1ST EDITION

THE IOS INTERVIEW GUIDE

The best way to pass any iOS interview is to know your stuff well. This book packs questions, answers, red flags, and general guidance for iOS developers to ace technical interviews.

ALEX BUSH

Foreword by Andrew Rohn

Interview Guide Series

The iOS Interview Guide

Questions, answers, and general guidance on what iOS developers should know to nail any tech interview. 1st edition, version 1.0.6.

Alex Bush

The iOS Interview Guide

Alex Bush

Preface

The iOS Interviews Guide V 1.0.6.

This is the final (1.0.6) version of The iOS Interviews Guide (1st edition) by Alex Bush.

This book is something I wish I had when I was a junior developer just starting out with iOS development. I wasn't sure what I should know as an aspiring iOS dev, what areas were important and I should pay close attention to, and what I could just glance over. There are a lot of resources and articles about interview questions out there, but I found that none of them really systematized what they were talking about. Some would have questions about memory management and byte shifting mixed in with questions about AutoLayout and design patterns, which does not make any sense and doesn't help developers who are seeking guidance in their quest. And this is how this book came to be. I hope you will enjoy reading it as much as I enjoyed working on it. I wish you all the best on your job hunt!

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!

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

- LinkedIn: https://www.linkedin.com/in/alexvbush/

Foreword

By Andrew Rohn, iOS Software Engineer at Reddit and Co-Host of Inside iOS Dev Podcast

Thanks to 'The iOS Interview Guide', I skillfully interviewed with Facebook, Uber, Reddit, Pandora, and many other great companies and I now work at Reddit as an iOS Software Engineer.

In one interview, I gave an answer to a system design question that was so good that the interviewer asked me "Have you solved this problem before?". To which I responded truthfully, "No." Later, he mentioned that the sophistication and quality of my answer greatly exceeded that of someone with only one year of experience.

My recent successful interview process is direct evidence of the authority and quality of this book. If you need more convincing about the quality of this book or if you'd like to learn more about the author – continue reading this foreword. Otherwise, I recommend you get straight to reading this book so that you can crush your next interview!

I think Alex is the perfect author for this book for three reasons: humility, pragmatism, and curiosity.

I'm a junior iOS Software Engineer with just one year of professional experience. My first year working professionally was done under the mentorship of Alex. I worked closely with Alex at Wanelo where we pair programmed together every other day. I think Alex might quibble at being called my "mentor". He might prefer something that defines our work relationship as equals

working together. This humility translates into what makes him a great teacher and mentor. Alex doesn't speak down to you. He speaks plainly and if you don't understand something he'll clarify it for you. Whenever I needed guidance, I knew Alex would graciously share his time with me until I had a full understanding.

Alex has a very pragmatic problem solving style. He understands that, at the end day, us software engineers are there to help a business make more money. If he talks architecture or algorithms he stays grounded in reality. He knows that your employer doesn't care what hip framework you used or what esoteric algorithm you vainly implemented. He realizes that good code is an asset to the company and that bad code can cripple a company from being able to keep up with competitors. I attribute this to his time as a successful consultant where clients needed a software system that would drive their business. Alex knows that employers want to see that you can consistently ship quality code in a timely manner.

Lastly, Alex has a boundless curiosity that ceaselessly propels his skills forward. He is constantly reading books, watching talks, and trying new things. After working with him, these habits have rubbed off on me. We had an informal book club every morning where he shared with me his new insights or learnings and encouraged me to share as well. This curiosity drove Alex to being knowledgeable not just in the iOS world but in the entire full stack. He has created systems that required both a backend application and client iOS application. Because of this, he has a rare and invaluable end-to-end system perspective. This unquenchable curiosity is what gives me confidence in Alex's word. He is an authority because he is always reaching for the next level of understanding.

Testimonials

Will's video testimonial on YouTube

– Will Lundy, iOS Developer at Wells Fargo

Yusuke's video testimonial on YouTube

– Yusuke Kawanabe, Lead iOS Engineer at Nima.

Alex Bush's book, "The iOS Interview Guide" is a very helpful resource for a variety of reasons. It helps the experienced developer prepare for their next career move by identifying concepts, and areas that technology companies will ask about during the application, and interview process. The book is an excellent resource because it's not simply a checklist of topics, and concepts, to study, but also discusses these concepts as well, and identifies potential pitfalls that the unsuspecting applicant may fall into during the interview process. This book is also a valuable resource for the junior developer who is trying to get an understanding as to what skills are expected from someone who is senior. This book helps chart a course for the junior developer in better improving their skills, and identifying those key areas which are important, thus allowing the junior developer to plan out their career development more efficiently. This book is indeed a valuable resource for developers in all stages of their careers. Thanks Alex for doing such a wonderful job!

– Fayyazuddin Syed, Senior iOS Developer

"Alex has hit a home-run with the iOS interview guide. It has been my go to reference while looking for a job in this field. I can't thank him enough for sharing his insight into what it takes to be prepared for an interview at any

level of your career!"

– Paul DeFilippi, iOS Developer

"As a junior developer, I just want to say that the information you're sharing is top notch and extremely eye-opening to the naive approaches that I have taken. Thank you so much for doing this."

– Jovanny Espinal, iOS Software Engineer at Blue Apron

"I am glad to inform you that your book on Swift Interview question helped me a lot , I have two current jobs under process for next rounds. All of the question I have answered the same way as you wrote in Expected Answer. It is really worth buying it ."

– Ramkrishna Baddi, iOS Developer

"An excellent guide to help self-starter iOS programmers land their dream jobs. This book can be your ultimate guide for your iOS development study as well as getting your first job as an iOS developer."

– Jon Lu, Freelance iOS Developer

Acknowledgments

First of all I'd like to thank all the early adopters of this book for all the feedback they gave me. Writing a book is a laborious endeavor and people like you giving invaluable feedback and validation make it worthwhile and fuel my enthusiasm for actually continuing this project.

I'd like to give special thanks to Artemij (Art) Fedosejev for showing me, with a personal example, that writing a book is actually something doable, and for supporting me throughout this journey with advice and inspiration. You will go far; the dots are connecting, mark my words.

Thanks to my mom and my grandpa, who were always supportive of me throughout life. Thank you for instilling in me this clarity and surety that I can achieve anything I want.

Thank you to Andrew Rohn for giving me feedback and for being the actual test pilot who applied ideas from the book in his interviews and code. I appreciate our shared values.

And another big thanks to the following early adopters of this book who I had the pleasure to either meet in person or to talk to over Skype or phone: Paul DeFilipi, Jon Lu, Kevin Zou, John Jacecko, Ronald Hernandez, Forrest Zhao, Fayyazuddin Syed, Will Lundy, Kurt Walker, Yusuke Kawanabe, Alex Qin, Natalia Chodelski, Sabita Samal, and Santi Gracia. Your feedback was invaluable!

I'd also like to thank the editor I collaborated with, Adaobi Obi Tulton. It was a pleasure and a breeze working with you.

Finally, I want to acknowledge you, the reader: Thank you for selecting this book. I hope it will help you in your development and job search.

Contents

Change Log

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog and this project adheres (somewhat) to Semantic Versioning.

[1.0.6] (current) - 2018-9-10

Added:

Changed:

- all the code samples shipped with the book (see ios_interview_guide_v_1_0_6_code_s zip file) were rewritten with Swift 4 and made more idiomatically "Swifty".
- this changelog moved to after the table of contents

Removed:

[1.0.5] - 2017-12-13

Added:

Changed:

- fixed and clarified struct inheritance in Fundamentals chapter.

- fixed storyboards mention in `What are the advantages and disadvantages of creating AutoLayouts in code versus using storyboards?` question in UI chapter.
- fixed minor typos

Removed:

[1.0.4] - 2017-08-16

Added:

- clarified how optionals work with `let`s
- clarified usage of scope defining `let`s

[1.0.3] - 2017-06-28

Changed:

- Fixed missing apostrophes in PDF version of the book

[1.0.2] - 2017-06-01

Changed:

- Fixed minor typos

[1.0.1] - 2017-05-28

Changed:

- Fixed minor typos

[1.0.0] - 2017-05-28

Added:

- Outro and Acknowledgments

[0.8.3] - 2017-05-24

Added:

- Added about the author info

[0.8.2] - 2017-05-23

Changed:

- Fixed and improved layout in all the chapters

[0.8.1] - 2017-05-15

Changed:

- Chapter 1 Intro (edited for typos and structural improvements).
- Chapter 2 Step One: Figure Out what the Big Picture Is (edited for typos and structural improvements).
- Chapter 5 Step Four: Get Productive with Networking (edited for typos and structural improvements).
- Chapter 6 Step Five: Learn How to Store Data (edited for typos and structural improvements).

- Chapter 7 Step Six. Go crazy responsive with UI layouts (edited for typos and structural improvements).
- Chapter 8 Step Seven: Beyond MVC. Design Pattens, Architecture, FRP, and Dependencies Management. (edited for typos and structural improvements).
- Chapter 9 Bonus Chapter: Storage Evolution (AKA You Don't Always Need Core Data!). (edited for typos and structural improvements).

[0.8.0] - 2017-04-30

Changed:

- Chapter 1 Intro was restructured.
- Chapter 2 Step One. Figure out what the big picture is. was restructured.

Added:

- Chapter 6 Step Five. Learn how to store data. (unedited)
- Added cross link references to chapters and questions/answers within chapters for easy navigation

Removed:

- all the TODOs, notes, and unfinished content was removed from chapters
- overflowing chapter title at the top of each page was removed

[0.7.1] - 2017-04-10

Changed:

- Chapter 3 Step Two. The Interview Game. (edited for typos and structural improvements).

[0.7.0] - 2017-03-27

Added:

- Chapter 8 Step Seven. Beyond MVC. Design Pattens, Architecture, FRP, Dependencies Management (unedited)

[0.6.0] - 2017-01-21

Added:

- This CHANGELOG
- Chapter 3 Step Two. The Interview Game. (unedited)

[0.5.0] - 2017-01-02

Added:

- Chapter 4 Step Three. Learn the fundamentals (unedited)

[0.4.0] - 2016-11-14

Added:

- Initial pre-release version containing 4 chapters (unedited)

Chapter 1

Intro

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

Do you hate them as much as I do? It takes so much time to prepare for them, 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. It will give you an overview of what there is to learn on the iOS platform and systematize that stuff so that there's clear structure and guidance on what you need to learn next.

The best way to not be nervous and to nail your interviews after all is to actually know more and better than your interviewer. If you want to know "advanced stuff," then this is the guide for you!

1.1 Who am I?

My name is Alex. I'm a fellow developer like you. I've been working with iOS for over six years, built over twenty apps, code reviewed thousands of lines of code, mentored several developers, and interviewed a lot of developers. I know all the struggles you go through with iOS development and I know what pitfalls there are. I founded Smart Cloud. I blog at http://www.sm-cloud.com/ and co-host Inside iOS Dev Podcast. And you can find me on LinkedIn here.

1.2 Structure of this book

This book is broken down into a series of **seven steps** that you can follow to get a good grasp on what there's to learn about the iOS platform and what kinds of questions you could be asked on technical iOS interviews. Questions are grouped into logical **layers of responsibility** which we will talk about more in Chapter 2 Step One: Figure Out What the Big Picture Is, and we will talk in more details about the architectural aspects of it in Chapter 8 Step Seven:

Beyond MVC: Design Pattens, Architecture, FRP, and Dependencies Management.

1.2.1 Step One: Figure Out What the Big Picture Is

Step one is covered in Chapter 2. As mentioned previously, this is where we will talk about the big picture of what there is to learn about iOS.

1.2.2 Step Two: The Interview Game

In this step we will talk about everything non-technical in the interview process. Chapter 3 outlines a typical interview process, covering phone screening, onsite interviews, resumes, and more. This chapter will also give you some tips on how to market and position yourself so that you stand out from the rest of the candidates.

1.2.3 Step Three: Learn the Fundamentals

Regardless of what position you're applying for, fundamentals such as memory management and Swift reference and value type are important. Chapter 4 will cover those questions.

1.2.4 Step Four: Get Productive with Networking

In Chapter 5 we will talk about one of the most common tasks you'd do on iOS, talk to external APIs and services. This chapter covers questions about HTTP requests, Alamofire, JSON parsing, and so on.

1.2.5 Step Five: Learn How to Store Data

Chapter 6 covers everything storage and persistence. There are multiple ways you could store data on iOS and in some scenarios certain solutions are better than others. You'll find answers to questions ranging from `NSUserDefaults` to `Core Data` in this chapter.

1.2.6 Step Six: Go Crazy Responsive with UI Layouts

This step covers something that we as iOS developers arguably work on the most - the user interface. Chapter 7 will walk you through interview questions about AutoLayout, Frames, storyboards, UITableView, and so on.

1.2.7 Step Seven: Beyond MVC: Design Patterns, Architecture, FRP, and Dependencies Management

Good architecture is important on every project. Chapter 8 covers interview questions about design patterns and architecture. We will discuss things ranging from MVC and MVVM to Functional Reactive Programming and SOLID principles.

1.3 Bonus Content

This book also has a bonus Chapter 9 Storage Evolution (AKA You Don't Always Need Core Data!), the contents of which didn't really fit into the interview question/answer format of this book. They are more suited for another book. But things discussed and shown in that chapter, such as storage development and refactoring using SOLID principles, are nevertheless useful and important to know. This is why I decided to still add it to this book, because knowing these techniques could also help you in your interviews.

Alright, without further ado, let’s get into it.

Chapter 2

Step One: Figure Out what the Big Picture Is

A big picture overview makes it easier to orient yourself. So first things first - find out what the iOS world is all about overall and what the high-level overview of what you could possibly be asked about on iOS interviews is. You can figure out the details later when necessary.

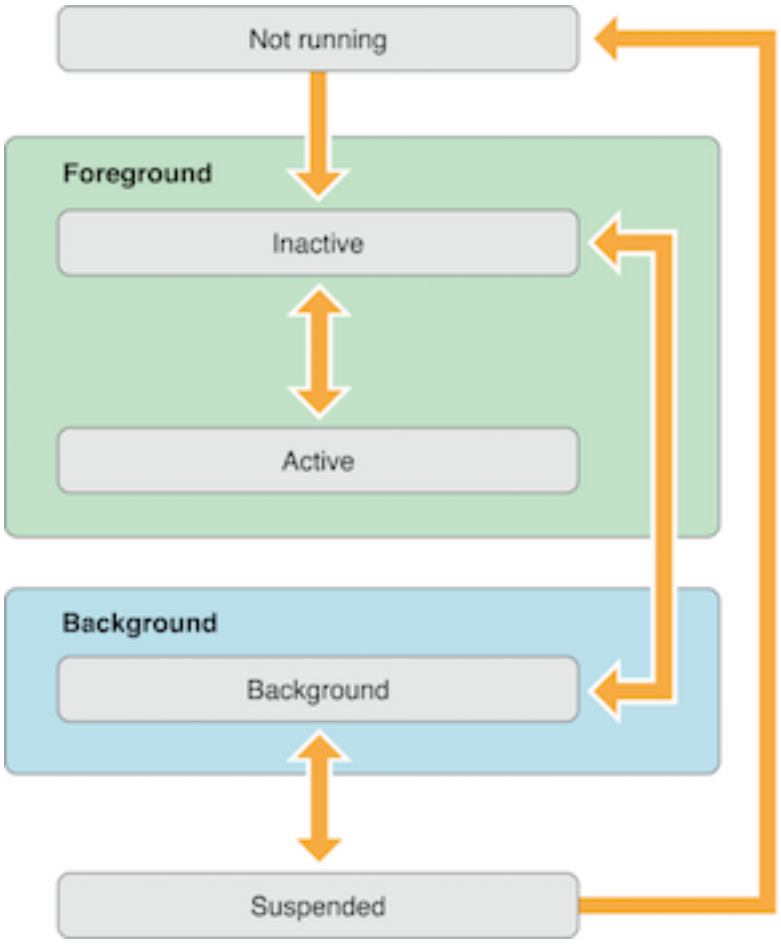
If you build enough apps you'll start noticing patterns. You'll see that there are things you do over and over again in one form or another that are essentially the same. When that happens, you realize how all apps are similar to each other. Sure they might differ in looks and what they do for the user, but overall, the way you build them is the same. Therefore when developers are interviewed for any iOS position, they will be asked a similar set of questions revolving around broad iOS topics. The main idea is that interviewers need to figure out what you know about building iOS apps.

In this chapter, we'll look at what iOS apps are, where they fit in the iOS system, the big picture design patterns that emerge out of building iOS apps, and how you can group and structure interview questions around those topics to systematize your own learning.

2.1 What is an iOS application and where does your code fit into it?

If you think about it long 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, home button press, and other app life cycle events.

Following is Apple's diagram of the iOS app life cycle:



It is indeed that simple and straightforward. The app is launched, and then it

sits and waits for user input, whether it's a touch or a home button click to put the app in the background, or something else.

`UIApplication` is just an object built around the `main()` loop to augment it and give us more usability that calls convenient callbacks to your `UIAppDelegate` subclass. Those "convenient" callback methods would be:

- `application:willFinishLaunchingWithOptions:`
- `application:didFinishLaunchingWithOptions:`
- `applicationDidBecomeActive:`
- `applicationDidEnterBackground:`
- `applicationWillResignActive:`
- `applicationWillEnterForeground:`

and everything else that we are used to working with in a typical app delegate.

What you actually do as an iOS app developer is just plug 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 and 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]?) -> Bool
    {
        let storyboard = UIStoryboard(name: "Main", bundle: nil)

        let rootViewController = storyboard.instantiateInitialViewController()

        self.window = UIWindow(frame: UIScreen.main.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")
    }
}
```

It is very typical to have something like that where you'd either use a storyboard or create an 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 the 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 launch! Heck, even all the callback methods that the **`main()`** loop under the hood sends to us will be received as they would be with a normal application that has a UI:

```
import UIKit

//@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {

    var window: UIWindow?

    func application(application: UIApplication,
                     didFinishLaunchingWithOptions launchOptions:
        [NSObject: AnyObject]?) -> Bool
    {
//        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` and `applicationWillResignActive` still call when you click on the home button and open your app again. The UI is just your app's code; the system doesn't care if you have any or not. It just keeps running its `main()` loop.

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

2.2 Patterns and Layers

After you build a few iOS applications of various complexities, one thing you might start to notice is that there are distinct layers of responsibility in each app's codebase. Regardless of whether you're building an Instagram-like application or a mail client or any other kind of app, all of them will be doing one or more of the following things: HTTP networking, storing data to disk, location GPS work, JSON parsing, data serialization, UI composition, resources and objects coordination, and other tasks.

All of those things in your code can by grouped into the following layers of responsibility: `storage layer`, `service layer`, `business logic layer`, and `UI layer`.

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

2.2.1 Storage Layer

The 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 responsibility of this layer is to store data for your application and play

the role of the **"ultimate source of truth"** for the rest of your code. Examples of what goes into this layer could be the following: `Core Data`, `Realm`, `NSUserDefaults`, `KeyChain`, `Disk File storage`, and `in-memory arrays and dictionaries/sets`.

We'll cover interview questions around storage in Chapter 6 Step Five: Learn How To Store Data

2.2.2 Service Layer

This layer is responsible for all things involving networking and external communication. That could be, as needed by pretty much any app these days, 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 `Bluetooth Low Energy (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 some other piece of data. Or it could be a location service that connects with a 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.

We'll cover interview questions around networking and services in Chapter 5 Step Four: Get Productive with Networking

2.2.3 UI Layer

The 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`**, and **`Bar Buttons`**. Two other things that also belong to this layer that might not be obvious are **`View Controllers`** and **`View Models`**. **`View Controllers`** are suppose to do 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 a happy and healthy iOS codebase is a skinny controller.

We'll cover interview questions around UI and layout in Chapter 7 Step Six: Go Crazy Responsive with UI Layouts

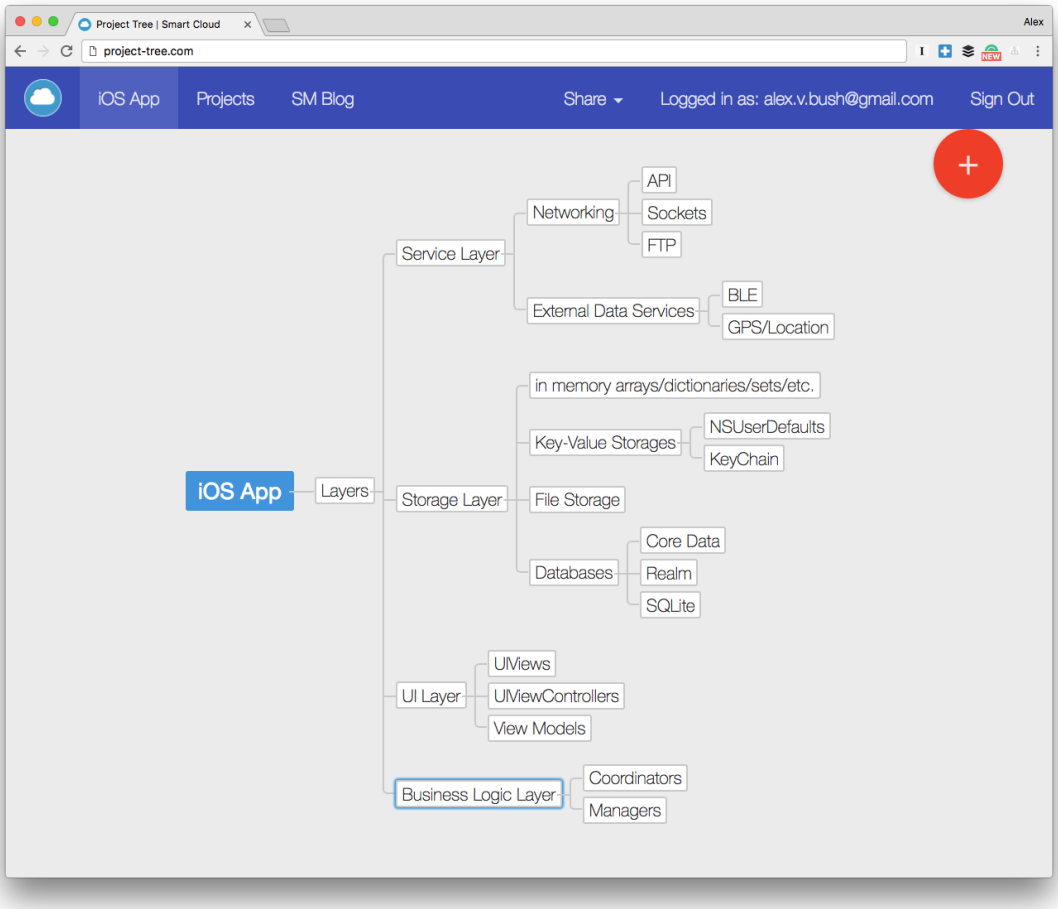
2.2.4 Business Logic Layer

In this layer are objects that are responsible for the actual application's business logic, objects that use components of other layers to achieve results and do the work for the user. Coordinators that use **`HTTP`** service objects in conjunction with storages to orchestrate receiving data from backend **`APIs`** and persisting it to **`Core Data`** would be one example.

Another example of what goes into this layer could be a manager object that takes care of token encryption and saving to keychain using keychain storage and some kind of encryption service in it.

The main idea is that this layer helps us keep services, storages, and other layers decoupled from each other and tell them (aka orchestrate and coordinate) what to do to achieve results. This layer is what actually makes your application useful.

This is how the layers structure looks overall:



You might be wondering why you're seeing this particular layers breakdown. It is inspired by the Single Responsibility Principle (SRP), one of the SOLID principles (which stands for Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation and Dependency Inversion). We'll discuss it in more detail in Chapter 8 Step Seven: Beyond MVC: Design Patterns, Architecture, FRP, and Dependencies Management

2.3 Zooming Out

So what does all of this mean for you and your interview prep? Layers in iOS codebases that we just discussed essentially group all the things that you should know as an iOS developer. That effectively means that when you prepare for interview questions each one of those questions could be placed in a respective group according to the purpose of the thing the question is asking about.

This is how this book is structured - instead of randomly preparing for questions that were put together in some arbitrary way, we instead will systematically take a look at each layer and the things you need to know there.

2.4 Zooming In

Now, when you know overall what there is to learn on iOS, you could go through each step and chapter one by one, reading questions and answers in order. Or you could skim them and skip the questions you already know answers to and are familiar with and instead focus on those areas where you're lacking.

Regardless of the way you approach it, systematizing your learning and knowing overall where you are in grasping iOS should give you a framework to work in and give you more confidence on the interviews themselves.

2.5 Conclusion

In this chapter we looked over the big picture of what there is to learn on iOS. We now have a plan of attack for prepping for questions about each layer of responsibility a typical iOS application has.

In the next chapter we are going to look at what a typical interview process consists of and how you can increase your chances to get noticed and get invited

to be interviewed.

Chapter 3

Step Two: The Interview Game.

In this chapter I'll outline a typical interview process from end to end. It will start with advice on searching for a job or receiving invitations from recruiters (or both), and then guide you through the interview itself before finishing with some advice on salary negotiation. This chapter isn't going to be as tech heavy as others in this book. Instead it focuses on stuff you need to know around software development craft such as marketing (yes, as a developer you need to be able to market and position yourself well to get a better job), negotiation, and soft skills.

3.1 Before The Interview

Your success in landing your dream job starts even before you get your foot in the door at the company you are applying to. There are three things you need to do before you go on an interview: **Job Search**, **Marketing**, and **Knowing Your Shit (a.k.a. Preparation)**.

3.1.1 Job Search

Everything starts with a job search. There are several ways you can do this and they are not mutually exclusive. You can apply for jobs through various job boards and company websites, talk to your friends, get referrals through word-of-mouth, and work with recruiters.

Now, let's look at these job search methods in more detail:

Job Boards and Company Websites:

The most straightforward approach to the job search is to put together a resume and portfolio and apply for positions posted on the internet. This includes online job boards and listings on Stack Overflow, LinkedIn and other professional networking and recruitment websites. There is nothing wrong with this approach and it's a perfectly fine way to land a job. It is important, however, to have a good resume that makes you stand out. We'll talk more about that later in this chapter.

Another important thing to know about applying for jobs through job boards is that you cannot win by playing a numbers game. Don't blast the same resume and cover letter to every position and company you can find. To stand out you need to tweak them so that they specifically suit each job you are applying for.

Referrals/Friends:

Having a friend or acquaintance refer you for a position is another good way to land a job. Ask everyone you know if they've heard of anyone who needs a developer with your skill set. You might be surprised how close open positions are to you. Because you'll get points by proxy through the person that referred you, your resume is less important with this approach but you should still make it impressive.

Recruiters:

Oh, recruiters. No one likes them but they're actually very useful in landing a job After you get over the stereotype and start actually working with a recruiter, you would find that they do a great job of lifting the burden of searching for

positions off your shoulders. After all, they are on your side. Sure, they get a nice payout when you land a job. It's in their interests, though, to get you the best job they can because if you get a big salary, they can negotiate with your new employer for a bigger cut. Give them a good resume to use and make adjustments to it if they ask for them. Also, tell them what you are ideally looking for and your salary preference. Then sit back and relax and let them source new gigs for you (if they are any good, of course).

Always remember, too, that if the recruiter you are working with is not producing results, don't be discouraged. Try a different one and keep searching for jobs on your own.

3.1.2 Figure out what team/company size you want to work with

Team size matters. It should influence your expectations as a developer and will affect the way an interview is conducted. When you apply for a position in a company, you need to know how big the engineering team in that company is and what their culture is like. Then you'll know the kinds of questions to expect on your interview.

Small Company:

If they are a small startup/company (0-10 people in the engineering team) with "hackish" culture then they'll most likely ask you about prototyping things and will expect you to "build shit quickly" disregarding quality and praising the speed. They won't care much about good architecture and scalability or performance at early startup stage they care more about "time to market" which means quickly putting together something that works most of the time and shipping it. The expectation on you as a developer (especially if you'll be the sole iOS developer on the team) would be that you can deliver apps end to end - from the very first line of code down to the release submission to the app store with all the necessary provisioning profiles etc. You need to be able to figure things out quickly. Most likely you won't be working with legacy

codebases here, all the projects are apps started from scratch, so called “green field” projects.

Large Company:

Interviewing at a large company (50+ people in the engineering team) you will probably be asked generic computer science questions and sometimes interviewers won’t even go into iOS-specifics. Big organizations have typically already figured out what their product is and are scaling product and marketing efforts. For you, this means that the interview is likely to revolve around hypothetical problem-solving and scalability. Large organizations care about the performance impact of file downloads, network requests, and other computations made on the client side. Unlike small organizations, there most likely will be time for doing optimizations. Timelines in bigger organizations are longer and typically they are looking for people to fill specialized roles. You’ll often find that large organizations like Facebook or LinkedIn are looking for developers (and even entire teams) to focus on things like UI performance, networking download speeds or other deeper specializations in iOS software that smaller companies can’t spend time on. Also expect to be working with legacy codebases.

Mid-size Company:

When applying to a mid-size company (10 - 50 people), you should expect a mix of what small companies need and big companies expect. Mid-size companies are not big enough to spare resources on people who specialize in things like performance but aren’t small anymore to move super fast and risk breaking things. They’ll expect you to be a well-rounded and balanced developer. As with a small company, in most cases you will be expected to build and ship things end-to-end but you will also have to keep an eye on performance, scalability, and architecture for future maintainability. If you interview at a good mid-size company, the questions are likely to encompass all of these areas as well as cover broad iOS, architecture and CS knowledge.

I personally find teams at mid-size companies to be the most challenging and interesting to work with because they demand more than simply building an MVP. They need someone whose talents go beyond shaving microseconds off

networking performance or scrolling in table views.

The main message here is that you should figure out what type of company and team you'd like to join ahead of time and prepare for your interview accordingly. Regardless of your choice, though, the advice in the following chapters about interview questions on UI, Networking, Storage, and other technical iOS topics will help you prepare for interviews at any company regardless of its size.

3.1.3 Marketing

This is something that most developers neglect. I know the word "marketing" can sound sleazy and unappealing. Yet, when you're aiming at your dream job, you need to stand out from the crowd if you want to get a foot in the door.

The way to do this is through marketing yourself with your resume, your blog, your GitHub profile, and other links and resources that showcase you as a great developer and an appealing candidate.

Resume

There's a notion that resumes are overrated and that some companies don't even look at them. Ignore it. There are still plenty of HR personnel and other recruitment gatekeepers that filter candidates out based on resumes so it makes sense to have a good one.

The main purpose of a resume is to briefly show that you have relevant experience for the position you're applying for. It won't get you the job though, you will still have to interview but it will increase your application response rate.

HR departments and other people involved in hiring devs typically get hundreds of resumes a day and have very limited time to spend on each one. Therefore, keep your resume under two pages and don't overly complicate it with long, smart-sounding words and corporate BS. Again, people who look at your re-

sume don't have much time, so make it easier for them and keep it short and concise. They'll appreciate that.

If you're applying to a US company, your resume structure should probably be as follows (sections going from top to bottom):

1. Your name and contact information, such as e-mail and phone number, should go at the very top. Below these you should add links to your blog and profiles on GitHub, Stack Overflow and anywhere else that showcases you as a great developer. Don't overdo it by including more than a few, though.

2. Your work experience. Include the names of companies you've worked at and your positions along with the dates of your employment. Also, add a very short (one or two sentence) description of what you did along the lines of, "I was working on the main iOS app and successfully implemented integration with internal JSON API". You can also mention one or two big technologies or architectural concepts you used at each job.

3. Your personal projects. This would include any relevant interesting stuff you've done aside from paid work.

4. Your education. State the name of the college or university you went to and what you majored in but nothing else. Employers won't really care about your education unless they are Google, or Amazon, or a similar size company.

5. Optionally, you can also have a section with any special expertise you have that might be relevant. In my resume, for example, I have Functional Reactive Programming (FRP) and SOLID principles. This section is not essential and can be omitted if it makes your resume too long or complicated.

Don'ts:

Do NOT include the following things in your resume:

- photo of yourself
- marital status
- date of birth
- career objective

It's unfortunately true that by excluding the first three of these from your resume you can avoid sources of negative bias. If you represent a minority group, are of a certain age or have a some kinds of social status, it might influence a recruiter's decision to consider you as a candidate. This is a complicated issue that could be the topic for another book so simply take my word for it and don't include these things in your resume.

As for stating your career objective, it's just filler that no-one cares about in a resume. If an employer is interested, they'll ask you at the interview.

References:

Don't include references in your resume but have a few available. Offer to provide them upon request.

Cover Letter

Cover letters are important. When you apply directly for jobs, the cover letter will be the first impression you make. It can be a deal breaker or a deal maker. If a company sees that you copy-and-pasted a form letter, they could take that to mean you don't care about joining them specifically. If it looks like you've taken the time to craft a cover letter for the company and the position you're applying for, you'll greatly increase your chances of getting an interview. In more general terms, keep your cover letter succinct, to the point and, for God's sake, spell check!

Writing a good cover letter means doing your research. Unless you already know everything you need to know about the company you're applying to,

Google it a lot. Find out what their team size is and what tech they work with. Read any technical and product blogs they have. Most importantly of all, research and, if possible, try out the product they make. You can't imagine how many developers are disregarded for jobs simply because they show no interest in the products made by the company they're applying to.

Github

Your code speaks more loudly about you than anything else. GitHub is the de facto standard for code version control nowadays. Not only should you know how to use `git` well enough, you should also have an up-to-date GitHub profile.

For some companies GitHub is the main tool used for filtering out applicants for developer jobs. Having an active GiHub account shows that you're a part of the software developer tribe and active in that community.

Your GitHub profile should have a very short description of what you can do and showcase what you've worked on. Potential employers find it handy to go to someone's GitHub profile and see their code before deciding whether to invite them for an interview.

In that rare case where the code you wrote can't be made public because of a non-disclosure agreement (NDA) or something similar, try to get a few code samples together that you can share. We'll discuss this more in the next section.

Code Samples

Good code samples that are on a GitHub profile or have been included in your application often make potential employers more inclined to send out an interview invitation. If you can't share much in your GitHub profile due to an NDA, ask your previous employers or clients if you can share only a portion of the code you wrote. That could be a set of classes that constitute a feature that you built, networking code you've written, or something similar. Try to find

something that shows complete end-to-end coding for a feature you've built and reassure the owner of the intellectual property that you won't reveal too much about the rest of the application.

Blog

Your blog can also help you stand out. If you regularly blog, it's easier for others to put you in a bucket of professionals. Share everything you learn. If you've just read an article about MVVM and tried it in your app, blog about it. If you found a new cool pod that helps with animation, blog about that. Just like your GitHub profile, your blog helps communicate your skills and your value to potential employer before you even meet the person who's going to interview you. Trust me, there's a special feeling you get when an interviewer casually says that they read something on your blog that they found useful or they want to talk about. Use blogging to your advantage!

3.1.4 Preparation (Know Your Shit!)

Aside from the job search and marketing, the most important thing you need to do before every interview is to prepare and learn as much iOS stuff as you can. Tech skills and soft skills are actually what you'll need day-to-day on your job so you've got to be ready. That is what the rest of this book is about. Specific questions that you should prepare for on UI, Storage, Networking and other things that you'll need to know are broken down in subsequent chapters. Read them.

3.2 At The Interview

When you're interviewing with a company you'll actually have multiple interviews at different stages of the process. Typically these break down into the

following steps: **phone intro**, **phone/skype/hangout/voip interview**, **onsite interview**, and optional **negotiation interview**.

3.2.1 Phone Intro

The phone intro is the first thing you'll have after company you applied to expresses interest in you. Its purpose is to allow you and the company to get acquainted and see if your mutual goals are aligned. Generally speaking, a company will try to sell themselves to you as a great place to work and simultaneously gauge your level of excitement and eagerness to work with them or in the field they specialize in. Don't stress too much about this interview. There won't be any technical questions and you'll only be expected to briefly talk about your previous experience. Have a quick and short summary of it in mind and don't get into lengthy details about what you've done before. You should have a few questions ready about the company interviewing you. They wouldn't want to hire someone who's not interested in them. Typically, this interview takes only 15 to 30 minutes.

3.2.2 Phone and/or Skype/Hangout/Voip Interview

A second interview can take various forms. Typically it is a technical interview that could either be purely Computer Science-oriented (i.e., about algorithms, data structures, etc.), or iOS-oriented (iOS tech questions with short answers to gauge your overall knowledge). It might also have a mix of both. These interviews can be either over the phone, so you'd be expected to just talk, or they could be conducted via Skype, Google Hangouts or another VoIP service. You might be asked to use VoIP so you can share your screen and the interviewer can see what you're typing while you solve a problem they've given you.

These interviews are typically not as crazy hard as you might think and they usually take from 30 minutes to an hour. If they take longer than this that can be a good sign.

Again, get ready for the technical questions by reading the chapters of this book.

It's okay to ask the interviewer to repeat a question that you didn't understand. If they give you a problem to solve, talk through your solution before typing it out. Usually, the interviewer just wants to understand how you think rather than see you get the right answer of the bat.

I have one other important piece of advice if you are asked to interview via a VoIP service: ensure that you have a stable internet connection (preferably through a cable because WiFi is unreliable) and a quiet spot to talk for an hour because you don't want the interview to get interrupted or cut off in the middle.

3.2.3 Onsite Interview

Onsite interviews are generally the hardest and longest. The interview style, questions, organization, and other details will vary from one company to another depending on their size and culture. In general, though, expect there to be at least one whiteboarding session (with or without CS and iOS questions, architectural discussion and problem solving), one or more pair programming sessions or a mix of each of these. Again, questions covered in the rest of this book will help you prepare.

Dress well to make a good impression, even if you're going to a hip startup in Silicon Valley

Even if you've asked questions in a previous interview about the company and the team that you could be working with, prepare more. You will be given a chance to ask them and they will show the extent of your interest in working specifically for the company that is interviewing you.

Whiteboarding

Whiteboarding can be intimidating for people but don't get discouraged. It's not an exam and you're not in school anymore. What interviewers really want

to see is how you tackle problems and how well you know iOS stuff. Take a deep breath, be calm, and talk your solutions through. Ask questions if you need to clarify something.

Pair Programming

Pair programming is one of the best ways to gauge a candidate's level of experience and general cultural fit. That's because it's the closest thing possible to what the interviewee will actually be doing on the job if hired. The same advice applies here as with other types of interviews: be calm, don't hesitate to ask questions, and talk your solutions through because your interviewer will want to know how and why you've decide to write a particular piece of code before you do it.

3.2.4 Salary Negotiation Interview

A salary negotiation interview sometimes happens right after your on-site interview or might occur later over the phone or VoIP. Negotiating is usually the hardest thing for developers to do and many simply agree to whatever is offered. There's a lot of benefit to be gained from discussing salary, though. Listen to Ruby Rogues Podcast Episode #274 for an at-length discussion of why it's important and what tactics will help you do it successfully. John Sonmez also covers salary negotiation in his great book Soft Skills.

If you work with a recruiter, they should take care of this part for you.

3.3 Importance of Soft Skills

So called "soft skills" are a set of your skills such as people skills, communication skills, productivity skills, organizational skills, attitudes and emotional intelligence (EQ). In other words, anything that is not directly related to, but

is just as important as, your "hard skills" or technical knowledge. Sometimes teams will prefer someone who is a great communicator over a coding genius who can't get along with others. So, in your interviews be at your best and smile to make a good impression on the people you're interviewing with

Read that John Sonmez book for more on this topic.

3.4 Keep Track of Progress

Finally, I highly recommend to have a Trello board for your job search to keep track of progress, e-mails, and phone calls about the jobs you're interviewing for.

3.5 Conclusion

In this chapter we went through the overall structure of the interview process. It should be easier for you to handle now that you know what to expect. The rest of this book covers the technical knowledge you will need to be calm and confident enough in your interviews to crush them!

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, such as memory management, protocols, extensions, let/var, optionals, KVO and delegates. Depending on the position you're applying for (junior, mid, senior, etc.) you'll either have a lot of these questions or just a few but expect them in one form or another. These questions can be tedious and boring for an experienced developer but it's 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 would you use one or the other?
- How is memory management 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 are they 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 does iOS application lifecycle consist 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 in either direction.

Expected answer: The `let` keyword is used to declare a constant variable and the `var` keyword is used to declare a variable. Variables created with these 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 (or within the calling scope) and you can't reassign it. In contrast, 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. In Objective-C everything is dynamic and can be `nil`. Also, `nil` can receive messages

without breaking everything (i.e. throwing an exception). In Swift, though, 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 in 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 talk at length about the big picture architectural implications of this and how it is going to affect how you write your code.

Expected answer: In Objective-C `nil` used to be a very handy "value" for variables. It typically meant an absence of value or 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`). With the introduction of `let` and `var` in Swift, however, it became apparent that not all constants and variables can be defined and set at the time of declaration. We needed to somehow declare that a variable has not been determined yet and that it potentially could have a value or no value. 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 set one 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 can be used with constants (**let**s) only if they were give a value right away (whether it's **nil** or an actual value). In general a constant has to be defined at the time of its declaration and therefore it has a value and is not an Optional.

In Swift, unlike in Objective-C, sending messages to **nil** causes a runtime exception. There is, though, a way of sending a message to an Optional in Swift and if the value is a **nil**, it will just ignore the message and return **nil** instead of raising an exception. This is much like the old Objective-C behavior and allows you to do method chaining calls. If one of the Optional values in the call sequence is **nil**, the whole chain will return **nil**.

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

In general, you should use **nil** and consequently **Optionals** to represent an absence of value as little as possible. Every time you declare an Optional, ask yourself if you really, really need it.

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

Red flag: Besides not knowing what Optionals are and how to work with them, the biggest red flag for an interviewer would be if you speak in favor of Optional Binding and explicit Optional Unwrapping. The former leads to poor design and cognitive overhead of if/else statements and with the latter there is the potential danger of runtime exceptions.

4.3 What is the difference between `struct` and `class` in Swift? When would you use one or the other?

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

Expected answer: Both `structs` and `classes` in Swift can have properties, methods, subscripts or initializers, be extended, and 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 (structs can't do that), type casting, and deinitializers (former `dealloc`).

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

The key thing about 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.), use `struct`. However, if you need custom behavior where passing by reference is preferable (so that you refer to the same instance everywhere), use `class`.

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

4.4 How is memory management handled in iOS?

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

Expected answer: Swift uses Automatic Reference Counting (ARC). This is conceptually the same thing in Swift as it is 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. It deallocates memory used by objects which reference count got down to zero. ARC does not increase or decrease the reference count of *value types* because, when assigned, these are copied. By default, if you don't specify otherwise, all the references will be strong references.

One of the gotchas of ARC that you need to be aware of is Strong Reference Cycles. For a class instance to be fully deallocated under ARC, it needs to be free of all strong references to it. But there is a chance that you could structure your code in such a 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 these approaches will assign an instance without keeping a strong reference to it. Use the `weak` keyword for one and the `unowned` keyword for the other before a property or variable declaration. *Weak reference* is used when you know that a reference is allowed to become `nil` whereas *unowned reference* is used when you are certain that the reference has a longer lifecycle and will never become `nil`. Since `weak` references can have a value or no value at all, they must be defined as optional variables. An 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`. If `self`, in turn, retains that closure, you'd have a mutual strong reference cy-

cle between closure and class instance. This often occurs when you use lazy loaded properties for example. To avoid it, 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 are defined with a pair of square brackets and every element in them has a `weak` or `unowned` keyword prefix and is separated from other elements by a comma. The same thinking applies to a closure capture list as to variable references: define a capture variable as a `weak` Optional if it could become nil' and the closure won't be deallocated before then, and define a 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 a memory management question or a standalone question. It is very important to understand properties, instance variables, constants, and local variables when working with Objective-C and Swift because they define how you refer to and work with your data.

Expected answer: `Properties` in Objective-C are used to store data in instances of classes. They define the memory management, type, and access attributes of the values they store such as `strong`, `weak`, `assign`, `readonly` and `readwrite`. 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 that declaration will also synthesize it, meaning create a getter and setter to access and set the underlying instance variable.

The **strong**, **weak** and**assign** property attributes define how memory for a 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** are **strong** by default. They can be declared as weak or unowned references with the **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, however.

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 the **static** keyword for value types (**struct**, **enum**) and with the **class** keyword for class types. In Objective-C, since Swift 3 and Xcode 8, you can also define class properties using the **class** keyword in property declaration.

Properties in both Swift and Objective-C 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. Nonetheless, you do 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 is it used?

Protocols are vital for any strongly typed OO language. Both Objective-C and Swift use them and you should expect to be asked about them 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. A protocol only has a description or signature of the methods, properties, operators, etc. that a type implements without the actual implementation.

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 OO language because they allow you to clearly and explicitly declare interfaces of things in 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 to dynamically change objects that conform to the same protocol at runtime. It also lets you abstract things out and code against interfaces rather than specific classes or other types. It helps, for example, with the implementation of core Cocoa Touch design patterns such as `Delegation`. Also, developing against protocols could help with test-driven development (TDD) because stubs and mocks in tests 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 to not only use and adopt existing protocols that Cocoa Touch provides but also 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 when developing with Objective-C and Swift. Having a handle on the benefits and limitations of categories and extensions is an important skill so expect this question on pretty much every interview.

Expected answer: `Categories` in Objective-C and `Extensions` in Swift are ways to extend existing functionality of a class or type. They allow you to add additional methods in Objective-C and Swift 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, `categories` are typically used to extend the functionality of 3rd-party or Apple framework classes. You can also use them 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 the functionality of existing types or to make a type conform to a protocol.

The drawback of extensions and categories is that they are globally applied, unlike protocols. This means that 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.

Neither categories nor extensions can add new stored properties.

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, you can't predict what implementation will take precedence at runtime. To avoid that collision, you should namespace your methods with a prefix and an underscore; i.e., something like `func ab_myExtensionMethodName()` where `ab` is your codebase's class/type name prefix (same convention as with the `NS` prefix for Cocoa's legacy NextStep).

Red flag: Extensions/Categories used to be an advanced feature of Objective-C and Swift but not any more. The key is not to abuse them.

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

Blocks and closures are an integral part of Objective-C and Swift development. This question 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 in 100% of interviews.

Expected answer: Blocks in Objective-C and closures in Swift declare and capture a piece of executable code that will be launched at a later time. You can either define them in-line 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 work that needs to be executed at later time (a.k.a. asynchronously).

Blocks/closures are reference types and will retain/strongly reference every-

thing put in them unless otherwise specified. You can avoid strong reference cycle issues by using the `__block` and `__weak` keywords in Objective-C (or, better still, use @strongify/@weakify) and `[weak self]`/`[unowned self]` in Swift.

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

If those domain names are offensive to you, try these more friendly alternatives: 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, good old MVC. This is a fundamental design pattern Apple keeps pushing onto iOS developers. Every single interviewer will ask a question about this.

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 are captured and represented by Models. Views are responsible for drawing things on the screen. 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 an 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, `NSManagedObject` subclasses and similar are the `Models`

MVC is a great general purpose design pattern but using it solely limits your architecture and often leads to notorious "Massive View Controller". "Massive View Controller" is the state of a codebase where a lot of logic and responsibility has been shoved into View Controllers that doesn't belong in them. That practice makes your code rigid, bloated, and hard to change. There are other design patterns that can help you remedy this, such as MVVM and the general SRP principle. Even though Apple keeps telling us that MVC is everything, don't be fooled by it and stick to SOLID principles. We'll talk more about MVC, MVVM, SOLID principles, and design patterns in general in Chapter 8.

Red flag: You absolutely have to know what MVC is. It's basic to any iOS development. At the same time, though, 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 a background in something other than just iOS.

Expected answer: `Singleton` is a class that returns only one-and-the-same instance no matter how many times you request it.

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

Singletons are often misused and can breed in global state, which makes debugging and working with your code difficult. It starts off very innocently when

you think you'll have only one instance of a class and you make it globally available across your codebase. But at some point you need to either reset it, do something else with the data stored on it, or realize that sharing it across the whole app doesn't make sense anymore. This is when you get into trouble because your singleton is everywhere now and data stored in it is unreliable because you don't know who might've changed it and when.

Using singletons makes it hard for you to inject dependencies because with a singleton there's only one instance of your singleton class. That prevents you from injecting it as a dependency for the 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 this way leads to a disaster.

4.11 What is Delegate pattern in iOS?

Like MVC, this is one of the fundamental Cocoa design patterns. This 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`. These are both represented by a protocol that an object conforms to and UITableView uses the single object it is provided with to send messages/events. Unlike with Observer pattern, there can 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 because that might lead to unexpected consequences. The delegate/delegatee relationship is tightly coupled.

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 provides mechanics through which you can observe changes on properties in iOS. In contrast to Delegate KVO entails a one-to-many relationship. Multiple objects could subscribe to changes in a property of another object. As soon as that property changes, all objects subscribing to it will be notified.

Under-the-hood implementation uses 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, this 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 that you can build on. There are a lot of great FRP projects like ReactiveCocoa and RxSwift that were built using KVO mechanics.

4.13 What does iOS application lifecycle consist of?

As iOS developers we simply have to know what's going on with the app we're building. Application lifecycle questions are intended to show how you under-

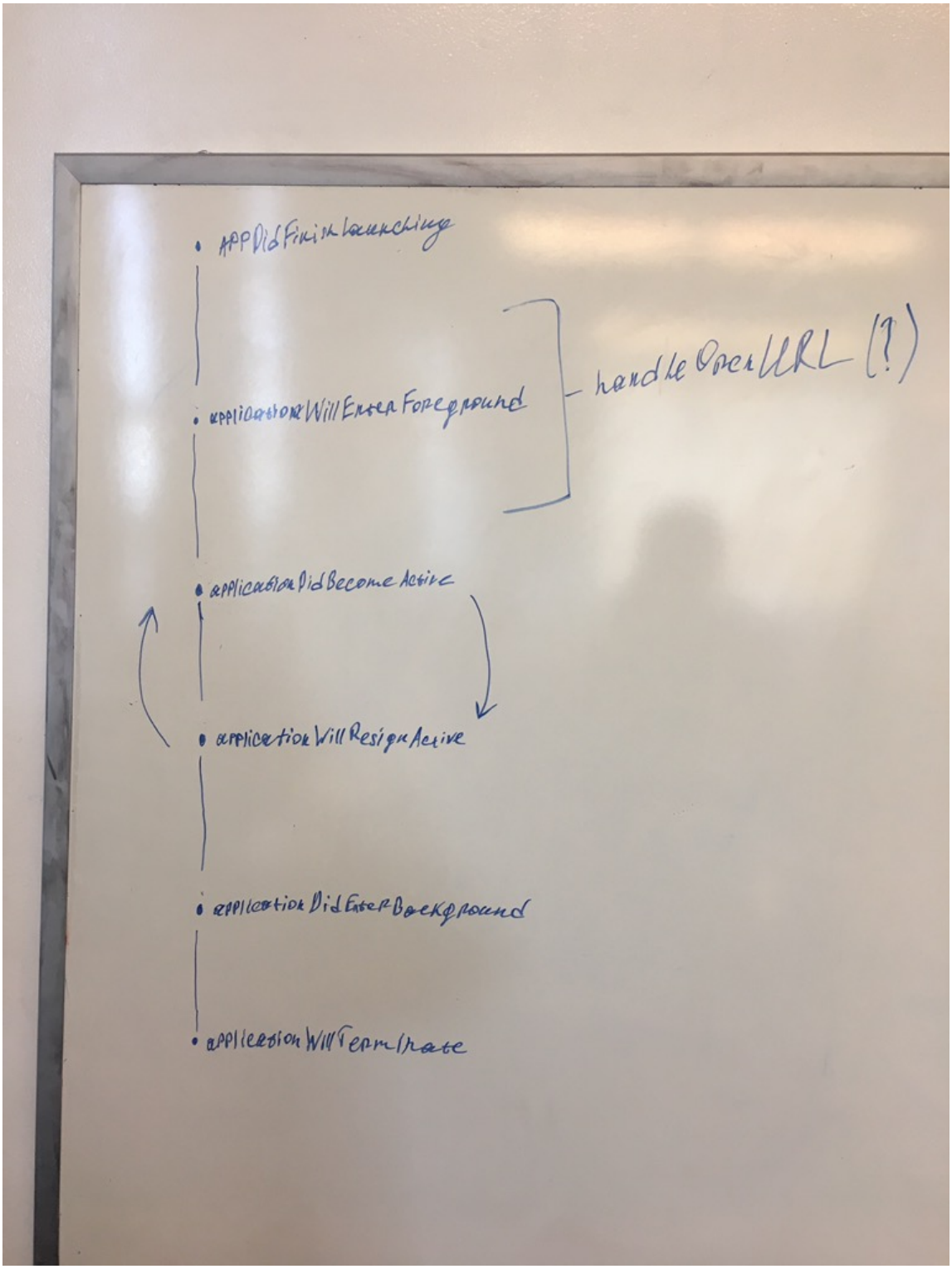
stand an iOS app's overall behavior in the system.

Expected answer:

The main point of entry into iOS apps is `UIApplicationDelegate`. `UIApplicationDelegate` 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:

(see a picture on the next page)


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

When an iOS app is launched the first thing called is `application: willFinishLaunchingWithOptions:-> Bool`. This method is intended for initial application setup. Storyboards have already been loaded 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 has 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 `applicationWillEnterForeground:` to finish up the transition to the foreground.

Termination

- `applicationWillResignActive:` is called when the app is about to become inactive (for example, when the phone receives a call or the user hits the Home button).
- `applicationDidEnterBackground:` is called when your app enters a background state after becoming inactive. You have approximately five 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 the given activity or url.

Knowing your app's lifecycle is important to properly initialize and set up your app and objects. You don't have to run everything in `application: didFinishLaunchingWithOptions`, which often becomes a kitchen sink 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 an expansion on the MVC question.

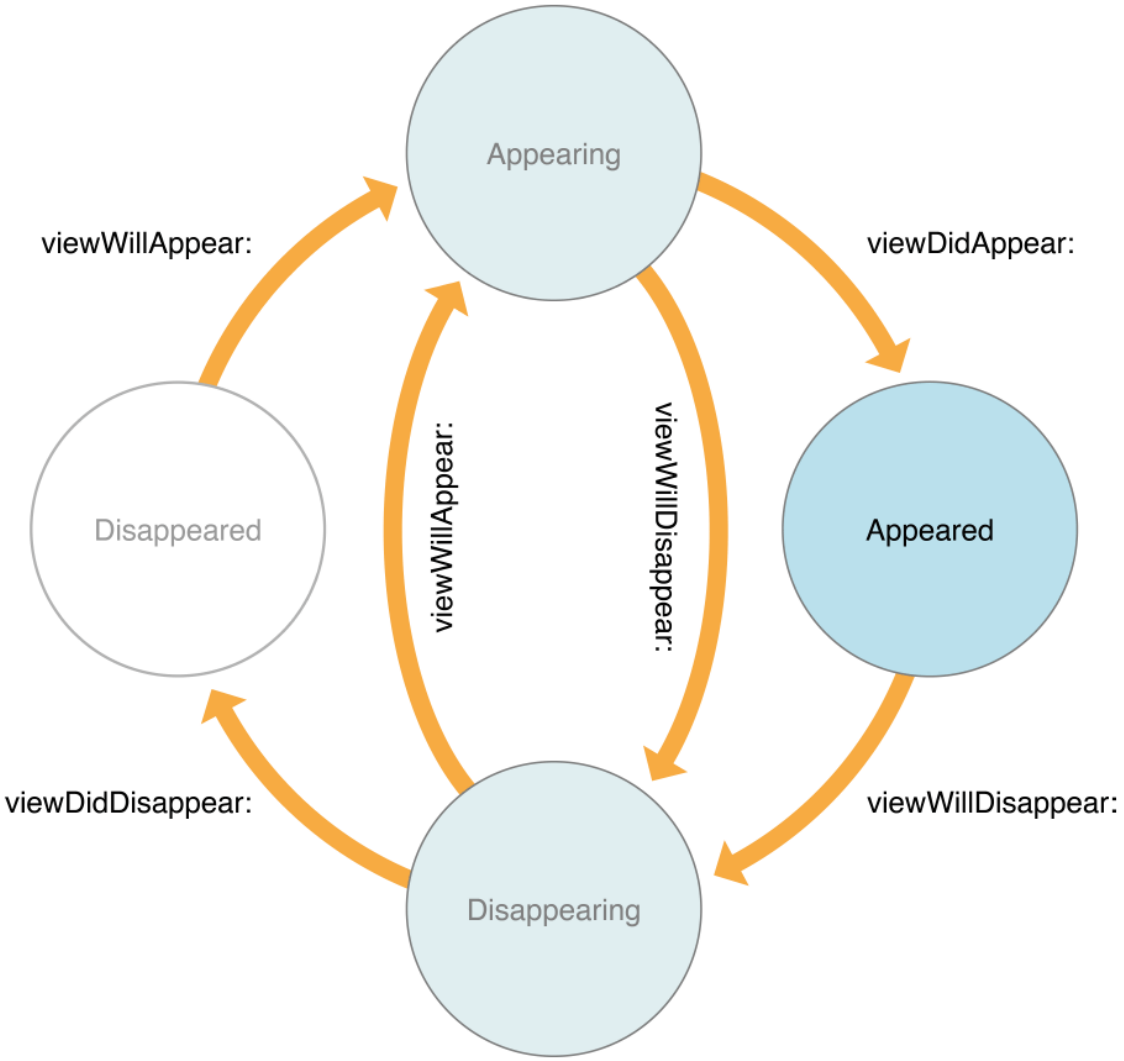
Expected answer:

View Controllers (VC) are the Controller part of the MVC triangle. They are responsible for controlling the view. Every time you want to display more or less significantly complex pieces of UI, you would want to use a View Controller to handle the lifecycle and user interaction callbacks of that UI. In a 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 by another 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 once when your VC's view was loaded in memory for the first time. Do any additional setup and initializations here. Typically, this is 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. Later still, if the user taps the Back button, **`viewWillAppear`** and **`viewDidAppear`** for that first VC will be called because it becomes the top VC again. Use this method to do final UI customizations, hook up UI observers, etc.

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

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

- **`viewDidAppear()`**: this method is called right after the view was shown on the screen and follows a **`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, because 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 it will be via `-initWithCoder`.

Red flag: As with memory management, you simply have to know this stuff to be able to develop iOS apps.

4.15 Conclusion

You'll encounter fundamental questions on every interview in various forms. These are the basic questions and answers that it is absolutely necessary to know and understand in order 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 interconnection age of ours. Therefore it's 100 percent guaranteed you'll be asked the questions covered in this chapter at every interview you go on. 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 implement networking on iOS?
- What are the concerns and limitations of networking on iOS?
- What should go into the 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? What are the advantages and disadvantages?
- What could you 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 know the meaning of the verbs used with it. You won't be tested on theory and the history of HTTP but you should be able to talk about the basics of the protocol that powers the modern-day 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 different 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 used to, well, 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 is supposed 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 APIs adhere to HTTP standards and respect 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` requests 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 meaning of HTTP verbs and the expected server behavior when using them.

5.2 What is REST?

REST stands for Representational State Transfer. REST is an API architecture built on top of HTTP protocol. Its main focus is resources and the ability of client applications to access, write, delete, and alter them. As far as iOS developers are concerned, it is the most popular API architecture for third-party services and many internal product APIs. Knowing what **REST** 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** users create in your app. A typical CRUD REST API for it would look like this:

- **https://yourawesomeproduct.com/posts** accepts **GET** requests and returns a list of **posts** available on the server.
- **https://yourawesomeproduct.com/posts/123** accepts **GET** requests and returns 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 the iOS client application.
- **https://yourawesomeproduct.com/posts/123** accepts **PATCH** requests to alter certain data in a specific **post** with a given **ID**.
- **https://yourawesomeproduct.com/posts/123** accepts **PUT** requests to replace an entire set of data in a specific **post** with given **ID**.
- **https://yourawesomeproduct.com/posts/123** accepts **DELETE** requests to destroy a **post** with a specified **ID**.

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

If the API you're using is truly RESTful then 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 have at least a basic idea of what `REST` and `RESTful` backends are.

5.3 How do you typically implement networking on iOS?

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

Expected answer: Networking falls into the `service layer` of your application since it deals with external communication. In general, you should decouple everything HTTP/network-related in your app into a set of service and client objects that handle all the nitty-gritty 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 service 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 the de facto standard for doing HTTP networking on iOS and following the Single Responsibility Principle (SRP) is vital for codebases big or small.

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

The aim of this question is to gauge your understanding of the 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. When developing the networking layer of the app, you should always issue as few HTTP requests as possible

and retry requests if they suddenly fail due to a poor connection or other issues.

There's also a bandwidth issue; it is not a good idea to upload or download large files and chunks of information when using a cellular connection and it is advised to use Wi-Fi instead.

5.5 What should go into the networking/service layer?

This is a conceptual and architectural question. Every application consists of several layers of responsibility and the 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 the service and networking layer in iOS apps according to SRP.

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, and so on. They are all 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 Bluetooth Low Energy (BLE):

- `BLEClient` object that owns and manages `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 a connection to BLE, discovery, and communication with the specific device/peripheral you're trying to connect to. `SpecificDeviceService` is also responsible for mapping data received from Characteristic to your custom objects.

- `CustomCharacteristicData` is just like `Post`. In the case of JSON API it is a domain model object that is mapped from raw data received from BLE to conveniently work with that piece of data throughout your application.

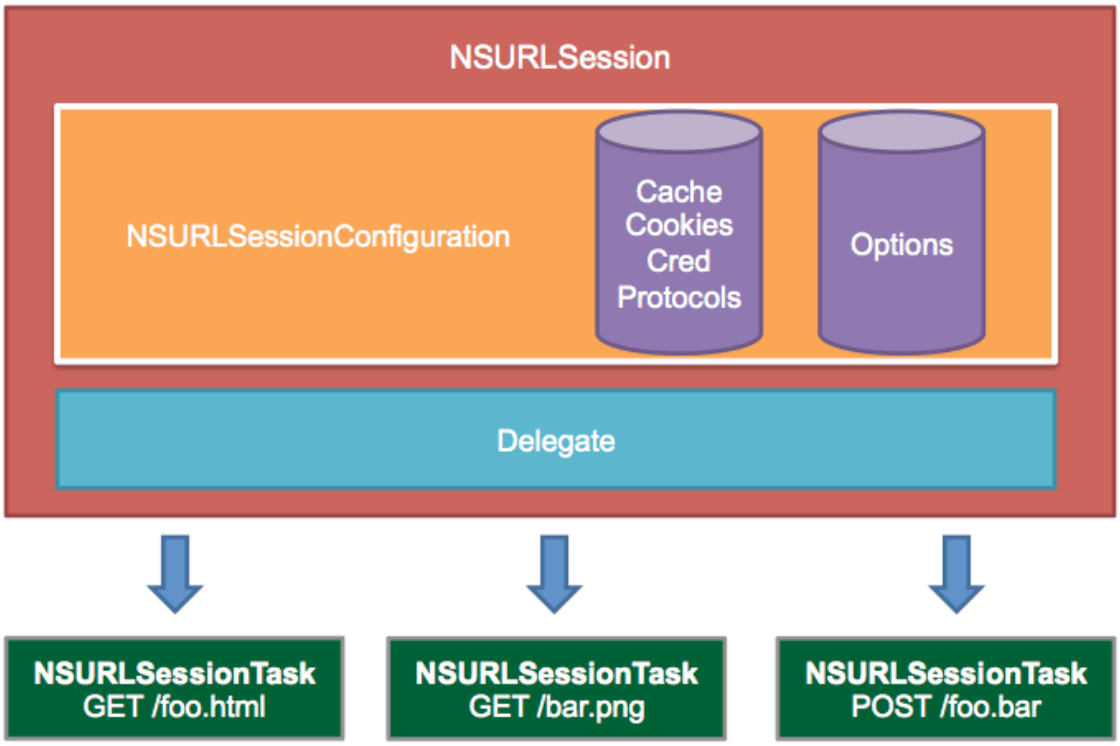
As you can see, both the HTTP and BLE examples are similar in what they do. Both of those examples wrap some kind of external service (HTTP or BLE respectively) and make it convenient and easy to work with those external services. At the end of the day, your application is going to interact only with `PostsService` and `Post` objects to do its 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, are all hidden behind those class interfaces. This design makes the code robust and reliable and separates low-level, unimportant logic from the business logic of your app.

Red flag: Simply saying the service layer has only an HTTP client and you create HTTP requests "when needed" for each endpoint isn't a red flag per se, but you should show a 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 to FTP, and to download files. You can optionally configure cache and execute your requests in a 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 a 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 the de facto 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 the de facto 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 the de facto standard of how you do HTTP networking on iOS and probably the most commonly used third-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, and more.

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 set up 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 `NSURL-Session`.

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 for a 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 the iOS system frameworks will complain and print logs or crash if you use them outside of 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` and `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 thread.

`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` makes 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 are common tasks 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, 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>(Awesomeness, Coolness, Greatness!),
"body": loads of text .....,
"title": This is an awesome post!]
```

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

Data serialization is only the first piece of the puzzle when working with JSON data. The other piece is data mapping. Even though we now 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 yourself manually, 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. 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 a 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 the differences are between them, and why it is important to have 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.

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. For 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 it 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 fetched from the web on iOS: downloading and caching. Downloading involves actually issuing an HTTP request to get the raw image data from a server. Caching is concerned with storing downloaded images to disk, database, or in-memory cache (or any combination of the three).

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 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 when doing it.

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

`In-memory cache` could be as simple as a `Dictionary` that keeps a reference to `UIImage` objects and uses the URLs they were downloaded from as unique keys 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 a similar database. Typically this is used for images of a very small size because databases were not necessarily made to handle large files. The best use case for that is small thumbnail images.

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

Ultimately the best solution for caching is going to vary case by case, but a lot of apps either use `SDWebImage` or a similar library or roll their own solution using 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 a PDF or image or video file that you need your app to download. And as usual, this question gives you an opportunity to either go over the basics or to dig deeper and explain the different techniques there are for downloading files.

Expected answer: At the very basic level, file download is just fetching a bunch of bytes 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 three 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 the received file in a temporary folder for later use, or 3. store the 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 be too expensive to handle in the callback block.

2. Storing the 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 in a temp folder allows you postpone handling the file until a later time and lets you move on in your download callback block. Files are kept in the temp folder only for the duration of your app running, though.

3. Storing the 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 typical for most of teams and companies, but those that work with messaging/chat applications are most likely going to ask it. Pub/sub systems are growing in popularity for solving problems other than chat/messaging on iOS though, 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 view controller with a specific chat then subscribes to a channel using sockets or Pubnub or another pubsub system. As soon as it's subscribed to its chat channel, it will receive the latest batch of messages since the 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/message applications work.

5.14 What is RestKit? What is it used for? What are the advantages and disadvantages?

RestKit used to be a very popular data synchronization framework 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 the 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. request sending and enqueueing,
- JSON request and response serialization,
- JSON response parsing and mapping,
- Core Data synchronization with mapping 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 is virtually not used anymore on new projects is because it was doing too much for you and forced you into its convoluted API. RestKit is so big that entire book can be 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 that it broke SRP. 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 a general understanding of what it offers and does for you so that you can make a conscious decision to pick it or avoid it.

5.15 What could you use instead of RestKit?

Since RestKit is practically obsolete these days, you could be asked about alternatives you could 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, and promises API out of the box. It takes on a lot of responsibilities just like RestKit and therefore is 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 a library like `Mantle`.

- Core Data synchronization and mapping from/to custom domain models can be taken care of by `Mantle`.

And that's everything you need. Rolling your own solution, and only when you need to, 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 are becoming more and more popular as tools for testing evolve in the iOS ecosystem. This question carries with it a lot of baggage and unfortunately is still somewhat controversial in the iOS community.

Expected answer: In general client-side applications do not integrate test network requests, they only do unit-testing or mock them if really necessary. The reason 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 `OCMock`-based libraries like Cedar, Quick, Specta, and Expecta are the go-to tools for unit-testing on iOS.

5.17 Conclusion

Service- and networking-related questions are 100 percent guaranteed 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 the 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 you really need to accomplish your task.

Chapter 6

Step Five: Learn How to Store Data

This chapter covers storage layer questions and answers and is good for quick interview prep for those questions. But there's more to it and the coverage in this chapter is continued in a bonus chapter, Chapter 9 Storage Evolution (AKA You Don't Always Need Core Data!). where we walk through building a small storage layer. We start with an in-memory array and then evolve and refactor it to use `NSUserDefaults`, File/Disk storage, and eventually `Core Data`, looking at advantages and disadvantages of each along the way.

The storage layer is present in every application because all of them need to have a state in one form or another. Often apps need to persist that state as well. This is why it is important for iOS devs to know storage options. In this chapter we'll cover questions about arrays and dictionaries, NSUserDefaults, file disk storage, Keychain, database solutions such Core Data, and more.

Interview questions covered in this chapter:

- What is the storage layer for in iOS applications?
- What can you use to store data on iOS?

- What is NSCoding?
- What is NSUserDefaults?
- What is Keychain and when do you need it?
- How do you save data to a disk on iOS?
- What database options are there for iOS applications?
- How is data mapping important when you store data?
- How would you approach major database/storage migration in your application?

6.1 What is the storage layer for in iOS applications?

This is a broad, open-ended question that could be asked in many forms. Effectively your interviewer is trying to gauge how you work with data, state, and persistence in iOS applications.

Expected Answer: The storage layer is responsible for storing data and keeping track of state. Objects and classes in this layer perform storing, saving, persisting, and mapping/serialization operations on data that other layers of the apps, such as the service layer, provide. At the end of the day you'll have things as simple as in-memory arrays and dictionaries and as complex as your own custom model objects and Core Data and Realm databases in this layer. The main point is that this layer decouples everything related to data storage and persistence from other classes and layers of your application.

Here's a typical set of classes that you'd have in your storage layer:

- a wrapper around Keychain

- a wrapper around `NSUserDefaults`
- a wrapper around file manager
- a wrapper around AVFoundation to store or retrieve audio and video files to/from disk
- `Repository` object that performs actually read and write to disk or database
- `NSManagedObject`s and its subclasses used to persist your domain model objects to Core Data (if you use Core Data)
- `Post` custom domain model class that represents instances of each post in your application
- `PostsStorage` object that initiates storing/fetching of `Post` models using a `repository` object and mapping its `NSManagedObject` results to `Post` model objects

Red Flag: This is not really a red flag, but quite often developers think that only Core Data or Realm belong in the storage layer. In fact, you don't have to use either of those technologies, and sometimes a simple in-memory array of objects will suffice. Use only what you need in your specific context.

6.2 What can you use to store data on iOS?

Interviewers ask this question to grasp your understanding of what tools and ways you have available to store and persist data on iOS.

Expected Answer: Generally there are the following ways to store data in order from simple to complex:

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

- NSUserDefaults/Keychain
- File/Disk storage
- Core Data, Realm
- SQLite

`In-memory arrays, dictionaries, sets, and other data structures` are perfectly fine for storing data. They are fast and simple to use. The main disadvantage though is that they can't be persisted without some work and can't be used to store large amounts of data.

`NSUserDefaults/Keychain` are simple key-value stores. One is insecure and another one is secure respectively. Advantages are that they are easy to use, relatively fast, and are actually able to persist things to disk. Disadvantages are that they were not made as a replacement for databases and can't handle large amounts of data or extensive querying.

`File/Disk storage` is actually a way of writing pieces of data (serialized or not) to/from a disk using **`NSFileManager`**. The great thing about it is that it can handle big files / large amounts of data but the disadvantage is that it was not made for querying.

`Core Data` or **`Realm`** are frameworks that simplify work with databases. They are great for large amounts of data and perfect for querying and filtering. Disadvantages are the setup overhead and learning curve.

Red Flag: You should be aware of different ways you could store data on iOS and their advantages or disadvantages. Don't limit yourself to only one solution that you're used to (like Core Data, for example). Know when one is preferable over the other.

6.3 What is NSCoding?

NSCoding is a widely used protocol for data serialization necessary for some of the data-storing operations using `NSUserDefaults`, `NSFileManager`, and `Keychain`. Interviewers will most likely ask this as part of a discussion about storage options on iOS, `NSUserDefaults`, `Keychain`, and so on.

Expected Answer: `NSCoding` is a Cocoa protocol that allows objects that adopt it to be serialized for `NSUserDefaults`, `NSFileManager`, or `Keychain` storage. The way it works is you implement the `init?(coder decoder: NSCoder)` and `encodeWithCoder` methods in the objects that comply to that protocol. Those methods decode and encode the object respectively for persistence and retrieval.

The gotcha with implementing `NSCoding` is that every property that you encode and decode needs to comply to the `NSCoding` protocol as well. All the "primitive" values such as `String`, `Number`, and `Int` already do that and every custom object that you're trying to serialize as one of the properties needs to comply to that protocol as well.

Red Flag: `NSCoding` is one of the fundamental protocols to use "lightweight" persistence implementation in iOS applications. Every iOS dev should be familiar with it.

6.4 What is NSUserDefaults?

`NSUserDefaults` is one of the common tools used in virtually every application for lightweight storage. Every iOS developer should be familiar with it.

Expected Answer: `NSUserDefaults` is a key-value storage that can persist serialized `NSCoding` compliant objects and primitives. Unlike Keychain, it is not secure and does not persist between app uninstalls. It's main purpose is to store small objects that are easily retrievable but also not important to lose.

A typical use case for it is some locally stored user preferences and/or flags. Do not use it as a database replacement because it was not built for extensive querying or for handling large amounts of data.

Red Flag: Using user defaults for data that needs to be secure is a red flag. For example, you would never want to store a user's password or access token in user defaults; use Keychain for that instead.

6.5 What is Keychain and when do you need it?

Storing data securely is important for every iOS app, big or small. This question is assessing your experience with iOS secure key-value storage.

Expected Answer: Keychain is a secure alternative to `NSUserDefaults`. It is a key-value store that is encrypted by the system and persists between app reinstalls unlike other types of data such as `NSUserDefaults`, files on disk, and Core Data databases. The advantage of `Keychain` is that it is secure, but the disadvantage is that its API is difficult to use.

The main use case for Keychain is to store small objects and primitives, such as tokens and passwords, securely. Use it instead of `NSUserDefaults` for that purpose, and just like with `NSUserDefaults` do not use it to store large amounts of data, such as databases, images, and videos.

Red Flag: You should be familiar with Keychain and what it's used for. The main red flag would be to either say that you use it instead of `NSUserDefaults` or vice versa. Both have their own purpose.

6.6 How do you save data to a disk on iOS?

Storing files on a disk is a more or less common thing to do in iOS applications. Don't expect this question that often, but note that it could come up from time to time in the context of storage.

Expected Answer: File storage is used to persist large amounts of data on a disk such as images, videos, and other kinds of files. `NSFileManager` is the class you would use to manipulate you app's folder on a disk. It is capable of creating subdirectories and storing files. You can store or read any `NSData` object whether it's an image, video, or an object serialized through the `NSCoding` protocol.

Red Flag: There isn't a specific red flag for this question. File disk storage is often not used directly because these days there are a lot of libraries and pods that take care of that low-level detail for you. But as a good iOS dev you need to be familiar with `NSFileManager` and how you can use it to persist stuff on disk.

6.7 What database options are there for iOS applications?

Interviewers ask this question to gauge your experience with database solutions on iOS.

Expected Answer: The go-to database solution on iOS is Core Data. There is also an option to use SQLite directly but tools are not that advanced for that, so you'll have to come up with some customizations of your own.

Another popular database framework is Realm. Each one of them has their own advantages and disadvantages.

Core Data is an object graph and persistence framework that is the go-to solution for local database storage on iOS. Advantages of that framework are that it is widely used and is supported by Apple. You can use it almost out of the box in your project, and it does a decent job of persisting data and making querying more or less straightforward.

A disadvantage is that the Core Data API is not that easy to use in some scenarios and specifically in a multi-threading environment. Another big disadvantage of Core Data is that there's a learning curve to it since it is not a straight-

forward addition on top of a relational database where each object represents a row in a table (like in `ActiveRecord`, for example), but rather an object graph storage.

Realm is an alternative to the Core Data database solution. It was built from the ground up to be easier to use and faster than Core Data or SQL. Advantages of Realm are that it's fast, has reactive features, is easier to use, is secure, and has entire cloud platform for syncing and other more advanced features. A disadvantage is that it is still in development - although the Realm team made a lot of progress recently - and as of the time of this writing, it doesn't have all the features on par with Core Data's `NSFetchedResultsController`. There are also issues with the size of realm databases. Due to their playback feature, it has to store way more data to replay the events that happened as compared to Core Data or SQL, which stores only the latest snapshot without keeping a history of all the changes.

Realm has a lot of potential to become the most popular solution for database storage on iOS in the long run, especially with all the backend/syncing functionality they are building into it.

SQLite is a relational database that powers Core Data under the hood. It can be accessed directly on iOS and used without Core Data, but it will require implementing custom tooling for accessing, reading, and writing to it. The main advantages of using SQLite directly are that it is going to be fast, and if you have SQL skills you can leverage them. The main disadvantage though is that you'll have to do all the heavy lifting of setting things up and accessing and migrating the database yourself; there are no good tools out there to help with that.

Red Flag: These days saying that there's only Core Data on iOS for databases would raise a red flag because the expectation is that developers are constantly looking for better solutions and are aware of other alternatives such as Realm or SQLite.

6.8 How is data mapping important when you store data?

Interviewers will most likely ask this question as part of a general discussion around the storage layer and the responsibilities it has. Just like with the service/networking layer, you need to understand the vital parts of it and what functions it performs, even if they are hidden by a library or a framework you use.

Expected Answer: One of the three main purposes of the storage layer, besides actually storing and persisting the data, is **data serialization**. Just like when you get data in the service layer in JSON or another format from external APIs and then serialize and map it into your custom domain model in the storage layer, you will need to serialize and map your data to and from your custom domain model objects to the format that your storage understands. The "*mapping*" chain for reading data looks like this: `db -> raw data format -> custom domain models`. And for writing like this: `custom domain models -> raw data format -> db`.

For example, that means that if you use Core Data, then serialization of your data that you'll make before saving it in Core Data will be mapping it to `NSManagedObject`s and then saving those to the Core Data database. And vice versa, when you need to retrieve data from Core Data, you'll create a predicate to query it and then you'll get back a bunch of `NSManagedObject`s and/or their subclasses as the result. You'll then need to map those objects into your own custom domain model objects to be able to easily work with them.

Specifically, in the case of `NSManagedObject`s, there are different approaches to working with data and quite often `NSManagedObject` subclasses are used directly as model objects throughout application. It is convenient after all to use them since mapping of values and properties is easily defined in the Core Data entity schema UI in Xcode. But there's a disadvantage to that approach that lies in coupling of responsibilities in `NSManagedObject` subclasses. If you use them throughout your application as domain models, then you couple yourself to Core Data directly and carry all the functionality of `NSManagedObject`

with them throughout the application. This issue is especially apparent when, inevitably, issues with multi-threading and concurrency arise. A cleaner way of doing it would be to use `NSManagedObject` and/or its subclasses only for data persistence and retrieval and use your own custom objects throughout the application as domain models.

Another example of similar mapping and serialization that you'll have to do would be `NSCoding` protocol serialization. If you use `NSUserDefaults` for storage of custom objects in your app then you'd need to do a similar serialization step. It is not that apparent because you typically implement the `NSCoding` protocol on your actual domain model objects, but effectively you go through the same process of retrieving data and then mapping it to your custom objects (or serializing your custom objects to raw data and then saving them). For reading, it would look like this: `NSUserDefaults -> NSData -> your custom object`. And for writing: `your custom object -> NSData > NSUserDefaults`. The reason it is not apparent is because the `NSData -> model` mapping step is "hidden" thanks to the `NSCoding` protocol, and you get your objects back without that intermediate step. But the same argument about coupling as with Core Data's `NSManagedObject` could apply, and it would be valid. But the difference between implementing `NSCoding` and coupling yourself to it and inheriting from `NSManagedObject` is that in one case it is a loose coupling to an interface and in the other case it is a tight coupling to an implementation. As you'll read in Chapter 8, according to SOLID principles, it's always better to couple yourself as loosely as possible.

Red Flag: The main red flag for this question is not being aware of serialization actually happening in the storage layer. Even if you don't implement it yourself explicitly, you should still be aware of its existence and of the cost you're paying for hiding that implementation and coupling yourself to a library or a framework.

6.9 How would you approach major database/storage migration in your application?

This question could be a part of an architectural discussion or come out of refactoring talks with your interviewer. Typically interviewers for bigger teams that are concerned with maintainability of the code ask this question.

Expected Answer: In practice, database or underlying storage migrations happen very rarely on iOS applications. Typically codebases end up getting stuck with whatever they picked as the initial storage/database solution (quite often Core Data). But there's a way you could organize your code using the Single Responsibility Principle where your codebase will be completely decoupled and agnostic of the persistence framework you use.

The main idea is to have a clear separation between your code that needs to access and use data from the database and the code that actually knows what database to use and how to access data in it. Typically that role is played by some kind of `storage` object that is the main object responsible for getting data in and out of database for the rest of the application. Internally that object would use one or more other objects that actually know how to work with an underlying database, let's say Core Data. And only those objects in the `storage` class actually refer to Core Data and know how to query it and how to write to it.

Since the rest of the application doesn't know anything about Core Data or whatever database solution you use, when the time comes, you could easily swap the underlying database for Realm, for example. You'd have to write some data migration code that will map and copy data from the existing Core Data to the new Realm. But the main approach will remain the same - the rest of your application continues to rely on the `storage` object to get the data and that object knows how to actually work with it.

This decoupling and Single Responsibility approach is described in detail in the bonus chapter, Storage Evolution. In that chapter we'll go through an example of the `PostsStorage` class that will start as simple in-memory storage, and

we will evolve and migrate it to use `NSUserDefaults`, then file/disk storage, and then eventually Core Data, keeping `PostsStorage`'s API consistent and unchanged throughout the whole process while the rest of the app will have no idea that we used various persistence solutions under the hood.

6.10 Conclusion:

The storage layer is one the building blocks of every iOS application. There are various approaches to storage and persistence when it comes to iOS apps, and a good developer knows what options are available and knows when to pick the right type of persistence solution.

This chapter gives you an overview of questions and answers around storage on iOS. If you want a practical example of all of the types of storages mentioned in this chapter, please look at the bonus chapter of this book, Chapter 9 Storage Evolution (AKA You Don't Always Need Core Data!).

Chapter 7

Step Six: Go crazy responsive with UI layouts

Quite often, creating a UI is one of the biggest parts of an iOS project. Being able to make it viewable on different-sized screens 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 Auto-Layout. The problem is it is notoriously difficult to work with and especially to debug.

But there's hope - there are libraries like Masonry that help you to 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 in working with UI on iOS?
- What do you use to lay out your views correctly on iOS?
- What are CGRect Frames? When and where would you use them?

- What is AutoLayout? When and where would you use it?
- What are compression resistance and content hugging priorities for?
- How does AutoLayout work with multi-threading?
- What are the advantages and disadvantages of creating AutoLayouts in code versus 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 alternative ways of working with UI do you know?
- How do you make a pixel-perfect UI according to a designer's specs?
- How do you unit and integration test UI?

7.1 What are the challenges in working with UI on iOS?

This question is typically asked to assess 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 the responsive and adjustable nature of iOS UI. There are several things you as a developer need to be concerned with when 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`

Mentioning all of the concerns above 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 toward a Frames versus 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`/`UICollectionVi`

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 lay out 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. When 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 auto-resizing 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 various devices.

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 and LayoutKit, that are more or less inspired by `React`. These alternatives are good in certain scenarios when, for example, you need to build highly dynamic and fast table views and collection views. AutoLayout is not always perfect for that and knowing 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 sane person 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 them?

This question is asked to learn if you have a background in building UI "the hard way" with using view position and size calculation. Before AutoLayout, Frames were used 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 UI rendering and how well you know a lower level of it.

Expected answer: The simplest way to position views on the screen is to give them 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 the upper-left corner where the view starts to draw itself and the 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 views 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 the different screen sizes 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 the de facto standard for doing layouts these days. Frames are considered to be an outdated concept that is very error prone. Say-

ing 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 a 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 for for some time and now it is the de facto standard. Your interviewer is either expecting a very brief answer to get an understanding of whether you're versed in the topic or is going to drill down and ask you for 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 a 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 specific `CGRect` frames to set for each view. It will adjust to different screen sizes or orientations 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 the width of an item or margin, etc.
- `priority` defines how high of a priority a given constraint should take. Constraints with a higher priority number take precedence over the others. `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 the de facto standard today for developing UI on iOS. Disregarding it or trying to prove that the Frames approach is better most likely 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 are compression resistance and content hugging priorities 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 while under the pressure of other constraints demanding its resizing. The higher compression resistance is, the less chance it's going to "budge" under the other constraint's pressure to compress it.

Hugging priority is the opposite of compression resistance. This constraint defines how likely it it the view will 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. Interviewers ask this question 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 constraint's addition or removal or constant change needs to be done on the main thread. After you change constraints, call the `setNeedsLayout` method.

Red flag: Saying 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 versus using storyboards?

Bigger teams sometimes ask this question 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 storyboards is considered to be more typical, and Apple pushes a lot of examples showing how to do that. The **advantages** are that it's visual, drag-and-drop/plug-and-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 constraints need to be removed at a glance. And quite often, teams working with one storyboard modify it in different `git` branches, causing merge conflicts.

Also, the **advantages** of defining AutoLayout in code are that it's very explicit, clear, and merge- and conflict-free. **Disadvantages**, on the other hand, are 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?

Bigger teams ask this question. They especially suffer from a poor team development support from Apple tools.

Expected answer: The main problem when working with storyboards in a big team is dealing with `.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 need to use frames will have to override the `layoutSubviews()` method where they can do the 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?

Interviewers ask this question 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 is used the most often. It can be triggered by running the `UIView.animateWithDuration()` set of methods. Things that are "animatable" this way are `frame`, `bounds`, `center`, `transform`, `alpha`, and `backgroundColor`.
- `Core Animation` is used for more advanced animation, things that `UIKit` isn't capable of doing. With Core Animation, you will manipulate

the view's `layer` directly and use classes like `CABasicAnimation` to set up 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 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 simply change frames in `UIView.animateWithDuration:animations:` and then assign new frames to your views and that's it - the animation will be performed. It's almost the same thing with AutoLayout, but instead of changing frames directly you change your constraints and their constants in the `animations:` block of the `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 extent of your answer will vary, and if interviewers 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 variable or set heights 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 the application demands. Two of the biggest constraints on mobile devices are memory and performance. This is why table views are designed to dequeue and reuse `UITableViewCells` they are displaying instead of creating new objects as user scrolls. It helps 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' heights (unless it's done automatically with `UITableViewAutomaticDimension`) and for the other cell and section life cycle callbacks like `tableView(UITableView, willDisplay: UITableViewCell, forRowAt: IndexPath)` and `tableView(UITableView, didSelectRowAt: IndexPath)`. 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 table views performance for smooth, fast scrolling?

One of the important questions 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 then 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 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 UIs 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 two or more items in a row or a grid where each item could be a different size. Each item in a `UICollectionView` is a subclass of `UICollectionViewCell`. `UICollectionView` mimics `UITableView` in its API and has similar `UICollectionViewDelegate` 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 common UI component used in iOS apps. Interviewers typically ask this question to gauge your level of experience working with either big, scrollable and zoomable content or gauge your level of 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 where all of the items cannot be displayed on the screen at the same time. `UIScrollView` is a superclass of `UITableView`, `UICollectionView`, and `UITextView`; therefore, all of them get the same features as `UIScrollView`.

When you work with `UIScrollView`, you define yourself as its delegate by adopting the `UIScrollViewDelegate` protocol. There are a lot of methods that you get with that delegate but the main one 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 sizes in a container into a column or a row. Interviewers ask this question to determine how up to date you are 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. Before iOS 9, you had to align your UI in a stack using constraints manually; it was very tedious and error prone, especially if you had to change the contents of your stack view at runtime. With `UIStackView`, it is as simple as a drag-and-drop in storyboards, and programmatically you add or remove views from the stack with just one command. `UIStackView` will take care of resizing for you.

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

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

This is an advanced question that interviewers ask to gauge how well informed you are of current trends in UI development.

Expected answer: Talk about React and React-like trends in UI development on the web and iOS. There's React Native, which is a great alternative for declarative UI development, but unfortunately, it 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 you make a pixel-perfect UI according to a designer's specs?

Teams that are very heavy on design and sleek UI typically ask this question.

Expected answer: The short answer is 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 laid out on various devices sizes and talked to you about cases where there's no content, and so on, then you could hypothetically build a "pixel-perfect UI." But, in reality, that's often not the case; you discover inconsistencies or edge cases in UI and UX as you build them. Designing 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 the UI/UX as you go.

Red flag: Don't say that you "just use a Photoshop or Sketch file and eyeball it."

7.19 How do you unit and integration test UI?

Interviewers typically ask this question in addition to 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 as well developed on iOS as it is for other platforms. The options you have today are libraries like Cedar that are built on top of Apple's `OCUnit`. But when using those you'll have to

do all the heavy lifting of setting it up, instantiating the UI, filling it with data, and so on.

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

Red flag: Saying that you don't test your UI is not a red flag per se 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 the time spent building iOS apps will be views-related work. For some apps it is crucial to build a sleek and nice UI; others can go by with just bare bones. As usual, things you should keep in mind are reusability and the 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, Architecture, FRP, and Dependencies Management.

Understanding design patterns and architecture is what distinguishes great developers from just good ones. They are the hardest concepts to grasp, but they give you the best return if you take time to study and practice them.

Design patterns give you a common language to use to talk about concepts in your code with other developers. They improve the readability and testability of your code.

Architecture helps you build maintainable software that is easy to change because the only thing constant in software development is that software is going to change.

In this chapter we'll cover questions about general programming, and iOS specifically, architecture and design patterns. The topics will vary from good old MVC, Delegate, Singleton, and so on, to SOLID and FRP.

Interview questions covered in this chapter:

- What design patterns are commonly used in iOS apps?
- What is MVC?
- What is MVVM?
- What are the common layers of responsibility that an iOS application has?
- What are the SOLID principles? Can you give an example of each in iOS/Swift?
- How do you manage dependencies in iOS applications?
- What is functional programming (FP) and functional reactive programming (FRP)?
- What are the design patterns besides common Cocoa patterns that you know of?

8.1 What design patterns are commonly used in iOS apps?

This question is a common one on interviews for positions of all levels, maybe with the exception of junior positions. Essentially the idea is that in working with the iOS platform, you as a developer should be familiar with commonly used techniques, architecture, and design patterns used on iOS.

Expected Answer: Typical commonly used patterns when building iOS applications are those that Apple advocates for in their Cocoa, Cocoa Touch, Objective-C, and Swift documentation. These are the patterns that every iOS developer learns. They include **MVC**, **Singleton**, **Delegate**, and **Observer**.

8.1.1 MVC

The good old Model-View-Controller is Apple's go-to application architecture design pattern. It's good for small/simple apps, but not sustainable in the long run. We'll cover it in more details in the following section.

8.1.2 Singleton

This is a common OOP design pattern where you create the one and only instance of a class that will be used everywhere in the application where an instance of that class is necessary. This is a useful design pattern but commonly overused to the point of becoming an anti-pattern. The main issue is that developers often use singletons to store a global state which is never a good idea due to race conditions and other types of data overrides that inevitably happen.

8.1.3 Delegate

Delegate is one of the core Cocoa design patterns. It is a variation of the Observer pattern where only one object can observe or be delegated to events from another object. It's a one-to-one relationship that is implemented through protocols. Cocoa itself uses this pattern a lot with `UITableViewDelegate`, `UITableViewDataSource`, `UIPickerViewDelegate`, and similar protocols that are exposed by the framework for developers to use. (We also discussed Delegate in the fundamentals chapter.)

8.1.4 Observer

This pattern is a common one in iOS. It's a design pattern that helps objects observe state changes or events in other objects without coupling that observation to internal implementation of those objects. Developers can always implement the Observer pattern themselves, but there are two built-in implementations in

Cocoa already - Delegate, one-to-one observation, and KVO (key-value observing), one-to-many observation.

Red Flag: When interviewer asks this question (in one form or another) what they are looking for is something besides MVC. Because MVC is the go-to design pattern, the expectation is that every iOS developer knows what it is. What they want to hear from you though, is what else is commonly used and available out of the box.

8.2 What is MVC?

MVC is Apple's design pattern of choice and a question about it is unavoidable at any iOS interview. When you are asked this question, it is a great opportunity to spin it into a deeper conversation about software architecture and design. If you're applying for a mid- or senior-level position, talk in length about MVC and its advantages, disadvantages, and alternatives.

Expected Answer: `MVC` stands for `Model-View-Controller` and is Apple's go-to design pattern for iOS applications. `Models` represent data, Views represent the UI, and Controller, the business logic. That more or less maps into

- `Models` being your NSObject subclasses or Core Data objects that represent your data;
- `Views` being your UIView subclasses and UIViewControllers that draw things on the screen;
- `Controller` being your application's logic and classes responsible for that.

Pay attention to where UIViewController is. It is in the `View` layer. The reason being that at the end of the day, it does (or should do) what it name entails - control the view, no more, no less. Too often though, and Apple's code examples are guilty of that too, developers put all the business logic into view

controllers and it quickly grows out of proportion and the whole MVC architecture becomes a `Massive-View-Controller` instead.

Advantages

Some architecture is better than no architecture and I can understand why Apple chose MVC as its main base design pattern of choice - it's simple to understand! Even novice developers can quickly wrap their heads around it and get going, cranking up a bunch of views, models, and view controllers. Where it falls short though is more complicated cases.

Disadvantages

The main disadvantage of MVC is its simplicity, which pretty quickly starts to limit you. As I've mentioned previously, a lot of developers tend to abuse view controllers and give them too much responsibility. The remedy for that is more explicit layer boundaries and the Single Responsibility Principle in general that we'll talk about later in this chapter.

MVC quickly falls short in edge cases. For example, where do you put a service object that does HTTP networking? It's certainly not a view. Is it a model? Nope. Is it a controller? Hmm... not really a controller either...

Alternatives

There are several things that can be done about MVC's shortcomings but the main two solutions are

- MVVM (Model-View-View-Model) design pattern
- SRP (Single Responsibility Principle)

MVVM helps with slimming down the notorious Massive-View-Controller by extracting the business logic and the data out from view controllers into view models (and ultimately other objects). SRP helps with setting firm boundaries for your code for object responsibilities. Each of them does only one thing and will change only for one reason. That greatly reduces the complexity of your code and makes it more composable, maintainable, and receptive to change.

We'll talk about both of those later in this chapter.

Red Flag: If you're applying for anything beyond a junior position, it is unacceptable to talk about MVC as a design pattern that is the best out there because Apple picked it. Apple is good at what they do - building hardware and frameworks. And their goal with their documentation and sample code is to get beginner developers up to speed as soon as possible. What they are not good at is building apps. They never had a case for a complex and big enough application, so following their advice with regard to architecture is wise only in the beginning and only to a degree. If you're working on a serious enough application with locally stored data and an HTTP connection, you have to use something better than MVC. As mentioned before, MVVM is a low-hanging fruit, but you could and should go farther and apply SRP and other design and architecture best practices to your code.

8.3 What is MVVM?

MVC is a fine pattern for the simplest apps. When you build something more complex you need a better architectural and design approach for your codebase than that. One of the alternatives that is embraced by the iOS community is MVVM. This question inevitably will arise through conversations about architecture and design. Answering it well will make you stand out from the crowd because, to my knowledge, not that many developers actually use this very useful design pattern.

Expected Answer: MVVM stands for Model-View-View-Model. This design pattern is effectively a subset and extension of MVC. With MVVM on iOS, in addition to models, views, and controllers, we'd also have view models that play an important role in data presentation and delegating business logic triggered by the view layer (views and view controllers). It fits nicely into existing MVC architectures and extends it by making it more testable and less coupled.

To illustrate where and how we use view models, consider this example:

```
import UIKit

class MyViewController: UIViewController {

    private var someButton: UIButton!

    private var buttonTitle: String!

    convenience init(buttonTitle: String) {
        self.init()

        self.buttonTitle = buttonTitle
    }

    override func viewDidLoad() {
        super.viewDidLoad()

        self.someButton = self.setupButton(title: self.buttonTitle)
    }

    private func setupButton(title: String) -> UIButton {
        let button = UIButton()
        button.setTitle(title, for: .normal)
        button.addTarget(self,
                         action: Selector(("buttonClick:")),
                         for: .touchUpInside)
        return button
    }

    public func buttonClick(sender: AnyObject) {
        // let's assume there's some business logic happening here
        print("there's an action here that
                  \ relies on the state of your application")
        print("such as button title - for example, \(self.buttonTitle!)")
    }
}

// creating a new instance of our VC
let myVC = MyViewController(buttonTitle: "some button title")
// simulating our VC's appearance on the screen
myVC.view

// simulating the user clicking our button
myVC.buttonClick(sender: NSObject())


// output:
// there's an action here that relies on the state of your application
// such as button title - for example, some button title
```

As you can see there's typical MVC stuff going on here. We have a view controller that upon initialization takes in some state to be displayed in its button title when the view is loaded. And it also prints that title when the user clicks on that button.

Seems ok, doesn't it? Well, there's a responsibility and coupling problem here that quite often leads to a "Massive View Controller" issue. The problem is that our view controller is a view layer object that is responsible for displaying a UI, but it currently also tries to get into the business of managing state and executing business logic. The way it does it is by keeping a reference to state, in our case the **`buttonTitle`** string that is passed to it upon initialization, and by having the business logic in the **`buttonClick`** method.

A way to improve is to introduce a **`viewmodel`** object for business logic and state and inject it as a dependency in our view controller:

```
import UIKit

class MyViewModel {

    let title: String

    init(title: String) {
        self.title = title
    }

    func printAction() {
        print("executing business logic")
        print("printing button title: \(self.title)")
    }
}

class MyViewController: UIViewController {

    private var viewModel: MyViewModel!

    private var someButton: UIButton!

    convenience init(viewModel: MyViewModel) {
        self.init()

        self.viewModel = viewModel
    }

    override func viewDidLoad() {
```

```
        super.viewDidLoad()

        self.someButton = self.setupButton(title: self.viewModel.title)
    }

    private func setupButton(title: String) -> UIButton {
        let button = UIButton()
        button.setTitle(title, for: .normal)
        button.addTarget(self,
                         action: Selector(("buttonClick:")),
                         for: .touchUpInside)
        return button
    }

    public func buttonClick(sender: AnyObject) {
        // trigger business logic here
        self.viewModel.printAction()
    }
}

// creating an instance of MyViewModel to keep track of state
// and to execute business logic
let myViewModel = MyViewModel(title: "some button title")

// creating a new instance of our VC and injecting our viewmodel
let myVC = MyViewController(viewModel: myViewModel)
// simulating our VC's appearance on the screen
myVC.view

// simulating the user clicking our button
myVC.buttonClick(sender: NSObject())


// output:
// executing business logic
// printing button title: some button title
```

So what we did here was extract all the state (**`title`** string) from our view controller since it's not the view controller's responsibility to keep track of the state. And we extracted the business logic (**`printAction`** that used to be in its **`buttonClick`** method) from it as well. The reason is the same, it's just a UI and it shouldn't know what our app does. There are many benefits of extracting that stuff into a view model object: state decoupling, responsibility decoupling, and better testability of your code. Now to test your business logic and state changes, you don't have to instantiate the view controller; all you have to do is to create a view model, send messages to it, and examine the changes. The

view controller simply becomes an input device, just like terminal or voice is.

You can also do the same thing with other UIView subclasses - not only view controllers and create view models for them when they grow out of proportion and carry too much state and logic.

This in a nutshell is what the MVVM pattern is and what it helps with in your code. For me personally it's been an invaluable tool for refactoring and code improvement that I've used when joining projects. Slimming down view controllers is typically the lowest-hanging fruit for improvement on iOS projects.

Red Flag: MVVM is not a silver bullet, but it becomes a more and more widespread tool for refactoring and decoupling on iOS projects. If you're aiming for something higher than a junior position, you need to at least be aware of it and have some experience using the pattern and recognizing when it could be of use to you.

8.4 What are the common layers of responsibility that an iOS application has?

The interviewer will ask this question in some form when your conversation gets deeper into architecture and the high-level concepts. You can steer the conversation to go in a way that aligns more or less with your experience if you want to.

Expected Answer: Every iOS application, no matter how big or small, has the following layers of responsibility: **UI Layer**, **Service Layer**, **Storage Layer**, and **Business Logic Layer**.

8.4.1 UI Layer

The UI Layer is responsible for displaying things on the screen. Every iOS application has this component since every iOS app has some kind of user

interface. This layer includes `UIWindows`, `UIViews`, AutoLayout, `UIViewControllers`, table views, collection views, `CALayers`, animations, touch events, app delegate, and other things with which the user interacts with your app.

The main purpose of this layer of responsibility is to display UI elements on the screen and to take user input in and delegate it to the rest of your application. The key here is not to put too much code that is responsible for storage or service or business logic in the UI layer because it could cause overblown view controllers and views issues, which is never good for any codebase.

8.4.2 Service Layer:

The service layer is responsible for all external communication your application has. HTTP API client objects and classes, Bluetooth Low Energy (BLE) code, analytics services, third-party (non-UI-related) services, location services, GPS/gyroscope, and the respective data mappings from JSON and other formats to your domain objects would all constitute this layer.

The key thing here, as with every other layer, is not to mix up responsibilities from other layers. For example, never put UI code inside of your service objects. Alert pop-ups and UI updates have no business being in HTTP networking code, it's the UI layer's responsibility to handle that. Same goes for storage - don't save things to disk or the database in your networking code. It doesn't make any sense.

8.4.3 Storage Layer:

The storage layer is responsible for storing things. That layer contains your custom domain model classes and complex things such as Core Data, Realm, SQL, and NSFileManager. And it also has simpler storage solutions such as NSUserDefaults, Keychain, and even in-memory arrays, sets, and dictionaries. The idea for that layer is to abstract out and decouple everything that has to do

with data management and persistence. Stuff in the storage layer is supposed to be the ultimate source of truth for your application. You should be able to rely on it and definitely say if you have something or not; other things and data from other layers are more temporary and ephemeral.

8.4.4 Business Logic Layer:

Into the business logic layer go objects that are responsible for the application's business logic - objects that use components and objects from other layers to achieve results and the work for the user. Coordinators that use 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 tells them what to do to achieve results. This layer is where the actual interesting stuff that your application does happens.

Virtually every iOS app has the previously described layers of responsibility. Even an app that does everything locally and never connects to an HTTP API would have to eventually track user behavior and would do so using a tracking/analytics service for that, which is the *service layer*. It definitely has some data to store, even in its in-memory arrays, so it has a *storage layer*. It has some views and view controllers to display, aka the *UI layer*. And of course, to be of any use to the user, it needs to coordinate all of those things, so here's your *business logic layer*.

Red Flag: Simply answering that every iOS app has a view, model, and controller as layers doesn't cut it. Every iOS app is way more than that, and MVC doesn't cover a lot of edge cases when a class doesn't strictly belong to either a model, view, or controller. This is why you need to look at your code's layers of responsibility more broadly.

8.5 What are SOLID principles? Can you give an example of each in iOS/Swift?

Interviewers could ask this question on senior or architect position interviews. SOLID principles are relatively old but incredibly useful concepts to apply to any OOP codebase in any language. Watch a few of Uncle Bob's talks on the topic to fully appreciate the history behind them.

On YouTube Bob Martin SOLID Principles of Object Oriented and Agile Design

Expected Answer: SOLID stands for Single Responsibility Principle, Open/Closed Principle, Liskov Substitution Principle, Interface Segregation Principle, and Dependency Inversion Principle. These principles feed into and support each other and are one of the best general design approaches you could take for your code. Let's go through each of them.

8.5.1 Single Responsibility Principle

The Single Responsibility Principle (SRP) is the most important one of them. It states that every module should have only one responsibility and reason to change. SRP starts with small concrete and specific cases such as a class and/or an object having only one purpose and being used only for one thing. The idea is that when, for example, you create a new *model* class called `Post`, its single purpose and responsibility is to hold the data and information about a post. It's a model class, it should do no more, no less. It should not be accessing the database to save itself. It should not be creating underlying comments or changing them in any way. It should not be parsing JSON to create a new post out it. All of those things are single responsibilities of other objects that should not be mixed into that `Post` class. The `Post` class has only one reason to change - it changes when we need to change the data structure of our posts in our application. It should not change because we decided to swap the underlying database to Realm from Core Data or because our backend decided to

return a different type of JSON.

This principle is the basis of the architecture and approach described in the previous answer that I use myself when building any kind of application. Consider this: all of those things around `Post` described previously are related to the same layer of responsibility of your application - the storage layer. The reason why they are grouped into that layer is because they are responsible for storing things and because the only reason for them to change is when you need to change how you store things, not when you need to change your networking code, for example.

8.5.2 Open/Closed Principle

The Open/Closed Principle (OCP) states that your modules should be open for extension but closed for modification. It's one of those things that sounds easy enough but is kind of hard to wrap your head around when you start to think about what it means. Effectively it means that when writing your code you should be able to extend the behavior of your objects through inheritance, polymorphism, and composition by implementing them using interfaces, abstractions, and dependency injection. If, let's say, you have a `PostsStorage` class that has a certain interface that allows you to store `Post` models in the database. According to that principle, when you want to extend and add behavior and features to your `PostsStorage`, you should be able to do that through inheritance and through injecting new dependencies into that storage. For example, if you want to change the database that the storage saves posts to from Core Data to Realm you have two options: either you subclass from it and override methods that call Core Data and use Realm there instead or you inject a different database adapter/accessor dependency that complies to the same protocol as the Core Data one but uses Realm under the hood instead. In both scenarios though, every object that was previously using `PostsStorage` should still be able to use it as before without any changes because in both scenarios, the `PostsStorage`'s interface that they relied on hasn't changed. We effectively extended `PostsStorage` behavior without modifying it. It nicely

aligns with SRP because `PostsStorage` hasn't had a reason to change when we swapped the underlying database to Realm; it was not `PostsStorage`'s responsibility to work with it in the first place.

8.5.3 Liskov Substitution Principle

The Liskov Substitution Principle (LSP) states that objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program. What that means is that when you inherit from a class or an abstract class or implement an interface (protocol), your objects should be replaceable and injectable wherever that interface or class that you subclassed from was used. This principle is often referred to as design by contract or, as of late in the Swift community, referred to as *protocol-oriented programming*. The main message of this principle is that you should not violate the contract that your interfaces that you subclass from promise to fulfill and that by subclassing, those subclasses could be used anywhere where the superclass was previously used. If we look at our `PostsStorage` as an example again, then according to Liskov's Substitution Principle we could say that if we subclass from it, let's call it `BetterPostsStorage`, then everywhere we were using the original `PostsStorage`, we could be using `BetterPostsStorage` instead and our app won't break or misbehave in any way.

8.5.4 Interface Segregation Principle

The Interface Segregation Principle (ISP) says many client-specific interfaces are better than one general-purpose interface. It also states that no client should be forced to depend on and implemented methods it does not use. What that means is that when you create interfaces (protocols) that your classes implement, you should strive for and depend on abstraction over specificity but not until it becomes a waste where you have to implement a bunch of methods your new class doesn't even use. For a lack of a better (shorter) example, let's pretend that we have the following classes and interfaces:

```
protocol WorkerInterface {
    func eat()

    func work()
}

class Worker: WorkerInterface {

    func eat() {
        print("worker's eating lunch")
    }

    func work() {
        print("worker's working")
    }
}

class Contractor: WorkerInterface {
    func eat() {
        print("contractor's eating lunch")
    }

    func work() {
        print("contractor's working")
    }
}


class Manager {

    private let workers: [WorkerInterface]

    init(workers: [WorkerInterface]) {
        self.workers = workers
    }

    func manage() {
        workers.forEach { (worker: WorkerInterface) in
            worker.work()
        }
    }
}

let worker1 = Worker()
let worker2 = Worker()
let contractor = Contractor()

let manager = Manager(workers: [worker1, worker2, contractor])

manager.manage()
```

Here we have a **`WorkerInterface`** that has two methods **`eat()`** and **`work()`**. And we have two classes that implement it: **`Worker`** and **`Contractor`**. And we have a **`Manager`** that relies on **`WorkerInterface`** to call **`work()`** on each one of the passed worker and contractor objects to initiate the work. It's all nice and good and we are assuming here that all workers and contractors are humans who can work but also need to eat so implementing the **`eat()`** method in both of them is perfectly reasonable (we are assuming that the **`eat()`** method is called on those objects somewhere else in the application).

But this quickly becomes unreasonable when we introduce a **`Robot`** class that complies to the same **`WorkerInterface`**:

```
protocol WorkerInterface {
    func eat()

    func work()
}

class Worker: WorkerInterface {

    func eat() {
        print("worker's eating lunch")
    }

    func work() {
        print("worker's working")
    }
}

class Contractor: WorkerInterface {

    func eat() {
        print("contractor's eating lunch")
    }

    func work() {
        print("contractor's working")
    }
}

class Robot: WorkerInterface {

    // do nothing here. cuz robots don't eat.
    func eat() {}

    func work() {
```

```
        print("robot's working")
    }
}


class Manager {

    private let workers: [WorkerInterface]

    init(workers: [WorkerInterface]) {
        self.workers = workers
    }

    func manage() {
        workers.forEach { (worker: WorkerInterface) in
            worker.work()
        }
    }
}

let worker1 = Worker()
let worker2 = Worker()
let contractor = Contractor()
let robot = Robot()

let manager = Manager(workers: [worker1, worker2, contractor, robot])

manager.manage()
```

This violates ISP because our `Robot`s don't need to eat and `Robot` is forced to implement an interface it doesn't fully need, hence an empty `eat()` method.

What we need instead is to extract better, more concrete interfaces and use them instead:

```
protocol WorkableInterface {
    func work()
}

protocol FeedableInterface {
    func eat()
}

class Worker: WorkableInterface, FeedableInterface {

    func eat() {
        print("worker's eating lunch")
```

```
    }

    func work() {
        print("worker's working")
    }
}

class Contractor: WorkableInterface, FeedableInterface {

    func eat() {
        print("contractor's eating lunch")
    }

    func work() {
        print("contractor's working")
    }
}

class Robot: WorkableInterface {

    func work() {
        print("robot's working")
    }
}


class Manager {

    private let workers: [WorkableInterface]

    init(workers: [WorkableInterface]) {
        self.workers = workers
    }

    func manage() {
        workers.forEach { (worker: WorkableInterface) in
            worker.work()
        }
    }
}

let worker1 = Worker()
let worker2 = Worker()
let contractor = Contractor()
let robot = Robot()

let manager = Manager(workers: [worker1, worker2, contractor, robot])

manager.manage()
```

Now we rely on a more specific `WorkableInterface` that `Manager` uses, and `Robot` doesn't have to implement what it doesn't need. This in a nutshell is what the Interface Segregation Principle is all about. You have to either do a little bit more design up front to get your interfaces/protocols right or you resolve it with adapters in existing systems. But ISP helps you maintain Liskov's Substitution Principle in your code as well.

This principle ties back into the current trend of *protocol-oriented programming*.

8.5.5 Dependency Inversion Principle

The Dependency Inversion Principle (DIP) states, depend on abstractions, not concretions. The best example that showcases this principle is the **Dependency Injection (DI)** technique. With the Dependency Injection technique, when you create an object, you supply and *inject* all of its dependencies upon it's initialization or configuration rather than let the object create or fetch/find its dependencies for itself. Let's look at the following example where DI is not applied:

```
class MyAPIClient {

    func httpGet(url: String, success: () -> Void, failure: () -> Void) {
        // let's assume we do some http networking stuff here

        // and let's pretend it succeeds

        success()
    }
}

class PostsService {

    lazy var apiClient: MyAPIClient = { [unowned self] in
        return MyAPIClient()
    }()

    func fetchPostsFromServer(success: () -> Void, failure: () -> Void) {
        // more business logic to prepare url and params here
```

```
        let postsUrl = "some_endpoint_url"

        apiClient.httpGet(url: postsUrl, success: {
            success()
        }, failure: {
            failure()
        })
    }
}


let postsService = PostsService()
postsService.fetchPostsFromServer(success: {
    print("change some UI upon successful fetch of posts here")
}, failure: {
    print("show some alert with an error")
})
```

Here when posts service creates its own instance of api client internally when it needs it and then stores it in a property for future use. Overall it's not that bad because here we at least separate some responsibilities and delegate low level networking code implementation to api client instead of keeping it posts service. But the problem with this is that **`PostsService`** is tightly coupled to **`MyAPIClient`**; whenever **`MyAPIClient`**'s public interface changes, **`PostsService`** will have to change as well. This violates the Dependency Inversion Principle because our high-level module (i.e., **`PostsService`**) depends on the low-level module **`MyAPIClient`**. Instead, both should depend on abstractions. The Dependency Injection technique can help us with that.

With the Dependency Injection technique, instead of creating an instance of **`MyAPIClient`** internally in **`PostsService`**, we would inject in posts a service object upon its initialization. Also we'd refactor the **`PostsService`** class in a way that it does not depend on a specific and concrete **`MyAPIClient`** class but instead relies on an interface abstraction:

```
protocol AbstractAPIClient {
    func httpGet(url: String, success: () -> Void, failure: () -> Void)
}

class MyAPIClient: AbstractAPIClient {
```

```
    func httpGet(url: String, success: () -> Void, failure: () -> Void) {
        // let's assume we do some http networking stuff here

        // and let's pretend it succeeds

        success()
    }
}

class PostsService {

    let apiClient: AbstractAPIClient

    init(apiClient: AbstractAPIClient) {
        self.apiClient = apiClient
    }

    func fetchPostsFromServer(success: () -> Void, failure: () -> Void) {
        // more business logic to prepare url and params here

        let postsUrl = "some_endpoint_url"

        apiClient.httpGet(url: postsUrl, success: {
            success()
        }, failure: {
            failure()
        })
    }
}

let myApiClient = MyAPIClient()

let postsService = PostsService(apiClient: myApiClient)

postsService.fetchPostsFromServer(success: {
    print("change some UI upon successful fetch of posts here")
}, failure: {
    print("show some alert with an error")
})
```

As you can see **PostsService** now depends on an abstraction, **AbstractAPIClient**, and gets its **apiClient** object injected upon initialization. The great thing about that is that now we can easily create a new type of API client (let's say one with better password security or something) and we'd simply inject instances of that into **PostsService** as long as that new API client conforms to **AbstractAPIClient** that **PostsService** depends on. Posts service doesn't have to change when that happens.

By applying the Dependency Injection technique we've not only complied with the Dependency Inversion Principle (DIP) but we also achieved better decoupling between our objects, complied with the Single Responsibility Principle (SRP), achieved the Liskov's Substitution Principle (LSP), and made our `PostsService` open for extension / closed for modification (OCP).

SOLID principles are the bedrock of good OOP design. Applying these principles will help you build better, more maintainable software.

8.6 How do you manage dependencies in iOS applications?

This is not necessarily an architecture or design patterns question but it is nevertheless related and important. By dependencies they mean the code you don't write yourself but use to build your application, that is, third-party libraries and frameworks. Your interviewer will gauge your level of experience setting projects up and managing dependencies on big and small projects by asking this question.

Expected Answer: Dependencies management is something you quite often don't think about right from the beginning of the project, usually not until you get to the point when you need to use a third-party library or a framework. Then a question presents itself - how do you do that? A naive approach would be to copy third-party code that your app depends on and just drag-and-drop or copy it into your project. The problem with that solution is that third-party libraries themselves have their own dependencies and sometimes dependencies from different libraries conflict with each other. For example a common scenario is the following: you import library `A version 1.1.0` and it depends on library `B versions from 2.1.0 to 2.2.0`. To satisfy the requirements of library `A` you add library `B version 2.2.0` (the latest) to your project. Then later you add library `C version 0.5.0` to your project. But library `C` depends on library `B version 2.1.0`. So now you have to resolve the version conflict between library `A` and `C` depending on `B`. The resolution in

this case is to install library `B version 2.1.0` because it will satisfy both `A` and `C`. But the problem is that you have already added library `B version 2.2.0` to your project. Now you'll have to remove it and remove all the code that uses APIs that are only in `2.2.0` and not in `2.1.0`. And then you'll have to rewrite your code so that it uses APIs of library `B version 2.1.0`. This is a lot of manual work that you as a developer should not be handling because this example is simplified and the real-life issues related to dependency management version conflicts are way more severe.

The solution to those types of issues is a dependency management tool, and the most popular one in the iOS community is Cocoapods. Cocoapods is a Ruby gem that helps you manage all the dependency's complexity, resolves version numbers of libraries (and their dependencies) that your project depends on, and just in general makes your life easier when setting up iOS projects. The way it works - it first figures out what libraries (called pods) our project depends on by reading a list in your project's `Podfile`. Then it will download those libraries from their respective github repositories and put them together in an Xcode project called `Pods`. After that it will create a new Xcode workspace for your project and that new `Pods` project and will put them both in. It will also set all the project settings and workspace settings up the way that your code is fully ready to import the libraries and start using them.

If you need help setting up your project, have a look at this video where I walk you through a typical Cocoapods setup: *iOS Project Setup with Cocoapods*

There's an alternative to Cocoapods called Carthage. Carthage uses a different approach to dependency management where it creates framework binaries for your dependencies but leaves it up to you to integrate them into your project. It's an alternative that's more flexible than Cocoapods but is harder to use.

Another error prone way of handling dependencies is to use git submodules but you'll have to do all the configurations and imports yourself.

Red Flag: The biggest red flag would be to say that you're copying or dragging and dropping external libraries/code manually into your codebase. This is an unmaintainable solution that will not work in the long and short run.

8.7 What is Functional Programming and Functional Reactive Programming?

Functional programming (FP) is the new hotness in iOS/Swift, JavaScript, and other dev communities. Except that it's actually not that new. Expect this question either in regards to Swift features or as a bigger architectural and conceptual discussion question.

Expected Answer: Functional programming (FP) is a style of programming that puts emphasis on functions as the main computational unit and treats them like first-class citizens in your code. Those functions are akin to mathematical functions. The functional programming paradigm avoids mutability and state change either completely or as much as possible. FP is a declarative style of programming where you would declare what your code should do instead of telling it explicitly (i.e., imperatively) how to do it (what steps to take, etc.) like in imperative programming. In contrast, imperative programming is a set of steps to execute that usually heavily relies on state and mutability to do so.

Swift introduces more FP concepts built into the language than Objective-C: value types, functions as first-class citizens, higher-order functions, and so on. Those concepts make Swift functional friendly but not a fully functional language.

Functional Reactive Programming (FRP) is a declarative programming paradigm that combines in itself *functional programming* and *reactive (async dataflow programming)* paradigms. It is also a declarative style of programming where you would declare what your code does rather then explicitly state how it does it. The reactive component of FRP allows us to introduce and describe the concept of time, which is hard to work with in pure functional programming. FRP helps us deal with user input and the asynchronous nature of iOS applications in general (user input happens at some point in time, networking will finish some time in the future, etc.).

FP and FRP rely heavily on higher-order functions such as *map*, *reduce*, and *filter* that take functions as arguments and return other functions which makes

them highly composable.

Swift doesn't have a native support for FRP but there are two excellent libraries out there that implement functional reactive programming concepts and make them easily available to us. Those libraries are ReactiveCocoa and RxSwift.

ReactiveCocoa offers composable, declarative, and flexible primitives that are built around the grand concept of streams of values over time. These primitives can be used to uniformly represent common Cocoa and generic programming patterns that are fundamentally an act of observation. ReactiveCocoa is a great way of getting FRP in your codebase, and Ash Furrow has written an entire book on the subject of FRP where he uses ReactiveCocoa. The book's called Functional Reactive Programming on iOS.

RxSwift is an implementation of Reactive Extensions (Rx) in Swift. Reactive Extensions is a library for composing asynchronous and event-based programs using observable sequences. RxSwift is a great (and in my opinion the best) implementation of functional reactive programming concepts in Swift and many other languages. The advantage of learning that library is that knowledge is transferable and can be applied to any other platform/language where Reactive Extensions is available (currently they have implementations in Java, JavaScript, C#, C#(Unity), Scala, Clojure, C++, Lua, Ruby, Python, Groovy, JRuby, Kotlin, Swift, PHP, Elixir, etc.).

Red Flag: These days the expectation is that developers at least understand some basic concepts of FP. You don't have to know what FRP is but you need to be able to explain how FP differs from the typical imperative style of programming.

8.8 What are the design patterns besides common Cocoa patterns that you know of?

This is an advanced question that an interviewer will ask when you interview for a senior or architect position. Be ready to recall a bunch of Gang of Four

patterns and similar. This could be a followup to "What design patterns are commonly used in iOS apps?" question.

Expected Answer: Besides commonly used *MVC*, *Singleton*, *Delegate*, and *Observer* patterns there are many other that are perfectly applicable in iOS applications: **Factory Method**, **Adapter**, **Decorator**, **Command**, and **Template**.

8.8.1 Factory Method

Factory Method is used to replace class constructors, abstract and hide objects initialization so that the type can be determined at runtime, and to hide and contain **`switch/if`** statements that determine the type of object to be instantiated.

Let's expand our previous Workers/Contractors/Robots example to demonstrate it:

```
protocol WorkableInterface {
    func work()
}

class Worker: WorkableInterface {

    func work() {
        print("worker's working")
    }
}

class Contractor: WorkableInterface {

    func work() {
        print("contractor's working")
    }
}

class Robot: WorkableInterface {

    func work() {
        print("robot's working")
    }
}

enum WorkerType {
    case Worker, Contractor, Robot
```

```
}

class TrainingAndPreparationCenter {
    func workerableUnit(_ workerType: WorkerType) -> WorkableInterface {
        switch workerType {
        case .Contractor:
            return Contractor()
        case .Robot:
            return Robot()
        default:
            return Worker()
        }
    }
}

class Manager {

    private let workers: [WorkableInterface]

    init(workers: [WorkableInterface]) {
        self.workers = workers
    }

    func manage() {
        workers.forEach { (worker: WorkableInterface) in
            worker.work()
        }
    }
}

let trainingAndPreparationCenter = TrainingAndPreparationCenter()

let worker1 = trainingAndPreparationCenter.workerableUnit(.Worker)
let worker2 = trainingAndPreparationCenter.workerableUnit(.Worker)
let contractor = trainingAndPreparationCenter.workerableUnit(.Contractor)
let robot = trainingAndPreparationCenter.workerableUnit(.Robot)

let manager = Manager(workers: [worker1, worker2, contractor, robot])
manager.manage()
```

As you can see we now have a **`TrainingAndPreparationCenter`** object that creates instances of **`WorkableInterface`** and the concrete instance type instantiated is determined at runtime. The idea is that the Factory Method helps you with SOLID principles by abstracting out how and what instances of concrete type are created and lets you rely on the abstract interface instead. There is another related design pattern called *Abstract Factory* that is also useful in iOS apps, but I rarely see it in the wild.

8.8.2 Adapter

Adapter is a design pattern that helps you, as the name suggests, adapt the interface of one object to the interface of another. This pattern is often used when you try to adapt third-party code that you can't change to your code, or when you need to use something that has an inconvenient or incompatible API. Here's an example:

```
protocol Shareable {

    func socialNetworkingTitle() -> String
    func socialNetworkingUrl() -> NSURL
}

class User {

    let email: String
    let username: String

    init(email: String, username: String) {
        self.email = email
        self.username = username
    }
}

class Post {

    let title: String
    let body: String

    init(title: String, body: String) {
        self.title = title
        self.body = body
    }
}

struct SomeUserInput: Shareable {

    let textContent: String

    func socialNetworkingTitle() -> String {
        return self.textContent
    }

    func socialNetworkingUrl() -> NSURL {
        let escapedContentString = self.textContent.
            \.addingPercentEncoding(withAllowedCharacters: .urlHostAllowed)!
        return NSURL(string: "http://mywebsite.com/\(escapedContentString)")!
```

```
    }
}

class UserShareableAdapter: Shareable {

    let user: User

    init(user: User) {
        self.user = user
    }

    func socialNetworkingTitle() -> String {
        return "Check out this user \(self.user.username)"
    }

    func socialNetworkingUrl() -> NSURL {
        let escapedUsernameString = self.user.username.
            \.addingPercentEncoding(withAllowedCharacters: .urlHostAllowed)!
        return NSURL(string:
          \ "http://mywebsite.com/users/\(escapedUsernameString)")!
    }
}

class PostShareableAdapter: Shareable {

    let post: Post

    init(post: Post) {
        self.post = post
    }

    func socialNetworkingTitle() -> String {
        return "Check out this post \(self.post.title)"
    }

    func socialNetworkingUrl() -> NSURL {
        let escapedPostTitleString = self.post.title.
            \.addingPercentEncoding(withAllowedCharacters: .urlHostAllowed)!
        return NSURL(string:
          \ "http://mywebsite.com/posts/\(escapedPostTitleString)")!
    }
}

class SocialSharingService {

    func shareShareable(shareable: Shareable) {
        print("sharing this on social networking")
        print("with the following title: \(shareable.socialNetworkingTitle())")
        print("and url: \(shareable.socialNetworkingUrl())")
    }
}
```

```
let user = User(email: "some@email.com", username: "some_username")
let post = Post(title: "some post title", body: "post content")

let someUserInout = SomeUserInput(textContent: "this is some user text")
let userShareableAdapter = UserShareableAdapter(user: user)
let postShareableAdapter = PostShareableAdapter(post: post)

let socialSharingService = SocialSharingService()

socialSharingService.shareShareable(shareable: someUserInout)
socialSharingService.shareShareable(shareable: userShareableAdapter)
socialSharingService.shareShareable(shareable: postShareableAdapter)
```

We have **Shareable** protocol. Those objects that conform to it can get us data necessary for sharing on social networks (think Facebook, Twitter, etc.). We also have a **SocialSharingService** that takes objects that conform to **Shareable** and knows how to send them up on network (or some other way) to share on social networks (side note: notice the name - service, i.e., external communication).

We also have a **SomeUserInput** model object that directly conforms to **Shareable** to be available for sharing by **SocialSharingService**. There are also two other model objects that we'd like to share: **User** and **Post**. The problem is that they don't conform to the **Shareable** protocol, nor should they. They could either be unavailable to us to change (think a third-party library classes) or, in this particular case, it doesn't make sense and breaks SRP to have them implement **Shareable**. The reason it doesn't make sense and breaks SRP is that if we ever want to change the way we share things and the data we want to have for sharing, we'd have to modify those model classes. It is not their responsibility to change when we change sharing. Their single responsibility is to represent User- and Post-domain-specific data in our application. No more, no less.

The solution to that problem is adapters. In our case we introduce two adapters **UserShareableAdapter** and **PostShareableAdapter** that themselves conform to the **Shareable** protocol and take in and wrap respective **User** and **Post** objects as parameters. Later, when **SocialSharingService** asks,

they will use those `user` and `post` objects' data to satisfy `Shareable` and `SocialSharingService` API, therefore "adapting" `user` and `post` objects to its API.

So now if we ever want to share another model object, let's say a `Product`, then we don't have to break its SRP and can just create another adapter that will be supplying the right data from product to `SocialSharingService` which preserves the `SocialSharingService` API and keeps it unchanged regardless of new objects it needs to share. That way we not only implemented the **Adapter** pattern but we also covered a lot of SOLID principles by doing it.

8.8.3 Decorator

Decorator is a wrapper around another class that enhance its capabilities. It wraps around something that you want to decorate, implements its interface, and delegates messages sent to it to the underlying object or enhances them or provides its own implementation.

Let's take a look at the following example:

```
protocol Product {
    func price() -> Int
    func name() -> String
}

class FullPriceProduct: Product {

    func price() -> Int {
        return 1000
    }

    func name() -> String {
        return "I'm a product"
    }
}

class DiscountedProductDecorator: Product {

    private let decoratedProduct: Product

    init(decoratedProduct: Product) {
```

```
        self.decoratedProduct = decoratedProduct
    }

    func price() -> Int {
        return Int(Float(decoratedProduct.price()) * 0.75)
    }

    func name() -> String {
        return decoratedProduct.name()
    }
}

class CheckoutManager {

    func checkout(product: Product) {
        let name = product.name()
        let price = Double(product.price() / 100)
        print("charging customer $\(price) for \(name)")
    }
}


let fullPriceProduct = FullPriceProduct()
let discountedProduct = DiscountedProductDecorator(decoratedProduct:
                                                    \ fullPriceProduct)

let checkoutManager = CheckoutManager()

checkoutManager.checkout(product: fullPriceProduct)
checkoutManager.checkout(product: discountedProduct
```

Here we have a **`Product`** protocol that defines the interface all of our products will have. There's a**`FullPriceProduct`** class that implements **`Product`** protocol, and it is a simple model class, nothing to it. We also have a **`CheckoutManager`** class, instances of which operate with objects that implement the **`Product`** protocol. The interesting thing is **`DiscountedProductDecorator`**. It is used to apply one or many discounts to a product. The way it works is it implements a **`Product`** interface and wraps around a decorated product object. It delegates all the messages sent to it to an underlying **`decoratedProduct`** and adds ("decorates with") additional behavior in the **`price()`** method to apply a discount to the resulting product price. At the end of the day you can wrap your objects in multiple decorators and use them just like the objects they decorate because they comply to the same protocol. The users of the **`Product`**

protocol don't have to know that they are working with a decorator that enhances the original object. We have complied with multiple SOLID principles again, especially the Open/Closed Principle.

8.8.4 Command

Command is a design pattern where you'd implement an object that represents an operation that you would like to execute. That operation can have its own state and logic to perform the task it does. The main advantages of this design pattern are that you can hide internal implementation of the operation from the users, you can add undo/redo capabilities to it, and you can execute operations at a later point in time (or not at all) instead of right away where the operation was created. Let's look at the following example:

```
import Foundation

protocol Command {
    func execute()
}

class HTTPGetRequestCommand: Command {

    private let url: URL

    var result: String?

    init(url: URL) {
        self.url = url
    }

    func execute() {
        print("fetching data from \(self.url)")
        print(".....")
        print("done")
        self.result = "this is some json that we got from the backend"
    }
}

class StorageSaveCommand: Command {

    private let dataToSaveToDisk: String

    init(dataToSave: String) {
```

```
        self.dataToSaveToDisk = dataToSave
    }

    func execute() {
        print("saving \(self.dataToSaveToDisk) to disk")
        print("......")
        print("done")
    }
}


let productUrl = URL(string: "http://my-awesome-app.com/api/v1/products/12345")!

let getRequestCommand = HTTPGetRequestCommand(url: productUrl)
getRequestCommand.execute()
let jsonResult = getRequestCommand.result!

let saveToStorageCommand = StorageSaveCommand(dataToSave: jsonResult)
saveToStorageCommand.execute()
```

In the preceding example we have the**Command** protocol that has the **execute()** method that will be the common interface to start execution of our operations. We have the **HTTPGetRequestCommand** that fetches data at a given URL when executed. It also has a **result** variable that holds the result of command execution (we could've also used blocks or direct value return from the **execute()** method).

We also have **StorageSaveCommand** that saves given data to disk when executed. There's nothing much to it.

Notice how both commands are initialized but they don't do anything except holding data until the **execute()** method is called. That's what makes commands so powerful. If you have a more complicated command it could aggregate data it needs for execution over time and can even change the values it has stored in itself before the actual execution happens or a command could be never executed, for example.

To add undo/redo mechanics you'd utilize an array of **Command** objects and execute them as you push or pop from the list. Showing an example of that is another discussion for another chapter. If you want to learn more, please refer to the resources at the end of this chapter.

8.8.5 Template

Template is a design pattern where the main concept is to have a base class that outlines the algorithm of what needs to be done. The base class has several abstract methods that are required to be implemented by its concrete subclasses. These methods are called hook methods. Users of the Template Method classes only interact using the base class that implements the algorithm steps, concrete implementations of those steps are supplied by subclasses.

The following example demonstrates template method pattern:

```
class Report {

    let title: String
    let text:  [String]

    init(title: String, text: [String]) {
        self.title = title
        self.text = text
    }

    func outputReport() {
        outputStart()
        outputHead()
        outputBodyStart()
        outputBody()
        outputBodyEnd()
        outputEnd()
    }

    internal func outputStart() {
        preconditionFailure("this method needs to
                                be overridden by concrete subclasses")
    }

    internal func outputHead() {
        preconditionFailure("this method needs to
                                be overridden by concrete subclasses")
    }

    internal func outputBodyStart() {
        preconditionFailure("this method needs to
                                be overridden by concrete subclasses")
    }

    private func outputBody() {
        text.forEach { (line) in
```

```
            outputLine(line: line)
        }
    }

    internal func outputLine(line: String) {
        preconditionFailure("this method needs to
                                be overridden by concrete subclasses")
    }

    internal func outputBodyEnd() {
        preconditionFailure("this method needs to
                                be overridden by concrete subclasses")
    }

    internal func outputEnd() {
        preconditionFailure("this method needs to
                                be overridden by concrete subclasses")
    }
}

class HTMLReport: Report {

    override func outputStart() {
        print("<html>")
    }

    override func outputHead() {
        print("<head>")
        print("     <title>\(title)</title>")
        print("</head>")
    }

    override func outputBodyStart() {
        print("<body>")
    }

    override func outputLine(line: String) {
        print("     <p>\(line)</p>")
    }

    override func outputBodyEnd() {
        print("</body>")
    }

    override func outputEnd() {
        print("</html>")
    }
}

class PlainTextReport: Report {
```

```
    override func outputStart() {}

    override func outputHead() {
        print("==========\(title)==========")
        print()
    }

    override func outputBodyStart() {}

    override func outputLine(line: String) {
        print("\(line)")
    }

    override func outputBodyEnd() {}

    override func outputEnd() {}
}


let htmlReport = HTMLReport(title: "This is a a great report",
                            text: ["reporting something important 1",
                                   "reporting something important 2",
                                   "reporting something important 3",
                                   "reporting something important 4"])

htmlReport.outputReport()

let plainTextReport = PlainTextReport(title: "This is a a great report",
                                      text: ["reporting something important 1",
                                             "reporting something important 2",
                                             "reporting something important 3",
                                             "reporting something important 4"])

plainTextReport.outputReport()
```

Here we have **Report** outline the structure of the algorithm that we have to print reports. But it's an abstract class that doesn't know the specific concrete implementations to get all the bits and pieces in place to actually print a report. It is a template. Subclasses **HTMLReport** and **PlainTextReport** provide the specifics by implementing "hook" methods.

Users of **Report** will rely on its abstract interface instead of concrete **HTML-Report** or **PlainTextReport**, that way we conform to SOLID principles again.

One of the most used Template Method implementations on iOS is **UIView-**

`Controller`. Every time we subclass it, it provides "the algorithm implementation" and we override "hook methods" such as `viewDidLoad()`, `viewWillAppear()`, and so on.

More about patterns:

For more details on **Factory Method**, **Adapter**, **Decorator**, **Command**, **Template**, and many other design patterns' implementations refer to the books Pro Design Patterns in Swift and Pro Objective-C Design Patterns for iOS.

The patterns here are not the only ones you can use on iOS, but those are the most common ones besides MVC and basics that I've seen in the wild.

Red Flag: Sticking only to *MVC*, *Singleton*, *Delegate*, and *Observer* patterns is fine when you're starting up with the iOS platform, but for advanced things you need to reach deeper into more abstract and high-level stuff like Gang of Four OOP Design Patterns. They are very useful and make your codebase more flexible and maintainable.

8.9 Conclusion:

In this chapter we've covered design patterns and iOS apps architecture. These are some of the most important things to know for iOS developers to get better at their craft. By applying good architecture and design patterns, you help yourself and your colleagues to have the common ground, language, and nomenclature for things in code that you work with day to day. It improves the readability, recognizability, maintainability, and flexibility of your code. It helps you follow SOLID principles, which will help your code stand the test of time and the most important and inevitable test that your codebase could ever face - *change!*

Chapter 9

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

This chapter is a continuation of *Chapter 6 Step Five: Learn How to Store Data*. In this chapter we'll go over a refactoring process where we will evolve storage classes that our application uses, starting with a simple in-memory array, to `NSUserDefaults`, to on-disk file storage, and then eventually to `Core Data`. All along this process we will preserve the API of our storage unchanged, adhering to SOLID principles so that users of our objects and classes are not concerned and coupled to our internal implementation. That way it will be easy for us to change it, as you will see.

The reason this chapter is called a **bonus**" chapter is because it is not technically focused on interviews and interview questions, but is instead about **practical day-to-day iOS development itself**. This chapter is a **sneak peek of my next book** that **covers more practical stuff** like what you see in this chapter. If you're interested in hearing about updates and progress on the next book please sign up for the wait list here:

http://iosinterviewguide.com/next_book

Every app, big or small, needs to store data. Typically when iOS developers think about storing data they think about Core Data or a similar database solution. But the goal for us is to be practical and to know our options. It turns out that Core Data is not always the best solution, and sometimes something simpler might suffice.

9.1 Storage Layer

A quick recap:

The **storage layer** can be as simple as an array or a dictionary of data that holds models in memory for your app. Or it can be as complex as a Core Data or custom SQL ORM solution that can be observed and queried with advanced predicates. The main purpose and responsibility of that layer is to store data for your application and to play the role of the ultimate source of truth for the rest of your code.

9.2 Typical tools Used for Persistence in the Storage Layer

The following classes, objects, and libraries/frameworks (in ascending order of complexity) are used in the storage layer:

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

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

Probably when you hear the words "storage layer" you instantly think about Core Data or a similar database technology that helps you persist things into tables. But surprisingly enough, your storage layer could be as simple as 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`, and `NSDictionary` could all 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 it is actually the only thing you need
- can use key-value observing (KVO) to be notified of 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:

Let's say your app is displaying posts that are fetched from the backend. A typical storage class for **`Posts`** will look like this:

```
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))
```

Compared 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 losing data from one launch of the app to another and can quickly and easily fetch data from the backend, in-memory storage is an invaluable, straightforward, and easy tool to use.

```
NOTE: ## Models and Collections

In this section, we talk about tools used to store data. In all cases, the data
we are storing are 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, 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 the underlying storing mechanism. Abstraction usefulness will be more apparent.

9.4 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 because there's no underlining table or graph structure. At the same time, they do persist objects to disk, unlike an in-memory solution. Simply put, **`NSUserDefaults`** and **`Keychain`** are just key-value storage that you can write primitive data to.

9.4.1 NSUserDefaults

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

Typically we think of **NSUserDefaults** as a solution to store user settings or preferences or tokens in (although tokens 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 following example, 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 in
            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 the **`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` to 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 use `storageNameSpacePrefix` so that we get all the keys for our stored `Posts` later (otherwise `self.userDefaults.dictionaryRepresentation().keys` will return all they keys in `NSUserDefaults`).

The important 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:

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

Disadvantages:

- can't easily use KVO for notification (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

9.4.2 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 anything stored in `NSUserDefaults`, will persist and survive an 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 that 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.

9.5 File/Disk Storage

File and disk storage are typically used to persist bigger chunks of data like images and videos but they can also be used as a substitute for your database. File and disk storage are perfectly capable of storing the same type of objects as **`NSUserDefaults`**: primitives like **`String`** and **`NSNumber`**, and dictionaries, arrays, and custom objects that conform to the **`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) -> Bool in
                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_storage"
    }

    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 the underlying storing mechanics are 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 the **`NSFileManager`** API, but other than that, it remains the same overall. We still use the **`remoteId`** of each post as a unique key to identify and access each post. To get all posts that were stored in the **`getAllPosts()`** method, we examine the folder 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 them to disk. And when retrieving individual **`Post`** objects from memory in the **`findPostByRemoteId()`** method, we get **`NSData`** for that

unique `remoteId` and then decode it to the`Post` object.

And the great thing is that, 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:

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

This storage mechanism 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.

9.6 Core Data

Finally, we've got the 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 uses 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.

```
NOTE: There are other database alternatives to Core Data such as Realm and SQLite.
We will not cover them in this edition, but you can learn more here
https://realm.io and here https://github.com/ccgus/fmdb
```

There is a lot to Core Data and it is a fairly complex piece of technology but overall you typically work with it using the following classes and objects:

- **`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 the database.
- **`NSFetchRequest`** is a "query" object that you use to retrieve **`NSManagedObjects`** from the 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: use `NSManagedObject` subclasses as your models or map and serialize your custom model objects to `NSManagedObjects` and use them only for persisting data to the database.

9.6.1 Going the 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` you use to retrieve, update, and save them. It is a 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 very bulky and difficult to work with.

9.6.2 Going the 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)
                                                    -> NSEntityDescription
    {
        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: managedObjectModel)
        do {
            try persistentStoreCoordinator.
                \.addPersistentStoreWithType(NSInMemoryStoreType,
                                                    configuration: nil,
                                                             URL: nil,
                                                         options: nil)
        }
        catch {
            print("error creating persistentStoreCoordinator: \(error)")
        }

        managedObjectContext = NSManagedObjectContext(
                                    concurrencyType: .MainQueueConcurrencyType)
        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? [PostManagedObject]
    }
}
```

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 the database. **PostManagedObject** is only used internally by **PostsStorage** to actually get data in and out from the database.

PostsStorage had experienced quite a change and now has **NSPersistentStoreCoordinator** to set up the database and entities that are going to be stored in it, and **NSManagedObjectContext** to help with data persistence and fetching.

We are setting up our database in a **PostsStorage** initializer with **NSPersistentStoreCoordinator**, **NSEntityDescription**, and **NSAttributeDescriptions** (for **remoteId** and **name** properties). Normally this setup will happen somewhere else in the iOS application with the help of Xcode's **Data Model** files, and instead of **NSInMemoryStoreType** 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 a **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 retrieve objects from the database, we get back **PostManagedObjects** in the **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()**, and **findPostByRemoteId()** methods that save and find **Posts** in and from the database:

```
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))
```

9.7 Storage Layer Plays Dual Role: Persistence and Data Mapping and Serialization

As you saw with the previous examples, no matter how complex or simple those examples were, 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 have some kind of permanent storage mechanism (in-memory dictionary, `NSUserDefaults`, `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 the 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 previous examples but there are plenty of libraries out there that can help you with that. Mantle and MTLManagedObjectAdapter are my go-to choices when working with Core Data and JSON, for example.

9.8 Switching Storage

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 say you start working on a new app or a new feature and don't know yet if you need persisted storage on a disk database. 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.

9.9 FRP in the Storage Layer.

Core Data, 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 functional reactive tools in iOS.

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

9.10 Be Practical in Your Storage Layer Implementation and Decisions

I'm a big believer in the **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. When you get a handle on Core Data, it sounds great to use it everywhere for every application. This is not always the best approach. When you have a hammer, everything looks like a nail.

In my experience `in-memory`, `NSUserDefaults`, and `file/disk` storage are useful for prototyping and for applications with small data footprints that can survive data wipes 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 and sort and filter them.

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

9.11 Conclusion

As mentioned in the introduction to this chapter, it is not focused on interviews and questions but rather on practical day-to-day coding. This is a **sneak peek** of my next book where I talk more about practical approaches like this one to the other layers of responsibility in iOS applications, such as in the service layer, UI layer, and so on. If you'd like to follow along with the book writing progress, feel free to sign up here:

http://iosinterviewguide.com/next_book

Chapter 10

Outro

Ok, so you've made it. You got to the end of this book.

I hope you learned something, or even a lot, along the way. And my best hope is that you've gotten your dream job with the help of this book.

As I mentioned in the preface, I wrote this book because there wasn't anything like it out there, and when I was less experienced, I wished there was. The main message of this book isn't knowing all the questions and answering them "just right" so you can pass that current pseudoexam called a technical interview. The main message instead is you should know your shit, and then interviews will be easy. The best approach I found to learning software development concepts and languages, or anything you're trying to learn, really, is to systematize your learning. Get the big picture overview of what there is to learn about a subject and then start to drill down into each branch of that tree. It's as simple as that.

Good luck!