The IOS INTERVIEWS GUIDE

Questions, Answers, & General Guidance on what iOS developers should know to nail any interview.

ALEX BUSH

The iOS Interview Guide

v 0.5.0

Alex Bush

Preface

The iOS Interviews Guide V 0.5.0.

This is a pre-order version of The iOS Interviews Guide by Alex Bush and it contains current content of the book with the exclusion of many draft raw materials and notes that have any meaning only to the author.

Please be aware that the book still has a lot of unpolished and raw edges such as unfinished proofreading and content structure. Table of contents and text of the book can change in the next versions. But the main value and message the book sends will stay the same course.

As an early adopter, I appreciate your interest and desire to support an indie author. People like you make it worthwhile to take the time and put my thoughts into this book. You'll receive future updates of the book as I write them and will be the first to get the final result of this laborious venture.

Feel free to reach out if you have any questions, find a typo, find something unclear, or if you'd just like to say hi or share your recent development or interview experience!

```
Note: code samples are currently overflowing pages.
It will be fixed when I'd get a chance to run the content by an editor.
```

- My email is **alex.bush@smartcloud.io**
- Twitter: **alex_v_bush**

- Gihub: **alexvbush**

Contents

Chapter 1

Intro

Ok, here we are. As developers we love our craft, and even more we love to be paid for our craft. That is why we get jobs. And to get a better job we need to go to those notorious interviews...

Do you hate them as much as I do? It takes so much time to prepare and you still can't guess what crazy thing they'll ask you on the interview making you sweat and jitter.

If so read on. This is a no bs, down to business, pragmatic guide on how to get ready for your iOS interview. Whether you're applying for a senior position or just starting out as a junior. This guide will help you get over your anxiety and actually give you concrete steps and guidance on what you need to know as a modern iOS developer. The best way to not be nervious and nail your interviews is to actaully know more and better than your interviewer :) If you wanted to know "advanced stuff" than this is the guide for you!

Working with iOS for 6+ years I've built 20+ apps, code reviewed thousands of lines of code, mentored several developers and interviewed a lot. I know all the struggles you go through with iOS development and know what pitfalls there are.

Chapter 2

Step One. Figure out what is the big picture.

First things first - find out what iOS world is all about and what possibly you could be asked about. Big picture strategy makes it easier to orient yourself. You can figure out the details later when necessary.

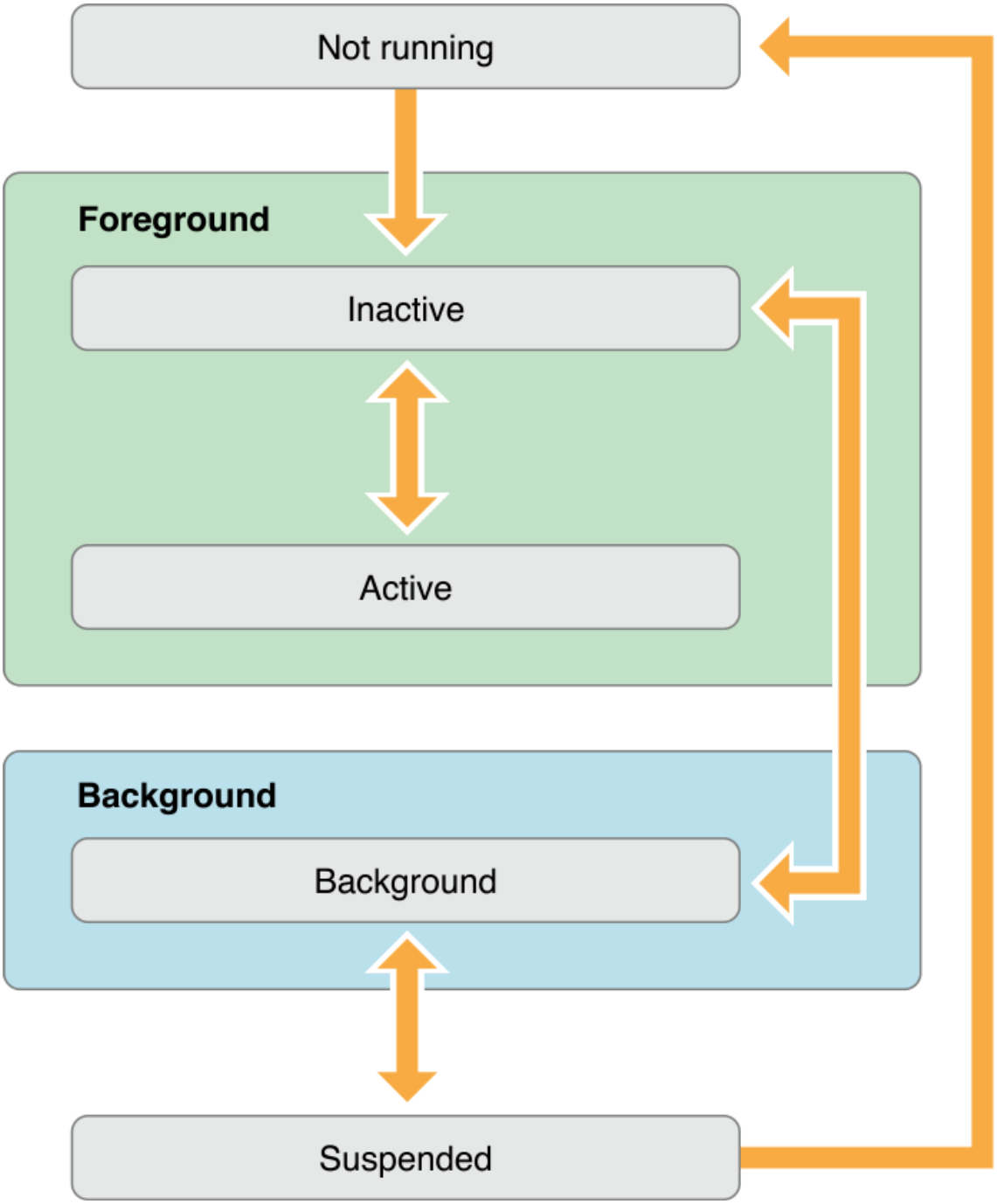
If you build enough apps you'll start noticing patterns. Things that you do over and over again in one form or another which essentially are the same. When that happens you realize how all apps are similar to each other. Sure they might differ in looks and what they do for the user but overall, the way you build them is the same. In this chapter, we'll look at what iOS apps are, where they fit in iOS system, the big picture, and design patterns that emerge out of building iOS apps.

2.1 What is an iOS application and where your code fits into it

If you think about it hard enough your typical iOS application is just a giant glorified run loop. It waits for user input and gets interrupted by external signals

such as phone calls, push notifications, and other app life cycle events.

// TODO: needs a better diagram



That is really it. App is launched, and then it sits and waits for user input, whether it's touches or "home" button click to put the app in the background. **`UIApplication`** is just a glorified **`main()`** loop that calls convenient callbacks to your **`UIAppDelegate`** subclass. Those "convenient" callback methods would be **`application:willFinishLaunchingWithOptions:`**, **`application:didFinishLaunchingWithOptions:`**, **`applicationDidBecomeActi`** **`applicationDidEnterBackground:`**, **`applicationWillResignActive:`**, **`applicationWillEnterForeground:`**, etc. Everything that we used to working with in a typical app delegate.

What you actually do as an iOS app developer is just "plug-in" into those callbacks to run your application's code and business logic. As soon as you understand that you will realize where the line is drawn between your app and Cocoa Touch code. It is an important distinction to make. Of course, the reality of day-to-day development is that your code will be very tightly dependent or coupled to Apple's iOS frameworks. But nevertheless, you should do your best to decouple your code from it so that it is more maintainable and stays sane over the course of your project's evolution. Because we all know that in software development only one thing remains constant and that thing is change.

One way to see that for yourself is to create a new single screen project in Xcode and strip out everything related to UI (View Controllers and Window) in your **`AppDelegate`** subclass.

For example a brand new project's **`AppDelegate`** would look like this:

```
import UIKit

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {

    var window: UIWindow?


    func application(application: UIApplication,
                     didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?) -> Bo
    {
        let storyboard = UIStoryboard(name: "Main", bundle: nil)

        let rootViewController = storyboard.instantiateInitialViewController()
```

```
        self.window = UIWindow(frame: UIScreen.mainScreen().bounds)
        self.window?.rootViewController = rootViewController
        self.window?.makeKeyAndVisible()

        return true
    }

    func applicationWillResignActive(application: UIApplication) {

    }

    func applicationDidEnterBackground(application: UIApplication) {

    }

    func applicationWillEnterForeground(application: UIApplication) {

    }

    func applicationDidBecomeActive(application: UIApplication) {

    }

    func applicationWillTerminate(application: UIApplication) {

    }
```

It is very typical to have something like that where you'd either use a storyboard or create initial view controller in code. But at the end of the day what happens is that you create a **`UIWindow`** to be the main window of UI of your application and then you create the first View Controller that is going to be displayed to the user.

But if you'd remove all that UI code your application is still going to be a perfectly valid iOS app and it's even going to lunch! Heck, even all the callback methods that the **`main()`** loop under the hood sends to us will be received as in a normal application:

```
import UIKit

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {

    var window: UIWindow?
```

```
    func application(application: UIApplication,
                     didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?) -> Bo
    {
//        let storyboard = UIStoryboard(name: "Main", bundle: nil)
//
//        let rootViewController = storyboard.instantiateInitialViewController()
//
//        self.window = UIWindow(frame: UIScreen.mainScreen().bounds)
//        self.window?.rootViewController = rootViewController
//        self.window?.makeKeyAndVisible()

        return true
    }

    func applicationWillResignActive(application: UIApplication) {
        print("applicationWillResignActive")
    }

    func applicationDidEnterBackground(application: UIApplication) {
        print("applicationDidEnterBackground")
    }

    func applicationWillEnterForeground(application: UIApplication) {
        print("applicationWillEnterForeground")
    }

    func applicationDidBecomeActive(application: UIApplication) {
        print("applicationDidBecomeActive")
    }

    func applicationWillTerminate(application: UIApplication) {
        print("applicationWillTerminate")
    }
}
```

Try to run this and you'll see a black screen instead of any kind of UI but notice that all the methods like **`applicationDidBecomeActive`**, **`applicationWillResignA`** etc. still called when you click on "home" button and open your app back again. UI is just your app's code, the system doesn't care if you have any. It just keeps running its **`main()`** loop.

To iOS system your app is just yet another building block, yet another run/main loop that can be launched on user demand or when some other even in the system like push notification or location change happens.

2.2 Patterns and layers

After you build a few iOS application of various complexity one thing you might start to notice is that there are distinct layers of responsibility in each app. Whether the app is doing some location GPS work, or networking, or stores things to disk, they all at the end of the day have the following groups of responsibility: Storage layer, Service layer, Business Logic layer, UI layer.

What exactly goes into each layer varies from app to app but roughly the following things could be grouped in each layer:

2.2.1 Storage Layer

Storage layer can be as simple as an array or dictionary of data that holds models in memory for your app. Or as complex as a Core Data or custom SQL ORM solution that can be observed and queried with advanced predicates. The main thing and responsibility of that layer is that it stores data for your application and can play the role of "ultimate source of truth" for the rest of your code. Examples of what goes into that layer could be the following: Core Data, Realm, NSUserDefaults, KeyChain, Disk File storage, in-memory arrays and dictionaries/sets.

2.2.2 Service Layer

This layer is responsible for all things networking and external communication. That could be, as pretty much any app these days needs, an HTTP client and a set of accompanying objects that do networking for the app and connect with the backend JSON API. Or it could be a BLE client wrapper code that helps your app communicate and send or receive data from external Bluetooth devices. Or it could be a socket connection code that allows your app to subscribe to server events and receive, let's say, comments from another chat participant. Or it could be a location service that connects with device's GPS delegates and

gets location change updates. You get the picture, the bottom line is that it's the code that knows how to work with external interfaces, whether it's HTTP or BLE or something else. Also quite often data serialization and mapping (let's say from JSON to your custom objects) are included in this layer as well.

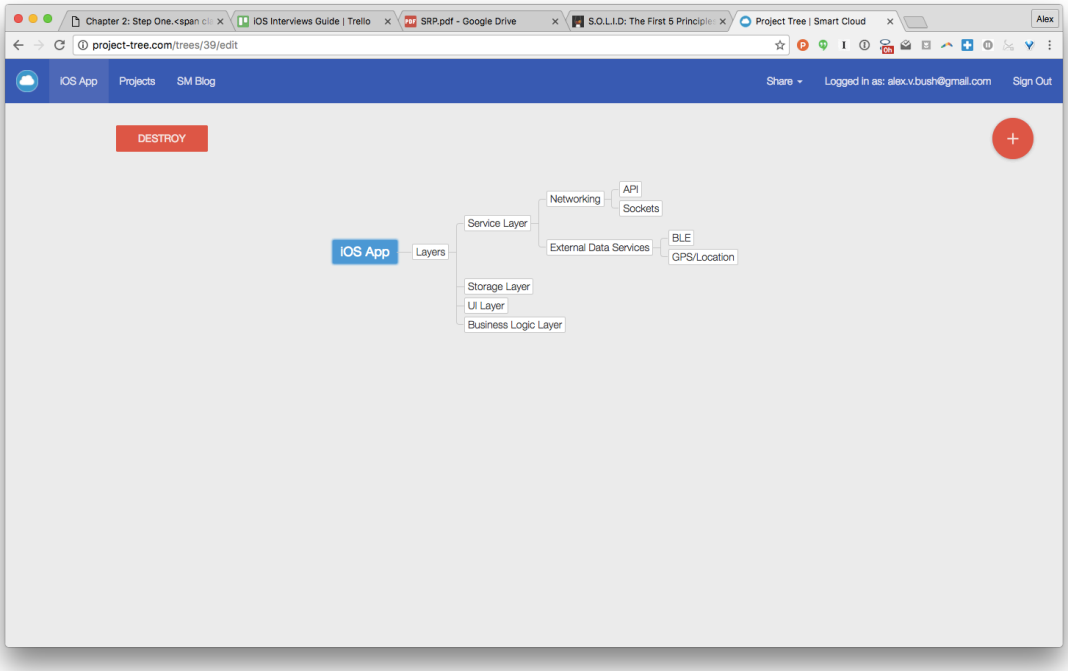
2.2.3 UI Layer

UI layer is responsible for drawing things on the screen. This is all the stuff that naturally goes into that bucket like UIView subclasses, Autolayout, Table Views, Buttons, Collection Views, Bar Buttons etc. One other thing that is also might not be obvious that belongs to this layer is View Controllers and View Models. View Controllers suppose to do, well just that, control the view. View Models are complimentary objects that help with decluttering and decoupling views from other layers of responsibility. Remember the key to happy and healthy iOS codebase is a skinny controller :)

2.2.4 Business Logic Layer

In this layer go objects that are responsible for actual application's business logic, objects that use components of other layers to achieve results and the work for the user. Coordinators that use HTTP service objects in conjunction with storages to orchestrate data receiving from backend APIs and persistence to Core Data would be one example. Another could be a manager that takes care of token encryption and saving to keychain using keychain storage and some kind of encryption service. The main idea is that this layer helps us keep services, storages, and other layers decoupled from each other and tell them what to do to achieve results.

This is how it looks overall:



2.3 Decoupling and Single Responsibility Principle

The layers we've looked at and that whole approach of decoupling things is an extension and application of SOLID principles and Single Responsibility principle especially. It states that "A class should have only one reason to change". What that means is that for example, if we have an object from Service Layer that connects to our backend API and retrieves a list of todos to display then the only reason for this object to change is if the way we retrieve objects from the API changed. It should never change because we want to store those todos in memory differently now, it's storage's responsibility. Keeping things separate like that helps maintain sanity in your codebase and makes it very flexible and receptive to change.

2.4 Zooming In

All of the layers above constitute, roughly speaking, the big picture of any iOS application, big or small. There's always a lot of other things and areas we could look at like all the iOS frameworks: Accelerate, Accounts, AddressBook, AddressBookUI, AdSupport, AssetsLibrary, AudioToolbox, AudioUnit, AVFoundation, AVKit, CFNetwork, CloudKit, CoreAudio, CoreAudioKit, CoreBluetooth, CoreData, CoreFoundation, CoreGraphics, CoreImage, CoreLocation, CoreMedia, CoreMIDI, CoreMotion, CoreTelephony, CoreText, CoreVideo, EventKit, EventKitUI, ExternalAccessory, Foundation, GameController, GameKit, GLKit, GSS, HealthKit, HomeKit, iAd, ImageIO, IOKit, JavaScriptCore, LocalAuthentication, MapKit, MediaAccessibility, MediaPlayer, MediaToolbox, MessageUI, Metal, MobileCoreServices, MultipeerConnectivity, NetworkExtension, NewsstandKit, NotificationCenter, OpenAL, OpenGLES, PassKit, Photos, PhotosUI, PushKit, QuartzCore, QuickLook, SafariServices, SceneKit, Security, Social, SpriteKit, StoreKit, SystemConfiguration, Twitter, UIKit, VideoToolbox, WebKit. But those are only the tools that if necessary will be used by our applications in one of the layers described above.

The main thing that will help you as an iOS developer is the ability to figure out where each piece goes and when/how to use those tools. This is what we are going to cover in future chapters.

2.5 SOLID principles

There are many patterns that were developed by software developers over the years. There is good old MVC (we'll talk about it a bit later). There's MVVM. There is VIPER, and many many others. But the main thing that unites them all is that they were born from and follow SOLID principles.

Where your application fits in the world of iOS devices and Apple frameworks is the direct application of SOLID principles. Every component of your app should be responsible for one thing, be open for extension, ... And your iOS

applications is a sum and composition of those individual components that in turn is also should be responsible for one thing in the whole eco-system of iOS. I hope now you see how understanding that your app is just a `main()` run loop fits into all of this. It all depends on how high of an altitude you're looking at it.

2.6 Beyond MVC

// TODO

Chapter 3

Step Two. Find out what type of team and company you'd want to work for.

Chapter 4

Step Three. Learn the fundamentals

Fundamental iOS questions are the questions about things that you'll be 100% working with day to day as an iOS developer. Things like memory management, protocols, extensions, let/var, optionals, KVO, delegates, etc. Depending on the position you're applying for (jr, mid, sr, etc.) you'll either have a lot of those questions or just a bit but nevertheless expect them in one form or another. Those questions can be tedious and boring for an experienced developer but its always good to brush up your skills.

Interview questions covered in this chapter:

- What is let and var in Swift?
- What is Optional in Swift and nil in Swift and Objective-C?
- What is the difference between `struct` and `class` in Swift? When you'd use one or another?
- How memory management is handled on iOS?
- What are properties and instance variables in Objective-C and Swift?

- What is a protocol (both Obj-C and Swift)? When and how it is used?
- What is a category/extension? When is it used?
- What are closures/blocks and how they are used?
- What is MVC?
- What are Singletons? What are they used for?
- What is Delegate pattern on iOS?
- What is KVO (Key-Value Observation)?
- What iOS application lifecycle consists of?
- What is View Controller? What is its lifecycle?

4.1 What is let and var in Swift?

This is a basic Swift question that might open up opportunities for deeper discussions around language semantics and mutability/immutability in languages in general and their respective advantages and disadvantages. Be ready to go either direction.

Expected answer:

`let` keyword is used to declare a constant and `var` keyword is used to declare a variable. Variables created with both of them are either references/pointers or values. The difference between them is that when you create a constant with `let` you have to give it a value upon declaration and you can't reassign it. And when you declare a variable with `var` it can either be assigned right away or at a later time or not at all (i.e. be `nil`).

This is a fundamental Swift thing that you should be familiar with. Unlike with Objective-C where everything is dynamic and can be `nil` and `nil` in turn

can receive messages without breaking everything in Swift you have to be very explicit about what you are declaring.

At the end of the day `let`, `var`, `nil`, and `Optionals` (as you'll see in the next section) help define how you handle state in your apps. Swift forces you to be more explicit about it.

4.2 What is Optional in Swift and nil in Swift and Objective-C?

This is another fundamental Swift question that you should be expecting on iOS interviews. Different from Objective-C treatment of nils and introduction of `Optionals` makes Swift development style in some cases dramatically different from Objective-C. Be ready to potentially talk at length about the big picture architectural implications of that in Swift and how it is going to affect how you're writing your code.

Expected answer:

In Objective-C `nil` used to be a very handy "value" for variables. It typically meant an absence of value, simply "nothing". You could send a message to a `nil` and instead of your app blowing up with an exception it would simply ignore it and do nothing (or return `nil`). In Swift, though, with the introduction of `let` and `var` it became apparent that not all of the constants and variables can be defined and set at the time of declaration. We needed to somehow declare that a variable is undetermined yet and potentially could have a value or could have no value at all. That's where `Optional` comes into play.

Optional is defined with `?` appended at the end of the variable type you declare. You can set a value to that variable right away or at a later time or not even set it at all. When you use Optional variables you have to either explicitly unwrap them, using `!` at the end of the variable, to get the value stored in it or you could do a so-called `Optional Binding` to find out whether an optional contains a value. To do that you'd use a

```
if let unwrappedOptional = someOptional {
  // your code here
}
```

construct.

Optionals cannot be used with constants (**let**s) because a constant has to be defined at the time of its declaration and therefore has value is not an Optional.

Unlike in Objective-C in Swift sending messages to nil causes a runtime exception. But there is a way to do something somewhat similar in Swift where you could send a message to an Optional but if optional is a nil than instead of raising an exception it will just ignore it and return nil, much like the old Objective-C behavior. That allows you to do method chaining calls and if one of the optional values in the call sequence is nil then the whole chain will return nil.

Optionals make Swift lean more towards the functional side of programming languages partially mimicking **Maybe** concept of Haskel and similar languages. At the end of the day just like **let**, **var**, and **nil** Optional is a helpful construct that forces you to be more mindful of you handle the state of your applications.

In general, you should be very mindful of the state of your applications and use **nil** and consequently **Optionals** to represent an absence of value as less as possible. Every time you declare an Optional ask yourself a few times if you really really need it.

```
NOTE: Objective-C now has nonull and nullable directives to give it more Swift-like
explicit variable type declaration.
```

Red flag:

Besides not knowing what optionals are and how to work with them the biggest red flag would be to speak in favor of Optional Binding and explicit Optional Unwrapping. The former leads to poor design and cognitive overhead of if/else statements and the latter to the potential danger of runtime exceptions.

4.3 What is the difference between `struct` and `class` in Swift? When you'd use one or another?

Structs and classes in Swift are very similar and different at the same time. This is another fundamental language question asked to gauge your level of understanding of Swift and the features it offers.

Expected answer:

`Structures` and `Classes` are very similar in Swift. Both can have properties, methods, subscripts, initializers, be extended/have a super type, conform to protocols.

`Classes` are *reference types*, they increase their reference count when passed to a function or assigned to a variable or constant. They also have some extra stuff like inheritance from a superclass, type casting, and deinitializers (former `dealloc`).

`Structs` are so called *value types*. That means that when a `struct` is assigned to a variable or constant or passed to a function its *value* is copied instead of increasing its reference count.

The key thing choosing between using a class or a struct is reference or value passing. If you need to store some primitives (i.e. Ints, Floats, Strings, etc.) then use `struct` if you need custom behavior where passing by reference is preferable (so that you refer to the same instance everywhere) then use `class`.

Red flag:

A red flag for this kind of question would be saying that you don't really use `structs` and prefer `classes` everywhere, just look in good old Objective-C. Structures is a great modern addition to Swift language that yet again, just like strong typing, `let`/`var`, and `Optionals`, forces developers to think harder about the data they use in their apps.

4.4 How memory management is handled on iOS?

Memory management is very important in any application and especially so in iOS apps that have memory and other constraints. This is one of the standard questions asked in one form or another and it refers to ARC, MRC, *reference types*, and *value types*.

Expected answer:

Swift uses Automatic Reference Counting (ARC) which is conceptually the same thing as in Objective-C. ARC keeps track of **strong** references to instances of classes and increases or decreases their reference count accordingly when you assign or unassign instances of classes (reference types) to constants, properties, and variables. ARC does not increase or decrease reference count of *value types* because, when assigned, they are copied. By default, if you don't specify otherwise all the references will be strong references.

One of the gotchas of ARC to be aware of is Strong Reference Cycles. Under ARC for a class instance to be fully deallocated it needs to be free of all the strong references to it. But there is a chance you could structure your code the way that two instances strongly reference each other and therefore never let each other's reference count drop down to zero. There are two ways of resolving this in Swift: *weak references* and *unowned references*. Both of them will assign an instance without keeping a strong reference to it. Use `weak` keyword for one and `unowned` keyword for another before a property or variable declaration. *Weak reference* is used when you know that reference is allowed to become `nil` and *unowned reference* is used when you are certain that that reference has a longer lifecycle and will never become `nil`. Since `weak` references can have a value or can have no value at all they must be defined as optional variables. And unowned reference has to be defined as non-optional since it is assumed to always have a value.

Another important gotcha is Strong Reference Cycle in Closures. When you use closures within a class instance they could potentially capture `self` in them and if self, in turn, retains that closure than you'd have a mutual strong reference cycle between closure and class instance. It often occurs when you use

lazy loaded properties, for example. To avoid that you'd use the same keywords `weak` and `unowned`. When you define your closure you should attach to its definition a so called *capture list*. A capture list defines how the closure would handle references captured in it. By default, if you don't use a capture list everything will be strongly referenced. Capture lists are defined either on the same line where the closure open bracket is or on the next line after that. They defined with a pair of square braces and each element in it has `weak` or `unowned` keyword prefix and is separated by commas. The same thinking as with variable references applies to closure capture list - define a capture variable as a `weak` optional if it could become nil at some point and the closure won't be deallocated before then, and define captured reference as `unowned` if it will never become nil before the closure is deallocated.

Red flag:

This is a must know for every iOS developer! Memory leaks and app crashes are all too common due to poorly managed memory in iOS apps.

4.5 What are properties and instance variables in Objective-C and Swift?

This could be a part of memory management question or a standalone question. Properties, instance variables, constants, and local variables are very important to understand working with Objective-C and Swift because they define how you refer and work with your data.

Expected answer:

`Properties` in Objective-C are used to store data in instances of classes. They define memory management, type, and access attributes of the values they store such as `strong`, `weak`, `assign`, `readonly`, `readwrite`, etc. Properties store values assigned to them in an instance variable that by convention has the same name as the property but starts with an underscore prefix. When you declare a property in Objective-C it will also synthesize it, meaning create a getter and

setter to access and set the underlying instance variable.

strong, **weak**, **assign** property attributes define how memory for that property will be managed. It is going to be either strongly referenced, weakly referenced (set to **nil** if deallocated), or assigned (not set to **nil** if deallocated).

One great feature of Objective-C properties that is often overlooked is **Key Value Observation (KVO)**. Every Objective-C property can be observed for changes enabling low-level Functional Reactive Programming capabilities.

In Swift, however, properties defined with a simple **let** or **var** and are **strong** by default. They can be declared as weak or unowned references with **weak** and **unowned** keywords before **let/var**. Swift properties in types are called stored properties. Unlike Objective-C properties, they do not have a backing instance variable to store their values. They do declare setters and getters that can be overridden though.

Swift enforces basic dependency injection with properties. If you define a **let** or **var** property it has to be either initialized in the property declaration and will be instantiated with that type's instance or it has to be injected in a designated initializer instead. Optional properties don't have to be initialized right away or injected because by their nature they can be **nil**.

Also, Swift properties can't be KVOed and instead have a greatly simplified mechanic built in - **Property Observers** (**willSet/didSet**). The only way to have property KVO in Swift is to subclass from **NSObject** or its subclasses.

Class or type properties are the properties defined for the entire type/class rather than individual instances of that type. In Swift, they can be defined with **static** keyword for value types (**struct**, **enum**) and with **class** keyword for class types. In Objective-C since Swift 3 and Xcode 8, you can also define class properties using **class** keyword in property declaration.

In both Swift and Objective-C properties can be lazy loaded. In Swift, you'd use **@lazy** directive in front of a property declaration. In Objective-C, you'd have to override property getter and set and initialize its value only if the underlying instance variable is **nil**.

Red flag:

You don't have to go too deep into the details of properties implementations and features in Swift and Objective-C but you have to know at least the basics of strong/weak/unowned referencing.

4.6 What is a protocol (both Obj-C and Swift)? When and how it is used?

Protocols are vital for any strongly typed OO language. Both Objective-C and Swift utilize them and you should be expecting this kind of question on every iOS interview. You have an option to either just quickly go over the functionality and purpose of protocols or to steer your conversation to a deeper discussion of protocol-oriented programming. It's up to you.

Expected answer:

`Protocols` or in other languages `Interfaces` are declarations of what a type that adopts them should implement. Protocol only has description or signature of the methods, properties, operators, etc. that a type implements without the actual implementation of those.

In both Swift and Objective-C protocols can inherit from one or more other protocols.

In Objective-C, protocols can declare properties, instance methods, and class methods. They can be adopted only by classes. You could define methods and properties as optional or required. They are required by default.

In Swift, protocols can declare properties, instance methods, type methods, operators, and subscripts. They can be adopted by classes, structs, and enums. By default, everything in Swift protocols is required. If you'd like to have optional methods and properties you have to declare your Swift protocol as Objective-C compatible by prefixing it with `@objc`. If you prefix your protocol with `@objc` it can only be adopted by classes.

Swift also lets you provide a default implementation for your protocols with a protocol extension.

Protocols are a great addition to any OOP language that allows you to clearly and explicitly declare interfaces of things your code and be able to rely on them. It is a way to abstract internal implementation details out and care about types rather than about inheritance structures. Declaring clear protocols allows you dynamically change objects that conform to the same protocol at runtime and to abstract things out and code against interfaces rather than specific classes or other types. It helps with the implementation of core Cocoa Touch design patterns such as `Delegation` for example. Also developing against protocols could help with test-driven development (TDD) since stubs and mocks in test could adopt the necessary protocols and substitute or "fake" the real implementation.

Red flag:

Protocols are one of the fundamental features of Objective-C and Swift. Being able not only use and adopt existing protocols that Cocoa Touch provides but also to create your own is crucial for any iOS developer.

4.7 What is a category/extension? When is it used?

Categories and extensions are super useful developing with Objective-C and Swift. Having a handle on benefits and limitations of categories and extensions is an important skill so expect this question pretty much on every interview.

Expected answer:

`Categories` in Objective-C and `Extensions` in Swift are a way to extend existing functionality of a class or type. They allow you to add additional methods in Objective-C without subclassing. And in Swift to add computed properties, static properties, instance/type methods, initializers, subscripts, new nested types, and make existing type conform to a protocol without subclassing.

In Objective-C, they are typically used to extend functionality of 3rd party or

Apple framework classes or in your own classes to distribute implementation into separate source files or to declare private or "protected" methods.

In Swift, extensions are used to extend functionality of existing types or to make a type conform to a protocol.

The drawback of extensions and categories is that unlike protocols they are globally applied. Meaning after you define an extension/category for a class or type it will be applied to all the instances of that type, even if they were created before the extension/category was defined.

Categories and Extensions cannot add new stored properties in either Swift or Objective-C.

Another important gotcha with categories and extensions is name clashes - if you define the same name from another category/extension or existing class/type in an extension/category then it is unpredictable what implementation will take precedence at runtime. To avoid that collision it is advised to namespace your methods with a prefix and an underscore. Something like `func ab_myExtensionMethod` where `ab` is your codebases class/type name prefix (same convention as with `NS` prefix for Cocoa's legacy NextStep).

Red flag:

Extensions/Categories used to be an advanced feature of Objective-C and Swift but not anymore. The key is not to abuse it.

4.8 What are closures/blocks and how they are used?

Blocks and closures are an integral part of Objective-C and Swift development. This question is used to be an advanced one for Objective-C developers but nowadays it is a standard for both Objective-C and Swift so it is going to be asked 100%.

Expected answer:

Blocks Objective-C and closures in Swift declare and capture a piece of exe-

cutable code that will be launched at a later time. You can either define them inline or give them dedicated type names to be referenced and used later. Blocks and closures are the first steps to multi-threading and asynchronicity in Swift and Objective-C since they are the building blocks that capture the work that needs to be executed at later time aka asynchronously.

Blocks/closures are reference types and will retain/strongly reference everything put in them unless otherwise specified. You can avoid strong reference cycle issues by using `__block` and `__weak` keywords in Objective-C (or better use `@strongify/@weakify`) and `[weak self]`/`[unowned self]` in Swift.

Blocks and closures syntax is notoriously hard to remember so you find yourself stuck check out these two websites: http://fuckingblocksyntax.com/ http://fuckingclosur

If those domain names are too offensive to you try these more friendly counterparts: http://goshdarnblocksyntax.com/ http://goshdarnclosuresyntax.com/

Red flag:

The main red flag with blocks and closures is memory management. Make sure you talk about strong reference cycle and how to avoid it with blocks/closures.

4.9 What is MVC?

Oh, the good old MVC. This is a fundamental design pattern Apple keeps pushing onto iOS developers. Every single interviewer will ask that question.

Expected answer:

`MVC` stands for Model View Controller. It is a software design pattern Apple chose to be the main approach to iOS application development. Application data is captured and represented by Models. Views are responsible for drawing things on the screen. And Controllers control the data flow between Model and View. Model and View never communicate with each other directly and instead rely on Controller to coordinate the communication.

A typical representation of each MVC layer in iOS application would be the following:

- `UIView` subclasses (Cocoa Touch or custom) are the `Views`
- `UIViewControllers` and their subclasses are the `Controllers`
- and any data objects and `NSManagedObject` subclasses and similar are the `Models`

MVC is a great general purpose design pattern but using it solely limits your architecture and quite often leads to notorious "Massive View Controller". "Massive View Controller" is the state of a codebase where a lot of logic and responsibility that doesn't belong there was shoved into View Controllers. That practice makes your code rigid, bloated, and hard to change. There are other design patterns that can help you remedy that such as MVVM and just general SRP principle. Even though Apple keeps telling us that MVC is everything don't be fooled by it and stick to SOLID principles.

Red flag:

You absolutely have to know what MVC is. That's basic to any iOS development. But at the same time explore alternatives and additions such as MVVM.

4.10 What are Singletons? What are they used for?

Singleton is a common design pattern used in many OOP languages and Cocoa considers it one of the Cocoa Core Competencies.. This question comes up from time to time on interviews to either gauge your experience with singletons or to find out if you have some other than iOS background.

Expected answer:

`Singleton` is a class that returns only one and the same instance no matter how many time you request it.

`Singletons` sometimes considered to be an anti-pattern. There are multiple disadvantages using singletons. The two main are global state/statefulness and object lifecycle and dependency injection.

Singletons are often misused and breed global state that makes debugging and working with our code difficult. It starts off very innocent, you think that you'll have only that one instance of that class. But later at some point you'd need to either reset it do something else with the data stored on it or sharing it across the whole app doesn't make sense anymore.

Using singletons makes it hard for you to inject dependencies since, well, with a singleton there's only one instance of your singleton class. That prevents you from injecting it as a dependency for purposes of testing and just general inversion of control architecture.

Red flag:

Never say that singletons are good for global values and storages. Architecting your apps that way leads to a disaster.

4.11 What is Delegate pattern on iOS?

Just like MVC this is one of the fundamental Cocoa design patterns. Will be asked on every interview.

Expected answer:

Delegate pattern is a variation of Observer pattern where only one object can observe events coming from another object. That effectively makes Delegate pattern a one-to-one relationship. Delegates are commonly used across iOS frameworks. Two of the arguably most commonly used examples would be `UITableViewDelegate` and `UITableViewDataSource`. Both of them are represented by a protocol that an object conforms to and UITableView uses

that single object provided to it to send messages/events. Unlike with Observer pattern there could be only one delegate object.

Delegation is sometimes abused by iOS developers. Be careful not to reassign your delegate object throughout the flow of your app, that might lead to unexpected consequences. Delegate/delegatee relationship is the one of tightly coupled nature.

4.12 What is KVO (Key-Value Observation)?

KVO is one of the core parts of Cocoa Touch and is used widely across the platform.

Expected answer:

`KVO` stands for Key-Value Observation and is a mechanics through which you can observe changes on properties on iOS. In contrast to Delegate KVO is one-to-many relationship. Multiple objects could subscribe to changes on a property of another object and as soon as it changes they all will be notified.

Under the hood implementation utilizes instance variables defined with properties to store the actual value of the property and setters/getters supplied by synthesization of those properties. Internally when you assign a property it will call `willChangeValueForKey:` and `didChangeValueForKey:` to trigger the change broadcast to observers.

Another way that KVO is used in iOS apps is public broadcasting of messages through `NSNotificationCenter`. The underlying mechanics are the same as with property KVO but the broadcasting can be triggered via a `post:` method on NSNotificationCenter default center rather than a property change.

Originally an Objective-C feature it is also available in Swift to classes subclassed from NSObject.

KVO on its own is a fairly bulky technology but it opens up a lot of possibilities to build on top of. There are a lot of great FRP projects like ReactiveCocoa and

RxSwift that were built using KVO mechanics.

4.13 What iOS application lifecycle consists of?

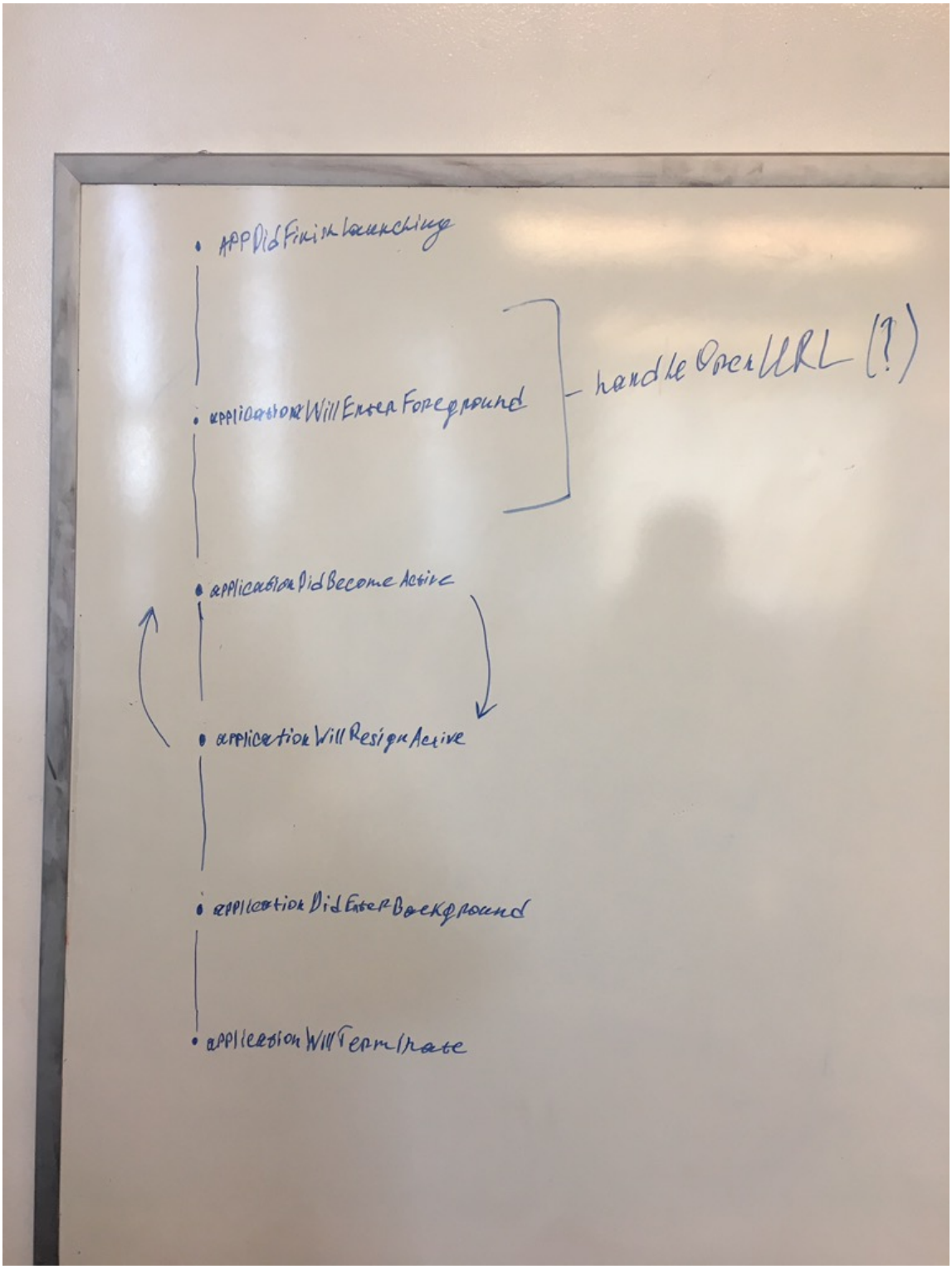
As iOS developers we simply have to know what's going on with the app we're building. Application lifecycle questions show how you understand iOS app's behavior overall in the system.

Expected answer:

The main point of entry into iOS apps is `UIApplicationDelegate`. `UIApplicationDel` is a protocol that your app has to implement to get notified about user events such as app launch, app goes into background or foreground, app is terminated, a push notification was opened, etc.

Lifecycle methods:

// TODO: needs better graphics for lifecycle here


• APPDidFinishLaunching
• applicationWillEnterForeground
handleOpenURL (?)
• applicationDidBecomeActive
• applicationWillResignActive
• applicationDidEnterBackground
• applicationWillTerminate

When the app is launched the first thing called is `application: willFinishLaunchin` `Bool`. This method is intended for initial application setup. Storyboards were loaded already at this point but state restoration hasn't occurred yet.

Launch

- `application: didFinishLaunchingWithOptions: -> Bool` is called next. This callback method is called when the application finished launching and restored state and can do final initialization such as creating UI.

- `applicationWillEnterForeground:` is called after `application: didFinishLaunchingWithOptions:` or if your app becomes active again after receiving a phone call or other system interruption.

- `applicationDidBecomeActive:` is called after `applicationWillEnterForeg` to finish up transition to foreground.

Termination

- `applicationWillResignActive:` is called when the app is about to become inactive (when iPhone receives a phone call or the user hits home button for example).

- `applicationDidEnterBackground:` is called when your app enters background state after becoming inactive. You have approximately 5 seconds to run any tasks you need to back things up in case the app gets terminated later or right after that.

- `applicationWillTerminate:` is called when your app is about to be purged from memory. Call any final cleanups here.

Both `application: willFinishLaunchingWithOptions:` and `application: didFinishLaunchingWithOptions:` can potentially be launched with options identifying that the app was called to handle a push notification or url or something else. You need to return `true` if your app can handle given activity or url.

Knowing app's lifecycle is important to properly initialize and setup your app and objects. You don't have to/should run everything in `application: didFinishLau` which quite often becomes a kitchen sync of setups and initializations of some sort.

4.14 What is View Controller? What is its lifecycle?

View Controllers are one of the core fundamental building units of Cocoa Touch applications. This question could be a part of or expansion on MVC question.

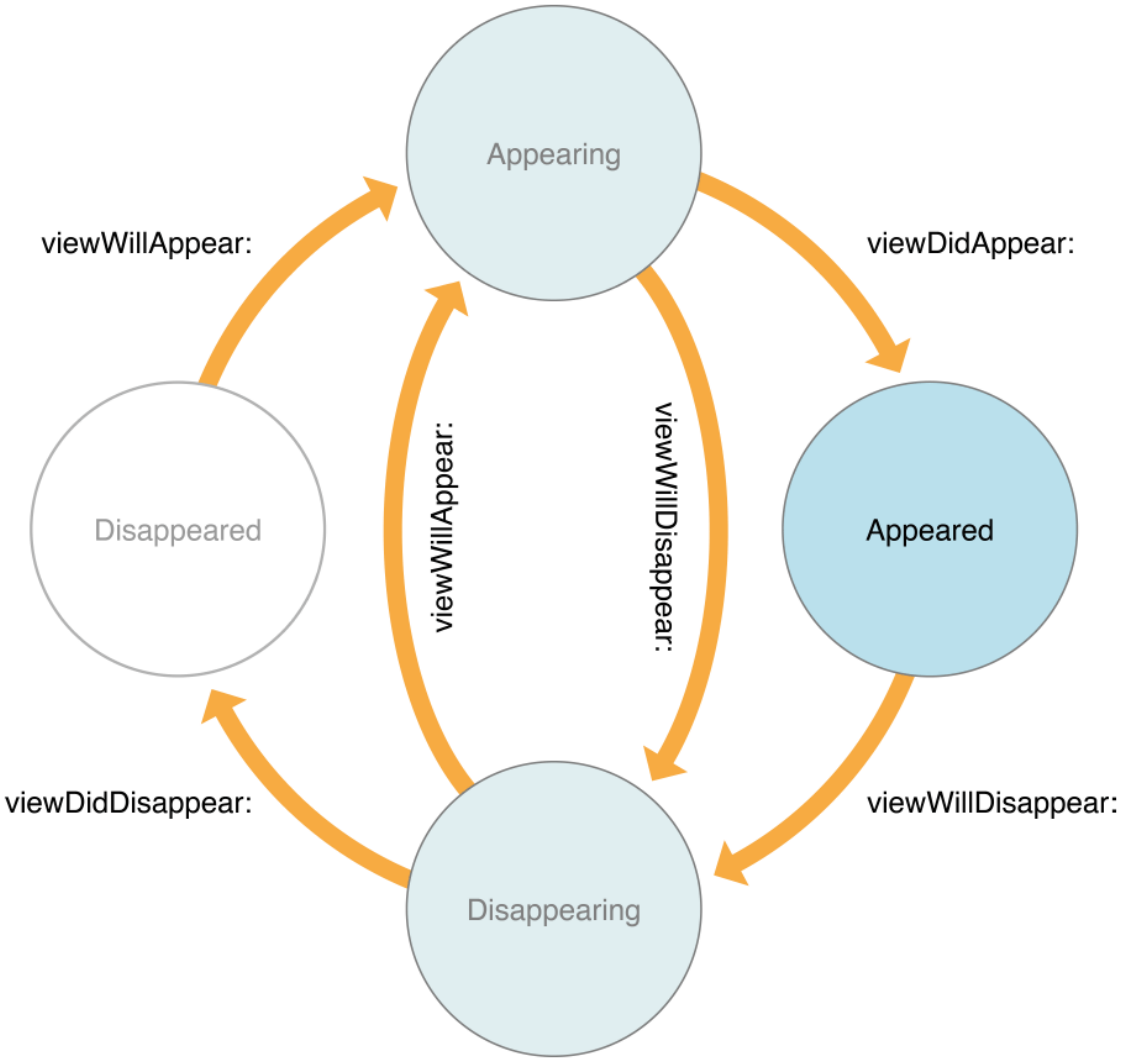
Expected answer:

View Controllers are the Controller part of MVC triangle. They are responsible for, which is clear from their name, controlling the view. Every time you want to display more or less significantly complex piece of UI you would want to use a View Controller to handle lifecycle and user interaction callbacks of that UI. In the nutshell a view controller is a simple subclass of `UIViewController` that has to have a view to draw the UI on. It is either attached to a `UIWindow` as the root view controller of that window or managed by a `UINavigationController` or other VC or system to be presented to the user.

At the end of the day when you develop iOS apps there are two main reasons you'd want to use View Controllers:

- get lifecycle callback for the view that the VC is managing (when the view was loaded, displayed, hidden, etc.)
- get handy built-in system integrations to present and dismiss VCs using `UINavigationController`, modal presentation, or parent/child containment API

View Controller lifecycle callbacks:



- **`loadView()`** you can override this method if you'd like to create the view for your VC manually.

- **`viewDidLoad()`** this method is called one time when your VC's view was loaded in memory for the first time. Do any additional setup and initializations here. Typically that's the method where most of your custom view initialization and autolayout setup will go. Also start your services and other async data related stuff here.

- **`viewWillAppear()`** this method is called when the view is about to appear on the screen. It will be called after **`viewDidLoad`** and every subsequent time after view disappears from screen and then appears again. For example, when you present a view in a navbar it will call **`viewDidLoad`** and then **`viewWillAppear/viewDidAppear`** for that VC. Later if you push a new VC on top of it **`viewWillDisappear`** and **`viewDidDisappear`** will be called because it's no longer the foreground/top VC. And then later if the user taps back button then **`viewWillAppear`** and **`viewDidAppear`** for that first VC will be called again because it became the top VC again. Use this method to do final UI customizations, hook up UI observers, etc.

- **`viewWillLayoutSubviews()`** is called right before **`layoutSubviews()`** is called in underlying UIView for that VC. Rarely used to adjust your view positioning.

- **`viewDidLayoutSubviews()`** is called right after **`layoutSubviews()`** is called in underlying UIView for that VC. Rarely used to adjust your view positioning.

- **`viewDidAppear()`** this method is called right after the view was show on the screen and follows **`viewWillAppear`** call.

- **`viewWillDisappear()`** is called when the view is about to become "hidden" i.e. not the top view controller presented to the user (see example above).

- `viewDidDisappear()` is called after `viewWillDisappear` and indicates that the view is "hidden".
- `didReceiveMemoryWarning()` is called when the system is low on memory and needs to release additional resources. Deallocate as much as you can here. Don't go crazy about it though, nowadays phones are so powerful that memory warnings rarely happen.

Additionally, view controller just like Views can be initialized either programmatically in code using `init...` constructor/initializer methods or loaded from a storyboard/xib file. In the former case, one of the initializer methods will be called and in the latter `-initWithCoder`.

Red flag:

Just like memory management you simply have to know this to be able to develop iOS apps.

4.15 Conclusion

You'll encounter fundamental question on every interview in various forms. These are the basics that are absolutely necessary to know and understand to do iOS development.

Chapter 5

Step Four. Get productive with networking.

Virtually every iOS app does some kind of networking. It's an integral part of our lives in this interconnect age of ours. Therefore you'll 100% be asked questions covered in this chapter on every interview you go. The depths and details may vary but overall every iOS developer should know how to handle networking and parse JSON data and how to structure iOS as a client-side app in general.

Alright, without further ado let's dive in!

Interview questions covered in this chapter:

- What is HTTP?
- What is REST?
- How do you typically do networking on iOS?
- What are the concerns and limitations of networking on iOS?
- What should go into networking/service layer?

- What is NSURLSession? How is it used?
- What is AFNetworking/Alamofire? How do you use it?
- How do you handle multi-threading with networking on iOS?
- How do you serialize and map JSON data coming from the backend?
- How do you download images on iOS?
- How would you cache images?
- How do you download files on iOS?
- Have you used sockets and/or pubsub systems?
- What is RestKit? What is it used for? Advantages and disadvantages?
- What to use instead of RestKit?
- How do you test network requests?

5.1 What is HTTP?

Even though you could think that this is a purely backend question it is very beneficial and even necessary for iOS developers to know what HTTP is and verbs it has with the meaning behind them. You won't be examined on theory and history of HTTP but you should be able to tell about the basics of the protocol that powers nowadays web.

Expected answer: HTTP stands for Hypertext Transfer Protocol and is the foundation of today's internet. What it means for us iOS developers is that when we build client-side applications we connect with backend APIs via HTTP. When we send requests to HTTP APIs we use "verbs" such as `HEAD`, `GET`, `POST`, `PATCH`, `PUT`, `DELETE`, etc. Each verb represents a differnt type of action you'd like the backend to do. You'd typically work with the following verbs in a properly implemented API:

- `HEAD` returns header information about a resource. Typically it has a status code (200, 300, 400, etc.) and caching details.
- `GET` returns actual data for the resource you requested. Typically it's your domain model data.
- `POST` is an action you do, well, to post something to your server. Typically used to submit data only.
- `PATCH` is used to change a resource's data. Unlike `PUT` it changes only certain values for the resource instead of overriding the whole thing.
- `PUT` is like `PATCH` but instead of altering only certain values in a resource it suppose to replace everything about the resource with the data you submit leaving only the unique `id` intact.
- `DELETE` not surprisingly destroys a resource on the backend.

iOS applications that communicate with server APIs using the above verbs can achieve most of the networking goals, except real-time connection/sockets, as long as the API adheres to HTTP standard and respects the meaning of those verbs. It is incredibly difficult to work with a backend that does some data changes on `POST` requests and returns some data on `PUT`s and so on. Contracts between server and client were made for the purpose of not only convenience but consistency and predictability.

Red flag: not knowing what HTTP is :) today's developers working with the web (and yes, as an iOS developer you do work with the web through requests to server APIs) simply can't afford not to know the fundamental meanings of HTTP verbs and expected server behavior using them.

5.2 What is REST?

`REST` stands for Representational State Transfer. REST is an API architecture build on top HTTP protocol. Its main idea is resources and the ability of client

applications access, write, delete, and alter them. As far as iOS developers concerned it is the most popular API architecture for third-party services and many internal product APIs. Knowing what it is and what it means is vital for iOS app development.

Expected answer: `REST` is an API architecture that revolves around resources and HTTP verbs. Each resource is represented by a set of endpoints that can receive some of the HTTP verb requests to do various `CRUD`(Create, Read, Update, Destroy) operations. For example, let's say you have an API that lets you manage `posts` user creates in your app. A typical CRUD REST API for it would look like this:

- `https://yourawesomeproduct.com/posts` accepts `GET` requests and returns back a list of `posts` available on the server
- `https://yourawesomeproduct.com/posts/123` accepts `GET` requests and returns back a single `post` with given `id` (123) available on the server
- `https://yourawesomeproduct.com/posts` accepts `POST` requests to create new `post` objects with the data provided by iOS client application
- `https://yourawesomeproduct.com/posts/123` accepts `PATCH` requests to alter certain data in a specific `post` with given `id`
- `https://yourawesomeproduct.com/posts/123` accepts `PUT` requests to replace entire set of data in a specific `post` with given `id`
- `https://yourawesomeproduct.com/posts/123` accepts `DELETE` requests to destroy a `post` with specified `id`

RESTful APIs also suppose to return the right status codes in response to your requests such as `200` for successful `GET` request or `201` for successful `POST`, etc.

If the API you're using is truly RESTful than it will be predictable and easy to work with. Again, protocols and contracts in software development were made not only for convenience but for reliability as well.

Red flag: you should be able to have at least a basic idea what `REST` and `RESTful` backends are.

5.3 How do you typically do networking on iOS?

This is a general networking question that could prompt and imply either big picture architectural discussion about decoupling and single responsibility around APIs on iOS or some tactical specific things on how you would implement networking/service layer in your applications. It's up to you where to steer the discussion.

Expected answer: Networking falls into `Service Layer` of your application since it does external communication. In general, you should decouple everything HTTP/network related in your app into a set of service and client objects that have all the nitty gritty details of HTTP connection. Those objects would perform requests and API calls for your application decoupling it from other layers of responsibility (like storage, business logic, UI, etc.) of the app.

A typical small "starter" implementation of a service layer in your app could look like this:

- a networking/HTTP manager of some kind (either `NSURLSession` or `AFNetworking/Alamofire` manager).
- an `APIClient` object that can be configured with a networking manager to actually perform HTTP requests against your API domain. APIClient usually is responsible for signing every request with authentication token/credentials.
- a set of `Service` objects that work with individual resources of your RESTful API such as `PostsService`, `UsersService`, etc. These ser-

vice objects use shared `APIClient` to issue specific concrete HTTP requests to their respective `/posts` and `users` endpoints. They compose params and other necessary data for requests.

At the end of the day, all other parts of the app are working directly only with `service` objects and never touch low-level implementation such as `APIClient` or `NSURLSession/AFNetworking/Alamofire`. That separation of concerns ensures that if your authentication or individual endpoints change they won't affect each other in your codebase.

Red flag: simply saying that you use `NSURLSession` and issue requests in View Controllers when necessary isn't gonna cut it. These days `AFNetworking` and `Alamofire` are de facto the standard for doing HTTP networking on iOS and following Single Responsibility Principle is vital for codebases big or small.

5.4 What are the concerns and limitations of networking on iOS?

This question aims to gauge your understanding of constraints of networking on iOS.

Expected answer: the main networking constraints on iOS are battery and bandwidth. iOS devices have limited battery capacity and sporadic network connection that can drop in and out frequently. Developing networking layer of the app you should always issue as little HTTP requests as possible and retry requests if they suddenly fail due to poor connection or other issues.

There's also bandwidth issue, it is not a good idea to upload or download large files and chunks of information when on cellular connection and is advised to use WiFi instead.

5.5 What should go into networking/service layer?

This is a conceptual and architectural question. Every application consists of several layers of responsibility and Service Layer is responsible for all the external data communication. You are asked about this to gauge your level of understanding of what is going to services and networking layer in iOS apps according to SRP (Single Responsibility Principle).

Expected answer: every iOS app that works with external data has a `Service Layer` that is responsible for communication with things like HTTP APIs, GPS location, BLE peripherals, Gyroscope, iCloud, sockets, etc. All of those are external to your app resources and to work with them you need a set of objects that can communicate with those resources (for example HTTP Client or BLE manager) and can serialize/deserialize data sent to or received from those resources.

Here is a typical service layer that does networking with some kind of API:

- `APIClient` object that has an HTTP manager
- `PostsService` object that owns `APIClient` and issues requests to specific endpoints to POST and GET `posts`. `PostsService` maps JSON data to your custom domain model objects
- `Post` class that subclasses from `MTLModel` to map JSON received by `PostsService` to your custom `Post` objects

And here's what will go in the same service layer for BLE (Bluetooth Low Energy):

- `BLEClient` object that owns and manager `CBCentralManager` and executes low-level connection to BLE peripherals
- `PeripheralsClient` object that discovers peripheral services, characteristics, and executes low-level stuff to get and send values to and from peripherals

- `SpecificDeviceService` that uses both `BLEClient` and `PeripheralsClient` to orchestrate connection to BLE, discovery, and communication with specific device/peripheral you're trying to connect to. `SpecificDeviceService` is also responsible for mapping data received from `Characteristic` to you custom objects
- `CustomCharacteristicData` is just like `Post` in case of JSON API is a domain model object that is mapped from raw data received from BLE to conveniently work with it throughout your application

As you can see both HTTP and BLE examples are similar in what they do. They wrap some kind of external service (HTTP or BLE in this case) and make it convenient and easy to work with it. At the end of the day, your application is going to interact only with `PostsService` and `Post` objects to do it's API networking, and with `SpecificDeviceService` and `CustomCharacteristicData` objects to work with external BLE devices. Low-level implementation details like HTTP GET/POST requests and BLE connection, peripheral and characteristics discovery, all hidden behind those class interfaces. It makes it robust and reliable and separates that low level, unimportant, logic from the business logic of your app.

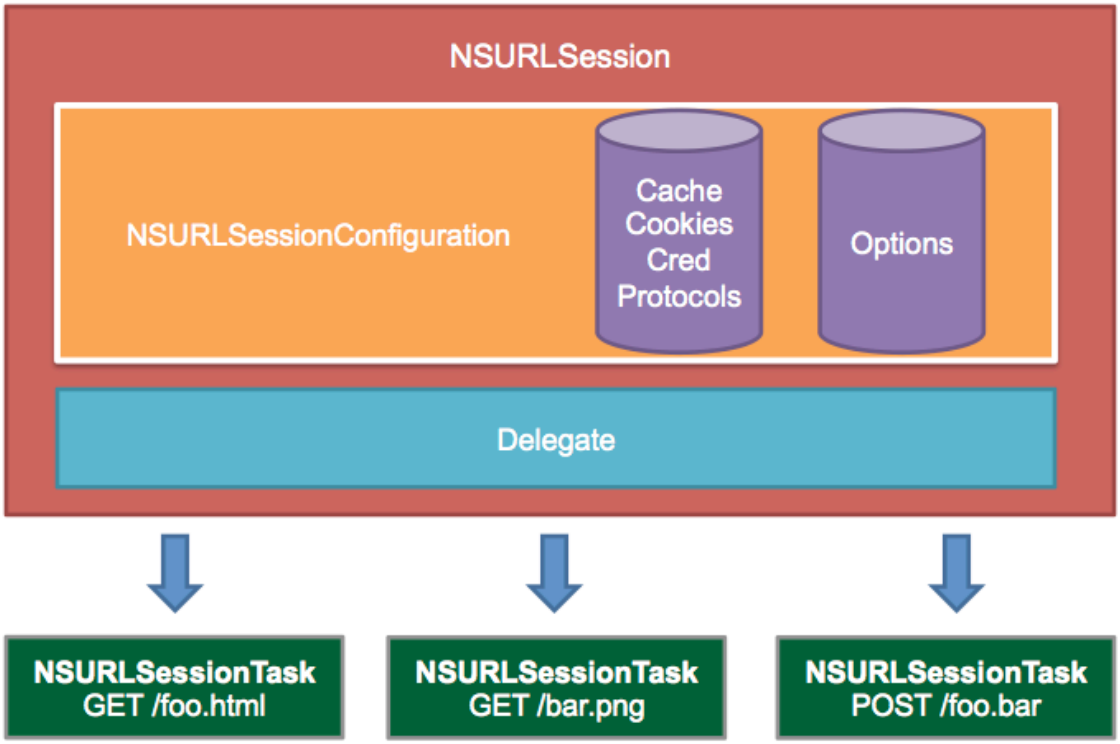
Red flag: simply saying that service layer has only HTTP client and that you create HTTP requests "when needed" for each endpoint isn't a red flag perse but you should show deeper architectural understanding of separation of concerns in iOS apps.

5.6 What is NSURLSession? How is it used?

That's one of the basic iOS networking questions. Go into deep details only if asked.

Expected answer: since iOS 7 `NSURLConnection` became obsolete and the new Apple standard for implementing HTTP networking is `NSURLSession`. `NSURLSession` and related classes do a lot of heavy lifting for basic HTTP

connection and authentication for you. It allows you to send HTTP verb (GET, POST, etc.) requests, connect FTP, and to download files. You can optionally configure cache and execute your requests in background/app suspended state. Overall the structure of `NSURLSession` related things looks like the following:



The way you typically work with it is to use `NSURLSessionDownloadTask` objects to execute requests against given urls. It has a block based and delegate based API which means there are two ways you can issue HTTP requests with `NSURLSession`: either by receiving a completion handler block callback or by implementing delegate methods and receiving notifications as the data comes in. Either way is fine and has a different purpose depending on your use case (for example if you'd like to receive download progress notification you would want to implement delegate callbacks rather than completion block). Also `NSURLSession` allows you to resume, cancel, or pause networking task.

All and all `NSURLSession` is a very robust way of doing HTTP and other networking but in reality, it is a bit too low level and in the majority of the cases

you're better off using a wrapper library like `AFNetworking` or `Alamofire`. Both of them are de facto the standard for networking on iOS and use `NSURLSession` under the hood to run HTTP requests for you.

Red flag: even though these days we all use `AFNetworking` and `Alamofire` it is beneficial to know what's going on under the hood and how conceptually `NSURLSession` works.

5.7 What is AFNetworking/Alamofire? How do you use it?

`AFNetworking` and `Alamofire` became de facto the standard for networking on iOS. Expect this question on every interview you have.

Expected answer: `AFNetworking` and `Alamofire` are wrappers around standard Apple iOS technologies for networking such as `NSURLSession` that make working with it more convenient and reduce the boilerplate setup you have to do when you work with `NSURLSession` directly. Nowadays `AFNetworking` and `Alamofire` are de facto the standard of how you do HTTP networking on iOS and probably the most commonly used 3rd party library. As a wrapper around Apple's `NSURLSession`, it has access to pretty much every feature it provides and more. Overall it takes care of HTTP requests, JSON data serialization into `Dictionary` objects, response caching, and status code response validation. With it, you can setup HTTP request headers, params, issue HTTP GET/POST/PUT/etc. requests, serialize JSON response, do basic HTTP authentication, upload and download files, etc.

Alamofire has a block based API. You use it either directly with minimal setup by creating requests using `Alamofire` class methods or by creating a session manager object (and providing it with `URLSessionConfiguration`) that can take callback blocks.

Here's an example of a typical minimal setup request:

```
Alamofire.request("https://httpbin.org/get").responseJSON { response in
    print(response.request)  // original URL request
    print(response.response) // HTTP URL response
    print(response.data)     // server data
    print(response.result)   // result of response serialization

    if let JSON = response.result.value {
        print("JSON: \(JSON)")
    }
}
```

And this is how you'd setup a session manager and use it to send requests:

```
let configuration = URLSessionConfiguration.default
let sessionManager = Alamofire.SessionManager(configuration: configuration)

sessionManager.request(urlString, method: .post, parameters: parameters,
                       encoding: JSONEncoding.default)
            .responseJSON { [weak self] response in

                if let json = response.result.value as? [String: String] {
                  // do something if it was success
                } else {
                  // do something if it was failure
                }

            }
```

Red flag: Alamofire and AFNetworking are the workhorses for today's HTTP networking on iOS and every developer should be familiar with it. You can get away with not knowing about them only if you're very good with **`NSURLSession`**.

5.8 How do you handle multi-threading with networking on iOS?

Multi-threading is very important when you work with networking on mobile devices. Blocking the main thread making your UI unresponsive for the duration of HTTP requests long time is not an option. This question most likely will be asked in every interview in one form or another.

Expected answer: the general idea with any kind of multi-threading on iOS is that you don't want to block the main UI thread. That means that every HTTP or other service/networking layer request should be executed on a background thread. In fact, some of iOS system frameworks will complain and print logs or crash if you use them not on the main thread (Autolayout for example). There are various mechanics in iOS and third party libraries to help you with this but the most common solutions are **GCD**, **NSOperation**. Most of the third party libraries (i.e. **Alamofire** and **AFNetworking**) and **NSURLSession** already have threading mechanics built in and execute their requests on a background thread and call completion blocks on the main.

GCD is a low-level library for managing threading and queues on iOS. It has a C-based interface (with Swift 3 it has finally become an object based API) and is very powerful. You'd use it in conjunction with **NSURLSession** for example. All the HTTP requests the **NSURLSession** does are executed on a background thread and it could be either configured to execute completion callback on the main thread or on a background thread. Also if your completion callback is executed on a background thread but you need to do some UI updates you can use **GCD** blocks like this:

```
dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{

    NSURLRequest *request = [NSURLRequest requestWithURL:
                             [NSURL URLWithString:@"http://smartcloud.io/"]];
    NSURLResponse *response;
    NSError *error;
    NSData *data = [NSURLConnection sendSynchronousRequest:request
                                         returningResponse:&response
                                                     error:&error];
    if (error) {
        // handle error
        return;
    }
    dispatch_async(dispatch_get_main_queue(), ^{
        // update your UI safely
    });
});
```

Red flag: these days every developer should know that you shouldn't block the main thread with background operations such as HTTP networking. Not

explaining what the issue is will definitely raise a red flag.

5.9 How do you serialize and map JSON data coming from the backend?

JSON serialization and mapping is a common task when you're doing HTTP networking with an API on iOS. Expect this question either as a standalone one or as a part of other HTTP/networking questions and follow-ups.

Expected answer: every time you receive JSON or XML or any other kind of response from a backend API that you most likely get it in a JSON or binary or other "inconvenient" format. The first thing you need to do to be able to work with the data you've received is to serialize it in something your app understands. At the most simplest and basic level that would be a dictionary or array of objects containing other dictionaries, arrays, and primitives from that response. So let's say if your JSON response looks like this:

```
{
  post: {
    title: 'This is an awesome post!',
    body: 'loads of text .....',
    tags: ['Awesomeness', 'Coolness', 'Greatness!'],
  }
}
```

Then a serialized object from it is going to look like this:

```
var post = Dictionary<String, AnyObject>();

post["title"] = "This is an awesome post!" as AnyObject?
post["body"] = "loads of text ....." as AnyObject?
post["tags"] = ["Awesomeness", "Coolness", "Greatness!"] as AnyObject?

print(post)

========== print output
["tags": <Swift._SwiftDeferredNSArray 0x60000003abe0>(Awesomeness, Coolness, Greatness!),
```

```
"body": loads of text .....,
"title": This is an awesome post!]
```

Obviously in the example above we've created the Dictionary manually ourselves but this is exactly what something like **`NSJSONSerialization`** would do for you. **`NSJSONSerialization`** is the go-to tool for JSON to dictionaries/arrays/primitives serialization.

Data serialization is only the first piece of the puzzle working with JSON data. The other piece is data mapping. Even though now we have a serialized dictionary object that represents the post data that we've received from the backend it's still just a **`Dictionary`** and is a poor choice for us to work with throughout the app. We need a better **`domain model object`** and this is where data mapping comes into play. In order for us to work with our own custom domain model objects and classes that dictionary needs to be mapped into those custom classes. You can either do it manually yourself, take a dictionary object, take values for each key and assign them to properties on your custom **`Post`** class or struct object. But that is a very tedious and error-prone boilerplate code that leads to a lot of errors and human mistakes. A better solution for this would be a library such as **`Mantle`** or **`ObjectMapper`**. Both of those help you declare your key/property mapping and automate the process. As the end result you'd get your custom domain model objects crafted specifically for the tasks your application does reducing errors.

Red flag: too many developers neglect proper data mapping in their applications. Understanding what serialization and mapping are, what is the difference between them, and why it is important to have a well-defined domain models will set you apart from other devs.

5.10 How do you download images on iOS?

A lot of iOS applications work with images fetched from the web. This is a typical networking interview question that also touches upon data storage and

caching a little.

Expected answer: as with most of the networking things there's a manual naive way and there are automations, libraries, and best practices that you can utilize. As the most basic implementation of image downloading on iOS you could use good old Apple's `NSURLSession` and download an image at a given url as binary data, convert it from `NSData` to `UIImage`, and then display it in a `UIImageView`. That will work but is too raw and inefficient and has a lot of performance implications.

There are two main things you need to worry about when working with images fetches from the web on iOS: downloading and caching. Downloading involves actually issuing an HTTP request to get the raw image data from a server. And caching is concerned with storing downloaded images to disk, database, or in-memory cache (or all of them in a combination).

At the end of the day you're better off using a library that takes care of a lot image downloading and caching boilerplate for you. Typical options to pick from are `AFNetworking` or `Alamofire` themselves or a powerhouse when it comes to image download and caching `SDWebImage`. `SDWebImage` gives you a lot of flexibility with your image downloading and caching and provides a set of `UIImageView` extension methods that you can call on it to download images for given urls along with placeholder images and download progress reporting.

5.11 How would you cache images?

Image caching is important for every iOS application that fetches graphics from the web. Due to mobile device constraints on memory, battery, and bandwidth, it is important to cache images and be efficient with it.

Expected answer: when it comes to caching images there are really 3 ways you can go about it: in-memory cache, database storage, disk storage.

`In-memory cache` could be as simple as `Dictionary` that keeps a reference

to `UIImage` objects and uses urls they were downloaded from as a unique key or it could actually be `NSCache` that performs similar stuff for you and also can be configured.

`Database storage` is used for image caching when you save downloaded image binary to `Core Data` or `Realm` or similar database. Typically this is used for images of a very small size because databases were not necessarily made to handle large files stored in them. The best use case for that is small thumbnail images.

`Disk storage` is what you expect that is - storing downloaded files to the disk for later quick retrieval instead of another fetch from the server. Files usually stored with a unique name identifier to quickly look them up.

Ultimately the best solution for caching is going to vary case by case but a lot of apps either use `SDWebImage` or similar library for that or roll their own solution as a combination of in-memory, database storage, and disk storage.

5.12 How do you download files on iOS?

File download is a common task for iOS apps. It could be PDF or image or video file that your app needs to download. And as usual, this question gives you an opportunity to either go over the basics or to dig deeper and explain different techniques there are for downloading files.

Expected answer: at the very basic level file download is just a bunch of bites you fetch from a url over HTTP somewhere on the web. Either `NSURLSession`'s `NSURLSessionDataTask` or `Alamofire`'s download GET request will do the trick. When you get the data there are 3 ways you can deal with it. You either: 1. work with received data right there in-memory in the callback where you received it. 2. store received file in a temporary folder for later use. 3. store received file permanently on the disk.

1. working the file right after you received it is the easiest. You have it in `NSData` form and it's accessible without any further ceremony. The

disadvantage though is that you can work with only small size files in that fashion. If a file is too big it could take too long to download and too expensive to handle it in the callback block.

2. storing received file in a temp folder is a mid-ground solution that typically is the best compromise and a great way to handle the data you just downloaded. Storing it temp folder allows you postpone handling the file to a later time and lets you move on in your download callback block. You can count on files to be in temp folder only for the duration of your app running, though.

3. storing received file on disk allows you to access it later and sometimes is the only way to handle downloaded files when they are too big to operate on in the download callback block.

Red flag: you should have at least a basic understanding of how to download files on iOS.

5.13 Have you used sockets and/or pubsub systems?

This question isn't as typical for most of the teams and companies but those that work with messaging/chat applications most likely are going to ask you this question. Pub/sub systems are growing in popularity for solving problems other than chat/messaging on iOS thought, so it is beneficial for iOS developers to be at least familiar with the topic.

Expected answer: sockets is a specific technology for persistent connection communication and you can think of it as a subset of pub/sub systems. Sockets and pub/sub systems such as `Pubnub` allow you to build apps that can connect and observe external data streams and react and process received data in nearly real-time.

Consider this example - you're building an app similar to Facebook Messenger. In that app you have your normal view controller with a list of chats you have

open and when you open a specific chat it will open another view controller for that chat. This new VC with a specific chat then subscribes to a channel using sockets or Pubnub or other pubsub system. As soon as it's subscribed to its chat channel it will receive the latest batch of messages since last connection and then will start receiving and sending new messages in real time as participants of that chat type them. That is the general idea of how chat/messages applications work.

5.14 What is RestKit? What is it used for? Advantages and disadvantages?

RestKit used to be a very popular data synchronization framework that is used by many companies especially with legacy codebases. It is not as popular these days but if you're joining a team that has to support that legacy technology then expect to be asked about RestKit.

Expected answer: RestKit is a framework that was made for a purpose of data synchronization between client iOS applications and RESTful web services. RestKit has several responsibilities it takes onto itself such as:

- HTTP url composition and building (Routing),
- HTTP GET/POST/etc. requests sending and enqueueing,
- JSON request and response serialization,
- JSON response parsing and mapping,
- Core Data synchronization with mapped from JSON domain models received from the backend,
- POSTing/PUTing/etc. domain models created locally and synced with Core Data with a remote RESTful service.

As you can see it takes on quite a lot :)

At the end of the day the reason RestKit became obsolete and virtually is not used anymore on new projects because it was doing too much for you and forced you into a convoluted API that it has. RestKit is so big that it can have an entire book written about it. If you want to learn more or unfortunately have to support it on a legacy project head over to **restkit on github** to dig deeper into it.

As an alternative to RestKit you're better off rolling your own solution for data synchronization. RestKit's downfall was its breaking SRP (Single Responsibility Principle). Choose wisely what features and functionality you need from your libraries and how they should be used in your applications.

Red flag: you don't have to have experience with RestKit these days but it is very beneficial to have an overview and understanding of what it offers and does for you so that you make a concious decision to pick or avoid it.

5.15 What to use instead of RestKit?

Since RestKit is practically obsolete these days you could be asked what are the alternatives you can use instead of RestKit to synchronize data with backend APIs.

Expected answer:

You have several options instead of RestKit to use for data synchronization with backend APIs:

- Overcoat
- Roll-Your-Own-Solution

Overcoat is another library that takes care of a lot of things for you like RestKit but unlike RestKit its API is way easier to use. It takes care of routing,

HTTP requests, JSON response parsing, object mapping from JSON to custom objects, object mapping from custom objects to Core Data, promises API out of the box. It takes on a lot of responsibilities just like RestKit and therefor not advisable to use for every app.

But the better option is to roll your own solution. If you think about it - everything that RestKit does is more or less necessary for any complex enough iOS application. Things that it does can be implemented using other libraries and tools available. For example:

- HTTP url composition/routing can be implemented as a simple custom url builder
- HTTP GET/POST/etc. requests sending and enqueueing can be handled by `AFNetworking` and `Alamofire`
- JSON request and response serialization is taken care of by `NSJSONSerialization` and/or `Alamofire/AFNetworking`
- JSON response parsing and mapping can be handled by library like `Mantle`
- Core Data synchronization and mapping from/to custom domain models can be taken care of by `Mantle`

And that's everything that you need. Rolling your own solution, and only when you currently need will also help you evolve your codebase gradually without introducing things with unnecessary functionality and baggage. You just need to know what you need.

5.16 How do you test network requests?

Unit and integration testing becoming more and more popular as tools for testing evolve in iOS ecosystem. This question carries with it a lot of baggage of opinions and unfortunately still is somewhat controversial in iOS community.

Expected answer: in general client-side applications do not integration test network requests, they only do unit-testing or mock them if really necessary. The reason be is that it is not common to have a dedicated server for unit-testing that can receive and adequately respond to those test requests. And also there's a challenge of keeping it in sync with the current client-side codebase.

Typically an `OCMock` based libraries like `Cedar`, `Quick`, `Specta`, `Expecta` are the go to tools for unit-testing on iOS.

5.17 Conclusion

Service and networking related questions are 100% going to be asked on every iOS interview. Networking is the building block of pretty much any iOS application these days, this is what makes apps useful - the ability to connect to external services and internet. In order to be a good iOS citizen and create efficient apps that don't waste bandwidth and sync data just in time you should know your options and know what do you really need to accomplish your task.

Chapter 6

Step Five. Learn how to store data.

Interview questions covered in this chapter:

- What Storage Layer is for in iOS applications?
- What can you use to store data on iOS?
- What is NSUserDefaults?
- How and when do you use NSUserDefaults?
- What are advantages and disadvantages of NSUserDefaults?
- What is NSCoding?
- What is Keychain and when do you need it?
- Does Keychain persist data when your app is uninstalled?
- How do you save data to disk on iOS?
- What disk storage is good and bad for?

- What database options there are for iOS applications?
- What is Core Data? What are the advantages and disadvantages of Core Data?
- How do you use Core Data?
- How data mapping is important when you store data?
- How would you approach major database/storage migration in your application?

Chapter 7

Step Six. Go crazy responsive with UI layouts.

Quite often creating UI is one of the biggest parts of iOS project. Being able to make it right for different screen sizes is a very crucial skill for any kind of project and team. Back in the day, we were only able to do frame size calculations and a little bit of auto-resizing masks. These days we have AutoLayout. The problem is it is notoriously difficult to work with and especially debug.

But there's hope - there are libraries like Masonry that help you declaratively define your AutoLayout constraints in code.

This chapter is going to be especially useful if you're applying for a company that is heavy on UI and values design a lot.

Interview questions covered in this chapter:

- What are the challenges working with UI on iOS?
- What do you use to layout your views correctly on iOS?
- What are CGRect Frames? When and where would you use it?
- What is AutoLayout? When and where would you use it?

- What compression resistance and content hugging priorities are for?
- How does AutoLayout work with multi-threading?
- What are the advantages and disadvantages of creating AutoLayouts in code vs using Storyboards?
- How do you work with Storyboards in a large team?
- How do you mix AutoLayout with Frames?
- What options do you have with animation on iOS?
- How do you do animation with Frames and AutoLayout?
- How do you work with UITableView?
- How do you optimize table views performance for smooth fast scrolling?
- How do you work with UICollectionView?
- How do you work with UIScrollView?
- What is UIStackView? When would you use it and why?
- What other alternative ways of working with UI do you know?
- How do make pixel-perfect UI according to designer's specs?
- How do you unit and integration test UI?

7.1 What are the challenges working with UI on iOS?

This question is typically asked to asses whether you understand that it's not that simple and straightforward to do UI on iOS anymore. Now we have multiple screen sizes and resolutions, not to mention iPad and Multi-Tasking support

where your views and view controllers can be displayed in various forms and formats.

Expected answer: show them that you are aware of responsive and adjustable nature of iOS UI. There are several things you as a developer need to be concerned with developing UI for iOS:

- multiple screen sizes/dimensions for iPhone 5, 6, 6 Plus, etc.
- multiple screen sizes/dimensions for iPads
- potential reusability of UIViews between iPhone and iPad
- adaptability of your UI to resizable views used for multi-tasking feature on iPad (i.e. size classes)
- UI performance, especially with dynamic content of various sizes in `UITableViews` and `UICollectionViews`

All of the above concerns show that you are aware of the issues. Also, it is good if you mention here that Apple has AutoLayout to address a lot of the challenges related to UI scalability and that it is a replacement of the previously used Frames and auto-resizing masks approach. These answers will likely make your interviewer steer towards Frames vs AutoLayout discussion.

Table views and Collection views performance is especially important for social networking applications for example. They typically have content of arbitrary size posted by users that need to be displayed in lists. The challenge there is to quickly calculate cell and other UI elements sizes when the user scrolls the content quickly. Mentioning that will most likely prompt your interviewer to ask probing questions about Frames, AutoLayout, and `UITableView`/`UICollectionVie`w questions.

Red flag: not mentioning various iPhone/iPad screen sizes and not mentioning AutoLayout as one of the solutions most likely is going to raise a flag.

7.2 What do you use to layout your views correctly on iOS?

Knowing your options for laying out things on the screen is crucial when you need to solve different UI challenges on iOS. This question helps gauge your knowledge about how you put and align views on the screen. Answering this question you should at least mention `CGRect` Frames and AutoLayout but it would be great to mention other options such a `ComponentKit` and other Flexbox and React implementation on iOS.

Expected answer: go to options for laying out views on the screen are good old `CGRect` Frames and AutoLayout. Frames along with autoresizing masks were used in the past before iOS 6 and are not a preferred option today. Frames are too error-prone and difficult to use because it's hard to calculate precise coordinates and view sizes for devices of various size and form.

Since iOS 6 we have AutoLayout which is the go-to solution these days and is preferred by Apple. AutoLayout is a technology that helps you define relationships between views, called constraints, in a declarative way letting the framework calculate precise frames and positions of UI elements instead.

There are other options for laying out views such as `ComponentKit`, `LayoutKit` that more or less inspired by `React`. These alternatives are good in certain scenarios when, for example, you need to build a highly dynamic and fast table views and collection views. AutoLayout is not always perfect for that and knowing that there are other options is always good.

Red flag: not mentioning at least AutoLayout and the fact that Frames are notoriously hard to get right is definitely going to be a red flag for your interviewer. These days no one in their sane mind would do `CGRect` frame calculations unless it is absolutely necessary (for example when you do some crazy drawings).

7.3 What are CGRect Frames? When and where would you use it?

This question is asked to learn if you have a background in building UI "the hard way" with using view position and size calculation. Frames were used previously before AutoLayout to position UI elements on the screen but these days there are other options you have to solve that problem. The interviewer is trying to figure out how advanced you are in and how well you know a lower level of UI rendering.

Expected answer: The simples way to position views on the screen is to give them explicit and specific coordinates and sizes with `CGRects`. `CGRect` is a struct that represents a rectangle that a view is placed at. It has `origin` with `x` and `y` values, and `size` with `width` and `height` values. They represent upper-left corner where the view starts to draw itself and width and height of that view respectively. Frames are used to explicitly position views on the screen and have the most flexibility in terms of what and how you position on the screen. But the disadvantage is that you have to take care of everything yourself (with great power comes great responsibility, you know). Meaning even though you're in full control of how your UI is drawn on the screen you will have to take care of all the edge cases and different screen size and resolutions.

A better option these days is AutoLayout. It helps you with layout positioning through constraints and sets specific frames for views for you. It makes your views scalable and adaptive to different screen sizes and resolutions.

Red flag: AutoLayout is de facto the standard of doing layouts these days, Frames are considered to be an outdated concept that is very error prone. Saying that frames are perfectly fine for laying out views would raise a red flag because most likely your interviewer would think that you don't know how to build adaptive and responsive UI.

7.4 What is AutoLayout? When and where would you use it?

This is very common UI related question on any interview. Virtually no interview will go without it. AutoLayout is one of the fundamental technologies that Apple pushed on for some time and now it is de facto the standard. Your interviewer either is expecting a very short and brief answer to get an understanding if you're versed in the topic or they'll drill down and ask you all the details about it. Be prepared for both.

Expected answer: AutoLayout is a technology that helps you define relationships between views, called constraints, in a declarative way letting the framework calculate precise frames and positions of UI elements instead. AutoLayout came as an evolution of previously used `Frames` and auto-resizing masks. Apple created it to support various screen resolutions and sizes of iOS devices.

In the nutshell using AutoLayout instead of setting view frames you'll create `NSLayoutConstraint` objects either in code or use nibs or storyboards. `NSLayoutConstraints` describe how views relate to each other so that at runtime UIKit can decide what `CGRect` frames specifically to set for each view. It will adjust to different screen sizes or orientation changes based on the "rules" you defined using constraints.

The main things you'll be using working with AutoLayout are: `NSLayoutRelation`, `constant`, and `priority`.

- `NSLayoutRelation` defines the nature of a constraint between two UI items/views. It can be `lessThanOrEqual`, `equal`, or `greaterThanOrEqual`.
- `constant` defines constraint value. It usually defines things like the distance between two views, or width of an item, or margin, etc.
- `priority` defines how high of a priority a given constraint should take. Constraints with higher priority number take precedence over other. Priority typically is used to resolve conflicting constraints. For example,

when there could be an absence of content we may want to align elements differently, in that scenario we'd create several constraints with different priority.

Bonus points: working with Apple's API for constraints in code is sometimes problematic and difficult. There are several different open source libraries out there that can help with it such as `Masonry` and `PureLayout` that dramatically simplify the process.

Red flag: AutoLayout is de facto the standard today for developing UI on iOS, disregarding it or trying to prove that Frames approach is better most likely is going to raise a red flag. There are alternatives of course but most likely your interviewer expects you to be very familiar with the technology since it's such a vital part of any iOS application.

7.5 What compression resistance and content hugging priorities are for?

This is an advanced question about AutoLayout typically asked along with other questions around constraints.

Expected answer:

- compression resistance is an AutoLayout constraint that defines how your view will behave under the pressure of other constraints demanding its resizing. The higher compression resistance is the less chance it's going to "budge" under other constraint's pressure to compress it.
- hugging priority is the opposite of compression resistance, this constraint defines how likely the view to grow under pressure from other constraints

Red flag: you should be familiar with these constraints if you worked with AutoLayout extensively.

7.6 How does AutoLayout work with multi-threading?

Pretty much every iOS application these days has some kind of multi-threading. This question is asked to gauge your general understanding of how to work with the main thread and background threads and with UI in particular.

Expected answer: all UI changes have to be done on the main thread. Just like working with `Frames` working with `AutoLayout` is UI work and it needs to be performed on the main UI thread. Every AutoLayout constraints addition or removal or constant change need to be done on the main thread. After you change constraints call `setNeedsLayout` method.

Red flag: saying that you can change AutoLayout constraints in any thread will raise a red flag.

7.7 What are the advantages and disadvantages of creating AutoLayouts in code vs using Storyboards?

This question is sometimes asked by bigger teams because they experience particular challenges when it comes to working with UI using storyboards. There's no "right" or "wrong" answer here, every approach has its advantages and disadvantages.

Expected answer: working with AutoLayout in code is considered to be more typical and Apple pushes a lot of examples showing how to do that. The **advantages** are so that it's visual, drag-n-drop/plug-n-play-able, and you can in some scenarios actually render your UI in Interface Builder without actually running the app and waiting for the entire build process to happen. Neat. But the **disadvantages** are very apparent when you need to debug your constraints or work in a team of more than two people. It is difficult to tell what constraints need to be there and what need to be removed at a glance. And quite often teams working with one storyboard modify it in different `git` branches

causing merge conflicts.

On the other hand **advantages** of defining AutoLayout in code is that it's very explicit, clear, merge conflict free. **Disadvantages** on the other hand is that it's difficult to work with Apple's AutoLayout constraints API in code (it can be helped if you use a library like `Masonry`) and you have to compile your app to see the results of rendering.

7.8 How do you work with Storyboards in a large team?

This question is asked by bigger teams. They especially suffer from a poor support of team development from Apple tools.

Expected answer: the main problem working with storyboards in a big team is `.storyboard` file merge conflicts. When two developers change the same storyboard in different branches they most likely will have a merge conflict. The benefits a unified monolith storyboard gives are quickly outweighed by the struggle teams experience with those merge conflicts. There are two solutions:

1. don't use storyboards and define your AutoLayout in code.
2. split your monolithic storyboard into multiple storyboards, typically one per view controller. That way storyboard changes will happen only when one view controller is modified which likely will help you avoid most of the merge conflicts.

7.9 How do you mix AutoLayout with Frames?

This question could be asked by a team that has an existing application and they are trying to either migrate to AutoLayout fully or to support both Frames and AutoLayout at the same time for legacy reasons.

Expected answer: AutoLayout and Frames can coexist together only in scenarios when you're not mixing them up directly. Basically, you can have a superview lay out itself and its subviews using constraints and have one or all of those subviews position themselves with frames. Views that would like to use frames will have to override `layoutSubviews()` method where they can do precise calculations for `CGRects` necessary to align things in them.

Red flag: never say that you can just simply change frames of views that use AutoLayout. That would not work because with AutoLayout frames are set by the system based on the constraints you've created.

7.10 What options do you have with animation on iOS?

This question is asked to probe your level of experience with animation on iOS. Depending on the team and project focus you could either answer briefly or extensively about each option available.

Expected answer: there are three major things you can use on iOS to animate your UI: `UIKit`, `Core Animation`, and `UIKit Dynamics`.

- `UIKit` is the basic animation that most of used many times. It can be triggered by running `UIView.animateWithDuration()` set of methods. Things that are "animatable" this way are: `frame`, `bounds`, `center`, `transform`, `alpha`, `backgroundColor`.

- `Core Animation` is used for more advanced animation, things that `UIKit` isn't capable of doing. With Core Animation, you will manipulate view's `layer` directly and use classes like `CABasicAnimation` to setup more complex animations.

- `UIKit Dynamics` is used to create dynamic interactive animations. These animations are a more complex kind where the user can interact with

your animation half-way through and potentially even revert it. With UIKit Dynamics you'll work with classes like `UIDynamicItem`. Note: there's also a very handy dynamics animation library by Facebook called `Pop` that can help with it.

Red flag: most likely your interviewer won't expect you to be very familiar with advanced animation techniques unless you claim that you're an expert. But nevertheless, you should be at least aware of other options beyond `UIKit` animations.

7.11 How do you do animation with Frames and AutoLayout?

This is a more specific question about views animation. Depending on the project and team focus they either would like to know how do you handle basic animations or they want to know if you know how to work with advanced animations using Core Animation.

Expected answer: most likely talking about how to animate views with `UIKit` is sufficient enough. With frame based views you'd simply change frame in `UIView.animateWithDuration:animations:` and then assign new frames to your views and that's it, the animation will be performed. With Auto-Layout almost the same thing but instead of changing frames directly you would change your constraints and their constants in `animations:` block of `UIView.animateWithDuration:animations:` method and then call `layoutIfNeeded()` on the views you've changed.

7.12 How do you work with UITableView?

`UITableView` is one of the most used and important UI classes in iOS applications. You can expect this question in one form or another on pretty much any

interview. The extense of your answer will vary and if the interviewer wants to dig deeper they'll ask additional questions around table views.

Expected answer: `UITableView` is a class that lets you display a list of static or dynamic content of various or set height with optional section grouping. Each row in a table is a `UITableViewCell` class or subclass. Table views and cells can be as complex or as simple as application demands. One of the big constraints on mobile devices is memory and performance. This is why table views were designed to dequeue and reuse `UITableViewCells` they are displaying instead of keep creating new objects as user scrolls. That helps to avoid memory bloat and improves performance.

When you work with `UITableView` you usually instantiate an instance of it and then implement `UITableViewDelegate` and `UITableViewDataSource` protocols.

- `UITableViewDelegate` is responsible for calculating cells' and sections' height (unless it's done automatically with `UITableViewAutomaticDimensi` and for the other cell and section lifecycle callbacks like `tableView(UITableView, willDisplay: UITableViewCell, forRowAt: IndexPath)`, `tableView didSelectRowAt: IndexPath)`, etc. It also dequeues section views.
- `UITableViewDataSource` is the source of data for the table. It provides the model data your table is displaying. It is also responsible for dequeuing cells for specific `indexPath`.

7.13 How do you optimize the performance of UITableView for fast scrolling?

One of the important things that is sometimes asked on interviews along with UITableView questions is a question about table view scrolling performance.

Expected answer: Scrolling performance is a big issue with `UITableViews` and quite often can be very hard to get right. The main difficulty is cell height

calculation. When the user scrolls every next cell needs to calculate its content and then height before it can be displayed. If you do manual Frame view layouts than it is more performant but the challenge is to get the height and size calculations just right. If you use AutoLayout then the challenge is to set all the constraints right. But even AutoLayout itself could take some time to compute cell heights and your scrolling performance will suffer.

Potential solutions for scrolling performance issues could be:

- calculate cell height yourself
- keep around a prototype cell that you fill with content and use to calculate cell height

Alternatively, you could take a completely radical approach which is to use different technology like `ComponentKit`. ComponentKit is made specifically for list views with dynamic content size and is optimized to calculate cell heights in a background thread which makes it super performant.

7.14 How do you work with UICollectionView?

This is the same questions as the one about `UITableView`. Your interviewer is trying to figure out if you've worked with more complex UI for lists of items.

Expected answer: `UICollectionView` is the next step from `UITableView` and it was made to display complex layouts for lists of items, think a grid where you have 2-3 or more items in a row or where each item could be of different size. Each item in a `UICollectionView` is a subclass of `UICollectionViewCell`. `UICollectionView` mimics `UITableView` in its API and has similar `UICollectionVie` and `UICollectionViewDataSource` to perform the same functions.

A very distinct feature of `UICollectionView` is that unlike `UITableView` it is using `UICollectionViewLayout` to help it lay out views it is going to display in its list.

7.15 How do you work with UIScrollView?

`UIScrollView` is a very comment UI component used in iOS apps. This question is typically asked to gauge your level of experience working with either big scrollable and zoomable content or deeper understanding of `UITableView` and `UICollectionView`.

Expected answer: `UIScrollView` is responsible for displaying content that is too big and cannot be fully displayed on the screen. It could be a big picture that the user can pinch to zoom or it could be a list of items where all of them cannot be displayed on the screen at the same time. `UIScrollView` is a superclass of `UITableView`, `UICollectionView`, and `UITextView` therefor all of them get the same features as `UIScrollView`.

When you work with `UIScrollView` you define yourself as its delegate by adopting `UIScrollViewDelegate` protocol. There are a lot of methods that you get with that delegate but the main one that you usually work with is `scrollViewDidScroll(UIScrollView)`. In this method, you can do additional work when the user scrolls table view content for example.

7.16 What is UIStackView? When would you use it and why?

`UIStackView` is a powerful new way to lay out views of various size in a container into a column or a row. This question asked to determine how up to date are you with the latest UI tools from Apple. `UIStackView` was introduced in iOS 9 but a surprising number of developers never heard about it.

Expected answer: `UIStackView` is used to align views in a container and "stack" them one after another. If you ever worked with flexbox on the web or with linear layouts on Android the concept will be familiar to you. Previously before iOS 9, you'd have to align your UI in a stack using constraints manually, it was very tedious and error prone especially if you had to change contents of

your stack view at runtime. With `UIStackView` it is as simple as a drag-n-drop in storyboards and programmatically you add or remove views from the stack with just one command. Resizing will be taken care of by `UIStackView` for you.

Note: be very cautious of using `UIStackView` in a table view cell - due to its dynamic sizing nature it could affect scrolling performance negatively.

7.17 What other alternative ways of working with UI do you know?

This is an advanced question that is asked to gauge how well rounded you are in today's trends in UI development.

Expected answer: talk about React and React-like trends in UI development on the web and iOS these days. There's React Native that is a great alternative for declarative UI development but unfortunately, comes with Javascript baggage. There are also libraries like `ComponentKit`, `LayoutKit`, and `IGListKit` that take a different approach from Apple's AutoLayout.

Red flag: you probably shouldn't say that you've never heard of other approaches. It's fine if you never had a chance to try them out in real apps, though.

7.18 How do make pixel-perfect UI according to designer's specs?

This question is typically asked by teams that are very heavy on design and sleek UI.

Expected answer: the short answer you don't :) the long answer is that it depends :) It depends on how you define "pixel-perfect UI". Ideally, if your

designer thought through all the edge cases of your UI laying out on various devices sizes and talked to you about cases when there's no content, etc. then you could hypothetically build a "pixel-perfect UI". But in reality, often that's not the case you discover inconsistencies or edge cases in UI and UX as you build it. Design and UI/UX is not a finite thing - it's a constantly evolving process that is never done. Your best bet is to do your best today and have a short and quick feedback loop with your designer and stakeholders to adjust it as you go.

Red flag: don't say that you "just take photoshop or sketch file and eyeball it".

7.19 How do you unit and integration test UI?

This question is typically asked in addition or as a part of a bigger unit-testing question. There's a lot of controversy around testing on iOS in general.

Expected answer: tooling around UI testing is not that well developed on iOS unlike for other platforms. The options you have today are libraries like `Cedar` that are built on top of Apple's `OCUnit`. But using those you'll have to do all the heavy lifting of setting it up, instantiating UI, filling it with data, etc.

There's a very promising alternative though `LayoutTest-iOS`. `LayoutTest-iOS` helps you test UI and automates a lot of tedious setup, AutoLayout constraint checks, data variations, etc.

Red flag: saying that you don't test your UI is not a red flag perse but you should at least acknowledge that if you don't do it you should be doing it.

7.20 Conclusion

UI questions are very common on iOS interviews because virtually more than half of the time spent building iOS apps will be views related work. For some apps it is crucial to build sleek and nice UI, other can go by with just bare

bones. In any case as usual things you should keep in mind are reusability and single responsibility principle. If your UI is not tightly coupled to other parts of your app then it's going to be very easy to update it if it's not perfect or if specs have changed.

Chapter 8

Step Seven. Beyond MVC. Design Pattens, Dependency Management, FRP, etc.

Chapter 9

Bonus Chapter: Storage Evolution (AKA You Don't Always Need Core Data!).

Every app, big or small, needs to store data. Questions about persistence are very typical on iOS interviews. Usually, you are asked what types of storage you worked with and if you worked with Core Data and what's good or bad about it. Expected answer is yes and "it's an object graph storage that helps you persist data organized by entities" or something along those lines. But for us, to nail the interview, the goal is to know even more than the person asking the questions. The goal is to know your options and that Core Data is not always the best solution. In this chapter, we'll look at what Storage Layer is in iOS apps and different ways you can persist data and evolve your storage mechanics using in-memory arrays and dictionaries, NSUserDefaults, File/Disk storage, Core Data and Realm, SQL.

Storage layer can be as simple as an array or dictionary of data that holds models in memory for your app. Or as complex as a Core Data or custom SQL ORM solution that can be observed and queried with advanced predicates. The main thing and responsibility of that layer is that it stores data for your

application and can play the role of ultimate source of truth for the rest of your code.

9.1 Typical tools used for persistence in Storage Layer

The following classes, objects, and libraries/frameworks (in ascending order of complexity) are used in Storage Layer:

- In-memory arrays, dictionaries, sets, and other data structures
- NSUserDefaults/Keychain
- File/Disk storage
- Core Data, Realm
- SQLite

9.2 In-memory arrays, dictionaries, sets, and other data structures

Probably when you hear words "storage layer" you instantly think about Core Data or similar database technology that will help you persist things into tables. But surprising enough your storage layer could be as simple as just an in-memory array where you store a list of things you've fetched from the backend API for example. The main thing is that you abstract that internal implementation out from the rest of your application.

All Swift Collection Types and corresponding Objective-C types can be used as the underlining mechanism for storage for your application. `Array`, `NSArray`,

`Set`, `NSSet`, `Dictionary`, `NSDictionary` all could be used to save things in the storage layer.

Advantages:

- easy and quick to create (they are just plain old arrays and hashes after all)
- quite often is actually the only thing you need
- can be KVO observed for changes

Disadvantages:

- can't be persisted to disk on its own without additional help (NSCoding interface for example)
- because they can't be persisted they can't be restored from persistent memory later
- can't be used to store large amounts of data

Example:

For example, let's say your app is displaying posts that are fetched from the backend. A typical storage class for Posts will look like that:

```
struct Post {
    let remoteId: NSNumber
    let name: String
}

class PostsStorage {

    private var posts = Dictionary<NSNumber, Post>()

    func savePost(newPost: Post) {
        self.posts[newPost.remoteId] = newPost
    }
```

```
    func getAllPosts() -> [Post] {
        return Array(self.posts.keys.map { self.posts[$0]! })
    }

    func findPostByRemoteId(remoteId: NSNumber) -> Post? {
        return self.posts[remoteId]
    }
}
```

As you can see, there's nothing crazy to it. It has an internal dictionary that uses **`remoteId`** of **`Post`** structs as keys to store those objects.

The way you'd use that storage is pretty straightforward as well:

```
let postsStorage = PostsStorage()

let post1 = Post(remoteId: 1, name: "Post 1")
let post2 = Post(remoteId: 2, name: "Post 2")
let post3 = Post(remoteId: 3, name: "Post 3")
let post4 = Post(remoteId: 4, name: "Post 4")

postsStorage.savePost(post1)
postsStorage.savePost(post2)
postsStorage.savePost(post3)
postsStorage.savePost(post4)


print(postsStorage.getAllPosts())

print(postsStorage.findPostByRemoteId(post2.remoteId))
```

Comparing to other storage layer tools this is the simplest one but it is often everything you really need. I've seen applications where using Core Data was an overkill and switching to an in-memory array of model objects was the best solution for storing data. It removed the overhead of dealing with Core Data setup, contexts, and coordinators.

Given that a lot of apps are actually fine with loosing data from one launch of the app to another and can quickly and easily fetch it from the backend using in-memory storage is an invaluable, straightforward, and easy tool to use.

```
NOTE: ## Models and Collections

In this section, we spoke about tools used to store data. In all cases, the data that
we are storing is a custom model class objects. A lot of naive implementations of
models and storage layers work with `NSDictionary`s as their models and access
values through keys. This is a very error prone approach and a custom `struct` or
`class` is always way better instead. So, again, just to reiterate, if we use arrays
and dictionaries we use them to store collections of things, we do not make them
represent individual model objects.
```

In the next sections, we'll see how this storage can be "evolved" and changed by swapping underlying storing mechanism. Abstraction usefulness will be more apparent.

9.3 NSUserDefaults and Keychain

The next step up from in-memory storage is **`NSUserDefaults`** and **`Keychain`**. Both of them, unlike in-memory storage, persist things to disk. They are also way simpler than Core Data or Realm because there's no underlining table or graph structure. At the same time, they do persist objects to disk, unlike in-memory solution. To put simply they **`NSUserDefaults`** and **`Keychain`** are just key-value storage that you can write primitive data to.

9.3.1 NSUserDefaults

NSUserDefaults can store by key primitive values like **`NSNumber`**, **`NSString`**, etc. or objects that comply to **`NSCoding`** protocol. Also, it can store arrays or dictionaries that contain objects that comply to **`NSCoding`** protocol. The objects can be retrieved easily by accessing them with a key they were stored with.

Typically we think of **`NSUserDefaults`** as a solution to store user settings or preferences or token in it (although token really should be stored in Keychain). But in reality, for some apps, it's a perfectly good option for storing the main

application data that acts as a database. As you will see in the example below it's a perfectly reasonable substitution for our in-memory solution from the previous section.

Example:

```
class Post: NSObject, NSCoding {
    let remoteId: NSNumber
    let name: String

    init(remoteId: NSNumber, name: String) {
        self.remoteId = remoteId
        self.name = name
    }

    required convenience init?(coder decoder: NSCoder) {
        guard let remoteId = decoder.decodeObjectForKey("remoteId") as? NSNumber,
            let name = decoder.decodeObjectForKey("name") as? String
        else { return nil }

        self.init(remoteId: remoteId, name: name)
    }

    func encodeWithCoder(coder: NSCoder) {
        coder.encodeObject(self.remoteId, forKey: "remoteId")
        coder.encodeObject(self.name, forKey: "name")
    }
}

class PostsStorage {

    private let userDefaults = NSUserDefaults.standardUserDefaults()

    private let storageNameSpacePrefix = "my_posts_"

    func savePost(newPost: Post) {
        let newPostData = encodePost(newPost)
        self.userDefaults.setObject(newPostData, forKey: self.postKey(newPost.remoteId))
    }

    func getAllPosts() -> [Post] {
        return Array(self.allPostKeys().map { (key) -> Post in
            let postData = self.userDefaults.objectForKey(key)
            return decodeToPost(postData as! NSData)
        })
    }

    func findPostByRemoteId(remoteId: NSNumber) -> Post? {
        if let postData = self.userDefaults.objectForKey(self.postKey(remoteId)) as? NSData {
            return decodeToPost(postData)
```

```
        }
        return nil
    }

    private func encodePost(post: Post) -> NSData {
        return NSKeyedArchiver.archivedDataWithRootObject(post)
    }

    private func decodeToPost(data: NSData) -> Post {
        return NSKeyedUnarchiver.unarchiveObjectWithData(data) as! Post
    }

    private func postKey(remoteId: NSNumber) -> String {
        return "\(self.storageNameSpacePrefix)\(remoteId.stringValue)"
    }

    private func allPostKeys() -> [String] {
        return Array(self.userDefaults.dictionaryRepresentation().keys.filter { (key) -> Bool i
            return key.containsString(self.storageNameSpacePrefix)
        })
    }
}
```

In this example, we replaced dictionary storage with **`NSUserDefaults`**. In order for us to be able to save **`Post`** objects to **`NSUserDefaults`**, we have to implement **`NSCoding`** protocol on them. So we convert **`Post`** into a class and implement **`init?(coder decoder: NSCoder)`** and **`encodeWithCoder`** to code and decode individual **`Post`** objects.

Also, we slightly change our save and retrieve methods. Now they use **`NSKeyedArchiver`** and **`NSKeyedUnarchiver`** convert **`Post`** objects to **`NSData`** or decode them back from **`NSData`** to **`Post`** type before they can be written or read from **`NSUserDefaults`**.

Oh, and notice that we have to **`storageNameSpacePrefix`** so that we get all the keys for our stored **`Posts`** later (otherwise **`self.userDefaults.dictionaryRepre`** will return all they keys in **`NSUserDefaults`**).

The main thing though is that our public API for the storage remains the same. Everyone who was using it and relying on it will continue to do it the same way but now the storage actually persists **`Posts`** in memory.

```
let postsStorage = PostsStorage()

let post1 = Post(remoteId: 1, name: "Post 1")
let post2 = Post(remoteId: 2, name: "Post 2")
let post3 = Post(remoteId: 3, name: "Post 3")
let post4 = Post(remoteId: 4, name: "Post 4")

postsStorage.savePost(post1)
postsStorage.savePost(post2)
postsStorage.savePost(post3)
postsStorage.savePost(post4)


print(postsStorage.getAllPosts())

print(postsStorage.findPostByRemoteId(post2.remoteId))
```

Advantages:

- actually, persists things to disk (so the data can be restored between app launches)
- easy to use key/value storage

Disadvantages:

- can't be easily KVO observed (you'll have to roll your own notification/observation system)
- can't be used to store large amounts of data (it was not made for that)
- not that helpful when you need to filter and sort data

9.3.2 Keychain

Keychain is the tool for storing data securely. This is where you'd store user passwords and tokens, not NSUserDefaults.

Typically working with Keychain directly is a bit gnarly and tedious due to its C-based API. I recommend using a library wrapper like KeychainAccess or samkeychain instead.

Advantages:

- key-value storage for primitive values
- secure

Disadvantages:

- inconvenient API
- errors out and fails quite often

An interesting fact is that stuff saved in Keychain, unlike stored in NSUserDefaults, will persist and survive app uninstall/reinstall. The reason being is that NSUserDefaults is the storage that is tightly coupled with your application and Keychain is a global secure system storage managed by Apple. That is both an advantage and disadvantage which allows us to do nice things like storing a flag on the first application launch in Keychain indicating that the app was installed for the first time. Next time, if the user uninstalls and then reinstalls your app, you can check whether the flag is present or not and go with default onboarding flow for a new user for example and do some other custom onboarding for returning users.

9.4 File/Disk storage

File and disk storage are typically used to persist bigger chunks of data like images and videos but it can also be used as a substitution for your database. It is perfectly capable of storing the same type of objects as `NSUserDefaults`:

primitives like String, NSNumber, etc., dictionaries, arrays, and custom objects that conform to **NSCoding** protocol.

We will iterate over our previous storage example and swap the underlying storage with an **NSFileManager**.

```
class Post: NSObject, NSCoding {
    let remoteId: NSNumber
    let name: String

    init(remoteId: NSNumber, name: String) {
        self.remoteId = remoteId
        self.name = name
    }

    required convenience init?(coder decoder: NSCoder) {
        guard let remoteId = decoder.decodeObjectForKey("remoteId") as? NSNumber,
            let name = decoder.decodeObjectForKey("name") as? String
            else { return nil }

        self.init(remoteId: remoteId, name: name)
    }

    func encodeWithCoder(coder: NSCoder) {
        coder.encodeObject(self.remoteId, forKey: "remoteId")
        coder.encodeObject(self.name, forKey: "name")
    }
}

class PostsStorage {

    private let fileManager = NSFileManager.defaultManager()

    private let storageNameSpacePrefix = "my_posts_"

    func savePost(newPost: Post) {
        let newPostData = encodePost(newPost)
        let key = postKey(newPost.remoteId)
        saveDataToDisk(key, directoryPath: documentsDirectory(), data: newPostData)
    }

    func getAllPosts() -> [Post] {

        return allPostKeys().map({ (fileName) -> Post in
            let postData = self.fileManager.contentsAtPath(fullPostPath(fileName))!
            return decodeToPost(postData)
        })
    }

    func findPostByRemoteId(remoteId: NSNumber) -> Post? {
```

```
        let postPath = fullPostPath(postKey(remoteId))
        if let postData = self.fileManager.contentsAtPath(postPath) {
            return decodeToPost(postData)
        }
        return nil
    }

    private func postKey(remoteId: NSNumber) -> String {
        return "\(self.storageNameSpacePrefix)\(remoteId.stringValue)"
    }

    private func allPostKeys() -> [String] {
        do {
            let directory = self.documentsDirectory()
            return try self.fileManager.contentsOfDirectoryAtPath(directory).filter({ (path) ->
                return path.containsString(self.storageNameSpacePrefix)
            })
        } catch {
            return []
        }
    }

    private func encodePost(post: Post) -> NSData {
        return NSKeyedArchiver.archivedDataWithRootObject(post)
    }

    private func decodeToPost(data: NSData) -> Post {
        return NSKeyedUnarchiver.unarchiveObjectWithData(data) as! Post
    }

    private func documentsDirectory() -> String {
        return NSSearchPathForDirectoriesInDomains(.DocumentDirectory,
                                                   .UserDomainMask, true).first! + "/posts_stora
    }

    private func saveDataToDisk(fileName: String, directoryPath: String, data: NSData) -> Bool

        let filePath = "\(directoryPath)/\(fileName)"

        do {
            try self.fileManager.createDirectoryAtPath(directoryPath,
                                    withIntermediateDirectories: true,
                                                    attributes: nil)
            let success = self.fileManager.createFileAtPath(filePath,
                                                     contents: data,
                                                     attributes: nil)
            return success

        } catch {
            return false
        }
```

```
    }

    private func fullPostPath(postKey: String) -> String {
        return self.documentsDirectory() + "/" + postKey
    }
}
```

Here we still keep serializing our **`Post`** objects to **`NSData`** before we store them but underlying storing mechanics is now using a file manager that saves each post to disk as a file with a unique namespaced name.

The implementation is grown a little bit more with a few extra private methods and **`do/catch`** blocks to accommodate **`NSFileManager`** API but other than that it remains the same overall. We still use **`remoteId`** of each post as a unique key to identify and access each post. To get all posts that were stored in **`getAllPosts()`** method we examine the folder that used by the storage and get **`NSData`** for each file and decode it back to **`Post`** objects. When we store **`Posts`** in **`savePost()`** method we encode them into **`NSData`** and persist to disk. And when retrieving individual **`Post`** objects from memory in **`findPostByRemoteId()`** method we get **`NSData`** for that unique **`remoteId`** and then decode it to **`Post`** object.

And the great thing is as with the previous iteration our public API for **`PostsStorage`** remains the same. Everyone who's been using it will continue to do so in the same fashion:

```
let postsStorage = PostsStorage()

let post1 = Post(remoteId: 1, name: "Post 1")
let post2 = Post(remoteId: 2, name: "Post 2")
let post3 = Post(remoteId: 3, name: "Post 3")
let post4 = Post(remoteId: 4, name: "Post 4")

postsStorage.savePost(post1)
postsStorage.savePost(post2)
postsStorage.savePost(post3)
postsStorage.savePost(post4)


print(postsStorage.getAllPosts())
```

```
print(postsStorage.findPostByRemoteId(post2.remoteId))
```

Advantages:

- actually, persists things to disk (so the data can be restored between app launches)
- can store large amounts of data (big media files for example)

Disadvantages:

- can't be easily KVO observed (you'll have to roll your own notification/observation system)
- not that helpful when you need to filter and sort data

This storage mechanic is a step up from `NSUserDefaults` and is pretty robust when you need something more stable than just a key-value store but you are not sure if you need a full-fledged database solution yet.

9.5 Core Data

And we got to good old Core Data database storage solution. Core Data is an object graph persistence framework that helps you save objects to a database. Under the hood, it is using SQLite (with options to use in-memory and binary stores) but the interface is completely abstracted out and we are interacting only with Core Data framework objects and classes when we use it.

There is a lot to it and it is a fairly complex piece of technology but overall you typically work with it using the following classes and objects: `NSManagedObject`, `NSManagedObjectContext`, `NSFetchRequest`, `NSFetchedResultsController`, `NSPredicate`, `NSSortDescriptor`.

`NSManagedObject`s represent data stored in the database. You can think of them as model objects.

`NSManagedObjectContext` allows you to insert, save, and retrieve (using `NSFetchRequest`) `NSManagedObjects` from database.

`NSFetchRequest` is a "query" object that you use to retrieve `NSManagedObjects` from database optionally filtered with an `NSPredicate` and sorted with an `NSSortDescriptor`.

`NSFetchedResultsController` is a more functional reactive way of getting notifications about object changes in the database that are filtered by criteria set in `NSFetchRequest` (think of it as notifications about database changes).

`NSPredicate` lets you add filters to `NSFetchRequest` queries.

`NSSortDescriptor` lets you add sorting to your queries.

There are two major schools of thought when it comes to working with Core Data: using `NSManagedObject` subclasses as your models or map and serialize your custom model objects to `NSManagedObjects` and use them only for persisting data to database.

9.5.1 Going NSManagedObject subclass route

Typically Core Data examples and tutorials will show you that you need to subclass your model object from `NSManagedObject` in order to be able to persist them to disk. But subclassing couples you to the underlying implementation details and behavior that comes with `NSManagedObject`. Also when you do asynchronous work and operate with `NSManagedObject` subclasses you need to keep a close eye on what `NSManagedObjectContext` do you use to retrieve, update, and save them. It is typically intricate and error-prone approach that causes a lot of bugs and confusion in the code. A good example of a library that implements this approach is RestKit. As a side note: it mixes networking and data persistence responsibilities together and is a very bulky and difficult to work with.

9.5.2 Going Data mapping/serialization route

A better approach is to map your model objects (just plain old **NSObject** subclasses or structs) to **NSManagedObjects** and use **NSManagedObjects** only to persist data to disk. That way your code stays completely decoupled from Core Data and model objects do not carry the burden of **NSManagedObject**'s underlying behavior because they are not subclassing from them. There's no need to worry about multi-threading and **NSManagedObjectContexts** because most of your code operates with simple and straightforward **NSObject/Object** subclasses or structs.

Let's look at an implementation of such approach:

```
struct Post {
    let remoteId: NSNumber
    let name: String
}

class PostManagedObject: NSManagedObject {
    @NSManaged var remoteId: NSNumber
    @NSManaged var name: String

    static func postManagedObject(remoteId: NSNumber,
                                  name: String,
                                  context: NSManagedObjectContext) -> PostManagedObject
    {
        let entity = entityDescription(context)
        let postManagedObject = PostManagedObject(entity: entity,
                                                  insertIntoManagedObjectContext: context)
        postManagedObject.remoteId = remoteId
        postManagedObject.name = name
        return postManagedObject
    }

    private static func entityDescription(context: NSManagedObjectContext) -> NSEntityDescripti
        return NSEntityDescription.entityForName(NSStringFromClass(self),
                                                 inManagedObjectContext: context)!
    }
}

class PostsStorage {

    private let persistentStoreCoordinator: NSPersistentStoreCoordinator
    private let managedObjectContext: NSManagedObjectContext

    init() {
```

```
        let postEntityDescriptior = NSEntityDescription()
        postEntityDescriptior.name = NSStringFromClass(PostManagedObject)
        postEntityDescriptior.managedObjectClassName = NSStringFromClass(PostManagedObject)

        let remoteIdAttributeDescriptor = NSAttributeDescription()
        remoteIdAttributeDescriptor.name = "remoteId"
        remoteIdAttributeDescriptor.attributeType = .Integer64AttributeType
        remoteIdAttributeDescriptor.optional = false
        remoteIdAttributeDescriptor.indexed = true

        let nameAttribute = NSAttributeDescription()
        nameAttribute.name = "name"
        nameAttribute.attributeType = .StringAttributeType
        nameAttribute.optional = false
        nameAttribute.indexed = false

        postEntityDescriptior.properties = [remoteIdAttributeDescriptor, nameAttribute]

        let managedObjectModel = NSManagedObjectModel()
        managedObjectModel.entities = [postEntityDescriptior]

        persistentStoreCoordinator = NSPersistentStoreCoordinator(managedObjectModel: managedOb
        do {
            try persistentStoreCoordinator.addPersistentStoreWithType(NSInMemoryStoreType,
                                                                      configuration: nil,
                                                                      URL: nil,
                                                                      options: nil)
        }
        catch {
            print("error creating persistentStoreCoordinator: \(error)")
        }

        managedObjectContext = NSManagedObjectContext(concurrencyType: .MainQueueConcurrencyTyp
        managedObjectContext.persistentStoreCoordinator = persistentStoreCoordinator
    }

    func savePost(newPost: Post) {
        encodePost(newPost)
        saveDataToDatabase()
    }

    func getAllPosts() -> [Post] {

        let fetchRequest = baseFetchRequest()

        if let postManagedObjects = executeFetchRequest(fetchRequest) {
            return postManagedObjects.map({ (postManagedObject) -> Post in
                return decodeToPost(postManagedObject)
            })
        } else {
            return []
```

```
        }
    }

    func findPostByRemoteId(remoteId: NSNumber) -> Post? {

        let fetchRequest = baseFetchRequest()

        fetchRequest.predicate = NSPredicate(format: "remoteId == %@", remoteId)

        if let postManagedObject = executeFetchRequest(fetchRequest)?.first {
            return decodeToPost(postManagedObject)
        } else {
            return nil
        }
    }

    private func encodePost(post: Post) {
        PostManagedObject.postManagedObject(post.remoteId,
                                            name: post.name,
                                            context: managedObjectContext)
    }

    private func decodeToPost(postManagedObject: PostManagedObject) -> Post {
        return Post(remoteId: postManagedObject.remoteId, name: postManagedObject.name)
    }

    private func saveDataToDatabase() -> Bool {
        if managedObjectContext.hasChanges {
            do {
                try managedObjectContext.save()
                return true
            } catch {
                return false
            }
        } else {
            return false
        }
    }

    private func baseFetchRequest() -> NSFetchRequest {
        let fetchRequest = NSFetchRequest(entityName: NSStringFromClass(PostManagedObject))

        let sort = NSSortDescriptor(key: "remoteId", ascending: true)
        fetchRequest.sortDescriptors = [sort]

        return fetchRequest
    }

    private func executeFetchRequest(fetchRequest: NSFetchRequest) -> [PostManagedObject]? {
        return (try? managedObjectContext.executeFetchRequest(fetchRequest)) as? [PostManagedOb
    }
```

```
}
```

Here we are creating a struct **Post** that will be our actual model structure (that is the thing that the rest of the application works with). And **PostManagedObject**, a subclass of **NSManagedObject**, is used to persist data mapped from **Post** objects to database. **PostManagedObject** is only used internally by **PostsStorage** to actually get data in and out from the db.

PostsStorage had experienced quite a change and now has an **NSPersistentStoreCoo** to setup database and entities that are going to be stored in it, and an **NSManagedObjectCon** to help with data persistence and fetching.

We are setting up our database in **PostsStorage** initializer with **NSPersistentStoreCo** **NSEntityDescription** and **NSAttributeDescriptions** (for **remoteId** and **name** properties). Normally this setup will happen somewhere else in iOS application with a help of Xcode's **Data Model** files and instead of **NSInMemoryStoreTyp** we'll have it use actual SQLite under the hood. But doing it explicitly in code like this is a perfectly fine approach as well.

To fetch and save objects we use **NSManagedObjectContext** and **NSFetchRequests**. When the data is saved on the outside we work with **Post** object but then we map it into a **PostManagedObject** that can be saved to Core Data and save it using **managedObjectContext.save()** in **saveDataToDatabase()** method. When we retrive objects from the database we get back **PostManagedObjects** in **getAllPosts()** and **findPostByRemoteId()** methods and then we map them back to **Post** objects that our application can work with.

But the bottom line again is that we have the same public API in the storage as before. Users of that storage can still rely on having **savePost()**, **getAllPosts()**, **findPostByRemoteId()** methods that save and find **Posts** in and from the db:

```
let postsStorage = PostsStorage()

let post1 = Post(remoteId: 1, name: "Post 1")
let post2 = Post(remoteId: 2, name: "Post 2")
```

```
let post3 = Post(remoteId: 3, name: "Post 3")
let post4 = Post(remoteId: 4, name: "Post 4")

postsStorage.savePost(post1)
postsStorage.savePost(post2)
postsStorage.savePost(post3)
postsStorage.savePost(post4)


print(postsStorage.getAllPosts())

print(postsStorage.findPostByRemoteId(post2.remoteId))
```

9.6 Realm

// TODO: add Realm example

9.7 SQLite

// TODO: add SQLite example

9.8 Storage Layer plays dual role: Data Storage/Persistenc and Data Mapping/Serialization

As you saw in examples above, no matter how complex or simple, they all had the same public API in the storage and the work the storage has done internally has similar parallels across implementations. They all do have some kind of permanent storage mechanism (in-memory dictionary, or **`NSUserDefaults`**, or **`Disk file storage`**, or **`Core Data database`**, etc.). And they all (except in-memory dictionary) encode or decode data before saving or retrieving it.

That is due to the nature of storage layer itself. It has to map data to some kind of structure it can easily persist and when retrieved back it is not useful to the rest of the application. So it needs to be mapped back to model objects and types that are convenient for us to work with.

You have the option to implement data serialization/mapping yourself just like we've done in the examples above but there are plenty of libraries out there that can help you with that. Mantle and MTLManagedObjectAdapter are my go-to choice working with Core Data and JSON for example.

9.9 Switching storages

As was mentioned in the beginning of this section one of the biggest advantages of abstracting out the storage layer is that you can swap underlying persistence mechanics when needed. Let's you started working on a new app or a new feature and don't know yet if you'd need on disk database persisted storage. So instead you can start implementing your storage as a simple array or a dictionary and prototype or deliver your feature with that. And later when you have more information and you know you really need to write the data to a database you can easily replace that under the hood array with a Core Data model and table. For the rest of your application nothing really changes, it accesses the data the same way as before because you had a clearly defined interface for that.

9.10 FRP in Storage Layer.

Core Data, Realm, or other Centralized Storage are awesome especially when you use them to observe data in the functional reactive way.

Although they can be used to pull (i.e. query things manually) they are the best when you can observe changes to your storage. The simplest example of that would be `NSFetchedResultsController`. In fact, this is one of the few

Apple's functional reactive tools in iOS.

A similar thing can be applied to in-memory data storages like arrays but you'll have to use either bare bones KVO or get help from a library like `RxSwift` or `ReactiveCocoa`.

9.11 Be practical in your Storage Layer implementation and decisions

I'm a big believer in "using only what you need" philosophy. In the case of storage, that means don't overthink what you really need to have from your storage layer. Even if when you get a handle on Core Data or Realm it sounds great to use it everywhere for every application. This is not always the best.

In my experience in-memory, `NSUserDefaults`, and File/Disk storage are useful for prototyping and for applications with small data footprint that can survive data whipe between app launches. Core Data, Realm, and SQL storage, on the other hand, give you an advantage of data observation and are very good database solutions when you need to store large amounts of data, sort and filter them.

Keep your storage API clear and abstract out internal implementation following Single Responsibility Principle and you'll be fine no matter what storage solution you've picked.