# Winter Midterm Progress Report

Brandon Lee, Rutger Farry, Michael Lee

February 18, 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.

# 1 Introduction/Recap

As a quick 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 application's purpose is to allow customers of a gym to easily be able to access their heath 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.

# 2 Brandon Lee

## 2.1 Current Progress

As we delegated through various agile methods such as kan-ban boards, sprint meetings, and etcetera, we've decided to delegate the work based off of the tabbed views in the iOS application. Thus, I've been put in charge of the Store tab and the Profile tab. As of right now I have the profile tab functionally sufficient for a minimum viable product (MVP). The profile tab is essentially a profile page where a user is able to input various forms of health data including their name, bio, profile picture, height, weight, repetition records of various workout routines, mile time, and maxes of various workout routines. Essentially, after much trial and error in implementing our application, I was able to hook up a Firebase backend with the client side code to enable many core features such as a login/auth system, account creation, profile fetching and perform batch updating. In pursuit of this, I developed various classes such as a Firebase Data Manager, User model, and the respective profile views and view controllers. I developed the respective table view cell classes, setup Firebase on the server side, and developed the schema in which users would be interpreted. All of the following work has been merged from their respective feature branches, into our main development branch.

## 2.2 Remaining Work

There is much remaining work to be done both on my end and in my groupmates' ends. Firstly, I have yet to develop a minimum viable product for the store tab. There is much development to be done in order to implement it. The store tab is essentially a tab where users can purchase various types of workout equipment from yoga balls to dumbbells. The store tab is geared towards using the eBay APIs and should deep link the listings on eBay. In order to implement this, much work needs to be done on developing request, response, and data manager classes for the eBay API as well as developing the client side code such as the view controllers, models, and views for the various listings we will support. These items and listings will be curated upon request of the client. In addition to my development of the Store and Profile tabs, I will need to work on writing tests in these sections to ensure code quality and easy code maintenance for potential future developers. In respect to my groupmates' remaining workloads, we have a minimum viable product of the Schedule tab, which simply links to Club Seven Fitness' online schedule as well as their contact info. Unit and integration tests should need to be developed on these sections as well.

## 2.3 Impeding Problems

We faced some problems throughout the past few months. Firstly, one of the biggest problems we ran into being the client unable to support the initial integration of MindBody, their adopted enterprise solution for scheduling individual training and classes. This resulted in us having to revert much of our already developed work and restart afresh. Additionally, there was much annoyance in moving over and setting up Firebase as Google (owners of Firebase) only allow for integration of their backend solutions in a very architecturally intrusive manner. As a result, much of the Firebase data manager code and Profile view controller are full with Firebase data flow and design patterns. Outside of coding and development, there have been a few problems on slow communication and collaboration between group mates and clients. This is mainly due to everyone's conflicting busy schedules and distance between the client and the developers.

## 2.4 Interesting Pieces of Code

An interesting piece of code I developed in the Profile tab is where I had to apply the adapter pattern in order to bridge the interface between Apple's UIPickerView and UITableViewCell. This allows for custom UIPickerViews in the various cells in a table view. To my surprise, this was not as easy to implement as I had initially perceived, as I had assumed UITableView wouldn't need an adapter pattern to have multiple custom picker views.

```
// MARK: - UIPickerViewDelegate

extension AbstractHealthCell: UIPickerViewDelegate {
    public func pickerView(_ pickerView: UIPickerView, titleForRow row: Int, forComponent componen
        return delegate?.pickerView(pickerView, titleForRow: row, forComponent: component, forCell
    }

    public func pickerView(_ pickerView: UIPickerView, didSelectRow row: Int, inComponent componen
        delegate?.pickerView(pickerView, didSelectRow: row, inComponent: component, forCell: self)
    }
}

// MARK: - UIPickerViewDataSource

extension AbstractHealthCell: UIPickerViewDataSource {
    public func numberOfComponents(in pickerView: UIPickerView) -> Int {
        return dataSource!.numberOfComponents(in: pickerView, forCell: self)
    }

    public func pickerView(_ pickerView: UIPickerView, numberOfRowsInComponent component: Int) ->
        return dataSource!.pickerView(pickerView, numberOfRowsInComponent: component, forCell: sel
    }
}

// MARK: - PickerCell Protocols

/// UIPickerView Hook Delegate
public protocol PickerCellDelegate: class {
    func pickerView(_ pickerView: UIPickerView, titleForRow row: Int, forComponent component: Int,

    func pickerView(_ pickerView: UIPickerView, didSelectRow row: Int, inComponent component: Int,

    /**
        Called on picker view open
        - Parameter cell: Target
     */
    func onPickerOpen(_ cell: AbstractHealthCell)

    /**
        Called on picker view close
        - Parameter cell: Target
     */
    func onPickerClose(_ cell: AbstractHealthCell)
}

/// UIPickerView Hook DataSource
public protocol PickerCellDataSource: class {
    func numberOfComponents(in pickerView: UIPickerView, forCell cell: AbstractHealthCell) -> Int
    func pickerView(_ pickerView: UIPickerView, numberOfRowsInComponent component: Int, forCell: A
}
```

### 2.5 Experimental Design

There were a few attempts at building the user interface for the profile view. Initially, there was an attempt to conform with Firebase's data flow paradigm. The client implementation was through using UITextFields and UITextViews to update per character change input. For example, when the user starts typing, the query is sent to Firebase as an incomplete input that continues to update per character stroke. This results in multiple queries being made, which would raise the price ceiling due to the amount of requests made per user field input. A solution to this was to optimize the Firebase request by pushing the save down the query through implementing a save button, which would build a request based off of the existing state changes, and perform a batch update.

### 2.6 Interface Design

The user interface is designed in respect to ease of accessibility to the user. While our alpha build is still a bit blocky, the tabbed design to navigate between the different views proved to be a successful approach towards intuitive user navigation. Additionally, putting most of our views into custom table view cells and table views has allowed for users to be able to find the data or sections they are looking for intuitively.

## 3 Rutger Farry

### 3.1 Current Progress

As someone who's been designing and programming iOS apps for a while now (4ish years), I've got a lot of prejudices and opinions on the best way to do things. The development studio I work for recently began work on its first iOS application. Since I'm the most experienced iOS developer there, I've taken somewhat of a project manager role, designing the structure of the application and laying out fail safes, such as required code reviews and continuous integration. I've been applying the lessons I've learned at my workplace to our senior project and have taken a bit of a similar role. While I have done some work on views and backend code, most of my efforts have been focused on designing the app structure and implementing continuous integration/deployment, style guidelines, code reviews, etc.

One of the coolest tools I've come across that allows us to automate testing and continuous deployment is called Fastlane. Fastlane is a task runner built in Ruby for iOS development that basically connects a series of tools together in tasks called lanes. After proper setup, even the most incredibly tedious multistep processes, such as deploying an app to the Apple App Store, can be completed in a few minutes with a single command. We are currently using Fastlane to test, lint, and generally ensure good "code smell," though when we get closer to deployment we intend to use it for deploying to test devices and even the App Store.

I've also taken advantage of two other tools that really speed up our development time and code quality. The first is TravisCI, a continuous integration platform that should be relatively familiar to most developers, especially in the open-source world. TravisCI runs our tests on remote servers to ensure we are writing portable, workable code. The second is SwiftLint, which powers our listing efforts. Linting is the process of statically analyzing code to ensure consistent style and best practices. For example, in Swift, there are different ways to write opening brackets to a function:

```
func getGoodGrades(useBribes: Bool) {

}
```

or we can do:

```
func getGoodGrades(useBribes: Bool)
{

}
```

Both ways are syntactically correct and compile fine. But considering the amount of time developers spend trying to figure out other people's code, the least we can do is ensure it is formatted consistently. Therefore, we use SwiftLint to complain at developers when they write code that's hard to read.

Additionally, I've done work on the data flow design for the application. We're taking a Redux-style approach to how data moves throughout our app, meaning there is a single root (we call it a data store) where the most important raw data for the application is placed. This is then passed to the main view controllers, which tell their descendant view controllers what they need to know by performing transformations on the central data store, creating a tree-like data flow

Using this tree-like data flow grants us some great advantages. The first and foremost advantage is that it is easy to know where views are getting their data from. Large view controllers that need access to vast swaths of information are given access to the entire data store, and minor views are just given subsets that might've had a transformation applied by their parent. All this information is traceable upwards, though, making it easy to hunt down bugs and just generally understand the application.

Another cool advantage is that since the most important parts of the application's state are stored in a single class, the application's state can be easily preserved and restored. Instead of searching across the app's view controllers for important information, we just preserve the central data store to a text file on app shutdown and then restore it when the app is started again.

I've also done work on the "Home" view controller, which the user sees when the open the application. This is currently just a simple table view containing three rows: a motivational or instructional YouTube video that the coach has chosen to show to their clients that day, a motivational quote that the coach has read out and recorded, and a list of trainers that work at Club Seven Fitness along with a picture and quick bio.

The YouTube video display is pretty interesting. Since YouTube is very protective of the video uploaded to their website, they don't provide a method of obtaining an HTTP video stream from their website. The only way to "stream" a YouTube video legally is by embedding their JavaScript video player inside a web browser in your application. Luckily, it's possible to pare down the player to remove the interface and only display video, allowing us to implement video controls using native elements.

## 3.2 Remaining Work

In taking our app to the alpha stage, we've eschewed a few of my ideas on separation of concerns and data flow. Several view controllers are holding their own state and information we fetch from Firebase is bypassing the central data store. Since the app's codebase is still pretty small, it's still pretty understandable, but I'd like to refactor the data flow before the app grows too much.

The application is still pretty ugly. We're just using Apple's native UIKit elements with a few colors here and there. We did create a bunch of wireframes for every screen in the Fall, but the app has changed a little since then. I foresee the need to revise those wireframes soon and ensure we are putting out a beautiful app that people will want to download and use.

## 3.3 Impeding Problems

We discovered in December that the wellness business management API we were intending to use, Mindbody, is prohibitively expensive for a small business such as Club Seven Fitness and decided to drop the use of that API. This required us to rethink a lot of the application since most of it was focused around that API.

I think the chances of us having another revelation like that is unlikely, however. Considering we don't have a similar revelation, we are not currently facing any difficult problems and I predict we will easily finish and deploy the application in time for the OSU Engineering Expo.

## 3.4 Experimental Design

We're planning on using RxSwift, a reactive functional library, for making the UI automatically react to changes the in the app's state. Reactive functional programming can be a deep hole, though, and we've experimented with making a few sample apps with primarily functional programming instead of imperative programming. These apps end up looking pretty cool and work,

but are hard to debug for programmers at our experience level and also hard to understand for people unfamiliar with functional programming.

Therefore, we've decided to stick with using RxSwift just to call functions in response to changes in app state instead of piping everything up in a functional programming way.

### 3.5 Summary

Overall I think we've done a good job so far and haven't made any serious mistakes. I'm looking forward to being totally feature-complete and being able to begin focusing on refining the design, performing user studies and refactoring the data flow to be super readable and understandable.

## 4 Michael Lee

### 4.1 Current Progress

The purpose of our project is to create an iOS application for the gym Club Seven Fitness. This application will be targeted towards the members of the gym and allow them to construct personal profiles that track their fitness goals and serve as way for the member to access the gym through their phone. More specifically, the application will be the starting hub from which they can access daily fitness videos, gym schedules, and activity trackers. The current goal of the project is to develop a working minimum viable product from the sketches we've been given, that can be shown to the gym owner and we can further decide on the direction of the application. The project purpose used to be to incorporate the Mind Body API to allow users ability to schedule classes and further interact with the gym, but due to financial constraints we've switched to using a cheap database/user base tool called Firebase that will give the app more functionality. The final goal of this project is to have a fully functioning application that can be uploaded to the Apple Application Store.

### 4.2 Remaining Work

The work on our project has been divided among our group members based on the Main screens that will be in the final application. These screens are the Main, schedule, activity, profile, and shop screens. The application has more parts like user authentication, storage, and the Ebay API, but for the minimum viable product we have focused on the screen distribution. I have been working on the schedule screen and the activity tracker screen. The schedule screen is essentially complete as it just consists of links and methods to contact the gym. The next screen I will be working on is the Activity screen which will be more substantial. It will use a number of tools already present like the countdown and stopwatch features, and it will interact with other parts of the phone like HealthKit and MapKit. After this is finished my main work on the screens will be done, the rest of the work will likely remain in the store functionality.

### 4.3 Impeding Problems

The problem that has impeded my progress is mainly my inexperience in iOS development; however, with the help from my group members, my own research, and looking through their code has helped me develop faster. In general, our project and gym ran into financial problems as they were unable to purchase the API that our project was based on. Overall, this may have been beneficial to our work as the API used outdated technology and the removal of the Mind Body API has made our work easier. This change does give us substantial restructuring to do on our previous documentation from Fall term, however we will primarily just be shifting focus toward the eBay shop API.

### 4.4 Interesting Pieces of Code

A particularly interesting piece of code came from developing the activity view. In this view, I needed to gather information from the phone's native HealthKit application. To manage this information I had to develop a data manager class of my own called healthKitManager. Within this function is the code below that is native to the Apple process: to access any of the user's information we must first authorize each piece of HealthKit data. You can see how I select each

piece of information that I want read and write permissions for and then I finally request at the bottom. This is interesting because Apple streamlines this process and it gives the user complete power over their own data, which is vital for the usage

```
func authorizeHealthKit() {
    if HKHealthStore.isHealthDataAvailable() {
        let healthKitRead = Set([
            HKObjectType.characteristicType(forIdentifier: HKCharacteristicTypeIdentifier.biol
            HKObjectType.characteristicType(forIdentifier: HKCharacteristicTypeIdentifier.date
            HKObjectType.characteristicType(forIdentifier: HKCharacteristicTypeIdentifier.bloo
            HKObjectType.quantityType(forIdentifier: HKQuantityTypeIdentifier.bodyMassIndex)!,
            HKObjectType.quantityType(forIdentifier: HKQuantityTypeIdentifier.height)!,
            HKObjectType.quantityType(forIdentifier: HKQuantityTypeIdentifier.bodyMass)!,
            HKObjectType.quantityType(forIdentifier: HKQuantityTypeIdentifier.bodyFatPercentag
            HKObjectType.quantityType(forIdentifier: HKQuantityTypeIdentifier.heartRate)!,
            HKObjectType.quantityType(forIdentifier: HKQuantityTypeIdentifier.distanceWalkingR
            HKObjectType.workoutType()
        ])

        let healthKitWrite = Set([
            HKObjectType.quantityType(forIdentifier: HKQuantityTypeIdentifier.bodyMassIndex)!
        ])

        healthKitStore.requestAuthorization(toShare: healthKitWrite, read: healthKitRead) { (s
            if (success == false) {
                print("error requesting authorization")
            }
        }

    } else {
       return
    }

}
```

Another Interesting piece of code is under the schedule view where the user has the ability to call the gym. The calling function is simple, but it links to a the telephone application with a simple URL. The "tel://" phone string tells phone which shared UIApplication we want to use and it calls the phone number "229929292".

```
func callGym() {
     let gymPhone = "229929292"
     let phoneString = "tel://" + gymPhone
     let url = URL(string:phoneString)!
     UIApplication.shared.open(url, options: [:], completionHandler: nil)
 }
```

## 4.5 Experimental Design

The completed design for the schedule view went through a couple phases to get where it was. At first it started out as three equal sized table cells that contained the separate UIViews, but because the contact cell and the picture was large it looked disproportionate with regards to the functionality that each element offers. Instead I shrunk the contact to a slimmer button at the bottom of the view and made the schedule link with picture take up most of the screen. We don't have any details on the club bio as of yet so I filled it in with some placeholder text and ample room to for any message the club wants to place. There is a lot of unused space on this screen so the contact button can take the place of what could be other real estate. For the future if we want to add anything, that button could be hidden and only appear when swiped.

## 4.6 Interface Design

The user interface is designed to be easy for the user to navigate, so we've separated everything into distinct sections. We have a lot of data to pull from separate sources like HealthKit and Firebase, so we're working on deciding how to integrate these two parts together. Where should data go and if it should have repetition or not. Other than that the majority of my cells are structure simply as a table view with table cells that, when clicked, lead to other expanded sub-view cells. This structure is very basic and conforms to the standard look and feel of other similar applications. In the final applicationn we may add more customization to give the gym a unique look and feel.