Fully supports Xcode 10, Swift 4.2, iOS 12 and iPhone X

Beginning iOS 12 Programming with Swift

Learn how to build a real world app for iPhone and iPad from scratch

SIMON NG

APPCODA

SAMPLE

Table of Contents

Published by AppCoda Limited

What You Will Learn in This Book

I know many readers have an app idea but don't know where to begin. Hence, this new book is written with this in mind. It covers the whole aspect of Swift programming and you will learn how to build a real-world app from scratch. You'll first learn the basics of Swift, then prototype an app, and later add some features to it in each chapter. After going through the whole book, you'll have a real app. During the process, you will learn how to exhibit data in table view, customize the look & feel of a cell, design UI using Stack Views, create animations, work on maps, build an adaptive UI, save data in local database, upload data to iCloud, use TestFlight to arrange beta test, etc.

This new book features a lot of hands-on exercises and projects. You will get the opportunities to write code, fix bugs and test your app. Although it involves a lot of work, it will be a rewarding experience. I believe it will allow you to master Swift 4.2, Xcode 10, and iOS 12 programming. Most importantly, you will be able to develop an app and release it on App Store.

Audience

This book is written for beginners without any prior programming experience and those who want to learn Swift programming. Whether you are a programmer who wants to learn a new programming language or a designer who wants to turn your design into an iOS app or an entrepreneur who wants to learn to code, this book is written for you.

I just assume you are comfortable using macOS and iOS as a user.

What People Say About This Book

"This book got me an internship and a job. After one week of following this book's tutorial, I was immediately able to begin developing my own app! 4 months later, I got an offer at Ancestry to intern as an iOS developer. Best money I ever spent!!"

- Adriana, iOS developer at Ancestry

"I have published 8Cafe and 8Books, apps based on, and inspired by, the AppCoda Swift iOS book; it was a pleasure to learn and develop with your team. In fact, a lot of my apps/games utilise ideas and techniques from your excellent Beginner and Intermediate Swift books. To me, and a lot of developers, your talent, knowledge, expertise and willingness to share have been simply a godsend."

- Mazen Kilani, creator of 8Cafe

"I've been developing iOS apps for about a year now and am greatly indebted to the team at AppCoda. The Swift books I've purchased from them have dramatically increased my productivity and understanding of the entire Xcode and iOS development process. I've learned much more than I ever would have by scouring StackOverflow and github for hours and hours, which is what I had been doing prior to deciding to use AppCoda. All of the information is updated and accurate, simple to read and follow, and the sample projects are fantastic. I really can't recommend these books highly enough. If you're trying to jump-start your Swift education, go for it."

- David Gagne, creator of Bartender.live

"AppCoda's books are fantastic. They are clearly written, assuming no knowledge, but still push you to think for yourself and internalize the concepts. No other resource is so comprehensive."

- JP Sheehan, Ingot LLC

"The book is well written, concise, with excellent example code and real-world examples. It's really helped me get my first App on the App store, and given me many ideas for further enhancements and updates. I also use it as a reference guide ongoing as well with the language, and the updates produced as Swift and iOS change are much gratefully received."

- David Greenfield, creator of ThreadABead

"Thanks for making such an awesome book! This book helped me develop my first real app and have made $200 on the app store in less than 2 months since launch. I was also able to get a software developer job where now I am running the Mobile department.

Thanks again for the great book, I always try to promote it when people ask me about learning how to code."

- Rody Davis, Developer of Pitch Pipe with Pitch Assistant

"The book is really good. I was taking other courses of Swift from Udemy and the instructor did not have much background as a developer. In your case, I know you have a good background as a developer. By the way, you explain the things."

- Carlos Aguilar, creator of Roomhints Interior Design Ideas

"For years, I'd been looking in vain for good quality resources to help sharpen my app development skills. Your books saved my life. They're the best explained programming books I've ever read in my 10 years of programming. They're so easy to understand and they hit everything. I will never thank you enough for writing the books and I owe you a lot."

- Eric Mwangi

"This book is clearly written with lots of examples. It is also great for experienced programmers new to Swift."

- Howard Smith, Flickitt

"Without this book, I couldn't become an iOS developer."

- Changho Lee, SY Energy

"I wanted to learn about iOS programming with Swift. For this, I turned to this book. It's an absolutely great way to learn Swift and iOS app development. If you have some programming background, you'll be able to do real stuff within a couple of days. But even if you do not, you'll still be able to learn to develop apps."

- Leon Pillich

"This book is the best book I have found on the Internet. It is very straightforward. I started my programming journey three years ago and currently, all my app achievement was due to this book."

- Aziz, Engineer at Kuwait Concepts

"Insightful, helpful, and motivational. The books are full of knowledge and depth of the subject, providing hints and tips on many aspects of iOS development, and encourages the student/reader to push forward and to not be afraid of seeking a deeper understanding of the concepts. Just awesome."

- Moin Ahmad, Creator of Guess Animals

"This book taught me how to build the structure for the type of app I wanted to create. The lessons are well laid out, each one is just the right length to avoid overload. I would highly recommend this book as an excellent introduction to creating your first app and beyond."

- Stephen Donnelly, Director at Rascalbiscuit

"I tried multiple learning sources including the Stanford training. Although I already did learn some topics like auto layout, delegates, segues, etc through other sources, your book was the first one that really made me understand them!"

- Nico van der Linden, SAP developer at Expertum

"Over the past three years, I have purchased more than a dozen books on Objective C and Swift. As a high school AP computer science teacher I work mainly with Java but I also teach several other programming languages so I tend to keep a large library of books on-hand. While many of the other books and online video tutorials I purchased these past years were very good, I found AppCoda's to be far above all others. Simon has a way of presenting a topic in such a manner where I felt he was teaching me in a classroom environment rather than just me reading words on a screen. The best way to describe his writing style is to say it feels like he is speaking to you, not just giving you instructions."

- Ricky Martin, Gulf Coast High School

"This was one of the easiest books I have found to learn Swift. As a beginner, it was extremely easy to follow and understand. The real-life examples you include as your work through the book and build the app being taught is genius and makes it all worth it in the end. I was able to take many things away and apply to my own apps. I find myself referring back to it many times. Great Work."

- Bill Harned, creator of Percent Off

"Best books on iOS development, well designed and easy to follow, and a great development journey companion."

- Ali Akkawi, freelancer in mobile apps development for iOS and Android

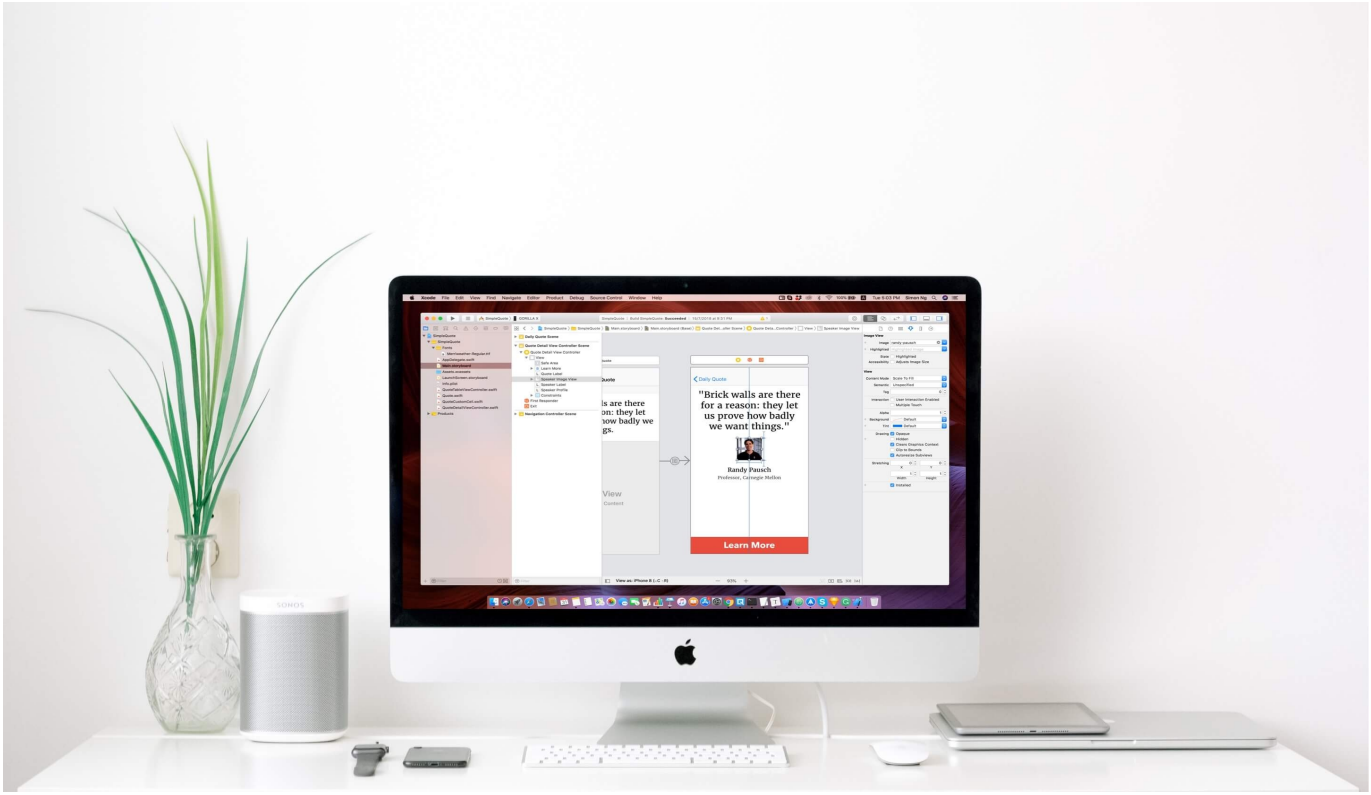
"I like the book. The contents are well structured. We have almost all of the latest concepts covered."

- Barath V, Lead iOS developer at Robert Bosch LLC

"I have purchased both the Beginning and Intermediate iOS 11 Programming with Swift books. I am a Java developer turned iOS mobile developer and these books really helped me learn the concepts of building a mobile application. The FoodPin application that you build in the Beginner book is an excellent way to learn all of the most common components of a mobile app. Even though I have now been working on iOS apps for over three years, I still regularly go back to the AppCoda swift books as a reference."

- Stacy Chang

Chapter 1
The Development Tools, the Learning Approach, and the App Idea



So you want to create your own app? That's great! Creating an app is a fun and rewarding experience. I still remembered the joy when I first created an app years ago, even though the app is just so simple and elementary.

Before we dive into iOS programming, let's go through the tools you need to build an app and get prepared the mindset for learning iOS app development.

The Tools

Apple has favored a closed ecosystem over the open system. iOS can be only run on Apple's own devices including iPhone and iPad. It is very much unlike its competitor, Google, that Android is allowed to run on mobile devices from different manufacturers. As an aspiring iOS developer, what this means to you is that you will need a Mac for app development.

1. Get a Mac

Having a Mac is the basic requirement for iOS development. To develop an iPhone (or iPad) app, you need to get a Mac with an Intel-based processor running on macOS version 10.13.4 (or later). If you now own a PC, the cheapest option is to purchase the Mac Mini. As of this writing, the retail price of the entry model is US$499. You can hook it up to the monitor of your PC. The basic model of Mac mini comes with a 1.4GHz dual-core Intel Core i5 processor and 4GB memory. It should be good enough to run the iOS development tools smoothly. Of course, if you have a bigger budget, get the higher model or an iMac with better processing power.

```
What about Hackintosh? Is it an option if you do not have a
Mac? I heard that we can use it to run Mac on Windows machines.
While you may have heard of some success cases of using
Hackintosh for iOS development, it is not the recommended
approach. If you are serious about learning iOS development and
afford the upfront cost, a Mac is a worthwhile investment.
```

2. Register Your Apple ID

You will need an Apple ID to download Xcode, access iOS SDK documentation, and other technical resources. Most importantly, it will allow you to deploy your app to a real iPhone/iPad for testing.

If you have downloaded an app from the App Store, it is quite sure that you already own an Apple ID. In case you haven't created your Apple ID before, you have to get one. Simply go to Apple's website (https://appleid.apple.com/account) and follow the

procedures for registration.

3. Install Xcode

To start developing iOS apps, Xcode is the only tool you need to download. Xcode is an integrated development environment (IDE) provided by Apple. Xcode provides everything you need to kick start your app development. It already bundles the latest version of the iOS SDK (short for Software Development Kit), a built-in source code editor, graphic user interface (UI) editor, debugging tools and much more. Most importantly, Xcode comes with an iPhone (and iPad) simulator so you can test your app without the real devices.

You have two ways to install Xcode: 1. Download it through the Mac App Store. 2. Manually download it from Apple's developer website.

Download Xcode from the Mac App Store

To download Xcode, launch the Mac App Store on your Mac. If you're using the latest version of Mac OS, you should be able to open the Mac App Store by clicking the icon in the dock. In case you can't find it, you may need to upgrade your Mac OS.

Figure 1-1. App Store icon in the dock

In the Mac App Store, simply search "Xcode" and click the "Get" button to download it.

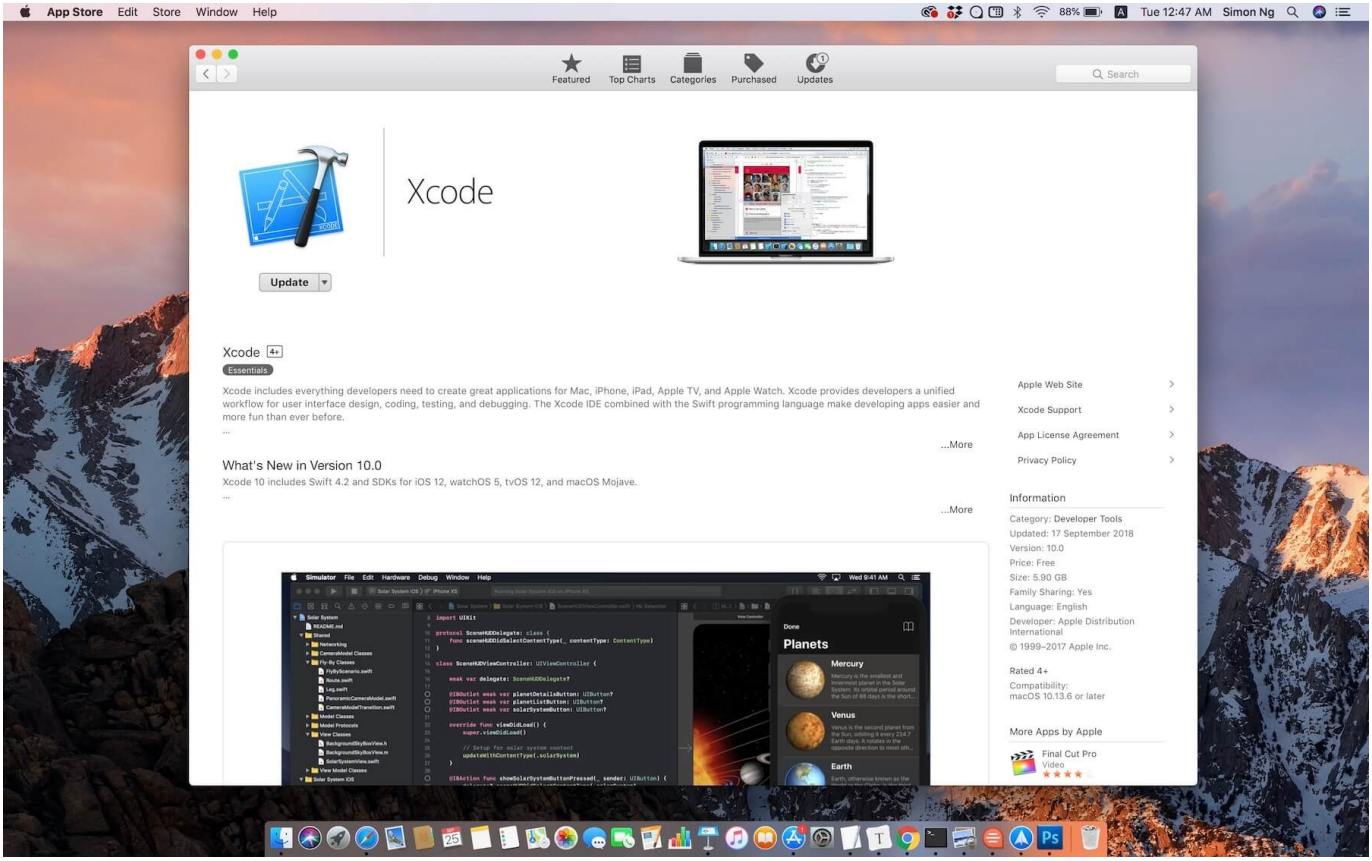


Figure 1-2. Download Xcode 10

Once you complete the installation process, you will find Xcode in the Launchpad.

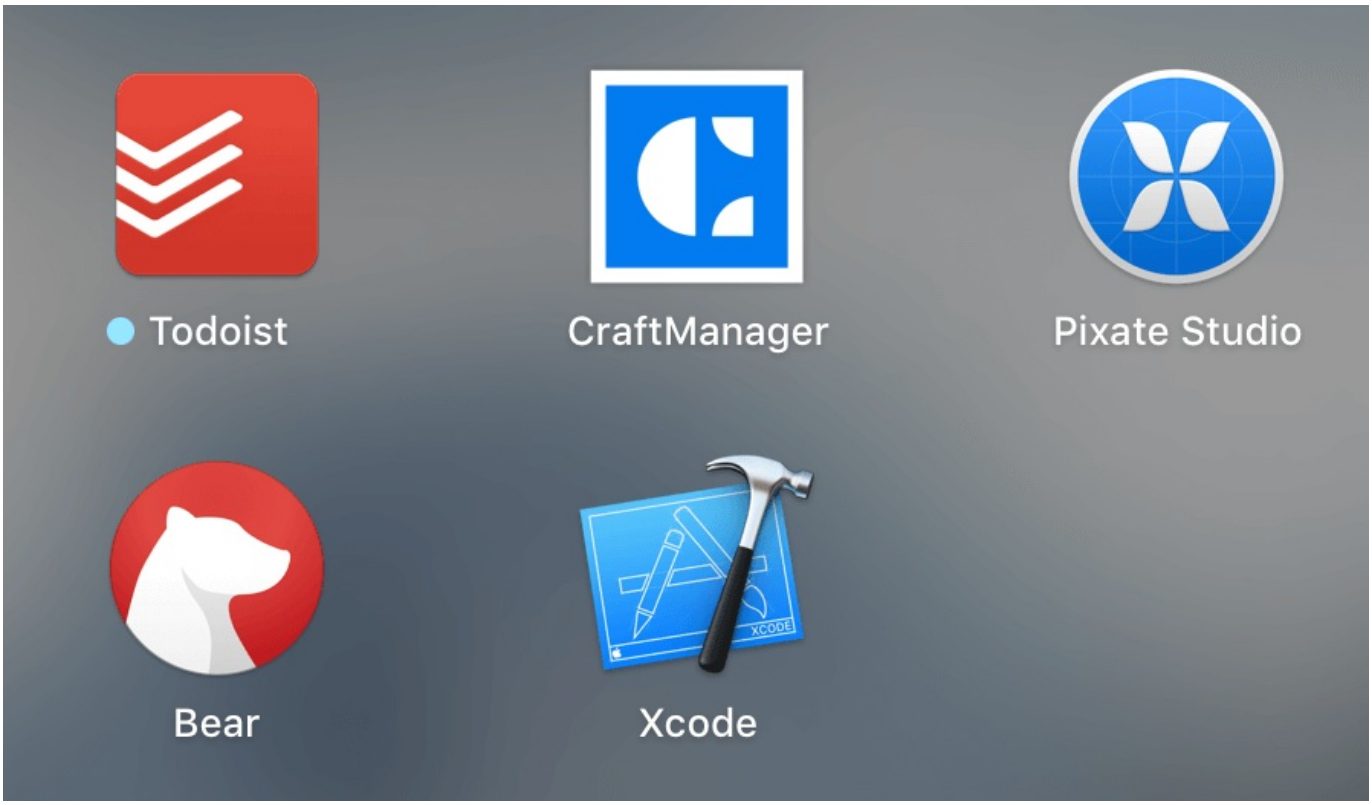


Figure 1-3. Xcode icon in the Launchpad

At the time of this writing, the latest version of Xcode is 10.0. Throughout this book, we will use this version of Xcode to create the demo apps. Even if you have installed Xcode before, I suggest you upgrade to the latest version. This should make it easier for you to follow the tutorials.

Download Xcode from Developer's website

Normally you can download Xcode from the Mac App Store, which is the recommended way for beginners. For any reason you don't want to use Mac App Store, you can download Xcode 9 manually. To get a copy of it, you have to sign into the Apple Developer website (http://developer.apple.com/register/). Select Download Tools and then click Download Xcode 10 GM.

Once the file is downloaded, double-click and install it.

4. Enroll in the Apple Developer Program (Optional)

A common question about developing an iOS app is whether you need to enroll in the Apple Developer Program (https://developer.apple.com/programs/). The short answer is optional. First, Xcode already includes a built-in iPhone and iPad simulator. You can develop and test out your app on Mac, without enrolling in the program.

Starting from Xcode 7, Apple has changed its policy regarding permissions required to build and run apps on devices. Before that, the company required you to pay US$99 per year in order to deploy and run your apps on a physical iPhone or iPad. Now, program membership is no longer required. Everyone can test their apps on a real device without enrolling in the Apple Developer Program. Having said that, if you want to try out some advanced features such as in-app purchase, push notifications or CloudKit, you still need to apply for the program membership. Most importantly, you're not able to submit your app to App Store without paying the annual membership fee.

So, should you enroll in the program now? The Apple Developer Program costs US$99 per year. It's not big money but it's not cheap either. Since you're reading this book, you're probably a newcomer and just start exploring iOS development. The book is written for beginners. We will first start with something simple. You are not going to tap into the advanced features until you grasp the basic skills.

Therefore, even if you do not enroll into the program, you will still be able to follow most of the content to build an app and test it on your device. For now, save your money. I will let you know when you need to enroll in the program. At that time, you're encouraged to join the program as you're ready to publish the app to the App Store!

The Learning Approach

I have been teaching iOS programming from 2012 through blogging, online courses, and in-person workshops. What I found is that it is the learning approach and the mindset that make the difference between failing and achieving. Before we talk about Swift and iOS programming, I want to get you equip with the right mindset and understand the most effective way to learn programming.

Get Your Hands Dirty

One of the most popular questions about learning how to code is:

What's the best way to learn iOS programming?

First, thanks for reading this book. Unfortunately, I have to tell you that you cannot learn programming just by reading books. This book has everything you need to learn Xcode, Swift, and iOS app development.

But the most important part is **taking action**.

If I have to provide an answer to the question, I will say "Learn by Doing". It is at the heart of my teaching approach.

Let me change the question a little bit:

What's the best way to learn English (or other foreign languages)?

What's the best way to learn cycling (or any other sports)?

You probably know the answer. I especially like this answer on Quora about learning a new language:

> Follow this routine: listen 1 hour a day, speak 1 hour a day, publish 1 journal entry.
>
> - Dario Mars Patible

You learn through practice, not by just studying grammar. Learning programming is somewhat very similar to learning a language. You need to take actions. You have to work on a project or some exercises. You have to sit in front of your Mac, immerse yourself in Xcode, and write the Swift code. It doesn't matter how many mistakes you make during the process. Just remember to open Xcode and code while reading this book.

Motivations

Why do you want to learn app development? What motivates you to sacrifice the weekends and holidays to learn how to code?

Some people begin learning app development just because of money. There is nothing wrong with that. You may want to build your app business to earn some side income and eventually turn it into a full-time business. That's completely understandable. Who doesn't want to live a rich life?

As of March 2018, however, there are over 2.1 million apps on the App Store. It is really hard to put up an app on the App Store and expect to make a load of money overnight. You'll be easily discouraged or even give up if money is your primary reason for building apps, especially when you come across articles like this:

- How Much Money I Made on the App Store (https://sitesforprofit.com/how-much-money-app-store)

> **Then reality set in.**
>
> 199 units old = US$209 in sales = US$135 proceeds (net to me). In order to get the app on the app store I needed to pay the $99 developer fee.
>
> So after 2 months and 1 week my (before tax) profit was $36.
>
> - James

Programming is hard and challenging. I find people who successfully master the language are those who have a strong desire to build apps and are enthusiastic to learn programming. They usually have an idea in their mind and want to turn it into a real app. Making money is not their number one concern. They know the app can solve their own problems and will be beneficial to others. With such a powerful purpose in mind, they can overcome any obstacles come up.

So, think again why you want to learn programming.

Find a Buddy

"The best way to learn is to teach" is an old saying. It still works in the modern world, however. You don't need to be an expert to teach. I'm not talking about giving a lecture at a university or teaching a bunch of students in a formal class. Teaching does not always

happen that way. It can be as simple as sharing your knowledge with a colleague or a classmate sitting next to you.

Try to find someone who is also interested in learning iOS programming. When you learn something new, try to explain the materials to your buddy. For example, after building your first app, teach your close friend how it works and how he/she can create an app too.

What if you can't find a buddy to share what you've learned? No worries. Start a blog on medium.com (or whatever platforms you like), write a blog post every day, and document everything you learn.

This is one of the most effective ways of learning as I learn so much while publishing tutorials on appcoda.com, as well as, developing my first book.

Sometimes you think you know the materials well. But once you need to explain the concept to someone else and answer questions, chances are that you didn't understand the material thoroughly. And this will motivate you to study the materials even harder. Give this method a shot while you learn iOS programming.

Be Patient

> Grit is passion and perseverance for very long-term goals. Grit is having stamina. Grit is sticking with your future, day-in, day-out. Not just for the week, not just for the month, but for years. And working really hard to make that future a reality. Grit is living life like it's a marathon, not a sprint.
>
> - Dr. Angela Lee Duckworth

Some of my students asked, "How long would it take to become a good developer?"

It takes time to master programming and become a great developer. It usually takes years. Not weeks, not months but years.

This book will help you kick start the journey. You will learn all the basics of Swift and iOS programming and eventually, build an app. That said, it takes time and lots of practices to become a professional programmer.

Be patient. Don't set your expectations too high for your first app. Just enjoy the process, create something simple and fun. Keep reading and coding daily. You will eventually master the skill.

Find Your App Idea

I always encourage my students to come up with their own app idea when start learning app development. This idea doesn't have to be big. You do not need to build the next Uber app or come up with a new idea to change the world. You just need to start with a very small idea that solves a problem.

Let me give you a couple of the examples.

One classic example that I used to mention is Cockpit Dictionary. It is an app built by Manolo Suarez, who is a pilot by profession. He had an app idea while learning app programming. The idea was not fancy but solved his own problem. There are tens of thousands of Aviation terms in abbreviated form. Even for an experienced pilot with over 20 years of aviation experience, it is impossible to remember all the acronyms and technical terms. Instead of using a print dictionary, he thought of building a handy app for pilots to look up all kinds of Aviation terms. A simple, yet a great idea to solve his own problem.

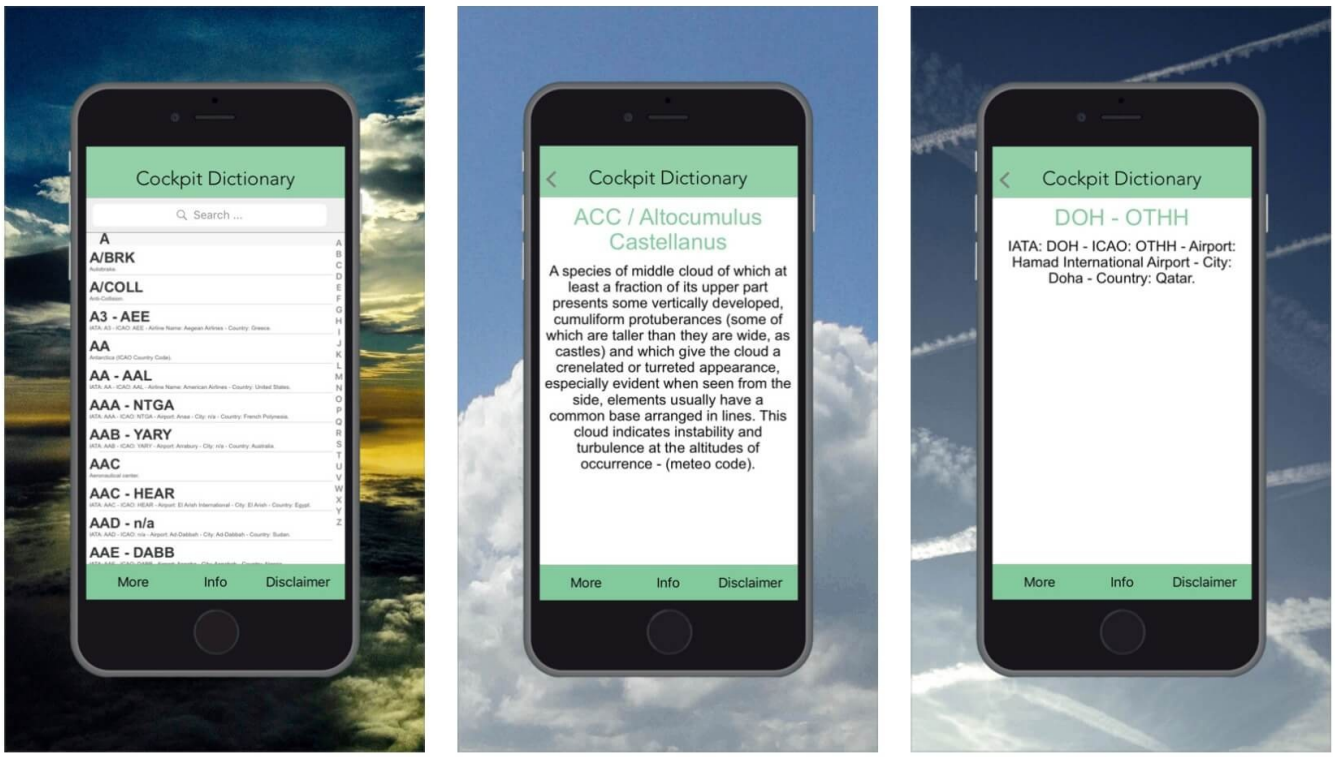


Figure 1-4. Cockpit Dictionary

Another example is the NOAA Buoy Data app. This app retrieves the latest weather, wind, and wave data from the National Oceanic and Atmospheric Administration's (NOAA) National Data Buoy Center (NDBC). Developed by Leo Kin, he came up with the app idea during his recovery from surgery.

> "After the surgery, I had to wear a neck brace for three months. During those three months, I couldn't move a lot and even had a hard time walking or even raising my arms. My physical therapist advised that I go walking as much as I can to get exercise and to build back my atrophied leg muscles.
>
> There is an island close to where I was living that I really enjoyed walking to. The only problem was that it can only be reached during low tide. And if the tide came in, there's no way to get back home except by swimming. Since I was very physically weakened, I was very scared of getting stuck on the island with no way back. While walking, I was always going to NOAA's website to check how high or low the tide was and if I had enough time to walk to the island and back.
>
> During one of my walks, the idea came to me that I should build an app. Even if no one else uses the app, it wouldn't matter because it would help me keep track of the tides and get back in time."
>
> - Leo Kin

His app may not interest you, but it was solving a problem he faced at the time. Probably people on that island would benefit from his app too.

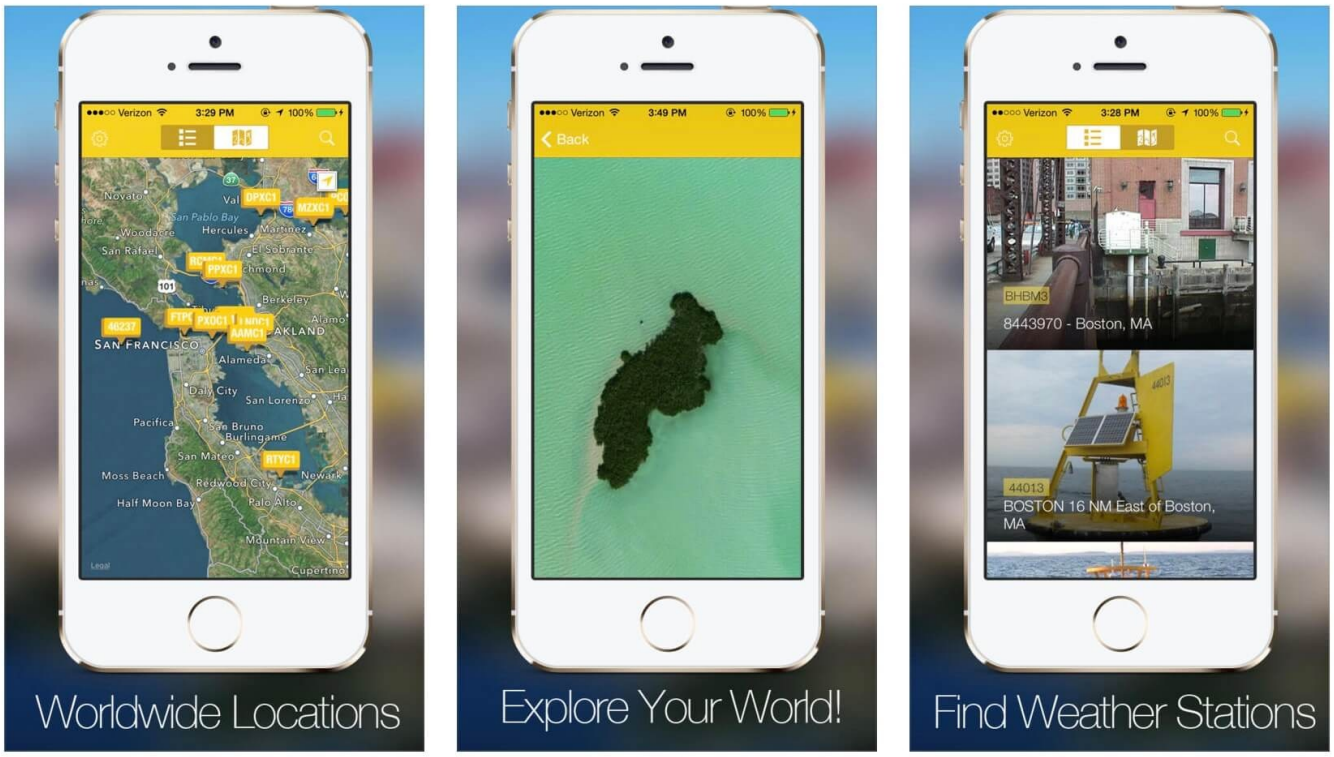


Figure 1-5. NOAA Buoy Data app

Having your own app idea will give you a clear goal and motivate you to keep learning. Now spare some time and write down three app ideas below:

1.

2.

3.

Summary

That's all for the introduction. Take some time to install Xcode on your Mac, and come up with your own app idea. Even though I may not teach you to build the exact same app, you will learn the coding techniques that empower you to build your own app.

When you proceed to the next chapter, we will start programming in Swift.

So get ready!

Chapter 2
Your First Taste of Swift with Playgrounds

Now that you have configured everything you need to start iOS app development, let me answer another common question from beginners before moving on. A lot of people have asked me about what skills you need in order to develop an iOS app. In brief, it comes down to three areas:

- **Learn Swift** - Swift is now the recommended programming language for writing iOS apps.
- **Learn Xcode** - Xcode is the development tool for you to design the app UI, write Swift code, and build your apps.
- **Understand the iOS software development kit** - Apple provides the software

development kit for developers to make our lives simpler. This kit comes with a set of software tools and APIs that empowers you to develop iOS apps. For example, if you want to display a web page in your app, the SDK provides a built-in browser that lets you embed right in your application.

You will have to equip yourself with knowledge on the above three areas. That's a lot of stuff. But no worries. You'll learn the skills as you read through the book.

Let me start off by telling you a bit about the history of Swift.

In the Worldwide Developer Conference 2014, Apple surprised all iOS developers by launching a new programming language called Swift. Swift is advertised as a "fast, modern, safe, interactive" programming language. The language is easier to learn and comes with features to make programming more productive.

Prior to the announcement of Swift, iOS apps were primarily written in Objective-C. The language has been around for more than 20 years and was chosen by Apple as the primary programming language for Mac and iOS development. I've talked to many aspiring iOS developers. A majority of them said Objective-C was hard to learn and its syntax looked weird. Simply put, the code scares some beginners off from learning iOS programming.

The release of Swift programming language is probably Apple's answer to some of these comments. The syntax is much cleaner and easier to read. I have been programming in Swift since its beta release. It's more than 4 years for now. I can say you're almost guaranteed to be more productive using Swift. It definitely speeds up the development process. Once you get used to Swift programming, it would be really hard for you to switch back to Objective-C.

It seems to me that Swift will lure more web developers or even novice to build apps. If you're a web developer with some programming experience on any scripting languages, you can leverage your existing expertise to gain knowledge on developing iOS apps. It would be fairly easy for you to pick up Swift. Being that said, even if you're a total beginner with no prior programming experience, you'll also find the language friendlier and feel more comfortable to develop apps in Swift.

In June 2015, Apple announced Swift 2, and that the programming language goes open source. This is a huge deal. Since then, developers created some interesting and amazing open source projects using the language. Not only can you use Swift to develop iOS apps, companies like IBM developed web frameworks for you to create web apps in Swift. Now you can run Swift on Linux too.

Following the release of Swift 2, Apple introduced Swift 3 in June 2016. This version of the programming language, integrated into Xcode 8, was released in Sep, 2016. This was considered as one of the biggest releases since the birth of the language. There were tons of changes in Swift 3. APIs are renamed and more features were introduced. All these changes helped to make the language even better and enabled developers to write more beautiful code. That said, it took all developers extra efforts to migrate their projects for these breaking changes.

In June 2017, Apple brought you Swift 4, along with the release of Xcode 9, with even more enhancements and improvements. This version of Swift had a focus on backward compatibility. That meant ideally projects developed in Swift 3 could be run on Xcode 9 without any changes. Even if you had to make changes, the migration from Swift 3 to 4 would be much less cumbersome than that from 2.2 to 3.

This year, Apple only releases a minor update for Swift, pushing Swift's version number to 4.2. Even though it's not a major release, the new version also comes with a lot of language features to improve productivity and efficiency. You may wonder if Apple is slowing down the development of Swift by just releasing a minor update. Actually, it's a good news for aspiring developers. In some ways, this means the Swift language becomes more stable and mature.

If you're a total beginner, you may have a couple of questions in mind. Why does Swift keep changing? If it keeps updating, is Swift ready for use?

Nearly all programming languages change over time. The same is for Swift. New language features are added to Swift every year to make it more powerful and developer friendly. It is somewhat similar to our spoken languages. Let's say, for English, it still changes over time. New vocabulary and phrases such as freemium are added to the dictionary every year.

> All languages change over time, and there can be many different reasons for this. The English language is no different.
>
> Source: https://www.english.com/blog/english-language-has-changed

While Swift keeps evolving, it doesn't mean it is not ready for production use. Instead, if you are going to build an iOS app, you should build it in Swift. It has become a de facto standard for iOS app development. Companies such as LinkedIn, Duolingo and Mozilla had already written apps entirely in Swift since its early versions. Since the release of Swift 4, the programming language is more stable, and definitely ready for enterprises and production uses.

Let's Get Started

Enough for the background and history. Let's begin to look into Swift.

To get a taste of Swift programming language, let's take a look at the following code snippets.

Objective-C

```
const int count = 10;
double price = 23.55;

NSString *firstMessage = @"Swift is awesome. ";
NSString *secondMessage = @"What do you think?";
NSString *message = [NSString stringWithFormat:@"%@%@", firstMessage, secondMessag
e];

NSLog(@"%@", message);
```

Swift

```
let count = 10
var price = 23.55

let firstMessage = "Swift is awesome. "
let secondMessage = "What do you think?"
var message = firstMessage + secondMessage

print(message)
```

The first block of code was written in Objective-C, while the second one was written in Swift. Which language do you prefer? I guess you would prefer to program in Swift, especially if you're frustrated with the Objective-C syntax. It's clearer and readable. There is no @ sign and semi-colon at the end of each statement. Both statements below concatenate the first and second messages together. I believe you can probably guess the meaning of the following Swift code:

```
var message = firstMessage + secondMessage
```

but find it a bit confusing for the Objective-C code below:

```
NSString *message = [NSString stringWithFormat:@"%@%@", firstMessage, secondMessag
e];
```

Trying out Swift in Playgrounds

I don't want to bore you by just showing you the code. There is no better way to explore coding than actually writing code. Xcode has a built-in feature called *Playgrounds*. It's an interactive development environment for developers to experiment Swift programming and allows you to see the result of your code in real-time. You will understand what I mean and how Swift Playgrounds works in a while.

Assuming you've installed Xcode 10 (or up), launch the application (by clicking the Xcode icon in Launchpad). You should see a startup dialog.

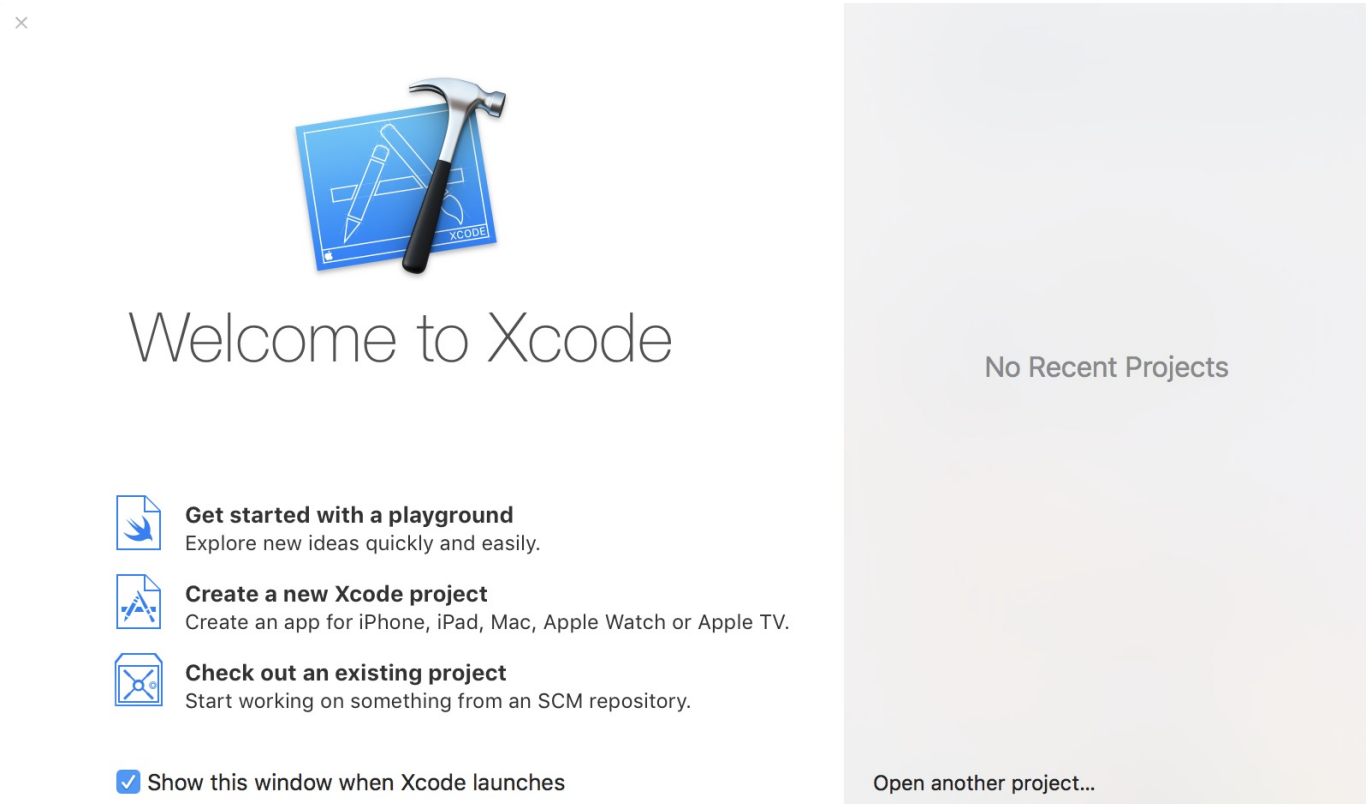


Figure 2-1. The startup dialog

A Playground is a special type of Xcode file. To begin, click "Get started with a playground." You'll then be prompted to select a template for your playground. Since we focus on exploring Swift in iOS environment, choose *Blank* under the *iOS* section to create a blank file. Click *Next* to continue.

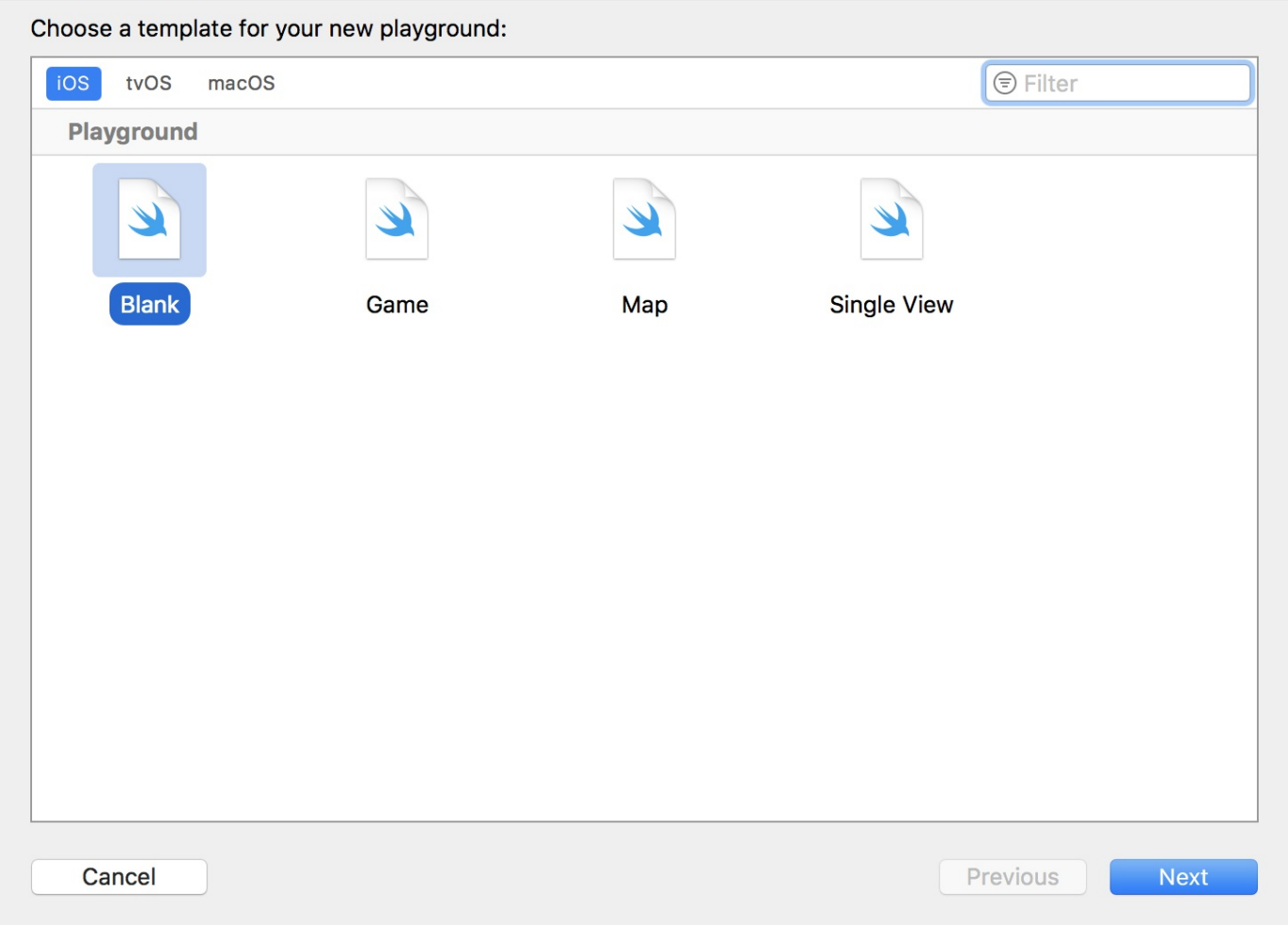
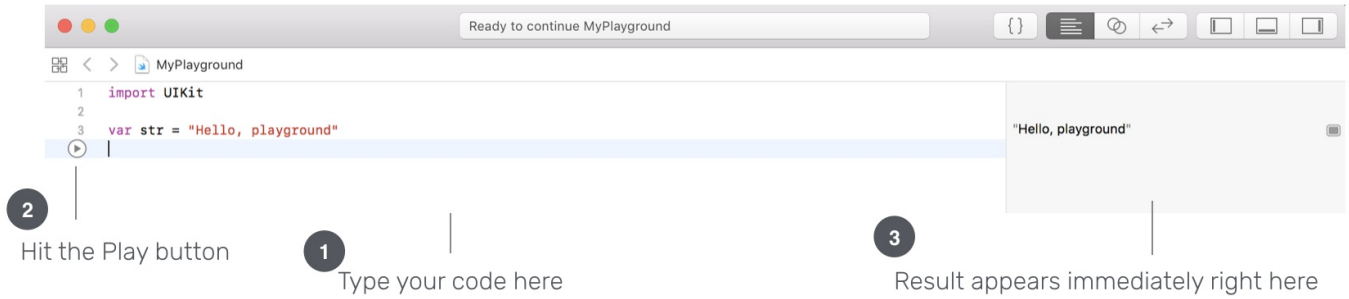


Figure 2-2. Creating a Playground file

Once you confirm to save the file, Xcode opens the Playground interface. Your screen should like this:



On the left pane of the screen, it is the editor area where you type the code. When you want to test your code and see how it works, hit the *Play* button. Playground immediately interprets the code (up to the line of the Play button) and displays the result on the right

pane. By default, Swift Playgrounds includes two lines of code. As you can see, the result of the `str` variable appears immediately on the right pane after you hit the *Play* button at line 4.

We'll write some code in Playgrounds together. Remember the purpose of this exercise is to let you experience Swift programming and learn its basics. I will not cover every feature of Swift. We will only focus on these topics:

1. Constants, variables and type inference
2. Control flow
3. Collection types like arrays and dictionaries
4. Optionals

These are the basic topics that you need to know about Swift. You will learn by example. However, I'm quite sure you will be confused by some of the programming concepts, especially you are completely new to programming. No worries. You will find my study advice in some sections. Just follow my advice and keep studying. And, don't forget to take a break when you're stuck.

Cool! Let's get started.

Constants and Variables

Constants and variables are two basic elements in programming. The concept of variables (and constants) is similar to what you learned in Mathematics. Take a look at the equation below:

```
x = y + 10
```

Here, both `x` and `y` are variables. `10` is a constant, meaning that its value is unchanged.

In Swift, you declare variables with the `var` keyword and constants using the `let` keyword. If you write the above equation in code, here is what it looks like:

```
let constant = 10
var y = 10
var x = y + constant
```

Type the code above in Playgrounds and then hit *Play* at line 5. You will see the result below.

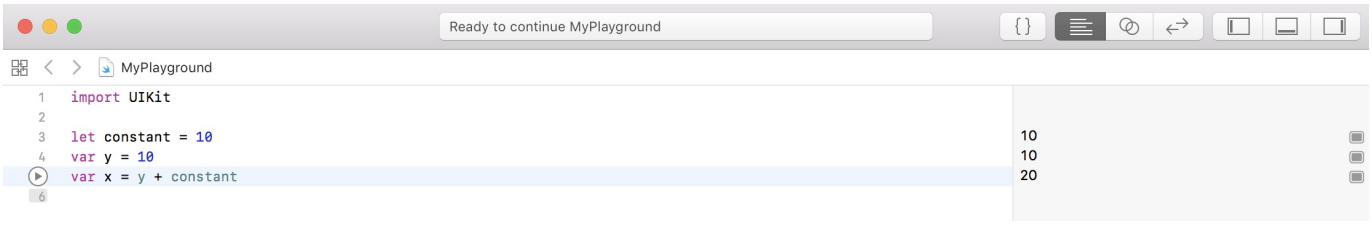


Figure 2-4. The result of the equation

You can choose whatever name for variables and constants. Just make sure they are meaningful. For example, you can rewrite the same piece of code like this:

```
let constant = 10
var number = 10
var result = number + constant
```

To make sure that you clearly understand the difference between constants and variables in Swift, type the following code to change the values of `constant` and `number`:

```
constant = 20
number = 50
```

After that, press shift+command+enter to execute the code. Other than using the *Play* button, you can use the shortcut keys to run the code.

You simply set a new value for the constant and variable. But as soon as you change the value of the constant, Xcode gives you an error in the console. Conversely, there is no issue for `number`.

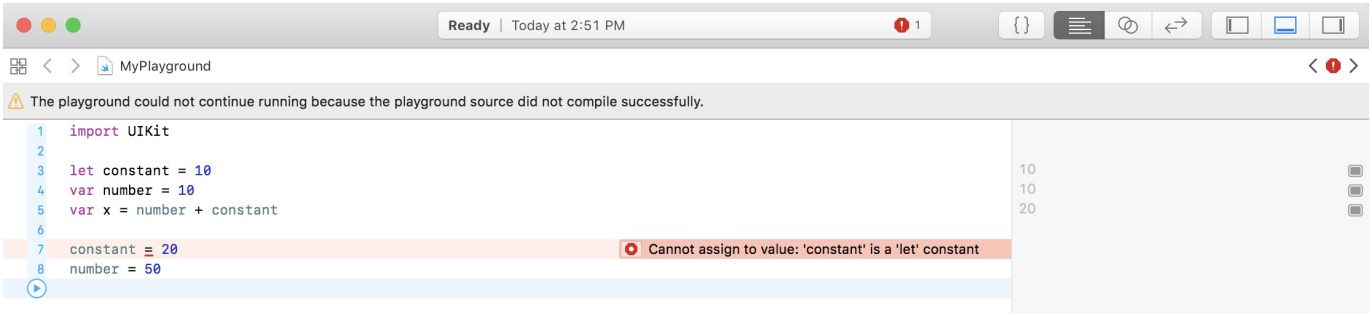


Figure 2-5. Errors in Playgrounds

This is the core difference between constants and variables in Swift. Once a constant is initialized with a value, you can't change it. If you have to change the value after initialization, use variables.

Understanding Type Inference

Swift provides developers with a lot of features to write clean-looking code. One feature is known as *Type Inference*. The same code snippet we just discussed above can be explicitly written as follows:

```
let constant: Int = 10
var number: Int = 10
var result: Int = number + constant
```

Each variable in Swift has a type. The keyword `Int` after colon (`:`) indicates the type of the variable/constant is an *integer*. If the value stored is a decimal number, we use the type `Double` .

```
var number: Double = 10.5
```

There are other types like `String` for textual data and `Bool` for boolean values (true/false).

Now back to *Type Inference*, this powerful feature in Swift allows you to omit the type when declaring a variable/constant to make your code look cleaner. The Swift compiler can deduce the type by examining the default value given by you. This is why we can write the code like this earlier:

```
let constant = 10
var number = 10
var result = number + constant
```

The given value (i.e. `10`) is an integer, so the type is automatically set to `Int`. In Playgrounds, you can hold the *option* key, and click any variable name to reveal the variable type, deduced by the compiler.

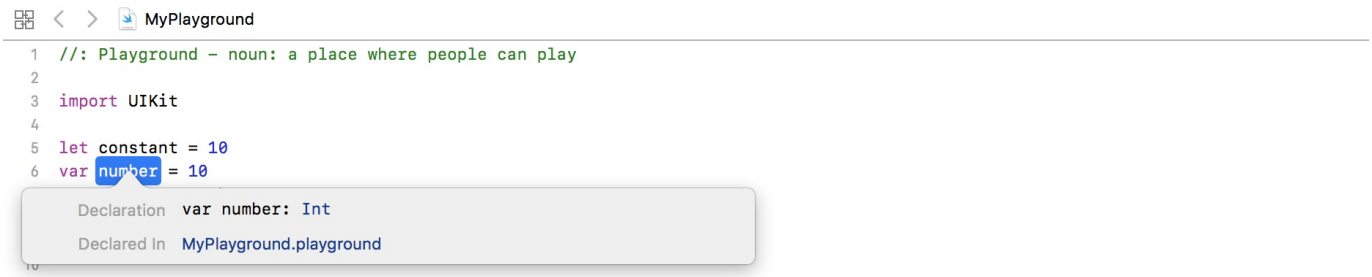


Figure 2-6. Hold option key and select the variable to reveal its type

> Feeling overwhelmed by all the new programming concept?
>
> Just a take a break. You don't have to go through this chapter without a rest. You can even skip this chapter, and read the next one if you can't wait to build your first app. You can always come back to this chapter to study the basics of Swift.

Working with Text

So far, we only work with variables of the type `Int` and `Double`. To store textual data in variables, Swift provides a type called `String`.

To declare a variable of the type `String`, you use the `var` keyword, give the variable a name and assign the variable with the initial text. The text specified is surrounded by double quotes (`"`). Here is an example:

```
var message = "The best way to get started is to stop talking and code."
```

After you key in the line of code above in Playgrounds and hit *Play*, you will see the result on the right pane.

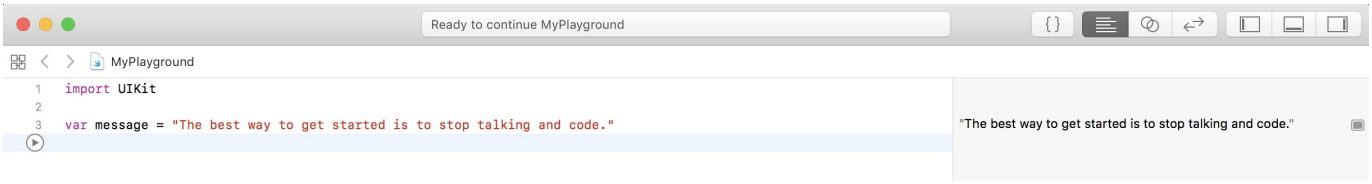


Figure 2-7. The string is shown immediately on the right pane

Swift provides different operators and functions (or methods) for you to manipulate strings. For example, you can use the addition (`+`) operator to concatenate two strings together:

```
var greeting = "Hello "
var name = "Simon"
var message = greeting + name
```

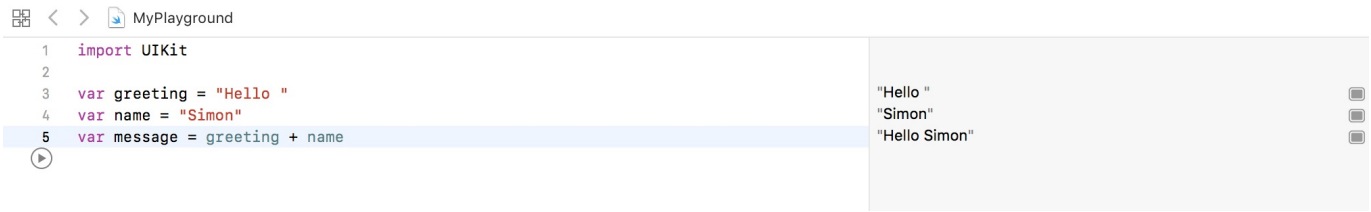
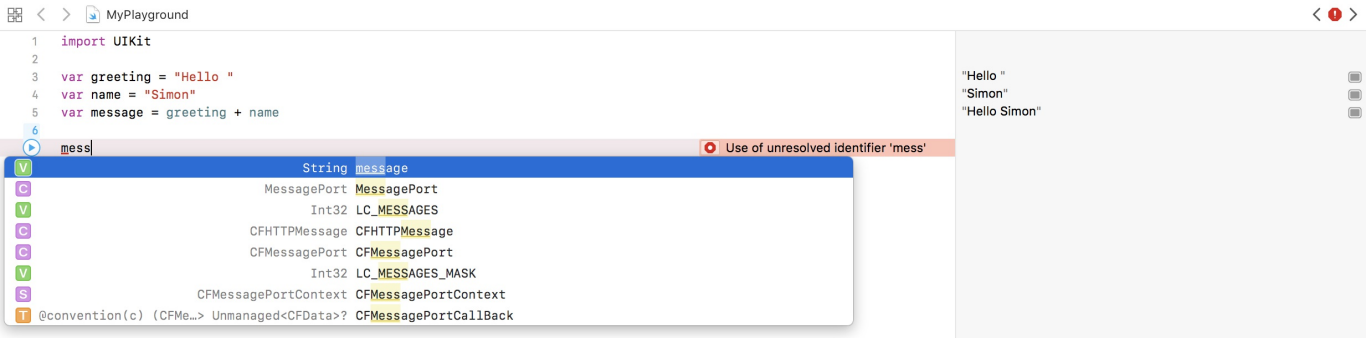


Figure 2-8. String concatenation

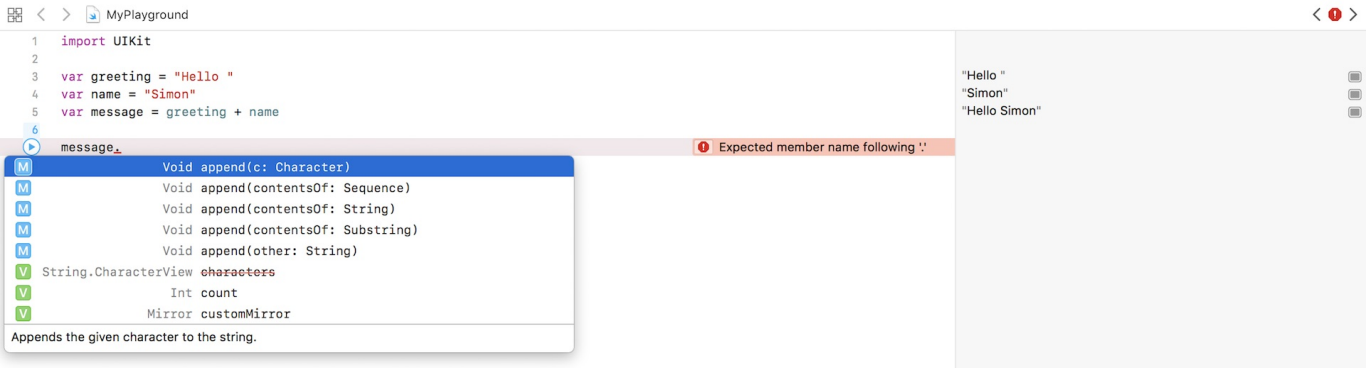
What if you want to convert the whole sentence into upper case? Swift provides a built-in method named `uppercased()` to convert a string to upper case. You can type the following code to have a try:

```
message.uppercased()
```

Xcode's editor comes with an auto-complete feature. Auto-complete is a very handy feature to speed up your coding. As soon as you type `mess`, you'll see an auto-complete window showing some suggestions based on what you have keyed in. All you need to do is to select `message` and hit enter.



Swift employs the dot syntax for accessing the built-in methods and the properties of a variable. As you type the dot after `message`, the auto-complete window pops out again. It suggests a list of methods and properties that can be accessed by the variable. You can continue to type `uppercase()` or select it from the auto-complete window.



Once you complete your typing, you would see the output immediately. When we use `uppercased()` on `message`, the content of `message` is converted to upper case automatically.

`uppercased()` is just one of the many built-in functions of a string. You can try to use `lowercased()` to convert the message to lower case.

```
message.lowercased()
```

Or if you want to count the number of characters of the string, you can write the code like this:

```
message.count
```

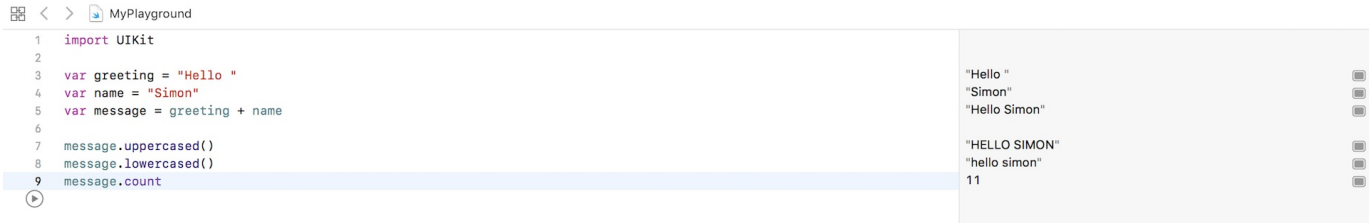


Figure 2-11. Manipulating a string using the built-in functions

String concatenation looks really easy, right? You just add two strings together using the `+` operator. However, it is not always trivial. Let's write the following code in Playgrounds:

```
var bookPrice = 39
var numOfCopies = 5
var totalPrice = bookPrice * numOfCopies
var totalPriceMessage = "The price of the book is $" + totalPrice
```

It is quite usual to create a string that mixes both a string and a number. In the example above, we calculate the total price of the books, and create a message that shows the total price to the user. If you have typed the code in Playgrounds, you will notice an error.

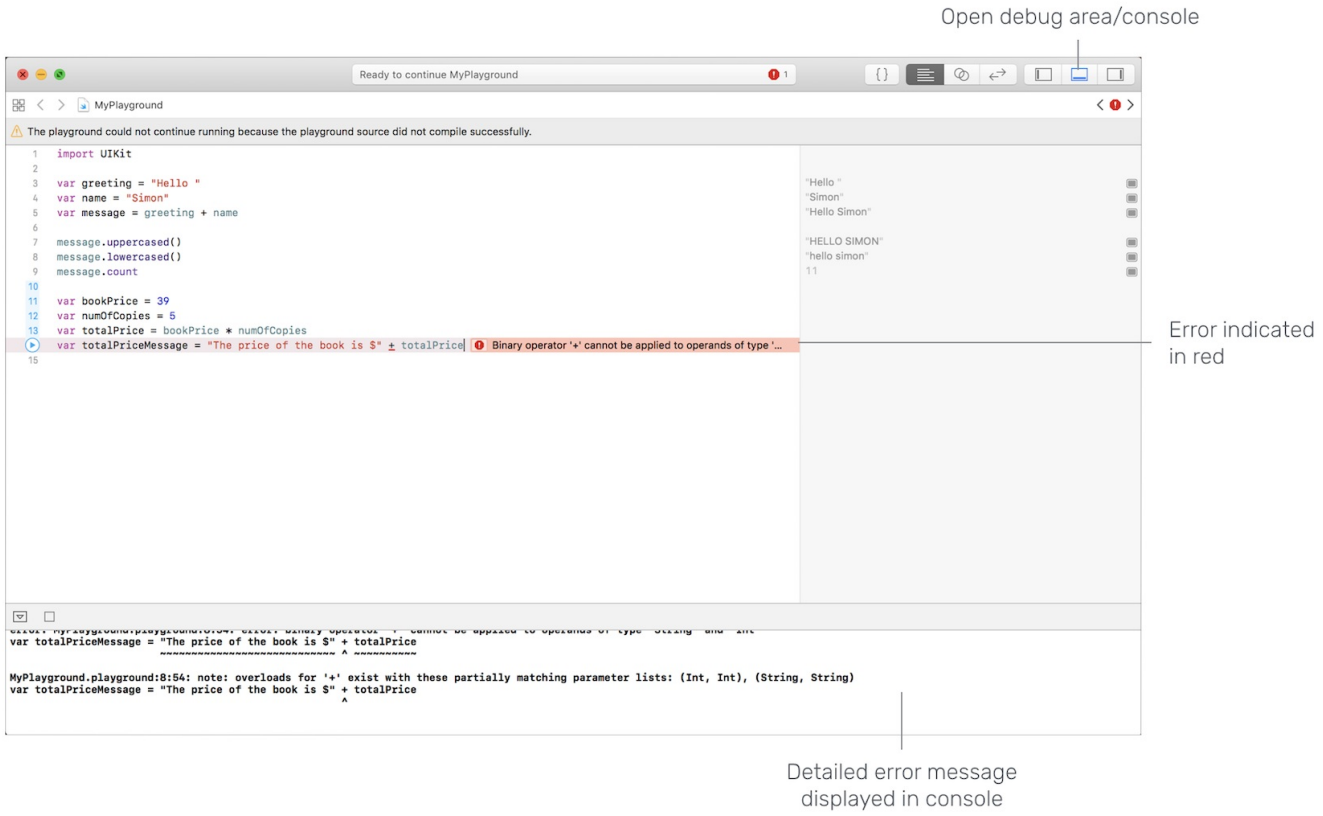


Figure 2-12. Debug area/Console

When Xcode finds an error in your code, the error is indicated by a red exclamation mark with a brief error message. Sometimes, Xcode shows you the possible fixes of the errors. But sometimes it does not.

To reveal the error details, you can refer to the debug area/console. If the console doesn't show up in your Playground, click the *debug area* button at the top-right corner.

Before I show you the solution, do you know why the code doesn't work? Give yourself a few minutes to think about that.

First, always remember that Swift is a type-safe language. This means each variable has a type that specifies what kind of values it can store. Let me ask you: what is the type of `totalPrice`? Recall what we learned earlier, Swift can determine the type of a variable by examining its value.

Since `39` is an integer, Swift determines that `bookPrice` has a type of `Int`, so does `numOfCopies` and `totalPrice`.

The error message displayed in the console mentioned that the operator `+` cannot concatenate a `String` variable with an `Int` variable. They must have the same type. In other words, you have to convert `totalPrice` from `Int` to `String` in order to make it work.

You can write the code like this by converting the integer to a string:

```
var totalPriceMessage = "The price of the book is $" + String(totalPrice)
```

There is an alternate way known as *String Interpolations* to do that. You can write the code like this to create the `totalPriceMessage` variable:

```
var totalPriceMessage = "The price of the book is $ \(totalPrice)"
```

String interpolations is the recommended way to build a string from multiple types. You wraps the variable for string conversion in parentheses, prefixed by a backslash.

After making the changes, re-run the code by hitting the *Play* button. The error should be fixed.

Control Flow Basics

> As far as the confidence goes, I think you will appreciate that it is not because you succeeded at everything you did, but because with the help of your friends, you were not afraid to fail. And if you did fail, you got up and tried again. And if you failed again, you got up and tried again. And if you failed again, it might be time to think about doing something else. But it was not just success, but not being afraid to fail that brought you to this point.
>
> - John Roberts, Chief Justice of the United States
>
> Source: http://time.com/4845150/chief-justice-john-roberts-commencement-speech-transcript/

Every day we make numerous decisions. Different decisions lead to different outcomes or actions. For example, you decide if you can wake up at 6AM tomorrow, you will cook yourself a big breakfast. Otherwise, you will go out for breakfast.

When writing programs, you use `if-then` and `if-then-else` statements to examine a condition and determine what to do next. If you turn the example above into code, it will be like this:

```
var timeYouWakeUp = 6

if timeYouWakeUp == 6 {
    print("Cook yourself a big breakfast!")
} else {
    print("Go out for breakfast")
}
```

You declare a variable `timeYouWakeUp` to store the time (in 24-hour) you wake up. You use `if` statement to evaluate a condition and determine what to next. The condition is placed after the `if` keyword. Here we compare the value of `timeYouWakeUp` to see if it equals `6`. The `==` operator is used for comparison.

If `timeYouWakeUp` does match `6`, the actions (or statements) enclosed in the curly brackets are executed. In the code, we simply use the `print` function to print a message to console.

Otherwise, the statements specified in the `else` block will be run to print another message.

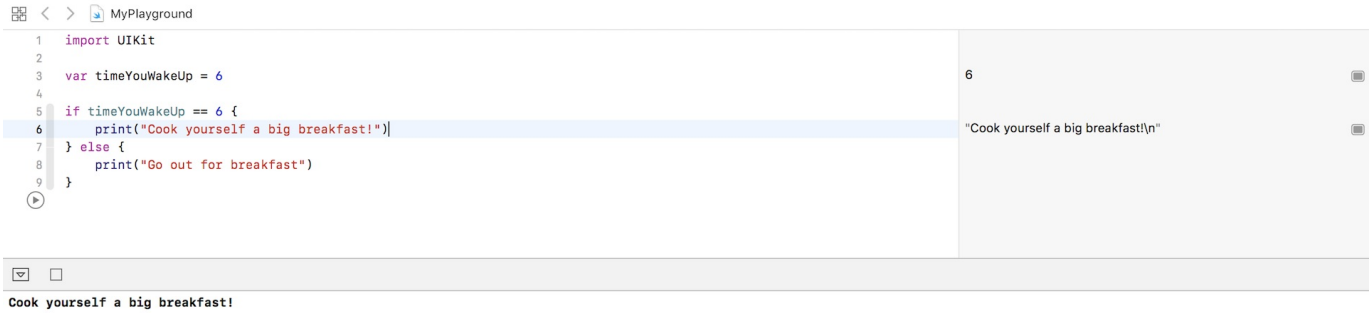


Figure 2-13. An example of If statement

In Playgrounds, you will see the message "Cook yourself a big breakfasts!" in the console because the value of `timeYouWakeUp` is initialized to `6`. You can try to change it to other values and see what you get.

Conditional logic is very common in programming. Imagine you are developing a login screen that requires users to input the username and password. The user can only be logged into the app with a valid account. In this case, you may use `if-else` statement to verify the username/password.

The `if-else` statement is one of the ways in Swift to control program flow. Swift also provides `switch` statements to control which code block to run. You can rewrite the example above using `switch`.

```
var timeYouWakeUp = 6

switch timeYouWakeUp {
case 6:
    print("Cook yourself a big breakfast!")
default:
    print("Go out for breakfast")
}
```

It will achieve the same result if `timeYouWakeup` is set to `6`. A `switch` statement considers a value (here, it is the value of `timeYouWakeUp`), and compare with the value specified in the `case`. The default case is indicated by the `default` keyword. It is very

much like the `else` block in the `if-else` statement. If the value being evaluated doesn't match any of the cases, the default case will be executed. So if you change the value of `timeYouWakeUp` to `8`, it will display the message "Go out for breakfast."

There is no universal rule about when to use `if-else` and when to use `switch`. Sometimes, we prefer one over the other just because of readability. Let's say, you typically get a bonus at the end of each year. Now you are making a plan for your next travel destination. Here is the plan:

- If you get a bonus of $10000 (or more), you will travel to Paris and London.
- If the bonus is between $5000 and $9999, you will travel to Tokyo.
- If the bonus is between $1000 and $4999, you will travel to Bangkok.
- If the bonus is less than $1000, you just stay home.

If you write the above plan in code, it looks like this:

```
var bonus = 5000

if bonus >= 10000 {
    print("I will travel to Paris and London!")
} else if bonus >= 5000 && bonus < 10000 {
    print("I will travel to Tokyo")
} else if bonus >= 1000 && bonus < 5000 {
    print("I will travel to Bangkok")
} else {
    print("Just stay home")
}
```

`>=` is a comparison operator, indicating "greater than or equal to". The first `if` condition checks if the value of `bonus` is greater than or equal to `10000`. To specify two simultaneous conditions, you use `&&` operator. The second `if` condition checks if the value is between 5000 and 9999. The rest of the code should be self explanatory.

You can rewrite the same piece of code using a `switch` statement like below:

```
var bonus = 5000

switch bonus {
case 10000...:
    print("I will travel to Paris and London!")
case 5000...9999:
    print("I will travel to Tokyo")
case 1000...4999:
    print("I will travel to Bangkok")
default:
    print("Just stay home")
}
```

Swift has a very handy range operator (`...`) that defines a range from lower bound to upper bound. For example, `5000...9999` defines a range that runs from 5000 to 9999. For the first case, `10000...` indicates a value that is great than 10000.

Both code blocks work exactly the same, but which way do you prefer? In this case, I prefer the `switch` statement which makes the code clearer. Anyway, even if you prefer to use `if` statement for the problem above, it is still correct. As you continue to explore Swift programming, you will understand when to use `if` or `switch`.

Understanding Arrays and Dictionaries

Now that you have a very basic knowledge of variables and control flow, let me introduce another programming concept that you will usually work with.

So far, the variables that we used can only store a single value. Referring to the variables in the earlier code snippet, `bonus`, `timeYouWakeUp` and `totalPriceMessage` can hold a single value, regardless of the variable type.

Let's consider this example. Imagine you are creating a bookshelf application that organizes your book collection. In your program, you will probably have some variables holding your book titles:

```
var book1 = "Tools of Titans"
var book2 = "Rework"
var book3 = "Your Move"
```

Instead of storing a single value in each variable, is there any way to store more than one value in it?

Swift provides a collection type known as **Array** that lets you store multiple values in a variable. With an array, you can store your book titles like this:

```
var bookCollection = ["Tool of Titans", "Rework", "Your Move"]
```

You can also initialize an array by writing a list of values, separated by commas, surrounded by a pair of square brackets. Again, since Swift is a type-safe language, all values should be of the same type (e.g. String).

Accessing the values of an array may look weird to you if you just begin to learn programming. In Swift, you use the subscript syntax to access the array elements. The index of the first item is zero. Therefore, to refer to the first item of an array, you write the code like this:

```
bookCollection[0]
```

If you type the code above in Playgrounds and hit *Play*, you should see "Tool of Titans" shown in the result pane.

When you declare an array as `var`, you can modify its elements. For example, you can add a new item to the array by calling the built-in method `append` like this:

```
bookCollection.append("Authority")
```

Now the array has 4 items. How can you reveal the total number of items of an array? Use the built-in `count` property:

```
bookCollection.count
```

Let me ask you, how can you print the value of each item of the array to console?

Don't look at the solution yet.

Try to think.

Okay, probably you will write the code like this:

```
print(bookCollection[0])
print(bookCollection[1])
print(bookCollection[2])
print(bookCollection[3])
```

It works. But there is a better way to do it. As you can see, the code above is repetitive. If the array has 100 items, it will be quite tedious to type a hundred lines of code. In Swift, you use a `for-in` loop to execute a task (or a block of code) for a specific number of time. For example, you can simplify the code above like this:

```
for index in 0...3 {
    print(bookCollection[index])
}
```

You specify the range of number (`0...3`) to iterate over. In this case, the block of code in the `for` loop is executed for 4 times. The value of `index` will be changed accordingly. When the `for` loop is first started to execute, the value of `index` is set to `0` and it will print `bookCollection[0]` . After the statement is executed, the value of `index` will be updated to `1` , and it will print `bookCollection[1]` . This process continues until the end of the range (i.e. `3`) is reached.

Now I have a question for you. What if there are 10 items in the array? You probably change the range from `0...3` to `0...9` . How about later the total number of items are increased to 100? Then you will change to range to `0...100` .

Is there a generic way to do that, instead of updating the code every time the total number of items changes?

Do you notice a pattern for these ranges: `0...3`, `0...9` and `0...100`?

The upper bound of the range equals to the total number of items minus 1. You can actually rewrite the code like this:

```
for index in 0...bookCollection.count - 1 {
    print(bookCollection[index])
}
```

Now regardless of the number of array items, this code snippet works.

Swift's `for-in` loop offers an alternate way to iterate over an array. The sample code snippet can be rewritten as follows:

```
for book in bookCollection {
    print(book)
}
```

When the array (i.e. `bookCollection`) is iterated, the item of each iteration will be set to the `book` constant. When the loop is first started, the first item of `bookCollection` is set to `book`. In the next iteration, the second item of the array will be assigned to `book`. The process keeps going until the last item of the array is reached.

Now that I believe you understand how `for-in` loop works and how you can repeat a task using loop, let's talk about another common collection type called **dictionary**.

A dictionary is similar to an array that allows you to store multiple values in a variable/constant. The main difference is that each value in a dictionary is associated with a key. Instead of identifying an item using an index, you can access the item using a unique key.

Let me continue to use the book collection as an example. Each book has a unique ISBN (short for International Standard Book Number). If you want to index each book in the collection by its ISBN, you can declare and initialize a dictionary like this:

```
var bookCollectionDict = ["1328683788": "Tool of Titans", "0307463745": "Rework",
"1612060919": "Authority"]
```

The syntax is very similar to an array initialization. All values are surrounded by a pair of square brackets. The key and value pair is separated by a colon (`:`). In the sample code, the key is the ISBN. Each book title is associated with a unique ISBN.

So how can you access a particular item? Again, it is very similar to array. However, instead of using a numeric index, you use the unique key. Here is an example:

```
bookCollectionDict["0307463745"]
```

This gives you the value: *Tool of Titans*. To iterate over all items of the dictionary, you can also use the `for-in` loop:

```
for (key, value) in bookCollectionDict {
    print("ISBN: \(key)")
    print("Title: \(value)")
}
```

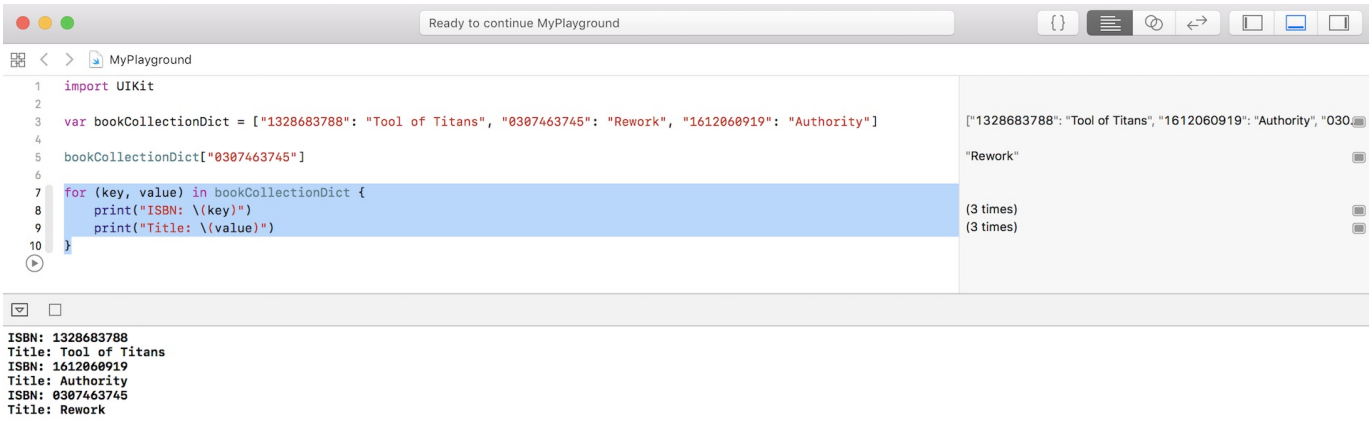


Figure 2-14. Iterate over a dictionary

As you may reveal from the message in the console, the order of the items doesn't follow the order in the initialization. Unlike an array, this is a characteristic of dictionaries that stores items in an unordered fashion.

You may still wonder when you will need to use a dictionary when building an app. Let's take a look another example. There is a reason why it is known as a dictionary. Think about how you use a dictionary, you look up a word in a dictionary, and it gives you the word's meaning. In this case, the word is the key, and its meaning is the associated value.

Before you move onto the next section, let's have a very simple exercise to create an Emoji dictionary, which stores the meaning of emoji characters. To keep things simple, this dictionary has the meaning of the following emoji characters:

- 👻 - Ghost
- 💩 - Poop
- 😤 - Angry
- 😱 - Scream
- 👾 - Alien monster

Do you know how to implement the emoji dictionary using the Dictionary type? Below is the code skeleton for the emoji dictionary. Please fill in the missing code to make it work:

```
var emojiDict = // Fill in the code for initializing the dictionary //

var wordToLookup = // Fill in the Ghost emoji //
var meaning = // Fill in the code for accessing the value //

wordToLookup = // Fill in the Angry emoji //
meaning = // Fill in the code for accessing the value //
```

To type an emoji character on Mac, press control-command+space.

Are you able to complete the exercise?

Let's take a look at the solution and the output in figure 2-16.

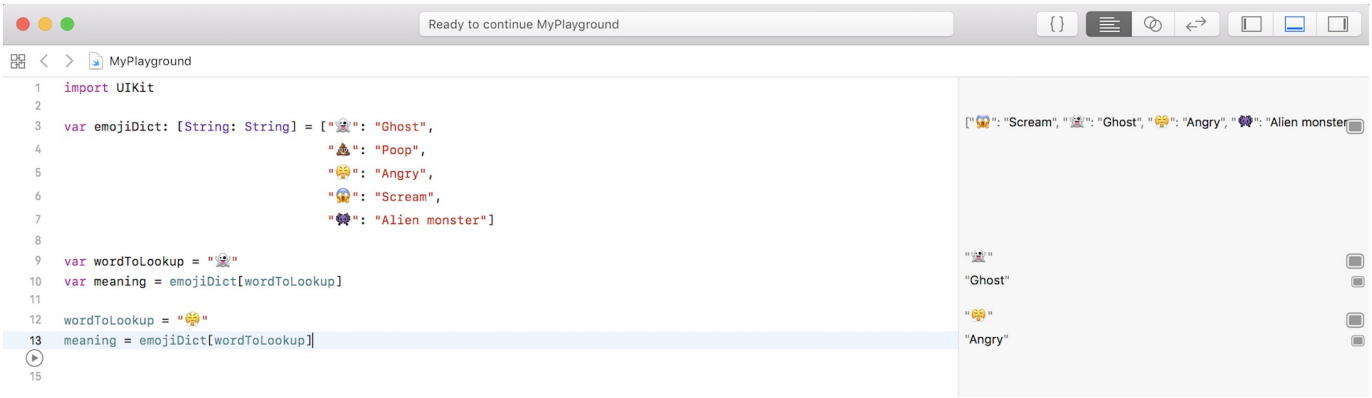


Figure 2-16. Solution to the emoji dictionary exercise

I believe you can figure out the solution by yourself.

Now let's add a couple of lines of code to print the `meaning` variable to console.

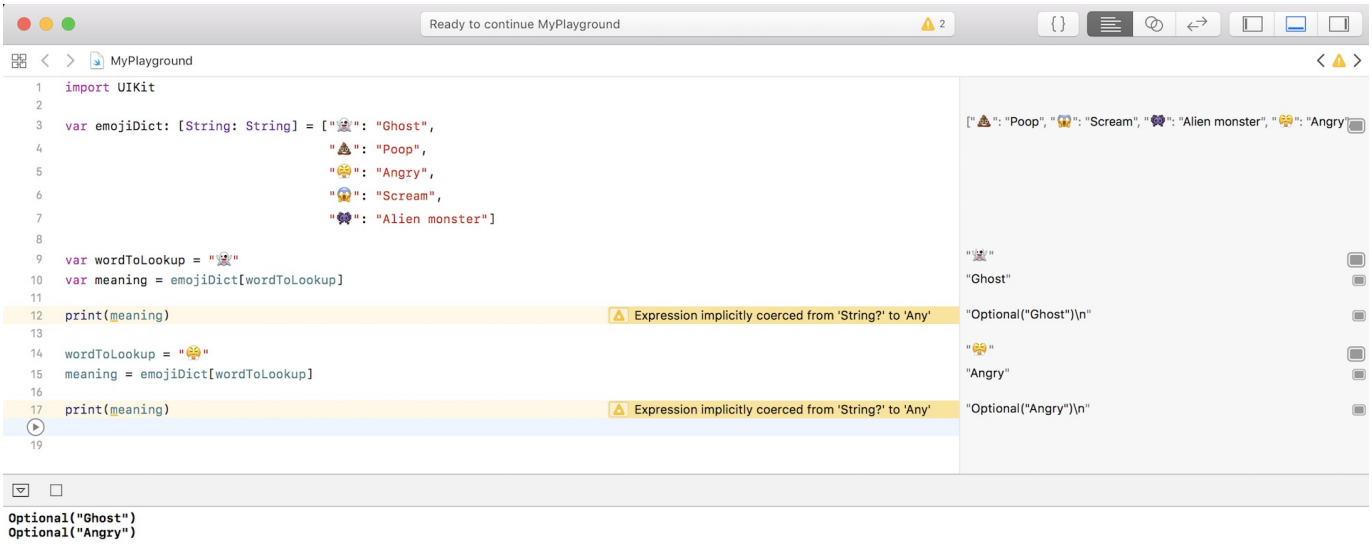


Figure 2-16. Print the meaning variable

You will notice two things:

1. Xcode indicates both print statements have some issues.
2. The output in the console area looks a bit different from other output we went through before. The result is correct, but what does *Optional* mean?

Note: In Xcode, warnings are indicated in yellow. One main difference between warnings and errors is that your program can still be run even if it has some warnings. As the name suggests, a warning gives you an advanced notice of some issues. You better fix the warnings to avoid any potential issues.

Both issues are related to a new concept in Swift called **Optionals**. Even if you have some programming background, this concept may be new to you.

> I hope you enjoy what you've learned so far. But if you are feeling stuck, take a break here. Grab a coffee and relax. Or you can even skip the rest of the chapter and try to build your first app in the next chapter. You can always revisit this chapter anytime.

Understanding Optionals

Do you have such experience? You open an app, tap a few buttons, and it suddenly crashes. I am quite sure you have experienced that.

Why does an app crash happen? One common cause is that the app tries to access a variable that has no value at runtime. Then the unexpected happens.

So is there a way to prevent the crashes?

Different programming languages have different strategies to encourage programmers to write good code or less-error-prone code. The introduction of Optionals is Swift's way to help programmers write better code, thus prevent app crashes.

Some developers struggle to understand the concept of Optionals. The fundamental idea is actually quite simple. Before accessing a variable that may have no value, Swift encourages you to verify it first. You have to make sure it has a value before any further processing. Thus, this can avoid app crashes.

Up till now, all the variables or constants we worked with have an initial value. This is a must in Swift. A non-optional variable gurantees to have a value. If you try to declare a variable without a value, you get an error. You can give it a try in Playgrounds and see what happens.

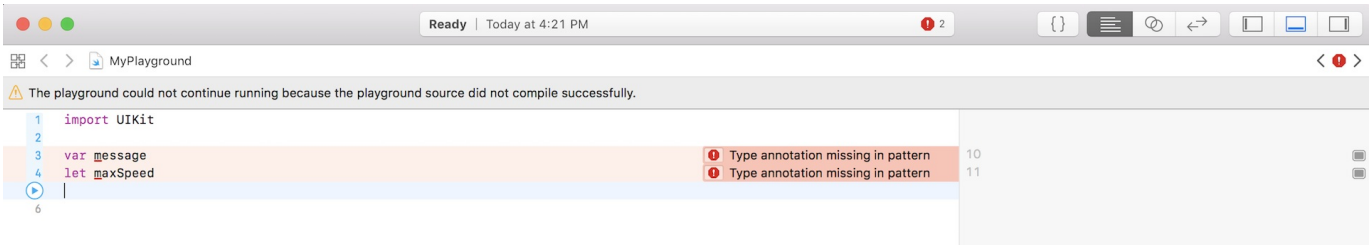


Figure 2-17. Declaring a variable/constant without an initial value

In some situations, you have to declare a variable without an initial value. Imagine, you are developing an app with a registration form. Not all fields in the form is mandatory, some fields (e.g. job title) are optional. In this case, the variables of those optional fields may have no values.

Technically, optional is just a type in Swift. This type indicates that the variable can have a value or no value. To declare a variable as an optional, you indicate it by appending a question mark (`?`). Here is an example:

```
var jobTitle: String?
```

You declare a variable named `jobTitle` which is of type `String` and it is also an optional. If you place the code above in Playgrounds, it will not show an error because Xcode knows that `jobTitle` can have no value.

Unlike a non-optional variable that the compiler can deduce the type from its initial value, you have to explicitly specify the type of an optional variable (e.g. `String` , `Int`).

If you have followed my instruction to enter the code in Playgrounds (and hit *Play*), you may notice that `nil` is displayed in the resulting pane. For any optional variable with no value, a special value called `nil` is assigned to it.

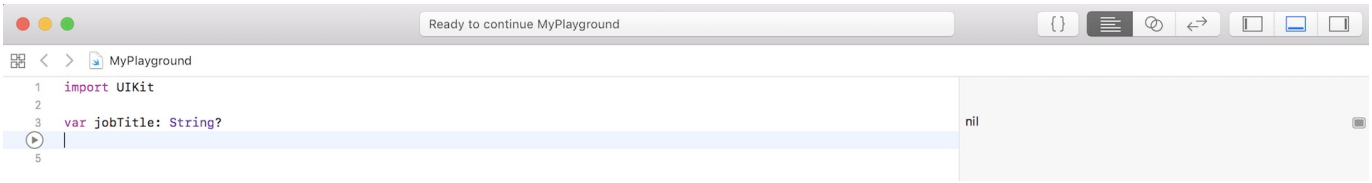


Figure 2-18. A special value "nil" is assigned to an optional variable with no value

In other words, `nil` indicates the variable does not have a value.

If you have to assign a value to an optional variable, you can just do that as usual like this:

```
jobTitle = "iOS Developer"
```

Now that you should have some knowledge of optionals, but how can it help us write better code?

Try to key in the code as displayed in figure 2-19.

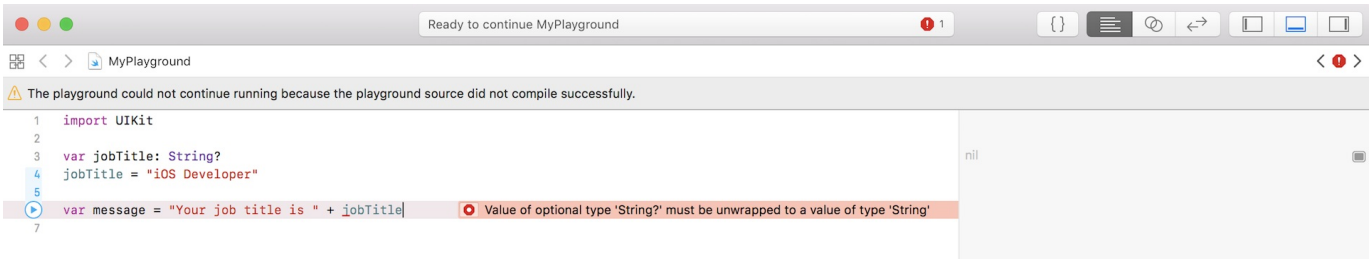


Figure 2-19. An error is shown when you access an optional variable

As soon as you complete typing the following line of code, Xcode indicates it by giving an error message.

```
var message = "Your job title is " + jobTitle
```

Here `jobTitle` was declared as an optional variable. Xcode told you that there was a potential error for that line of code because `jobTitle` might have no value. You have to do some checkings before using the optional variable.

This is how optionals can prevent you from writing bad code. Whenever you need to access an optional variable, Xcode forces you to perform verification to find out whether the optional has a value.

Forced Unwrapping

So how can you perform such verification and unwrap the value of the optional variable? Swift offers a couple of ways to do that.

First, it is known as *if statements and forced unwrapping*. In simple words, you use a `if` statement to check if the optional variable has a value by comparing it against `nil`. If the optional does have a value, you unwrap its value for further processing.

This is how it looks like in code:

```
if jobTitle != nil {
    var message = "Your job title is " + jobTitle!
}
```

The `!=` operator means "not equal". So if `jobTitle` does not equal to `nil`, it must have a value. You can then execute the statements in the body of `if` statement. When you need to access the value of `jobTitle`, you add an exclamation mark (`!`) to the end of the optional variable. This exclamation mark is a special indicator, telling Xcode that you ensure the optional variable has a value, and it is safe to use it.

Optional Binding

Forced unwrapping is one way to access the underlying value of an optional variable. The other way is called *optional binding*, and it is the recommended way to work with optionals. At least, you do not need to use `!`.

If optional binding is used, the same code snippet can be rewritten like this:

```
if let jobTitleWithValue = jobTitle {
    var message = "Your job title is " + jobTitleWithValue
}
```

You use `if let` to find out whether `jobTitle` has a value. If yes, the value is assigned to the temporary constant `jobTitleWithValue`. In the code block, you can use `jobTitleWithValue` as usual. As you can see, there is no need to add the `!` suffix.

Do you have to give a new name for the temporary constant?

No, you can actually use the same name like this:

```
if let jobTitle = jobTitle {
    var message = "Your job title is " + jobTitle
}
```

Note: Even though the names are the same, there are actually two variables in the code above. jobTitle in black is the optional variable, while jobTitle in blue is the temporary constant to be assigned with the optional value.

This is pretty much about Swift's optionals. Are you confused by various `?` and `!` symbols? I hope you do not. In case you are struggled to understand optionals, post your questions to our Facebook group (https://facebook.com/groups/appcoda).

Okay, do you still remember the warning displayed in figure 2-16? When you tried to print the value of `meaning`, Xcode gave you some warnings. In console, even though the value was printed, it was prefixed by "Optional".

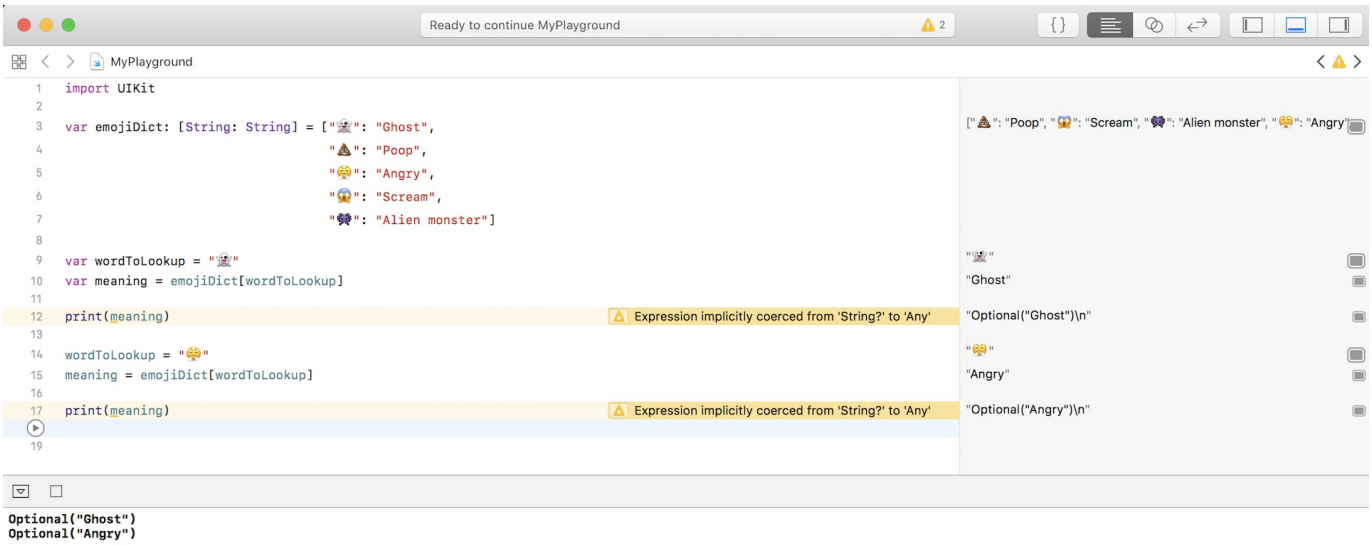


Figure 2-20. Same as figure 2-16 showing you the warning messages

Now can you figure out why? Why the `meaning` variable is an optional? How can you modify the code to remove the warning messages?

Again, don't look at the solution yet. Think.

If you look into the code, `meaning` is actually an optional. It is because the dictionary may not have a value for the given key. For example, if you write this code in Playgrounds:

```
meaning = emojiDict["😍"]
```

The `meaning` variable will be assigned with `nil` because `emojiDict` doesn't have a value for the key 😍.

Therefore, whenever we need to access the value of `meaning`, we have to check if it has a value. To avoid the warning messages, we can use optional binding to test the existence of the value.

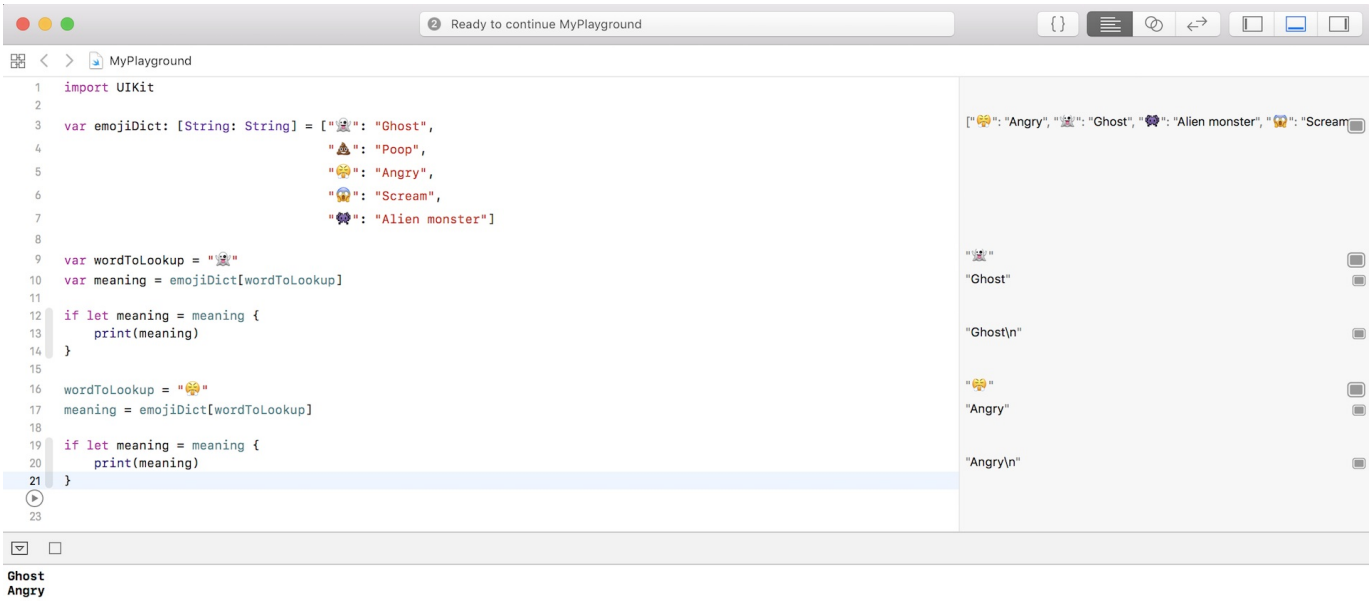


Figure 2-21. Use optional binding to check if meaning has a value and unwrap it accordingly

After you made the changes, the warning messages disappears. You also notice that the values displayed in the console is no longer prefixed with Optional.

Playing around with UI

Before I close this chapter, let's have some fun to create some UI elements. What we are going to do is to display the emoji icon and its corresponding meaning in a view (see figure 2-22) instead of just printing out the messages in console.

Figure 2-22. Displaying emoji in a view

As I mentioned at the very beginning of the chapter, other than learning Swift, you will need to familiarize yourself with the iOS SDK in order to build apps.

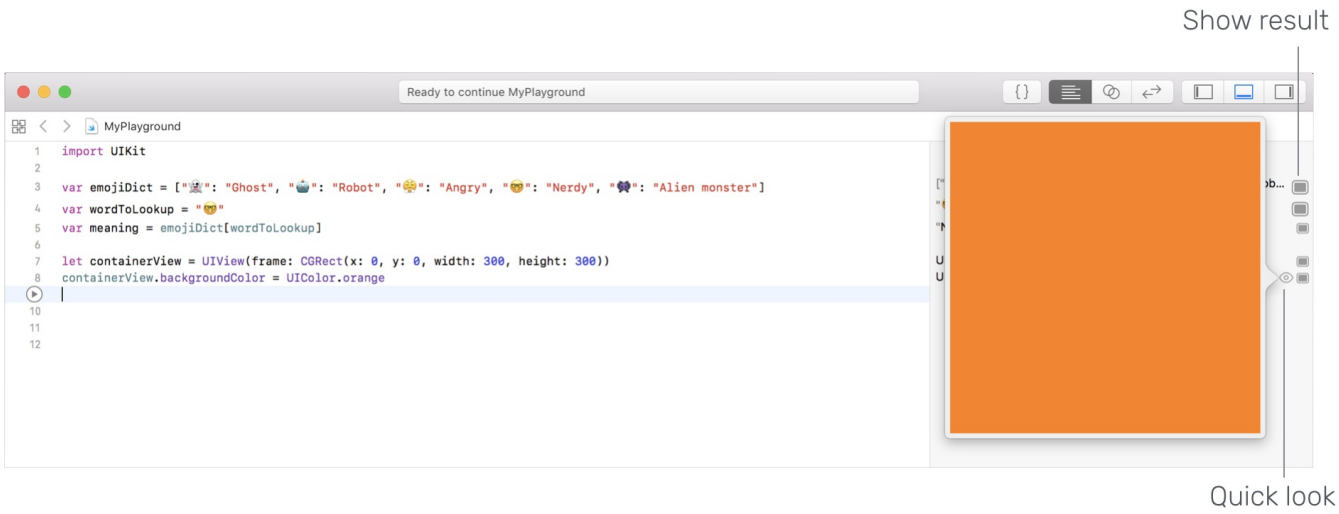
You can also use Playgrounds to explore some of the UI controls, provided by the iOS SDK. A view is the basic UI element in iOS. You can think of it as a rectangular area that is used for showing content. Now key in the following code in your Playground and then hit *Play* to run the code:

```
var emojiDict = ["👻": "Ghost", "🤖": "Robot", "😤": "Angry", "🤓": "Nerdy", "👾": "Alien monster"]
var wordToLookup = "🤓"
var meaning = emojiDict[wordToLookup]

let containerView = UIView(frame: CGRect(x: 0, y: 0, width: 300, height: 300))
containerView.backgroundColor = UIColor.orange
```

You should be very familiar with the first three lines. Let's focus on the last two lines of code. To create a view, you can instantiate a `UIView` object with its size. Here we specify both width and height to `300` points. `x` and `y` indicates the position. The last line of code changes the view's background colour to *orange*.

To preview a UI element in Playgrounds, you can click the Quick Look or Show Result icon. The Quick Look feature displays the label as a pop-over. If you use the Show Result feature, Playground displays the view inline, right below your code.



Now the view (i.e. `containerView`) is empty. Wouldn't it be great if you can add the emoji to the view? Continue to type the following lines and click the Show Result icon.

```
let emojiLabel = UILabel(frame: CGRect(x: 95, y: 20, width: 150, height: 150))
emojiLabel.text = wordToLookup
emojiLabel.font = UIFont.systemFont(ofSize: 100.0)

containerView.addSubview(emojiLabel)
```

An emoji is just a character. In iOS, you can use a label to display text. To create a label, you instantiate a `UILabel` object with your preferred size. Here the label is 150x150 points. The `text` property contains the text to be displayed in the label. To make the text bigger, you can alter the font property to use a larger font size. Lastly, to display the label in `containerView`, you add the label to the view by calling `addSubview`.

Continue to type the following code to add another label:

```
let meaningLabel = UILabel(frame: CGRect(x: 110, y: 100, width: 150, height: 150))
meaningLabel.text = meaning
meaningLabel.font = UIFont.systemFont(ofSize: 30.0)
meaningLabel.textColor = UIColor.white

containerView.addSubview(meaningLabel)
```

After running the code, click the Show Result button to view the result like this:

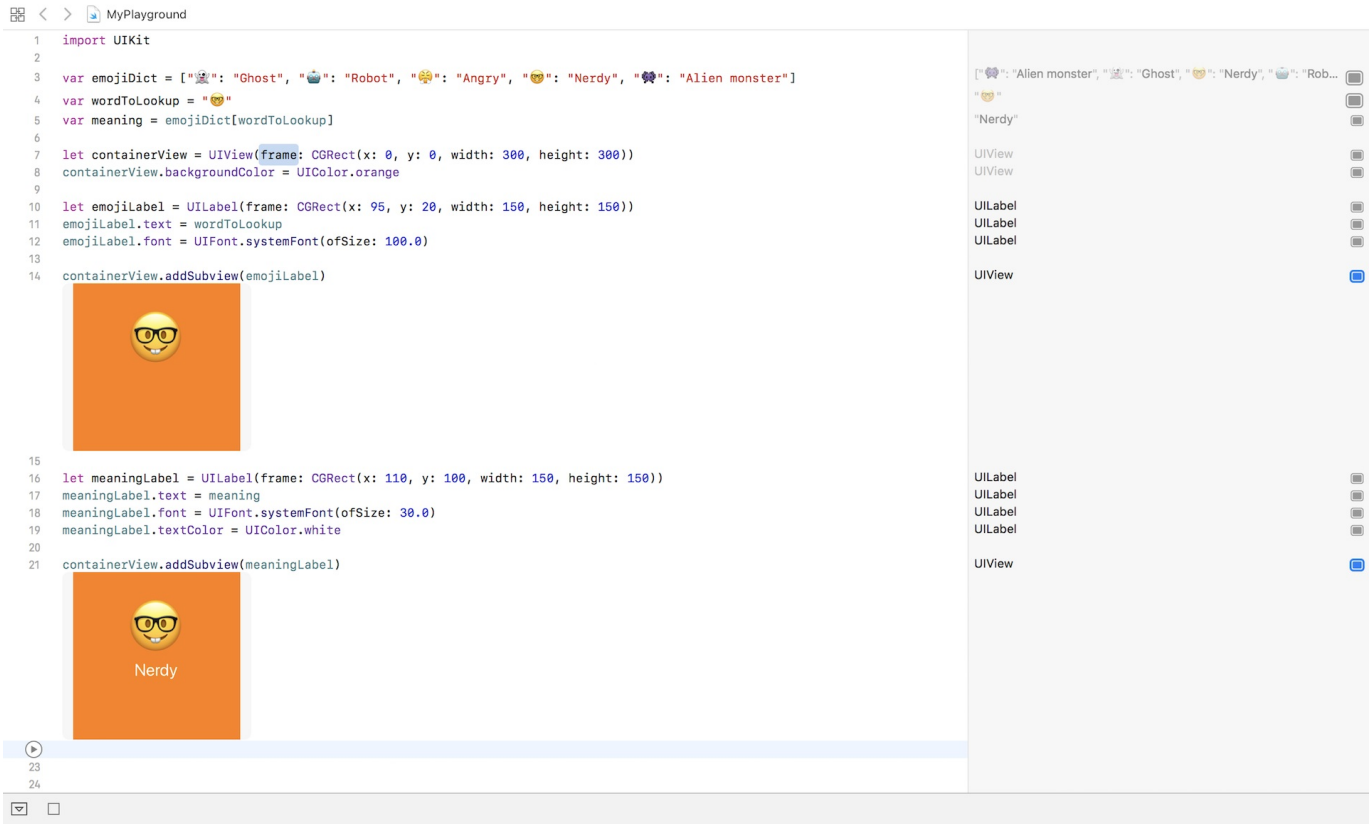


Figure 2-24. Nerdy emoji icon in the view

This is the power of iOS SDK. It comes with tons of pre-built elements and allows developers to customize them with few lines of code.

Don't get me wrong. For the rest of the book, you do not need to write code to create the user interface. Xcode provides a feature called Interface Builder that lets you design UI using drag-and-drop. We'll go through it in the next chapter.

Now you've got a taste of Swift. What do you think? Love it? I hope you find Swift a lot easier to learn and code. Most importantly, I hope it don't scare you away from learning app development.

What's Next

What's coming up is that I will teach to build your first app. You can now move onto the next chapter. However, if you want to learn more about the Swift programming language, I would recommend you to check out Apple's official Swift Programming Language guide (https://docs.swift.org/swift-book/). You will learn the language syntax, understand functions, optionals, and many more.

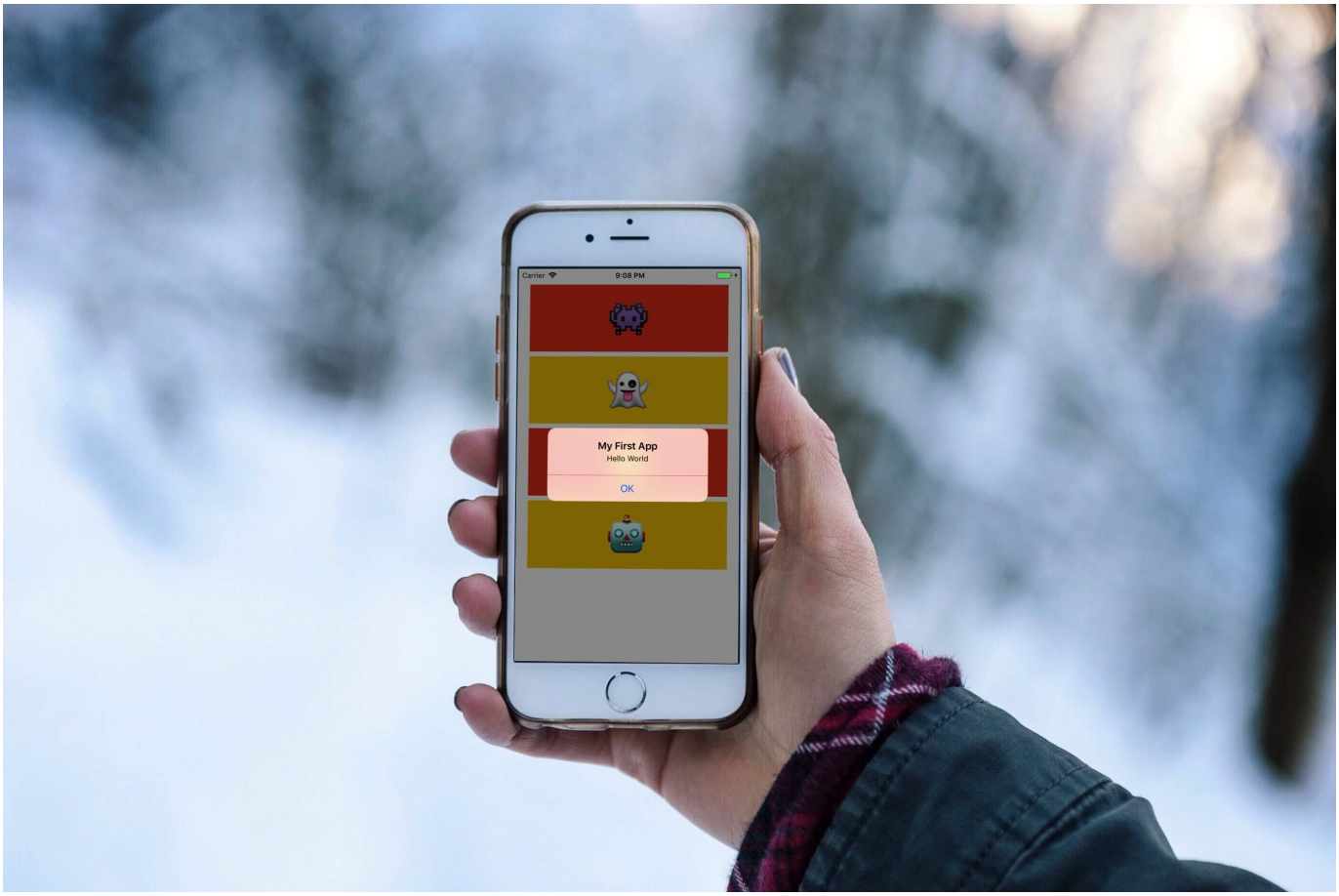
But it's not a must.

If you can't wait to build your first app, flip it over to the next chapter, and read the guide later. You can learn more about Swift along the way.

For reference, you can download the sample Playground file from http://www.appcoda.com/resources/swift4/ch2-playgrounds.zip.

Love this chapter and want to continue to learn iOS app development? Please get the full copy of the book here. You will also be able to access the full source code of the project.

Chapter 3
Hello World! Build Your First App in Swift



> Learn by doing. Theory is nice but nothing replaces actual experience.
>
> – Tony Hsieh

By now you should have installed Xcode and some understandings of Swift. If you have skipped the first two chapters, please stop here and go back to read them. You need to have Xcode 10 installed in order to work on all exercises in this book.

You may have heard of the "Hello World" program if you have read any programming book before. Hello World is a program for the first-time programmer to create. It's a very simple program that outputs "Hello, World" on the screen of a device. It's a tradition in the programming world. Let's follow the programming tradition and create a "Hello World" app using Xcode.

Despite its simplicity, the "Hello World" program serves a few purposes:

- It gives you an overview of the syntax and structure of Swift, the new programming language of iOS.
- It also gives you a basic introduction to the Xcode environment. You'll learn how to create an Xcode project and lay out your user interface using Interface Builder. Even if you've used Xcode before, you'll learn what's new in the latest version of Xcode.
- You'll learn how to compile a program, build the app and test it using the built-in simulator.
- Lastly, it makes you think programming is not difficult. I don't want to scare you away from learning programming. It'll be fun.

Your First App

Your first app, as displayed in figure 3-1, is very simple and just shows a "Hello World" button. When a user taps the button, the app shows a welcome message. That's it. Extremely simple but it helps you kick off your iOS programming journey.

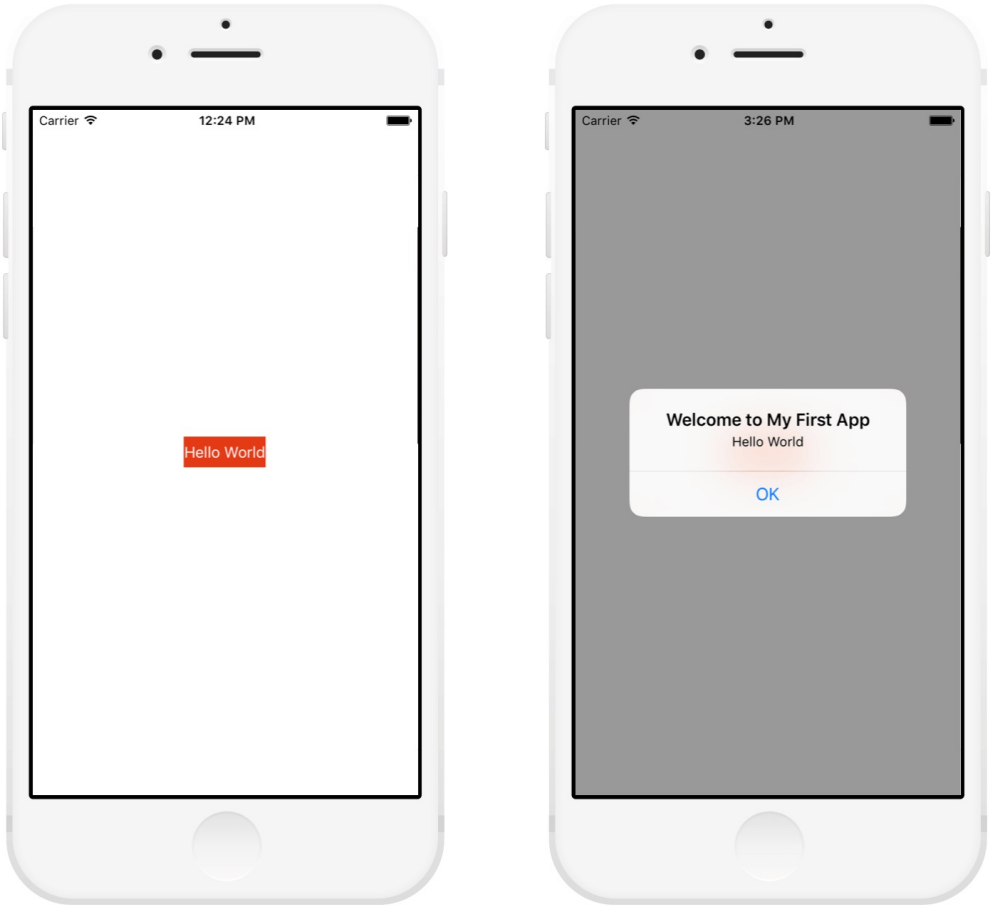


Figure 3-1. HelloWorld App

But this is just a start. After going through the challenges in this chapter, you will keep improving your first app to make it more interesting. To give you a sneak peak, figure 3-2 is your final deliverable.

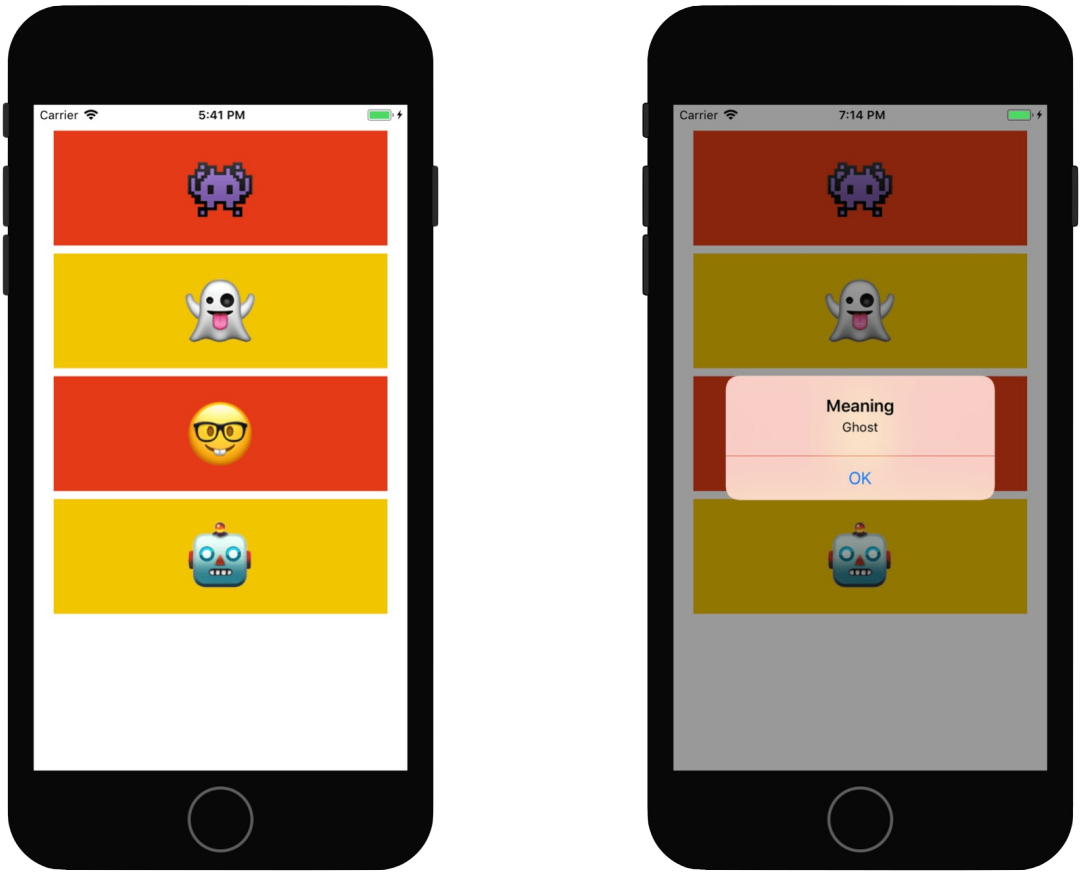


Figure 3-2. Your final deliverable

When you're building your first app, I want you to keep one thing in mind: *forget about the code and keep doing*. Even though you now have some basic knowledge of Swift, I'm quite sure that you will find some of the code difficult to understand. No worries. Just focus on the exercises, and get yourself familarized with the Xcode environment. We will talk about the code in the next chapter.

Let's Jump Right Into the Project

First, open Xcode. Once launched, Xcode displays a welcome dialog. From here, choose "Create a new Xcode project" to start a new project.

Figure 3-3. Xcode - Welcome Dialog

Xcode shows various project templates for selection. Each template serves different purposes and help you easier to get started with the development of a particular type of application. Say, if you want to develop a sticker pack for the Messages app, you will use the *Sticker Pack App* template. However, in most case, the *Single View Application* template is good enough for creating an iOS app. So, choose iOS > Single View Application and click *Next*.

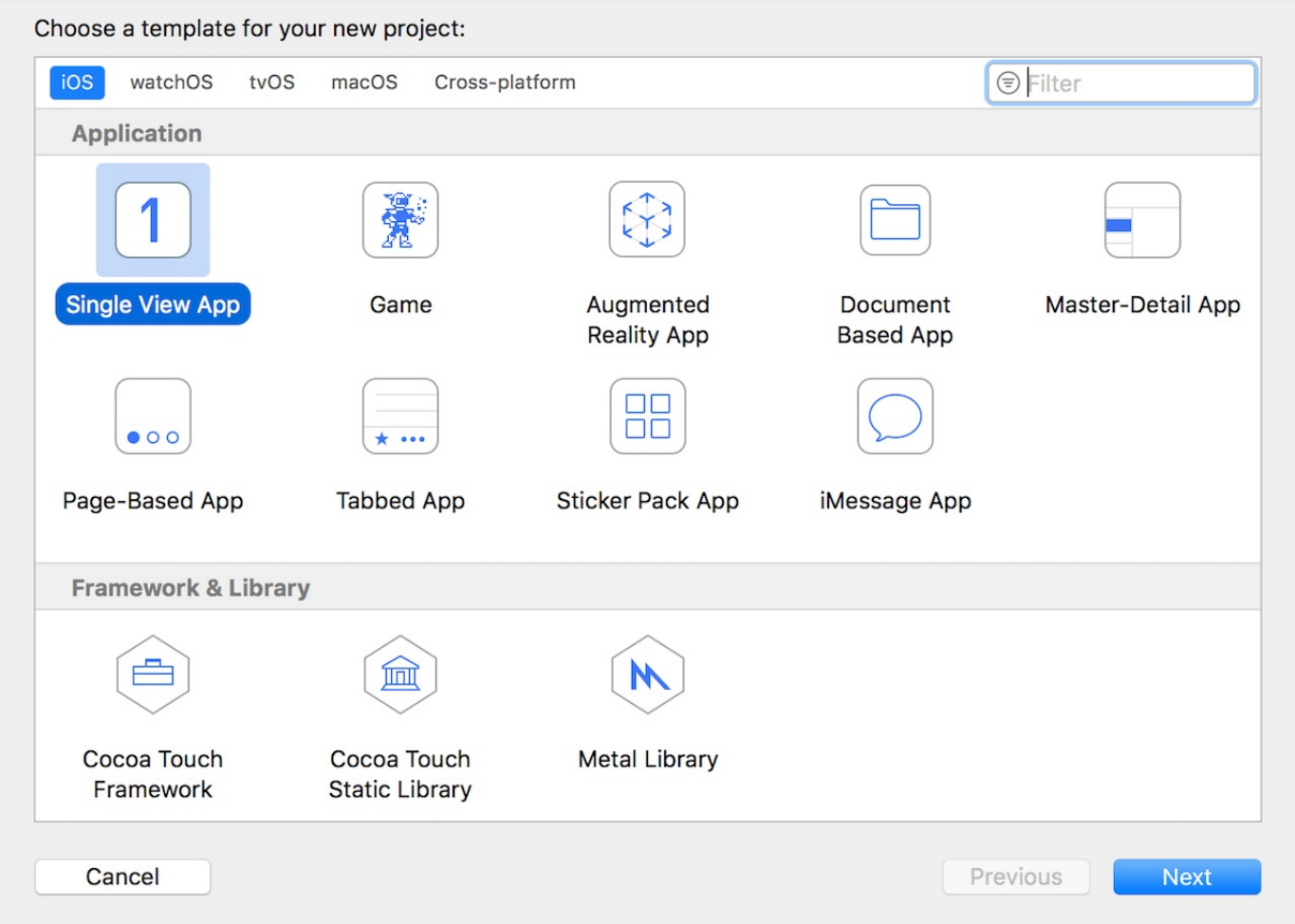


Figure 3-4. Xcode Project Template Selection

This brings you to the next screen to fill in all the necessary options for your project.

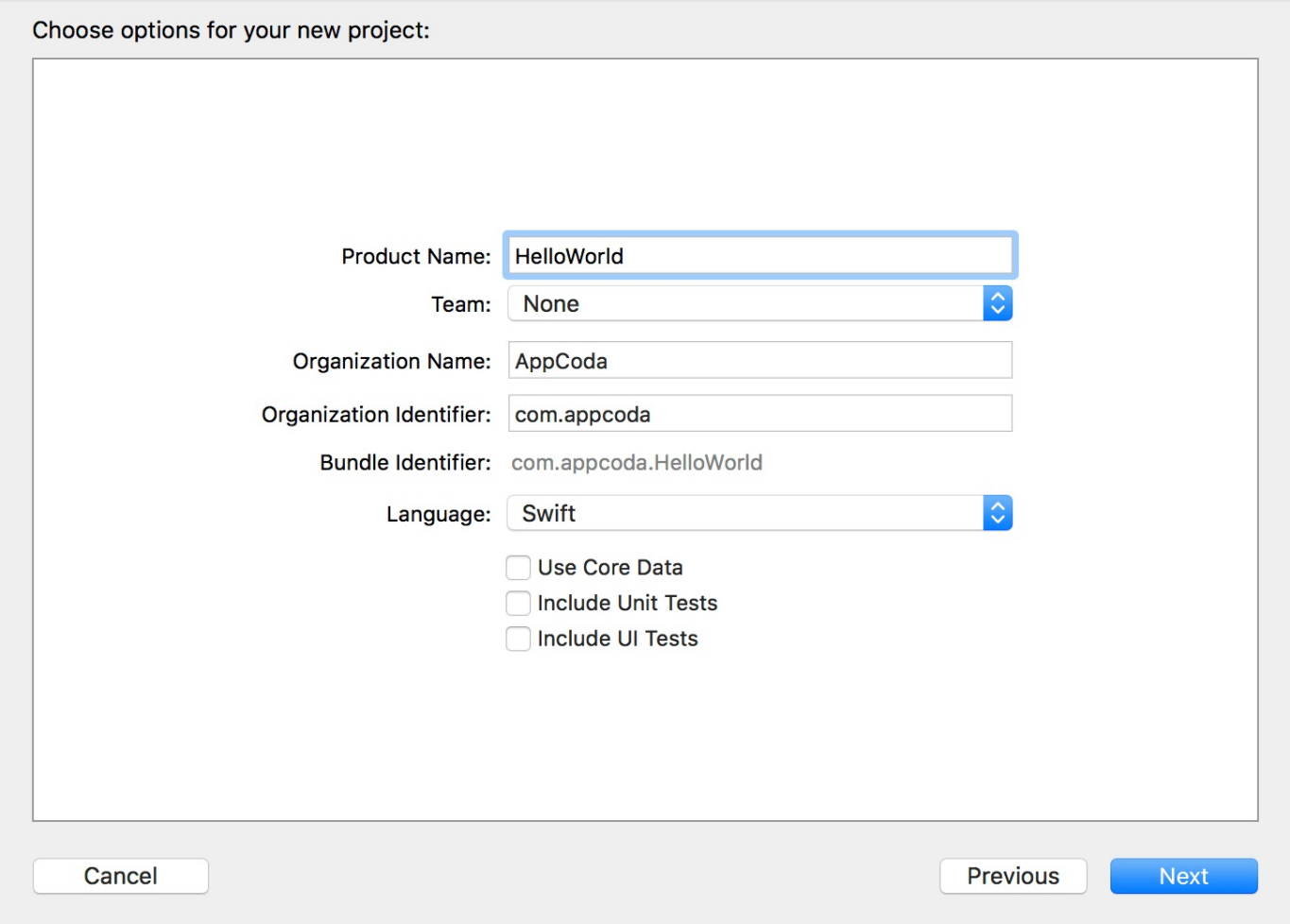


Figure 3-5. Options for your Hello World project

You can simply fill in the options as follows:

- **Product Name: HelloWorld** – This is the name of your app.
- **Team:** Just leave it as it is. You do not set the team yet. For your first app, just skip this step.
- **Organization Name: AppCoda** – It's the name of your organization. If you are not building the app for your organization, use your name as the organization name.
- **Organization Identifier: com.appcoda** – It's actually the domain name written the other way round. If you have a domain, you can use your own domain name. Otherwise, you may use "com.". For instance, your name is Pikachi. Fill in the

organization identifier as "com.pikachi".

- **Bundle Identifier: com.appcoda.HelloWorld** - It's a unique identifier of your app, which is used during app submission. You do not need to fill in this option. Xcode automatically generates it for you.
- **Language: Swift** – Xcode supports both Objective-C and Swift for app development. As this book is about Swift, we'll use Swift to develop the project.
- **Use Core Data: [unchecked]** – Do not select this option. You do not need Core Data for this simple project. We'll explain Core Data in later chapters.
- **Include Unit Tests: [unchecked]** – Do not select this option. You do not need unit tests for this simple project.
- **Include UI Tests: [unchecked]** – Do not select this option. You do not need UI tests for this simple project.

Click "Next" to continue. Xcode then asks you where to save the "HelloWorld" project. Pick any folder (e.g. Desktop) on your Mac. You may notice there is an option for source control. Just deselect it. We do not need to use the option in this book. Click "Create" to continue.

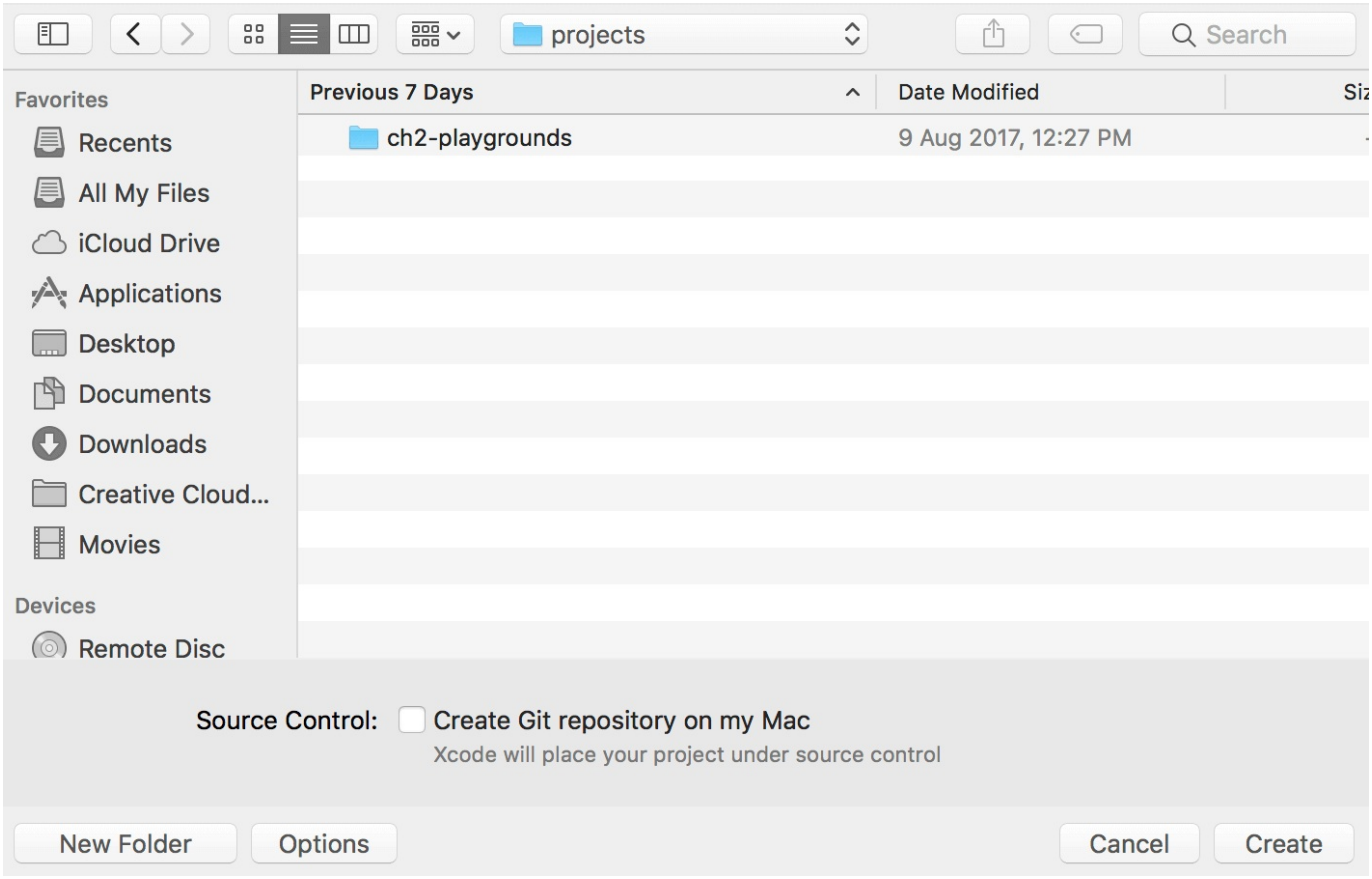
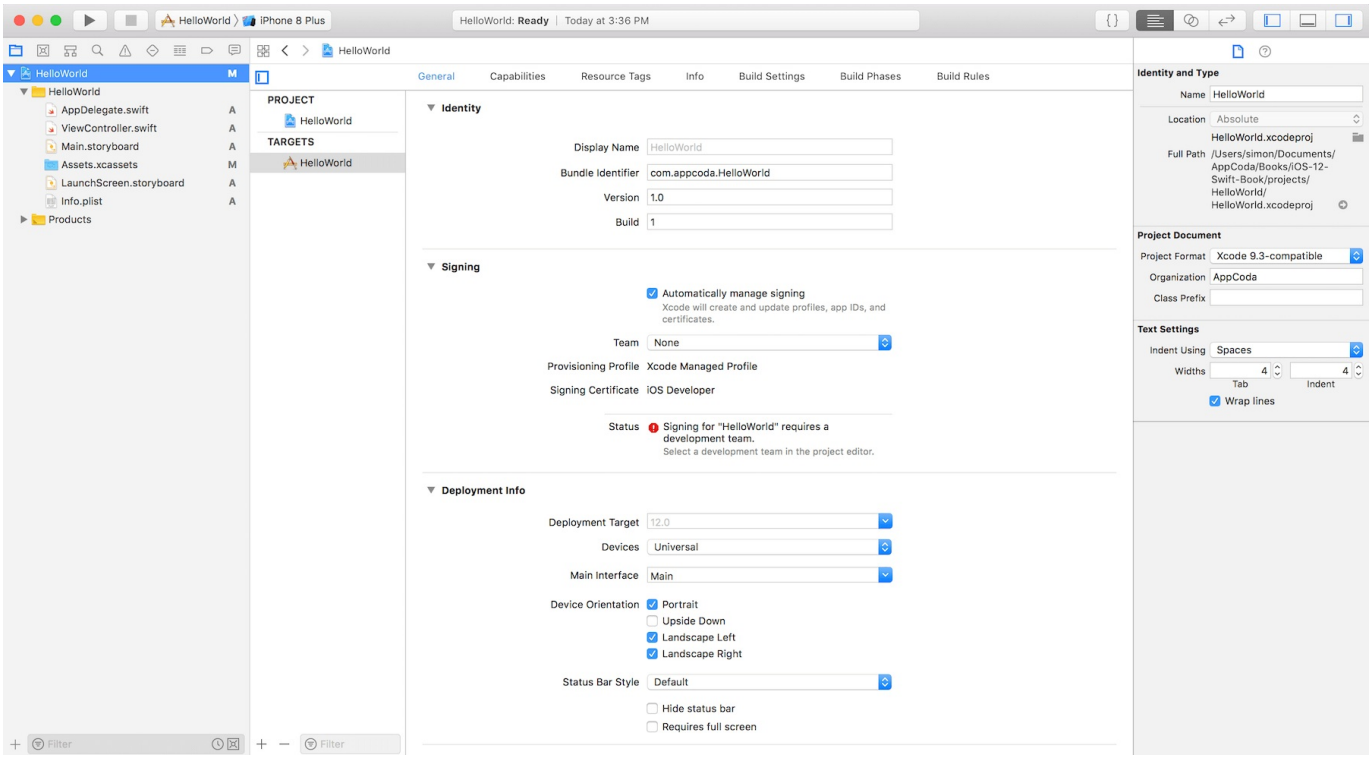


Figure 3-6. Choose a folder and save your project

After you confirm, Xcode automatically creates the "Hello World" project. The screen will look like the screenshot shown in figure 3-7.



Familiarize Yourself with Xcode Workspace

Before we start to implement the Hello World app, let's take a few minutes to have a quick look at the Xcode workspace environment. In the left pane is the project navigator. You can find all your project files in this area. The center part of the workspace is the editor area. You do all the editing stuff here (such as editing the project setting, source code file, user interface) in this area.

Depending on the type of file, Xcode shows you different interfaces in the editor area. For instance, if you select `ViewController.swift` in the project navigator, Xcode displays the source code in the center area (see figure 3-8).

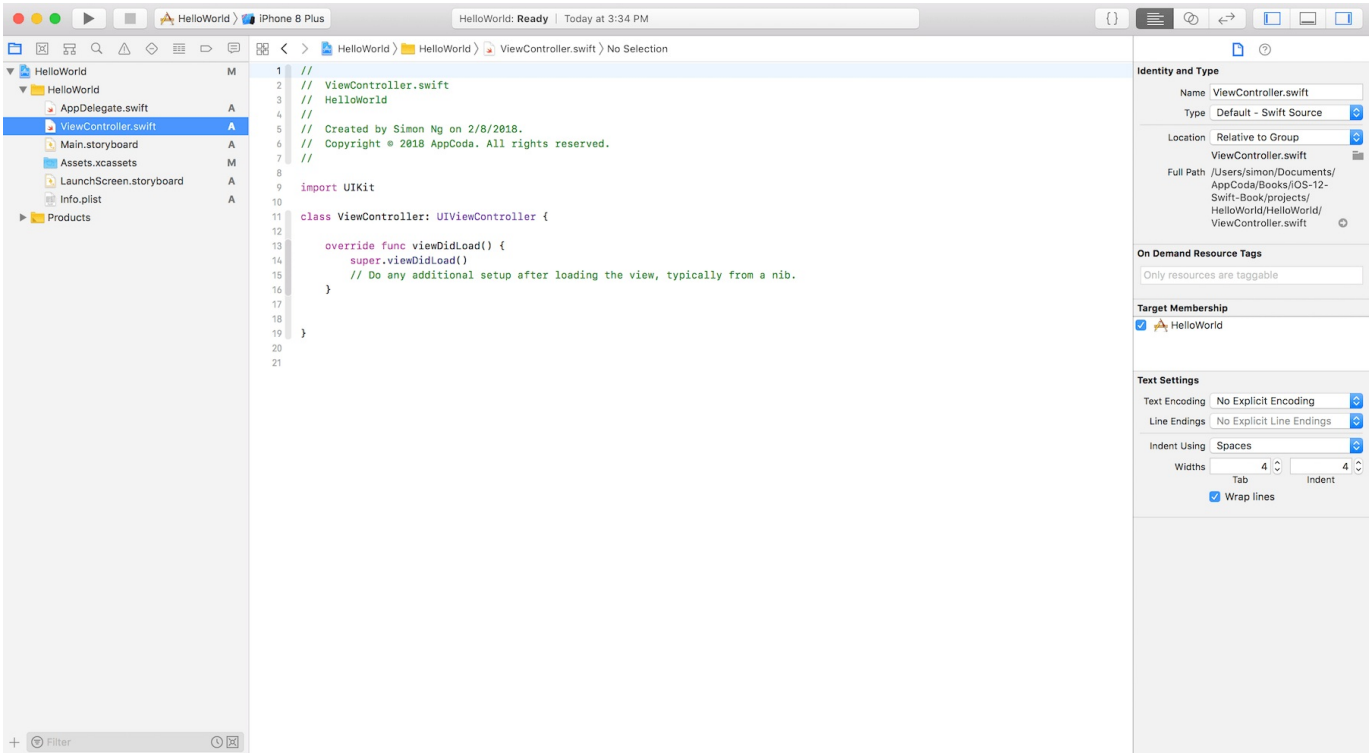


Figure 3-8. Xcode Workspace with Source Code Editor

If you select Main.storyboard, which is the file for storing user interface, Xcode shows you the visual editor for storyboard (see figure 3-9).

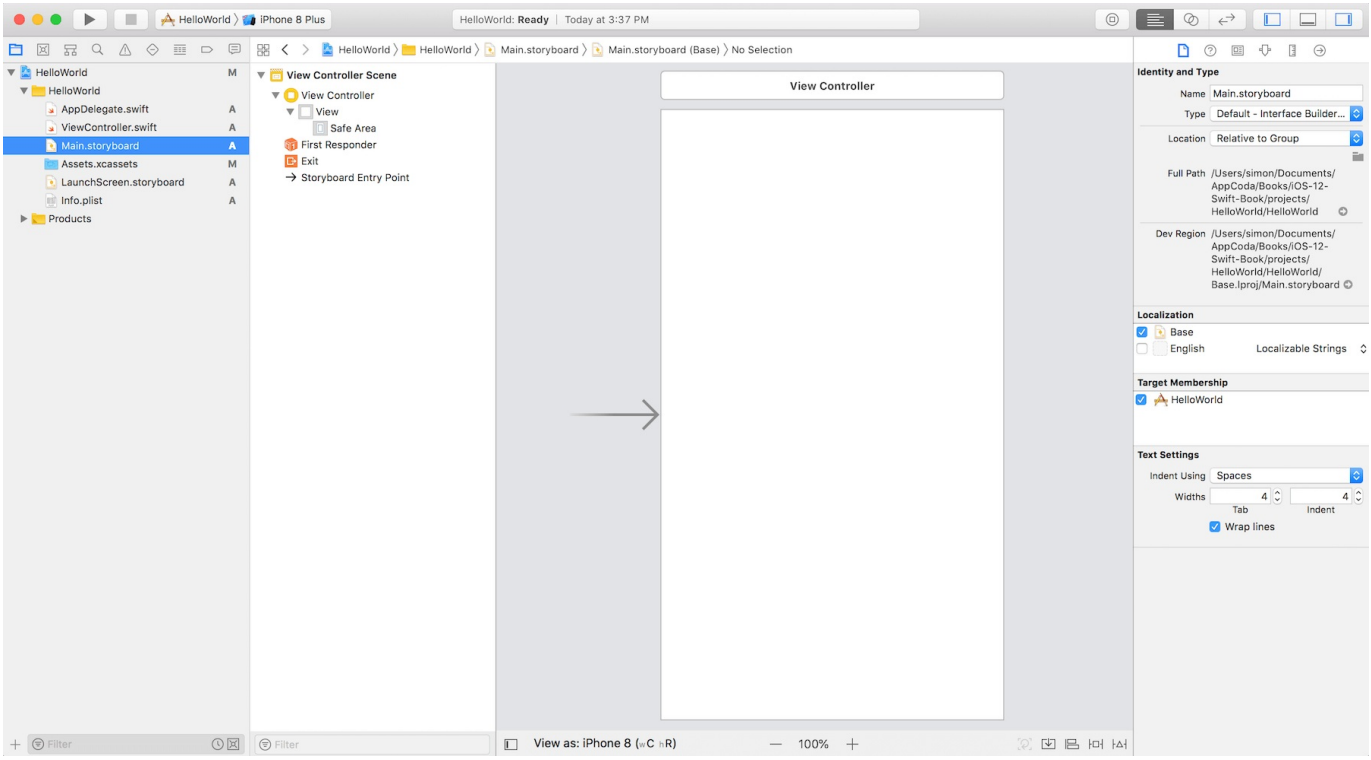


Figure 3-9. Xcode Workspace with Storyboard Editor

The rightmost pane is the utility area. This area displays the properties of the file and allows you to access Quick Help. If Xcode doesn't show this area, you can select the rightmost button in the toolbar (at the top-right corner) to enable it.

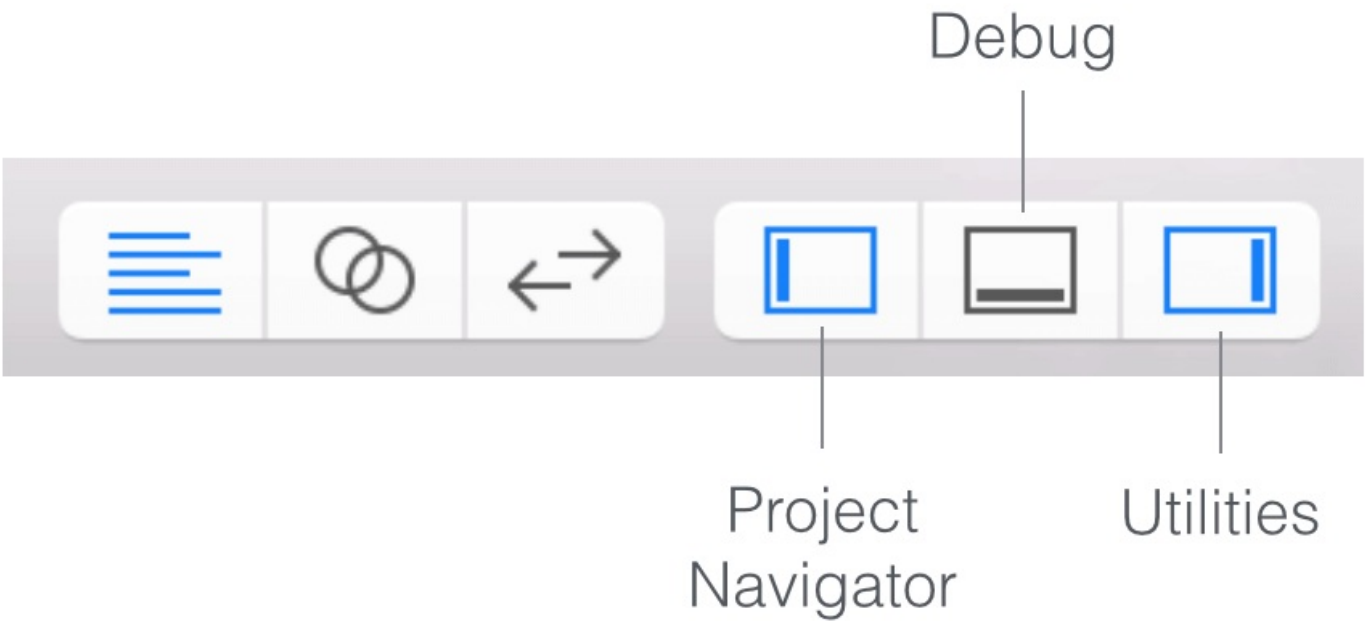


Figure 3-10. Show/hide the content areas of your workspace

The middle view button of the view selector is deselected by default. If you click on it, Xcode displays the debug area right below the editor area. The debug area, as its name suggests, is used for showing debug messages. We'll talk about that in a later chapter, so don't worry if you do not understand what each area is for.

Run Your App for the First Time

Until now, we haven't written a single line of code. Even so, you can run your app using the built-in simulator. This will give you an idea how to build and test your app in Xcode. In the toolbar you should see the Run button.

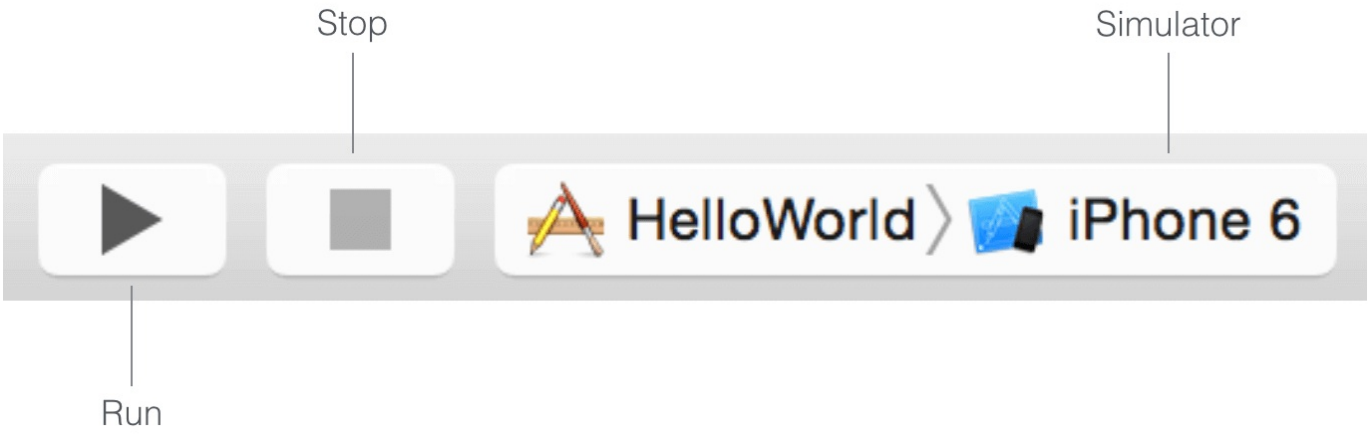


Figure 3-11. Run and Stop Buttons in Xcode

The *Run* button in Xcode is used to build an app and run it in the selected simulator. By default, the Simulator is set to *iPhone 8 Plus*. If you click the *iPhone 8 Plus* button, you'll see a list of available simulators such as iPhone 8 and iPhone X. Let's select *iPhone 8* as the simulator, and give it a try.

Once selected, you can click the *Run* button to load your app in the simulator. Figure 3-12 shows the simulator of an iPhone 8.

Figure 3-12. The Simulator

> **Quick tip:** On non-retina Mac, it may not be able to show the full simulator window. You can select the simulator, and press command+1 to scale it down. Alternatively, you can place the mouse cursor near one of the corners of the device's bezel and scale down the size.

A white screen with nothing inside?! That's normal. So far we haven't designed the user interface or written any lines of code. This is why the simulator shows a blank screen. To terminate the app, simply hit the *Stop* button in the toolbar.

Try to select another simulator (e.g. iPhone X) and run the app. You will see another simulator showed up on screen. Since the release of Xcode 9, you are allowed to run multiple simulators simultaneously.

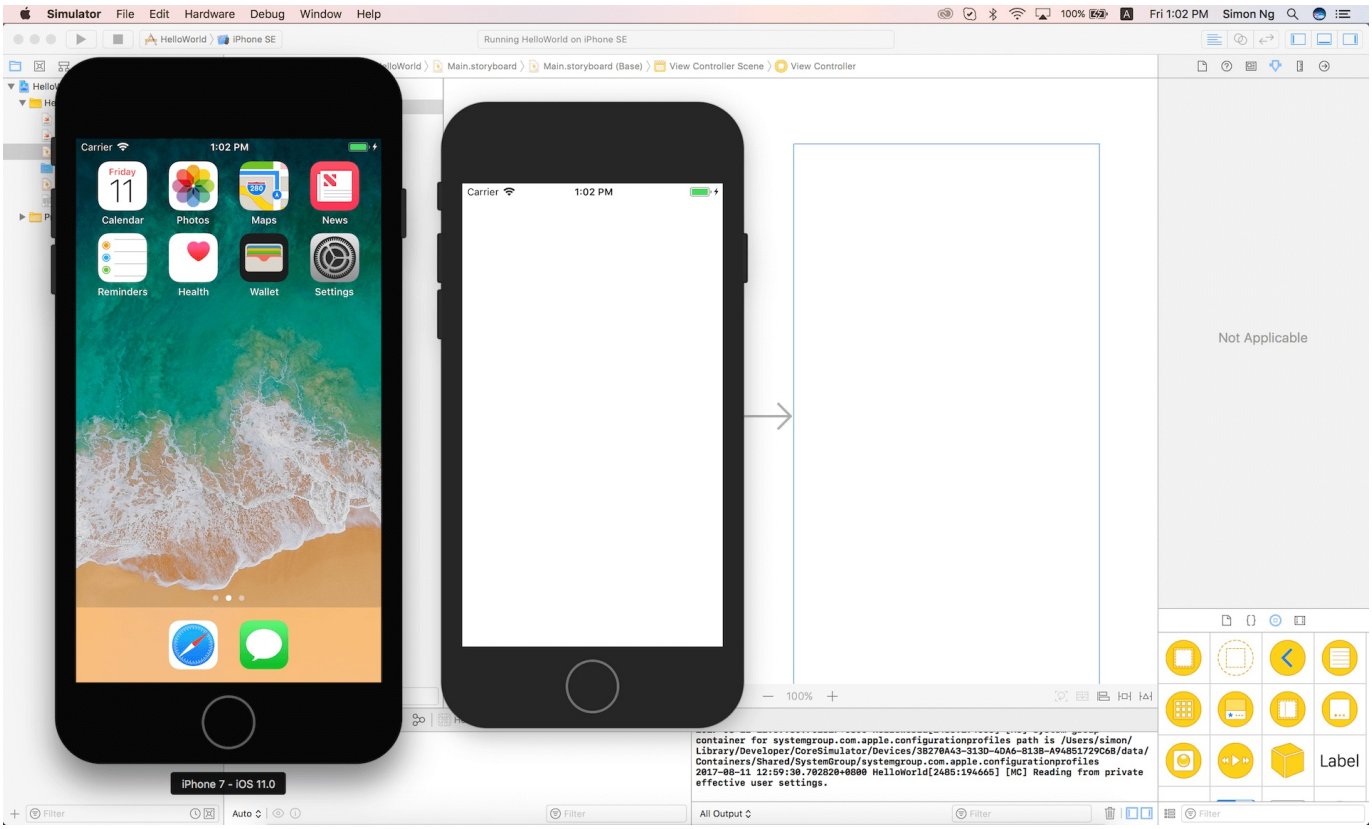


Figure 3-13. Running multiple simulators

The simulator works pretty much like a real iPhone. You can click the home button to bring up the home screen, and it comes with some built-in apps. Just play around with it to familiarize yourself with Xcode and simulator environment.

> **Quick tip:** Running multiple simulators at the same time requires extra memory usage on your Mac. If you do not need any of the simulators, you can select the simulator and press command+w to close it.

A Quick Walkthrough of Interface Builder

Now that you have a basic idea of the Xcode development environment, let's move on and design the user interface of your first app. In the project navigator, select the `Main.storyboard` file. Xcode then brings up a visual editor for storyboards, known as

Interface Builder.

The Interface Builder editor provides a visual way for you to create and design an app's UI. Not only can you use it to design individual view (or screen), the Interface Builder's storyboard designer lets you lay out multiple views, and chain them together using different types of transitions, so as to create the complete user interface. All these can be done without writing a line of code.

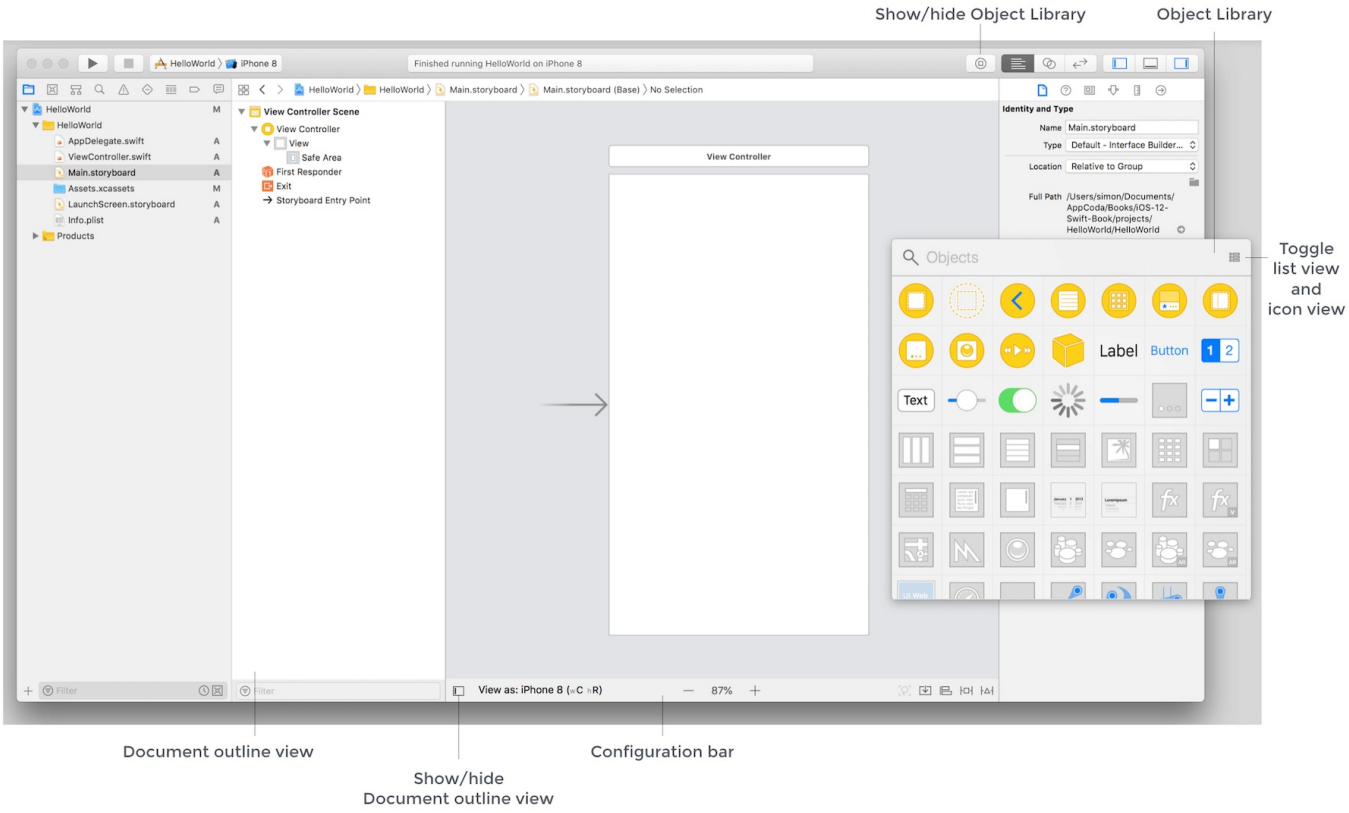


Figure 3-14. The Interface Builder Editor

In Xcode 10, Apple has made a change to the Object library. If you've used Xcode 9 (or earlier version) before, the Object library was situated at the lower right pane. Now the Object library is hidden by default. You have to click the Object library button at the top menu, as displayed in figure 3-14, to make it appear as a floating window.

The Object library contains all the available UI objects such as button, label, image view for you to design your user interface. You can view the Object library in two difference modes: *list view* and *icon view*. I prefer to use icon view in this book. But if you want to change to the list view mode, simply use the toggle button to switch between them.

Since we chose to use the *Single View Application* template during project creation, Xcode generated a default view controller scene in the storyboard. In your Interface Builder, you should see a view controller in the editor area. This view controller is where you design the app's user interface. Each screen of an app is usually represented by a view controller. Interface Builder allows you to add multiple view controllers to the storyboard and link them up. Later in this book, we will further discuss about that. Meanwhile, focus on learning how to use Interface Builder to lay out the UI for the default view controller.

What is a Scene?

A scene in storyboard represents a view controller and its views. When developing iOS apps, views are the basic building blocks for creating your user interface. Each type of view has its own function. For instance, the view you find in the storyboard is a container view for holding other views such as buttons, labels, image views, etc.

A view controller is designed to manage its associated view and subviews (e.g. button and label). If you are confused about the relationship between views and view controllers, no worries. We will discuss how a view and view controller work together in the later chapters.

The Document Outline view of the Interface Builder editor shows you an overview of all scenes and the objects under a specific scene. The outline view is very useful when you want to select a particular object in the storyboard. If the outline view doesn't appear on screen, use the toggle button (see figure 3-14) to enable/disable the outline view.

Lastly, there is a configuration bar in the Interface Builder. To reveal the bar, place the mouse cursor on `View as: iPhone 8`, and then single-click on it. The configuration bar was first introduced in Xcode 8 that lets you live preview your app UI on different devices. Furthermore, you can use the `+` and `-` buttons to zoom in/out the storyboard. We will talk about this new feature later on.

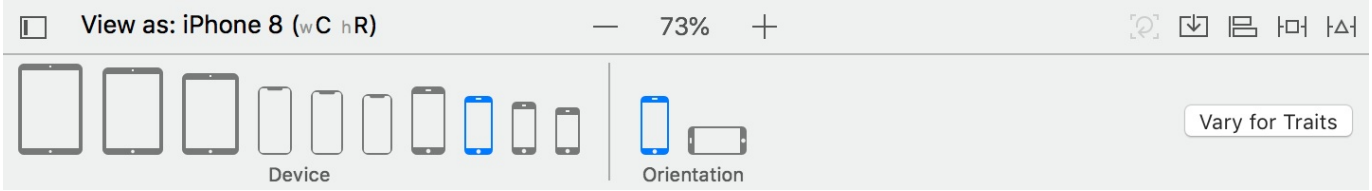


Figure 3-15. The configuration bar in Xcode

Designing the User Interface

Now we are going to design the app's user interface. First, we will add a Hello World button to the view. Click the Object library button to shows the Object library. You can then choose any of the UI objects, and drag-and-drop them into the view. If you're in the icon view mode of the Object library, you can click on any of the objects to reveal the detailed description.

Okay, it's time to add a button to the view. All you need to do is drag a Button object from the Object library to the view.

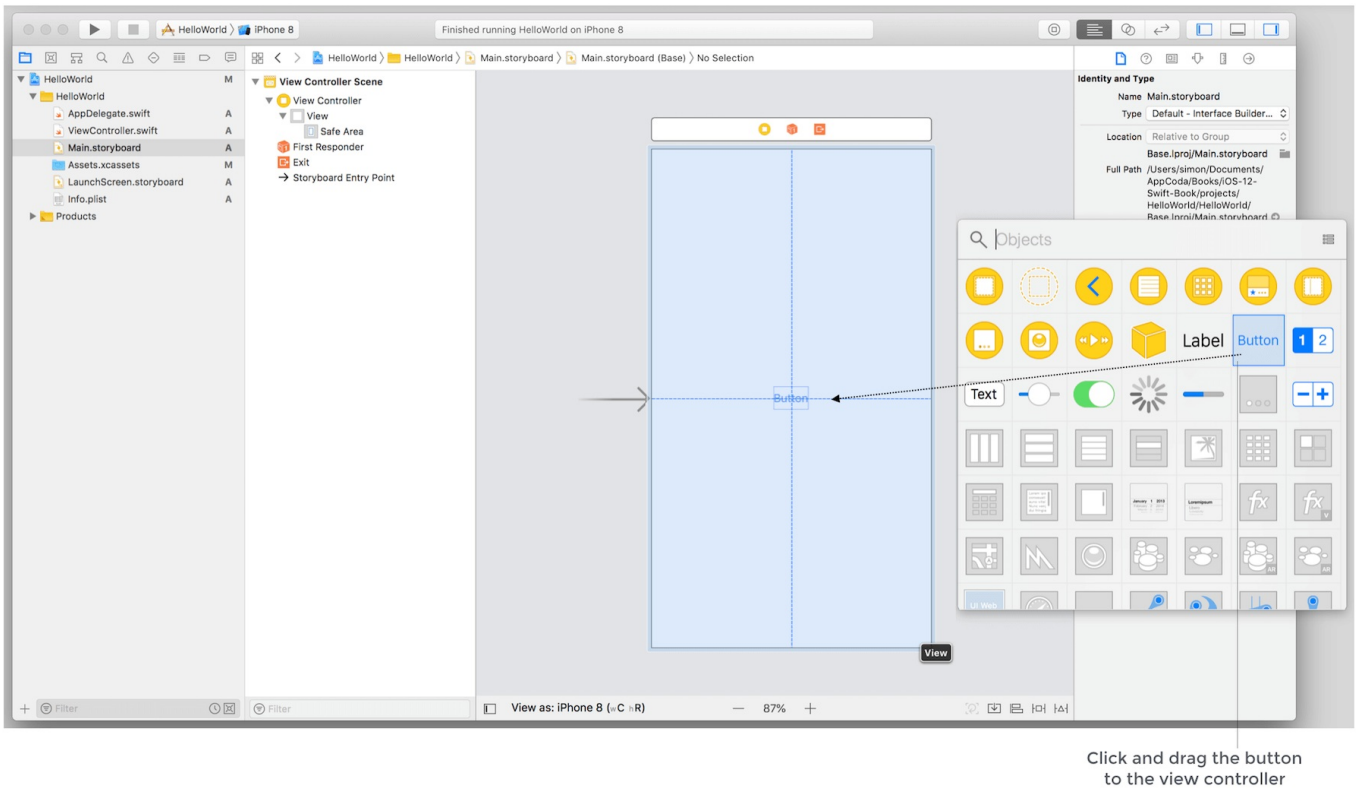


Figure 3-16. Drag the Button to the View

As you drag the Button object to the view, you'll see a set of horizontal and vertical guides if the button is centered. Stop dragging, and release your button to place the Button object there.

Next, let's rename the button. To edit the label of the button, double-click it and name it "Hello World". After the change, you may need to center the button again.

Figure 3-17. Renaming the button

In case the words are truncated, you can resize the button to make it fit or press command+= to let Xcode resize it for you.

Figure 3-18. Resizing the Hello World button

Great! You're now ready to test your app. Select the iPhone 8 simulator and hit the Run button to execute the project, you should see a Hello World button in the simulator as shown in figure 3-19. Cool, right?

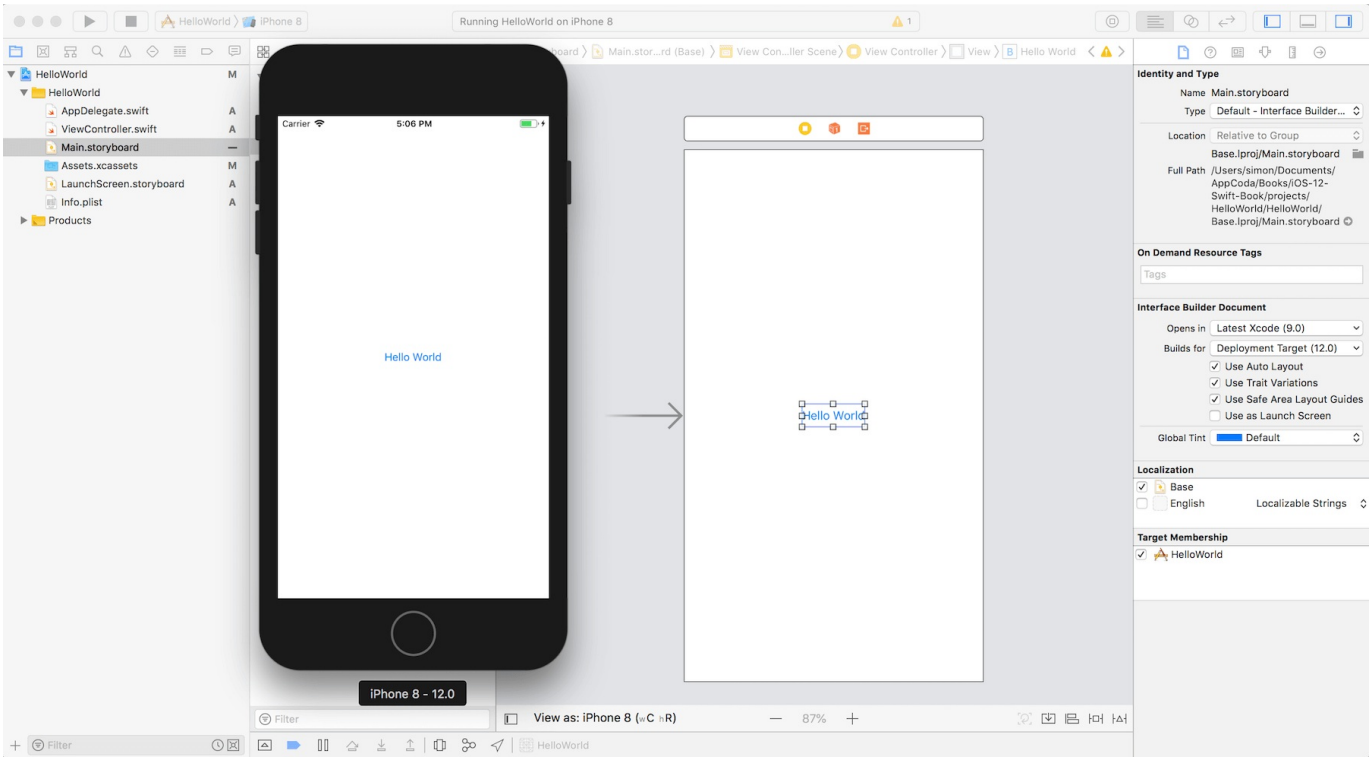


Figure 3-19. Hello World app with a Button

However, when you tap the button, it shows nothing. We'll need to add a few lines of code to display the "Hello, World" message.

> **Quick note:** This is the beauty of iOS development. The code and user interface of an app are separated. You're free to design your user interface in Interface Builder and prototype an app without writing any lines of code.

Coding the Hello World Button

Now that you've completed the UI of the HelloWorld app, it's time to write some code. In the project navigator, you should find the `ViewController.swift` file. Because we initially selected the *Single View Application* project template, Xcode already generated a `ViewController` class in the `ViewController.swift` file. This file is actually associated with the view controller in the storyboard. In order to display a message when the button is tapped, we'll add some code to the file.

> **Swift versus Objective-C**
> If you have written code in Objective-C before, one big change in Swift is the consolidation of header (.h) and implementation file (.m). All the information of a particular class is now stored in a single .swift file.

Select the `ViewController.swift` file and the editor area immediately displays the source code. Type the following lines of code in the `ViewController` class:

```
@IBAction func showMessage(sender: UIButton) {
    let alertController = UIAlertController(title: "Welcome to My First App", mess
age: "Hello World", preferredStyle: UIAlertController.Style.alert)
    alertController.addAction(UIAlertAction(title: "OK", style: UIAlertAction.Style
.default, handler: nil))
    present(alertController, animated: true, completion: nil)
}
```

Note: I encourage you to type the code, rather than copy & paste it.

Your source code should look like this after editing:

```
import UIKit

class ViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view, typically from a nib.
    }

    @IBAction func showMessage(sender: UIButton) {
        let alertController = UIAlertController(title: "Welcome to My First App",
message: "Hello World", preferredStyle: UIAlertController.Style.alert)
        alertController.addAction(UIAlertAction(title: "OK", style: UIAlertAction.
Style.default, handler: nil))
        present(alertController, animated: true, completion: nil)
    }
}
```

What you have just done is added a `showMessage(sender: UIButton)` method in the `ViewController` class. The Swift code within the method is new to you. I will explain it to you in the next chapter. Meanwhile, just consider the `showMessage(sender: UIButton)` as an

action. When this action is called, the block of code will instruct iOS to display a "Hello World" message on screen.

```
Note: Did you notice that some lines of the code were begun
with "//"? In Swift, if the line is prefixed with "//", that
line of code becomes a comment, which will not be executed.
```

You can try to run the project in the simulator. The behaviour of the app is still the same. When you tap the button, it still doesn't show you any response. The reason is that we haven't made the connection between the button and the code.

Connecting the User Interface with Code

I said before that the beauty of iOS development is the separation of code (.swift file) and user interface (storyboards). But how can we establish the relationship between our source code and the user interface?

To be specific for this demo, the question is:

- How can we connect the "Hello World" button in the storyboard with the `showMessage(sender: UIButton)` method in the `ViewController` class?

You need to establish a connection between the "Hello World" button and the `showMessage(sender: UIButton)` method you've just added, such that the app responds when someone taps the Hello World button.

Now select `Main.storyboard` to switch back to the Interface Builder. Press and hold the control key of the keyboard, click the "Hello World" button and drag it to the View Controller icon. Release both buttons (mouse + keyboard) and a pop-up shows the `showMessageWithSender:` option under Sent Events. Select it to make a connection between the button and `showMessageWithSender:` action.

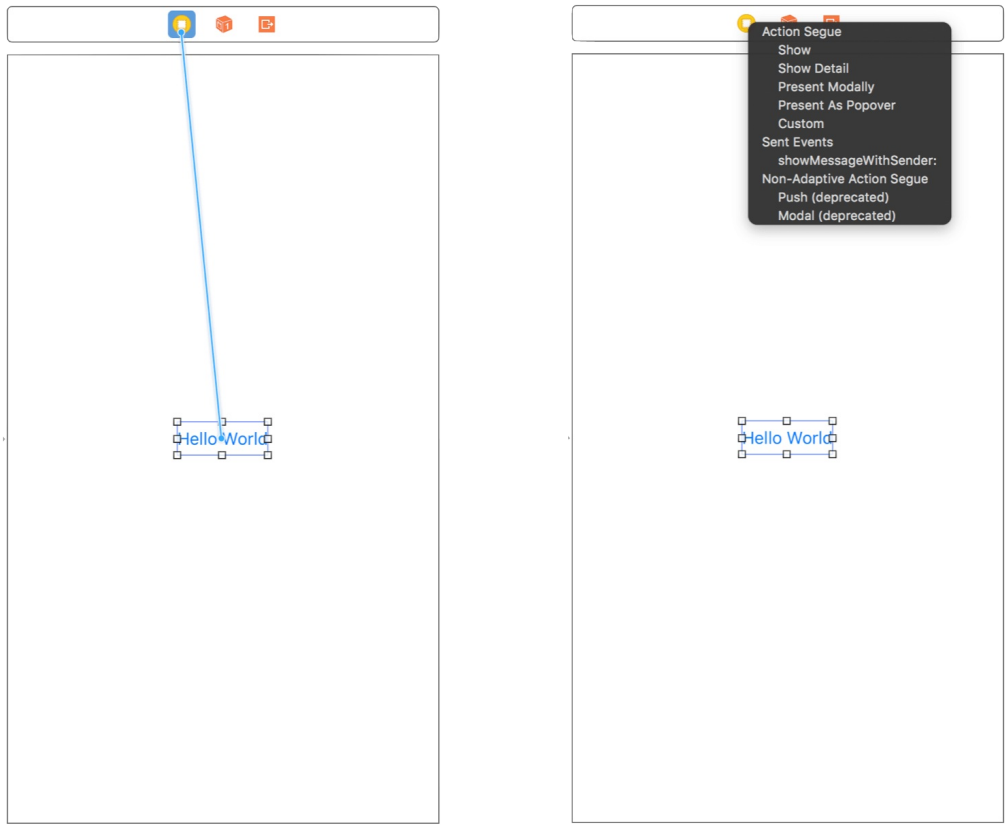


Figure 3-20. Drag to the View Controller icon (left), a pop-over menu appears when releasing the buttons (right)

Test Your App

That's it! You're now ready to test your first app. Just hit the *Run* button. If everything is correct, your app should run properly in the simulator. This time, the app displays a welcome message when you tap the Hello World button.

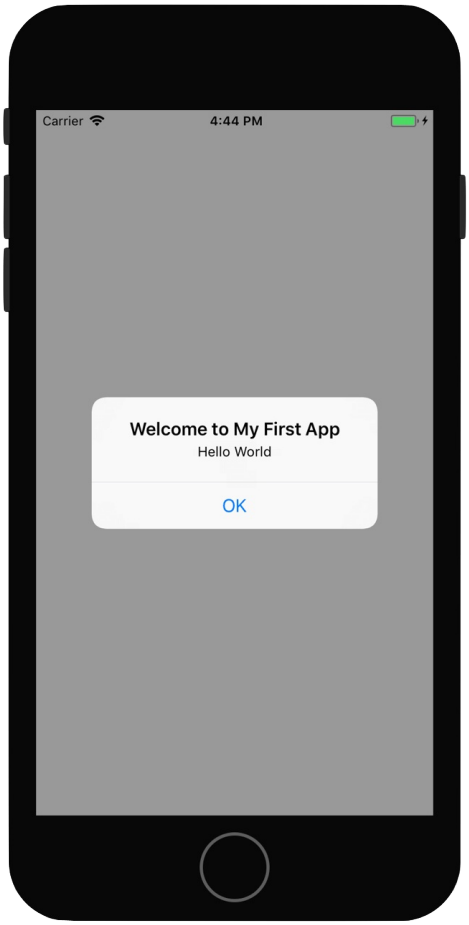


Figure 3-21. Hello World app

Changing the Button Color

As mentioned before, you do not need to write code to customize a UI control. Here, I want to show you how easy it is to change the properties (e.g. color) of a button.

Select the "Hello World" button and then click the Attributes inspector under the Utility area. You'll be able to access the properties of the button. Here, you can change the font, text color, background color, etc. Try to change the text color (under Button section) to *white* and background (scroll down and you'll find it under View section) to *red* or whatever color you want.

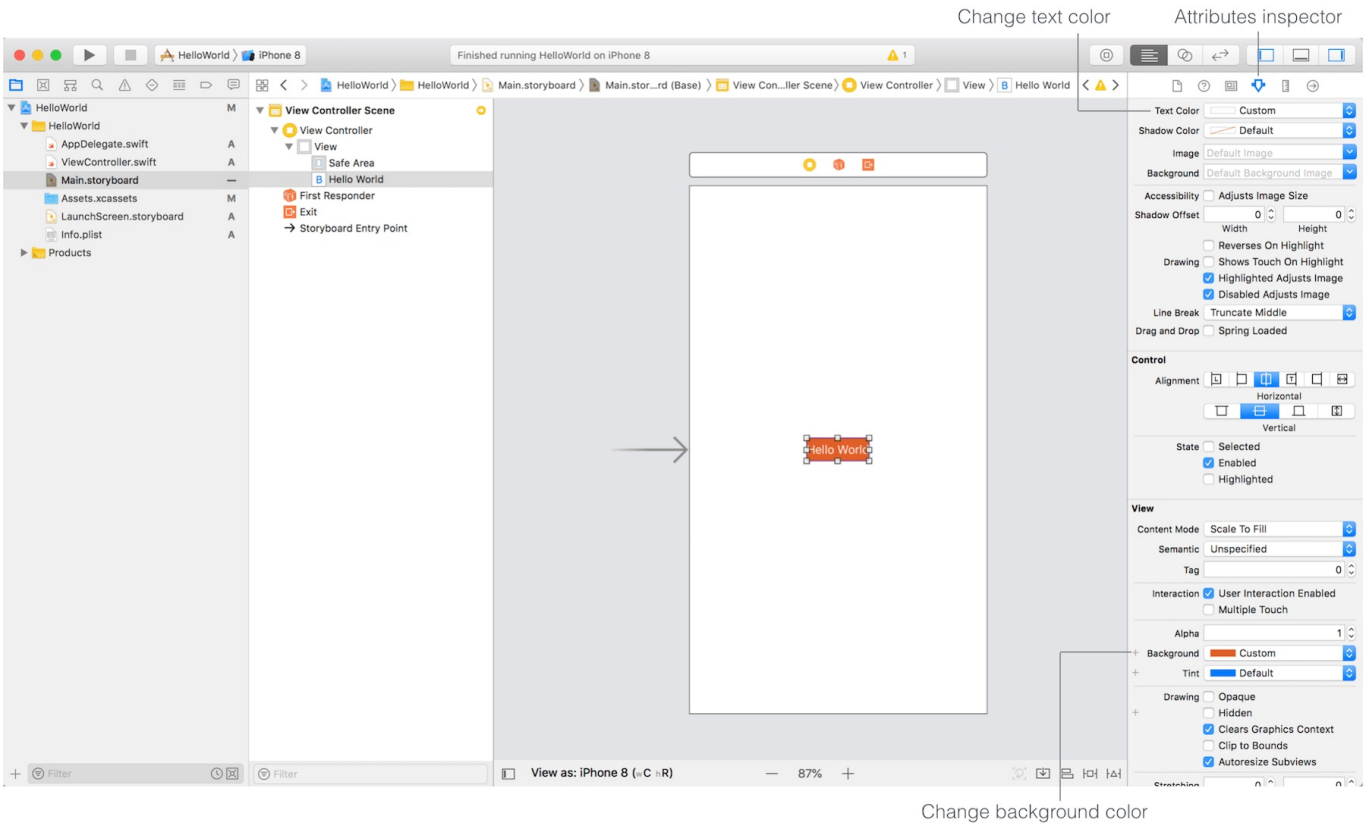


Figure 3-22. Changing the color of the Hello World button

Run the project again and see what you get.

Your Exercise #1

Not only can you change the color of a button, you can modify the font type and size in the Attributes inspector by setting the *Font* option. Your task is to continue to work on the project and create a user interface like figure 3-23. When a user taps any of the buttons, the app displays the same Hello World message.

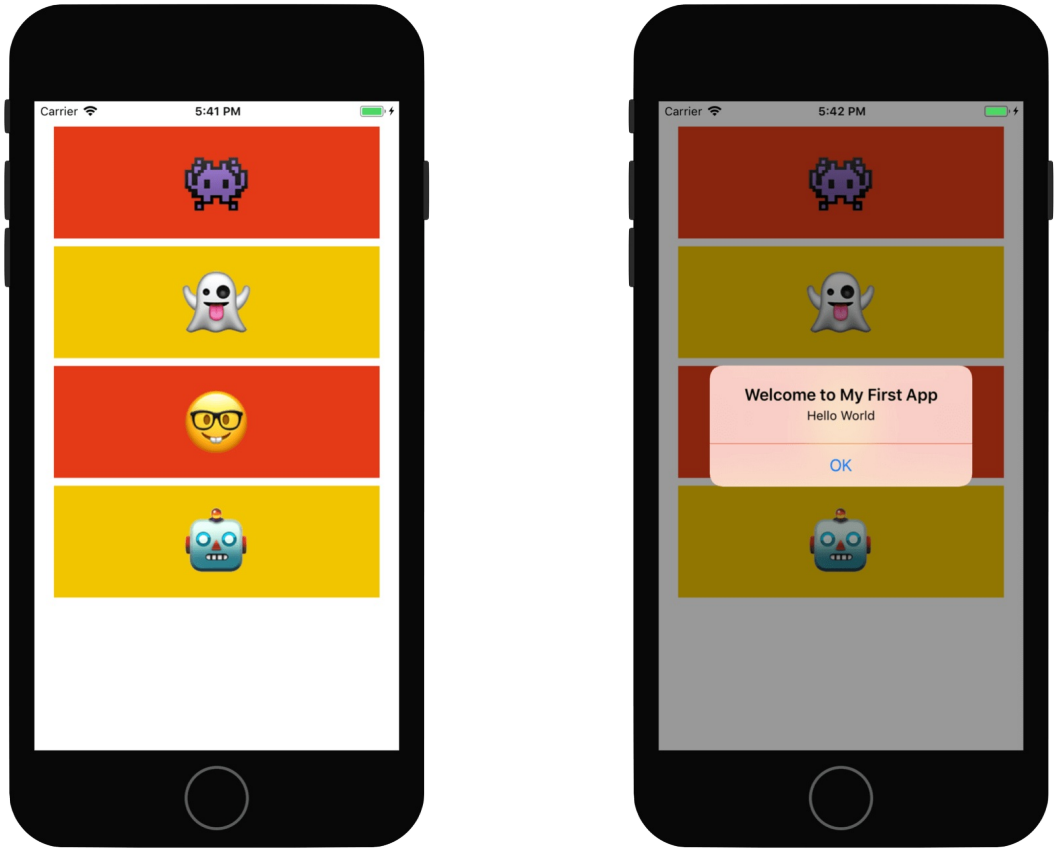


Figure 3-23. Design this app

To give you some hints, here are the things you will need to work on:

1. Resize the "Hello World" button and change its font size to *70 points*. Change its title from *Hello World* to 👾. To key in emoji characters, you can hold down control+command and then press spacebar.
2. Add three more buttons. Each of them has an emoji icon as its title.
3. Establish a connection between the buttons and the `showMessage(sender: UIButton)` method.

Your Exercise #2

Wouldn't it be great if you can display the meaning of the emoji icon instead of the same "Hello World" message like what I have shown you in figure 3-2?

You've learned how to use a dictionary to store the meaning of emoji icons. Now try to modify the `showMessage(sender: UIButton)` method such that it displays the meaning of the emoji icon. I will give you a hint. Here is the code skeleton. Replace the code of your existing `showMessage(sender: UIButton)` method with the following and fill in the missing code:

```
@IBAction func showMessage(sender: UIButton) {

    // Initialize the dictionary for the emoji icons
    // If you forgot how to do it, refer to the previous chapter
    // Fill in the code below




    // The sender is the button that is tapped by the user.
    // Here we store the sender in the selectedButton constant
    let selectedButton = sender

    // Get the emoji from the title label of the selected button
    if let wordToLookup = selectedButton.titleLabel?.text {

        // Get the meaning of the emoji from the dictionary
        // Fill in the code below



        // Change the line below to display the meaning of the emoji instead of He
llo World
        let alertController = UIAlertController(title: "Meaning", message: meaning
, preferredStyle: UIAlertController.Style.alert)

        alertController.addAction(UIAlertAction(title: "OK", style: UIAlertAction.
Style.default, handler: nil))
        present(alertController, animated: true, completion: nil)
    }

}
```

This exercise is more difficult than the first one. Try your best to complete it. It is really fun to see that you turn a Hello World app into a simple Emoji Translator.

What's Coming Next

Congratulations! You've built your first iPhone app. It's a simple app, but I believe you already have a better understanding of Xcode and understand how an app is built. It's easier than you thought, right?

Even if you couldn't complete exercise #2, that is completely fine. Don't be discouraged. I have included the solution for your reference. Just keep reading, you will get better and better as you have more coding practices.

In the next chapter, we'll discuss the details of the Hello World app and explain how everything works together.

For reference, you can download the complete Xcode project from http://www.appcoda.com/resources/swift42/HelloWorld.zip.

To continue reading and access the solution of the exercise, please get the full copy of the book here.

Chapter 4
Hello World App Explained

> Any fool can know. The point is to understand.
>
> – Albert Einstein

Isn't it easy to build an app? I hope you enjoyed building your first app.

Before we continue to explore the iOS SDK, let's take a pause here and have a closer look at the Hello World app. It'll be good for you to understand the basics of the iOS APIs and the inner workings of the app.

So far you followed the step-by-step procedures to build the Hello World app. As you read through the chapter, you may have had a few questions in mind:

- How does the View Controller in the storyboard link up with the `ViewController` class in the `ViewController.swift` file?
- What does the block of code inside the `showMessage(sender:)` method mean? How does it tell iOS to show a Hello World message?
- What does `@IBAction` keyword mean?
- What is behind the "Hello World" button? How can the button detect tap and trigger the `showMessage(sender:)` method?
- What is `viewDidLoad()`?
- How does the Run button in Xcode work? What do you mean by "compile an app"?

I wanted you to focus on exploring the Xcode environment so I didn't explain any of the questions above. Yet it's essential for every developer to understand the details behind the code and grasp the basic concept of iOS programming. The technical concepts may be a bit hard to understand, in particular, if you have no prior programming experience. Don't worry because this is just the beginning. As you continue to study and write more code in the later chapters, you will gain a better understanding of iOS programming. Just try your best to learn as much as possible.

Understanding Implementation and Interface

Before diving into the programming concept, let's take a look at a real life example.

Consider a TV remote control. It's convenient to control the volume of a TV set wirelessly with a remote. To switch TV channels, you simply key in the channel number. To increase the speaker volume, you press the Volume + button.

Let me ask you. Do you know what happens behind the scene when pressing the Volume button or the channel button? Probably not. I believe most of us don't know how a remote control communicates with a TV set wirelessly. You may just think that the remote sends a certain message to the TV and triggers the volume increase or channel switch.

In this example, the button that interacts with you is commonly characterized as the *interface* and the inner details that hide behind the button are referred as the *implementation*. The interface communicates with the implementation via a message.

This concept can also be applied in the iOS programming world. The user interface in storyboard is the *interface*, while the code is the *implementation*. The user interface objects (e.g. button) communicate with the code via messages.

Specifically, if you go back to the Hello World project, the button you added in the view is the interface. The `showMessage(sender:)` method of the `ViewController` class is the implementation. When someone taps the button, it sends a `showMessageWithSender` message to `ViewController` by invoking the `showMessage(sender:)` method.

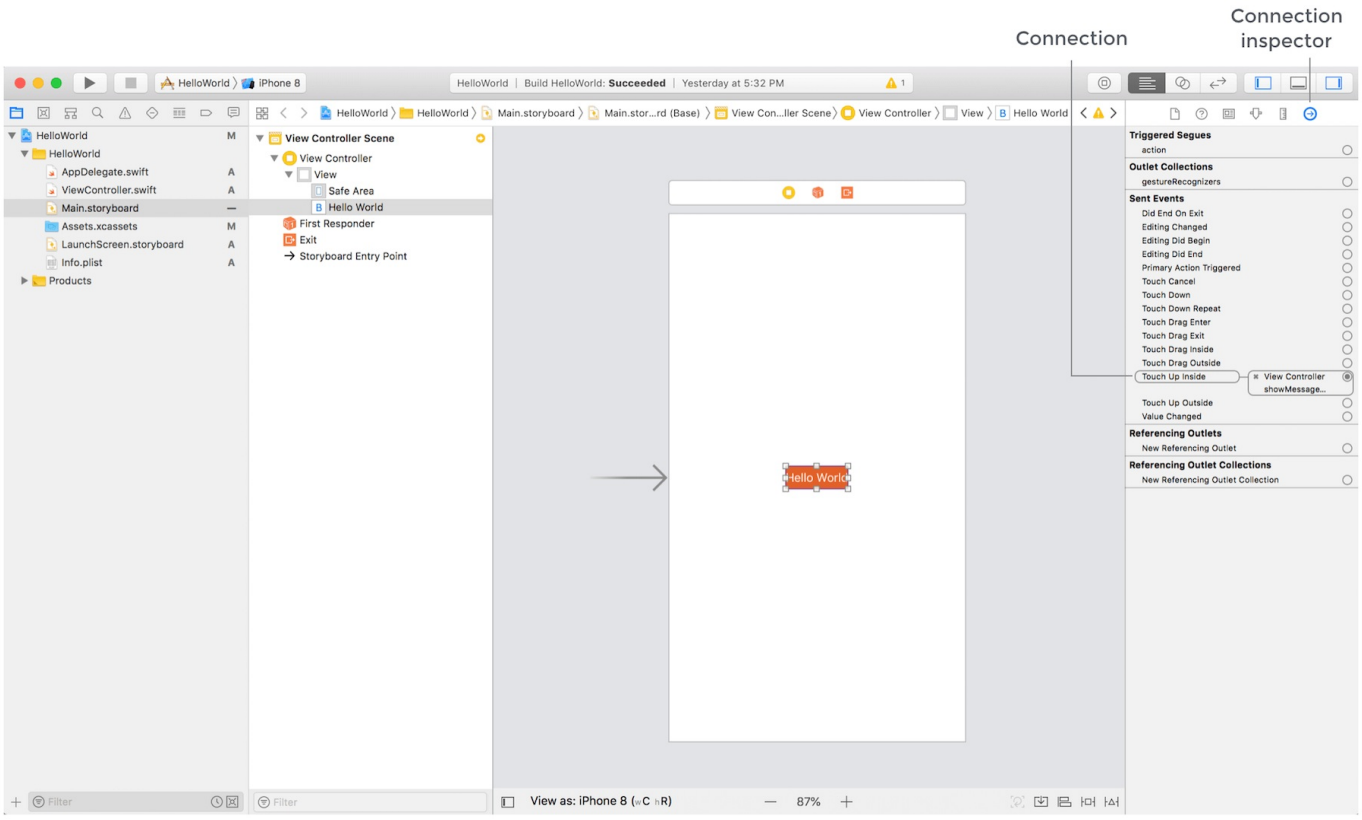
What we have just demonstrated is one of the important concepts behind Object Oriented Programming known as *Encapsulation*. The implementation of the `showMessage(sender:)` method is hidden from the outside world (i.e. the interface). The Hello World button has no idea how the `showMessage(sender:)` method works. All it knows is that it needs to send a message. The `showMessage(sender:)` method handles the rest by displaying a "Hello World" message on the screen.

Quick note: Like Objective-C, Swift is an Object-Oriented Programming (OOP) language. Most of the code in an app in some ways deals with objects of some kind. I don't want to scare you away by teaching you the OOP concepts here. Just keep reading. As we go along, you'll learn more about OOP.

Behind the Touch

Now that you should have a basic understanding about how the button in the UI communicates with the code, let's take a look at what actually happens when a user taps the "Hello World" button? How does the "Hello World" button invoke the execution of the `showMessage(sender:)` method?

Do you remember how you established the connection between the `Hello World` button and the `showMessage(sender:)` event in Interface Builder?



In iOS, apps are based on event-driven programming. Whether it's a system object or UI object, it listens for certain events to determine the flow of the app. For a UI object (e.g. Button), it can listen for a specific touch event. When the event is triggered, the object calls up the preset method that associates with the event.

In the Hello World app, when users lift up the finger inside the button, the "Touch Up Inside" event is triggered. Consequently, it calls up the `showMessage(sender:)` method to display the "Hello World" message. We use the "Touch Up Inside" event instead of "Touch Down" because we want to avoid an accidental or false touch. The illustration shown in figure 4-3 sums up the event flow and what I have just described.

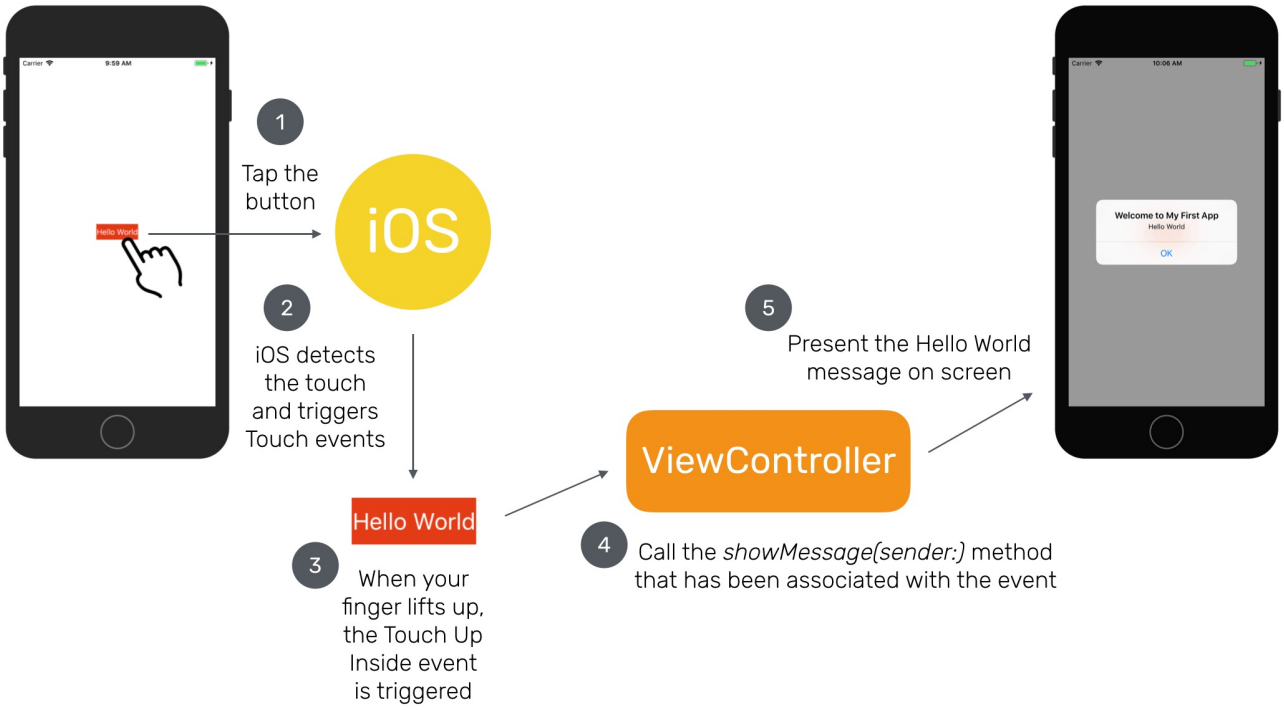


Figure 4-3. Event flow in the Hello World App

Inside the showMessage Method

You should now have a better understanding of iOS programming. But what is the block of code in the `showMessage(sender:)` method?

First things first, what is a method? As mentioned before, most of the code in an app in some ways deals with objects of some kinds. Each object provides certain functions and performs specific tasks (e.g. display a message on screen). These functions when expressed in code are known as *methods*.

Now, let's take a closer look at the `showMessage(sender:)` method.

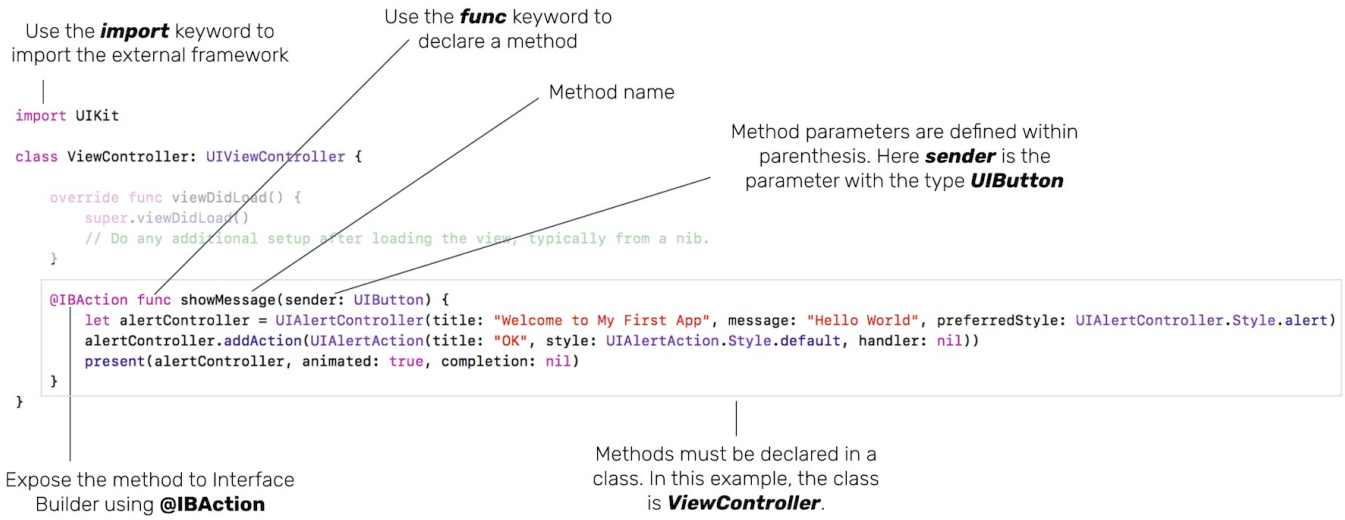


Figure 4-4. showMessage() method explained

> **Quick note:** I know it may be a bit hard for you to understand the code. If you're completely new to programming, it'll take some time to get used to Object Oriented Programming. Don't give up, as you'll gain a better understanding of objects, classes, and methods as we move along. You can also take a look at the Appendix to learn more about the Swift syntax.

In Swift, to declare a method in a class, we use the `func` keyword. Following the `func` keyword is the name of the method. This name identifies the method and makes it easy for the method to be called elsewhere in your code. Optionally, a method can take in parameters as input. The parameters are defined within the parentheses. Each of the parameters should have a name and a type, separated by a colon (`:`). In our example, the method accepts a `sender` parameter which has a type `UIButton` . The `sender` parameter indicates the object that sends the request. In other words, it tells you the button that the user has tapped.

Note: Do you remember exercise #2 in the previous chapter? The *sender* parameter can tell you which emoji button the user has tapped.

A method doesn't have to take in parameters. In this case, you simply write an empty pair of parentheses like this:

```
func showMessage()
```

There is one keyword in the method declaration that we haven't discussed. It's the `@IBAction` keyword. This keyword allows you to connect your source code to user interface objects in Interface Builder. When it is inserted in the method declaration, it indicates the method can be exposed to Interface Builder. This is why the `showMessageWithSender` event appeared in a pop-over when you established the connection between the Hello World button and the code in chapter 3. Refer to figure 3-19 if you do not understand what I mean.

Okay, enough for the method declaration. Let's talk about the block of code enclosed in the curly braces. The code block is the actual implementation of the task performed by the method.

If you look at the first line of the code block closely, we make use of `UIAlertController` to construct the Hello World message. What the heck is `UIAlertController` ? Where does it come from?

When developing apps in iOS, we don't need to write all functions from scratch. Say, you don't need to learn how to draw the alert box on screen. The iOS SDK, bundled in Xcode, already provides you with tons of built-in functions to make your life easier. These functions are usually known as *APIs* and organized in the form of *frameworks*. The `UIKit` framework is just one of them, that provides classes and functions to construct and manage your app's user interface. For example, `UIViewController` , `UIButton` , and `UIAlertController` actually come from the `UIKit` framework.

There is one thing to take note. Before you can use any functions from the framework, you have to first import it. This is why you find this statement at the very beginning of `ViewController.swift`:

```
import UIKit
```

Now, let's continue to look into the `showMessage(sender:)` method.

The first line of code creates a `UIAlertController` object, and store it in `alertController`. The syntax of constructing an object from a class is very similar to calling a method. You specify the class name, followed by a set of initial values of properties. Here we specify the title, message and preferred style of the alert:

```
let alertController = UIAlertController(title: "Welcome to My First App", message:
"Hello World", preferredStyle: UIAlertController.Style.alert)
```

Right after creating the `UIAlertController` object (i.e. alertController), we call the `addAction` method to add an action to the alert so that it displays an "OK" button. When programming in Swift, you call a method by using dot syntax.

```
alertController.addAction(UIAlertAction(title: "OK", style: UIAlertAction.Style.de
fault, handler: nil))
```

You may wonder how you can find out about the usage and available methods of a class. A simple answer is: *Read the documentation*. You can go up to Google to look up the class. But Xcode provides a convenient way to access the documentation of the iOS SDK.

In Xcode, you can press and hold the option key, point the cursor to the class name (e.g. `UIAlertController`) in your code and then click. A pop-over will appear displaying the class description and sample code. If you need further information, scroll down and click the class reference link. This will bring you to the official documentation of the class.

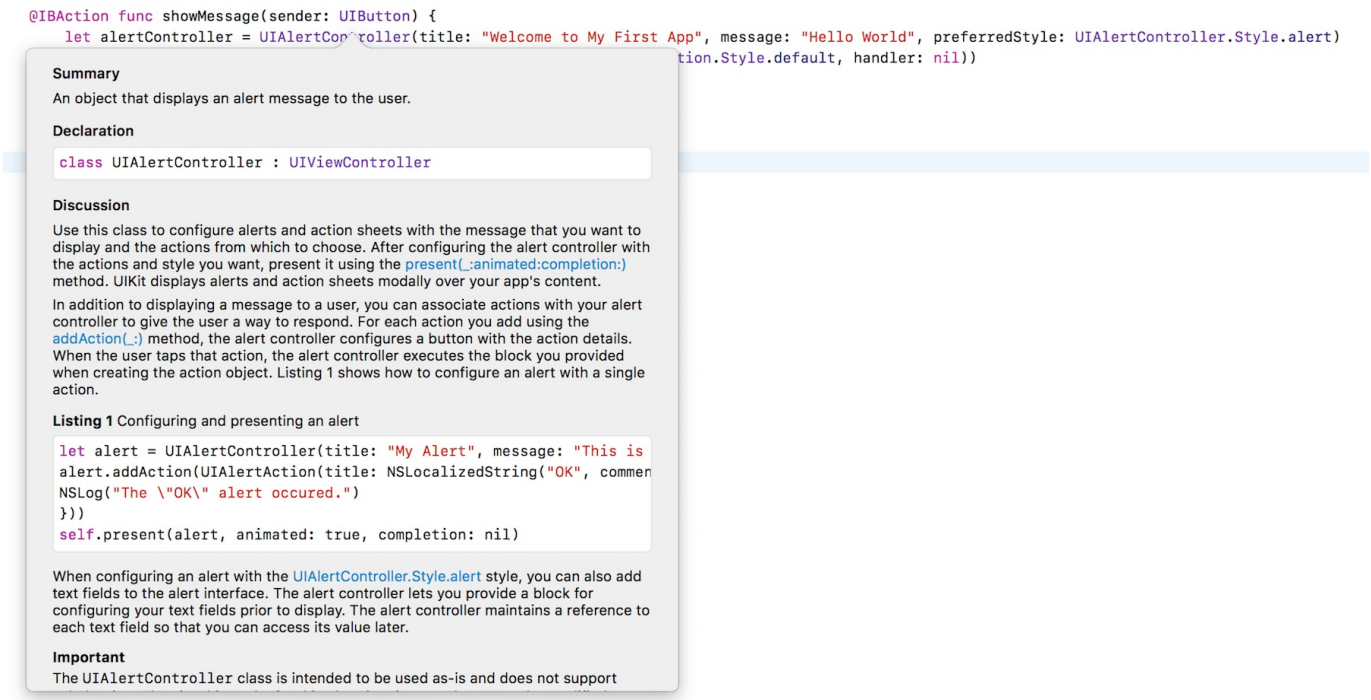


Figure 4-5. Class Information

After the `UIAlertController` object is configured, the last line of the code is for showing the alert message on screen.

```
present(alertController, animated: true, completion: nil)
```

To display the alert, we ask the current view controller to present the `alertController` object with animation.

Sometimes, you may see some developers write the above line of code like this:

```
self.present(alertController, animated: true, completion: nil)
```

In Swift, you use the `self` property to refer to the current instance (or object). In most cases, the `self` keyword is optional. So you can omit it.

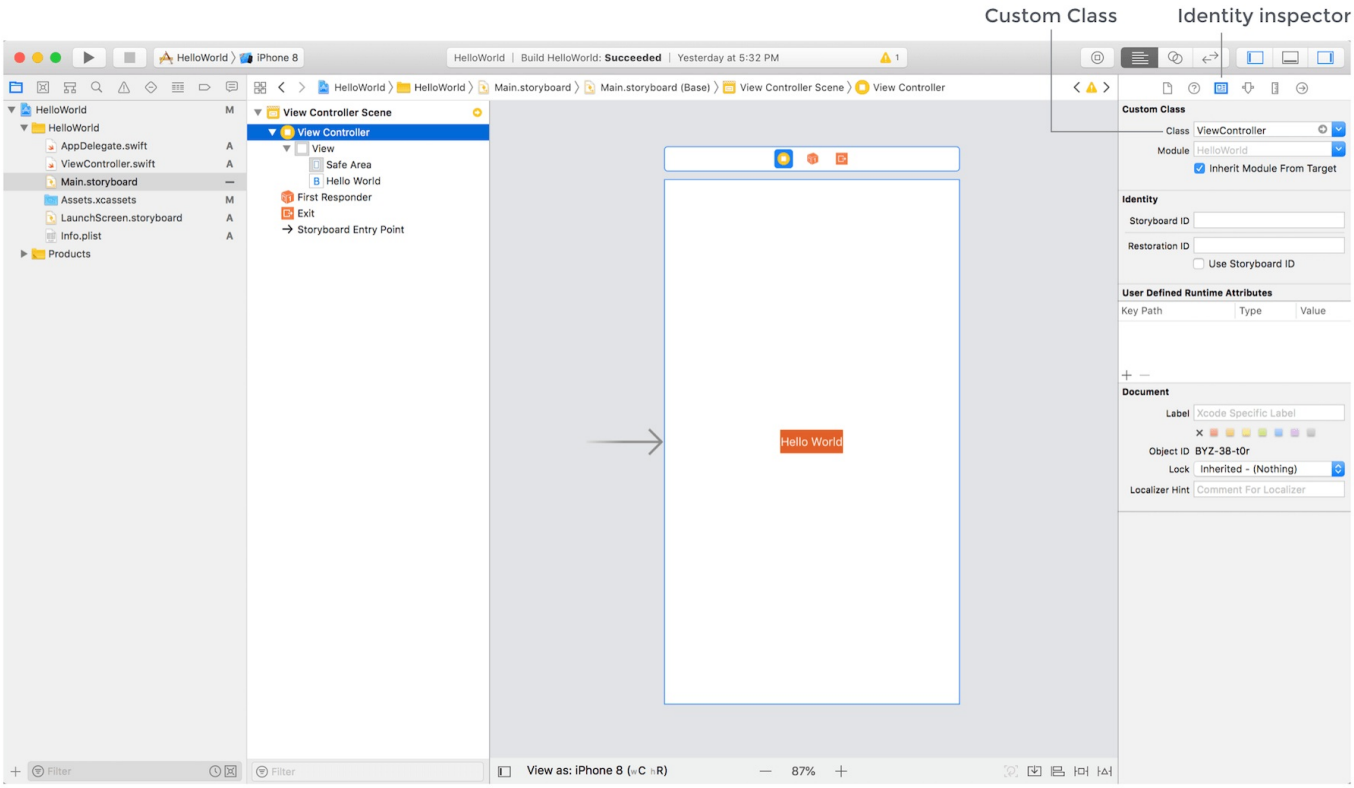
Relationship Between User Interface and Code

How does Xcode know that the View Controller in Interface Builder links up with the `ViewController` class defined in `ViewController.swift`?

The whole thing seems to be trivial but actually it is not. Do you remember the project template that we chose when creating the Xcode project? It is the *Single View Application* template.

When this project template is used, it automatically creates a default view controller in the Interface Builder editor and generates `ViewController.swift`. On top of that, the view controller is automatically linked with the `ViewController` class defined in the swift file.

Go to `Main.storyboard`, select View Controller in the document outline view. In the Utility area, select the Identity inspector icon and you'll find that `ViewController` is set as the custom class. This is how the objects in Interface Builder associate with the classes in Swift code.



UIViewController and View Controller Life Cycle

Do you have questions about this part of the code? You know that `ViewController` is the name of the custom class, but what is `UIViewController` ? And, what is `viewDidLoad()` method? Why was the method here?

```
class ViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view, typically from a nib.
    }


  .
  .
  .

}
```

Like I said before, we rely on Apple's iOS SDK to build our apps. We rarely write our own code to draw a warning dialog or message dialog in order to present some messages on screen. We rarely draw our own buttons either. Instead, we rely on `UIAlertController` and `UIButton` to do the heavy lifting. The same concept applies to the view, the rectangular area we present on screen to users.

`UIViewController` is the fundamental building block of most iOS apps. It holds other UI elements (e.g. buttons) and controls what to display on screen. By default, `UIViewController` has an empty view. As you tested it in the previous chapter, it just displayed a blank screen without any functions or interactions with users. It is our responsibility to provide a custom version of `UIViewController` .

To do that, we create a new class (i.e. `ViewController`) that extends from `UIViewController` . By extending from `UIViewController` , `ViewController` inherits all its functionalities. For instance, it has a default empty view. In code, it is written like this:

```
class ViewController: UIViewController
```

In the body of `ViewController`, we provide our customizations. In the Hello World demo, we added a method called `showMessage(sender:)` to present a Hello World message.

This is one of the fundamental iOS development concepts you have to keep in mind. We are not writing everything from scratch. The iOS SDK already provides us with the skeleton of an iOS app. We build on top of the skeleton to create the UI and functionalities of our own app.

Now that I believe you have a basic idea of `UIViewController`, but what is `viewDidLoad`?

Similar to the "Hello World" button that we discussed earlier, the view of the view controller also receives different events due to the changes of the view's visibility or state. At an appropriate time, iOS automatically calls a specific method of `UIViewController` when the view's state changes.

When the view is loaded, the `viewDidLoad` method will be automatically called, so that you can perform additional initialization. In the Hello World app, we keep it unchanged. As a quick example, you can modify the method of your Hello World app like this to give it a try:

```
override func viewDidLoad() {
    super.viewDidLoad()

    view.backgroundColor = UIColor.black
}
```

Run the app to have a quick test. The background of the view controller's view becomes black. This is one of the many examples when you need to customize the `viewDidLoad()` method. In later chapters, you will learn how we use `viewDidLoad()` for other customizations.

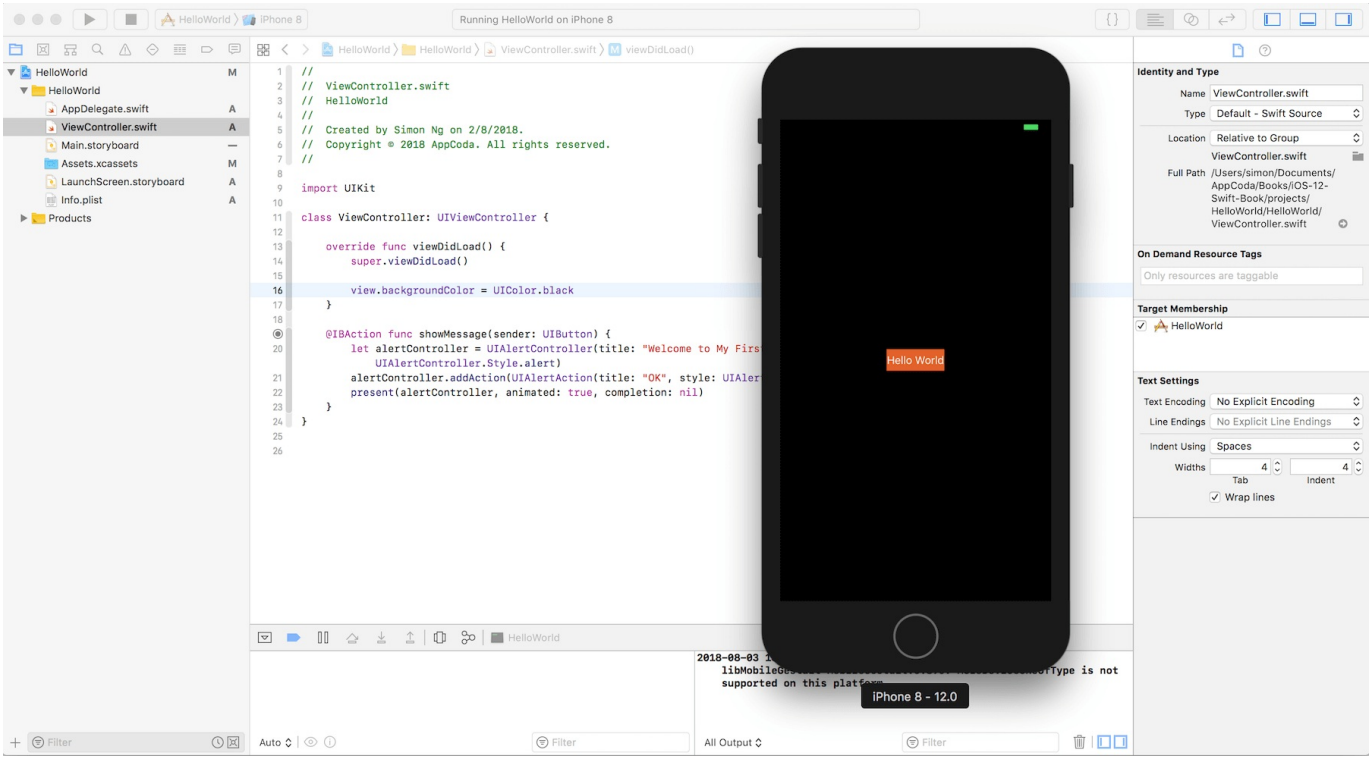


Figure 4-7. Changing the background color of the view

`viewDidLoad` is just one of the methods for managing the view's states. Say, when a user clicks the home button to go back to the Home screen, the `viewWillDisappear` and `viewDidDisappear` methods will be automatically called. Again, you can provide your own customization of these methods to perform additional operations.

> **Note:** You have to add the *override* keyword before "func viewDidLoad()". This method is originally provided by UIViewController. In order to provide customizations, we indicate to *override* its default implementation by using the *override* keyword.

Behind the Scene of the Run Button

One final thing I would like to talk about is the Run button. When you click the Run button, Xcode automatically launches the simulator and runs your app. But what happens behind the scene? As a developer, you have to look at all the pieces.

The entire process can be broken into three phases: *compile*, *package* and *run*.

- **Compile** – You probably think iOS understands Swift code. In reality, iOS only reads machine code. The Swift code is for developers to write and read. To make iOS understand the source code of the app, it has to go through a translation process to translate the Swift code into machine code. This process is referred as "compile". Xcode already comes with a built-in compiler to compile the source code.
- **Package** – Other than source code, an app usually contains resource files such as images, text files, sound files, etc. All these resources are packaged to make up the final app. We used to refer to these two processes as the "build" process.
- **Run** – This actually launches the simulator and loads your app.

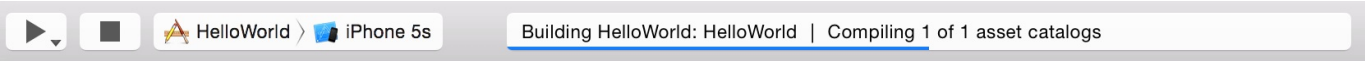


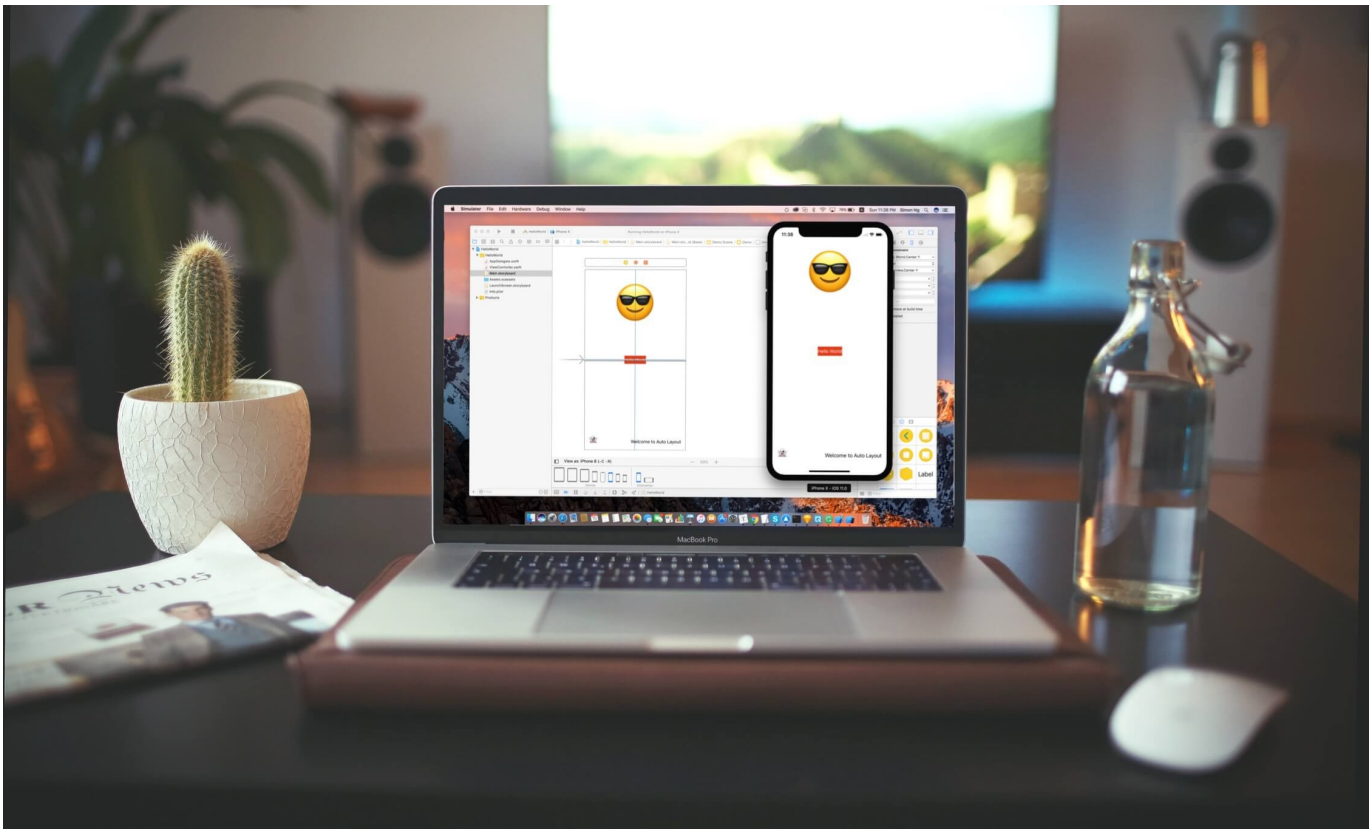
Figure 4-8. Build Process

Summary

You should now have a basic understanding of how the Hello World app works. As a beginner without any prior programming experience, it's not easy to understand all the programming concepts we just discussed. No worries. As you write more code and develop a real app in the upcoming chapters, you'll have a better idea about Swift and iOS programming.

Love this chapter? Consider to purchase the full version of the book and access our private Facebook group to get support. Please check out the full copy here.

Chapter 5
Introduction to Auto Layout



> Life is short. Build stuff that matters. – Siqi Chen

Wasn't it fun to create the Hello World app? Before we move onto building a real app, we will look into Auto Layout in this chapter.

Auto layout is a constraint-based layout system. It allows developers to create an adaptive UI that responds appropriately to changes in screen size and device orientation. Some beginners find it hard to learn. Even some developers avoid using it. But believe me, you won't be able to live without it when developing an app today.

When iPhone was first released over ten years ago, it only came with a single screen size: 3.5-inch. Later we had the 4-inch iPhone. In September 2014, Apple introduced the iPhone 6 and 6 Plus. Now Apple's iPhones are available in different screen sizes including

3.5-inch, 4-inch, 4.7-inch, 5.5-inch and 5.8-inch displays. When you design your app UI, you have to cater for all these screen sizes. If your app is going to support both iPhone and iPad (also known as a universal app), you will need to make sure the app fits additional screen sizes including 7.9-inch, 9.7-inch, 10.5-inch, and 12.9-inch.

Without using auto layout, it would be very hard for you to create an app that supports all screen resolutions. This is why I want to teach you auto layout at the very beginning of this book, rather than jumping right into coding a real app. It will take you some time to master the skill but I want you to understand the underlying concept. In this chapter and the one that follows, I want to help you build a solid foundation on designing an adaptive user interface.

> **Quick note:** Auto layout is not as difficult as some developers thought. Once you understand the basics, together with stack views you will learn in the next chapter, you will be able to use auto layout to create complex user interfaces for all types of iOS devices.

Why Auto Layout?

Let me give you an example, and you'll have a better idea why auto layout is needed. Open the HelloWorld project you built in chapter 3. Instead of running the app on iPhone 8 simulator, run it using the iPhone SE, 7/8 Plus or iPhone X simulators. You'll end up with the results illustrated in figure 5-1. It turns out that the button isn't centered when running on other iPhone devices, except the one with a 4.7-inch screen.

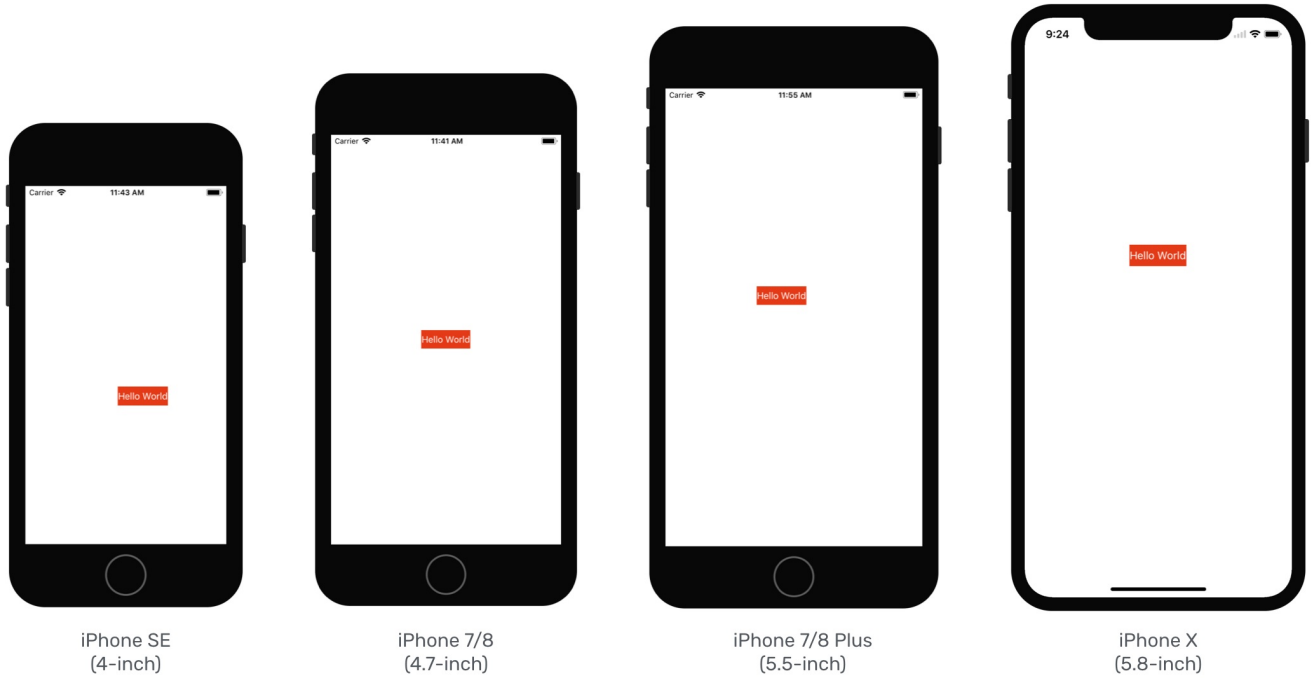


Figure 5-1. The app UI looks different when running on iPhone SE (4-inch), iPhone 7 (4.7-inch) and iPhone 7 Plus (5.5-inch)

Let's try one more thing.

Click the Stop button and run the app using the iPhone 8 simulator. After the simulator launches, go up to the menu and select Hardware > Rotate Left (or Rotate Right) from the menu. This rotates the device to landscape mode. Alternatively, you can press command+left arrow/right arrow to rotate the device sideway. Again, the Hello World button is not centered.

Why? What's wrong with it?

As you know, the iPhone devices have different screen dimensions:

- For iPhone 5/5s/SE, the screen in portrait mode consists of 320 points (or 640 pixels) horizontally and 568 points (or 1136 pixels) vertically.
- For iPhone 6/6s/7/8, the screen consists of 375 points (or 750 pixels) horizontally and 667 points (or 1334 pixels) vertically.
- For iPhone 6/6s/7/8 Plus, the screen consists of 414 points (or 1242 pixels)

horizontally and 736 points (or 2208 pixels) vertically.
- For the brand-new iPhone X, the screen consists of 375 points (or 1125 pixels) horizontally and 812 points (or 2436 pixels) vertically.
- For iPhone 4s, the screen consists of 320 points (or 640 pixels) and 480 points (or 960 pixels).

Why Points instead of Pixels?

Back in 2007, Apple introduced the original iPhone with a 3.5-inch display with a resolution of 320x480. That is 320 pixels horizontally and 480 pixels vertically. Apple retained this screen resolution with the succeeding iPhone 3G and iPhone 3GS. Obviously, if you were building an app at that time, one point corresponds to one pixel. Later, Apple introduced iPhone 4 with retina display. The screen resolution was doubled to 640x960 pixels. So one point corresponds to two pixels for retina display.

The point system makes our developers' lives easier. No matter how the screen resolution is changed (say, the resolution is doubled again to 1280x1920 pixels), we still deal with points and the base resolution (i.e. 320x480 for iPhone 4/4s or 320x568 for iPhone 5/5s/SE). The translation between points and pixels is handled by iOS.

Without using auto layout, the position of the button we lay out in the storyboard is fixed. In other words, we "hard-code" the frame origin of the button. In our example, the "Hello World" button's frame origin is set to (147, 318). Therefore, whether you're using a 4-inch or 4.7-inch or 5.5-inch simulator, iOS draws the button in the specified position. Figure 5-2 illustrates the frame origin on different devices. This explains why the "Hello World" button can only be centered on iPhone 7, and it is shifted away from the screen center on other iOS devices, as well as, in landscape orientation.

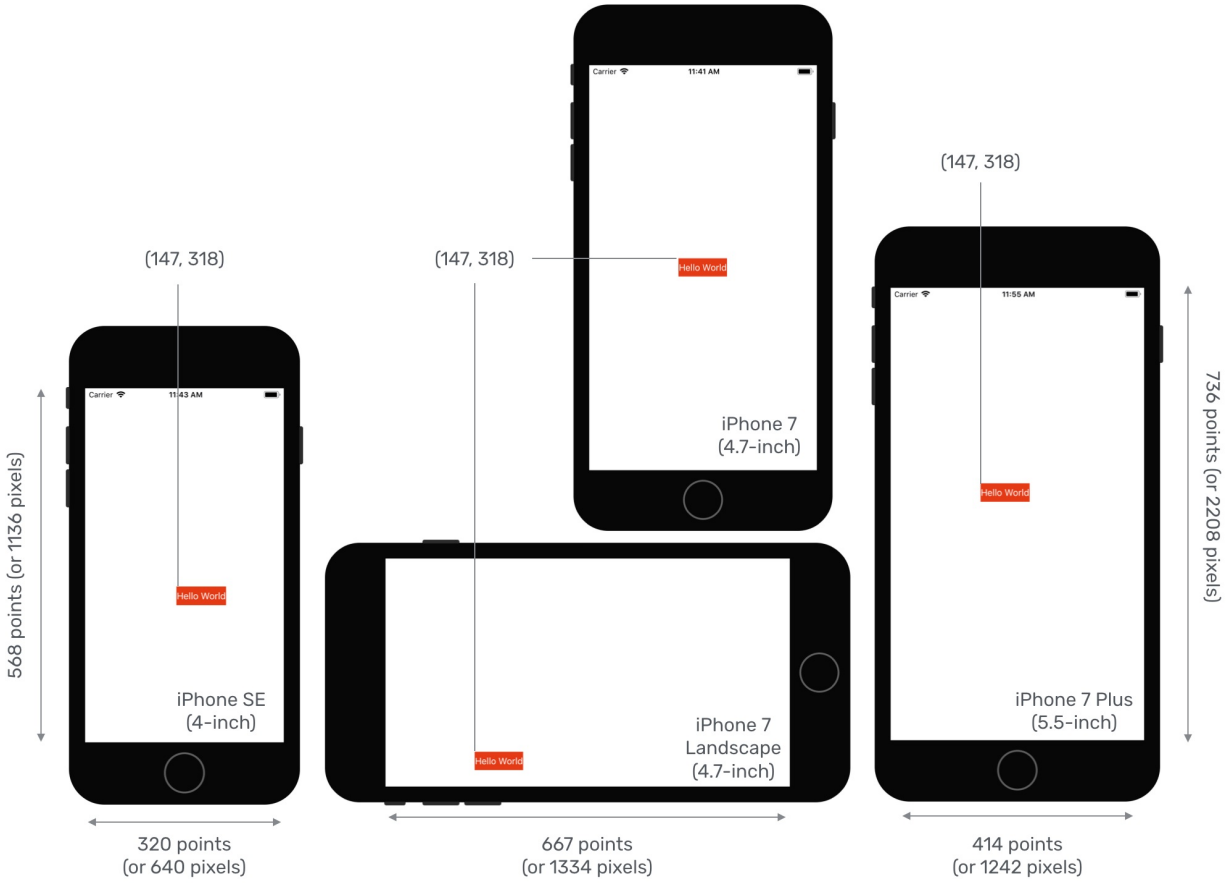


Figure 5-2. How the button is displayed on iPhone 7 Plus, iPhone 7 and iPhone SE

Obviously, we want the app to look good on all iPhone models, and in both portrait & landscape orientation. This is why we have to learn auto layout. It's the answer to the layout issues, that we have just talked about.

Auto Layout is All About Constraints

As mentioned before, auto layout is a constraint-based layout system. It allows developers to create an adaptive UI that responds appropriately to changes in screen size and device orientation. Okay, it sounds good. But what does the term "constraint-based layout" mean?

Let me put it in a more descriptive way. Consider the "Hello World" button again, how do you describe its position if you want to place the button at the center of the view? You would probably describe it like this:

The button should be centered both horizontally and vertically, regardless of the screen resolution and orientation.

Here you actually define two constraints:

- center horizontally
- center vertically

These constraints express rules for the layout of the button in the interface.

Auto layout is all about constraints. While we describe the constraints in words, the constraints in auto layout are expressed in mathematical form. For example, if you're defining the position of a button, you might want to say "the left edge should be 30 points from the left edge of its containing view." This translates to `button.left = (container.left + 30)`.

Fortunately, we do not need to deal with the formulas. All you need to know is how to express the constraints descriptively and use Interface Builder to create them.

Okay, that's quite enough for the auto layout theory. Now let's see how to define layout constraints in Interface Builder to center the "Hello World" button.

Live Previewing in Interface Builder

First, open `Main.storyboard` of your HelloWorld project (or download it from http://www.appcoda.com/resources/swift42/HelloWorld.zip). Before we add the layout constraints to the user interface, let me introduce a handy feature in Xcode.

You can test the app UI in simulators to see how it looks in different screen sizes. However, Xcode provides a configuration bar in Interface Builder for developers to live preview the user interface.

By default, Interface Builder is set to preview the UI on iPhone 8 (4.7-inch). To see how your app looks on other devices, click `View as: iPhone 8` button to reveal the configuration bar and then choose your preferred iPhone/iPad devices to test. You can also alter the device's orientation to see how it affects your app's UI. Figure 5-3 shows a live preview of the Hello World app on iPhone 8 in landscape orientation.

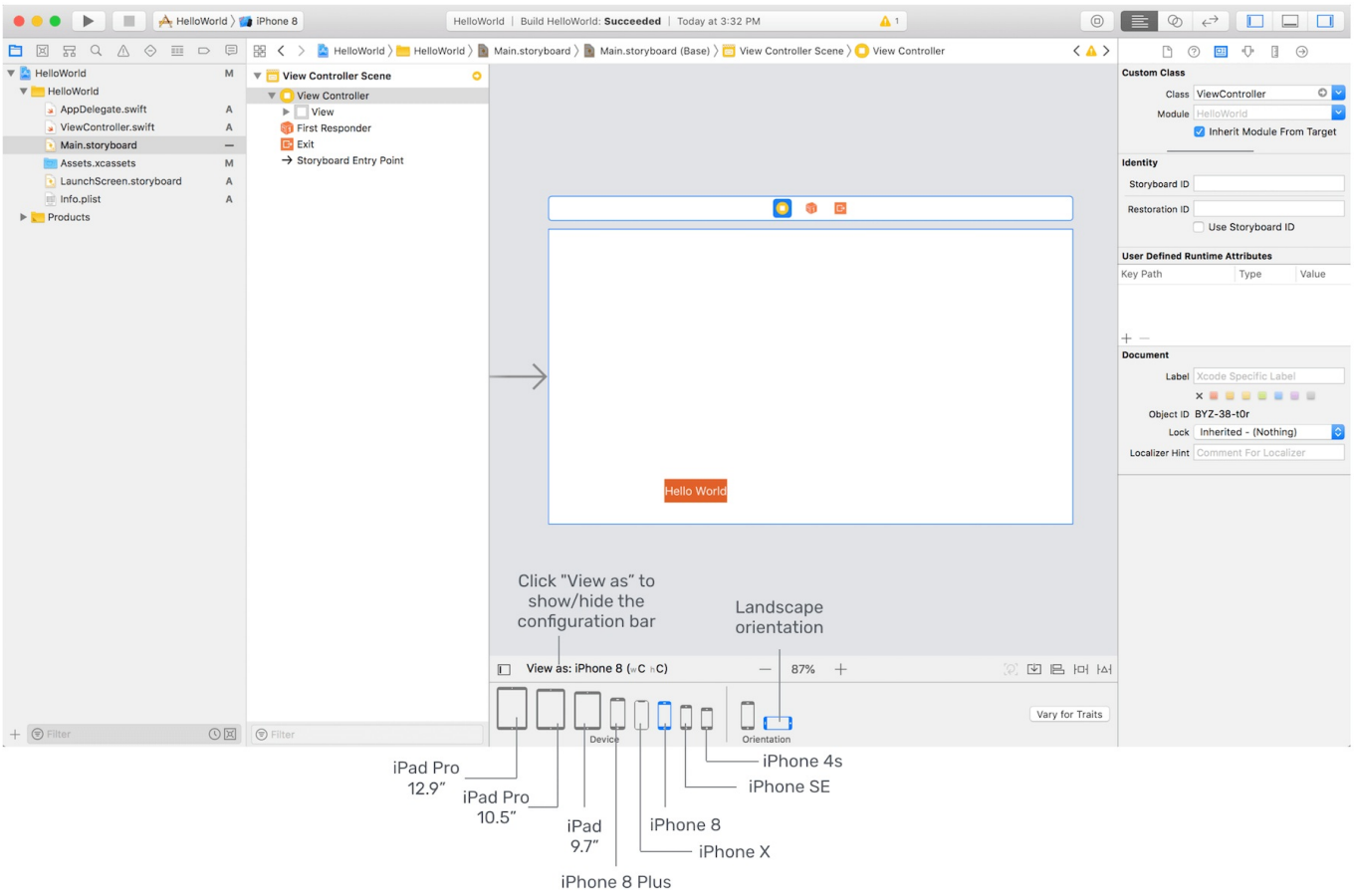


Figure 5-3. Live preview using Xcode's configuration bar

The configuration bar is a great feature in Xcode introduced since version 8. Take some time to play around with it.

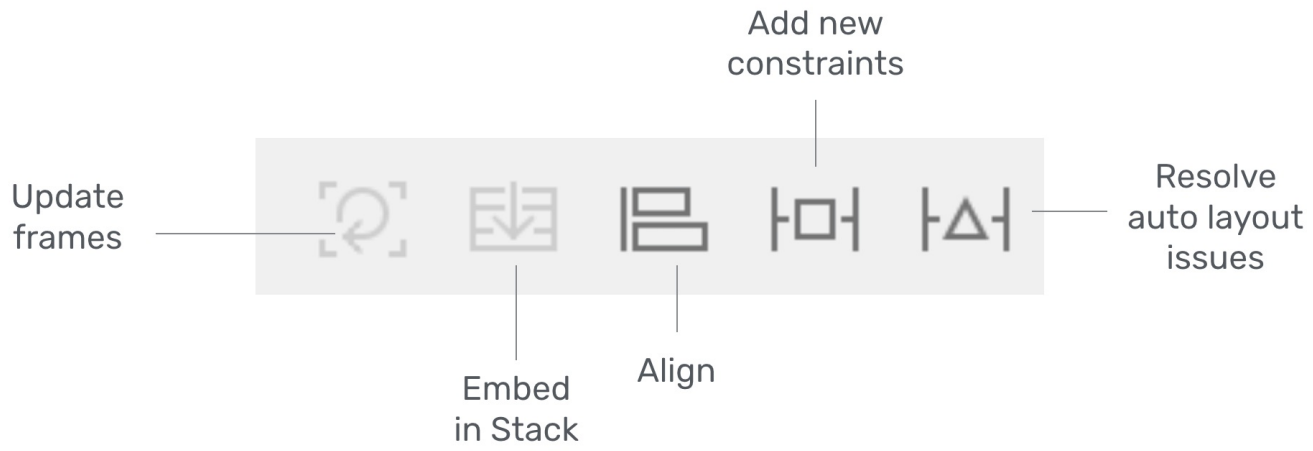
> **Quick note:** You may wonder what "(wC hC)" means. Just forget about it right now and focus on learning auto layout. I will discuss about it in the later chapter.

Using Auto Layout to Center the Button

Now let's continue to talk about auto layout. Xcode provides two ways to define auto layout constraints:

1. Auto layout bar
2. Control-drag

We'll demonstrate both approaches in this chapter. First, we begin with the auto layout bar. At the bottom-right corner of the Interface Builder editor, you should find 5 buttons. These buttons are from the layout bar. You can use them to define various types of layout constraints and resolve layout issues.



Each button has its own function:

- **Align** – Create alignment constraints, such as aligning the left edges of two views.
- **Add new constraints** – Create spacing constraints, such as defining the width of a UI control.
- **Resolve auto layout issues** – Resolve layout issues.
- **Stack** – Embed views into a stack view. We will further discuss it in the next chapter.
- **Update frames** - Update the frame's position and size in reference to the given layout constraints.

As discussed earlier, to center the "Hello World" button, you have to define two constraints: *center horizontally* and *center vertically*. Both constraints are with respect to the view.

To create the constraints, we use the Align button. First, select the Hello World button in Interface Builder and then click the Align icon in the layout bar. In the pop-over menu, check both "Horizontal in container" and "Vertically in container" options. Then click the "Add 2 Constraints" button.

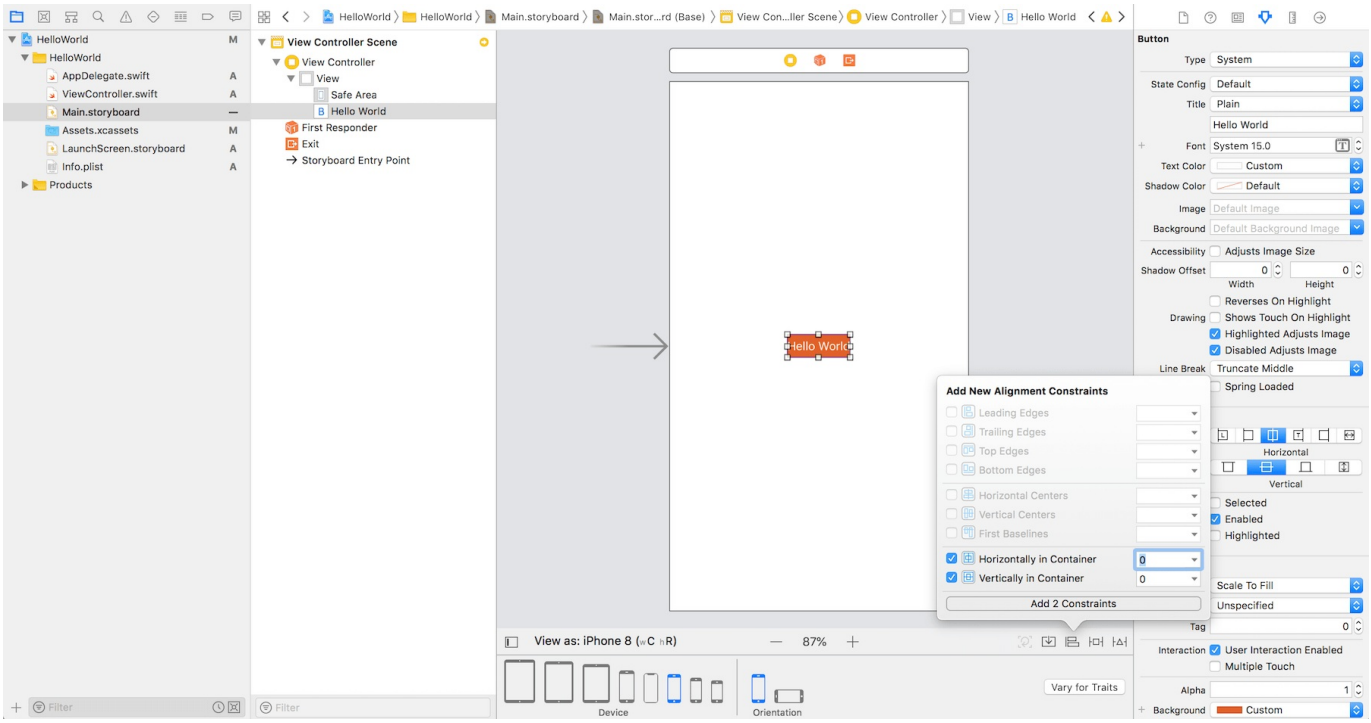


Figure 5-5. Adding constraints using Align button

> **Quick tip:** You can press command+0 to hide the project navigator. This will free up more screen space for you to work on the app design.

You should now see a set of constraint lines in blue. If you expand the Constraints option in the document outline view, you will find two new constraints for the button. These constraints ensure the button is always positioned at the center of the view. Alternatively, you can view these constraints in the Size inspector.

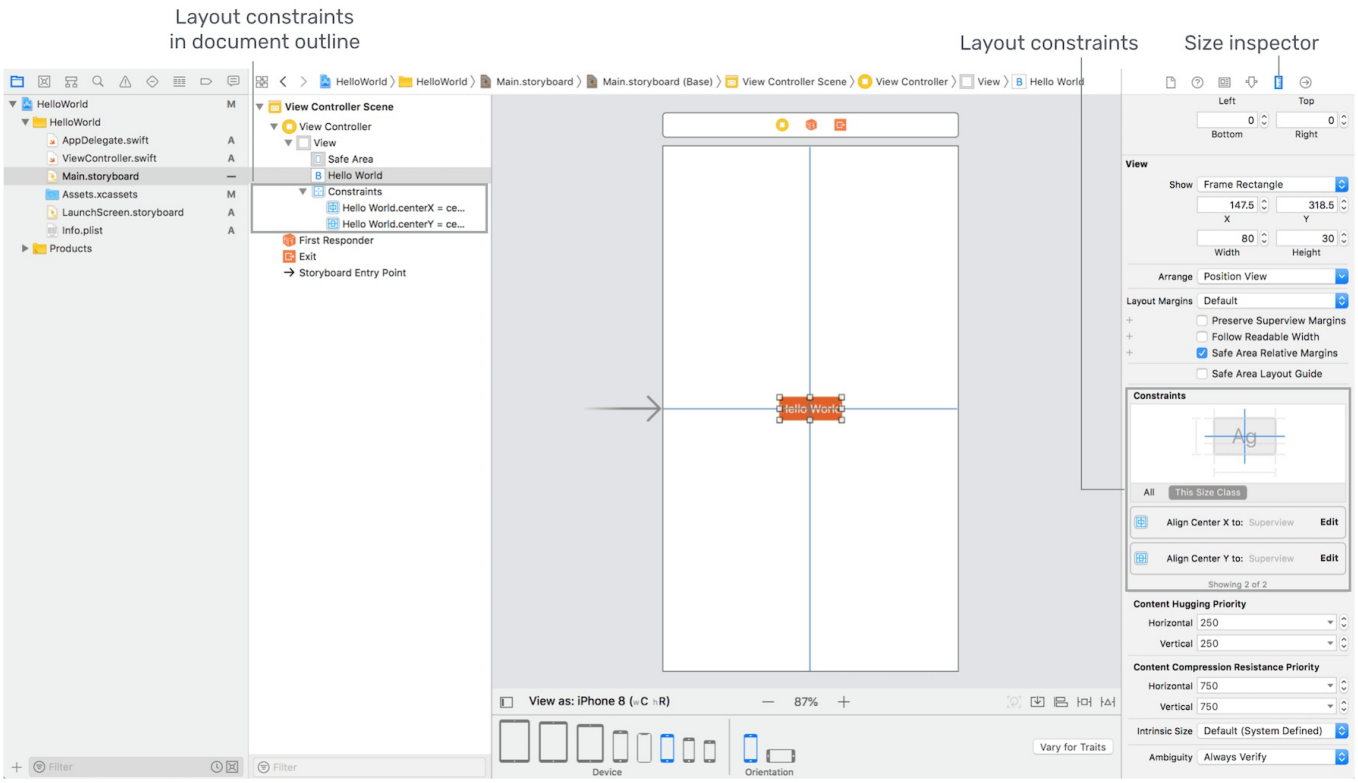


Figure 5-6. View the constraints in Document Outline and Size Inspector

> **Quick note:** When your view layout is being configured correctly and there is no ambiguity, the constraint lines are in blue.

Okay, you're ready to test the app. You can click the Run button to launch the app on iPhone 7/8 Plus (or iPhone SE). Alternatively, just switch to another device or change the device's orientation by using the configuration bar to verify the layout. The button should be centered perfectly, regardless of screen size and orientation.

Resolving Layout Constraint Issues

The layout constraints that we have just set are perfect. But that is not always the case. Xcode is intelligent enough to detect any constraint issues.

Now, try to drag the Hello World button to the lower-left part of the screen. Xcode immediately detects some layout issues and the corresponding constraint lines turn orange that indicates a misplaced item.

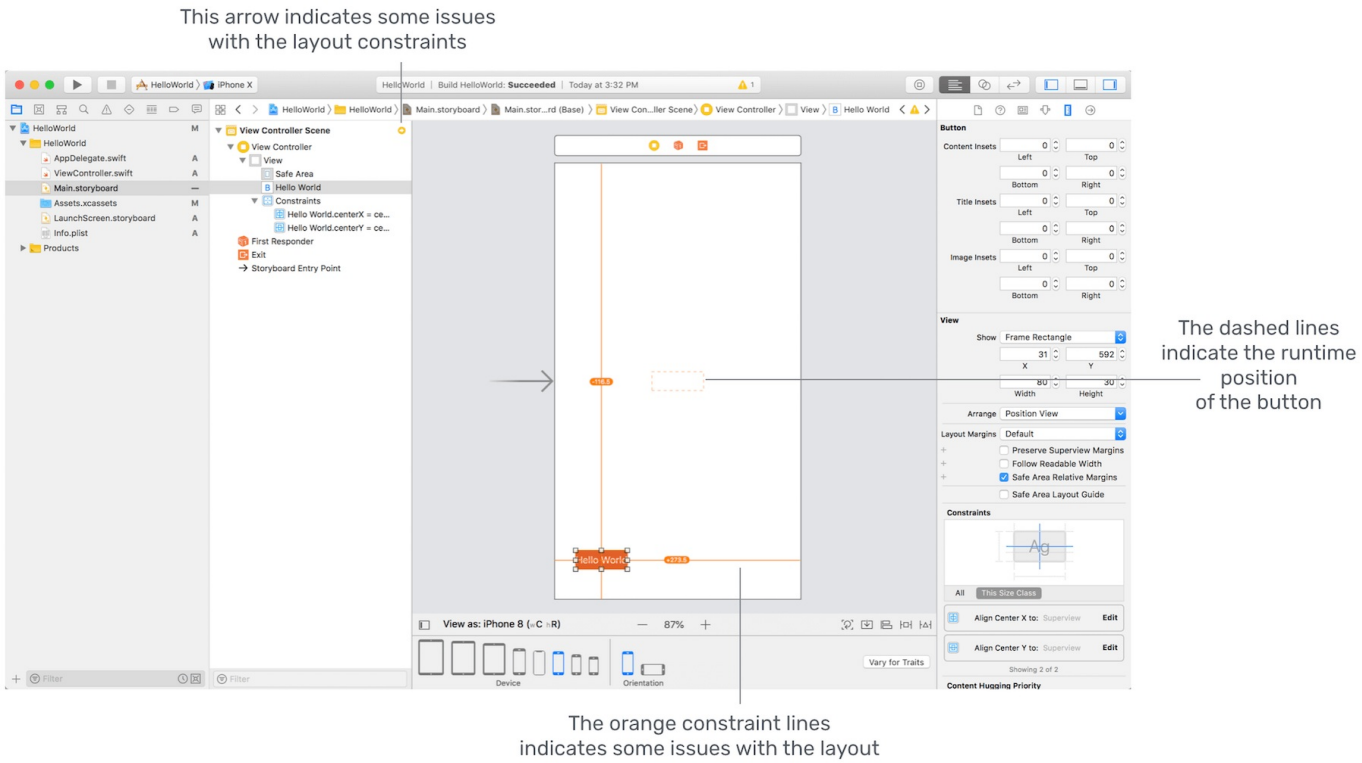
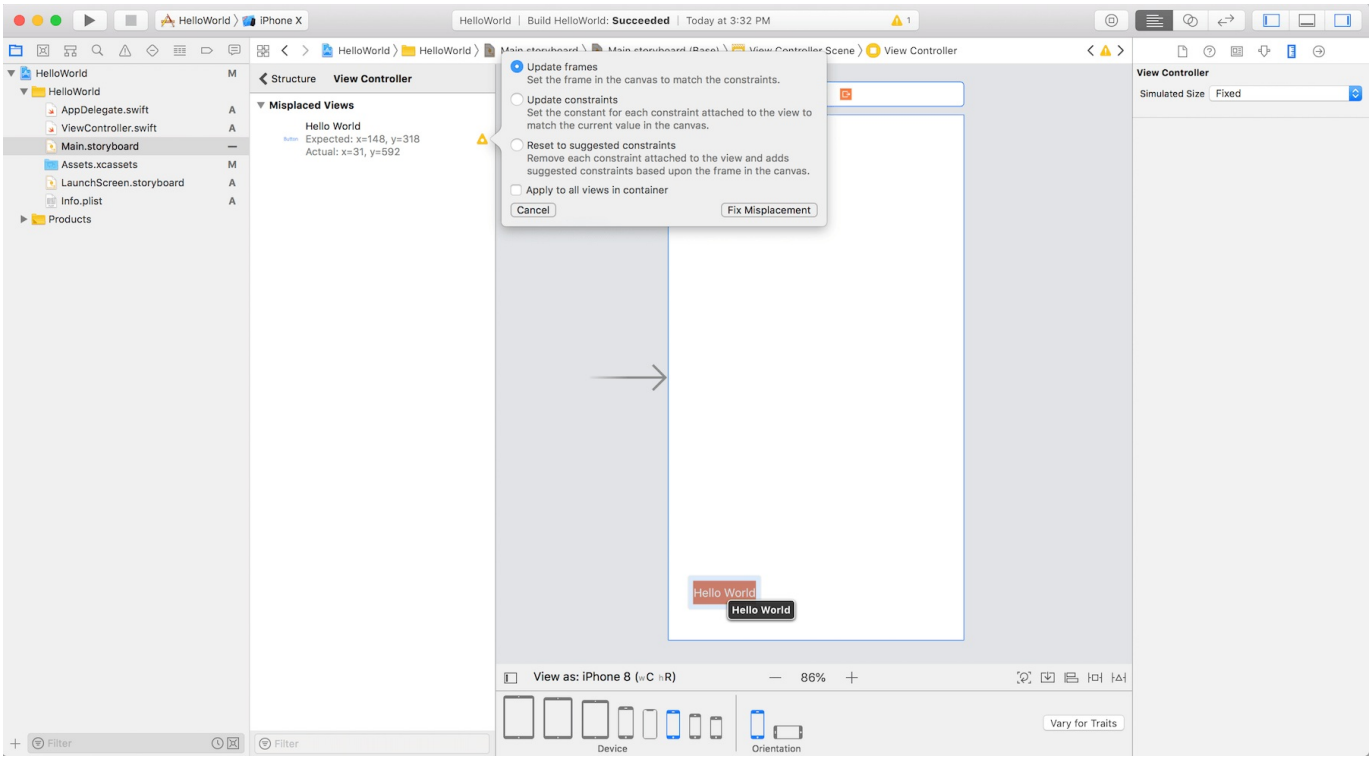


Figure 5-7. Interface Builder uses orange/red lines to indicate Auto Layout issues

Auto layout issues occur when you create ambiguous or conflicting constraints. Here we said the button should be vertically and horizontally centered in the container (i.e. the view). However, the button is now placed at the lower-left corner of the view. Interface Builder found this confusing, therefore it uses orange lines to indicate the layout issues. The dash lines indicate the expected position of the button.

When there is any layout issue, the Document Outline view displays a disclosure arrow (red/orange). Now click the disclosure arrow to see a list of the issues. For layout issues like this one, Interface Builder is smart enough to resolve the layout issues for us. Click the indicator icon next to the issue and a pop-over shows you a number of solutions. In this case, select the "Update frames" option and click "Fix Misplacement" button. The button will then be moved to the center of the view.



Alternatively, you can simply click the "Update frames" button of the layout bar to resolve the issue.

This layout issue was triggered manually. I just wanted to demonstrate how to find the issues and fix them. As you go through the exercises in the later chapters, you will probably face a similar layout issue. You should know how to resolve layout issues easily and quickly.

An Alternative Way to Preview Storyboards

While you can use the configuration bar to live preview your app UI, Xcode provides an alternate Preview feature for developers to preview the user interface on different devices simultaneously.

In Interface Builder, open the Assistant pop-up menu > Preview (1). Now press and hold the `option` key, then click Main.storyboard (Preview). You can refer to figure 5-9 for the steps.

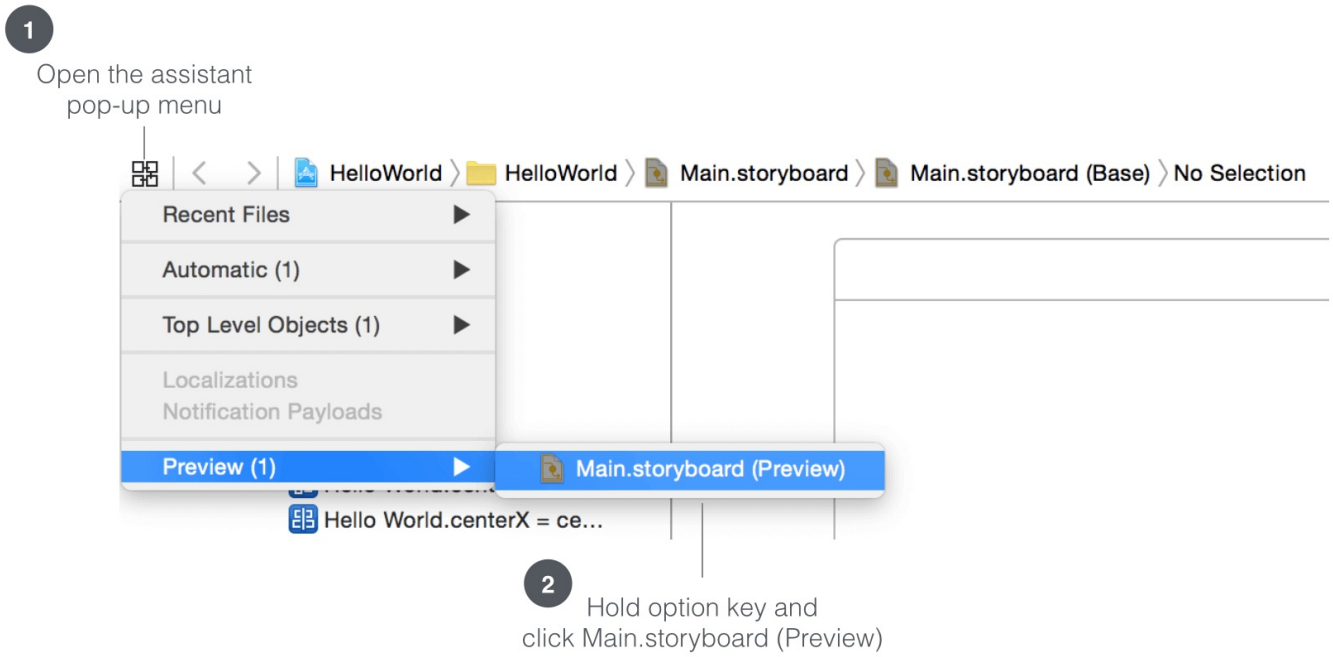
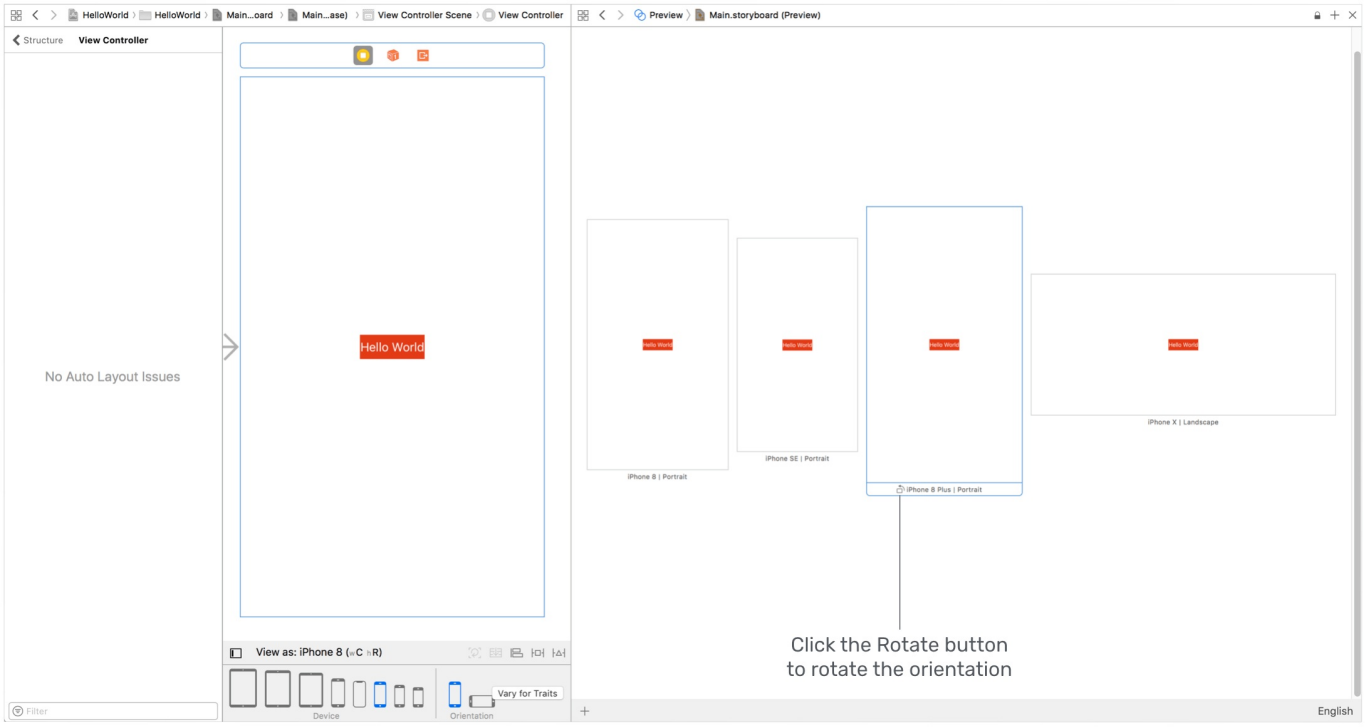


Figure 5-9. Assistant Pop-up Menu

Xcode will display a preview of your app's UI in the assistant editor. By default, it shows you the preview on the iPhone 8 device. You can click the + button at the lower-left corner of the assistant editor to add other iOS devices (e.g. iPhone SE/8 Plus) for preview. If you want to see how the screen looks like in landscape orientation, simply click the rotate button. The Preview feature is extremely useful for designing your app's user interface. You can make changes to the storyboard (say, adding another button to the view) and see how the UI looks on the chosen devices all at once.



If you want to free up some more screen space for the preview pane, hold both command and option keys and then press 0 to hide the Utility area.

> **Quick tip:** When you add more devices in the preview assistant, Xcode may not be able to fit the preview of all devices sizes into the screen at the same time. If you're using a trackpad, you can scroll through the preview by swiping left or right with two fingers. What if you're still using a mouse with a scroll wheel? Simply hold the shift key to scroll horizontally.

Adding a Label

Now that you have some idea about auto layout and the preview feature, let's add a label to the lower-right part of the view and see how to define the layout constraints for the label. Labels in iOS are usually used for displaying simple text and messages.

In the Interface Builder editor, click the Object library button to open the Object library. Drag a label from the Object library and place it near the lower-right corner of the view. Double-click the label and change it to "Welcome to Auto Layout" or whatever title you

want.

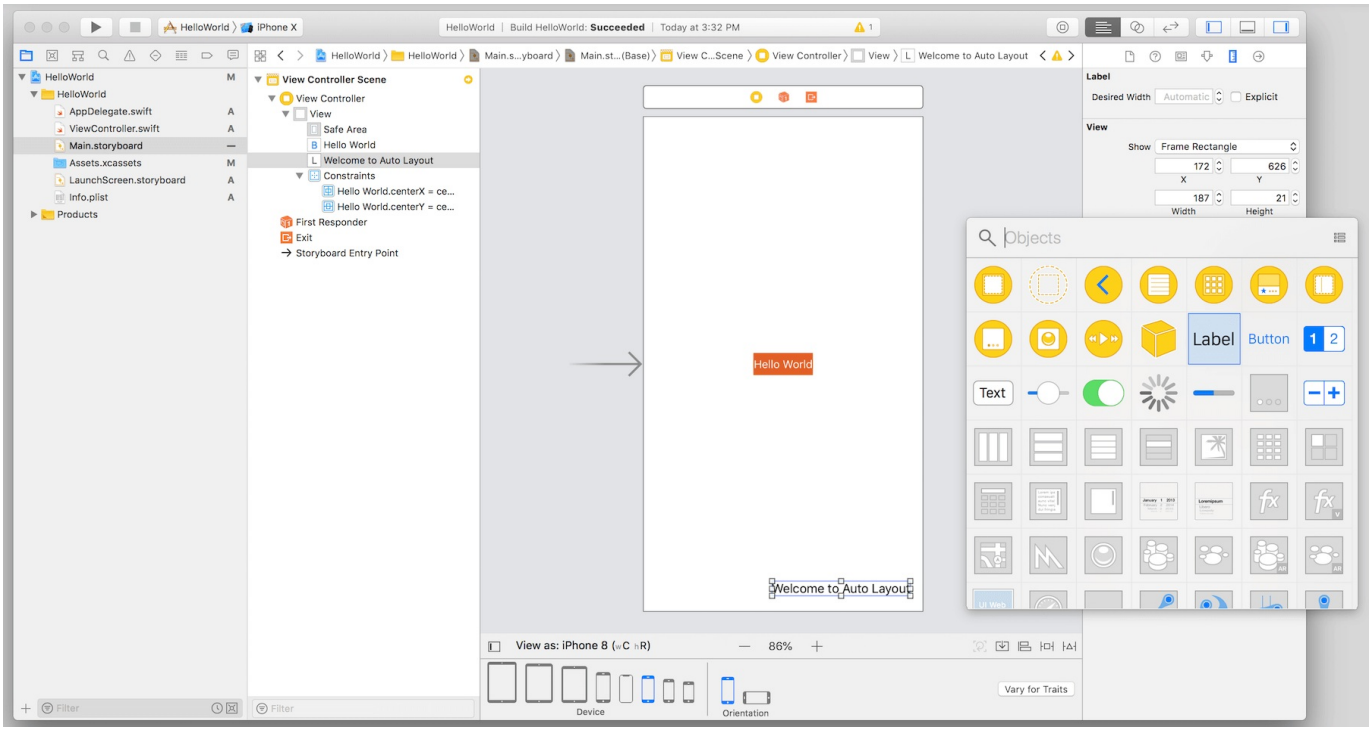


Figure 5-11. Adding a label to the view

If you opened the preview assistant again, you should see the UI change immediately (see figure 5-12). Without defining any layout constraints for the label, you are not able to display the label on all iPhone devices except iPhone 8 and 8 Plus.

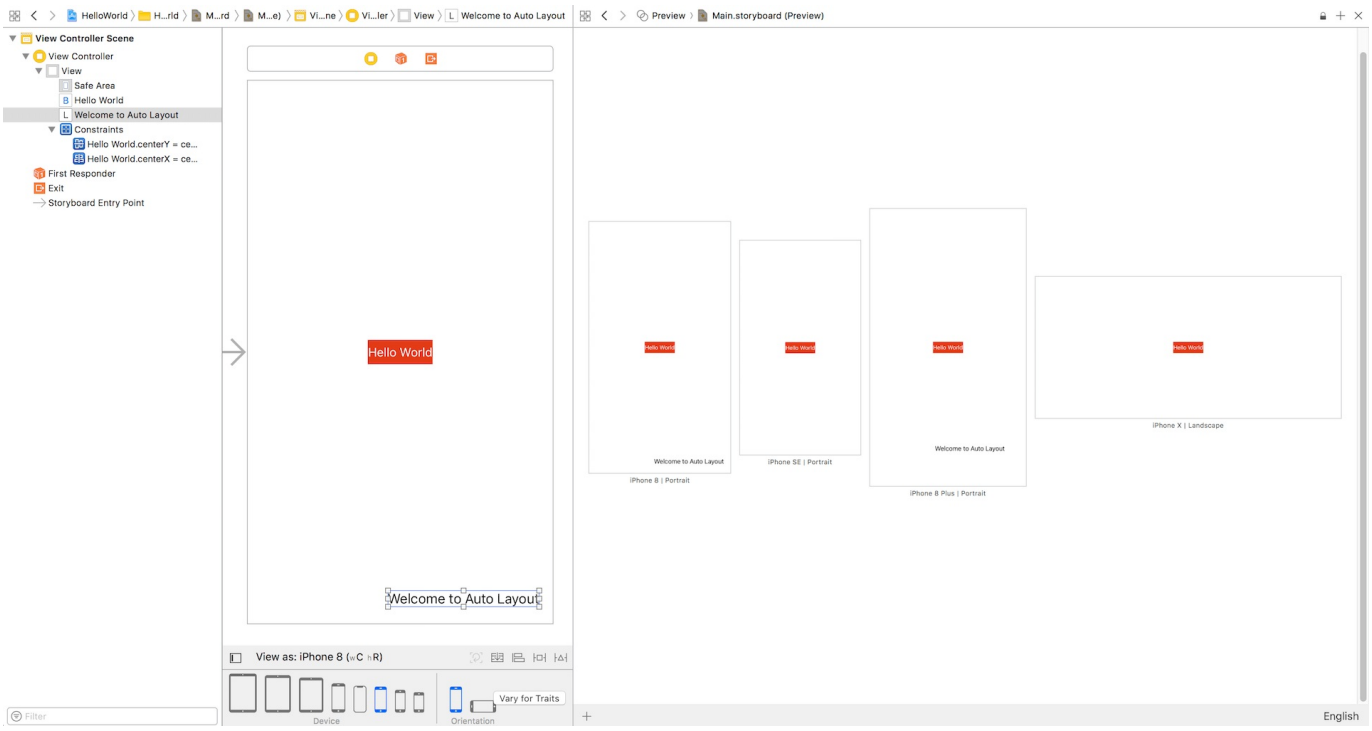


Figure 5-12. The label can't be displayed properly

How can you resolve this issue? Obviously, we need to set up a couple of constraints to make it work properly. The question is: *what constraints should we add?*

Let's try to describe the requirement of the label in words. You probably describe it like this:

The label should be placed at the lower-right corner of the view.

That's okay, but not precise enough. A more precise way to describe the location of the label is like this:

The label is located 0 points away from the right margin of the view and 20 points away from the bottom of the view.

This is much better. When you describe the position of an item precisely, you can easily come up with the layout constraints. Here, the constraints of the label are:

1. The label is 0 points away from the right margin of the view.

2. The label is 20 points away from the bottom of the view.

In auto layout, we refer this kind of constraints as *spacing constraints*. To create these spacing constraints, you can use the "Add new constraints" button of the layout button. But this time we'll use the Control-drag approach to apply auto layout. In Interface Builder, you can *control-drag* from an item to itself or to another item along the axis for which you want to add constraints.

To add the first spacing constraint, hold the control key and drag from the label to the right until the view becomes highlighted in blue. Now release the button, you'll see a pop-over menu showing a list of constraint options. Select "Trailing space to Safe Area" to add a spacing constraint from the label to the view's right margin.

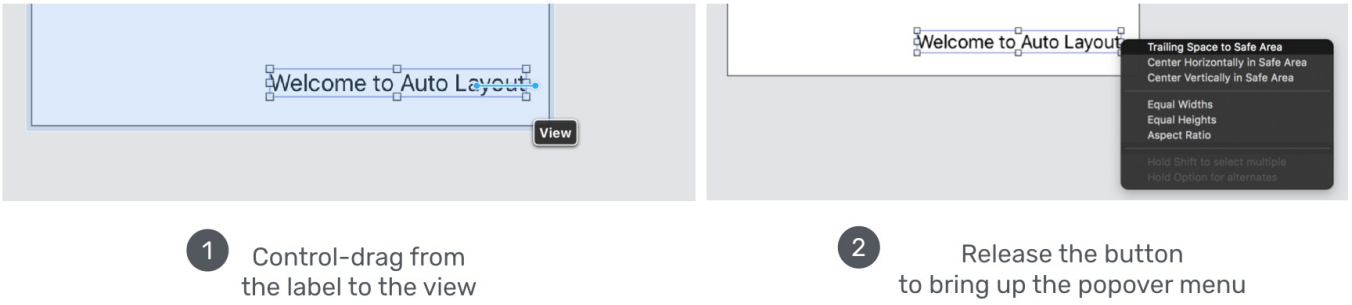


Figure 5-13. Using Control-drag to add the first constraint

In the document outline view, you should see the new constraint. Interface Builder now displays constraint lines in red indicating that there are some missing constraints. That's normal as we haven't defined the second constraint.

Now control-drag from the label to the bottom of the view. Release the button and select "Bottom Space to Safe Area" in the shortcut menu. This creates a spacing constraint from the label to the bottom layout guide of the view.

Figure 5-14. Using Control-drag to add the second constraint

Once you added the two constraints, all constraint lines should be in solid blue. When you preview the UI or run the app in the simulator, the label should display properly on all screen sizes, and even in landscape mode.

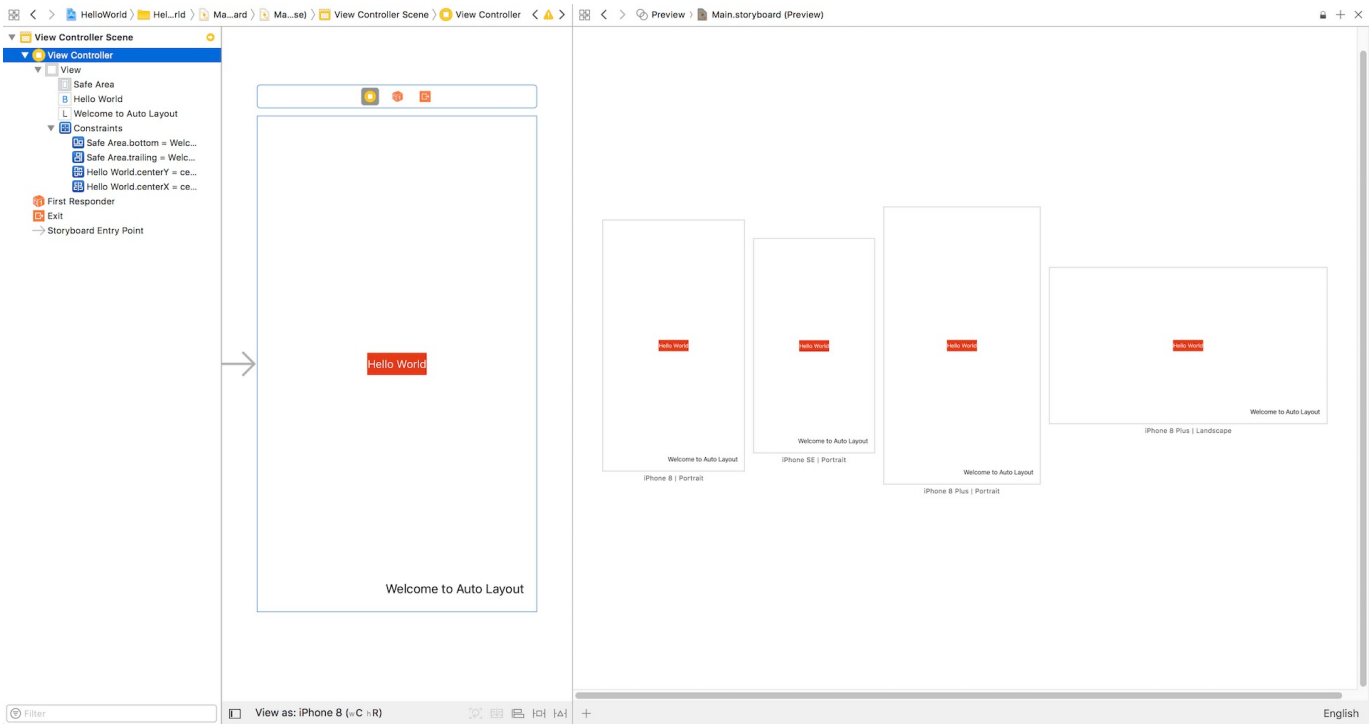


Figure 5-15. The UI now supports all screen sizes

Great! You've defined the constraints correctly. Having that said, you may notice the yellow indicator in the document outline. If you click the indicator, you will find a layout warning related to localization.

Why is that?

This is due to a feature introduced since the release of Xcode 9. Our demo app now only supports English. The layout constraints we defined work perfectly for English. But what if it requires to support other languages? Will the current layout constraints still work for the right-to-left languages (e.g. Arabic)?

In Xcode 10, Interface Builder automatically reviews your layout constraints and sees if they are suitable for all languages. If it finds some problems, it will issue a localization warning. To fix the issue, you can choose the second option to add the leading constraint.

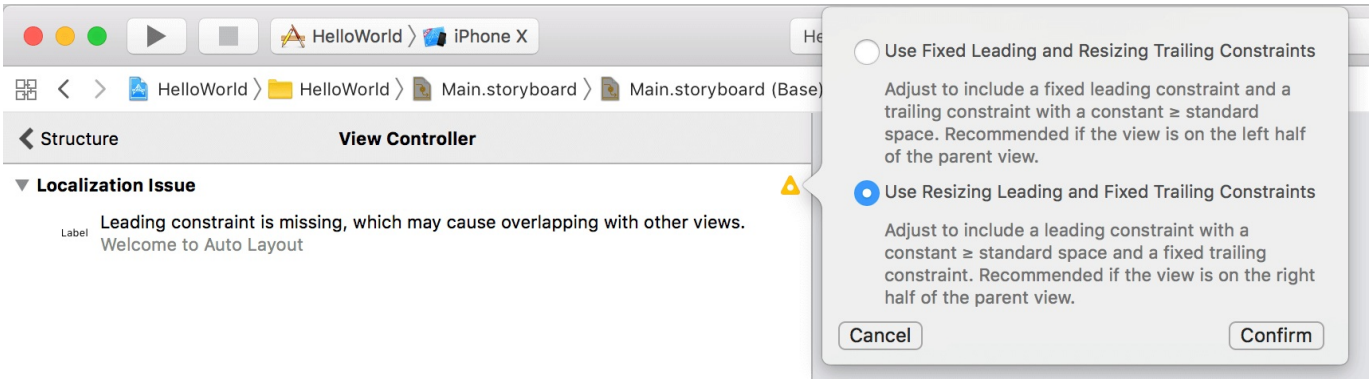


Figure 5-16. Layout issues related to localization

Safe Areas

In the document outline, did you notice an item called Safe Area? And, did you remember the spacing constraints we defined earlier were related to Safe Area? We defined two spacing constraints:

1. Trailing space to Safe Area
2. Bottom space to Safe Area

So, what is safe area?

Safe area is first introduced in Xcode 9 to replace top & bottom layout guides used in older versions of Xcode. Rather than explaining the term in words, it is better to show you what Safe Area is.

Go to the document outline and select Safe Area. The area in blue is the safe area. Safe area is actually a layout guide, representing the portion of your view that is unobscured by bars and other content. In the view shown in figure 5-17, the safe area is the entire view excluding the status bar.

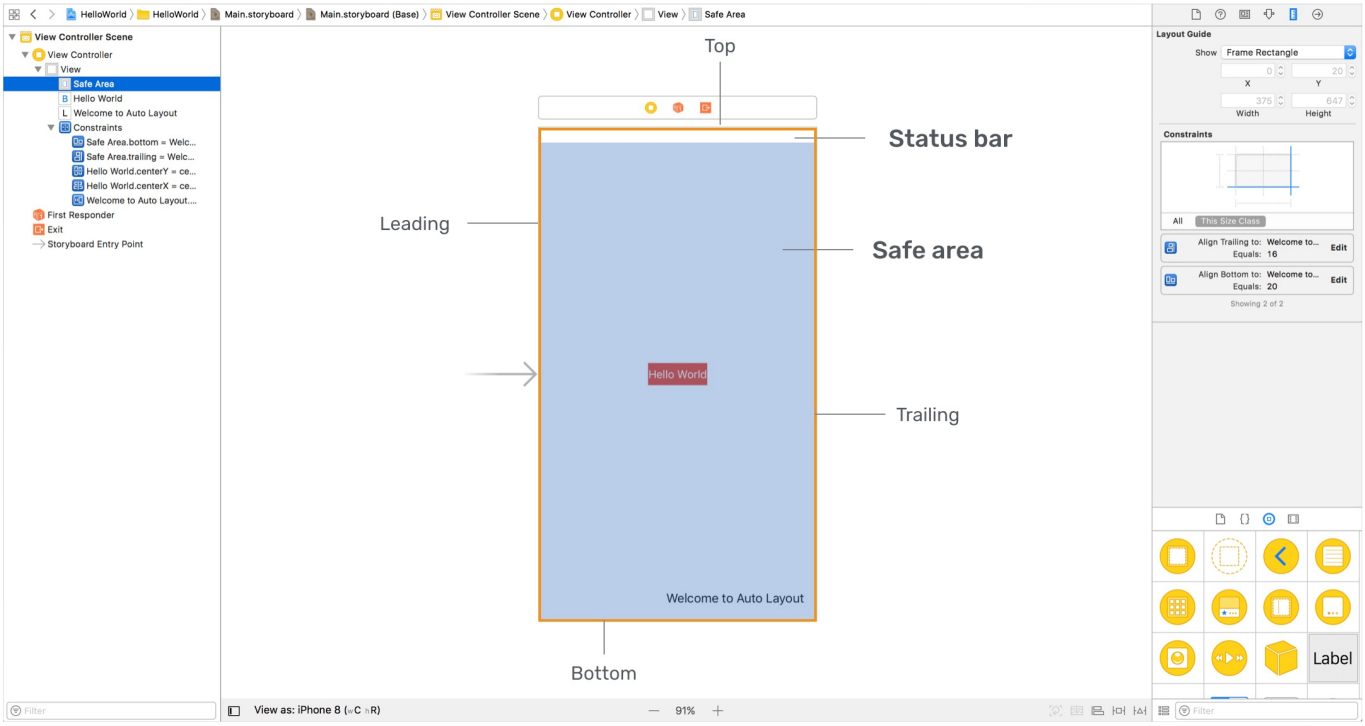


Figure 5-17. Safe area

The safe area layout guides help developers easier to work with layout constraints because the safe area updates itself automatically when the view is covered by navigation bar or other content.

Take a look at figure 5-18. The Hello World button is defined to place 20 points under the top anchor of the safe area. If the view doesn't have any navigation bar or tab bar, the safe area is the entire view excluding the status bar. Thus, the button is placed 20 points below the status bar.

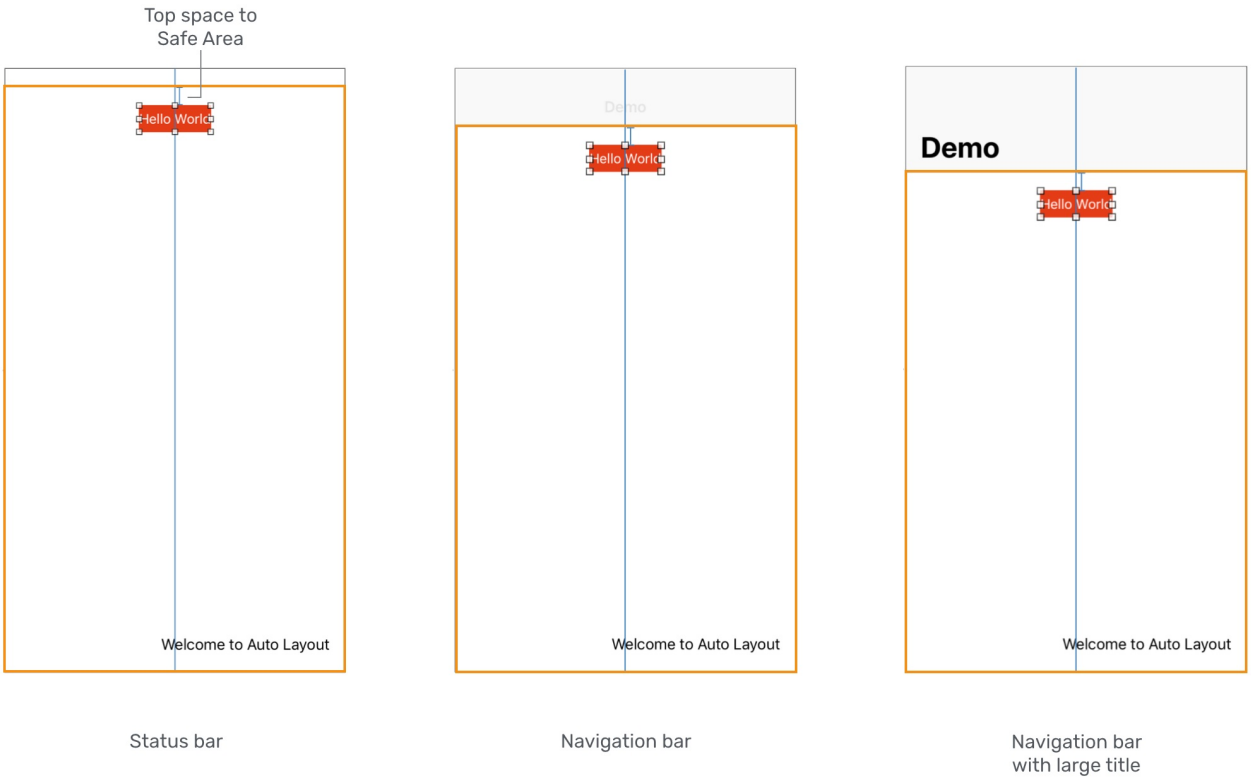


Figure 5-18. Safe areas adjust automatically

In case the view contains a navigation bar, whether it uses the standard title or large title in iOS 11, the safe area adjusts itself automatically. The button is then placed below the navigation bar. Therefore, as long as the UI object is constrained relative to the safe area layout guides, your interface will layout correctly even if you add a navigation or tab bar to the interface.

Editing Constraints

The "Welcome to Auto Layout" label is now located 16 points away from the trailing anchor of the safe area. What if you want to increase the space between the label and the right side of the view? Interface Builder provides a convenient way to edit the constant of a constraint.

You can simply choose the constraint in the document outline view or select the constraint directly. In the Attributes inspector, you can find the properties of this constraint including relation, constant, and priority. The constant is now set to `16`. You can change it to `30` to add some extra space.

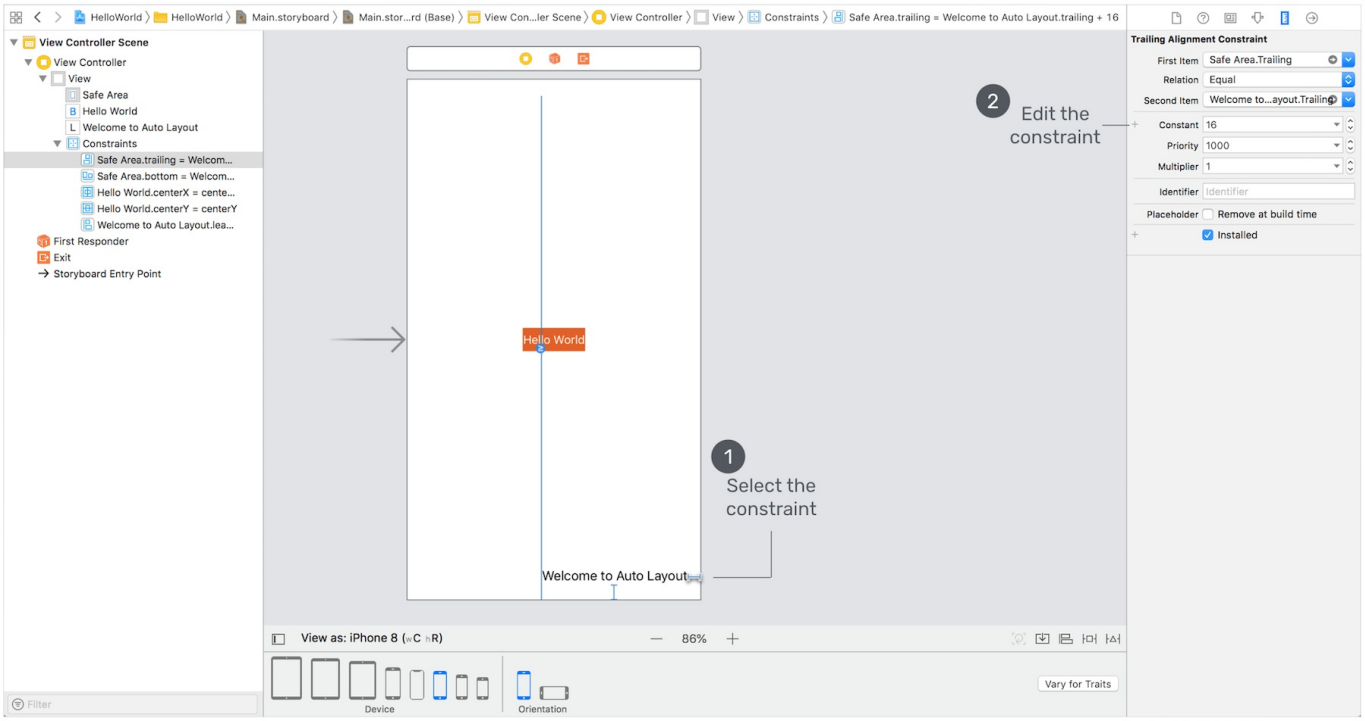


Figure 5-19. Editing a constraint using Attributes inspector

Alternatively, you can double-click the constraint to edit its properties through the popover menu.

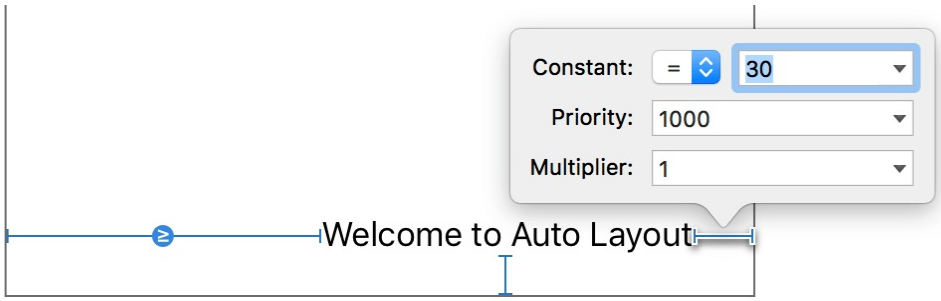


Figure 5-20. Editing a constraint by double clicking the constraint

Your Exercise

By now, I hope you should have some basic ideas about how to lay out your app UI and make it fit for all screen sizes. Let's have a simple exercise before moving on to the next chapter. All I need you to do is add two more emoji labels to the view. Figure 5-21 shows the expected result. Let me give you some hints:

- The "smiling face" emoji label should be 10 points away from the top anchor of the safe area, and centered horizontally.
- The "ghost" label has two spacing constraints.

You can adjust the font size of the label by editing its Font option in the Attributes inspector. However, even if you do not know how to do it, that's completely fine. I will show you how in later chapters. For now, just focus on defining the layout constraints.

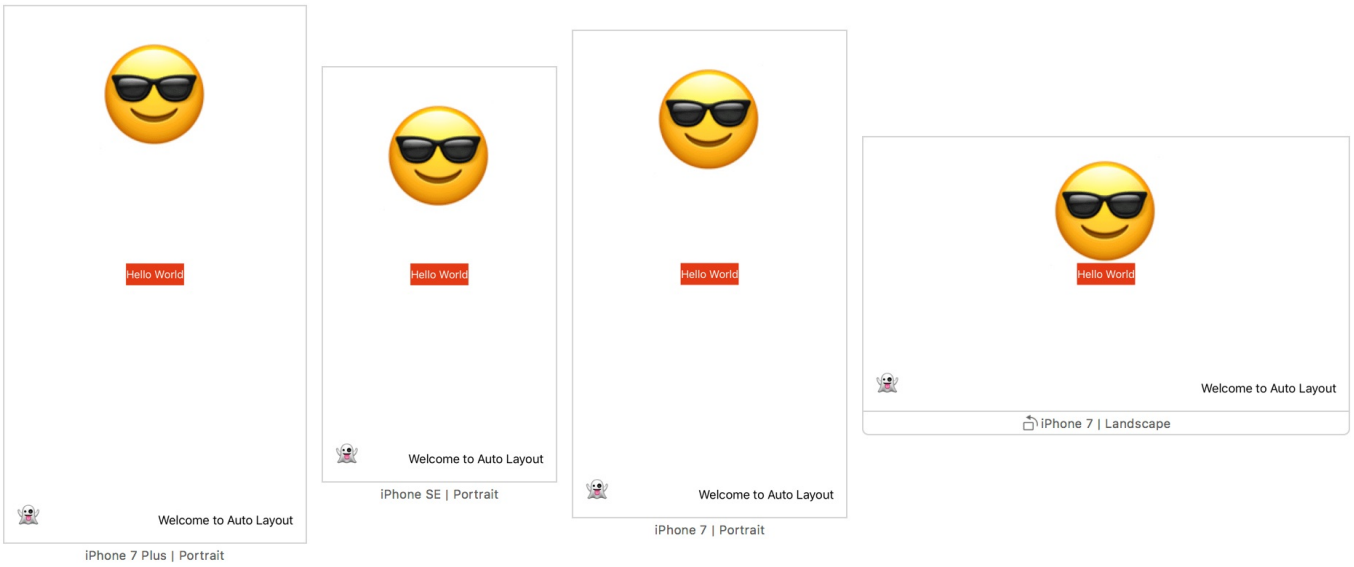


Figure 5-21. The expected app layout

Summary

In this chapter, we went through the basics of Auto Layout. It's just the basics because I don't want to scare you away from learning auto layout. As we dig deeper and create a real app, we'll continue to explore some other features of auto layout.

Most of the beginners (and even some experienced iOS programmers) avoid using auto layout because it looks confusing. If you thoroughly understand what I have covered in this chapter, you're on your way to becoming a competent iOS developer. The original iPhone was launched in 2007. Over these years, there have been tons of changes and advancements in the area of iOS development. Unlike the good old days when your apps just needed to run on a 3.5-inch, non-retina device, your apps now have to cater to various screen resolution and sizes. This is why I devoted a whole chapter to Auto Layout.

So take some time to digest the materials.

To continue reading and access the full source code, please get the full copy of the book here.

Chapter 6
Designing UI Using Stack Views

> To the user, the interface is the product.
>
> - Aza Raskin

I have given you a brief overview of auto layout. The examples that we have worked on were pretty easy. However, as your app UI becomes more complex, you will find it more difficult to define the layout constraints for all UI objects. Starting from iOS 9, Apple introduced a powerful feature called Stack Views that would make our developers' life a little bit simpler. You no longer need to define auto layout constraints for every UI objects. Stack views will take care of most of that.

In this chapter, we will continue to focus on discussing UI design with Interface Builder. I will teach you how to build a more comprehensive UI, which you may come across in a real-world application. You will learn how to:

1. Use stack views to lay out user interfaces.
2. Use image views to display images.
3. Manage images using the built-in asset catalog.
4. Adapt stack views using Size Classes.

On top of the above, we will explore more about auto layout. You'll be amazed how much you can get done without writing a line of code.

What is a Stack View

First things first, what is a stack view? The stack view provides a streamlined interface for laying out a collection of views in either a column or a row. In Keynote or Microsoft Powerpoint, you can group multiple objects together so that they can be moved or resized as a single object. Stack views offer a very similar feature. You can embed multiple UI objects into one by using stack views. In most cases, for views embedded in a stack view, you no longer need to define auto layout constraints.

```
Quick note: For views embedded in a stack view, they are
usually known as arranged views.
```

The stack view manages the layout of its subviews and automatically applies layout constraints for you. That means the subviews are ready to adapt to different screen sizes. Furthermore, you can embed a stack view in another stack view to build more complex user interfaces. Sounds cool, right?

Don’t get me wrong. It doesn’t mean you do not need to deal with auto layout. You still need to define the layout constraints for the stack views. It just saves you time from creating constraints for every UI element and makes it super easy to add/remove views from the layout.

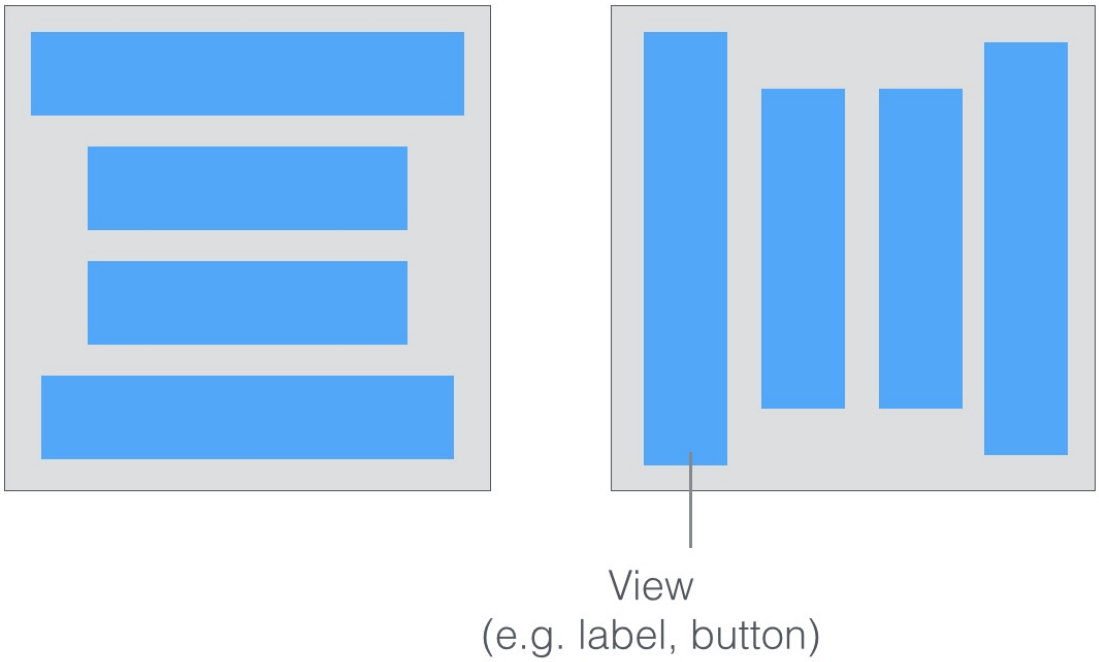


Figure 6-1. Horizontal Stack View (left) / Vertical Stack View (right)

Xcode provides two ways to use stack view:

1. You can drag a Stack View (horizontal / vertical) from the Object library, and put it right into the storyboard. You then drag and drop view objects such as labels, buttons, image views into the stack view.
2. Alternatively, you can use the Stack option in the auto layout bar. For this approach, you select two or more view objects and then choose the Stack option. Interface Builder then embeds the objects into a stack view and resizes it automatically.

If you still have no ideas about stack views, no worries. We'll go through both approaches in this chapter. Just read on and you'll understand what I mean in a minute.

The Sample App

Let's first take a look at the demo app we're going to build. I will show you how to lay out a welcome screen like this using stack views:

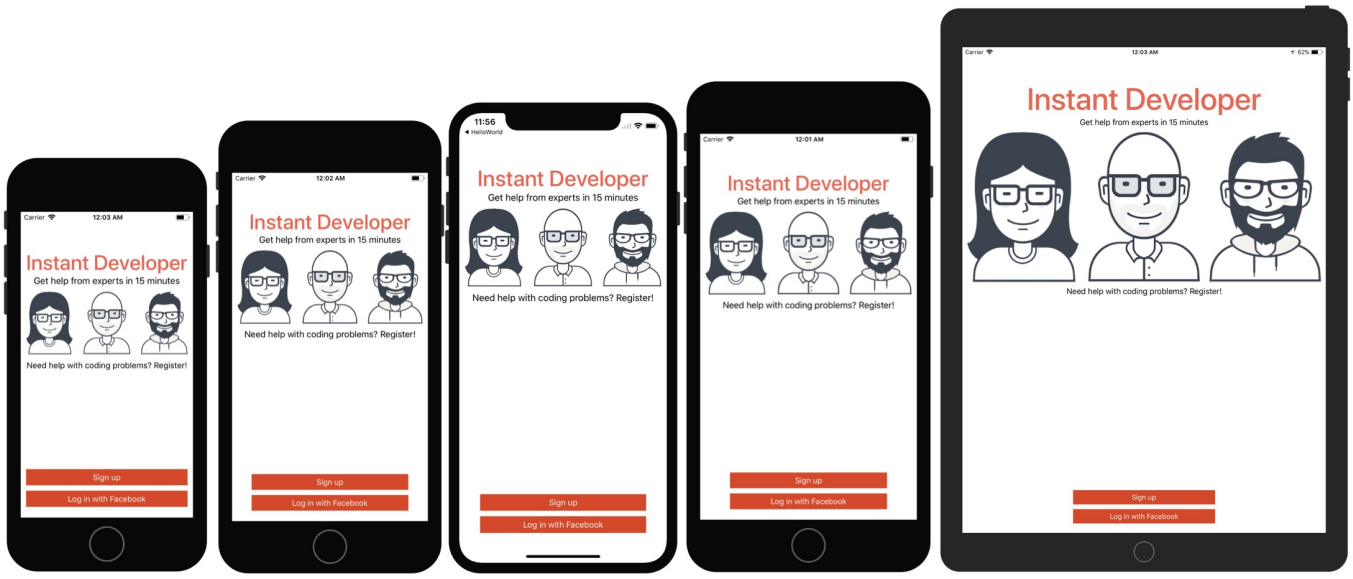


Figure 6-2. The sample app

You can create the same UI without using stack views. But you will soon see how stack views completely change the way how you layout user interfaces. Again, there is no coding in this chapter. We will just focus on using Interface Builder to build an adaptive user interface. This is a crucial skill you will need in app development.

Creating a New Project

Now fire up Xcode and create a new Xcode project. Choose Application (under iOS) > "Single View Application" and click "Next". You can simply fill in the project options as follows:

- **Product Name: StackViewDemo** – This is the name of your app.
- **Team:** Just leave it as it is.
- **Organization Name: AppCoda** – It's the name of your organization.
- **Organization Identifier: com.appcoda** – It's actually the domain name written the other way round. If you have a domain, you can use your own domain * name. Otherwise, you may use "com.appcoda" or just fill in "edu.self".
- **Bundle Identifier: com.appcoda.StackViewDemo** - It's a unique identifier of your app, which is used during app submission. You do not need to fill in this option.

Xcode automatically generates it for you.

- **Language: Swift** – We'll use Swift to develop the project.
- **Use Core Data: [unchecked]** – Do not select this option. You do not need Core Data for this simple project.
- **Include Unit Tests: [unchecked]** – Do not select this option. You do not need unit tests for this simple project.
- **Include UI Tests: [unchecked]** – Do not select this option. You do not need UI tests for this simple project.

Click "Next" to continue. Xcode then asks you where to save the StackViewDemo project. Pick a folder on your Mac. Click "Create" to continue.

Adding Images to the Xcode Project

As you may notice, the sample app has three images. The question is how can you bundle images in Xcode projects?

In each Xcode project, it includes an asset catalog (i.e. Assets.xcassets) for managing images and icons that are used by your app. Go to the project navigator and select the `Assets.xcassets` folder. By default, it is empty with a blank `Appicon` set. We are not going to talk about app icons in this chapter, but will revisit it after building a real-world app.

Figure 6-3. Asset Catalog

Now download this image set (https://www.appcoda.com/resources/swift4/stackviewdemo-images.zip) and unzip it on your Mac. The zipped archive contains a total of 5 image files:

- user1.pdf
- user2.png
- user2@2x.png
- user2@3x.png
- user3.pdf

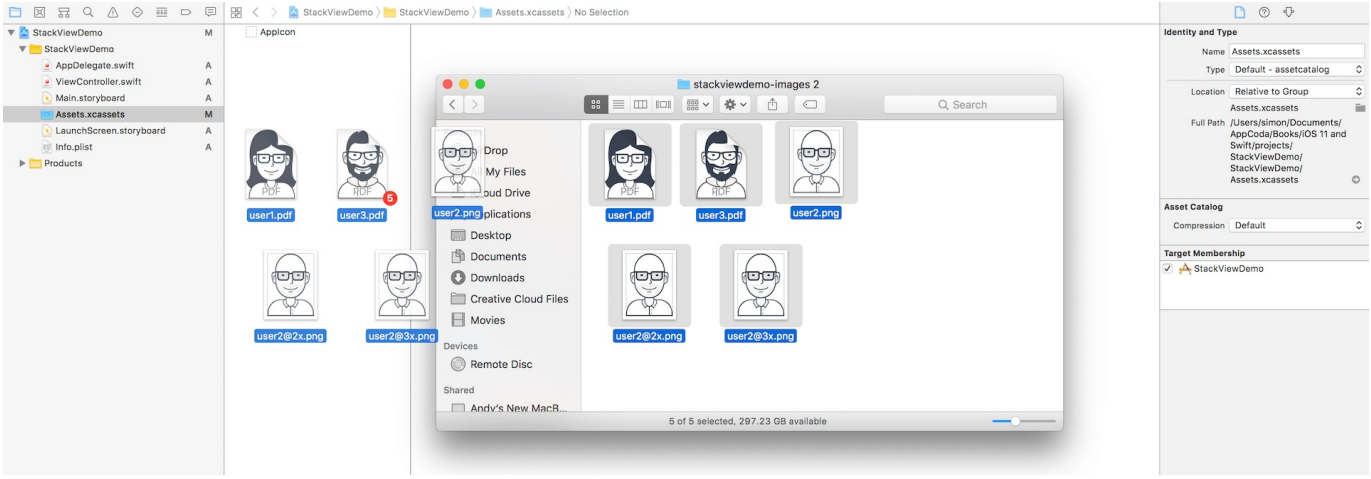
```
Credit: The images are provided by usersinsights.com.
```

iOS supports two categories of images: *raster images* and *vector images*. Common image formats like PNG and JPEG are classified as raster images. Raster images use a grid of pixels to form a complete image. One problem of raster images is that it doesn't scale up well. Increasing the size of a raster image usually means a significant loss of quality. This is why Apple recommends developers to provide three different resolutions of images when PNG is used. In this example, the image files, prefixed by , comes with three versions. The one with @3x suffix, which has the highest resolution, is for iPhone 6/7/8 Plus and iPhone X. The one with @2x suffix is for iPhone SE/6/7/8, while the one without the @ suffix is for older devices with non-Retina display (e.g. iPad 2).

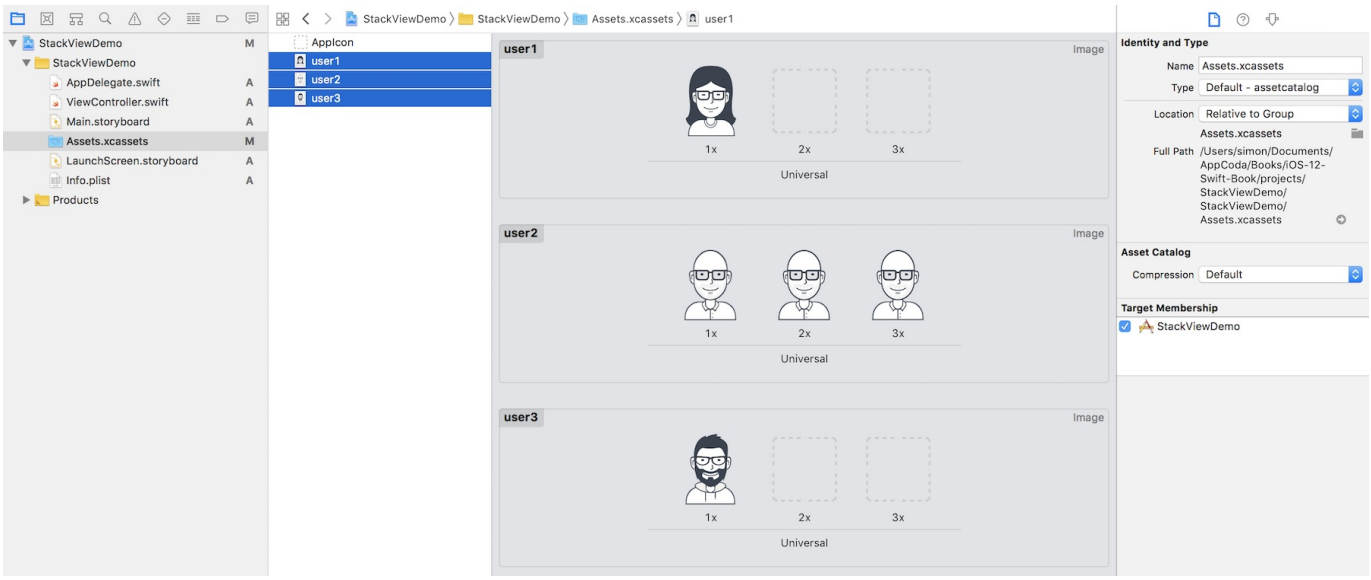
Vector images usually have file types such as PDF and SVG. You can use tools like Sketch and Pixelmator to create vector images. Unlike raster images, vector images are comprised of paths instead of pixels. This allows the images to scale up without losing any image quality. Because of this feature, you just need to provide a single version of the image in PDF format for Xcode.

I intentionally include both image types in the example for illustration purpose. When developing a real world app, you usually work with either one or the other. So which image type is more preferable? Whenever possible, ask your designer to prepare the images in PDF format. The overall file size is smaller and the images are scalable without losing quality.

To add the images to the asset catalog, all you need to do is drag the images from Finder, and drop them into the set list or set viewer.



Once you add the images to the asset catalog, the set view automatically organizes the images into different wells. Later, to use the image, you just need to use the set name of a particular image (e.g. user1). You can omit the file extension. Even if you have multiple versions of the same image (e.g. user2), you don't have to worry about which version (@2x/@3x) of the image to use. All these are handled by iOS accordingly.

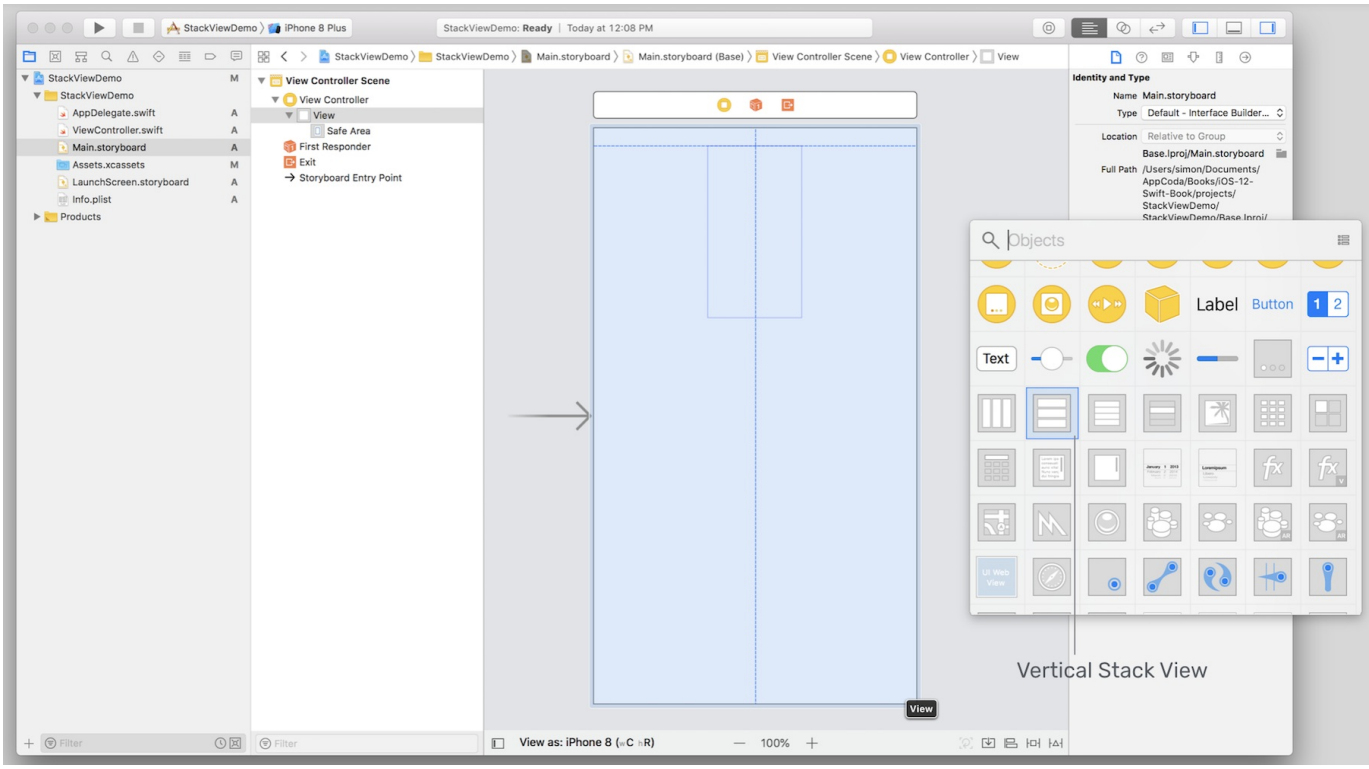


Layout the Title Labels with Stack Views

Now that you've bundled the necessary images in the project, let's move onto the creation of stack views. First, open `Main.storyboard`. We'll start with the layout of these two labels.

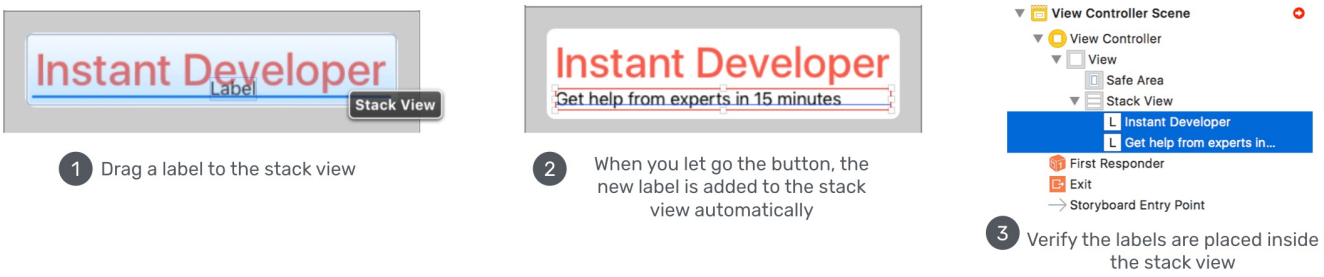
Stack view can arrange multiple views (known as arranged views) in both vertical and horizontal layouts. So first, you have to decide whether you want to use a vertical or horizontal stack view. The title and subtitle labels are arranged vertically. Therefore, vertical stack view is a suitable choice.

Click the Object library button to display the Object library. Drag a Vertical Stack View object to the view controller in the storyboard.



Next, drag a label from the Object library and put it right into the stack view. Once you drop the label into the stack view, the stack view automatically embeds it and resizes itself to fit the label. Double click the label and change the title to "Instant Developer". In the Attributes inspector, increase the font size to *40* points, and font style to *medium*. Optionally, you can change the color to *red*.

Now, drag another label from the Object library to the stack view. As soon as you release the label, the stack view embeds the new label and arranges the two labels vertically like this:



Edit the title of the new label and change it to "Get help from experts in 15 minutes". If you've done it correctly, the two labels should be placed inside the stack view. You can verify it by reviewing the objects in the document outline.

Currently, both labels are aligned to the left of the stack view. To change its alignment and appearance, you can modify the built-in properties of the stack view. Select the stack view and you can find its properties in the Attributes inspector.

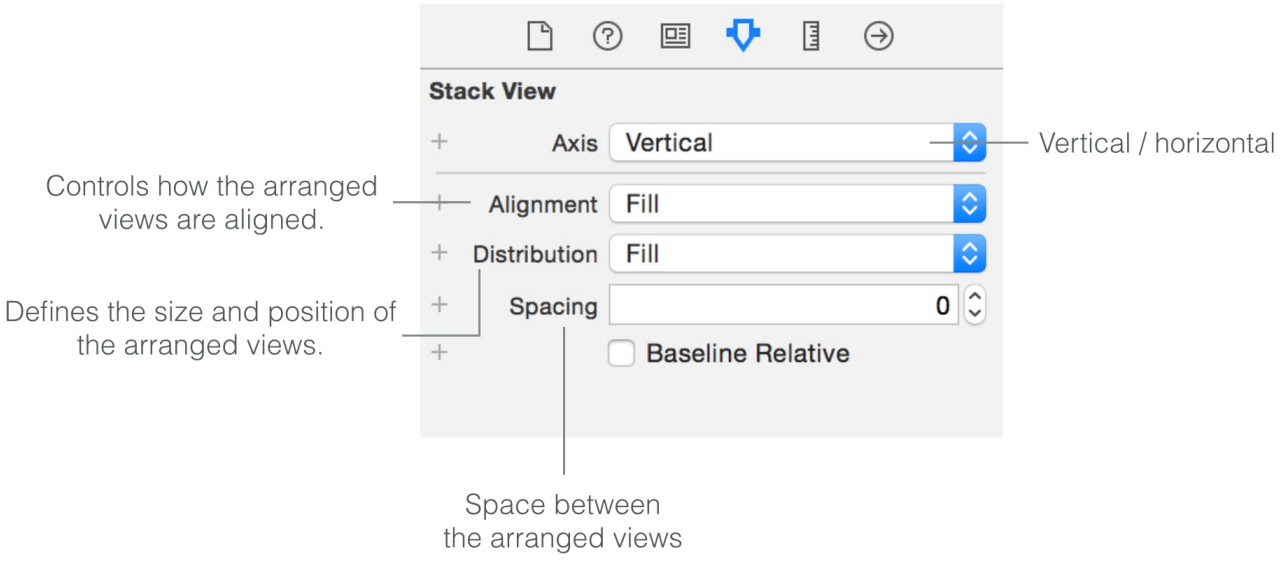


Figure 6-9. Sample properties of a stack view

As a side note, if you have any problems selecting the stack view, you can hold the shift key and right-click the stack view. Interface Builder will then show you a shortcut menu for selection. Alternatively, you can select the stack view in the document outline. Both approaches would work.

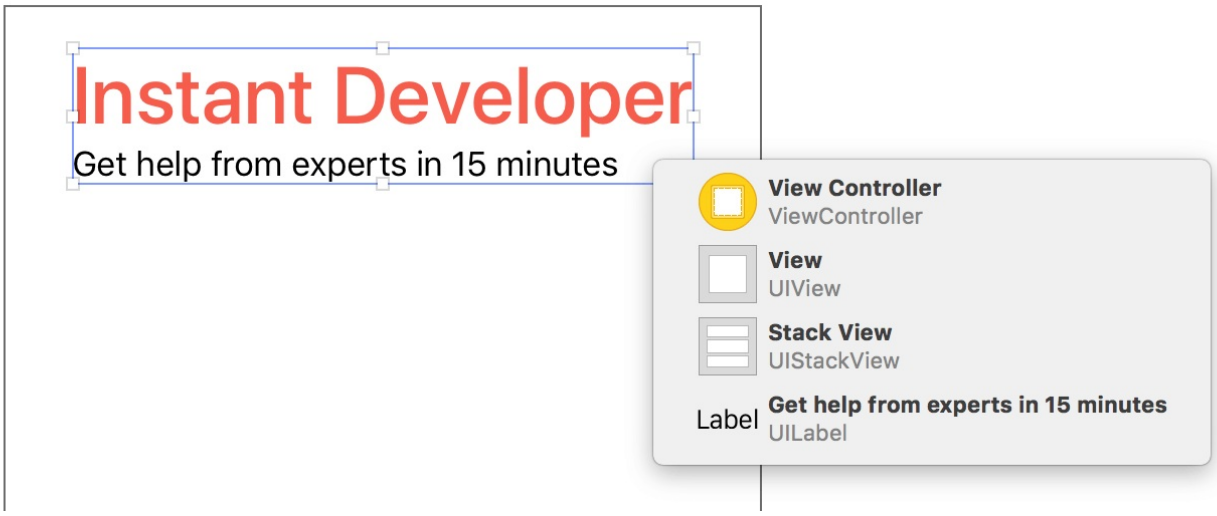


Figure 6-10. Shortcut menu for selection

Let's briefly talk about each property of the stack view:

- The *axis* option indicates whether the arranged views should be layout vertically or horizontally. By changing it from vertical to horizontal, you can turn the existing stack view to a horizontal stack view.
- The *alignment* option controls how the arranged views are aligned. For example, if it is set to Leading, the stack view aligns the leading edge (i.e. left) of its arranged views along its leading edge.
- The *distribution* option defines the size and position of the arranged views. By default, it is set to *Fill*. In this case, the stack view tries its best to fit all subview in its available space. If it is set to *Fill Equally*, the vertical stack view distributes both labels equally so that they are all the same size along the vertical axis.

Figure 6-11 shows some sample layouts of different properties.

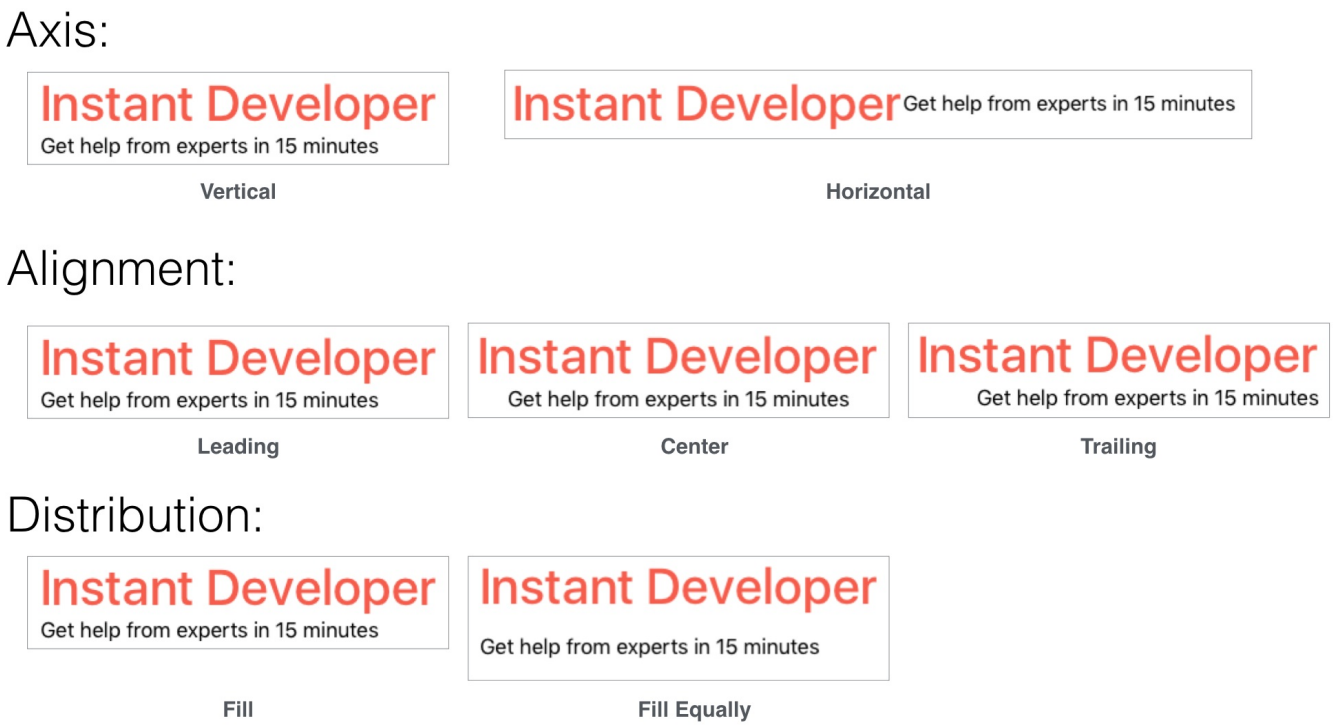


Figure 6-11. A quick demo of the stack view's properties

For our demo app, we just need to change the Alignment option from *Fill* to *Center* and keep other options intact. This should center both labels.

Before moving onto the next section, make sure you position the stack view correctly. This would help you easier to follow the rest of the materials. You can select the stack view and go to the Size inspector. Ensure you set the value of X to `28` and Y to `70`.

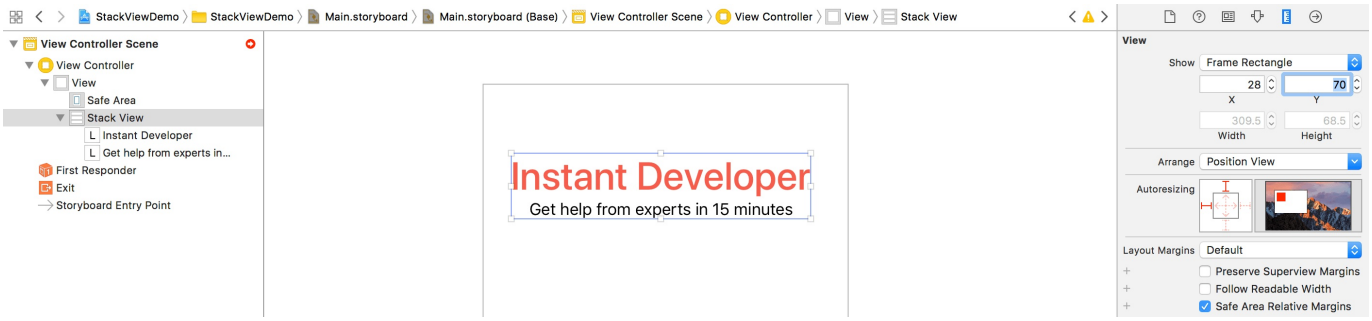


Figure 6-12. Verify the position of the stack view

Layout the Images Using the Stack Button

At the very beginning of this tutorial, I mentioned that there are two ways to use stack views. Earlier, you added a stack view from the object library. Now I'd like to show you another approach.

We're going to lay out the three user images. In iOS, we use image views to display images. From the Object library, look for the image view object, and drag it into the view. Once you added the image view, select the Attributes inspector. The image option already loads the available images from the asset catalog. Simply set it to `user1`. In the Size inspector, set the width of the image to `100` and height to `134`.

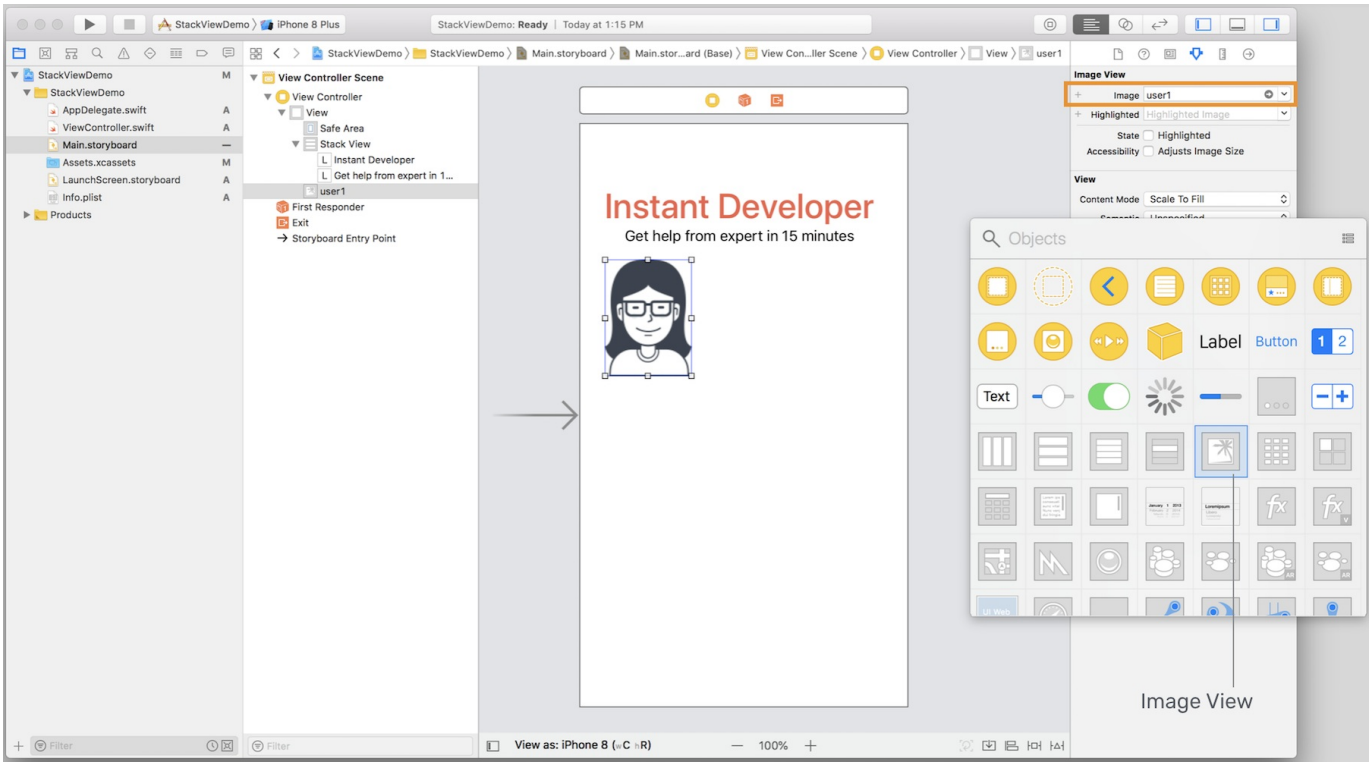


Figure 6-13. Adding an image view for displaying images

Repeat the procedures to add two more image views, and place them next to each other. Set the image of the second and third image view to `user2` and `user3` respectively. Your layout should look like this:

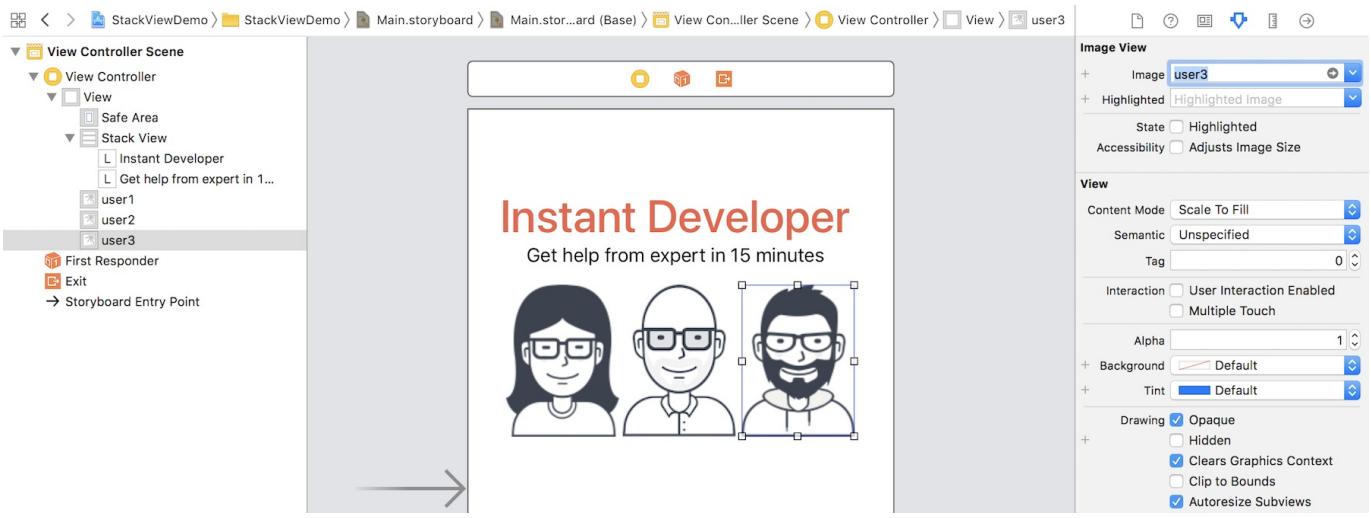


Figure 6-14. Assign the images to the image views

Using stack views minimizes the number of layout constraints you have to define, but it doesn't mean you do not need to define any auto layout constraints. For the image views, we want each of them to keep its aspect ratio. Later, the size of these images varies depending on the screen size. We want them to retain their aspect ratio, no matter they are stretched or squeezed.

To do that, in the document outline view, control-drag horizontally on the image view of `user1`. In the shortcut menu, select `Aspect Ratio`. Repeat the same procedure for the other two image views.

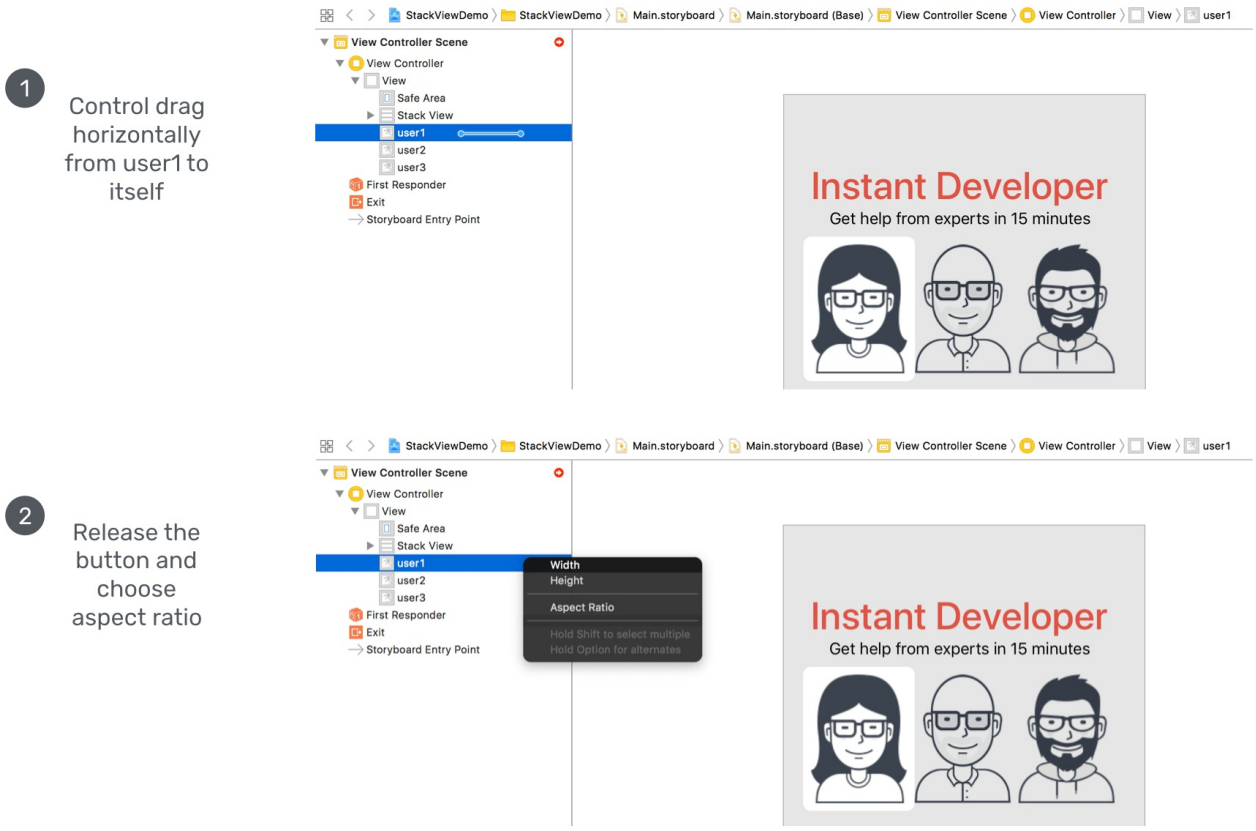


Figure 6-15. Adding an aspect ratio constraint

Now I want to group these three image views together using a stack view, so it is easier to manage. However, we will use an alternative approach to create the stack view.

Hold the command key and click the three image view to select them. Then click the *Embed in* button in the layout bar and choose *Stack View*. Interface Builder automatically embeds the image views in a horizontal stack view.

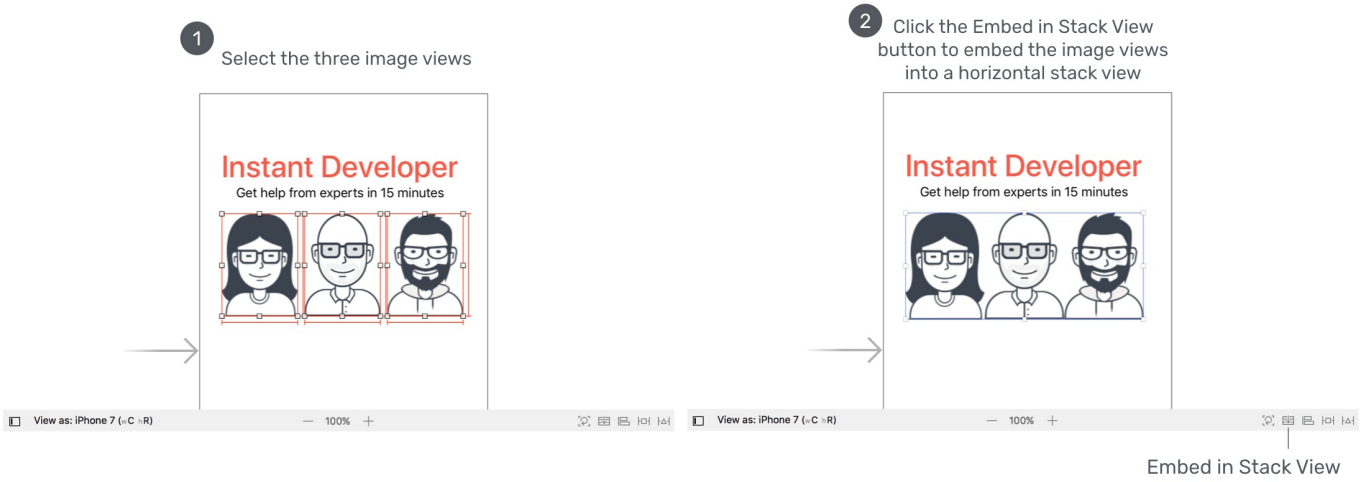


Figure 6-16. Group the image views in a horizontal stack view

To add some spacing between the image views, select the stack view and set the spacing to `20`. Also, change the Distribution option from *Fill* to *Fill Equally*.

> **Quick note:** It seems that the stack view has already distributed the image views equally. Why do we need to change the distribution to Fill Equally? Remember that you're now designing the UI for all screen sizes. If you do not explicitly set the distribution to Fill Equally, iOS will layout the image views using Fill distribution policy. It may not look good on other screen sizes.

Now you have two stack views: one for the labels, and the other for the image views. These two stack views can actually be combined together for easier management. One great thing about stack views is that you can nest multiple stack views together.

To do that, select both stack views in the document outline view. Then click the *Embed in > Stack View* to embed both stack views in a vertical stack view. After that, make sure that the alignment of the new stack view is set to *Fill*. This size of the stack view increases

as the screen size grows. By setting the alignment property to *Fill*, this ensures the image views are resized automatically on larger screens. You will understand what I mean when we preview the UI on iPad.

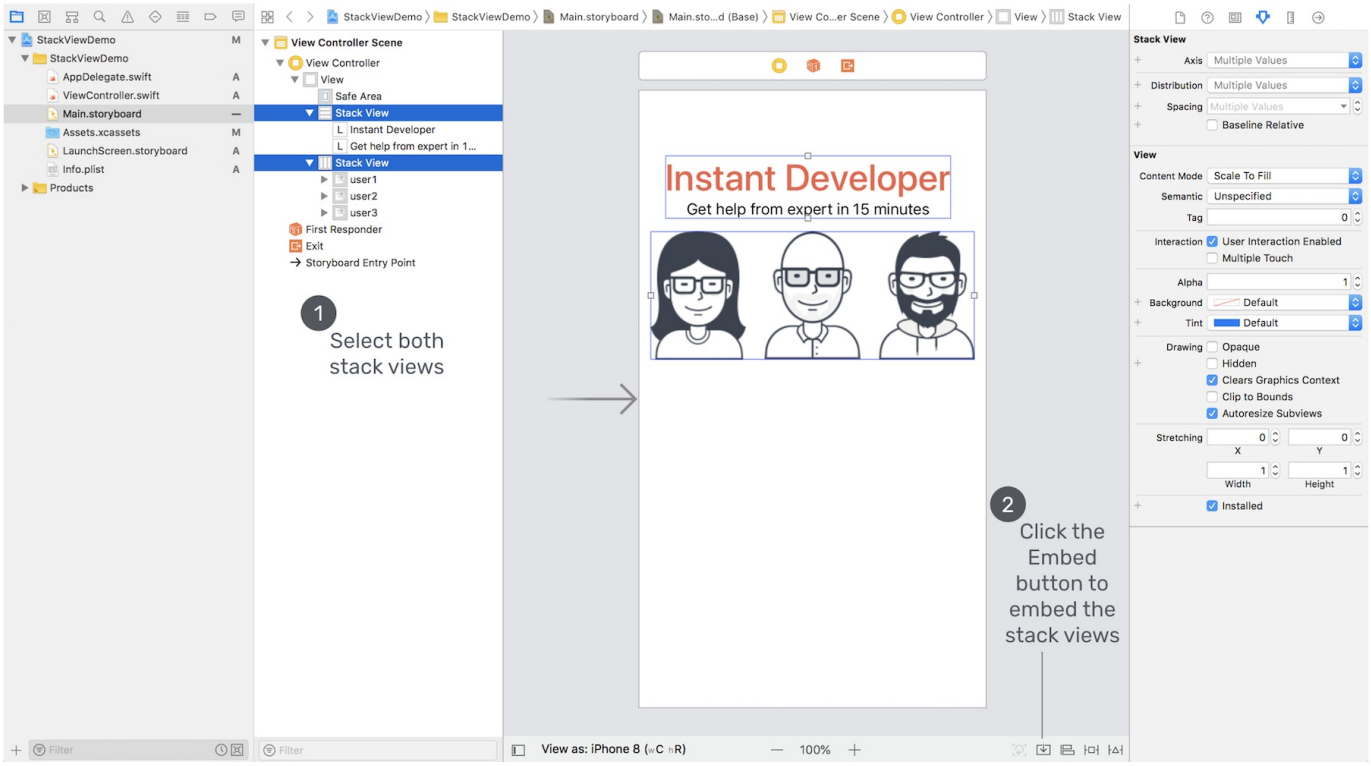


Figure 6-17. Nested stack views

Cool, right? So far the UI looks good on iPhone 8. But if you preview the UI on other devices, it doesn't look as expected. The reason is that we haven't defined the layout constraints for the stack views.

Adding Layout Constraints for the Stack View

For the stack view, we will define the following layout constraints:

- Set a spacing constraint between itself and the top layout guide, such that it is 50 points away from the top anchor of the safe area layout guide.
- Set a spacing constraint between the left side of the stack view and the leading anchor of the safe area, such that there is a space of 10 points between them.

- Set a spacing constraint between the right side of the stack view and the trailing anchor of the safe area, such that there is a space of 10 points between them.

To continue reading and access the full version of the book, please get the full copy here. You will also be able to access the full source code of the project.

Chapter 7
Introduction to Prototyping

> If a picture is worth 1000 words, a prototype is worth 1000 meetings.
>
> - @IDEO

How many times have you heard of someone say, "I got an app idea!"

Probably you already got one. So what's the next step?

Now that you have some basic concepts of iOS programming and Interface Builder, should you just open Xcode and start coding your app?

As I always said, coding is just a part of the app development process. Before you begin to code your app, you will have to go through other preparation stages. This is not a book about software engineering. I am not going to discuss every stage of the software development lifecycle. Instead, I want to focus on prototyping, which is an integral part of the mobile development process.

Every time I mentioned prototype to beginners, two questions pop up:

- What's a prototype?
- Why prototyping?

A prototype is an early model of a product used for testing a concept or visualizing an idea. Prototyping has been used in many industries. Before constructing a building, an architect needs to draw a plan of the building and make a model of the building. An aircraft company builds a prototype of an aircraft to test any design flaws before building and assembling an airplane. Software companies also build software prototypes to explore an idea before creating the actual application. In the context of app development, a prototype can be an early sample of an app which is not fully functional and contains a basic UI or even sketches.

Prototyping is the process of developing a prototype and offers many advantages. First, it helps you visualize your idea and better communicate your idea to your team members and users. While you are now learning and developing an app on your own, app development rarely happens like that in a real-world environment.

You probably work in a team of programmers and UI/UX designers to build apps for your clients. Even if you're an indie (or solo) developer, you're probably developing an app that targets a specific group of users or a niche market. Or you hire a designer to design the app UI for you. You have to find some ways to communicate the app idea with your designer or test your idea with your potential users. You can explain your idea in words, tell your users how the app works but this is not effective enough. There is no better way than showing your app idea as a fully functional demo.

By creating a prototype, you can involve everyone (developers, designers, and users) of the project earlier. All the involved parties will better understand how the app works and figure out what's missing at the early development stage, way ahead of the final product

is built.

Prototyping also allows you to test an idea without building an actual app. You can show your prototype to your potential users and get early feedback before an app is built. This saves you both time and money. Figure 7-1 illustrates the benefits of prototyping.

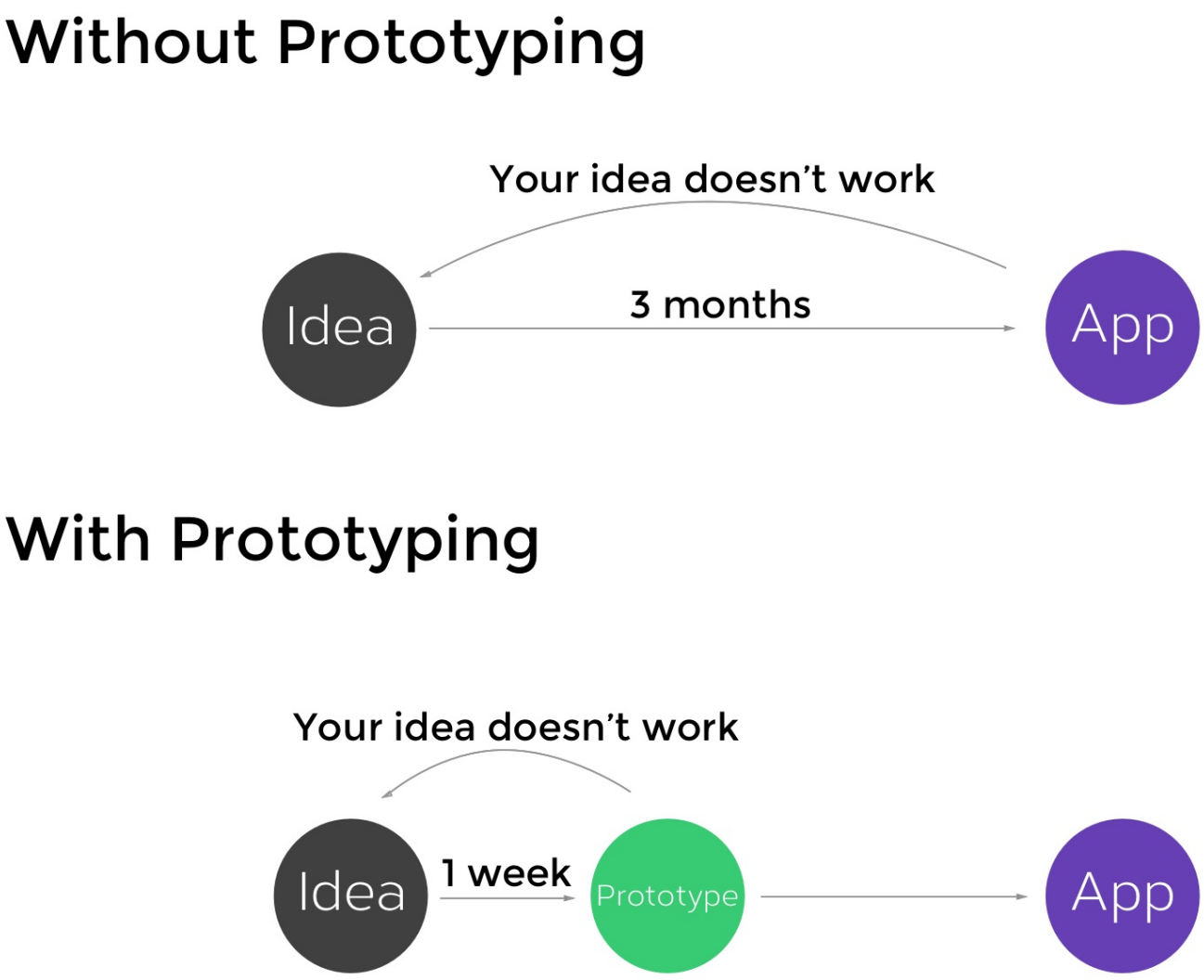


Figure 7-1. Prototyping saves you money and time

To continue reading and access the full version of the book, please get the full copy here. You will also be able to access the full source code of the project.

Chapter 8
Creating a Simple Table Based App

> That's been one of my mantras - Focus and Simplicity. Simple can be harder than complex: You have to work hard to get your thinking clean to make it simple. But it's worth it in the end because once you get there, you can move mountains.
>
> - Steve Jobs

Now that you have a basic understanding of prototyping and our demo app, we'll work on something more interesting and build a simple table-based app using UITableView in this chapter. Once you master the technique and the table view customization (to be discussed in the next chapter), we'll start to build the Food Pin app.

First of all, what exactly is a table view in an iPhone app? A table view is one of the most common UI elements in iOS apps. Most apps (except games), in some ways, make use of table view to display content. The best example is the built-in Phone app. Your contacts

are displayed in a table view. Another example is the Mail app. It uses a table view to display your mail boxes and emails. Not only designed for listing textual data, table view allows you to present the data in the form of images. The TED, Google+ and Airbnb are also good examples. Figure 8-1 displays a few more sample table-based apps. Though they look different, all in some ways utilize a table view.

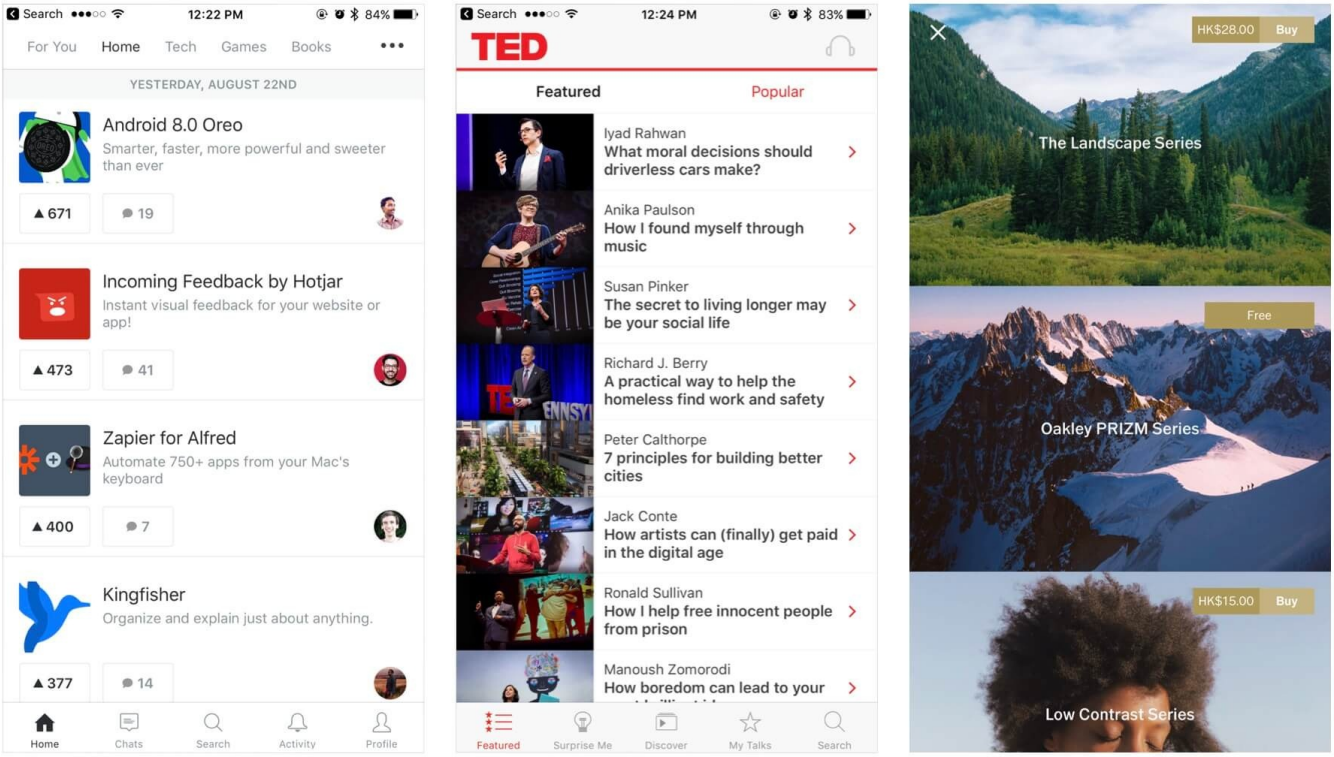
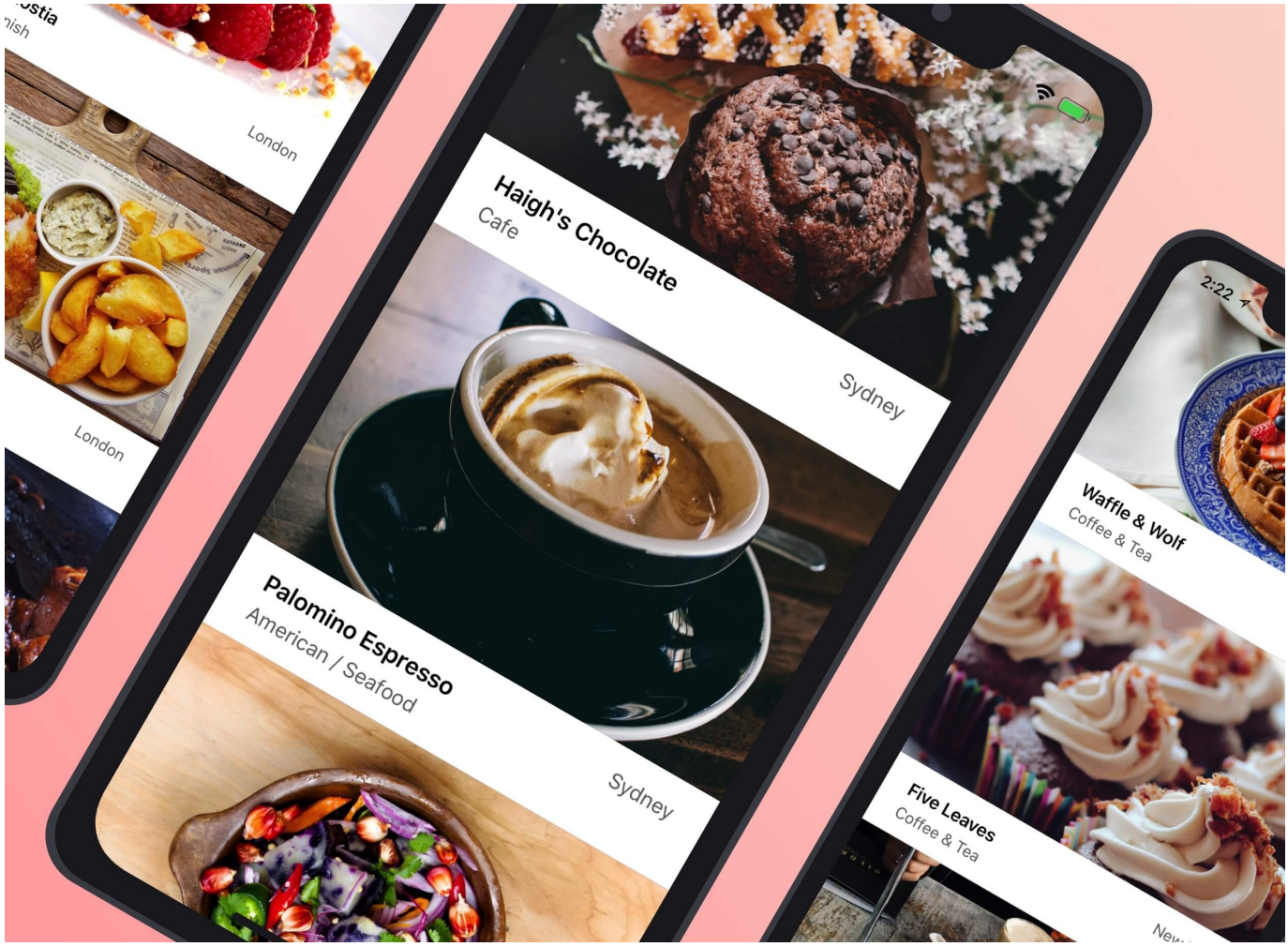


Figure 8-1. Sample table-based apps (left: Product Hunt, middle: TED, right: VSCO)

What we are going to do in this chapter is create a very simple table view and learn how to populate data (images and text) into it.

To continue reading and access the full version of the book, please get the full copy here. You will also be able to access the full source code of the project.

Chapter 9
Customize Table Views Using Prototype Cell



> I think that's the single best piece of advice: Constantly think about how you could be doing things better and questioning yourself.
>
> - Elon Musk, Tesla Motors

In the previous chapter, we created a simple table-based app to display a list of restaurants using the basic cell style. In this chapter, we'll customize the table cell and make it look more stylish. There are a number of changes and enhancements we are

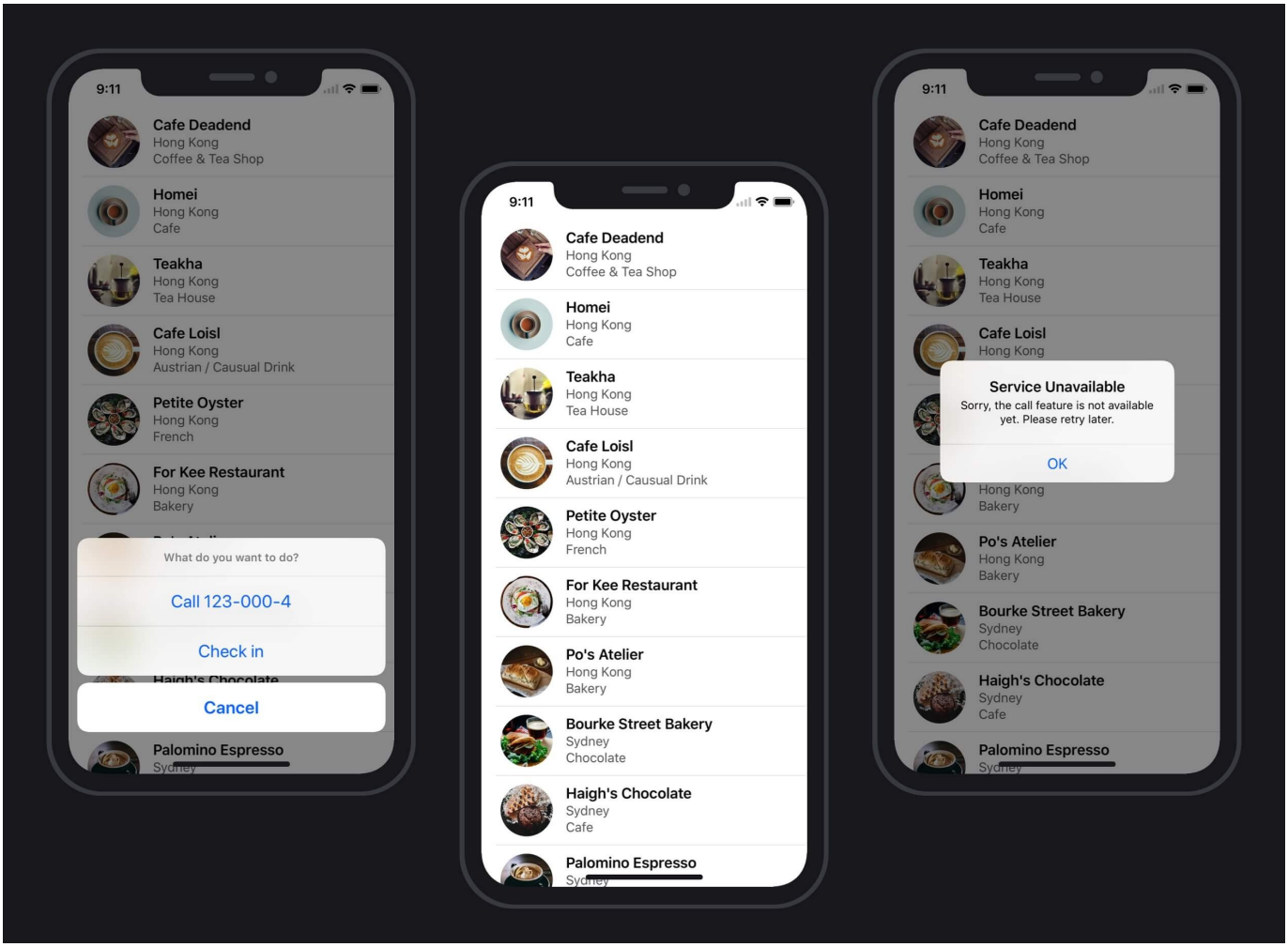
going to work on:

- Rebuild the same app using `UITableViewController` instead of `UITableView`
- Display a distinct image for each restaurant rather than showing the same thumbnail
- Design a custom table view cell instead of using the basic style of table view cell

You may wonder why we need to rebuild the same app. There are always more than one way to do things. Previously, we used `UITableView` to create the table view. In this chapter, we'll use `UITableViewController` to create a table view app in Xcode. Will it be easier? Yes, it's going to be easier. Recalled that we needed to explicitly adopt both `UITableViewDataSource` and `UITableViewDelegate` protocols, `UITableViewController` has already adopted these protocols and established the connections for us. On top of this, it has all the required layout constraints right out of the box.

Starting from this chapter and onwards, you begin to develop a real-world app called *FoodPin*. It's gonna be fun!

Chapter 10
Interacting with Table Views and Using UIAlertController



> There is no learning without trying lots of ideas and failing lots of times.
>
> - Jonathan Ive

Up till now, we only focus on displaying data in a table view. I guess you are thinking how we can interact with the table view and detect row selections. This is what we're going to discuss in this chapter.

We'll continue to polish the FoodPin app, which we have built in the previous chapter, and add a couple of enhancements:

- Bring up a menu when a user taps a cell. The menu offers two options: *Call* and *Check-in*.
- Display a heart icon when a user selects *Check-in*.

Through implementing these new features, you will also learn how to use UIAlertController, which is commonly used to display alerts in iOS apps.

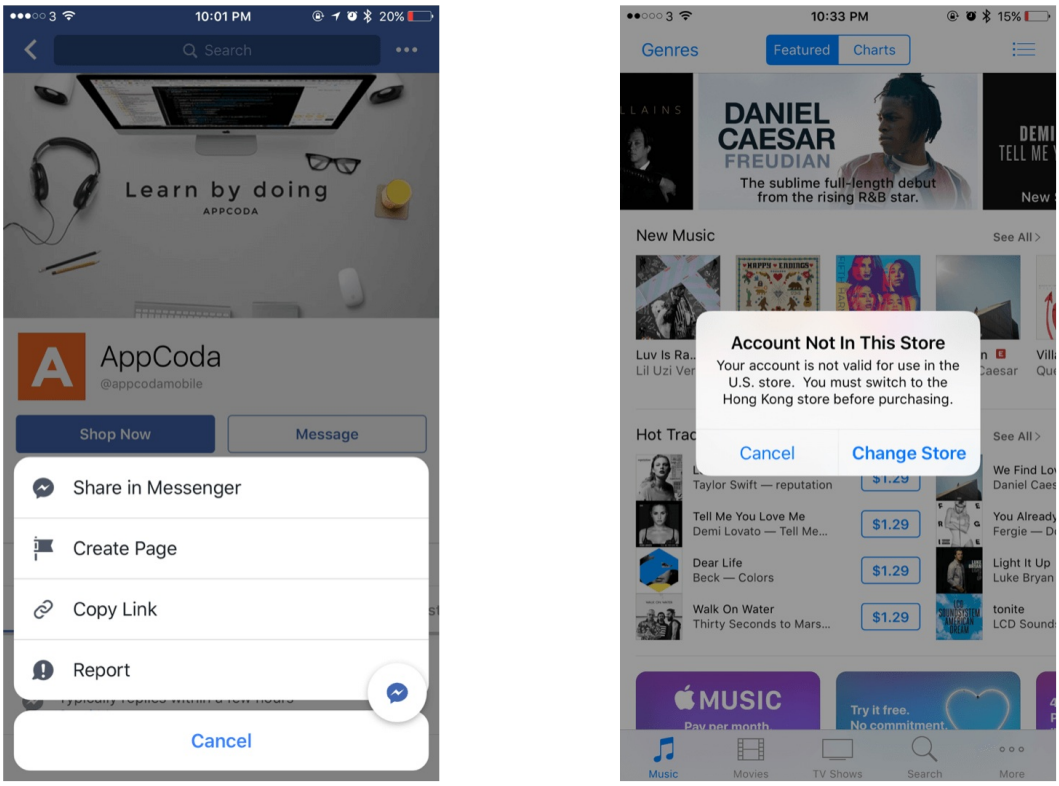


Figure 10-1. Sample alerts in Facebook and iTunes apps

> **Quick note:** This class replaces the UIActionSheet and UIAlertView classes for displaying alerts in iOS 8 (or up).

Understanding the UITableViewDelegate Protocol

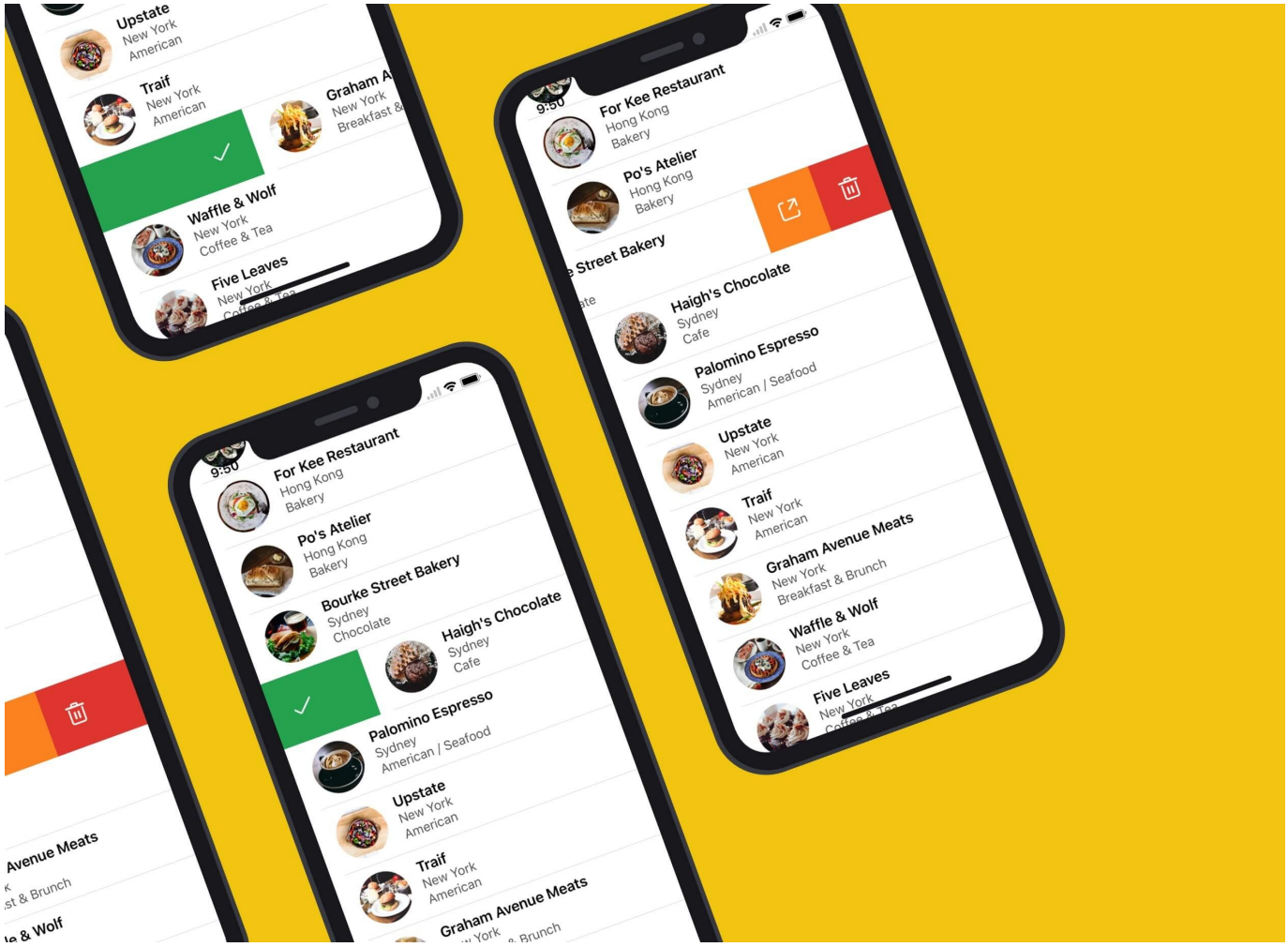
When we first built the SimpleTable app in Chapter 8, we adopted two delegates, `UITableViewDelegate` and `UITableViewDataSource`, in the `RestaurantTableViewController` class. I have discussed with you the `UITableViewDataSource` protocol but barely mentioned about the `UITableViewDelegate` protocol.

As said before, the delegate pattern is very common in iOS programming. Each delegate is responsible for a specific role or task to keep the system simple and clean. Whenever an object needs to perform a certain task, it depends on another object to handle it. This is usually known as "separation of concerns" in software design.

The `UITableView` class applies this design concept. The two protocols are designed for different purposes. The `UITableViewDataSource` protocol defines methods, which are used for managing table data. It relies on the delegate to provide the table data. On the other hand, the `UITableViewDelegate` protocol is responsible for setting the section headings and footers of the table view, as well as, handling cell selections and cell reordering.

To manage the row selection, we will implement some of the methods in the `UITableViewDelegate` protocol.

Chapter 11
Table Row Deletion, Swipe for Actions, Activity Controller and MVC



> If you spend too much time thinking about a thing, you'll never get it done. Make at least one definite move daily toward your goal.
>
> – Bruce Lee

Now you know how to handle table row selection. But how about deletion? How can we delete a row from a table view?

It's a common question when building a table-based app. Select, delete, insert and update are the basic operations when dealing with data. We've discussed how to select a table row. Let's talk about deletion in this chapter. In addition, we'll go through a couple of new features to the FoodPin app:

1. Adding a custom action button when a user swipes horizontally in a table row. This is usually known as *Swipe for More* action.
2. Adding a social sharing feature to the app, that enables users to share the restaurants on Twitter or Facebook.

There is a lot to learn in this chapter, but it's going to be fun and rewarding. Let's get started.

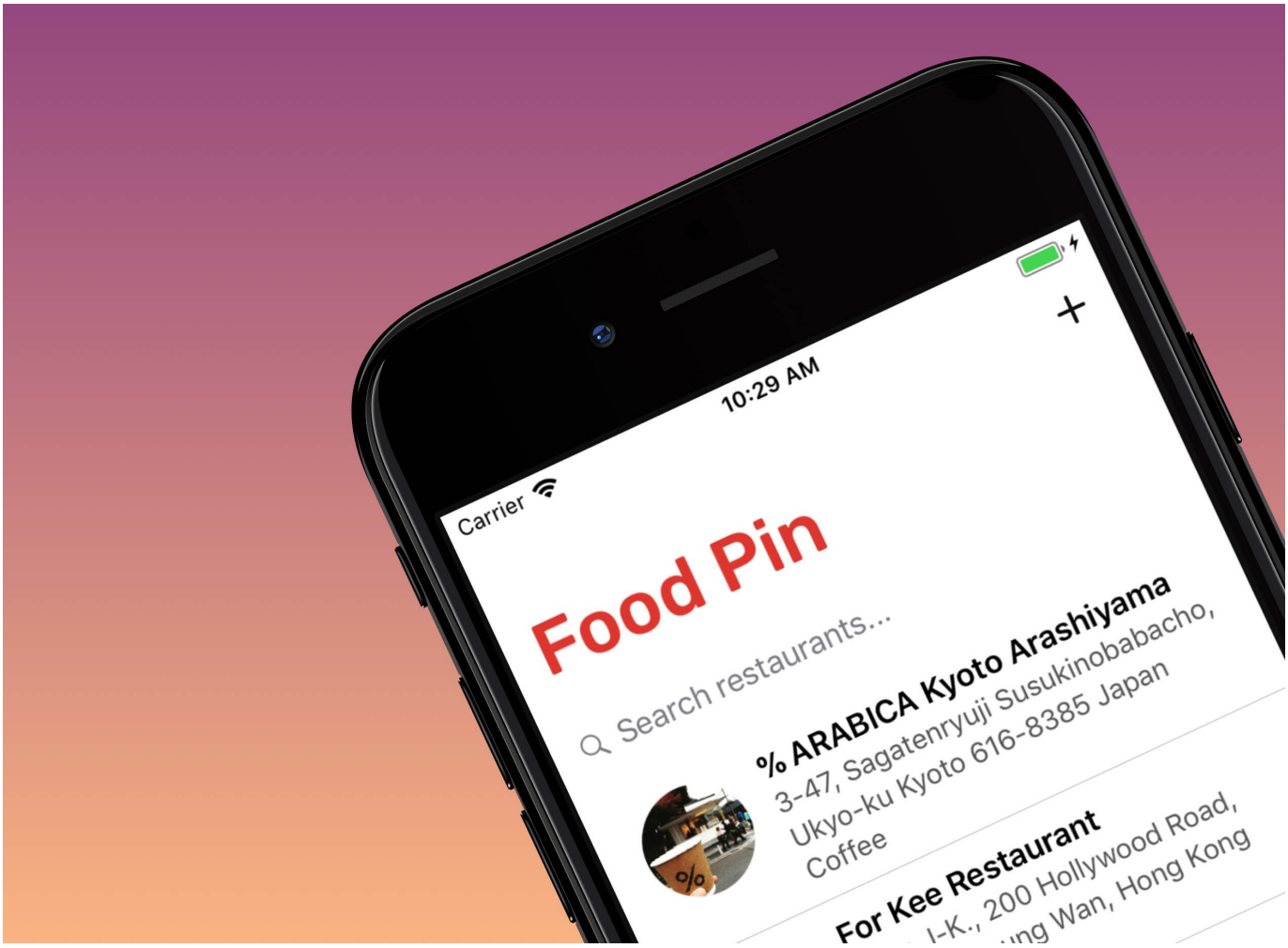
A Brief Introduction to Model View Controller

Before jumping into the coding part, I would like to give you an introduction of Model-View-Controller (MVC) model, which is one of the most quoted design patterns for user interface programming.

I try to keep this book as practical as possible and seldom talk about the programming theories. That said, you can't avoid learning Model-View-Controller, especially your goal is to build great apps or become a competent programmer. MVC is not a concept that applies to iOS programming only. You may have heard of it if you've studied other programming languages, such as Java or Ruby. It is a powerful design pattern used in designing software applications, whether it is a mobile app and a web app.

To continue reading and access the full version of the book, please get the full copy here. You will also be able to access the full source code of the project.

Chapter 12
Introduction to Navigation Controller and Segue



> Just build something that you'd want to use today, not something you think people would use somehow.
>
> – Paul Graham

First things first, what's navigation controller? Like table views, navigation controllers are another common UI components in iOS apps. It provides a drill-down interface for hierarchical content. Take a look at the built-in Photos app, YouTube, and Contacts. They

all use navigation controllers to display hierarchical content. Generally, you combine a navigation controller with a stack of table view controllers to build an elegant interface for your apps. Being that said, this doesn't mean you have to use both together. Navigation controllers can work with any types of a view controller.

Scenes and Segues in Storyboards

Up till now, we just lay out a table view controller in the storyboard of the FoodPin app. Storyboarding allows you to do more than that. You can add more view controllers in the storyboard, link them up, and define the transitions between them. All these can be done without a line of code. When working with storyboards, *scene* and *segues* are two of the terms you have to understand. In a storyboard, a scene usually refers to the on-screen content (e.g. a view controller). Segues sit between two scenes and represent the transition from one scene to another. *Push* and *Modal* are two common types of transition.

> **Note:** You can make storyboards more manageable and modular by using a feature called storyboard references. When your project becomes large and complex, you can break a large storyboard into multiple storyboards and link them up using storyboard references. This feature is particularly useful when you are collaborating with your team members to create a storyboard.

Creating Navigation Controller

We'll continue to work on the Food Pin app by embedding the table view controller into a navigation controller. When a user selects any of the restaurants, the app navigates to the next screen to display the restaurant details.

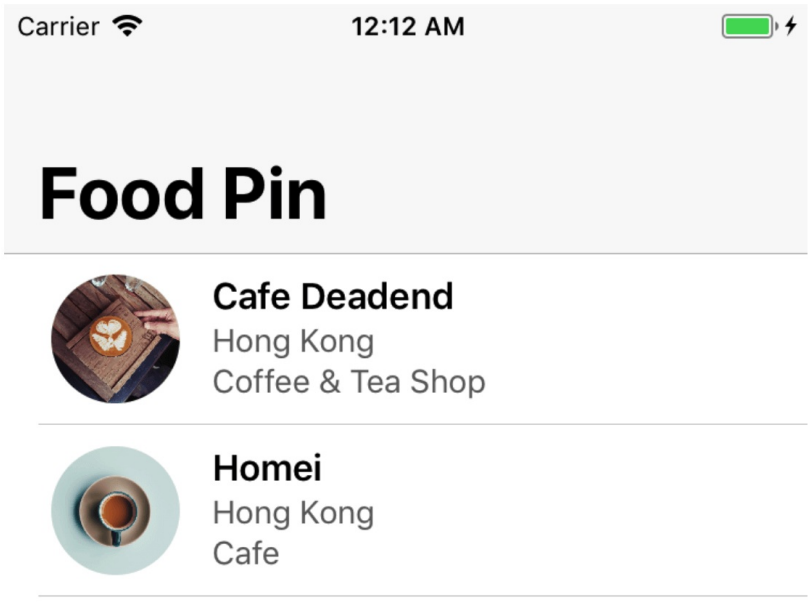


Figure 12-1. FoodPin app with a navigation bar

To continue reading and access the full version of the book, please get the full copy here. You will also be able to access the full source code of the project.

Chapter 13
Object Oriented Programming, Project Organization and Code Documentation

> Most good programmers do programming not because they expect to get paid or adulation by the public, but because it is fun to program.
>
> - Linus Torvalds

If you read from the very beginning of the book and have worked through all the projects, you've gone pretty far. By now, you should be able to build an iPhone app with navigation controllers and table views using Interface Builder. We'll further enhance the FoodPin

app, and improve the detail view. Did you manage to complete the previous exercise and develop your own detail view? This shouldn't be difficult to implement if you understand the materials and I intentionally left out that part for you as an exercise.

Anyway, we'll revisit it and show you how to improve the detail screen. But before that, I have to introduce to you the basics of *Object Oriented Programming*. In the next chapter, we'll build on top of what we'll learn in this chapter and enhance the detail view screen.

Don't be scared by the term "Object Oriented Programming" or OOP in short. It's not a new kind of programming language, but a programming concept. While some programming books start out by introducing the OOP concept, I purposely left it out when I began writing the book. I want to keep things simple and show you how to create an app. I don't want to scare you away from building apps, just because of a technical term or concept. Having said that, I think it's time to discuss OOP. As you're still reading the book, I believe you're determined to learn iOS programming, and you want to take programming skills to the next level.

Okay, let's get started.

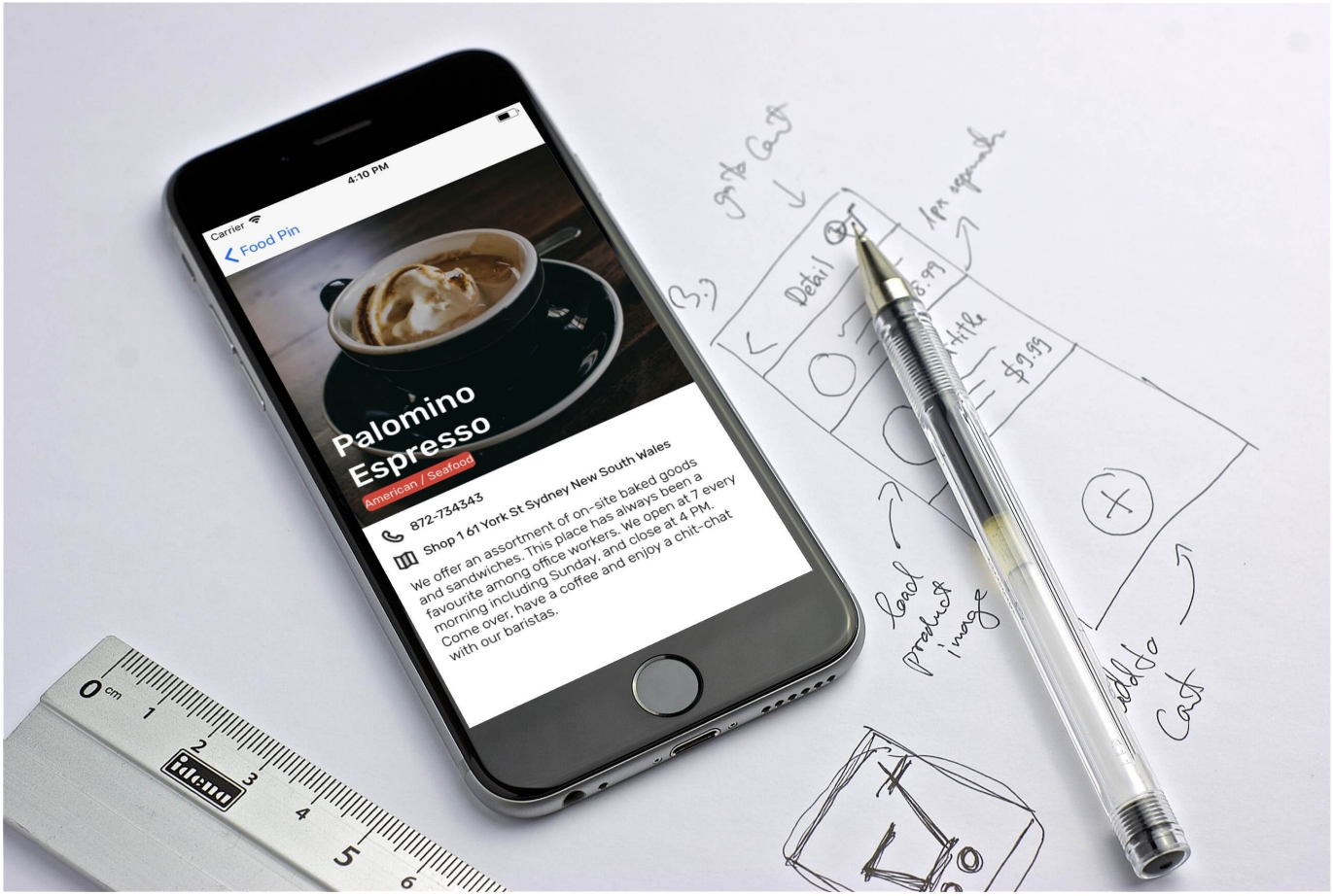
The Basic Theory of Object Oriented Programming

Like Objective-C, Swift is known as an Object Oriented Programming (OOP) language. OOP is a way of constructing software application composed of objects. In other words, the code written in an app in some ways deals with objects of some kinds. The `UIViewController`, `UIButton`, `UINavigationController`, and `UITableView` objects that you have used are some sample objects that come with the iOS SDK. Not only can you use the built-in objects, you have already created your own objects in the project, such as `RestaurantDetailViewController` and `RestaurantTableViewCell`.

First, why OOP? One important reason is that we want to decompose complex software into smaller pieces (or building block) which are easier to develop and manage. Here, the smaller pieces are the objects. Each object has its own responsibility, and objects

coordinate with each other in order to make the software work. That is the basic concept of OOP. To continue reading and access the full version of the book, please get the full copy here. You will also be able to access the full source code of the project.

Chapter 14
Detail View Enhancement, Custom Fonts, and Self Sizing Cells



> To create something exceptional, your mindset must be relentlessly focused on the smallest detail.
>
> - Giorgio Armani

The detail view is a bit primitive. Wouldn't it be great to improve the detail view to the one shown above? In this chapter we'll focus on three areas:

- Design the detail view of the FoodPin app using a custom table view.
- Learn how to use your own fonts.

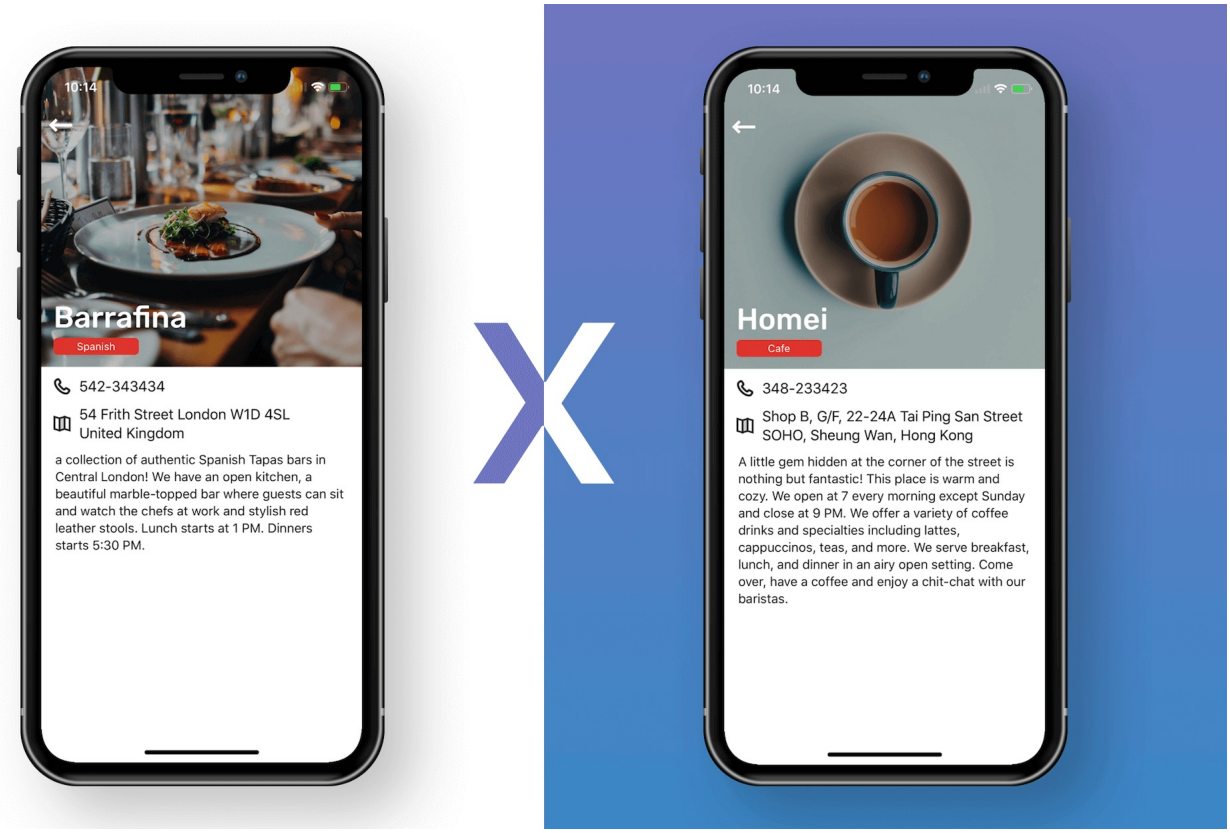
- Learn how to automatically resize a table view cell to fit its content.

We will cover a lot of materials in this chapter. You will probably need several hours to work on the project. I suggest you set aside other stuff and let yourself focus on it. On top of that, I assume you already understand how to build a custom table using prototype cells. If you forget what it is, go back and read chapter 9 again.

If you are ready, let's begin to tweak the detail view to make it look great.

What we're going to do is to present the restaurant information including name, type, location, phone and description in a well-designed table view. The restaurant image will be used as the header of the table.

Chapter 15
Navigation Bar Customization, Extensions, and Dynamic Type



> Fun is one of the most important and underrated ingredients in any successful venture. If you're not having fun, then it's probably time to call it quits and try something else.
>
> - Richard Branson

The detail view that we built is already great, but we will make it even more attractive in this chapter. Figure 15-1 shows the improved design with the following changes:

1. Customizing the navigation bar to make it transparent, so that the restaurant image shifts to the top of the screen.
2. Customizing the back button. Instead of using the default back button, we use our

own back icon and remove the title of the back button.

3. Changing the color of the status bar. To cater for dark content, it is better for us to change the color of the status bar to white.

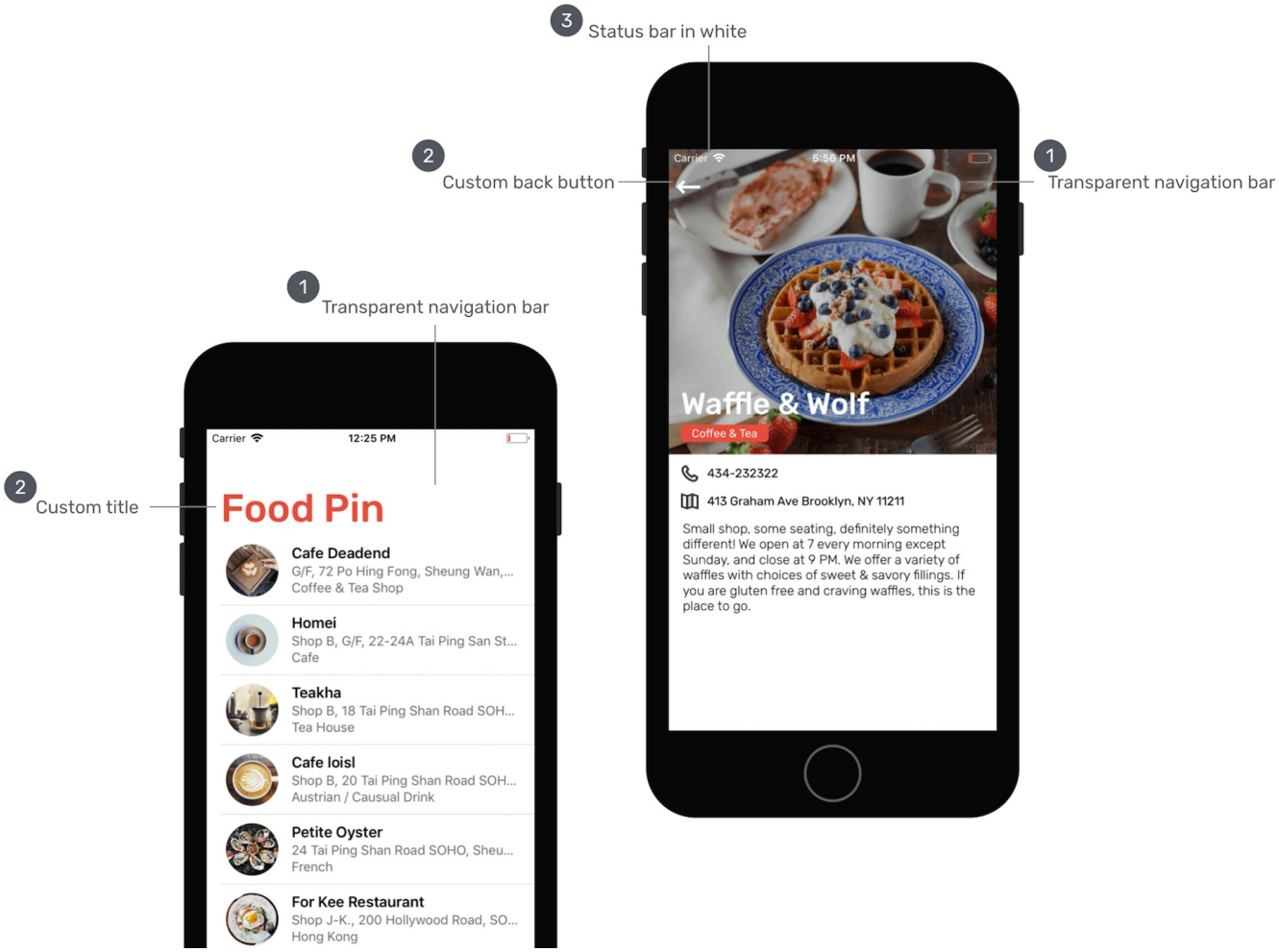


Figure 15-1. Customizing the navigation bar and status bar

On top of that, I want to show you how to customize the large title of the navigation bar on the home screen. There are three changes we are going to perform:

1. Customizing the background of the navigation bar and make it transparent.
2. Change the font of the large title.
3. Hide the navigation bar when users swipe the table view.

In the first two section of this chapter, we will focus on the customization of the navigation bar and status bar. Thereafter, I will discuss with you about Dynamic Type, a technology built into iOS that lets users choose their preferred text size. While working on the project, I will also show you how to use Swift extensions to simplify our code.

Let's start with the navigation bar customization.

To continue reading and access the full version of the book, please get the full copy here. You will also be able to access the full source code of the project.

Chapter 16
Working with Maps

> The longer it takes to develop, the less likely it is to launch.
>
> -Jason Fried, Basecamp

The MapKit framework provides APIs for developers to display maps, navigate through maps, add annotations for specific locations, add overlays on existing maps, etc. With the framework, you can embed a fully functional map interface into your app without any coding.

Starting from iOS 9, the MapKit framework allows developers to provide pin customization, transit and flyover support. In iOS 11, Apple introduced a new MarkerAnnotationView that gives the annotation a modern look. I will go over some of these features with you. In particular, you will learn a few things about the framework:

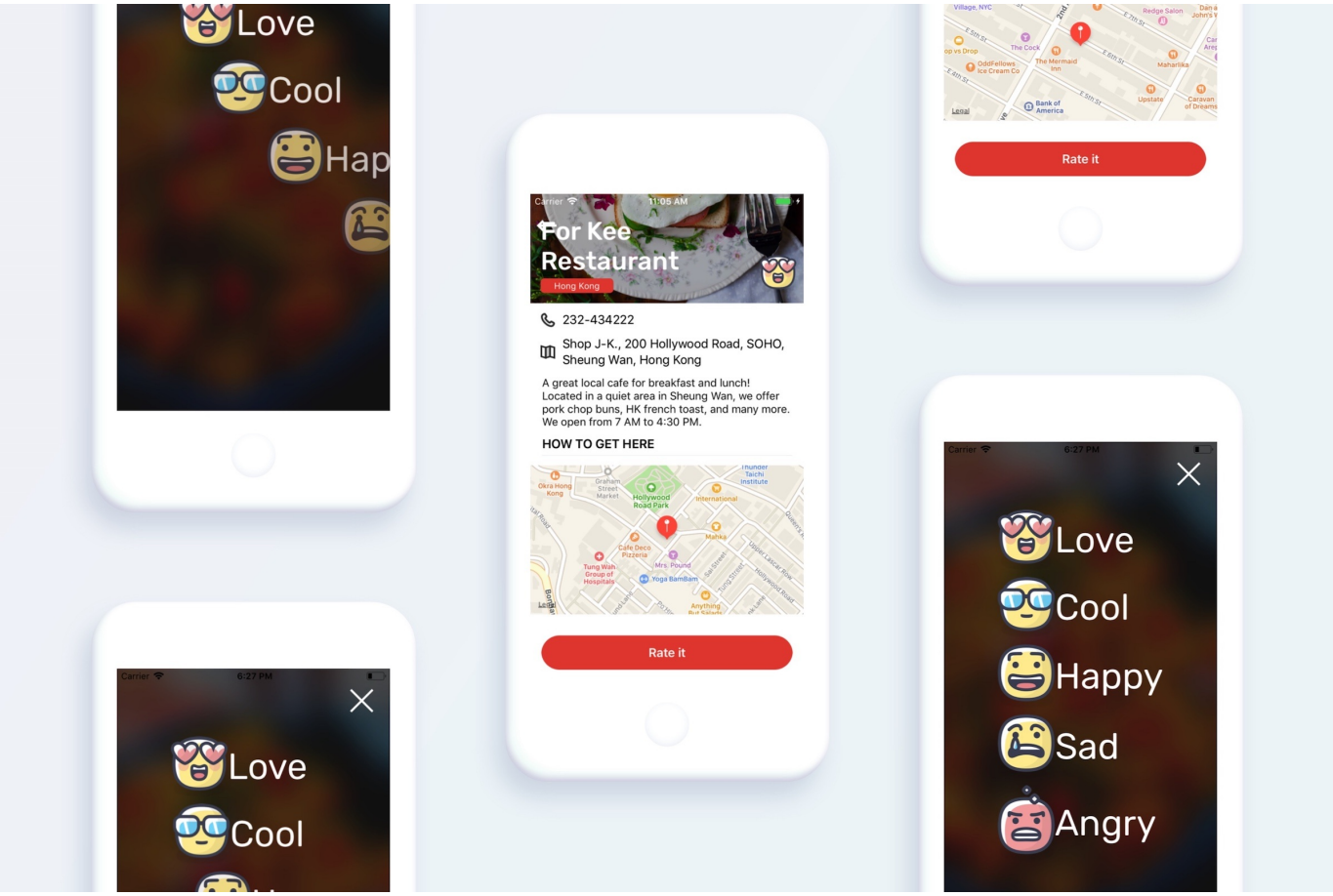
- How to embed a map in a view and a table view cell
- How to translate an address into coordinates using Geocoder
- How to add and customize a pin (i.e. annotation) on map
- How to customize an annotation

To give you a better understanding of the MapKit framework, we will add a map feature to the FoodPin app. After the change, the app will display a small map view in the detail screen. When a user taps that map view, the app will further bring up an interactive map that takes up the whole screen.

Cool, right? It's gonna be fun. Let's get started.

To continue reading and access the full version of the book, please get the full copy here. You will also be able to access the full source code of the project.

Chapter 17
Basic Animations, Visual Effects and Unwind Segues



> Animation can explain whatever the mind of man can conceive. This facility makes it the most versatile and explicit means of communication yet devised for quick mass appreciation.
>
> – Walk Disney

In iOS, creating sophisticated animations does not require you to write complex code. All you need to know is a single method in the `UIView` class:

```
UIView.animateWithDuration(1.0, animations)
```

There are several variations of the method that provide additional configuration and features. This is the basis of every view animation.

First things first, what's an animation? How is an animation created? Animation is a simulation of motion and shape change by rapidly displaying a series of static images (or frames). It is an illusion that an object is moving or changing in size. For instance, a growing circle animation is actually created by displaying a sequence of frames. It starts with a dot. The circle in each frame is a bit larger than the one before it. This creates an illusion that the dot grows bigger and bigger. Figure 17-1 illustrates the sequence of static images. I keep the example simple so the figure displays 5 frames. To achieve a smooth transition and animation, you'd need to develop several more frames.

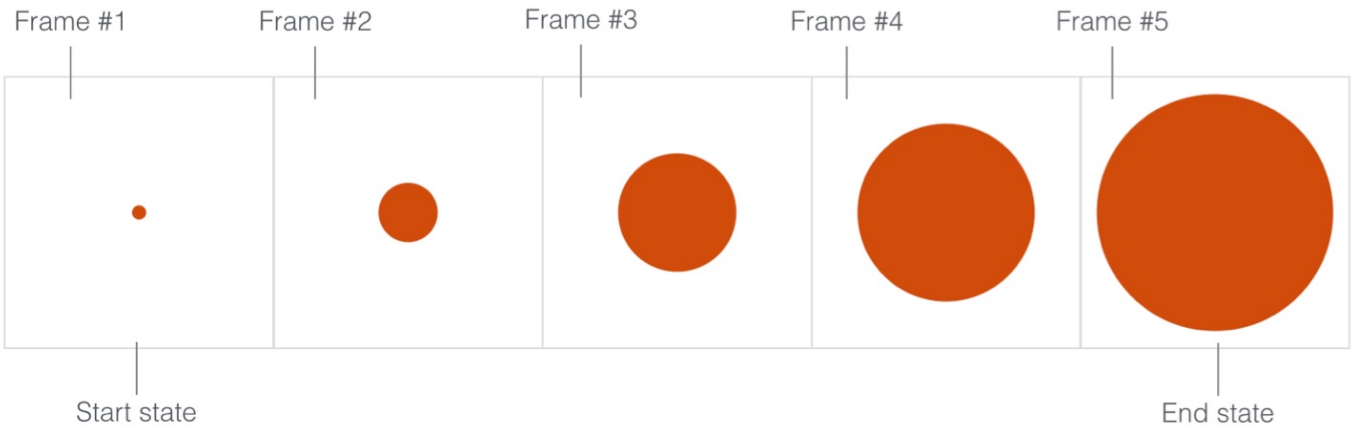


Figure 17-1. Sequence of frames for creating an animation

Now that you have a basic idea of how animation works, how will you create an animation in iOS? Consider our growing circle example. You know the animation starts with a dot (i.e. start state) and ends with a big red circle (i.e. end state). The challenge is to generate the frames between these states. You may need to think of an algorithm and write hundreds of lines of code to generate the series of frames in between. UIView animation helps you compute the frames between the start and end state resulting in a

smooth animation. You simply specify the start state and tell UIView the end state by calling the `UIView.animateWithDuration` method. The rest is handled by iOS. Sounds good, right?

There is no better way to understand the technique than by working on a real example. We will add some basic animations to our FoodPin app. Here is what we're going to do:

- Add a "Rate it" button in the detail view
- When a user taps the button, it brings up a review view controller, in which the buttons are animated, for the user to rate the restaurant.

Through building the review view controller, I will show you how to create basic animations using `UIView`. On top of that, I will show you how to create a blurred background using the built-in API and pass data between view controllers using unwind segues.

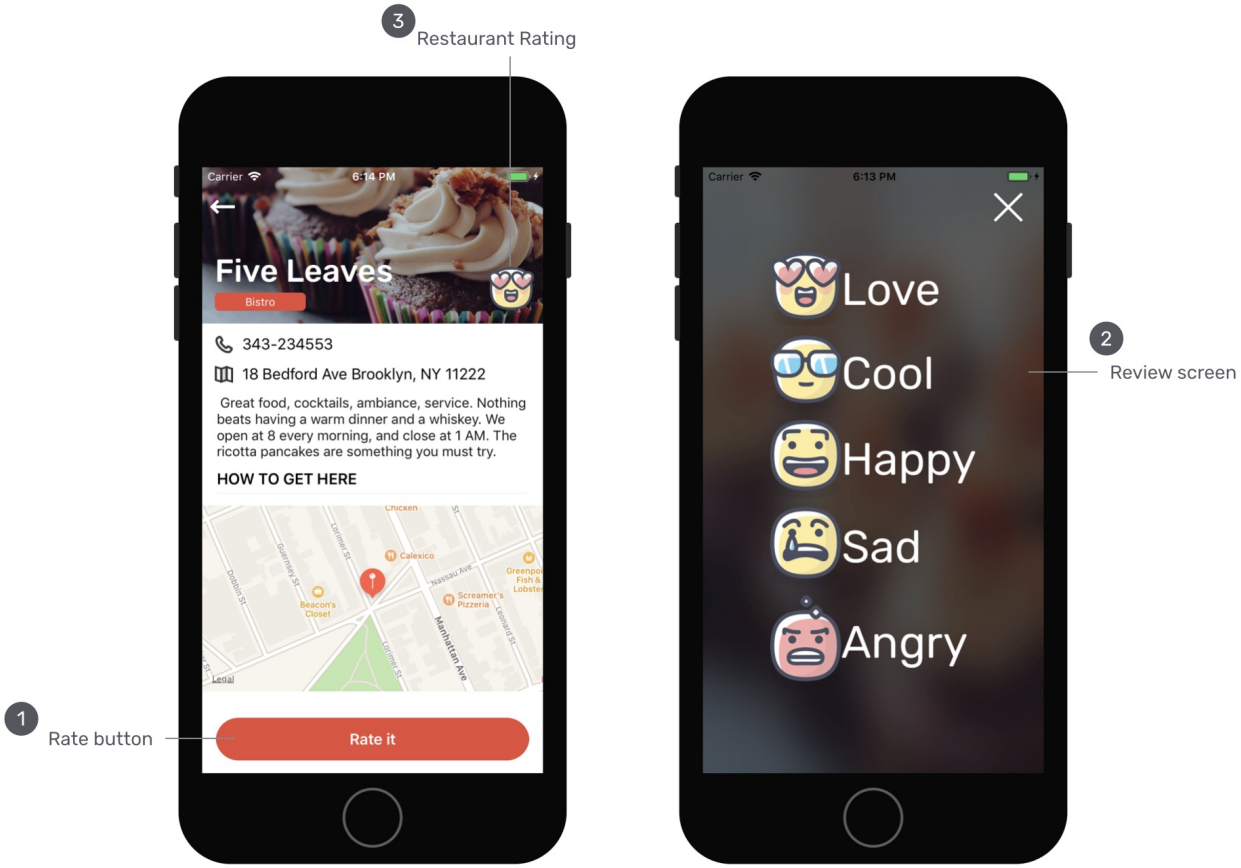
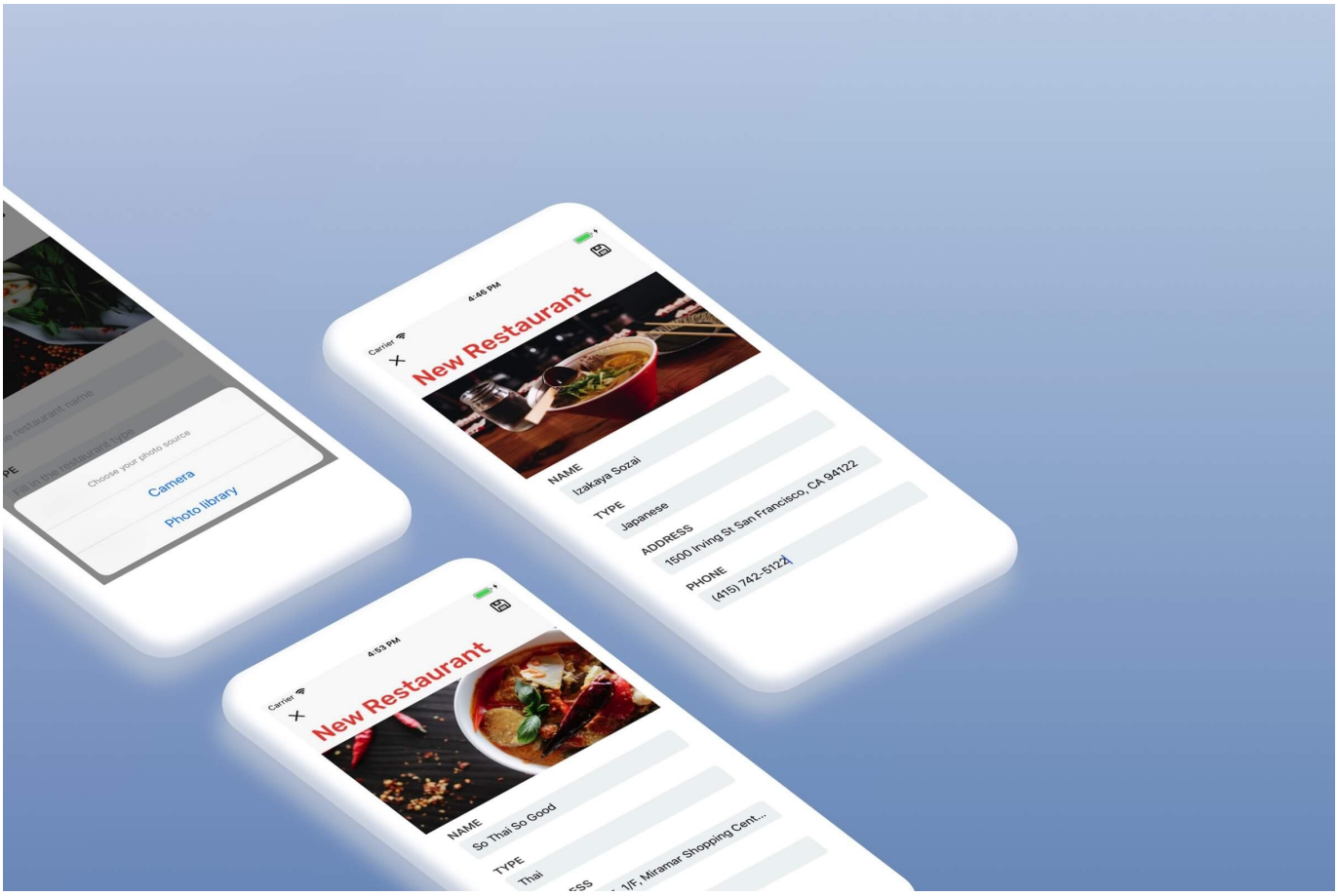


Figure 17-2. Adding a rating feature in the detail screen

To continue reading and access the full version of the book, please get the full copy here.

Chapter 18
Working with Static Table Views, Camera and NSLayoutConstraint



> My biggest motivation? Just to keep challenging myself. I see life almost like one long University education that I never had. Every day I'm learning something new.
>
> - Richard Branson

Up till now, the FoodPin app is only capable of displaying content. We need to find a way for users to add a new restaurant. In this chapter, we will create a new screen that displays an input form for collecting restaurant information. In the form, it will let users pick a restaurant photo from the built-in photo library. You'll learn a number of techniques:

- How to create a form using a static table view
- How to use `UIImagePickerController` to select a photo from the built-in photo library and take photos
- How to define auto layout constraints programmatically using `NSLayoutConstraint`

In the first few chapters of the book, we have gone through the basics of table views. The table views that I covered are dynamic in nature. Usually, you create a prototype cell and populate it with dynamic content. However, table views are not limited to present dynamic content. Sometimes, you may just want to use a table view to present a form or a setting screen. In this case, a static table view is what you need. Static table views are ideal for situations where there are a pre-defined number of data items to be displayed.

Xcode allows developers to create static table views with minimal code. To illustrate how easy you can use Interface Builder to implement a static table view, we will build a new screen for adding a new restaurant.

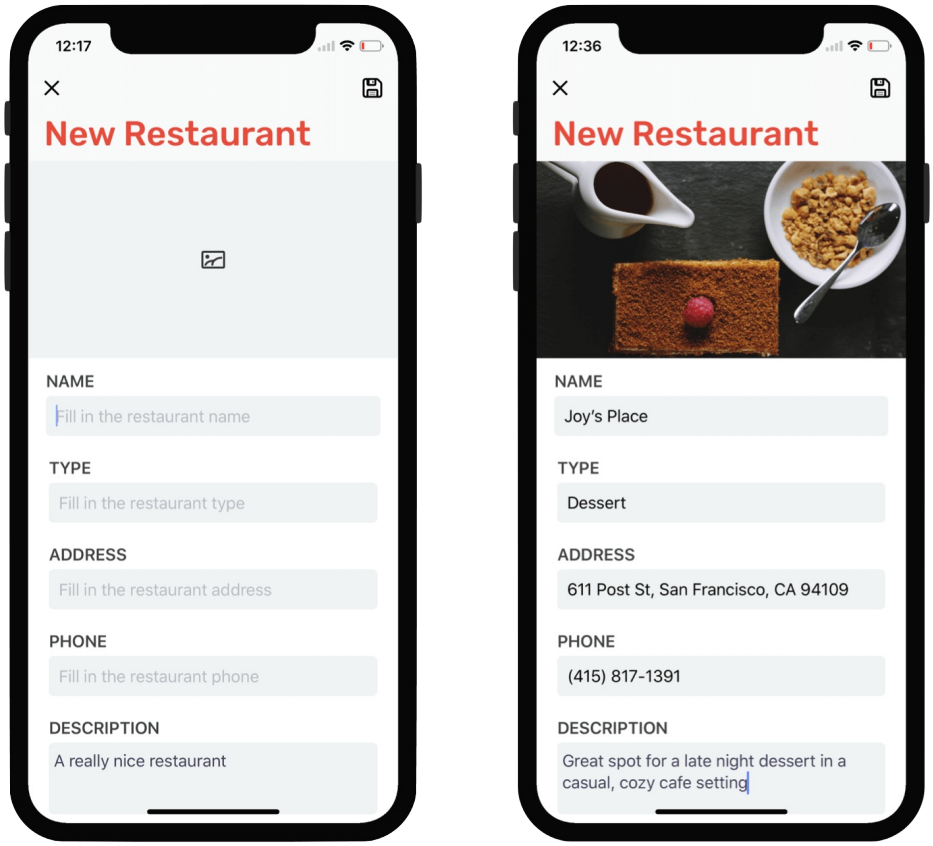
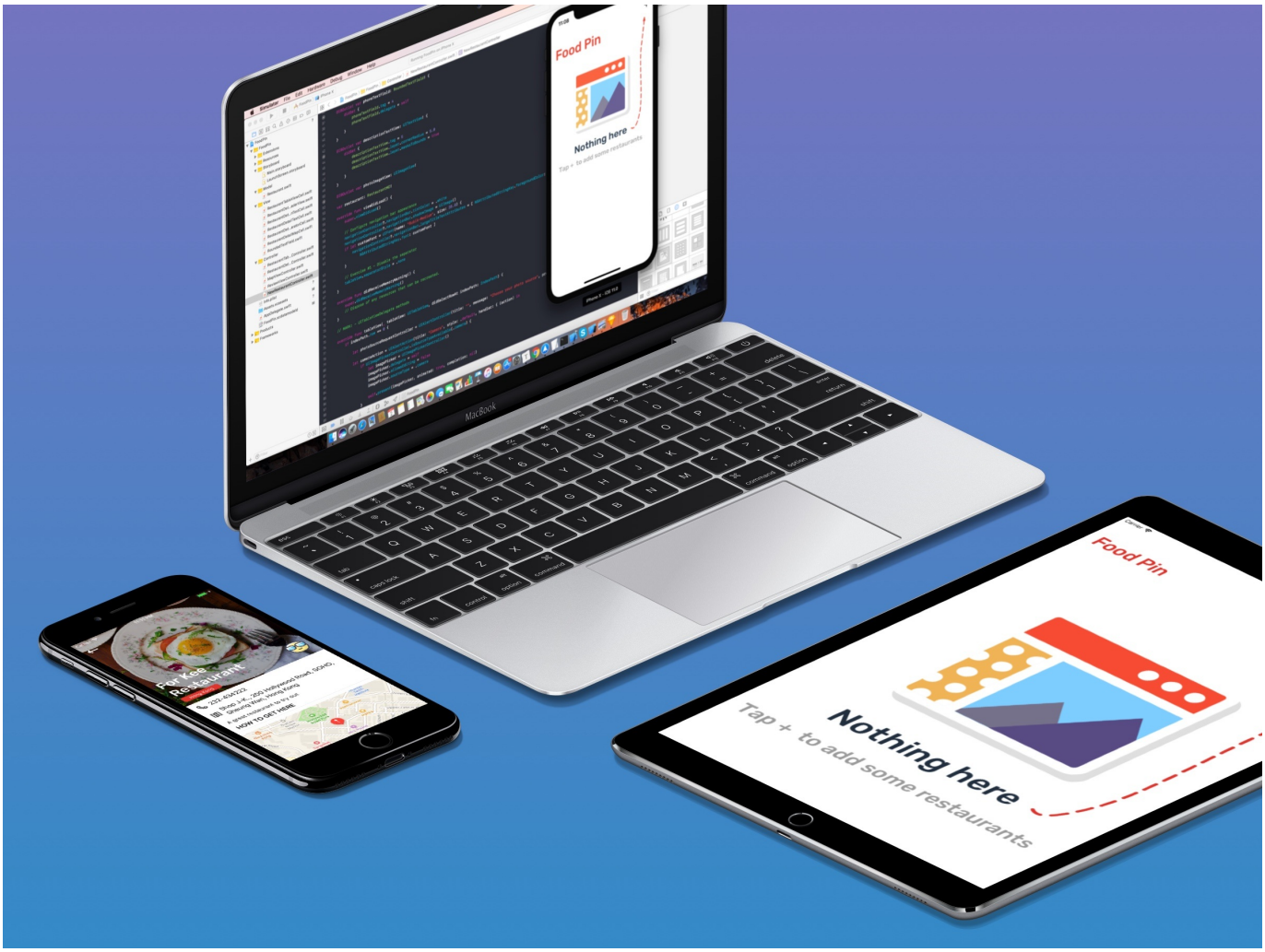


Figure 18-1. Creating a New Restaurant screen for adding a new restaurant

Let's get started.

To continue reading and access the full version of the book, please get the full copy here. You will also be able to access the full source code of the project.

Chapter 19
Working with Core Data



> Learn not to add too many features right away, and get the core idea built and tested.
>
> – Leah Culver

Congratulations on making it this far! By now you've already built a simple app for users to list their favorite restaurants. If you've worked on the previous exercise, you should understand the fundamentals of how to add a restaurant; I've tried to keep things simple

and focus on the basics of UITableView. Up to this point, all restaurants have been predefined in the source code and stored in an array. If you want to add a restaurant, the simplest way is to append the new restaurant to the existing `restaurants` array.

However, if you do it that way, you can't save the new restaurant permanently. Data stored in memory (e.g. array) is volatile. Once you quit the app, all the changes are gone. We need to find a way to save the data in a persistent manner.

To save the data permanently, we'll need to save in a persistent storage-like file or database. By saving the data to a database, for example, the data will be safe even if the app quits or crashes. Files are another way to save data, but they are more suitable for storing small amounts of data that do not require frequent changes. For instance, files are commonly used for storing application settings like the Info.plist file.

The FoodPin app may need to store thousands of restaurant records. Users may also add or remove the restaurant records quite frequently. In this case, a database is a suitable way to handle a large set of data. In this chapter, I will walk you through the Core Data framework and show you how to use it to manage data in the database. You will make a lot of changes to your existing FoodPin project, but after going through this chapter your app will allow users to save their favorite restaurants persistently.

Chapter 20
Search Bar and UISearchController

> I knew that if I failed I wouldn't regret that, but I knew the one thing I might regret is not trying.
>
> – Jeff Bezos

For most of the table-based apps, it is common to have a search bar at the top of the screen. How can you implement a search bar for data searching? In this chapter, we will add a search bar to the FoodPin app. With the search bar, we will enhance the app to let users search through the available restaurants.

Since the release of iOS 8, a new class called `UISearchController` was introduced to replace the `UISearchDisplayController` API that has been around since iOS 3. The old API is now deprecated. If you have some experience with iOS 7 or older version of SDK, remember to use `UISearchController` instead.

The `UISearchController` API simplifies the way to create a search bar and manage search results. You're no longer limited to embed search in table view controller but can use it in any view controller like collection view controller. Even more, it offers developers the flexibility to influence the search bar animation through a custom animator object.

In iOS 11, Apple further simplified the implementation of the search bar. It introduces a new `searchController` property on navigationItem of the navigation bar. With just a few lines of code, you will be able to add a search bar to the navigation bar. You will understand what I mean in a while.

With `UISearchController`, adding a search bar to your app is quite an easy task. Let's get started to implement a default search bar and see how we can filter the restaurant data.

To continue reading and access the full version of the book, please get the full copy here. You will also be able to access the full source code of the project.

Chapter 21
Building Walkthrough Screens with UIPageViewController and Container Views

> If you're interested in the living heart of what you do, focus on building things rather than talking about them.
>
> - Ryan Freitas, About.me

For the very first time launching an app, you probably find a series of walkthrough (or tutorial) screens. It's a common practice for mobile apps to step users through a multi-screen tutorial where all the features are demonstrated. Some said your app design probably fails if your app needs walkthrough screens. Personally, I don't hate walkthrough screens and find most of them pretty useful. Just make sure you keep it short. Don't take it too far to include long and boring tutorials. Here, I'm not going to argue with you whether you should or should not include walkthrough screens in your app. I just want to show you how.

In this chapter, we'll discuss how to use `UIPageViewController` to create walkthrough screens and you will learn how to use a special type of view known as *Container View*.

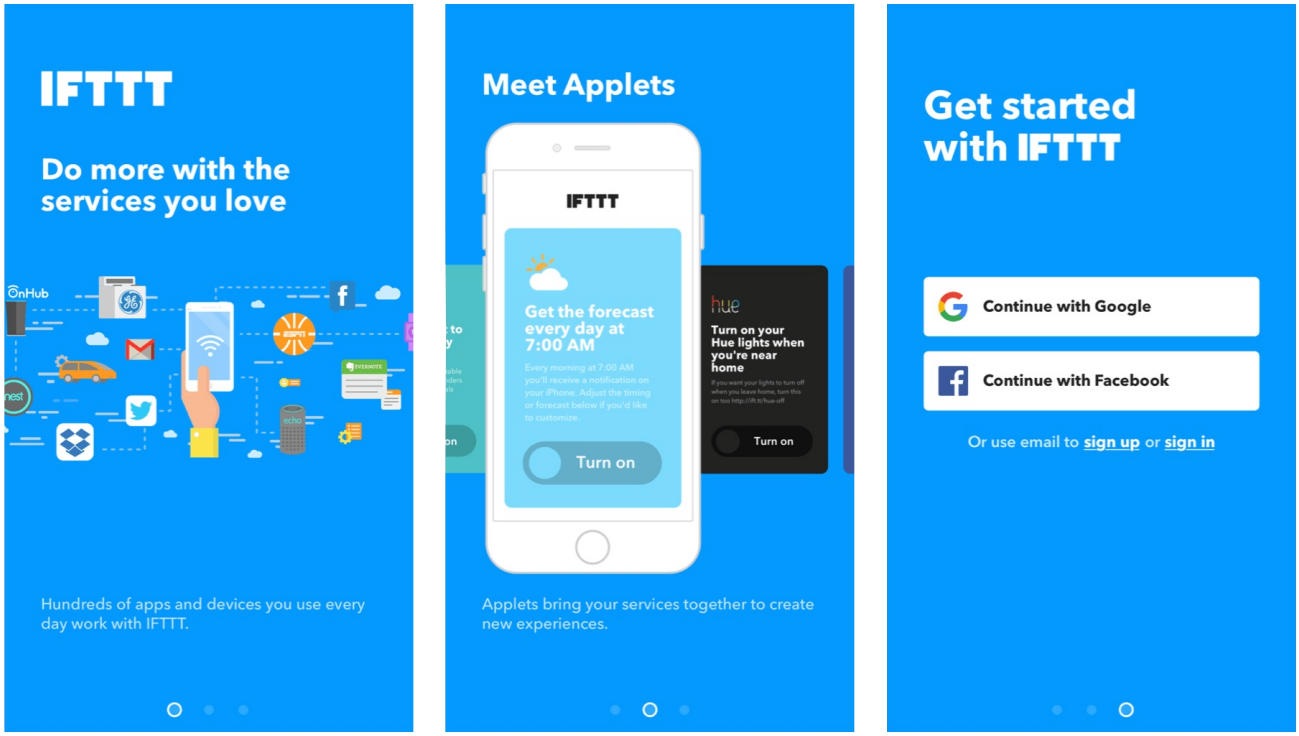


Figure 21-1. Sample Walkthrough Screens of IFTTT

One of the ways to implement this type of walkthrough screen is to utilize the `UIPageViewController` class. It lets developers build pages of content, for which each page is managed by its own view controller. The class has a built-in support for scrolling transition. With `UIPageViewController`, users can easily navigate between multiple pages

through simple gestures. The page view controller is not limited to creating walkthrough screens. You can find examples of page view implementation in games like Angry Birds (the page that shows all available levels) or in book apps (the page that displays the table of contents).

The `UIPageViewController` class is a highly configurable class. You're allowed to define:

- **the orientation of the page views** – vertical or horizontal
- **the transition style** – page curl transition style or scrolling transition style
- **the location of the spine** – only applicable to page curl transition style
- **the space between pages** – only applicable to scrolling transition style to define the inter-page spacing

We will add a simple walkthrough for the FoodPin app. By implementing this new feature, you will learn how `UIPageViewController` works along the way. That said, we'll not demonstrate every option of `UIPageViewController`; we'll just use the scrolling transition style to display a series of walkthrough screens. With the basic understanding of the UIPageViewController, however, I believe you should be able to explore other features in the page view controller.

Let's get started.

A Quick Look at the Walkthrough Screens

Let's have a quick look at the walkthrough screens. The app will display a total of three tutorial screens. The user will be able to navigate between pages by swiping through the screen or tapping the *NEXT* button.

In the last screen of the walkthrough, it displays a *Get Started* button. When the user taps the button, the walkthrough screen will be dismissed and never be shown again. At any time, the user can skip the walkthrough screens by tapping the *Skip* button. Figure 21-2 shows the screenshots of the walkthrough.

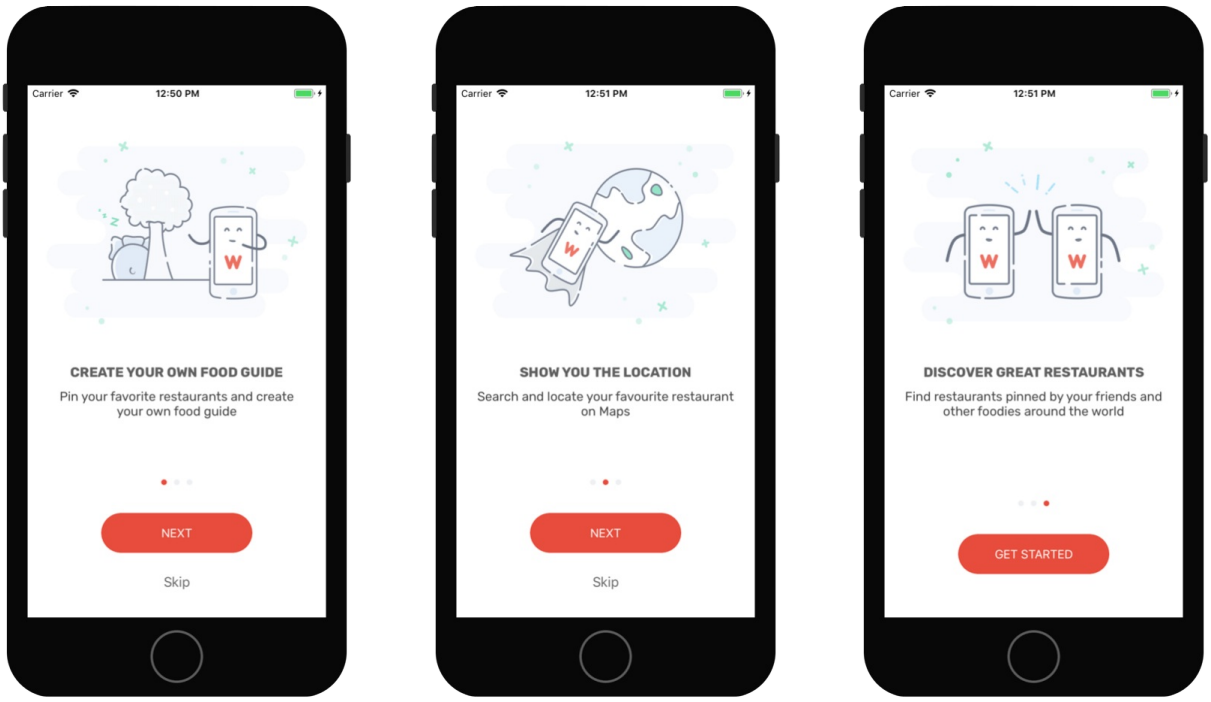


Figure 21-2. Walkthrough screens for the FoodPin app

To continue reading and access the full version of the book, please get the full copy here. You will also be able to access the full source code of the project.

Chapter 22
Exploring Tab Bar Controller and Storyboard References

> If you're trying to achieve, there will be roadblocks. I've had them; everybody has had them. But obstacles don't have to stop you. If you run into a wall, don't turn around and give up. Figure out how to climb it, go through it, or work around it.
>
> - Michael Jordan

The tab bar is a row of persistently visible buttons, which open different features of the app, at the bottom of the screen. Once a less-prominent UI design in the mainstream, the tab bar design becomes popular again lately.

Before the debut of the large screen devices, there were only 3.5-inch and 4-inch iPhones. One drawback of tab bars is that you sacrifice a bit of screen estate. This was really an issue for small screen devices. As Apple rolled out the iPhone 6 and 6 Plus with larger screen sizes in late 2014, app developers started to replace the existing menus of their apps with tab bars. The Facebook app has switched from a sidebar menu design to a tab bar. Other popular apps like Whatsapp, Twitter, Quora, Instagram, and Apple Music also adopt tab bars for navigation.

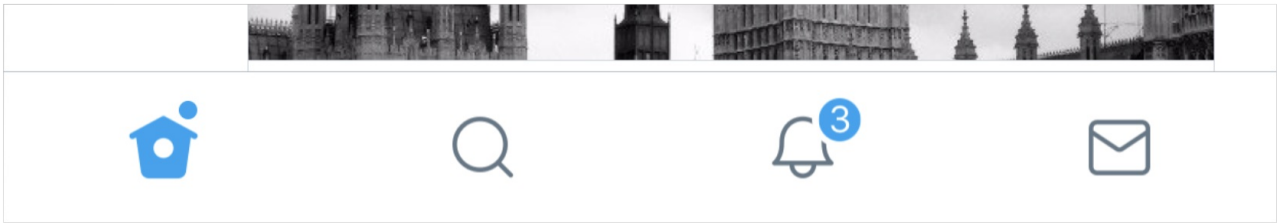


Figure 22-1. A tab bar in the Twitter app

Tab bars allow users to access the core features of an app quickly, with just a single tap. Though it takes up a bit of screen estate, it is worth it.

While navigation controllers let users navigate hierarchical content by managing a stack of view controllers, tab bars manage multiple view controllers that don't necessarily have a relation to one another. Normally a tab bar controller contains at least two tabs and you're allowed to add up to five tabs depending on your needs.

In this chapter, I will walk you through the implementation of tab bars and see how we can customize its appearance. We will also take a look at another feature of Interface Builder called *storyboard references*.

Building a Tab Bar Controller

First, let's open the FoodPin project. We're going to create a tab bar with three items:

- **Favorites** - this is the restaurant list screen.
- **Discover** - this is a new screen to discover favorite restaurants recommended by

your friends or other foodies around the globe. We will implement this tab in the iCloud chapter.

- **About** - this is the "About" the about screen of the app. Once again we'll keep this page blank until the next chapter.

Creating a tab bar is easy and doesn't require to write a line of code. All you need to do is embed a set of view controllers in a tab bar controller using Interface Builder.

To continue reading and access the full version of the book, please get the full copy here. You will also be able to access the full source code of the project.

Chapter 23
Getting Started with WKWebView and SFSafariViewController

> I've got a theory that if you give 100% all of the time, somehow things will work out in the end.
>
> - Larry Bird

It is very common you need to display web content in your apps. From iOS 9 and onward, it provides three options for developers to show web content:

- **Mobile Safari** - the iOS SDK provides APIs for you to open a specific URL in the built-in Mobile Safari browser. In this case, your users temporarily leave the application and switch to Safari.
- **UIWebView / WKWebView** - Before the release of iOS 9, this is the most convenient way to embed web content in your app. You can think of `UIWebView` as a stripped-down version of Safari. It is responsible to load a URL request and display the web content. `WKWebView`, introduced in iOS 8, is an improved version of `UIWebView`. It has the benefit of the Nitro JavaScript engine and offers more features. If you just need to display a specific web page, `WKWebView` is the best option for this scenario.
- **SFSafariViewController** - this is a new controller introduced in iOS 9. While `UIWebView` allows you to embed web content in your apps, you have to build a custom web view to offer a complete web browsing experience. For example, `UIWebView` doesn't come with the Back/Forward button that lets users navigate back and forth the browsing history. In order to provide the feature, you have to develop a custom web browser using `UIWebView`. In iOS 9, Apple introduced `SFSafariViewController` to save developers from creating our own web browser. By using `SFSafariViewController`, your users can enjoy all the features of Mobile Safari without leaving your apps.

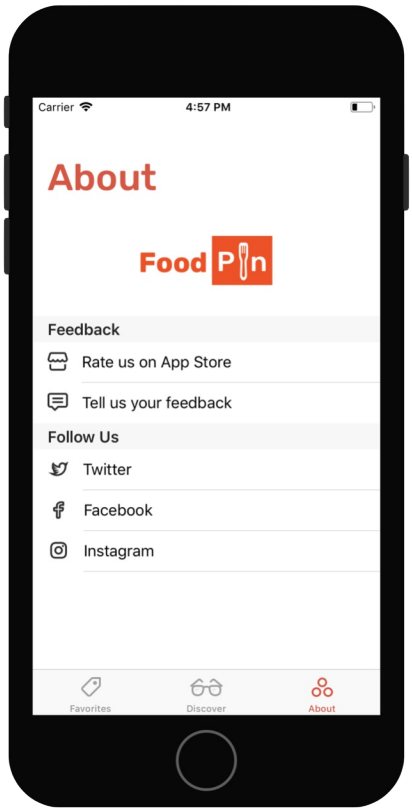


Figure 23-1. The About Screen

In this chapter, I will walk you through all the options and show you how to use them to display web content. We will not talk about `UIWebView` as it is deprecated. While the class is still available in the iOS SDK, you should always use `WKWebView` instead.

We have created `about.storyboard` in the previous chapter. However, we didn't provide any implementation yet. Take a look at figure 23-1. That is the *About* screen we're going to create, and here are how each of the rows works:

- **Rate us on App Store** - when selected, we will load a specific iTunes link in Mobile Safari. Users will leave the current app and switch to the App Store.
- **Tell us your feedback** - when selected, we will load a *Contact Us* web page using `WKWebView`.
- **Twitter / Facebook / Pinterest** - each of these items has its own link for the corresponding social profile. We will use `SFSafariViewController` to load these links.

Sounds interesting, right? Let's get started.

To continue reading and access the full version of the book, please get the full copy here. You will also be able to access the full source code of the project.

Chapter 24
Exploring CloudKit

> The most impressive people I know spent their time with their head down getting shit down for a long, long time.
>
> - Sam Altman

Let's start with some history. When Steve Jobs unveiled iCloud to complement iOS 5 and OS X Lion at Apple's annual Worldwide Developers Conference (WWDC) in 2011, it gained a lot of attention but came as no surprise. Apps and games could store data on the cloud and have it automatically synchronize between Macs and iOS devices.

But iCloud fell short as a cloud server.

Developers are not allowed to use iCloud to store public data for sharing. It is limited to sharing information between multiple devices that belong to the same user. Take our Food Pin app as an example - you can't use the classic version of iCloud to store your favorite restaurants publicly and make them available for other app users. The data, that you store on iCloud, can only be read by you.

If you wanted to build a social app to share data amongst users at that time, you either came up with your home-brewed backend server (plus server-side APIs for data transfer, user authentication, etc) or relied on other cloud service providers such as Firebase and Parse.

```
Note: Parse was a very popular cloud service at the time. But
Facebook announced the demise of the service on January 28,
2016.
```

In 2014, the company reimagined iCloud functionality and offered entirely new ways for developers, as well as, users to interact with iCloud. The introduction of CloudKit represents a big improvement over its predecessor and the offering is huge for developers. You can develop a social networking app or add social sharing features easily using CloudKit.

What if you have a web app and you want to access the same data on iCloud as your iOS app? Apple further takes CloudKit to the next level by introducing CloudKit web services or CloudKit JS, a JavaScript library. You can develop a web app with the new library to

access the same data on iCloud as your app.

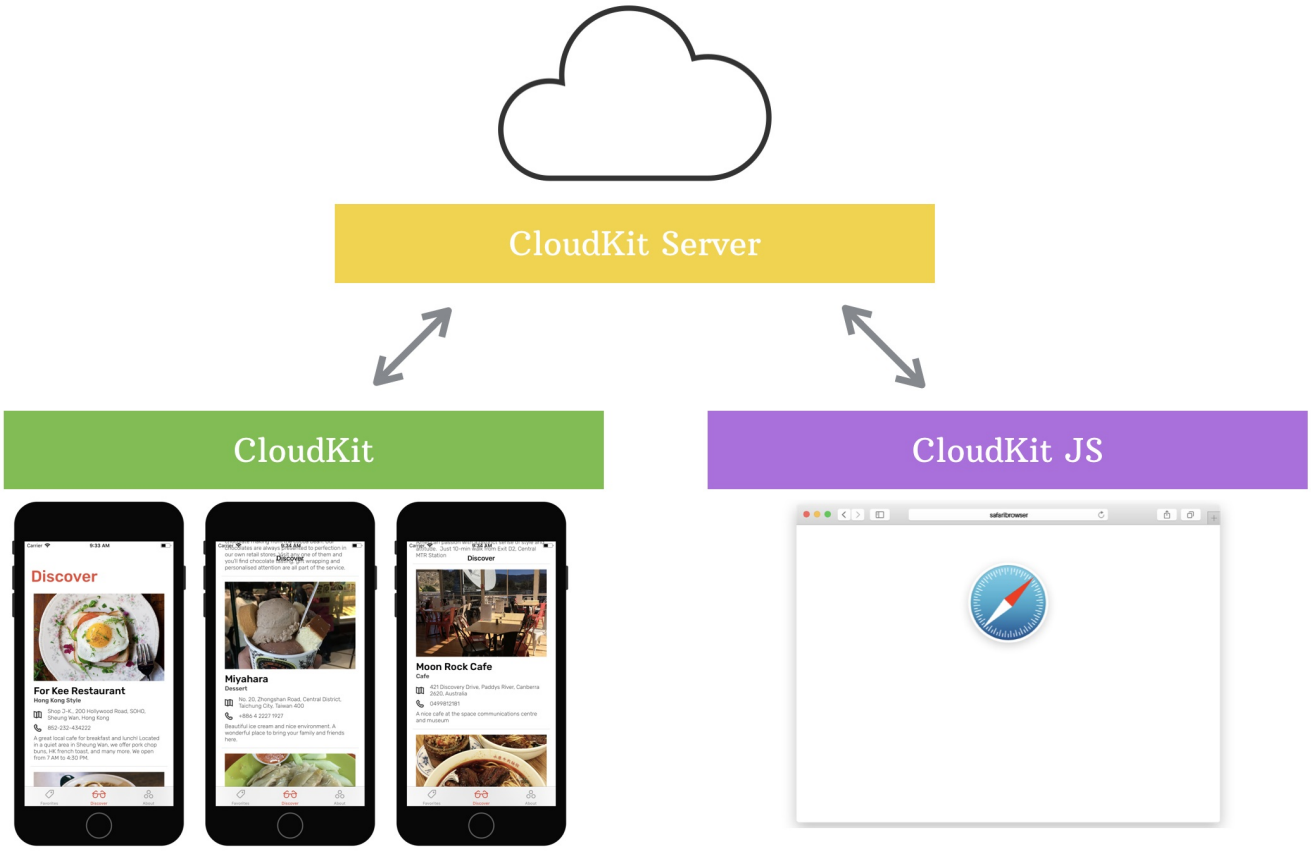


Figure 24-1. Storing your data to the cloud

In WWDC 2016, Apple announced the introduction of Shared Database. Not only can you store your data publicly or privately, CloudKit now lets you store and share the data with a group of users.

CloudKit makes developers' lives easier by eliminating the need to develop our own server solutions. With minimal setup and coding, CloudKit empowers your app to store data, including structured data and assets, in the cloud.

Best of all, you can get started with CloudKit for free (with limits). It starts with:

- 10GB for assets (e.g. images)
- 100MB for database
- 2GB for data transfer

As your app becomes more popular, the CloudKit storage grows with you and adds an additional 250MB for every single user. For each developer account, you can scale all the way up to the following limits:

- 1PB assets
- 10TB database
- 200TB data transfer

That's a massive amount of free storage and is sufficient for the vast majority of apps. According to Apple's iCloud calculator, the storage should be enough for about 10 million free users.

> With CloudKit, we were able to focus on building our app and even squeeze in a few extras.
>
> - Hipstamatic

In this chapter, I will walk you through the integration of iCloud using the CloudKit framework. But we will only focus on the Public database.

As always, you will learn the APIs by implementing a feature of the FoodPin app. We will enhance the app to let users share their favorite restaurants anonymously, and all users can view other's favorites in the *Discover* tab. It's going to be fun.

There is a catch, however. You have to enroll in the Apple Developer Program (USD99/year). Apple opens up the CloudKit storage for paid developers only. If you're serious about creating your app, it's time to enroll in the program and build some CloudKit-based apps.

Chapter 25
Localizing Your App to Reach More Users

> Good code is its own best documentation. As you're about to add a comment, ask yourself, "How can I improve the code so that this comment isn't needed?" Improve the code and then document it to make it even clearer.
>
> - Steve McConnell

In this chapter, let's talk about localization. The iOS devices including iPhone and iPad are available globally. The App Store is available in more than 150 countries around the world. Your users are from different countries and speak different languages. To deliver a great user experience and reach a global audience, you definitely want to make your app available in multiple languages. The process of adapting an app to support a particular language is usually known as *localization*.

Xcode has the built-in support for localization. It's fairly easy for developers to localize an app through the localization feature and a few API calls.

You may have heard of the terms: *localization* and *internationalization*. You probably think that both terms refer to the process of translation; that's partially correct. In iOS development, internationalization is considered a milestone in building a localized app. Before your app can be adapted to different languages, you design and structure the app to be language and region independent. This process is known as *internationalization*. For instance, your app displays a price field. As you may know, some countries use a dot to indicate decimal place (e.g. $1000.50), while many other countries use a comma instead (e.g. $1000,50). The internationalization process involves designing the price field so that it can be adapted to different regions.

Localization is the process of adapting an internationalized app for different languages and regions. This involves translating static and visible text to a specific language and adding country-specific elements such as images, videos, and sounds.

In this chapter, we'll localize the FoodPin app into Chinese and German. However, don't expect me to translate all the text in the app - I just want to show you the overall process of localization using Xcode.

Chapter 26
Deploying and Testing Your App on a Real iOS Device

> If we want users to like our software, we should design it to behave like a likable person: respectful, generous and helpful.
>
> - Alan Cooper

Up till now, you have been running and testing your app on the built-in simulator. The simulator is a great companion for app development especially if you do not own an iPhone. While the simulator is good, you can never rely completely on the simulator. It's not recommended to submit your app to App Store without testing it on a real device. Chances are there are some bugs that only show up when your app runs on a physical iPhone or over the cellular network. If you're serious about building a great app, this is a must to test your app on a real device before releasing it to your users.

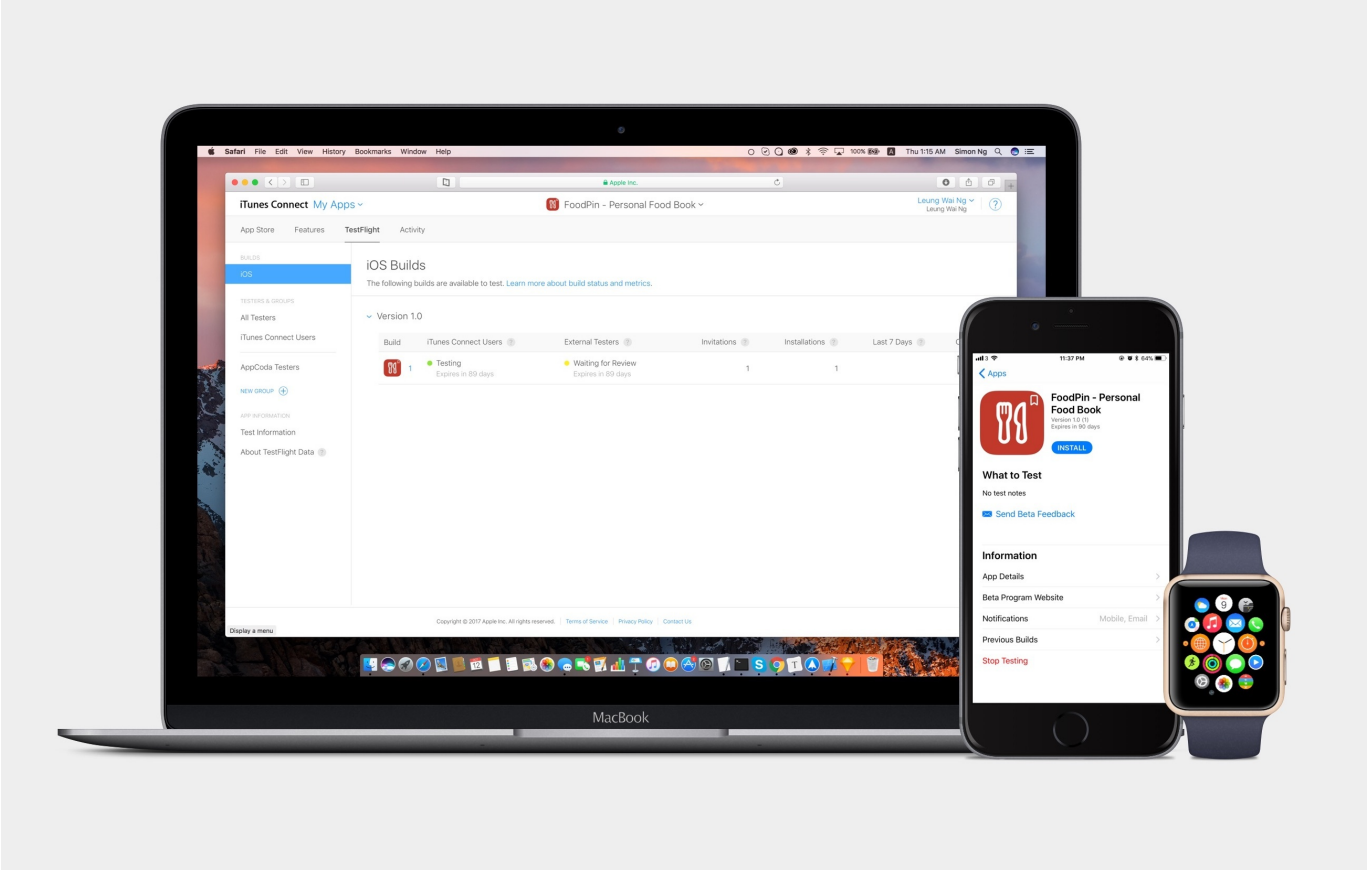
One great news, especially for aspiring iOS developers, is that Apple no longer requires you to enroll in the Apple Developer Program before you can test your app on an iOS device. You can simply sign in Xcode with your Apple ID, and your app is ready to run on your iPhone or iPad. However, please note that if your app makes use of the services like CloudKit and Push Notifications, you still need to enroll in the Apple Developer Program, which costs $99 per year. I know, for some, this is a significant amount of money. But if you read the book from the very beginning and are still with me, I believe you have demonstrated a strong determination to build an app and deploy it to your audience. It's not easy to make it this far! So why stop here? If you're not on a tight budget, I highly recommend you enroll in the program so that you can continue to learn the rest of the materials and most importantly, submit your app to App Store.

To test an app on a physical device, you will need to perform a few configurations:

- Request your development certificate
- Create an App ID for your app
- Configure your device for development
- Create a provisioning profile for your app

In the old days of iOS development, you had to manage the above configurations all on your own through the iOS Provisioning Portal (or Member Center). The modern version of Xcode automates the whole signing and configuration processes by using a feature called *Automatic Signing*. This makes your life a lot easier. You will see what I mean shortly.

Chapter 27
Beta Testing with TestFlight and CloudKit Production Deployment



> If you are not embarrassed by the first version of your product, you've launched too late.
>
> - Reid Hoffman, LinkedIn

Now that you have completed the testing of your app on a real device, what's next? Submit your app directly to App Store and make it available for download? Yes, you can if your app is a simple one. If you're developing a high-quality app, don't rush to get your app out, as I suggest you beta test your app before the actual release.

A beta test is a step in the cycle of a software product release. I know you have tested your app using the built-in simulator and on your own device. Interestingly, you may not be able to uncover some of the bugs, even though you're the app creator. By going through beta tests, you would be amazed at the number of flaws discovered at this stage. Beta testing is generally opened to a select number of users. They may be your potential app users, your blog readers, your Facebook followers, your colleagues, friends or even family members. The whole point of beta testing is to let a small group of real people get their hands on your app, test it, and provide feedback. You want your beta tester to discover as many bugs as possible in this stage so that you can fix them before rolling out your app to the public.

You may be wondering how can you conduct a beta test for your app, how beta testers run your app before it's available on App Store and how testers report bugs?

In iOS 8, Apple released a new tool called **TestFlight** to streamline the beta testing. You may have heard of TestFlight before. It has been around for several years as an independent mobile platform for mobile app testing. In February 2014, Apple acquired TestFlight's parent company, *Burstly*. Now TestFlight is integrated into App Store Connect (previously known as iTunes Connect) and iOS that allows you to invite beta testers using just their email addresses.

TestFlight makes a distinction between beta testers and internal users. Conceptually, both can be your testers at the beta testing stage. However, TestFlight refers internal users as members of your development team who have been assigned the Technical or Admin role in App Store Connect. You're allowed to invite up to 25 internal users to test your app. A beta tester, on the other hand, is considered as an external user outside your team and company. You can invite up to 10,000 users to beta test your app.

If you are going to let external users test your app, the app must be approved by Apple before you can send out the invitation This restriction doesn't apply to internal users. Your internal users can begin beta testing once you upload your app to App Store Connect.

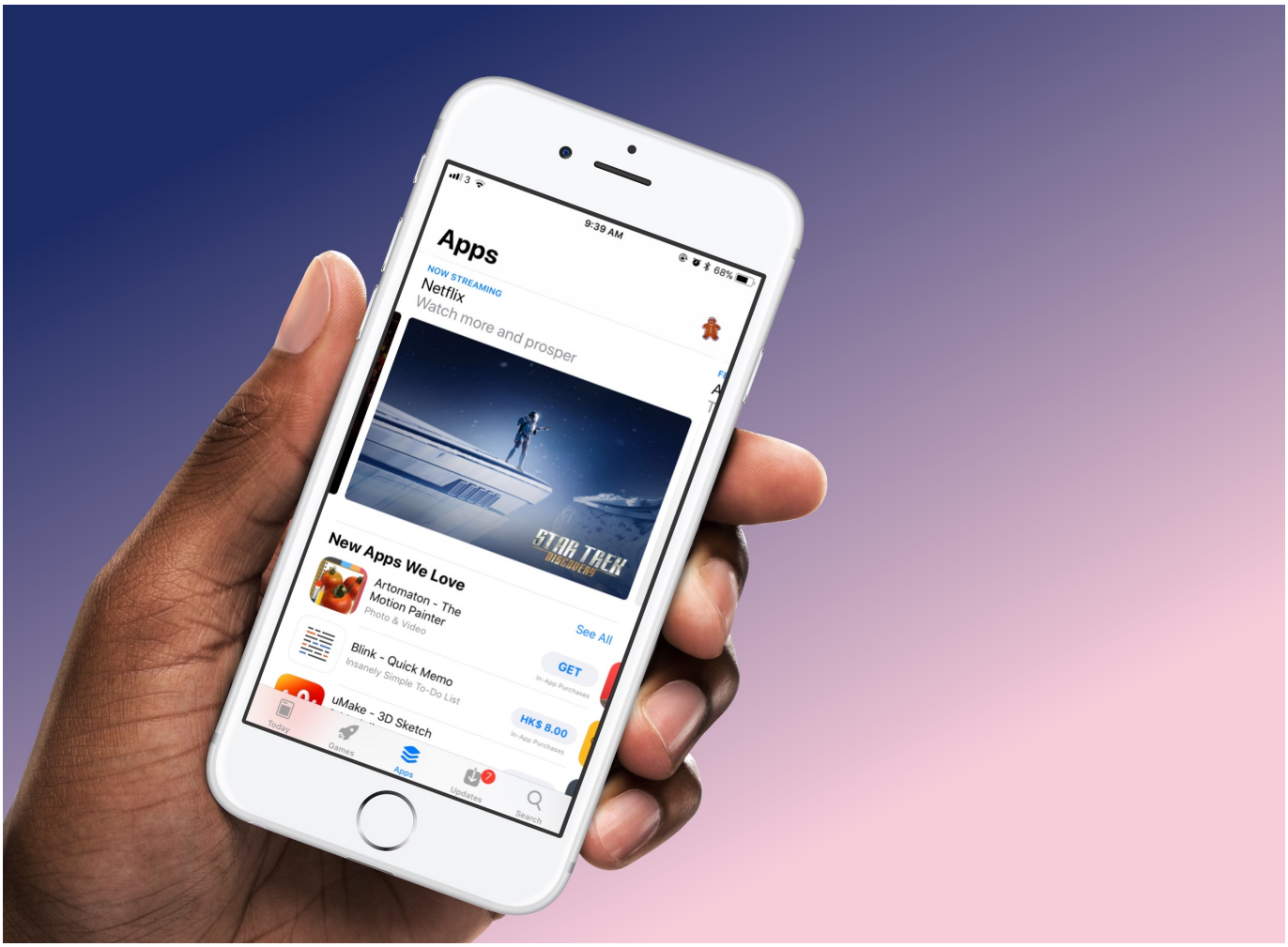
Similar to CloudKit, TestFlight is not available for free. You have to enroll in the Apple Developer Program before you can access TestFlight.

In this chapter, I will walk you through the beta test process using TestFlight. In general, we will go through the tasks below to distribute an app for beta testing:

- Create an app record on App Store Connect.
- Update the build string.
- Archive and validate your app.
- Upload your app to App Store Connect.
- Manage beta testing in App Store Connect.

To continue reading and access the full version of the book, please get the full copy here. You will also be able to access the full source code of the project.

Chapter 28
Submit Your App to App Store



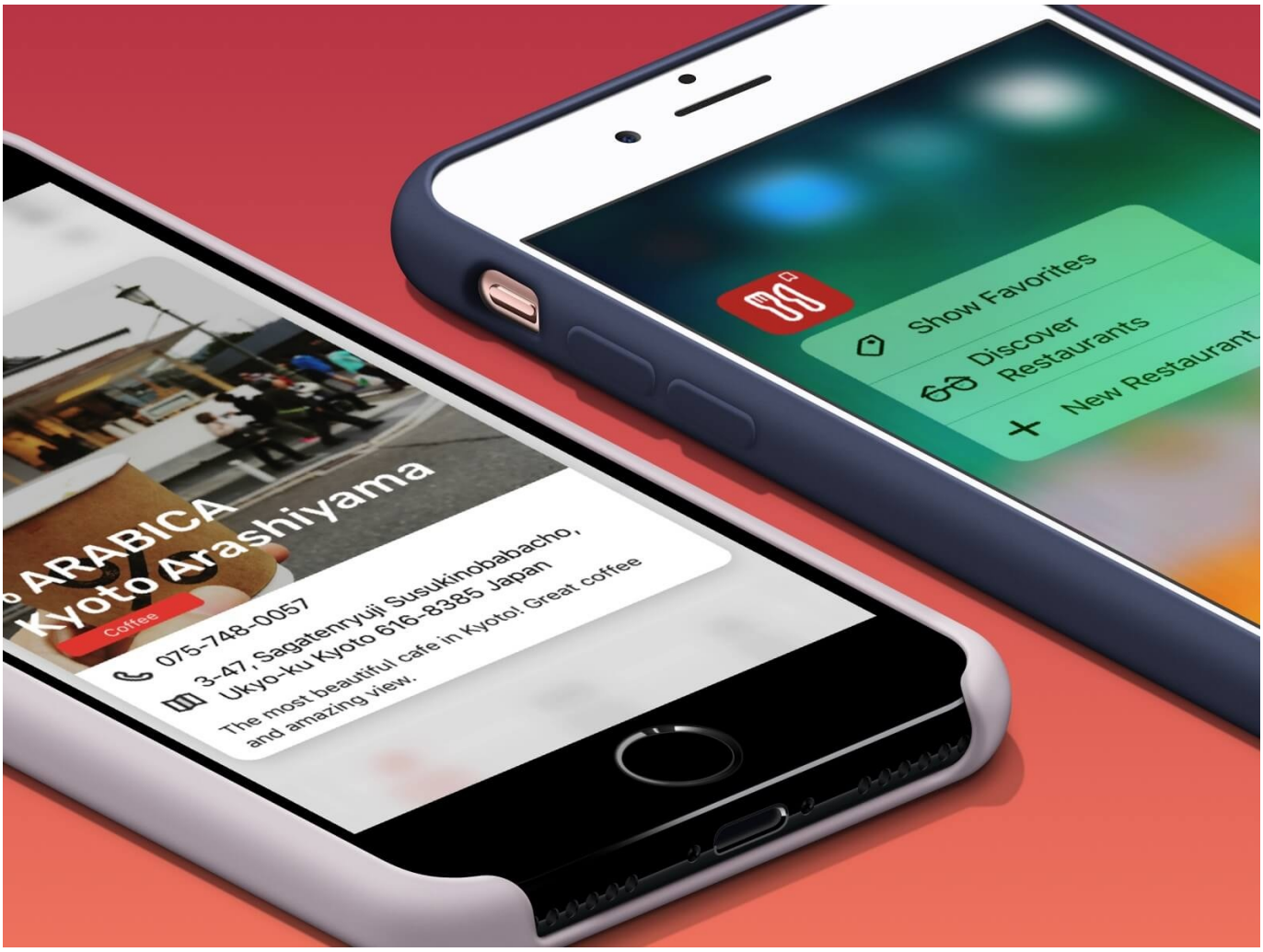
> Don't let people tell you your ideas won't work. If you're passionate about an idea that's stuck in your head, find a way to build it so you can prove to yourself that it doesn't work.
>
> - Dennis Crowley, FourSquare

Congratulations! You have probably worked weeks or months before coming to the last step of app development. After beta testing and numerous bug fixing, your app is finally ready for submission.

You already uploaded your app binary to iTunes Connect in the previous chapter, so it is quite straightforward to submit your app to App Store. Once you submit your app, it will be reviewed by Apple's App Review team before publishing it onto App Store. For a first-time app developer, submitting an app to the app store can be a nightmare. You may need to submit your app multiple times before Apple approves it.

In this chapter, I will walk you through the app submission process and give you some guidelines to minimize the possibility of app rejections.

Chapter 29
Adopting 3D Touch



As your first app, the FoodPin app is pretty good. That said, if you want to make it even better and adopt some modern technologies provided by the iOS devices, I have two more chapters for you.

Since the release of the iPhone 6s and 6s Plus, Apple introduced us an entirely new way to interact with our phones known as 3D Touch. It literally adds a new dimension to the user interface and offers a new kind of user experience. Not only can it sense your touch, iPhone can now sense how much pressure you apply to the display.

With 3D Touch, you now have three new ways to interact with the iPhones: *Quick Actions*, *Peek*, and *Pop*. Quick actions are essentially shortcuts for your applications. When you press an app icon a little harder, it shows a set of quick actions, each of which allows you to jump straight to a particular part of an app. It simply saves you a few "taps".

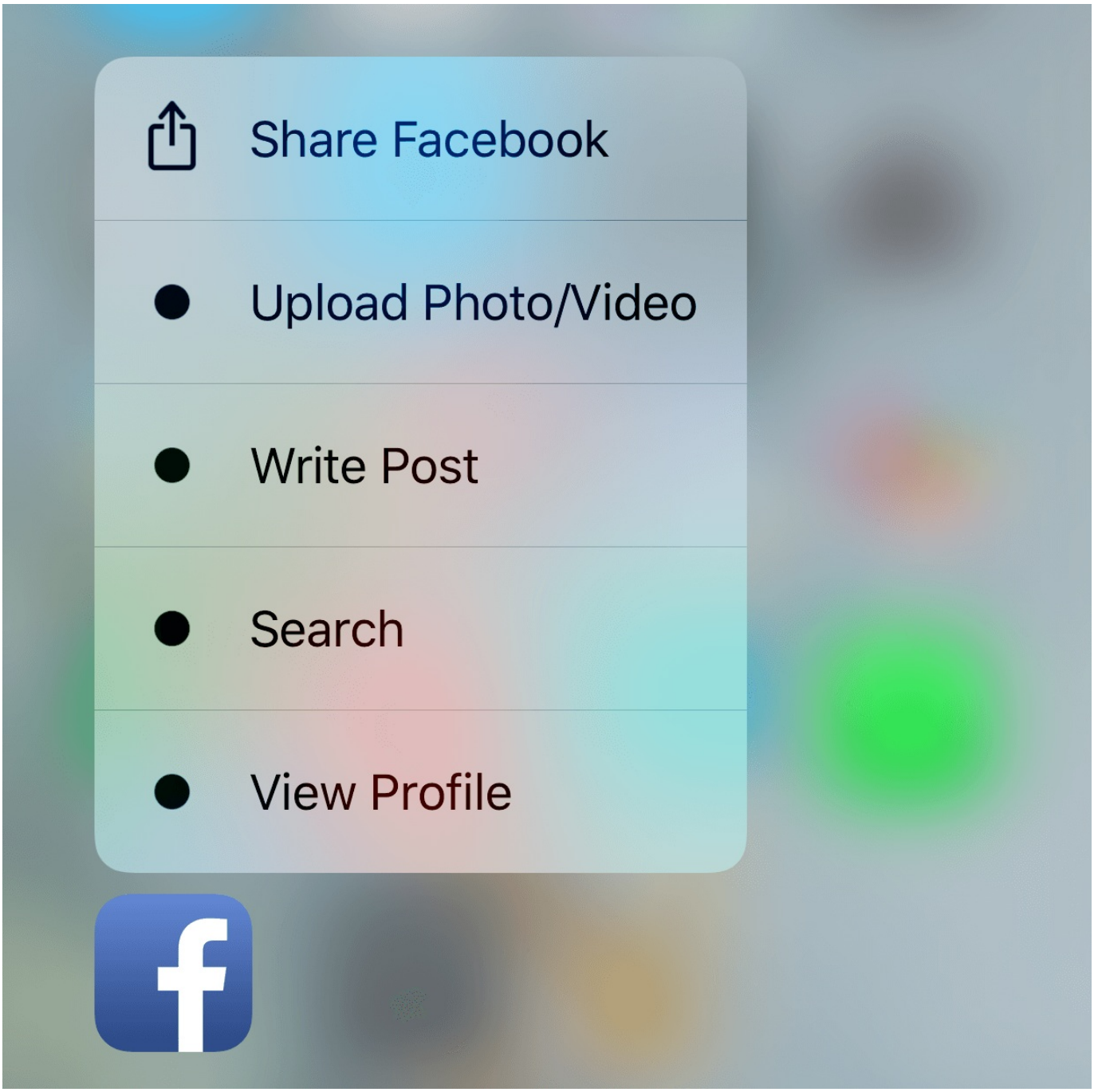


Figure 29-1. Sample Quick Actions

Peep and Pop purposely want to give users a quicker access to the app's contents. Take the built-in Photos app as an example. When you press a little harder on a photo, the app "Peeks" the photo in a pop-up. If the preview is good enough for you, simply release your

finger and you're back to the photo album. But if you want more than a preview, just press a bit harder to "Pop" into a full view.

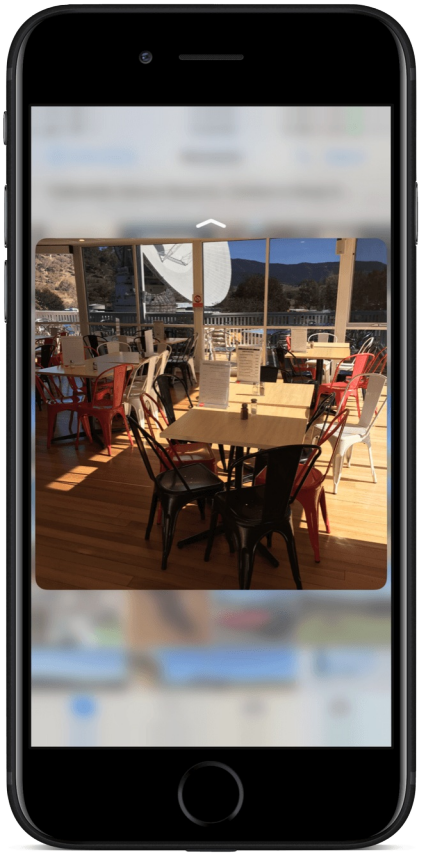


Figure 29-2. Peek and Pop in Photos app

Since iOS 9, Apple provided a set of APIs for developers to work with 3D Touch. In this chapter, I will go through some of the new APIs with you. More specifically, we will add Quick Actions, Peek, and Pop features to the FoodPin app.

Home Screen Quick Actions

First, let's talk about Quick Actions. Apple offers two types of quick actions: *static* and *dynamic*. Static quick actions are hardcoded in the `Info.plist` file. Once the user installs the app, the quick actions will be accessible, even before the first launch of the app. As the name suggests, dynamic quick actions are dynamic in nature. The app creates and

updates the quick actions at runtime. Take the News app as an example. Its quick actions show some of the most frequently accessed channels. They must be dynamic because these channels will change over time. For some quick actions, it even bundles a widget that shows users some useful information without opening the app.

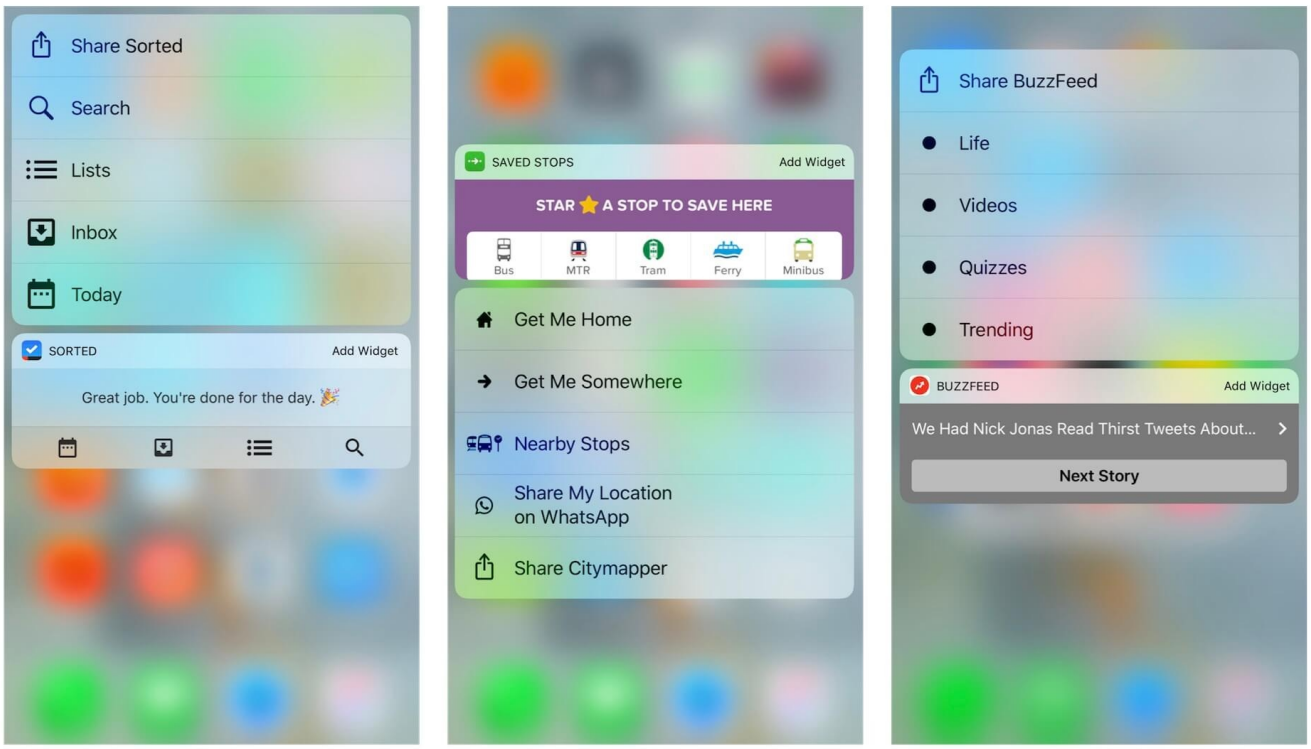


Figure 29-3. Quick actions Widgets

To continue reading and access the full version of the book, please get the full copy here.

Chapter 30
Developing User Notifications in iOS

Prior to iOS 10, user notifications are plain and simple. No rich graphics or media. It is just in text format. Depending on the user's context, the notification can appear on the lock screen or home screen. If the user misses any of the notifications, they can bring up the Notification Center to reveal all pending notifications.

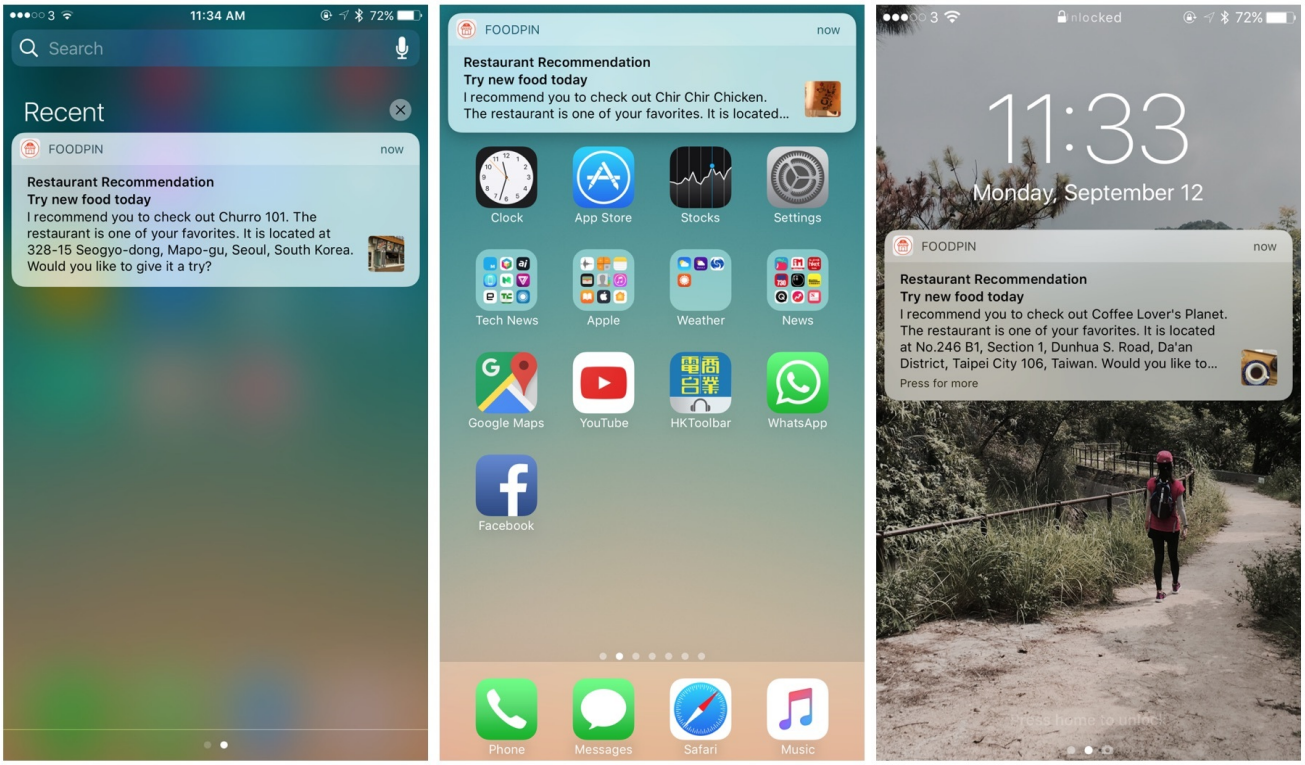


Figure 30-1. Sample user notifications in lock screen, home screen, and notifcation center

Since the release of iOS 10, Apple has revamped the notification system to support user notifications in rich content and custom notification UI. By rich content, it means you can include static images, animated GIFs, videos, and audios in the notifications. Figure 30-2 gives you an idea of the new notifications.

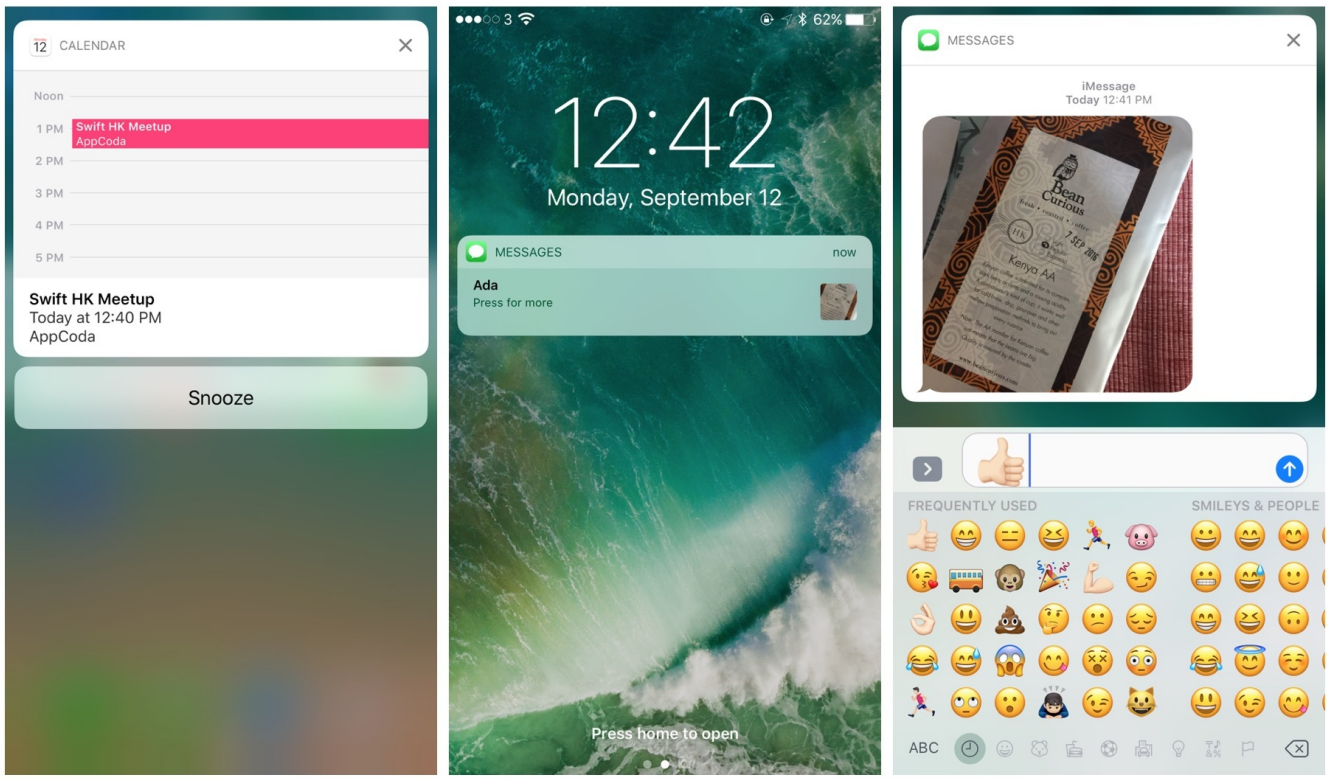


Figure 30-2. Sample user notifications in rich content

You may have heard of push notifications, which have been widely adopted in messaging apps. Actually, user notifications can be classified into two types: *local notifications* and *remote notifications*. Local notifications are triggered by the application itself and contained on the user's device. For example, a location-based application will send users a notification when they are in a particular area. Or a to-do list app displays a notification when an item is close to the due date.

Remote notifications are usually initiated by server side applications that reside on remote servers. When the server application wants to send messages to users, it sends a notification to Apple Push Notification Service (or APNS for short). The service then forwards the notification to users' devices.

We're not going to talk about the implementation of remote notifications in this chapter. Instead, we will focus on discussing local notifications, and show you how to use the new User Notifications framework to implement the rich-content notifications.

To continue reading and access the full version of the book, please get the full copy here. You will also be able to access the full source code of the project.

Appendix - Swift Basics

Swift is a new programming language for developing iOS, macOS, watchOS and tvOS apps. As compared to Objective-C, Swift is a neat language and will definitely make developing iOS apps easier. In this appendix, I will give you a brief introduction of Swift. This doesn't serve as a complete guide for the programming language but gives you all the essentials to kick start Swift programming. For the full reference, please refer to the official documentation (https://swift.org/documentation/).

Variables, Constants, and Type Inference

In Swift, you declare variables with the `var` keyword and constants using the `let` keyword. Here is an example:

```
var numberOfRows = 30
let maxNumberOfRows = 100
```

These are the two keywords you need to know for variable and constant declaration. You use the `let` keyword for storing a value that is unchanged. Otherwise, use `var` keyword for storing values that can be changed.

Isn't it easier than Objective-C?

What's interesting is that Swift allows you to use nearly any character for both variable and constant names. You can even use an emoji character for the naming.

You may notice a huge difference in variable declaration between Objective-C and Swift. In Objective-C, developers have to specify explicitly the type information when declaring a variable. Be it an `int` or `double` or `NSString`, etc.

```
const int count = 10;
double price = 23.55;
NSString *myMessage = @"Objective-C is not dead yet!";
```

It's your responsibility to specify the type. In Swift, you no longer need to annotate variables with type information. It provides a huge feature known as *Type inference*. This feature enables the compiler to deduce the type automatically by examining the values you provide in the variable.

```
let count = 10
// count is inferred to be of type Int
var price = 23.55
// price is inferred to be of type Double
var myMessage = "Swift is the future!"
// myMessage is inferred to be of type String
```

It makes variable and constant declaration much simpler, as compared to Objective-C. Swift provides an option to explicitly specify the type information if you wish. The below example shows how to specify type information when declaring a variable in Swift:

```
var myMessage: String = "Swift is the future!"
```

No Semicolons

In Objective-C, you need to end each statement in your code with a semicolon. If you forget to do so, you will end up with a compilation error. As you can see from the above examples, Swift doesn't require you to write a semicolon (`;`) after each statement, though you can still do so if you like.

```
var myMessage = "No semicolon is needed"
```

Basic String Manipulation

In Swift, strings are represented by the `String` type, which is fully Unicode-compliant. You can declare strings as variables or constants:

```
let dontModifyMe = "You cannot modify this string"
var modifyMe = "You can modify this string"
```

In Objective-C, you have to choose between `NSString` and `NSMutableString` classes to indicate whether the string can be modified. You do not need to make a choice in Swift. Whenever you assign a string to a variable (i.e. `var`), the string can be modified in your code.

Swift simplifies string manipulating and allows you to create a new string from a mix of constants, variables, literals, as well as, expressions. Concatenating strings is super easy. Simply add two strings together using the `+` operator:

```
let firstMessage = "Swift is awesome."
let secondMessage = "What do you think?"
var message = firstMessage + secondMessage
print(message)
```

Swift automatically combines both messages and you should see the following message in console. Note that print is a global function in Swift to print the message in console.

Swift is awesome. What do you think? You can do that in Objective-C by using the `stringWithFormat:` method. But isn't the Swift version more readable?

```
NSString *firstMessage = @"Swift is awesome. ";
NSString *secondMessage = @"What do you think?";
NSString *message = [NSString stringWithFormat:@"%@%@", firstMessage, secondMessag
e];
NSLog(@"%@", message);
```

String comparison is more straightforward. You can use the `==` operator to compare two strings like this:

```
var string1 = "Hello"
var string2 = "Hello"
if string1 == string2 {
    print("Both are the same")
}
```

Arrays

The syntax of declaring an array in Swift is similar to that in Objective-C. Here is an example:

Objective-C:

```
NSArray *recipes = @[@"Egg Benedict", @"Mushroom Risotto", @"Full Breakfast", @"Ha
mburger", @"Ham and Egg Sandwich"];
```

Swift:

```
var recipes = ["Egg Benedict", "Mushroom Risotto", "Full Breakfast", "Hamburger",
"Ham and Egg Sandwich"]
```

While you can put any objects in `NSArray` or `NSMutableArray` in Objective-C, arrays in Swift can only store items of the same type. In the above example, you can only store strings in the string array. With type inference, Swift automatically detects the array type. But if you like, you can also specify the type in the following form:

```
var recipes : String[] = ["Egg Benedict", "Mushroom Risotto", "Full Breakfast", "H
amburger", "Ham and Egg Sandwich"]
```

Swift provides various methods for you to query and manipulate an array. Simply use the count method to find the number of items in the array:

```
var numberOfItems = recipes.count
// recipes.count will return 5
```

Swift makes operations on array much simpler. You can add an item by using the `+=` operator:

```
recipes += ["Thai Shrimp Cake"]
```

This also applies when you need to add multiple items:

```
recipes += ["Creme Brelee", "White Chocolate Donut", "Ham and Cheese Panini"]
```

To access or change a particular item in an array, pass the index of the item by using subscript syntax just like that in Objective-C and other programming languages:

```
var recipeItem = recipes[0]
recipes[1] = "Cupcake"
```

One interesting feature of Swift is that you can use `...` to change a range of values. Here is an example:

```
recipes[1...3] = ["Cheese Cake", "Greek Salad", "Braised Beef Cheeks"]
```

This changes the item 2 to 4 of the recipes array to "Cheese Cake", "Greek Salad" and "Braised Beef Cheeks". (Remember the first item in an array starts with the index 0. This is why index 1 refers to item 2.)

If you print the array to console, here is the result:

- Egg Benedict
- Cheese Cake
- Greek Salad
- Braised Beef Cheeks
- Ham and Egg Sandwich

Dictionaries

Swift provides three primary collection types: *arrays*, *dictionaries*, and *sets*. Now let's talk about dictionaries. Each value in a dictionary is associated with a unique key. To declare a dictionary in Swift, you write the code like this: