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