU will then label members as either a debtor or a creditor, if one is a debtor, he/she should be aware that is his/her turn to buy the next round of products.

A list lifecycle can last as long as it's members find suitable, when it is time to close the list, they may review how much each members owe each other, exchange money (which happens without any interference of IOU), then close the list.

For example, a list of expenses between two long term friends may never be closed and a list between housemates may be closed once one or more of the housemates decides to live somewhere else.

By no means IOU guarantees that the members who overspent money in a list will get paid back, it assumes that users have a "Gentlemen's" agreement among each other and should simply be used to track who's turn to buy products is and somewhat act as a reminder as to what should be bought.

At any given time, a user can open any given list and view who owes him/her and who he/she

owes money to. This number should be optimised in terms of change, to avoid hassle in terms of money exchange, that is, if one is a creditor in a list, he/she should never need to pay anyone back, likewise, if one is a debtor, he/she should only pay back creditors their given percentage. This in turn should avoid users paying someone who will then use part of the received money to pay someone else.

To make this all work, IOU is distributed as a mobile phone application and enforces users to login using their Facebook profiles. What makes IOU possible now is the fact that most of the population nowadays happen to own a Facebook account as well as a smartphone with access to the internet. This makes the application very convenient, enabling users to easily find their Friends and check what needs to be bought on the go.

Notes about this documentation

Code blocks appear as the following Scheme hello world example:

```
(define hello-world
    (lambda ()
         (begin
        (write 'Hello-World)
            (newline)
        (hello-world))))
```

Code meant to be run in the terminal will be prefixed with a dollar '$' sign, please note that the '$' should not be typed in, as the following Grunt initialise function demonstrates:

```
$ grunt init
```

Figures and captions are displayed like so:

Fig 1 - WebSockets architecture

Chapter 2

Literature review and Context

Although a good idea, IOU is not alone, in fact, there are several other smartphone applications that allow expense sharing and beyond.

This chapter aims to rationalise each of the main alternatives in contrast with IOU, highlighting where it exceeds and where it falls short.

The list of IOU phone apps comparison in this chapter is by no means exhaustive, neither in number of features nor in the number of applications. It simply names the most popular applications available and analyses their main features.

Splitwise

What is it

A fully featured money splitting application that allows users to track expenses and even caters for fairness common bill sharing problems such as, to which extent is a girlfriend a housemate and how rent should be split.

Where it falls short

Although it integrates with Google+, it has no integration with Facebook, making it a little harder to find your friends and rather difficult to use in different devices, having to remember which email address and password was used when one first signed up for a service seems slightly out of fashion now.

For some, this may be a plus, but I find it has too many features, making it complicated to get started with, for example, when creating an item, the user has to stipulate how to split it, debts simplification is an option and it does not enforce the usage of groups.

It only tracks what users have spent, not what is due to be bought.

Where it exceeds

Other than the points made above, it is really hard to find something wrong with this application, it is available for all major platforms, including a web version, it is very complete, boasts a beautiful interface and is never wrong when the subject comes to maths accuracy. Most of all, it is free.

If I were to choose an application other than IOU, Splitwise would be my first choice.

I.O.U - I Owe You

What is it

It is a simple application to keep track of IOU notes.

Where it falls short

It is only available for Android users and the interface is not particularly pleasant to look at. It does not integrate with social media of any kind, operating more as a stand alone application.

Where it exceeds

Simplicity, it does what it says in the tin, tracks IOU notes.

Still Waiting

What is it

Yet another IOU reminder application.

Where it falls short

It does not integrate with Facebook and the app is not consistent through different platforms, the iOS version being much more superior both in terms of features and interface. The iOS version is not free of charge. Much like 'I.O.U - I Owe You', it is constrained to track IOU notes and not having a 'due to buy queue'.

Where it exceeds

It has a pleasant interface and a pragmatic approach towards grouping bills based on people. Has a good set of options and even allows adding pictures and geolocation to your bills.

IOU - by INEBAS

What is it

An application for tracking expenses, bills and IOU notes.

Where it falls short

It is only available in the iOS App Store and is not free of charge.

Where it exceeds

It has a beautiful and pleasant to use interface. Tracks lending and borrowing money.

Conclusion

In short, if one is looking for an IOU application, the options are endless, but what sets me into still want to create yet another IOU application is the fact that although all applications mentioned above (even those not mentioned), may do their job quite well, they fall short in the most fundamental issue, they do not solve my problem.

The problem

All applications tested and tried during the research time for IOU are consistently ignoring the workflow of the user and simply focusing on a more general approach to bill sharing, if they were to narrow their scope to a smaller niche, then they would be able to appeal better to that given group.

Some did narrowed their focus, however, constantly towards restaurant bills sharing.

The main feature that sets IOU apart, is that fact that it allows users to set a desire to buy something, for example:

- If I am throwing a party, I may set up items to be bought, on a list, as my friends buy them, they mark the items as bought along with how much they spent.
- If I finish the milk at home, I may add it to my housemates list, perhaps another housemate will notice it when strolling around the supermarket before going home.

IOU is not only a maths tool, but also, a communication mechanism that seamlessly integrates in the users day-to-day activities.

How does IOU approach it

The way I envision IOU usage is by being an app that is checked every time one goes to the supermarket or remembers something to be bought for the weekend party or drinks the last bit of milk from the fridge. Once it becomes a habit to use it, you no longer need to make groceries lists or ask anyone what is missing for the party or for the house in general.

In addition, IOU's data-feed happens in real time, being always connected via a WebSocket, this means users do not need to trigger a page refresh. When the data changes, the screen updates automatically.

Finally, having Facebook integration, allows users to simply refer to IOU as an extension of an

application they are already using. Making the eco system simple and familiar.

Some aid literature

- AngularJs from Novice to Ninja, by Sandeep Panda, is the single book that has helped me the most getting started in the world of Angular.js, explaining in a simple and effective way the various components that compose the eco-system.

- Mastering Web Application Development with AngularJS, by Pawel Kozlowski and Peter Bacon Darwin proposes a by feature approach to structuring Angular.js apps, this is gradually becoming the de facto structure for most Angular.js applications.

- Hackers and Painters, by Paul Graham, although not directly related to the work done towards IOU, when feeling overwhelmed by a technology or exhausted to the point of giving up, reading about Paul's experiences at ViaWeb uplifted my motivation and placed me back in the right track.

- Both the Ionic and the Firebase blog have been essential in this journey too.

Chapter 3

Research and Development Method

Chapter two described the problem domain, this chapter explains the rationale behind the technological decisions taken in order to achieve the best results in the implementation of IOU.

The functional requirements

IOU is mostly an online application, user data gets saved in the server and devices being used to access the data should be ubiquitous, that is, accessible from both a native mobile application or from a web application running on a browser.

The application should only allow login and registration via Facebook and the presence of WebSockets is a big plus, so that users do not need to refresh their views in order to see the most up-to-date information, much like a chat room.

Users should be able to create an unlimited amount of lists, on which they can add both members and products. Each list will also have a name and an icon associated to them. Both of which are editable. A list can only be viewed, edited or closed by their members.

Once a product is added to a list, it should be treated as an item that is still due to be bought. Any user from any given list can 'buy' those products. Adding or editing the price and name of any product, both bought and due to be bought products, can be done at any time. There should be a timestamp for each bought product stating the last time it was edited, in addition, any bought product that has been edited should be flagged as edited.

Every list should also have a members view, where more members can be added and it should clearly state the breakdown of how much each member owes or is owed in relation to the user currently logged in.

The main list view should display the current user overall balance, that is, taking into account all lists he/she belongs to. Within the single list views, the overall balance presented should be constrained to the current list.

The Login page, Terms and conditions of usage an Privacy Policy pages should be open to public access, however, all other pages should be private.

Only users that accepted the Terms and conditions of usage can register and login.

A left navigation menu should be present on all private pages, showing the picture and name of who is logged in as well as links to the home view, the terms and conditions view, the privacy

policy view and a logout.

Upon logout, all local storage data should be cleared, the user should be logged out from the application and redirect to the login page.

A go home button should be present on all screens to aid navigation.

The main action of each page should be presented in a block button in the bottom of each page.

The backend technology choices

There is no doubt that to achieve it, data should be provided by means of a Web Service Restful API, where the endpoints expose data in a convenient JSON format to be consumed by either a native mobile application or from a web application running on a browser. However, the options on how to implement such structure are endless. Below is a list of some of the available choices highlighting their benefits and constraints:

- PHP backend using Laravel:

  - Pros: This would've been the easiest solution, Laravel is a very flexible framework that provides powerful and yet simple to use features. It's routing system could easily cater for the necessities of IOU, queues are easy to integrate and packages such as socialite can take care of the Facebook authentication.
  - Cons: Although there is a possibility to use other databases, MySQL is still the predominant option within the PHP world. WebSockets are rather difficult to integrate. Even though HHVM can compile PHP code to C, it still lacks the extra speed boost necessary for a real time apps.

- Node.js:

  - Pros: Works really well with noSQL databases and WebSockets. Frameworks like Sails make it really easy to create and expose a Restful API and it performs quite well in terms of speed.
  - Cons: Not as easy as with PHP to find a suitable free server to start the work.

- Firebase:

  - Pros: No need to write an API, store your data on Firebase, set up your security rules and you are ready to go. Full support for WebSockets and social media integration, such as Facebook. Fast speed from noSQL databases and the entry plan is free of charge.
  - Cons: If you want to process anything other than data, such as images and payment, you will still need a backend system other than Firebase. As your application expands in number of users, plans become more expensive.

The frontend technology choices

For the frontend app, the choices were more limited, that is, in order to run a native application on a phone, one must develop constrained to the architecture he/she is writing if for. For example, Java is the native environment for Android apps and iOS developers have a choice between Swift and Objective-C.

Another option was to create a hybrid phone application, that is, write it using common web application technologies such as HTML, CSS and JavaScript, then compile it using either PhoneGap or Apache Cordova to run inside the target architectures web view. As a side note, PhoneGap is a branch from the open source project Apache Cordova administered by Adobe.

The last option was to use Titanium AppCelerator, which sits somewhere in between a native and a hybrid application, Titanium exposes a JavaScript API and converts them to the native function calls to the target architecture on compiling time.

Despite having to choose which frontend technology is best suited for IOU, one more element had to be considered. That is, the addition of a framework that would allow faster development and provide amenities such as high level abstractions for the platform and testing facilities. This is specially important, since the app has to run on more than one platform, including web.

At this point, the idea of using either a native application solution or Titanium, were beginning to fade, native apps would be constrained to one architecture only and therefore, have to be re-developed for every other platform. Titanium, solves that problem, but constraints the developer to a rather cryptic API that does not look like JavaScript.

That's when I came across Ionic, Ionic is a frontend SDK for developing hybrid mobile apps. Frameworks such as Lungo and Topcoat, do cover part of what Ionic does, that is, they provide consistent widgets that can be used interchangeably throughout different platforms. However, Ionic goes the extra mile, not only providing frontend widgets, but also exposing an API that wraps Apache Cordova and a tight integration with Angular.js, making it the ultimate production tool when the subject comes to developing hybrid phone applications.

The decisions

The final choice was to stick to Ionic for the frontend and Firebase for the backend.

The main rationale behind this decision was how both technologies combine and complement each other. Firebase provides a driver that integrates with Angular.js, converting it into a true 3-way data binding tool.

Angular.js is a tool backed up by Google which aims to structure JavaScript apps and impulse the development of applications that otherwise would be impossible to develop or to the very least take too long.

Besides, Angular's Eco System boasts a vibrant community. Integration with Node.js workflow tools such as Yeoman, Bower and Grunt are a breeze, E2E and Unit testing tools options are vast and Directives truly extend the vocabulary of HTML.

It still falls short in a sense that the application is not native, that is, some of the transition effects may not be as smooth as those provided natively and despite Firebase being a really interesting backend solution, in future, if the app needs to consume more powerful features, such as image resize and integration with a payment gateways, although in their pipeline for future development, Firebase will not yet suffice.

Nevertheless, the combination of those tools have been proven to be right for IOU, future improvements will dictate what other technologies will be used.

The diagram below represents the communication between the clients and the server. Every request that goes to Firebase must first satisfy the security rules. The double arrows represent the WebSockets:
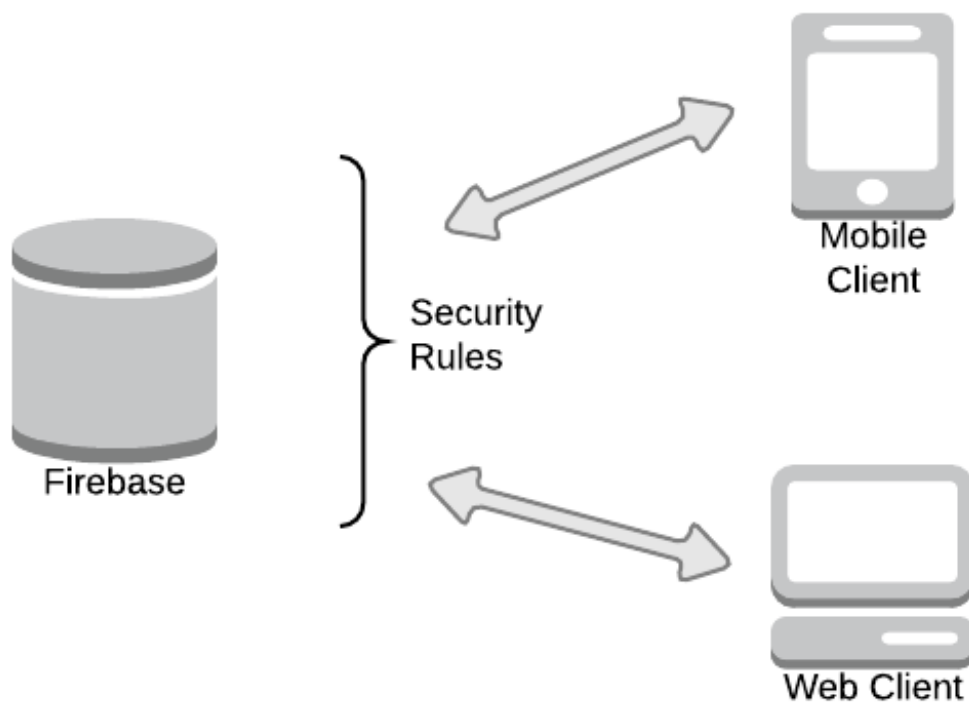

Fig 1 - WebSockets architecture

The newer technologies

After the technologies that were used in this project were decided, two notable alternatives arisen,

although not necessarily a better option, exploring the following will be essential in future implementation decisions:

- React.js: This is a library created and maintained by Facebook, as opposed to Angular.js, which uses HTML as its' templating language, React relies on a virtual DOM, that is, an abstraction of the DOM is created within JavaScript, allowing the app to to update the views in real time, without the necessity of triggering an expensive digest loop. This in turn seems to resolve the issue some of the Ionic apps suffer from in terms of slow DOM updates that make the app look and feel sluggish.

- Monaca: This is a paid framework, that has a limited features version available free of charge. Much like Ionic, Monaca uses Angular.js to power the application API and seems to have a very similar set features as Ionic, the only plus side being that it integrates Crosswalk in all apps by default. Crosswalk is the latest Chromium version, that when compiled along side a hybrid mobile application, renders animations much more gracefully than the default Android Web View. Of course, Ionic apps can also be packaged with Crosswalk, but it does not come free of some configuration overhead.

The workflow

Now that the technologies have been chosen, an efficient workflow must be devised in order to maximize production and generate good quality builds. Below is a non exhaustive list of the technologies used for this application:

- Git
  - Git-Flow

- Node.js
  - Cordova
  - Yeoman
  - generator-ionic
  - Bower
  - Angular
  - Angular Animate
  - Angular Local Storage
  - Angular UI Router
  - Firebase
  - Angularfire
  - Ionic
  - Firebase
  - ngCordova
  - Grunt

- autoprefixer
    - contrib-compass
    - contrib-concat
    - contrib-cssmin
    - contrib-htmlmin
    - contrib-jshint
    - contrib-uglify
    - contrib-watch
    - ng-annotate
    - ng-constant
    - usemin
    - wiredep
    - stylish

- Ruby Gems
    - SASS
    - Compass

- Android SDK
- iOS SDK

In addition, the Cordova in-app-browser plugin has also been installed for both Android and iOS platforms in order to aid Facebook oAuth login with Firebase within a pop-up window.

Git and Git-Flow

Git has been the version control system of choice for this assignment. Git, originally created by Linus Torvalds, despite being relatively new, is now a mature tool that as opposed to more conservative VCSs' takes a distributed approach. Without getting in too much detail, what it really means is that branches are 'cheap' to create, there is no need to deploy every branch to the remote repository, every developer keeps the relevant branches to himself in his local repository. Nevertheless, in practice, it is a good idea to have one central repository to keep track of the projects' progress. For this assignment, GitHub has been chosen as the origin.

In addition, in order to keep a reasonable workflow, this assignment adheres to git flow guidelines. Originally a blog post by Vincent Driessen:

Fig 1 - Git-flow workflow

Although not directly relevant for this project, as it has been developed by only one person, Git-flow provides a mean to keep the repository organised ensuring different developers can

collaborate in different points of a project lifecycle. Should this project ever expand into a bigger application, new developers will be able to collaborate immediately, without too much overhead. The main selling point for implementing Git-flow, is this instance, was so that commits happen in appropriate branches and that Releases and Hotfixes are tagged appropriately. To ensure that the branching model follows the workflow correctly, Vincent has written a Git plugin that facilitates adherence to his proposed model. It can be found in the repository below:

https://github.com/nvie/gitflow

This project uses the above plugin and adheres to Vincets' model.

Node.js

Despite being originally a backend tool, it's powerful package manager (NPM), has now been proven to be an excellent tool to manage any JavaScript project, being it a backend, frontend or hybrid application. Since this assignment relies on Firebase as the backend technology, Node.js has been used to manage the frontend assets.
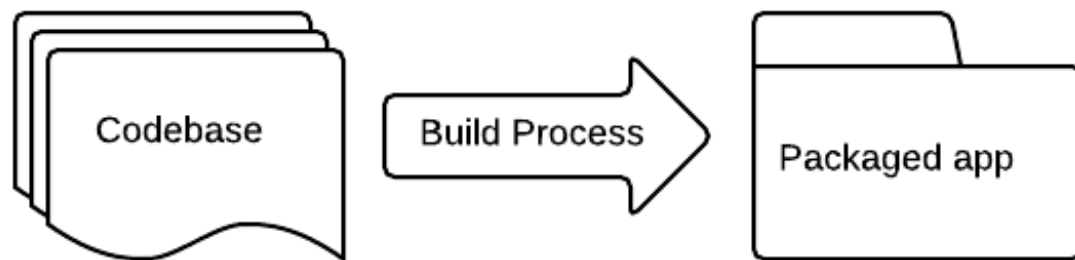
In a nutshell, three tools were used:

- Yeoman: Used to scaffold the app, Yeoman is an scaffolding tool, created and maintained by Google, whereby developers can share their workflow with others. The generator used for this application was created by Diego Netto, his generator is recommended by the creators of Ionic, the framework of choice for this assignment.

- Bower: Bower: Bower is being used to manage the frontend CSS and JavaScript assets. It takes care of all dependencies allowing the Git repository to be slimmer, since the dependencies do not need to make part of it. Furthermore, this project uses Wiredep, a Grunt task that triggers every time a new bower dependency is added. And automatically injects the dependency within Usemin blocks. Of course, there is a chance this process may fail, since not all bower dependencies are developed in a way that Wiredep understands, however, as a rule of thumb, all bower dependencies must be added using the --save flag, for example:
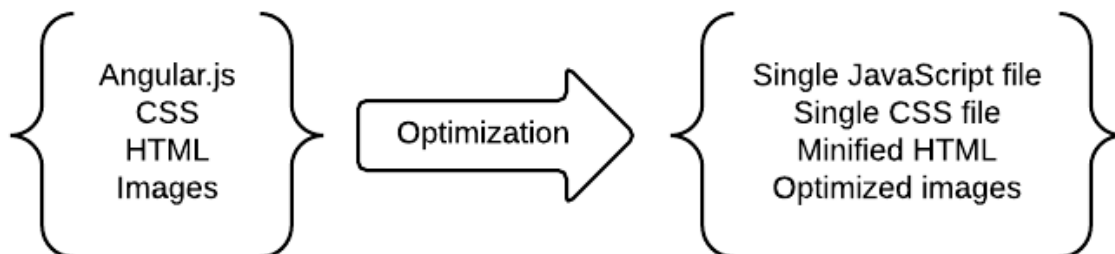
  $ bower install bootstrap --save

  - Since we used the save flag, our bower.json should now contain a new entry, in this case 'bootstrap' and 'jQuery' (Bootstraps' dependency), and since bower.json has changed, our watcher must have triggered Wiredep, which in turn must have added the dependencies to our index.html. This brings us to the second rule of thumb, one must always check that Wiredep has added the dependency appropriately in the HTML file.

- Grunt: Grunt is used as the overall manager for all the tasks running. Several watchers are running simultaneous and a complex build process is happening involving concatenating, obfuscating and minifying code. All this adds up to less HTTP requests for the final

application and a more 'machine ready' production code. In order to keep up with a good coding style, a JSLint task has also been incorporated and runs every time a JavaScript file is modified, it runs a series of strict checks and outputs its' results within the Grunt-CLI. The heuristic behind JSLint is to always declare a 'use strict' statement at the top of every JavaScript file, this ensures that the most strict set of rules will be applied for all checks. The diagram below represents the building process, whereby the left side is the Git repository that contains the application code and the right side is a replaceable built result:
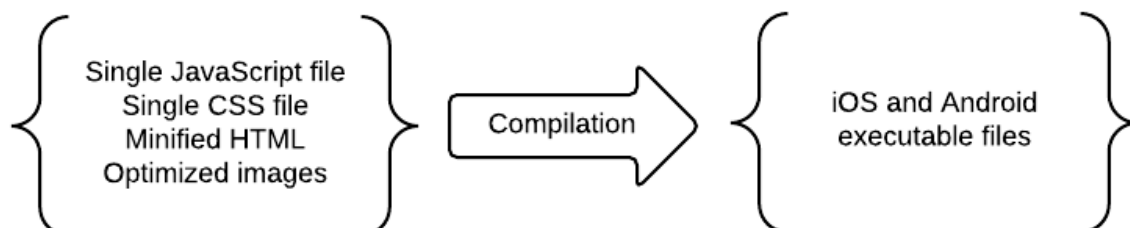
Codebase → Build Process → Packaged app

Fig 1 - Build Process

The code is then optimised:

{ Angular.js CSS HTML Images } → Optimization → { Single JavaScript file Single CSS file Minified HTML Optimized images }

Fig 1 - Optimisation Process

And finally compiled to the target mobile platforms:

{ Single JavaScript file Single CSS file Minified HTML Optimized images } → Compilation → { iOS and Android executable files }

Fig 1 - Compilation Process

20

Angular js

Developed and maintained by Google, Angular adds several advantages to the development process of JavaScript applications. The four main points to mention are:

- Two way data binding: Angular has models that bind to the view via a $scope. Every time a model changes its value, the view is updated. This reflects into shorter code that does not rely on DOM structure and manipulation. Freeing the developer to concentrate in the applications' logic rather than its' formatting.

- Directives: When DOM manipulation is absolutely necessary, Angular provides directives, whereby developers can write reusable pieces of code that operate in a set of data in combination with the DOM structure. Throughout this application, a reusable directive has been used to cater for the search feature.

- Structure: Angular adds structure to an application. Although commonly following an MVC architecture, Angular calls itself an MVW framework, which literally stands for Mode-View-Whatever, it lays out the guidelines, but leaves it up to the developer on how to best implement them. All applications typically have an entry point, called the run method; Templates can be decorated with directives and expressions, further DOM manipulation can be encapsulated within reusable Directives; Shared functionality can be created as services, which in turn can be configured during different points of the applications' lifecycle; Code behind view blocks can be stored in controllers; application state can be managed via routes, filters provide formatting and filtering functionality; And it can all be grouped within modules, which are very similar to namespaces, other than modules will also declare their external dependencies and provide the methods to implement all the above.

- Dependency Injection: As mentioned by Brian Ford in a Google developer conference, dependency injection can be defined with one sentence: "Don't call me, I'll call you". What he means is that it is a design pattern that implements inversion of control, one of the SOLID principles. Instead of instantiating objects within our methods, we simply declare the dependencies in our methods' signature, Angular will recognise the services a module depends on and will automatically inject the correct instantiated objects when that method is called during run time. This makes the methods more testable and more decoupled from one another, hence, immediately providing higher adherence to SOLID principles.

In addition, this application makes use of an Angular service called Angular Local Storage, this dependency provides a simple API to interact with HTML5 local storage for different clients. It will also fall back to cookies, should Local Storage not be available in the current system.

Ionic and Cordova

In the core of this entire application is Ionic, a Framework for building hybrid (native/web)

applications for smart devices (phones/tablets). Finding and deciding on using Ionic was the single most laborious research for this project.

Prototypes were written using different frameworks, most of those frameworks would either only have a set of UI interfaces or provide the low level integration required by Cordova, such as Topcoat and the Angular-Cordova bridge respectively. Nonetheless, Ionic that provided all the necessary tools and workflow to accomplish the final task in one framework.

Being backed by Drifty Co. a company that works closely with Google, and having their CEO, Max Lynch, writing the Angular.js port for Material Design, places Ionic in a very respectable position.

Despite all that, Ionic created and makes use of ngCordova a tool that seamlessly integrates Angular.js with Apache Cordova, making it easy to call the low level Cordova API without compromising on the application architecture.

It also provides a beautiful set of predefined UI elements and Directives that package them together for advance usage. Plus, Angular-UI-Router is already packaged in the bundle, so nested routes and states can also be implemented.

In addition, Ionic keeps an up-to-date forum and documentation, not to mention a nightly built code sample library on Code-Pen.

Ionic has also recently released a service called Ionic View, which allows developers to view and share their application on different mobile platforms without going through the hassle of deploying their app in any app store. And will soon release a push notification system, now closing the loop with all necessary services to develop complex hybrid mobile applications.

Firebase

Combining noSQL with a layer of WebSockets, comes Firebase, it is not only a database system in the cloud but also a full eco system that caters for security annotations on top of your data structure and oAuth integration with third party providers such as Facebook and Google+. This project integrates Firebase by means of a library called Angularfire. Angularfire is a library written by the Firebase team that aids easier integration with Angular, converting the application into a Three way data binding tool, when data changes in the database the view for all clients is immediately updated, real-time. Combining a top notch support team and being now part of Google, I am confident Firebase will soon become one of the biggest entities for backend solutions.

Chapter 7 develops further the obstacles imposed by Firebase, noSQL in general and the techniques employed to overcome those limitations.

SASS/Compass

This projects makes use of CSS pre-processors in order to simplify the stylesheets necessary to render the application. A Grunt task then aids it further, by means of running the compass watch task necessary to transform the SASS code into CSS. It also makes use of another task called Autoprefixer, which in turn prefixes CSS3 declarations with the browser vendors annotations. For example, as a developer, I may wish to use border radius for my buttons, my CSS class may look like the following:

```
.btn {
  border-radius: 5px;
}
```

After running Autoprefixer, the browser vendors will be added like so:

```
.btn {
  -webkit-border-radius: 5px;
    -moz-border-radius: 5px;
         border-radius: 5px;
}
```

This process runs automatically whenever a SASS file is changed throughout the application, providing, of course, that the Grunt serve tasks is running.

In addition, Compass, provides a full library of Mixins, aiding further with common CSS patterns. In the SASS world, CSS development is seen much like Object Oriented Development, that is, a class can be extended, variables can be created and even nesting is allowed. Mixins are simply a piece of reusable CSS, that can be mixed into any other CSS class.

The SDKs'

Firstly, in order to develop an iOS application, one must be developing on a Macintosh computer, this is necessary since the SDKs' come bundled with XCode a Mac specific application. Providing that, the set up for iOS is rather simplistic, one only needs to download and install XCode, accept the terms and conditions and ensure the command line tools are installed and available.

In the other hand, the Android SDK can be proven to be slightly more difficult to install and configure. The first dependency to be installed is the Java JDK, almost all the other dependencies can be met by means of installing the Android Studio, an excellent IDE built on top of the JetBrains IntelliJ. However, if you wish to develop in the terminal there are a few other things that must be made available on your path:

- The SDK platform-tools
- The JAVA_HOME variable
- Ant

- ADB (The Android Debug Bridge)

With all that in place, Android development should be possible, however, if that seems like too much overhead, Ionic offers a Vagrant box already preconfigured to develop Ionic apps for Android, it can be obtained from the link below:

https://github.com/driftyco/ionic-box

Vagrant is a Virtual Machines manager that integrates with Oracles Virtual Box. Both tools are available free of charge. Furthermore, it allows port forwarding and directories mapping with the local development system. The Vagrant box above, therefore allows Android development without the need to install and configure an Android environment.

Chapter 4

Data Findings

As a result, three applications were developed for this assignment, an Android application, an iOS application and a web application. In summary, all apps do not diverge from the original source code, they have simply been compiled for different platforms.

Unfortunately, by the time of this writing, the iOS application is still under approval by Apple, please keep on checking the live version of this documentation at the link below as it will be made available as soon as the app goes live:

http://iou.rocks

The Android application can be downloaded and installed from the link below:

https://play.google.com/store/apps/details?id=com.jbonigomes.IOU

A web version of the application has also been made available at the following link:

https://ioutest.firebaseapp.com

It is important to note that since the codebase for the web app and the mobile apps are the same, a greater amount of care has been taken towards the mobile apps, and therefore, the web application lacks a slight level completeness. The main issue lies within the fact that page refreshes may render the application unusable, should that occur, one must click the home button in the top left of the application then refresh the page again. This issue has been added to the backlog and once the web app is deemed to be sufficiently good it will be published in a new URL that does not contain the word 'test' in it.

Chapter 6 develops further towards what is recommended for future improvements, however, an archiving system that was originally planned could not be implemented yet, this has however been added to the backlog and will be developed in a future version. To cover the lack of an archiving system, for now, users have the option to completely delete a list, IOU recommends to only delete a list once all members are satisfied that the amount owed to each other meets their expectancy.

Information Architecture

The information architecture for IOU takes in consideration the overall user experience aiming to logically group sections that relate to each other and layer them down based on importance.
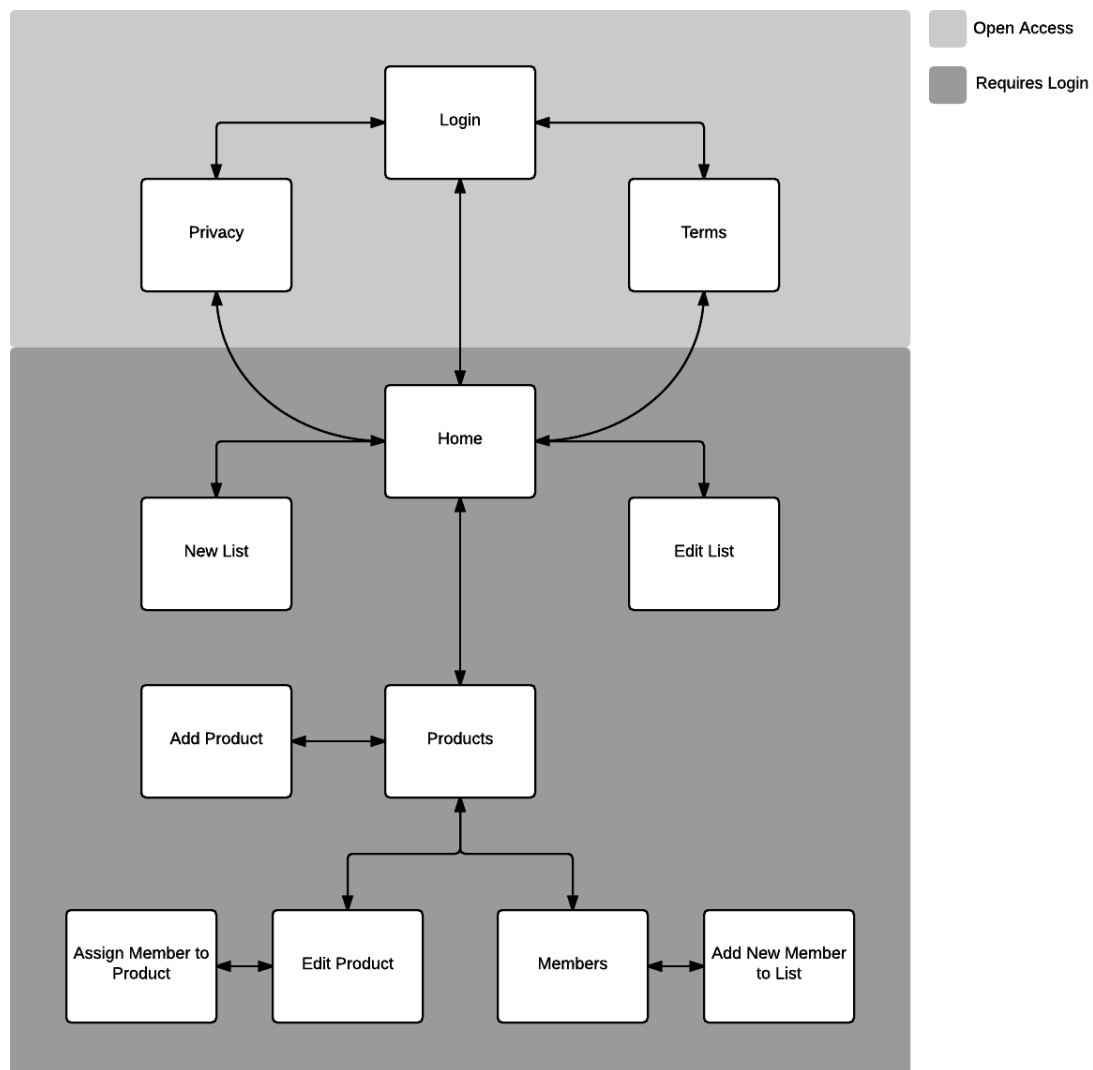
Fig 1 - Information architecture

Although not explicitly demonstrated, all states that require login present a link that goes straight back to the home page as well as a quick access slide menu that displays information about the currently logged user and present links to the home page, privacy page, terms and conditions page and logout.

Angular Architecture

The application dependencies look like the following diagram:
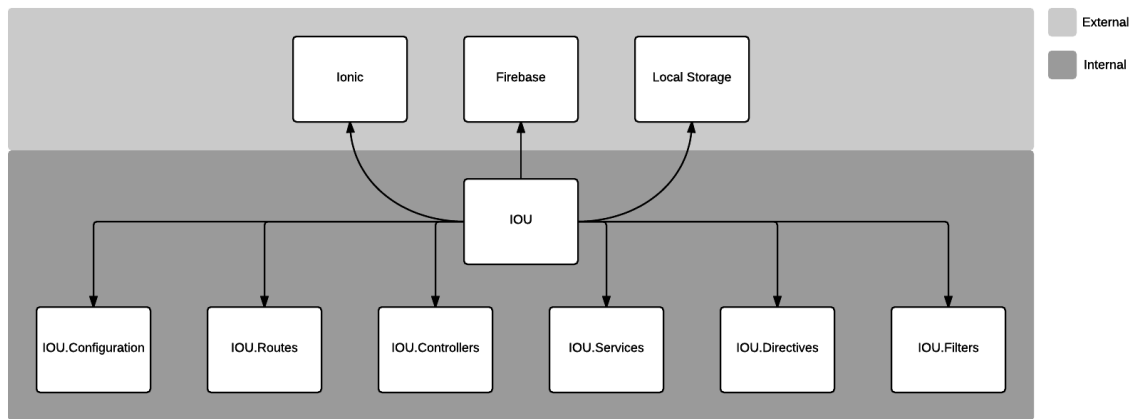
Fig 1 - Angular architecture

Centrally located is the IOU module, where the applications' run method is located, this module declares the external and internal dependencies that together form the final product.

Use cases

In addition to the information architecture depicted above, this section develops further into the user journey analysing each scenario a user may encounter.

The use case diagrams below assumes that 'Common Links' refers to the following links:

- Home page
- Terms and Conditions page
- Privacy Policy page

Also, the diagrams do not explicitly show that for every stage that requires a login, a logout link is also provided.

Before having access to the application, the user must first accept the terms and conditions and login.
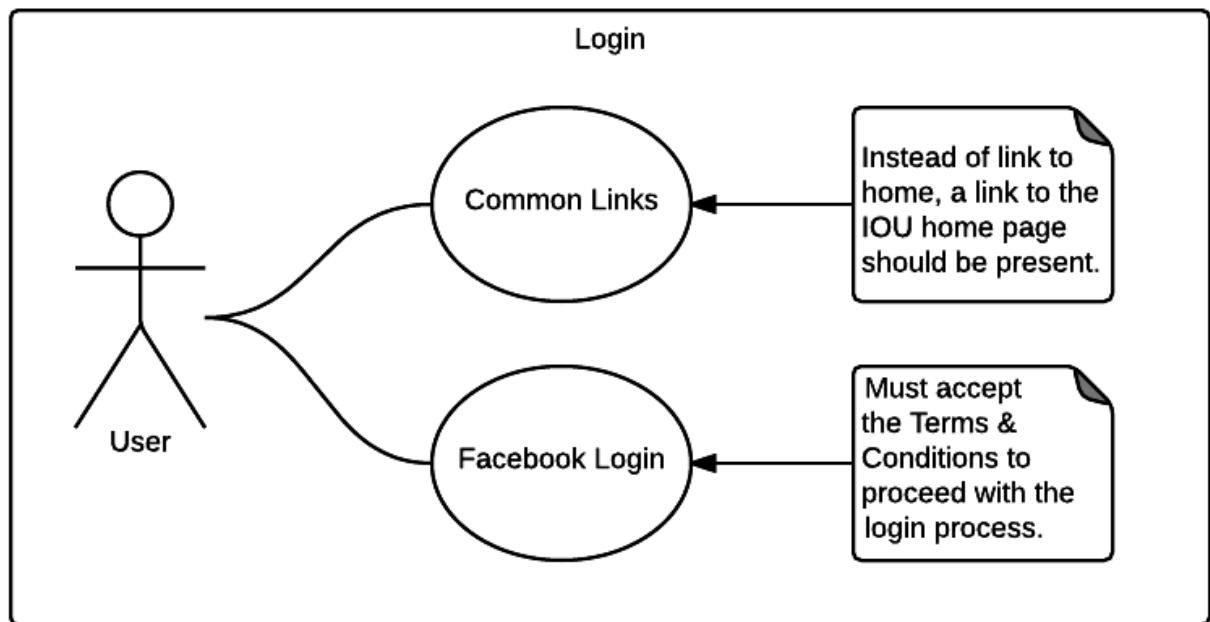
Fig 1 - Login Use Case

The home page shows all lists that the current logged user belongs to. Allows creating, editing, deleting and filtering lists. Each list also displays their total in relation to the logged user. The top of the page should give a summary of the whole total between all lists in relation to the currently logged in user.
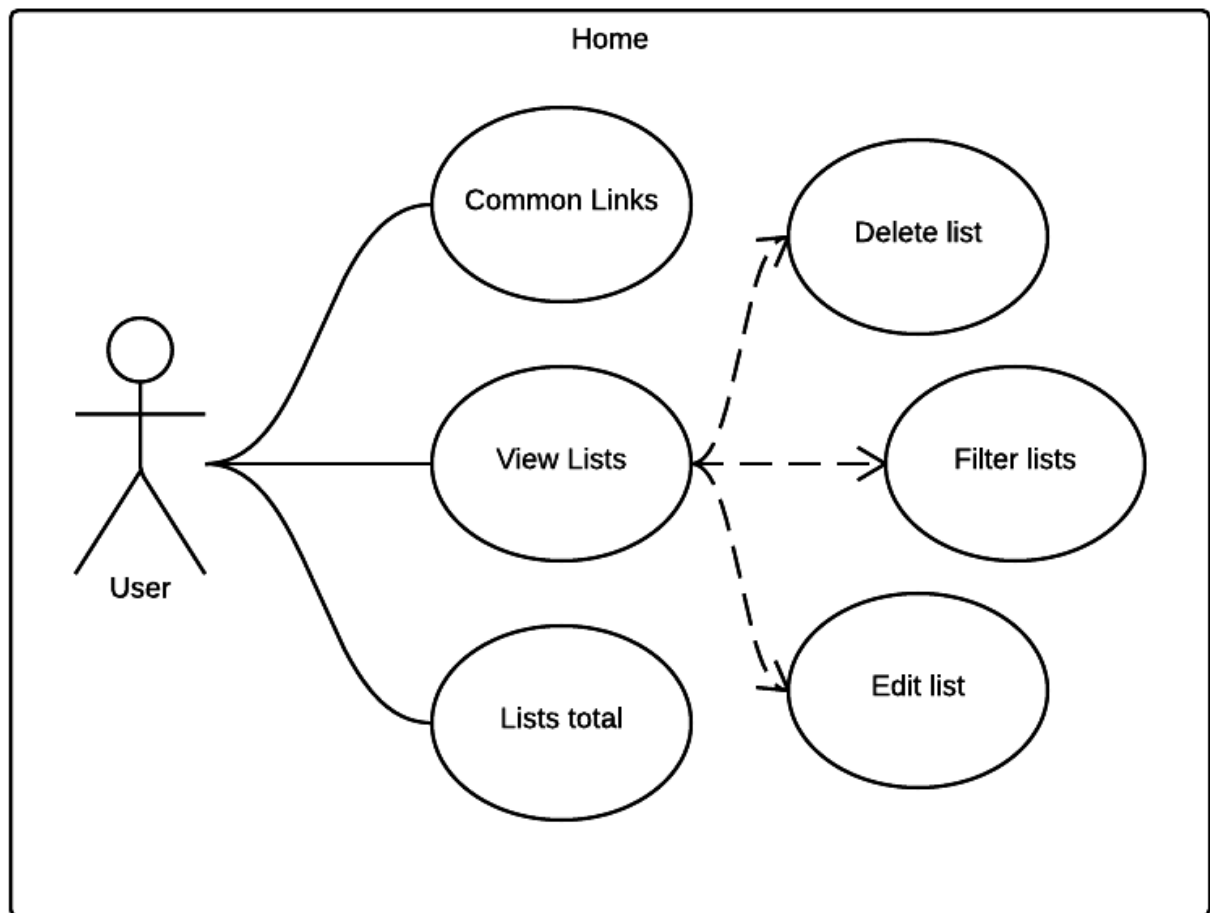
Fig 1 - Home Use Case

The products view should display a list of all products that are due to be bought as well as all the products that have been bought. It should allow creating new products, buying products and editing products. It should also display a total for the current list in relation to the logged in user.
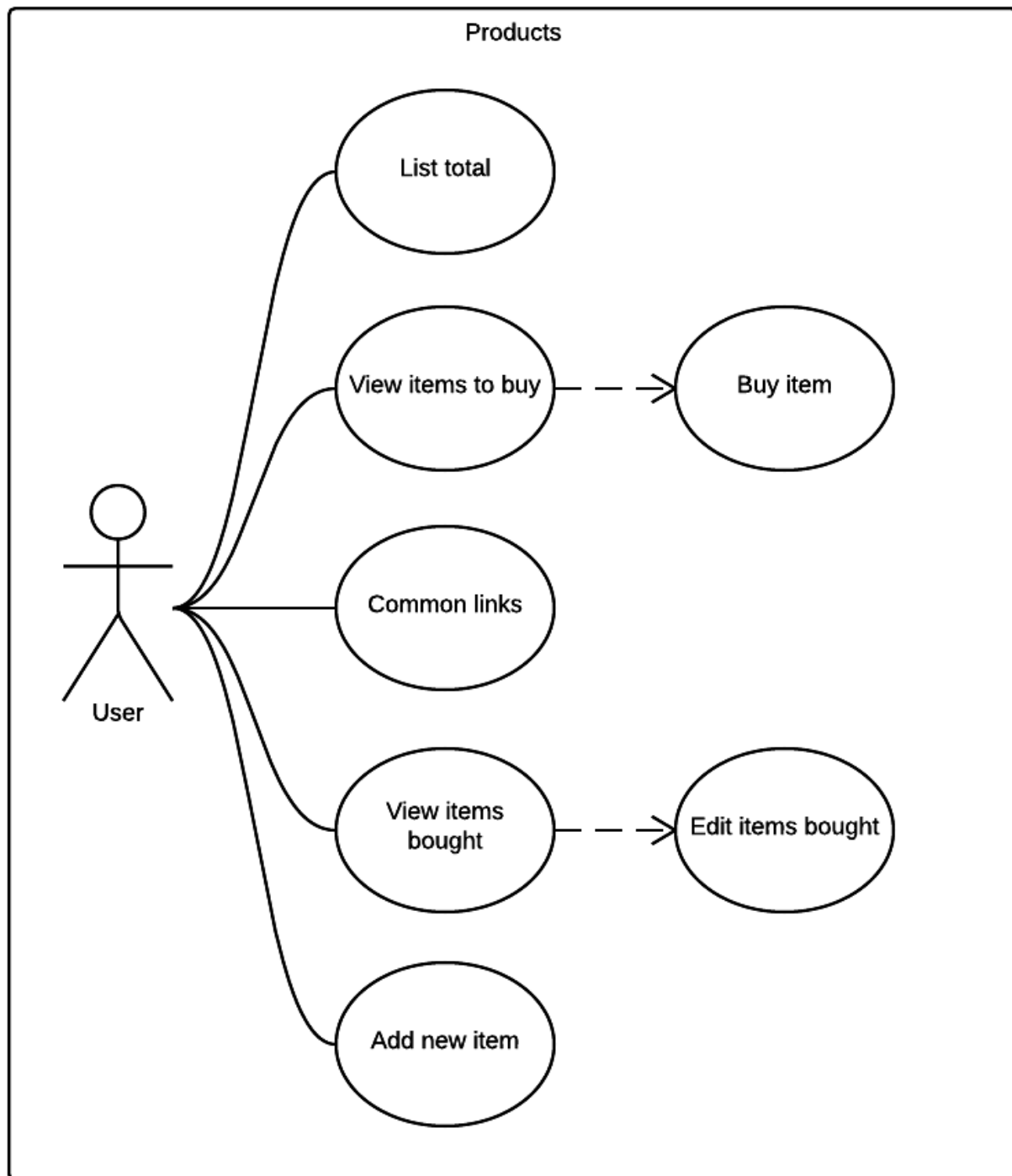
Fig 1 - Products Use Case

The members view should show all members that belong to the current list. Each member should be displayed with a calculated field of how much they owe or are owed in relation to the currently logged in user. The same list overall total that appears in the products page should be displayed at the top of the members page too. From the members view, new members could also be added.
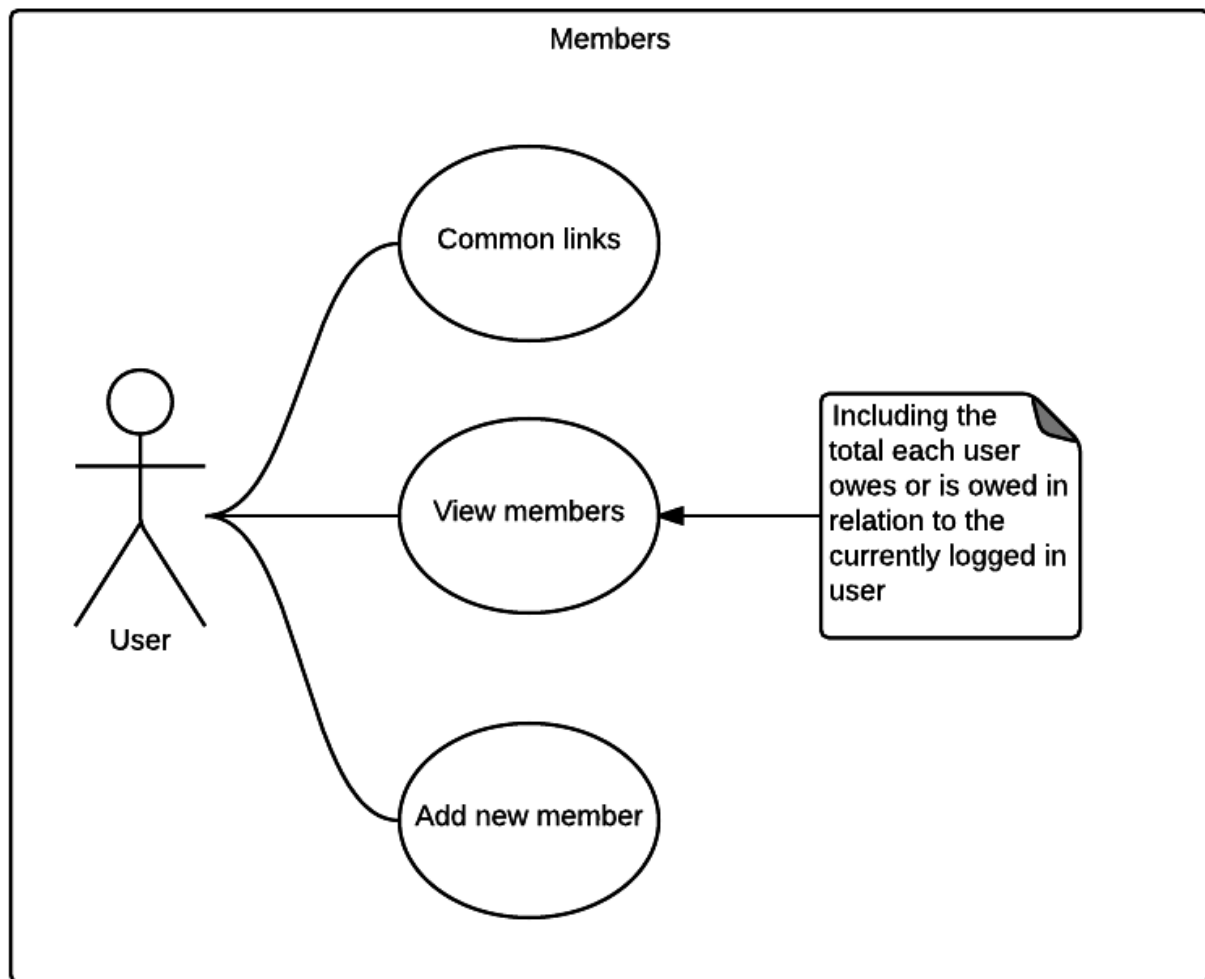
Fig 1 - Members Use Case

The user workflow for using the IOU app should look like the following:

- When a user remembers that something has to be bought, a new item should be created in the products view.
- When this newly created product is bought in the 'real world', the user should 'buy' this product within IOU, by means of adding a price to the given product.
- Once all members decide it is time to close the list, they should check the members tab to know how much they owe each other.
- Finally, in the 'real world' they should exchange the proposed amounts then delete the list from IOU.

Application Structure

The following is the file structure powering IOUs' source code, this is the default structure created by Diego Nettos' Yeoman Generator:

```
├── Gruntfile.js        - Configuration of all Grunt tasks
├── package.json        - Dev dependencies and required Cordova plugins
├── bower.json          - Lists front-end dependencies
├── config.xml          - Global Cordova configuration
├── .gitignore          - Best practices for checking in Cordova apps
├── resources/          - Scaffolded placeholder Icons and Splashscreens
│   ├── ios/
│   ├── android/
├── app/
│   ├── index.html      - Main Ionic app entry point
│   ├── lib/            - Libraries managed by Bower
│   ├── scripts/        - Custom AngularJS Scripts
│   ├── styles/         - Stylesheets
│   ├── templates/      - HTML views
├── platforms/          - Targeted operating systems
├── plugins/            - Native plugins
├── hooks/              - Cordova lifecycle hooks
├── merges/             - Platform specific overrides
├── coverage/           - Istanbul reports
├── test/               - Unit tests
│   ├── spec/
├── www/                - Copied from app/ to be used by Cordova
```

Chapter 5

Analisys Evaluation

After completing the IOU application, one major concern was how smooth the animations and transitions would perform in a real device. Up to that point, the application was only tested using emulators.

Surprisingly, as opposed to the bad reputation of hybrid apps performance, IOU responds quite well on Android devices.

Unfortunately, by the time of this writing, Apple is still reviewing IOU and, I am, therefore unable to comment on its' performance yet.

Testing

IOU has been tested with several users, each of which presented their own problems, most of which were not bugs but simply different points of view on how the workflow should be approached.

The main goal of getting users to test the application early, was to not only collect early feedback, but also to ensure that the application validates and handles user input correctly, as well as ensure that the calculation algorithms work correctly.

The scenarios tested were:

- Login

    - Can log in with valid Facebook credentials
    - Cannot log in with invalid Facebook credentials
    - Must accept the terms and conditions before login
    - User can read the terms and conditions
    - User can read the privacy policy
    - User can visit IOUs' home page

- Home

    - Only the lists that the currently logged user belongs to should be visible
    - User can create a new list
    - User can delete a list
    - User can edit a list
    - Name and image

- An overall calculation of all lists in relation to the currently logged in user should be presented
- Clicking on a list should navigate to the products view of that list

- Products

  - A total of the current list in relation to the currently logged in user should be presented in the top of this page
  - User can create a new product
  - A new product should appear as a 'To buy' product
  - User can buy a 'To buy' product
  - User can edit products that have been bought
  - Owner
  - Price
  - Name
  - A product that has been edited, must display an '@amended' tag

- Members

  - A total of the current list in relation to the currently logged in user should be presented in the top of this page
  - A list of all members that belong to the current list should be displayed here
  - Each member should display a calculated field of how much he/she owes or is owed in relation to the current logged in user
  - A new member can be added to this list

- All

  - Pages that require a login should present common links:
  - Home
  - Terms and conditions
  - Privacy policy
  - Logout
  - Validations
  - Any price entered must be a numeric value that is bigger than 0
  - All string fields are required and must therefore be completed

Each scenario was tested using different inputs to ensure that the appropriate validations triggered correctly.

The future

Unfortunately, due to the tight deadline for this project, automated testing could not be

implemented, however, the set up for this to happen is already in place and tests will be written in future iterations.

The most exciting testing framework present is Protractor, and End-to-end testing framework developed specifically for Angular.js.

Protractor uses a Selenium driver to simulate users actions and ensure the expected outcome is met.

In addition, Mocha and Chai are already set up to perform the unit tests, those tests run using Angulars' Karma Runner and using Istanbul, produce static websites with the results and the code coverage stats.

Chapter 6

Conclusion and Recommendations

As a final result, this project delivered the planed basic set of features. Users are already using it and the project has been proven to be fit for purpose.

Although still being in the process of being approved by Apple, the Android version is already live and iOS users have access to the alternative web version.

Despite all that, the list below has been compiled from the feedback heard from users and will be addressed in future iterations:

- Send a welcome email when user registers for the first time
- Create an explanation video to be displayed in the app store
- Display message and disable all functions when user is offline
- Send a push notification when a user is added to list by someone else
- Improve overall look an feel such as adding a back button where relevant
- Make the app more snappy (this may include compile the add along with Crosswalk, a full version of the Chromium browser)
- Improve the web version, the current list and product id being edited should be saved to local storage in case the user presses the browser refresh button.
- have a loading icon when pages are being loaded with an overlay that does not allow users to interact with the application until its' state is ready

In addition, creating a concept of list owners may be a good idea, therefore a list can only be managed by whomever created it, instead of allowing all participants to take this role. In this case, only a list owner can:

- Delete products
- Archive/close a list
- Add support for different languages
- Remove a user from a list
  - If he/she did not buy anything yet, delete the user
  - Otherwise, have a warning message allowing the delete action to be canceled add state that removing this user will also remove all products he/she has bought

Some extra features could also improve the final product:

- Add different currencies
- Create an invite friends feature
- Add a users guide/help page within the app

- Allow archiving lists, so that a list is never deleted should users want to review it some time in future

In terms of the codebase, due to the small size of this application, files have been grouped based on how Angular qualifies them, that is, if it is a Directive, it goes to the directives file and so on. However, as the application expands, this pattern may no longer be sustainable. Grouping files by feature will be a better option as the app expands. More specifically, the file that would benefit from this pattern the most is the controllers file. This approach has been demonstrated by several experts in the field, including Pawel Kozlowiski and Sandeep Panda.

Simply as a proof of concept, proving that Ionic and Cordova can produce a hybrid mobile app to all major platforms, it may be a good idea to compile the application for both Blackberry and Windows phone.

Although already mentioned in Chapter 5, implementing automated tests will be paramount in future versions of this app.

Finally, this project demonstrated that it is possible to create hybrid mobile phone apps using web technologies. Although fundamentally similar, mobile development comes with a different set of problems to be solved, most of which are technically challenging. However, completing this project gives me the confidence and enthusiasm to improve it in future and engage in new exciting development endeavours.

Chapter 7

Review/Reflections

This chapter reflects on the challenges encountered while developing IOU. Each section develops further on what those challenges were and how they were approached.

The IOU algorithm

The entire application would be meaningless if a good money splitting algorithm was not implemented. It seems money splitting algorithms are somewhat complicated to implement successfully. Moreover, IOU not only aims to split expenses in a fair manner, but also, optimise the change each participant has to pay, so that users do not need to exchange money among each other more than once.

The first step taken followed Steve Bobs' approach:

http://stevebob.net/iou/howitworks.html

This is only a pseudocode that had to be implemented in JavaScript ensuring that the concept works in a real life scenario.

The final algorithm can be found in the source code under 'app/scripts/services.js' under a factory called MembersWithTotal.

The algorithm works as follow:

```
calculate the list total
calculate how much each individual user spent in the list
calculate the list average by dividing the total per the number of users
calculate how much has been overspent, this is the sum of the difference be
tween the amount of money each member has spent and the list average
qualify the current logged user as a 'creditor', 'debtor' or 'even', a user
 is a 'debtor' if he/she spent less than the list average, a 'creditor' if
he/she spent more and 'even' if he/she spent the exact same amount as the l
ist average

if the logged user is a creditor
  for each debtor
    calculate a percentage owed to the current logged in user, this is the
percentage of the total overspent that the current logged user has spent
    based on how much the debtor has underspent, calculate how much he/she
owes to the current logged user by 'slicing' his/her percentage (calculated
 right above)
    also keep accumulating this number to present as a list overall total

if the logged user is a debtor
  for each creditor
    calculate a percentage the current logged in user owes to the creditor,
 that is, from the total list overspent, this will be the percentage the cr
editor has overspent
    based on how much the current logged in user has underspent, calculate
how much he/she owes the creditor by 'slicing' his/her percentage (calculat
ed right above)
    also keep accumulating this number to present as a list overall total

Otherwise, the logged user must have spent the exact same amount as the lis
t average and therefore does not owe or is owed any money, zero can be retu
rned as the his/her total
```

Arriving at this final algorithm required a successive amount of trial and error. Although not particularly elegant, and somewhat naive, after testing with several users, amounts and scenarios, I believe it calculates the totals correctly.

Following the above, the way the optimised change works is that it will only return one of three scenarios:

- An even list, which means, no actions for the user, he/she spent the ideal amount
- A creditor list, this user does not have to pay any money and is simply informed how much he/she is owed
- A debtor list, this user owes money to the creditors, the app will notify how much is owed to each member

There is still room for improvement over how this process work, most of which goes towards the efficiency and readability of the JavaScript implementation. Going forward, once the unit tests for this factory have been written, more effort will be applied in its' mathematical foundations.

Data architecture

Coming from a relational databases background, one of the most difficult mind set to adapt, was to start thinking in a noSQL way.

In noSQL, data is stored as key value pairs, which in turn, can hold nested key value pairs, in practice, data is represented as a JSON notation object that can be queried by traversing the tree structure.

The IOU app is powered by, Firebase, an implementation of a noSQL database, that although very fast and scalable comes with a few limitations, for example, it only allows a depth of up to 32 levels when nesting data objects and although data can be queried as arrays, it does not support storing data as arrays, all values must be strictly stored as JSON key value pairs.

From the Firebase documentation, the initial data structure was organised as the sample JSON object below:

https://www.firebase.com/docs/web/guide/structuring-data.html

```
{
  "lists": {
    "123": {
      "name": "Households",
      "image": "lightbulb",

      "members": {
        "444" : true,
        "555" : true
      }

      "products": {
        "567" : true,
        "891" : true
      }
    },

    "234": { ... },

    "345": { ... }
  },

  "products": {
    "567" : {
      "amended" : true,
      "date"    : 1412635438623,
      "name"    : "Potatoes",
      "owner"   : "444",
      "price"   : "28"
      "bought"  : true
    },

    "891" : { ... }
  }

  "members": {
    "444": {
      "name" : "Jose Gomes"
    },

    "555": { ... }
  },
}
```

In contrast with a more traditional relational data structures, there is not much going on in the sample above. There are three main collections (lists, products and members) and each hold a series of entries, represented by a unique key which in turn holds the values for that row. The only

extraneous notation is how the relations are declared, instead of foreign keys, Firebase recommends setting relations like so:

```
"members": {
    "444" : true,
    "555" : true
}
```

This may look intriguing at first, however, since Firebase does not support arrays in their storage, this is the nearest achievable way to represent a one-to-many relation.

Although easy to understand, this initial set up does not make use of the full potential from a noSQL database, within a key value architecture, the developer is not constrained to the limitations of data normalisation, the database can be moulded in whichever way it best suits the application using it. Besides, joins considerably deteriorate the performance of data processing.

The following blog post by Anant Narayanan, was the first stage in denormalising the initial concept:

https://www.firebase.com/blog/2013-04-12-denormalizing-is-normal.html

In conversation with Sara Robinson, Developer Evangelist at Firebase, during the a Google Developers Conference, it became clearer that a more nested structure would boost the application performance yet making it easier to perform queries that return meaningful data with less processing.

As a result, the second iteration architecturally looks like the following:

```
{
  "lists" : {
    "123" : {
      "name"  : "Households",
      "image" : "lightbulb",

      "bought" : {
        "567" : {
          "amended" : true,
          "date"    : 1412635438623,
          "name"    : "Potatoes",
          "owner"   : "444",
          "price"   : "28"
        },
        "891" : { ... }
      },

      "tobuy" : {
        "678" : {
          "date" : 1412635438623,
          "name" : "Steak"
        },
        "789" : { ... }
      },

      "members": {
        "444" : true,
        "555" : true
      }
    },

    "234" : { ... },

    "345" : { ... }
  },

  "members" : {
    "444" : {
      "name" : "Jose Gomes"
    },

    "555" : { ... }
  }
}
```

Since the IOU app differentiates products that have been bought from products that are still due to be bought, the first step taken was to break down products into two categories (bought and tobuy).

This, in turn, raised the awareness that both structures did not have the same attributes. Secondly, since a set of products belong to a list only, much like a one-to-many relationship, it was only natural to nest those two new objects within a list object, those two actions not only improved the speed to which data can be read, but also represent, if not, describe, the IOU app architecture.

Finally, during the last iteration and after coming to a realisation of how Firebase security rules work, it was decided that a collection of members was not necessary. This decision was made for two reasons, firstly, simply to make the structure cleaner, thus completely avoid joins and secondly because the objects are too simple, the only data stored is their Facebook id and their names. Besides, if a username is stored in the database, should the user update his/her name on Facebook, IOU will never be notified and will therefore always display an out-of-date username. The final data structure simply removes the members object completely:

```
{
  "lists" : {
    "123" : {
      "name"  : "Households",
      "image" : "lightbulb",

      "bought" : {
        "567" : {
          "amended" : true,
          "date"    : 1412635438623,
          "name"    : "Potatoes",
          "owner"   : "444",
          "price"   : "28"
        },
        "891" : { ... }
      },

      "tobuy" : {
        "678" : {
          "date" : 1412635438623,
          "name" : "Steak"
        },
        "789" : { ... }
      },

      "members": {
        "444" : true,
        "555" : true
      }
    },

    "234" : { ... },

    "345" : { ... }
  },
}
```

This final decision raises one last problem, now every time a use is displayed, an AJAX call must be made to Facebooks' Graph API in order to return a users' username. Although simplistic, this approach generates a non-obvious problem, when retrieving a list of users of unknown length, we have to wait for all Asynchronous calls to be finished before rendering the list.

Before going on details, it is important to clarify what a promise is. In Angular.js, most Asynchronous calls are treated as promises, this concept is not new, as mentioned in Wikipedia, the term promise was proposed in 1976 by Daniel P. Friedman and David Wise, and Peter Hibbard called it eventual. However, in the JavaScript world, it's implementation is still taking 'baby

steps'. In Angular.js, when a function is meant to return the result of an asynchronous call, an AJAX call for example, this call will be said to return a promise that will be resolved some time in the future. For example, one may wish to stipulate what should be done in the event of an error, or bind the data returned to a model in the event of a successful call.

Fortunately, Angular provides a service called $q, this service is packed with methods that aid promises handling. This project uses a method of $q called all(). The all() method receives two arguments, the first is an array of promises and the second is a callback function that will be executed when all the promises in the array are resolved, in addition, the data returned by each promise will be packed as an array and passed as an argument to this function. Taking this approach resolved the problem imposed when rendering a list of users, the code sample below has been used the the users:

```
// userswithnames refers to an array of promises
// created making sucessive AJAX calls to Facebooks'
// Graph API, based in the stored user ids'
$q.all(userswithnames).then(function(members) {
  // bind each member to the view
});
```

In addition, to minimise the number of AJAX calls, since the member that is currently logged in will have his/her data presented throughout most states of the application, during login time IOU will store the users information on Local Storage, hence minimising the usage of $q only to render the members list view.

Finally, although not closely related with the data structure, getting acquainted with handling asynchronous calls in JavaScript has been paramount to the success of this project. Although Firebase simplifies the data handling with Angular.js by means of using Angularfire, in addition to the $q problem described above, sometimes when data is updated outside the Angular application a new digest loop has to be triggered. This problem has been overcome by setting up a watcher in the root of the Firebase data architecture, every time data changes in Firebase, this watcher is notified and will then run a data refresh within the given controller.

oAuth

Despite simplifying the entire oAuth process with third parties, Firebase still requires some configuration in order to work as expected.

Since IOU uses Facebook authentication, the first step is to create a Facebook developer account, then generate a new Facebook app. This process will create an unique Facebook app ID and a secret key. Those two parameters must then be configured accordingly inside Firebase's admin panel.

The next step is to choose an appropriate firebase oAuth method, there are several options,

however, IOU is constrained to use the authWithOAuthPopup method, since it is supported both on all major web browsers and within Cordova's In-App-Browser extension.

The final step is ensure sufficient information about the users is collected from Facebook during the login process. this is done via a scope variable, IOU collects the basic public user information, such as the users email address, profile picture and full name, in addition, IOU will also ask permission to have access to the users friends list.

It is important to note that although IOU has access to the users friends list, Facebook will only return the list of friends of a user that also happen to be an IOU user. This limitation exist to avoid abuses such as spamming, but can, however be quite confusing for a first time Facebook app developer.

Once all this is in place, Firebase will take care of all the rest, such as token management and providing a consistent API for logging users in and out of the application as well as providing basic information about users and authentication status.

Security rules

Firebase provides a set of annotations that follow the data structure of the application. In addition to authentication, those annotations dictate who has access to read and write specific sets of data.

It is important to note that the security rules are not part of the application code, those annotations are set up in the server via Firebase's admin panel and are therefore not visible even to those who have access to the application source code.

Once a user is logged in, Firebase will assign a variable named auth, which is null by default, to an object containing the logged user information. Rules can then be written to ensure that a given user does not have access to data he/she should not have access to.

Much like the Firebase structure, those rules are written in a JSON format and follow the same structure of the Firebase data. For example, in order to control access of a list, so that only users that belong to a list can view and edit it, the following rule may be used:

```
{
  "rules": {
    ".read": false,
    ".write": false,
    "lists": {
      "$list" {
        ".read"  : "(root.child('members/' + auth.uid).val() === auth.uid)"
,
        ".write" : "(root.child('members/' + auth.uid).val() === auth.uid)"
,
      }
    }
  }
}
```

In order to test security rules, Firebase provides a tool in their admin interface which simulates data reads and writes.

When developing more complex rules, Firebase has created a tool called Blaze, which compiles Yaml formatted data into workable security rules and allows usage of reusable variables and functions.

Play and App stores

Last but not least, as opposed to web applications, apps published in the Google Play Store and Apple App Store, must be digitally signed. Each platform operates in a different way, however the principle is the same, the developer generates a key that is used to sign the first version of the application. Subsequent updates to the same will have to be signed with the same key.

Apple makes this process easier within Xcode, the generated key is automatically added to the developers keychain and the signing process is carried out seamlessly within Xcode.

Android has a slightly more complicated process, whereby three steps are necessary when creating the first version of an app:

- Firstly, a key must be generated, replacing 'my-release-key' with the desired key name and 'alias_name' with the desired alias name:

  $ keytool -genkey -v -keystore my-release-key.keystore -alias alias_name -keyalg RSA -keysize 2048 -validity 10000

- This command will produce a key, and it is the developers' responsibility to keep it safe for future updates, the key generation process will ask several questions about the developers identity and will ask for a password to be created in the end of the process.

- The next step involves signing the unsigned app with the newly generated key:

  $ jarsigner -verbose -sigalg SHA1withRSA -digestalg SHA1 -keystore my-release-key.keystore HelloWorld-release-unsigned.apk alias_name

- Finally, the apk that will be uploaded to the Play Store can now be generated:

  $ zipalign -v 4 HelloWorld-release-unsigned.apk HelloWorld.apk

It is worth mentioning that the first step should only be executed once, subsequent updates will then reused the key for the jarsigner before it can be zipaligned. Failure to provide the same key will block the app from being updated.

Chapter 8

References

Chapter 9

Bibliography

conrad barski land of lisp

paul graham hackers and painters

Quentin Charatan and Aaron Kans java in two semesters

Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Matthias Felleisen How to design programs

Gerald Jay Sussman, Hal Abelson structure and interpretation of computer programs

Appendices

Appendix A - About this documentation

The dilemma

At the time of this writing, I faced a dilemma, which word processor is the best processor for writing this documentation.

The most obvious choices were:

- Microsoft Word
- Apple Pages
- Google Docs

Surely any of the choices above would be suitable, however, in my humble opinion, such options can be somewhat 'evil' when it comes to document formatting, turning most documentation written in one platform unreadable in the other or to the very least almost certainly not looking as intended.

In general, converting the document to a PDF format remediates this problem, however, I am now left with the fact that my content is tightly coupled to the editor that created the documentation. Modification is difficult, for example, someone else may wish to edit the document, or perhaps if at some point in time in the near future I decide to present my documentation as an HTML document on the web, or perhaps as a deck of slides or even publish it in a book format, I may have to revise the entire text looking for any 'evil' formatting issues that was not visible in the former format.

Finally, as a writer, when writing documentation, I should concentrate on writing the documentation, and not about auto generated formatting issues that may arise and drag productivity in a typical writing session. The writer should only worry about the semantics and the content of his/her writing, formatting should be done separately, perhaps not even by the writer himself, or better, simply choose a new format from predefined options written by a talented designer.

LaTeX is a really good option to move away from the formatting problems mentioned above, and beyond, however, it does trap the writer with a little clutter to tinker with in terms of settings and so on. What I am trying to say is, once the document is finalised, it no longer consists of the content and the content only, but also carries several formatting tags. This in one hand demonstrates how powerful LaTeX can be, but in the other, may confuse and distract the writer.

The solution

Since this course is about computer science, I set off to find a solution that would allow me and

any other developer to run away from the masses and write simple interchangeable documentation with ease. Writing a solution that could potentially be further enhanced to the point a non programmer could also benefit from.

The requirements were:

- Write content and content only, without distractions
- Formatting should be written separately and be interchangeable/themeable
  - This also means that if some time in future I want to publish my content to a different type of media, I should be able to do so without too much effort

- It should not be coupled to any specific text editor
- It should have an automated building and deployment solution

The answer was always there, Markdown. Markdown is a really simple and easy to use markup language, it is the de facto standard for readme files in software development and widely used in blogs throughout the web.

A short introduction to Markdown syntax can be found in the link below:

http://daringfireball.net/projects/Markdown/syntax

Although Markdown may not be as powerful as LaTeX, the community around it is immense and really keen in providing further enhancements to it, the following link lists many projects and plugins that address some of those short-comes:

https://github.com/cben/mathdown/wiki/math-in-Markdown

The community also provides an excellent open source Markdown editor called Mou, although Mac specific, all other major operating systems have free alternatives to Mou:

http://25.io/mou/

Finally, there is an open source project which is now maintained by Jakob Truelsen and Ashish Kulkarni, called wkhtmltopdf, this headless command line tool, allows any HTML document to be converted to PDF, and since Markdown is easily convertible to HTML, we now have all the tools we need in order to create our documentation.

http://wkhtmltopdf.org/

The workflow

To make this work, we will need to automate every step of the process, so that we can only write Markdown, then compile/deploy our work with only one command and in the process, if we wish to do so, add some personalised styles to our document.

The application that holds the documentation has been scaffolded using Yeoman, more specifically, using a generator called generator-jekyllrb.

https://github.com/robwierzbowski/generator-jekyllrb

Although Yeoman is an NPM package, backed by Node.js, it combines three simple but powerful Ruby Gems:

- Jekyll: A static blog generator created by the GitHub Team. It provides an easy to use templating language called Liquid as well as giving the means to transform Markdown files into HTML, finally compiling the entire application into deployable static websites.
- Redcarpet: A Markdown to HTML converter.
- Compass: A CSS pre-processor extension of SASS.

Moreover, Jekyll makes it really easy to deploy and host your application directly on Github free of charge.

This Yeoman generator also leverages the power of Grunt and Bower into the workflow.

Most of the Grunt tasks used for this documentation have also been used for the main IOU application, however, a few notable differences are mentioned below:

- grunt-build-control: Allows deployment to Github via Git.
- grunt-jekyll: integrates Jekyll with Node.js.
- grunt-wkhtmltopdf: Compiles HTML files to PDF.

Developing locally

To use this workflow for other projects or simply try out what has been created so far, please follow the following instructions and make sure all dependencies are met and available on your path:

Dependencies:

- Git
- Ruby and Ruby Gems
    - SASS
    - Compass
    - Jekyll

- Node.js and NPM
    - Grunt
    - Bower

- wkhtmltopdf

Git clone this repository and `cd` into it:

```
$ https://github.com/jbonigomes/ioudocs && cd $_
```

Install the NPM packages:

```
$ npm install
```

Install the Bower dependencies:

```
$ bower install
```

Serve the app:

```
$ grunt serve
```

Other than `serve`, you may wish to run:

```
$ grunt serve:dist
```

The latter will first build the application then serve the optimised code, the former, may be a better option when debugging as files will not be minified and changes will be automatically refresh in the browser using 'browser sync'.

Once displayed in the browser, you may wish to explore the documentation and view the PDF generated version too, link found right below the left navigation bar.

The directory structure will look as follow:

- .jekyll
- .sass-cache
- .tmp
- app
- dist
- node_modules

And the root directory hold the following files:

- .bowerrc
- .csslintrc
- .editorconfig
- .gitattributes
- .gitignore

- .jshintrc
- _config.build.yml
- _config.yml
- bower.json
- Gemfile
- Gruntfile.js
- package.json

The top most directories, prefixed with a dot '.' only hold temporary files required by Jekyll to serve the application locally.

All files in the root directory are common configuration files for Git, Jekyll and Node.js.

The node_modules directory holds all the Grunt tasks.

The app directory is where we, developers will actually work.

The dist directory holds the last build from our application, that is, the result of our code. For every build, Grunt will delete this directory and re-create it based on our changes. The dist directory holds the optimised code that is ready for production.

The app directory tree structure looks like this:

- _bower components
- _includes
- _layouts
- _posts
- _scss
- docs
- fonts
- img
- js
- pdf

The top most directory `_bower_components` holds the Bower dependencies. All the directories prefixed with an underscore '_' will not be copied to the dist directory during the build process.

- _includes: holds the cover page and the xsl file used to generate the table of contents in the PDF.
- _layouts: hold the layout for the main web page, the layout for the documentation page and the layout for the PDF file.
- _posts: holds the actual Markdown that makes up this documentation, they have been separated by chapters.

- _scss: the SASS styles for the app and the print.scss, used by the PDF file.
- docs: holds the documentation landing page.
- fonts: the web fonts used in this project.
- img: the images.
- js: the JavaScript.
- pdf: a placeholder HTML file containing all chapters and extending the PDF layout, as well as the actual generated PDF file.

During your workflow, you may notice that the PDF file looks well formatted when running `grunt serve:dist` however, this statement does not hold true when running `grunt serve`, this is because Jekyll compiles the temporary CSS file in a different location when running on debug mode, to avoid too much overhead, when testing the PDF's layout you may use this short-hand:

```
$ grunt build
```

This will build the latest changes and save them under the dist directory, including a well formatted PDF file. You may now open the PDF directly:

```
$ open dist/pdf/index.pdf
```

If you wish to deploy this app to GitHub, you will need to update the remote option in the buildcontrol task, found in the Gruntfile.js, swap it with a valid GitHub Page repository path, more details about GitHub Pages, Jekyll and BuildControl can be found in the link below:

- https://pages.github.com/
- http://jekyllrb.com/
- https://github.com/robwierzbowski/grunt-build-control

Once set up, you may deploy your code using:

```
$ grunt deploy
```

Roadmap

- Keep improving this workflow and gathering feedback from others on how to improve it.
- Create a more generic version to use as a starting point for future documentations, perhaps a Yeoman generator.
- Fix the issue with no styles in the PDF's generated with `grunt serve`.
- Create new themes for the Markdown's, so far it is using the generic GitHub flavour styles.

Details

The source code for this documentation can be found at:

https://github.com/jbonigomes/ioudocs

The latest built source for this documentation can be found at:

https://github.com/jbonigomesbbk/jbonigomesbbk.github.io

And the final outcome here:

http://iou.rocks/

Appendix B - Users Guide

This is the users guide for IOU app:

http://iou.rocks

IOU allows keeping track of expenses with Facebook friends.

Why use IOU?

- Every time you remember to buy something, just like a shopping list, for example:

    - When the washing power is reaching its' end
    - When making a list of groceries for the next BBQ party

- Every time you go shopping, consult your IOU lists

- Every time you buy something that should be shared with your friends

IOU lets you create lists. A list is a container which holds three other lists:

- The members of this list
- The products you wish to buy
- The products bought in this list

IOU will keep you informed of how much you spent and based on how your lists are grouped.

A list can be closed at any time, simply glance at the members view before closing it to know how much you owe or how much each member owes you.

To get started, you must log in using your Facebook, IOU will never share your information with 3rd Parties, neither will it create posts on your behalf.

Before login, you must read and accept the terms and conditions:
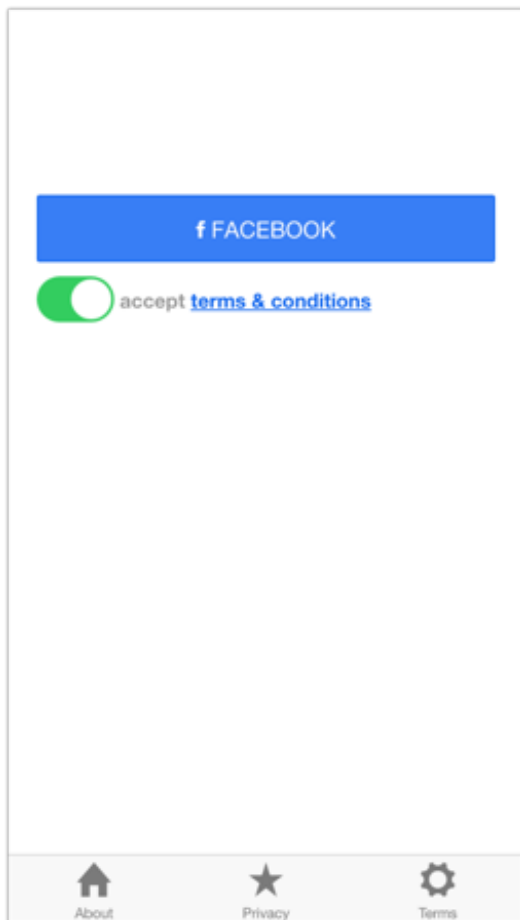
Unless one of our friends has already added you to an existing list, chances are that your first screen will look blank, the first step is to create a new list, clicking the 'NEW' button in the bottom of the page:
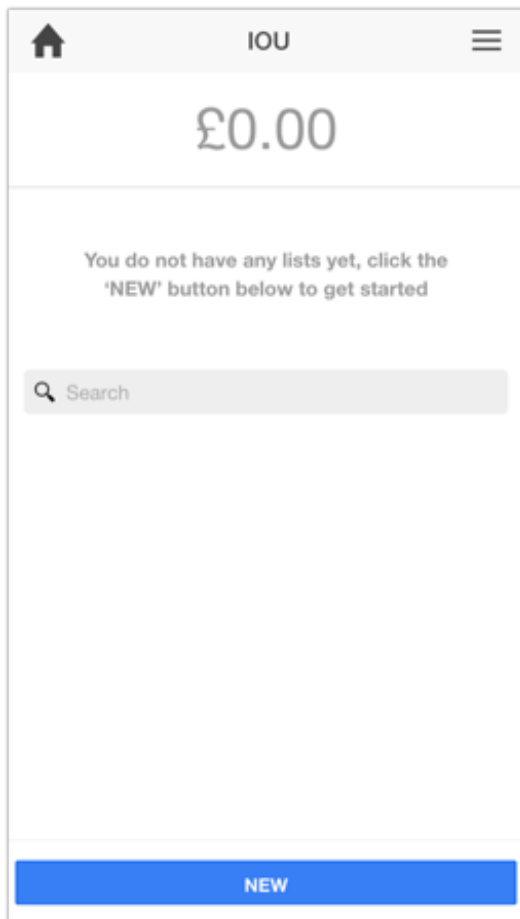
Let's name our list, Households, since it will be a list with items that we share with our housemates. We will also choose the bag icon for this list:
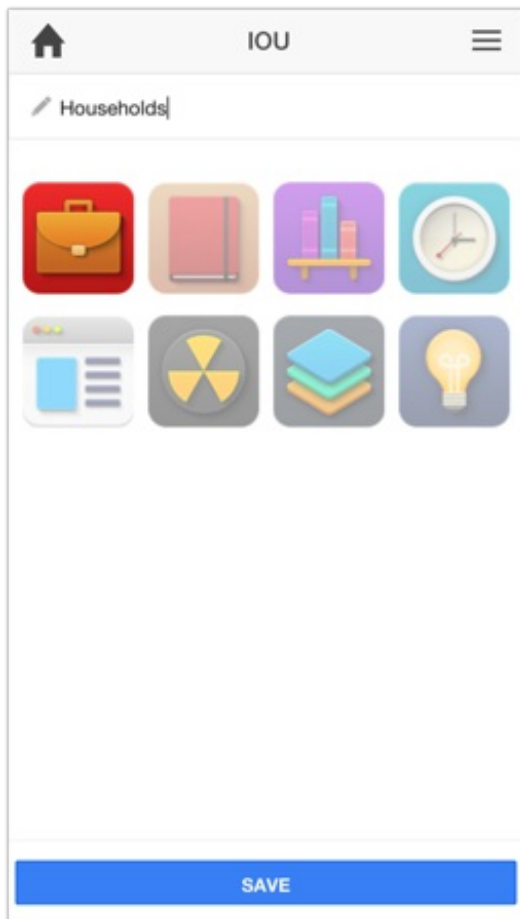
We now have our first list, please note the search box at the top, if we have many lists this feature can come handy to quickly find the list we want. Lets click on it to explore further:
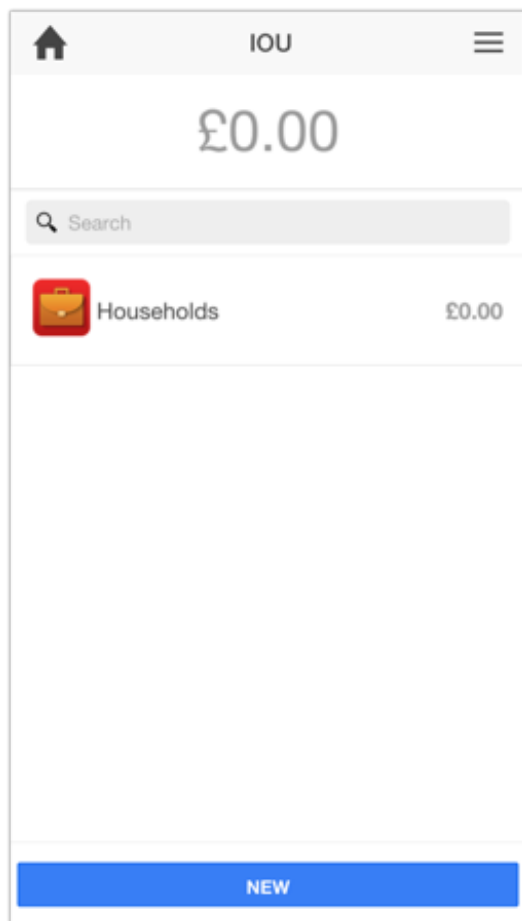
Our list is still empty, we can add a new product by pressing the 'NEW' button in the bottom of the page:

We now have to choose what we want to buy, lets go with potatoes since we are having chips tonight:
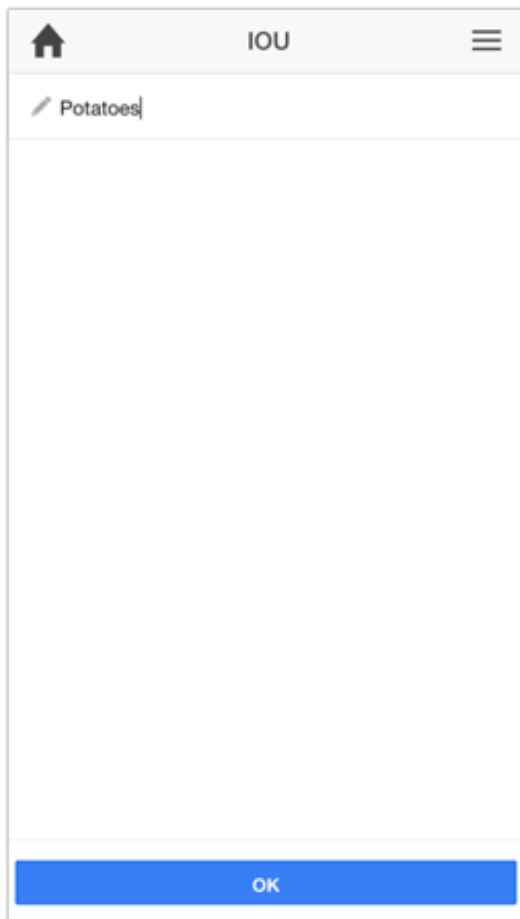
Our item now appears in the to buy list, this is much like a shopping list of things we want to buy. Lets go ahead and buy this product within IOU. All we need to do is click on the item itself:

Ok, we can now add the price and press 'BUY':
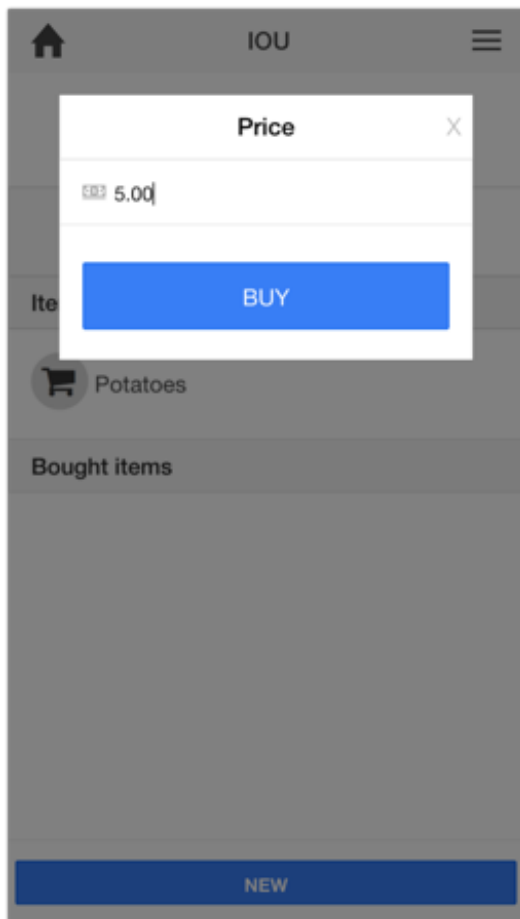
Our product will now appear in the bought items list, along with its name, price, date bought and a picture of who bought it. Please note that the big gray number at the top of the page still reads '£0.00' despite the fact we have bought an item. That is because we are the only member of our list. If we, want to be the only member of this list and only use it to track personal expenses, we can leave it as it is, however, if we want to split the expenses with others, we have to add new members. Lets click on the members tab:
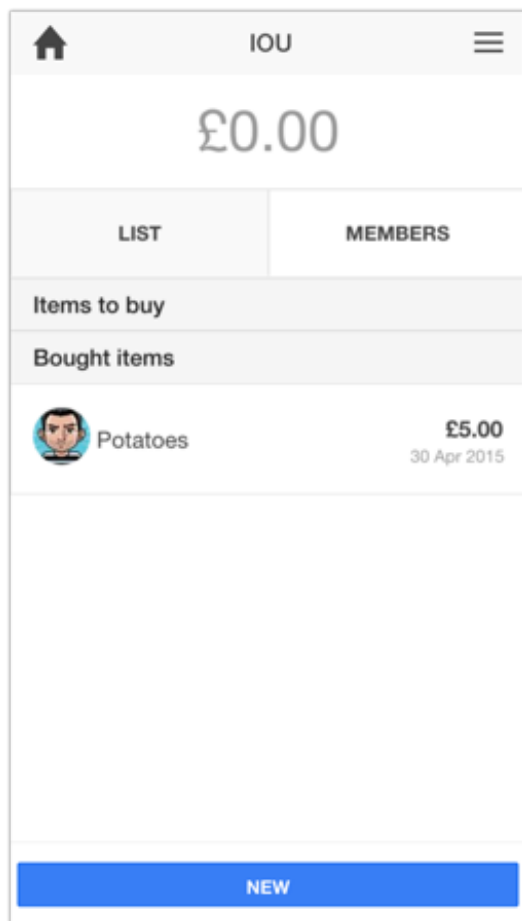
To create add a new member, simply click on the 'NEW' button at the bottom of the page. Please note that you will only be able to see a list of your Facebook friends that also happen to be members of the IOU app:
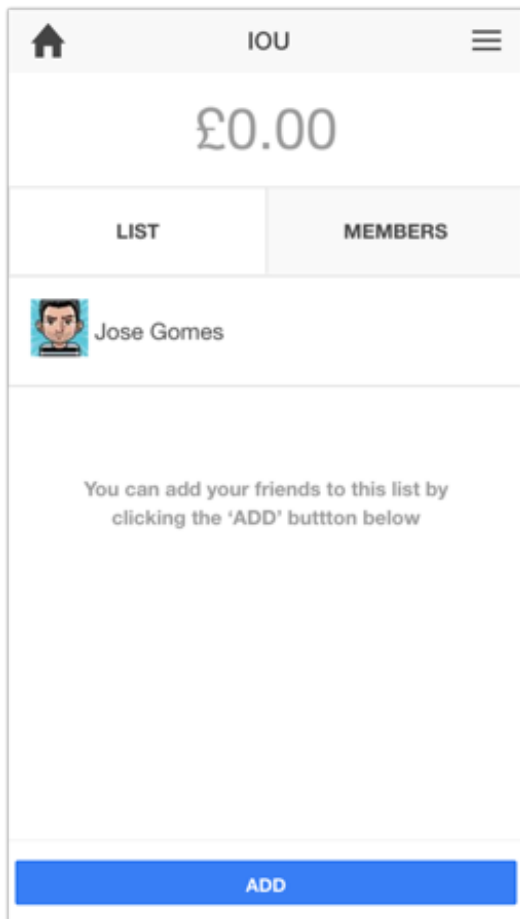
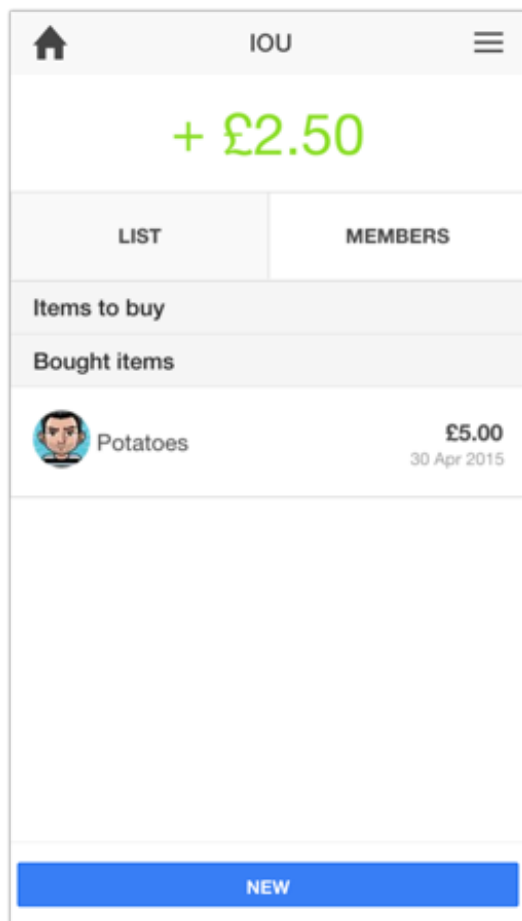Lets choose one of our friends by clicking the 'ADD' button. We can also cancel this action and filter the list, much like in the home page.

Ok, we are now £2.50 positive in our list:
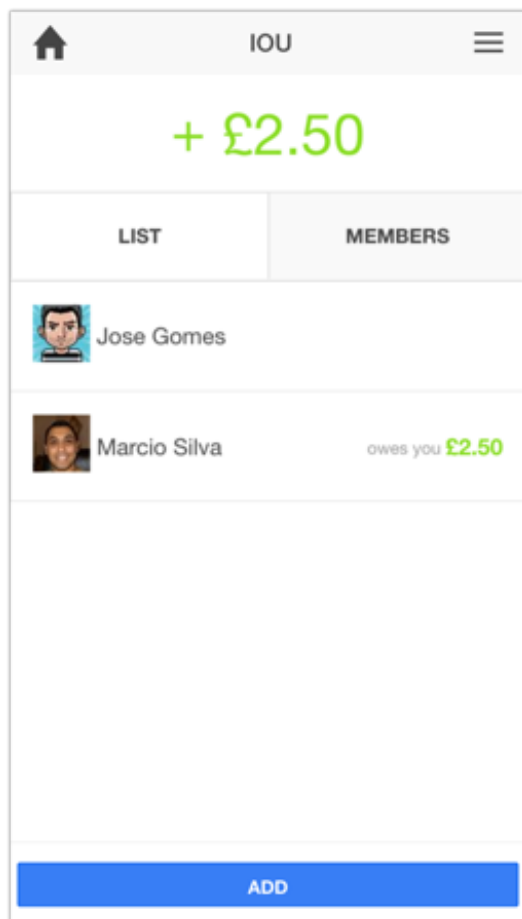
We can always go back to our products page if we want to add a new product or edit a product we bought in the past. Let's edit the product we bought by clicking it:
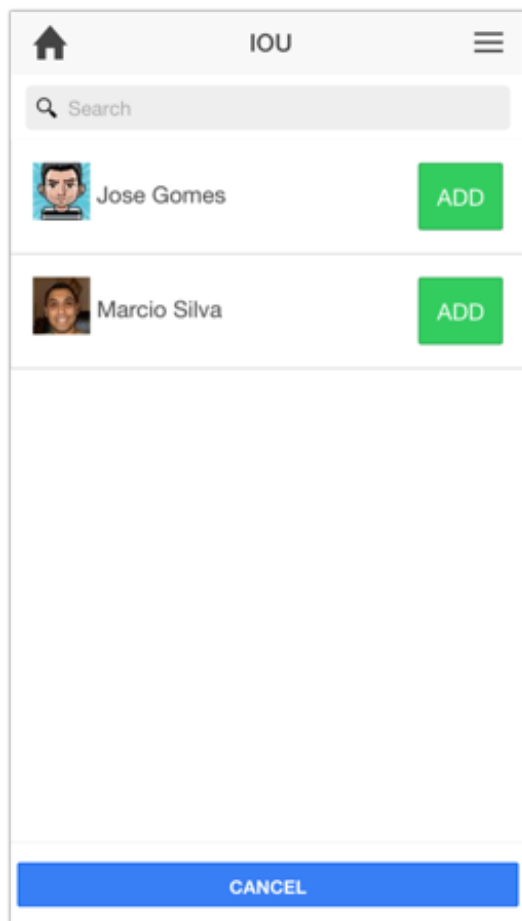
We can change its name, price and who bought it. Lets go ahead and change the person who bought it by clicking our picture:
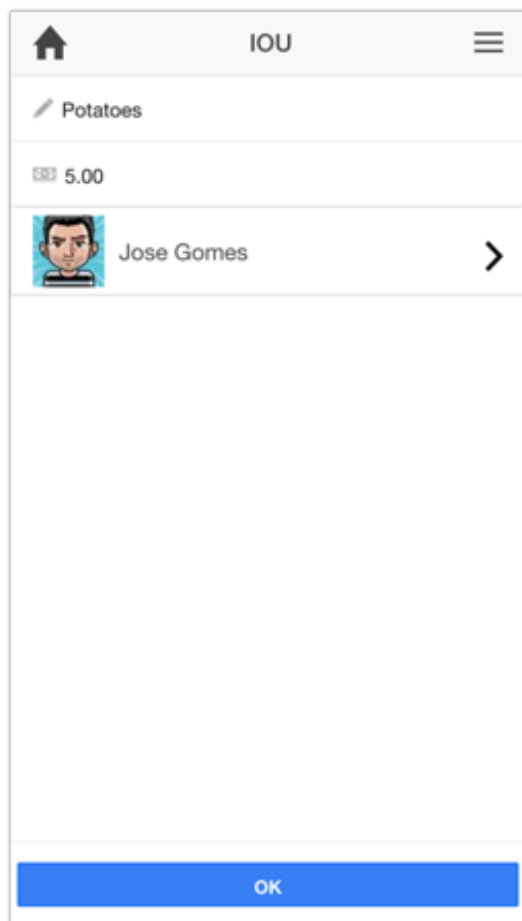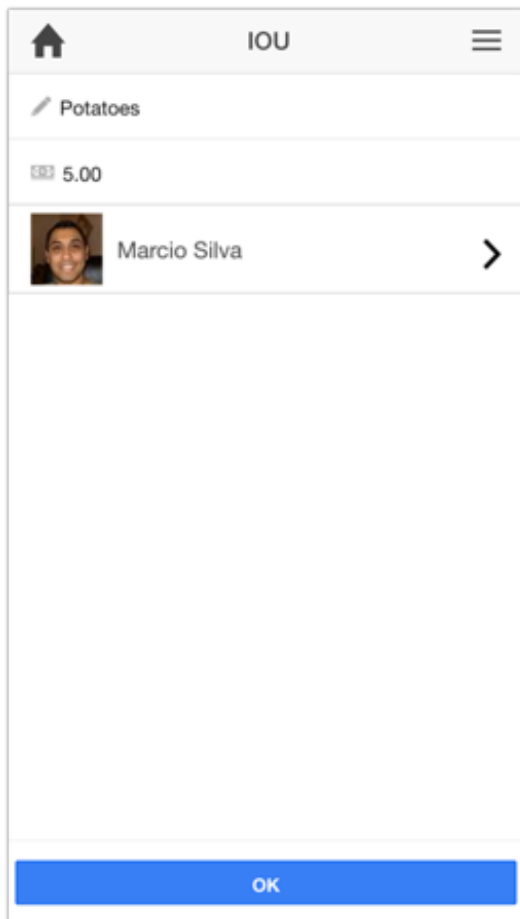
We can now only choose those who are members of our list:
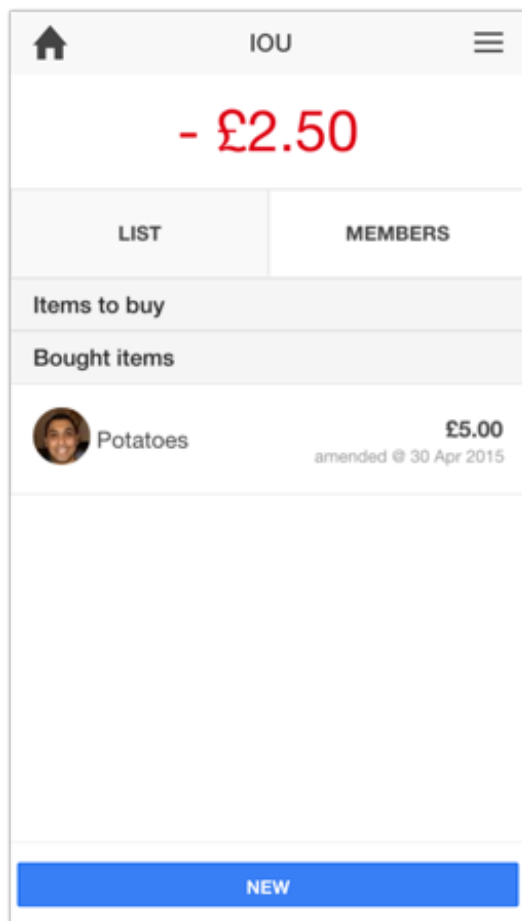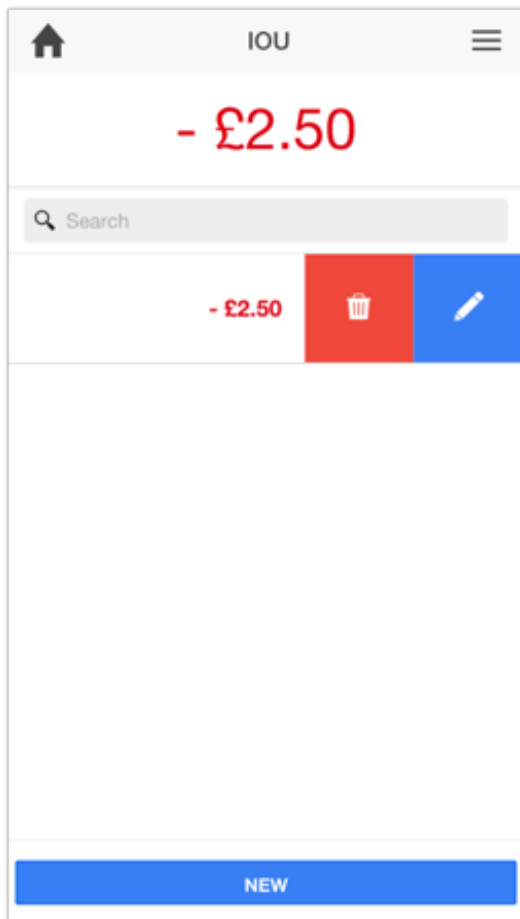
Ok, we have successfully reassigned who bought our first product, lets press ok to save the changes:
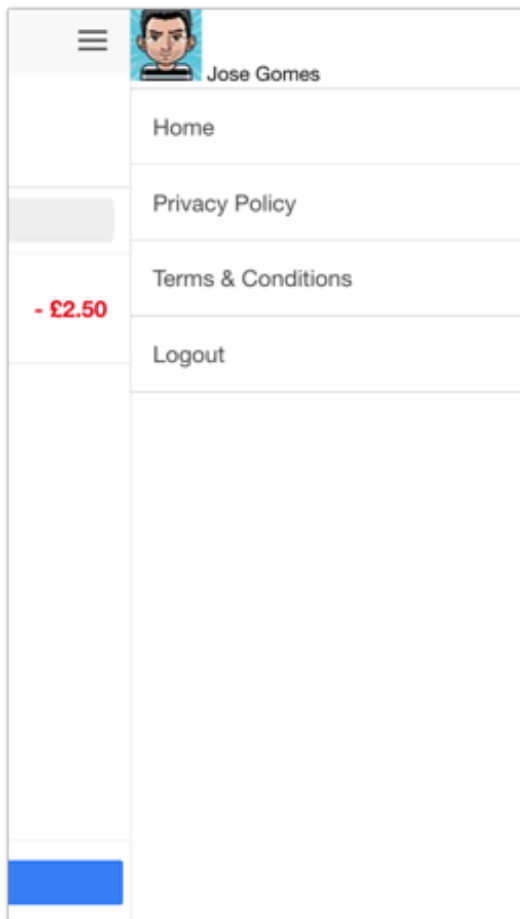
Please note that we are now £2.50 in debt. We can always navigate back to the home page by pressing the 'HOME' icon at the top left corner of the screen:

Lists can be edited and deleted, those options can be revealed by swiping the list name from right to left:

A quick access menu is also available on all pages:

We hope you enjoy IOU as much as we do.

Appendix C - The included USB

The included USB stick contains the source code for both the documentation and the application itself named implementation.

In order to create local builds and even make amendments, each section imposes a set of dependencies.

Appendix A explains in details how to compile and run the documentation files locally.

For the main IOU app, the following are the dependencies required for development:

- Git
  - Git-flow

- Ruby and Ruby Gems
  - SASS

- Compass

- Node.js and NPM
  - Grunt
  - Bower
  - Ionic
  - ios-sim (mac only)

- A UNIX based shell
- SDKs
  - Android SDK
  - iOS SDK (mac only, comes with XCode)

Below is a quick guide on how to build the IOU implementation:

1 - Either copy the content of the implementation directory or clone the repository directly from GitHub then 'cd' into it:

```
$ git clone https://github.com/jbonigomes/iou && cd $_
```

2 - Install the npm dependencies:

```
$ npm install
```

3 - Install the bower dependencies:

```
$ bower install
```

4 - To view the app run:

```
$ grunt serve
```

5 - To build a new distribution run:

```
$ grunt build
```

6 - To serve a built app run:

```
$ grunt serve:dist
```

7 - To add a new platform run either/or:

```
$ grunt platform:add:ios
$ grunt platform:add:android
```

8 - To emulate a platform run:

```
$ grunt emulate:ios
```

9 - To build a new release:

```
$ grunt build:ios --release
```

Testing is still being implemented, however, the following command is already available for running unit tests:

```
$ grunt test
```

Should you wish to try the protractor tests, first install Protractor globally:

```
$ npm install -g protractor
```

Update the webdriver:

```
$ webdriver-manager update
```

Spin up the server:

```
$ webdriver-manager start
```

You can now run end-to-end tests like so:

```
$ protractor test/e2e/conf.js
```