CODE STANDARDS

iOS Development

Table of Contents

1. Project Structure

1.1. Structure

- To keep all those hundreds of source files from ending up in the same directory, it's a good idea to set up some folder structure depending on your architecture. For instance, you can use the following:

```
├─ Models
├─ Views
├─ Controllers
├─ Resources
├─ Frameworks

├─ Assets
```

- First, create them as groups (little yellow "folders") within the group with your project's name in Xcode Project Navigator. Then, for each of the groups, link them to an actual directory in your project path by opening their File Inspector on the right, hitting the little gray folder icon, and creating a new subfolder with the name of the group in your project directory.

- **Models:**
  - The Model encapsulates a particular set of data, and contains logic to manipulate that data. When you think about accounting software, an `Invoice` is a model. When you think of a Twitter app, a `Tweet` is a model.

- **Views:**
  - Instead of manipulating view frames directly, you should use size classes and Autolayout to declare constraints on your views. The system will then calculate the appropriate frames based on these rules, and re-evaluate them when the environment changes.

- **Controllers:**
  - Use dependency injection, i.e. pass any required objects in as parameters, instead of keeping all state around in singletons. The latter is okay only if the state *really* is global.

```
let fooViewController = FooViewController(withViewModel: fooViewModel)
```

  - Try to avoid bloating your view controllers with logic that can safely reside in other places.

- **Resources**:
  - Some libraries which needs to be added and not supported by cocoapods or carthage for dependency management will be included into resources folder.

- **Assets**:
  - Asset catalogs are the best way to manage all your project's visual assets. They can hold both universal and device-specific (iPhone 4-inch, iPhone Retina, iPad, etc.) assets and will automatically serve the correct ones for a given name.

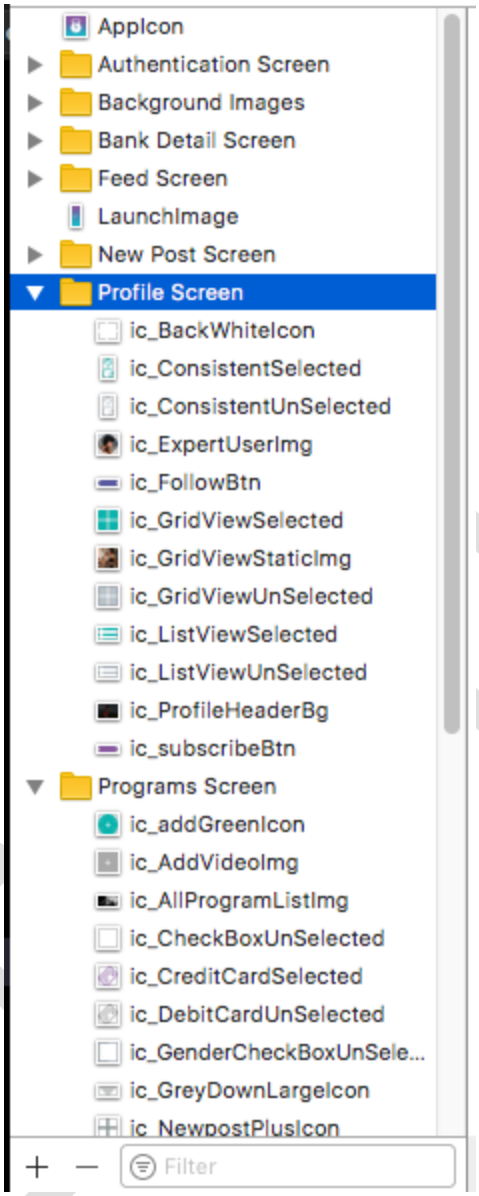
AppIcon
Authentication Screen
Background Images
Bank Detail Screen
Feed Screen
LaunchImage
New Post Screen
Profile Screen
ic_BackWhiteIcon
ic_ConsistentSelected
ic_ConsistentUnSelected
ic_ExpertUserImg
ic_FollowBtn
ic_GridViewSelected
ic_GridViewStaticImg
ic_GridViewUnSelected
ic_ListViewSelected
ic_ListViewUnSelected
ic_ProfileHeaderBg
ic_subscribeBtn
Programs Screen
ic_addGreenIcon
ic_AddVideoImg
ic_AllProgramListImg
ic_CheckBoxUnSelected
ic_CreditCardSelected
ic_DebitCardUnSelected
ic_GenderCheckBoxUnSele...
ic_GreyDownLargeIcon
ic_NewpostPlusIcon
Filter

Figure 1.1.1

1.2. Module Structure:

- The Module name should be descriptive enough to understand the functionality of module. It should be unique identifier for project itself. The image below explains how the modules within the project will be visible in Xcode.

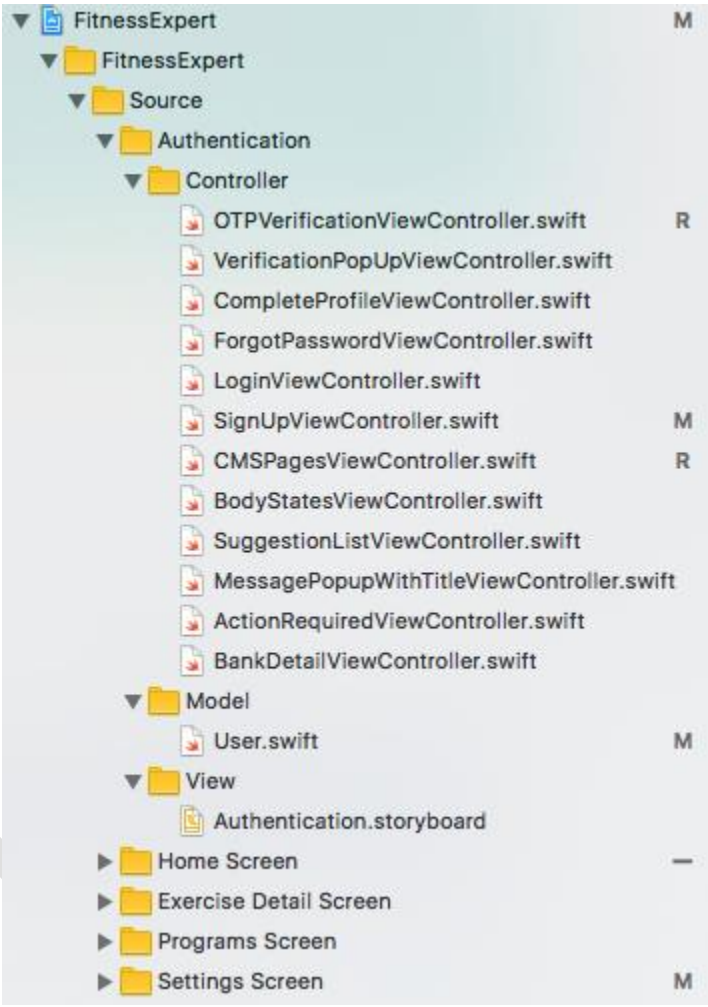

Figure 1.2.1

2. Naming Rules

2.1. Class files

- Class names are written in UpperCamelCase.
- For classes that extend an Android component, the name of the class should end with the name of the component.
- For Example: UserModel, UserPosts, ImageUploaderService.
- Do not add a class prefix. Swift types are automatically namespaced by the module that contains them. If two names from different modules collide you can disambiguate by prefixing the type name with the module name.

```
// SomeModule.swift
public class UsefulClass {
    public class func helloWorld() {
        print("helloWorld from SomeModule")
    }
}

// MyApp.Swift
class UsefulClass {
    class func helloWorld() {
        print("helloWorld from MyApp")
    }
}

import SomeModule

let appClass = UsefulClass.helloWorld()
let moduleClass = SomeModule.UsefulClass.helloWorld()
```

Figure 2.1.1

2.2. Image files

- Image file names are written in lowercase_underscore.

2.3. UI Controls

- Naming conventions for control:

Control	Prefix	Example
Text Field	txt	txtEmail
Button	btn	btnSend
Label	lbl	lblTitle
Image View	img	imgProfile
Table View	tbl	tblUsers
Picker	picker	pickerDOB
Scroll View	scroll	scrollLogin
View	view	viewContainer

Figure 2.3.1

- Naming conventions for icons:

Asset Type	**Prefix**	**Example**
Icons	ic_	ic_star.png
Menu icons and Action Bar icons	ic_menu	ic_menu_archive.png
Status bar icons	ic_stat_notify	ic_stat_notify_msg.png
Tab icons	ic_tab	ic_tab_recent.png
Dialog icons	ic_dialog	ic_dialog_info.png

Figure 2.3.2

- Naming conventions for selector states:

State	**Suffix**	**Example**
Normal	_normal	btn_order_normal.png
Pressed	_pressed	btn_order_pressed.png
Focused	_focused	btn_order_focused.png

Disabled	_disabled	btn_order_disabled.png
Selected	_selected	btn_order_selected.png

Figure 2.3.3

2.4. View Controller files

- View Controller files should match the name of the iOS components that they are intended for but moving the top level component name to the beginning. For example, if we are creating a layout for the SignIn, the name of the view controller file should SignInViewController.
- Class names must be descriptive. In particular (don't forget to add a three-letter prefix for Objective-C):

Component	Class Name	File Name
View Controller	SignInViewController	SignInViewController.swift
Subviews	YourSubViewName	YourSubViewName.swift
Objects	YourObject	YourObject.swift
Custom TableView Cells	UserCell	UserCell.swift

Figure 2.4.1

- A view controller in iOS is an instance of a subclass of UIViewController. The UIKit provides several special-purpose subclasses of UIViewController, such as

UITableViewController. You must extend the framework view-controller classes to have the controller mediate the data between models and views. View controllers are typically the delegate or data source objects for many types of framework objects.

3. Code Guidelines

3.1. Native Swift Types

Use Swift types whenever possible (Array, Dictionary, Set, String, etc.) as opposed to the NS*types from Objective-C. Many Objective-C types can be automatically converted to Swift types and vice versa. See Working With Cocoa Data Types for more details.

```
let pageLabelText = "\(currentPage)/\(pageCount)"
let alsoPageLabelText = currentPage + "/" + pageCount
```

Swift Collection Types

Do not make NSArray, NSDictionary, and NSSet properties or variables. If you need to use a specific method only found on a Foundation collection, cast your Swift type in order to use that method.

```
var arrayOfJSONObjects = [[String: AnyObject]]()
let names: AnyObject? = (arrayOfJSONObjects as NSArray).value(forKeyPath:
"name")
```

Consider if there is a Swiftier way to do what you're trying to do:

```
var arrayOfJSONObjects = [[String: AnyObject]]()
let names: [String] = arrayOfJSONObjects.flatMap { object in
    return object["name"] as? String
}
```

3.2. Class Prefixes

Do not add a class prefix. Swift types are automatically namespaced by the module that contains them. If two names from different modules collide you can disambiguate by prefixing the type name with the module name.

```
// SomeModule.swift
public class UsefulClass {
    public class func helloWorld() {
        print("helloWorld from SomeModule")
    }
}

// MyApp.Swift
class UsefulClass {
    class func helloWorld() {
        print("helloWorld from MyApp")
    }
}

import SomeModule

let appClass = UsefulClass.helloWorld()
let moduleClass = SomeModule.UsefulClass.helloWorld()
```

Figure 3.2.1

3.3. Optionals

Force Unwrapping:

Avoid force unwrapping optionals by using ! or as! as this will cause your app to crash if the value you are trying to use is nil. Safely unwrap the optional first by using things like guard let, if let, guard let as?, if let as?, and optional chaining. A rare reason to force-unwrap would be if you have a value you expect to never be nil and you want your app to crash if the value actually is nil due to some implementation mistake. An example of this would be an @IBOutlet that accidentally gets disconnected. However, consider this an edge-case and rethink whether your own code could be refactored to not use force-unwrapping.

```
guard let url = URL(string: "http://www.example.com/") else {
   return
}

UIApplication.shared.open(url)
```

Downcast

```
guard let detailViewController = segue.destination as? DetailViewController else {
   return
}
detailViewController.person = person
```

Optional chaining

```
self.delegate?.didSelectItem(item)
```

if let Pyramid of Doom

Use multiple optional binding in an if let statement where possible to avoid the pyramid of doom:

```
if
   let id = jsonObject[Constants.Id] as? Int,
   let firstName = jsonObject[Constants.firstName] as? String,
   let lastName = jsonObject[Constants.lastName] as? String,
   let initials = jsonObject[Constants.initials] as? String {
      // Flat
      let user = User(id: id, name: name, initials: initials)
      // ...
}
```

If there are multiple unwrapped variables created, put each on its own line for readability (as in the example above).

3.4. Error Handling

- **Forced-try Expression:**

Avoid using the forced-try expression try! as a way to ignore errors from throwing methods as this will crash your app if the error actually gets thrown. Safely handle errors using a do statement along with try and catch. A rare reason to use the forced-try expression is similar to force unwrapping optionals; you actually want the app to crash (ideally during debugging before the app ships) to indicate an implementation error. An example of this would be loading a bundle resource that should always be there unless you forgot to include it or rename it.

```
do {
   let json = try JSONSerialization.jsonObject(with: data, options: .allowFragments)
```

```
    print(json)
} catch {
    print(error)
}
```

3.5. Let vs. Var

- Whenever possible use `let` instead of `var`.
- Declare properties of an object or struct that shouldn't change over its lifetime with `let`.
- If the value of the property isn't known until creation, it can still be declared `let`: assigning constant properties during initialization.

3.6. Access Control

Prefer private properties and methods whenever possible to encapsulate and limit access to internal object state.

For private declarations at the top level of a file that are outside of a type, explicitly specify the declaration as fileprivate. This is functionally the same as marking these declarations private, but clarifies the scope:

```
import Foundation

// Top level declaration
fileprivate let foo = "bar"

struct Baz {
...
}
```

If you need to expose functionality to other modules, prefer public classes and class members whenever possible to ensure functionality is not accidentally overridden. Better to expose the class to open for subclassing when needed.

3.7. Spacing

- Open curly braces on the same line as the statement and close on a new line.
- Put else statements on the same line as the closing brace of the previous if block.
- Make all colons left-hugging (no space before but a space after) except when used with the ternary operator (a space both before and after).

```
class SomeClass: SomeSuperClass {
   private let someString: String

   func someFunction(someParam: Int) {
      let dictionaryLiteral: [String: AnyObject] = ["foo": "bar"]

      let ternary = (someParam > 10) ? "foo" : "bar"

      if someParam > 10 {
         ...
      } else {
         ...
      }
   }
}
```

3.8. Closures

Shorthand Argument Syntax

- Only use shorthand argument syntax for simple one-line closure implementations:

```
let doubled = [2, 3, 4].map { $0 * 2 } // [4, 6, 8]
```

- For all other cases, explicitly define the argument(s):

```
let names = ["George Washington", "Martha Washington", "Abe Lincoln"]

let emails = names.map { fullname in
        let dottedName = fullname.replacingOccurrences(of: " ", with: ".")
        return dottedName.lowercased() + "@whitehouse.gov"
}
```

Capture lists

- Use capture lists to resolve strong reference cycles in closures:

```
UserAPI.registerUser(user) { [weak self] result in
  if result.success {
    self?.doSomethingWithResult(result)
  }
}
```

3.9. Protocols

Protocol Conformance:

- When adding protocol conformance to a type, use a separate extension for the protocol methods. This keeps the related methods grouped together with the protocol and can simplify instructions to add a protocol to a type with its associated methods.
- Use a // MARK: - SomeDelegate comment to keep things well organized.

```
class MyViewcontroller: UIViewController {
  ...
}

// MARK: - UITableViewDataSource

extension MyViewcontroller: UITableViewDataSource {
  // Table view data source methods
}

// MARK: - UIScrollViewDelegate

extension MyViewcontroller: UIScrollViewDelegate {
  // Scroll view delegate methods
}
```

Delegate Protocols:

- Limit delegate protocols to classes only by adding class to the protocol's inheritance list (as discussed in Class-Only Protocols).
- If your protocol should have optional methods, it must be declared with the @objc attribute.
- Declare protocol definitions near the class that uses the delegate, not the class that implements the delegate methods.
- If more than one class uses the same protocol, declare it in its own file.
- Use weak optional vars for delegate variables to avoid retain cycles.

```
//SomeTableCell.swift

protocol SomeTableCellDelegate: class {
    func cellButtonWasTapped(cell: SomeTableCell)
}

class SomeTableCell: UITableViewCell {
    weak var delegate: SomeTableCellDelegate?
    // ...
}
```

```
//SomeTableViewController.swift

class SomeTableViewController: UITableViewController {
    // ...
}

// MARK: - SomeTableCellDelegate

extension SomeTableViewController: SomeTableCellDelegate {
    func cellButtonWasTapped(cell: SomeTableCell) {
        // Implementation of cellbuttonwasTapped method
    }
}
```

3.10. Arrays and Dictionaries

Type Shorthand Syntax

- Use square bracket shorthand type syntax for Array and Dictionary as recommended by Apple in Array Type Shorthand Syntax:

```
let users: [String]
let usersByName: [String: User]
```

Trailing Comma

- For array and dictionary literals, unless the literal is very short, split it into multiple lines, with the opening symbols on their own line, each item or key-value pair on its own line, and the closing symbol on its own line. Put a trailing comma after the last item or key-value pair to facilitate future insertion/editing. Xcode will handle alignment sanely.

```
let anArray = [
   object1,
   object2,
   object3,
]

  let aDictionary = [
   "key1": value1,
   "key2": value2,
]
```

3.11. Constants

- Define constants for unchanging pieces of data in the code. Some examples are CGFloat constants for cell heights, string constants for cell identifiers, key names (for KVC and dictionaries), or segue identifiers.
- Where possible, keep constants private to the file they are related to.
- File-level constants must be declared with fileprivate let.
- File-level constants must be capital camel-cased to indicate that they are named constants instead of properties.
- If the constant will be used outside of one file, fileprivate must be omitted.
- If the constant will be used outside of the module, it must be declared public (mostly useful for Pods or shared libraries).
- If the constant is declared within a class or struct, it must be declared static to avoid declaring one constant per instance.

Example:

```
//SomeTableCell.swift

//not declared private since it is used in another file
let SomeTableCellIdentifier = "SomeTableCell"

class SomeTableCell: UITableViewCell {
    ...
}


//ATableViewController.swift

//declared fileprivate since it isn't used outside this file
fileprivate let RowHeight: CGFloat = 150.0

class ATableViewController: UITableViewController {
    ...
    private func configureTableView() {
        tableView.rowHeight = RowHeight
    }

    func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath:
NSIndexPath) -> UITableViewCell {
        return tableView.dequeueReusableCellWithIdentifier(SomeTableCellIdentifier,
forIndexPath: indexPath)
    }
}
```

Figure 3.11.

3.12. Function Parameters

- Name functions with parameters so that it is clear what the first parameter is.
- If that is not possible, provide an explicit label for the first parameter.
- Avoid using the underscore character to prevent explicit external parameter names.

Examples:

```
class Person {
    init(firstName: String, lastName: String) {
        // Called as Person(firstName: "...", lastName: "...")
    }

    func setName(name: String) {
        // Called as setName("...")
    }

    func setName(first firstName: String, last lastName: String) {
        // Called as setName(first: "...", last: "...")
     }
}
```

Figure 3.12.1

3.13. Flow Control

- For single conditional statements, do not use parentheses.
- Use parentheses around compound conditional statements for clarity or to make the order of operations explicit.
- When using optional booleans and optional `NSNumber`s that represent booleans, check for `true` or `false` rather than using the nil-coalescing operator:

```
if user.isCurrent?.boolValue == true {
    // isCurrent is true
} else {
```

```
    // isCurrent is nil or false
}
```

3.14. Switch Statements

- break is not needed between case statements (they don't fall through by default)
- Use multiple values on a single case where it is appropriate:

```
var someCharacter: Character
  ...
  switch someCharacter {
  case "a", "e", "i", "o", "u":
     print("\(someCharacter) is a vowel")
  ...
  }
```

- When pattern matching over an enum case with an associated value, use case .CASENAME(let ...)rather than case let ... syntax for value binding.

```
enum AnEnum {
    case foo
     case bar(String)
     case baz
  }

  let anEnumInstanceWithAssociatedValue = AnEnum.Bar("hello")

  switch anEnumInstanceWithAssociatedValue {
              case .foo: print("Foo")
             // Correct
             case .bar(let barValue): print(barValue) // "hello"
              case .baz: print("Baz")
```

3.15. Managing Libraries

- CocoaPods manages library dependencies for your Xcode projects.
- The dependencies for your projects are specified in a single text file called a Podfile. CocoaPods will resolve dependencies between libraries, fetch the resulting source code, then link it together in an Xcode workspace to build your project.
- Ultimately the goal is to improve discoverability of, and engagement in, third party open-source libraries by creating a more centralised ecosystem.

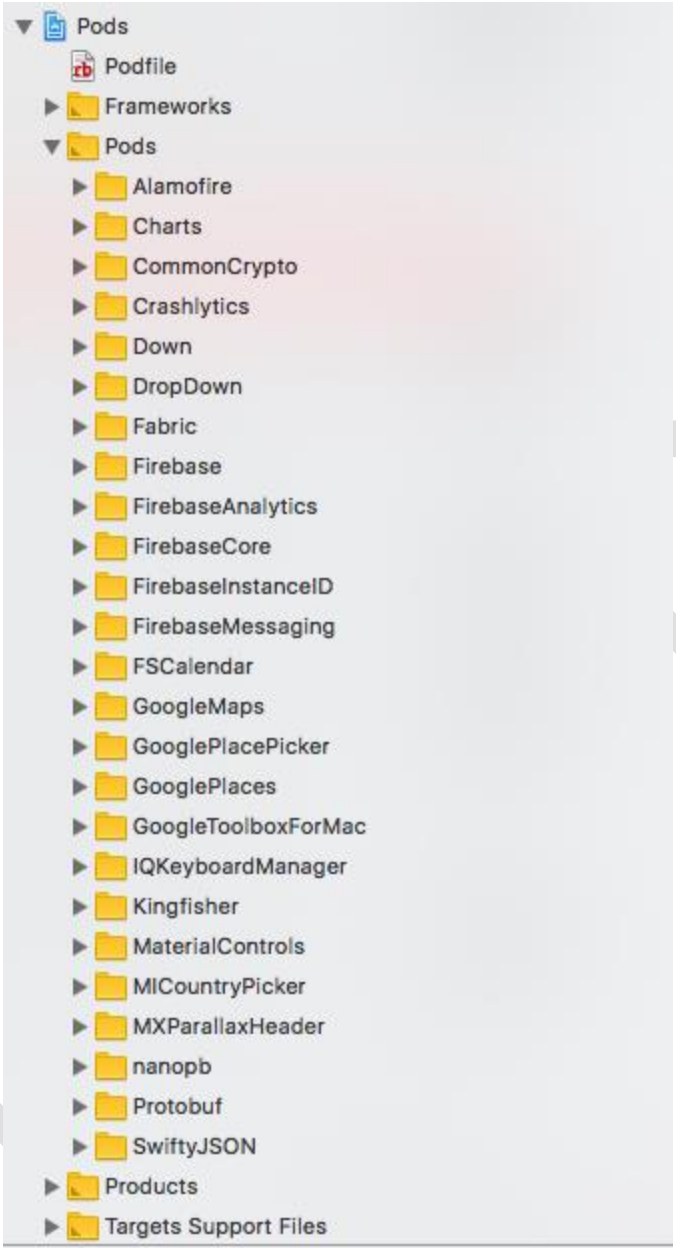

Figure 3.15.1

4. Database

4.1. Normalization

- Normalization is a database design technique which organizes tables in a manner that reduces redundancy and dependency of data.

SYSTEM: Hospital		**DATE:** / /	**AUTHOR:**
Source ID No.:		**Name of Source:** Prescription Record	
UNF	**1NF**	**2NF**	**3NF**
Pat No Surname Forename Ward No Ward Name Presc Date Med Code Med Name Dosage Lgth Treat	Pat No Surname Forename Ward No Ward Name Pat No Presc Date Med Code Med Name Dosage Lgth Treat	Pat No Surname Forename Ward No Ward Name Pat No Presc Date Med Code Dosage Lgth Treat Med Code Med Name	Pat No Surname Forename Ward No Ward No Ward Name Pat No Presc Date Med Code Dosage Lgth Treat Med Code Med Name

Figure 4.1.1

4.2. Schema & Table

- It divides larger tables to smaller tables and links them using relationships.

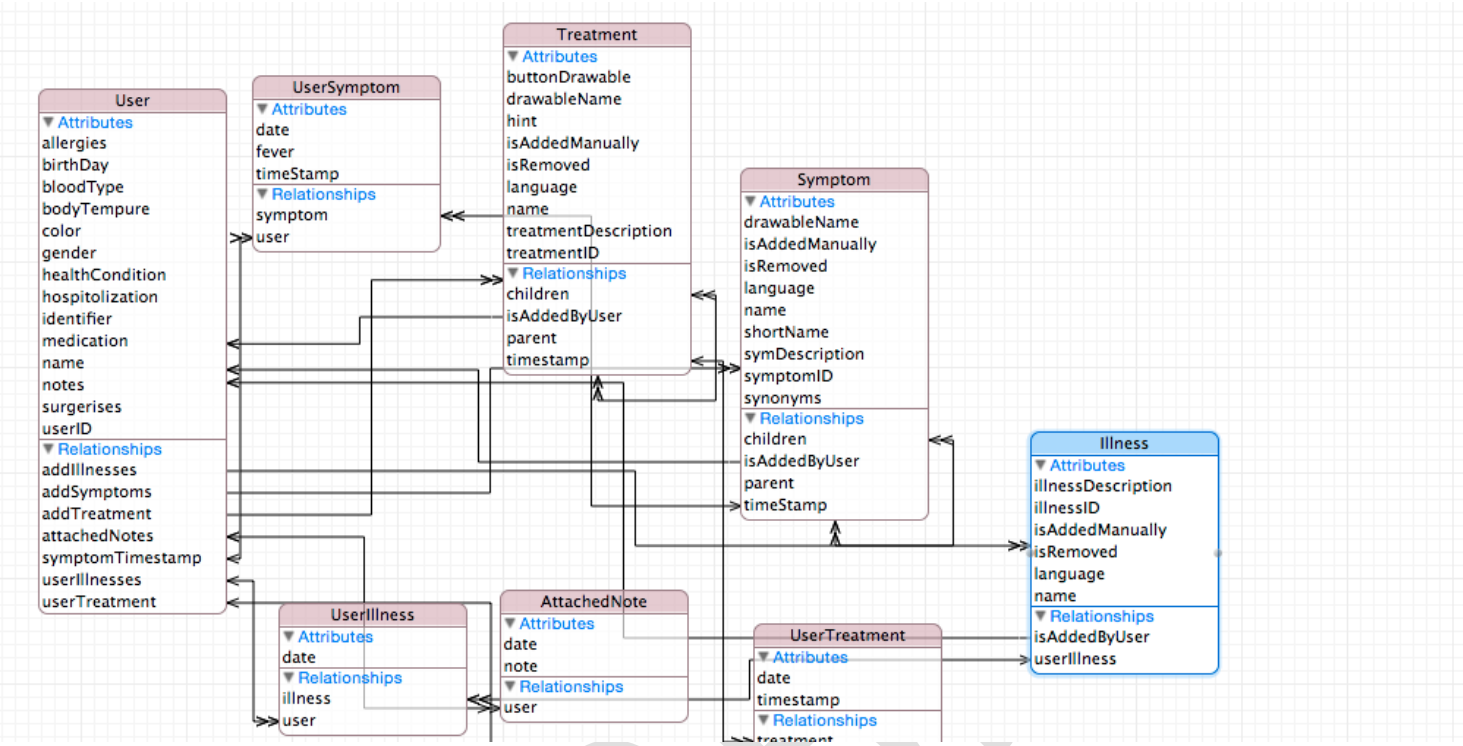

Figure 4.2.1

5. Commenting

- Inline Comment:
    - An inline comment will be used when any methods contain the main base logic which are important or for a complex logic which has been used to get some specific result.
    - In below example, an inline comment has been shown for code which has been used to apply a font style.

Inline Comment

```
///Data array to store data which maps to corresponding cells
var dataArray: [[CellData]] = []

///Refresh control to refresh TableView
private var refreshContorl = UIRefreshControl()

///Instance of DataRequestObject closure.
private var dataRequest: DataRequestClosure?

///Instance of CellForDataObject.
private var getCellForRow: CellForDataClosure?

///Current status of TableView
var currentStatus = CurrentStatus.loading

///Delegate instance which conforms to PBTableViewDelegate
var PBDelegate: PBTableViewDelegate?
```

Figure 5.1

- Outer Comment:
    - An outline comment will be used outer side of any methods which will indicate the purpose of the class method, input parameters to be passed, output result return by the method and exception which can be thrown by a method.

Outer Comment example

```
/// Closure which populates tableview with the dataArray and sets isLast. Otherwise calls the error closure.
/// - parameter response: The dataArray which is used to display tableview and isLast(Indicates End of Table).
/// - parameter error: Error if error occurrend.
typealias DataResponseClosure = (_ response: (dataArray: [[CellData]], isLast: Bool)?,_ error: (String)?) -> ()

/// Closure which takes "fromIndex" as argument to call the request closure which provides the data to be loaded in
    tableView.
/// - parameter fromIndex:  The current offset of the list.
/// - parameter response:  The response closure of type DataResponseClosure.
typealias DataRequestClosure = (_ fromIndex: Int, _ response: @escaping DataResponseClosure) -> ()

/// Closure which takes data of type CellData as an input(Which was retrieved from the DataRequestClosure). It loads the
    new cell in the tableview.
/// - parameter indexPath:  IndexPath of cell to be loaded.
/// - parameter data:  The data of type CellData which is used to load tableView.
/// - returns: tableView: An Object of self.
typealias CellForDataClosure = (_ indexPath: IndexPath, _ cellData: CellData, _ tableView: PBTableView) -> UITableViewCell
```

Figure 5.2

6. Image Loading

- We are using SDWebImage & Kingfisher image loading library.
- Kingfisher is a lightweight, pure-Swift library for downloading and caching images from the web. This project is heavily inspired by the popular SDWebImage. It provides you a chance to use a pure-Swift alternative in your next app.

```
let url = URL(string: "url_of_your_image")
imageView.kf.setImage(with: url)
```

- Basic usage of it is much simpler
- Asynchronous image downloading and caching.
- URLSession-based networking. Basic image processors and filters supplied. Multiple-layer cache for both memory and disk.
- Cancelable downloading and processing tasks to improve performance.
- Independent components. Use the downloader or caching system separately as you need.
- Prefetching images and showing them from cache later when necessary.
- Extensions for UIImageView, NSImage and UIButton to directly set an image from a URL.
- Built-in transition animation when setting images.
- Customizable placeholder while loading images.
- Extensible image processing and image format support.

7. API Calling Library

- We use Alamofire, which is the best swift library for networking and it has good support from community.
- We have our own wrapper around Alamofire in our networking layer which defines all the required endpoints.
- It also handles other things like Authorization with backend services, network failures, retrying network calls on failures.

```
let desiredStatus = !self.notification.isSelected
setNotificationStatus(
    id: self.notification.id,
    isSelected: desiredStatus,
    indicator: Indicator.grayedOut(view: self)
) { (response) in
    switch response
    {
    case .success(_):
        self.notification.isSelected = desiredStatus
        if self.notification.isSelected
        {
            self.btnisSelected.setImage( , for: .normal)
        }
        else
        {
            self.btnisSelected.setImage( , for: .normal)
        }
    case .failed(let errorMessage):
        JDStatusBarNotification.show(withStatus: errorMessage, dismissAfter: 5, styleName: JDStatusBarStyleError)
    }
}
```

```
func setNotificationStatus(
    id: String,
    isSelected: Bool,
    indicator: Indicator,
    response: @escaping (MessageOrErrorResponse)->Void)
{
    indicator.show()
    AlamofireModel.alamofireMethod(
        .post,
        apiAction: APIAction.setNotificationSettings,
        parameters: [
            id : isSelected ? "Y" : "N" /* Y - true, N - false */
        ],
        Header: [:],
        handler: { (res) in
            indicator.hide()
            if
                res.code == 200
            {
                response(.success(message: res.message))
            }
            else
            {
                response(.failed(errorMessage: res.message))
            }
    }) { (e) in
        indicator.hide()
        response(.failed(errorMessage: e.localizedDescription))
    }
}
```

```
class func alamofireMethod(
    _ method: Alamofire.HTTPMethod, //HTTP method which should be used
    apiAction: APIAction, //The API service endpoint to be used.
    parameters : [String : Any], //Paramters
    Header: [String: String], //Headers
    retryCount: Int = 3, //No. of retries, default is 3
    handler:@escaping CompletionHandler, //Closure for completion of API call
    errorhandler : @escaping ErrorHandler) //Closure in case of error
{
```

```
if retryCount == 0
{
    Crashlytics.sharedInstance().recordError(response.result.error!)
    errorhandler(response.result.error! as NSError)
}
else
{

    usleep(10000) // = 0.01 seconds
    print("unreachable, retry no. -> " + (abs(retryCount - 3)).description)
    alamofireMethod(
        method,
        apiAction: apiAction,
        parameters: parameters,
        Header: header,
        retryCount: (retryCount-1),
        handler: handler,
        errorhandler: errorhandler
    )
}
```

```
alamofireManager?.request(
    UrlFinal,
    method: method,
    parameters: _parameters,
    encoding: URLEncoding.default,
    headers: header)
    .responseJSON(
        queue: nil,
        options: JSONSerialization.ReadingOptions.allowFragments,
        completionHandler:
        {
            (response) in

            if response.result.isSuccess
            {
                if let val = response.result.value
                {
                    var message = ""
                    if let m = (response.result.value as AnyObject).value(forKey: "message")
                    {
                        message = m as! String
                    }

                    handler(AlamofireResponse(Object: val as AnyObject, code: response.response?.statusCode ??
                        200, message: message))
```