# Final Report

Brandon Lee, Rutger Farry, Michael Lee

June 13, 2017

**Abstract**

The following report evaluates and recaps the purposes and project goals. The following will evaluate project progress, milestones, challenges, problems, proposed solutions, design, and remaining work. The following document will describe these aspects in the technical implementation level as well as in an overall higher level for clarity.

# Contents

# 1 Introduction

As a recap, our project, C7FIT is an iOS health app built for the local Portland gym, Club Seven Fitness. This project is built in collaboration with eBay's mobile team located in Portland as well. The roles of our clients included providing mentorship along with code reviews, assets, and resources for our needs. The application's purpose is to allow customers of a gym to easily be able to access their health information already on their iPhones as well as integrate that data into something that will help them work with trainers at Club Seven Fitness. Our goal for this application was to build something that would be an effective tool to do such that. Group members include Brandon Lee – Team Leader, Michael Lee – Developer, and Rutger Farry – Developer.

# 2 Original Requirements Document

**Requirements Document**

**Brandon Lee, Rutger Farry, Michael Lee**

CS 461
Fall 2016
4 November 2016

**Abstract**

C7FIT is a mobile application for iPhone developed as a senior software engineering project under the supervision of eBay Inc. This app is essentially a health and fitness application that will be utilized by members of Club Seven Fitness in Portland. Currently Club Seven gym clients have difficulty tracking their workouts, goals, and schedules in the digital world. C7FIT aims to integrate existing technologies from Club Sevens website as well as various iOS frameworks to provide users an accessible interface for interacting with their health and fitness goals on a mobile platform. The following document exercises the requirements of this project and formulates the plan of approach in developing this application in the next coming months. Such areas focused on include the project purpose, scope, perspective, interfaces, functions, constraints, and attributes. This document will additionally have elements of the higher level overview of the projects technical components.

# Contents

# 1 Introduction

## 1.1 Purpose

The purpose of this document is to deliver a clear specification of the requirements for the C7FIT app. The document will define a clear objective for the development of C7FIT including software features, product interfaces, design constraints, product functions, user characteristics, assumptions, and dependencies.

## 1.2 Scope

C7FIT is an iOS mobile application designed for Club Seven Fitness in Portland. The app allows for gym members to engage with Club Sevens content on the mobile platform. In addition to interacting with user inputted data from Firebase, our app will utilize iOSs HealthKit, which brings a plethora of user health data. Such application includes displaying the users daily level of activity. Furthermore, an activity tracking feature will allow for users to record running workouts and store such data locally. Finally, this software will extend to incorporate features from the traditional fitness scope including timers/stopwatches, contacting the local gym (Club Seven) through an embedded messaging service, daily workouts and video tutorials, and health statistics and records (eg. mile time, max bench press). Links to the eBay store for fitness equipment will also be natively built. The above is a listing of the essential features required for base viability. Additional features may be incorporated in future development. Stretch goals will be evaluated as development progresses.

## 1.3 Definitions

| Term | Definition |
|---|---|
| User | An individual who interacts with the mobile application |
| C7FIT | Our iOS app |
| Firebase | Google's Backend as a Service |
| Club Seven Fitness | Our client's client, a gym business based in Portland |
| HealthKit | iOS Health Framework |
| MapKit | iOS Map Framework |
| API | Interface to a piece of software exposed to the developer |

## 1.4 Overview

The following document includes two main sections. The first section illustrates an overall description of the product. Additionally, this part will delve into user characteristics and constraints.

The second half of this document goes into the specific requirements of the products including external interface requirements, functional requirements, performance requirements, logical database requirements, and system attributes. Appendices at the end will provide structure for the results and release plans.

# 2 Overall Description

This section will provide an overview of the application and how it interacts with its related components. This section will also detail the general functionality and how the it will be used by customers. Finally it will discuss the constraints and assumptions in the design of the application.

## 2.1 Product Perspective

The product will consist of the mobile iOS application for Club Seven. The application will be used to interact with the Firebase API, the eBay store, and draw data from the native iOS frameworks: HealthKit and MapKit.

### 2.1.1 System Interfaces

The system will be contained and limited to iPhone, running iOS 10, as the product is purely an iOS application. The Firebase API will provide the users external account data. It will provide users the ability to create a profile and input relevant health data. The application will also provide location information using MapKit, so the user knows their relative distance for classes and trainers.

The application will also use the MapKit and HealthKit independently of the Firebase API. HealthKit will be used to track the users general fitness levels, such as: steps per day, sleep, calories burned, etc. MapKit will be used to chart the GPS location of the user during their exercise activities.

As a stretch goal, the eBay API will be included as a way to monetize the application. Users will be able to see and purchase relevant products.

### 2.1.2 User Interfaces

The application will consist of 5 main topical screens: Home, Schedule, Store, Activity, and Profile. Each screen will maintain a consistent layout shown in the wireframes drawn up in our second meeting. In general, the screens will have a navigation bar along the bottom of the screen divided up into the 5 topical screens. Certain screens that require information from the Firebase API will show different information depending on whether the user is signed in or not. More specifically, each screen will have table rows that either contain information or bring the user to other more specific parts of the app. Once the rows have been tapped, the user will be directed to a new screen and have the option to go back with a ¡ button on the top bar.

### 2.1.3 Hardware Interfaces

An iPhone capable of running iOS 10, the iOS version were targeting, will be required to use this application. In addition, the phone will need to have the available memory space to download the app.

### 2.1.4 Software Interfaces

The Firebase API and eBay API will require developer accounts and accounts in order to function in our application. Their purpose has been discussed above: the Firebase API is key to the health data storage of the application, and the eBay API will be used for monetization. The application will be primarily in portrait mode.

### 2.1.5 Communication Interfaces

Wifi and cellular data will provide internet access to get the data from the Firebase API.

#### 2.1.6 Memory Constraints

The specific memory constraints of the application will depend on the device running the app. We are currently targeting a multitude of devices, iPhones (4s, 5, 6, 7) so the specific constraints will vary depending on the RAM and space available to each of these devices. However, the application will need to store some data locally like the activity map and personal statistics.

### 2.2 Design Constraints

#### 2.2.1 Operations

The user will be required to sign in to Firebase to access the full capabilities of this application.

#### 2.2.2 Site Adaption Requirements

The User Interface of this application will be in English. We will not provide other languages (as of initial release). The application will fit and work on current iPhones running iOS 10: including iPhone 4S to 7 and sizes 4,5,6, and 6 plus.

### 2.3 Product Functions

This mobile application will be a personalized app that connects the functionality of the eBay API to the C7FIT gyms customers. The functionality again can be split up into the 5 main portions Home, Schedule, Store, Activity, and Profile, with Home having the most varied functionality. Home will be the starting point for the user and provide general overview on their daily options. This includes a daily workout video from YouTube, and a daily motivational quote. The schedule screen links to the online schedule and contact information for the gym. The Store screen will display shopping information for curated fitness products from the eBay API. The Activity screen will display the users current daily activity, and their PR in various fitness tests. Additionally, this screen will have stopwatches and maps to assist their workouts. Finally, the Profile screen will contain their profile and personal information.

### 2.4 User Characteristics

The target audience for the product are the members of C7FIT gym. These gym members will only be able to access the services that are provided by the gym.

### 2.5 Constraints, Assumptions, and Dependencies

This application depends on having a Firebase account for the majority of the data. Additionally, it requires the user have internet connection as most of the data is pulled through external APIs. The key aspects of the application is constrained by the limitation of this API.

Additional caching of the Firebase data will not be supported outside of the suite of tools Firebase already provides, and as such the application will not display information when the user does not have access to the API be it through internet or lack of account. The daily videos and motivational quotes come from Firebase and will cycle through a static set of quotes and video links.

We assume that the mobile device has the capability to run our application at an acceptable level. If the phone does not have enough resources, then the application may fail to run as intended or at all.

The application is dependent on the Firebase API, native HealthKit and MapKit, for most of its functions.

# 3 Specific Requirements

## 3.1 External Interface Requirements

### 3.1.1 User Interfaces

C7FIT is a user-facing application, so naturally most of the design emphasis will be on the user interface and presenting information from external sources (such as the ebay API, GPS, external files etc) in a pleasing and easily digestible manner on the user interface. The main objective of the user interface is to help the user accomplish their most common tasks in as little time and effort as possible. Secondary goals are to represent Club Seven Fitnesss brand, while not being distracting, and to make tasteful use of animations to keep the user engaged, especially during operations that may require some time to complete such as API calls. (Designers often refer to these animations as microinteractions.)

Apples UIKit API provides minimally styled basic design elements, such as buttons, labels, sliders, text fields, and more. We intend to use these as the building blocks for our application wherever possible. In the MVP phase, unless a special element is absolutely necessary, we will just use the basic elements given to us by UIKit in their standard form.

Following completion of the MVP, we intend to add styling and animations to the most used elements, keeping minimalism, consistency, and reusability as a goal. Near the beginning of the project, we will design each screen using Sketch (a vector design program ideally suited for UIs). We will then implement this design down to the pixel in our app after the MVP stage.

The UI will have five tabs: Home, Store, Schedule, Activity, and Profile. The tabs will be displayed using UIKits UITabBar, and each underlying view will be managed by a single UITabBarViewController. Each tab is explained below:

1. Home Tab
2. Home will be a tableview with three cells. Some of these will be tappable, allowing users to view more detail than the snippet shown in the cell:
3. Todays workout video
4. This will display a Youtube video
5. The video URL will be retrieved from a static file either in Firebase or a GitHub repo
6. The video will be different every day
7. Selecting the video will open it in a landscape, full-screen view
8. Todays Motivation
9. This is a simple cell that just contains a motivational phrase
10. The phrase will be fetched from a static file in a GitHub repo

11. Store
12. The Store view will be a tableview of items available for sale from the Club Seven eBay store
13. Each cell will contain a picture of the item, its price, and name. Other metadata from eBay may be included
14. The store will use the eBay API to list products sold by C7Fit
15. Tapping a cell will show more information about the item, with the option to purchase it on eBay
16. Schedule Tab
17. The schedule will be a WKWebView (basically an embedded web browser) that opens the schedule webpage from Club Seven Fitness's website
18. As a stretch goal, we may hook into a Google Calendar or parse the HTML to display a calendar using native elements
19. Activity Tab
20. The activity tab screen will be a collection of tools, user activity, and health results.
21. Todays Activity
22. The top section of the Activity Tab shall have a title Today Activity.
23. The content in the today activity section will include steps and active time from the Health Kit SDK.
24. If the user does not have a profile on Health Kit SDK then a message will be displayed to the user . To get activity go to health app on your phone
25. Workout Tools
26. The section session shall have a title Workout Tools
27. There will be three tools in this section.
28. The first tool will be stop watch.
29. The stopwatch cell shall have text saying Stop Watch and shall have a right caret.
30. If the user taps the stopwatch cell a new screen shall be started.
31. The stopwatch screen shall have a title Stop Watch and a left caret.
32. If the left caret is selected the user will be returned to the activity tab.
33. The stopwatch screen shall have a timer.
34. The stopwatch screen shall have a Start button that starts the timer.
35. The stopwatch screen shall have a Stop button that stops the timer.
36. The time shall remain on the screen until the user taps the start button again.
37. The second tool will be countdown time.
38. The countdown cell shall have text saying Countdown Timer and shall have a right caret.

39. If the user taps the countdown cell a new screen shall be started.
40. The countdown screen shall have a title Countdown Timer and a left caret.
41. If the left caret is selected the user will be returned to the activity tab.
42. The countdown screen shall have a timer.
43. The countdown screen shall have an option to select a timer duration.
44. The countdown screen shall have a Start button that starts the timer.
45. The countdown screen shall have a Stop button that stops the timer.
46. The time shall remain on the screen until the user taps the start button again.
47. Map
48. The map screen shall have a title saying Map and a right caret.
49. The map shall use the Mapkit SDK.
50. The map shall show map with current location.
51. The map shall a Start Activity - have a start activity option.
52. The map shall a Track Activity - track activity to record the track and draw it on the map.
53. The map shall a Finish Activity - stop the activity.
54. The map shall a Activity Details - Time and Distance for activity.
55. The map shall a Save Activity - Save locally on device in core data with time as title.
56. The map shall a View an Activity - Stretch goal to remap an activity.
57. Test Results
58. The test results section shall list all current recorded results.
59. The test results section will have an edit option.
60. The edit option will launch a new screen that list the test results and allows the user to enter the correct data.
61. Test results will include: Mile Time, Number of Push Up in a Minute, Number of Situps in a Minute, Leg Press, Bench Press, Lat Pull.
62. Profile Tab
63. The more tab shall display the signed in users profile.
64. The elements that will be displayed include picture, name, and info from the HealthKit service.

### 3.1.2 Hardware Interfaces

C7FIT is an iOS 10 application, meaning it will run on any Apple iOS device capable of running iOS 10, or the next approximately 5 iOS operating systems produced in the future. We will ensure to not use any deprecated APIs, and since Apple usually does not deprecate APIs without several years advance notice, we can expect C7FIT to be compatible with future Apple devices for at least the next three years.

The app will be specifically designed with the iPhone 5s and up in mind, but will be compatible with all iPhones down to the iPhone 4s. We intend to use an adaptive design to stretch to make maximum use of the variety of displays on these devices.

### 3.1.3 Software Interfaces

The primary software the app will interface with is the iOS operating system, as well as the UI framework provided by iOS called UIKit, and a large collection of helper libraries contained in Apples Foundation framework. These libraries provide for everything from app instantiation to making network calls, to saving to disk, to drawing UI elements such as buttons and sliders.

We may also elect to bring in a few external 3rd party libraries to use instead of the Foundation / UIKit framework in areas that Apples API is known to be a little rough. We intend to minimize the use of third party frameworks / libraries as little as possible however. We will only use the leader in that field to reduce the likelihood of it being deprecated, and will wrap it in our own API to enable easy swapping of underlying APIs and prevent building dependencies in our code on potentially fickle open-source projects.

### 3.1.4 Communication Interfaces

C7FIT will make extensive use of Internet REST APIs, especially the Firebase API to provide functionality to the user. It will also make use of the eBay API if we meet our stretch goals to allow users to buy gym equipment online. Finally, the app will make use of the iPhones GPS unit to communicate with satellites and determine the users precise location during a workout. Luckily, this functionality is largely abstracted by the iOS Location API, and just involves a few lines of code.

The most challenging external communications will be those with the eBay API, as we will have to build a communication library with them based on their JSON REST APIs.

## 3.2 Classes and Objects/Functional Requirements

The main objective behind our program design is to reduce state to as few places as necessary. We plan on implementing our architecture with the principles of functional reactive programming in mind, so that developers can focus on building functionality of the app instead of worrying about bugs in their data flow.

### 3.2.1 DataStore

We will use the single-source-of-truth principle, along with dependency injection to reduce confusion about where data is coming from. All of the applications state will be kept in a central DataStore.

This will have fields containing data structures capable of holding all the data needed for all the apps views, organized hierarchically roughly around the purpose / source of the data. The DataStore will be initialized upon app instantiation and a reference to it will be dependency-injected into each View Controller. This way, every View Controller will share the same source of truth, ensuring consistency between views. This will also ease caching and local storage.

Information will be requested from the DataStore using functions exposed by the stores interface. Upon fetching the data through whatever means necessary (network call or GPS call, retrieval from disk, etc) the view controller will be notified with the requested data, allowing it to update its UI. The View Controller will not perform any modifications to the data, besides what is necessary to display it nicely. No methods that modify the display of the information should require asynchronous operations. The method for notifying the ViewController of state changes will be discussed in the next section.

#### 3.2.2 Responding to State Changes

There are several ways of responding to state changes, from callbacks to promises. We intend to use a functional reactive approach, in which the state of views is never explicitly changed, only described. Apple does not provide a functional reactive library, but there are two leading 3rd party libraries widely in use today: RxSwift and ReactiveCocoa. Both are similar, but ReactiveCocoa has deeper integration with Apples UIKit and Cocoa Touch frameworks. Receiving data from the API, the user changing screens, inputting text, pushing buttons, etc are state changes the app will respond to.

### 3.3 Performance Requirements

Modern iOS devices are relatively overpowered for this sort of application, so computational performance shouldnt be a worry. A much bigger priority than performant code should be readable code.

Places we should focus on performance are asynchronous calls, especially network calls. After the MVP, we should put priority on caching and prefetching data. If no cached data exists, we should display a useful animation that indicates progress.

### 3.4 Logical Database Requirements

We will be using Firebase for all database needs. Apple does provide a few set of APIs for caching on-device, but our source of truth will be coming from the Firebase API.

### 3.5 Software System Attributes

#### 3.5.1 Reliability

We will implement measures to ensure that the app functions well in all edge situations. Our programming methodologies described in the Classes and Objects section will ensure that developers are unable to compile the application without first accounting for all possible data states, including lack of GPS hardware, lack of network capability, disk failure, and more. Developers will be able to display helpful, user readable messages in all of these situations.

### 3.5.2 Availability

Most of the apps features will require internet access. However we dont intend to restrict access to the applications other features that dont require connectivity in the event that a network connection is lost. Features such as using the GPS or timer workout tracker, should still be functional. Additionally, the UI should display informative, but not disruptive information in disabled interface elements.

While we could cache network requests to the Firebase API while the device is offline, we will opt to just make the API attempt fail quickly and show an error to the user. Weve found that this helps prevent users from making unintentional actions to their account.

### 3.5.3 Security

The C7FIT app will contain some potentially sensitive health information, but not to the point where most users would want to add additional security measures such as TouchID or a passcode (common security measures on iOS).

Purchasing from eBay will be done in the eBay app/mobile website, which will handle all payment information and verification. C7FIT will just deep-link to the eBay app (or website if the app is not installed on the users device)

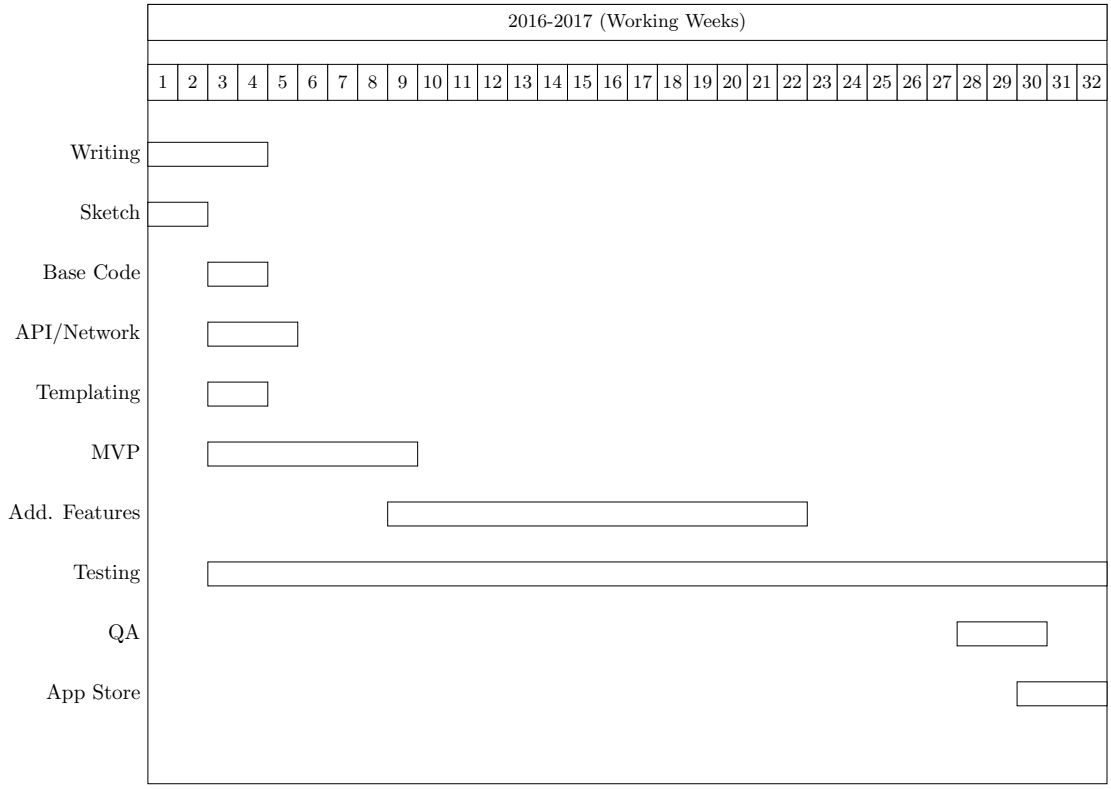
### 3.5.4 Maintainability

As the application will be used by Club Seven gym goers long after we complete development, there will need to be some form of maintenance in order to ensure C7FIT retains its quality and user base. In order to attain this, the app will be handed off to the eBay mobile team in Portland after publishing on the App Store. A small team of developers, whom we are currently working with will ensure the maintenance. Additionally, eBay Inc. will have the documentation we write this term for reference in the situation the app requires to be maintained long term.

### 3.5.5 Portability

The app is an iOS iPhone application. Because of the architecture of Apples development ecosystem, it should be relatively easy to port C7FIT to other Apple devices such as iPad. However for an Android application or any non-iOS device, there would be a substantial amount of more work to do before being able to publish. In terms of translation portability, we can use NSLocalizableString to load localized strings from a centralized source of truth upon our client's request.

## 3.6 Gantt Chart



# 4 Signed Participants

**Students**
Brandon Lee
Rutger Farry
Michael Lee

**Client**

# 3 Project Changes – Requirements Document

## Final Requirements Table

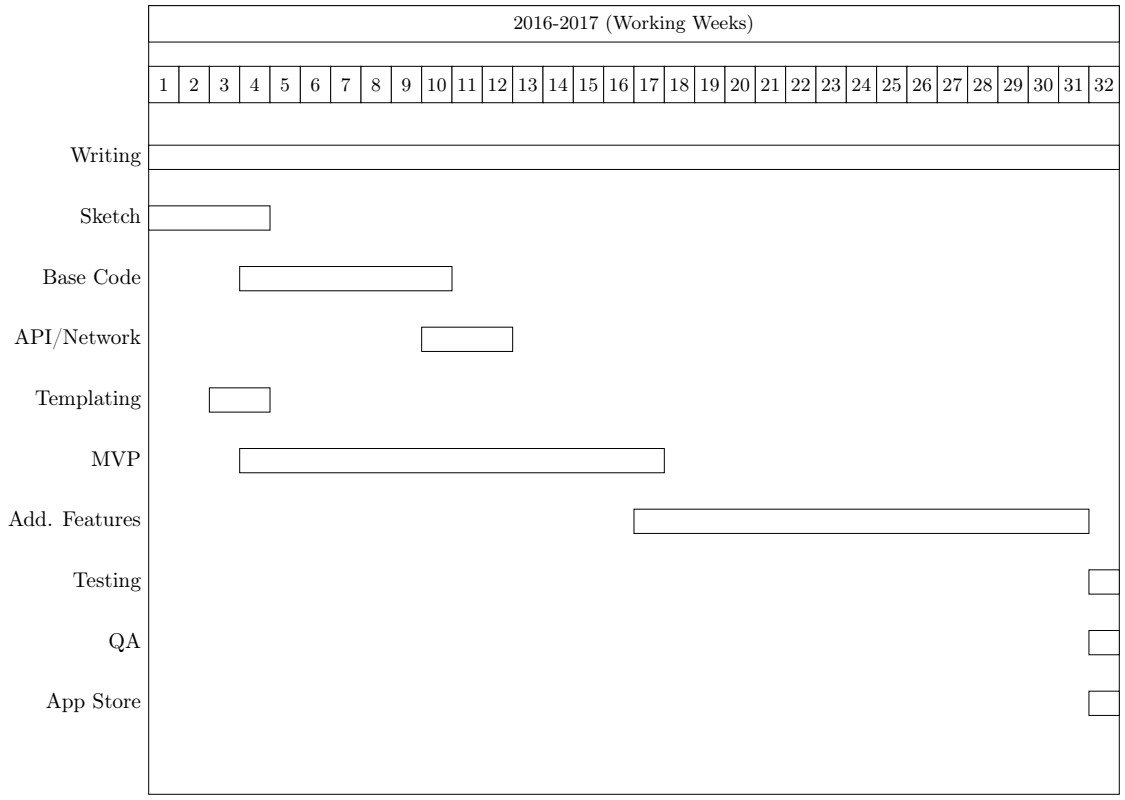
| Requirement | What happened to it | Comments |
|---|---|---|
| Home Tab | Implemented | Completed |
| Home will be a tableview with three cells. Some of these will be tappable, allowing users to view more detail than the snippet shown in the cell: | Implemented | Completed |
| Today's workout video | Implemented | Completed |
| This will display a Youtube video | Implemented | Completed |
| The video URL will be retrieved from a static file either in Firebase or a GitHub repo | Implemented | Ended up being stored in Firebase |
| The video will be different every day | Implemented | Completed |
| Selecting the video will open it in a landscape, full-screen view | Implemented | Completed |
| Today's Motivation | Implemented | Completed |
| This is a simple cell that just contains a motivational phrase | Implemented | Completed |
| The phrase will be fetched from a static file in a GitHub repo | Implemented | Storage has moved to Firebase |
| Store | Implemented | Completed |
| The Store view will be a tableview of items available for sale from the Club Seven eBay store | Implemented | Completed |
| Each cell will contain a picture of the item, its price, and name. Other metadata from eBay may be included | Implemented | Completed |
| The store will use the eBay API to list products sold by C7FIT | Implemented | Completed |
| Tapping a cell will show more information about the item, with the option to purchase it on eBay | Implemented | Completed |
| Schedule Tab | Implemented | Completed |
| The schedule will be a WKWebView (basically an embedded web browser) that opens the schedule webpage from Club Seven Fitness's website | Implemented | Completed |
| As a stretch goal, we may hook into a Google Calendar or parse the HTML to display a calendar using native elements | Attempted | Did not integrate in final product. |
| Activity Tab | Implemented | Completed |
| The activity tab screen will be a collection of tools, user activity, and health results. | Implemented | Completed |
| Today's Activity | Implemented | Completed |

| | | |
|---|---|---|
| The top section of the Activity Tab shall have a title Today Activity. | Implemented | Completed |
| The content in the today activity section will include steps and active time from the Health Kit SDK. | Implemented | Completed |
| If the user does not have a profile on Health Kit SDK then a message will be displayed to the user. “To get activity go to health app on your phone” | Implemented | Completed |
| Workout Tools | Implemented | Completed |
| The section session shall have a title “Workout Tools” | Implemented | Completed |
| There will be three tools in this section. | Implemented | Completed |
| The first tool will be stop watch. | Implemented | Completed |
| The stopwatch cell shall have text saying “Stop Watch” and shall have a right caret. | Implemented | Completed |
| If the user taps the stopwatch cell a new screen shall be started. | Implemented | Completed |
| The stopwatch screen shall have a title Stop Watch and a left caret. | Implemented | Completed |
| If the left caret is selected the user will be returned to the activity tab. | Implemented | Completed |
| The stopwatch screen shall have a timer. | Implemented | Completed |
| The stopwatch screen shall have a Start button that starts the timer. | Implemented | Completed |
| The stopwatch screen shall have a Stop button that stops the timer. | Implemented | Completed |
| The time shall remain on the screen until the user taps the start button again. | Implemented | Completed |
| The second tool will be countdown time. | Implemented | Completed |
| The countdown cell shall have text saying “Countdown Timer” and shall have a right caret. | Implemented | Completed |
| If the user taps the countdown cell a new screen shall be started. | Implemented | Completed |
| The countdown screen shall have a title Countdown Timer and a left caret. | Implemented | Completed |
| If the left caret is selected the user will be returned to the activity tab. | Implemented | Completed |
| The countdown screen shall have a timer. | Implemented | Completed |

| | | |
|---|---|---|
| The countdown screen shall have an option to select a timer duration. | Implemented | Completed |
| The countdown screen shall have a Start button that starts the timer. | Implemented | Completed |
| The countdown screen shall have a Stop button that stops the timer. | Implemented | Completed |
| The time shall remain on the screen until the user taps the start button again. | Implemented | Completed |
| Map | Implemented | Completed |
| The map screen shall have a title saying Map and a right caret. | Implemented | Completed |
| The map shall use the Mapkit SDK. | Implemented | Completed |
| The map shall show a map with current location. | Implemented | Completed |
| The map shall a Start Activity - have a start activity option. | Implemented | Completed |
| The map shall a Track Activity - track activity to record the track and draw it on the map. | Implemented | Completed |
| The map shall a Finish Activity - stop the activity. | Implemented | Completed |
| The map shall an Activity Details - Time and Distance for activity. | Implemented | Completed |
| The map shall a Save Activity - Save locally on device in core data with time as title. | Implemented | Completed |
| The map shall a View an Activity - Stretch goal to remap an activity. | Implemented | Completed |
| Test Results | Implemented | Completed |
| The test results section shall list all current recorded results. | Implemented | Completed |
| The test results section will have an edit option. | Implemented | Completed |
| The edit option will launch a new screen that lists the test results and allows the user to enter the correct data. | Implemented | Completed |
| Test results will include: Mile Time, Number of Push Up in a Minute, Number of Situps in a Minute, Leg Press, Bench Press, Lat Pull. | Implemented | Completed |
| Profile Tab | Implemented | Completed |
| The more tab shall display the signed in users profile. | Implemented | Completed |

| | | |
|---|---|---|
| The elements that will be displayed include picture, name, and info from the HealthKit service. | Implemented | Completed |

### Final Gantt Chart




# 4 Original Design Document

**Design Document**

**Brandon Lee, Rutger Farry, Michael Lee**

CS 461
Fall 2016
2 December 2016

**Abstract**

The purpose of this document is to elaborate the details of the design of our application, C7FIT. The following document will consist of five core parts discussing areas of design in the development of our application including an overview, definitions, conceptual models for software design descriptions, design description information content, and design viewpoints.

# Contents

# 1 Overview

## 1.1 Scope

This document will use a level of abstraction that is brief enough for an interested reader to absorb, but in-depth enough for a team of experienced programmers to implement. We cover all aspects of the application architecture, as well as some choices we've made to standardize our development practices and development environment.

## 1.2 Purpose

The intent of this paper is to document the architectural and design choices the C7Fit team has made in designing the C7FIT iOS application. As the app has not been implemented in code yet, this version of the document should not be considered as a guidebook to the final softeware implementation. The main purpose is to codify our intent as a reference when we begin implementing the software, and we intend to reflect these intentions in our software as closely as possible / reasonable. This document is subject to change as we begin implementing the application.

## 1.3 Intended Audience

C7FIT is an iOS application designed for the Club Seven Fitness Gym in Portland, Oregon. It is designed and developed by a team of seniors at Oregon State University as part of the Senior Capstone Project that is all members of the College of Engineering must undertake to graduate. The group's instructor is D. Kevin McGrath, and their corporate sponsor is eBay Inc., who has donated staff and other resources to oversee the project's completion.

## 1.4 Conformance

The C7FIT application shall conform to the requirement specifications as denoted by sections 4 and 5 in this design document. Please refer to the referenced sections for further details on application requirements and conformance.

# 2 Definitions

| Term | Definition |
|---|---|
| User | An individual who interacts with the mobile application |
| C7FIT | Our iOS app |
| MindBody | Health/Wellness Service |
| Club Seven Fitness | Our client's client, a gym business based in Portland |
| HealthKit | iOS Health Framework |
| MapKit | iOS Map Framework |
| EventKit | iOS Calendar Framework |
| API | Interface to a piece of software exposed to the developer |
| iOS | The Apple iPhone Operating System |
| HealthKit | The official iOS health framework |
| MapKit | The official iOS map framework |
| CoreData | The official iOS local data storage framework |
| Storyboard | A visual representation of a user interface of an app view |
| Data Store | The centralized source of truth for our data |
| Xib | XML Interface Builder for iOS application views |
| View Controller | A manager connecting model and view |
| Swift | Apple's new general purpose programming language |
| Objective C | Apple's old general purpose programming language |

# 3 Conceptual Model for Software Design Descriptions

## 3.1 Software Design in Context

The project will be designed in a modular fashion. In order to achieve such a design objective, we will be utilizing a object oriented application structure. Because this application will comprise of multiple views utilizing data and inputs from multiple sources, having a modular abstraction context for our application design is essential for success. With such a level of modularity through an object oriented design, new features and additional views can be added with moderate ease.

## 3.2 Software Design Descriptions within the Life Cycle

### 3.2.1 Influences on SDD Preparation

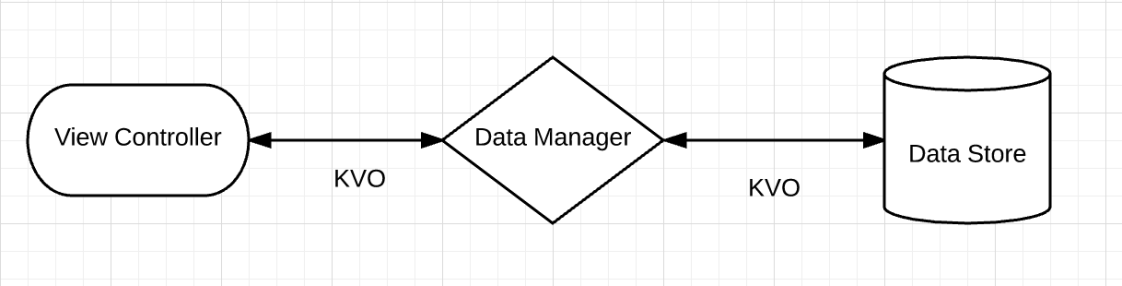
The design document and application is prepared through considerations between the interests of the capstone course as well as the clients both at eBay Inc. and Club Seven Fitness. Their interests provide a catalyst for direction in this design document.

### 3.2.2 Influences on Software Life Cycle Products

This project is a mobile application which both draws and sends data between client and server. A hefty majority of the features shall be integrated with the existing API on the server. Thus as a result, a significant element of the application will be developing the interfaces in interacting between client and the server. These modules will be critical in the success of our application. In order to ensure the quality of this aspect, we will look heavily on the feedback provided by our client for advice in developing such a significant piece of interface. Once the foundation for interface between client side application and server side application is hammered out, we can then focus on the other parts of the application.

### 3.2.3 Design Verification and Design Role in Validation

Because C7FIT is a mobile application which relies heavily on the server side MindBody API, there is definite significance on the integrity of the interface between client and server. Because the MindBody API is responsible for providing much of the fitness and course schedule data, the majority of the client side module's responsibilities will be working with the networking interface with the server side to display and interact with the returned data. Parsing and processing such data to elegantly display to the user will be a primary task as well. Additionally, another main aspect of the client side modules will be to receive user input in forms of registering for courses and modifying server side data. Such tasks will be done in correlation with the server side modules. Below is a diagram of such a potential design for abstraction between the movement of data in our application.




# 4 Design Description Information Context

## 4.1 Introduction

This section of the design document will provide information about the specific elements of design incorporated into the development of this application. Such areas include SDD identification, design stakeholders and their concerns, design views, design viewpoints, design elements, design overlays, design rationale, and design languages.

## 4.2 SSD Identification

This report is the initial SDD for the iOS Application in Swift project (C7FIT). The initial start date for this assignment was September 22, 2016. From there on out, we've been documenting design and specifications for our project C7FIT. As of currently, the design doc is not quite finalized yet The client of this project is eBay Inc., which is comprised of Luther Boorn, Splons Splonskowski, Wyatt Webb. The main objective of this design document is to illustrate the scope of the project C7FIT. The following sections will illustrate the concepts through visual diagrams.

## 4.3 Design Stakeholders and Their Concerns

The current stakeholders of the project are the clients at eBay Inc. including Luther Boorn, Splons Splonskowski, and Wyatt Webb. They are all developers and managers at eBay Inc. The main scope of the document is to layout the design of the C7FIT project. The main concerns in regards to this project is finishing the application on time with the current specifications. This document will utilize text descriptions as well as diagrams to portray the viewpoints of the project.

### 4.4 Design Views

This project is implemented through a MVC structure, which will allow for modular application development. Thus, adding new features and capabilities will be easily achievable. Because of the object oriented structure of the application, project development updates will be easily accomplished. Design views of the application will portray the user's schedule as well as user fitness data. The relationships between the varying MVC classes are easily defined. Dependencies render the the current facing challenges and potential future issues as well. Dynamic data displays the change of state between the various views in the application.

### 4.5 Design Viewpoints

Design viewpoints clearly display the expectations for interaction from the user in the application's various views. Dependency viewpoints display the various requirements of the system. Logical viewpoints portray the logical structure of the application. Interaction viewpoints render the interactions between the various classes of the application. These viewpoints are relevant to this project as the various states of the application encapsulate these fundamental components.

### 4.6 Design Rationale

Choices made for the design are decided accordingly with the specifications of the application's requirements. As the requirements shift and the client's evaluate the state of the application's development, the rational for the design may change as well. Each of the major design choices are commented on with relation to the rational behind each choice. This is done in order to allow for future developers to easily maintain the code base accordingly to the choices in design.

### 4.7 Design Languages

UML diagrams, Sketch designs, and user stories are utilized in order to layout the design of the application.

## 5 Design Viewpoints

### 5.1 Introduction

This section of the document will span various viewpoints for the application including context, composition, logical, dependency, information, pattern use, interface, structure, interaction, state dynamics, and resource viewpoints.

### 5.2 Developing the User Interface

To create the User interface, we will be coding the interface elements rather than using the drag and drop tools that come with Storyboards or Xibs. The custom code we have will be more complicated than the Storyboards or Xibs, but we can create custom stylings that are not possible with the other methods. The standard view layouts can be reused and repeated like a template. The code will make it hard to visualize our layout, however we will use the Storyboards to flesh out a prototype based off of the Sketches that weve created.

#### 5.2.1 Pattern Viewpoint

From a patterning viewpoint, using code to develop our user interface will allow us to maintain a consistent pattern. This is because each viewing screen is self contained and can be replicated

multiple times for different views. This replication gives us a consistent pattern between views. It makes it easier for the developer, as we dont have to recreate the designs, and it makes the views consistent for the user. This makes the user interface more intuitive for the user.

#### 5.2.2 Context/User Viewpoint

From the users viewpoint the popups created with code will be customized for each specific function. This will result in a better presentation, which will be more aesthetically pleasing for the app, and provide added functionality as Xib templates will not have exactly what were looking for. This means that we will be creating custom popups and views as shown by our Sketch Designs.

#### 5.2.3 Dependency Viewpoint

The User interface will depend on the back end code of the application to perform actual functions. Likewise, the backend will depend on the user interface being developed for certain parts of the application to function. Controller elements that interact with the view, like selecting date/time or viewing trainer data will require that the view is created.

### 5.3 Storing General User Data

To store our general user data, we will be using CoreData to store local data on our application without the need of a backend database. CoreData is focused on storing objects with attributes rather than the more typical SQL tables. This allows it to have more flexibility in data storage at the cost of more overhead. We will be using CoreData to store data locally rather than having a server database, because the data stored will be relatively simple and will not require the large overhead of a remote server.

#### 5.3.1 Dependency Viewpoint

All parts of the application that require the user to store data will be reliant on the local CoreData database to be set up. CoreData is natively supported by iOS so it will not be dependent on an external framework. The applications daily youtube videos and quotes will not require CoreData because they will be stored and pulled from a GitHub page. The Users personal record data, however, will require CoreData. They will store their PRs, such as push ups and mile times, locally on their device using CoreData. Additionally, the map tracking portion of the app will depend on a CoreData object to maintain information about the users activity path and time.

#### 5.3.2 Information Viewpoint

Storing the data locally on the app rather than on a remote server allows the data to be entered in and accessed at any time without an internet connection. Much of the app requires the MindBody api to function, this is an unavoidable limitation as we dont control the api; however, by storing user data locally the user will always have access to their information. This means that the information will persist whether or not there is an internet connection.

#### 5.3.3 User Viewpoint

From the users viewpoint, storing the data locally benefits them because they will always be able to access the data. Like a note-taking application they will not be reliant on internet, and so they will be able to plan workouts on their own. Local storage allows the user to map and enter in their workout information conveniently when they dont have internet. A use case would be when the user goes on runs in rural areas without wifi connectivity. This storage will still allow them to track their run information.

### 5.3.4 Resource Viewpoint

From a resource viewpoint, local storage puts a larger limitation on the application itself. If all of the users data is stored on the application they are limited to the size and power of their device. If they run out of space, then the application will be unable to store additional information. If we had used a remote server the user would not be dependent on the power of their device; However, we are limited financially to set up a server and consequently chose to be restricted by the devices limitations. In normal conditions, the user will not be storing an excess amount of data that will overload the devices capabilities.

## 5.4 Choosing an IDE

The IDE we will be using is XCode. XCode has all the built in support that we need for development using Swift 3, because the application is natively provided by Apple for development on iOS projects. This IDE is created and supported by Apple, and so it has extensive documentation. Additionally, it is the most popular IDE for the development of iOS applications so it has strong community support. It has a large amount of built in tools that allow the developer to simulate and debug their application without the need of the iOS device, although we will be testing both in the simulator and on our respective iOS devices (iphone 5,6 and 7).

### 5.4.1 Dependency Viewpoint

The XCode IDE has the frameworks and technologies that we will need to develop the application already built in. As a design focus we have chosen a majority of native technologies like Swift 3 and CoreData, and so the native IDE will naturally support all of these choices. There is a device simulator built into the IDE so we can test the application on devices that we dont own. On other IDEs we would have to depend on XCodes tools for this functionality as Apple limits much of their tools to their own software.

### 5.4.2 Resource Viewpoint

From a resource viewpoint, using the XCode IDE costs the team nothing. It provides some of the best functionality available for an iOS IDE, and is offered for free on Apples computers. The functionality from the IDE gives the team powerful tools without any resource expenditure.

## 5.5 Parsing Mindbody Data

Most of our app is built around the MindBody service. MindBody is a software suite that helps gyms, studios, and other wellness centers run their business. It manages responsibilities such as handling membership / service payments, scheduling appointments, tracking customer metrics, and more. Most of this functionality is accessible via a SOAP XML API, a type of web-based protocol to communicate with an external service.

Since SOAP XML APIs are considered obsolete, and have been mostly replaced with JSON-based APIs, there is little innovation in XML parsers, especially on newer platforms such as iOS. However, in the early 2000s, SOAP was the norm and there are therefore several performant and stable options available for our use. The only issue is that none of these parsers were written with a Swift API, so we will therefore need to write a wrapper around the libraries we choose. We have decided to use Apple's XMLParser library for this purpose

### 5.5.1 Logical Viewpoint

Our wrapper will consist of a functional interface that will simply convert NSData XML objects received from the API into Swift structures or classes. The functions we use to do this will know the structure of the Mindbody API responses and will use the XMLParser library to parse them.

### 5.5.2 Dependency Viewpoint

Our parsing library for Mindbody will be a wrapper for Apple's XMLParser library. This library is a 1st party dependency, as it is built into Apple's Foundation framework available to every iOS app.

### 5.5.3 Algorithm Viewpoint

Some data structures received from Mindbody may be nested inside other data structures. To parse a parent data structure with children data structures, we will use recursive programming - parsing the children before moving up into the parent data structure. This decision is partially guided by Swift's strict type system, which would require a parent structure's children to be of a Swift object or struct type, instead of a raw data type.

### 5.5.4 Patterns Viewpoint

When creating these functions, we may find it useful to create convenience parsers to parse different kinds of input. Instead of rewriting the logic of every function just to take different type parameters, we should use composition - creating a base function that all convenience functions will use underneath. Therefore, convenience parsing functions will only have to perform a type conversion for their parameter into the parameter required by the base function. The base function's input parameter should either use the most primitive type available, such as a string, or the input parameter required by the XMLParser library.

## 5.6 Handling Touch Events

Touch is the primary source of user interaction with iOS applications and handling touch events is likely the most important aspect of any app. In our opinion, the most important attributes of good touch handling are:
1. Correctly and swiftly determining intentional touch events, while filtering noise.
2. Correctly placing touch events on the screen. According to interviews with high-level Apple engineers[**touchpara1**], this is not just a case of being accurate where users actually touch the screen is a different location than where they expect to see the touch registered.
3. Providing a simple API that hides superfluous details.

Apples UIKit framework contains several UI elements (buttons, sliders, labels, etc), and most of these have gesture/touch-recognition built in. For elements that dont have built in gesture recognition, UIKit provides a UIGestureRecognizer class which can be easily overlaid on an elements view area. We have decided to use UIKit's elements exclusively unless it is absolutely necessary to build a custom component or to add touch handling to a existing UIKit component that lacks it.

### 5.6.1 Composition Viewpoint

Since touch is such a large part of iOS, Apple has the design patterns pretty hammered out. If we want to observe an element designed in a XIB file (graphical user interface designer), we can drag a touch handler from the the XIB file to our code. This touch handler is a function that is called whenever the element is touched. We can then filter out the undesired touch events using a switch

statement to appropriately respond to taps, drags, zooms, and more.

In the event the element we want to observe is dynamically added to the screen or otherwise not in the XIB file, it is possible to handle touches by calling:

`addTarget(_:action:for:)`

this method is called on a child of the UIResponder class and allows us to specify a target class and method to be called for a specified gesture.

## 5.7 Storing/Modifying Application State

Apple's frameworks provide a lot of functionality, but do little to tell developers how to manage their application's state. They are opinionated however, and they provide some components such as storyboards that make creating a simple app easy, but can become convoluted as an app grows. Since our app is relatively complicated, we want to remember to think big when designing how our application manages its state.

We have decided to opt for using a central data store in our application that is passed to each view controller via dependency injection. This solves a host of problems, including reducing data fragmentation, easing persistence, reducing confusion, and more. It also creates some cool benefits, such as making it easy to track and roll back app state, making it a lot easier to debug issues.

### 5.7.1 State dynamics

While it would be unreasonably difficult to design our entire state machine before implementing the app, we will try to outline the main components and behavior of the data store below.

When the app is launched for the first time, it will have no data from the Mindbody service to display. While subsequent launches will likely have some cached data to display, it is useful to first consider the initial state. Much of the information displayed will be in arrays of Swift classes or structures. To force the views to consider the null state, we will code these arrays as optionals (can be null). Once data is received from the API, the arrays will be initialized and filled with data. In the case of no data being received, the array could either stay null or be initialized with zero elements. We will likely utilize both methods, along with the use of exceptions to display useful information to the user at all times.

The datastore is mostly intended to handle the app's data state, but it would be interesting to also put view state inside either the central datastore or a additional datastore. This would allow us to easily fully restore the app's previous state. Likely this would just entail creating an enumeration that reflects the app's navigation stack of view controllers.

## 5.8 General

### 5.8.1 Context Viewpoint

The primary users of the C7FIT application will be members of the Club Seven Gym in Portland, Oregon. Club Seven, run by owner Brian Warner, is a small, personal training-focused gym that works closely with its members to achieve their individual goals. The gym also works with atheletes, especially student atheletes, and has hired an ex-OSU football player Sylvester Green to coach clients.
As a small gym, the owner likely doesn't have a full-time staff to answer phones, manage payments, and schedule training sessions. Luckily, these processes can be handled by the MINDBODY platform,

which the gym uses to manage these tasks. While MINDBODY provides a customizable client-facing web portal for gyms, it doesn't have any client-facing applications for scheduling, paying, viewing gym info, etc. This is mostly due to the fact that it is logistically difficult for a company such as MINDBODY to release and maintain individual apps for each of its partner gyms. The Apple App Store makes it tedious to release app updates, and keeping every version of a MINDBODY app up to date would be difficult, if not outright banned by Apple.
If a specific gym wants a mobile app, they need to build one themselves, which we are doing for Club Seven.

### 5.8.2 Patterns Use Viewpoint

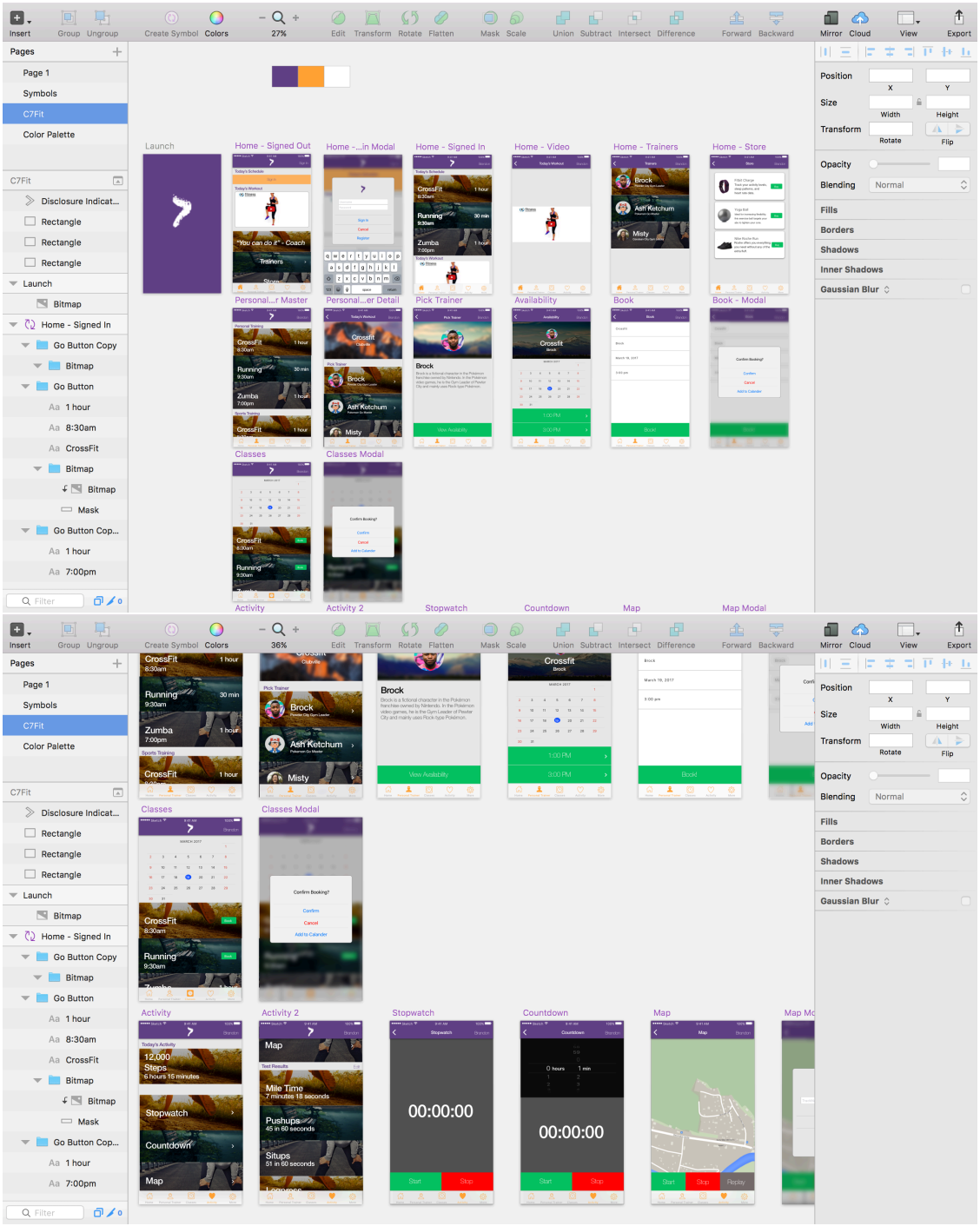
For this project, we will enforce a strict pattern for our data flow to ensure we always know where our data is stored. We will use the Dependency Injection pattern to enforce this, creating a single datastore object during our app's instantiation. As new views are presented, the root view controller should inject a reference to this datastore object into the new presenting view controller.

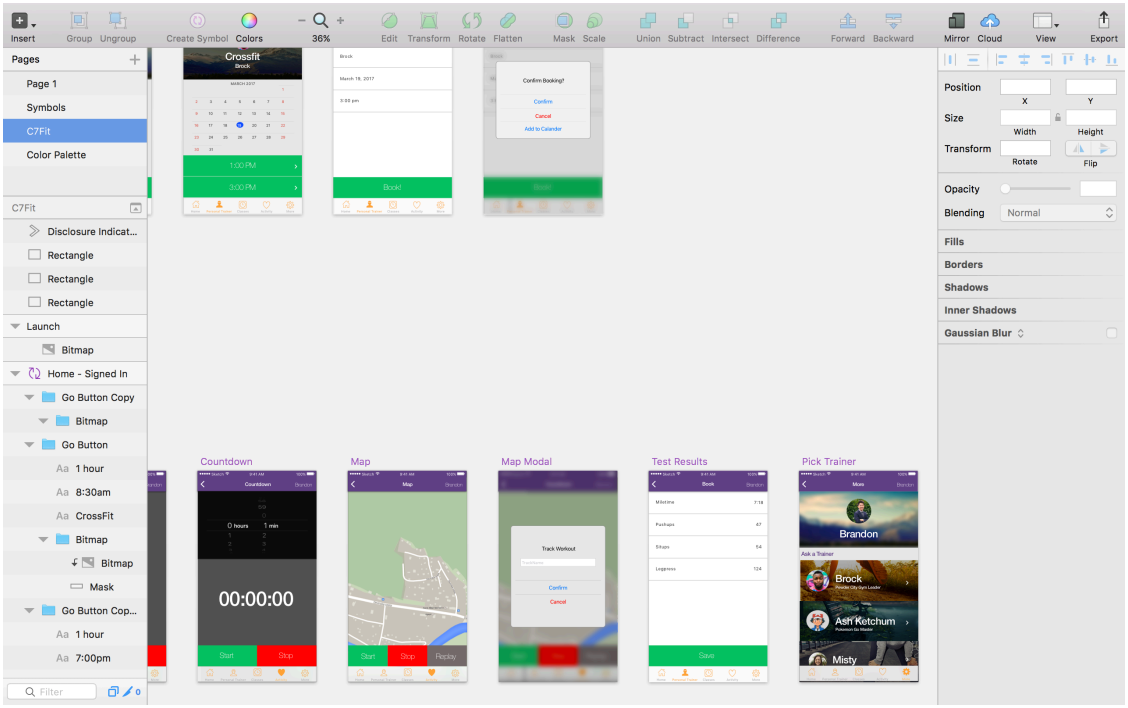
There are several advantages to this approach. First, since view controllers will require the datastore in their initializer, their datasource will be explicitly defined and it will be impossible to instantiate a view withiout it having valid data to display. Second, since all views share the same data source, they will not need to needlessly copy data for themselves. Thirdly, since all views will share the same data source passed in from their parent, there will not be inter-dependencies between views. This will result a tree-like data flow instead of the web-like data flow that develops in many large and poorly-designed applications. Finally, the app's full state will be shared among all views in the app, eliminating fragmentation as long as views are kept up-to-date with the datastore. This leads us to our second major design pattern, reactive programming.

There is another pattern that goes along well with this central datastore approach called functional reactive programming. Functional reactive programming consists of composing a series of pure functions that watches an underlying data source. When the data source changes, the functions run and transform the underlying data from one form to another. We intend to use reactive programming to ensure our views stay up to date with our app's state. This will be accomplished by binding view elements to the datasource, with functions in-between to convert the data source's data into a presentable view object. Whenever our data changes, the parts of the view that should be updated should then "react" without requiring additional code. This helps eliminate a lot of mistakes with writing "update" functions. Often these functions don't update the correct parts of view, update too much of the view, or cause unexpected side-effects.

### 5.8.3 Interface Viewpoint

Client side user interface is important to any application. Rendering a consistent and elegant design for the user to interface with is vital to the user experience. Below are a few Sketch designs for the various potential views in C7FIT.


Launch
Home - Signed Out
Home -...in Modal
Home - Signed In
Home - Video
Home - Trainers
Home - Store
Personal...r Master
Personal...er Detail
Pick Trainer
Availability
Book
Book - Modal
Classes
Classes Modal
Activity
Activity 2
Stopwatch
Countdown
Map
Map Modal
Pages
Page 1
Symbols
C7Fit
Color Palette
Position
Size
Transform
Opacity
Blending
Fills
Borders
Shadows
Inner Shadows
Gaussian Blur




# 6 Conclusion

The designs in this document were evaluated on different criteria, some of the more important criteria being native availability, functionality, and visual-ability. We wanted designs to be simple to understand, and at the same time to portray a clear message on what our intentions are. Many of our designs were driven for iOS.

# 7 Signed Participants

**Students**
Brandon Lee
Rutger Farry
Michael Lee

Sponsor

Luther Boorn

Luthe Boorn

12 - 2 - 2016

### 4.1 Design Document Changes

The biggest change we encountered while working on our senior project was the removal of MINDBODY functionality when working on our app. The ability to schedule and pay for appointments using the MINDBODY API was going to be a significant portion of the app, but we determined in December that it was going to be unfeasibly expensive to pay for the MINDBODY service.

Luckily for our group, the design direction that our client wanted to take was already accounted for in our plans. Their decision was to replace the scrapped MINDBODY functionality with our stretch goal of implementing a fitness shop using the eBay API. Since this was a stretch goal for our project, it was already noted in our design documents and we had already drawn up wireframes for the proposed eBay fitness shop. Therefore, switching directions in this way was relatively painless and only entailed removing MINDBODY from our design documents and rearranging the tab bar to include a button for the eBay fitness shop.

## 5 Original Tech Review

**Technology Review and Implementation Plan**

**Brandon Lee, Rutger Farry, Michael Lee**

CS 461
Fall 2016
14 November 2016

**Abstract**

The purpose of this document is to elaborate on the technologies we will be utilizing to develop our application, C7FIT. The following document will break down nine different technologies of which we will discuss our options, goals in design, criteria, discussion, and final decisions.Of the various areas of technology, the following document will look into choosing an IDE, capturing and sending MindBody API data, parsing MindBody API data, storing general user data and user login, iOS application language, handling touch events, storing/modifying application state, responding to changes in the application state, and developing the user interface.

# Contents

# 1 Introduction

By writing this document, we hope to clear up any confusion in our group about how we will implement our application and ensure that in the event of our untimely passing, our successors are able to complete our work. While the functionality provided by our app is not novel, it is still large and complex and needs to be well-designed to stay understandable as it grows. Rather than reinventing the wheel, we choose to stand on the head of giants, taking advantage of libraries provided by Apple and others where possible. However, as stated by the phsychonaut Terrance McKenna, "We are so much the victims of abstraction that with the Earth in flames we can barely rouse ourselves to wander across the room and look at the thermostat." We hope to harness abstraction, but not become slave to it to the point we cannot see the flames.

# 2 Choosing an IDE

## 2.1 Options

To create our application we will need a set of development tools or an Integrated Development Environment rather than a simple text editor. When developing in iOS there are only a few options available. The following IDE's, Xcode, Nuclide, and AppCode are all viable IDE's to develop mobile iOS application.

Xcode is the native iOS IDE, created by Apple, for developing software for iOS, macOS, and WatchOS. It supports programming for C, Objective-C, Swift, and many others. In addition to being a text editor, it has Interface builders, an iOS simulator for local debugging, a Version Editor, etc. It has extensive documentation and a lot of support from Apple and the community as it's the most popular iOS IDE.[15]

Nuclide is an open source IDE built on top of the Atom by Facebook for a unified web and mobile development. It has the typical built in debugger and supports the main iOS languages of React Native, Objective-C, and Swift. For running Applications the easiest way to build and run is still through using Xcode; however, there is also a build system called Buck, developed by Facebook, which gives the developer additional tools to build, run, and debug iOS apps.[4]

AppCode is a Smart IDE for iOS/macOS development created by JetBrains. AppCode supplants the development tools of Xcode, so it can't be used on it's own; however, JetBrains provides a multitude of developer tools so AppCode may provide functionality that tops Xcode.[1]

## 2.2 Goals

The main goal in choosing an IDE is to make all stages of the development process easier from coding to debugging. The IDE we choose needs to have Swift support, as that's the language our application will be written in. The IDE should be intuitive to use, but also have enough tools to make the job of coding easier.

## 2.3 Criteria

Based on the goals above, our chosen IDEs will be evaluated on the following categories: functionality, ease of use, and swift support. The IDE needs to be aid us in development not hinder us. It needs to be easy to use as we don't want to spend our entire time learning how to use the software.

Finally, it needs to support the language that we're programming in.

| Options | Functionality | Swift Support | Ease of Use |
|---|---|---|---|
| XCode | High | High | Moderate |
| AppCode | Moderate | Moderate | Moderate |
| Nuclide | Low | Moderate | Moderate |

### 2.4 Discussion

The benefits of Nuclide and AppCode come from their added text editing features. For example, AppCode has exceptional customization compared to Xcode, from appearance to code-styling. For functionality, AppCode has an excellent step through debugger and code completion. Xcode on the other hand, has been known to lack refactoring options and has an inaccurate debugger. Nuclide does support swift, but it has limited built-in support for the language. Both Nuclide and AppCode still rely on Xcode for a lot of their functionality. Nuclides documentation states that the best way to build and run applications is by using Xcode. AppCode is a lot more independent, but it fails to match Xcode's other functionality: it does not support storyboards, build settings, or other more advanced functionality found in Xcode. It will always lag behind Xcode as Apple has complete ownership over their products. Overall Xcode and AppCode have different functional benefits, but Xcode leads in functionality that doesn't pertain to the editor itself. AppCode is slightly easier to use as it allows for a lot of customization, but overall they're not too different.

### 2.5 Selection

From the discussion above, we can see that Xcode simply overpowers its competitors in functionality and support. Their are some drawbacks, but having the support of Apple and integration of its products is too great to pass up. We will be using Xcode to develop this application.

## 3 Capturing and Sending Data to MindBody

### 3.1 Options

To initially describe this piece, MindBody is the API service that our application will interact with in order to obtain data such as class availability, schedules, and user data. In order to capture such data from our service, we will need to develop some form of SOAP client as the service is a SOAP service, sending and receiving XML.

The first technology that comes to mind would be to find an existing framework for iOS Swift applications to build and make SOAP requests. After a bit of research, we found an open source framework, SOAPEngine that allows for us to deal with SOAP services.

SOAPEngine is a generic SOAP client that allows for capturing and sending data between service and user. One of the goals for using this framework would be that we would not have to develop such boilerplate code ourselves and instead be able to focus on higher level objectives.

Another technology we would be able to use would be Apple's native NSURLConnection. This class allows the application to load the contents of a URL through providing URL request objects. NSURLConnection's interface is scarce, only providing the basic controls to start/end asynchronous loads.[10] In order to enable configurations, one must perform them on the URL request object itself.

The final technology for this piece is another native Apple class - NSURLSession. NSURLSession is another class that establishes an API for downloading content from the internet.[11] There are multiple delegate methods within this framework that allows for background downloads while the application is either not running or suspended.

## 3.2 Goals

The main goal for this piece is to establish a route for data flow between our client side application and the MindBody API service. This is particularly important as much of our application's features revolve around data driven interactions such as registering for a workout class or viewing the user's daily schedule.

## 3.3 Criteria

With this in mind, there are certain criteria that will need to be met in order to establish a foundation for such requirements. These criteria include native availability, maintainability, and ease of use. Below is a table which highlights the pros and cons of each of the above candidates.

| Options | Native Availability | Maintainability | Ease of Use |
|---|---|---|---|
| SOAPEngine | No | Difficult | Easy |
| NSURLConnection | Yes | Moderate | Moderate |
| NSURLSession | Yes | Easy | Moderate |

## 3.4 Discussion

As displayed in the table above, it isn't quite clear which option yields the best results for our use cases. Thus, we will now discuss the details of the strengths and short comings of each. SOAPEngine is an open source framework, installable through CocoaPods, which provides an abstraction of what we would implement as a SOAP client. This comes with its pros and cons. Firstly, we get the luxury of not having to write our own boilerplate SOAP client and we would be able to focus on more higher level objectives. However, two negatives arise with this benefit. Maintainability would be difficult as we have limited control over the SOAPEngine framework. In the event that a bug arises in sending network requests which stem from this open source framework, we would end up debugging someone else's code rather than our own implementation. Next, NSURLConnection yields a bit more promising implementation. This is an Apple class under the native Foundation framework. This is better than SOAPEngine as we get more low level control over how we implement our network requests. However a setback from using this class and it's set of features is that NSURLConnection is a bit older than other native APIs. Its interface is a bit more scarce and many of its older methods have been deprecated over other possible options. And finally we have NSURLSession. This framework is also a native Apple class under the Foundation framework. It is currently well supported even under the new Swift 3 and has provided functionality to allow for background downloads and more. NSURLSession's interface is rich and full of methods for anything sending network requests would require.

## 3.5 Selection

After late client specification changes, we've decided to focus our RESTful HTTP interactions not with the MindBody API, but with the eBay and Firebase API. From the discussion above, it is clear which option we will move forward with. NSURLSession appears to be the main victor in sending

network requests to the eBay API. It contains essentially anything we would need in order to build our requests and send them to eBay.

# 4 Parsing Web Data

## 4.1 Options

We originally thought that our app would make heavy use of the Mindbody API, but that requirement changed to having a large focus on the eBay API instead. Luckily, the eBay API is more modern and uses the REST JSON protocol, meaning we no longer have any need for an XML parser. The following sections below are an outline of the XML tools we were considering, but they are likely no longer relevant to the project.

## 4.2 Goals

Speed is usually a huge consideration for high-performance apps such as web-browsers and databases. For us, it isn't. In fact, we care very little about the performance or memory usage of our XML parser, as it will be parsing very small documents with a very large processor. The delay of having a XML parser that is even 10x slower than an alternative will be unnoticeable compared to the huge time required for the HTTP request that precedes it. Therefore, the only things we care about when choosing an XML parser are accuracy and the quality of its API.

## 4.3 Criteria

There are five criteria we'll consider when choosing a XML parser: speed, memory usage, API language, whether it is accurate and an estimate of how long it will take to implement a parser for MindBody's API using it. Parsing data and memory usage is from several tests on a large XML document conducted by Ray Wenderlich.[16]

| Options | Speed | Memory Usage | API Language | Accurate | Implementation Time |
|---|---|---|---|---|---|
| XMLParser | 1.8s | 3 MB | Objective-C | Yes | 1 week |
| libxml2 | 1.2s | 3 MB | C | Yes | 2-4 weeks |
| Custom | Unknown | Unknown | Swift | Unknown | 1-6 months |

## 4.4 Discussion

Apples Foundation framework, which is bundled within the iOS platform, contains two XML-parsing libraries: XMLParser and libxml2. XMLParser (formerly NSXMLParser), is the most modern. Its most appealing feature is that its written with an Objective-C API, meaning its interoperable with Swift without a wrapper. While the slowest option, it is also the most reasonable choice for our project, as it would allow us to hit the ground running without worrying too much about XML parsing. iOS devices have extremely fast processors, and any speed advantage at this level would be unnoticeable.

libxml2 is a XML parsing library developed for the GNOME Project[14]. It is performant, dependable and used in many popular applications such as Chrome, Safari[13], GNOME, Python, PERL, PHP, and more. It is one of the few XML parsing libraries that is still being actively maintained, although the code base is very stable and we expect very few modifications to it in the future.
If we were building a very large application with a high dependency on XML in which performance is key, libxml2 would be the ideal choice, as it has been for several important projects. For our purposes, however, we estimate to receive a barely-measurable speed improvement from using libxml2

in our app, and the lack of a Swift-compatible API means wed have to spend time writing a wrapper for it to use in our app, a distraction wed be better off avoiding.

Creating our own XML Parser is the most interesting and risky option of the three, although it would be satisfying to pull off. As of yet, we are unaware of any XML parsers written in Swift - most seem to be C-based. Creating a XML parser is no easy task though (that alone could be a senior-project), and would require a serious amount of research and design work. Additionally, the most performant XML libraries are written in C. The optimizations they can make regarding memory management are not possible in Swift, which used automatic reference counting (ARC).

### 4.5 Selection

We originally decided on using Apple's XML parsing library, but since we are now parsing JSON instead of XML, we will just use Apple's JSON parsing library instead.

## 5 Storing General User Data/User Login

### 5.1 Options

Our application will need a way to store the data that it collects on the user. This data will primarily consist of the user's fitness Personal Records (Test Results), their MindBody login data, and their saved Activities. The scope of this application does not include a backend to store this data, so we need to store the data locally. We have four main options for storing data CoreData, SQLite, NSUserDefaults, and Property Lists.

Core Data uses SQLite queries to store its data in .db files, which eliminates the need for a separate backend database. Unlike more traditional database tables, Core Data focuses on objects as it stores the contents of an object which is represented by an Objective-C class.

SQLite is an open source database engine. It implements a typical SQL database engine without the need of a server. SQLite is written in C needs to be embedded into an iOS application, FMDB is a popular Objective-C wrapper around SQLite.[5]

Property List are a set of files containing either an NSDictionary or an NSArray which contains the archived application data. A certain subset of classes can be archived into this property list: NSArray, NSDate, NSString, NSDictionary. Other objects cannot be used as a property list.

Finally, NSUserDefaults is typically used to store basic objects for user preferences. It is one of the most common methods for storing local data, and it can be used to store application states and user login access tokens.[8]

### 5.2 Goals

The main goal for storing user data, is to find an efficient solution for storing local data on our application without the need for a backend database. The solution needs to be capable of handling the capacity of our data as we'll be tracking activity in addition to more basic data objects.

### 5.3 Criteria

With these goals in mind, we'll be evaluating our data storage possibilities on capacity, efficiency, and native availability.

| Options | Native Availability | Efficiency | Capacity |
|---|---|---|---|
| SQLite | No | High | High |
| CoreData | Yes | Moderate | High |
| NSUserDefaults | Yes | Low | Low |
| PropertyLists | No | Low | Moderate |

### 5.4 Discussion

From the information gathered, it's clear that CoreData and SQLite lead as our best options for storing the data. There are tradeoffs between both CoreData and SQLite: CoreData is more focused on objects so it's more powerful and there is more flexibility in data storage, but this means that it uses more memory and space than SQLite. On the other hand, objects are much easier to work with than SQL code, meaning it will be less error-prone to work with CoreData. SQLite is supported on multiple platforms (iOS, Android, Windows, etc.), however for our situation this is irrelevant as we're developing solely for iOS. CoreData wins in this respect as there additional support features specifically for iOS. Both Apple and our Client have recommended that we use CoreData to store the data. Property Lists and NSUserDefaults are meant for storing simpler data than our program requires.

### 5.5 Selection

After our client's specification changes, we've decided to move towards Firebase as the main option for storing user data. Additionally we will store data remotely rather than locally as remote data can be accessed more easily by users with multiple devices or multiple accounts.

## 6 iOS Development Language

### 6.1 Options

There are a few main technologies that comes to mind when talking iOS mobile development language. The first and oldest technology used by app developers is Objective C. Objective C is a general purpose object oriented programming language that is Apple's main programming language. It is based off of C and Smalltalk messaging and incorporates Cocoa and CocoaTouch for both mobile and desktop API interfaces.

The next programming language is Swift. Swift is a relatively new programming language developed by Apple. It is a general purpose, compiled programming language that utilizes the existing Objective C runtime library.[12] Swift is intended to work with large existing bodies of Objective C code bases and can compile under Apple's LLVM compiler alongside C, C++, and Objective C.

And finally we have React Native. React Native is a JavaScript framework developed by Facebook which allows developers to build mobile apps with JavaScript. This framework is also relatively new as it is only about a year old. One of main benefits of React Native's approach is that web developers would not need to learn another language in order to develop mobile apps for both Android and iOS. This would allow small startups and people with projects to rapidly develop mobile apps.

## 6.2 Goals

The main goal in choosing a viable candidate for a development language is to have an interface for as clear and concise code as possible. A language needs to provide access to all of the native Apple frameworks as well as be easy to build an application with in a matter of a few months. Additionally, a language should be easy to learn and comprehend.

## 6.3 Criteria

Some criteria must be established to rank our possible iOS development languages for our purposes. Such language criteria that shall be evaluated include native availability, maintainability, and ease of use. Below is a table to compare the options and their ranking criteria.

| Options | Native Availability | Maintainability | Ease of Use |
|---|---|---|---|
| Objective C | Yes | Easy | Moderate |
| Swift | Yes | Moderate | Easy |
| React Native | No | Difficult | Moderate |

## 6.4 Discussion

As displayed in the table above, each language option has its own strengths and weaknesses. Let's start with Objective C. Objective C is the oldest of the three options, yet is the most foundational as many of Apple's core frameworks such as Cocoa and CocoaTouch are based in Objective C. By far Objective C is safest option as the language is a bit more settled versus the latter two options. As a result, interfaces won't get deprecated and frameworks won't drastically change in a matter of months. However one weakness to Objective C is that because of the language's age, the Objective C's common practices feel outdated and the verbosity of the language at times can be a bit much. Additionally, Objective C's learning curve is not the best as many small common mistakes new programmers make can result in a crash. Swift does a bit better in this regard. While Swift is a new language that is constantly evolving (which makes it a bit difficult to maintain), the language is very safe in terms of common beginner programming errors such as null pointers. This safety comes with a trade off in performance. However for the purposes of our iOS application, this shouldn't be a problem. Lastly is React Native, which carry a few heavy cons. React Native is not native iOS development. Everything done by React Native does not quite have the articulate features and meticulous control that native iOS development brings. React Native is a good framework if we need to rapidly build an application on Android, iOS, and web. However, because this isn't the case, React Native is not a very strong candidate for us at this time.

## 6.5 Selection

From the discussion above, there is a distinct candidate that meets our requirements for our project, and that is Swift. Swift has the elegance of safe design and the functionality of a multi-paradigm programming language. While maintainability will be an obstacle due to the constant evolution of the language, this is strongly outweighed by the benefits Swift brings to our project.

# 7 Handling Touch Events

## 7.1 Options

Touch is the primary source of user interaction with iOS applications and handling touch events is likely the most important aspect of any app. In our opinion, the most important attributes of good touch handling are:
1. Correctly and swiftly determining intentional touch events, while filtering noise.
2. Correctly placing touch events on the screen. According to interviews with high-level Apple engineers[2], this is not just a case of being accurate  where users actually touch the screen is a different location than where they expect to see the touch registered.
3. Providing a simple API that hides superfluous details.

Apples UIKit framework contains several UI elements (buttons, sliders, labels, etc), and most of these have gesture/touch-recognition built in. For elements that dont have built in gesture recognition, UIKit provides a UIGestureRecognizer class which can be easily overlaid on an elements view area. While its widely recommended to use UIKits built in elements as much as possible, we will explore two alternatives below; masking custom components with UIKit's UIGestureRecognizer or working with UIResponder, the central handler for touch events in an application.

## 7.2 Goals

Since this is a problem that has been solved fantastically by Apple, and due to the fact that most of the raw input received by the touchscreen is hidden from the developer anyways, it would be best for us to choose the simplest solution that hides unnecessary details. There's no need to reinvent the wheel.

## 7.3 Criteria

Below we will rate the three options on three criteria; ease of use / conciseness, power, and the experience quality for the user we predict we could provide.

| Options | Ease of Use | Power | Experience |
|---|---|---|---|
| UIKit Components | Easy | Satisfactory | Best |
| UIGestureResponder | Medium | Satisfactory | Best |
| UIResponder | Hard | Most | Satisfactory |

## 7.4 Discussion

It is impossible to access input from an iOS device's touchscreen directly without jail-breaking. UIKit is a large framework however, and provides several level of access to touchscreen events. The highest level is through components with built in gesture recognizer. These are nice logical components to build an app with, including buttons, sliders, tables, text fields, image fields, grids, and more. They contain various properties that allow developers to easily set what data they contain and to manage how they display that data. Most also contain a sort of gesture recognizer which receives a signal when it is touched in a certain way. Buttons are touched, sliders are swiped, images are pinched and rotated, etc, etc.

For custom elements, or UIKit elements that don't contain a gesture recognizer, it is possible to make any object responsive to a type of gesture by using a UIGestureRecognizer. This is our second option. Many developers utilize this option when building highly custom components that are not in UIKit. However, when using standard components which already have an implementation in UIKit,

it is smarter to just use the UIKit option. This future-proofs your application when Apple releases new devices with new capabilities, such as when the iPhone 6s was released with Force Touch, since these capabilities are automatically added to UIKit's built-in components.
The final option is to use UIResponder. UIResponder is a global dependency of any iOS application that informs the application whenever the screen is touched. It then sends the coordinates of the touch every time the touch moves a significant amount (about a millimeter). This is a less object-oriented method of programming, as a global dependency of the application would handle touches instead of individual UI components. The component would then need to be notified to change state by the application.

### 7.5 Selection

While using UIResponder to notify the responder chain happens in the background, it is likely less confusing to not implement this behavior ourselves. That leaves using the gesture responders built into UIKit components, or layering UIGestureResponders on top of custom components. While we will likely do a mix of both, layering UIGestureResponders is redundant and unnecessary in most cases. Therefore, we will use the gesture responders built into UIKit by default.

## 8 Storing/Modifying Application State

### 8.1 Options

Apple's frameworks provide a lot of functionality, but do little to tell developers how to manage their application's state. They are opinionated however, and they provide some components such as storyboards that make creating a simple app easy, but can become convoluted as an app grows. Since our app is relatively complicated, we want to remember to think big when designing how our application manages its state.

There are a few options we can choose from. The first is Apple's recommendation, which is to use MVC (Model-View-Controller). The second is to use MVVM (Model-View-ViewModel), a paradigm popularized by Microsoft. The third is to use dependency injection to inject a central store into view controllers from the main application, an old functional programming paradigm which has been recently popularized by libraries such as React.

### 8.2 Goals

The main goal of a state model is to make it easy for developers to find the data they need to find. Secondarily, data shouldn't be replicated unnecessarily, especially when large. Third, the application's state should be easy to persist. Fourth, the data should be decoupled from the views, making it easy for developers to change the presentation without altering the underlying model.

### 8.3 Criteria

Here we will rate the three paradigms based on the goals mentioned above: Ease of Use, Unnecessary Redundancy, Ease of Persistence and whether the data is decoupled from the view.

| Options | Ease of Use | Unnecessary Redundancy | Ease of Persistence | Decoupled from View? |
|---|---|---|---|---|
| MVC | Easy | Yes | Hard | Tightly Coupled |
| MVVM | Medium | Yes | Medium | Loosely Coupled |
| Central Data Store | Easy | No | Easy | Decoupled |

### 8.4 Discussion

MVC is the paradigm recommended by Apple and works great for small applications. It is the epitome of object-oriented programming, in which each view is an object and contains all the data that it needs to operate. The problem is that most applications have more than one view, and moving data between views can be difficult. For each transition between views, a custom function must be written that is aware of the data in each view and how to transfer the data between these views. These functions can become very complicated and must be changed whenever a view is changed. Additionally, since every view holds its own data, it can be difficult to know where data is, how to persist data to permanent storage, and data can be unnecessarily duplicated.
MVVM solves some of the issues associated with MVC by moving the data to a data model. The data model knows what data the view needs, but not how it is presented. Since the data model doesn't need to know how the data is presented, the data does not have to be stored in view objects. Instead it is stored in the most primitive types possible - strings, arrays, integers, images, etc. This allows the developer to easily change the data's presentation.
MVVM doesn't solve issues with data duplication and persistence however. Using a central data store that is injected into views does however. Since views no longer store data, state is all in one place and therefore easy to keep track of and easy to persist. Views just hold pure transformation functions that make the store's data presentable. This methodology is best mixed with a reactive programming library which makes it easy for view elements to watch the applications state and respond to changes that concern them.

### 8.5 Selection

We intend to use a central data store for managing our application's state. We may start with MVC, since it is easy for a small app, but will definitely end up using a central data store to store our application's state, and hopefully will use a reactive library (or write our own) to ensure our views update to state changes.
Update: Currently a lot of our app state is held in different view controllers. After we reach MVC though, we intend to look over our data organization and transition to a central data store.

## 9 Responding to Changes in Application State

### 9.1 Options

There are three main options in native iOS development that allow for the communication between changes in data and the observation of such updates. The first of the three is Key Value Observing. Simply put, Key Value Observing (KVO) is the process in which one object is alert to the changes in another object's properties.[7] KVO is essentially event driven inspection between specific objects through listening on a unique key path.

Next is NSNotifications. The concept of this can be boiled down to a unified "Notification Center" singleton. which lets objects be notified of events that occur.[9] This concept enforces communication between controller and centralized object with minimal coupling. To differentiate between controllers, each is assigned a key to allow other objects to listen to specific objects. These listeners can also react to such events without coupling to the controller.

Finally we have delegation, which is one of the most popular of the options. Delegation is essentially the concept of giving the work of the controller to a "delegate", which in turn performs actions specific to the delegate on behalf of the delegator.[3] This provides functionality previously unavailable to a controller.

### 9.2 Goals

The main goals in choosing a method of object observation is to allow controllers to be self contained. This will promote controller reuse along with clean and concise code. A good candidate in controller communication pattern would be easy to implement and sync.

### 9.3 Criteria

With the above goals in mind, we will be evaluating candidates based on three categories: applicability, overhead, and ease of use. Below is a table of the options and respective categories.

| Options | Applicability | Overhead | Ease of Use |
|---|---|---|---|
| Key Value Observing | Moderate | Easy | Moderate |
| NSNotifications | Moderate | Moderate | Easy |
| Delegates | Moderate | Moderate | Easy |

### 9.4 Discussion

Each of the above options: KVO, NSNotifications, and Delegation - have their strengths and weaknesses. Additionally, each option has their own specific use cases. KVO is for more ad hoc communications between specific objects. NSNotifications is when the application has multiple objects observing the same event. Delegates are for designing class based interfaces with protocols. While some use cases may overlap, there are definitely situations where certain options are much more viable and effective rather than others. I find that delegation is essential in almost every application in iOS. NSNotifications and KVO can be selected in different levels, one over the other in some cases. Typically, it would be the most safe to KVO for property level events and the delegate patterns for all other cases. In the event that something is not available through these two options, shifting focus towards NSNotifications is the final choice to take.

### 9.5 Selection

Delegation is almost impossible to avoid in iOS world, but can be unclear sometimes to the coder. We will use delegation where necessary, but have decided to actually use a fourth option, ReactiveSwift, to perform reactive changes to the application's state.

## 10 Developing the User Interface

### 10.1 Options

Our application will need a well designed user interface to interact with the user. The designs for the screens were created initially on paper and then they were drawn up more formally on Sketch. To actually implement our wireframes in our application we have three main choices: Storyboards, Xibs, or Code.

The Storyboard is a visual representation of the screens of the application. It contains a sequence of scenes, with each scene having a view and a view controller. These scenes contain the objects and controls that the user can interact with and they're linked together transitionally with Segues. All of the visual aspects of the user interface are created during the design process.

Xib stands for XML Interface Builder it is the method introduced before the storyboards, where the programmer can design the full user interface by dragging and dropping windows, buttons, text fields, and other objects. This method maintains a separation between the graphical view and the

view controllers unlike with the storyboard.

Finally, the last option to creating the user interface would be to code it 'manually'. Foregoing the drag and drop tools that come with Storyboards or Xibs, and designing the code for the user interface gives added flexibility and customization.[6]

## 10.2 Goals

The main goal in choosing the User Interface development tool is to implement our wireframes in an efficient way. We want to be able to completely cover the aspects of our wireframes, so the method we choose needs to have enough flexibility to create what we need.

## 10.3 Criteria

Based on the goals above we have four categories to evaluate our UI development tool: Versioning, Performance, Prototyping, and Functionality. Versioning refers to the ease of development when our entire team will be working on the application at once. Performance is how well the UI will perform when it is implemented. Prototyping is the ability to see how the layout will look as we develop. Finally, functionality addresses how well the tool accomplishes creating we want from the UI.

| Options | Versioning | Performance | Prototyping | Functionality |
|---|---|---|---|---|
| Storyboard | Poor | Moderate | High | Moderate |
| Xibs | Poor | Moderate | High | Moderate |
| Code | Good | High | Low | High |

## 10.4 Discussion

Storyboards and Xibs are similar in function with storyboards being a newer method Apple has introduced which integrates the View and View Controller. Storyboards have the advantage of making prototyping the flow of an application very easy as everything in the UI is laid out before the coding. With this method it becomes very easy to create a working prototype of an application; however the integration of the view and view controller in storyboards means that controllers can't be reused, they're dependent on the rest of the Storyboard to function. Storyboards handle the transition between views with segues, but they don't handle the flow of data, this still has to be configured with code. Xibs on the other hand are useful as they are simply standard view layouts that can be reused as repeated templates. Custom code is more complicated to implement than Storyboards or Xibs, but anything that is technically feasible can be implemented with code. This will be useful for our application as we want to have dynamic UI features and effects that are not inherently present in the drag and drop features. Code is easier to develop in a team as it does not suffer from any additional merge conflicts, versioning happens as normal. Code does lose out to the other methods in its ability to prototype as it's hard to see the layout in action, however it makes up for this in its lack of overhead which results in increased performance.

## 10.5 Selection

For our application we will primarily be using code to design the user interface. It has the capability to implement the dynamic layouts and effects we want, while allowing our team to develop at once. We may use Storyboards or Xibs to help visualize initial prototypes to get a feel of how our application will look.

# 11 Conclusion

The tools in this document were evaluated on different criteria, some of the more important criteria being native availability, functionality, and ease of use. We wanted our choices to be simple to use, but at the same time we didnt want to add excessive overhead by using unneeded frameworks. Many of our choices were driven by native availability for iOS, as this is the most optimal choice for our targeted platform. Some of our choices were driven by Client request, as they want this project to be a learning experience both for them and us: for example, this is one reason were developing in Swift 3 instead of the Objective-C that eBay typically uses. Overall the technologies we have chosen will maximize the maintainability and effectiveness of our application.

## 12 Signed Participants

**Students**
Brandon Lee
Rutger Farry
Michael Lee

## 5.1 Tech Document Changes

The biggest change we encountered throughout our project was removing usage of the MINDBODY API from our feature list. This actually had a relatively low impact on our technology choices, and if anything simplified our decision-making. Since MINDBODY was a SOAP-based XML API that lacked a Swift wrapper, implementing MINDBODY in our app would require using a library for making HTTP calls, as well as an XML parsing library, for parsing the results of those calls. We considered this and spent a good deal of effort evaluating different options while writing our original tech document. However, in the end, almost our entire web stack was based on Firebase (which has a native Swift library). We only had to use an HTTP library for interacting with the eBay API and did not have to make use of an XML parsing library at all.

There was only one other major change from our original tech document, and that was the removal of a reactive library from our application. In our original document, we had considered using a reactive library such as RxSwift or ReactiveCocoa for our application to help manage our data and keep our UI in sync with our app's state. In the end, we found that Firebase provided a lot of this functionality out of the box, and decided we did not want to bring in an additional external dependency. For sections that used the eBay API, we just went with a more classic imperative model. Since the functionality was pretty simple, we just built some basic tests and did not run into any problems in that area.

# 6 Weekly Blog Posts

## 6.1 Brandon Lee

### 6.1.1 October 14, 2016

**Problems**

- Communication was a bit rough as clients and students were using email for half of the week before switching to Slack.
- Additionally clients and students have not met up in person yet to discuss details of the project, so as of right now some details are still vague due to the broad scope of specifications.

**Progress**

- Wrote Problem Statement.
- Created GitHub Organization and repo.
- Created Slack for remote communication.

**Plans**

- Next week Wednesday we are planning to go up to Portland to discuss further details about the app with clients at eBay in person.
- I plan to catch up on learning up a bit on some iOS concepts in the extra time before we formally begin.

### 6.1.2 October 21, 2016

**Problems**

- Since we're still just ramping up, currently still blocked on not having an exact direction to proceed in terms of app architecture and design. Need to confirm major requirements in accordance with these core principles. Will be sitting down with fellow students this weekend and discussing this weekend.
- Communication still seems to be slow as not many people (clients and students) seem to be using the Slack, so communication as of currently is still just a mixture of emails and Slack.

**Progress**

- Visited eBay Portland in person to discuss details and iron out plans.
- Got a clearer idea of what is required by the clients in terms of app functionality and features.
- Created a resources page with useful links.
- Received back Problem Statement, will update and turn in by Monday.

**Plans**

- Develop requirements document by next week.
- Sit down with team and discuss higher level app architecture.
- Create Sketch mockup design of app.
- Continue reading up on iOS.

### 6.1.3 October 28, 2016

**Problems**

- Busy week for both students and clients. Was not able to obtain an updated list of functional requirements until Thursday night, the night before the Requirements Document rough draft due.
- Difficulty organizing time to collectively work together as students on Problem Statement and Requirements Document due to scattered midterm schedules.
- Communication on Slack still not very strong between students and client.

**Progress**

- Hammered out final copy of Problem Statement as well as rough draft for Requirements Document.
- Got in more communication between students on Slack.
- Got more communication with clients through email.
- Laid out a bit of the technical designs (higher level architecture, data flow).

**Plans**

- Develop Sketch designs from wireframes
- Layout more architecture/design
- Continue reading up on iOS.

### 6.1.4 November 4, 2016

**Problems**

- Design and functionality is a bit broad at this time.
- Client wanted to obtain first draft of design in Sketch.

**Progress**

- Got initial sketch designs implemented and sent to client.

**Plans**

- Plan and write Technology Review and Implementation Plan
- Continue reading up on iOS.

### 6.1.5 November 11, 2016

**Problems**

- Busy week for both students and clients. Was not able to obtain feedback on initial sketch designs.
- Technology Review and Implementation Plan seldom planned or started at this time.

**Progress**

- Met together as students and laid out design for what we wanted to do with the document.

**Plans**

- Write Technology Review and Implementation Plan
- Continue reading up on iOS.

#### 6.1.6 November 18, 2016

**Problems**

- Schoolwork is piling up and getting busier.
- Communication between students and client seem to be getting sparse.

**Progress**

- Wrote and completed Technical Document

**Plans**

- Plan and write Design Document
- Continue reading up on iOS.

#### 6.1.7 November 25, 2016

**Problems**

- Schoolwork is piling up and getting busier.
- Thanksgiving Break

**Progress**

- Not much currently due to holiday week.

**Plans**

- Pick up where we left off last week after holidays.

#### 6.1.8 December 2, 2016

**Problems**

- Design Document not done.

**Progress**

- We completed Document.

**Plans**

- Wrap up this week and finish strong with finals next week!

#### 6.1.9 January 13, 2017

**Problems**

- Winter break ends, school is starting up again.

**Progress**

- Not much during the holidays.

**Plans**

- Meet up with eBay and make sure everyone's on the right page.

#### 6.1.10 January 20, 2017

**Problems**

- Donald Trump is President now.
- MindBody is no longer supported by our client.

**Progress**

- Met up with eBay through video conference.
- Update rest of the group members on current status of project.

**Plans**

- Now that we're all on the same page, we will begin implementation.
- Will meet up with eBay in the coming weeks as well.

#### 6.1.11 January 27, 2017

**Problems**

- Need to work on implementation more.

**Progress**

- Ramping up.
- Met up with everyone to get started together.
- GitHub sprint started, kanban board set up, work delegated.

**Plans**

- Start weekly meetings with teammates
- Organize project into different feature branches
- Continue work

#### 6.1.12 February 3, 2017

**Problems**

- Alpha build is due next week

**Progress**

- Organized GitHub repo
- Good progress as planned on the user login/profile screens
- Firebase (backend) successfully integrated
- CI put in place

**Plans**

- Wrap up alpha build goals
- Combine all our feature branches into main branch

#### 6.1.13 February 10, 2017

**Problems**

- Alpha build and midterm documentation due next week

**Progress**

- Ramping up development
- Currently working on various pickerViews for various tableview cells.. Can be annoying - best way to implement this is with hook delegates and extensions...

**Plans**

- Finish up as much as possible and merge into one branch
- Use use merged project to demo something cool

#### 6.1.14 February 17, 2017

**Problems**

- Everything is due today.
- Profile screen is a bit buggy.
- Slightly behind schedule due to the vast amounts of writing we have to do.

**Progress**

- Everything has been merged together.
- PRs have been code reviewed.
- Good chunk of the midterm report stuff is done.

**Plans**

- Update PRs with the code reviews.
- Work on store tab along with fixing PR and bugs (weight attribute).

#### 6.1.15 February 24, 2017

**Problems**

- School has been busy
- Still need to implement store screen
- Not sure how to proceed with OAuth 2.0 client implementation

**Progress**

- Demo to TA
- Email eBay on their opinion of Swift OAuth

**Plans**

- Attend class group demo next week
- Continue work on store screen

#### 6.1.16 March 3, 2017

**Problems**

- School is busy
- eBay API documentation is still needing to be read
- Need to learn collection views

**Progress**

- Got OAuth figured out
- Started store screen and data manager
- Went to class group demo this week

**Plans**

- Finish store screen

#### 6.1.17 March 10, 2017

**Problems**

- eBay data manager class is causing errors

**Progress**

- Fixed up data manager by moving OAuth stuff into its own separate struct
- Created PR for store screen
- Applied good swift design patterns throughout certain parts of the app

**Plans**

- Work on reports/presentation/poster
- Implement store screen item view

#### 6.1.18 March 17, 2017

**Problems**

- Dead week

**Progress**

- Fixed bugs in eBayItem struct
- Chugging along with item view
- Finished poster draft

**Plans**

- Work on reports/presentation
- Implement store screen item view

#### 6.1.19 March 24, 2017

**Problems**

- Finals week

**Progress**

- Implemented View item screen
- Made presentation and demo
- Wrote final progress report for winter term
- Merged all branches together, creating an MVP

**Plans**

- Spring break

#### 6.1.20 April 7, 2017

**Problems**

- Getting back into school mode

**Progress**

- Met up with teammates
- Merged everything together
- Started linting process

**Plans**

- Continue working on issues
- Merge linting branch with dev

#### 6.1.21 April 14, 2017

**Capstone Lists**

**Everything I need**

- Desk
- Outlet
- Macbook(s)
- Poster
- iPhone(s)
- Charger
- Lightning cable

**What I have**

- Macbook(s)
- iPhone(s)
- Charger
- Lightning cable

**What I still need**

- Desk
- Outlet
- Poster

**Problems**

**Progress**

**Plans**

#### 6.1.22 April 21, 2017

**Problems**

- Need to refine UI
- Profile screen, activity screen, home screen, schedule screen, and shopping screen all need to be polished

**Progress**

- Working on refactoring profile screen

**Plans**

- Work on store screen
- Meet up with eBay in the future.

#### 6.1.23 April 28, 2017

**Problems**

- Need to submit soon
- Need to merge work together

**Progress**

- Refactor profile screen
- Refactor store screen
- Refactor schedule screen

**Plans**

- Meet up Sunday to merge and wrap up
- Meet up with eBay on Monday

#### 6.1.24 May 5, 2017

**Problems**

- Wired article due
- Busy school week

**Progress**

- Met up with eBay
- Wrote Wired article

**Plans**

- Will need to meet up with eBay again in the future
- They gave us more things to fix that weren't in the initial requirements.

#### 6.1.25 May 12, 2017

**Problems**

- Midterm report coming up

**Progress**

- Powering through school

**Plans**

- Work on midterm report
- Work on midterm presentation

#### 6.1.26 May 19, 2017

**If you were to redo the project from Fall term, what would you tell yourself?** I would tell myself to get started earlier, ask for more code reviews from our client, and delegate work more evenly.

**What's the biggest skill you've learned?** I've learned a bunch on how to organize a team towards completing an objective. In addition, I realized how the software development process changes as we move away from school and more towards real world development.

**What skills do you see yourself using in the future?** I see myself using the skills I've learned here in iOS development a lot in the real world as an aspiring mobile developer. Namely, all things iOS, Swift, and Objective-C.

**What did you like about the project, and what did you not?** I liked how it was a mobile development project rather than either web development or research as this is where my interests are. One thing I didn't like about this project was how it was so geared towards writing. There was way too much written objectives versus actual development, the ratio could've easily been much better.

**What did you learn from your teammates?** I learned that it can be challenging working with others. But as long as everyone's consistently on the same page, it can definitely help in alleviating this struggle.

**If you were the client for this project, would you be satisfied with the work done?** Yes. As this is a student college project with the work done essentially for free and the base requirements being met, I'd say that the objectives have been satisfied.

**If your project were to be continued next year, what do you think needs to be working on?** More refinement, particularly in the UI state animations and towards the services side.

## 6.2 Rutger Farry

### 6.2.1 October 14, 2016

This week was focused on battening down the hatches on what our final product is going to look like. Up until last Friday, I only knew the platform and language our program would be implemented in. I had no idea what it would actually do, what it should look like, etc.

I now understand we will be producing a fitness app for iOS that connects to the Mindbody platform to communicate workouts between clients and coaches. We narrowed down the problem we'd be addressing in our Problem Statement, drew up wireframes, and began investigating what libraries we should use for things such as API calls, layouts, data management, etc.

Next week we will be driving up to Portland to meet with our team in the eBay offices, and should be able to begin work on the app. I'd like to get the structure outlined completely in code, and perhaps begin on some API calls. Looking forward to see what we can get done!

### 6.2.2 October 21, 2016

The highlight of this week was traveling to Portland and visiting the people at eBay's office that we'd be working with. We had a good high-level talk about the app's purpose and later drilled down into the specific requirements, with a more extensive functional requirements doc promised soon.

I now feel like we're nearing the level of confidence in the app's direction that we'll be able to start coding soon. The next steps we take are mostly to improve the quality of the code we write. Things like setting up continuous integration and developing UI redlines to create consistent expectations across team members and to ensure that those expectations are met. We will probably be focusing on this, along with writing / revising the class's homework documents before moving on to code.

### 6.2.3 October 28, 2016

We're getting into the thick of midterm season, which I think has everyone a little stretched. Most of our efforts this week were focused on getting our Requirements Document written and our Problem Statement revised. Working on the requirements document actually encouraged us to get some useful work done on designing our application architecture, however. I'm excited that we finally started drilling down into this and am excited to explore some of the decisions we made in the area of functional reactive programming and unidirectional dataflow in Swift.

The team has additionally been looking into where we want the UI design to go, but it hasn't progressed much beyond high-level discussions and looking at design inspirations. Hopefully as midterms start to wind down we can start laying down the foundation of the app.

#### 6.2.4 November 4, 2016

Progress is coming along well on the design and I feel that we have a very solid foundation to move forward on. We've written several design documents planning out the different stages of our implementation and have considered how to construct the different components of our app and how they will interact. With midterms winding down for me, I'm looking forward to starting to implement the app.

#### 6.2.5 November 11, 2016

We've started work on another document – a technology review to further explore possible implementation plans for the components that make up our app and to choose the best option. In my opinion, the best way to do this is to just start writing something with that library / architecture. The time constraints for the document make it impossible to do this effectively however, so we are just reading blog posts and software documentation from Apple instead.

Both us and our sponsors are excited for us to begin implementing our app, but most our work has gone into writing our documents however.

#### 6.2.6 November 18, 2016

This week was mostly coasting. We worked on our technical requirements doc and have started looking into the design doc. We've had a bit of communication with eBay about our mockups, but besides that haven't talked with them much. I've been working on some side iOS projects and have a good idea about how I'd like to implement state management in our app.

#### 6.2.7 November 25, 2016

This week was mostly spent eating turkey. I also ate some other things. Didn't do much related to the app.

#### 6.2.8 December 2, 2016

This week we worked on our design document along with the myriad of other homeworks we had assigned. We ended up writing quite a lot and turned it in Friday morning. We're looking forward to using the lack of homework over Christmas break to start working on the app.

#### 6.2.9 January 13, 2017

Winter break is over, and surprise(!) we didn't start working on the app. At least I can say that we are refreshed and ready to begin. We are planning to meet up with eBay soon to refine our designs before starting implementation in earnest.

#### 6.2.10 January 20, 2017

We met with eBay over a weird Skype clone. Something came up over the winter and it turns out the MINDBODY API costs like $25k a year (which is surprising since it's trash), so we are pivoting from using that to just making our stretch goal of implementing a fitness store using the eBay API into a primary goal. This shouldn't be too hard since we've already done some planning in this area, but should probably flesh those plans out a bit more it is now a primary goal.

#### 6.2.11 January 27, 2017

We still have not done much with implementation and now midterms are starting. We are feeling an initial sense of urgency and have decided to start holding weekly meetings to code.

#### 6.2.12 February 3, 2017

The alpha build is due soon. We've started coding in earnest and split out work into issues on GitHub. We've basically just made branches for every main screen and assigned everyone a branch.

#### 6.2.13 February 10, 2017

Worked on laying out the homescreen using a UICollectionView. This was pretty nice and allowed for a good amount of flexibility that I think we might want if we ever make this into an iPad app.

#### 6.2.14 February 17, 2017

We merged our branches this week and have an alpha build and midterm report due >.< We are going to reassess what remaining work needs to be done and will make some new issues and branches to split this work up.

#### 6.2.15 February 24, 2017

I'm kinda sad that we've gotten Firebase so deeply integrated into our app. While it is convenient, I've heard horror stories of companies who've built a similar reliance on a third-party service deeply into their application and then suffered when the service was shuttered or increased their prices.

In other news, we demoed our app to our TA today and it seemed to go well.

#### 6.2.16 March 3, 2017

Pretty much my only focus right now is organizing the 2017 Randall Fox Beaver Omnium this weekend and making sure that goes smoothly. I will be on the race course all day and probably only sleep about 4 hours this weekend, so am not going to work on the app.

#### 6.2.17 March 10, 2017

Still haven't worked on the app a lot. Need to start work on our term report and videos, as well as making a rough draft of our Expo poster. Cycling season has begun though, and although I'm not working to organize a race anymore, I will still be traveling / racing every weekend for the next six weeks, so won't have a lot of free time.

#### 6.2.18 March 17, 2017

I've been able to work on the app a bit during the week, but weekends are totally hectic. Luckily my course load is pretty easy and I'm not to pressured to work on school / the app during the weekend, but time during the week is definitely tight.

#### 6.2.19 March 24, 2017

Had a very hectic time submitting my group's term presentation in Seattle this weekend (we are racing at UW). Upon arriving at our home stay at 10pm, I holed myself up in our truck and recorded my portion of the app demo and then hitched a ride to a Starbucks so I could upload the 700MB file on my laptop. All next week I will be skiing around Montana with probably very rare access to WiFi, so will not be working on the app at all.

#### 6.2.20 April 7, 2017

After some fun skiing in Montana, our team came back from spring break refreshed and ready to work. I think we actually hit the ground running pretty well; we merged our progress together into `dev`, with Brandon and I merging first. (Michael finished his branch a few days later and had to resolve a bunch of merge conflicts). We met several times to resolve these as a team and got everything merged up by Thursday.

#### 6.2.21 April 14, 2017

Last weekend I got sick and got a really nasty ear infection on my way to my bike race at Whitman. Sat out for most of the events, but raced in the criterium. Upon returning on Sunday, I got so sick that I slept all day Monday and have been in pain all week. Didn't touch C7Fit.

#### 6.2.22 April 21, 2017

Still have an ear infection and have been taking it easy. I did some work on adding the trainer cell to the home screen. Now tapping on the trainer cell will open a list of trainers, but the list is kinda ugly and stretches images weird. Will probably work on that next week.

#### 6.2.23 April 28, 2017

Ear infection is abating and I'm starting to feel normal again. Worked a lot on a React Native side project during my illness but not too much on the app. This week I finished up the home screen for the most part though and merged my UI branch that cleans up the colors in the navbar and other areas.

#### 6.2.24 May 5, 2017

This week was super crazy. We went up to Portland on Monday to meet with eBay, which we didn't really have time for, but knew it needed to be done. (Did a little polishing on the app the night before, too). This week I had **nearly a million assignments due:** – 2 assignments in Operating Systems II – 2 assignments in AI – 2 assignments in Computer Architecture – 1 midterm in Computer Architecture – and had to work on the WIRED article for this class

Needless to say, no work was done on C7FIT.

#### 6.2.25 May 12, 2017

This week was a lot more chill, but still had a bit to do. Made everything about the app was pretty and ready for our midterm report / expo. Made the midterm report and video.

#### 6.2.26 May 19, 2017

**If you were to redo the project from Fall term, what would you tell yourself?** I would tell myself to do more work at the beginning to establish coding idioms for the project. I think that since we all started at about the same time, no one really looked at each other's work and for that reason our styles are all a little different. I'd also enable our linter and tests before the project started to ensure everyone is following a consistent style from the start. Changing our code to conform to our style guidelines halfway through the project was a little painful.

I'd also try to maintain more open communication with our client. They were pretty hands-off, but I think if we made the effort to communicate more we might have made the development process smoother for ourselves.

Finally I'd tell everyone on the team to commit / make PRs more. We worked on big feature branches that would consume a whole screen, and while siloing ourselves to screens was good, we still had some nasty merges and probably would've had more shared code if we merged more.

**What's the biggest skill you've learned?** Better discipline is probably the biggest takeaway from this. There were some times when there was really no pressure to work on the project. Well I think we did a decent job pacing ourselves, we still could've done better and avoided some mad dashes.

**What skills do you see yourself using in the future?** Discipline / task management.

**What did you like about the project, and what did you not?** The thing I liked about the project was also the thing I disliked. It was easy. Making an app is not breakthrough and is something I've lots of times before. While I enjoyed the extra free time to work on my own projects and go on bike races, part of me wishes that I worked on a project that captured my interest more, even if it would have taken more time.

**What did you learn from your teammates?** Brandon was super good at staying on top of things and scheduling out his work. I'd like to be more like that on the next project I do. Michael got up to speed on iOS super quickly and was a very good Swift developer by the end of the project. I don't know how he did it, but I should probably ask.

**If you were the client for this project, would you be satisfied with the work done?** Yes. App development is crazy expensive. Getting an app of similar quality developed by an US-based iOS shop would cost at least $30,000, and very likely much more. That being said, I think the app is overall very responsive and with a few tweaks, the user interface would be top-tier. Additionally, very few, if any gyms at the scale of Club Seven Fitness have an iOS app, or even a quality website. An iOS app will hopefully help them market their gym and provide a useful utility to current club members.

**If your project were to be continued next year, what do you think needs to be working on?** We'd probably start by working on feedback from our App Store launch. This would probably be mostly UI tweaks. Afterward, I think it'd be useful to build our own scheduling API and integrate that into the app, since the MINDBODY API is terrible *and* extremely expensive. Finally, building a web frontend for coaches to more easily view their client's workouts would be a cool expansion of the project, since using Firebase might be a little clunky for non-developers.

## 6.3 Michael Lee

### 6.3.1 October 14, 2016

The progress of this week has kept up with the pace of this class. Last week we had the first phone meeting with our client where we hashed out the basics details of our project. We had a group meeting after that to work out the details of our problem statement.

Overall the communication between our group and client were good. It was slightly difficult because they are remote, however we had good email communication and we set up a slack to help. There were some miscommunications due to online communication when getting our signatures, however because our client responded promptly we were able to complete the assignment on time.

Our group plans to go up to Portland to meet with our client next week Wednesday so we can talk about the app in detail and meet the team that were going to work with. As I haven't really worked in depth with IOS programming, I will need to catch up to my teammates and so I plan on learning some IOS in the coming week.

### 6.3.2 October 21, 2016

Our team hasn't had too many problems, as we've already met with our client twice. Our biggest problem would be working on the next requirements document and determining what is feasible in the time we have, as from the meetings we've had the project seems quite large.

On Wednesday we went up to Portland to meet with our client eBay. They showed us around their office and we were able to meet the team members we will be working with. During the meeting we discussed what technologies we will be using on our app and the look of each of the menus. We received our edited Problem statement Thursday night and we'll be revising that later this week.

For the future, we need to iron out a more official requirements document. Even though we have a preliminary version, we need to be clear on the particular functions. To do this we'll be using the rough wireframes that Luther provided us and our initial requirements document.

### 6.3.3 October 28, 2016

This week we were able to complete our requirements doc on time, but our team was a little strapped for time. Most of our team was busy throughout the week, so most of the requirements doc was done close to the deadline.

This process for this week was mainly just the creation of the requirements document. We did get a list of functional requirements from our client, however it was too late to be implemented into the current iteration of our requirements document. As a team we met to talk about some of the higher level design of the app, which will be useful for when we create the design document.

For the coming week we need to finalize our requirements document and continue to work on the higher level design of our application.

#### 6.3.4 November 4, 2016

This week we didn't do much as a group and I had two midterms to deal with, so the progress on capstone was stagnated.

For next week we need to get together as a group and plan our technology review and Implementation plan

#### 6.3.5 November 11, 2016

Personally, I'm more inexperienced than my other group members in IOS programming, so the implementation document requires more research for me to complete my section.

Our team met up and we discussed the implementation document, and have begun researching the different technologies we can use to complete our project.

For the next week I need to finish researching and writing the document so that it is complete by Monday.

#### 6.3.6 November 18, 2016

**Progress** This week we completed the technology review early in the week. We also met up to make some preliminary plans for our Design document.

**Problems** With midterms from other classes, it's hard to coordinate with group members, and it's unclear what we need to do for the design document

**Plans** Continue to make progress on the design document

#### 6.3.7 November 25, 2016

**Progress** We haven't made any progress this week due to the holidays.

**Problems** We didn't make any progress this week.

**Plans** We plan to work on the design document next week.

#### 6.3.8 December 2, 2016

**Progress** We completed our design document and turned it in without a signature, as we completed it late Thursday we didn't have time to get our client to sign it.

**Problems** The expectations of the design document were very unclear going into it so it took much longer to complete than expected.

**Plan** Start working on the Progress Report

#### 6.3.9 January 13, 2017

Winter break finished, but only Brandon started working on the project, our group as a whole has not made much progress.

#### 6.3.10 January 20, 2017

Our client has informed us that they cannot afford the MindBody API which means the direction of our project needs to be changed

#### 6.3.11 January 27, 2017

This week was spent working on the easier schedule screen that has been modified to be a simple weblink to the gym's website.

#### 6.3.12 February 3, 2017

The schedule screen is finished and I've begun looking into the healthkit api and how we can use it to query a user's pre-existing healthkit data.

#### 6.3.13 February 10, 2017

I've begun work on the location tracking portion of the application with the mapkit api. It is difficult to get the mapkit to record accurate data and display it as a line for the user to see as they run.

#### 6.3.14 February 17, 2017

Mapkit has been implemented roughly, as it works to track the user's location. There needs to be some fine tuning and bug fixes done so that the location is tracked when the user isn't moving or moves very little.

#### 6.3.15 February 24, 2017

The map portion of the screen is finished and it allows the user to save their runs to the offline database firebase.

#### 6.3.16 March 3, 2017

improved the look of the run lookup detail view, so the user can see a small snapshot of their run before they click to load the full details.

#### 6.3.17 March 10, 2017

Most of this week has been spent developing the less important features of the activity tools screen like the stopwatch and countdown timer. They are relatively simple features but must be implemented based on the requirements doc

#### 6.3.18 March 17, 2017

This week was spent trying to develop a heart rate monitor from the data that can be recorded using the flashlight and the camera application on the iPhone. Currently, the phone can record the data, but analyzing it has proven to be a harder task than initially thought and the results aren't very accurate. This wasn't in the requirements document, though, so it may not be implemented.

#### 6.3.19 March 24, 2017

Progress: This past week I spent most of the time refining portions of my code in preparation for the final clone that is coming on May 1st. The implementation of the heart rate screen is completed, but visually it is quite lacking. I added some additional modal views that displays a more informative message to the user based on the heart rate that they enter.

Problems: The finishing touches to our application will be primarily making it look better, but the challenge with this is we weren't given too many image assets and that will impact the final look of our project. Additionally, because we're developing primarily in code rather than using interface builder or storyboards making the views visually appealing takes much longer to do than normal. There is a time crunch coming up with the deadline approaching and so we need to focus to make the project as visually presentable as possible. We also want to tie in the healthkit data source with the firebase data that we've created, but the functions have all been written we just need to link up to a single source of truth

Plans: Next week we plan to go on a coding grind to make our project look as good as we can make it. Development will continue after May 1st, but we want to make it look as good as we can. The functionality is done so thats all we have left to focus on.

#### 6.3.20 April 28, 2017

Progress: This week we spend most of it refining all of the parts of the application to meet the specific requirements that we've outlined previously in our requirements document. This was done in preparation for the code freeze that we've been expecting on Monday. Our progress was good overall, and I would expect that we will finish on time.

Problems: The problem we've run into is that our requirements weren't evolving to an extent. We made some requirements so specific, that when we realized the application wouldn't use that anymore we couldn't change it. For the sake of the code freeze we made our application match the requirements document as close as possible.

Plans: We plan on going to meet with our client Monday to show them the current state of our application and see if they want to make any changes post code freeze for the sake of uploading to the apple iOS store.

#### 6.3.21 May 5, 2017

The code freeze has been implemented so development is relatively halted. There isn't much to do before expo anymore, and most of my work is centered on other classes as school gets busier

#### 6.3.22 May 19, 2017

Progress: We finished the midterm progress report early this week. We completed expo this weekend, so a major milestone is finished.

Problems: A lot of things to do before expo happens, along with the mental preparation that comes with it. Standing at expo for hours was not pleasant.

Plans: Meet-up with the client and finalize everything before the end of the term. Complete the remaining required documents for our class.

**If you could redo the project from fall term, what would you tell yourself?** If I could restart, I would interact more with my client and the software developers of eBay. The developers were a resource that I should've taken more advantage of, and seeing all of the cool projects that were created at expo I would've liked to try implementing things more innovative or advanced.

I would tell myself all the normal things like start earlier, and work continuously throughout the term, but overall I didn't lag too behind. It would have been better for me to start earlier as I had much more to catch up on than my group mates.

**What's the biggest skill you've learned?** The greatest skill I learned would be iOS development. I got experience with Apple's frameworks and SDKs like healthKit and mapKit, in addition to learning iOS. I would liken the experience to learning web development, and there is still much to learn in this aspect.

**What skills do you see yourself using in the future?** I definitely see myself bringing this skill with me in future jobs, as phones become more and more prevalent. All of the experience with Apple's technologies will also help me if I want to develop on any of their devices in the future.

**What did you like about the project, and what did you not?** I liked that it gave us practical experience in a field that does seem to be growing: phones are becoming more prevalent and thus it's useful to learn. On the other hand, I was not particularly interested in developing user applications. I do wish that our project had more research applications so it was more interesting, but it came with practical applications.

**What did you learn from your teammates?** I learned a lot about the best practices for developing in iOS, as both of them were experienced in that area being from OSU's app club. Communication and even though we are CS students, meeting up in person is very helpful. This is a little difficult as were all in school and have different schedules, but useful to note in the future workplace.

**If you were the client for this project, would you be satisfied with the work done?** I would be satisfied as we completed all of the requirements initially outlined for us by our client. In addition to just the requirements the application looks fairly professional especially given the assets that we were given. Our app looks just as good if not better than other comparable gym applications, and it looks more modern than their current website. Our application combines fitness tools and gym specific information together which many other applications do not.

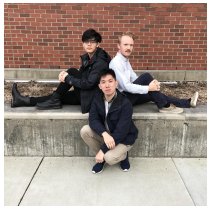
**If your project were to be continued next year, what do you think needs to be working on?** We would need to work on some documentation, but the requirements our client as outlined for us were all completed. The app could use development on additional features that would make it a better gym app like integration with watchKit for the apple watch or integration of some external api like the MindBody Api that the gym couldn't afford previously.

# 7 Final Poster

# C7FIT

## The Club Seven Fitness iOS app. Track your workouts and map your runs!

Brandon Lee, Michael Lee, Rutger Farry
Clients - Luther Boorn (eBay), Club Seven Fitness
leebran@oregonstate.edu
leemi@oregonstate.edu
farryr@oregonstate.edu



## Background

The local Portland gym, Club Seven Fitness, was looking for a convenient and personalized way for their clients to track their fitness progress. Our group created the C7FIT iOS application, which is personalized towards gym and will conveniently be with them throughout the day. It serves as a one stop portal for all of their client's needs, with multiple tools to assist and track workouts, a customized shopping center, and user profiles.

## FAQ

### Why did we undertake this project?

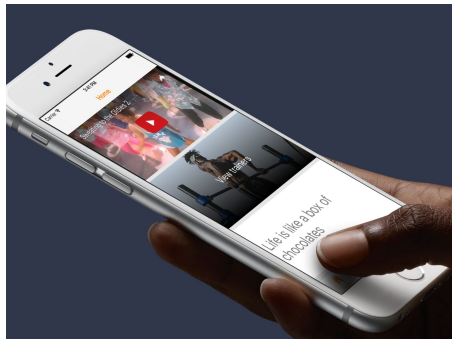
The C7FIT project was under taken to develop a mobile health and fitness application for local gym Club Seven Fitness in downtown Portland. At Club Seven Fitness, gym goers are unable to keep a clear record of their fitness goals in accordance to the classes they were enrolled in. This initial issue was the catalyst for our project, to develop a way for gym goers to track their workouts on the go.

### What generated the idea?

Because of the initial issue described above, we believed that a mobile approach to solving the problem on the go would be the best method in solving the problem. The various features of the iOS application stem from the many possible use cases of the gym clients.

### What goals and hypotheses did we have?

The main goal of the project is to build an iOS app worthy of deploying to the App Store. This app would solve the many use cases of Club Seven Fitness' clients. An initial hypothesis was to build an app where users could sign up for gym classes, but the initial goal fell through when the gym's backend services shifted their policies. Now our application is focused purely on health and fitness tracking along with a bit of fitness shopping.




## Implementation

### What did we do?

We built a multi tabbed iOS application with the following tabs: Home, Schedule, Store, Activity, and Profile. Each screen focuses on a specific component of the application as described in the feature section. The application supports any iPhone on iOS 10 or higher.

### How did we do it?

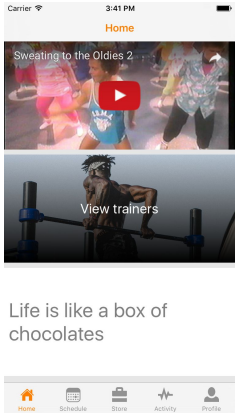
We developed the application through Apple's Xcode and Swift 3. In order to leverage cloud storage, we included Google's Firebase into our app through CocoaPods. All development was done as natively as possible through Apple's provided frameworks and libraries within iOS.

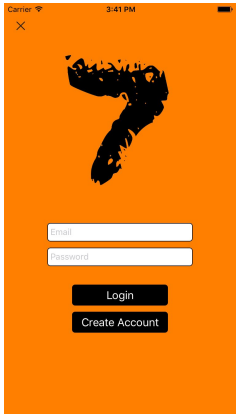
### Why did we do it this way?

We decided to develop as natively as possible in order to achieve a sense of unity throughout the whole application. While it may have been a bit easier to import frameworks to do some heavy lifting for us, doing things the native, and sometimes harder way yielded a greater learning experience as well as a less externally dependent app.

## Features

- Communication - Our application gives Club Seven Fitness owners and coaches the ability to interact personally with their clients through daily quotes and embedded instructional youtube videos.
- Profiles - The users can construct and save their own profiles to keep better track of their personal fitness goals. We have custom fields like push ups, sit ups, etc. in addition to integrating the fitness information from the native HealthKit API.
- Shopping - We integrated eBay's new shopping API to test their API and provide function to the gym's clients. We query their API for a list of products and the application displays the top rated product in that category.
- Fitness Tools - The app provides multiple fitness tools: a run tracker, heart-rate calculator, countdown clock, and timer. The run tracker is integrated with firebase which allows the user to save their runs. They can view their distance, time, and pace along with the path of their run.




## Importance

C7FIT is an important application for many reasons. As one of the first apps created for a local gym, it allows the app to make assumptions about the customer that would be infeasible to make about a larger corporation's customer base. In addition, it grants gym trainers holistic insight into their client's personal workouts, allowing them to tailor better personal workouts for their client.

## Results

While the app is not yet on the App Store, we plan to publish it by mid-June. Internal feedback among friends and family has been positive, and our mentors at both eBay and OSU have been satisfied with our work. We are looking forward to publishing to the App Store so our target audience can begin using the app.

## Conclusion

Building C7FIT was a great experience for all of us. For Brandon and Rutger, who had worked on iOS applications in previous internships, it was a chance to develop with a lot of freedom and an opportunity to help teach our partner, Michael about iOS development. It was very cool to develop an open source application. While there are some challenges with open-source development, such as hiding API keys, knowing that your project is in the public eye forces you to write better code, and enforces developers to take precautions they should be taking regardless.

Oregon State UNIVERSITY

# 8 Project Documentation

## 8.1 Installing

The recommended development environment is:

```
MacOS Sierra
iOS 10
Xcode 8.3
Swift 3.1
```

You'll need a few tools before getting started. Ensure you have a recent copy of Xcode downloaded. Then run the following two commands to install `bundler` and the Xcode command-line tools, if you don't have them yet.

```
sudo gem install bundler
xcode-select --install
```

Then, to download the code, run the following lines. We use some Ruby and Swift dependencies; the following commands ensure they're downloaded and hooked into the Xcode workspace.

```
git clone https://github.com/iOS-Capstone/C7FIT.git
cd C7FIT
bundle install
bundle exec pod install
```

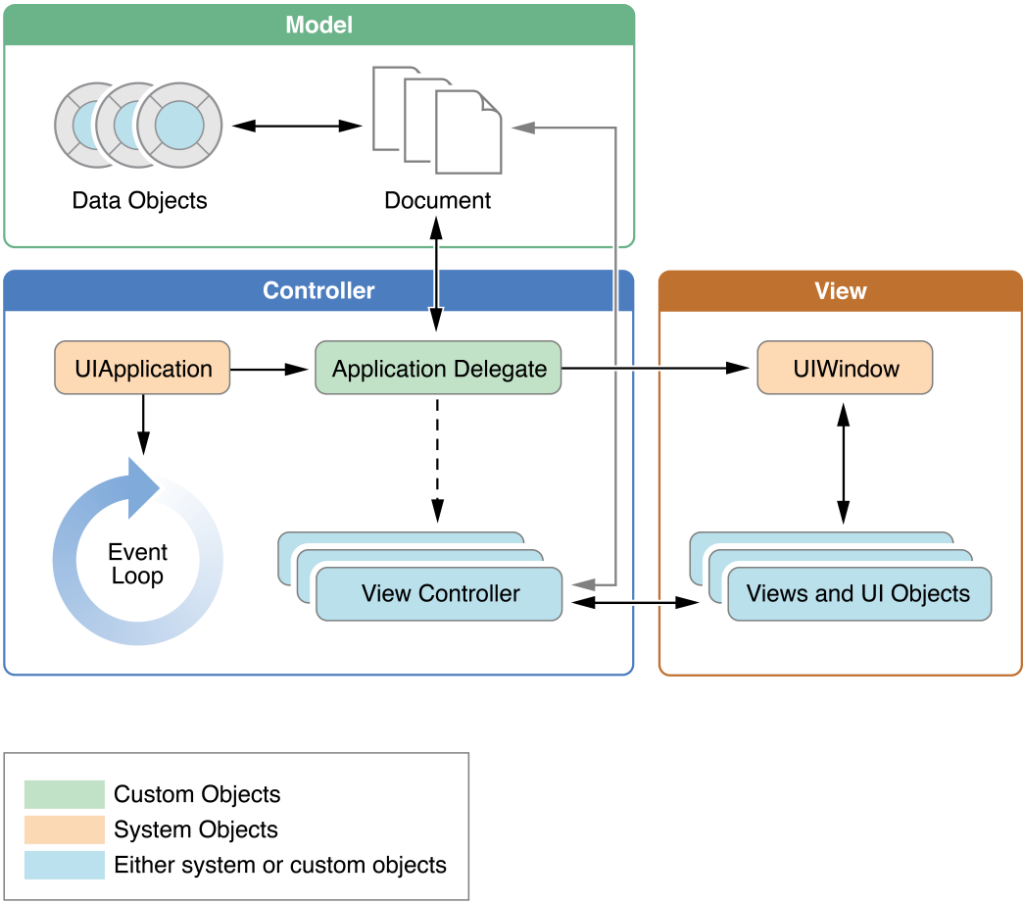
The project should now be openable and buildable through the `C7FIT.xcworkspace` file.

## 8.2 Running

Today, compiling and running an iOS project is as simple as plugging the target iPhone into your computer, opening the `C7FIT.xcworkspace` file, and hitting the "play" button in Xcode. For more detailed and up-to-date information on how to accomplish this, visit Apple's Developer documentation on launching your app on devices.

## 8.3 Application Life Cycle

Our application was designed under and obeys, for the most part, a standard MVC model and the life cycle of a typical iOS application. Shown below is a diagram of Apple's MVC life cycle that illustrates the performance of our application.




Application Life Cycle

## 8.4 Design

Now I will go over the typical structure of one a TableView Screen in our application with an example from the code. The C7Fit Application consists mainly of these TableView screens, but the principles talked about can be extrapolated to other views like UIView and CollectionViews.

### 8.4.1 TableViewController

Taking a look at the HealthKitTableViewController.swift we can see the general structure of the defined class:

```
class HealthKitTableViewController: UITableViewController {

    // MARK: - Properties
    let healthInfoID = "healthInfo"
    var healthKitManager = HealthKitManager()

    // MARK: - View Life cycle
    override func viewDidLoad() {
      ...
        healthKitManager.authorizeHealthKit()
        self.title = "Today's Activity"
      ...
    }
    override init(style: UITableViewStyle) {
      ...
```

```
    }
    required init?(coder aDecoder: NSCoder) {
      ...
    }
    // MARK: - Table view delegate
    override func numberOfSections(in tableView: UITableView) ->
Int {
        return 2
    }
    override func tableView(_ tableView: UITableView,
numberOfRowsInSection section: Int) -> Int {
        switch section {
        case 0:
            return 4
        case 1:
            return 2
        default:
            return 0
        }
    }
    override func tableView(_ tableView: UITableView,
titleForHeaderInSection section: Int) -> String? {
        switch section {
        case 0:
            return "Health Statistics"
        case 1:
            return "Recent Activity"
        default:
            return "Other Statistics"
        }
    }
    // MARK: - Table view data source

        override func tableView(_ tableView: UITableView,
cellForRowAt indexPath: IndexPath) -> UITableViewCell {
        switch indexPath.section {
        case 0:
            switch indexPath.row {
            case 0:
                let sampleType = HKSampleType.quantityType(
forIdentifier: HKQuantityTypeIdentifier.bodyMass)
                let titleLabel = "Weight"
                return queryData(titleLabel: titleLabel, sampleType
: sampleType!, path: indexPath)
            case 1:
                ...
            case 2:
                ...
            ...
            default:
                break
            }
    case 1:
          ...

        return UITableViewCell()
    }
    ... Function of the class ...
```

```
68 }
```

Starting from the beginning of the class definition, on line one, we can see that the TableViewController inherits the UITableViewController delegate. This means that within our new class we have to explicitly define a set of protocols that outlined by the delegate.

The first section of the class is the defined as the properties of the class. These are variables defined that need to be used within the global scope of the class. Most commonly they will be instances of data managers, title strings, cell reuse IDs, etc.

Following the properties, we have functions of the View Life Cycle. These functions are required and are called when the view is instantiated. Within these functions, there are a few required function calls initiated when Swift creates the skeleton function, and here we set up anything that must be loaded and defined once the view has been set up like permissions and titles.

Next is the delegate which outlines the look of the TableView. Here in the HealthKitTableView it defines the number of sections, rows, and titles for each row. There are other delegate protocols from UITableViewController that can define row height, section type, editing, button types, etc.

The last delegate methods are the data source methods. These methods define the function of the TableView, hence their name data source. This is where the contents of the cell and it is pulled from the ViewCell.

### 8.4.2 ViewCell

TableViewControllers make use of ViewCells to populate their cells. Keeping these separate from the Controller class allows reuse of common cells for added simplicity and efficiency. We will continue with the previous example and show the structure of the HealthInfoCell a simple TableViewCell.

```
1  class HealthInfoCell: UITableViewCell {
2
3      // MARK: - Properties
4      var titleLabel = UILabel()
5      var infoLabel = UILabel()
6
7      // MARK: - Initialization
8      override init(style: UITableViewCellStyle, reuseIdentifier:
       String?) {
9          super.init(style: style, reuseIdentifier: reuseIdentifier)
10         setup()
11         setupConstraints()
12     }
13
14     required init?(coder aDecoder: NSCoder) {
15       ...
16     }
17
18     // MARK: - Layout
19     func setup() {
20         addSubview(titleLabel)
21         addSubview(infoLabel)
22     }
23
24     func setupConstraints() {
25         titleLabel.translatesAutoresizingMaskIntoConstraints =
       false
26         let titleLead = titleLabel.leadingAnchor.constraint(equalTo
       : leadingAnchor, constant: 10)
```

```
27          let titleTop = titleLabel.topAnchor.constraint(equalTo:
    topAnchor, constant: 10)
28          let titleBottom = titleLabel.bottomAnchor.constraint(
    equalTo: bottomAnchor, constant: -10)
29          NSLayoutConstraint.activate([titleLead, titleTop,
    titleBottom])
30
31          infoLabel.translatesAutoresizingMaskIntoConstraints = false
32        ...
33          NSLayoutConstraint.activate([infoTrail, infoTop, infoBottom
    ])
34         }
35
36 }
```

From this example we can see that the structure begins similar to the TableViewController where common properties are defined at the top of the class. Continuing with that trend, right below the initialization functions initialize the cell by calling two setup functions that are defined below. These initialization functions have some functions that need to be defined in order for the delegate requirement to be satisfied.

Looking at the layout of the cell we can see that both the UILabels from the property section are added to our ViewCell. There is no text within them because if we go back to the TableViewController we can see that we defined the titles. This flexibility lets us use this one ViewCell structure for a multitude of cells in the TableView.

Finally, the last function sets up the constraints of the view. For our application, we coded all of our views using constraints, which means that we did not use any of Apple's drag and drop tools like Interface Builder. To do this we define some constraints on each of the UILabels to determine their appearance: the titleLabel in this case is constrained 10 units to the right of the left of the cell (leading), and 10 units above and below the bottom and top of the cell respectively. Anchors are the iOS defined hard properties of each view item, and in this case the dimensions of the cell defined within the overall TableViewController.

## 8.5 Firebase

The Firebase data manager is an essential part of our application as it handles all of the application's communications with the offline Firebase server. Fortunately, most of the hard work with Firebase is already done by Google and our manager. To use Firebase within our application all that needs to be done is to import Firebase, initialize the manager, and authenticate the user.

```
1 import Firebase
2 let firebaseDataManager = FirebaseDataManager()
3 firebaseDataManager.monitorLoginState { _, user in
4   guard let userID = user?.uid else { return self.present(
     LoginViewController(), animated: true, completion: nil) }
5     \\ Execute Code Here
6 }
```

The above code shows an example of how to use Firebase within our application. The data manager's monitorLoginState will handle user login if the user is not already authenticated, so all that needs to be done is encapsulate any Firebase related code within this wrapper function. More documentation on the usage of Firebase can be found on the Google Documentation site at https://firebase.google.com/docs/ios/setup.

### 8.6 Ebay API

Our application uses the eBay API to query for information on various gym related items. To do this we need to access the eBay API which uses oAuth for authentication. There was a discrepancy with their oAuth as it didn't conform to the standard oAuth protocols so our application has a wrapper function within the EbayAPITokenManager to handle this. One problem is that the oAuth token will expire after a certain amount of time, and so the application needs to be restarted after a certain amount of time or eBay queries will not respond. The ebayDataManager has two functions that allow easy searching and getting items. Below is a code snippet that shows a query in action.

```
let ebayAPITokenManager = EbayAPITokenManager()
let ebayDataManager = EbayDataManager()

ebayAPITokenManager.getOAuth2Token { OAuth2Token in
        guard let token = OAuth2Token else { return }
        self.ebayAPIToken = OAuth2Token
        self.ebayDataManager.searchItem(query: "Yoga Ball",
   OAuth2Token: token) { itemCategory in
            guard let itemCategory = itemCategory else { return }
            self.categoryCellData.append(itemCategory)
            self.collectionView?.reloadData()
       }
       self.ebayDataManager.searchItem(query: "Gym Shoes",
   OAuth2Token: token) { itemCategory in
           ...
       }
       ...
}
```

## 9 Learning New Technologies

### 9.1 How did you learn new technology?

https://www.raywenderlich.com/

https://developer.apple.com/

https://stackoverflow.com/

https://firebase.google.com/docs/

As we used the Swift language, we spent a lot of time on the Apple website reading their documentation. The most helpful resource was their documentation on HealthKit, MapKit, and Swift 3 (the newest version of the language). Ray Wenderlich is one of the primary guide and tutorial creators for iOS development. He had blog posts that provided tons of examples which helped us greatly in learning the basics. Finally, Stack Overflow was able to solve any of our more common problems.

### 9.2 What, if any, reference books really helped?

We didn't use any reference books for iOS development, as the Swift language is fairly new in itself most of the prevalent information was available on the Internet.

### 9.3 Were there any people on campus that were really helpful?

The go-to people for this project were mainly our group mates. Two of us were part of the OSU App club, and so they all had experience with this sort of development before. Other than that

we worked with a team of developers at eBay, our client, so they were our second source of help as they had years of experience. There wasn't really a go to on campus for such specific help.

# 10 What We Learned

## 10.1 Brandon Lee

### 10.1.1 What technical information did you learn?

From this experience, I was able to hone my mobile development skills, specifically in the realm of iOS. I was able to read up on various Cocoa frameworks and tools such as UICollectionViews and eBay's Browse APIs. Utilizing these technical skills, I built various screens including the Store screen and Profile screen. Additionally, I learned Google's Firebase and ways to integrate their services into an iOS project. Because of our client's expressive desire to keep as natively as possible, I developed knowledge in doing things 'the hard way' in iOS including laying out UI with Autolayout and NSAnchors, making network calls with URLSession rather than Alamofire and dealing with JSON natively rather than with SwiftyJSON.

### 10.1.2 What non-technical information did you learn?

I learned a lot about myself and my interests. I find that iOS development is definitely something that I want to remain in as a career field. In terms of work ethic, I find that as with anything I spend the majority of my time with, I occasionally need to take a break from it in order to renew my interest from time to time. All in all, though, I find that I found the field I wish to be in.

### 10.1.3 What have you learned about project work?

I found that project work can be difficult at times when the team is fragmented remotely. Communication is the main issue here and it is absolutely critical that there is some form of scheduled group meeting. This is to ensure that progress is steadily being maintained throughout the terms.

### 10.1.4 What have you learned about project management?

Project management is difficult. There needs to be some form of control between the team leader and the other members. It can be hard to do this when we're all peers as well. I found that meeting face to face on frequent occasions can help in alleviating this struggle and keeping development and progress on track. I also learned that I'd rather be a worker/follower rather than a leader.

### 10.1.5 If you could do it all over, what would you do differently?

If I could do it all over, I'd tell myself to start earlier, such as the Fall term as we spent a lot of time simply writing rather than developing. Additionally, I think I would have made weekly group meetings more mandatory so that all the group members would have the opportunity to touch bases more on a routine basis throughout the year.

## 10.2 Rutger Farry

### 10.2.1 What technical information did you learn?

This year I was provided with the opportunity to refine my iOS development skills and expose myself to various programming paradigms in iOS. Between work, capstone, freelancing, and personal projects, I was simultaneously working on at least three iOS projects during capstone, which actually turned out to be pretty great. Since I had my hands in so many projects, each with a different way of doing things, I was able to take my successes and failures and apply what I learned to C7FIT.

I learned that developing entirely in code is really good for large teams, but oftentimes using Xcode's Interface Builder is a good option for quickly developing complex interfaces, and shouldn't be left out of the iOS developer's toolbox for small projects.

Since we forced ourselves to develop the interface entirely in code though, I discovered that there are actually some great new ways to design with code in iOS that are much less painful than

previous methods. UIStackView, which is basically Flexbox for Cocoa, became my best friend in designing hierarchical interfaces.

### 10.2.2 What non-technical information did you learn?

One of the most important things I learned while working on my capstone project was how to organize my work over time. For most of my high school and university "career", I mostly just did projects the day before they were due, no matter the size. This year, with capstone, organizing a cycling race, and taking a lot of Computer Science classes, I learned that life is a lot less stressful if you prepare for things beforehand.

### 10.2.3 What have you learned about project work?

Once again, staying organized is very important. Splitting up work into small chunks and having frequent check-ins are helpful. We siloed off the app into large chunks and for the most part, didn't mess with each other's code. This helped us avoid merge conflicts but probably resulted in less shared code.

### 10.2.4 What have you learned about project management?

Splitting a project into sub-tasks is sometimes hard before starting work, but an effort should be taken to do this as soon as possible early on in the project. Also laying out a solid foundation and defining idioms is important for maintaining consistency.

### 10.2.5 What have you learned about working in teams?

I've worked in teams a lot in the past so this was not very new. I would say that constant communication is important. Also making regular contributions, even if you are busy and can only make small contributions, shows that you are "in" with the project and are concerned about its success.

### 10.2.6 If you could do it all over, what would you do differently?

I would've done more research into the new tools and services we were going to use – this would've helped us avoid the awkward mid-year transition away from MINDBODY. If we had known we couldn't use MINDBODY sooner we probably would've been able to plan better. I would also have made an effort to reach out to our clients at eBay and Club Seven Fitness more since that would give us a better idea of who our target audience is. When engineers get too out of touch with their clients, their product may make sense to them but can be absurd to normal users. While I don't think this is the case with our application, it would've still been nice to talk more.

Additionally, although the first term was very writing-intensive, I would've like to squeeze a little time into starting development in the first term. It seems like a lot of this class was based on the naive assumption that building a product will be much smoother if extremely detailed plans are written out beforehand. The truth is that many things can only be known after starting to get your hands dirty, and while planning is certainly essential, a lot of the details we were expected to write about were impossible to know before starting to code. Therefore I would've told myself to start development earlier.

## 10.3 Michael Lee

### 10.3.1 What technical information did you learn?

First and foremost, I learned about iOS development and with it comes all of the technologies and development kits that are native to Apple its iPhone products. This primarily includes the Xcode development environment, the Swift language, and HealthKit and MapKit SDKs. For our project, we chose to develop the interface purely in code which was also a learning hurdle for me. Normally for beginners, they start off using Apple's drag and drop interface builders which makes it much easier, but because of the eBay team and our design decisions we opted for the more difficult route: However, from this, I've learned the best design practices for iOS app development from the eBay team and mine.

### 10.3.2 What non-technical information did you learn?

Working long term within the group I've learned the importance of weekly meetings that keep the whole team on track. I'm sure in a working environment when everyone is in the same environment and not split between school teams are more on the same page. This has shown me the importance of communication between teams and how helpful it is to update teammates on your status.

### 10.3.3 What have you learned about project work?

I've learned that not only is good and thorough planning key to a successful project, not everything will always go to plan. For this project we spent a large amount of time planning, a whole term, simply writing documents about what we wanted to do. The whole term almost went to waste as we found out we weren't able to afford the API that our project was centered around. This showed me the importance of flexibility as we needed to

### 10.3.4 What have you learned about project management?

I've learned that to be successful a team needs to be able to respond to major changes quickly and efficiently. Planning is not just a one-time thing, the process of planning begins before the project and continues throughout the development process. As circumstances change you can't just stick to an archaic outdated plan, but instead, you must adapt and change to fit the new circumstances.

### 10.3.5 What have you learned about working in teams?

As I've stated before working within teams on such a large project takes a lot of communication and accountability. It's easy to have tunnel vision when you focus on your portion of the project and it definitely helps to have group mates review and debug your code for the sake of functionality and understandability. As a group member, you need to remain accountable for your portion of the code, as not only does the success of a project as a whole depend on your work, other group mates could depend on your code for their portions. When a group member doesn't complete their portion promptly it can lead to a general mistrust or doubt that one will fulfill their duties, but this can be solved conversely with good communication. Sometimes confrontation is necessary in order to get things done.

### 10.3.6 If you could do it all over, what would you do differently?

If I could restart, I would interact more with my client and the software developers of eBay. The developers were a resource that I should've taken more advantage of, and seeing all of the cool projects that were created at the OSU Engineering Expo, I would've liked to try implementing things more innovative or advanced even if they might not be entirely successful.

I would tell myself all the normal things like start earlier and work continuously throughout the term, but overall I didn't lag too behind. It would have been better for me to start earlier as I had much more to catch up on than my group mates.

# 11 Appendix I: Essential Code

```
//
//  EbayDataManager.swift
//  C7FIT
//
//  Created by Brandon Lee on 2/25/17.
//  Copyright © 2017 Brandon Lee. All rights reserved.
//

import Foundation

struct EbayDataManager {
```

```
// MARK: - Constants

let browseAPIbaseURL = "https://api.ebay.com/buy/browse/v1/"

// MARK: - Network Requests

/**
    Fetches an item based off of its ID.
     - Parameter itemID: ItemID
     - Returns completion: A callback that returns the item JSON
 */
func getItem(itemID: String, OAuth2Token: String, completion: @escaping ([String: Any]?) -> Vo
    let headers = [
        "authorization": OAuth2Token
    ]
    let urlEncodedString = "\(browseAPIbaseURL)item/\(itemID)"
        .addingPercentEncoding(withAllowedCharacters: .urlQueryAllowed)!
    let url = URL(string: urlEncodedString)!
    let request = NSMutableURLRequest(url: url)
    request.httpMethod = "GET"
    request.allHTTPHeaderFields = headers
    let dataTask = URLSession.shared.dataTask(with: request as URLRequest) { (data, _, error)
        guard let data = data, error == nil else {
            print("Error in retrieving item: \(String(describing: error?.localizedDescription)
            return completion(nil)
        }
        guard let dataJSON = try? JSONSerialization.jsonObject(with: data, options: []),
            let dataDict = dataJSON as? [String: Any] else {
            return completion(nil)
        }

        DispatchQueue.main.async {
            completion(dataDict)
        }
    }

    dataTask.resume()
}

/**
    Fetches a list of items based off of a search query with a limit of 10 items.
    - Parameter query: The item search query
    - Returns completion: A callback that returns an EbayItemCategory model
 */
func searchItem(query: String, OAuth2Token: String, completion: @escaping(EbayItemCategory?) -
    let headers = [
        "authorization": OAuth2Token
    ]
    let urlEncodedString = "\(self.browseAPIbaseURL)item_summary/search?q=\(query)&limit=10"
        .addingPercentEncoding(withAllowedCharacters: .urlQueryAllowed)!
    let url = URL(string: urlEncodedString)!
    let request = NSMutableURLRequest(url: url)
    request.httpMethod = "GET"
    request.allHTTPHeaderFields = headers
    let dataTask = URLSession.shared.dataTask(with: request as URLRequest) { (data, _, error)
        guard let data = data, error == nil else {
            print("Error in retrieving item: \(String(describing: error?.localizedDescription)
            return completion(nil)
```

```
            }
            guard let dataJSON = try? JSONSerialization.jsonObject(with: data, options: []),
                let dataDict = dataJSON as? [String: Any] else {
                return completion(nil)
            }

            // Convert raw JSON into respective models, TODO: - Potentially decouple this in the f
            guard let items = dataDict["itemSummaries"] as? [[String: Any]] else { return }

            var itemArray = [EbayItem]()
            for item in items {
                let newItem = EbayItem(itemJSON: item)
                itemArray.append(newItem)
            }

            DispatchQueue.main.async {
                completion(EbayItemCategory(title: query, items: itemArray))
            }
        }

        dataTask.resume()
    }
}

//
//  FirebaseDataManager.swift
//  C7FIT
//
//  Created by Brandon Lee on 1/25/17.
//  Copyright © 2017 Brandon Lee. All rights reserved.
//

import Foundation
import Firebase

/**
    A representation of C7FIT's Firebase services.
 */
struct FirebaseDataManager {

    // MARK: - Constants

    let ref = FIRDatabase.database().reference()

    // MARK: - Static Information

    /**
        Fetch trainers and daily content data for homescreen

         - Returns completion: A callback that returns a FIRDataSnapshot
     */
    func fetchHomeScreenInfo(completion: @escaping (_: FIRDataSnapshot) -> Void) {
        ref.child("homescreen").observeSingleEvent(of: .value) { snapshot in
            completion(snapshot)
        }
    }

    /**
        Fetch club information
```

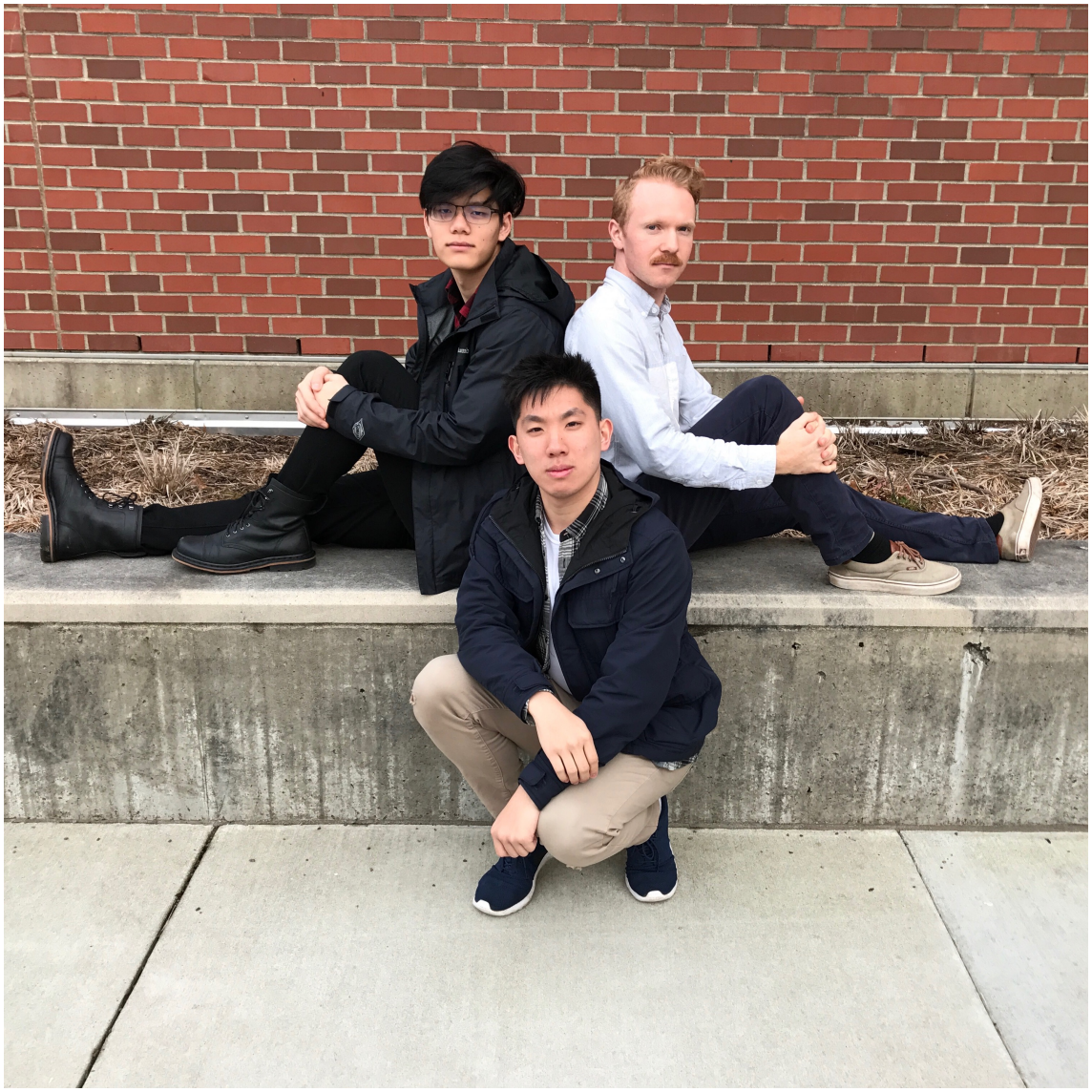
```
    - Returns completion: A callback that returns a FIRDataSnapshot
 */
func fetchClubInfo(completion: @escaping (_: FIRDataSnapshot) -> Void) {
    ref.child("clubInfo").observeSingleEvent(of: .value, with: { snapshot in
        completion(snapshot)
    })
}

// MARK: - User Account Login/Logout

/**
    Create new user account with credentials.

     - Parameter email: User email string
     - Parameter password: User password string
     - Returns completion: A callback that returns FIRAuthCallback
 */
func createAccount(email: String, password: String, completion: @escaping (_: FIRUser?, _:Erro
    FIRAuth.auth()?.createUser(withEmail: email, password: password) { user, error in
        completion(user, error)
    }
}

/**
     Submit login with credentials, display profile screen if valid.

     - Parameter email: User email string
     - Parameter password: User password string
     - Returns completion: A callback that returns FIRAuthCallback
 */
func signIn(email: String, password: String, completion: @escaping (_: FIRUser?, _: Error?) ->
    FIRAuth.auth()?.signIn(withEmail: email, password: password) { user, error in
        print("login screen user: \(String(describing: user?.email))")
        completion(user, error)
    }
}

/**
    Checks if the user is currently logged in.

    - Returns bool: A bool that represents if the user is logged in
 */
func isLoggedInUser() -> Bool {
    return FIRAuth.auth()?.currentUser != nil
}

/**
    Log user out of Firebase account.
 */
func logout() {
    do {
        try FIRAuth.auth()?.signOut()
    } catch let signoutError {
        print("Error signing out: \(signoutError.localizedDescription)")
    }
}
```

```
// MARK: - User Initialization

/**
    Build up a new user profile in Firebase database after account authentication.

    - Parameter uid: User's universal ID
    - Parameter email: User email string
 */
func buildUserProfile(uid: String, email: String) {
    let newUser = User(email: email,
                       photoURL: nil,
                       name: nil,
                       bio: nil,
                       weight: nil,
                       height: nil,
                       bmi: nil,
                       mileTime: nil,
                       pushups: nil,
                       situps: nil,
                       legPress: nil,
                       benchPress: nil,
                       lateralPull: nil)
    let newUserRef = self.ref.child("users").child(uid)
    newUserRef.setValue(newUser.toAnyObject())
}

// MARK: - User State

/**
    Fetch user from the database.

    - Parameter uid: User's universal ID
    - Returns completion: A callback that returns a FIRDataSnapshot
 */
func fetchUser(uid: String, completion: @escaping (_: FIRDataSnapshot) -> Void) {
    ref.child("users").child(uid).observeSingleEvent(of: .value, with: { snapshot in
        completion(snapshot)
    })
}

/**
 fetch user runs
*/
func fetchUserRun(uid: String, runTitle: String, completion: @escaping (_: FIRDataSnapshot) ->
    ref.child("userRun").child(uid).child(runTitle).observeSingleEvent(of: .value, with: { sna
        completion(snapshot)
    })
}

/**
 fetch list of user runs
*/
func fetchUserRunList(uid: String, completion: @escaping (_: FIRDataSnapshot) -> Void) {
    ref.child("userRun").child(uid).observeSingleEvent(of: .value, with: { snapshot in
        completion(snapshot)
    })
}
/**
```

```
    Monitor the login state of the user.

    - Returns completion: A callback that returns FIRAuthStateDidChangeListenerHandle
 */
func monitorLoginState(completion: @escaping (_: FIRAuth, _: FIRUser?) -> Void) {
    FIRAuth.auth()?.addStateDidChangeListener { auth, user in
        completion(auth, user)
    }
}

// MARK: - Data Modification

/**
    Updates any new attributes of a given existing user

    - Parameter uid: User's universal ID
    - Parameter user: User data to update
 */
func updateUser(uid: String, user: User) {
    let newUserRef = self.ref.child("users").child(uid)
    newUserRef.setValue(user.toAnyObject())
}

/**
    Updates any new attributes of a given user's run
*/
func updateUserRun(uid: String, runTitle: String, userRun: RunData) {
    let newUserRun = self.ref.child("userRun").child(uid).child(runTitle)
    newUserRun.setValue(userRun.toAnyObject())
}

/**
    Uploads new profile picture to Firebase Storage

    - Parameter uid: User's universal ID
    - Parameter data: Image data to upload
    - Returns completion: A callback that returns a URL?
 */
func uploadProfilePicture(uid: String, data: Data, completion: @escaping (_: URL?) -> Void) {
    let storageRef = FIRStorage.storage().reference(withPath: "profilePics/\(uid).jpg")
    let uploadMetaData = FIRStorageMetadata()
    uploadMetaData.contentType = "image/jpeg"
    storageRef.put(data, metadata: uploadMetaData) { (metaData, error) in
        if error == nil {
            // Update user profilePicURL
            completion(metaData?.downloadURL())
        } else {
            print("Upload profile pic error: \(String(describing: error?.localizedDescription)
            completion(nil)
        }
    }
}

/**
    Helper function to build runData from JSON, after retrived from fireBase Db
*/
func buildRunFromJson(json: [String: AnyObject]) -> RunData? {
    guard let runTitle = json["runTitle"] as? String,
```

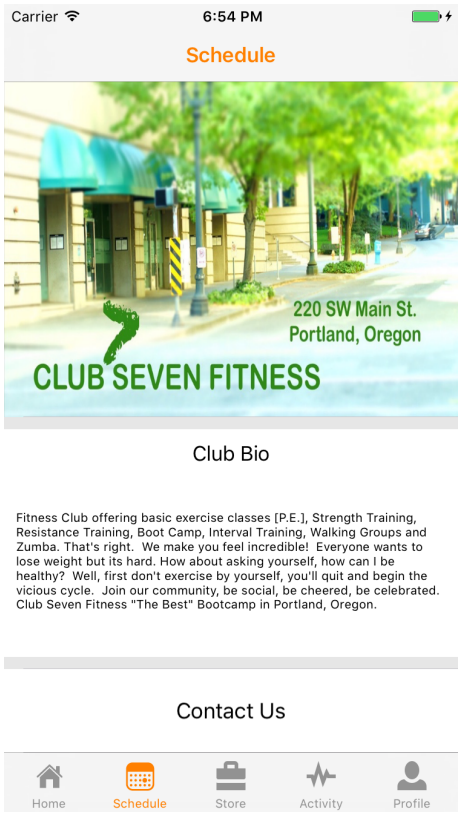
```
            let time = ((json["time"] as? Double)),
            let distance = ((json["distance"] as? Double)),
            let pace = json["pace"] as? String,
            let locationsString = json["locations"] as? [AnyObject],
            let dateDouble = json["date"] as? Double else { return nil }

        var convertedLoc = [Location]()
        for locString in locationsString {
            if let tempLocation = self.buildLocFromJson(json: locString as! [String : AnyObject])
                convertedLoc.append(tempLocation)
            }
        }

        let date = Date(timeIntervalSince1970: dateDouble)

        return RunData(runTitle: runTitle, time: time, distance: distance, pace: pace, locations:
    }

    /**
        Helper function to build location data from JSON, after retrieved from fireBase Db
    */
    func buildLocFromJson(json: [String: AnyObject]) -> Location? {
        guard let timestamp = json["timestamp"] as? String,
            let latitude = (json["latitude"] as? Double),
            let longitude = (json["longitude"] as? Double) else { return nil }
        return Location(timestamp: timestamp, latitude: latitude, longitude: longitude)
    }

}
```

## 12 Appendix II: Pictures and more

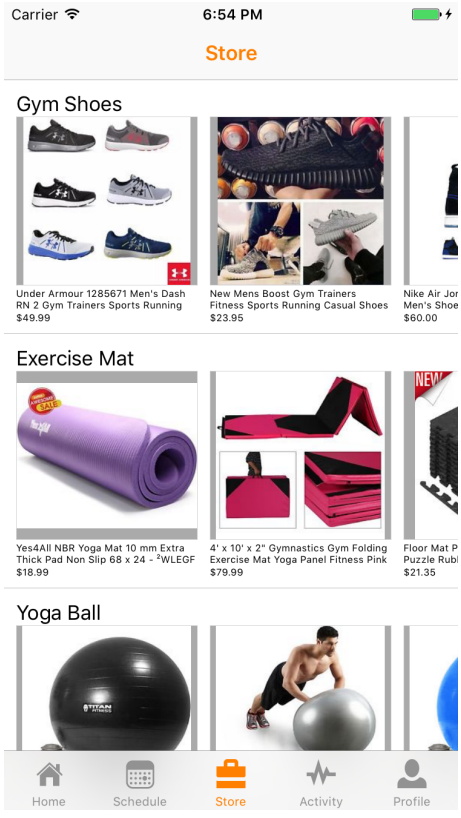


The C7FIT group photo

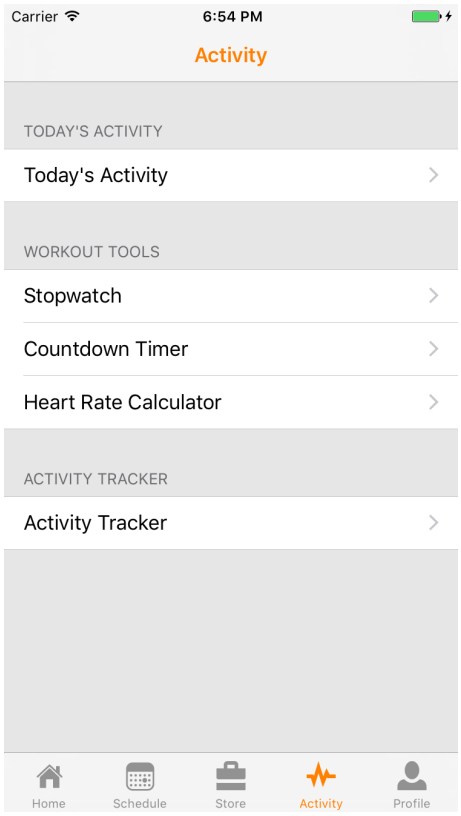
h



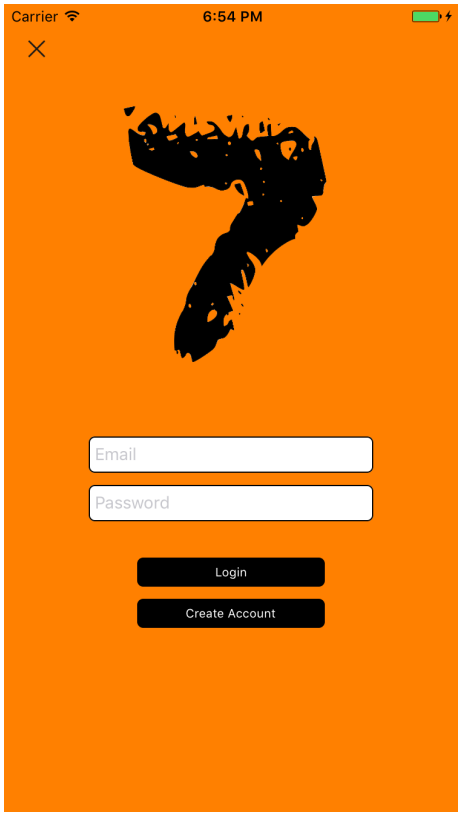

Schedule Tab




Store Tab




Activity Tab




Profile Tab