iOS Handbook

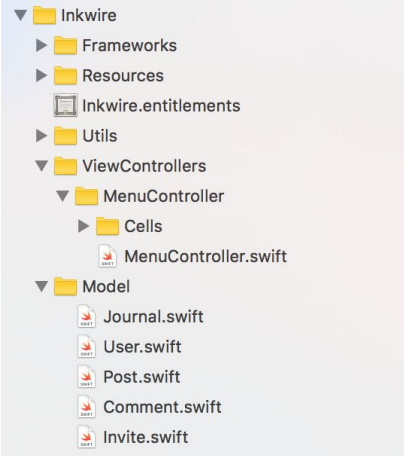
Overview

Software engineering is absolutely essential in any software product you make. A badly engineered product can have adverse effects on the end user experience and the engineers working on it. Your apps have to be beautiful, both on the outside and the inside. You want to engineer your product in a way that (1) reduces latency, (2) minimizes use of resources and (3) allows engineers to easily build on top of it in the future. This section covers some of the most important software engineering practices in iOS development. You should know these like the back of your hand and use them in every app you make.

Project Structure

You should do your best to have a clean project structure. In MDB, we encourage teams to connect blank ViewControllers on their storyboards with segues, and then do everything programmatically. This way, the storyboard serves as a high level visualization of the flow of your app but all the ViewControllers are empty in the storyboard. One person on your team should make this Storyboard before starting the project after planning out the whole frontend architecture of your app. This system allows for easy collaboration since storyboard modifications can cause merge issues.

Your directory structure should look similar to this. We have a frameworks folder to store lightweight libraries that you haven't integrated via cocoapods. The resources folder is for things such as fonts that you've added to your project.



MVC (Model View Controller)

Brief Explanation - The model represents your business logic and creates an abstraction for your backend. Views are how information is displayed to the user (UI logic). The controller communicates with the model and the views but models and views should not communicate with each other.

MVC is the most important concept when it comes to software architecture for iOS development. Typically when beginning to write an app, engineers will create a model first and test it with the database before writing any other code. If you need help understanding MVC, ask one of our instructors or senior devs to setup a private session with you ASAP. Please also refer to this raywenderlich tutorial.

Structs

Brief Explanation - any object that a class might be useful for, but does not need to be inherited. Classes have state and behavior. Structs have state but not behavior.

Example - assuming you have a constants struct as shown below, in all the places in your entire codebase where you would be writing UIColor(red:0.93, green:0.95, blue:0.96, alpha:1.0), you could just write Constants.backgroundColor instead. Why is this helpful? (1) it cleans up your code a lot and (2) let's say you want to change your app color; now instead of changing it in many places in your code, you just have to change it in the constants struct. Having a Constants struct (or multiple constants structs) is a common software engineering practice that you should take note of.

```
struct Constants {
    static let backgroundColor = UIColor(red:0.93, green:0.95, blue:0.96, alpha:1.0)
}
```

Enums

Explanation - defines a common type for a group of related values and enables you to work with those values in a type-safe way within your code.

If you need help understanding these, please refer to the apple documentation.

Closures

Brief Explanation - closures are self-contained blocks of functionality that can be passed around and used in your code. Closures are everywhere in Swift; they have many use cases! If you don't understand these, please setup a private session with our iOS instructor or our iOS senior devs.

Example - Consider the two code snippets below. They retrieve a user from the database and construct a local user object using the retrieved data. You probably have many similar Firebase calls in your code already that use .observe to get info from the database. The method signature for observe is: observe(FIRDataEventType, (FIRDataSnapshot) -> Void). The first argument it takes is a value from an enum. The second argument it takes is a closure! This may be hard to tell in snippet 1, but snippet 2 makes it easier to identify. In snippet 2, block is a closure. Snippet 1 and 2 are equivalent.

Snippet 1

```
let dbRef = FIRDatabase.database().reference()
let userId = "1203980128301238901293 80"
dbRef.child("Users/\(userId)").observe(.value, with: { snapshot -> Void in
    if snapshot.exists() {
        if let userDict = snapshot.value as? [String: AnyObject] {
            let retrievedUser = User(key: snapshot.key, userDict: userDict)
            print("The retrieved user is \(retrievedUser.name!)")
        }
    }
})
```

Snippet 2

```
let dbRef = FIRDatabase.database().reference()
let userId = "1203980128301238901293 80"

let block: ((FIRDataSnapshot) -> Void) = { snapshot -> Void in
    if snapshot.exists() {
        if let userDict = snapshot.value as? [String: AnyObject] {
            let retrievedUser = User(key: snapshot.key, userDict: userDict)
            print("The retrieved user is \(retrievedUser.name!)")
        }
    }
}

dbRef.child("Users/\(userId)").observe(.value, with: block)
```

Protocols/Delegates

Explanation - A protocol defines a blueprint of methods, properties, and other requirements that suit a particular task or piece of functionality. Delegation is a design pattern that enables a class or structure to hand off (or delegate) some of its responsibilities to an instance of another type.

Understanding protocols and delegates is really important because they help you communicate between different classes. Protocol oriented programming is one of the things that makes Swift so elegant. If you need help understanding how to use these, please refer to this article.

GCD

Explanation - Grand Central Dispatch is an easy higher level concurrency model to avoid synchronization issues in your code. If you are using Firebase or any BaaS, you will need to use this! If you don't understand this, please watch the MDB video tutorial.

Example - note the usage of DispatchQueue.main.async below and read the comment.

```
let dbRef = FIRDatabase.database().reference()
let postId = "adf234afadf"

dbRef.child("Posts/\(postId)").observe(.value, with: { snapshot -> Void in
    if snapshot.exists() {
        if let postDict = snapshot.value as? [String: AnyObject] {
            let retrievedPost = Post(key: snapshot.key, postDict: postDict)

            //We retrieved the post JSON from the database and created a local Post object out of
                it. Now, we can add it to our collectionview which displays a feed of posts. This
                needs to be done on the main thread and inside the callBack for the API call! The
                API call is async though and usually runs on the background thread. So how do we
                dispatch UI work to the main thread from within the callBack of an async call? We
                use GCD! This is basically the DispatchQueue.main.async line below.

            self.posts.append(retrievedPost)
            var indexPaths = Array<IndexPath>()
            let index = self.posts.index(where: {$0.postId == retrievedPost.postId})
            let indexPath = IndexPath(item: index!, section: 0)
            indexPaths.append(indexPath)

            DispatchQueue.main.async {
                self.collectionView.performBatchUpdates({Void in
                    self.collectionView.insertItems(at: indexPaths)
                    self.numPosts += 5
                    }, completion: nil)
            }


        }
    }
})
```

Utils

A common software engineering practice is to create one or more utility classes for commonly used methods. This cleans up your code base and prevents redundant code.

Example - Suppose you were making a money app, and in 20 places in your codebase, you needed to get the currency string representation of a double so that you could display it properly to the user in a label. You could create a method in your utility class (as shown below) and just call MoneyAppUtils.doubleToCurrencyString(value) wherever you need to do so.

```
class MoneyAppUtils {
    static func doubleToCurrencyString(val: Double) -> String {
        let formatter = NSNumberFormatter()
        formatter.numberStyle = NSNumberFormatterStyle.CurrencyStyle
        formatter.locale = NSLocale(localeIdentifier: "en_US")
        return formatter.stringFromNumber(val)!
    }
}
```

Caching & LocalStore

Network calls are expensive. You don't want to make more of them than you absolutely need to. You should implement caching in your projects and also LocalStore classes. Caching will save the most frequently accessed data in memory and evict objects using the LRU model (if you use the HanekeSwift cocoapod for caching). Caching is best for things such as images or objects in your feed. However, sometimes you may have data about the user that will rarely change and is needed to make other queries. Instead of querying the database to retrieve this data first each

time, you can implement a LocalStore class that has methods for you to easily save and access data from UserDefaults.

Style

As you've probably already noticed, swift is a very beautiful language! It's powerful and clean at the same time. It's important that you make sure your code is beautiful too! Please refer to this style guide: **https://github.com/raywenderlich/swift-style-guide** and refactor your code accordingly. The guide is long, so please feel free to skim. The most important sections for you to read are Naming, Use of Self, and Protocol Conformance.

Documentation

It's very important to document your code so that other developers can later understand it. If you don't know how to propely document your code, please check out this article. Once you've documented your code, you can easily produce Apple docs style html files using a library called Jazzy.

Design Resources

- https://dribbble.com/tags/ios
- https://www.invisionapp.com/tethr
- https://www.google.com/webhp?sourceid=chrome-instant&ion=1&ie=UTF-8&rct=j#q=sketchappresources
- https://designcode.io/sketch
- https://www.sketchapp.com/extensions/plugins/
- http://www.mobile-patterns.com/

Go-to libraries

- https://github.com/Alamofire/Alamofire
- https://github.com/JonasGessner/JGProgressHUD
- https://github.com/Haneke/HanekeSwift
- https://realm.io/docs/swift/latest/
- https://github.com/John-Lluch/SWRevealViewController
- https://github.com/Yalantis/Koloda
- https://github.com/hyperoslo/ImagePicker
- https://github.com/MengTo/Spring
- https://github.com/ViccAlexander/Chameleon
- https://github.com/maxep/MXParallaxHeader
- https://github.com/Krelborn/KILabel
- https://github.com/realm/jazzy
- https://github.com/SwiftyJSON/SwiftyJSON