Electronics and Computer Science
Faculty of Physical Sciences and Engineering
University of Southampton

Shakib-Bin Hamid
April 20, 2016

Collaborative Consumption for UoS

Project supervisor: Dr. Michael R. Poppleton
Second examiner: Dr Nicholas Gibbins

A project report submitted for the award of
MEng Software Engineering

Electronics and Computer Science
Faculty of Physical Sciences and Engineering
University of Southampton

ABSTRACT

A project report submitted for the award of MEng Software Engineering
by **Shakib-Bin Hamid**

The smartphone application industry is diverse and competitive. Emerging technologies and ever-changing user expectations turn the rigid classic engineering principles on its head. On the other hand, modern software engineering practices such as scrum, regular testing, participatory design and user experience evaluation hope to overcome the various challenges that developers usually face in the paradigm. This project demonstrates the application of such processes by developing a hybrid mobile application for University of Southampton students to benefit from Collaborative Consumption.

Declaration

I am aware of the requirements of good academic practice and the potential penalties for any breaches. I confirm that this project report and accompanying code is all my own work except the open-source frameworks and libraries in Appendix B, which are referenced and credited in the report and code.

Acknowledgements

I want to thank my parents, Mr Abdul Hamid and Mrs Dilshad Begum, for always supporting me, my project supervisor, Dr Michael Poppleton, for his directions in the project and finally, Miss Rosanna Lathwell for designing my app logo.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Technology has introduced new trading concepts that can expand a traditional sharing community far beyond that of close friends. Take eBay for example - an eBay user can post the items that they want to sell or put on auction, with name, description and pictures to help a buyer minimise the risks in transactions. A buyer can also find ratings-reviews about the seller's previous transactions etc. Both buyer and seller can securely pay via Paypal with buyer protection. These technology platforms themselves are not providing any items; they are not product brands. Rather, they provide some features that build trust in their communities, because sharing, swapping, renting etc. only happens when there is trust between the participating parties. Such Sharing Economy using technology is called *Collaborative Consumption* (CC).

University of Southampton (UoS) itself has a vibrant student community. Students here borrow calculators and books for the odd coursework, parking spots for their visiting parents, sell or give away their old mobiles, music players or laptops etc. This happens often enough that there are several social media groups for sharing items. A targeted Collaborative Consumption tool for UoS students can certainly thrive because of the existing communities. This tool needs to be mobile because an average student usually accesses the aforementioned social media sites from their phones, on the go.

However, the goal of this project is not promoting CC, rather it is to explore modern software engineering practices -

scrum : develop in short, targeted and self contained cycles,

participatory design : include end users in requirements elicitation and user experience testing,

user centred development : develop towards user stories rather than use cases,

behaviour driven development : write automated test scripts for each user story under preconditions.

Another goal is to experience both *server* and *client* side development to demonstrate how mobile application features such as push notifications etc. operate. The premise of a mobile CC tool is suitable to demonstrate such processes.

Mobile applications for smartphones usually target a wide range of users from very different backgrounds. So, evaluating User Experience (UX) for these applications is essential and time should be spent on user participations. However, development in the native language for a platform, Java on Android for example, usually requires a very steep learning curve. Together with the need to develop a full stack application in such a short time, a fast mobile development framework is required. Such tools exist that it is possible to build mobile applications entirely in JavaScript (both server and client) and deploy the same code to multiple platforms. As this project's resultant application is a proof-of-concept rather than a marketplace application, it is quite acceptable and even preferred to work with such *Hybrid* solutions.

Overall, the goal of this project is to demonstrate modern Software Engineering practices by implementing a Client Hybrid Mobile Collaborative Application and the corresponding Server Application.

Below is a summary of what content is to follow -

Chapter 2 reviews the relevant literature on collaborative consumption, hybrid mobile development and agile software development practices.

Chapter 3 lists the features of the client app and discusses what features were discarded and why.

Chapter 4 explains how the project was organised/structured and managed during different stages such as sprints etc.

Chapter 5 gives an overview of the technologies used and discusses the architecture of the system as a whole.

Chapter 6 explains the client and server implementation in-depth.

Chapter 7 discusses how the implementation was tested - automated and against scenarios.

Chapter 8 gives a detailed description of why and how the user sessions were organised, performed and documented.

Chapter 9 evaluates the success, shortcomings and usefulness of the project.

Chapter 10 concludes the project aims and results.

Chapter 2

Literature Review

2.1 Collaborative Consumption

2.1.1 Defining CC

"Collaborative Consumption (CC)" is people, in a community, acquiring or distributing physical goods or services for *some form of compensation* amongst themselves [6]. The form of compensation dictates whether the transaction can be labelled as *Sharing* ownership, *Swapping or Trading* for another resource or even *Buying and Selling* of used goods. One commonality in all such practices is the interaction amongst the community members using technology, especially Web 2.0 (otherwise known as the *interactive Web*). Sharing items within a small organisation like family or friends has been present in society for a very long time, but the technologies like social networks, e-commerce etc. available today allow people to make trusted transactions in large communities. In fact, the CC platforms are merely *"economical-technological coordination providers"* [26]. In other words, people have been willing to engage in 'Sharing Economy' if a suitable technological platform exists.

2.1.2 CC Categories

There are three categories of Collaborative Consumption [2] :

- Where consumers pay for using a resource. For example: Singapore based Rent-A-Toy rents expensive toys for a fee.

- Where members redistribute their rarely used or unwanted items. For example: Hey, Neighbor! is a site where those in particular locales can share/rent their equipment and 'micro-favours'.

- Where people share less tangible resources like skills, space, time etc. For example: Airbnb and Couchsurfing members temporarily share their living space with fellow members.

On the other hand, two categories of Collaborative Consumption exchanges are [26] -

- ***Access over ownership***: When a user offers and shares their resources for a limited time to another user in the community (with or without the need for an exchange). For example: Rent the Runway rents expensive designer cloths to people for a fee or Your Parking Space lets the users to list their parking space to be available to rent to others.

- ***Transfer ownership***: When a user completely relinquishes their claim on an item and gives it away to another (also with or without the need for an exchange). For example: Swapstyle lets its users swap their cloths.

2.1.3 Reasons to Participate

Four possible reasons why people take part in Collaborative Consumption [26] are -

- ***Sustainability***: People want to reuse and recycle their owned goods to bring wastage to a minimum and sustain a better environment.

- ***Enjoyment***: Users often enjoy sharing their produced item because it can help others. For example: Many software developers give away their software source code on Github because they enjoy software freedom.

- ***Reputation***: Users enjoy the earned reputation in the community from participating in many transactions. For

example: Well traded Ebay users usually carry out more successful transactions.

- ***Economic Benefits***: Members want to save some cost by swapping their items, buying used ones or earn some money by selling unwanted goods.

The followings affect the satisfaction of a sharing option [37]-

- ***Cost Savings***: A sharing option that saves some money for the consumer leads to a successful transaction more often.
- ***Familiarity***: Consumers trade more often on Collaborative Consumption platforms that they are familiar with (through exposure from various media).
- ***Service Quality***: Members use Collaborative Consumption platforms that give them easy options, suggestions and an overall better experience.
- ***Trust***: People only want to carry out transactions if they feel that the community is secure, welcoming and the members are trustworthy.

2.1.4 Collaborative Consumption in UoS

University of Southampton students also have a Facebook Group called Free & For Sale, where students put up their unwanted item to sell, swap or give away. These items can include calculators, books, cloths, fashion accessories, furniture, kitchen tools, garage tools, event tickets, transport tickets, mobiles, laptops, electronic boards etc. Students arrange the transactions within themselves. The group has been quite successful in recent years.

The above leads me to believe that University of Southampton students can truly benefit from a Collaborative Consumption platform which lets *only* them trade on the platform and integrates with their campus identity. There are existing platforms including **Ebay**, **Gumtree** and **Facebook** where students can co-ordinate their transactions. But none of these are suited for

this particular atmosphere and cannot provide support for a UoS community.

2.2 Hybrid Mobile Development

2.2.1 Mobile Applications

Native apps are those built using platform specific tools, for example: Swift or C# and XCode for iOS, Java (Dalvik runtime) and Android Studio for Android etc. Examples are - Gmail by Google, Pages by Apple etc.

Web apps are built using HTML, CSS and JavaScript and hosted on a *website* that can be accessed from any browser including a mobile. Examples are - Netflix website or Google Docs etc.

Hybrid apps sit between native and web apps (Figure 2.1 [42]). They are built using similar technologies as web apps, but are then compiled into platform specific installers and used as a native app. Underneath the surface, these apps usually run a *localhost* server on the mobile that runs the app and the OS accesses them using web-view. [34, 33]

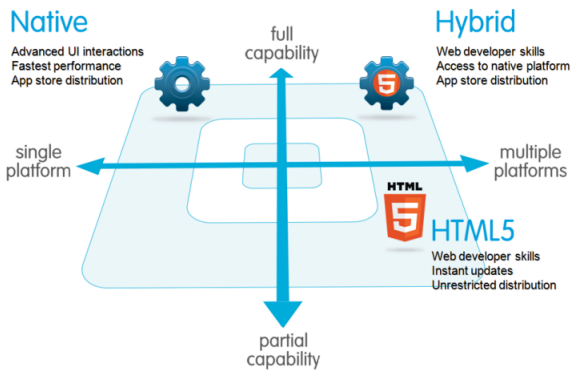


Figure 2.1: Hybrid Apps

2.2.2 Advantages of Hybrid Apps

A major difficulty in native development is that, it easily becomes difficult to simultaneously develop, maintain and support for different platforms [34]. Very little of one device's code

can be ported over to another because of lack of standards, e.g. the standard Android uses commas to separate items in a list, but Samsung phones use a semicolon [31]. Whereas, a hybrid app can be ported into multiple platforms simultaneously, with very little (if any) change.

Native apps usually perform better than their web and hybrid counterparts because they can leverage the full potential of the OS. While web apps have a disadvantage of being accessible only over network and are slow to respond, recent hybrid app development frameworks have started hardware acceleration. In fact, apart from 3D games, there is negligible or unnoticeable performance gap between the two paradigms [12].

Early web and hybrid apps were notorious for their lack of usability on a mobile, because the components such as buttons were not optimised for touch. Modern frameworks, however, provide the exact components as the native apps. Resultantly "end users value hybrid and native apps similarly" [35].

2.2.3 Challenges in Hybrid Apps

Lack of automated testing frameworks is a major drawback for production ready hybrid apps. The mobile app industry, in general, does not have a unified testing pattern for either usability or functional testing [31]. Each OS provider has its own IDE (Integrated Development Environment) and test environment. But most hybrid frameworks, being very recent and open-source, do not have such features.

Moreover, debugging is difficult in mobile apps, in general, because CPU, memory etc. monitoring is cumbersome, even impossible at times. This is specially true in hybrid frameworks, as most only support primitive emulation rather than a debugger.

Finally, security is an emerging issue. Being built on web technologies, it is subject to threats from that paradigm as well as mobile OS threats [9, 25].

Nevertheless, I believe hybrid technologies are perfect for a short project with intentions of fast prototyping and user evaluation.

2.3 Agile Development & Scrum

2.3.1 Overview of Agile Method

The "Agile Manifesto" [5] stated four core principles -

1. **Individuals and interactions** over processes and tools
2. **Working software** over comprehensive documentation
3. **Customer collaboration** over contract negotiation
4. **Responding to change** over following a plan

Put simple, agile software development is to - define a vision and scope for the project, define requirements together with customer, build in iterations and review the results with the customer to update the requirements until release [1]. This iterative framework is called **Scrum**.

2.3.2 Scrum

In Scrum framework, the development team works closely with the **Product Owner** (usually the customer). The team defines requirements in forms of **User Stories** (`As a <user>, I want <feature> so that <reason>` [41]); how the task is carried out is left open. The project is broken down into iterations (typically 1-4 weeks), each with a goal such that the vision of the project is fulfilled by the release period [43].

The long list of tasks is called the **Product Backlog**. At each iteration, tasks are selected (**Sprint Backlog**) to generate a *working* software of *value* to the customer. This is then demonstrated to the product owner and shippability is discussed. Backlog is updated and prioritised with possibly new or reduced items (Figure 2.2 [43]), changing the direction and contents of future deliveries. The development team also has

a **Review** meeting to improve internal development process.

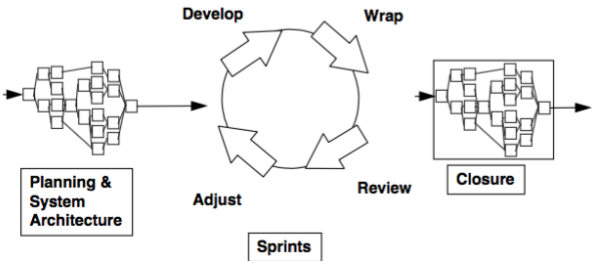


Figure 2.2: Scrum

However, agile methods are not perfect by any means. For example: it is not always possible to make upfront cost estimation because of the volatile nature of development [40]. As always, an early architectural design decision may adversely affect future development. Despite these, I think it is preferable to take an agile approach to any modern mobile app development [46] because of the nature of the industry itself.

2.4 Testing in Modern SE

2.4.1 Test Driven Development

Test Driven Development (TDD) is a software engineering practice where automated unit tests are written before development. These tests fail at first, but start to pass as the development proceeds. Thus development itself remains focused and targeted [47]. It is also a part of Extreme Programming (XP). Modern agile teams believe that the advantages include [20] -

- Efficiency in error detection in later stages because of comprehensive development stage testing.
- Fast regression testing possible because of the test driven code.
- Testability (code that is easy to test) increases in the entire codebase.
- Loosely coupled system as a result of the need for mock objects.

However, some shortcomings of the TDD process are [20] -

- Some code is inherently difficult to unit test, for example: GUIs etc.
- The code must coherently support mock objects to isolate other code which proves hard in practice.
- Testing hard-to-test code needs a certain level of experience and determination.

In the literature, I have found TDD to be the current trend across the industry, including myself using the process in IBM during internship. However, TDD requires an experienced team in the required technology to implement the architecture and have a coherent data model. In short projects it can actually be counter productive.

2.4.2 Behaviour Driven Development

In TDD most developers are unsure what and how much to test, where to start etc. There is also usually a gulf between developer testing and customer acceptance. Behaviour Driven Development (BDD) attempts to answer these questions by mandating automated test cases based *only* on the user stories [44]. Any other automated test is optional for the development teams. Below is a test structure for BDD, extracted from a user story -

```
Given <init_context>
When <event> occurs
Then <assert_outcome>
```

This style can be placed in between unit and integration tests in traditional testing. One major advantage is that it is easier to understand as a customer. BDD may or may not replace general unit testing itself. Also, regression testing mandatory as usual.

Chapter 3

Application Specifications

In this chapter, I have listed the features of the Client app, as well as which features were discarded and justifications for doing so.

3.1 Implemented Features

Using this app, students can borrow (with or without a deposit) and sell items amongst themselves (Figure 3.2h). They can search for their desired item from a list (Figure 3.1b) via text based search (Figure 3.1a) or from a list of categories (Figure 3.2b) as well as add their own items with textual details and pictures (Figure 3.2e, 3.2d). Users can navigate the app using a slide-in menu to other pages (Figure 3.1c), e.g. their profile where they find their avatars, simple ratings, items and comments by others (Figure 3.2c). They also receive push notifications about items they want to borrow/sell and find a list of such notifications in app (Figure 3.2a, 3.2i).

3.2 Unimplemented Features

This project is not about creating a production ready app, but rather to investigate and experience software engineering(SE) practices. If some feature did not add to a high-priority user story or did not add to the SE values, then it was ignored. Some of the important ones are below -

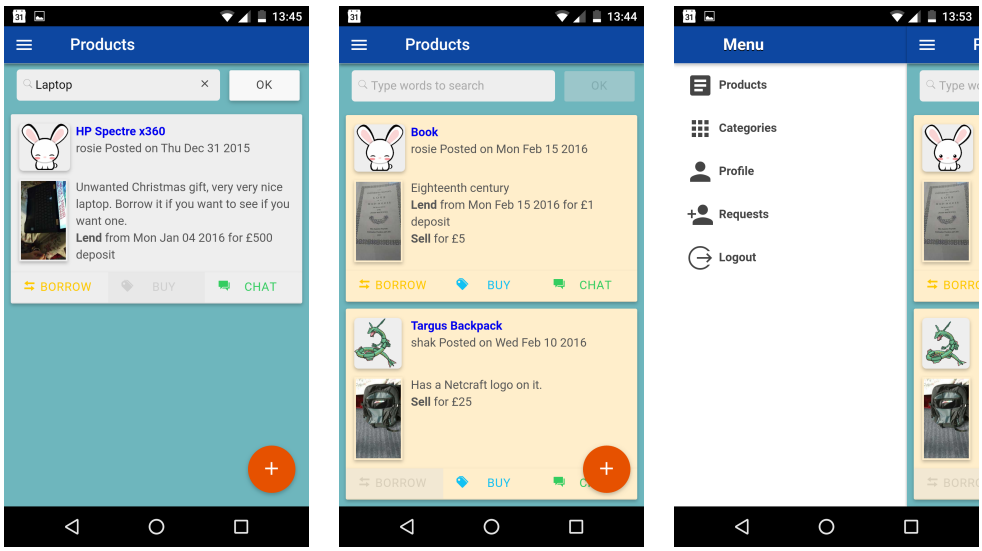
- Users cannot create their own accounts, i.e. there is no registration. One reason to do so is that, I imagined such a system to use the university authentication to guarantee

that only UoS students use the service. Although, there is a REST route available on the server side to do it, so that, on the first login in a production app, a User object will be created on the database.

- There is no chatting option on the app, even though the placement in the UI is there. The major reasons are - time constraints and less priority. Given the choice between push notification and chat, the user interviews led me to (rightly) prioritise push. I discuss more about it in later chapters.

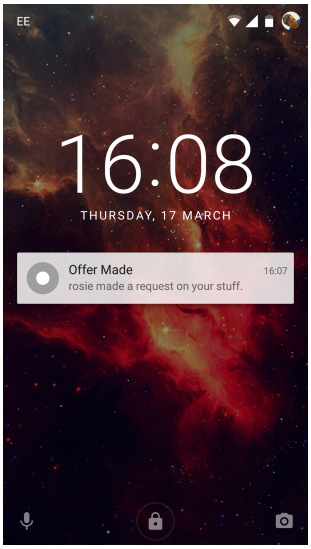
- Users cannot upload a profile picture, a default one is set for them. For demo purposes, I edited user objects on the database myself in the figures here. I have demonstrated how such features can be implemented by allowing to upload item picture, so other user stories took priority.

3.3 Screenshots

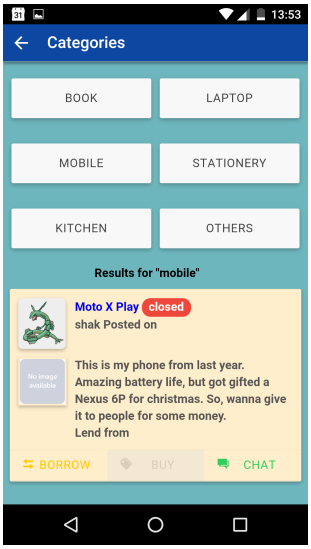


(a) Search (b) Product List (c) Menu

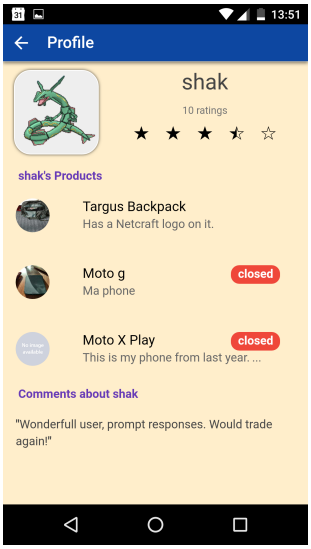
Figure 3.1: Screenshots



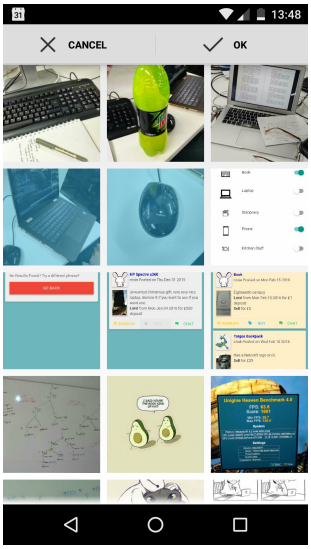
(a) Notification



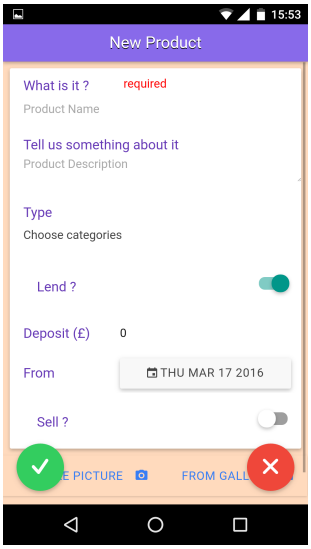
(b) Category List



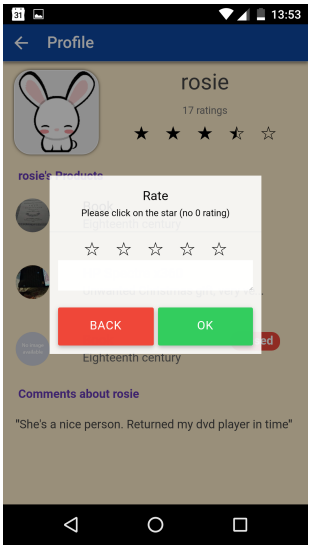
(c) Profile



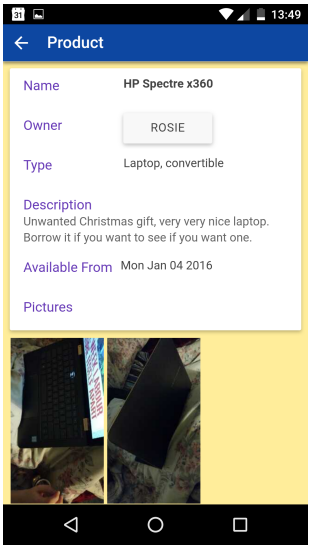
(d) Gallery Select



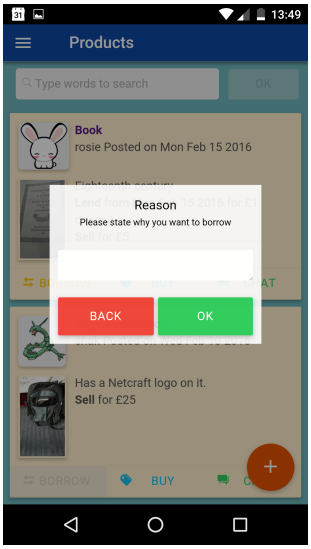
(e) Add Product



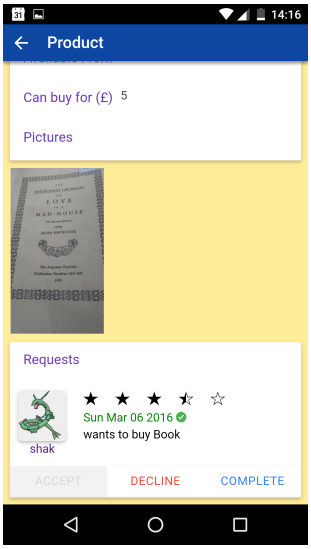
(f) Rate



(g) Product Detail



(h) Borrow



(i) Requests

Figure 3.2: Screenshots

Chapter 4

Project Organisation

In this chapter, I have discussed the overall project plan and the organisation in various stages in detail.

4.1 Plan Overview

The overall plan for the project was to - start with some simple assumptions and make a simple prototype, gather user stories through interviews, develop in four sprints, perform UX evaluation and finish development (Appendix D). I used Microsoft Visual Studio online to keep track of the user stories (backlog), a physical kanban board for bugs, Gantt chart for the general plan, Github for version control and Mendeley for formal background research. The project was to be completed between October and March.

4.2 Initial Sprint

I started the first sprint in early October with a goal of producing a very simple client app, working together with a simple server and database. The initial user stories were based on the following assumptions and put into the first iteration of product backlog, seen in Figure 4.1 -

- There will be a number of users and things.
- Most routes will require user authentication.
- Documentation should be adequate for future development.

This prototype was then used in a set of user interviews to elicit more requirements and form user stories.

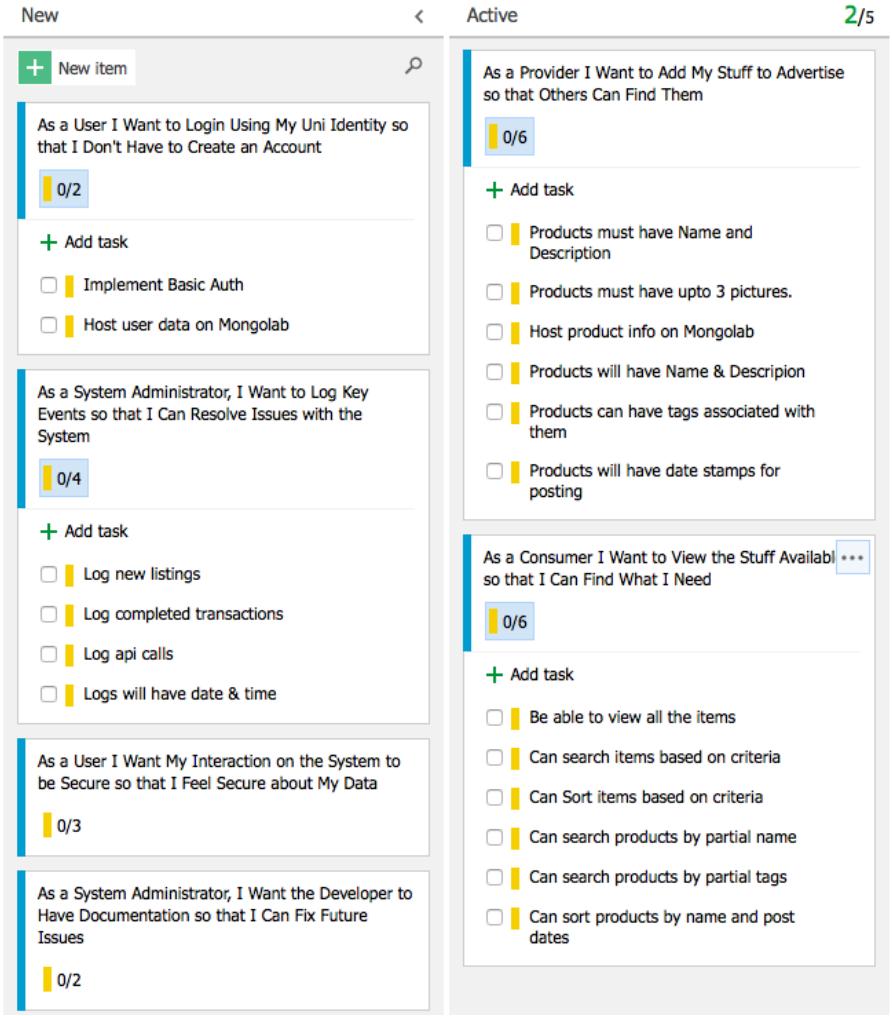


Figure 4.1: Backlog v1

4.3 RE with User Participation

As planned, I set up interviews with 5 potential users, all UoS students. *Participatory Design* is to invite end users in the designing process of software development. Chapter 8 describes the process and my experiences in detail. In short, my target was to ask the interviewees some open ended questions that will help me better understand their *pain points* and have them draw some wireframes on paper. I used the information gathered to include new user stories and select work for the second sprint.

4.4 Intermediate Sprints

The next stage was to develop the user stories from the updated product backlog (Figure 4.2) in three sprints. After each sprint, I spent 3-4 days in review to test the *user stories* (*not* development testing) to decide if they can be considered closed (*acceptance criteria*, usually performed by the product owner), updated with more work, be simplified etc. I also recorded the bugs, separate on a physical kanban board.

At that point, I also re-prioritised the backlog, so that I could complete the user stories that can have the most effect with the least effort (*impact vs feasibility*). I used both the user interviews and my abilities in required technologies as metrics for this procedure.

4.4.1 Sprint 2

This sprint's targets were as follows -

- A basic user profile functionality.
- Adding products, view list of added products and basic search.
- Simple logging for each HTTP request.

The review was shorter than planned, as the prototype was still quite simple. The updated product backlog can be seen in Figure 4.3. It introduced a new column - *Disregarded* where I placed the user stories that would not be included in the final version. A progress report was submitted after this sprint.

4.4.2 Sprint 3

The next sprint, performed during the Christmas period, was set to complete user stories regarding -

- Adding some trading options - lend, sell etc.
- Allow offers/requests to be made regarding products.
- Allow users to accept/decline offers.

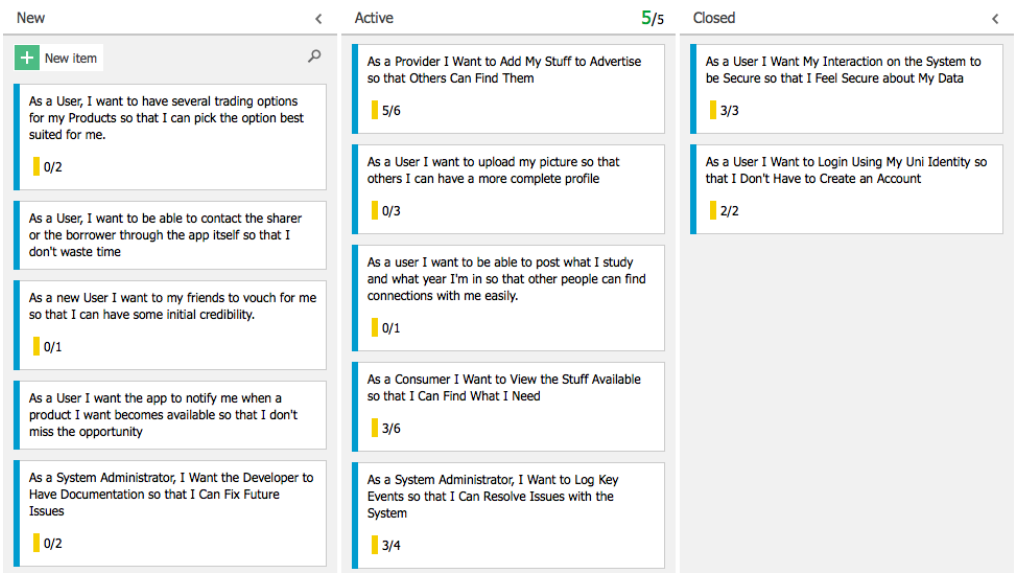


Figure 4.2: Backlog v2

Another major goal in this iteration was to start formal JavaScript testing, starting from server routes. I discuss about testing in more detail in Chapter 7. This sprint review revealed two choices for the next sprint -

Chat functionality between two users.

Push Notifications sent regarding offers made on products.

Whereas, every user wanted push notifications for not only offers, but also for several other user stories, chatting only serves a single user story - *in-app communication*. Each required a significant learning curve, and project plan and timings could only accommodate one to be done *well*. I chose notifications, because it could be used in various purposes and allow me to gain better insights into mobile development (*push notifications* being more prevalent on mobile platforms). The updated product backlog can be seen in Figure 4.4.

4.4.3 Sprint 4

As planned, this iteration, starting in the Semester 2, was to finish the user stories already present on the backlog. The primary goal in this iteration was to implement push notifications regarding the offers made on products. Some of the major tasks were -

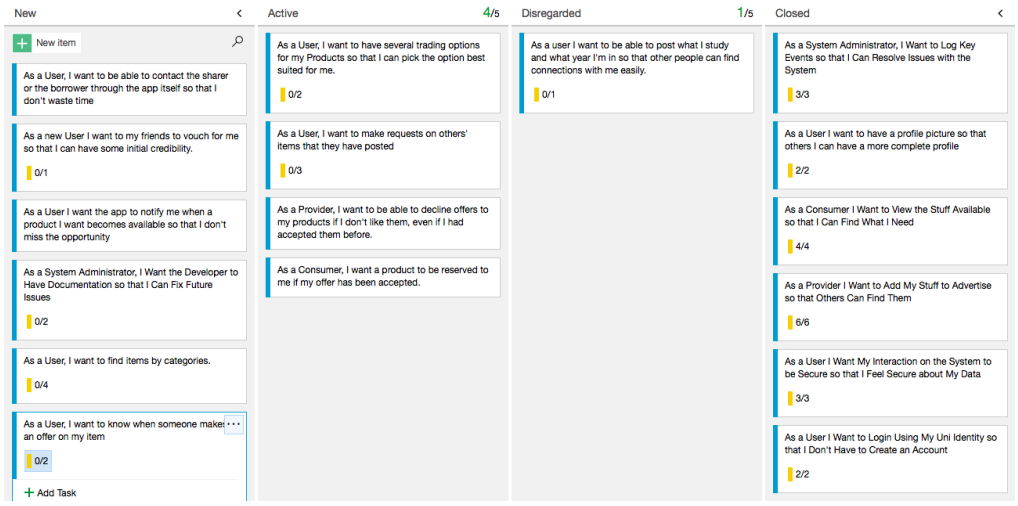


Figure 4.3: Backlog v3

- Integrate GCM with the server.
- Register client device for push notifications.
- Send push notifications to the correct users, in time.

During the development and review, I realised that push notifications were both unreliable and quite difficult to test. However, I marked the user stories as done by simplifying the acceptance criteria. The next stage was to perform User Experience (UX) evaluation on the current prototype.

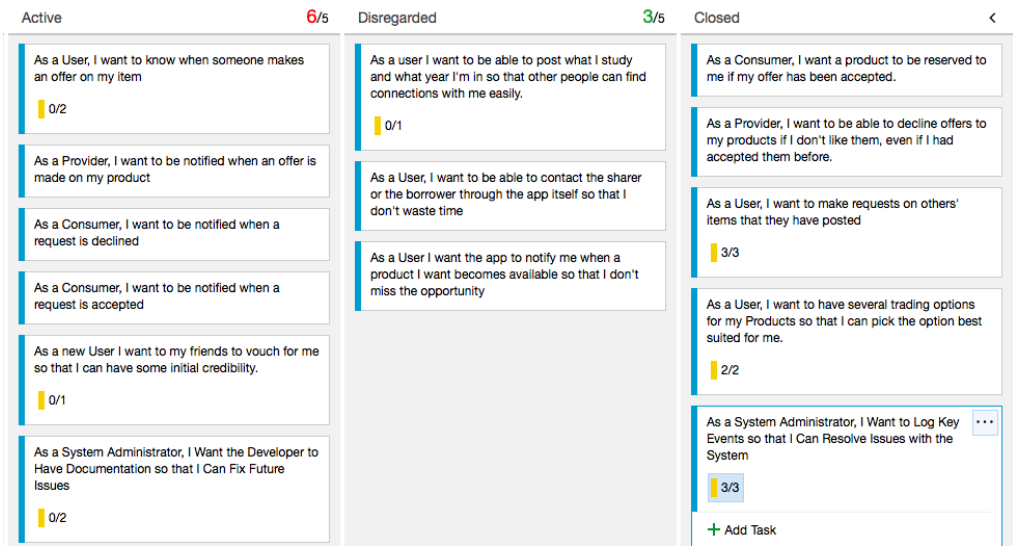


Figure 4.4: Backlog v4

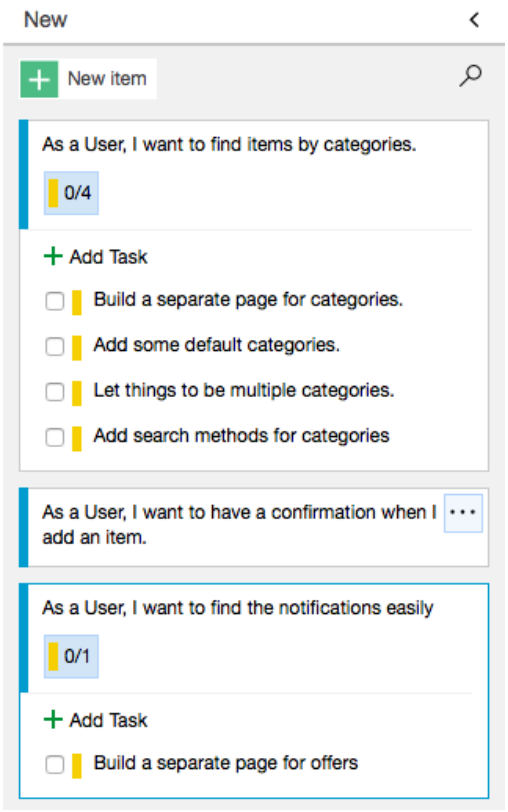


Figure 4.5: Final User Stories

4.5 UX Evaluation

Interviews were set up with the same users who were interviewed earlier to evaluate the prototype. I have discussed the process in detail in chapter 8. The results were largely satisfactory. However, it also gave rise to a number of issues, found in Appendix E. I evaluated the issues and prioritised them based on effort needed to address them. The final backlog was updated as seen in Figure 4.5.

4.6 Final Sprint

The final sprint's goal was to resolve most of the simple issues found in the UX evaluation and focus on at least half of the major issues. The results included -

- New app layout to utilise white space better.
- Separate sections for categories and offer notifications.

In the final sprint review, I found most of my originally planned specifications to be completed to a satisfactory level (for an

alpha stage prototype, not a production ready app). So, I discarded the final UX evaluation interviews, which were only planned if significant issues were found in the previous evaluation.

Chapter 5

Architecture & Technology Overview

In this chapter, I will briefly discuss the frameworks used in the project, followed by how well they served their intended purpose as well as how they work in tandem as a complete system.

5.1 Technologies Used

JavaScript is the base programming language in the project. However, each part of the system is built using a suitable framework. In this section, I will only list the frameworks.

5.1.1 Client Application

Ionic [13] is used as the 'app' framework. It provides the GUI components as HTML and CSS and plug-ins for platform specific support for mobile features like camera, calendar etc. from **Cordova** [14]. As for the JavaScript part of the application, it uses an opinionated framework, **AngularJS** [21], to provide interactivity and navigation.

5.1.2 Server Application

I used **NodeJS** [19], a non-blocking JavaScript runtime, and **Express** [18], an unopinionated framework, to build the server. **MongoDB** [38], a NoSQL database, was used for storage.

5.1.3 Test Suits

MochaJS [36], a JavaScript test framework, was used as the automated test runner and the assertion library was **ShouldJS** [29].

5.1.4 Cloud Platform

The server was hosted on **Heroku** [28], a PaaS (Platform-as-a-Service), while **mLab** [15] hosted the database.

5.2 Applicability of Technology

In this section, I will discuss how the aforementioned frameworks helped in development.

5.2.1 Ionic & AngularJS

Ionic provides HTML 5 and CSS built components like in Figure 5.1. These are easily customisable and provides great usability on mobiles. Using the CLI (Command Line Interface), I easily added plug-ins (`ionic add <plugin>`), different platform support (`ionic add <platform>`) and generated platform specific installable app (`ionic <build || run> <platform>`) without writing any native code.

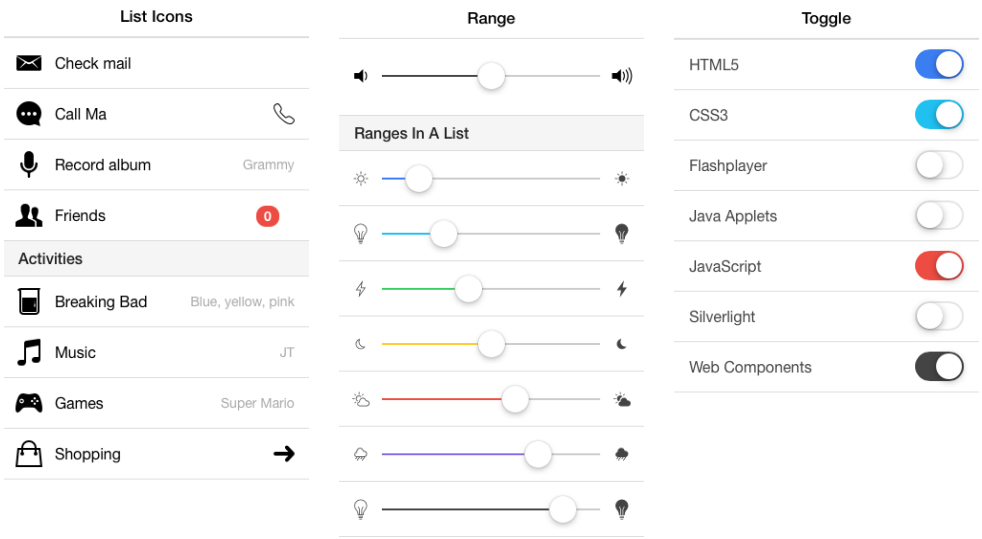


Figure 5.1: Ionic Component Examples

I structured the code in MVC (Model View Controller) pattern and created navigation between views using AngularJS. Angular's *directives* bind a view with the underlying data structure, for example - bind a table to a list, so updating views is made very easy.

5.2.2 NodeJS & MongoDB

NodeJS is completely asynchronous, so it can utilise IO operations (database query) very efficiently (compared to Apache for example) and serve magnitudes of more clients (Figure 5.2 [10]), despite being single-threaded. In fact, it is 2.5 magnitudes faster than PHP and cpu-memory efficient. As the system is meant to handle many users simultaneously, it is highly advised to use Node to handle such real-time communication, specially in tandem with hybrid platforms [11].

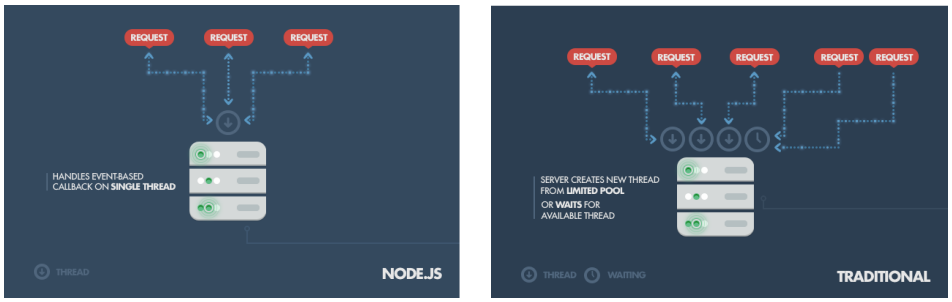


Figure 5.2: NodeJS

MongoDB is schema-less and document driven. As a NoSQL database, it is highly scalable and can handle unstructured data, multimedia and social-media content efficiently [24]. Indeed, Mongo's flexibility was necessary for fast prototyping and suitable for the app's intention.

5.2.3 Heroku & mLab

Out of the well known PaaS such as Bluemix [30], Openshift [27], AppEngine [22] etc., Heroku is by far the easiest to use (to get started and deploy), in my opinion, and has the best community support. A PaaS takes care of infrastructure tasks like hosting a server, scaling it etc. So, I could focus more on development. As for mLab, it is a generic MongoDB hosting

site with free usage upto 500 MB and easily integrated with Heroku.

5.2.4 Issues

There were some issues that could not be avoided with the selection of technology -

GCM (Google Cloud Messaging) [23] is the only entity that can send push notification to an Android phone (even though intermediate frameworks, e.g. Ionic Push, can be involved). While it is possible to verify if a request for notification has been accepted, when it gets delivered cannot be controlled in any way. Often a notification will be found as sent, but it will not actually arrive for hours. The probable reason is that, I used a *free* GCM service rather than a *paid* one.

Base64 images [17] are ASCII encodings of binary image files. Although, Amazon S3 [3] is the preferred service to store images, it is not free. So, I stored images as very large strings in MongoDB. This should be avoided a production-ready app.

5.3 Architecture

In this section I will discuss how different parts of the project comes together. Figure 5.3 is the complete architecture diagram which will be referred to in the following sections.

5.3.1 Client-Server Communication

The Ionic client app makes requests to the NodeJS server on Heroku to retrieve/create/update/delete various data artefacts. These requests are authorised by the *PassportJS Local Strategy* (by unique username-password, rather than a social network). The server then asynchronously handles each request.

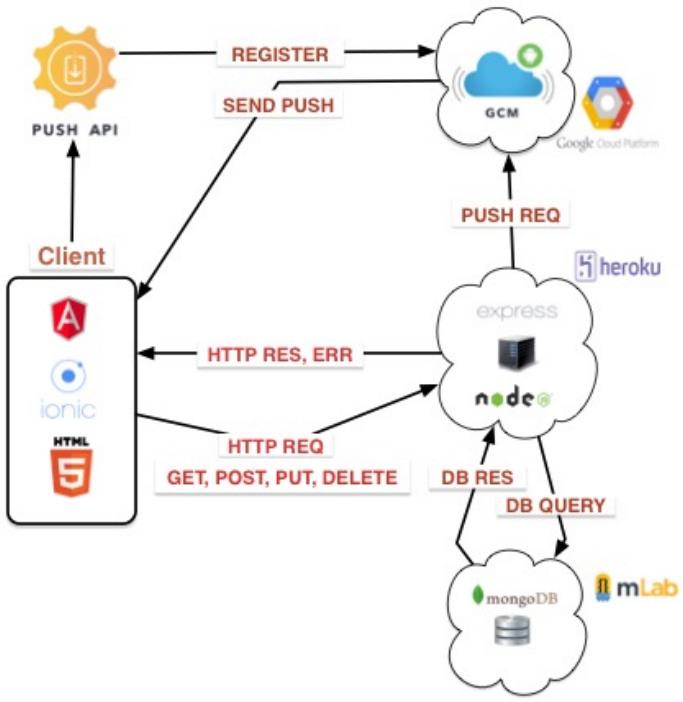


Figure 5.3: System Architecture

5.3.2 Server-Databse Communication

The server contacts the MongoDB database on mLab asynchronously (for which it has a key) for each client request. After the results reach the server, it asynchronously passes them to the Client. The client thus never directly contacts the database.

5.3.3 Push Notification

When the app is opened on a mobile first time, Client sends a GCM registration request via the Ionic Push API. This API registers the app, sends back a token to the client. Finally, the client sends the token to the server and it, in turn, saves it on mLab. In future, an event may be triggered on the server (e.g. new offer from a user) which may require a push notification to one-or-more clients. It sends a request to the GCM server to send a notification to the given tokens. GCM takes care of the actual sending of the notification.

Chapter 6

Implementation

In this chapter, I will explain in detail, how the different parts of the project were designed and built.

6.1 Data Models

The system needs to handle data regarding -

- *People* who will use the system.
- Details of *Something* that can be traded, lent, rented or sold etc.
- Record of *interaction* between people regarding some item of interest.

Data models were designed keeping these in mind. The models were developed as they were needed for user stories during a sprint.

6.1.1 Model Definitions

User : Users must have a unique username and a password (hashed). They can also have a default avatar (as Base64 or URI) and an initial rating of 0. Rating is primitive and only accounts for the average.

PushToken : Every User must have a PushToken object reference. These objects contain the device tokens generated by GCM by push registration.

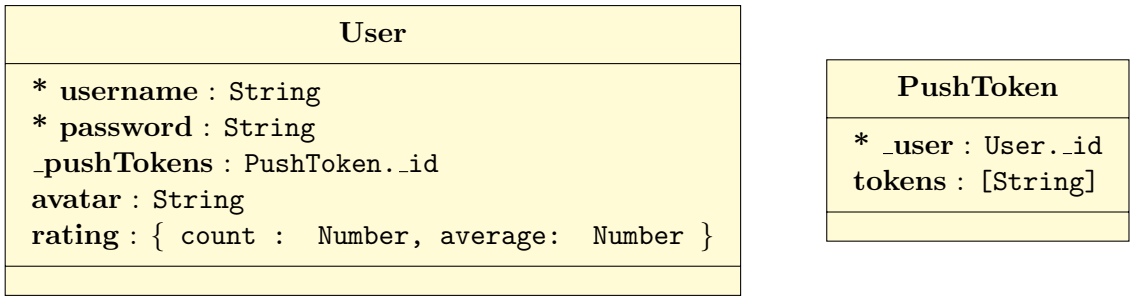


Figure 6.1: User Model

Thing : A Thing is an abstraction for either a service or an object. It must have a name and an owner. Other fields include name, description and type (category). `closed` records whether the item can be acquired or not, whereas `_reservedTo` is the User whose offer for the thing was last accepted.

Lend : A Lend object is created when a User wants others to borrow a Thing (which gets a reference to that object).

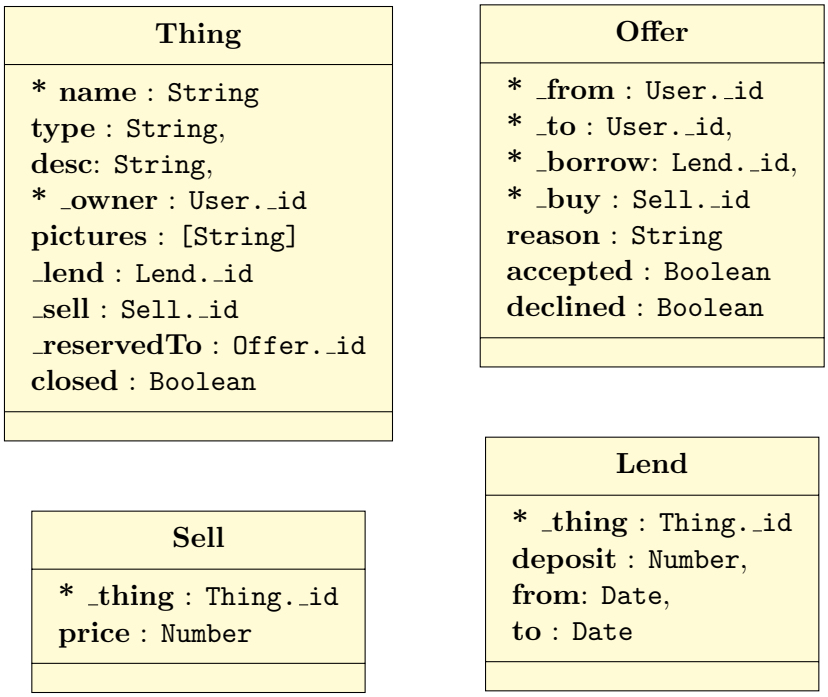


Figure 6.2: Thing Model

Sell : Similar to Lend, it only gets created when a User wants others to buy a Thing. A Thing can have both `_sell` and `_lend`.

Offer : An Offer is created when a User wants to buy/borrow (depending on which options are available) some item for some *reason*. It can be accepted or declined (both initially set to false).

6.1.2 Model Design Choices

Things can be *closed*, rather than removed from the system once a transaction takes place, so that, a borrowed item could be made *active* once it has been returned, rather than posting again.

Offers have both *accepted* and *declined* fields, rather than a toggle, because, the initial condition, where it is neither accepted nor declined, can be interpreted.

Images, as explained in Section 5.2.4, are saved directly as Strings, rather than BLOBs (Binary Large OBject) because of database free account restriction.

User objects do not directly have access to device tokens, so that, unused or invalid tokens can be destroyed in future without manipulating the main User object.

Lend or Sell options are whole objects, rather than a field on a Thing, because, an Offer should be distinguished between a borrow or a buy offer. Rather than creating an Offer hierarchy, these intermediate objects are referenced in respective Offer-Thing combos.

6.2 Server

In this section, I will discuss how the RESTful API was created, database and GCM integration was done and how authentication is performed.

6.2.1 Routes

REST (Representational State Transfer) is the software architectural style of WWW (World Wide Web) [16]. Simply put,

a client makes requests, usually HTTP (Hypertext Transfer Protocol) requests, at specific routes. The methods are usually CRUD (Create Read Update Delete).

This server was then hosted on heroku at http://project-api.herokuapp.com/. Each route is superseded by `/api` to denote a version number for the api. Table 6.1, 6.2 and 6.3 respectively shows routes for User, Thing and Offer objects. These routes were created with the following in mind -

- Users will need to add new items, retrieve all or searched items.
- The things will be updated, requested upon and eventually closed.
- Users will want to request for items, respond to requests and rate others.

Route `/users`	***Method***	***Use***	***Params***	***Returns***
`/`	`GET`	get all the users on the system	—	`[User]`
	`POST`	create a new user on the system	`User`	`User`
`/:uname`	`GET`	get a particular user	—	`User`
	`PUT`	update a single user	`User`	`User`
`/:uname/things`	`GET`	get all the things for a user	—	`[Thing]`
`/:uname/rating`	`PUT`	rate a user	`[User.rating]`	—
`/offers/to`	`GET`	get all the offers made to a user	—	`[Offer]`

Table 6.1: User Routes

Route `/things`	***Method***	***Use***	***Params***	***Returns***
`/`	`GET`	get all the things on system	—	`[Thing]`
	`POST`	create a new thing on system	`Thing`	`Thing`
`/:id`	`GET`	get a particular thing	—	`Thing`
	`PUT`	update a single thing	`Thing`	`Thing`
	`DELETE`	delete a single thing	—	`Thing`
`/:id/close`	`POST`	close a single thing	—	`Thing`

Table 6.2: Thing Routes

Route `/things/:id/offers`	***Method***	***Use***	***Params***	***Returns***
`/:offer_id`	`GET`	get all offers for a thing	—	`[Thing]`
`/:offer_id/accept`	`GET`	accept an offer	—	`Thing`
`/:offer_id/decline`	`POST`	decline an offer	—	`Thing`

Table 6.3: Offers Routes

6.2.2 Code Structure

There are 3 main sections in this server :

Models is where the `mongoose` data models are defined. Any data field validation, pre-post hooks to methods (e.g. a *post* hook to *delete* a User object deletes the corresponding *PushToken* object from DB) etc. are done here.

Controllers is where the callbacks for the http request handling are defined. Specially, `rest-framework.js` file contains my own simple framework to support HTTP GET, POST, PUT & DELETE methods. There is a controller for each model that is directly accessed from the client app.

Server is where the server configurations and http routes are defined. `server.js` attaches methods from the controllers

to respective routes, e.g. `ThingController.postThings` is attached to `/things` route.

6.2.3 Authentication using Passport

Passport is a NodeJS module which provides different strategies to authenticate applications including various social networks. In this project, I used the *Basic Strategy* or *BasicAuth* for authentication. Which means, each User object has a password that is hashed using *bcrypt* before it is saved to the database. A `passport.js` strategy is defined in `authController.js` which verifies a provided *plain text* password against the hashed password on the database. This strategy is then used to intercept every connection (except to create a new User), authenticate said connection and finally serve it. A failure to authenticate results in a HTTP 401 error.

6.2.4 Database Integration

Mongoose is a NodeJS module which provides built in type casting, validation, query building, hooks etc. for a MongoDB database. A database was created using a free service on mLab and an admin user was added. This admin, along with the database address was then put in configuration in `server.js`, thus establishing the connection. Mongoose provides an easy to understand *schema-like* definition structure to a NoSQL database. Mongoose also adds timestamps for object creation (`createdAt`) and latest update (`updatedAt`) to the object. It also provides an optional `populate` method to add all the referenced objects, when an object is retrieved.

6.2.5 GCM Integration

GCM is integrated in the server using the node-gcm module. First, I created a new project on GAE(Google App Engine), then added a server key for a GCM API. This key was then put in the configuration. At this point a notification request can be sent to GCM, given an array of device tokens and other parameters (e.g. the title, body, sound, icon etc.). GCM can be queried for the acceptance of the request, but the actual

sending is performed by GCM itself. The messages can have a time-to-live of 3 months, and can arrive at any point in between because only a *free* account is being used.

6.3 Client

6.3.1 Ionic Project Structure

An Ionic project, itself, is a NodeJS module. The `www/` directory must contain all the *code* using HTML, CSS and JavaScript. `www/index.html` is the starting point of the app. `plugins/` is where all the platform specific (e.g. written in Java for Android) plugins are kept. `platforms/` keeps generated code for different platforms. `ionic serve -l` will start a local server where both iOS and Android versions of the app are emulated (except for the plugins which are only activated on a mobile).

6.3.2 App Structure

`ui-router`, an Angular library, is used by Ionic to define the app in forms of *States*. Each state gets a *template* (an HTML file) as view and a *controller* (an AngularJS file) as the view's Angular controller. The controllers share data between themselves using Angular Factories. The template files are kept in `www/templates/` and all the JavaScript code is in `www/js/`. Figure 6.3 is a complete representation of the different components in the Client App.

6.3.3 States

Each state gets an URI (e.g. `/app/profile`). The URI routes to the template view of the app. States can be abstract (e.g. `app`), with child states (e.g `app.products`). Data from the database is requested and resolved by the state and injected into the controller before the views are loaded, for example: latest 20 products are fetched on loading the Products state. `ui-router` also automatically keeps track of navigation by using the states.

6.3.4 Controllers

Controllers hold the various data and functions that are shown or used on the view. For example: using the `ng-model` directive on a text-box, it can be bound to a variable in the view's controller. Thereafter, any changes to either the textbox or the variable will reflect on the other end. This is achieved by a regular *digest* cycle performed by the Angular engine. However, using many such directives in a view can lead to *significant* performance decrease.

6.3.5 Services

Controllers are view specific. However, there are common tasks that even distinctly different controllers will need to perform, database query for example. These are defined in Angular *Factories* or *Services*. These can be injected in the controllers or the state definitions, but are inaccessible from the view. If a function is needed on the view, it is defined in the controllers and those will, in turn, optionally use services.

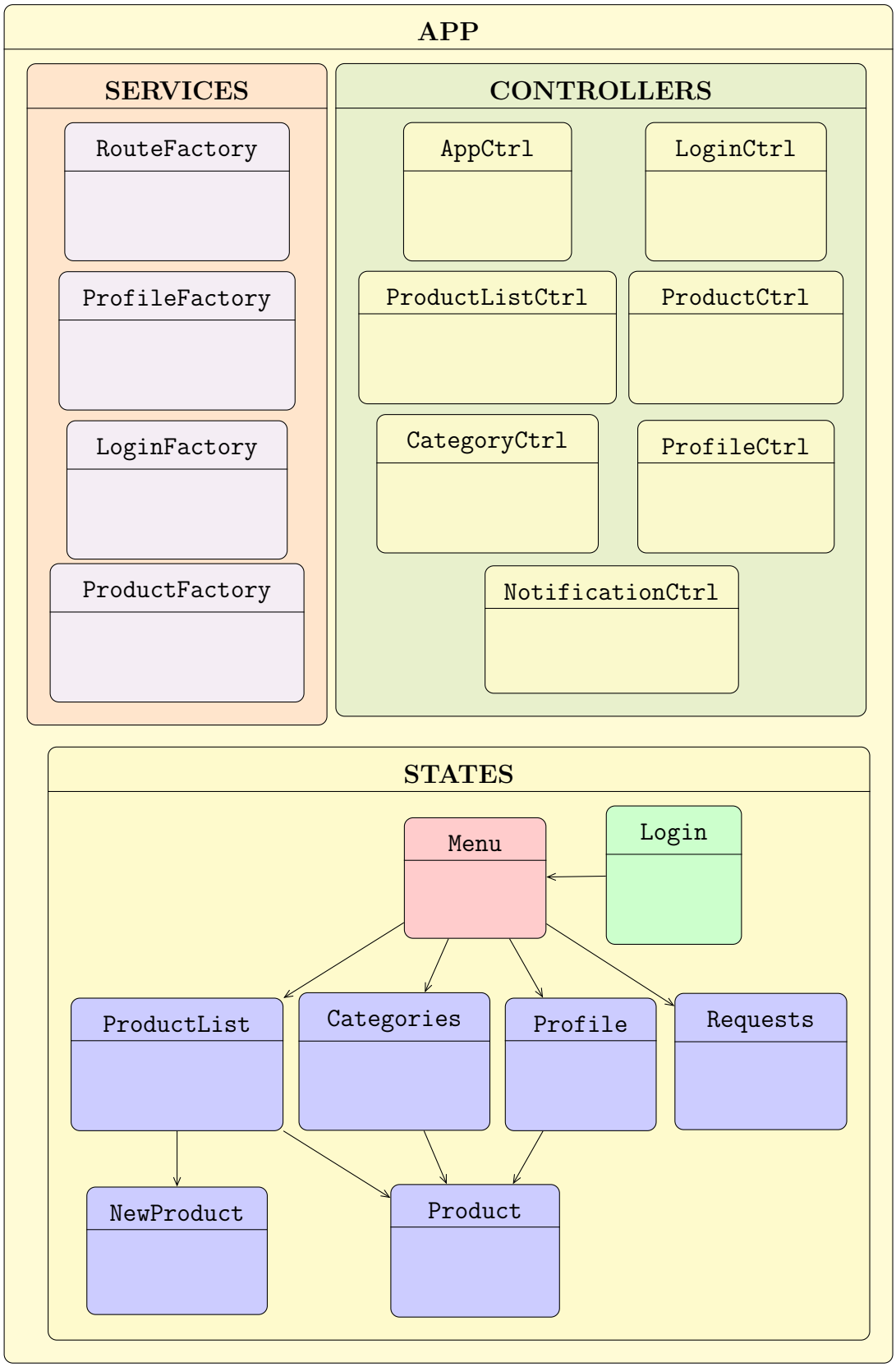


Figure 6.3: Client App Structure

Chapter 7

Testing

This project followed a Behaviour Driven Development (BDD) style from Sprint 3 for the server, and scenario driven GUI testing on the client. In this chapter, I discuss the procedure and the depth of testing in detail.

7.1 Server Test

As discussed in Section 2.4, comprehensive unit testing on the server for each function proved to be a large overhead to a prototyping project, even though agile SE dictates self-contained testing during development sprints. As a result, a more lenient and user-story-driven approach, BDD was used in the project. The format sits between integration and unit testing, but geared more towards the former. Each subsequent sprint also went through a comprehensive round of regression testing. Because the server's main purpose is to provide for a RESTful API, the routes for the API have been tested with boundary cases. Also, independent automated tests were performed to check the database objects' integrity. 100% of the REST routes that are exposed to the clients have been tested with automated functions given boundary conditions. Each test is self-contained because it creates a sandbox where the necessary objects are instantiated and cleaned up after success/fail. No external library except the development code was injected into the test suites. However, no granular unit testing was performed, which is necessary in a future production ready server, but avoidable for this short project.

7.1.1 REST API Test

The RESTful API exposes the routes about the User, Thing and Offer objects to the client. For example: a user story may be about borrowing an item. Thus, the route for borrowing is tested so that - users can only borrow a *borrowable* item which is *not closed*, not owned by *himself* etc. The result can be considered a *behaviour* given the *condition*. So, a test is performed for *each* such conditions that may appear on the client side. Figure 7.1 is the comprehensive list of the tests.

As seen in the figure, the tests are easily readable. These tests can be used (and *was* in this project) in the sprint review to negotiate with the customer to *accept* a user-story. For example: the unavoidable unreliability of GCM led me to accept the user-story about push notification, even though the test to expect a notification within 10 seconds would randomly fail.

```
ip-038-120:Server shakib-binhamid$ mocha test/routes.js --timeout 5000


  Routing
    User Account
      ✓ should not post a new user without a username (245ms)
      ✓ should not post a new user without a password (147ms)
      ✓ should successfully post a new user (733ms)
      ✓ should return error trying to save duplicate username (136ms)
      ✓ should return a correct user (272ms)
      ✓ should edit a user (399ms)
      ✓ should rate a user (804ms)
    Thing
      ✓ should not create a Thing because not authorisation
      ✓ should create a new Thing (662ms)
      ✓ should return the created Thing (413ms)
      ✓ should not update the Thing if not the owner (268ms)
      ✓ should update the Thing (402ms)
      ✓ should not delete the Thing if not owner (266ms)
    Offer
      ✓ should not make an offer to borrow to one's own item (144ms)
      ✓ should not make an offer to buy to oneself (154ms)
      ✓ should not make an offer to an invalid user (410ms)
      ✓ should not make an offer from an invalid user (271ms)
      ✓ should not make an offer to borrow that you cannot borrow (526ms)
      ✓ should not make an offer to buy that you cannot buy (532ms)
      ✓ should make an offer to borrow the thing (651ms)
      ✓ should make an offer to buy the thing (2478ms)
      ✓ should accept an offer to borrow the thing (reservation OK) (1302ms)
      ✓ should decline an offer to buy the thing (reservation NOT removed) (1309ms)
      ✓ should decline the previous offer to borrow the thing (reservation REMOVED) (1320ms)
      ✓ should close a product (406ms)
      ✓ should not close a closed product (275ms)
      ✓ should not accept offer to borrow a closed product (665ms)
      ✓ should not accept offer to buy a closed product (663ms)


  28 passing (20s)
```

Figure 7.1: REST API Behaviour Test

7.1.2 Database Integrity Test

These tests go through each object (*document* in this case) in the database. It then validates the referenced objects and constraints on their fields. For example: in Figure 7.2, a test fails to find a reservation for a Thing, an offer/request for which was accepted. It means that the offer that it was reserved to is *missing* and the Thing needs to be mended. Note that the actual *fixing* is *not* automated, rather I used these tests to monitor the database.

```
ip-038-120:Server shakib-binhamid$ mocha test/db_integrity_checker.js --timeout 150000


  Checking User Integrity
    ✓ check for PushToken integrities (1699ms)
    ✓ send Push notifications to all the active users (1029ms)

  Checking Thing Integrity
    ✓ check for owner for Thing (1064ms)
    ✓ check for Lend for Thing (781ms)
    ✓ check for Sell for Thing (648ms)
    1) check for reservations for Thing
    2) check for Closed for Thing

  Checking Offer Integrity
    3) check for sender for offers
    4) check for receiver for offers
    ✓ check for lend for offers that are made to borrow (376ms)
    ✓ check for sell for offers that are made to buy (1625ms)


  7 passing (11s)
  4 failing
```

Figure 7.2: DB Integrity Check

```
1) Checking Thing Integrity check for reservations for Thing:

     Uncaught AssertionError: expected 0 to be 1
     + expected - actual

     -0
     +1

     at Assertion.fail (node_modules/should/lib/assertion.js:91:17)
     at Assertion.Object.defineProperty.value (node_modules/should/lib/assertion.js:163:19)
     at test/db_integrity_checker.js:94:37
     at Query.<anonymous> (node_modules/mongoose/lib/query.js:2124:28)
     at node_modules/mongoose/node_modules/kareem/index.js:177:19
     at node_modules/mongoose/node_modules/kareem/index.js:109:16
```

Figure 7.3: Error in Testing

The database integrity tests were also performed each sprint. In fact, these tests proved to be so helpful, that I ran them atleast once a development day. I imagine the tests to pave a path to future monitoring mechanism that may be put in place.

7.2 Client Test

It was not possible to make automated GUI tests for the client app for the following reasons -

- Touch-event mocking is not possible in Ionic
- Ionic libraries cannot be isolated
- AngularJS can usually be mocked, but Ionic integrates AngularJS, so it cannot be extracted either

In short, BDD or TDD is not possible, because the Ionic framework does not support such features. Even the community is silent in this regard. In fact, as discussed in Chapter 2 in detail, this is a common problem across mobile app development, specially hybrid apps. I believe the reasons are -

- Ionic is undergoing a complete overhaul for version 2. So, the developers are mostly ignoring the current version.
- The production hybrid apps have proprietary, in-house test environments such as in Facebook etc.

As a result, I tested the client app regularly against pre-defined set of use case scenarios during development and with end users during UX evaluation.

7.2.1 GUI Testing

To test the GUI, I have exhaustively listed all the use case scenarios in Appendix F. This was built up as each feature had been added. At every sprint, I walked through every scenario multiple times, on two different Android mobiles - Moto X 1st Gen and Moto X Play. I used the list as my primary guide in client app testing. The list is comparable to a 100% coverage of BDD testing on client.

7.2.2 User Evaluation

I have also evaluated the client app with the end users. More about this is discussed in Chapter 8, Appendix C and E.

Chapter 8

User Participation

In this chapter, I discuss why and how I have interviewed the potential users of the app, as well as my experiences in doing so.

8.1 Reason

Even though I had a vision of what the app could be, the specific user stories must be structured and prioritised *only* to the *end users'* needs. So, the first round of interviews were run to empathise with my users' pain points. For example - I asked the users - *Which of these features are more important to you and in what order and why such order?* (Appendix C.1), and the response dictated where I needed to concentrate. This idea stems from the simple fact that - "those affected by a design should have a say in the design process" [7]. In fact, I turned the responses into user stories in backlog (See Chapter 4) and their pain points allowed me to prioritise them.

I also felt it necessary to define UX goals (e.g. evaluate navigation and white space use etc.) and run interactive interviews with the *end users*. It is because users typically interact with mobile apps very differently to their desktop counterparts (e.g. average user spends about an hour a day on apps, whereas each session only lasts around a minute! [8]). So, even established assumptions about interaction often falls behind [39]. Questions like - *How do you know if someone responded to the 'x' you posted earlier?* (Appendix C.2) pointed out some obvious faults in the app during UX evaluation when they could not

complete the task in expected time or take the expected navigation route. I turned the issues into estimated work items (Appendix E) and fixed them in the following sprint.

8.2 Methodology

For each user participation period, I defined a goal first e.g. gain a better understanding of sharing economy between students in UoS and produce wireframes for RE period. Then a data collection method (e.g. interview, questionnaire, focus group, observations etc.) was chosen. I chose individual structured interviews with consented audio recording, physical notes and observation for qualitative data. I also asked them to speak aloud what they were thinking or doing. This method was chosen because video recording is time consuming and difficult to get *right*, questionnaires often remove empathy and focus groups proved difficult to organise [4]. I also believed qualitative data will provide more information than quantitative for my purposes.

Then I made interview scripts (with generally open ended approach to the questions), estimated time taken, applied for ethical approval with consent form and data protection guarantee. The notes taken were cross checked with the recorded audio for each individual afterwards. Data was kept secure on my personal laptop. It was then turned into user stories and prioritised based on observation. I also used the information to confirm prior assumptions.

8.3 Experiences in RE

I found two very distinct groups of interviewees in my first interview period - one group will express themselves a lot more than the others, rarely there was an *ideal* persona. So, it soon proved difficult to keep track of the structured questions as the answers would often overlap or that there was barely any data from certain interviewees. As a result, interviews almost

always ran overtime than what was estimated.

The planned interactive wireframing session, where users draw out how they believed their most desired features would be implemented, was very successful and productive. Every interviewee engaged easily and provided me with crucial ideas on how to structure UI elements and implement navigation. Similar experiences with card sorting technique, where they prioritised the important featured, which helped me prioritise user stories even in later stages.

I had also found it extremely helpful to have a very early prototype running on the mobile device [32] that I used to develop the app, even during RE session. It gave a better perspective about wireframing to both the participants and I, and removed the usual gulf between the two parties [45].

8.4 Experiences in UX Evaluation

I had corrected my approach to the interview questions based on previous the experiences and made them slightly more specific and task based so as to keep the interviewees focused. Resultantly, the sessions ran almost exactly as long as they were estimated to run.

Very simple bugs, some of which I was aware of but never fixed because they were not prioritised (e.g. reverse chronological ordering of products etc.), distracted the participants from main tasks. Although they could have been fixed in between different sessions, I did not do so as it would jeopardise the study. In future, I will fix all the simple and obvious bugs before I run evaluation sessions.

The results from the sessions were very satisfactory. In general, the participants found the interface to be *intuitive*, *fun* and *easy* to navigate. They also found the information layout to be *clean* and animations to add *life* to the app.

There were 10 easily fixable (1-2 hours) and 4 much harder to

fix (more than 10 hours) issues found (see Appendix E). Most of these (9/10 of the former, 2/4 of the latter) were addressed in the sprint that followed soon after. I can truly appreciate the importance of user evaluation at this point and would recommend doing so at least after every two iterations in any future work.

Chapter 9

Evaluation & Future Work

In this chapter, I discuss how well the project went, which parts can be improved upon, what is left to complete as well as how others can use the results of the project in their own work.

9.1 Project Success

The selection of technologies proved to be very satisfactory. Interaction between the client and the server using JSON was made seamless due to full stack JavaScript development. The frameworks' community support was generally helpful, albeit their own rapid development made version specific support tedious. It was very easy to add new modules and libraries using `npm` and access mobile hardware plugin support by Ionic. Overall, fast prototyping was possible using the frameworks, as expected.

I believe that running the user interviews was a great success. I now have practical knowledge of how to prepare for such sessions, the methodology of interviews and observations and most importantly, interpreting the results. As found in the literature, it is extremely important to evaluate a prototype often; in my opinion - at least once every two sprints. This removes tunnel vision, sets realistic goals and prioritises the remaining work or bugs.

The server has been sufficiently tested with automated test suits, as discussed in Chapter 7 and the client was tested by

scenerios and user interviews. This is satisfactory for a prototyping project within these time constraints.

Finally, the project organisation was a success. Development started early and finished a little over a week earlier than planned. Sprints were regularly reviewed, backlog was managed throughout the development period and bugs were documented. I believe that the app has reached an alpha stage, where there are a number of bugs and a few features are missing, but it is ready to be field tested and will reach feature-lock and beta in 3-4 iterations.

9.2 Points of Improvement

The server was not satisfactorily *unit* tested, while the Ionic framework provides none to very little support in automated testing in general. If this were to be a production ready service, I would, at least, rebuild the server in TDD style.

Push notification support is poor in *every* hybrid app framework. The current choice of libraries is quite helpful, but will likely require some rewriting themselves to suit this app's needs. There needs to be a better guarantee of the delivery of the notifications.

Data models can be made simpler in the server. For example: Lend and Sell objects can be integrated in Thing, PushToken should be absorbed in User etc. This structure was chosen at first to prevent certain fields to be retrieved. In fact, later I discovered that the `mongoose` library natively provides methods that perform the intended behaviour. So, it is no longer necessary to separate the objects.

Finally, as a production-ready app, a paid Amazon S3 database should be chosen and integrated with the server to store images as BLOBs, rather than Base64 strings. This removes a large overhead from the main database.

9.3 Future Development

The app must have an improved rating system for the users, for example: there should be sub-ratings, e.g. timeliness, care-of-product etc. This project focused on the overall app, but a production-ready community driven app will heavily depend on a detailed and reliable rating system. In fact, some user research may be performed about how such a system may work before development.

Next, the in-app communication should be implemented. This could be done using the university email service, regular chatting frameworks, sms-call systems etc.

Finally, the app should have an automated recommendation notification system for desired items (wish-list) for the users. This feature should not require any additional frameworks, but such *recommendation* system is usually difficult to get *right*. Research should also be done on how similar systems handle the UX on client, e.g. Amazon or EBay.

9.4 Usefulness in Other Projects

The client app structure, feature list and server models can provide a useful framework for software projects that have community driven interaction models, e-commerce etc. They can also use the project plan or sprint times from the Gantt chart. Hopefully, it will also convince mobile app developers to invest resources in user interaction sessions.

This project has used technologies from various paradigms which can influence other projects. For example: student projects with asynchronous servers can get guidelines from the nodejs code, RESTful api developers can benefit from the relevant parts in server, app developers can also benefit enormously from the ionic project structure etc.

Chapter 10

Conclusion

Even though *Software Engineering* itself is now considered an established and mature study, application development for smartphones is a recent and diverse addition in the industry. Because of the wide variety of smartphones used by the people, there is also a plethora of development toolkits available for multiple different platforms. Although each major mobile OS has its own distinct engineering practices and architectural patterns, it is expected that an app is to be implemented and supported well simultaneously for multiple platforms. This project has tried to tackle the problem by using hybrid technologies to develop a mobile app.

It is also extremely difficult to rigidly define mobile app requirements in this volatile industry and satisfy user experiences for multiple platforms at the same time. I have successfully used scrum (iterative development and review) to develop the app, and organised interview sessions to engage the end-users in requirements elicitation, design and evaluate the app in hopes to evade such difficulties.

However, the benefits of fast development and multi-platform support comes with the price of automated testing on client side. While there are established practices in server building using NodeJS, apps built with such hybrid client frameworks as Ionic are nearly impossible to unit test because of lack of mocking and isolation support. In fact, there seems to be very few, if any, example of community support in this regard. This is certainly a barrier, but given the aims of the project, the is-

sue can be considered a discovery of fact, rather than a major shortcoming.

Finally, I consider the project to be successful, and valuable for future students, because it demonstrates relevant agile software development processes such as scrum, daily testing, evolving user stories etc. within the short project frame, experiments with emergent technologies in mobile app industry and evaluates the results against end users' expectations.

Bibliography

[1] A systematic literature review on agile requirements engineering practices and challenges. *Computers in Human Behavior*, 51, Part B:915 – 929, October 2015. doi:10.1016/j.chb.2014.10.046.

[2] Pia A. Albinsson and B. Yasanthi Perera. Alternative marketplaces in the 21st century: Building community through sharing events. *Journal of Consumer Behaviour*, 11(4):303 – 315, July 2012. doi:10.1002/cb.1389.

[3] Amazon. Amazon Simple Storage Service (S3) - Object Storage, February 2016. Last Accessed 10 March 2016. URL: https://aws.amazon.com/s3/.

[4] Leena Arhippainen and Marika Tähti. Empirical evaluation of user experience in two adaptive mobile application prototypes. In *Proceedings of the 2nd international conference on mobile and ubiquitous multimedia*, pages 27 – 34, 2003.

[5] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mallor, Ken Shwaber, and Jeff Sutherland. The agile manifesto. Technical report, The Agile Alliance, 2001.

[6] Russell Belk. You are what you can access: Sharing and collaborative consumption online. *Journal of Business Research*, 67(8):1595 – 1600, 2014. doi:10.1016/j.jbusres.2013.10.001.

[7] E. Bjögvinsson, P. Ehn, and P. A. Hillgren. Design things and design thinking: Contemporary participatory design

challenges. *Design Issues*, 28(3):101 – 116, July 2012. doi:10.1162/DESI_a_00165.

[8] Matthias Böhmer, Brent Hecht, Johannes Schöning, Antonio Krüger, and Gernot Bauer. Falling asleep with angry birds, facebook and kindle: A large scale study on mobile application usage. In *Proceedings of the 13th International Conference on Human Computer Interaction with Mobile Devices and Services*, MobileHCI '11, pages 47 – 56, New York, NY, USA, August 2011. ACM. doi:10.1145/2037373.2037383.

[9] Achim D. Brucker and Michael Herzberg. On the static analysis of hybrid mobile apps: A report on the state of apache cordova nation. In Juan Caballero and Eric Bodden, editors, *International Symposium on Engineering Secure Software and Systems (ESSoS)*, Lecture Notes in Computer Science, pages 72 – 88, Heidelberg, March 2016. Springer-Verlag. doi:10.1007/978-3-319-30806-7_5.

[10] Tomislav Capan. Why the hell would i use node.js? a case-by-case tutorial, February 2013. Last Accessed 11 March 2016. URL: https://www.toptal.com/nodejs/why-the-hell-would-i-use-node-js.

[11] Ioannis K. Chaniotis, Kyriakos-Ioannis D. Kyriakou, and Nikolaos D. Tselikas. Is node.js a viable option for building modern web applications? a performance evaluation study. *Computing*, 97(10):1023–1044, October 2015. doi:10.1007/s00607-014-0394-9.

[12] Andre Charland and Brian Leroux. Mobile application development: Web vs. native. *Communications of the ACM*, 54(5):49 – 53, May 2011. doi:10.1145/1941487.1941504.

[13] Drifty Co. Ionic: Advanced html 5 hybrid mobile app framework, February 2016. Last Accessed 10 March 2016. URL: http://ionicframework.com.

[14] Drifty Co. ngcordova - simple extensions for common cordova plugins - by the ionic team, February 2016. Last

Accessed 10 March 2016. URL: http://ngcordova.com/docs/plugins/.

[15] ObjectLabs Corporation. Mongodb hosting: Database-as-a-service by mlab, February 2016. Last Accessed 10 March 2016. URL: https://mlab.com.

[16] Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. In *Proceedings of the 22nd International Conference on Software Engineering*, ICSE '00, pages 407 – 416, New York, NY, USA, 2000. ACM. doi:10.1145/337180.337228.

[17] Internet Engineering Task Force. The base16, base32, and base64 data encodings, oct 2006. Last Accessed 10 March 2016. URL: https://tools.ietf.org/html/rfc4648.

[18] Node.js Foundation. Express - node.js web application framework, February 2016. Last Accessed 10 March 2016. URL: http://expressjs.com.

[19] Node.js Foundation. Node.js, February 2016. Last Accessed 10 March 2016. URL: https://nodejs.org/en/.

[20] Boby George and Laurie Williams. A structured experiment of test-driven development. *Information and Software Technology*, 46(5):337 – 342, 2004. Special Issue on Software Engineering, Applications, Practices and Tools from the ACM Symposium on Applied Computing 2003. doi:10.1016/j.infsof.2003.09.011.

[21] Google. Angularjs - superheroic javascript mvw framework", February 2016. Last Accessed 10 March 2016. URL: https://angularjs.org.

[22] Google. App engine - platform as a service - google cloud platform, February 2016. Last Accessed 10 March 2016. URL: https://cloud.google.com/appengine/.

[23] Google. Cloud messaging — google developers, February 2016. Last Accessed 10 March 2016. URL: https://developers.google.com/cloud-messaging/.

[24] C. Győrödi, R. Győrödi, G. Pecherle, and A. Olah. A comparative study: Mongodb vs. mysql. In *2015 13th International Conference on Engineering of Modern Electric Systems*, pages 1–6, June 2015. doi:10.1109/EMES.2015.7158433.

[25] M. L. Hale and S. Hanson. A testbed and process for analyzing attack vectors and vulnerabilities in hybrid mobile apps connected to restful web services. In *2015 IEEE World Congress on Services*, pages 181–188, June 2015. doi:10.1109/SERVICES.2015.35.

[26] Juho Hamari, Mimmi Sjöklint, and Antti Ukkonen. The sharing economy: Why people participate in collaborative consumption. *Journal of the Association for Information Science and Technology*, pages 1 – 13, July 2015. doi:10.1002/asi.23552.

[27] Red Hat. Openshift: Paas by red hat, built on docker and kubernetes, February 2016. Last Accessed 10 March 2016. URL: https://www.openshift.com.

[28] Heroku. Heroku — cloud application platform, February 2016. Last Accessed 10 March 2016. URL: https://www.heroku.com.

[29] TJ Holowaychuk. Should.js api documentation, February 2016. Last Accessed 10 March 2016. URL: https://shouldjs.github.io.

[30] IBM. Ibm bluemix - next generation cloud app development platform, February 2016. Last Accessed 10 March 2016. URL: https://console.ng.bluemix.net.

[31] M. E. Joorabchi, A. Mesbah, and P. Kruchten. Real challenges in mobile app development. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 15–24, October 2013. doi:10.1109/ESEM.2013.9.

[32] Eeva Kangas and Timo Kinnunen. Applying user-centered design to mobile application development. *Communi-*

cations of the ACM, 48(7):55 – 59, July 2005. doi:10.1145/1070838.1070866.

[33] A. Karadimce and D.C. Bogatinoska. Using hybrid mobile applications for adaptive multimedia content delivery. In *Electronics and Microelectronics (MIPRO), 2014 37th International Convention on Information and Communication Technology*, pages 686–691, May 2014. doi:10.1109/MIPRO.2014.6859654.

[34] Nabil Litayem, Bhawna Dhupia, and Sadia Rubab. Review of cross-platforms for mobile learning application development. *(IJACSA) International Journal of Advanced Computer Science and Applications*, 6(1):31 – 39, February 2015. doi:10.14569/IJACSA.2015.060105.

[35] I. Malavolta, S. Ruberto, T. Soru, and V. Terragni. End users' perception of hybrid mobile apps in the google play store. In *2015 IEEE International Conference on Mobile Services*, pages 25–32, June 2015. doi:10.1109/MobServ.2015.14.

[36] mocha. Mocha - the fun, simple, flexible javascript test framework, February 2016. Last Accessed 10 March 2016. URL: https://mochajs.org.

[37] Mareike Möhlmann. Collaborative consumption: determinants of satisfaction and the likelihood of using a sharing economy option again. *Journal of Consumer Behaviour*, 14(3):193 – 207, 2015. doi:10.1002/cb.1512.

[38] Inc. MongoDB. Mongodb for giant ideas, February 2016. Last Accessed 10 March 2016. URL: https://www.mongodb.com.

[39] M. J. O'Sullivan and D. Grigoras. User experience of mobile cloud applications - current state and future directions. In *2013 IEEE 12th International Symposium on Parallel and Distributed Computing*, pages 85 – 92. IEEE, June 2013. doi:10.1109/ISPDC.2013.20.

[40] B. Ramesh, L. Cao, and R. Baskerville. Agile requirements engineering practices and challenges: an empirical study.

Information Systems Journal, 20(5):449–480, 2010. doi:10.1111/j.1365-2575.2007.00259.x.

[41] M. J. Rees. A feasible user story tool for agile software development? In *Software Engineering Conference, 2002. Ninth Asia-Pacific*, pages 22 – 30, 2002. doi:10.1109/APSEC.2002.1182972.

[42] Salesforce. Native, html5, or hybrid: Understanding your mobile application development options, April 2015. Last Accessed 10 March 2016. URL: https://developer.salesforce.com/page/Native,_HTML5,_or_Hybrid:_Understanding_Your_Mobile_Application_Development_Options.

[43] Ken Schwaber. *Business Object Design and Implementation: OOPSLA '95 Workshop Proceedings 16 October 1995, Austin, Texas*, chapter SCRUM Development Process, pages 117 – 134. Springer London, London, 1997. doi:10.1007/978-1-4471-0947-1_11.

[44] C. Solis and X. Wang. A study of the characteristics of behaviour driven development. In *2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 383–387, August 2011. doi:10.1109/SEAA.2011.76.

[45] Arnold P. O. S. Vermeeren, Effie Lai-Chong Law, Virpi Roto, Marianna Obrist, Jettie Hoonhout, and Kaisa Väänänen-Vainio-Mattila. User experience evaluation methods: Current state and development needs. In *Proceedings of the 6th Nordic Conference on Human-Computer Interaction: Extending Boundaries*, NordiCHI '10, pages 521 – 530, New York, NY, USA, 2010. ACM. doi:10.1145/1868914.1868973.

[46] Anthony I. Wasserman. Software engineering issues for mobile application development. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, FoSER '10, pages 397 – 400, New York, NY, USA, November 2010. ACM. doi:10.1145/1882362.1882443.

[47] Laurie Williams, E. Michael Maximilien, and Mladen Vouk. Test-driven development as a defect-reduction practice. In *Proceedings of the 14th International Symposium on Software Reliability Engineering*, ISSRE '03, pages 34 –, Washington, DC, USA, 2003. IEEE Computer Society. doi:10.1109/ISSRE.2003.1251029.

Appendix A

Read Me

This project was done using a Macbook Air Mid 2012 with Mac OS X 10.11 El Capitan. So, the following guide is confirmed to work on such a system.

However, it should work without any changes on a recent Unix based system and with slight changes on Windows. The framework websites provide more knowledge on platform specific issues.

The following sections must be completed in order.

A.1 Install Node.JS

First install the `node.js` runtime from `https://nodejs.org/en/`. This will also install the package manager `npm`. On the development system it was `0.12.7`.

You can update existing versions using `sudo npm install npm -g`. Also, check the current version using `node --version`.

A.2 Install Express.JS

Use `sudo npm install -g express`. The `-g` option installs the package globally. This is necessary to start the server locally.

A.3 Add Database

This project only used a remote database on mLab at `https://mlab.com`. Create a free account here, add a database, add a *db user* and take note of the username-password. On mLab, over at your database page, you should see the *db URI*. Fill in the username-password and overwrite `server.js` and `test/config-debug.js` at the Server project root.

A.4 Install Heroku Toolbelt

If you are using Heroku, as was used in this project, then you must install the Heroku Toolbelt from `https://toolbelt.heroku.com`.

A.5 Heroku Deploy

Once you have created a Heroku account (free), `cd` over to the *Server* project root. Then use `heroku login` to log into your account from CLI. The server *must* be a *git* repository. If it is not, use `git init` to create one.

Use `heroku create <app-name>` where `<app-name>` is what you want to call your server. This will add a *Heroku git remote*.

Finally, use `git push heroku master` to push the `master` branch to Heroku. It will now be deployed to `<app-name>.herokuapp.com`.

You can use `heroku logs --tail` to see real-time logs from your remote server.

A.6 Server local

Use `npm install` while in the Server project root. It will install all the modules (including dev-dependencies) from `package.json`.

Finally, use `node server.js` to run the server. You should install *nodemon* by `sudo npm install -g nodemon` to automatically restart the server on code change.

A.7 Install Ionic & Cordova

Use `sudo npm install -g cordova ionic` to install both Ionic and Cordova CLI.

It is recommended that you use `Google Chrome` as your *default* browser.

A.8 Add Platforms to Ionic Project

Apart from *Android*, which should already be added to the project in the source code provided, you can add other platforms. For example, to add iOS, use `ionic platform add ios`.

You *must* install Android Studio from `http://developer.android.com/sdk/index.html` to build for Android and XCode from `https://developer.apple.com/xcode/download/` to build for iOS.

A.9 Install Plugins & Ionic Libraries

To add plugins for a platform, first add the platform, then use `ionic plugin add <plugin>`. Replace `<plugin>` with cordova-plugin-device, cordova-plugin-console, cordova-plugin-whitelist, cordova-plugin-splashscreen, cordova-plugin-statusbar, ionic-plugin-keyboard, com-badrit-base64, cordova-imagePicker.git, cordova-plugin-file. Use these *one by one*.

Some might already be installed depending on your Ionic version. If they are, just ignore and carry on.

You can also perform `ionic state reset` to re-download all the plugins from `package.json` file of your Ionic project.

A.10 Add GCM API Key

Create a Google account, go to `https://console.developers.google.com`, create a project. Add *GCM* as an API, create a *Server Key*

and take note of it.

Then update `controllers/offer.js` and add your GCM API key.

A.11 Set up Ionic Push

Push is not supplied as a standard with Ionic. Create an Ionic account. Look at `http://docs.ionic.io/v1.0/docs/push-android-setup` for GCM integration in Ionic.

In essence, use `ionic add ionic-platform-web-client` on the Client project root. Then `ionic push --google-api-key <your-google-api-key>` and `ionic config set gcm_key <your-gcm-project-number>`. Head over to `http://apps.ionic.io`, log in using your ionic account and you will see your push service connected with your app.

A.12 Change Host id in Client

Open up `<clinet-project-root>/www/js/services.js` and change `routes.APP` to your server app address. You can also toggle `development` to access either the local server or the remote one.

A.13 App Build

`cd` to Client project root. Set `development` is set to `false` in `services.js`. Once you have added the desired platform and plugins (and platform SDK), use `ionic build <platform>` to build an installer for the platform.

For android, a `android-debug.apk` file will be created at `/platforms/android/build/outputs/apk/`. Transfer the file over to the mobile and install it.

A.14 App local

Disable web-security to allow cross origin policies in chrome. Look online for you platform specific instruction. Otherwise, your app cannot receive data in browser emulation. At Client project root, use `ionic serve -l` to deploy both iOS and Android emulation.

Appendix B

Libraries Used

Below is the complete list of libraries, frameworks that were used in any way in the project and their github source links -

Ionic Framework v.1.1.1 by *Drifty Co*
https://github.com/driftyco/ionic

AngularJS v.1.4.3 by *Google*
https://github.com/angular/angular.js

Phonegap base64 v.0.2.0 by *Hazem Hagrass*
https://github.com/hazemhagrass/phonegap-base64

Cordova ImagePicker v.1.1.0 by *Wymsee*
https://github.com/wymsee/cordova-imagePicker

Cordova Datepicker v.0.9.3 by *Vitalii Blagodir*
https://github.com/VitaliiBlagodir/cordova-plugin-datepicker.git

Phonegap Plugin Push v.1.5.3 by *Adobe PhoneGap Team*
https://github.com/phonegap/phonegap-plugin-push.git

Angular UI Router v.0.2.7 by *Angular UI Team*
https://github.com/angular-ui/ui-router.git

Ionic Material v.0.4.0 by *Zach Fitzgerald*
https://github.com/zachsoft/Ionic-Material/

Ionic Rating v.0.1.0 by *Fraser Xu*
https://github.com/fraserxu/ionic-rating.git

ngCordova v.0.1.23-alpha by *Drifty Co*
https://github.com/driftyco/ng-cordova.git

NodeJS v.0.12.7 by *NodeJS Foundation*
https://github.com/nodejs/node

ExpressJS v.4.13.3 by *TJ Holowaychuk*
https://github.com/strongloop/express.git

MongooseJS v.4.2.2 by *Guillermo Rauch*
https://github.com/Automattic/mongoose/

Mongoose Deep Populate v.2.0.3 by *Buu Nguyen*
https://github.com/buunguyen/mongoose-deep-populate.git

Node GCM v.0.13.0 by *Marcus Farkas*
http://github.com/ToothlessGear/node-gcm

PassportJS v.0.2.2 by *Jared Hanson*
https:////github.com/jaredhanson/passport.git

Mocha v.2.3.4 by *MochaJS*
https://github.com/mochajs/mocha

ShouldJS v.8.0.2 by *TJ Holowaychuk*
https://github.com/shouldjs/should.js.git

Supertest v.1.1.0 by *TJ Holowaychuk*
https://github.com/visionmedia/supertest.git

Appendix C

Interview Scripts

C.1 Interview 1

- What items do you own, but not use often (Once a week or less)?
- Which of these items would you share with others?
- Why those items and not the others? (If none, then why?)
- If another student were to borrow your item, which items -
  - Would you have them pay for i.e. sell? Why?
  - Would you have them put a deposit for? Why?
  - Would you make them share an item of their own for i.e. swap? Why?
  - Would you let them borrow regardless? Why?
  - Anything else you have to add?
- How much does trusting the borrower or lender get in these decisions?
- Tell me how you would increase trust in the community to share your belongings?
- How can the borrower increase the trust?
- How can the lender increase the trust?
- How would you go about & carry out the transaction? How would set up how and where to meet or post?

- What would you expect to see in as you open the application?
- What would you want to do first after you open it?
- What other functions would you perform as a user? As a borrower ? As a sharer?
- Which of these features are more important to you and in what order and why such order?
- How would you perform the top 3 features that you have selected? (Ask them to draw out a flowchart like steps for the functions)

C.2 Interview 2

- You have just installed the app. You open it for the first time.
  - What do you see?
  - What do you want to do now?
-
  - Navigate back to the home page. (steps & time)
  - Try to find your profile. How else could you find your profile? (steps & time)
- You want to post this 'x' on the community to sell, but you also don't mind lending it. What do you do? (steps & time)
- You're looking for a used 'macbook' to buy. How will you find it? (steps & time)
- You've found a 'moto x'. You can use it for your project. How can you borrow it? (steps & time) What do you think happens next?
- You want to find out more about the item or its seller. How? (steps & time)
- How do you know if someone responded to the 'x' you posted earlier? Find such responses on the app. (steps & time)

- You want to accept a request. How? What do you think happens when you accept?
- You want to decline a request. How? What do you think happens when you decline a request?
- Note 2 points that could be 'improved' in the app.
- Note 2 points that the app does well.

Appendix D

Gantt Charts

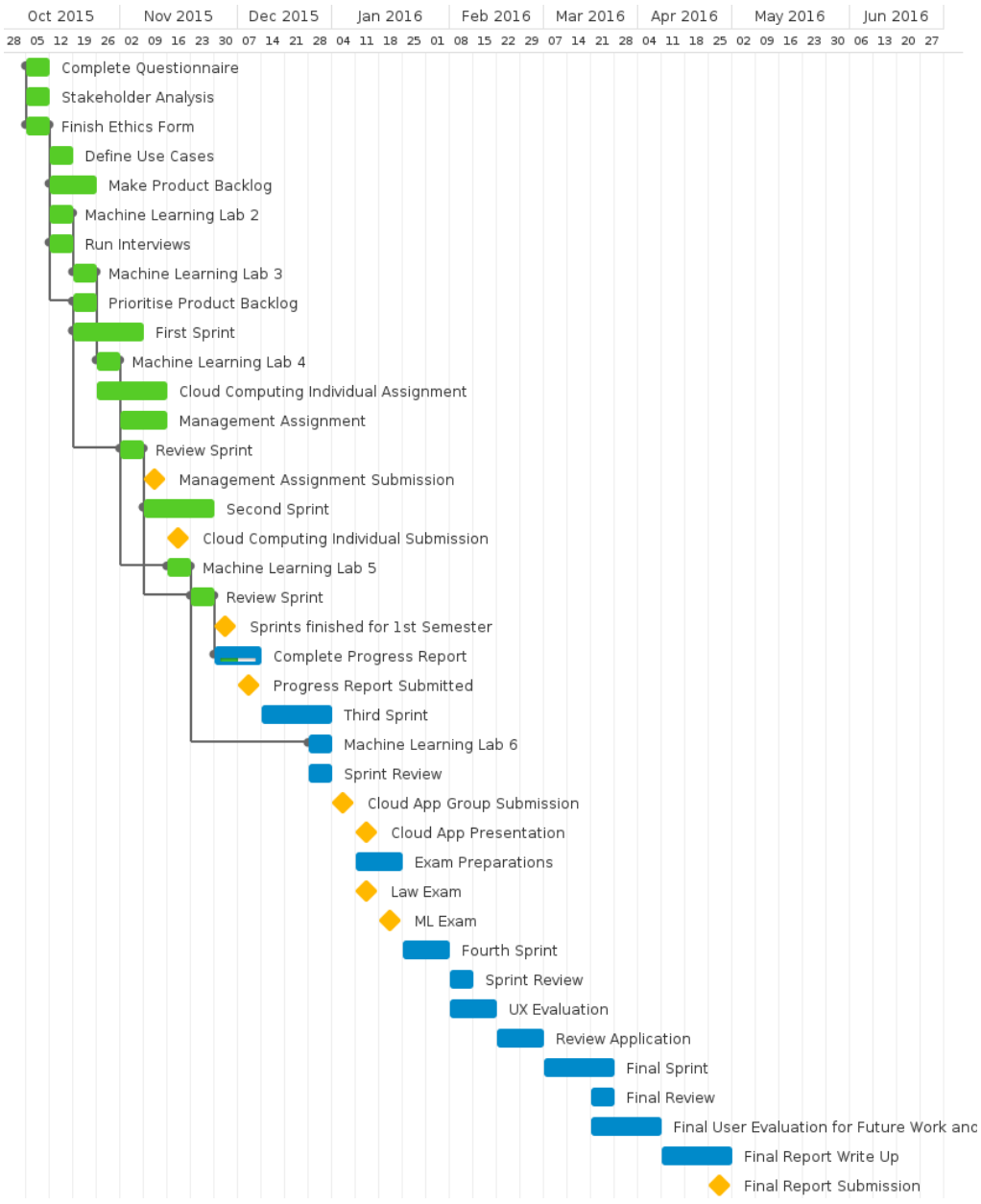


Figure D.1: Initial Gantt Chart

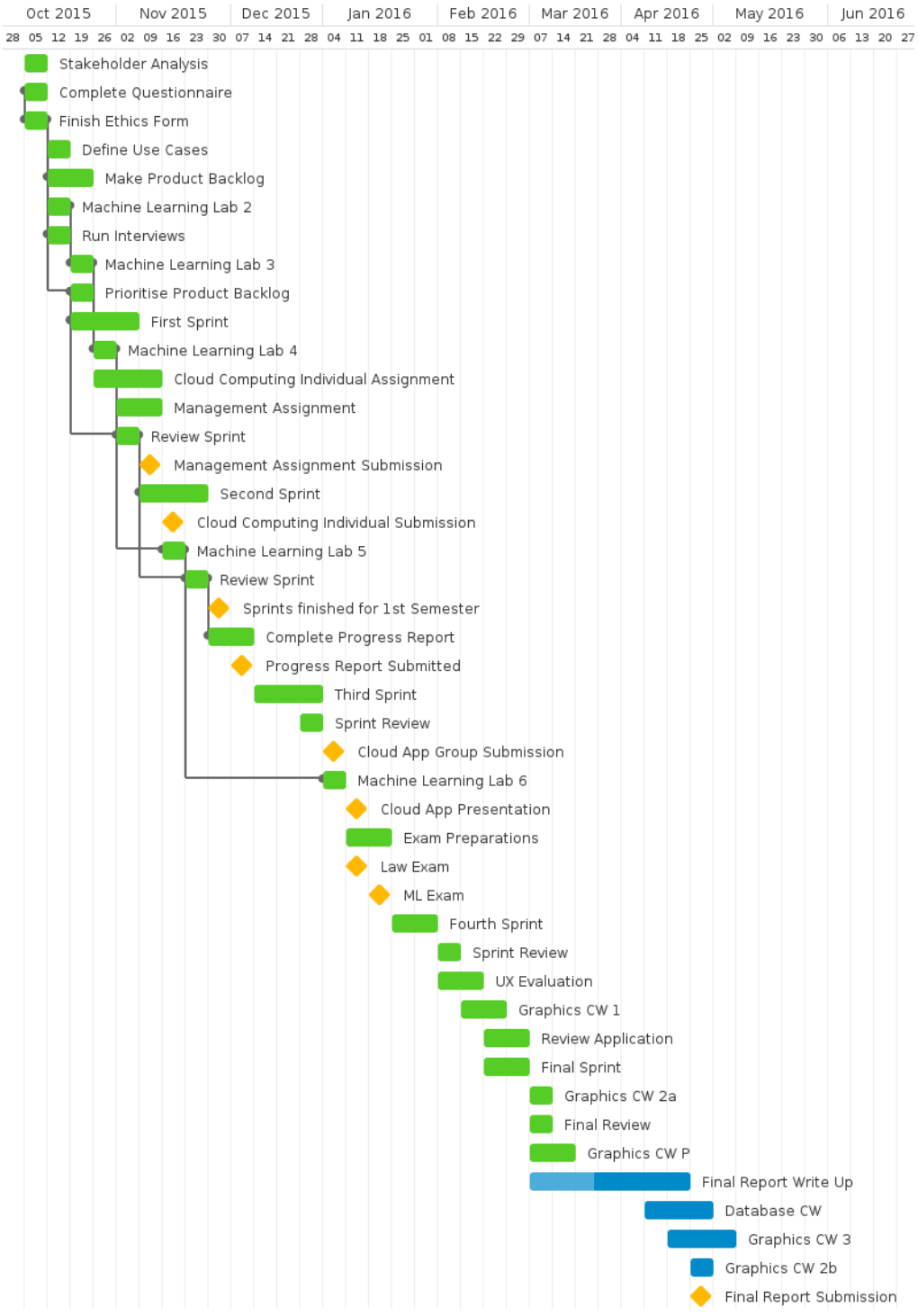


Figure D.2: Final Gantt Chart

Appendix E

UX Results

fixable easy (1-2 hrs) necessary but tough (>10 hrs) nice but cant

- You have just installed the app. You open it for the first time.
  - What do you see?
    - List of products **1**, **2**- ads, **4**, **5**
    - Expected the products in reverse chronological order **4**
    - Gray boxes. **3**
    - Details of products. **2**, **5**
    - Would rather not have a placeholder image for items without an image **1**, **4**
    - Pictures are a bit small. **5**
    - Circle buttons on bottom were getting in the way. **1**, **2**, **3**, **5**
    - Expected to have a category section **2**, **3**, **4**
    - GIF avatars of people **1**, **3**, **4**
    - Expects clicking on the user avatar on product card to bring up the user profile **1**
    - Expected owner's name to be on the product card along with their avatars **4**
    - Can see a nice chat option **1**
    - Borrow / buy is disabled means those options not available for the product **1**
    - Good that can't borrow own things **1**
    - Some items are closed seeing the red tag. **2**, **4**, **5**
    - Search bar for finding products. **2**, **3**
    - Wasn't sure what the blue person icon meant. **4**
    - Expected to have a tutorial to explain the basic usage **4**
  - What do you want to do now?
    - Search for something. **3**, **1**
    - Click on a product. **1**, **2**, **3**, **4**
    - Add a product. **1**, **5**

- – Navigate back to the home page. (steps & time)
    - * Everyone started looking about the app and made their way back to the home page. This task was trivial.
    - * Navigation stack messed up in some order. Couldn't reproduce how they did it. **1**
  - – Try to find your profile. How else could you find your profile? (steps & time)
    - * Everyone found the profiles before this question was asked. It was trivial.
    - * From the icon on home. **1**, **2**, **3**, **4**, **5**
    - * From the menu on the top left. **1**, **2**, **3**, **4**, **5**
    - * From one of own products. **3**
    - * Prefer the icon on home. **3**, **5**
    - * Prefer the menu option. **1**, **2**
    - * Cannot change avatar. **1**
    - * Liked the idea of rating. **1**, **4**, **5**
    - * Not sure if the comments are about the products, deals or person's own self. **1**, **2**, **4**
    - * Not sure what the ratings represent, i.e. expected to see granular ratings e.g. timeliness 4, care of product 3 etc **1**, **2**
    - * Good that can't rate oneself. **4**, **5**
    - * Rated someone. **3**, **5**
    - * Failed to rate someone. **1**

- You want to post this 'x' on the community to sell, but you also don't mind lending it. What do you do? (steps & time)
  - – Everyone followed the expected route, filled in the fields and clicked the green tick. Took less than two minutes.
  - – Click on (+) on home page. **1**, **2**, **3**, **4**, **5**
  - – Likes the use of toggle buttons for sell or lend options. **5**
  - – Prefer to have types in defined categories. **3**, **4**
  - – Relates types to tags. **4**
  - – Easy to add and see the scrollable picture. **1**, **5**
  - – Noticed the lending bug - where things are always put on for sale and lending regardless of optic
  - – Expected a confirmation **4**
  - – Expected to be taken to the summary page **5**
  - – Expected the product to be on top of the list **1**, **2**, **3**, **4**, **5**

- You're looking for a used 'macbook' to buy. How will you find it? (steps & time)

  - Trivial to search for a product on the home page (less than 30 seconds). But not trivial to get out of the search.
  - Search doesn't work on types or wrongly spelt names etc . **3**, **4**, **5**
  - Expected to see a message or suggestions when no results for a search were found **1**, **2**, **3**, **4**
  - Liked the cross to clear the search field. **3**, **4**, **5**

- You've found a 'moto x'. You can use it for your project. How can you borrow it? (steps & time) What do you think happens next?
  - Trivial to borrow an item. Less than 30 seconds.
  - Click on borrow button if available. **1**, **2**, **3**, **4**, **5**
  - If not available to borrow, contact the user via chat. **3**
  - Expected to have the options on the product page too. **2**, **5**
  - Buyer gets notified. **1**, **2**, **3**, **4**, **5**
  - Item was reserved. **2**, **3**, **4**
  - Get in touch with the buyer and exchange. Then complete. **1**, **3**, **4**
  - Complete immediately. **2**, **5**
  - Products should be considered closed if both parties complete it **4**

- You want to find out more about the item or its seller. How? (steps & time)
  - Trivial. Found within 30 seconds. Comments about own profile applies here too.
  - Went to product details page, then owner name. **1**, **2**, **3**, **4**, **5**
  - Expected the closed items on a product to be marked on a profile page like on the cards **4**, **5**

- How do you know if someone responded to the 'x' you posted earlier? Find such responses on the app. (steps & time)
  - The most difficult task here. Took quite a bit of looking around. More than 60 seconds.
  - Expected to get a notification (didn't always get sent a notification because of Google servers). **1**, **2**, **3**, **4**, **5**
  - Found on product details page. **1**, **2**, **4**, **5**, **3**
  - Expected to find separate place on profile for notifications about such things **1**, **2**, **3**, **4**, **5**
  - Expected to have a tab on the menu for notifications directly. **4**, **5**

- You want to accept a request. How? What do you think happens when you accept?

  - Trivial and took less than 30 seconds.
  - Click on accept button on an offer. **1**, **2**, **3**, **4**, **5**
  - The other side gets notified. **1**, **2**, **3**, **4**, **5**

- You want to decline a request. How? What do you think happens when you decline a request?
  - Trivial. Similar to above.

- Note 2 points that could be 'improved' in the app.
  - Add meaning to rating **1**
  - Improve the interaction with a failed search and make clearing a search better **1**
  - Make notifications clearer to find. **2**, **3**, **5**
  - Make scrolling better. **2**
  - Add categories for products. **3**
  - Separate tab in the menu for own products **5**
  - Add time tag to comments
  - Order of products in reverse chronological order.

- Note 2 points that the app does well.
  - Product card layout is clean with three buttons for obvious actions. **1**, **3**
  - Intuitive and clean interface. **2**, **4**, **5**
  - Animations, loading, fluidity adds life to the app. **4**
  - Adding pictures of products is easy, necessary and done well. **1**

Appendix F

GUI Testing

Table F.1, F.2, F.3, F.4 and F.5 contains a comprehensive list of the GUI tests that were performed by the end of development. It accounts for the expected behaviour, resultant behaviour and the possible bug in the system.

#	*Scene*	*Expected*	*Actual*	*Accept*	*Bug*
1	Open the app and stay on login page	The background to show up	OK, but takes some time	Y	–
		Labels on top, password covered, form filled	Labels misplaced on first login, rest OK	Y	Label stays on textfield without focus on first login
2	Login	Taken to product list	OK	Y	–
3	Products page examination	Reverse chronology of products	OK	Y	–
		Pull to refresh	OK	Y	–
		Only latest 20 at first	OK	Y	–
		Closed products at bottom	OK	Y	–
4	Product card examination	Owner's avatar, name, date	OK	Y	–
		user avatar goes to profile	OK	Y	–
		item name, description, picture, options	OK	Y	–
		Offer buttons disabled or enabled appropriately	OK	Y	–
		Default picture	OK	Y	–
		Appropriate closed tag	OK	Y	–

Table F.1: GUI Testing after Sprint 5a

#	*Scene*	*Expected*	*Actual*	*Accept*	*Bug*
5	Side menu examination	Profile takes to own profile	OK	Y	–
		Categories takes to categories	OK	Y	–
		Request goes to offers	OK	Y	–
		Logout returns to login	OK	Y	–
6	Add product button placement	Floats on bottom right	OK	Y	–
7	New Product examination	name required field	OK	Y	–
		Can't add product without name	OK	Y	–
		Description resizable	OK	Y	–
		Lend and sell toggle options	OK	Y	–
		Lend date from today	OK	Y	–
		Picture from gallery	OK	Y	–
		Picture from camera	OK	Y	–
		Multiple Category	OK	Y	–
		Clear category on cancel	Selection stays put	N	Scoped variable is not clearing

Table F.2: GUI Testing after Sprint 5b

#	*Scene*	*Expected*	*Actual*	*Accept*	*Bug*
7	New Product examination	Lend and Sell done correctly	Both selected always	N	Scoped variable not clearing on Lend
		Cancel closes modal	OK	Y	–
8	Search bar examination	Correctly aligned, button disabled	OK	Y	–
		Typing removes products	OK	Y	–
		Cross clears field	OK	Y	–
		Message for failed search	OK	Y	–
9	New Product	Added on top	OK	Y	Y
		confirmation given	OK	Y	–
9	Product page examination	Name, description, options, category	OK	Y	–
		Owner links to profile	OK	Y	–
		Offers iff for own item	OK	Y	–
		Picture slide viewable	OK	Y	–
		Navigable back	OK	Y	–
10	Loading examination	Animation shown	OK	Y	–

Table F.3: GUI Testing after Sprint 5c

#	*Scene*	*Expected*	*Actual*	*Accept*	*Bug*
11	Profile examination	Avatar, name displayed	OK	Y	–
		Rating count & average	OK	Y	–
		Cannot rate own	OK	Y	–
		Cannot rate 0	OK	Y	–
		Can rate others	OK	Y	–
		Comments in a list	OK	Y	–
		Own products list	OK	Y	–
		Products deletable if own	OK	Y	–
		Products not deletable if others'	OK	Y	–
		Products link to product page	OK	Y	–
		Navigable back	OK	Y	–
12	Requests examination	Recent requests first	OK	Y	–
		Older requests separate	OK	Y	–
		Request notification	Received most of the times	Y	GCM delivery unpredictable
		Options correctly enabled or disabled	OK	Y	–

Table F.4: GUI Testing after Sprint 5d

#	*Scene*	*Expected*	*Actual*	*Accept*	*Bug*
13	Requests examination	Product name links to product page	OK	Y	–
		Requester name and rating	OK	Y	–
		Request reason and date	OK	Y	–
		No request message	OK	Y	–
14	Categories examination	Category buttons displayed	OK	Y	–
		Search header displayed	OK	Y	–
		Failed search message	OK	Y	–
		Search results gives product cards	OK	Y	–

Table F.5: GUI Testing after Sprint 5e