

Xcode 12, Swift 5.3 and iOS 14 Ready

Beginning iOS Programming with Swift



SIMON NG

APPCODA

Table of Contents

Preface

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

- The Tools
- The Learning Approach

Chapter 2 - Your First Taste of Swift with Playgrounds

- Trying out Swift in Playgrounds
- Constants and Variables
- Understanding Type Inference
- Control Flow Basics
- Understanding Optionals
- Playing around with UI

Chapter 3 - Hello World! Build Your First App in Swift

- Familiarize Yourself with Xcode Workspace
- A Quick Walkthrough of Interface Builder
- Designing the User Interface
- Connecting the User Interface with Code

Chapter 4 - Hello World App Explained

- Understanding Implementation and Interface
- Relationship Between User Interface and Code
- UIViewController and View Controller Life Cycle

Chapter 5 - Introduction to Auto Layout

- Why Auto Layout?
- Live Previewing in Interface Builder
- Using Auto Layout to Center the Button
- Resolving Layout Constraint Issues
- Safe Areas
- Editing Constraints

Chapter 6 - Designing UI Using Stack Views

- What is a Stack View
- Layout the Title Labels with Stack Views
- Layout the Images Using the Stack Button
- Adapting Stack Views Using Size Classes
- Preserving Vector Data

Chapter 7 - Introduction to Prototyping

- Sketching Your App Ideas on Paper
- Wireframing Your App
- Make Your Sketch/Wireframe Interactive
- Prototyping Your App with App Prototyping Tools

Chapter 8 - Creating a Simple Table-based App

- Design the User Interface
- UITableView and its Protocols
- Connecting the DataSource and Delegate
- Add Thumbnail to Table View
- Hide the Status Bar

Chapter 9 - Customize Table Views Using Prototype Cell, Diffable Data Source and Dark Mode

Building a Table View App Using UITableViewController and UITableViewDiffableDataSource
Customize Table View Cells
Designing Prototype Cells in Interface Builder
Updating the Cell Provider
Rounding Image Corners
Testing the App Using Dark Mode
Chapter 10 - Displaying Alerts with UIAlertController and Handling Table View Selection
Building a more Elegant Cell Layout
Understanding the UITableViewDelegate Protocol
Managing Row Selections by Implementing the Protocol
Understanding UIAlertController
Chapter 11 - Object Oriented Programming, Project Organization and Code Documentation
The Basic Theory of Object Oriented Programming
Classes, Objects and Instances
Structures Explained
Using the Array of Restaurant Objects
Organizing Your Xcode Project Files
Documenting and Organizing Swift Code with MARK
Chapter 12 - Table Row Deletion, Swipe for Actions, Activity Controller and MVC
A Brief Introduction to Model View Controller
Deleting a Row from UITableView
Delete Row Data from the Table View Using Snapshot

Swipe for More Actions Using UIContextualAction
Understanding SF Symbols
Customizing UIContextualAction
Chapter 13 - Introduction to Navigation Controller and Segue
Scenes and Segues in Storyboards
Creating Navigation Controller
Large Navigation Bar Titles
Passing Data Using Segues
Chapter 14 - Detail View Enhancement and Navigation Bar Customization
Using Custom Fonts
Designing the Table View Header
Understanding Image View Scaling
Designing the Prototype Cells
Customizing the Table View Separator
Understanding Self Sizing Cells
Chapter 15 - Navigation Bar Customization, Extensions and Dynamic Type
Customizing the Navigation Bar
Hiding a Navigation Bar on Swipe
Swift Extensions
Adapting Colors for Dark Mode
Dynamic Type
Chapter 16 - Working with Maps
Using MapKit Framework
Adding a Map Interface to Your App
Displaying a Full Screen Map

Converting an Address into Coordinates Using Geocoder
Adding an Annotation to the Map
Customizing the Annotation
Chapter 17 - Basic Animations, Visual Effects and Unwind Segues
Creating a Rate Button
Applying a Blurring Effect to the Background Image
Understanding Outlet Collections
Animating Dialog View Using UIView Animation
Unwind Segues and Data Passing
Chapter 18 - Working with Static Table Views, Camera and NSLayoutConstraint
Designing the New Restaurant View Controller
Building a Rounded Corner Text Field
Customizing the Navigation Bar
Displaying the Photo Library Using UIImagePickerController
Adopting the UIImagePickerControllerDelegate Protocol
Defining Auto Layout Constraints Programmatically
Chapter 19 - Working with Core Data
What is Core Data?
Using Core Data Template
Using the Managed Object
Fetching Data Using Core Data
Deleting Data Using Core Data
Updating a Managed Object
Chapter 20 - Search Bar and UISearchController

Using UISearchController
Using Predicate for Searching the Results
Search Bar in the Table Header View
Customizing the Appearance of the Search Bar
Chapter 21 - Building Walkthrough Screens with UIPageViewController and Container Views
A Quick Look at the Walkthrough Screens
Understanding Page View Controllers and Container Views
Handling the Page Indicator and the Next/Skip Buttons
Updating the Page Indicator for Gesture-based Navigation
Working with UserDefaults
Chapter 22 - Exploring Tab Bar Controller and Storyboard References
Building a Tab Bar Controller
Hide Tab Bar When Pushed
Adding New Tabs
Customizing the Appearance of the Tab Bar
Storyboard References
Chapter 23 - Getting Started with WKWebView and SFSafariViewController
Designing the About View
Opening Web Content in Mobile Safari
Loading Web Content Using WKWebView
Loading Web Content Using SFSafariViewController
Chapter 24 - Exploring CloudKit
Understanding CloudKit Framework
Managing Your Records in CloudKit Dashboard

Fetching Data from Public Database Using Operational API
Optimizing the Performance
Lazy Loading Images
Caching Images Using NSCache
Saving Data Using CloudKit
Chapter 25 - Localizing Your App to Reach More Users
Internationalizing Your App
Adding Your Supported Languages
Export for Localization
Import Localizations
Testing the Localized App
Chapter 26 - Deploying and Testing Your App on a Real iOS Device
Understanding Code Signing and Provisioning Profiles
Automatic Signing in Xcode
Deploying the App on Your Device via USB
Deploying the App over Wi-Fi
Chapter 27 - Beta Testing with TestFlight and CloudKit Production Deployment
Creating an App Record on App Store Connect
Archiving and Validating Your App
Upload Your App to App Store Connect
Manage Internal Testing
Manage Beta Testing with External Users
CloudKit Production Deployment
Chapter 28 - Submit Your App to App Store

Get Prepared and Well-Tested
Submit Your App to App Store
Chapter 29 - Adopting 3D Touch, Haptic Touch and Context Menus
Home Screen Quick Actions
Context Menu and Preview
Chapter 30 - Developing User Notifications in iOS
Using User Notifications to Increase Customer Engagement
Using the User Notifications Framework
Creating and Scheduling Notifications
Interacting with User Notifications
Appendix - Swift Basics

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

APPCODA



Copyright ©2020 by AppCoda Limited

All right reserved. No part of this book may be used or reproduced, stored or transmitted in any manner whatsoever without written permission from the publisher.

Published by AppCoda Limited

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

"The Basic and Intermediate books are everything you need to understand all the concepts of design, coding, testing, and publishing an app. All you need is the idea."

- Rich Gabrielli

"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

"I probably could have done it without the book but it would've taken WAY longer and I probably wouldn't have understood what I was doing so well. If it weren't for this book my app probably won't be in the App Store right now. True story. Keep it up!"

- Marc Estwick

Preface

A couple years ago, people still asked, "Is Swift ready for app development?"

Now everyone knows Swift is the programming language to learn for developing iOS apps.

I really love this programming language, not because I teach Swift programming for a living and want you to buy my courses/books. I have been programming for over 15 years in various programming languages. Swift is my favorite language right now. This new programming language is well designed with a simplified and clean syntax. Swift is easier to learn and makes me more productive in iOS app development, as compared to the good old Objective-C.

Swift was first introduced by Apple in June 2014 and has gone through several updates. Fast forward to today, the company already released version 5.3 of the programming language along with Xcode 12 with more features. With over 6 years of development since its first release, Swift is no longer a brand new programming language. The language is now solidified, more mature and ready for any application development on iOS, macOS, watchOS, as well as, tvOS. Companies like Lyft, LinkedIn, and Mozilla have been developing their apps in Swift. Whether you are planning to develop your next iOS app or you want to build a career in iOS app development, Swift is for sure the programming language to learn and use.

This book covers everything you need to learn for iOS app development. Remember, Swift is just a programming language. To develop an iOS app, you need to learn far more than that. On top of the introduction to Swift, this book will teach you how to lay out the user interface using Xcode and master the essential APIs of the iOS 14 SDK. Most importantly, you will acquire the Swift programming skill by developing a real-world app from the ground up.

As an absolute beginner with no prior programming background, you may question if you could learn Swift programming and build a real iOS app.

I have been programming in Swift since it was first announced. Swift is more approachable and it is a lot easier for newbies to learn than Objective-C. Not everyone can become a great developer, but I believe everyone can learn programming and develop an app in Swift. All you need is hard work, determination, and the willingness to take actions.

I launched AppCoda about 8 years ago and started to publish iOS programming tutorials on a weekly basis. Since then, I have published several books on iOS app development. At first, I thought people, who want to learn app development, are those with programming experience and technical background. What's interesting is that people from different backgrounds are passionate to build their own apps. I have a reader from France, who is a surgeon by profession, started from zero programming experience to launching his first app, which allows anyone to share and advertise event information for free. Another reader is a pilot by profession. He began to learn iOS programming a couple years ago and is now building iPhone apps for his own use and other pilots. Boozy is an app for finding Happy Hours, Daily Deals and Brunches. It was built by a law school dropout. The creator of the app could not find a good place for a drink in DC area. So she decided to make an app to meet a real need. Similarly, she did not know coding when she came up with the idea. She just got started and learned along the way.

From time to time, I got emails from people who want to create an app. The emails usually mention something like this: "I have an app idea. Where do I begin? But I have no programming skills. Can I learn from scratch to make one?"

What I learned from these truly inspiring stories is that you don't need to have a degree in Computer Science or Engineering in order to build an app. These readers have one thing in common. They are committed to taking actions. They all put their hard work in to make things happen. This is what you need.

So you've got an idea to build an app? I believe you can make one on your own. Remember there is nothing to deter you from learning and get things done if you're really passionate about it. Let me borrow one of my favorite quotes from *Last Lecture* to conclude:

> Brick walls are there for a reason: they let us prove how badly we want things.
>
> - Randy Pausch

Lastly, thanks for picking up the book. I hope you will enjoy reading it and launch your first iOS app on App Store. If you'd like to share the story of your first app, drop me an email at simonng@appcoda.com. I would love to hear from you.

Simon Ng
Founder of AppCoda

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 5.3, Xcode 12, and iOS 14 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.

SwiftUI vs UIKit

Since Apple released a new UI framework called SwiftUI in late 2019, one common question is whether you should learn to use SwiftUI or UIKit (Storyboard) to build user interface.

My short answer is to learn both, especially you want to become a professional iOS developer and look for a job. Most of the apps on the App Store are developed using UIKit. It is very likely you will work on an app built with UIKit if you are employed. Even if you do not plan to look for a job, SwiftUI doesn't come with all types of UI components. In some cases, you may need to utilize the components provided by UIKit. This is why I suggest you to learn both frameworks during the transition period.

This book focuses on teaching you to use UIKit and Storyboard for creating user interface. If you want to learn the SwiftUI framework, you may refer to our Mastering SwiftUI book.

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 10.15.4 (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$799. You can hook it up to the monitor of your PC. The basic model of Mac mini comes with a 3.6GHz quad-core Intel Core i3 processor and 8GB 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.

To install Xcode, go up to the Mac App Store and download it. 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.

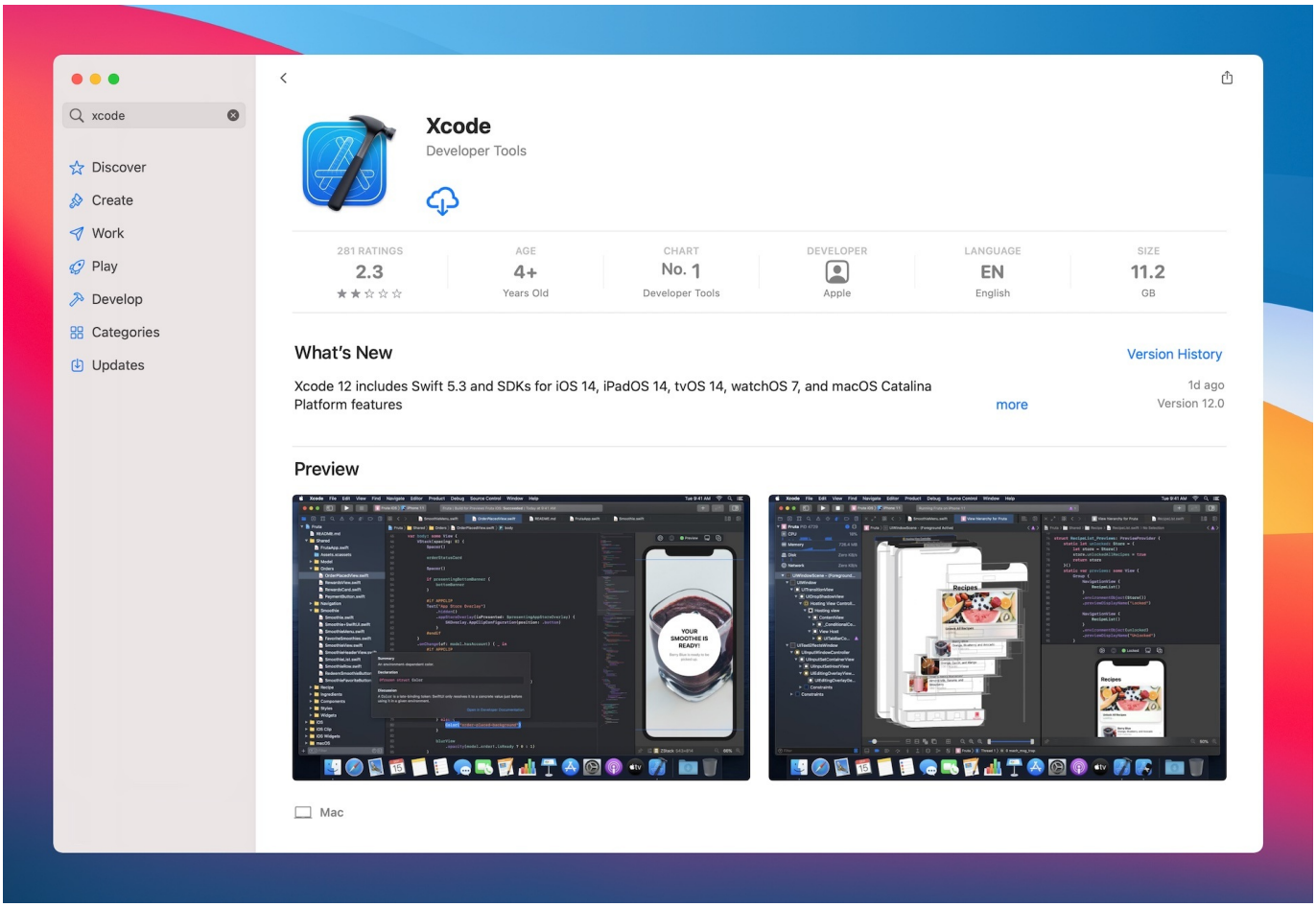


Figure 1-2. Download Xcode 11

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

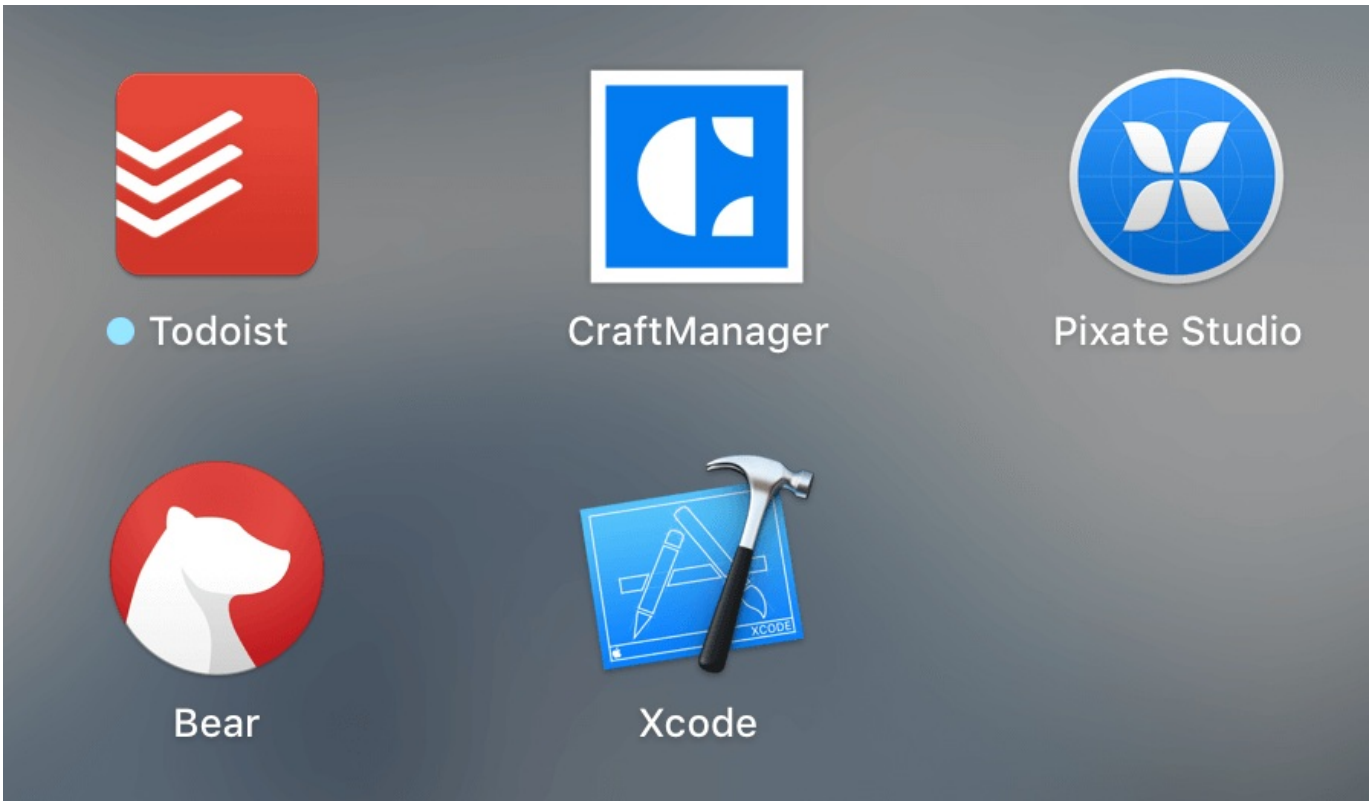


Figure 1-3. Xcode icon in the Launchpad

At the time of this writing, the latest version of Xcode is 12. 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.

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 February 2019, however, there were over 2.2 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. The 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.

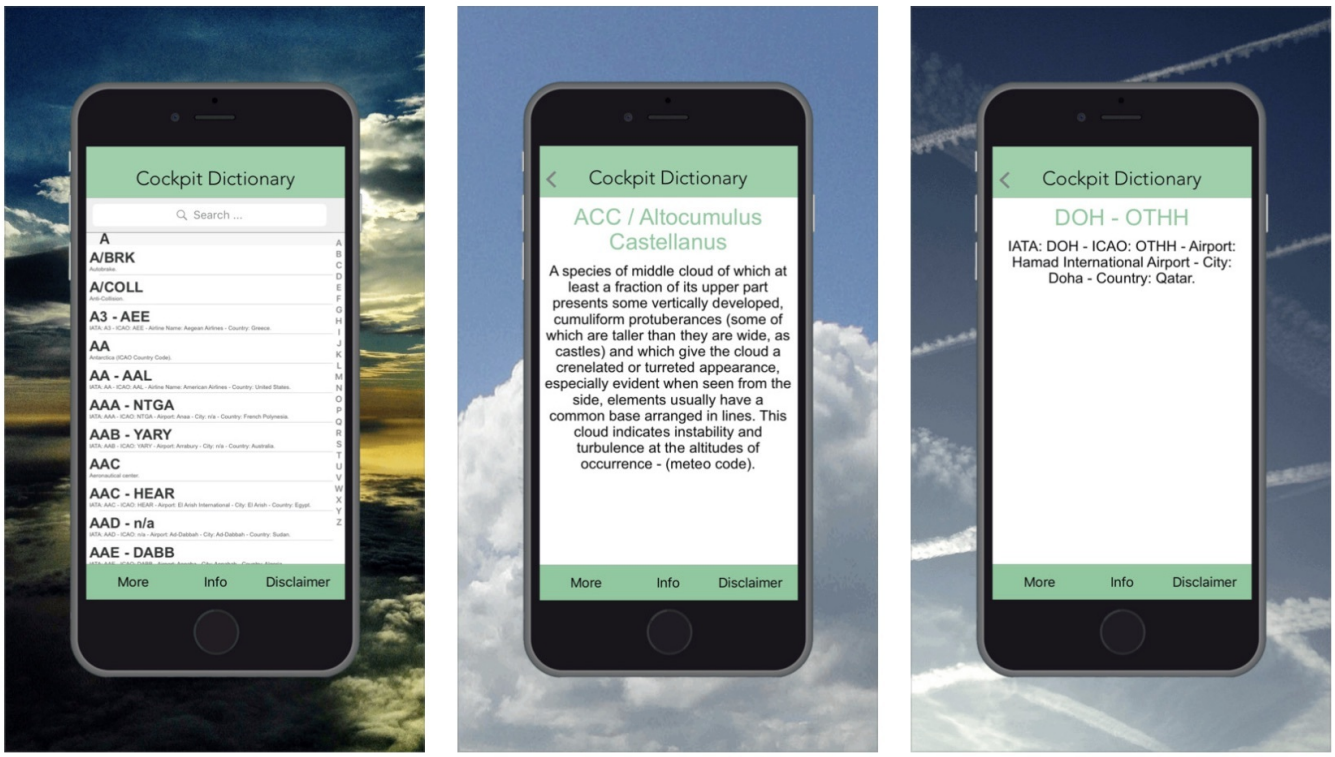


Figure 1-4. Cockpit Dictionary

Another example is the NOAA Buoy Data app. While this app is no longer available on the App Store, I still want to quote it as an example. The app was designed to retrieve 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.

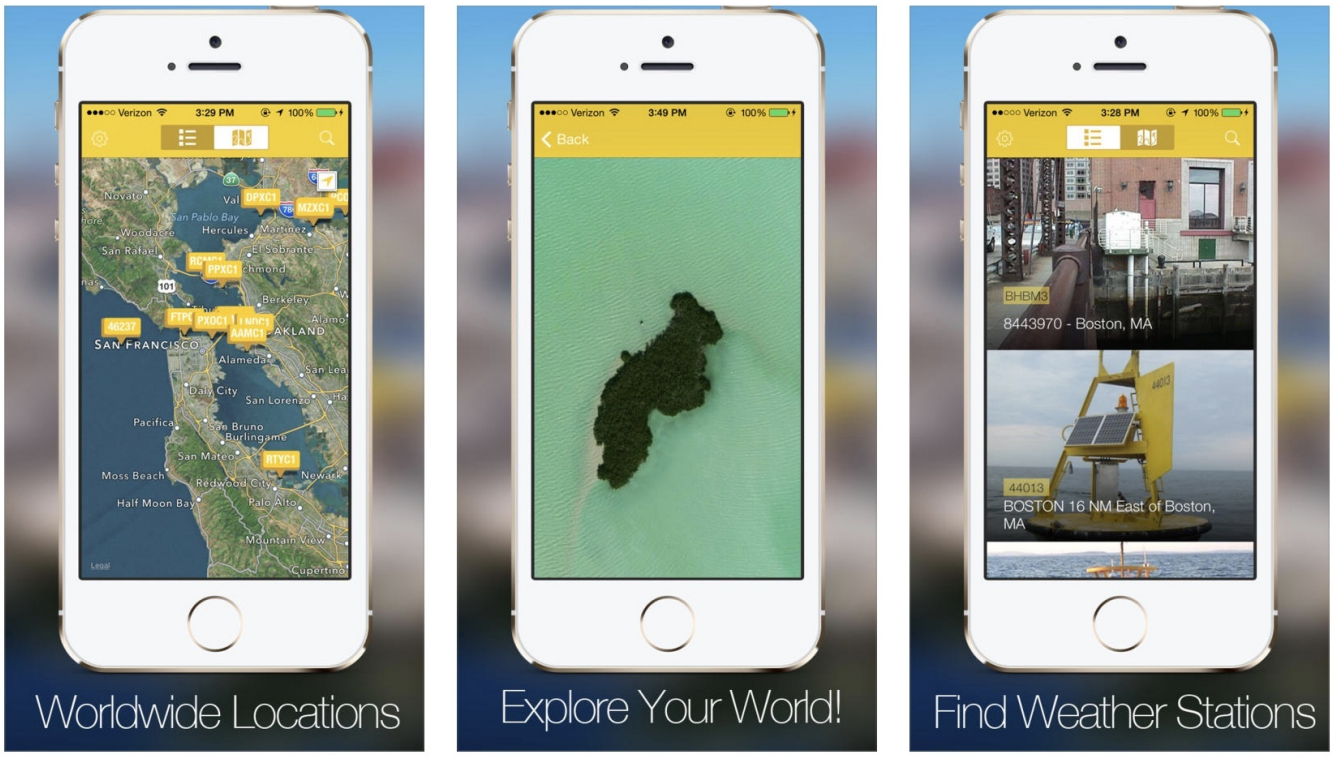


Figure 1-5. NOAA Buoy Data app

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.

In late 2018, Apple only released a minor update for Swift along with Xcode 10, pushing Swift's version number to 4.2. Even though it's not a major release, the new version also came with a lot of language features to improve productivity and

efficiency.

In late March 2019, Apple officially released Swift 5, which is a major milestone of the programming language. While it included many new features, the most important change was that the Swift runtime was included in Apple's platform operating systems including iOS, macOS, watchOS, and tvOS. Actually, it was a big news for aspiring developers. This meant the Swift language became more stable and mature.

This year, along with the release of Xcode 12, Apple released Swift 5.3 with more powerful features, allowing you to write better and clearer code.

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, seco
ndMessage];

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, seco
ndMessage];
```

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 12 (or up), launch the application (by clicking the Xcode icon in Launchpad). You should see a startup dialog.

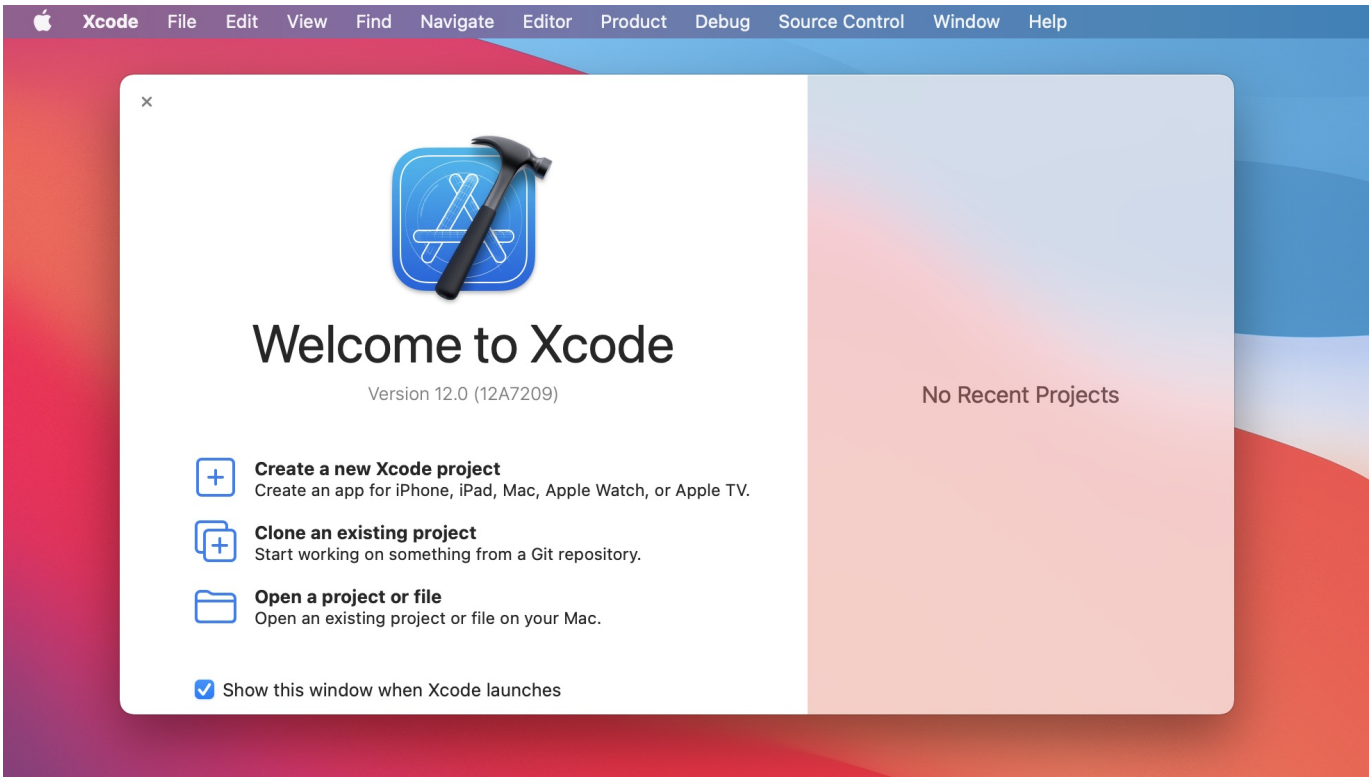


Figure 2-1. The startup dialog

A Playground is a special type of Xcode file. To create a Playground file, go up to the Xcode menu and choose File > New > 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.

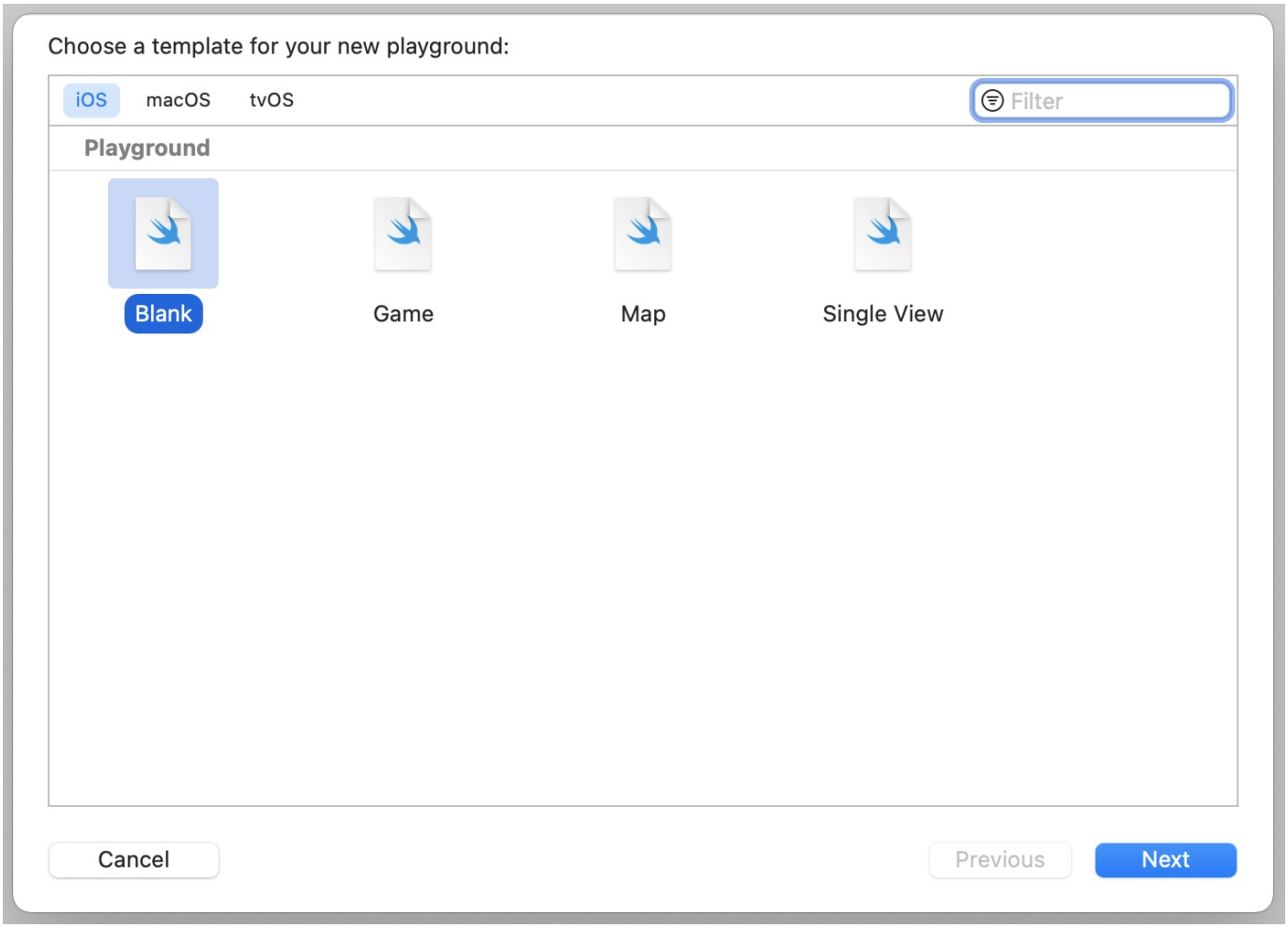
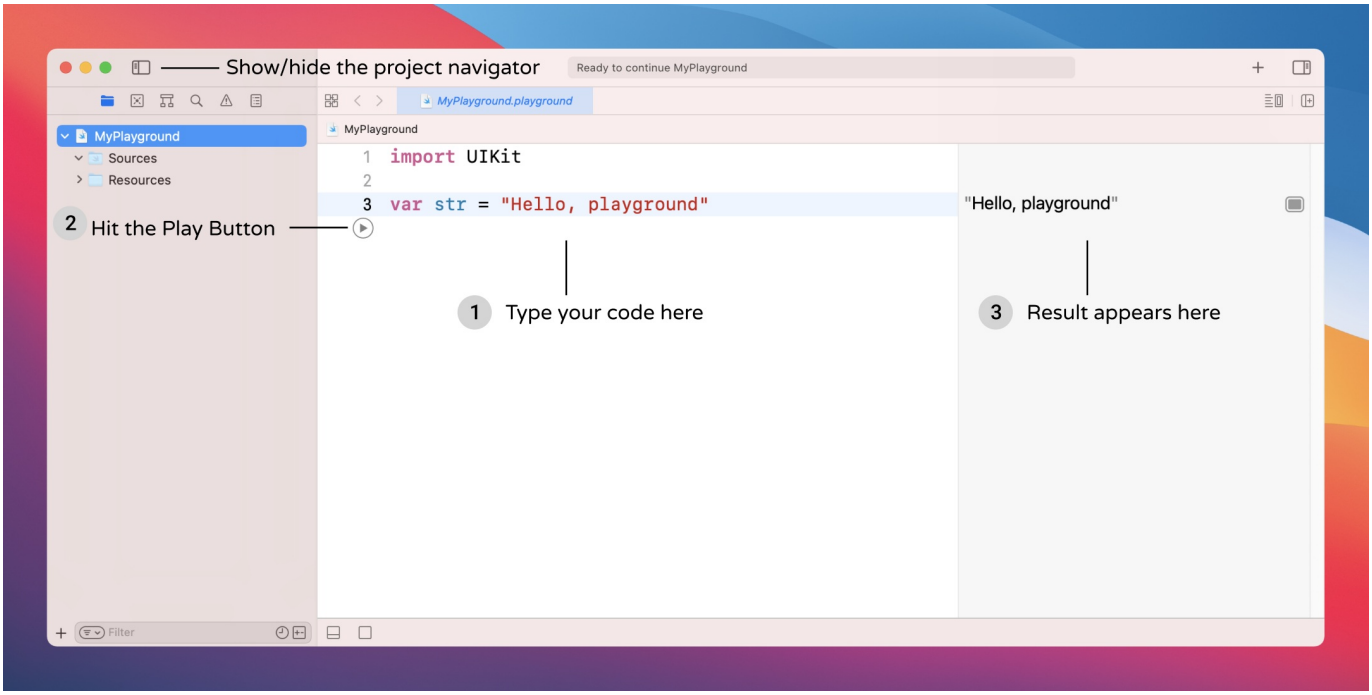


Figure 2-2. Creating a Playground file

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`.

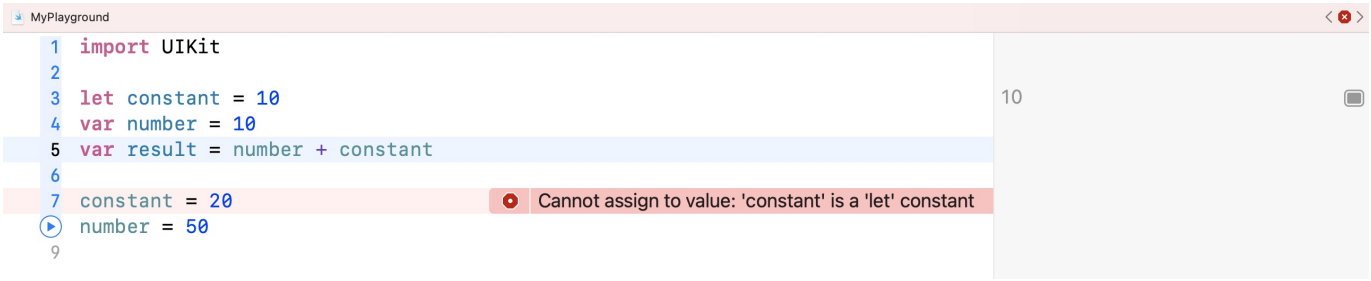


Figure 2-5. Errors in Playgrounds

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
```

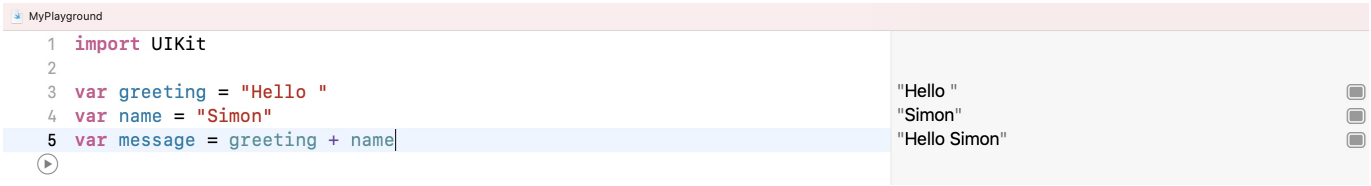
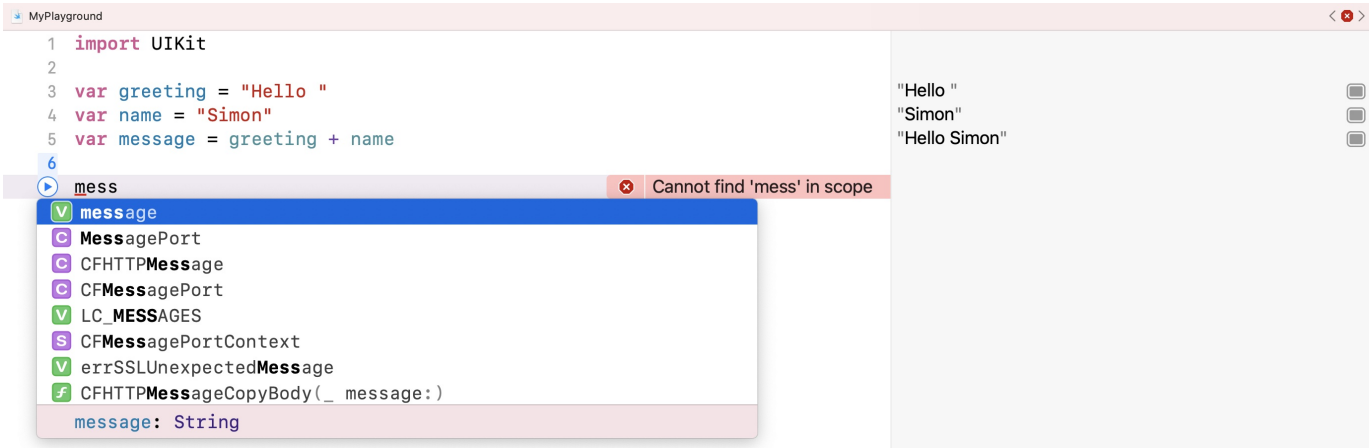


Figure 2-8. String concatenation

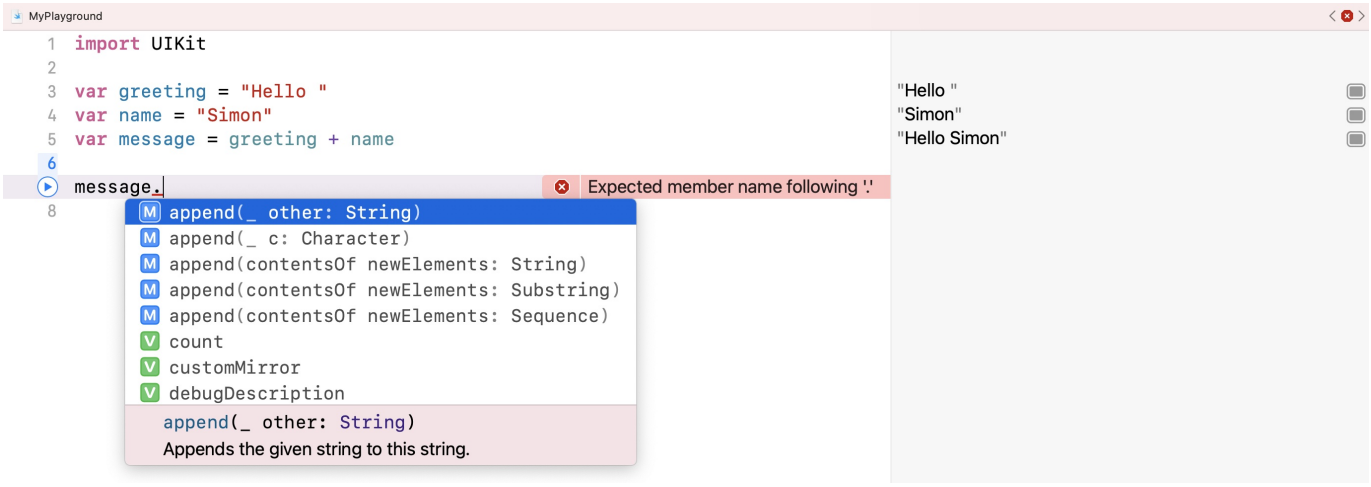
What if you want to convert the whole sentence into upper case? Swift provides a built-in method named `uppercased()` to convert a string to upper case. You can type the following code to have a try:

```
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
```

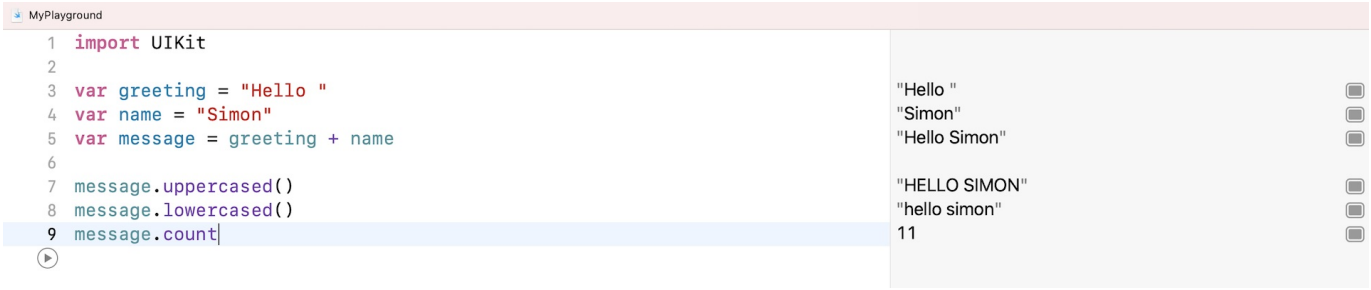


Figure 2-11. Manipulating a string using the built-in functions

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.

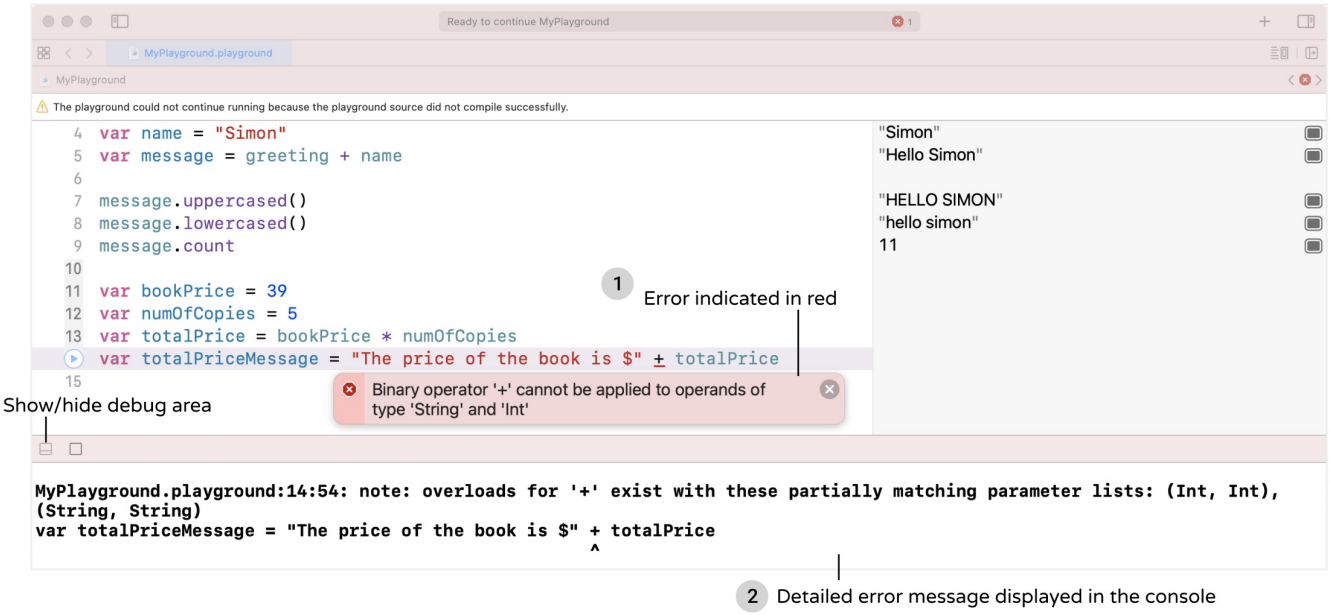


Figure 2-12. Debug area/Console

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

In addition to the error indicator, you should see the debug area/console which displays the detailed error message. If the console doesn't show up in your Playground, click the *show/hide debug area* button at the bottom-left corner. Alternatively, you can go up to the Xcode menu and choose *View* > *Debug Area* > *Activate Console*.

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.

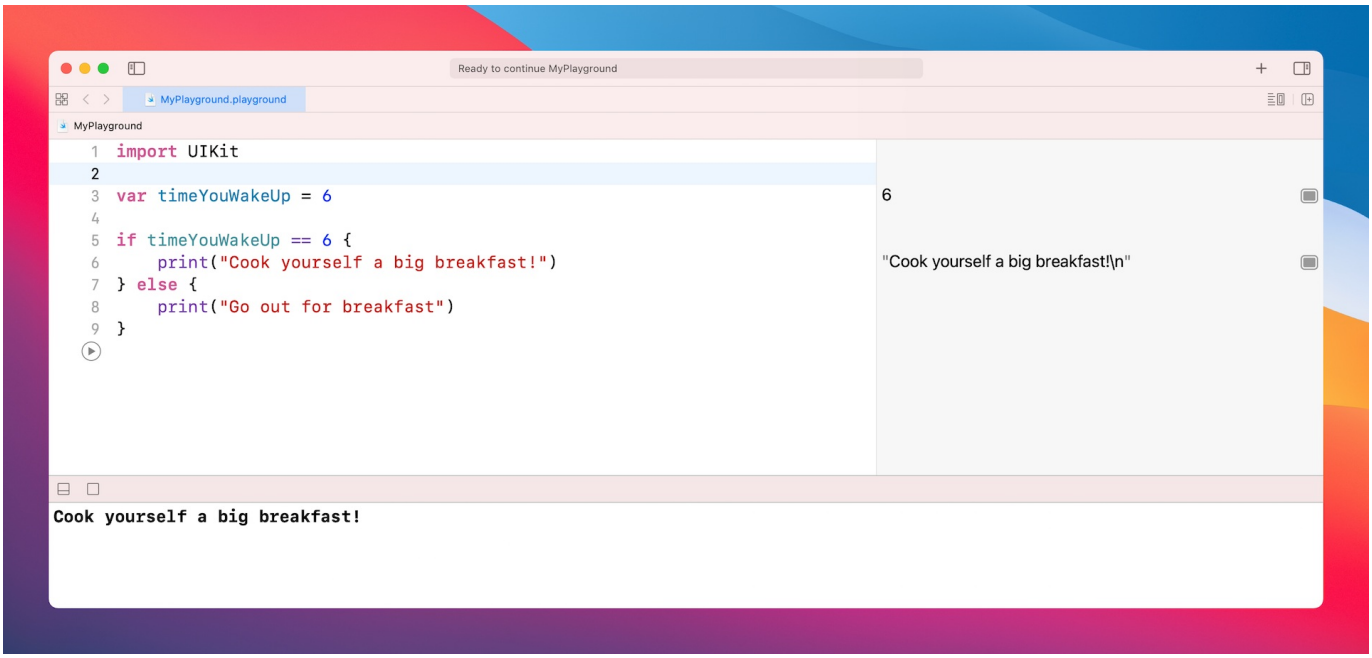


Figure 2-13. An example of If statement

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": "R
ework", "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)")
}
```

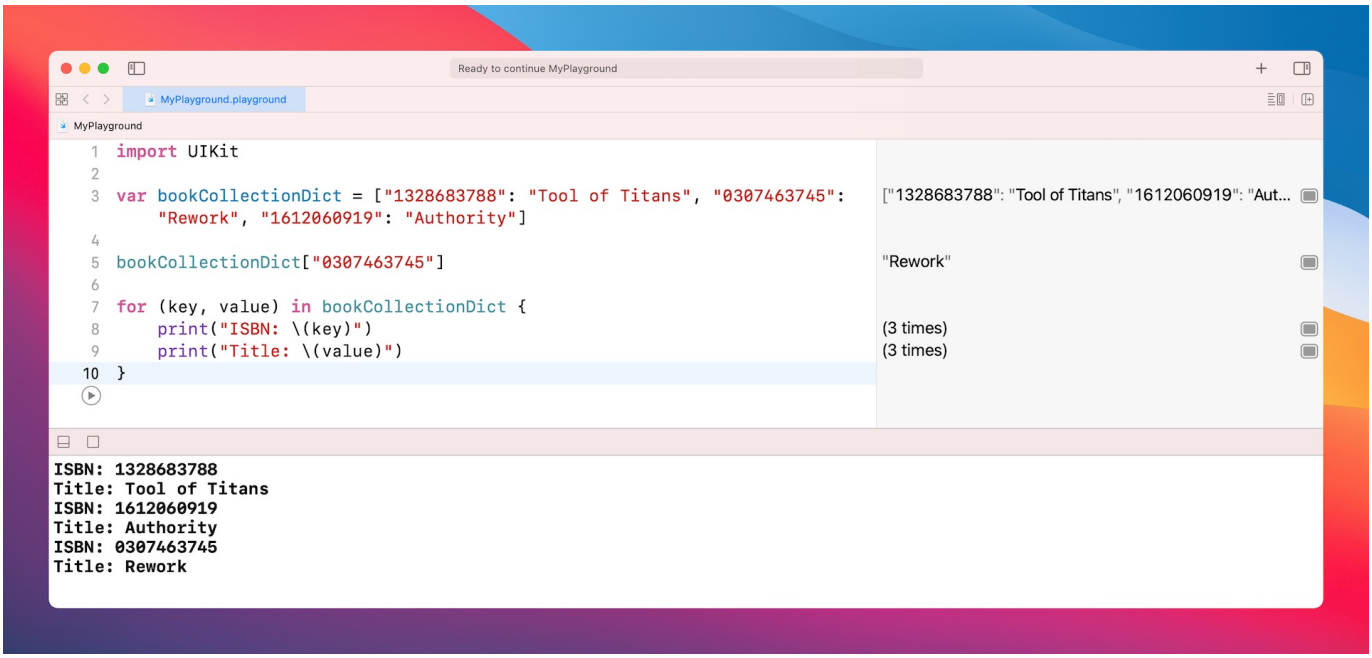


Figure 2-14. Iterate over a dictionary

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.

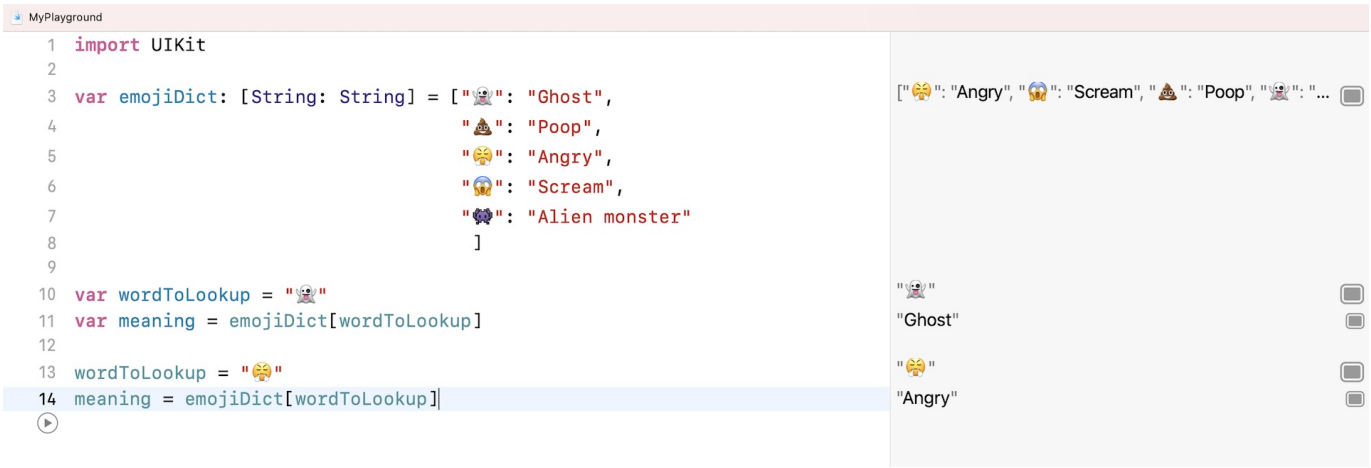


Figure 2-16. Solution to the emoji dictionary exercise

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.

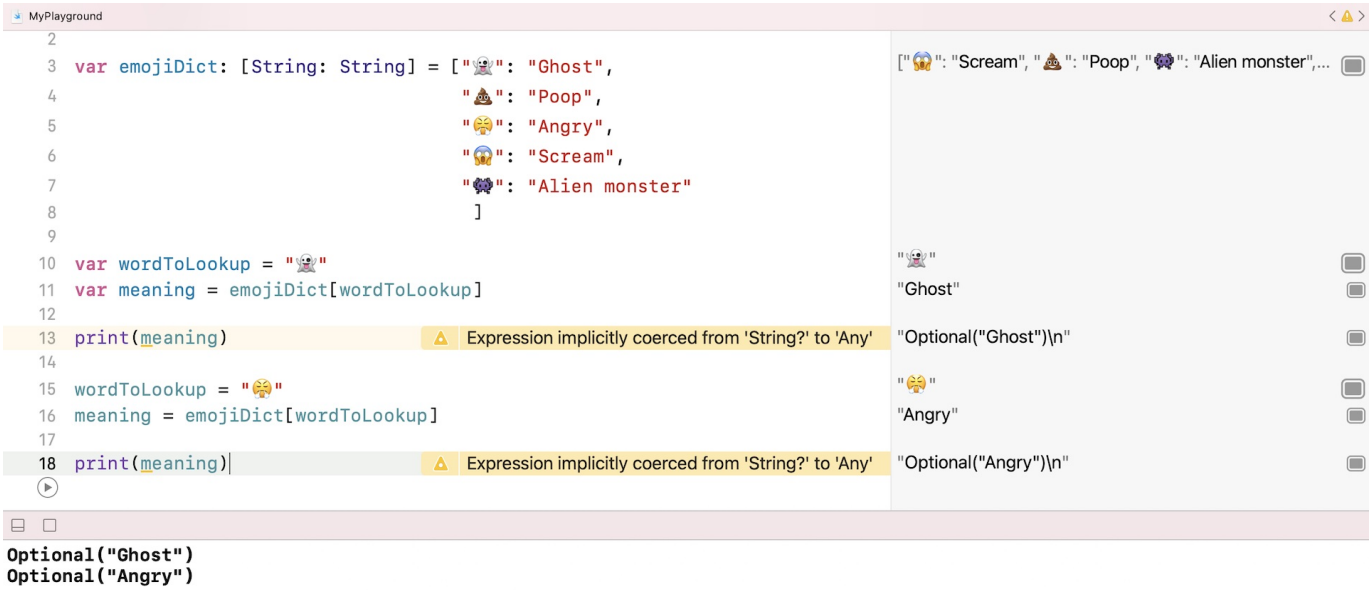


Figure 2-16. Print the meaning variable

You will notice two things:

1. Xcode indicates that 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? 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.

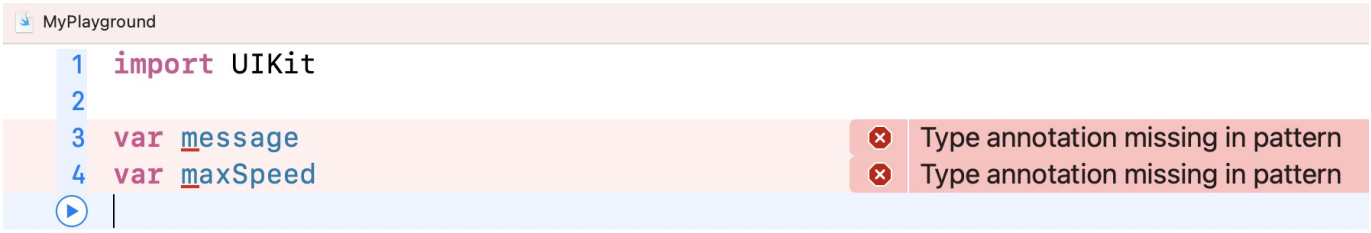


Figure 2-17. Declaring a variable/constant without an initial value

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.

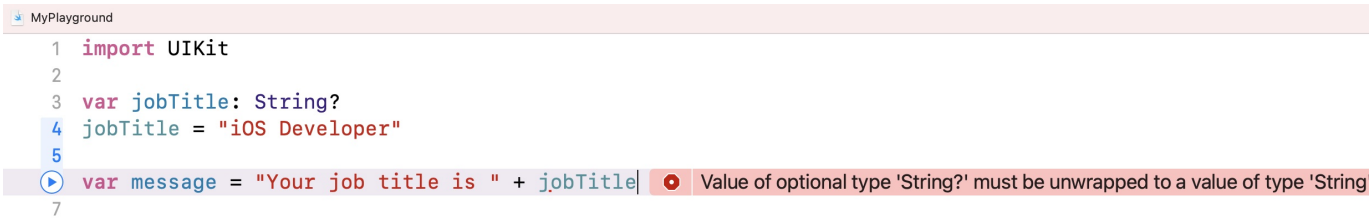


Figure 2-19. An error is shown when you access an optional variable

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".

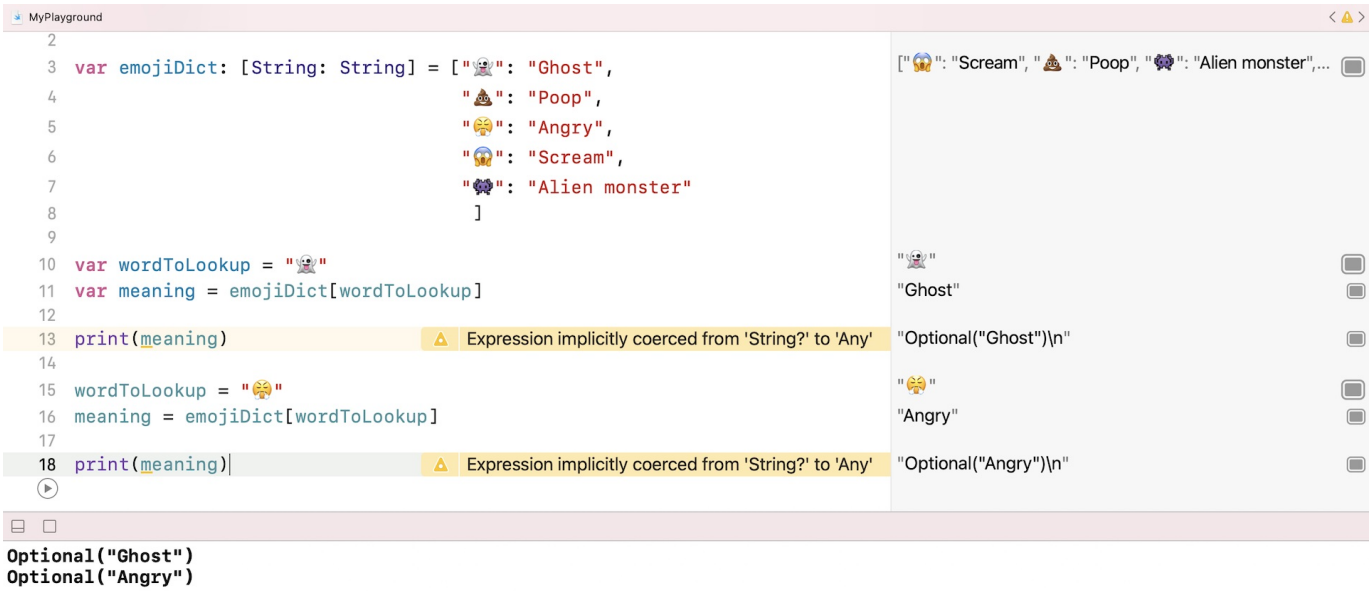


Figure 2-20. Same as figure 2-16 showing you the warning messages

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.

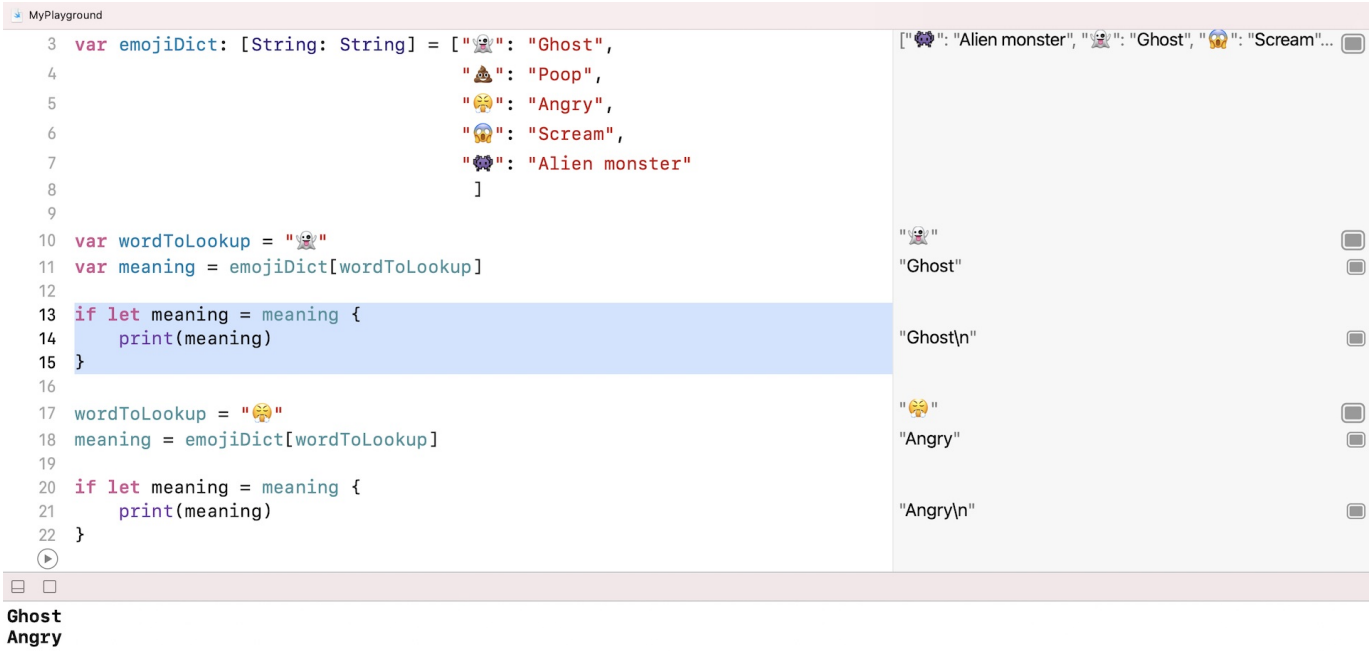


Figure 2-21. Use optional binding to check if meaning has a value and unwrap it accordingly

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: [String: String] = ["👻": "Ghost",
                                   "💩": "Poop",
                                   "😤": "Angry",
                                   "😱": "Scream",
                                   "👾": "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: 1
50))
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, heigh
t: 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

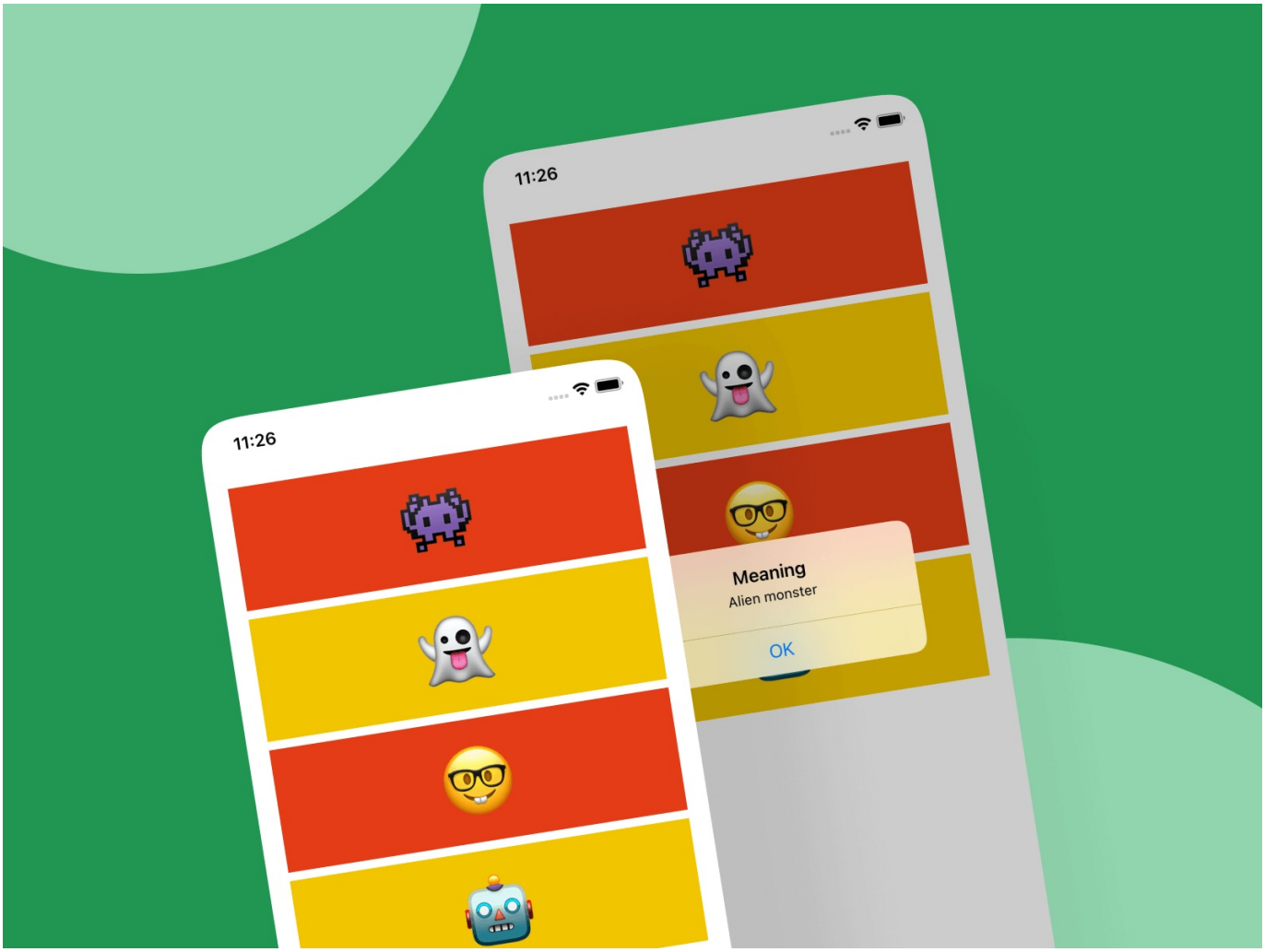
What's coming up is that I will teach to build your first app. You can now move onto the next chapter. However, if you want to learn more about the Swift programming language, I would recommend you to check out Apple's official Swift Programming Language guide (https://docs.swift.org/swift-book/). You 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/swift53/ch2-playgrounds.zip.

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 12 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.

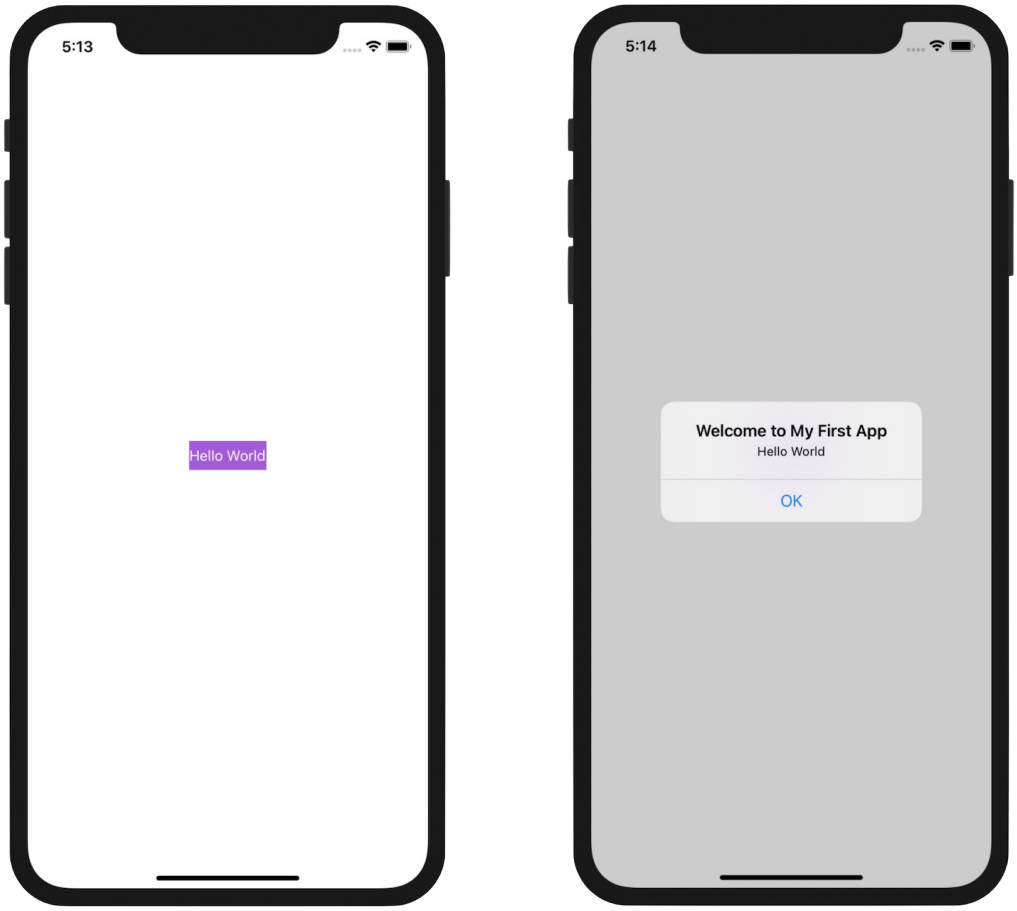


Figure 3-1. HelloWorld App

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.

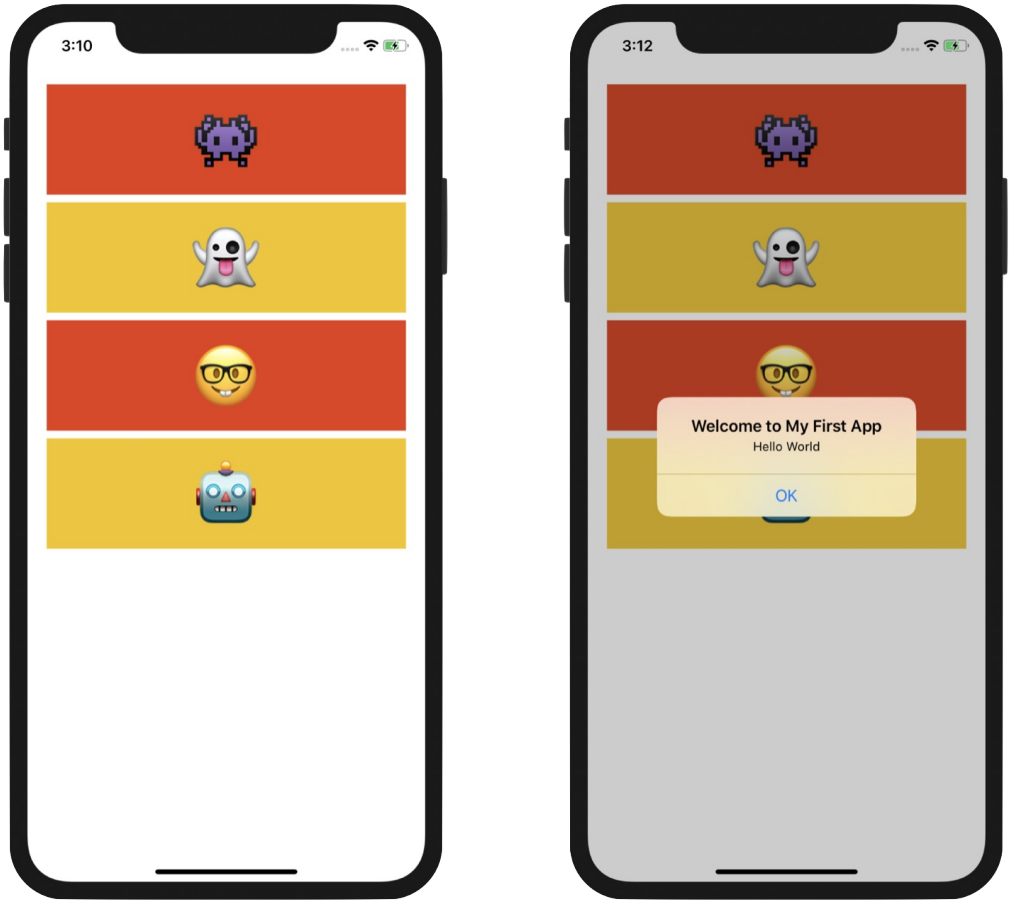


Figure 3-2. Your final deliverable

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.

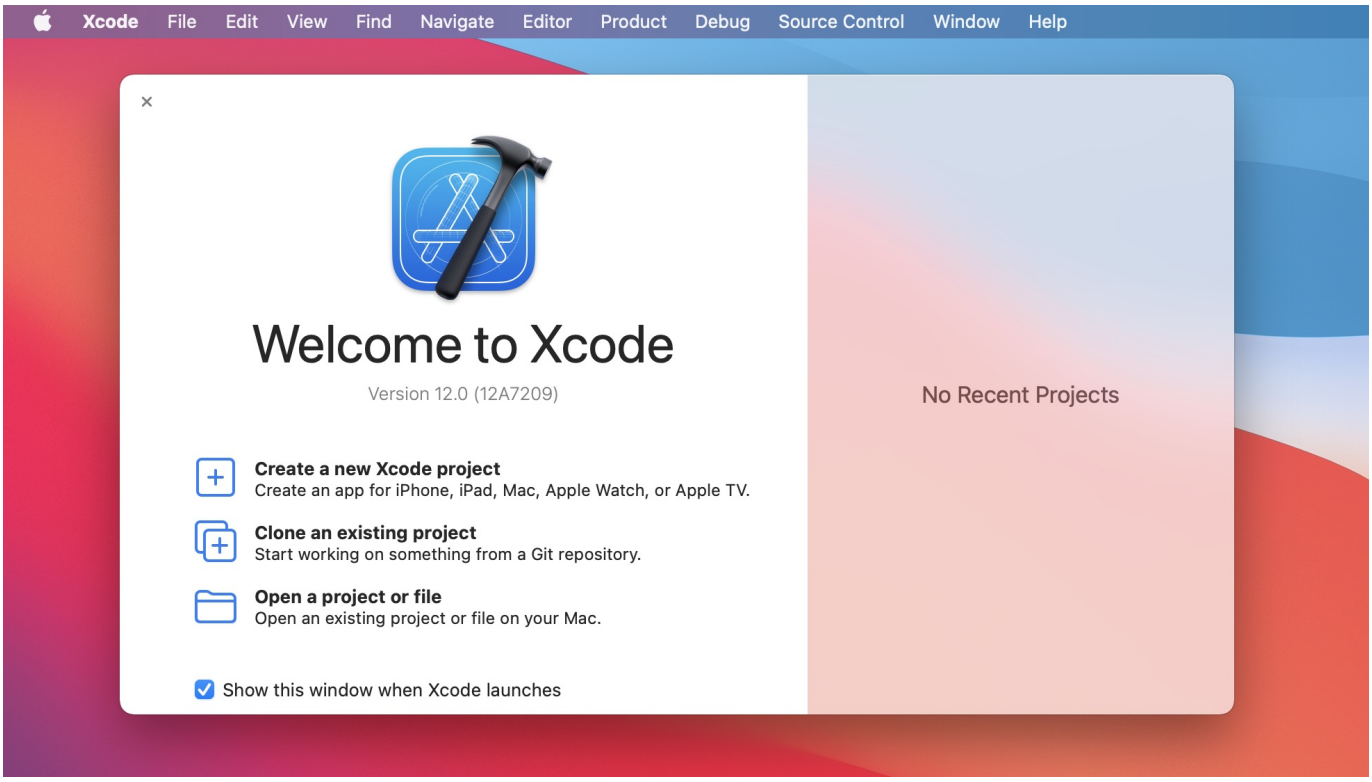


Figure 3-3. Xcode - Welcome Dialog

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 *App* template is good enough for creating an iOS app. So, choose *iOS* > *App* and click *Next*.

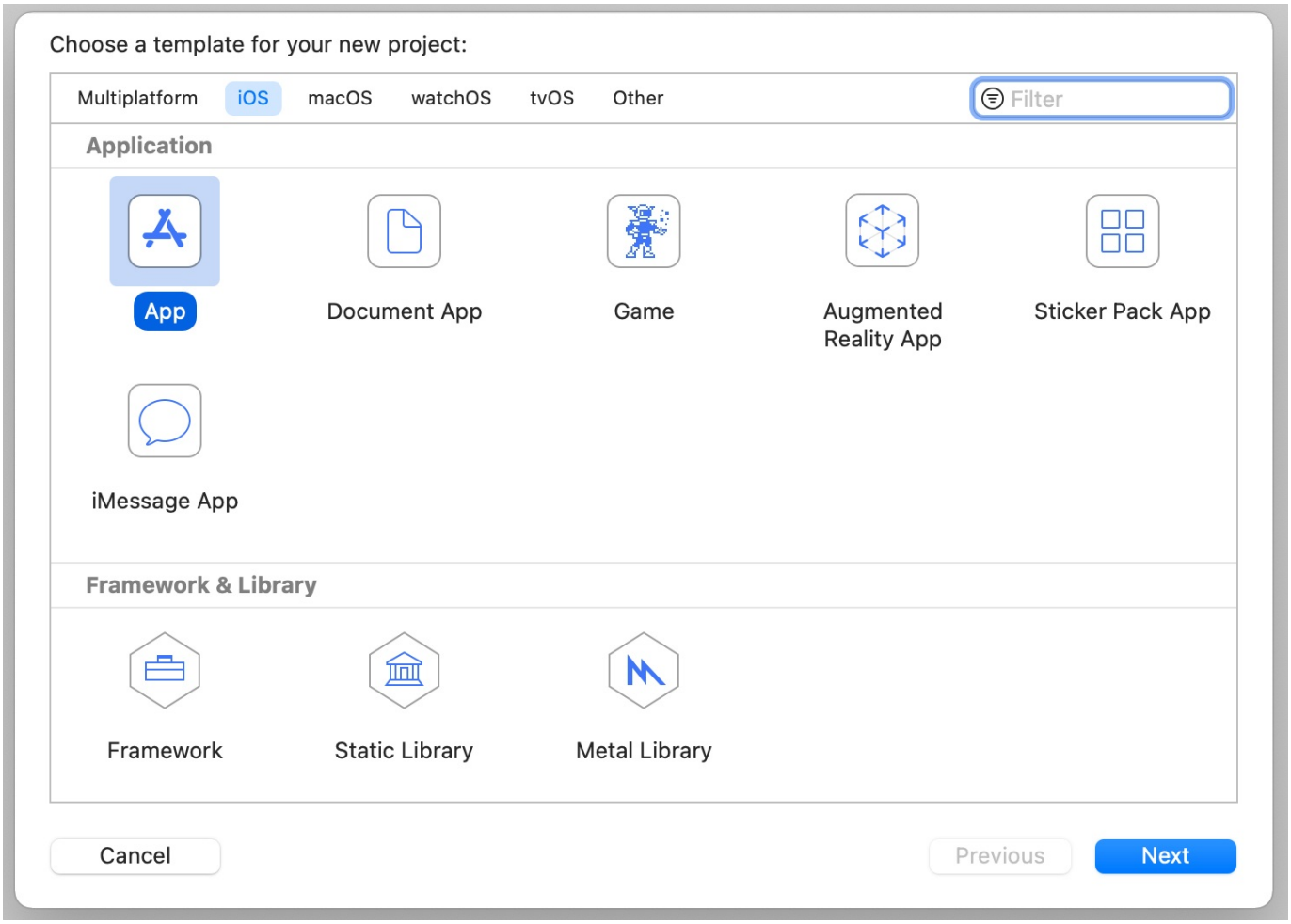


Figure 3-4. Xcode Project Template Selection

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

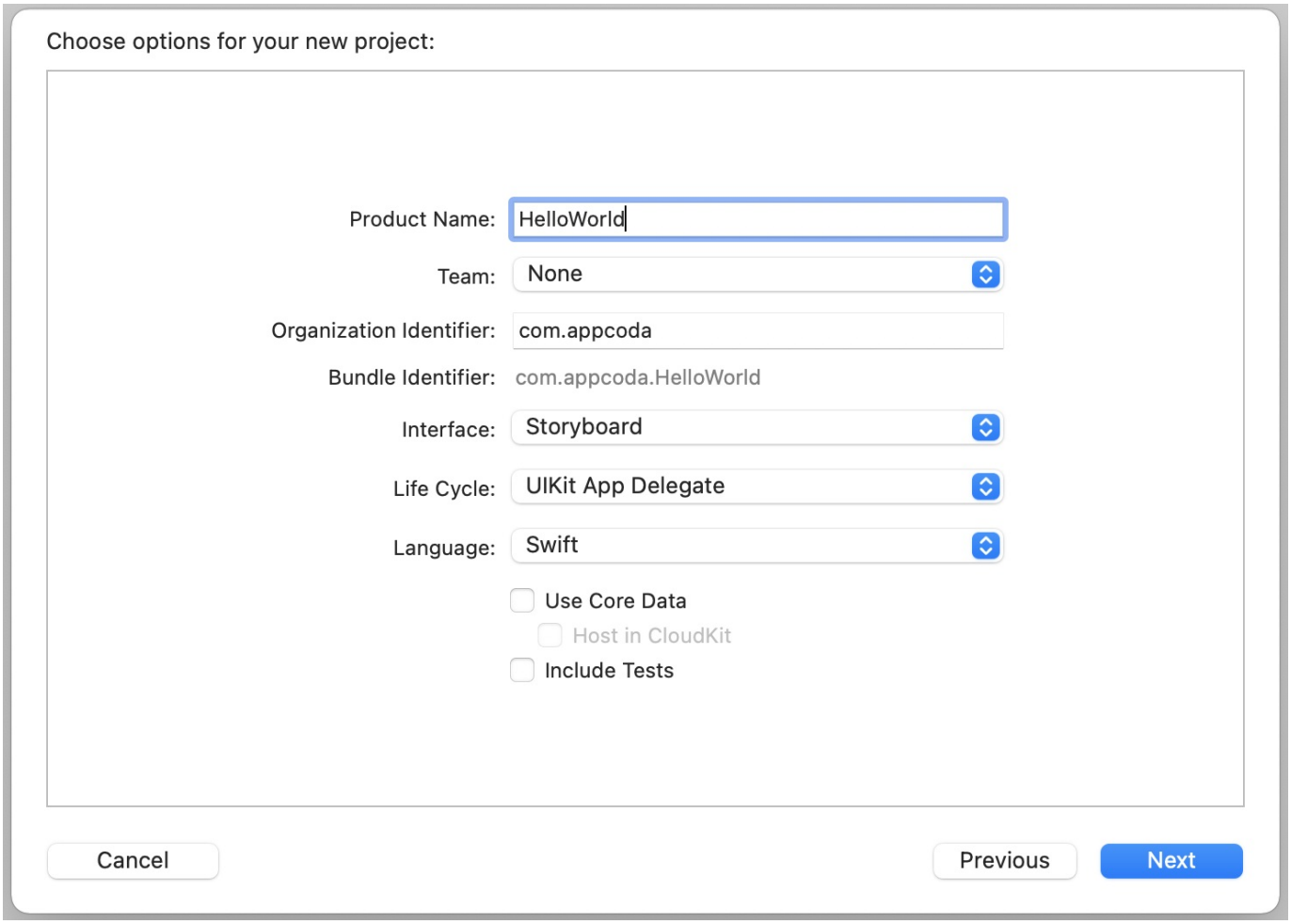


Figure 3-5. Options for your Hello World project

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 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.

- **Interface: Storyboard** - Xcode now supports two ways to build UI. Please change the option to `Storyboard`. In this book, we use Storyboard. If you want to learn SwiftUI, you can refer to our new book (https://www.appcoda.com/swiftui).
- **Life Cycle: UIKit App Delegate** - This is the default life cycle option for UIKit development.
- **Language: Swift** – Xcode supports both Objective-C and Swift for app development. As this book is about Swift, we 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 Tests: [unchecked]** – Do not select this option. You do not need any 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.

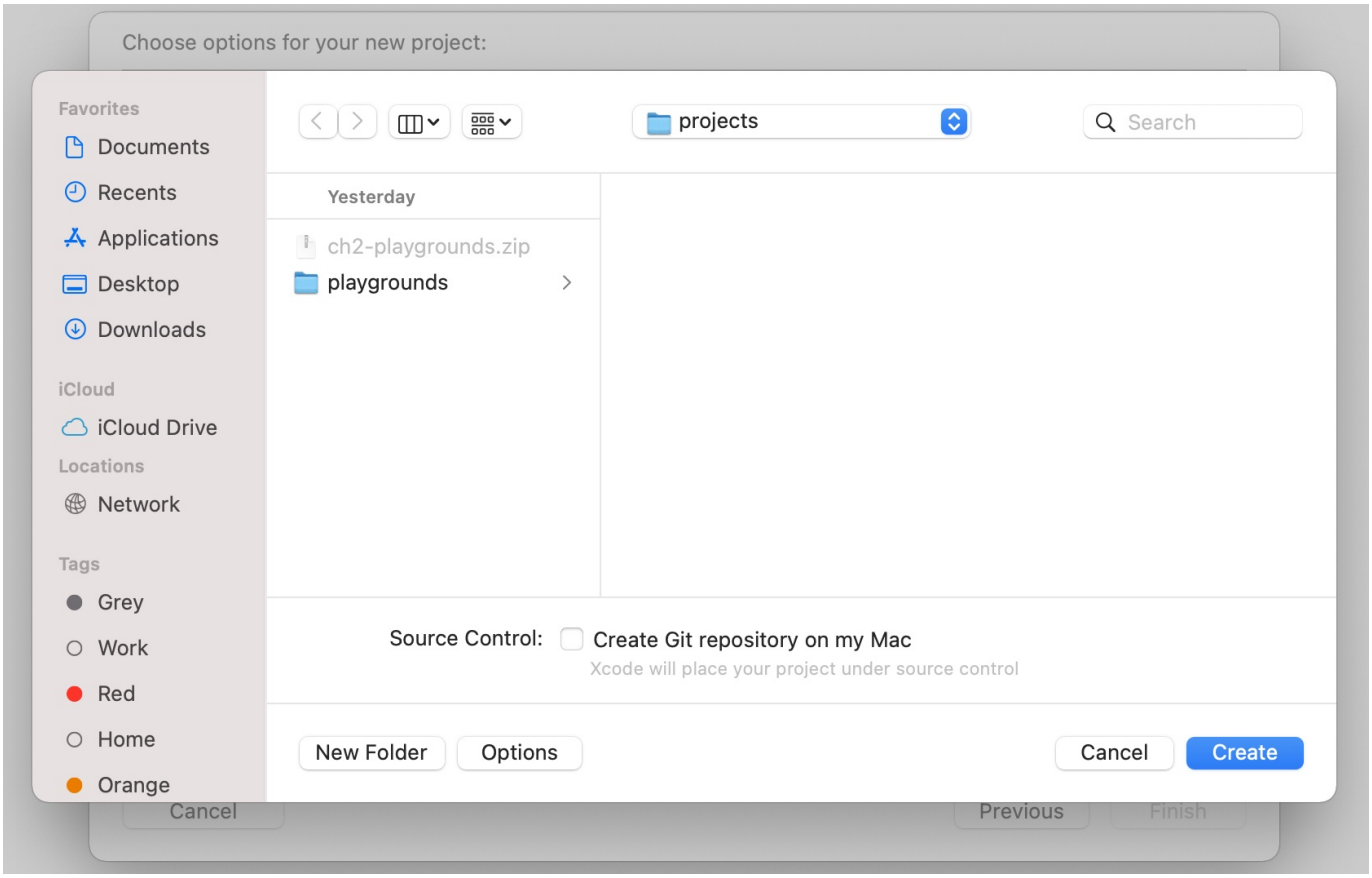
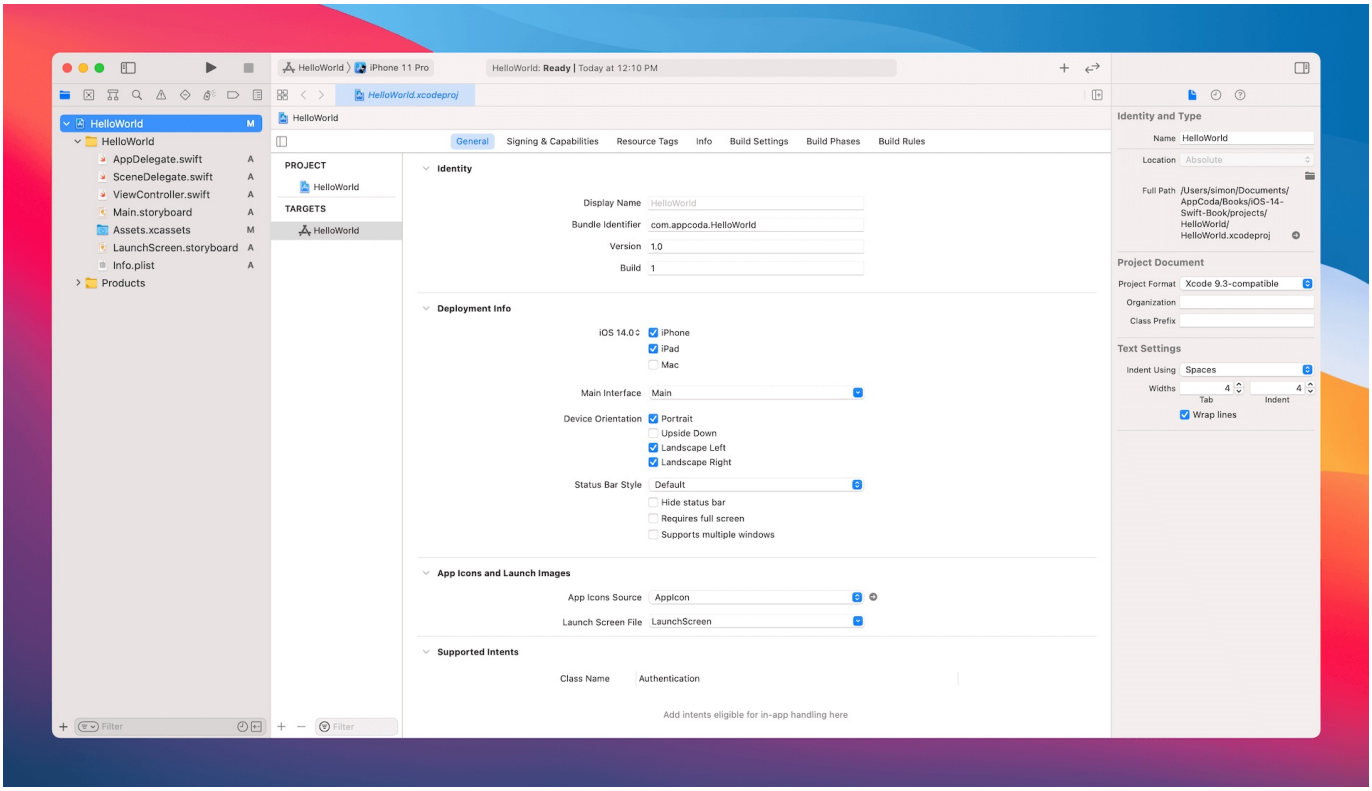


Figure 3-6. Choose a folder and save your project

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). Xcode comes with several themes for you to choose from. Say, if you prefer dark themes, you can go up to the menu and choose *Xcode* > *Preferences* > *Themes* to change it.

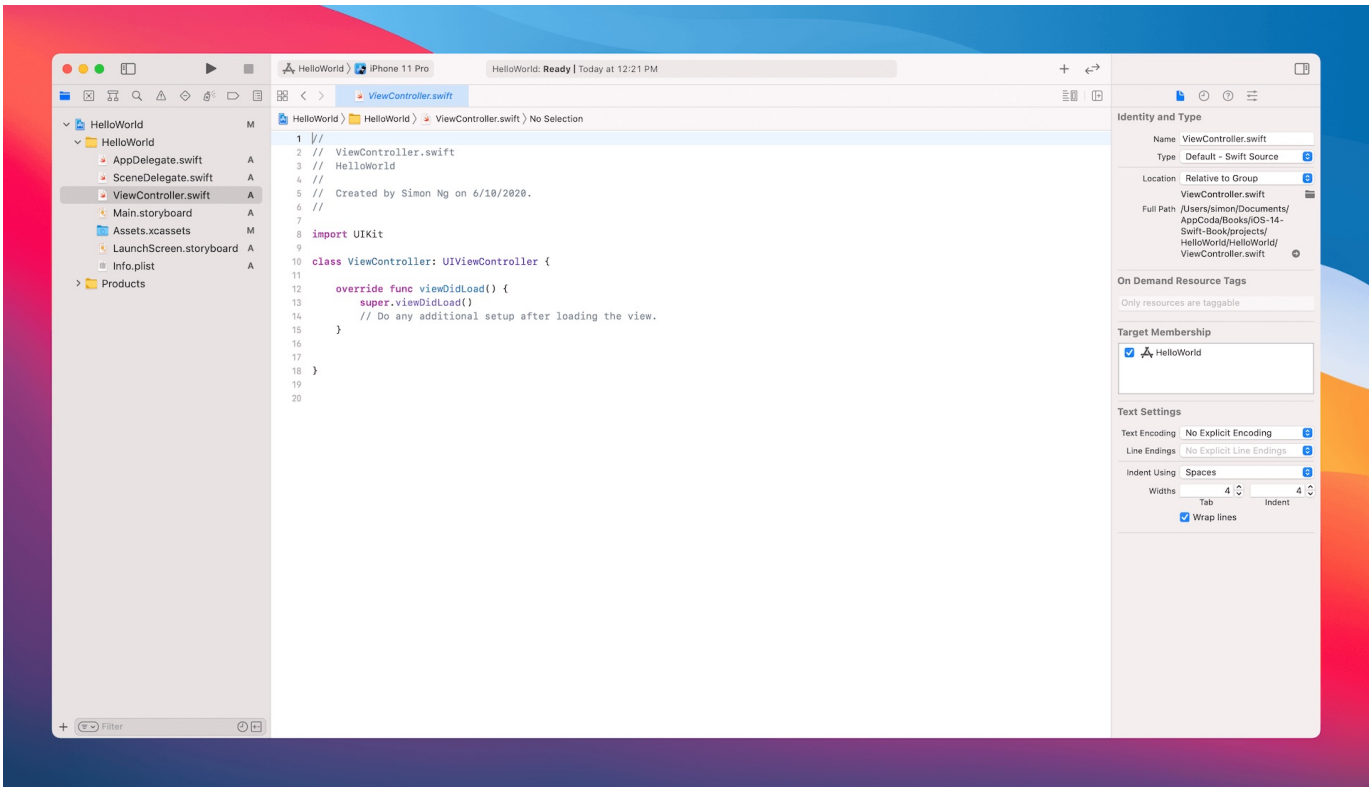


Figure 3-8. Xcode Workspace with Source Code Editor

If you select `Main.storyboard`, which is the file for storing user interface, Xcode shows you the Interface editor for designing the storyboard and app UI (see figure 3-9).

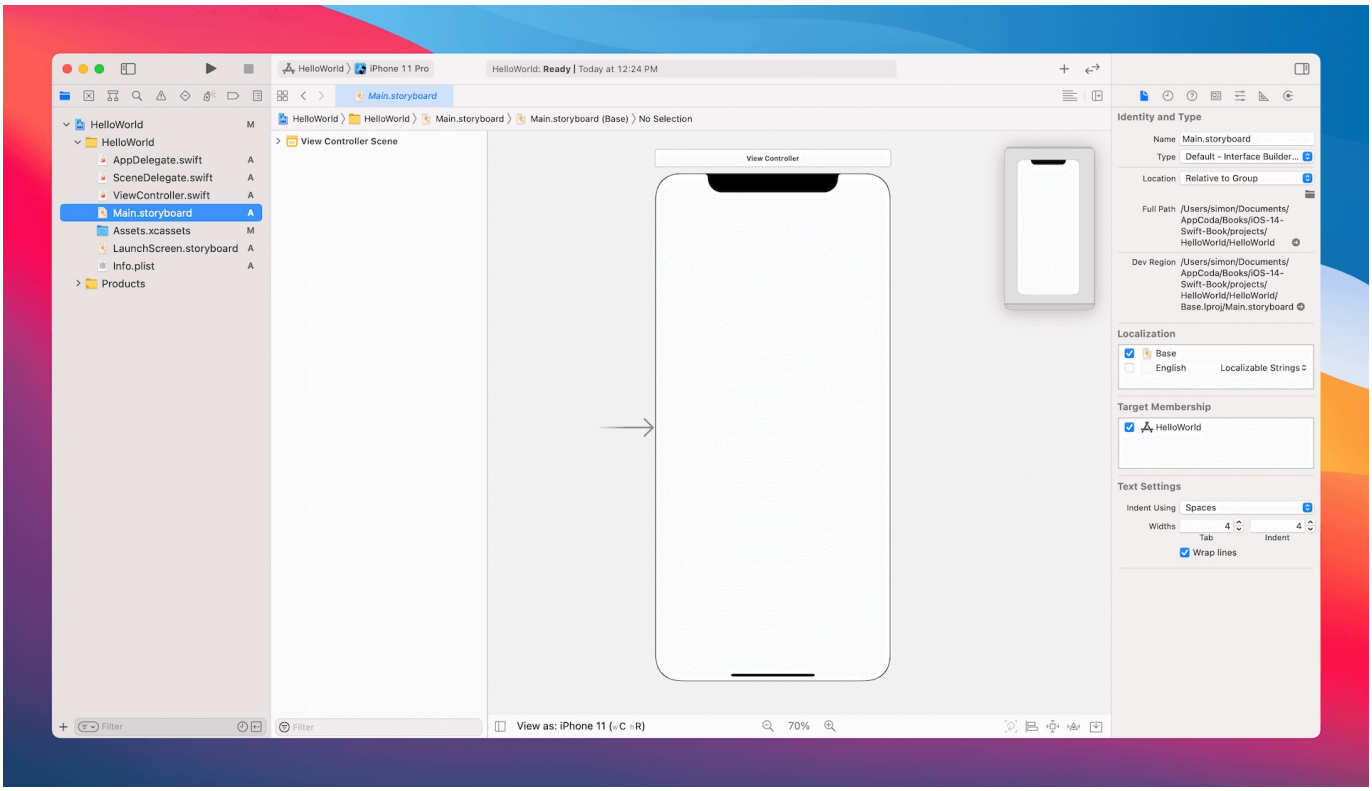


Figure 3-9. Xcode Workspace with Interface Editor

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.

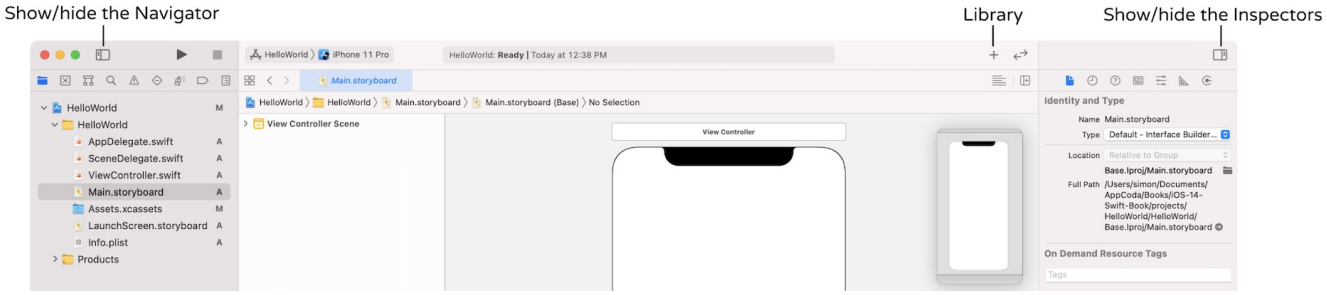


Figure 3-10. Show/hide the content areas of your workspace

Sometimes, you may want to hide the project navigator to free up more screen space for editing your code or user interface. You can click the button at the top-left corner of the window to control its appearance. The plus button is used to bring up the UI library. We'll talk about it later in the chapter, so don't worry if you do not understand what it is for.

```
Note: I used macOS Big Sur (i.e. 11.0) to run Xcode 12. If
your Mac is still using macOS Cataline (i.e. 10.15), the
buttons' positions are a bit different from mine.
```

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 *iPod touch*. If you click the *iPod touch* button, you'll see a list of available simulators such as iPhone SE and iPhone 11 Pro. Let's select *iPhone 11 Pro* 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 11 Pro.

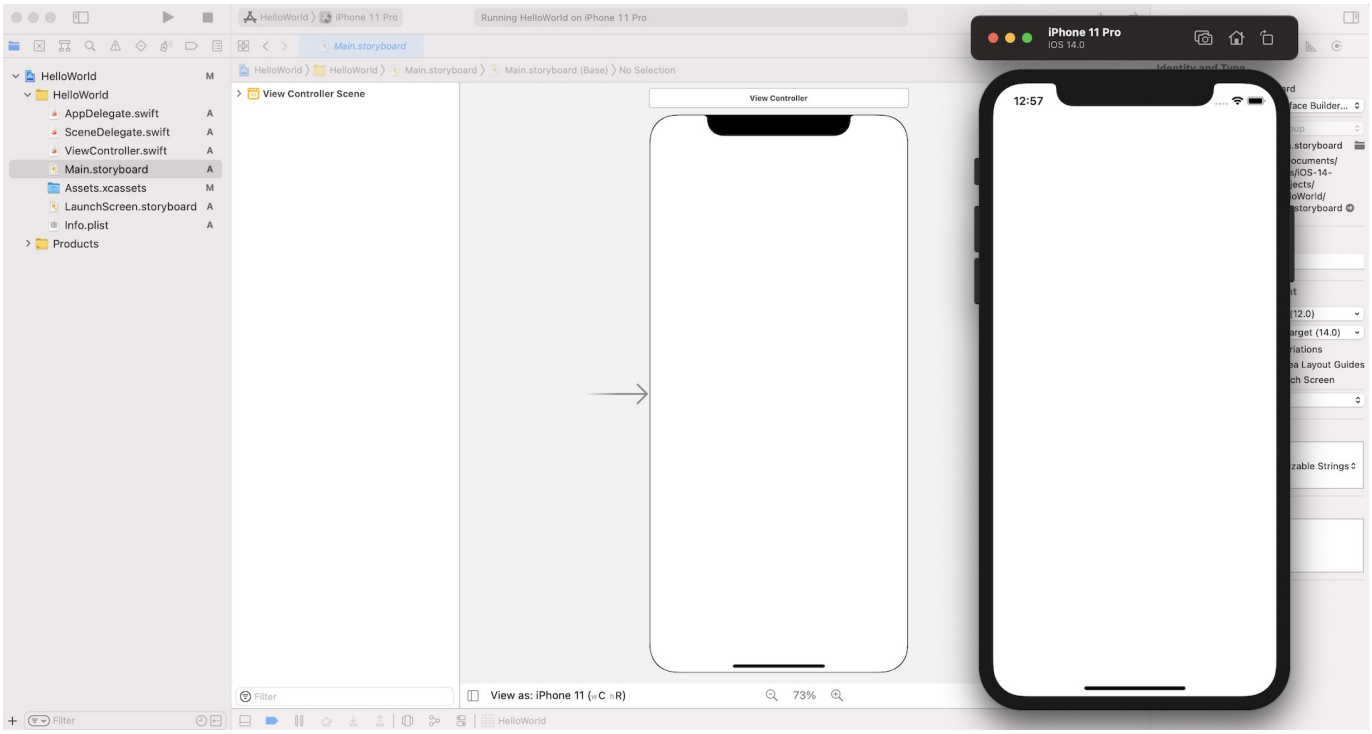


Figure 3-12. The Simulator

> **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 8/12) and run the app. You will see another simulator showing 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 (or press shift-command-h) to bring up the home screen. It also 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.

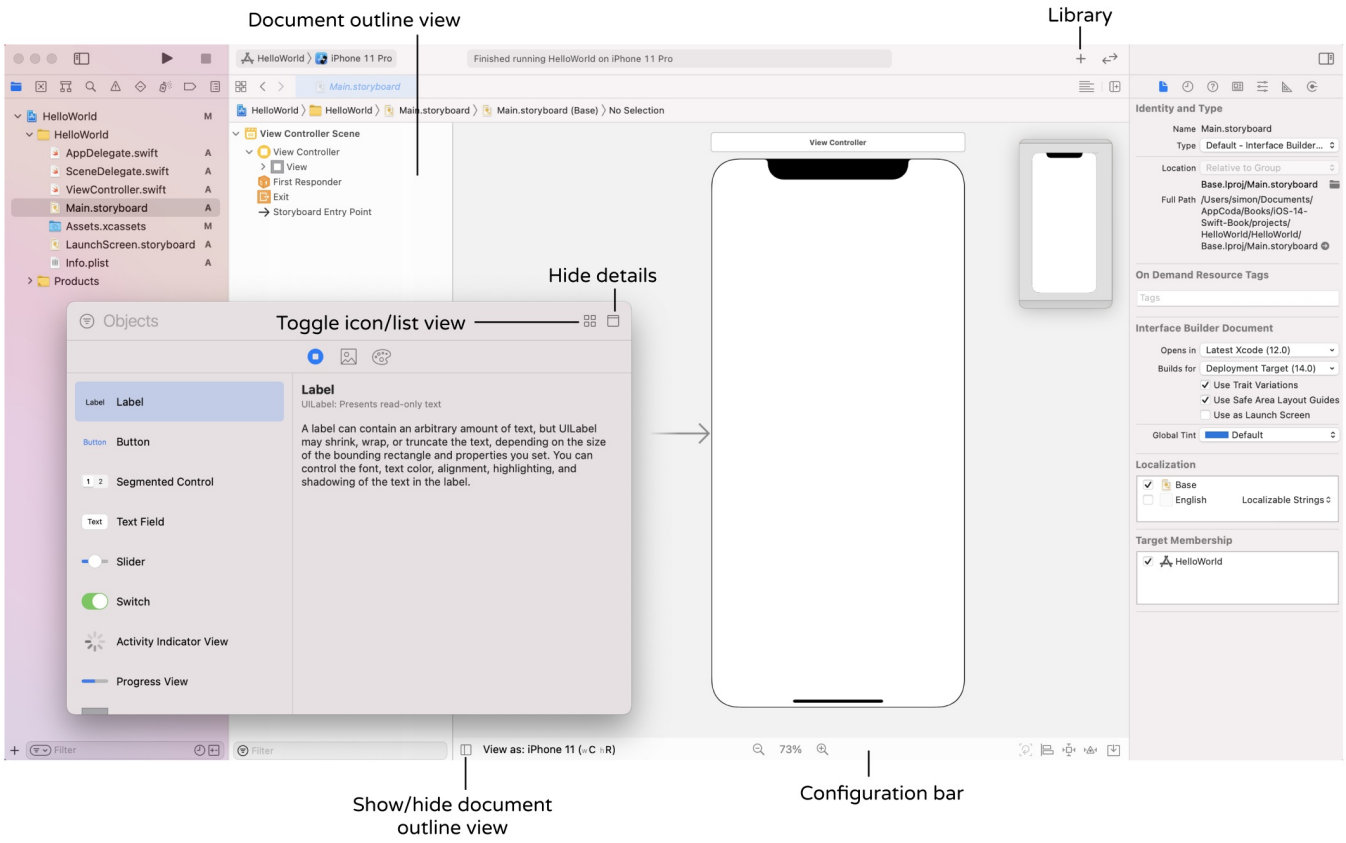


Figure 3-14. The Interface Builder Editor

In Xcode 12, Apple has made some visual changes. 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 + 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 *App* 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 11`, 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.

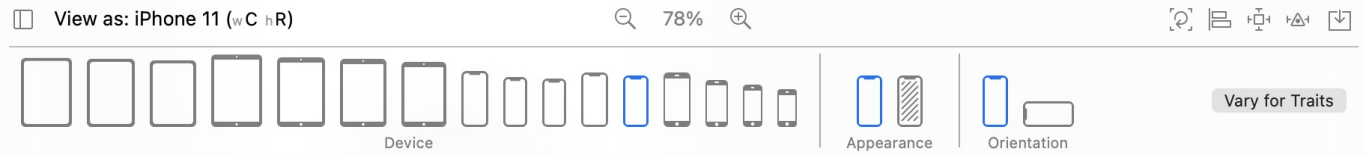


Figure 3-15. The configuration bar in Xcode

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 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 the *Show Details* button (refer to figure 3-14) 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.

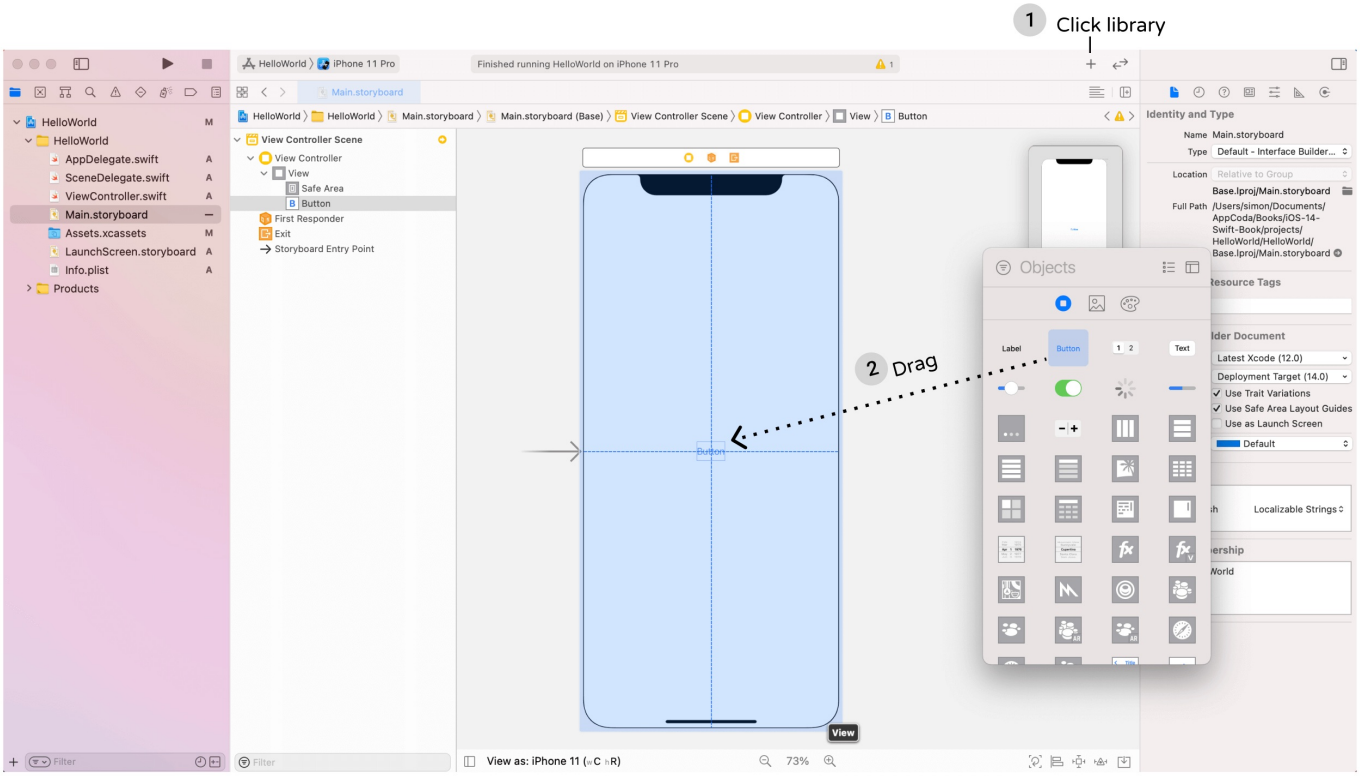


Figure 3-16. Drag the Button to the View

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.

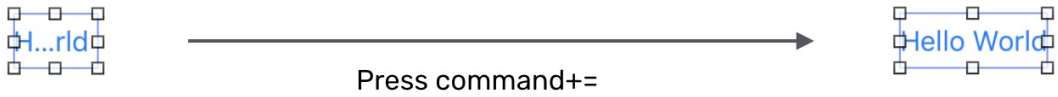


Figure 3-18. Resizing the Hello World button

Great! You're now ready to test your app. Select the iPhone 11 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?

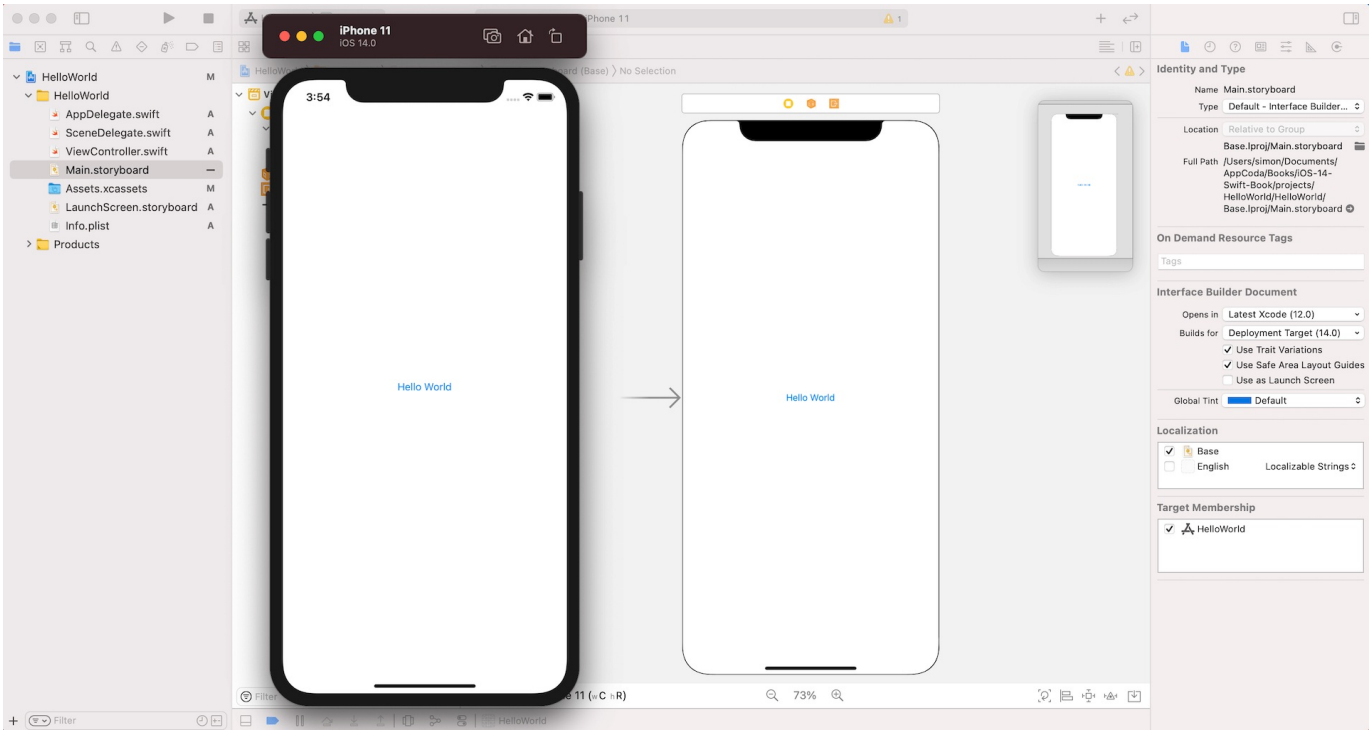


Figure 3-19. Hello World app with a Button

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 *App* 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. The editor area immediately displays the source code. Insert the following lines of code inside the `ViewController` class:

```
@IBAction func showMessage(sender: UIButton) {
    let alertController = UIAlertController(title: "Welcome to My First Ap
p", message: "Hello World", preferredStyle: UIAlertController.Style.alert)
    alertController.addAction(UIAlertAction(title: "OK", style: UIAlertAct
ion.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.
    }

    @IBAction func showMessage(sender: UIButton) {
        let alertController = UIAlertController(title: "Welcome to My Firs
t App", message: "Hello World", preferredStyle: UIAlertController.Style.al
ert)
        alertController.addAction(UIAlertAction(title: "OK", style: UIAler
tAction.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.

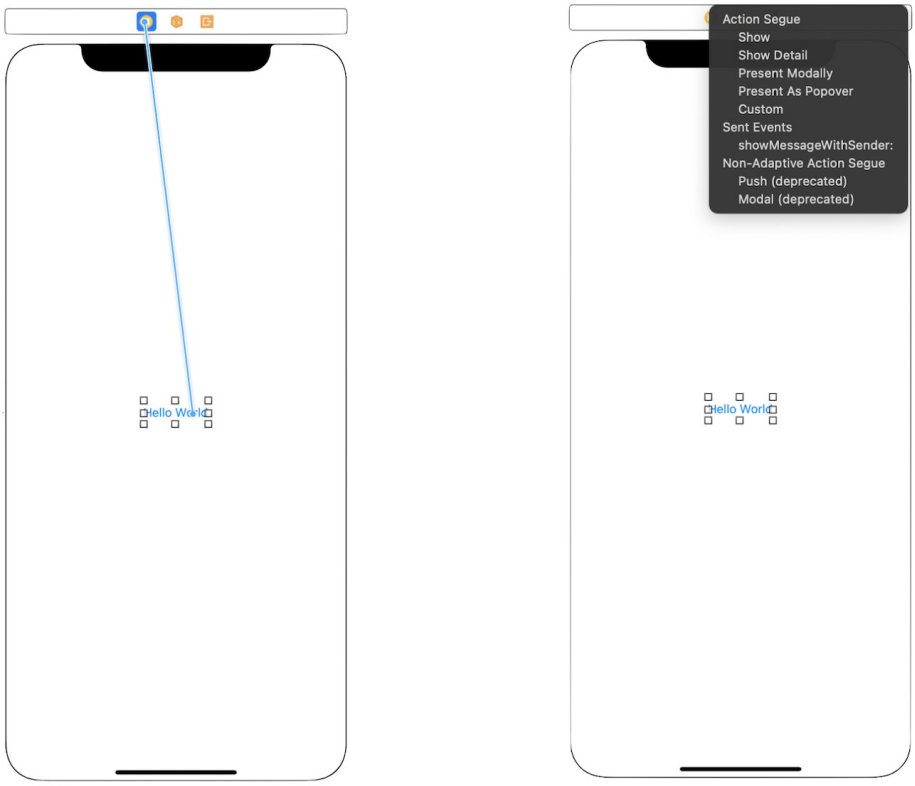


Figure 3-20. Drag to the View Controller icon (left), a pop-over menu appears when releasing the buttons (right)

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.

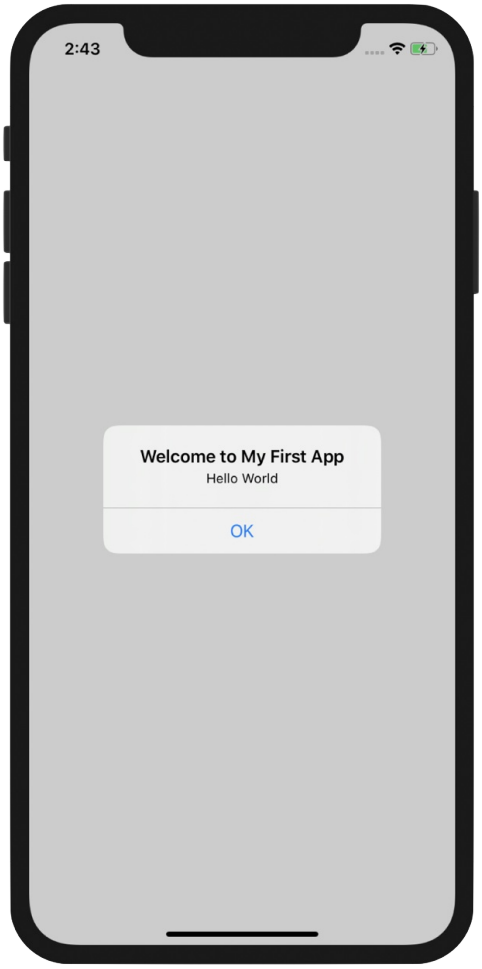


Figure 3-21. Hello World app

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 *System Purple Color* or whatever color you want.

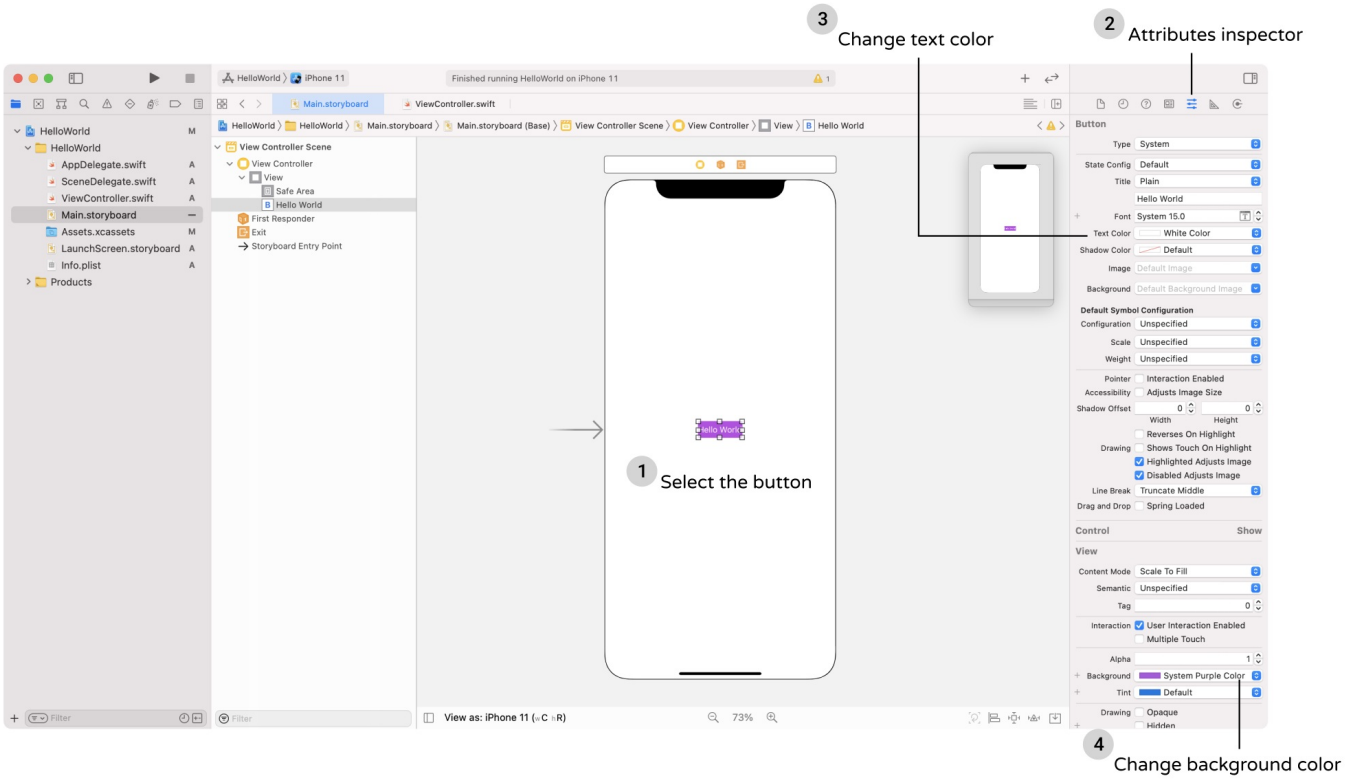


Figure 3-22. Changing the color of the Hello World button

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.

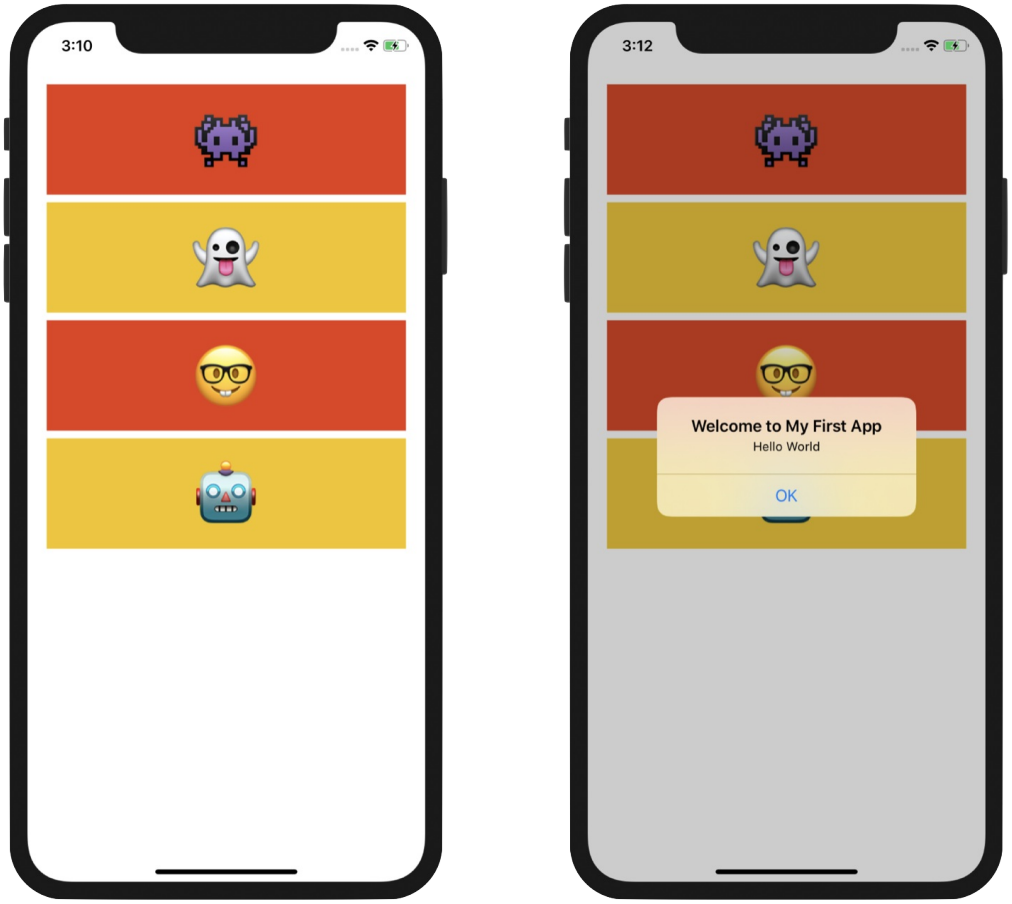


Figure 3-23. Design this app

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 inste
ad of Hello World
        let alertController = UIAlertController(title: "Meaning", message:
 meaning, preferredStyle: UIAlertController.Style.alert)

        alertController.addAction(UIAlertAction(title: "OK", style: UIAler
tAction.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/swift53/HelloWorld.zip.

For the solution of the exercise, you can download it from http://www.appcoda.com/resources/swift53/HelloWorldExercise.zip.

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.

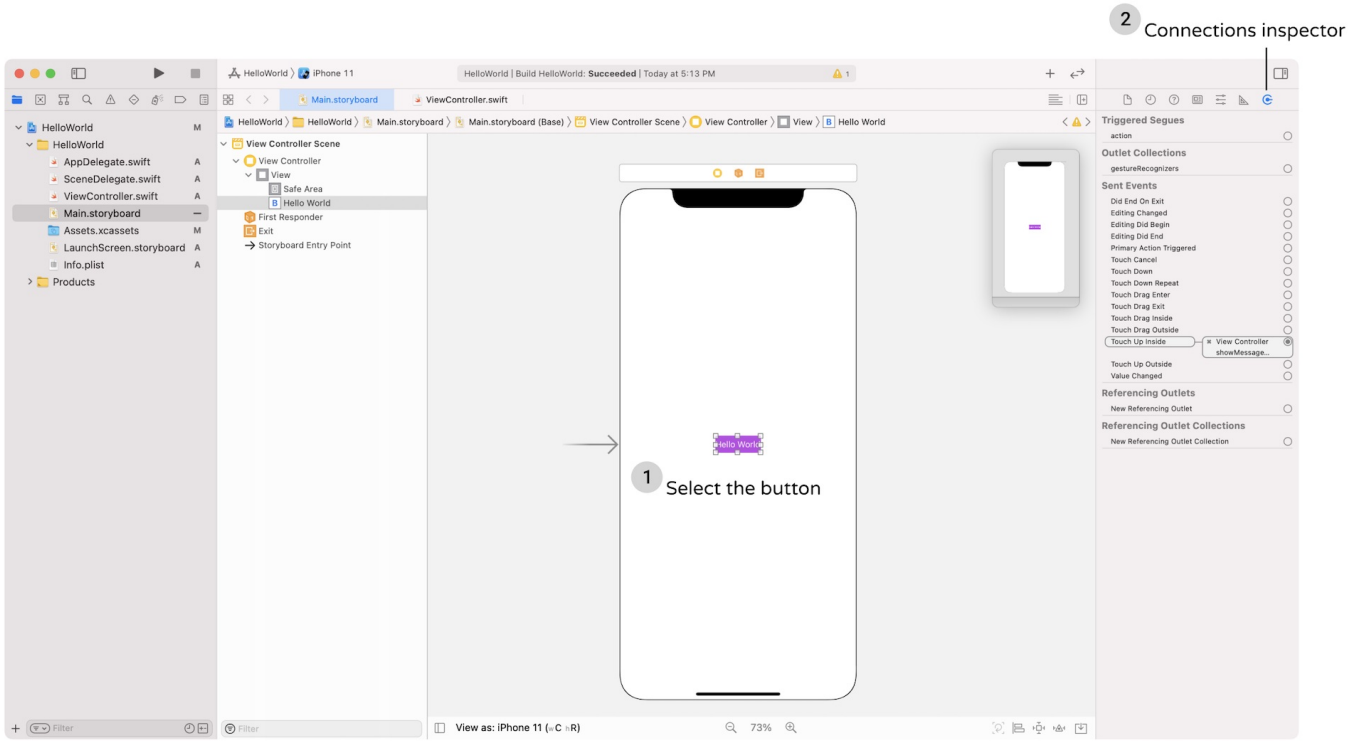
Quick note: Like Objective-C, Swift is an Object-Oriented Programming (OOP) language. Most of the code in an app in some ways deals with objects of some kind. I don't want to scare you away by teaching you the OOP concepts here. Just keep reading. As we go along, you'll learn more about OOP.

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?

Open `Main.storyboard` again and select the "Hello World" button. Click the Connection inspector icon in the Utility area. Under the Sent Events section, you should find a list of available events and its corresponding method to call. As you can see in the figure below, the "Touch Up Inside" event is connected to the `showMessage(sender:)` method.



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.

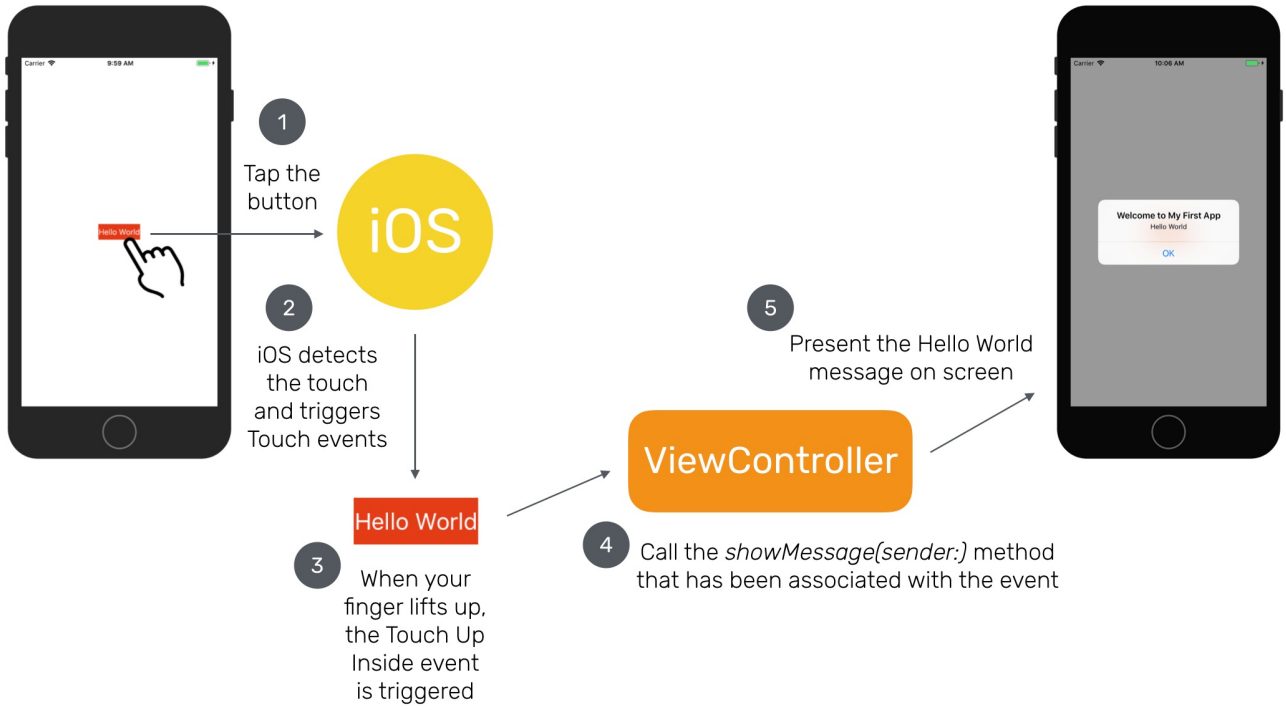


Figure 4-3. Event flow in the Hello World App

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.

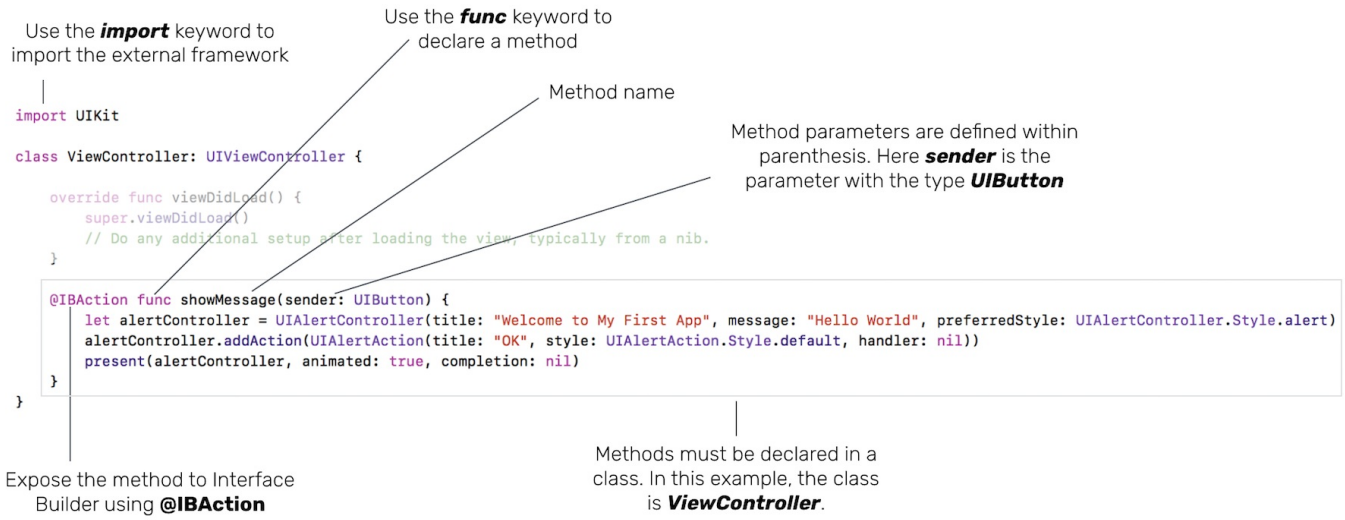


Figure 4-4. showMessage() method explained

> **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.default, 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, click the *Open in Developer Documentation* link. This will bring you to the official documentation of the class.

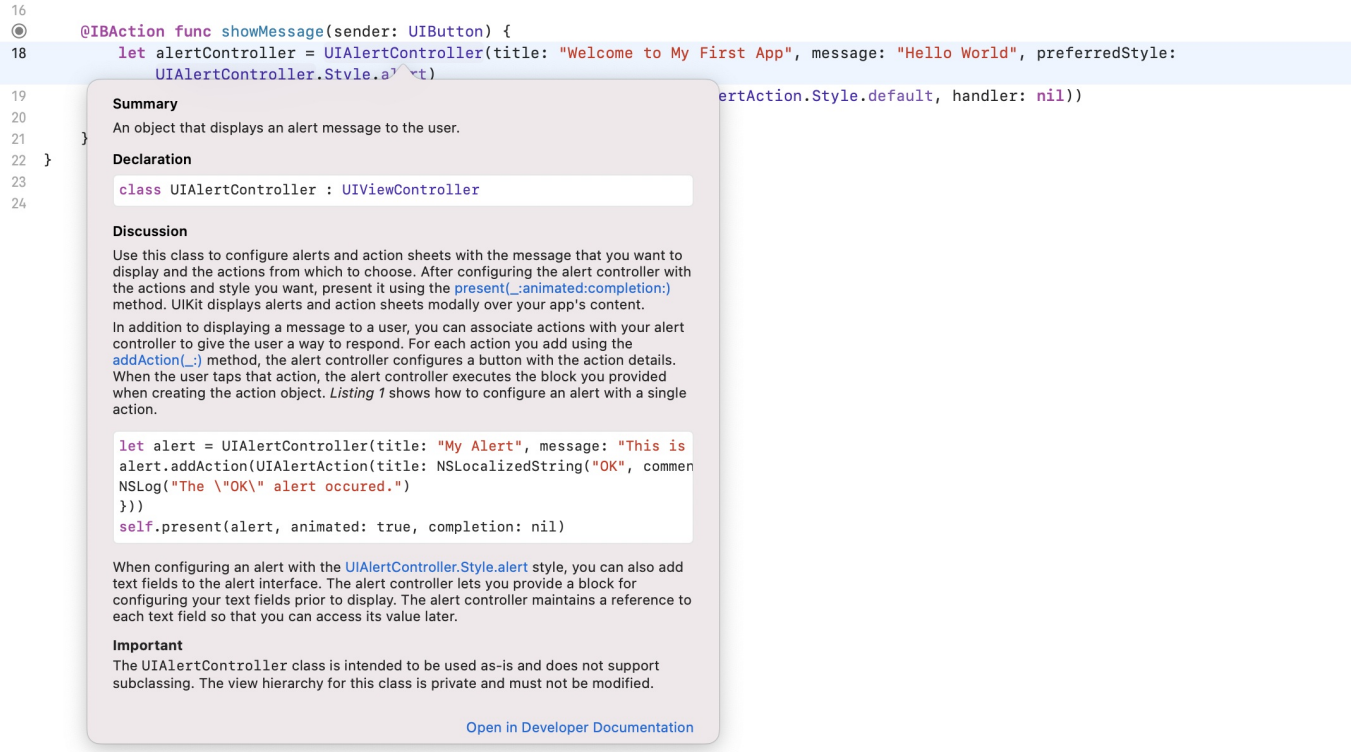


Figure 4-5. Class Information

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)
```

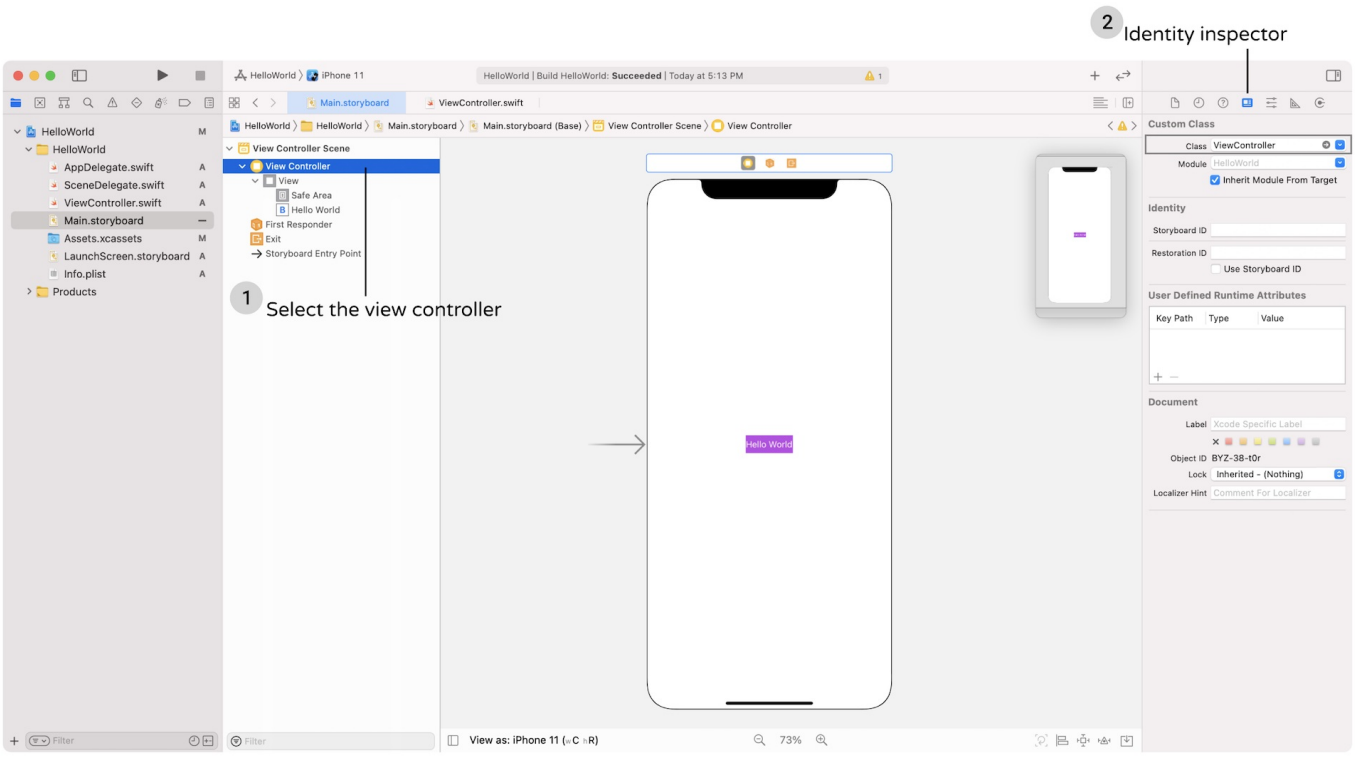
In Swift, you use the `self` property to refer to the current instance (or object). In most cases, the `self` keyword is optional. So you can omit it.

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 *App* 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.



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.

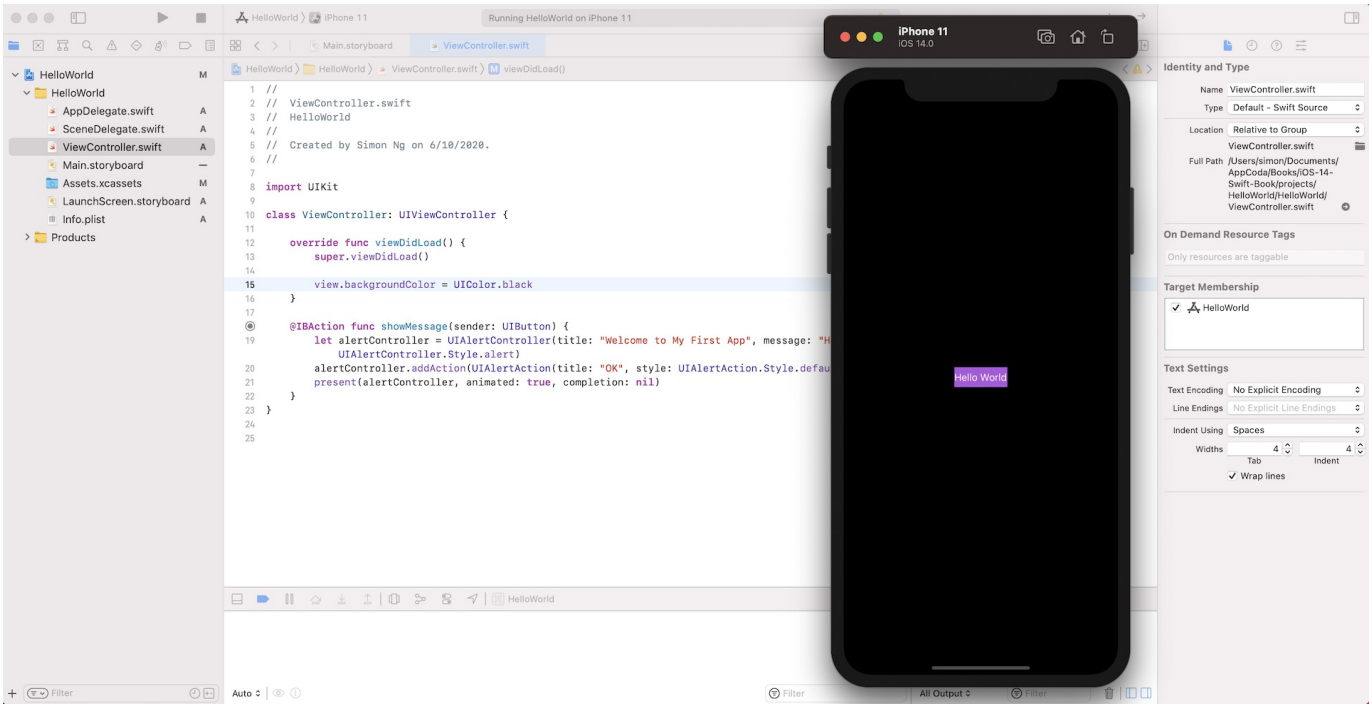


Figure 4-7. Changing the background color of the view

`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.

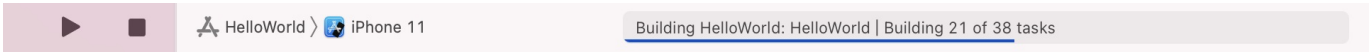


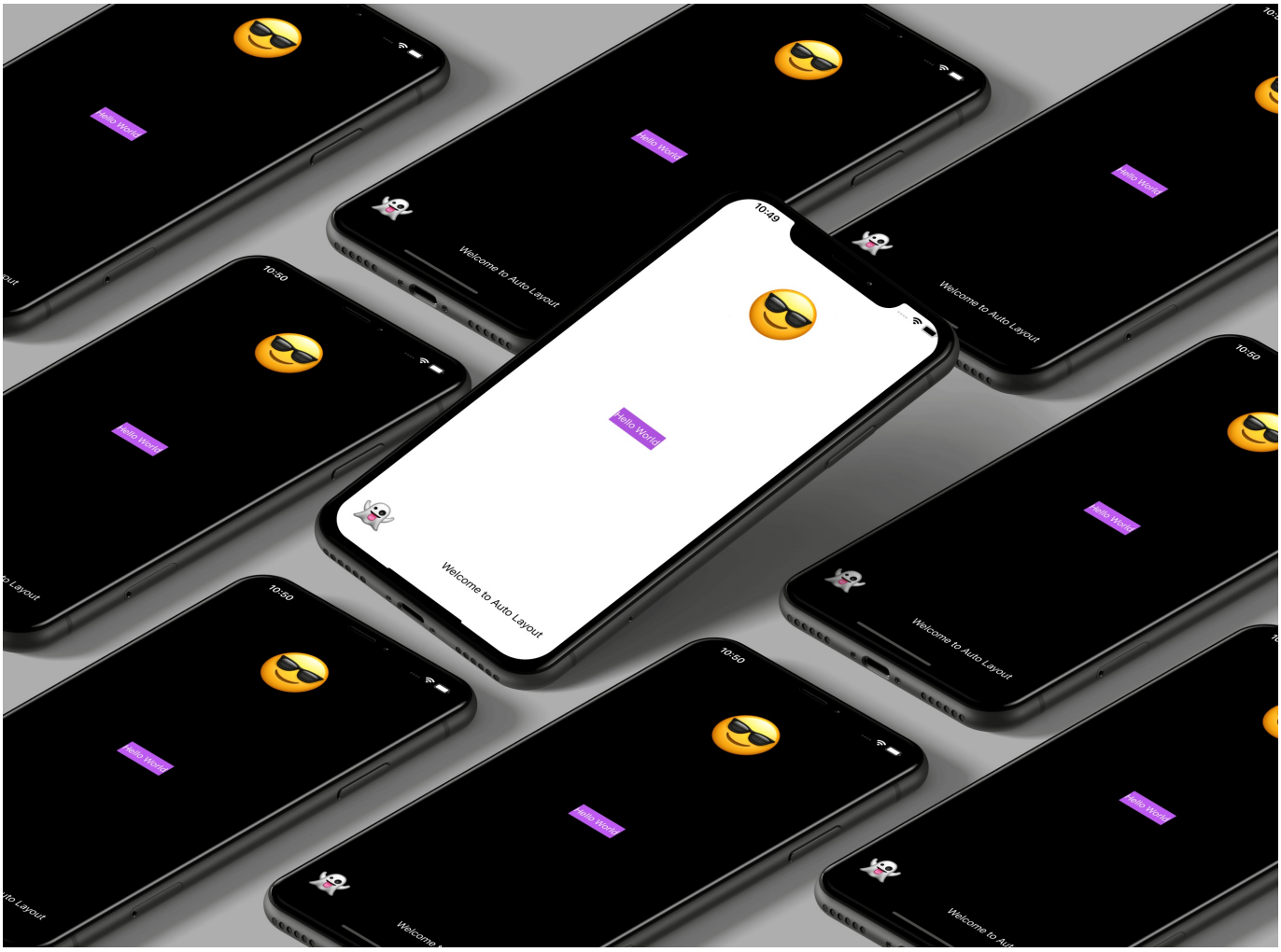
Figure 4-8. Build Process

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.

If you got any questions about this chapter, don't hesitate to ask. You can join our private Facebook group (https://facebook.com/groups/appcoda) to ask questions.

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, 5.8-inch, 6.1-inch, and 6.5-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 11 simulator, run it using the iPhone SE or 11 Pro 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 6.1-inch screen.

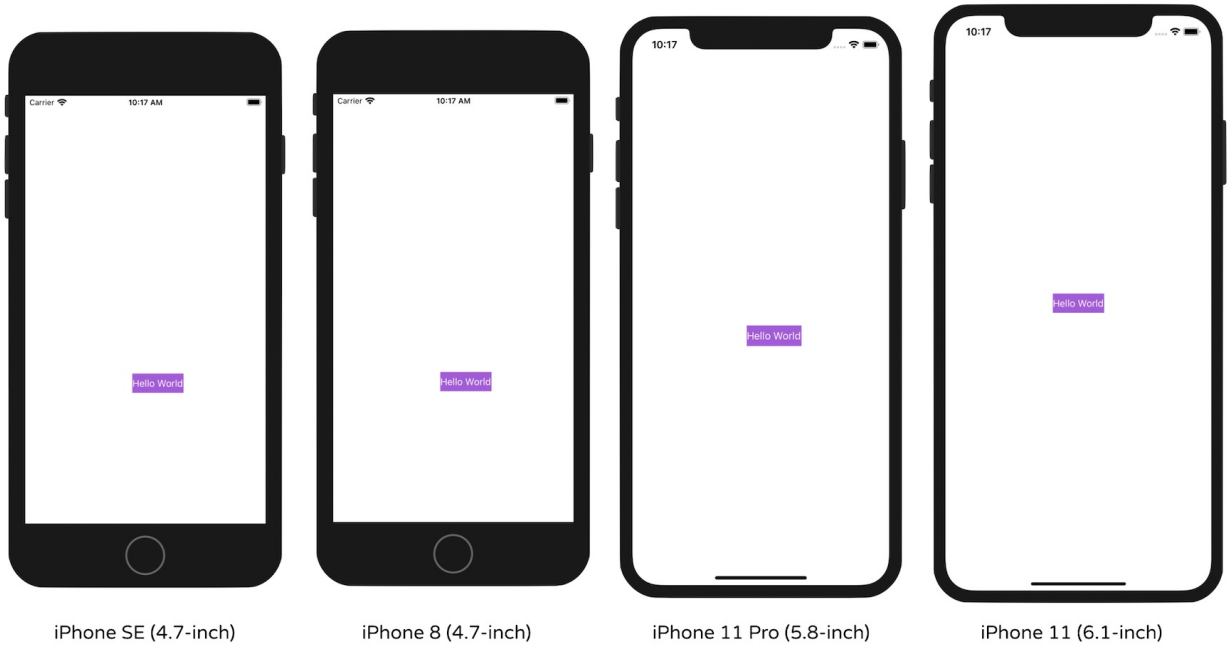


Figure 5-1. The app UI looks different when running on iPhone SE, iPhone 8, iPhone 11 and iPhone 11 Pro

Let's try one more thing.

Click the Stop button and run the app using the iPhone 11 simulator. After the simulator launches, go up to the menu and select *Device* > *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 (and it even goes off the screen).

Why? What's wrong with it?

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

- For iPhone SE (old model), 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/SE (new model), 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 iPhone X/XS/11 Pro, the screen consists of 375 points (or 1125 pixels) horizontally and 812 points (or 2436 pixels) vertically.
- For iPhone XR/11, the screen consists of 414 points (or 828 pixels) horizontally and 896 points (or 1792 pixels) vertically.
- For iPhone XS Max/11 Pro Max, the screen consists of 414 points (or 1242 pixels) horizontally and 896 points (or 2688 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 (167, 433). You can find out the absolute position in the Size inspector, which is the button next to the Attributes inspector.

Whether you're using a 4.7-inch or 5.8-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 11 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 SE, iPhone 11 Pro and iPhone 11

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/swift53/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 11 (6.1-inch). To see how your app looks on other devices, click `View as: iPhone 11` 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.

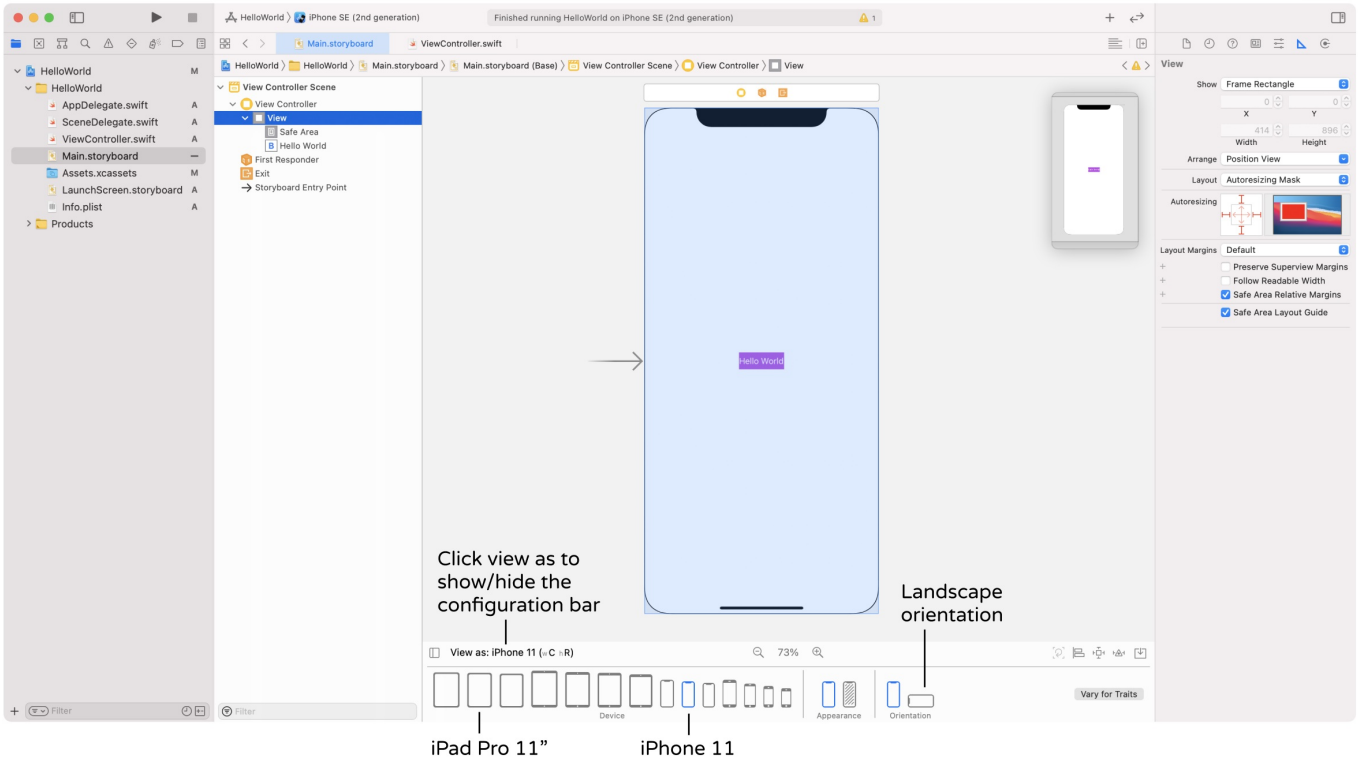


Figure 5-3. Live preview using Xcode's configuration bar

The configuration bar is a great feature for previewing the UI on different devices. Take some time to play around with it.

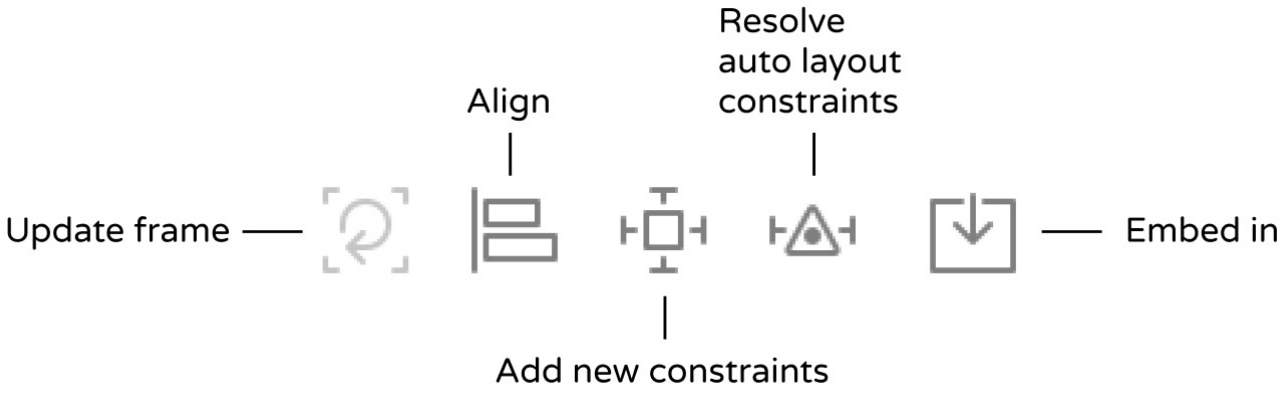
Quick note: You may wonder what "(wC hR)" means. Just forget about it right now and focus on learning auto layout. I will discuss about it in the later chapter.

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.
- **Embed in** – Embed views into a stack view (or other views). We will further

discuss it in the next chapter.

- **Resolve auto layout issues** – Resolve layout issues.
- **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 "Horizontally in Container" and "Vertically in Container" options. Then click the "Add 2 Constraints" button.

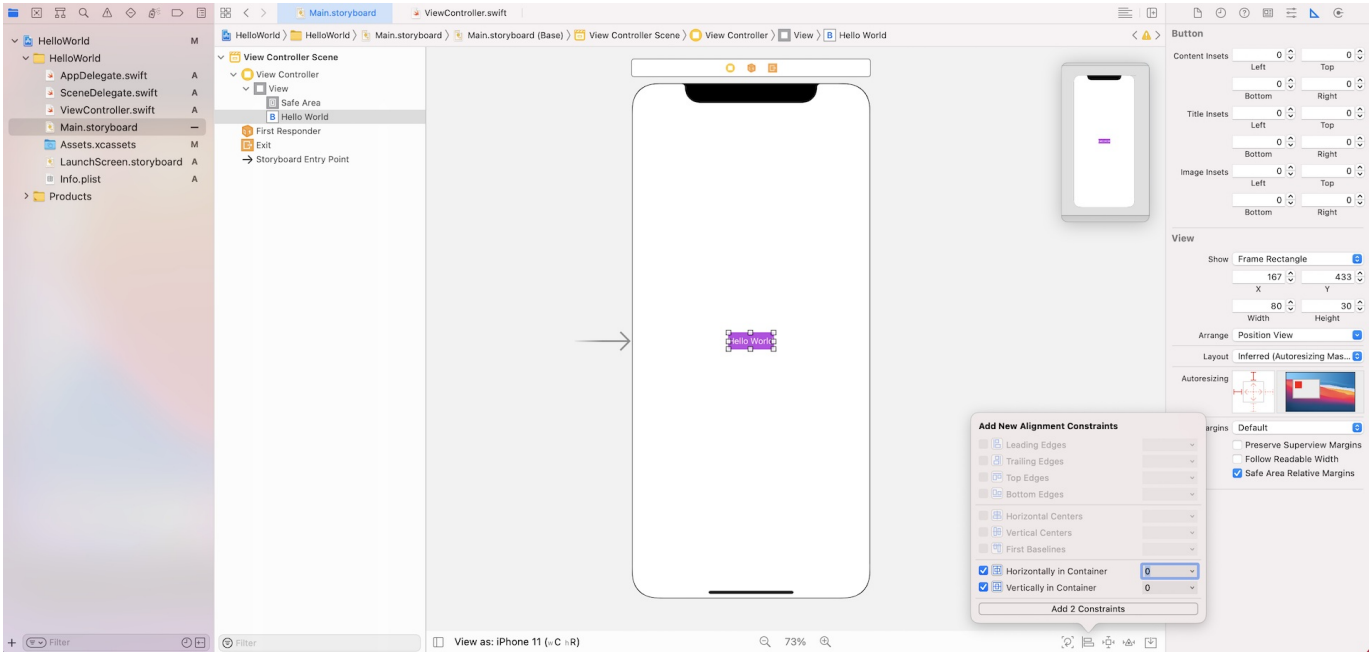


Figure 5-5. Adding constraints using Align button

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.

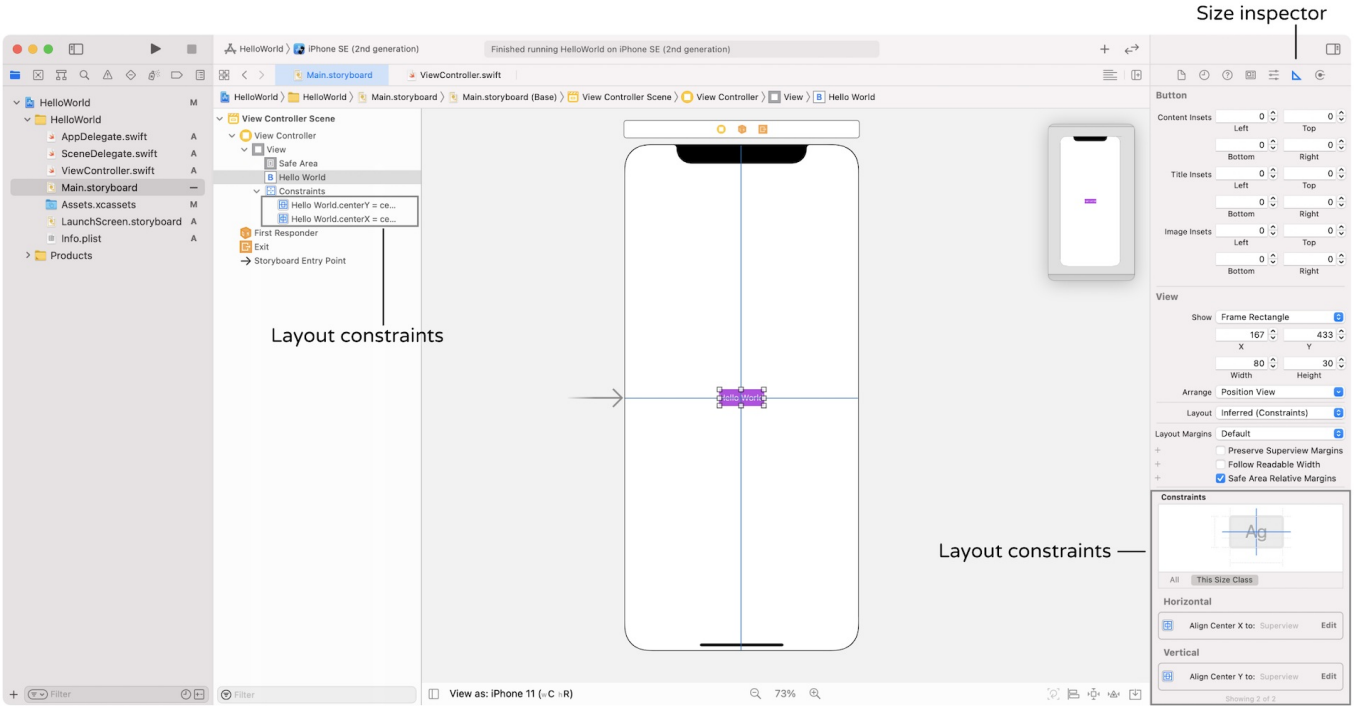


Figure 5-6. View the constraints in Document Outline and Size Inspector

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 8 (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.

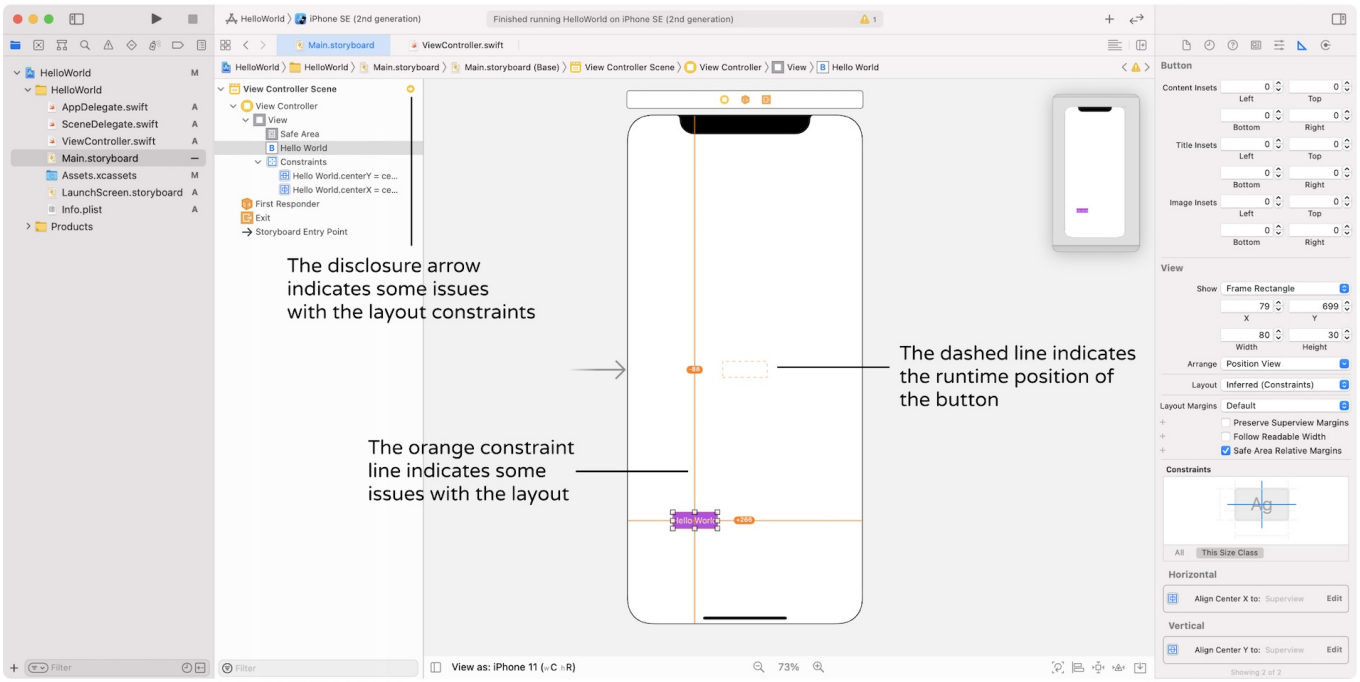
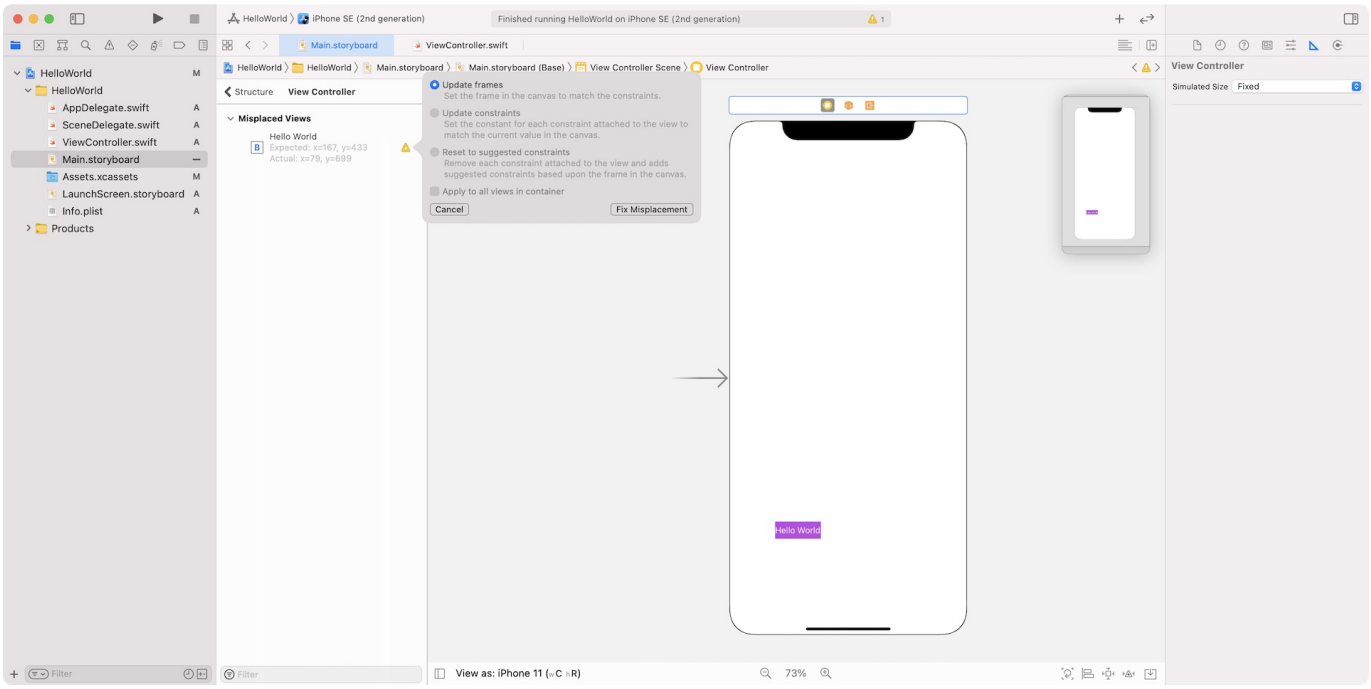


Figure 5-7. Interface Builder uses orange/red lines to indicate Auto Layout issues

Auto layout issues occur when you create ambiguous or conflicting constraints. Base on the constraints configured, they specified that 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 is now confused, thus 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, click the *Adjust Editor Options* button and choose *Preview*. You can refer to figure 5-9 for the steps.

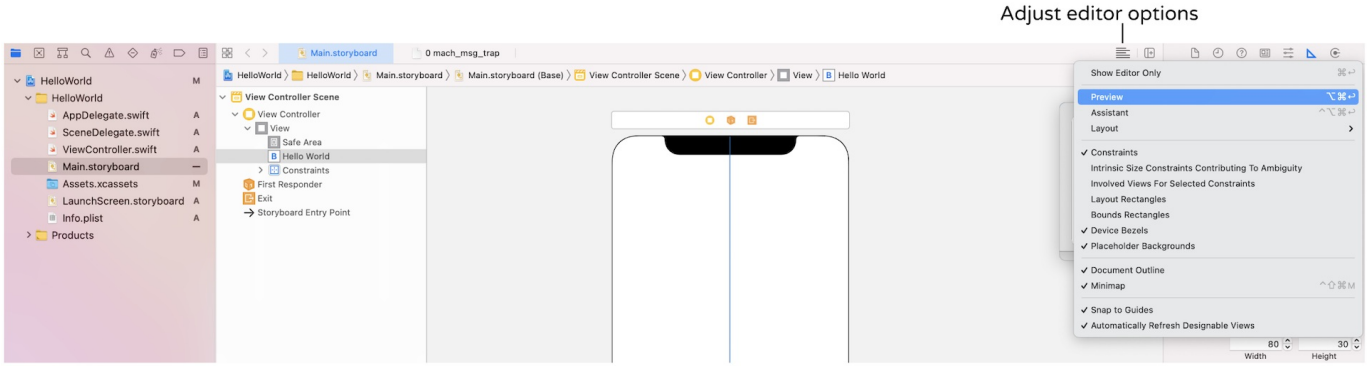
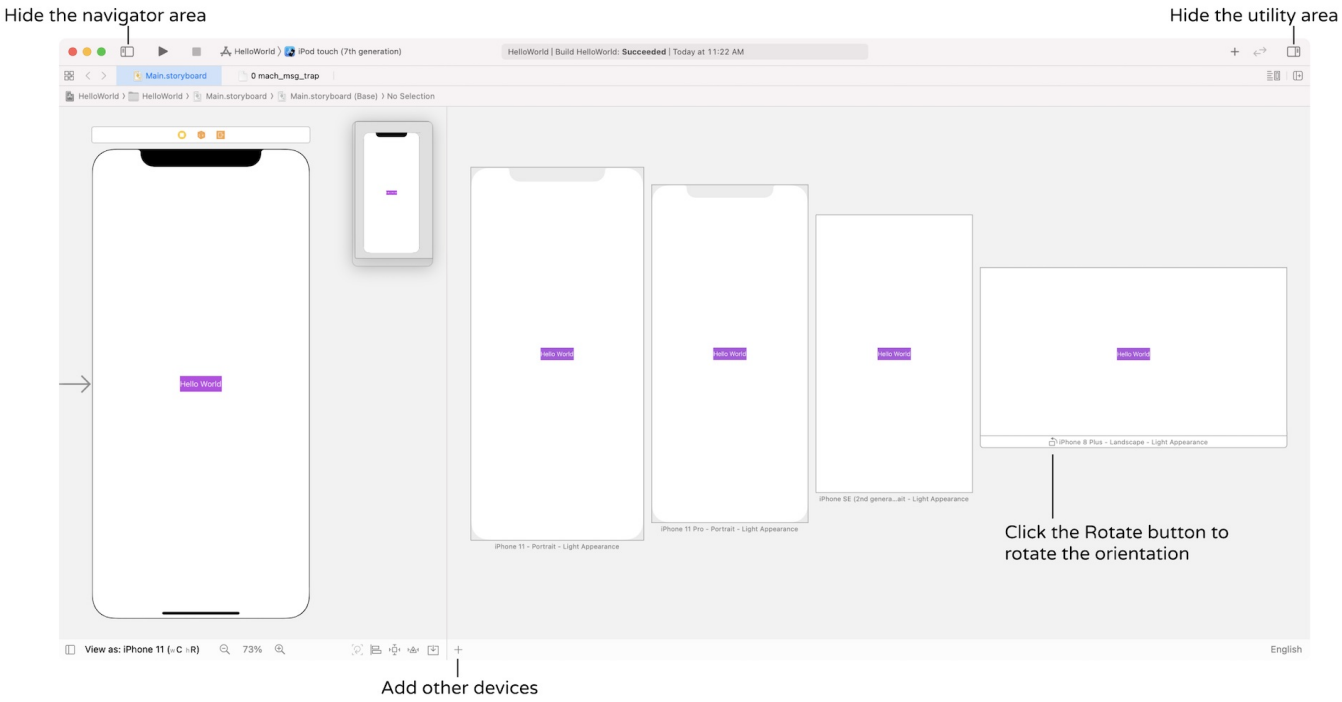


Figure 5-9. Adjust editor options

Xcode will display a preview of your app's UI in the assistant editor. By default, it shows you the preview on the iPhone 11 device. You can click the + button at the lower-left corner of the assistant editor to add other iOS devices (e.g. iPhone SE/8/11 Pro) 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. To close the preview, you can click the *Adjust Editor Options* button and choose *Show Editor Only*.

> **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 ideas 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.

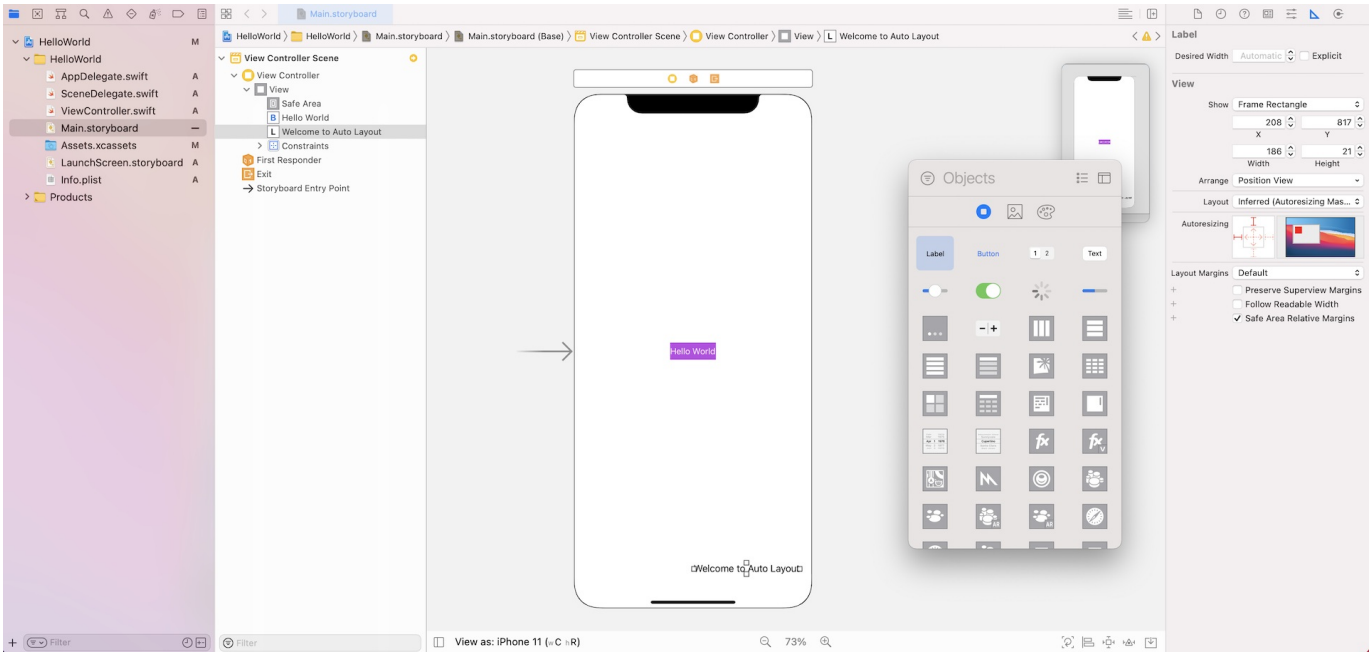


Figure 5-11. Adding a label to the view

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 11.

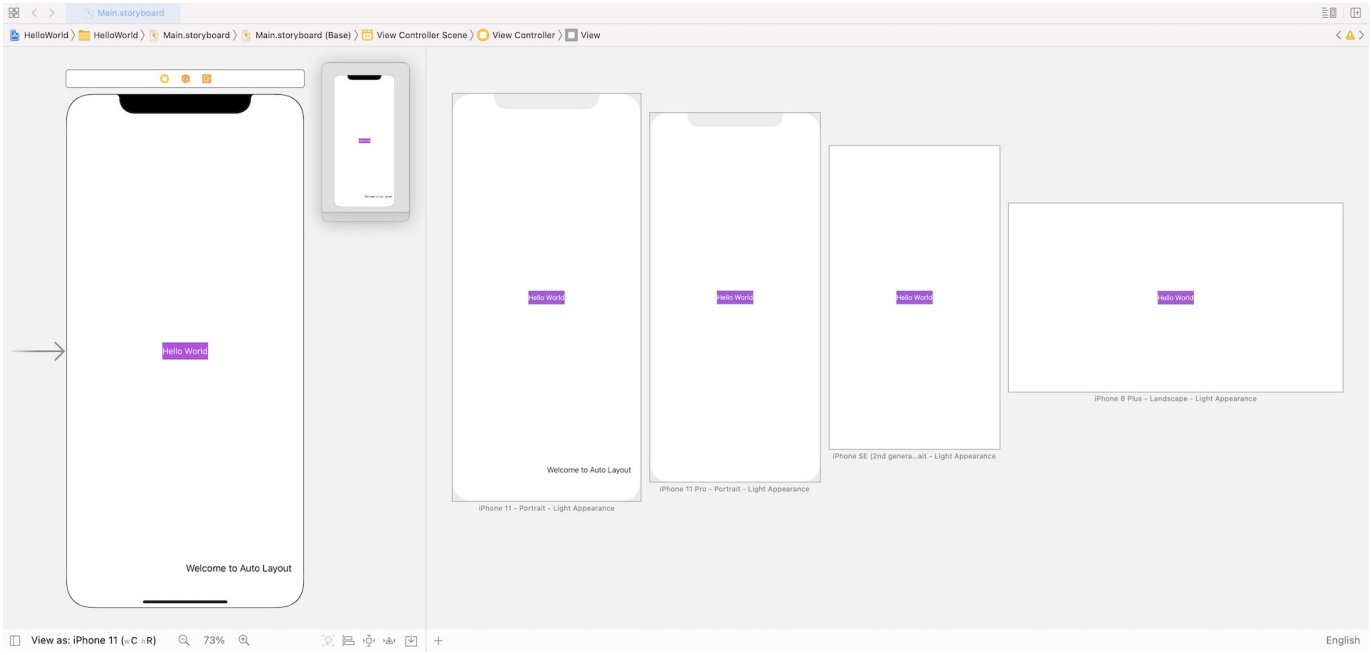


Figure 5-12. The label can't be displayed properly

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 20 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 20 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.

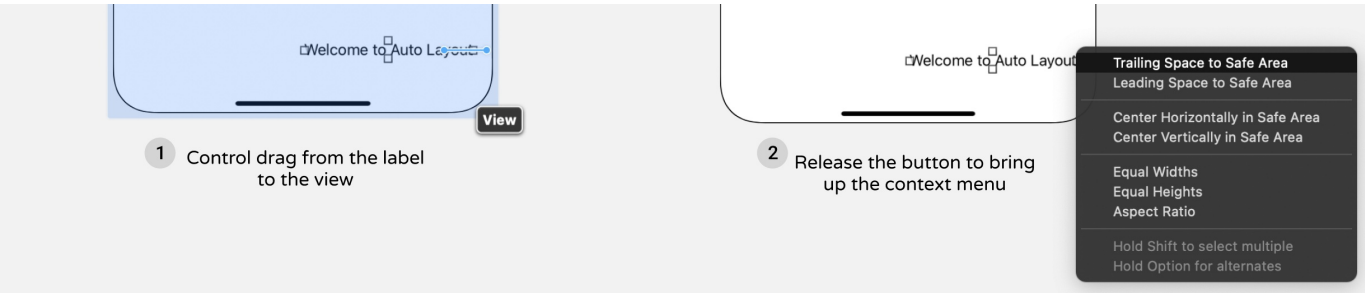


Figure 5-13. Using Control-drag to add the first constraint

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.

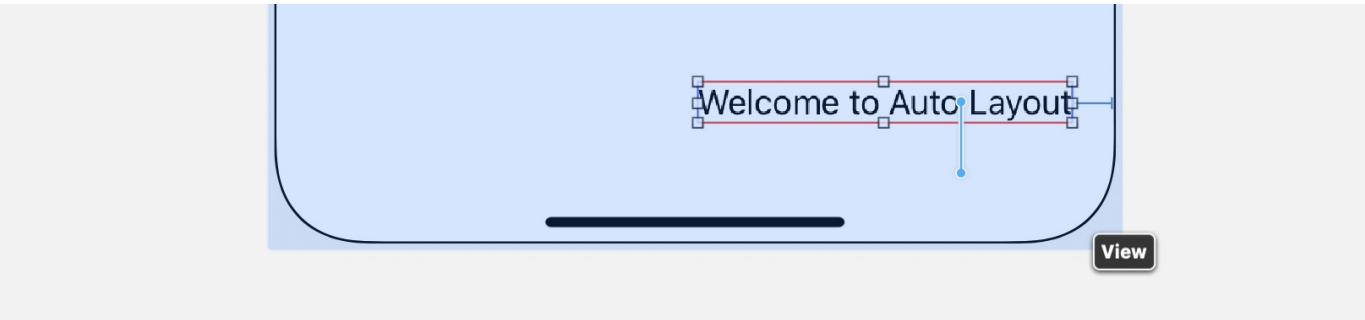


Figure 5-14. Using Control-drag to add the second constraint

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.

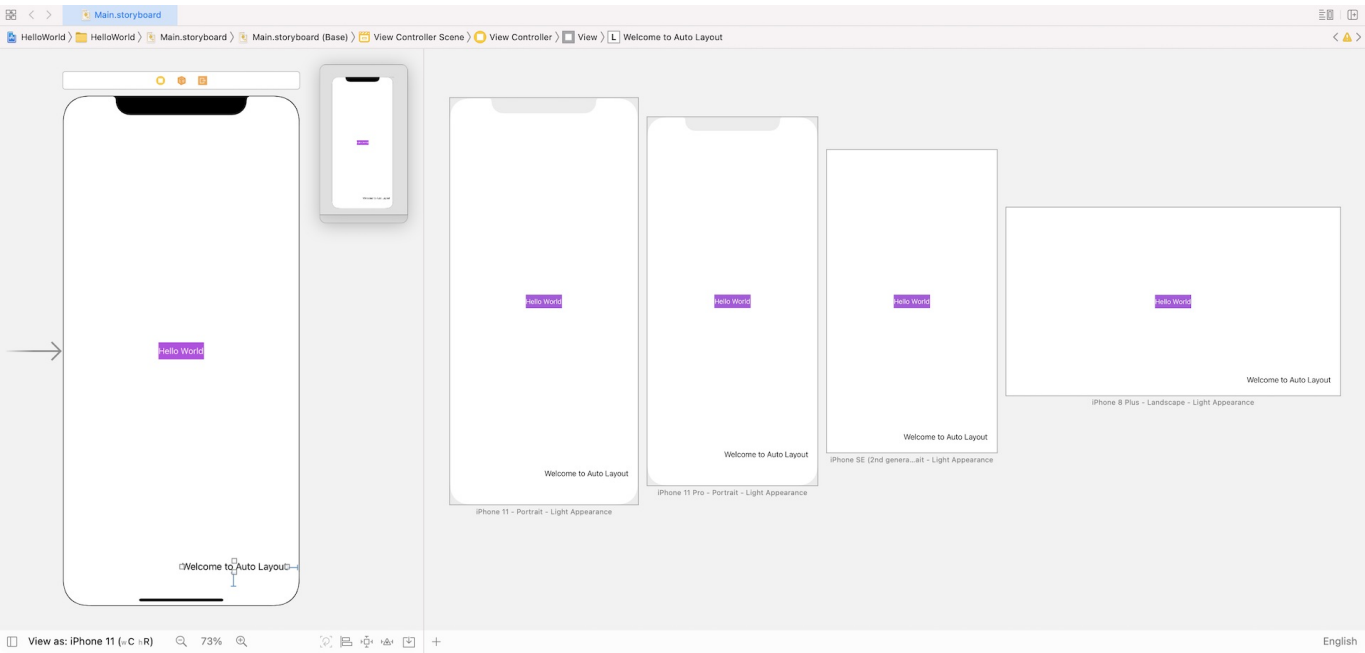


Figure 5-15. The UI now supports all screen sizes

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?

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 12, 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.

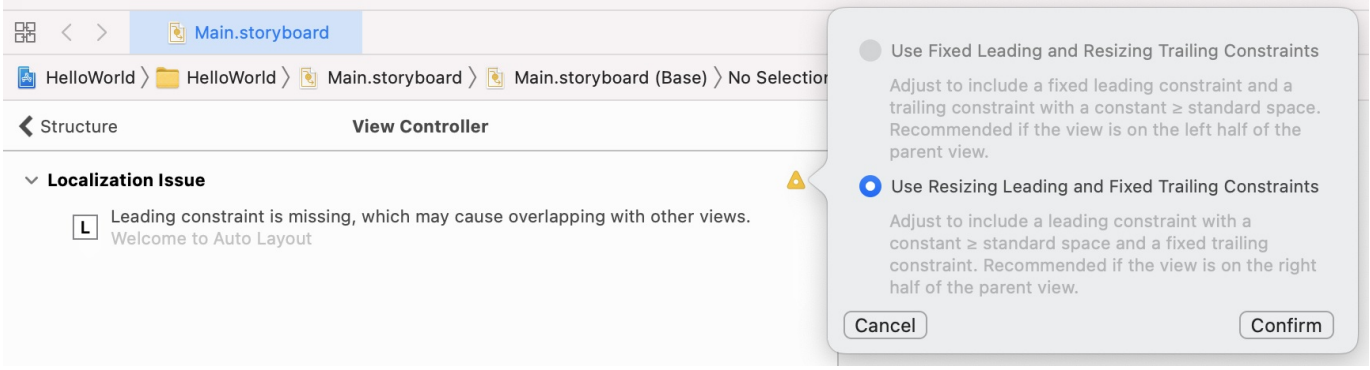


Figure 5-16. Layout issues related to localization

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 was 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.

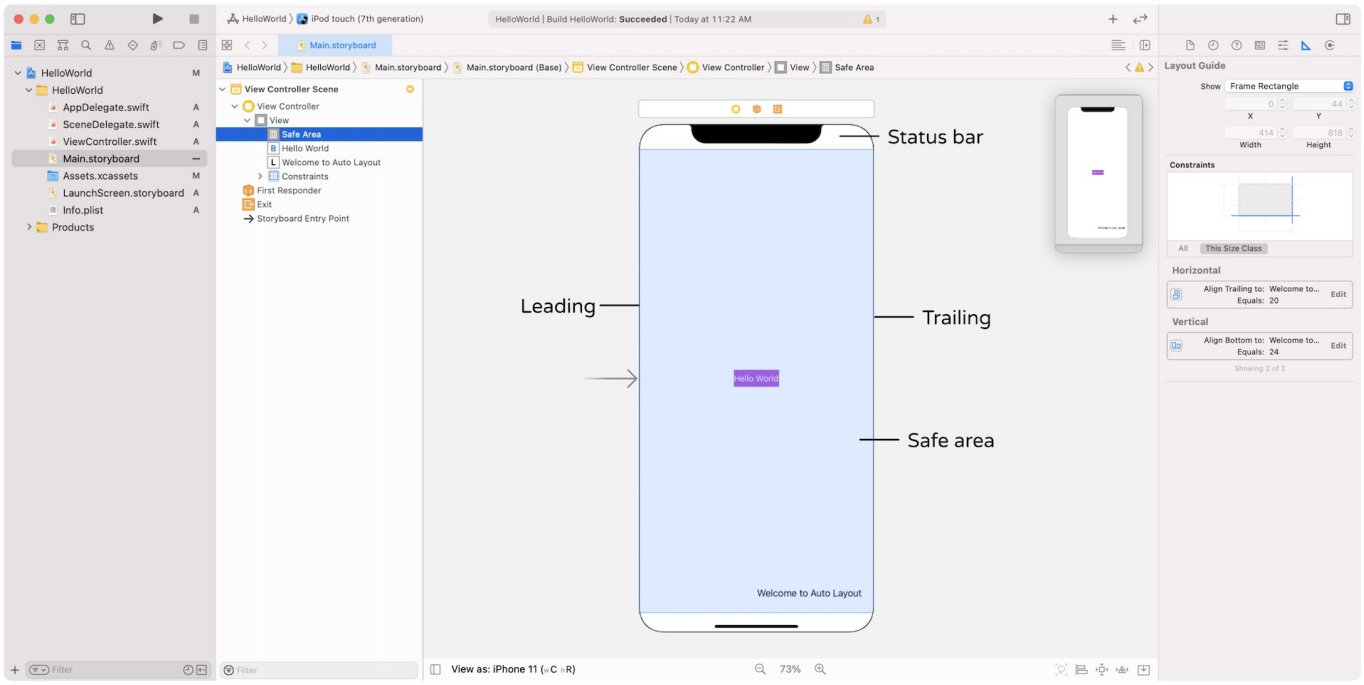


Figure 5-17. Safe area

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.

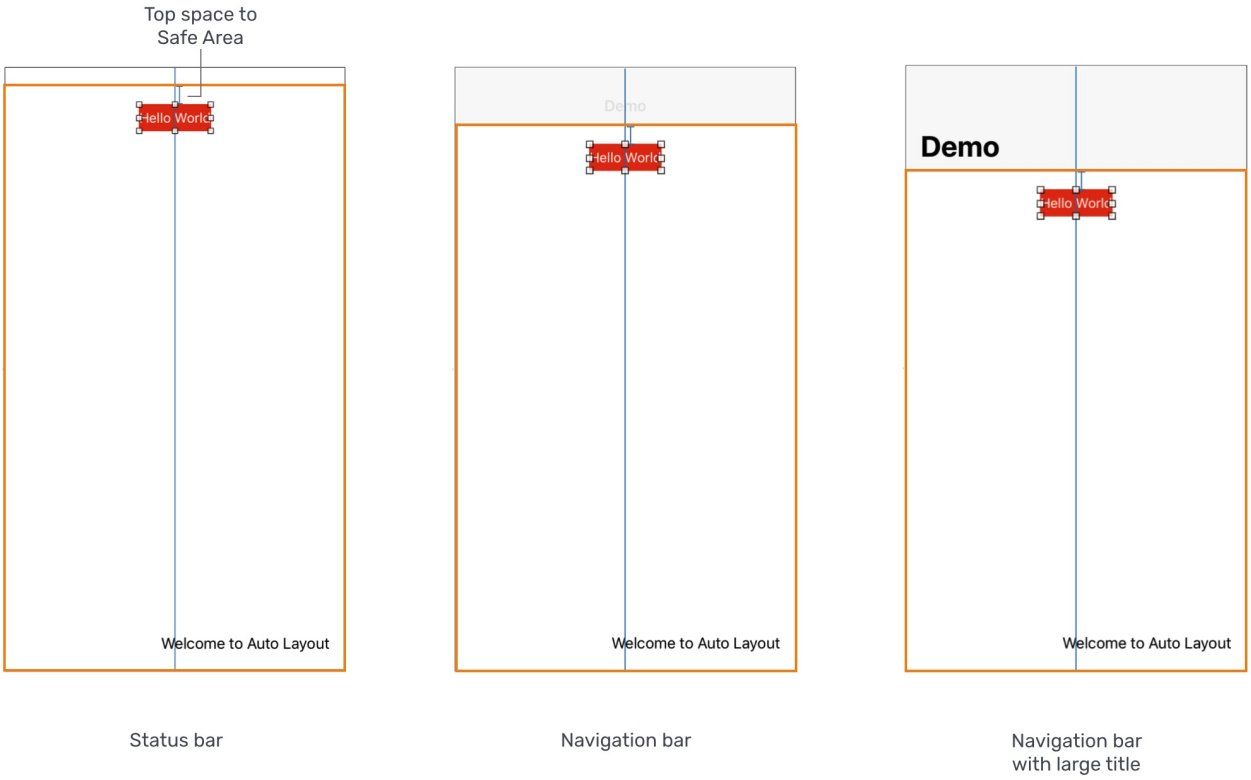


Figure 5-18. Safe areas adjust automatically

In case if the view contains a navigation bar, whether it uses the standard title or large title, 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. You can change the value of Constant to `30` to add some extra space.

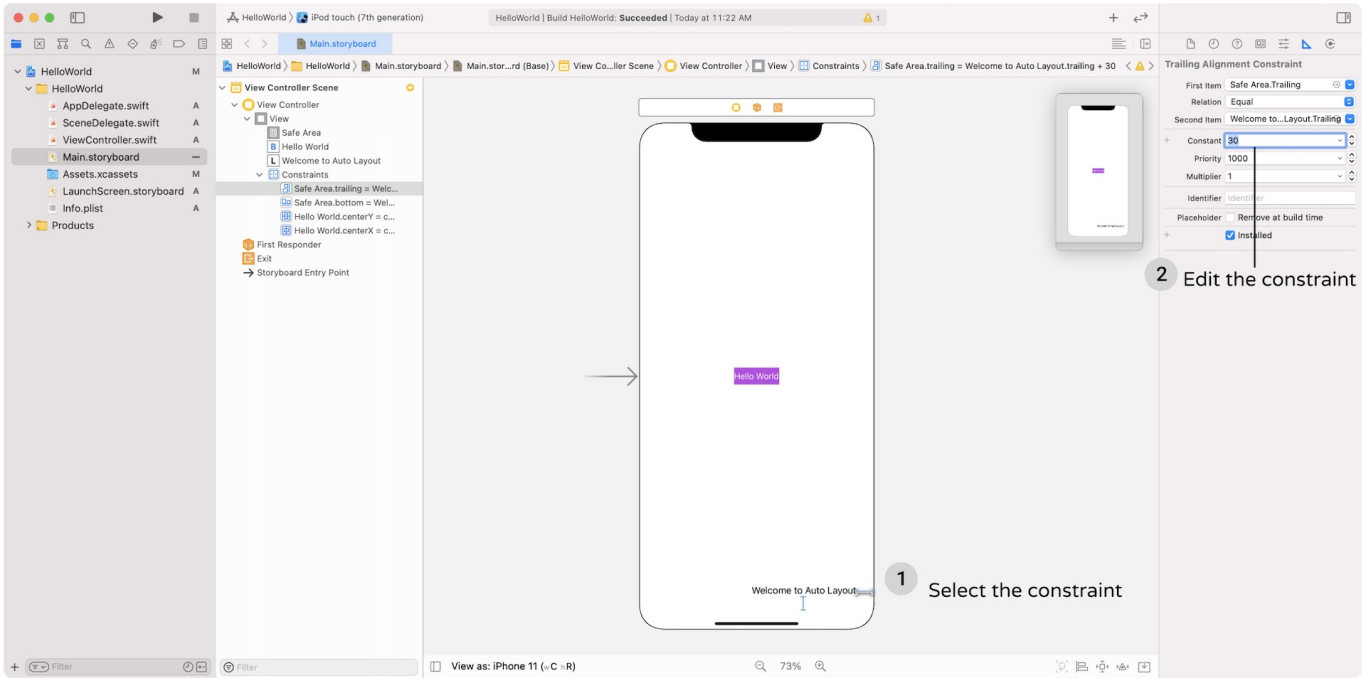


Figure 5-19. Editing a constraint using Attributes inspector

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

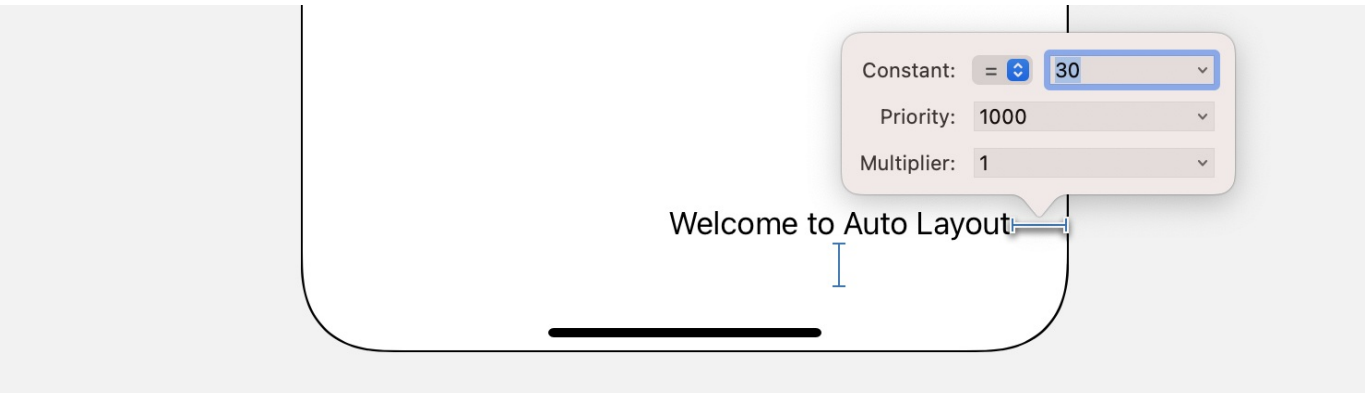


Figure 5-20. Editing a constraint by double clicking the constraint

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 33 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.

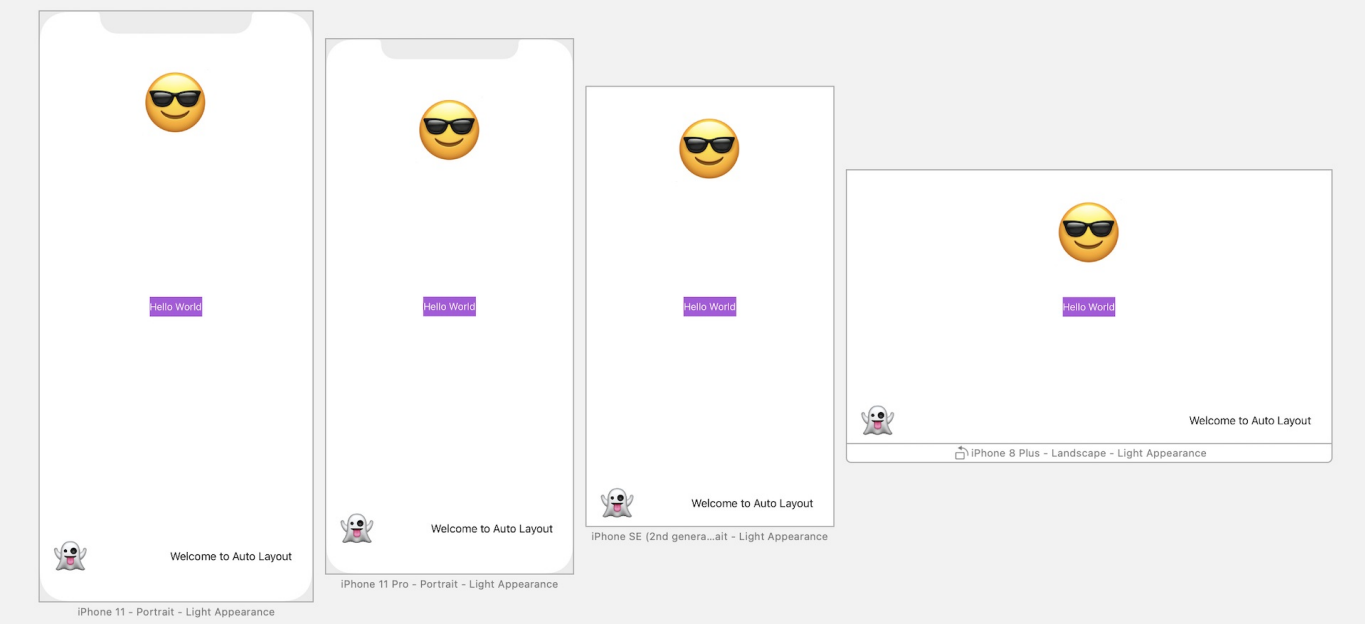


Figure 5-21. The expected app layout

Summary

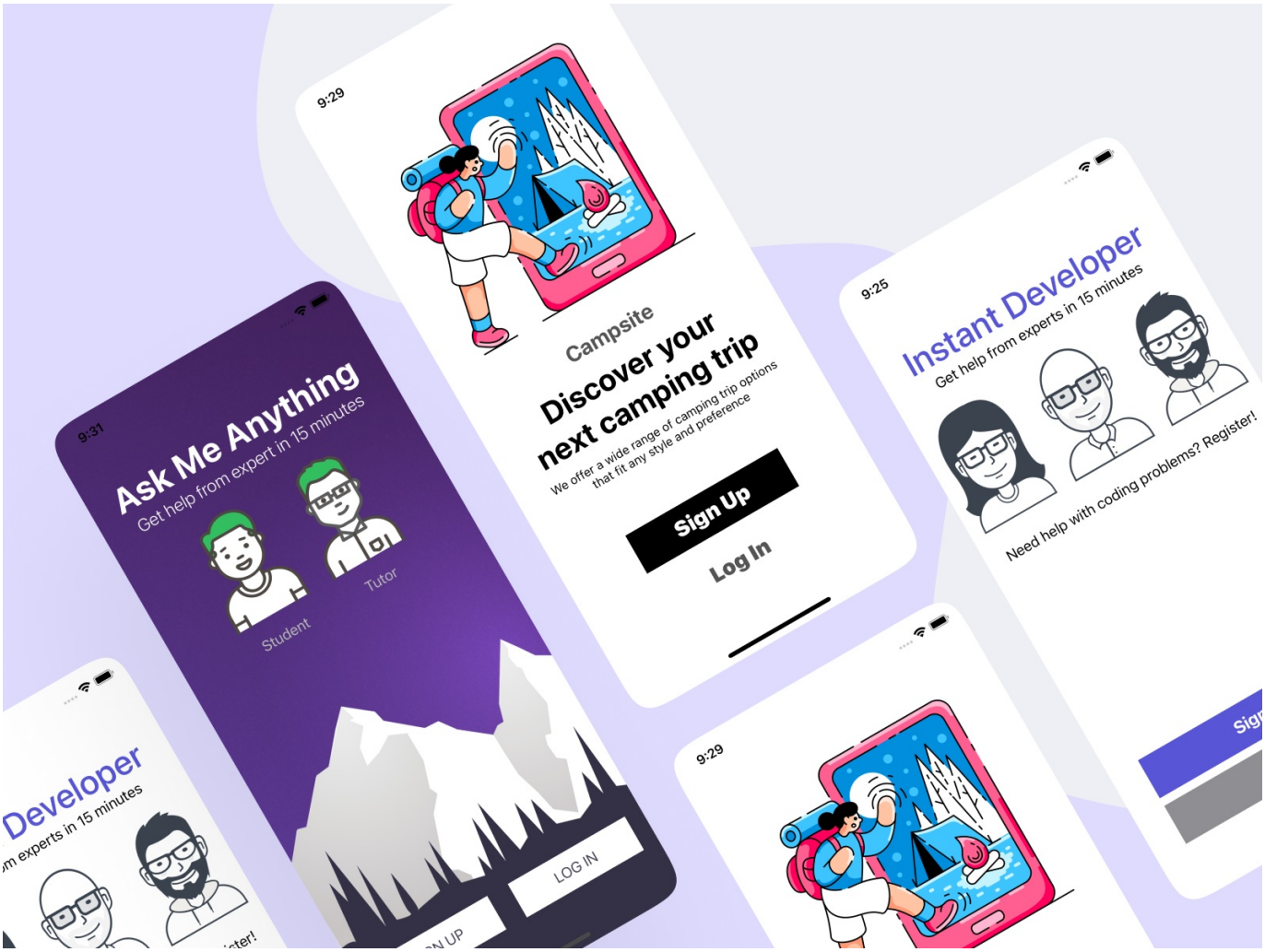
In this chapter, we went through the basics of Auto Layout. It's just the basics because I don't want to scare you away from learning auto layout. As we dig deeper and create a real app, we'll continue to explore some other features of auto layout.

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.

For reference, you can download the complete Xcode project from https://www.appcoda.com/resources/swift53/HelloWorldAutoLayout.zip.

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.

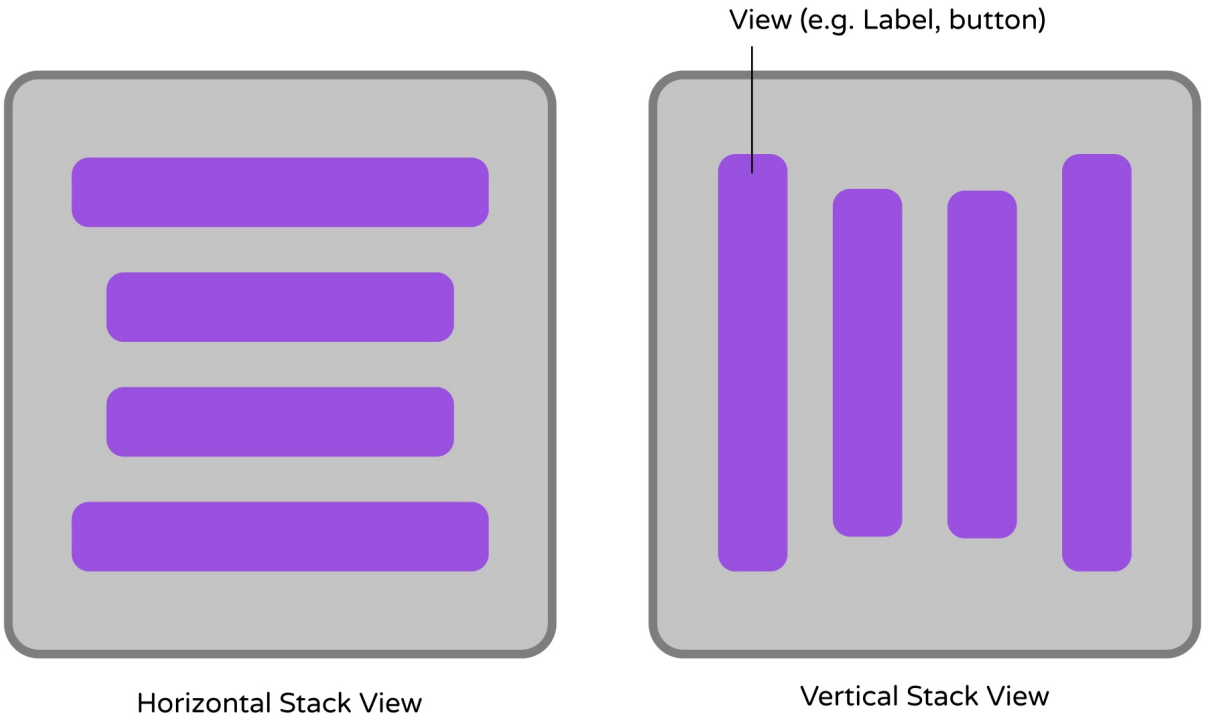


Figure 6-1. Horizontal Stack View (left) / Vertical Stack View (right)

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:

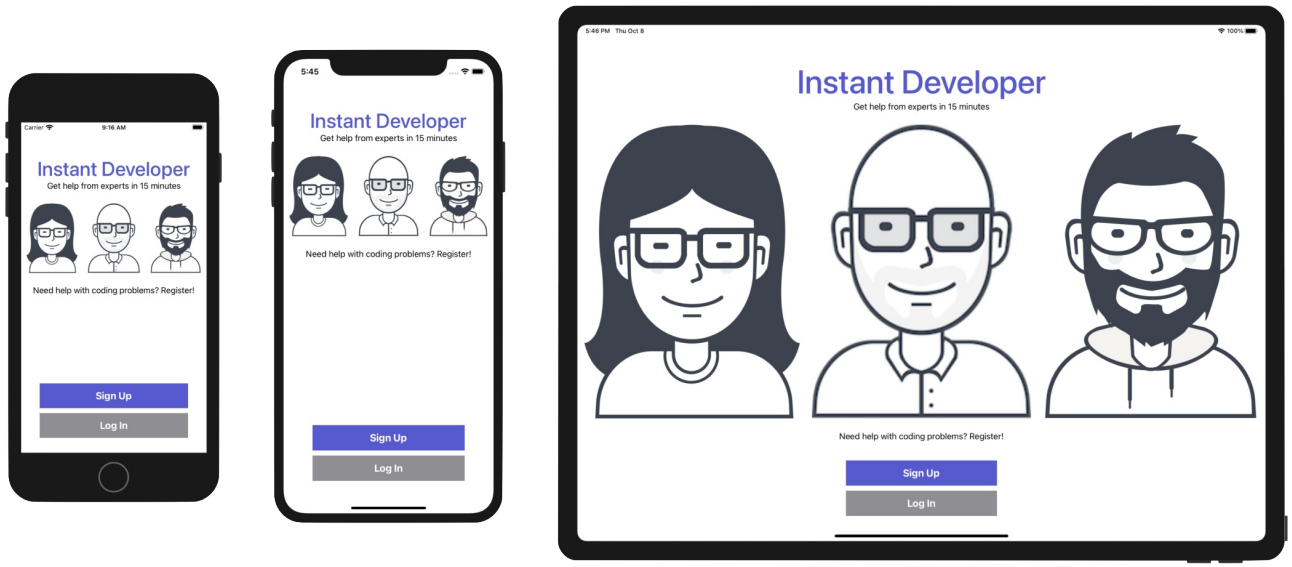


Figure 6-2. The sample app

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 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.
- **Interface: Storyboard** - As explained before, Xcode now supports two ways to build UI. Please change the option to `Storyboard` because this book uses UIKit and Storyboard for UI development.
- **Life Cycle: UIKit App Delegate** - This is the default life cycle option for UIKit development.
- **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 Tests: [unchecked]** – Do not select this option. You do not need any 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.

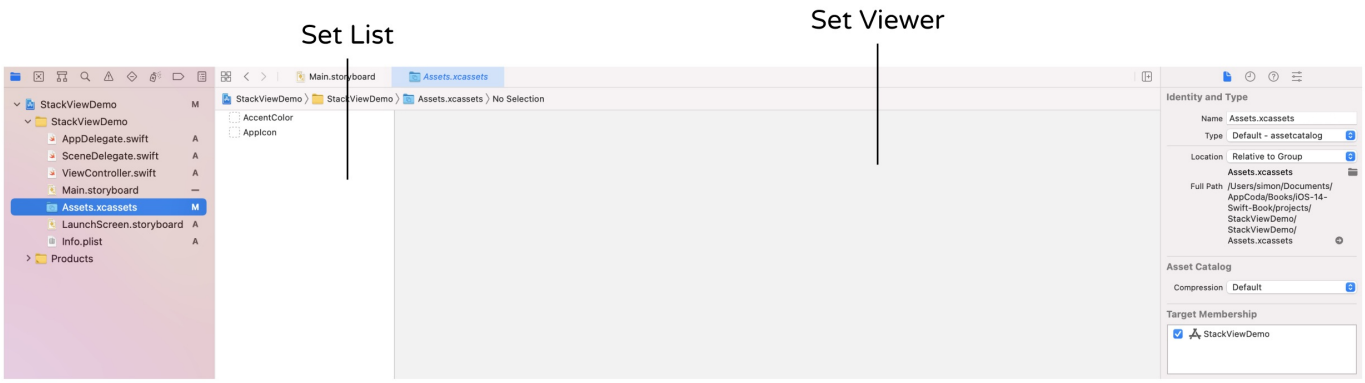


Figure 6-3. Asset Catalog

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.
```

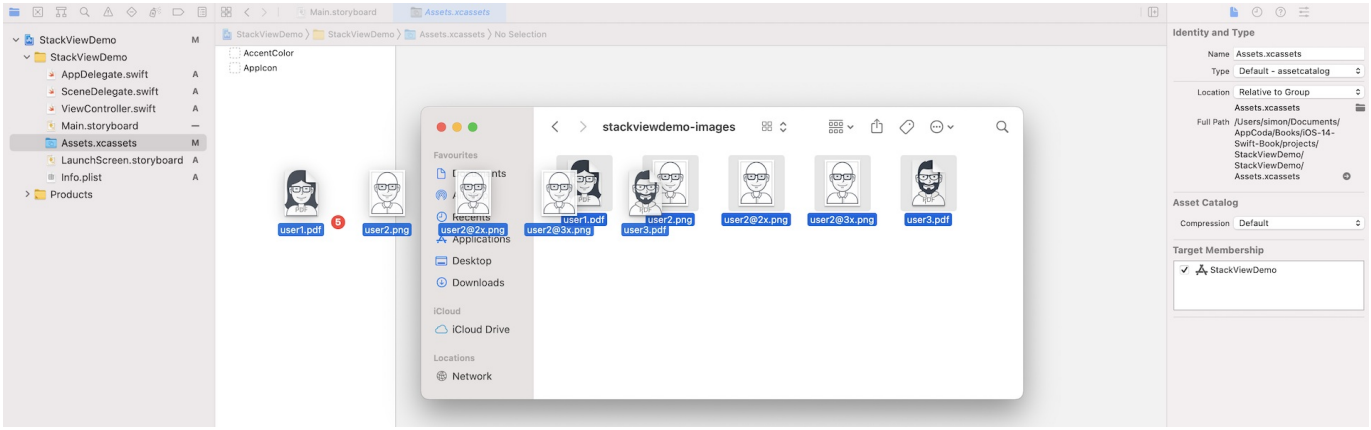
iOS supports two categories of images: *raster images* and *vector images*. Common image formats like PNG and JPEG are classified as raster images. Raster images

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 comes with three versions. The one with @3x suffix, which has the highest resolution, is for iPhone 8 Plus, iPhone 11/12 Pro and Pro Max. The one with @2x suffix is for iPhone SE, iPhone 8, and iPhone 11, while the one without the @ suffix is for older devices with non-Retina display (e.g. iPad 2). For details about how the images are used, you can further refer to this link (https://developer.apple.com/design/human-interface-guidelines/ios/icons-and-images/image-size-and-resolution/).

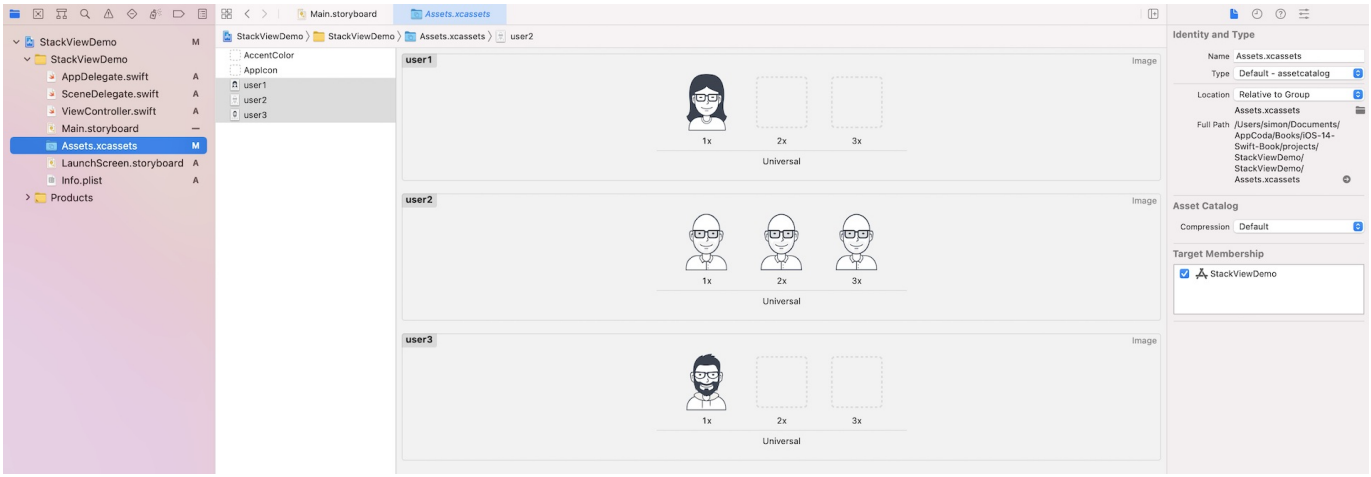
Vector images usually have file types such as PDF and SVG. You can use tools like Sketch and Pixelmator to create vector images. Unlike raster images, vector images are comprised of paths instead of pixels. This allows the images to scale up without losing any image quality. Because of this feature,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



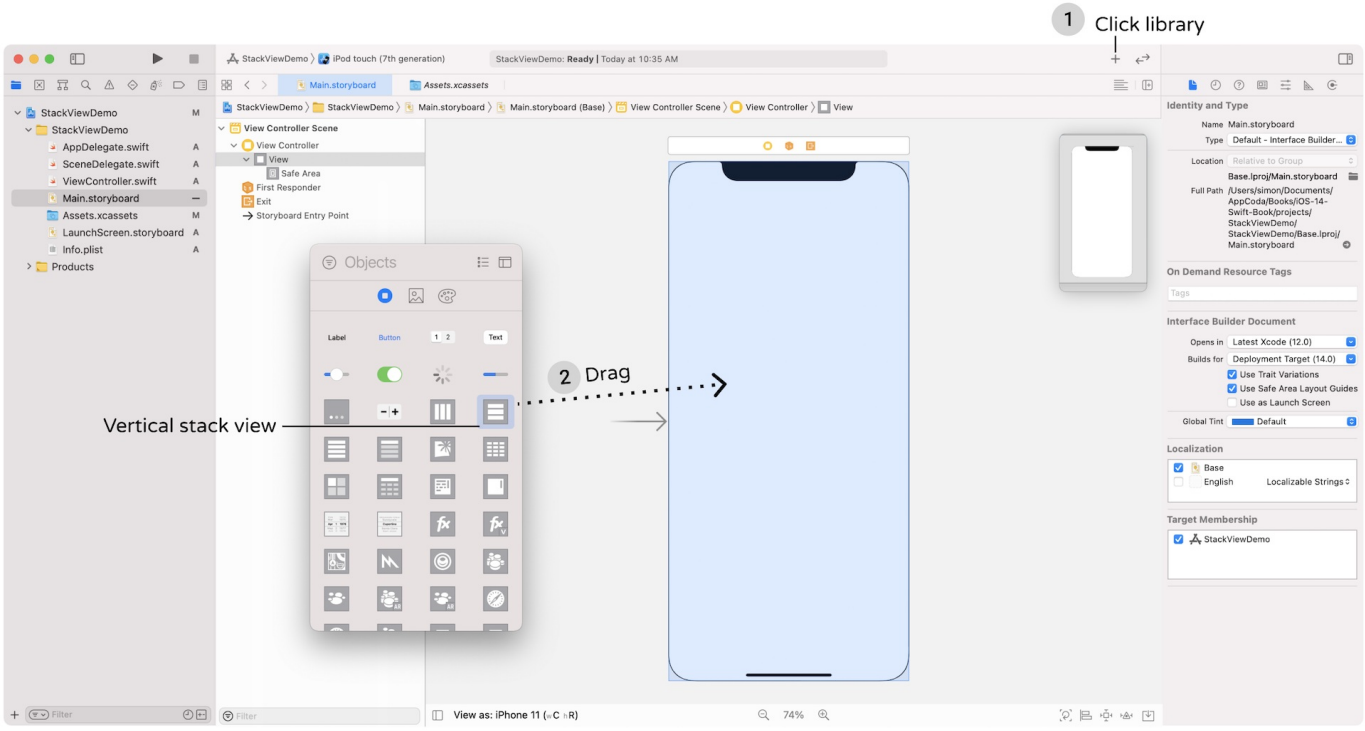
Layout the Title Labels with Stack Views

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.

Instant Developer
Get help from experts in 15 minutes

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.

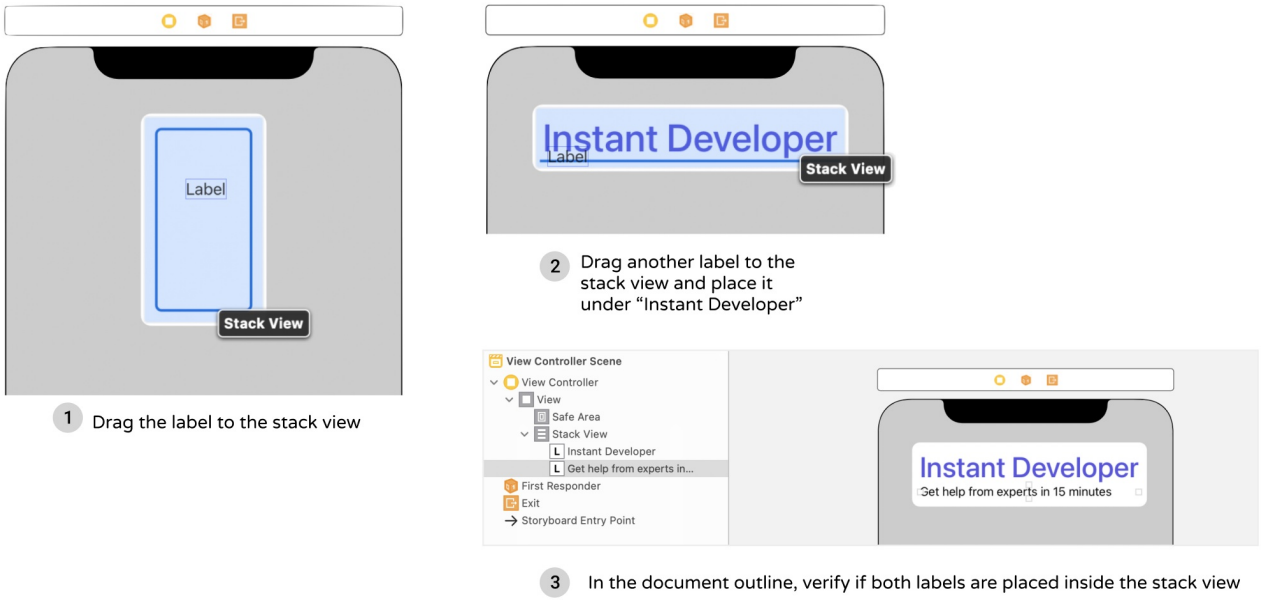
I used to lay out the UI on iPhone 11/12. Actually, you are free to use any device. However, to follow this chapter, please check the *View as* option in the configuration bar. Make sure you are currently using iPhone 11. *Library* button to display the Object library. Drag a Vertical Stack View object 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 *System Indigo Color* or whatever color you like.

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.

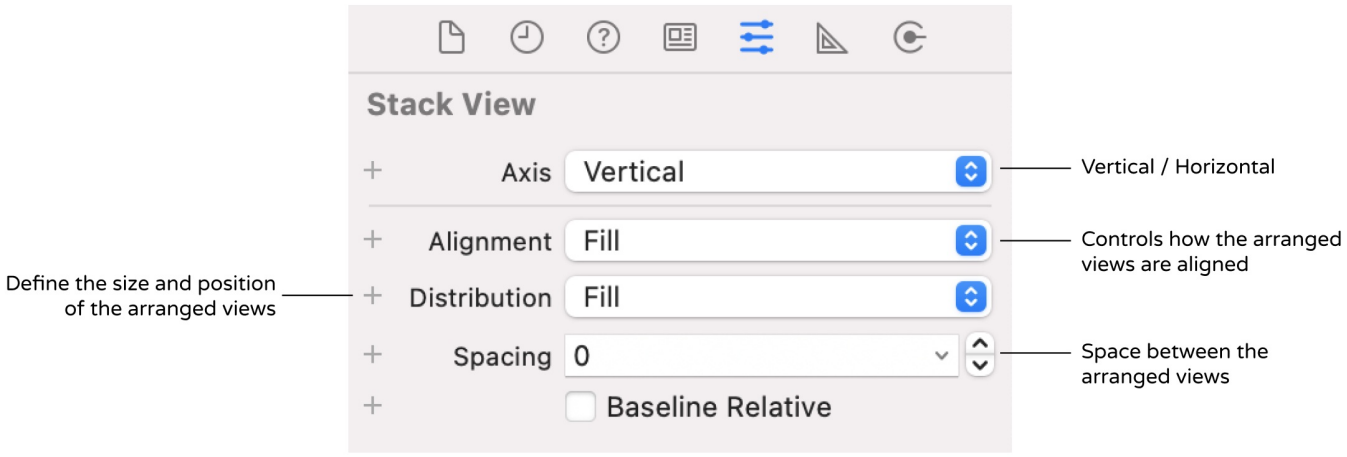


Figure 6-9. Sample properties of a stack view

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.

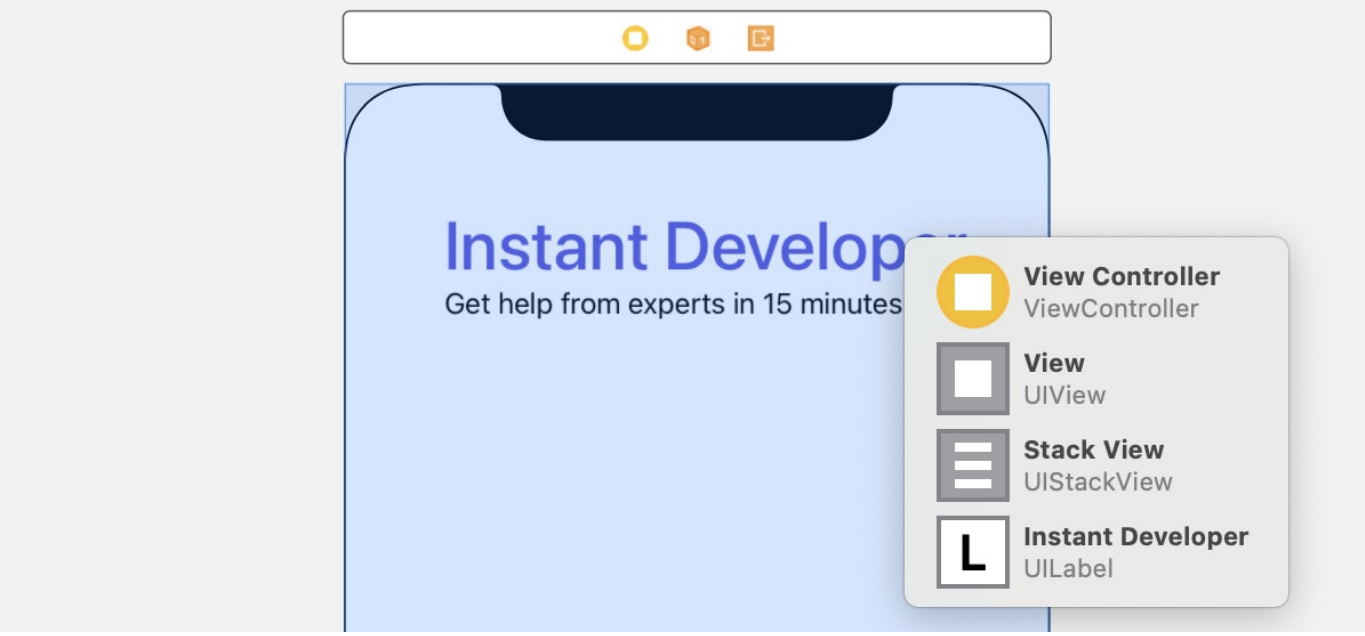


Figure 6-10. Shortcut menu for selection

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 `53` and Y to `70`.

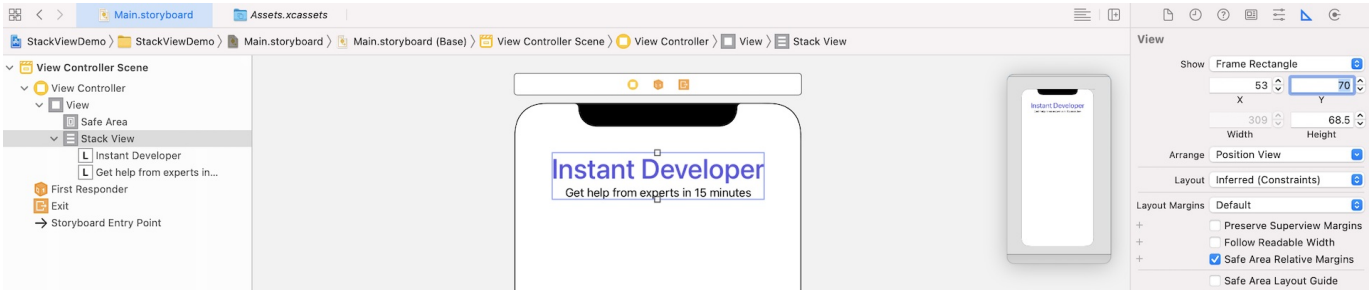


Figure 6-12. Verify the position of the stack view

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`.

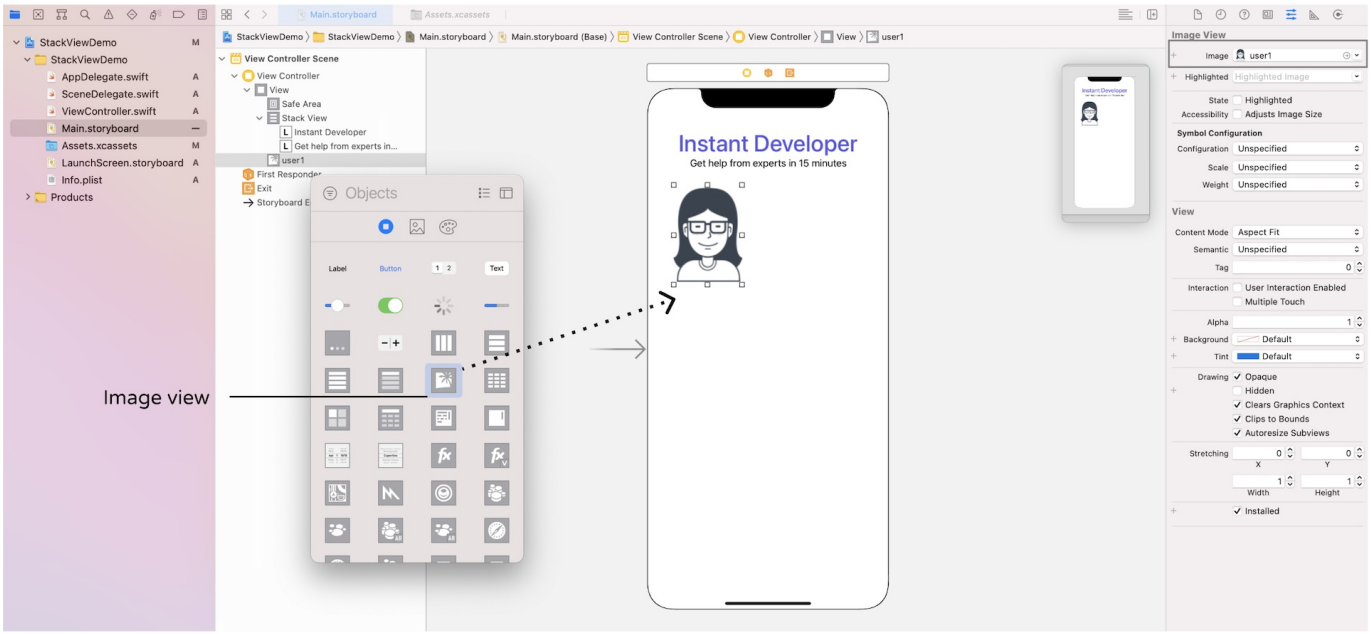


Figure 6-13. Adding an image view for displaying images

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:

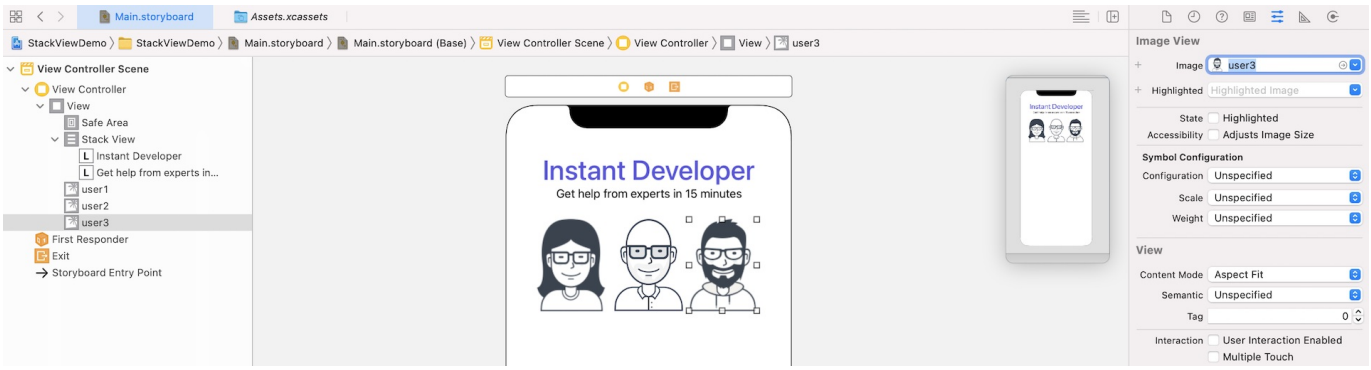


Figure 6-14. Assign the images to the image views

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.

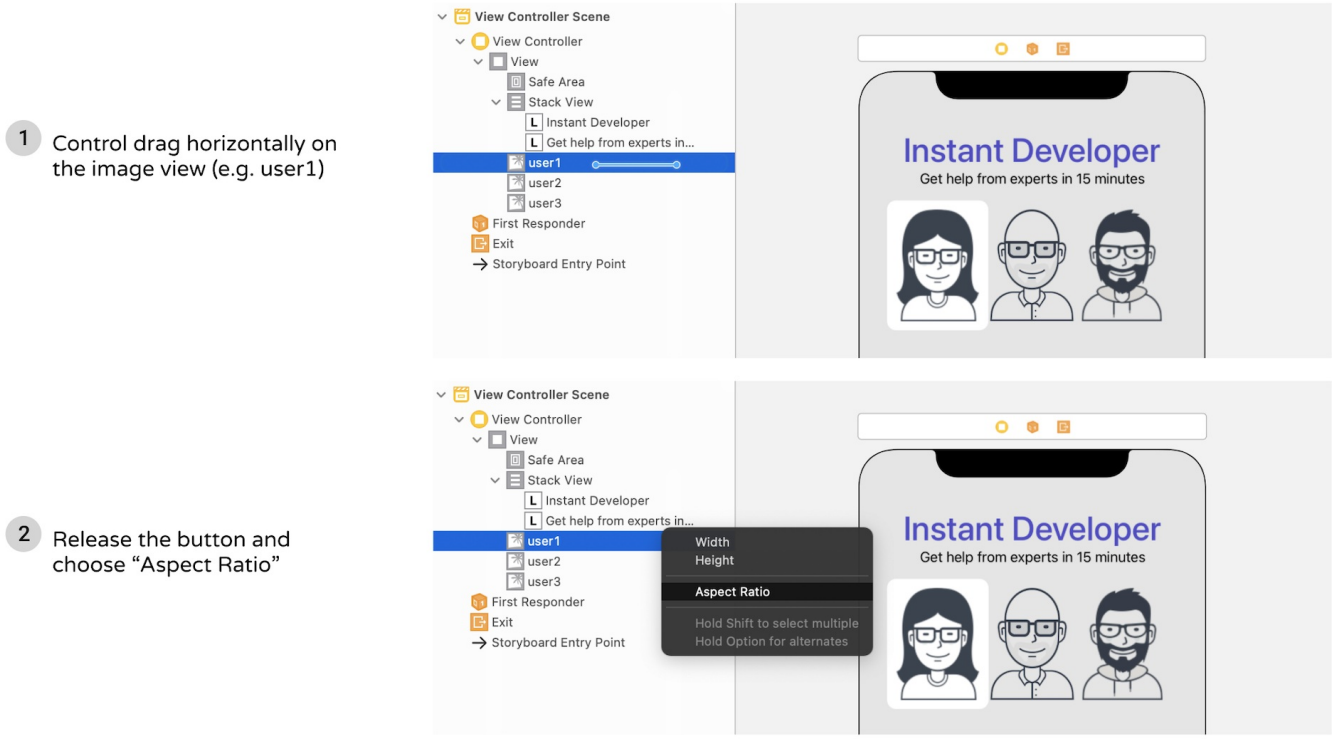


Figure 6-15. Adding an aspect ratio constraint

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 views 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.

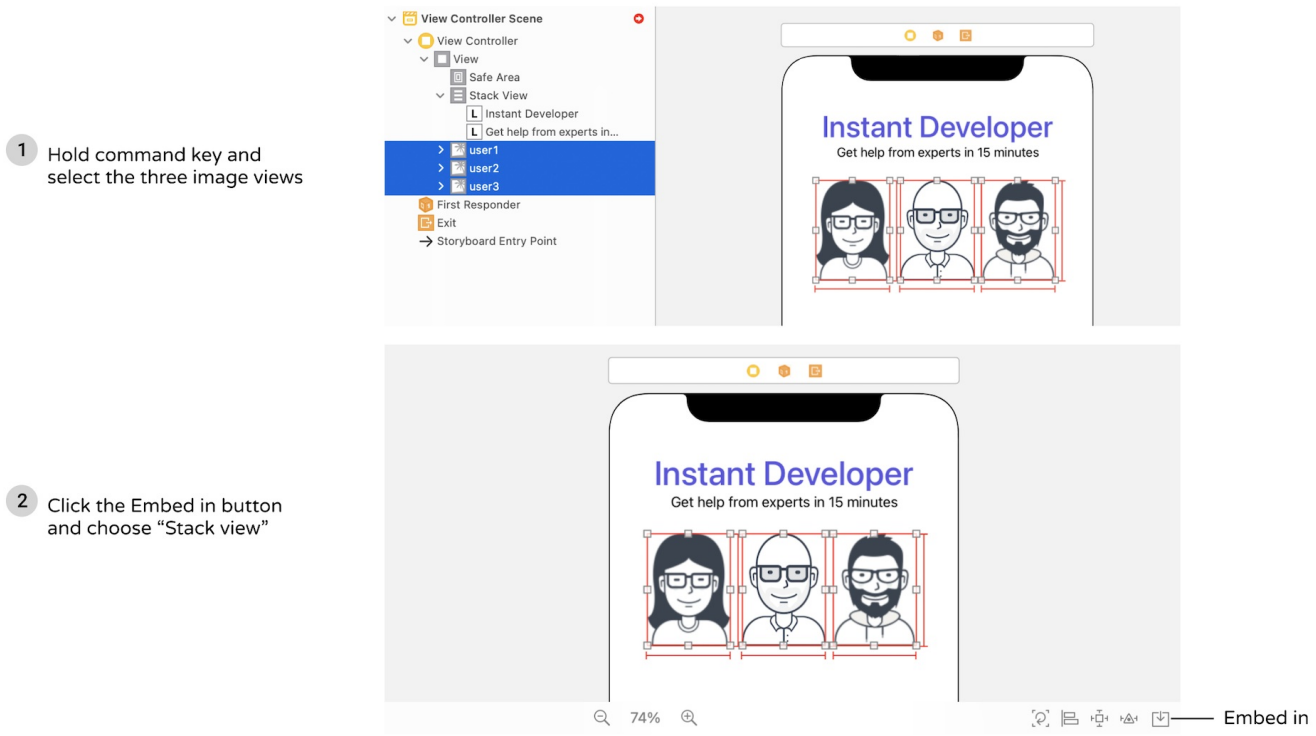


Figure 6-16. Group the image views in a horizontal stack view

To add some spacing between the image views, select the stack view and set the spacing to `20`.

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*. 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.

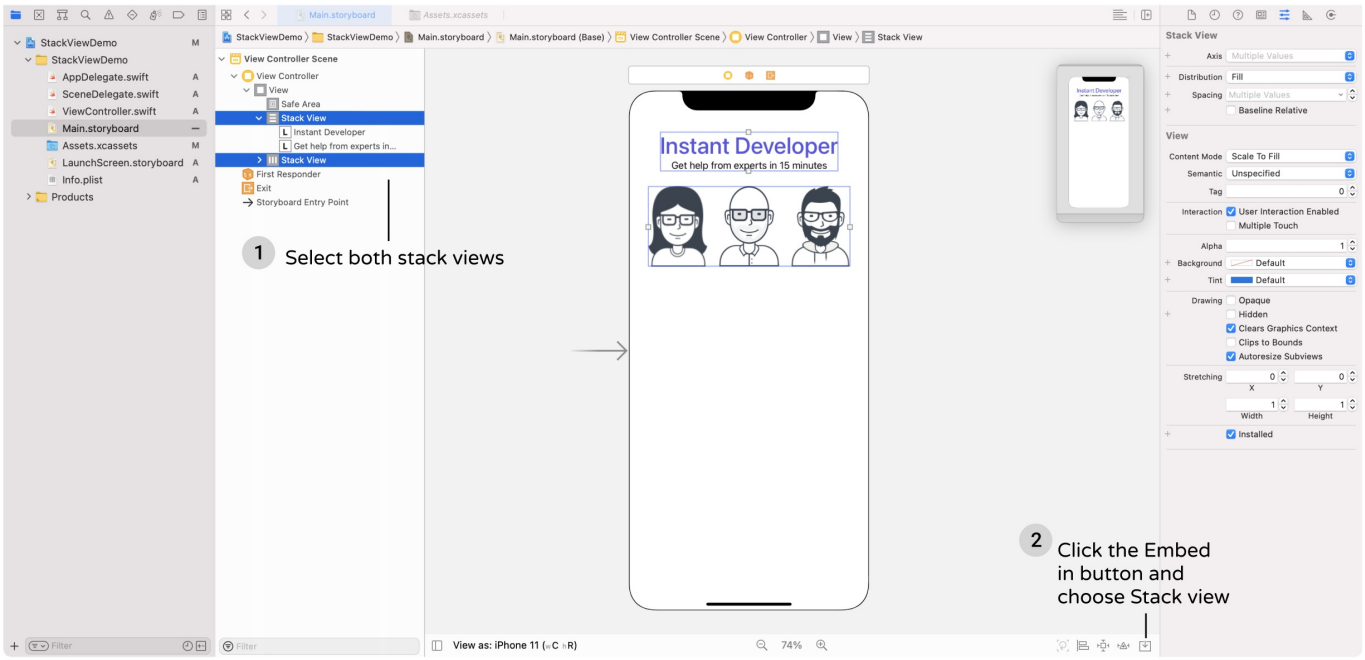


Figure 6-17. Nested stack views

Cool, right? So far the UI looks good on iPhone 11. 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.

Now click the *Add New Constraint* button in the layout bar. Set the space constraints of the top, left and right side to `50` , `10` and `10` respectively. When the constraint is enabled, it is indicated by a solid red bar. Then click the "Add 3 Constraints" button to add the constraints.

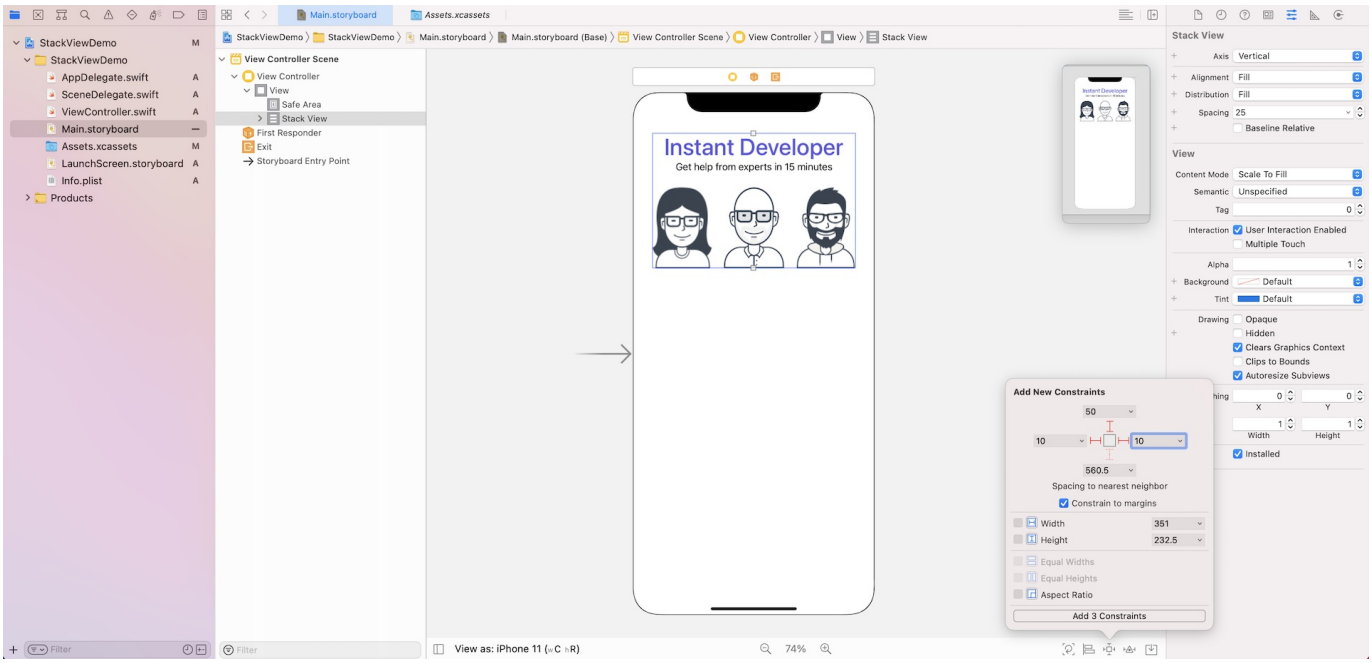


Figure 6-18. Adding multiple constraints using the Pin button

Once you added the constraints, Interface Builder automatically re-positions the stack view to the correct position that conforms to your layout constraints.

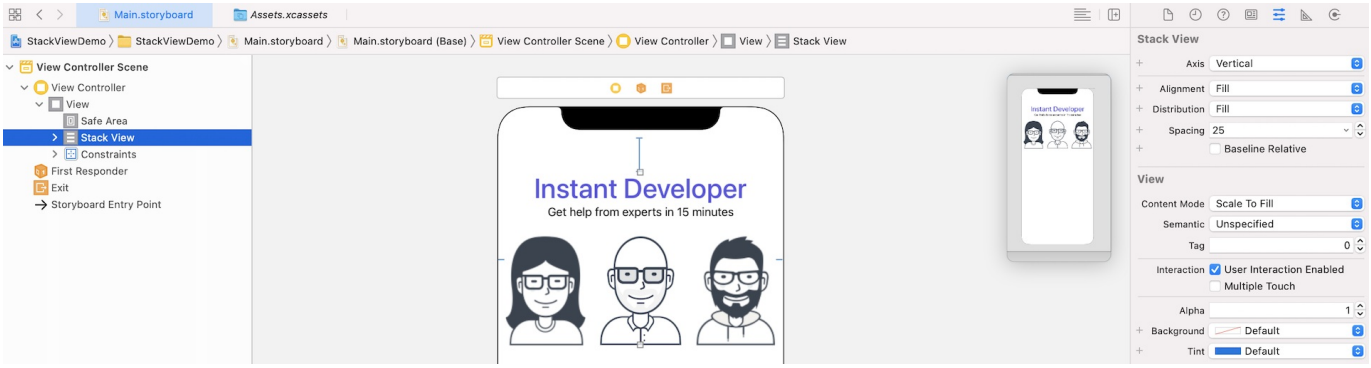


Figure 6-19. The position of the stack view after adding the spacing constraints

Now try to preview the interface on other devices. They should look pretty good. However, you may notice an issue:

- The *Instant Developer* label is truncated for iPhone SE (1st generation).

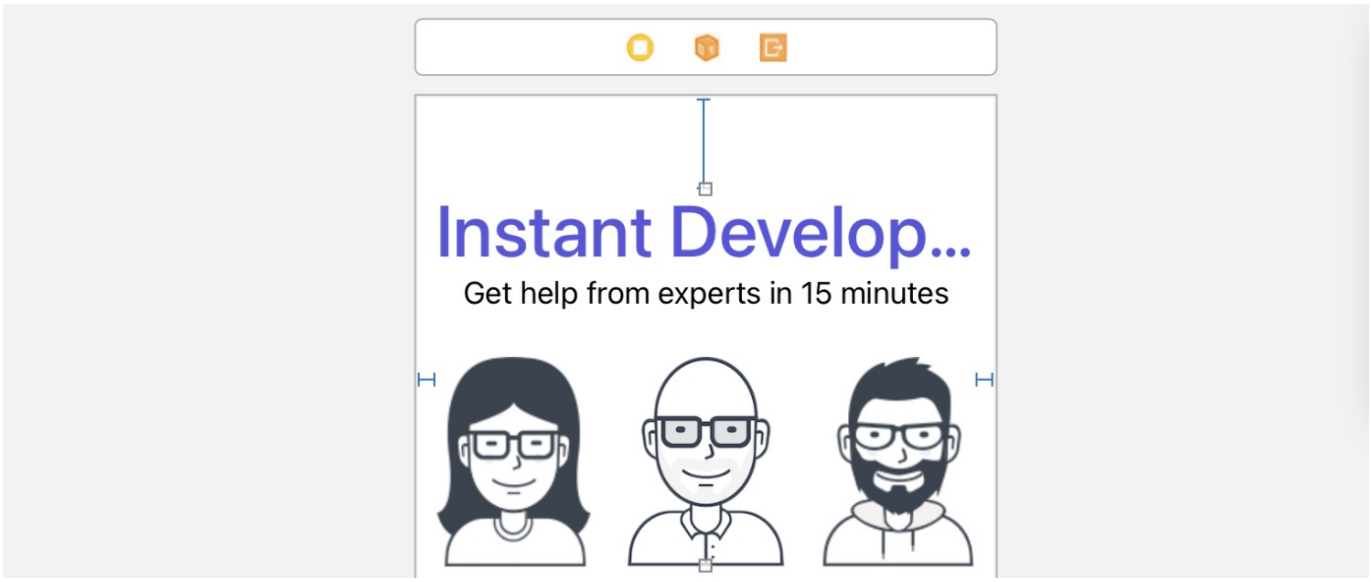


Figure 6-20. Previewing the UI on iPhone SE

So how can we fix the issue?

It is quite obvious that we can decrease the font size of the label. Xcode provides an automatic way to adjust the font size on the fly. Now select the *Instant Developer* label, and go to the Attributes inspector. Set the *Autoshrink* option to *Minimum Font Size*, and the value to `20`.

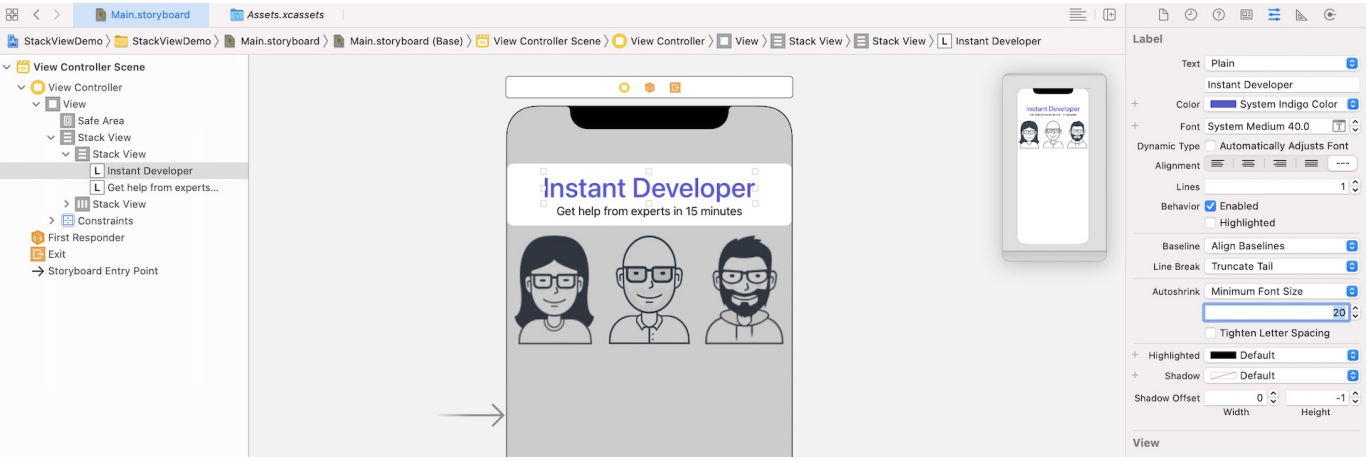


Figure 6-21. Changing the autoshrink option of the label

By doing so, you tell Xcode (or iOS) to determine the suitable font size for the label such that it can perfectly be displayed. Now choose iPhone SE (1st generation) in the configuration bar to test out the UI. The label should be fully displayed.

Adding a Label Below the Images

There is a label right below the images that we haven't added yet. I intentionally left it out so that I can show you how easy you can add an object to an existing stack view.

Open the Object library, drag a label object to the stack view holding the other two stack views. You will see a blue line that indicates the position of the insertion point.

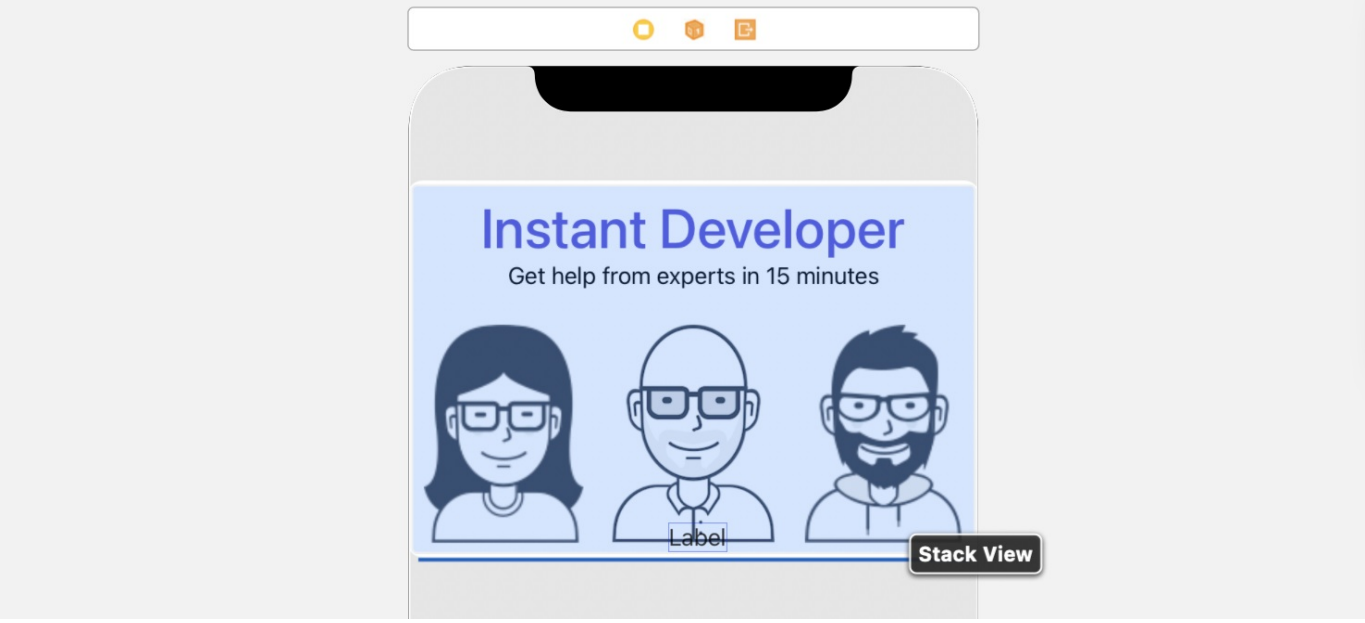


Figure 6-22. Adding a new label to an existing stack view

Next, select the new label. Go to the Attributes inspector, set the text to the following:

Need help with coding problems? Register!

Also, change the text alignment option to `center`.

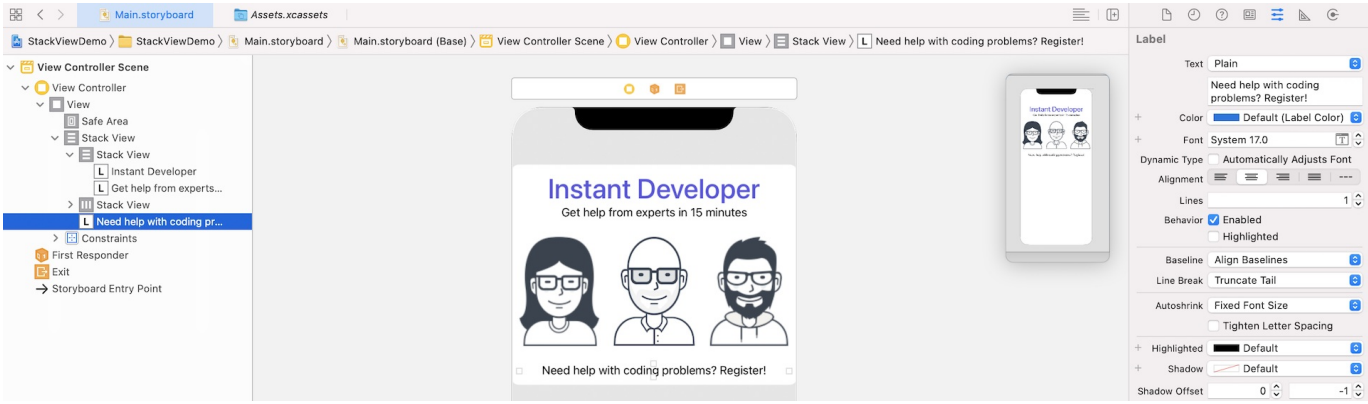


Figure 6-23. Changing the text alignment option to center

Now the UI looks pretty good on all devices. As you can see, stack views save you from defining constraints for the new label. It just inherits the constraints that you have defined earlier.

Layout the Buttons Using a Stack View

We haven’t finished yet. Let's continue to layout the two buttons at the bottom of the screen.

First, click the + button to open the object library and drag a button from the library to the view. Double click the button to name it "Sign up". In the Attributes inspector, change its background color to *System Indigo Color* and text color to *white*. Also, edit the Font option, set its size to `20` points and style to `bold`.

In the Size inspector, set the button's width to `200` and height to `50` points.

Next, let's create the *Log In* button. However, instead of repeating all the procedures. You can select the *Sign Up* button and press command-D to make a duplicate. Then move the new button and place it under the *Sign Up* button. You can change the new button's text to `Log In`. In the Attributes inspector, change its background color to *System Gray Color*. Your UI should now look like that shown in figure 6-24.

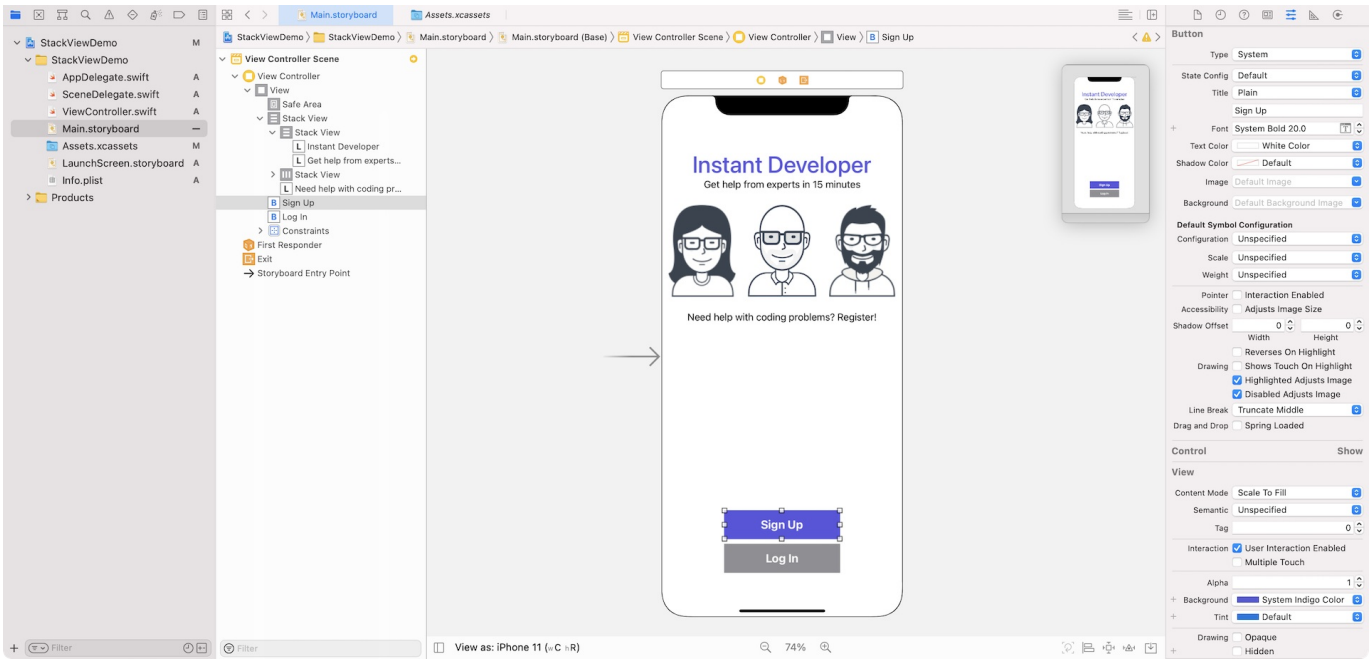


Figure 6-24. Adding two buttons to the view

Again, you do not need to set the layout constraints for these buttons, just let the stack view do the magic for you. Hold the command key and select both buttons. Then click the *Embed in > Stack View* in the layout bar to group them in a vertical stack view. Next, center the stack view horizontally. To add a space between the buttons, select the stack view. In the Attributes inspector, set the value of *spacing* to `10`.

Similarly, we have to define layout constraints for this stack view so that it is positioned close to the bottom of the view. Here are the layout constraints we're going to define:

- Center the stack view horizontally, with respect to the container view.
- Set a spacing constraint so that there is a space between the stack view and the bottom anchor of the safe area.

Select the stack view that you just created. Click the Align button in the layout bar. Check the *Horizontally in Container* option and click *Add 1 Constraint*.

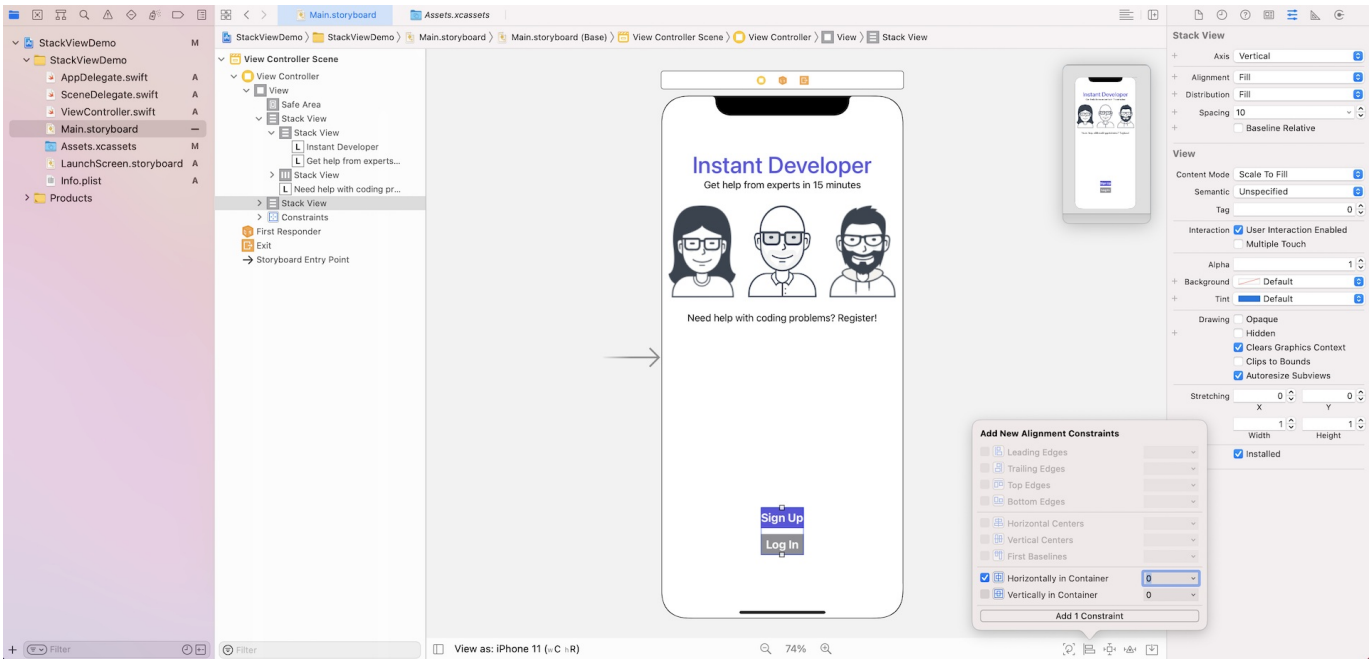


Figure 6-25. Adding a new constraint to center the stack view horizontally

Next, to add a spacing constraint, click the *Add New Constraints* button and set the value of the bottom side to `30`. Make sure the bar is in solid red. Click *Add 1 Constraint* to add the constraint.

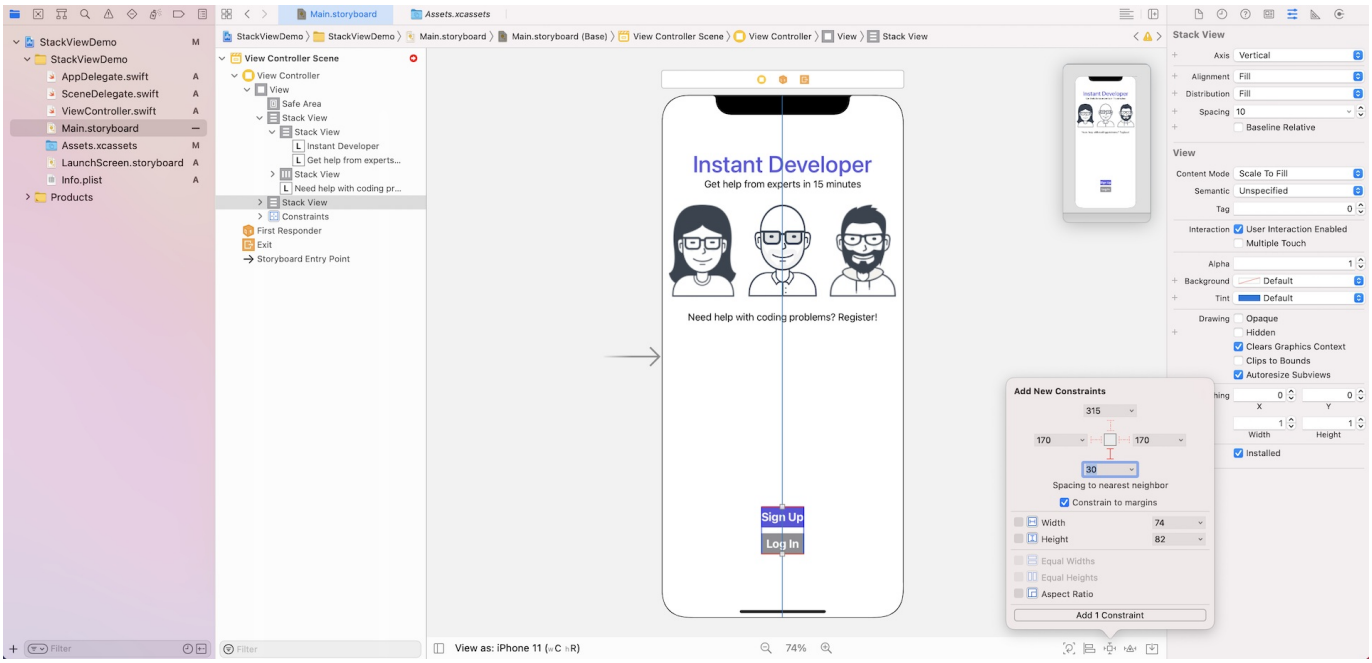


Figure 6-26. Adding a spacing constraint to the stack view

One thing you may notice is that the button is narrower than expected. As you embed the buttons in the stack view, they are resized to its intrinsic size. If you want to set the width to 200, you will need to add the size constraint to the stack view. In the document view, control-drag from the stack view to itself to bring up the context menu. Choose *Width* to add the width constraint.

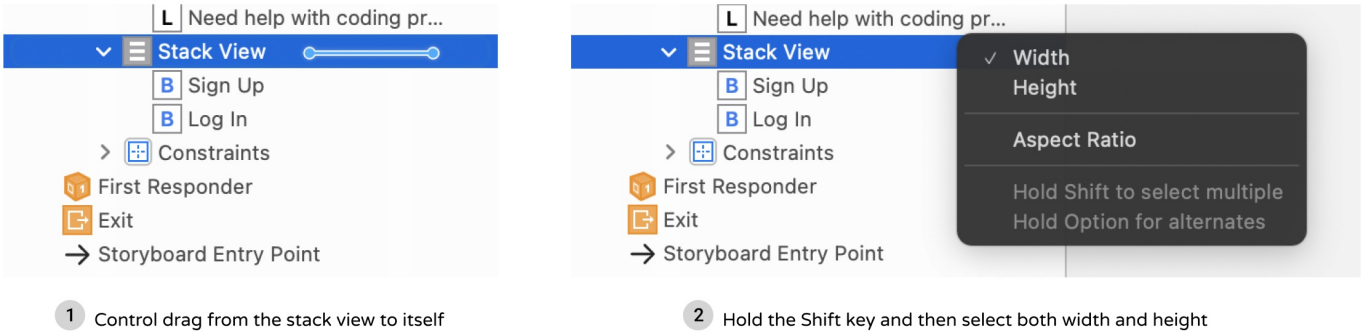


Figure 6-27. Adding the width constraint using control-drag

Once you added the constraints, you should see the constraints in the document outline. Expand it to reveal the width constraint. Since we need to change the width to 200 points, select the width constraint and go to the Attributes inspector. Change the constant value to `200`. The stack view now has a fixed width of 200 points.

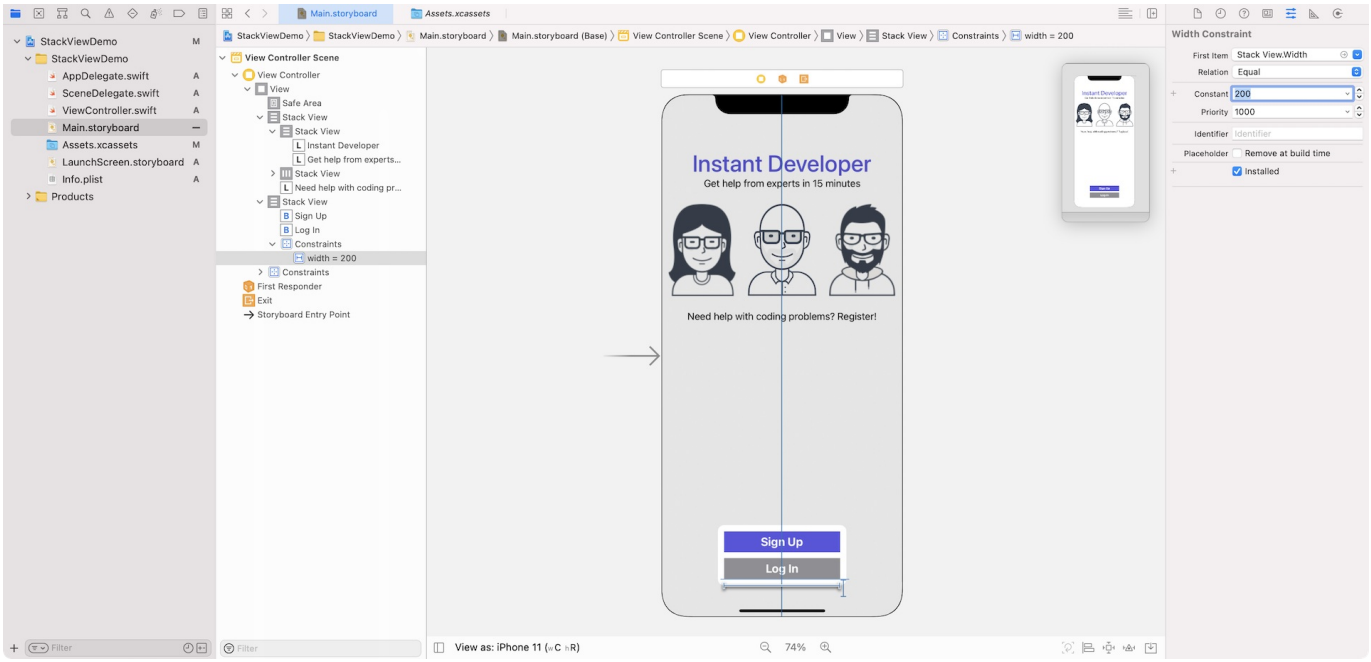


Figure 6-28. Editing the width constraint and change it to 200

Both buttons should have a height of 50 points. As you may know, we need to add a height constraint for both of them. Xcode allows you to add the same constraints to multiple objects at the same time. In this case, you can hold the command key and select both buttons. Then click the *Add new constraints* button and set the height to `50`. Once you confirm to add the constraint, Xcode applies the constraint to both items.

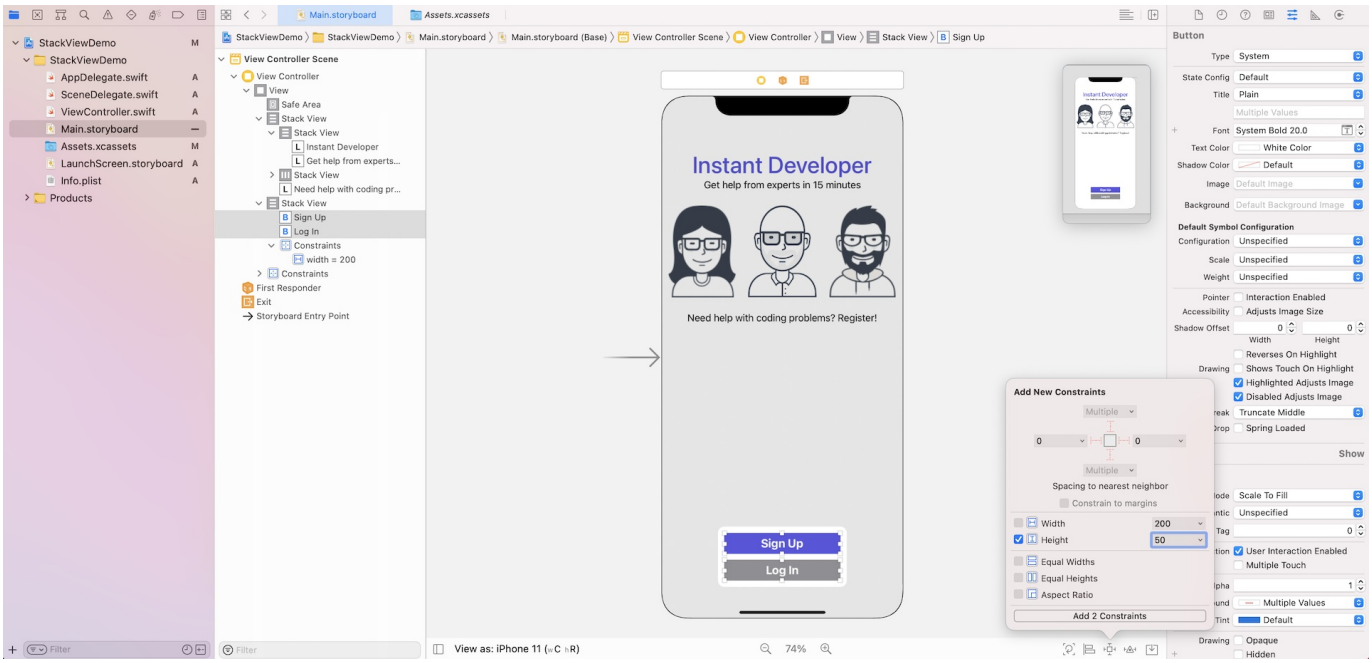


Figure 6-29. Adding the height constraints to both buttons

It's time to test the app again. You can preview the UI in Interface Builder or run the project on different devices. Your UI should look great on all devices.

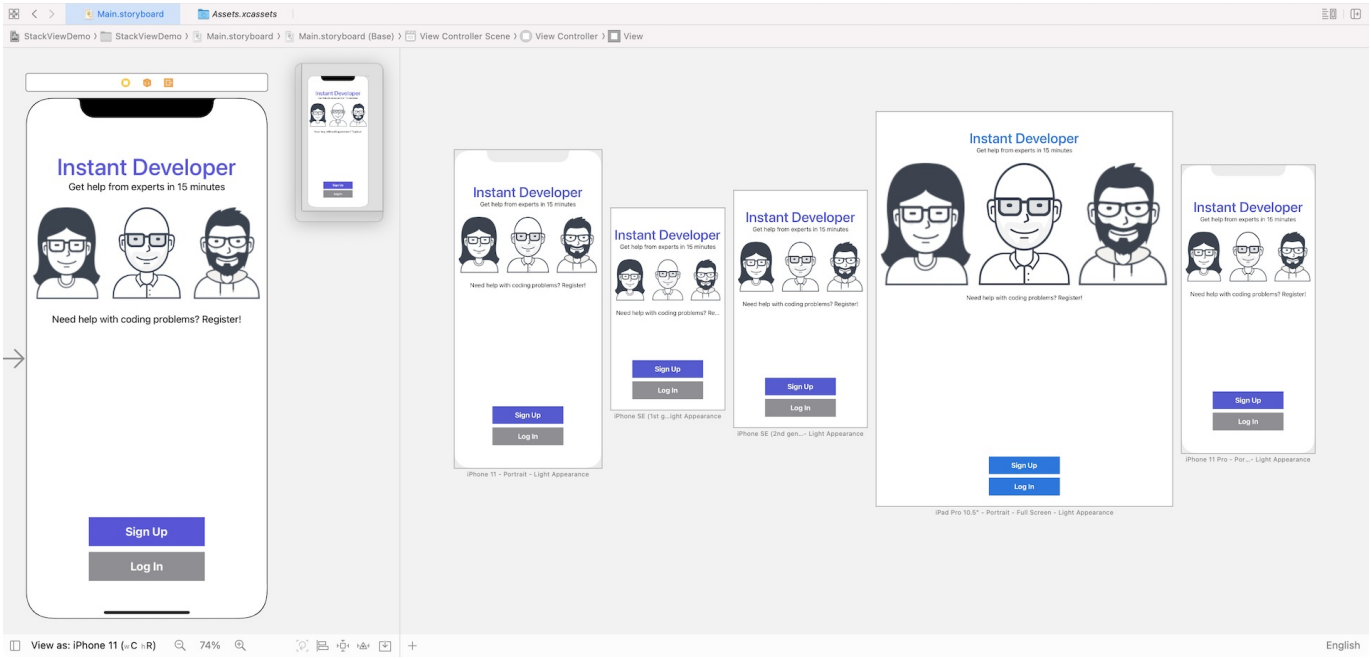


Figure 6-30. Previewing the UI on iPhone and iPad

Do you notice the message "Need help with coding problems? Register!" was truncated on iPhone SE (1st generation)? Try to fix it by yourself. If you don't know how to resolve the issue, go back to the earlier section to find the solution.

You may also notice that the buttons and label color displayed on iPad are different from the rest of the devices. I guess this is a bug in Xcode 12. If you run the app on the iPad simulator, all the buttons and label should have the same color as the original design.

Adapting Stack Views Using Size Classes

Now that you've already built a nice UI using stack views, have you tested the layout in landscape orientation? If you turn the iPhone (or iPad) simulator sideway, the UI in landscape orientation looks like this:

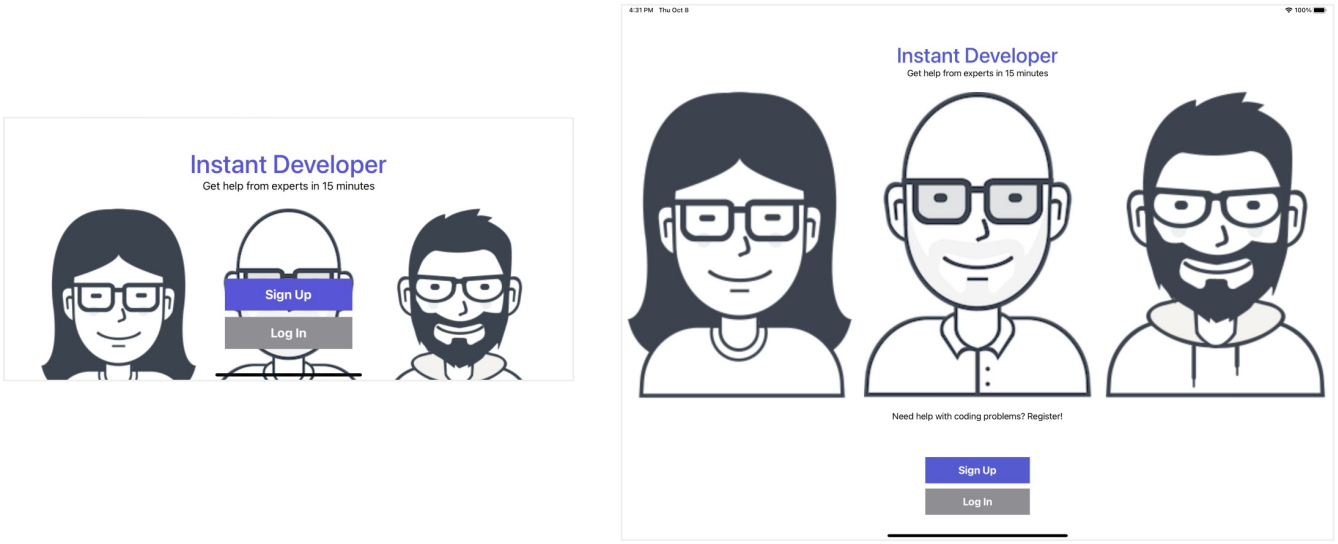


Figure 6-31. The app UI in landscape orientation on iPhone 11 Pro (left) and iPad Pro 12.9-inch (right)

It still looks pretty good on iPad. But it looks a bit weird on iPhone. There are a couple of things I want to customize to make it look better:

1. Move the whole stack view a little bit closer to the top edge of the view.
2. The images are too big and the middle one was overlapped with the buttons.

Please keep in mind that all the changes above are applied to iPhones in landscape orientation only. How can you do that?

This leads to the UI design concept known as Adaptive Layout, introduced since iOS 8. With adaptive layout, your apps can adapt their UI to a particular device and device orientation.

To achieve adaptive layout, Apple introduced a concept called *Size Classes*. This is probably the most important aspect which makes adaptive layout possible. Size classes are an abstraction of how a device is categorized depending on its screen size and orientation. You can use both size classes and auto layout together to design adaptive user interfaces.

A size class identifies a relative amount of display space for both vertical (height) and horizontal (width) dimensions. There are two types of size classes: *regular* and *compact*. A regular size class denotes a large amount of screen space, while a compact size class denotes a smaller amount of screen space.

By describing each display dimension using a size class, this will result in four abstract devices: *Regular width-Regular Height*, *Regular width-Compact Height*, *Compact width-Regular Height* and *Compact width-Compact Height*.

The table below shows the iOS devices and their corresponding size classes:

		Horizontal Size Class (Width)	
		Regular	Compact
Vertical Size Class (Height)	Regular	**iPad Portrait** **iPad Landscape**	**iPhone Portrait**
	Compact	**iPhone 6/7/8 Plus/ XR/XS Max/11/11 Pro Max Landscape**	**iPhone SE/6/7/8/ XS/11 Landscape**

Figure 6-32. Size Classes

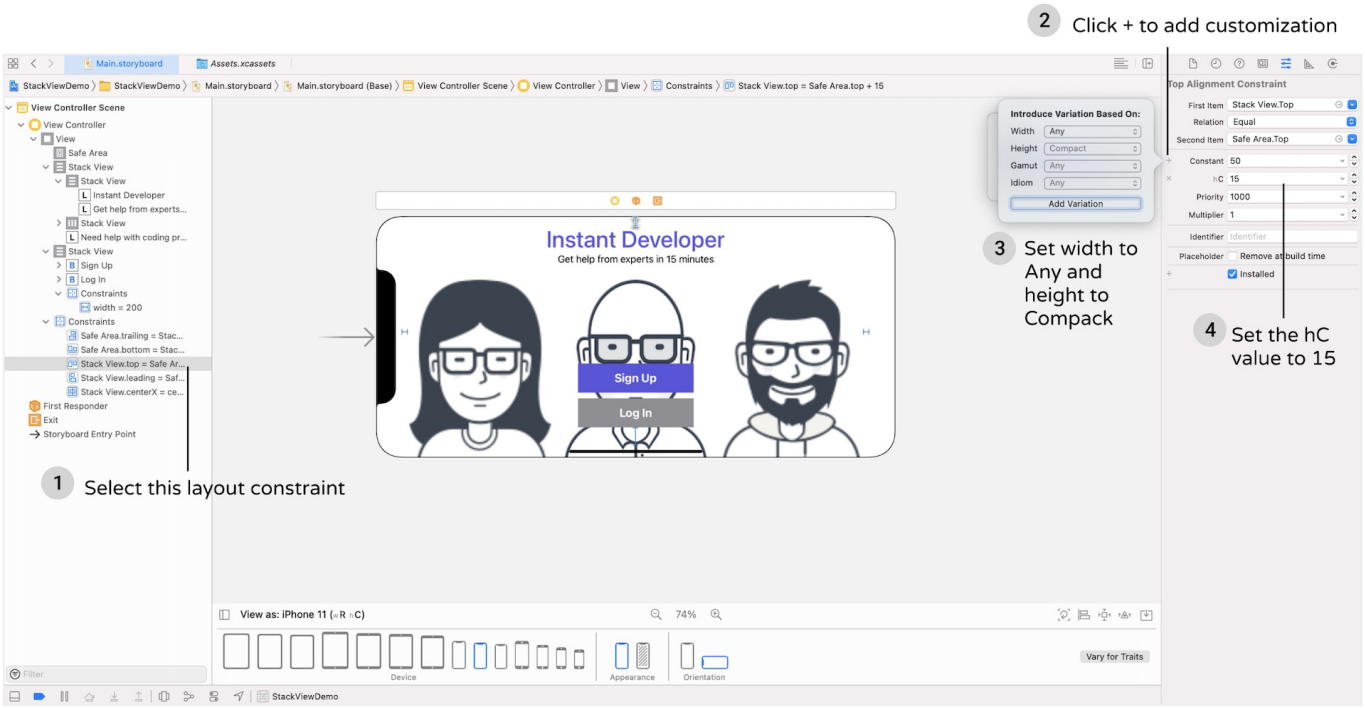
To characterize a display environment, you must specify both a horizontal size class and vertical size class. For instance, an iPad has a regular horizontal (width) size class and a regular vertical (height) size class. For our customization, we want to provide layout specializations for iPhones in landscape orientation. In other words, here are the two size classes we have to deal with:

- Compact width-Compact height
- Regular width-Compact height

You don't need to remember the above table. In Interface Builder, you can find out the current size class in the configuration bar. For example, when iPhone 11 Pro (landscape) is selected, it shows "View as: iPhone 11 Pro (wC hC)". "wC hC" means compact width and compact height.

With an understanding of size classes, now go back to the `Main.storyboard`. In the configuration bar, set the device to landscape orientation. Next, select the spacing constraint between the top anchor of the safe area and the top side of the stack view. In the Attributes inspector, click the `+` button next to the *Constant* option. Select *Any* Width, *Compact* height and *Any* Gamut, and then set the value of this size class to `15`.

Quick note: Here, any width means both compact and regular width.



By doing so, the space between the stack view and top edge of the view will be decreased when the iPhone is turned sideway. Preview the UI in Interface Builder or run the project on various iOS devices to see the result. You should notice that the spacing constraint only reduces its value when the device is an iPhone and in landscape mode.

Now let's see take a look at the second issue, and see how we can reduce the size of the images when iPhone is in landscape orientation.

You are not only allowed to provide customization for constraints. Xcode lets you specify the size-class-specific property for stack views. Select the root stack view, which is the one that holds other stack views.

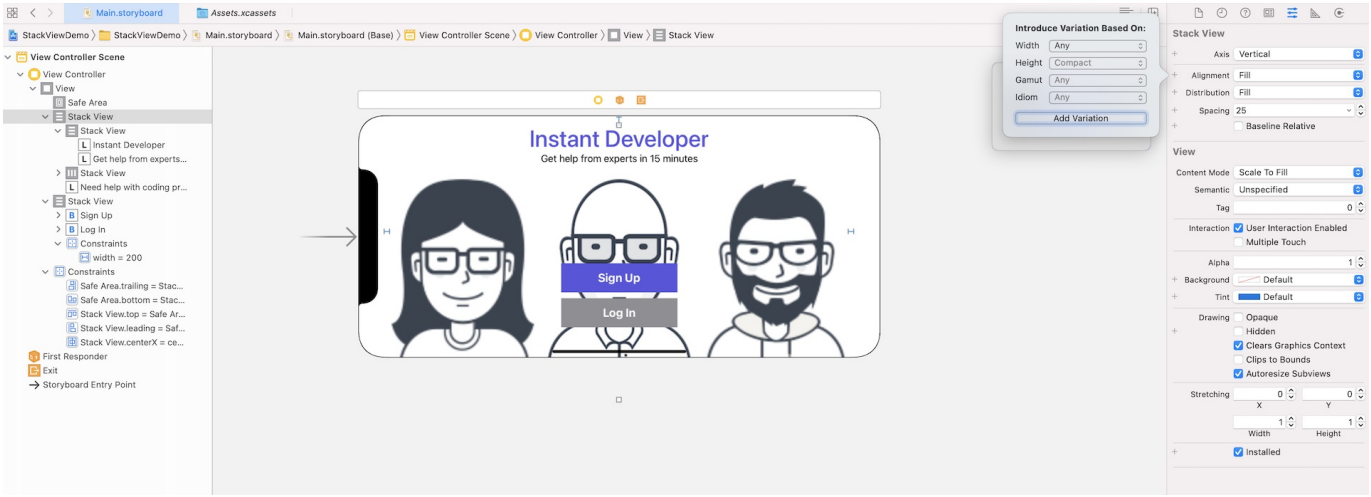


Figure 6-34. Adding a customization for the stack view

In the Attributes inspector, click `+` next to Alignment to add a customization. Choose *Any* width, *Compact* height and *Any* Gamut, then click *Add Variation*. This lets you provide a customization for the alignment. Set the *hC* field to *Center*. In landscape orientation (iPhone only), we will keep the alignment as *Center* so that the image will not resize.

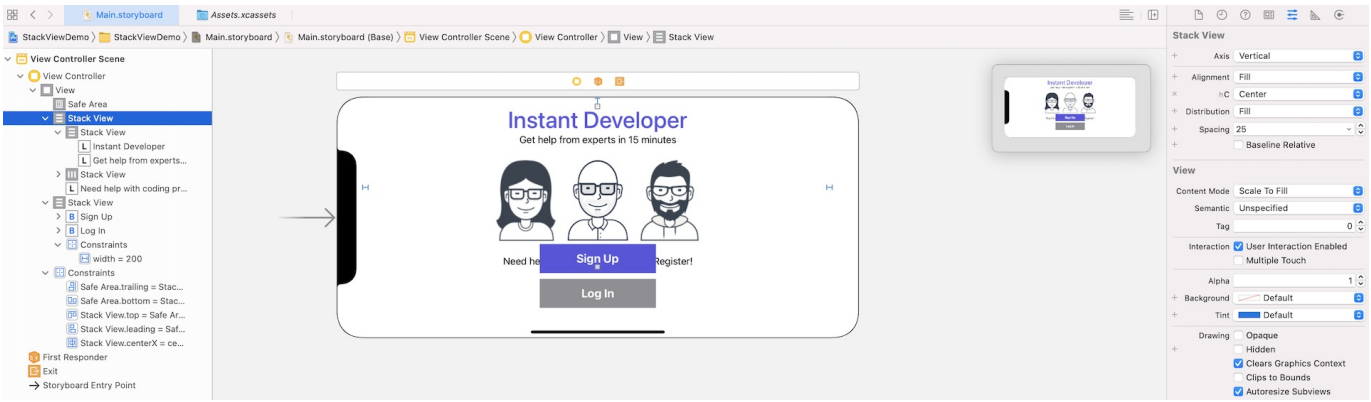


Figure 6-35. After adding customization for alignment

It now look better on iPhone landscape. But there is still an issue with the "Need help with coding problems? Register!" message. It is still covered by the buttons. To resolve the issue, let me show you another trick of using size classes.

Select the stack view which holds both *Sign Up* and *Log In* button. In the Attributes inspector, click the `+` button next to Axis. Choose *Any* width and *Compact* height, and click *Add Variation* to confirm. In the new hC field, set it to `Horizontal`. This means that both buttons should be arranged horizontally instead of vertically for iPhone landscape.

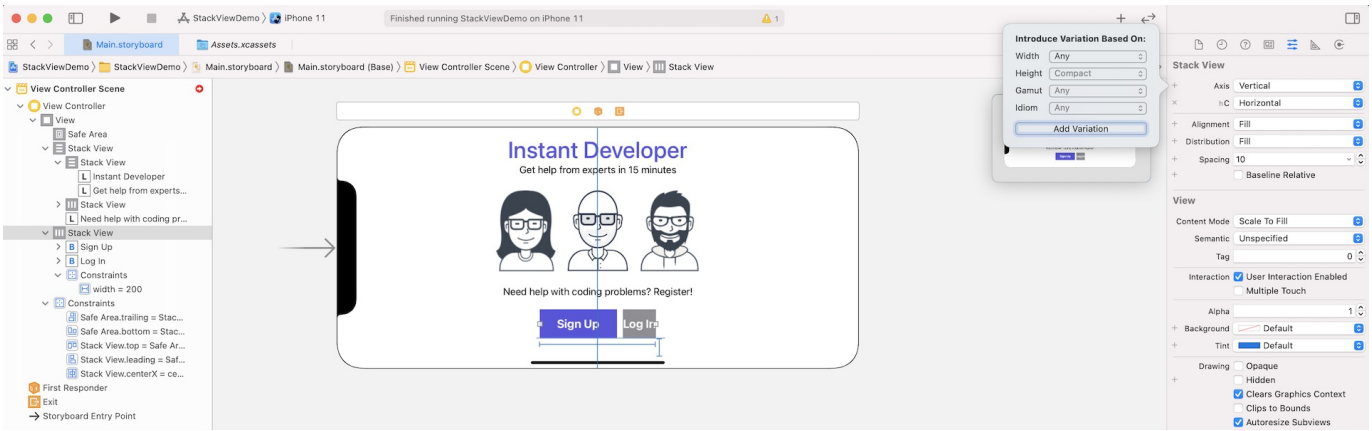


Figure 6-36. Adding layout specialization for the Axis option

Now the "Need help" message is no longer obscured by the buttons. That said, there is still one more thing we have to tweak. Assuming you have the stack view selected, click the `+` button next to Distribution. Again, choose *Any* width and *Compact* height, and click *Add Variation* to add a specialization for the distribution option. In the new hC field, set its value to `Fill Equally`.

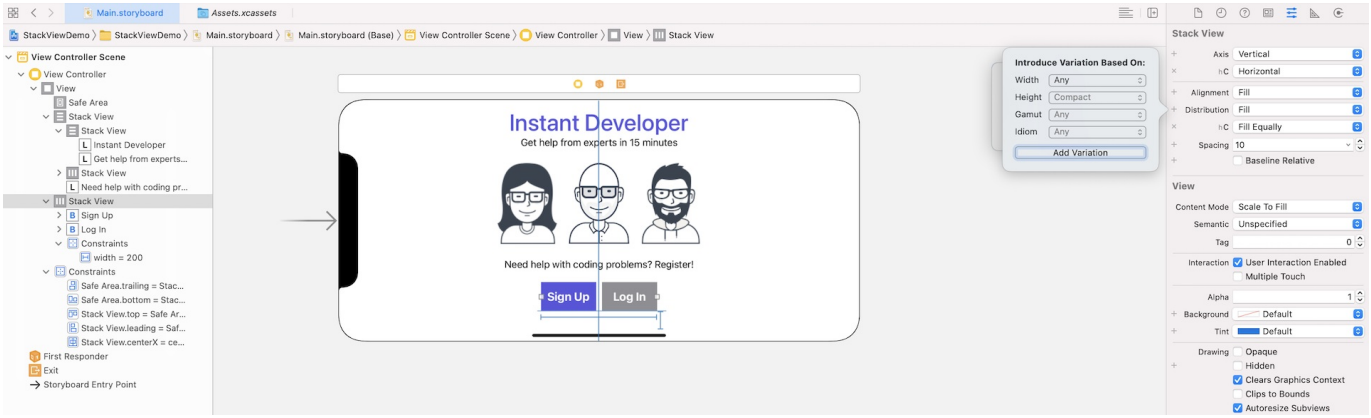


Figure 6-37. Adding a customization for the hidden property

Once you made the change, both buttons are having the same size. That's it! You can run the app on different simulators again. The user interface will be adapted to fit different screen sizes and orientation.

Preserving Vector Data

Before I end this chapter, I want to show you a feature in Xcode known as *Preserve Vector Data*. I mentioned that we prefer to use vector images over raster images in iOS development because they can scale up or down without losing their quality. That's partially true.

When a vector image is used, Xcode automatically converts it into static images (@1x, @2x @3x). It is pretty much like the `user2` image we prepared, but the conversion is handled by Xcode. In this case, the image quality will still be slightly affected when the image is enlarged. If you try to run the demo on iPad Pro (12.9-inch), you should find that the image quality is not perfect.

Starting from version 9, Xcode comes with a feature called *Preserve Vector Data* that lets you preserve the vector data of the images. The option is disabled by default. To enable it, you can go to `Assets.xcassets` and choose one of the images.

In the Attributes inspector, tick the *Preserve Vector Data* checkbox to enable the option.

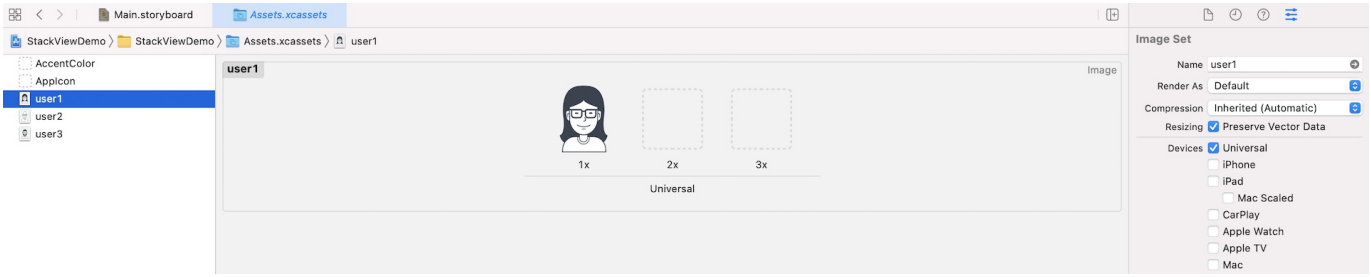


Figure 6-38. Enabled the Preserve Vector Data option

Now if you run the app on iPad Pro (12.9-inch) again, you will find the image looks much better. Figure 6-39 illustrates the image difference when the option is enabled or disabled.

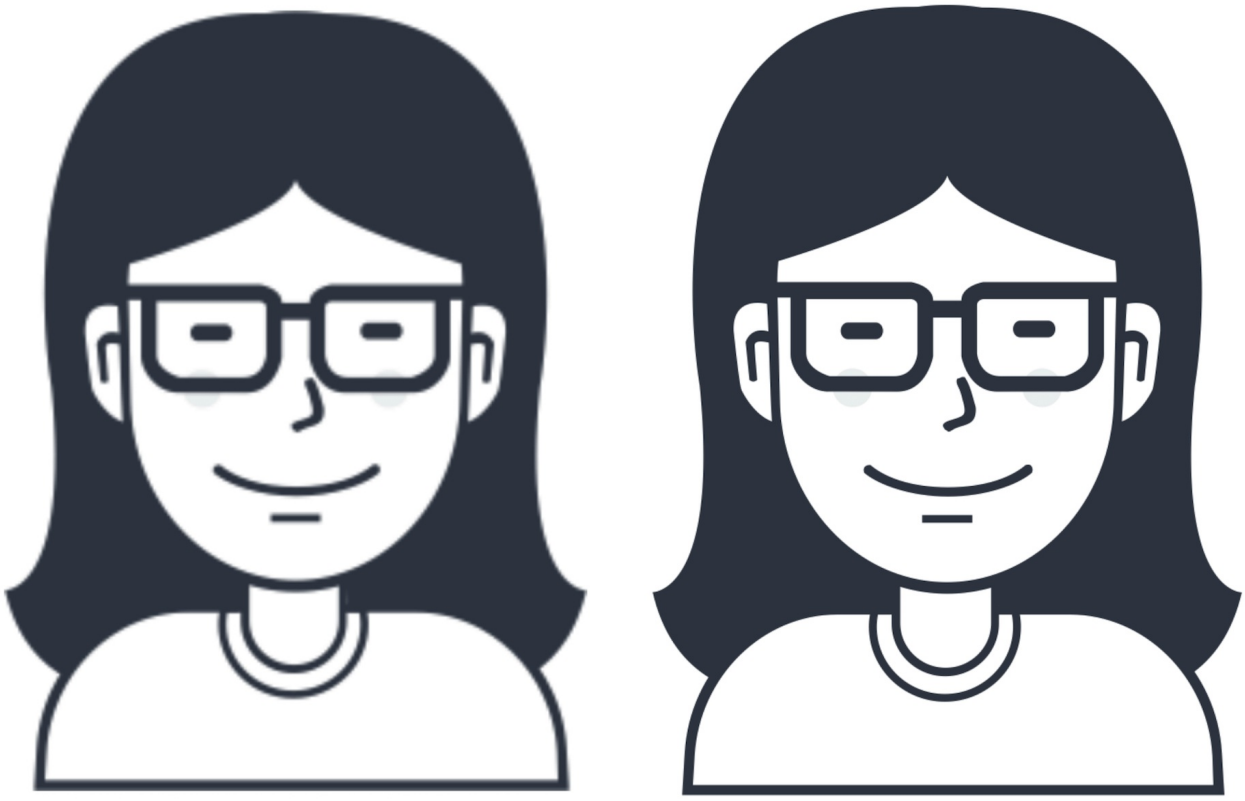


Figure 6-39. (left) with Preserve Vector Data disabled, (right) with Preserve Vector Data enabled

Exercise #1

To help you better understand how auto layout and size classes work, let's have another simple exercise. You are required to update some constraints and create another layout specialization for iPhone (Portrait) and iPad. Here are the requirements:

- For all devices, change the width of the "Sign Up" and "Log in" buttons from 200 to 300 points.
- For iPad, increase the font size of the "Instant Developer" label to 65 points.

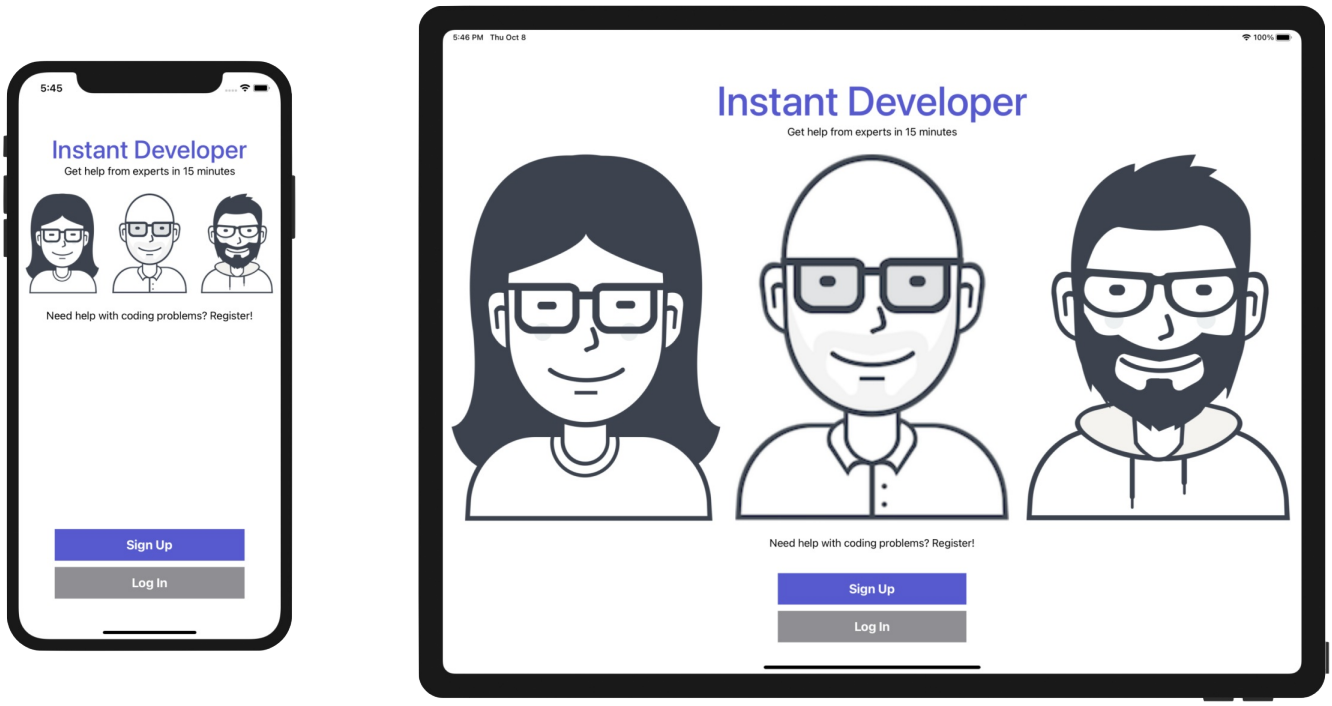


Figure 6-40. UI requirements for exercise #1

Exercise #2

If exercise #1 is too simple for you, exercise #2 is for you. Try to build a UI like the one shown in figure 6-41. You can download the required images from http://www.appcoda.com/resources/swift4/student-tutor.zip

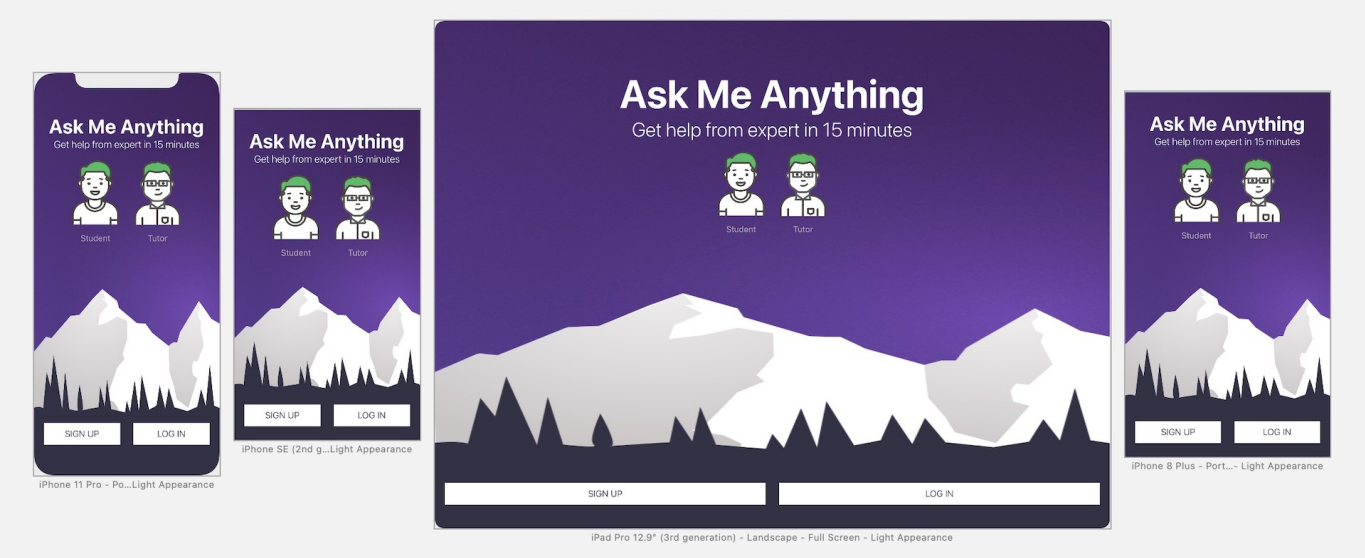


Figure 6-41. UI requirements for exercise #2

Credit: The background image is provided by Luka Dadiani.

Exercise #3

Here is one more exercise for you to practice what you learned. Try to build a UI like the one shown in figure 6-42. You can use any of your photos for building the user interface. There is one feature about the *Label* object that we haven't covered but you need to use it in the exercise. By default, a label is set to display text in a single line. If you need the label to display multiple lines of text, you need to change the value of the *Lines* option. Say, to display two lines of text, you can simply set it to 2.

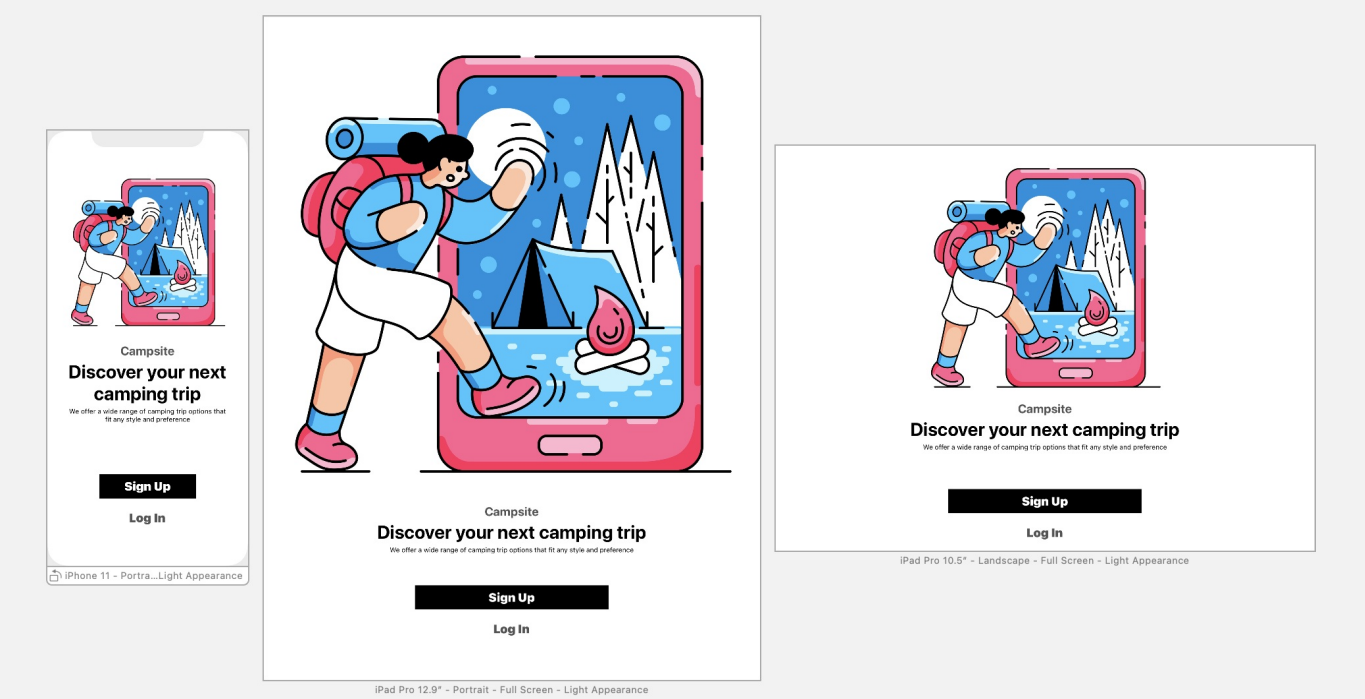


Figure 6-42. UI requirements for exercise #3

Summary

Congratulations! You have finished the chapter and learned how to build an adaptive UI using stack views and size classes.

Stack views streamline the way you build user interfaces on iOS, with very minimal constraints. One question you may have is: when should you use stack views? Apple's engineers recommended developers to adopt stack views first, and then only when you need to actually use raw constraints. So throughout this book, we will design the user interfaces using stack views.

For reference, you can download the complete Xcode project from:

- Complete project and solution to exercise #1: http://www.appcoda.com/resources/swift53/StackViewDemo.zip
- Solution to the exercise #2: http://www.appcoda.com/resources/swift53/StackViewDemoExercise2.zip
- Solution to the exercise #3: http://www.appcoda.com/resources/swift53/StackViewDemoExercise3.zip

Got a question? Join our Facebook group (https://www.facebook.com/groups/appcoda) to discuss with it with other developers.

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.

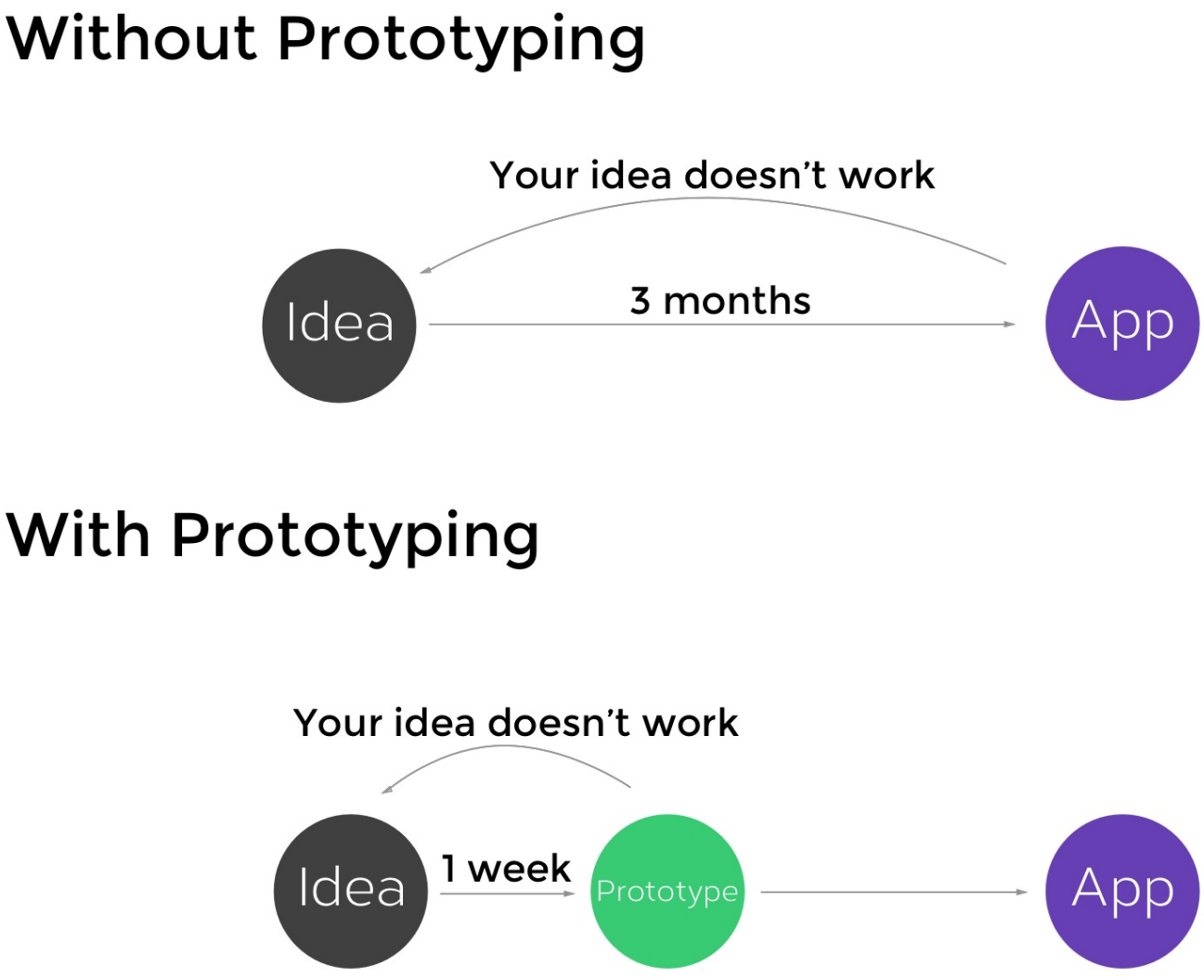


Figure 7-1. Prototyping saves you money and time

Sketching Your App Ideas on Paper

Now that you have an app idea, how can you create a prototype for your app?

A prototype can take many forms. It can be paper-based or digital-based. I always start with hand-drawn concepts and highly recommend you use paper to sketch out your app design. It's the simplest way to create an app prototype. For me, paper sketching is still the best way to quickly record all the ideas in my head. Of course, you can argue that iPad is better for you. Just use whatever works the best for you.

```
Tips: You can find some good printable sheet templates at
http://sneakpeekit.com
```

For example, I have an idea for building a food app that allows me to save my favorite restaurants. While Yelp is good, I want to build an app for me to create a personal restaurant guide. The app will have these features:

- List the favorite restaurant at home screen
- Create a restaurant record and import a photo from photo album as the restaurant image
- Save the restaurant locally and share it with other foodies in the world
- Show the location of a restaurant on maps
- View restaurants shared by other foodies

I think people may like this idea too. What I used to do is take out a piece of paper and draw my design. Some people said they're not good at drawing. You don't have to be an artist to draw your app design. My drawing did not look good either (see figure 7.2) but the point is to visualize your app idea and figure out the basic structure of your app.

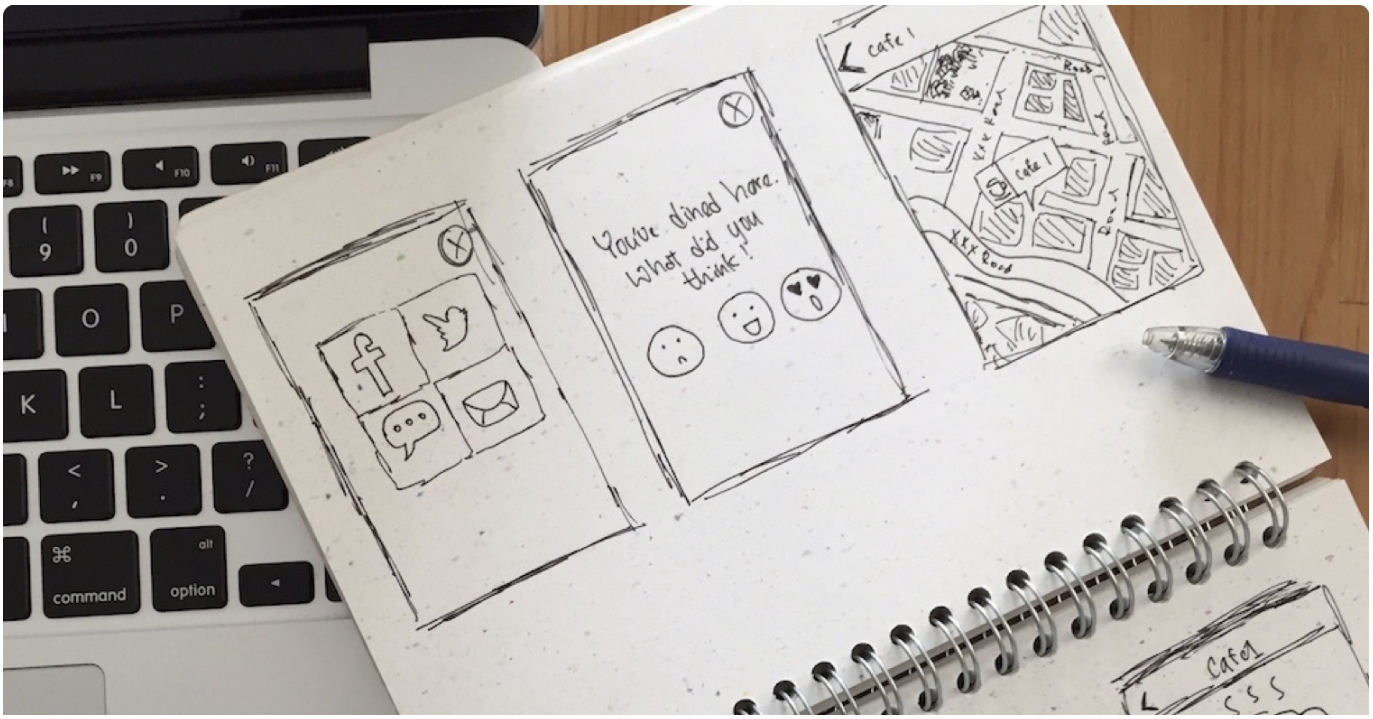


Figure 7-2. Draw your app on paper

Wireframing Your App

You ofter hear the term "wireframe". What I just demonstrated is a way to wireframe an app. The purpose of wireframing is not in the details. We focus on visualizing the structure and organization of the app without concerning about the color, graphics, and visual design. You can think of wireframes the foundation of your app. It lets you better understand the features you want to build and the navigation flow of the app.

Okay, you don't want to draw by hand. Are there any tools you can use to wireframe your mobile app?

You use boxes, circles, and lines to wireframe an app. So basically, you can use whatever drawing tool you prefer to create the wireframe. For me, I use Sketch, which I will further introduce to you in the later section. Figure 7-3 shows a sample wireframe of a mobile app created using the software.

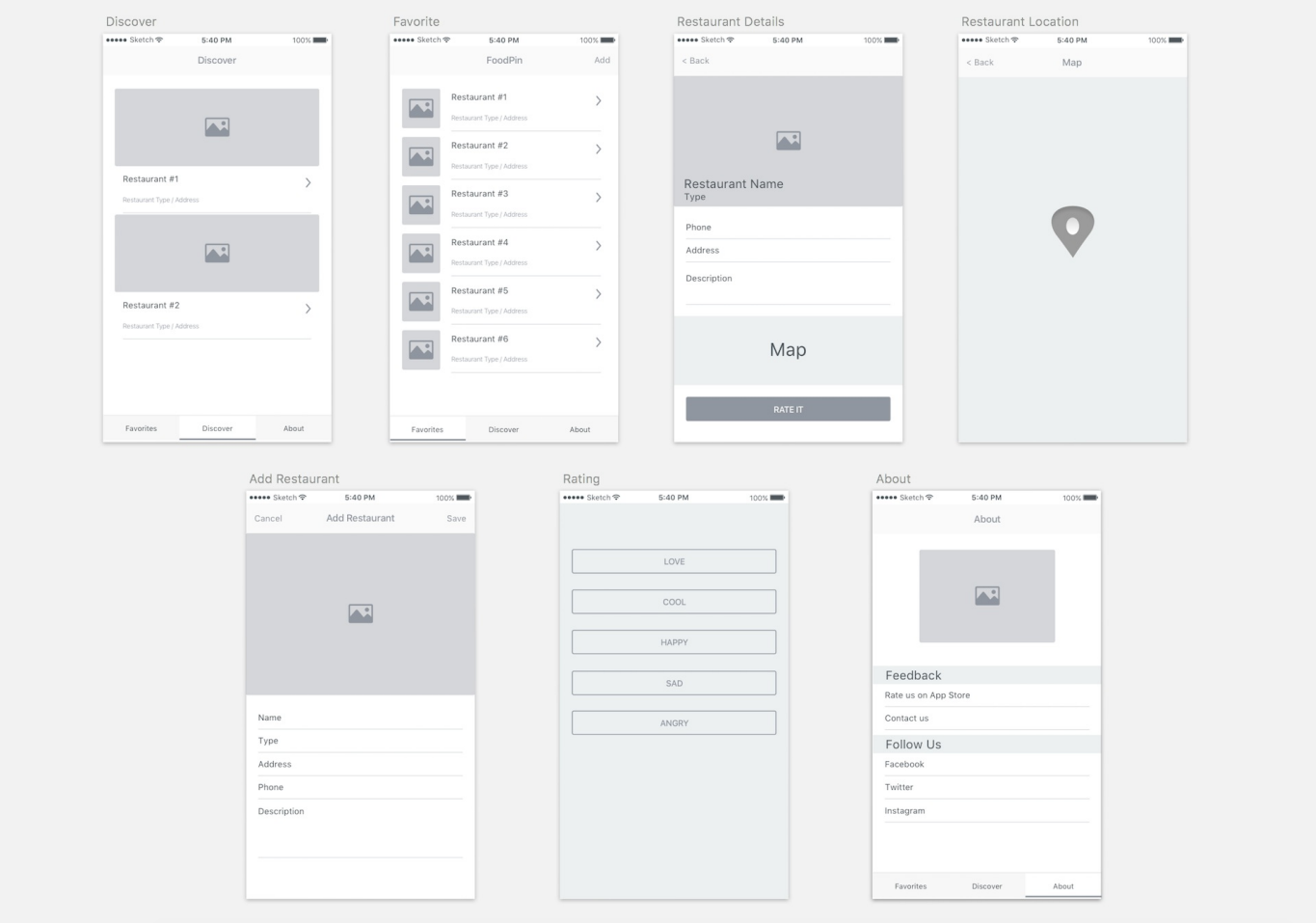


Figure 7-3. A sample wireframe of a mobile app

Make Your Sketch/Wireframe Interactive

You already sketch your app idea on paper. You may even wireframe your app. Can you make it even better to demonstrate how it works and show your potential users its workflow? There are a number of tools for developers to build interactive prototypes which I will further discuss in the next section. Meanwhile, I want to show you how to create a simple prototype using a very handy tool called POP (https://marvelapp.com/pop/) from Marvel.

POP was first developed by a Taiwan-based startup called Woomoo. This prototyping app is brilliant and has once been featured by Apple. Later, Woomoo's team was acquired by Priceline, while its flagship prototyping app POP found its new home at Marvel, a U.K. startup.

The POP app can easily turn your hand drawings or wireframes into a working prototype. It lets you capture your drawings using a camera or import them from your photo album. To interact with the images, the app offers various transitions for you to link your screens up. You'll understand what I mean in a while.

First, install the POP app on your iPhone and download the wireframe images from http://www.appcoda.com/resources/swift4/FoodPinWireframe.zip. You can unzip the images and import them to your iPhone via AirDrop.

POP is very easy to use. When you launch it for the first time, you will see your project listing. Click the `+` icon to create a new project and select your preferred device. Give your project a name (e.g. Food Pin). Once the project is created, click the `+` icon and choose the *Phone* option to import your wireframe images. Alternatively, you can capture your sketches using the built-in camera. Figure 7-4 shows a sample POP project.

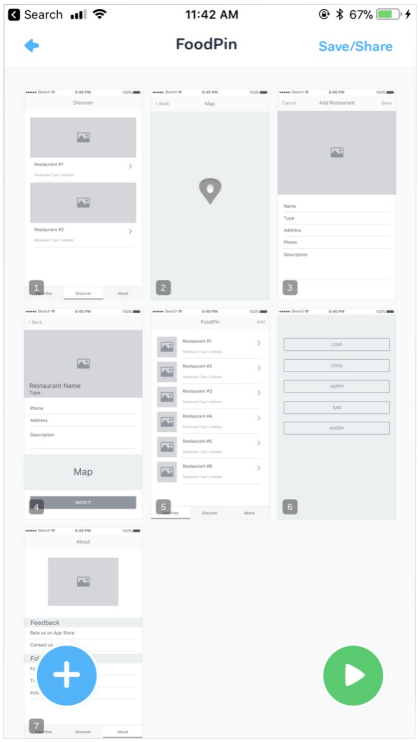


Figure 7-4. A sample POP project

Start with the home screen of the app and define the screen transition. POP lets you highlight a specific area of the image and specify the target page when that area is tapped. Then define the transition type including fade, next, back, rise and dismiss. Say, for the home screen, the app should navigate to the detail screen when a user taps any of the records. To configure a screen transition, you can click the *Add Link* button to highlight the records and then click *Link to image*. Select the target image, which is the restaurant detail. Optionally, you can set up the screen transition animation.

Once you have made the change, go back to the project home. Tap the *Play* button to interact with the prototype. The app will transit to the detail screen when any of the records are tapped.

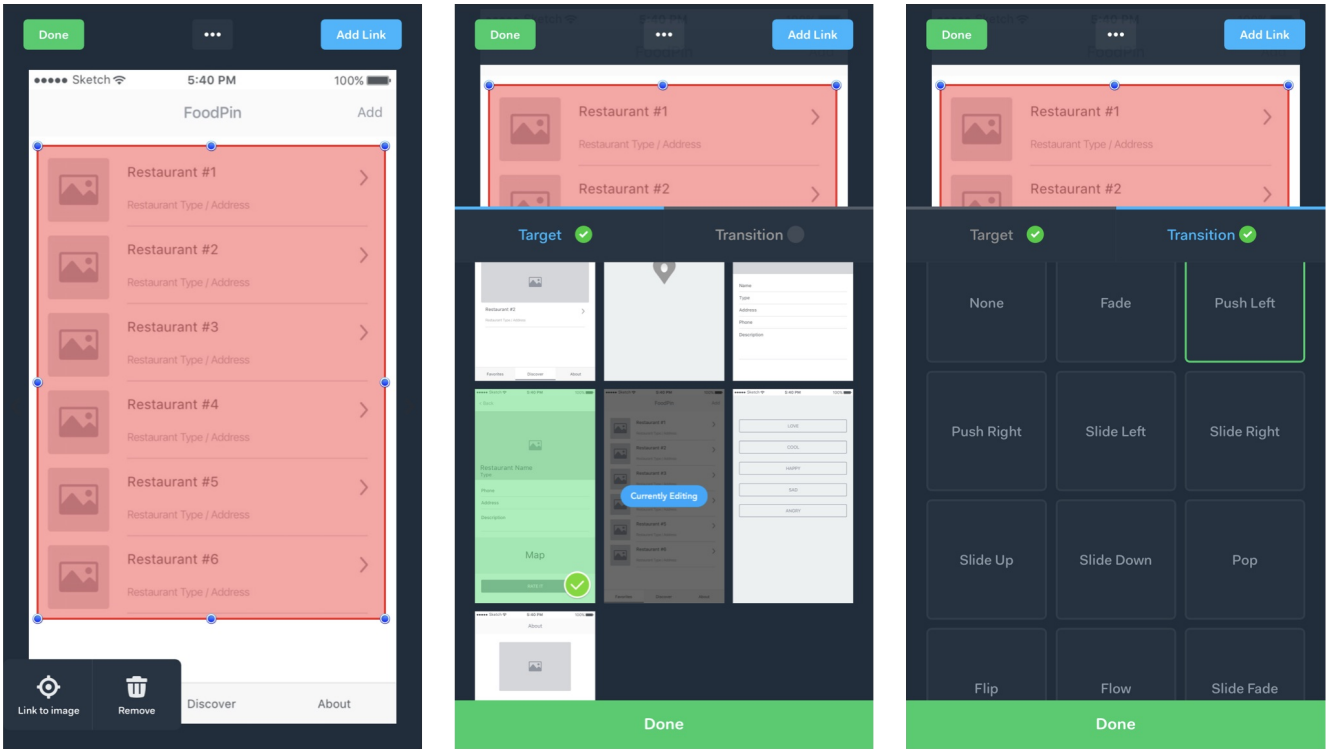


Figure 7-5. Defining screen transitions in POP

You just need to repeat the procedures to define the rest of the screen transitions. When the prototype is finished, you can share it with your team members and potential users using the Share option. Your users can try out the prototype using a web link:

https://marvelapp.com/10c52gg6

Prototyping Your App with App Prototyping Tools

This is how you visualize your idea. By building a simple prototype, it allows you to involve your users and solicit their feedback as early as possible. Wireframes are not visually pleasing. You can further build on the wireframes and add visual

elements to turn it into a high-fidelity prototype. In this section, I will explore some of the popular app prototyping tools that can help you design a fully functional prototype, which looks almost the same as the final product.

Sketch

I am a big fan of Sketch (https://www.sketchapp.com). Even if I am not an app designer, the tool allows me easily to turn my paper sketches/wireframes into a professional app design. The Sketch app comes with iOS app template to help you layout your UI. Most importantly, you can find tons of free/paid Sketch resources online (http://www.sketchappsources.com) that can elevate your app design. Sketch also has a companion iPhone app which lets you easily view the design on a real device.

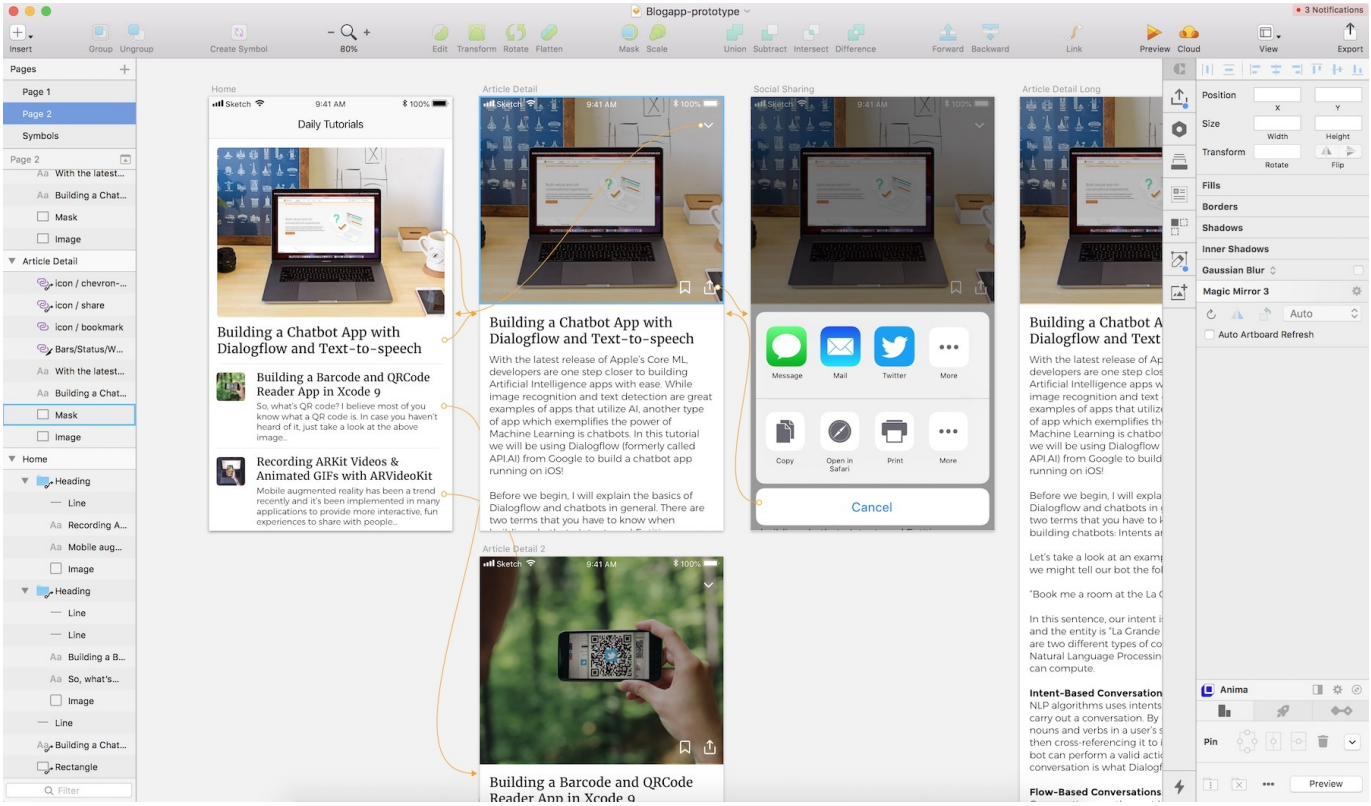


Figure 7-6. Designing the app UI using Sketch

The latest version of Sketch comes with a built-in prototyping feature. You can easily link up multiple screens (or artboards) to create interactive workflows and see the app in action by using the preview mode. If you have the Sketch app and want to see how the prototyping feature works, you can check out this link: https://sketchapp.com/docs/prototyping.

```
Note: If you want to learn more about Sketch, I highly
recommend you to check out Meng To's Design+Code book
(https://designcode.io).
```

Figma

Sketch has dominated the mobile design tool space for some time. The most notable alternative is a cloud-based prototyping tool called Figma (https://www.figma.com). Unlike Sketch which is a Mac app, Figma is a web application that can be used with any browser and puts emphasis on collaboration. All your design projects are stored in the cloud, allowing you to easily access your projects from anywhere and multiple designers to simultaneously collaborate on a single document.

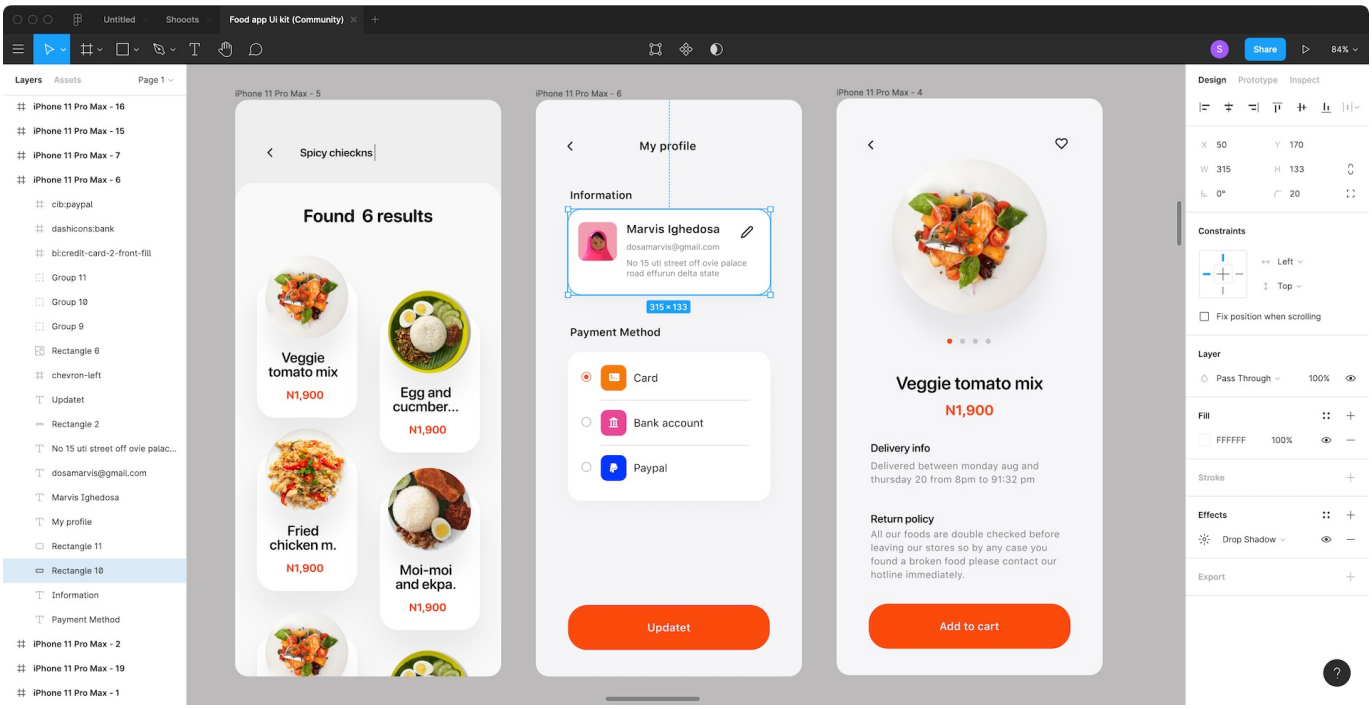


Figure 7-7. Designing the app UI using Figma

While Sketch doesn't have any free plan, Figma offers a free starter plan which allows you to create three prototyping projects. You will need to upgrade to other premium plans if you need to create more projects

If you've used Sketch before, it's not hard to switch to Figma. Both Sketch and Figma have their edges. However, if you need to work on a design with teams of remote designers, Figma is definitely a good choice.

Adobe Experience Design

In March 2016, Adobe announced a Sketch competitor called Adobe XD (short for Experience Design) for UI design. The company claimed it as an all-in-one tool for both web and mobile app design projects. At the time of writing, the app is free for download (http://www.adobe.com/products/experience-design.html).

If you've used Photoshop before, you will find Adobe XD a lot easier to use. The tool has two modes: *design* and *prototype*. In the Design mode, you can design the app UI with a set of layout tools. At a certain point, you can switch to the Prototype mode to build an interactive prototype with the app screens you just designed. It is pretty easy to connect one screen with the other. By clicking and dragging, you can create an app prototype with simple animations to demonstrate the workflow of your app.

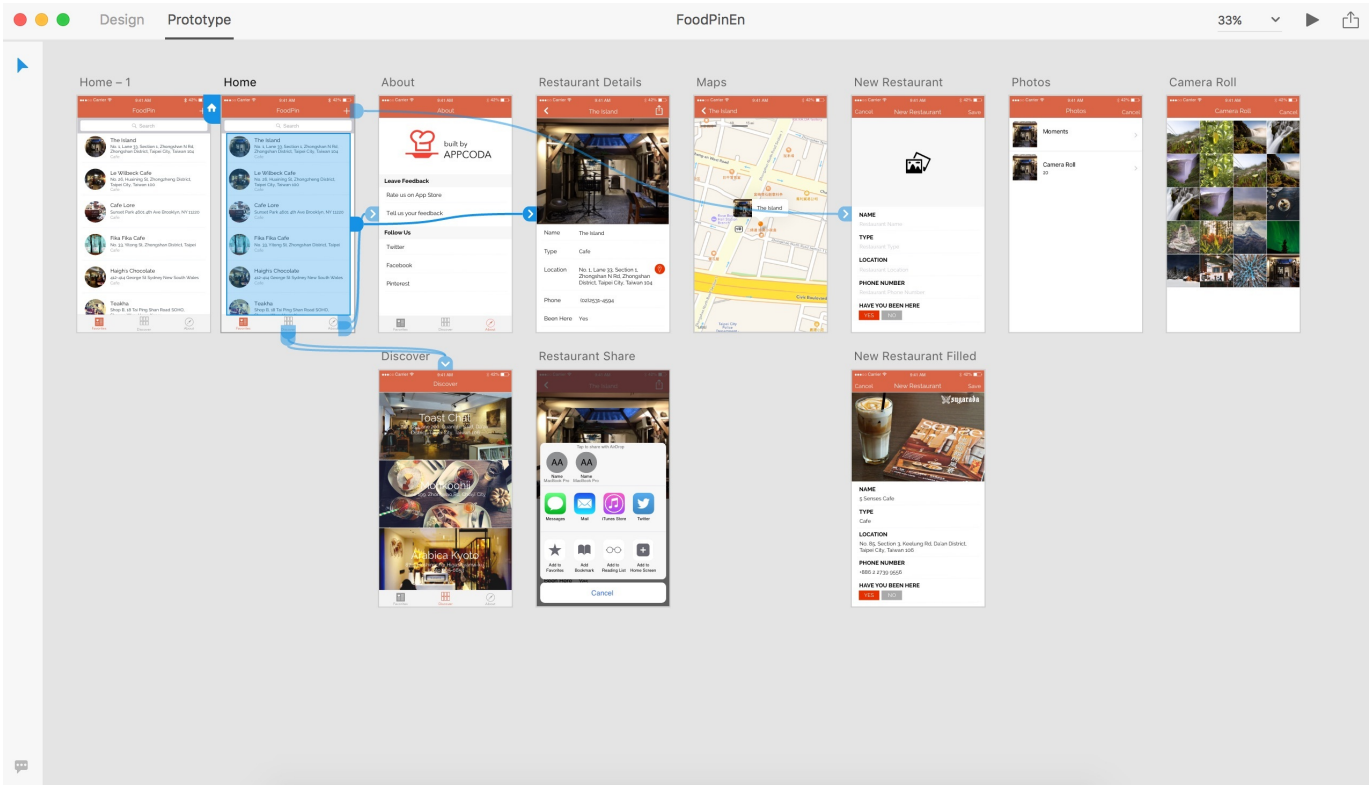


Figure 7-8. Prototyping the FoodPin App using Adobe XD

I have built a prototype demo, which is the app we're going to build in this book, using Adobe XD. You can download the project file from http://www.appcoda.com/resources/swift3/FoodPinEn.zip.

Invision

Invision (https://www.invisionapp.com/) is one of the leading prototyping and collaboration tools in the market. It allows you to quickly create interactive mockups without writing a single line of code. InVision supports Sketch files which means you can easily export your Sketch screens to InVision and turn them into an interactive prototype. Collaboration has been one of my favorite features of InVision. Let's say, you work with a UI designer to build the prototype. InVision lets you share feedback and add a comment to the screen design. This is particularly useful when you work with your design team or an external UI designer.

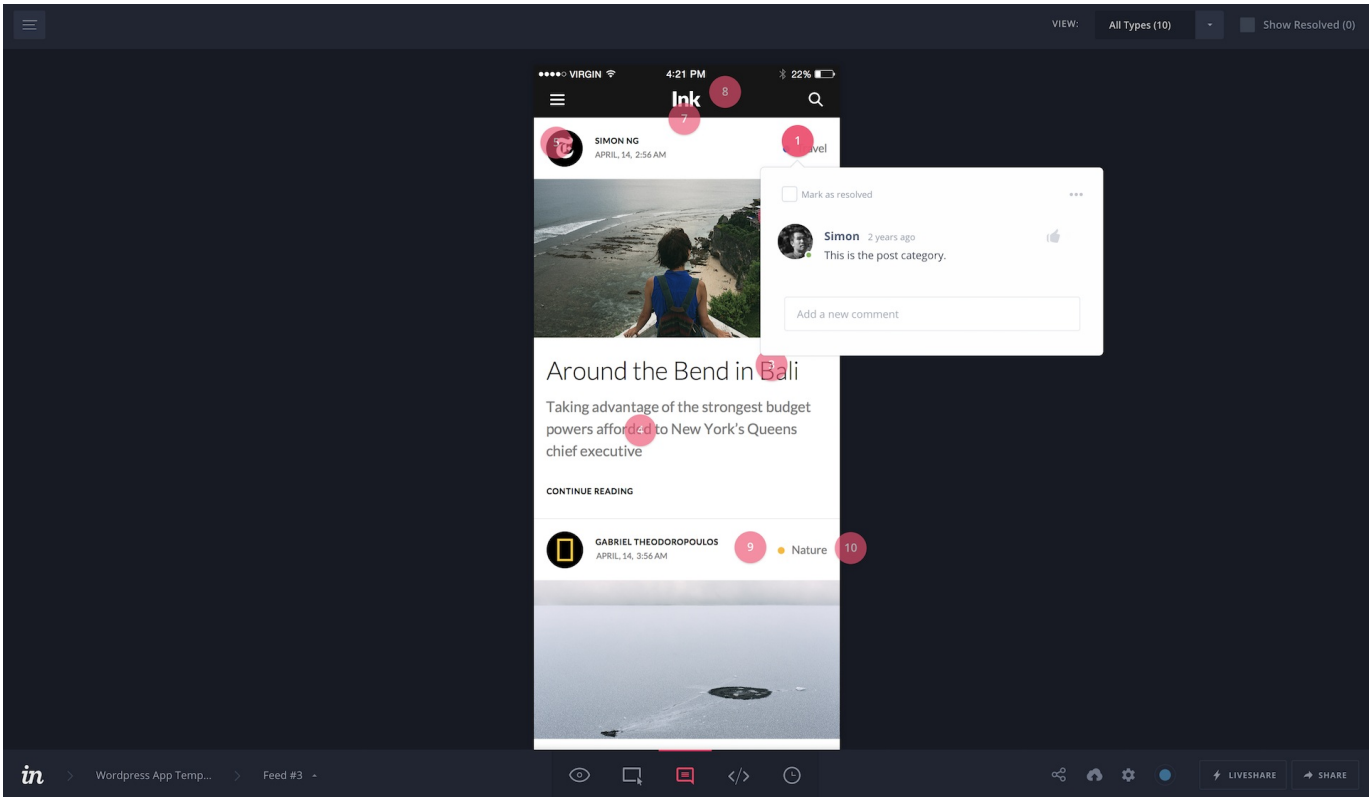


Figure 7-9. Prototyping using InVision

Flinto

Flinto (https://www.flinto.com) is another easy-to-use prototyping tool that lets you create realistic app prototype. The app is intuitive to use and pretty easy to learn. You can easily link up screens and create animated transitions.

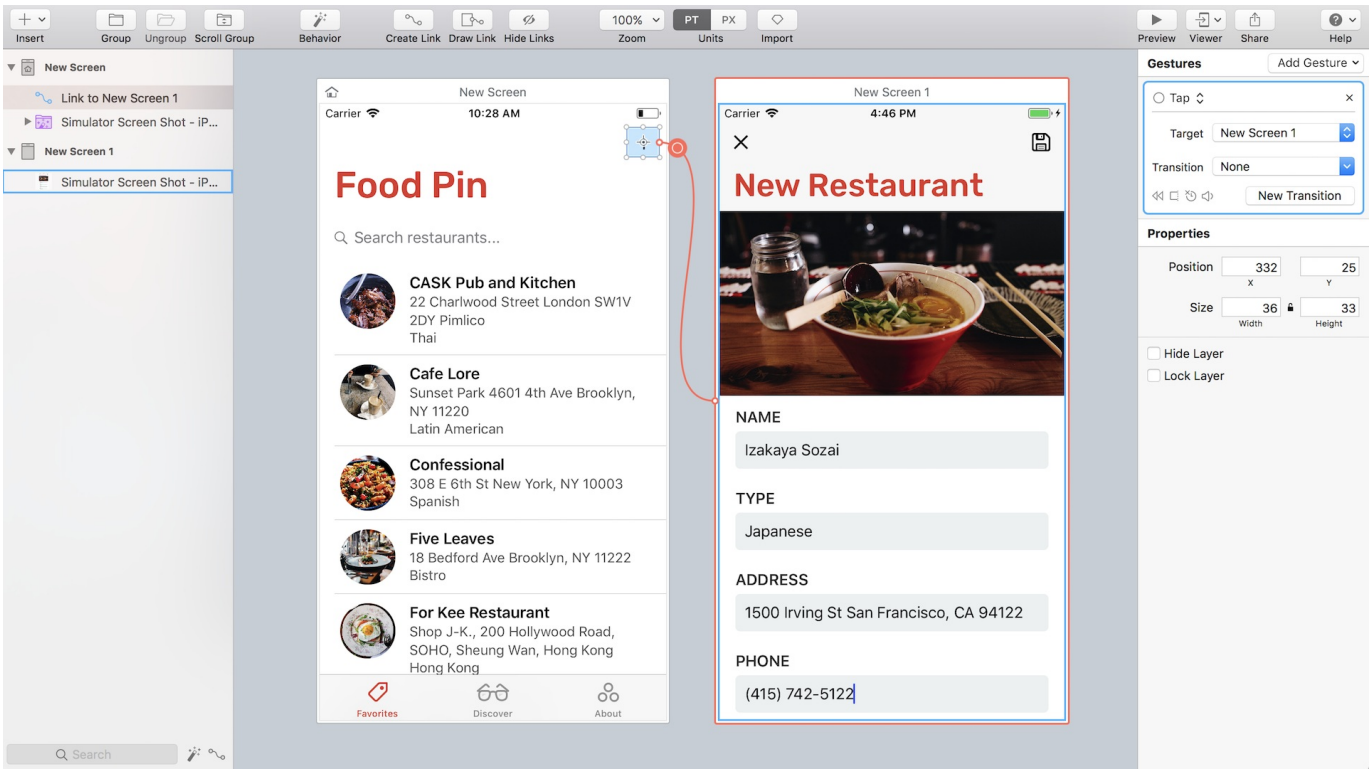


Figure 7-10. Prototyping using Flinto

Keynote

Keynote! Are you kidding me? You heard me right. Apple's Keynote can also be used to make quick prototypes. Actually, Apple's engineers mentioned in WWDC that they used the presentation tool to create a prototype for their app projects.

Keynote shouldn't be new to you if you've used it for creating presentations. The same set of drawing tools built in Keynote allow you to design a simple app UI. Keynotpia offers mock templates to streamline your sketches. Figure 7-10 shows a sample screen created using Keynote.

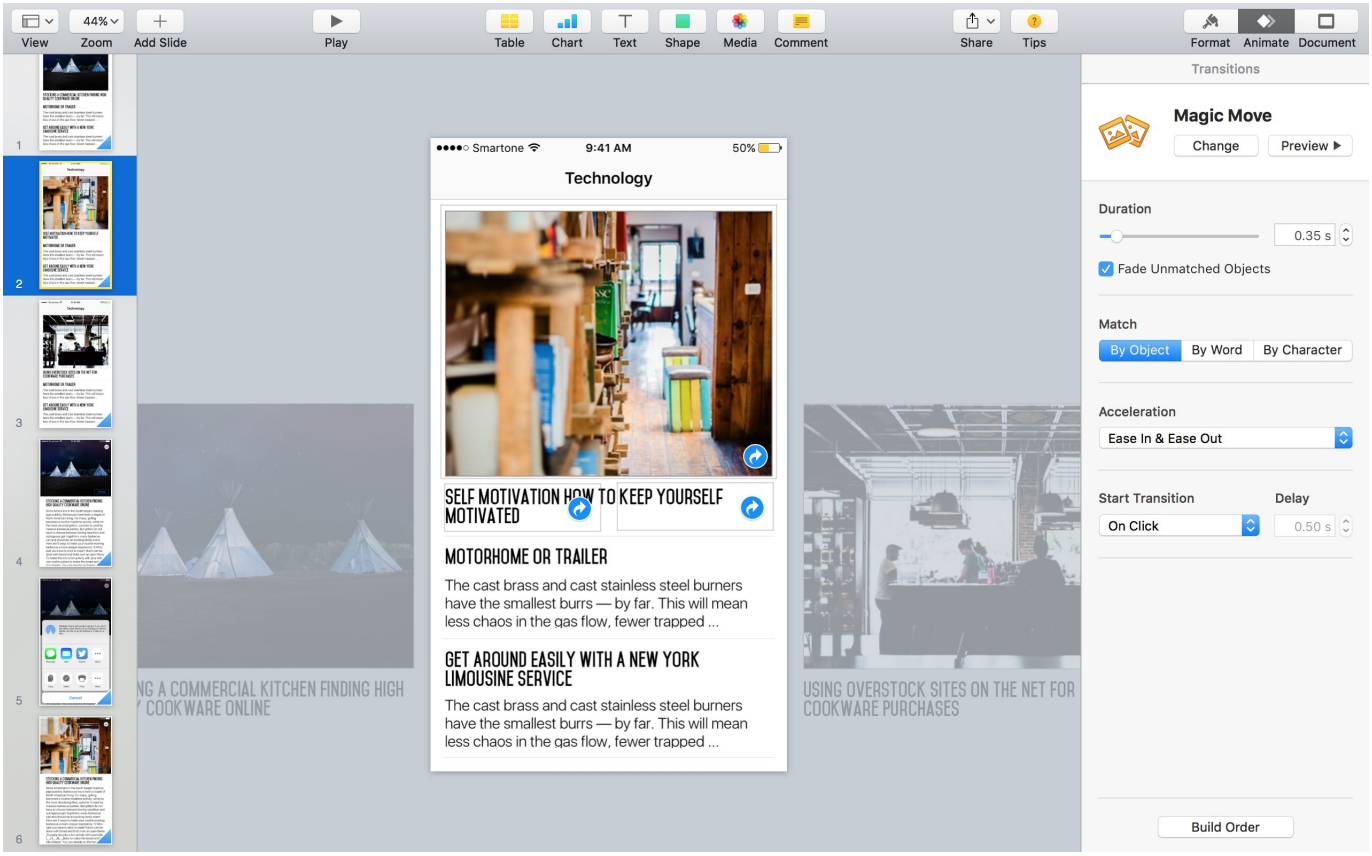


Figure 7-11. Design an app prototype using Keynote

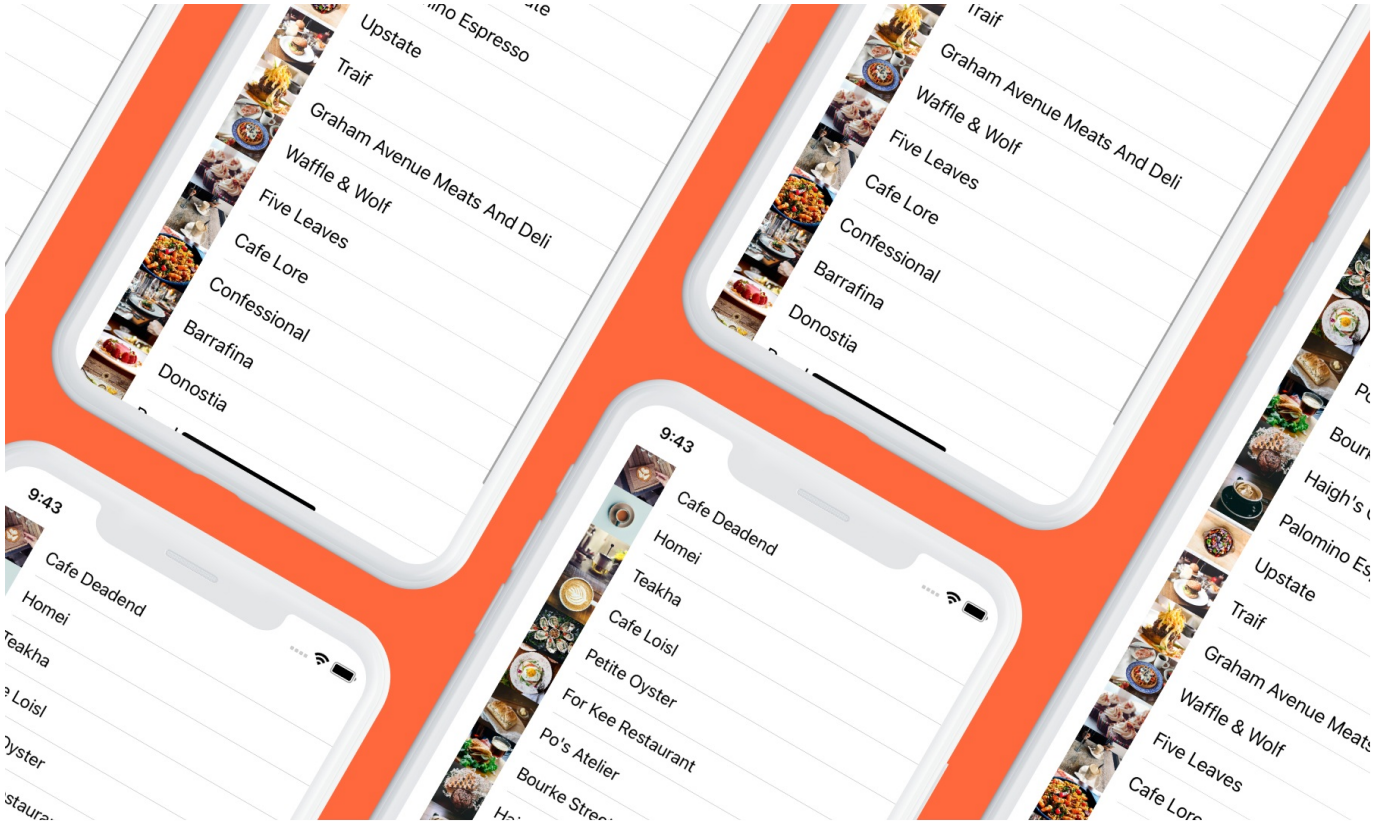
Not only can you design the app UI, interestingly, Keynote makes it very easy to animate the static mockups. The magic move transition lets you animate the screen transition to replicate the feel of native apps. I will not go into the details about how you can create a working prototyping with Keynote. If you want to learn more about how to prototype using Keynote, you can check out this article (http://webdesign.tutsplus.com/tutorials/how-to-demo-an-ios-prototype-in-keynote--cms-22279). You can also take a look at Apple's 60 Seconds Prototyping video (https://developer.apple.com/videos/play/wwdc2017/818/).

Summary

Prototyping is an essential process in app development. It allows you to quickly build something workable and show users. Prototyping is used to test an idea and get feedback as early as possible. If you're building an app for a client, building a prototype lets your client clearly understand the app design and how it works before seeing the final product.

Thus, no matter if you are a solo developer or a member of a development team, I want you to start prototyping today. Rather than jumping right in to build an app, lay out your idea on paper first and build a simple demo using POP or other prototyping tools. This will save you a lot of time and money creating a product with no appeal.

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.

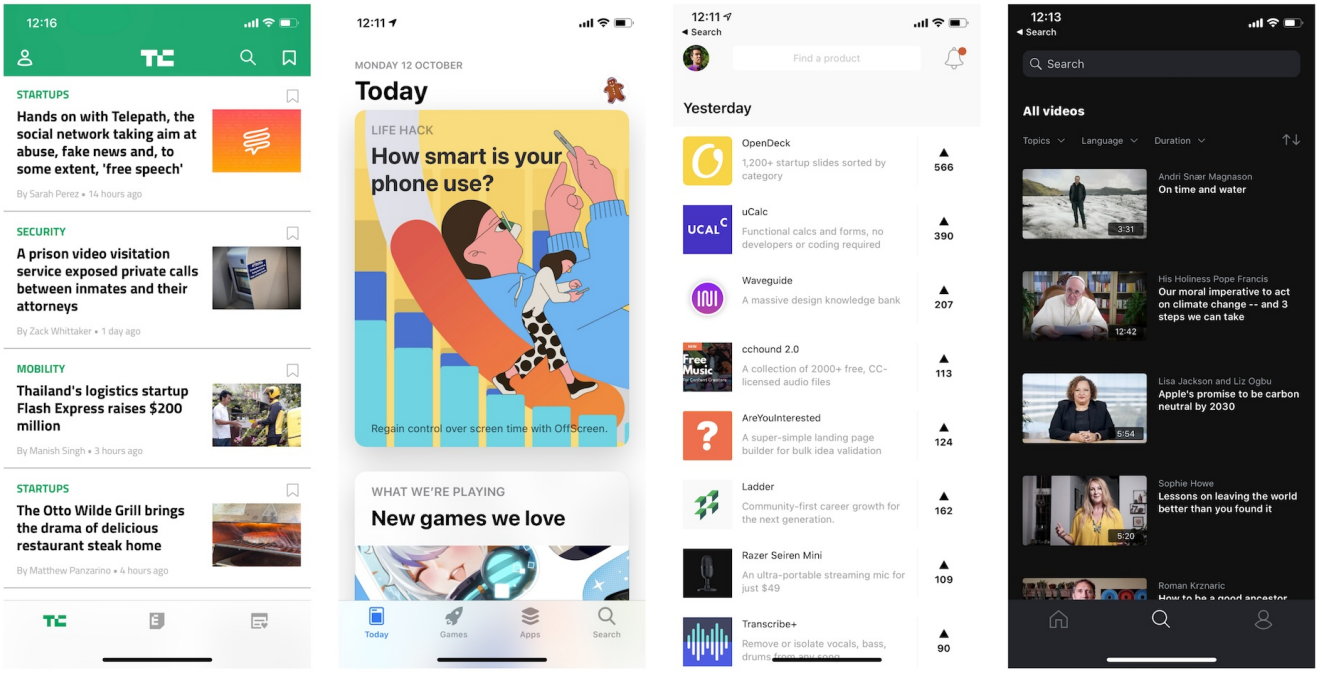


Figure 8-1. Sample table-based apps (left to right: Techcrunch, App Store, Product Hunt, and TED)

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.

Creating a SimpleTable Project

> Don’t just read the book. If you’re serious about learning iOS programming, stop reading. Open your Xcode and code! This is the best way to learn programming.

Let's get started and create a simple app. The app is really simple. We'll just display a list of restaurants in a plain table view. We'll polish it in the next chapter. If you haven't fired up Xcode, launch it, and create a new project using the *App* template under iOS.

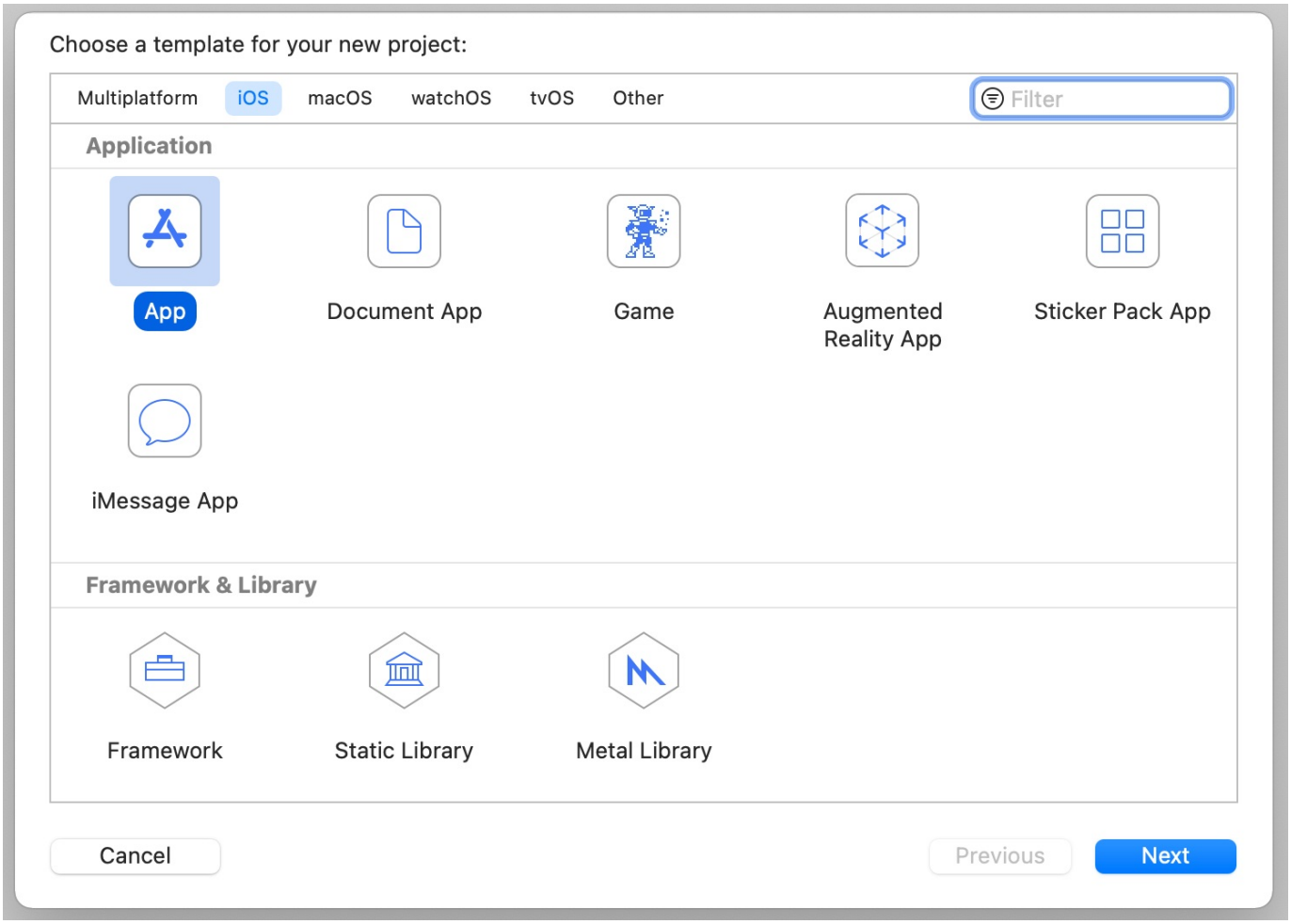


Figure 8-2. Xcode Project Template

Click "Next" to continue. Again, fill in all the required options for the Xcode project:

- **Product Name: SimpleTable** – 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.".
- **Bundle Identifier: com.appcoda.SimpleTable** - This field should be

automatically generated by Xcode.

- **Interface: Storyboard** - Xcode now supports two ways to build UI. In this book, we use Storyboard. If you want to learn SwiftUI, you can refer to our new book (https://www.appcoda.com/swiftui).
- **Life Cycle: UIKit App Delegate** - When Storyboard is used, the life cycle should be default to UIKit App Delegate.
- **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 *SimpleTable* project. Pick a folder on your Mac. Click "Create" to continue.

Design the User Interface

To present data in a table in an iOS app, all you need to use is the table view object. First, select `Main.storyboard` to switch to the Interface Builder editor. Again, I prefer to use iPhone 11/12 to design the UI. Select `View as:` button and change it to iPhone 11/12. Open the Object library, look for the `Table View` object and drag it into the view.

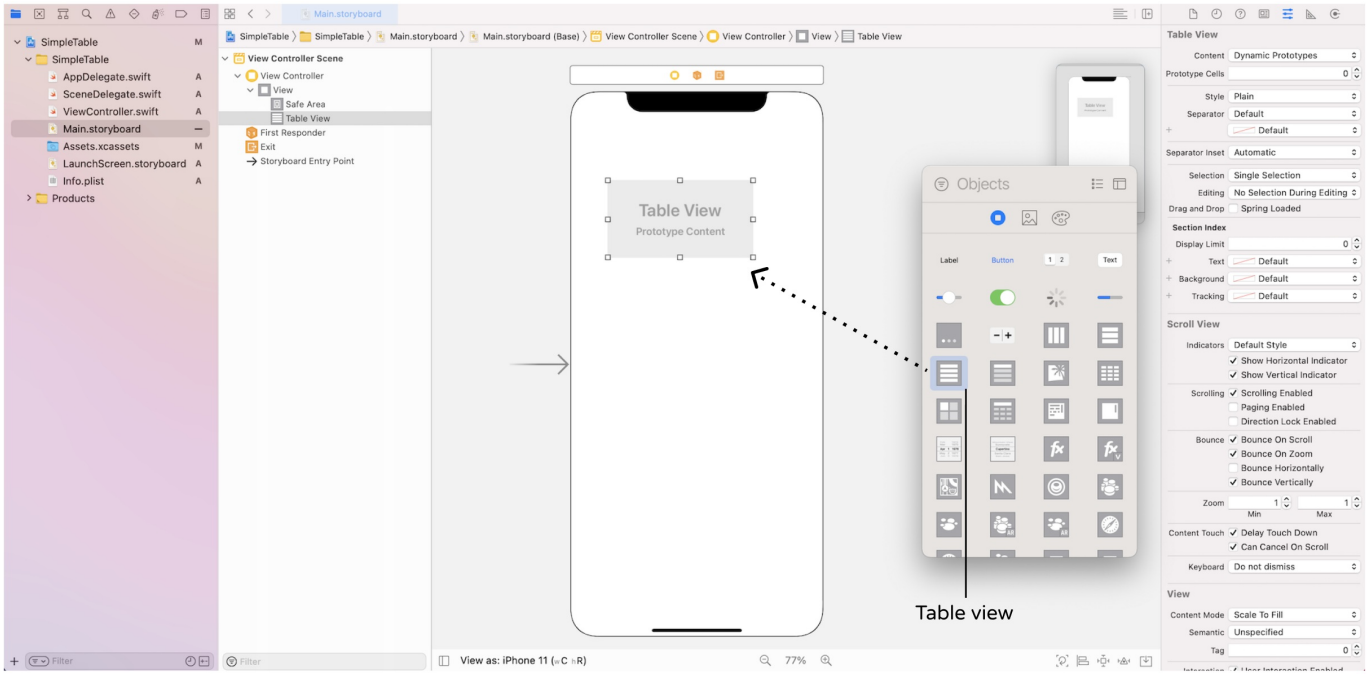


Figure 8-3. Drag a table view from the Object library to the View

Resize the table view to fit the whole view. Next, go to the Attributes inspector (if it doesn't appear in your Xcode, choose *View* > *Inspectors* > *Attributes*), change the number of Prototype Cells from `0` to `1`. A prototype cell will appear in the table view.

Prototype cells allow you to easily design the layout of your table view cell. You can think of it as a cell template such that you can reuse it in all the table cells.

By default, a prototype cell comes with several standard cell styles including basic, right detail, left detail and subtitle. You can create your cell design or simply choose from the default style. In this example, we will use the basic style. For cell customizations, I will leave it to the next chapter.

Select the cell and then open the Attributes inspector. Change the cell style to *Basic*. This style is good enough to display a cell with both text and image. Additionally, set the identifier to `datacell`. This is a unique key for identifying the prototype cell. We'll use it later when writing code.

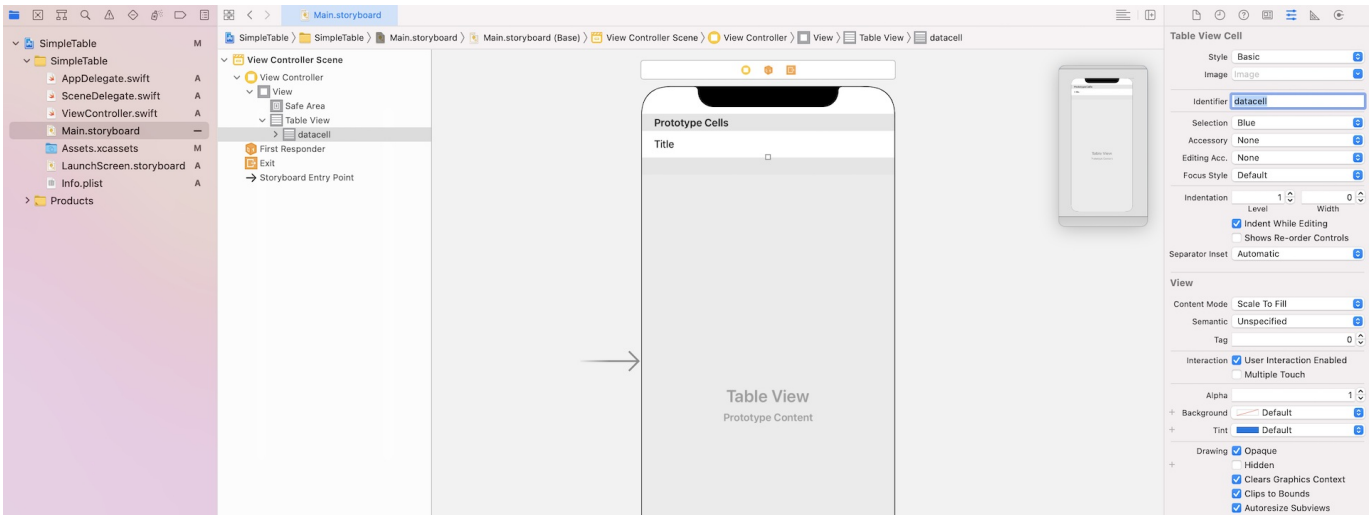


Figure 8-4. Prototype Cell in table view using Basic style

Run Your App to Have a Quick Test

Before moving on, try to run your app on both iPhone and iPad simulators. The Simulator screen should look like the one in figure 8-5.

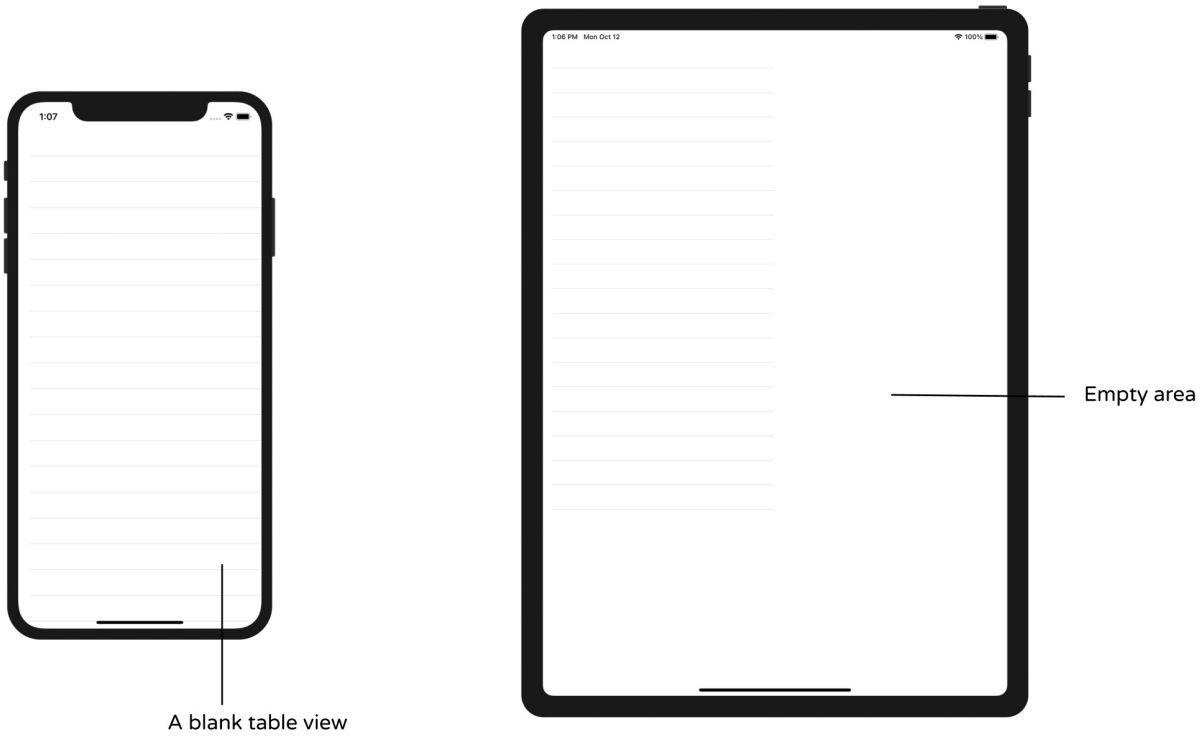


Figure 8-5. A blank table view on iPhone 11 (left) and iPad Pro 11-inch (right)

Easy, right? You have created the table view for your app. Meanwhile, it doesn't display any data. If you take a closer look at the table view, it is not perfectly stretched to take up the whole screen on iPad. I believe you should know the reason if you thoroughly understand auto layout.

Quick tip: Xcode allows you to run multiple simulators at the same time. If you've run the app on both iPhone and iPad, you now should have two simulators. To save system resources, you may select the iPad simulator and press command+w to close it.

So far we haven't defined any layout constraints for the table view. This is why it is not displayed properly on every device. Now select the *Table View* in Interface Builder, click the *Add new constraints* button in the layout bar. Set the spacing of the top, left, right and bottom side to `0`.

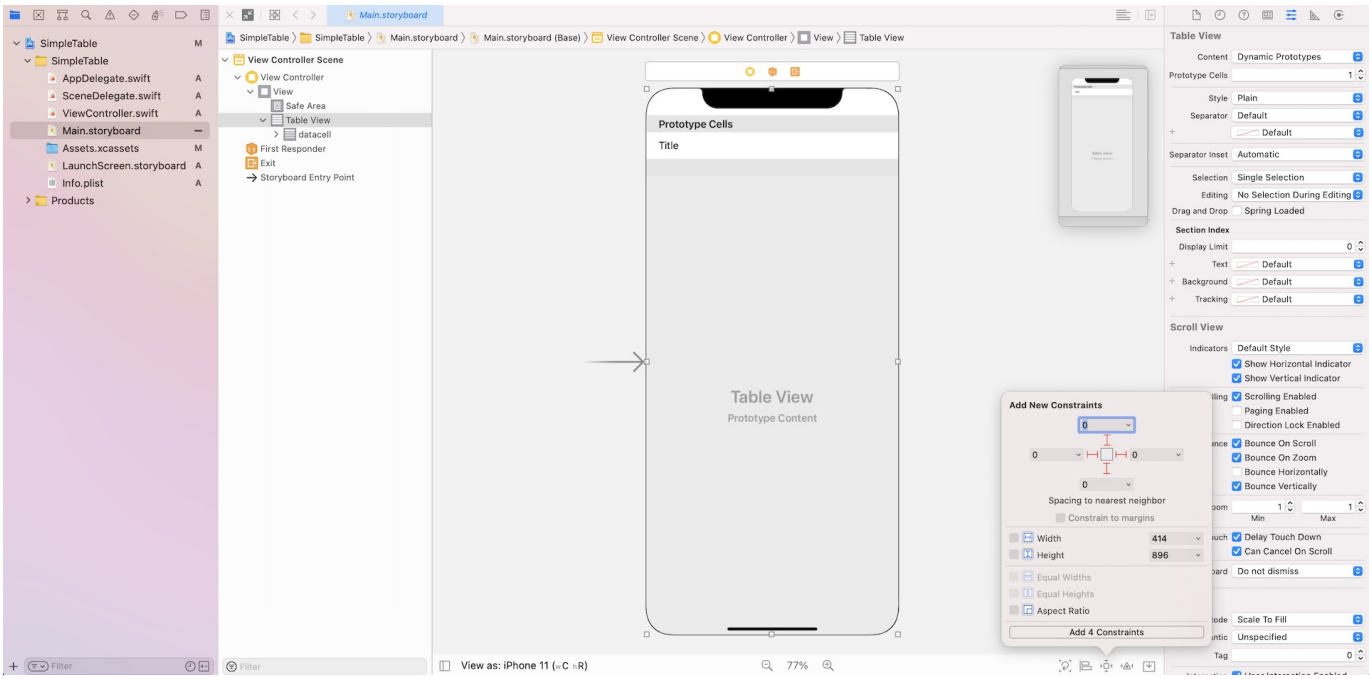


Figure 8-6. Adding layout constraint to the table view

Make sure the bars are in solid red and the *Constrain to margins* option is unchecked. Then click the *Add 4 Constraints* button to add the constraints.

Here we define 4 spacing constraints for each side of the table view. The *Constrain to margins* option is unchecked. In this case, the spacing constraints will be relative to the edge of the view. This ensures that all sides of the table view will stretch to the edge of the view. In other words, your table view will automatically resize to fit all screen sizes.

If you tick the *Constrain to margins* option, it will still work. Figure 8-7 shows you how the safe area is adjusted when the option is enabled or disabled.



Figure 8-7. The Safe Area layout guide varies based on the selection of the Constrain to Margins option

You can run the project again. The table view should now support all screen sizes. With the UI design ready, let's move onto the core part and write some code to insert the table data.

UITableView and its Protocols

I mentioned at the very beginning of the book that other than Swift, you have to learn the APIs and foundation classes provided by the iOS SDK in order to build an app. These APIs and classes are organized into what-so-called *frameworks*. The UIKit framework is one of the most commonly used frameworks.

It provides classes to construct and manage your app's user interface. All objects listed in the Object library of Interface Builder are provided by the framework. The Button object you used in the HelloWorld app, and the Table View object we're working on right now are from the UIKit framework. While we use the term *Table View*, the actual class is *UITableView*. Simply put, every UI component in the Object library has a corresponding class. You can first select *Table View*, and then go to the Identity inspector to reveal the actual class name in the custom class section.

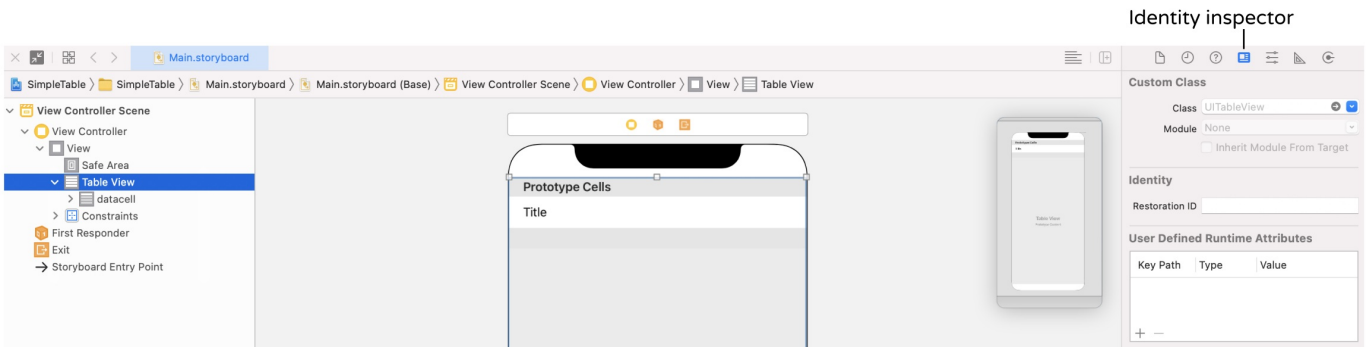


Figure 8-8. Click the object in the Object library to reveal the actual class name in UIKit framework

> I intentionally leave out the discussion of classes and methods to later chapters. If you have no ideas what a class is, don't worry. Just think of it as a code template. I will explain it to you in later chapters.

Now that you have an idea about the relationship between Table View and UITableView. We'll write some code to insert the table data. Select `ViewController.swift` in the project navigator to open the file in the editor pane. Append `, UITableViewDataSource, UITableViewDelegate` after `UIViewController` to adopt the protocols.

As soon as you insert the code after `UIViewController`, Xcode detects an error. The red exclamation mark at the top-right corner of the code editor indicates that there is a problem. On top of that, Xcode highlights the line of code, and display a message telling you the details of the issue. The message is helpful for telling the cause of the problem, and it also shows you the solution.

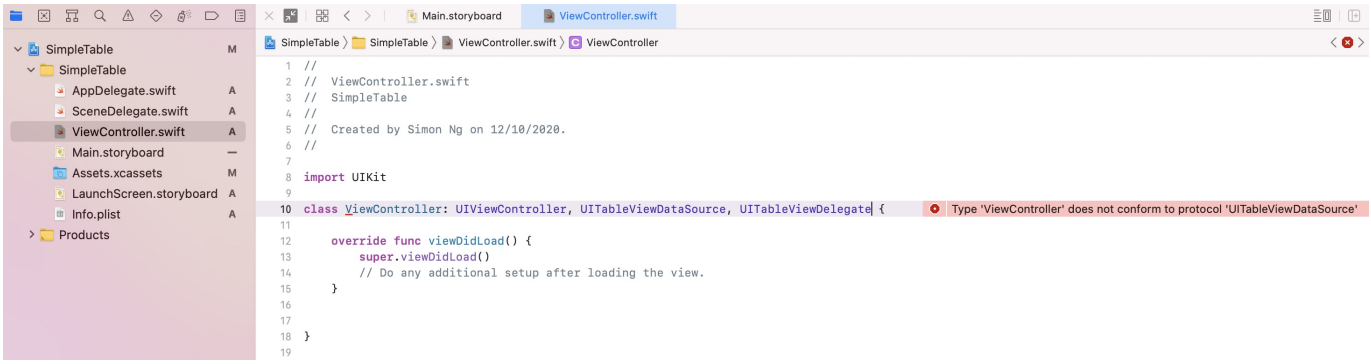


Figure 8-9. Implementing UITableViewDataSource and UITableViewDelegate protocols

Wait. Don't click the *Fix* button yet. What does it mean by "Type ViewController does not conform to protocol UITableViewDataSource"?

The `UITableViewDelegate` and `UITableViewDataSource` are known as protocols in Swift. To display data in a table view, we have to conform to a set of requirements, defined in the protocols. Here, the `ViewController` class is the one that adopts the protocol, and implements all the mandatory methods.

It may be confusing. What are these protocols? Why protocols?

Well, let's say you're starting a new business. You hire a graphic designer to design your company logo. The designer is skilled and very capable of creating any kind of logo. But the designer can't start the logo design right away. In the least, you need to provide the designer with some requirements such as company name, color

preference, business nature before he/she can create a logo. However, you're busy. You delegate the task to your personal assistant, and let the assistant deal with the designer directly, providing the designer with the logo requirements.

In iOS programming, the `UITableView` class is just like the graphic designer. It's flexible enough to display various kinds of data (e.g. image, text) in table form. You can use it to display a list of countries or contact names. Or in our project, we'll be presenting a list of restaurants with thumbnails.

But before UITableView can display the data for you, it requires someone to provide some basic information like:

- how many rows do you want to display in the table view?
- what is the table data? For example, what do you want to display for row 2? What do you want to display in row 5?

That "someone" is known as a delegate object. In the above analogy, the personal assistant is the delegate object. In iOS programming, it also applies the concept of delegation usually known as *delegation pattern*. An object relies on another object to perform a specific task. In our project, the ViewController is the delegate that provides the table data. Figure 8-10 illustrates the relationship of UITableView, the protocols and the delegate object.

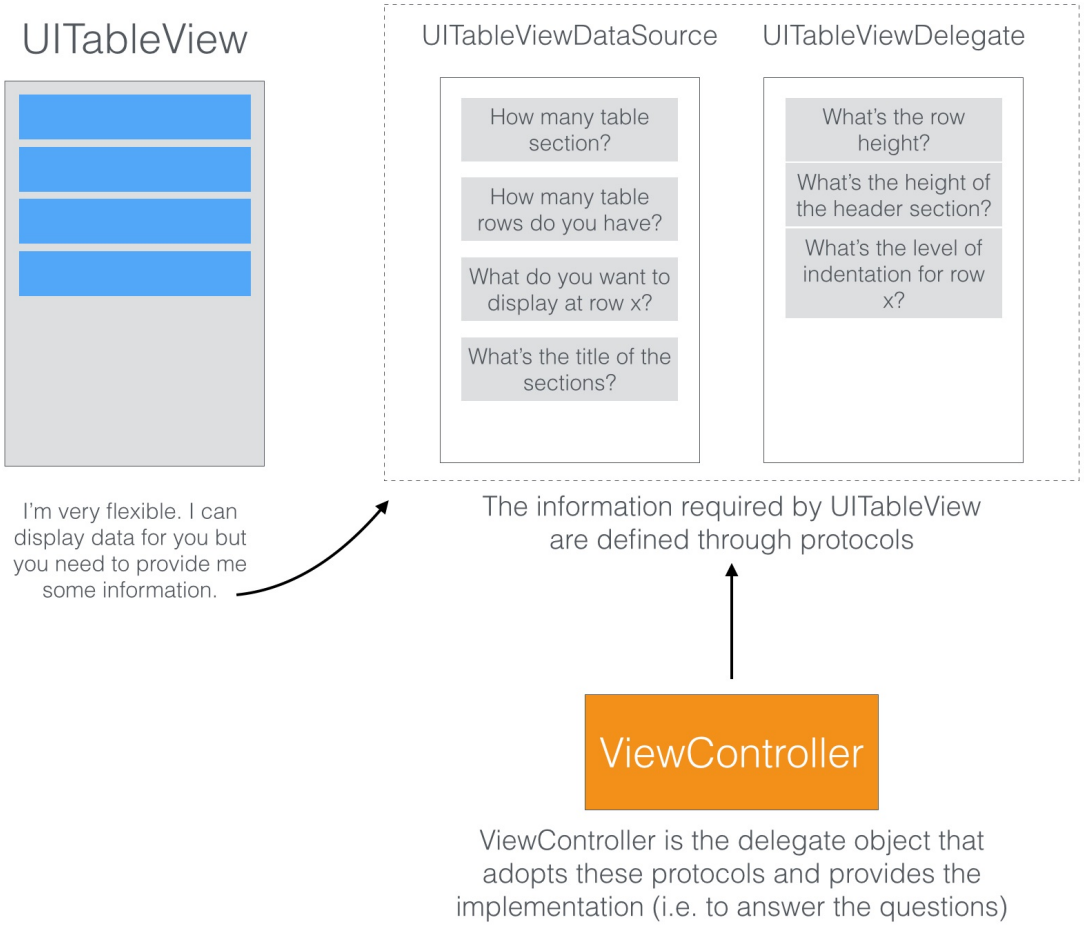


Figure 8-10. The relationship of UITableView, the protocols and the delegate object

So how do you tell UITableView what data to display? The `UITableViewDataSource` protocol is the key. It's the link between your data and the table view. The protocol defines a list of methods that you have to adopt.

Okay, what methods should I adopt? Now you are ready to click the *Fix* button in the error prompt shown in figure 8-9. As soon as you click the button, Xcode adds the following methods:

```
func tableView(_ tableView: UITableView, numberOfRowsInSection section: In
t) -> Int {

}

func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath)
 -> UITableViewCell {

}
```

These are the two required methods for the `UITableViewDataSource` protocol.

All you need is to provide the implementation so that `UITableView` knows the total number of rows to display and the data for each row. The protocol also defines some optional methods that we'll not discuss it here.

The `UITableViewDelegate` protocol, on the other hand, deals with the appearance of the `UITableView`. All methods defined in the protocols are optional. They let you manage the height of a table row, configure section headings and footers, re-order table cells, etc. We will not change any of these methods in this example. We will leave that for a later chapter.

With some basic knowledge of protocols, let's continue to code the app. Select `ViewController.swift` and declare a variable for holding the table data. Name the variable `restaurantNames` and insert the following line of code in the class:

```
var restaurantNames = ["Cafe Deadend", "Homei", "Teakha", "Cafe Loisl", "P
etite Oyster", "For Kee Restaurant", "Po's Atelier", "Bourke Street Bakery"
, "Haigh's Chocolate", "Palomino Espresso", "Upstate", "Traif", "Graham Av
enue Meats And Deli", "Waffle & Wolf", "Five Leaves", "Cafe Lore", "Confes
sional", "Barrafina", "Donostia", "Royal Oak", "CASK Pub and Kitchen"]
```

In this example, we use an array to store the table data. If you forgot the syntax of an array, refer to chapter 2. The values are separated by commas and surrounded by a pair of square brackets.

When I said "insert the code in the class", it means you have to declare the variable within the curly bracket of the class like this:

```
class ViewController: UIViewController, UITableViewDataSource, UITableView
Delegate {

    var restaurantNames = ["Cafe Deadend", "Homei", "Teakha", "Cafe Loisl"
, "Petite Oyster", "For Kee Restaurant", "Po's Atelier", "Bourke Street Ba
kery", "Haigh's Chocolate", "Palomino Espresso", "Upstate", "Traif", "Grah
am Avenue Meats And Deli", "Waffle & Wolf", "Five Leaves", "Cafe Lore", "C
onfessional", "Barrafina", "Donostia", "Royal Oak", "CASK Pub and Kitchen"
]

    .
    .
    .

}
```

Arrays for Absolute Beginners

An array is a fundamental data structure in computer programming. You can think of an array as a collection of data elements. Consider the `restaurantNames` array in the above code, it represents a collection of `String` elements. You may visualize the array like this:



Figure 8-11. restaurantNames array

Each of the array elements is identified or accessed by an index. An array with 10 elements will have indices from 0 to 9. That means `restaurantNames[0]` returns the first item of the array.

Let's continue to code. We adopt the two required methods of the `UITableViewDataSource` protocol like this:

```
func tableView(_ tableView: UITableView, numberOfRowsInSection section: In
t) -> Int {
    // Return the number of rows in the section.
    return restaurantNames.count
}

func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath)
 -> UITableViewCell {
    let cellIdentifier = "datacell"
    let cell = tableView.dequeueReusableCell(withIdentifier: cellIdentifie
r, for: indexPath)

    // Configure the cell...
    cell.textLabel?.text = restaurantNames[indexPath.row]

    return cell
}
```

The first method is used to inform the table view the total number of rows in a section. You can simply call the `count` method to get the number of items in the `restaurantNames` array.

> **Quick note:** A table view can have multiple sections but there is only one by default.

The second method will be called every time a table row is displayed. By using the `indexPath` object, we can get the current row (`indexPath.row`). Here what we did is to retrieve the indexed item from the `restaurantNames` array, and assign it to the text label (`cell.textLabel?.text`) for display.

> **Note:** You may wonder what the question mark (?), next to textLabel, is for. Do you still remember optionals we discussed in chapter 2? Here *textLabel* is an optional. To access the property of textLabel, we have to first verify if textLabel has a value. You can use *if let* to perform the verification. Alternatively, you can place a question mark (?) after textLabel.

Okay, but what is `dequeueReusableCell` in the second line of code?

The `dequeueReusableCell` method is used for retrieving a reusable table cell from the queue with the specified cell identifier. The `datacell` identifier is the one we defined earlier in Interface Builder.

You want your table-based app to be fast and responsive even when handling thousands of rows of data. If you allocate a new cell for each row of data instead of reusing one, your app will use excess memory and may result in a sluggish

performance when the user scrolls the table view. Remember every cell allocation has a performance cost, especially when the allocation happens over a short period of time.

The screen real estate of iPhone is limited. Even if your app needs to display 1,000 records, the screen may only be able to fit 10 table cells at most. Therefore, why on earth would the app allocate a thousand table view cells instead of creating 10 table cells and reuse them? This would save tons of memory and make table view work more efficiently. For performance reasons, you should reuse table view cells instead of creating new ones.

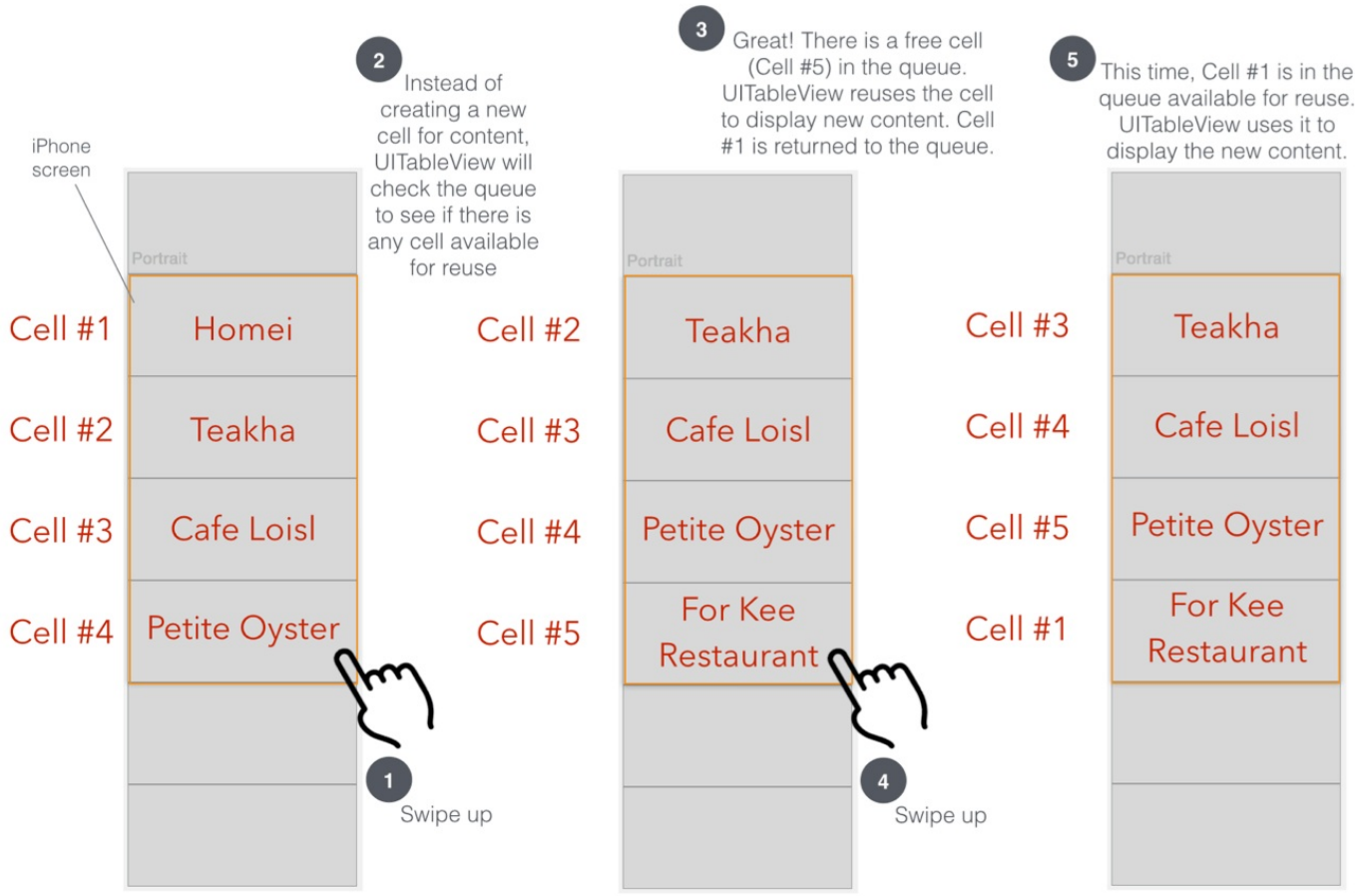


Figure 8-12. Illustration showing how UITableView reuses table view cells

Now, let's hit the "Run" button and test your app. Oops! The app still displays a blank table view just like before.

Why is it the table view still not showing the content as expected? We've written the code for displaying table data and implemented all required methods. But why the table view didn't show the content as expected?

There is still one thing left.

Connecting the DataSource and Delegate

Even though the `ViewController` class has adopted the `UITableViewDelegate` and `UITableViewDataSource` protocols, the `UITableView` object in storyboard has no idea about it. We have to tell the `UITableView` object that `ViewController` is the delegate object for the data source.

Go back to `Main.storyboard`. Select the table view. Press and hold the control key, drag from the table view to the View Controller object in the document outline view.

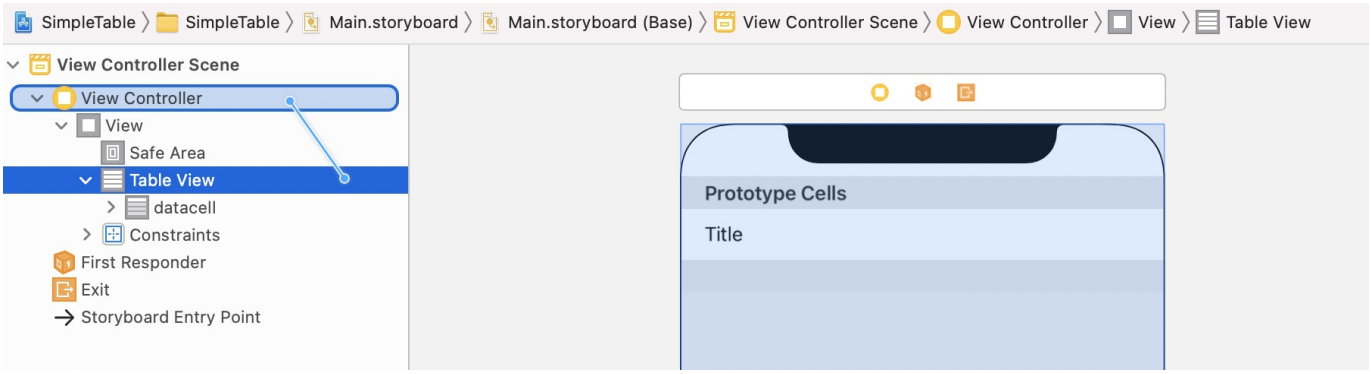


Figure 8-13. Connecting Table View with its Datasource and Delegate

Release both buttons. In the pop-over menu, select *dataSource*. Repeat the above steps and make a connection with *delegate*.

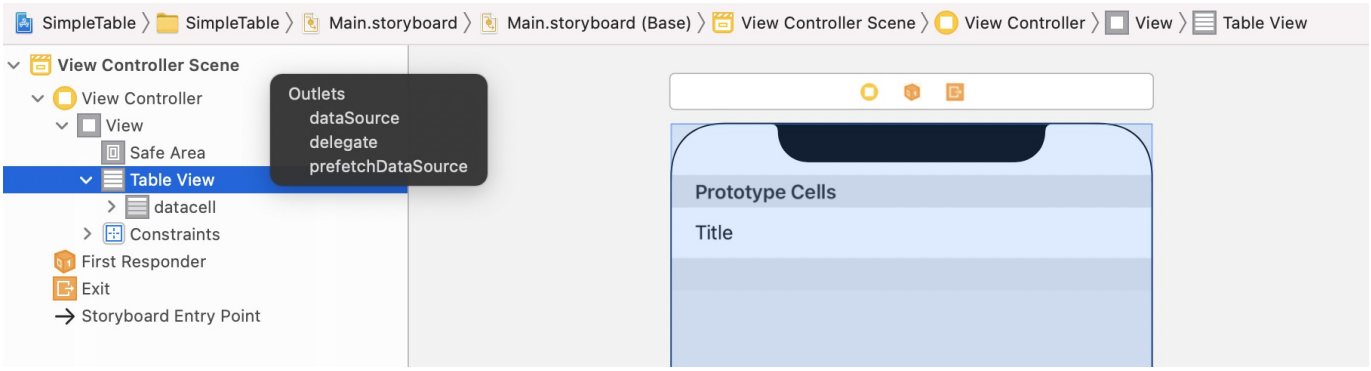


Figure 8-14. Selecting the dataSource and delegate outlets

That's it. To ensure the connections are linked properly, you can select the table view again. Click the Connections Inspector icon in the Utility area to reveal the existing connections. Alternatively, you can right-click the table view to reveal the connections as well.

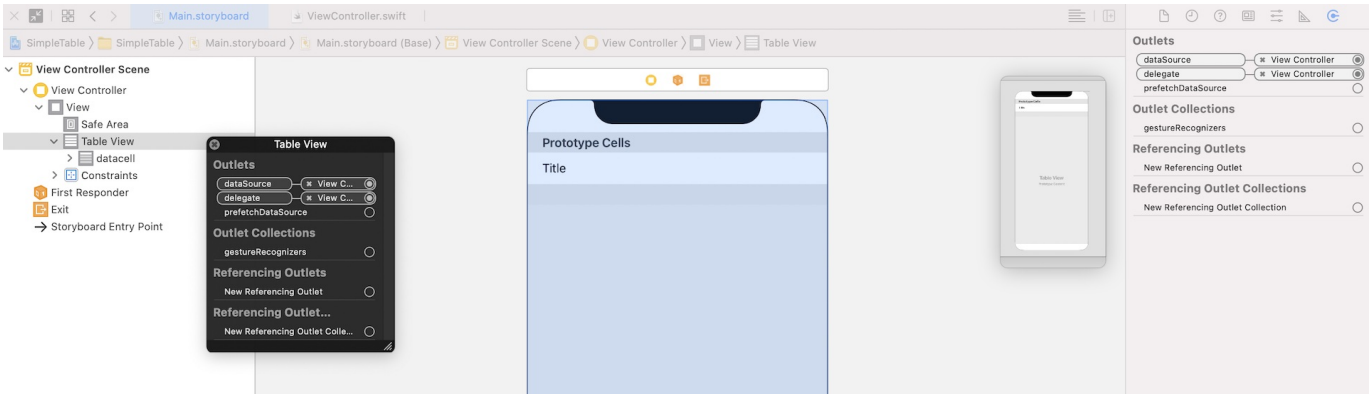


Figure 8-15. Two ways to reveal the connections

Test Your App

Finally, you're ready to test your app. Simply hit the *Run* button and load your app in the simulator. Your app should now show a list of restaurant names.

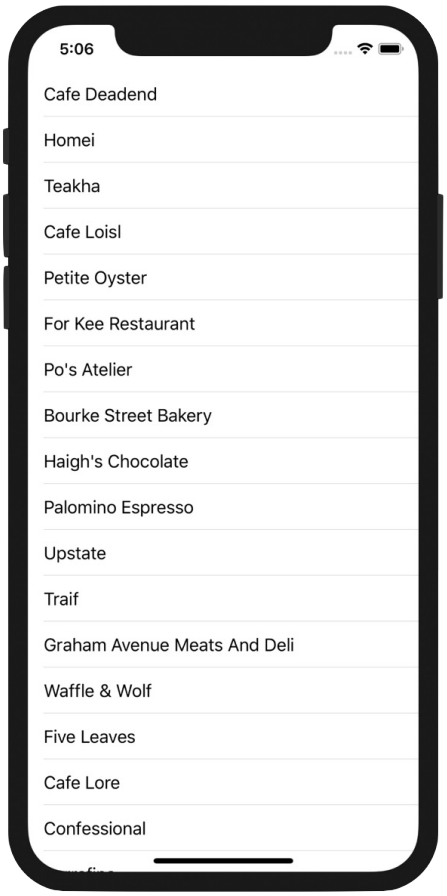


Figure 8-16. SimpleTable App

Quick tip: To scroll through the table in the iOS simulator, simply & hold on the table, drag it up and down to move it.

Add Thumbnail to Table View

Wouldn't it be great to add an image to each row? UITableView makes it extremely easy to do this. You just need to add a line of code to insert a thumbnail in each row.

First, download the sample images from http://www.appcoda.com/resources/swift53/simpletable-images1.zip. The zipped archive contains three image files. Unzip the file, and drag the images from Finder to the asset catalog (`Assets.xcassets`).

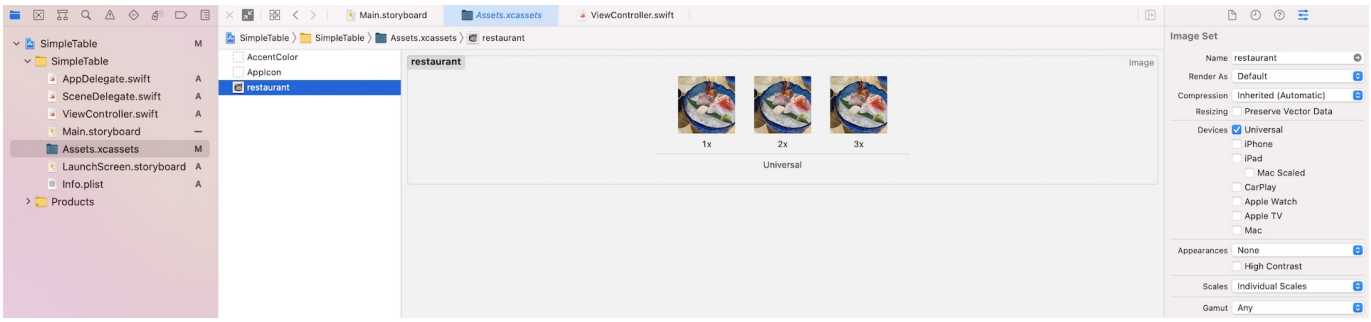


Figure 8-17. Adding images to the asset catalog

Now edit `ViewController.swift` and add the following line of code in the `tableView(_:cellForRowAtIndexPath:)` method. Put it right before `return cell` :

```
cell.imageView?.image = UIImage(named: "restaurant")
```

After the changes, the method should look like this:

```
func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath)
 -> UITableViewCell {
    let cellIdentifier = "datacell"
    let cell = tableView.dequeueReusableCell(withIdentifier: cellIdentifie
r, for: indexPath)

    // Configure the cell...
    cell.textLabel?.text = restaurantNames[indexPath.row]
    cell.imageView?.image = UIImage(named: "restaurant")

    return cell
}
```

The `UIImage` class provided by the UIKit framework lets you create images from files. It supports various image formats such as PNG, PDF and JPEG. You can pass the name of the image (file extension is optional) and the class will load the image from the asset catalog.

Earlier, we selected to use the `Basic` cell style for the table view cell. This type of cell style already comes with a default area for display images or thumbnails. This line of code instructs `UITableView` to load the image and display it in the image view of the table cell. Now, hit the "Run" button again and your SimpleTable app should display the image in each row.

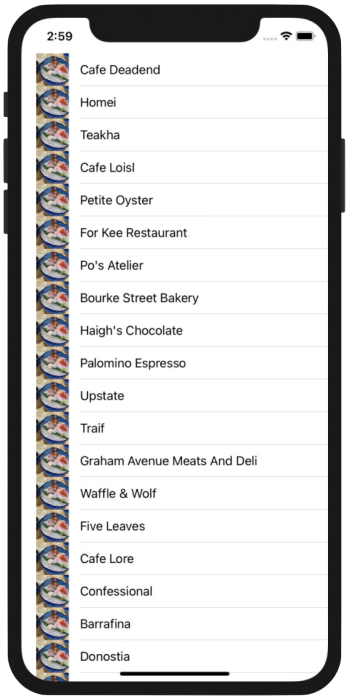


Figure 8-18. SimpleTable app with images

Hide the Status Bar

The content of table view now overlaps with the status bar. This doesn't look good. An easy remedy is to hide the status bar. You can control the appearance of the status bar on a per view controller basis. If you don't want to show the status bar in a particular view controller, simply add the following lines of code before the last closing curly bracket (`}`):

```
override var prefersStatusBarHidden: Bool {
    return true
}
```

The `prefersStatusBarHidden` property indicates whether the view controller should hide the status bar. By default, the value is set to `false`. To hide the status, we have to override the default value and set it to `true`. Insert the code to the

`ViewController` class and test the app again. You should have a full-screen table view without status bar.

Exercise #1

The demo app displays the same image for all table cells. Try to tweak the app such that it shows a different image in each cell (hint: Create another array for the images). You can download the image pack from http://www.appcoda.com/resources/swift4/simpletable-images-2.zip. Figure 8-19 shows the resulting screen.

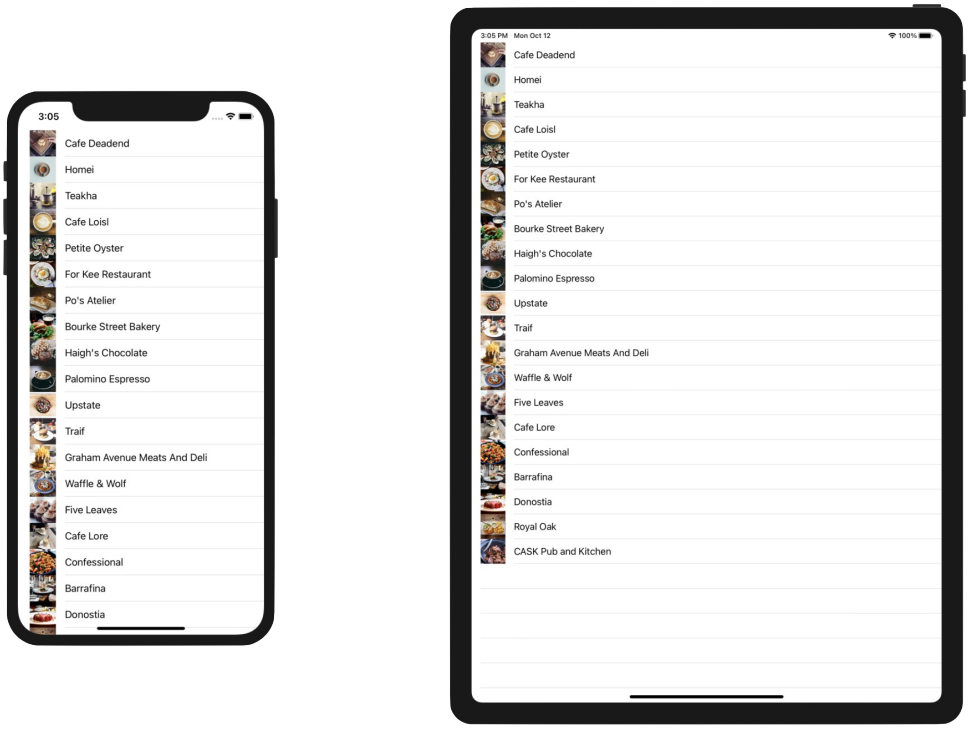


Figure 8-19. Displaying different images in the demo app

Quick note: In case you do not know how to complete the exercise, no worries. I will go through it with you in the next chapter or you can download the solution at the end of the chapter.

Credit: The images used in this demo project are provided by unsplash.com.

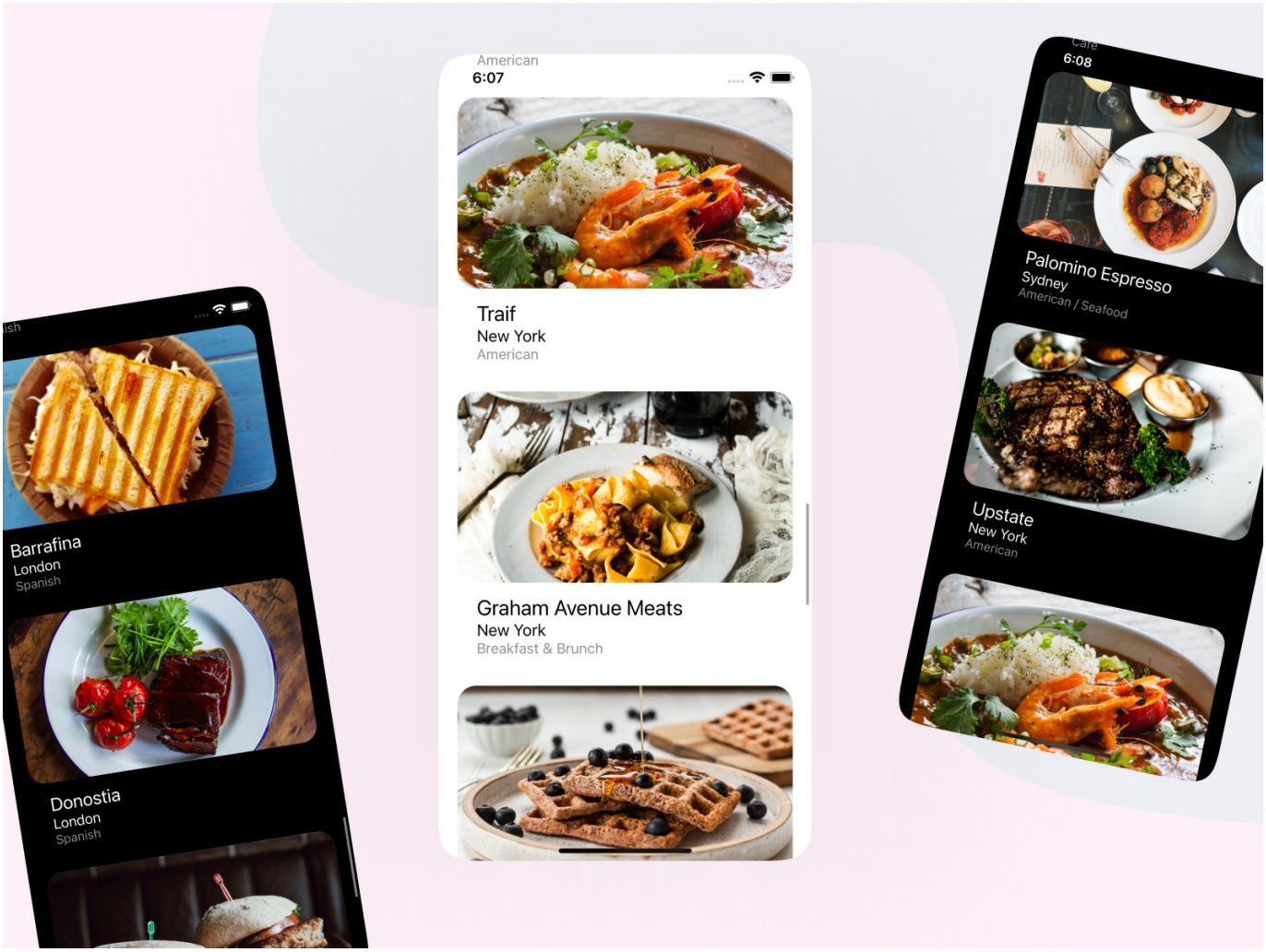
Summary

Table view is one of the most commonly used elements in iOS programming. If you thoroughly understood the materials and built the app, you should have a good idea of how to create your own table view.

I tried to keep everything simple in the demo app. In a real world app, the table data is generally not 'hard-coded'. Usually, it's loaded from a file, database or somewhere else. We'll talk about that later. Meanwhile, make sure you thoroughly understand how the table view works. Otherwise, go back to the beginning and study the chapter again.

For reference, you can download the complete Xcode project from http://www.appcoda.com/resources/swift53/SimpleTable.zip, and the solution to exercise #1 from http://www.appcoda.com/resources/swift53/SimpleTableExercise.zip.

Chapter 9
Customize Table Views Using Prototype Cell, Diffable Data Source and Dark Mode



> 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. And, I will show a modern way to handle the data source of a table view. There are a number of changes and enhancements we are going to work on:

- Rebuild the same app using `UITableViewController` instead of `UITableView`
- Use the new `UITableViewDiffableDataSource` to configure the data of the table view
- 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
- Test the app in dark mode

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` and `UITableViewDiffableDataSource` to create a table view app. 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.

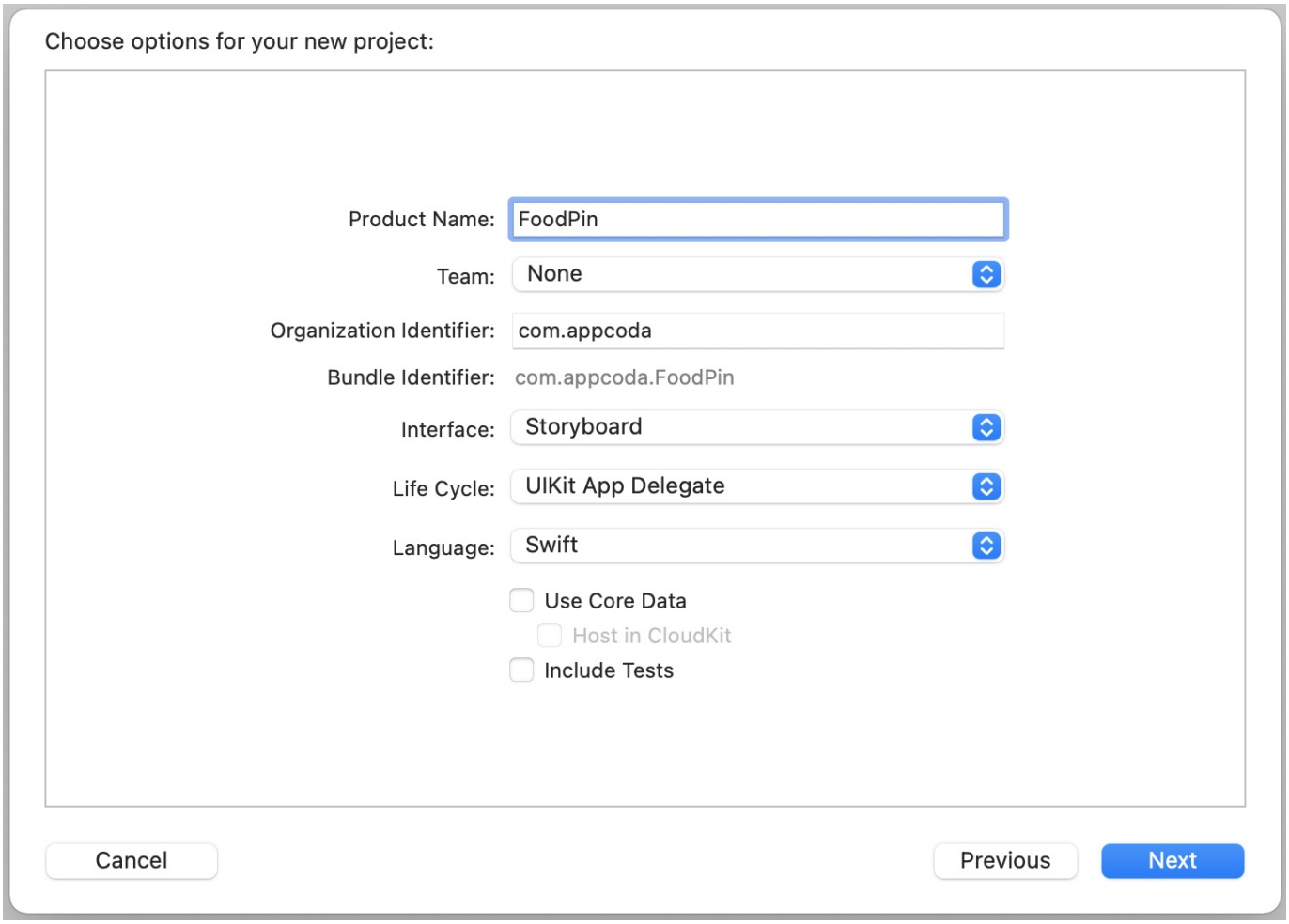
The `UITableViewDiffableDataSource` class was first introduced in iOS 13, which is a modern way to supply a table view with the data to display. You can still use the traditional way, which was discussed in chapter 8, to populate data into a table. This new approach replaces the `UITableViewDataSource` protocol and will make your life much easier to cope with data changes in a table view. As more devices are adopting iOS 13 (or up), the `UITableViewDiffableDataSource` approach will become the standard implementation of `UITableView`.

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

Building a Table View App Using UITableViewController and UITableViewDiffableDataSource

First, let's see how to use `UITableViewController` to re-create the same SimpleTable app. Fire up your Xcode, and create a new project using the *App* template. Name the project *FoodPin* and fill in all the required options for the Xcode project, just like what you did in the previous chapter.

> **Quick note:** We're building a real app, so let's give it a better name. Feel free to use other names if you want. Also, make sure you use your own organization identifier. Otherwise, you can't test your app on a real iPhone.



Once you create the Xcode project, select `Main.storyboard` and jump to the Interface Builder editor. As usual, a default view controller is generated by Xcode.

This time, we will not use the default controller. Select the view controller, and press the `delete` key to delete it. The view controller is associated with `ViewController.swift`. We do not need it either. In the Project Navigator, select the file and hit the `delete` button. Select "Move to Trash" when prompted. This will completely delete the file.

Go back to Interface Builder. At the time of this writing, iPhone 12 was just announced. We will design the UI using iPhone 12, so select *View as:* and change the device to *iPhone 12*.

Next, open the Object library and search for "Table". Then drag a Table View Controller (i.e. UITableViewController) from the library and place it in the storyboard. You have to designate that controller as the initial view controller. This tells iOS that the table view controller is the first view controller to load. All you need to do is go to the Attributes inspector, and tick the `Is Initial View Controller` option. You'll then see an arrow pointing to the table view controller (see figure 9-2).

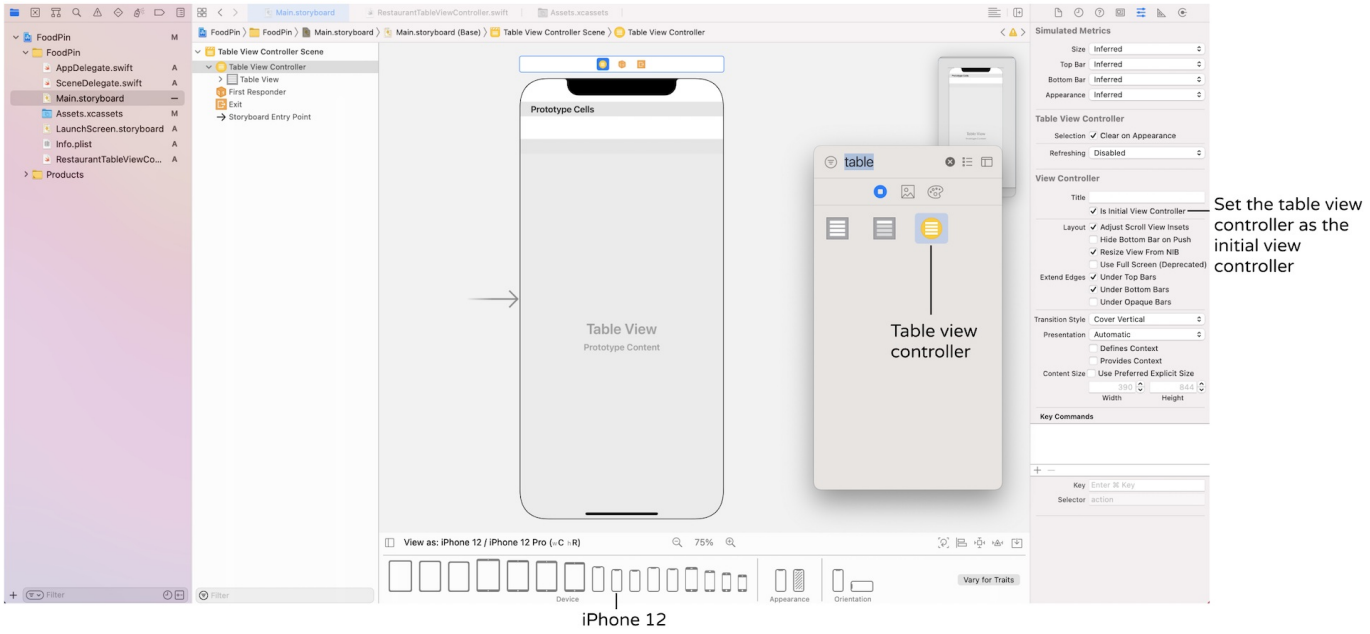


Figure 9-2. Drag a Table View Controller from the Object library and set it as the initial view controller

We haven't inserted any data into the table yet. If you compile and run the app now, you'll end up with a blank table.

By default, the table view controller is associated with the `UITableViewController` class (Go to the Identity inspector to take a look). In order to populate our own data, we have to associate it with our own class.

Go back to the Project Navigator and right-click the FoodPin folder. Select "New Files..." to create a new file.

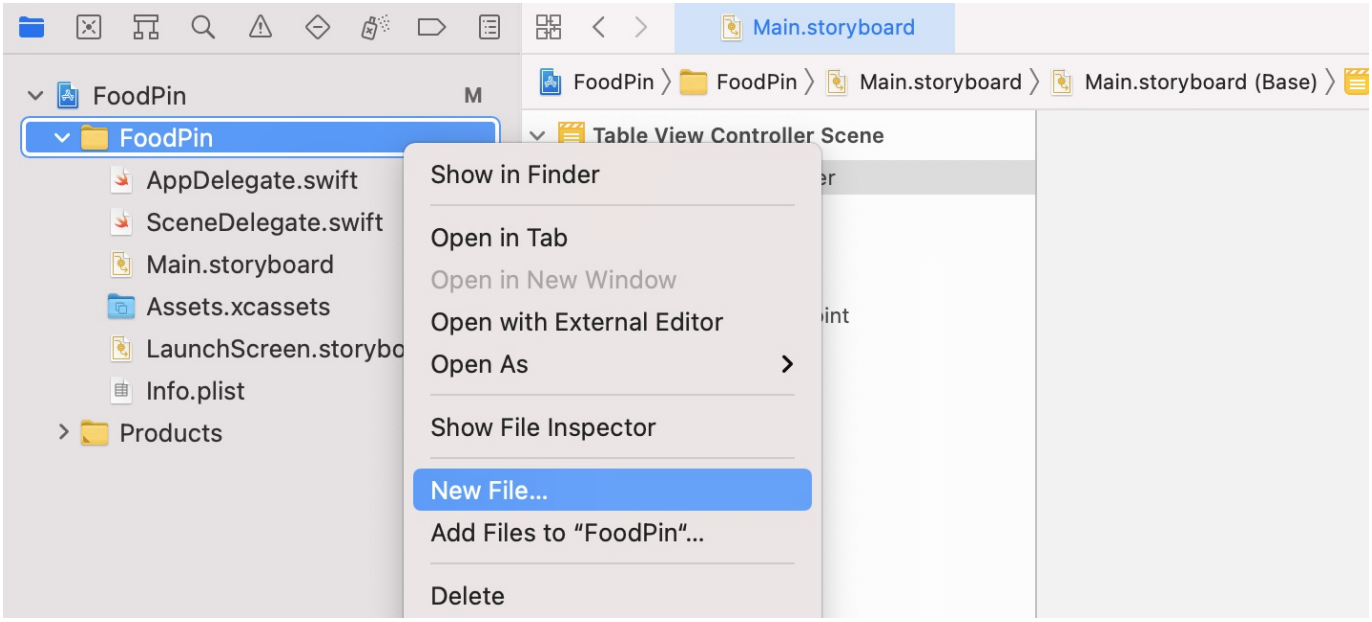
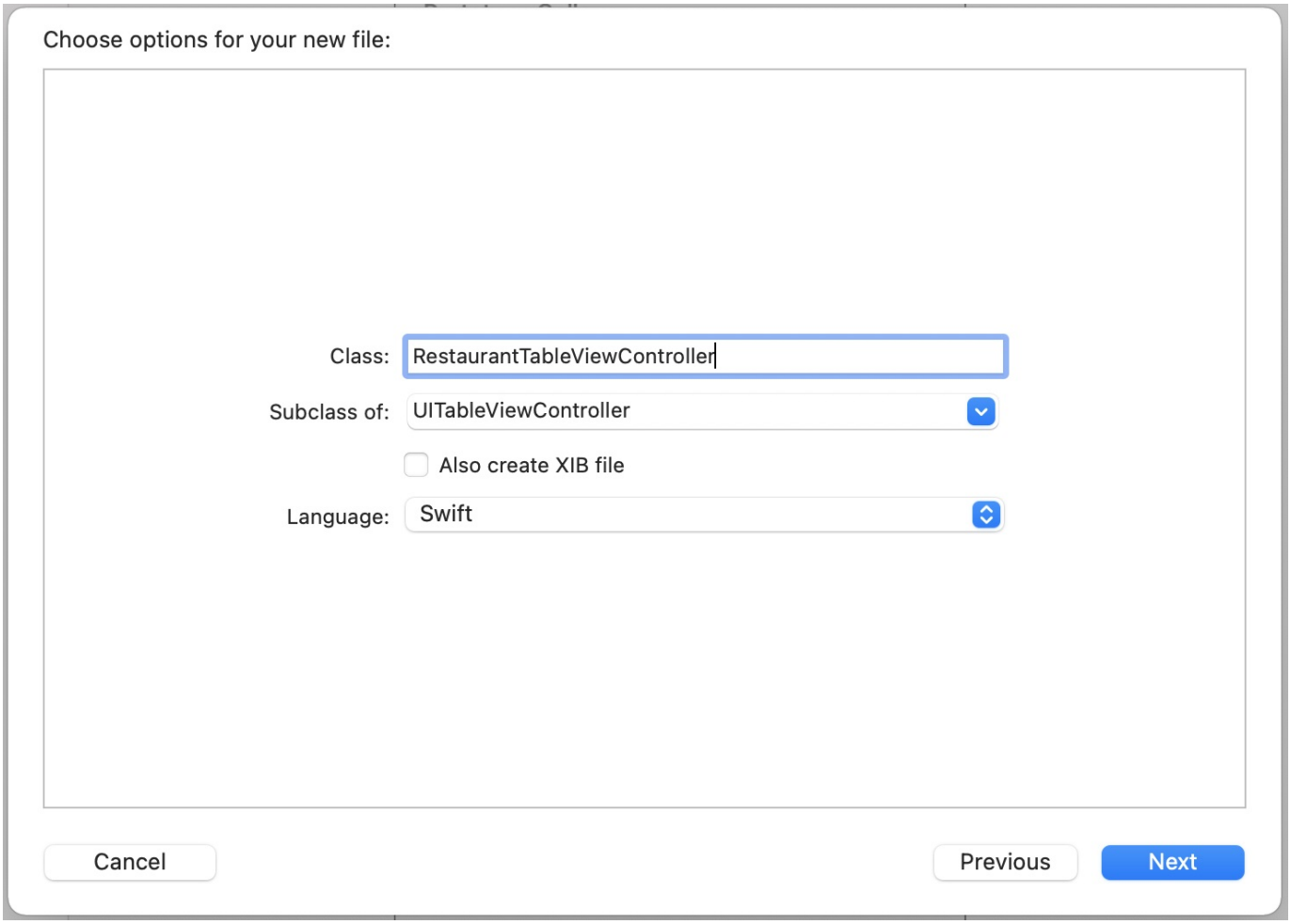


Figure 9-3. Create a new file

Under the iOS category, choose *Cocoa Touch Class* as the template and click `Next`. Name the new class `RestaurantTableViewController`. Because we're working with a table view controller, change the value of "Subclass of" to `UITableViewController`. Keep the rest of the values intact, click "Next" and save it to

the project folder. You should see the `RestaurantTableViewController.swift` file in the project navigator.



Superclass and Subclass

If you're new to programming, you may wonder what a subclass is. Swift is an object oriented programming (OOP) language. In OOP, a class can be inherited by another class. In the example, the `RestaurantTableViewController` class inherits from the `UITableViewController` class. It inherits all the states and functionalities provided by the `UITableViewController` class. The `RestaurantTableViewController` class is known as a subclass (or child class) of `UITableViewController`. In other words, the `UITableViewController` class is referred as the superclass (or parent class) of `RestaurantTableViewController`.

The table view controller in the storyboard has no idea about the `RestaurantTableViewController` class we just created. We have to assign the table view controller with the new custom class. Go to `Main.storyboard` and select the table view controller. In the Identity inspector, set the custom class to `RestaurantTableViewController` and hit the return key to confirm the change. Now we have established a relationship between the table view controller in the storyboard and the new class.

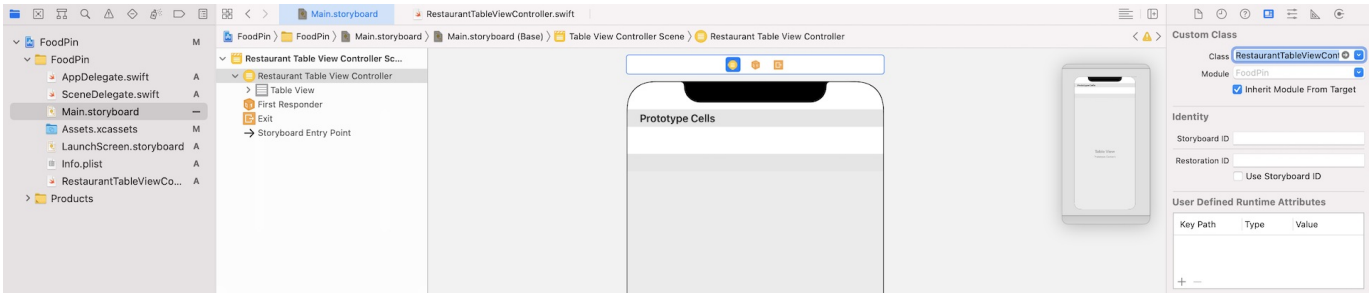


Figure 9-5. Set the custom class of the Table View Controller

There is one more thing to configure for the table view. In the document outline, expand *Table View* and select the table view cell. In the Attributes inspector, change the cell style to `Basic` and set the identifier to `datacell`. This is pretty much the same as what we have done in the previous chapter.

Okay, the user interface is ready. Let's move onto the code. Select the `RestaurantTableViewController.swift` file in the Project Navigator. By default, Xcode still uses the traditional way to handle the data source of the table view. If you look into the `RestaurantTableViewController.swift` file, you should find these lines of code:

```
// MARK: - Table view data source

override func numberOfSections(in tableView: UITableView) -> Int {
    // #warning Incomplete implementation, return the number of sections
    return 0
}

override func tableView(_ tableView: UITableView, numberOfRowsInSection se
ction: Int) -> Int {
    // #warning Incomplete implementation, return the number of rows
    return 0
}
```

As mentioned earlier, we will use a new approach (i.e. `UITableViewDiffableDataSource`) to populate the table data. You can delete the code above and remove those commented code. After the clean up, your `RestaurantTableViewController` class should be like this:

```
class RestaurantTableViewController: UITableViewController {

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

}
```

Now we are ready to adopt the new approach to create the same table view as shown in the previous chapter. First, declare an instance variable, which is used for holding the table data.

```
var restaurantNames = ["Cafe Deadend", "Homei", "Teakha", "Cafe Loisl", "P
etite Oyster", "For Kee Restaurant", "Po's Atelier", "Bourke Street Bakery"
, "Haigh's Chocolate", "Palomino Espresso", "Upstate", "Traif", "Graham Av
enue Meats", "Waffle & Wolf", "Five Leaves", "Cafe Lore", "Confessional",
"Barrafina", "Donostia", "Royal Oak", "CASK Pub and Kitchen"]
```

Previously, we implemented the following required methods of the `UITableViewDataSource` protocol to provide the table data:

- func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int
- func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell

The new approach no longer requires us to deal with the `UITableViewDataSource` protocol and all the methods above. Instead, we have to create a `UITableViewDiffableDataSource` object to configure the table view cell and then create a `NSDiffableDataSourceSnapshot` object to tell the table view what data to display.

`UITableView` supports multiple sections, thus the `UITableViewDiffableDataSource` object also requires us to specify the number of sections and the cell configuration of each section. For the demo app, we only have one section. So, declare an enum variable in `RestaurantTableViewController` like this:

```
enum Section {
    case all
}
```

We use `enum` to define a new `Section` type. In the enumeration, it only has one case because the table has a single section. I name the case `all`, which means all records. But you can actually name it to whatever name you like. Anyway, let's keep it to `all`.

Now that we have defined the section, the next step is to create an instance of `UITableViewDiffableDataSource`. We will write a new function called `configureDataSource()` for this purpose. Insert the following code inside `RestaurantTableViewController`:

```
func configureDataSource() -> UITableViewDiffableDataSource<Section, String
> {

    let cellIdentifier = "datacell"

    let dataSource = UITableViewDiffableDataSource<Section, String>(
        tableView: tableView,
        cellProvider: {  tableView, indexPath, restaurantName in
            let cell = tableView.dequeueReusableCell(withIdentifier: cellI
dentifier, for: indexPath)

            cell.textLabel?.text = restaurantName
            cell.imageView?.image = UIImage(named: "restaurant")

            return cell
        }
    )

    return dataSource
}
```

The function looks a bit complicated but if you study the code, some of the lines should be familiar to you. As you are pretty new to Swift, let me walk you through the code line by line. The first line is the function declaration. The symbol (->) means return and the type that follows is the type of returned object.

```
func configureDataSource() -> UITableViewDiffableDataSource<Section, String
> {
  ...
}
```

So, in the code above, we create a function named `configureDataSource()` which returns an instance of `UITableViewDiffableDataSource<Section, String>`.

What's `Section` and `String` here?

`UITableViewDiffableDataSource` is a generic object which is capable to handle different types of section and item of a table view. The type `Section` and `String` inside the angle bracket indicated that we use the `Section` type for the table section and the `String` type for the table cell data. The reason why `String` type is used is that the restaurant name is of the type `String`.

Inside the `configureDataSource()` function, we create an instance of `UITableViewDiffableDataSource`. When initializing the object, it is required to provide the instance of table view that the data source is connected, plus the cell provider. Obviously, the data source will connect to the `tableView` of `RestaurantTableViewController`. The `cellProvider` parameter is where you set up each cell of the table view. In the closure of the cell provider, you have three parameters including `tableView`, `indexPath`, and `restaurantName`. The `restaurantName` here is the restaurant name of the current cell.

The code inside `cellProvider` is pretty much like those in the `tableView(_:cellForRowAt)` function you wrote in the previous chapter:

```
override func tableView(_ tableView: UITableView, cellForRowAt indexPath:
IndexPath) -> UITableViewCell {

    let cellIdentifier = "datacell"
    let cell = tableView.dequeueReusableCell(withIdentifier: cellIdentifie
r, for: indexPath)

    // Configure the cell...
    cell.textLabel?.text = restaurantNames[indexPath.row]
    cell.imageView?.image = UIImage(named: "restaurant")

    return cell
}
```

The core difference is that we have to figure out the restaurant name from the `restaurantNames` array using index path. For `cellProvider`, it already passes us the restaurant name to display.

Now that we have created the function for configuring the data source, declare the following variable in `RestaurantTableViewController` to use it:

```
lazy var dataSource = configureDataSource()
```

You may wonder why we add a modifier `lazy` before the variable declaration. If you try to omit the `lazy` keyword, Xcode will show you the following message:

"Cannot use instance member 'configureDataSource' within property initializer; property initializers run before 'self' is available"

In this case, you must declare the variable with the `lazy` modifier because its initial value cannot be retrieved until after the instance initialization completes.

Okay, we have configured the data source of the table view. That said, your app is not yet ready to display the restaurants. We still have one more step to implement before the data shows up in the table view. Update the `viewDidLoad` function like this:

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

    tableView.dataSource = dataSource

    var snapshot = NSDiffableDataSourceSnapshot<Section, String>()
    snapshot.appendSections([.all])
    snapshot.appendItems(restaurantNames, toSection: .all)

    dataSource.apply(snapshot, animatingDifferences: false)
}
```

We first assign our custom `dataSource` to the table view's data source. The final step is to create a snapshot of the data to display in the table view. The snapshot is an instance of `NSDiffableDataSourceSnapshot`. This line of code creates an empty snapshot using the section identifier type `Section` and item identifier type `String`:

```
var snapshot = NSDiffableDataSourceSnapshot<Section, String>()
```

With the empty snapshot, we call `appendSections` to add a section to the snapshot. Then we call `appendItems` to add all items in the `restaurantNames` array to the `.all` section. Lastly, we apply the snapshot to the data source.

Before you run the app, download the image pack from http://www.appcoda.com/resources/swift53/simpletable-images1.zip and drag all images to the `Assets.xcassets`. Now, hit the "Run" button and your FoodPin app should look the same as the one we built earlier.

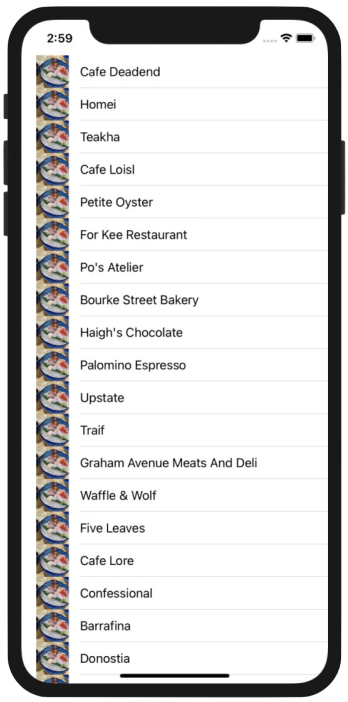


Figure 9-6. A simple table app

This is how we use the `UITableViewDiffableDataSource` approach to present data in a table view. To recap, here is the steps we used with this new approach:

1. Create an enum which indicates the table sections.
2. Create a `UITableViewDiffableDataSource` object to connect with your table and provide the configuration of the table view cells.
3. Assign the diffable data source to your table view.
4. Generate the current state of the table data by creating a snapshot
5. Call the `apply()` function of the data source to populate the data

Comparing with the traditional approach, which adopt the `UITableViewDataSource` protocol, you may wonder which approach is preferred. In general, use the diffable data source approach to create your table view. While the code may be more difficult to understand, this modern way of table view implementation will save

you a lot of work when you need to update or change the table data. And, I forgot to mention this approach can automatically animate the data change. You will understand what this means in later chapters.

Display Different Thumbnails

Did you go through the exercise in the previous chapter? I hope you've put your effort in it. In this section, we will modify the existing app to display different restaurant images. First, download another image pack from http://www.appcoda.com/resources/swift53/simpletable-images2.zip, and unzip it. Add all the images into the asset catalog (i.e. Assets.xcasset). The pack includes some food and restaurant images. If you like, you're free to use your own images.

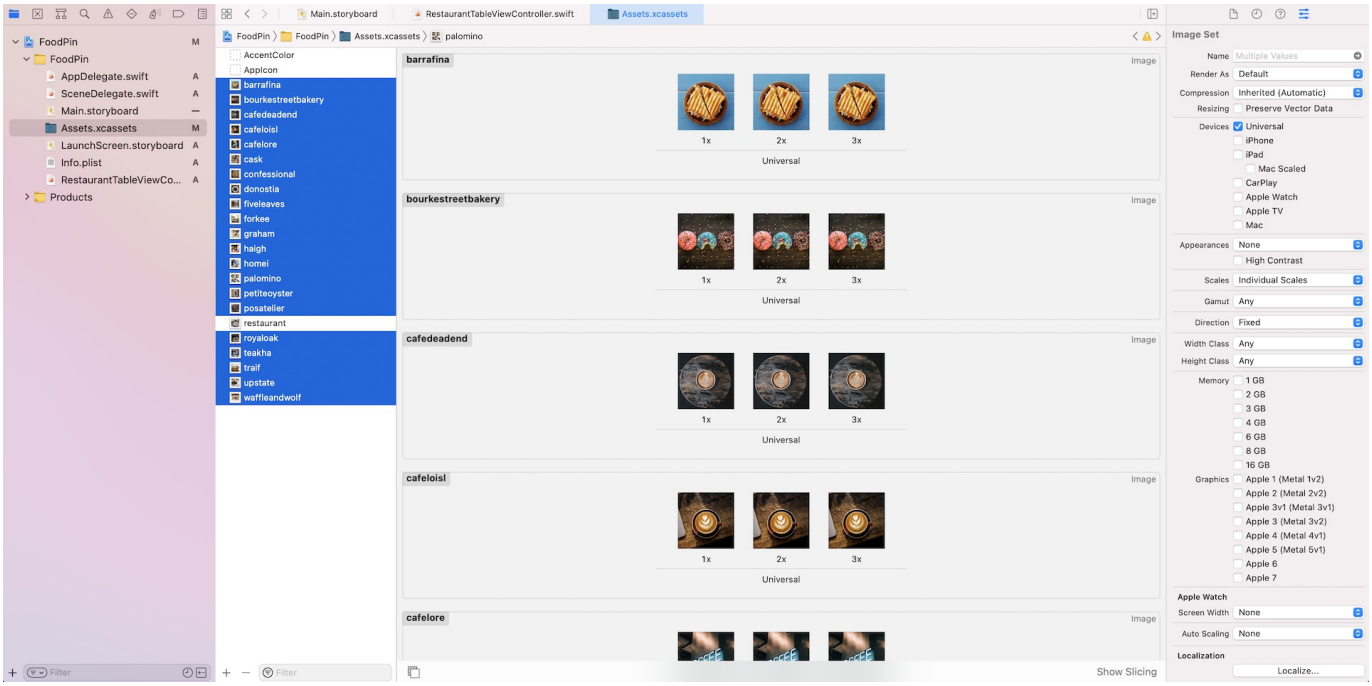


Figure 9-7. Adding restaurant images to the asset catalog

First, let's declare a new array named `restaurantImages` for holding the file name of the restaurant images. Insert the following line of code in the `RestaurantTableViewController` class:

```
var restaurantImages = ["cafedeadend", "homei", "teakha", "cafeloisl", "petiteoyster", "forkee", "posatelier", "bourkestreetbakery", "haigh", "palomino", "upstate", "traif", "graham", "waffleandwolf", "fiveleaves", "cafelore", "confessional", "barrafina", "donostia", "royaloak", "cask"]
```

Note that the order of images are aligned with that of the `restaurantNames`.

Before we continue to modify the code, let's revisit the code for displaying the thumbnails in table row:

```
cell.imageView?.image = UIImage(named: "restaurant")
```

The above line of code, inside the `cellProvider` closure of the `configureDataSource` function, tells `UITableView` to display `restaurant.jpg` in each cell. In order to display a different image, we need to modify this line of code.

As explained before, one of the parameters provided by the `cellProvider` closure is `indexPath`. With the index path, you can use `indexPath.row` to find out which row is now being processed.

Therefore, in order to load the corresponding image of the restaurant, change the line of code in the `configureDataSource` function from:

```
cell.imageView?.image = UIImage(named: "restaurant")
```

to:

```
cell.imageView?.image = UIImage(named: self.restaurantImages[indexPath.row
])
```

After saving the changes, try to run your app again. Each restaurant should display its own image.

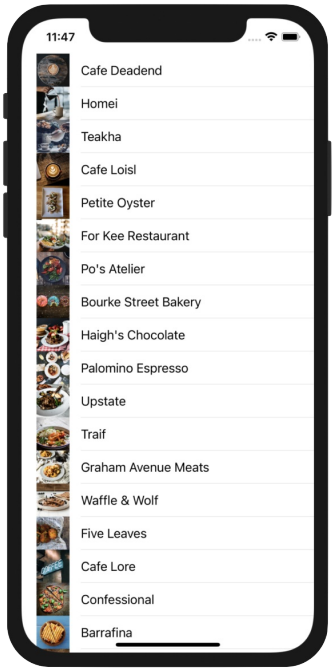
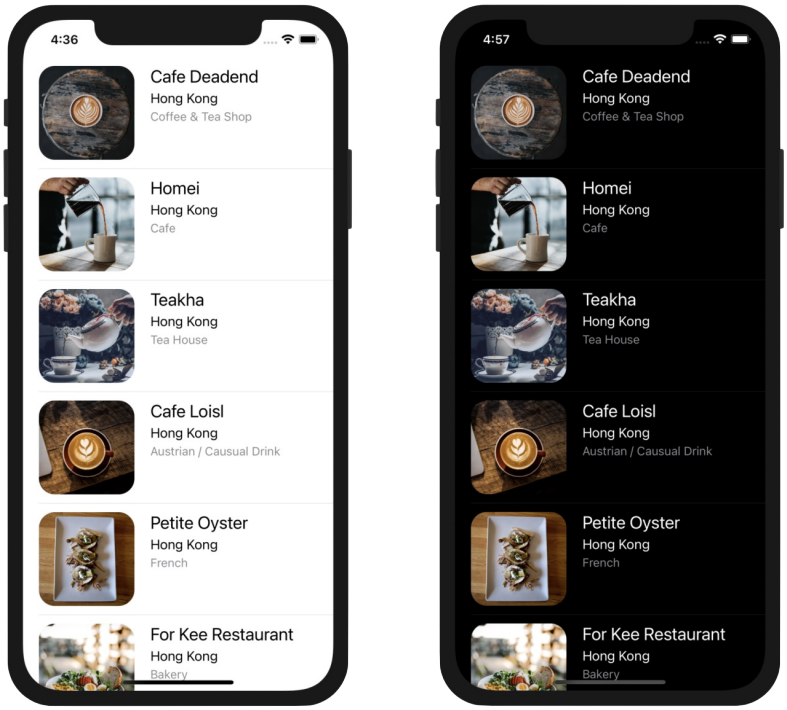


Figure 9-8. Loading the table view with different restaurant images

Customize Table View Cells

So far we use the *basic* cell type for the table view. We're going to make it even better by customizing the table cells. Here is what we are going to do. We will redesign the table view cell such that it will be a bit bigger and display a larger restaurant image. Also, we will show more information about the restaurant such as location and type. On top of that, we will round the corners of the images. To give you a better idea about how the cell is customized, take a look at figure 9-9.

The cell looks awesome, right? And, I forgot to mention that the table view will also look great in dark mode.



Designing Prototype Cells in Interface Builder

The beauty of prototype cells is that it allows developers to customize the cell right inside the table view controller. To build a custom cell, you simply add other UI controls (e.g. labels, image views) to the prototype cell.

First, let's change the style of the cell. You can't customize the cell when it's set to the `Basic` style. Select the prototype cell and change the style from `Basic` to `Custom` in the Attributes inspector.

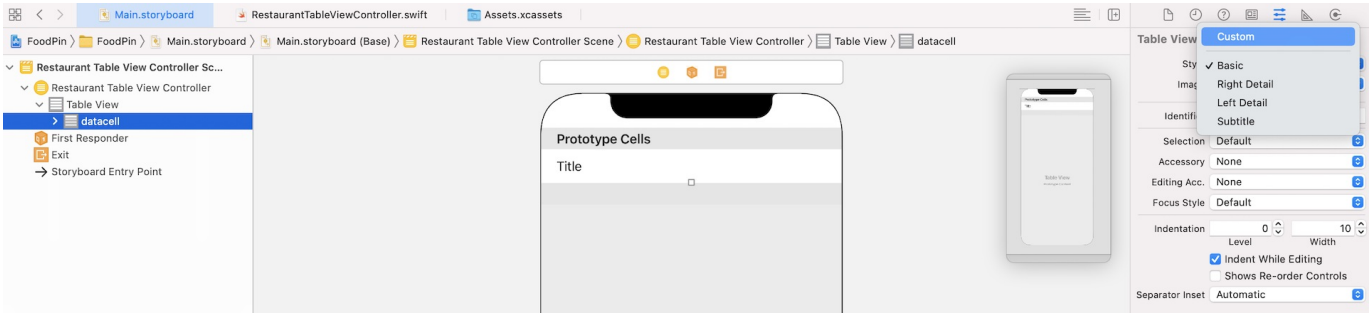


Figure 9-10. Changing the cell style from Basic to Custom

In order to accommodate a larger thumbnail, we have to make the cell a little bit bigger. You'll need to change the row height of the prototype cell. Select the prototype cell and change the row height to `140`.

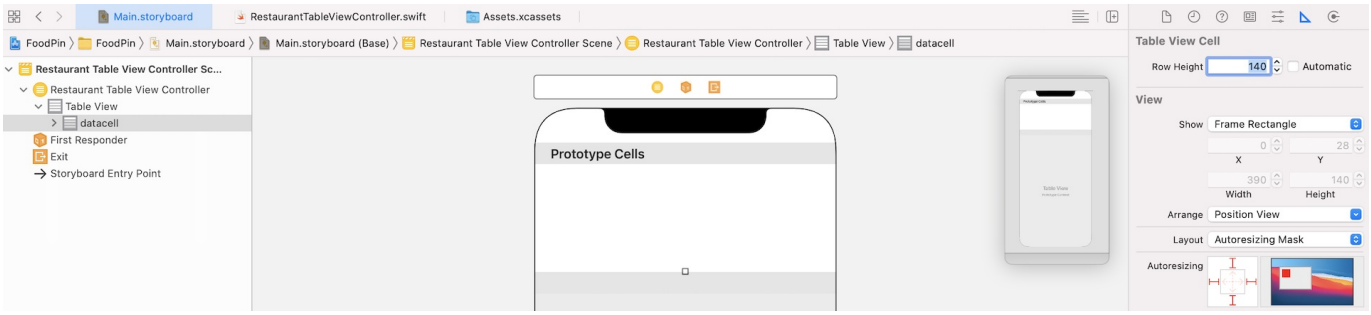


Figure 9-11. Changing the row height of the prototype cell

After altering the row height, open the Object library and drag an Image View object to the prototype cell. Adjust its size to 120x118 points. You can select the image view, click the Size inspector and change the attribute of *X*, *Y*, *Width* and *Height* like that shown in figure 9-12. Once you set the size of the image view, switch over to the Attributes inspector and set the *Content Mode* of the image view to `Aspect Fill`. Since our image view is not a perfect square, the image will look better in this mode.

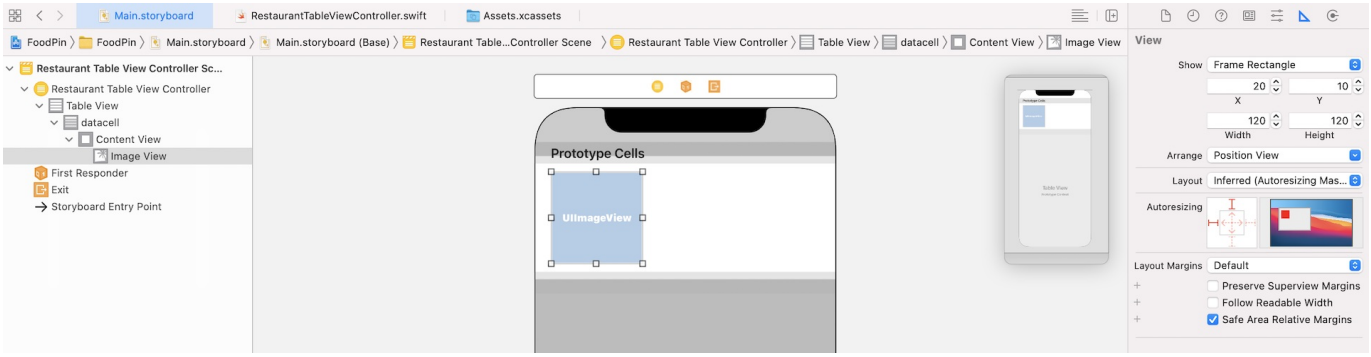


Figure 9-12. Adding an image view to the prototype cell

Next, we will add three labels to the prototype cell:

- **Name** - for restaurant name
- **Location** - for restaurant location (e.g. New York)
- **Type** - for restaurant type (e.g. tea house)

To add a label, drag a Label object from the Object library to the cell. Name the first label *Name*. Instead of using the system font for the label, we'll use a text style. I will explain to you the difference of a fixed font and a text style in later chapters. For now, just go to the Attributes inspector, and change the font to `Text Styles - Title 2`. To change the font to the *Title 2* style, click the `T` button of the font option and change the font from *System* to *Title 2*.

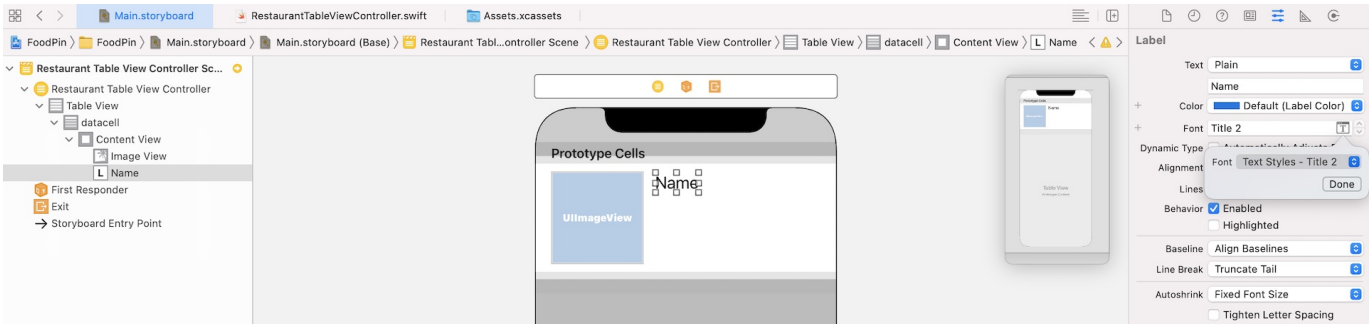


Figure 9-13. Adding name label to the prototype cell

Drag another label to the cell and name it *Location*. Change the font to `Text Styles - Body`. Lastly, create another label and name it *Type*. For this label, change the font style to `Text Styles - Subhead` and set the font color to `System Gray Color`.

I've covered stack views in chapter 6. Not only can you use stack views in a view controller, you can also apply stack views to layout the components in a prototype cell. Thus, instead of defining the layout constraints for each of the labels and the image view, we will use stack views to group them together.

First, hold the `command` key, and select the three labels. Click the `Embed in` button in the layout bar and then choose *Stack View* to embed them in a vertical stack view. Go to the Attributes inspector, and change the spacing of the stack view to `4`. This will add a space between the labels.

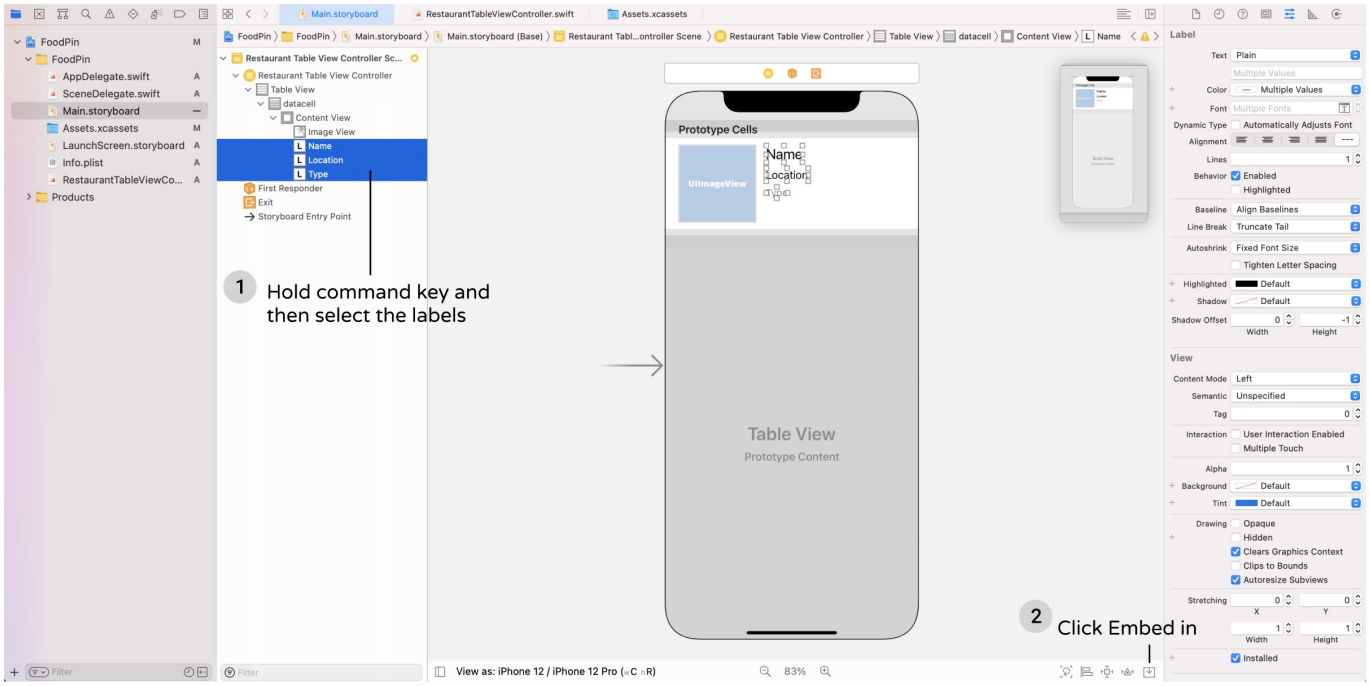


Figure 9-14. Embedding the labels using stack view

Next, select both the image view and the stack view that we have just created. Click the `Embed in` button and choose *Stack View* to embed them into a horizontal stack view. You can go up to the Attribute inspector, and change the spacing option to `20`. Awesome, right? You can nest stack views to create complex layouts.

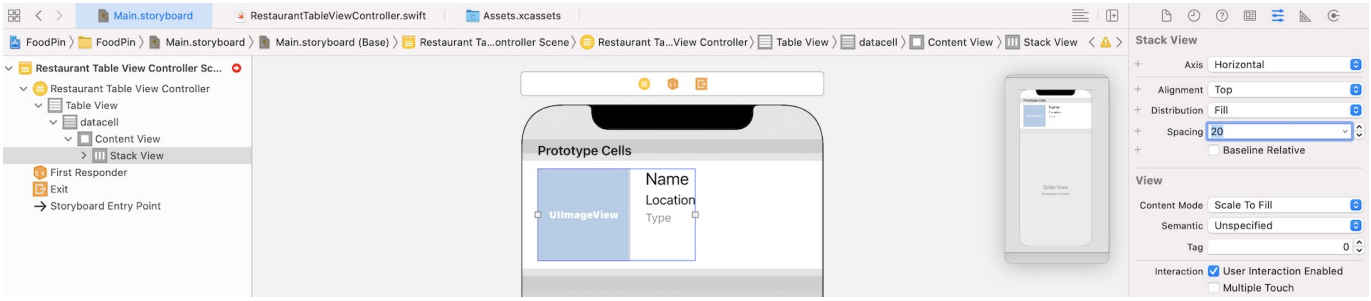


Figure 9-15. Combining the label stack view and the image view into a horizontal stack view

Again, it doesn't mean you do not need to use auto layout. We still need to define the layout constraints for the stack view. Figure 9-16 depicts the layout requirements of the cell.

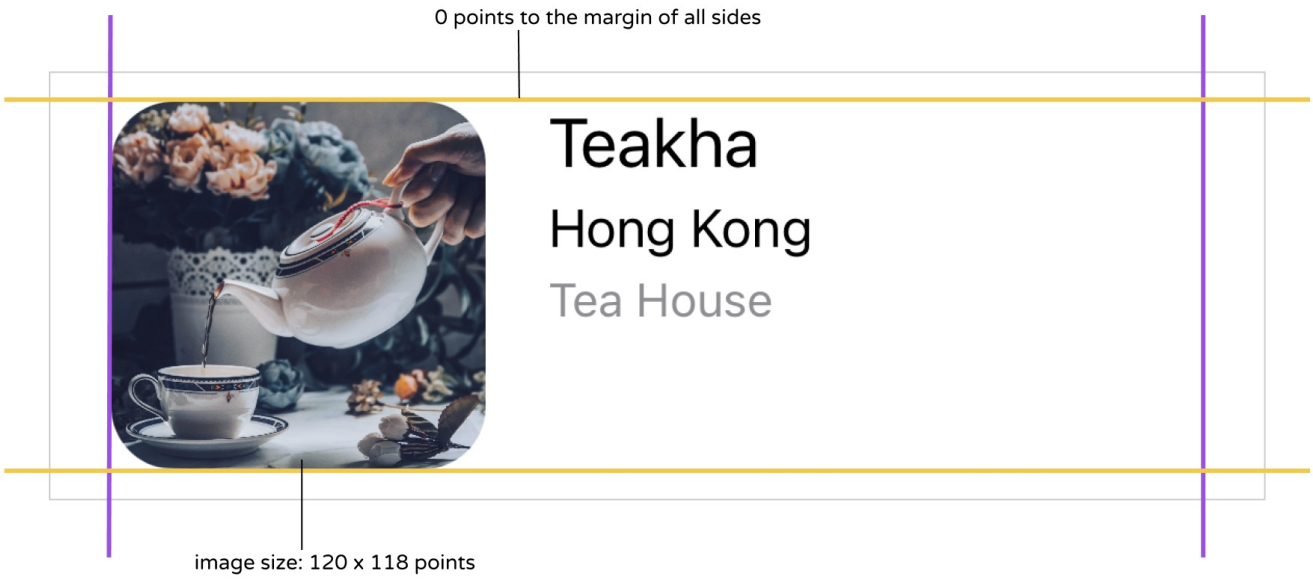


Figure 9-16. The layout requirement of the prototype cell

In brief, we want the cell content (i.e. the stack view) is confined to the viewable area of the cell. This is why we're going to define a spacing constraint for each side of the stack view. On top of this, the size of image view should be fixed to 120x118 points.

Now select the stack view and click the *Add New Constraints* button in the layout bar. Set the values of the top, left, bottom and right sides to 0, 0, 0 and 0 respectively. Also, make sure the *Constrain to margins* option is checked.

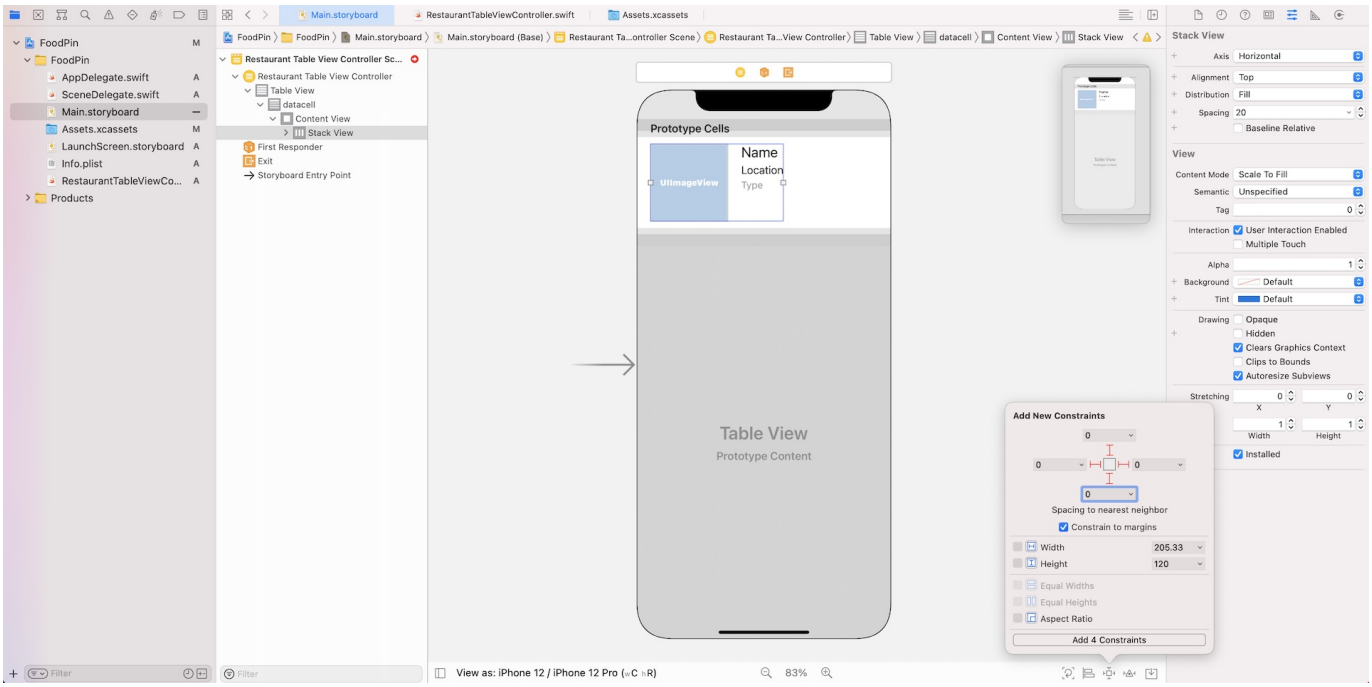


Figure 9-17. Adding spacing constraints for the stack view

Once you add the 4 constraints, the stack view should be resized automatically. Next, we need to add a couple of constraints to control the size of the image view. In the document outline, drag horizontally from the image view to itself. In the pop over menu, hold shift key and choose both "width" and "height" options. Press enter to confirm. This ensures the size of the image view is fixed.

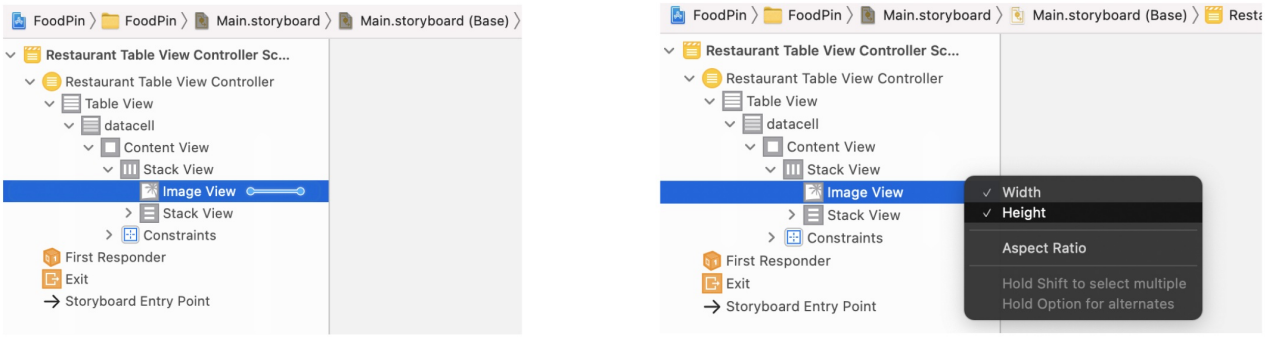


Figure 9-18. Adding the width and height constraints for the image view

Cool! You've completed the layout of the prototype cell. Let's move on and write some code.

Creating a Class for the Custom Cell

So far, we've designed the table cell. But how can we change the label values of the prototype cell? These values are supposed to be dynamic. By default, the prototype cell is associated with the `UITableViewCell` class. You can select the `datacell` in document outline and then go to the Identity inspector to reveal the association.

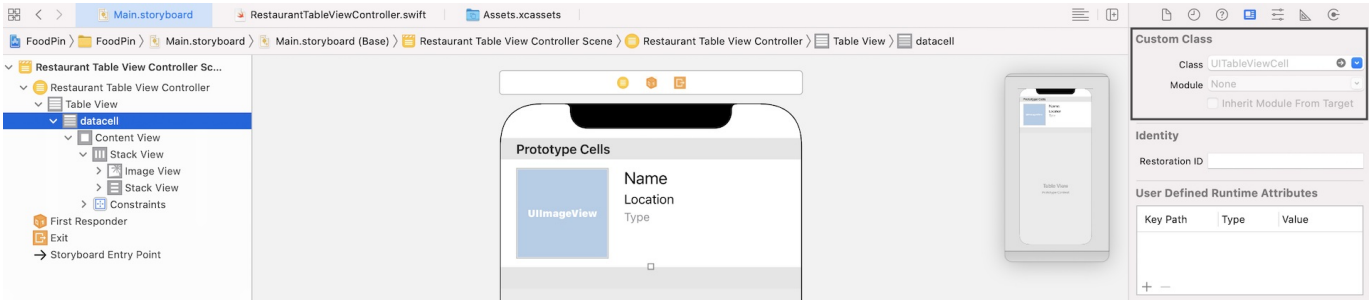


Figure 9-19. The default class of a table view cell is set to UITableViewCell

In order to update the cell data, we're going to create a new class, which extends from `UITableViewCell`, for the prototype cell. We want to create a custom version of `UITableViewCell` that associates with the custom cell in Interface Builder.

As usual, right click the "FoodPin" folder in Project Navigator and select "New File...". After selecting the option, Xcode prompts you to select a template. Since we need to create a new class for the custom table view cell, select "Cocoa Touch Class" and click "Next". Fill in `RestaurantTableViewCell` as the class name and set the value of "Subclass of" to `UITableViewCell`.

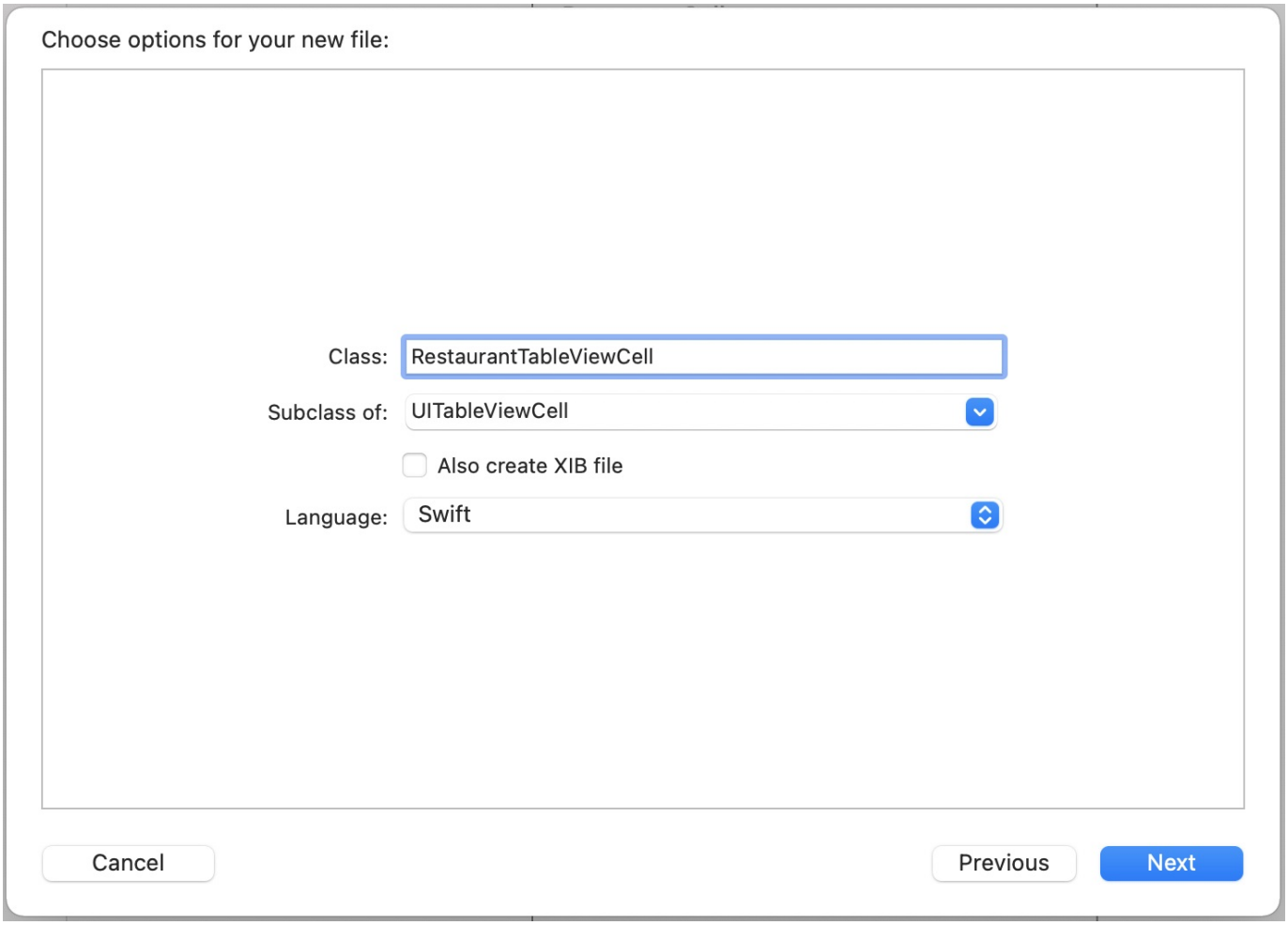


Figure 9-20. Creating a new class for the custom table view cell

Click "Next" and save the file in the FoodPin project folder. Xcode should create a file named `RestaurantTableViewCell.swift` in the Project Navigator.

Next, declare the following outlet variables in the `RestaurantTableViewCell` class:

```
@IBOutlet var nameLabel: UILabel!
@IBOutlet var locationLabel: UILabel!
@IBOutlet var typeLabel: UILabel!
@IBOutlet var thumbnailImageView: UIImageView!
```

The `RestaurantTableViewCell` class serves as the view model of the custom cell. In the cell, we have 4 properties that are changeable:

- thumbnail image view
- name label
- location label
- type label

The view model stores and provides the values for the cell to display. Each of them is required to connect with the corresponding user interface object in Interface Builder. By connecting the source code with the UI objects, we can change the values of UI objects dynamically.

This is a very important concept in iOS programming. Your UI in storyboard and code are separated. You design the UI in Interface Builder and write your code in Swift. If you want to change the value or properties of a UI element (e.g. label), you have to establish a connection between them such that an object in your code can obtain a reference to an object defined in a storyboard. In Swift, you use `@IBOutlet` keyword to indicate a property of a class, that can be exposed to Interface Builder. For properties annotated with the `IBOutlet` keywords, we call it *outlets*.

So in the above code, we declare four outlets. Each outlet is going to connect with its corresponding UI object. Figure 9-21 depicts the connections.

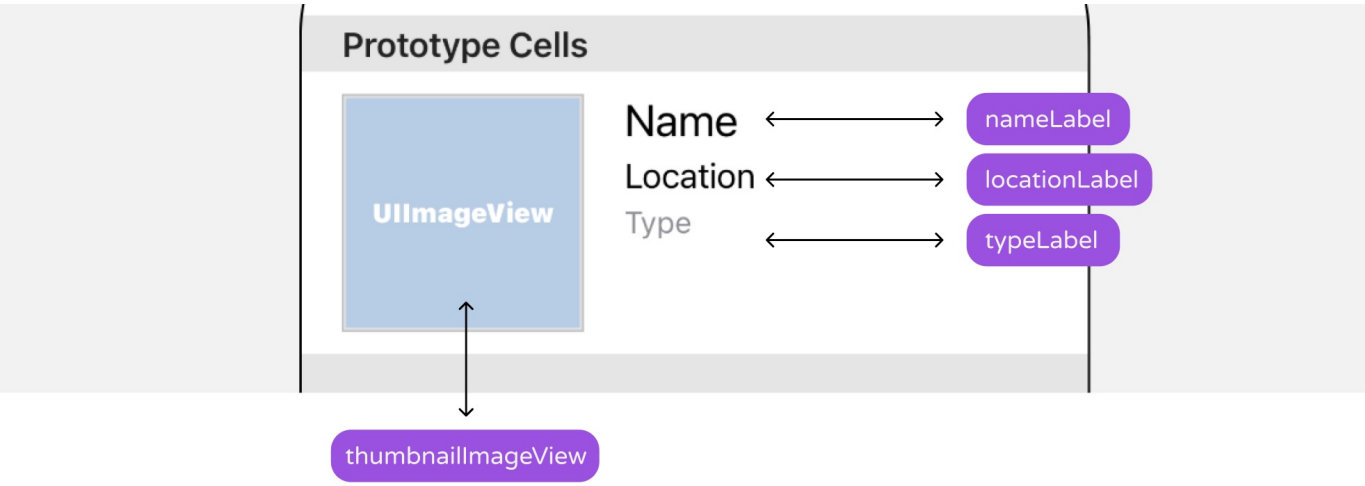


Figure 9-21. Outlet connections

@IBAction vs @IBOutlet

We've used @IBAction to indicate action methods when developing the HelloWorld app. What's the difference between @IBAction and @IBOutlet? @IBOutlet is used to indicate a property that can be connected with a view object in a storyboard. For example, if the outlet is connected with a button, you can use the outlet to change the color or title of the button. On the other hand, @IBAction is used to indicate an action method that can be triggered by a certain event. For example, when a user taps a button, it can trigger an action method to do something.

Before we can establish a connection between the outlets of `RestaurantTableViewCell` class and the prototype cell in Interface Builder. We have to first set the custom class.

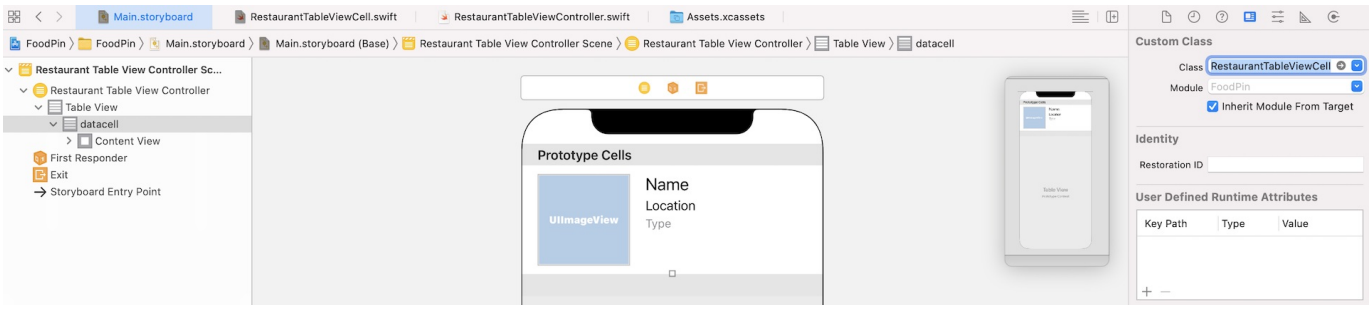


Figure 9-22. Set the custom class of the prototype cell

As said earlier, the prototype cell is associated with the default `UITableViewCell` class. To assign the prototype cell with the custom class, select the cell in storyboard. In the Identity inspector, set the custom class to `RestaurantTableViewCell` (see figure 9-22).

Establishing the Connections

Next, we will establish the connections between the outlets and UI objects in the prototype cell. In Interface Builder, right click the cell (i.e. `datacell`) in the Document Outline view to bring up the Outlets inspector. Drag from the circle (next to `thumbnailImageView`) to the image view object in the prototype cell (see figure 9-23). Xcode automatically establishes the connection when you release the button.

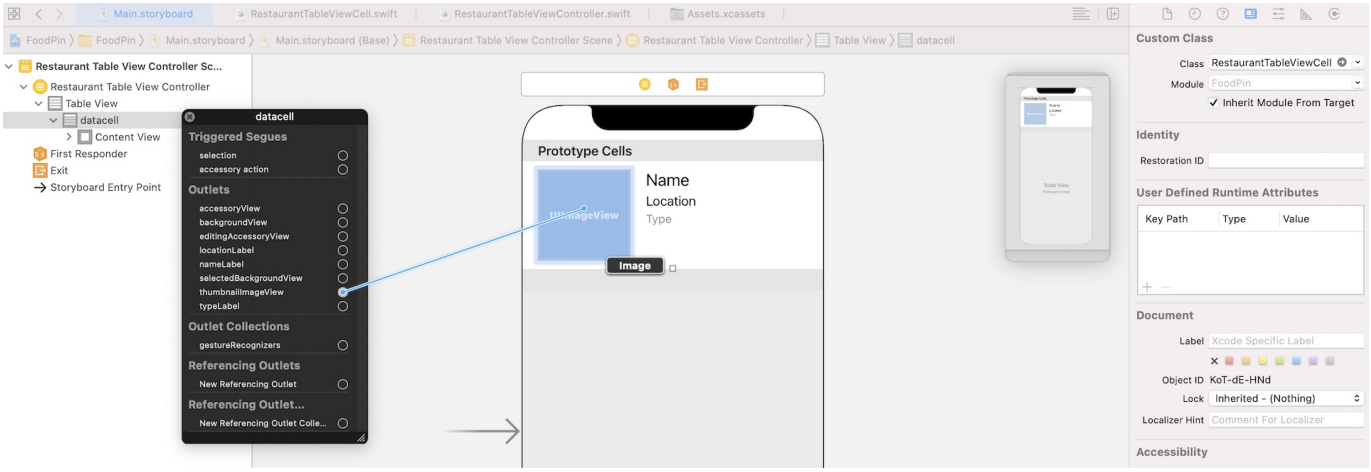


Figure 9-23. Connecting the outlet with the image view

Repeat the above procedures for the following outlets:

- **locationLabel** - connects to the *Location* label the cell
- **nameLabel** - connects to the *Name* label of the cell
- **typeLabel** - connects to the *Type* label of the cell

After you've made all the connections, the UI should look like the screenshot shown in figure 9-24.

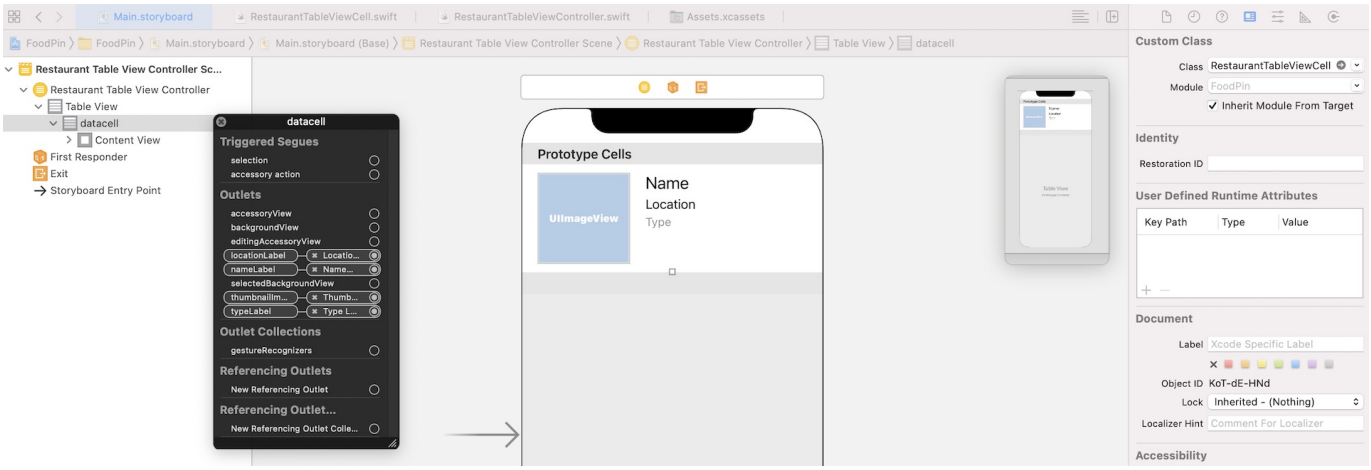


Figure 9-24. The outlet connections

You can also switch back to `RestaurantTableViewCell.swift` to review the connections. If an outlet variable is connected, it will be indicated by a gray circle indicator. You can even click on the indicator to reveal the connection.

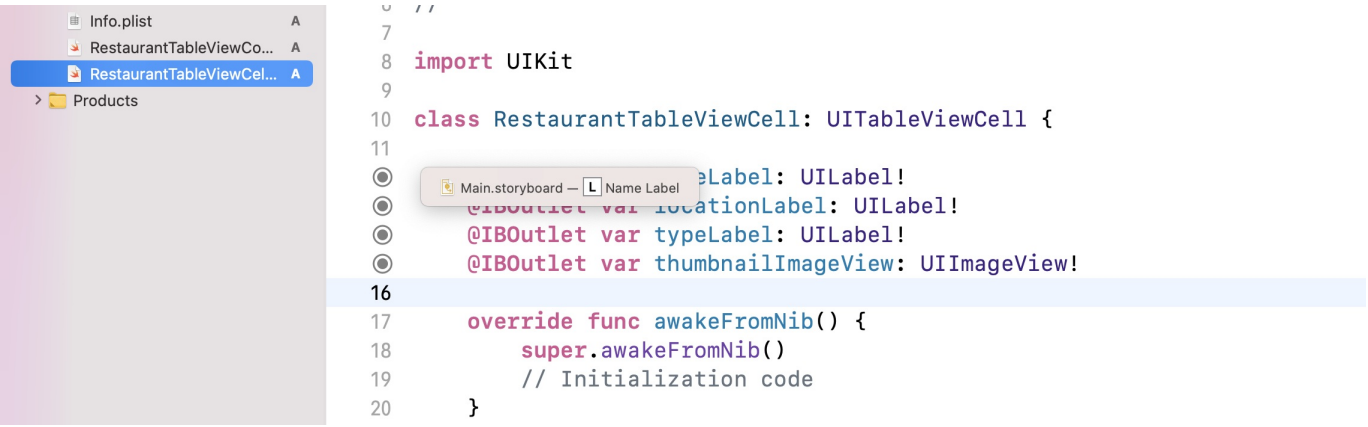


Figure 9-25. A gray circle indicates the outlet is connected properly

Updating the Cell Provider

Finally, we come to the last part of the change. In the `RestaurantTableViewController` class, we're still using `UITableViewCell` (i.e. the default cell) to display the content. We need to modify a line of code to replace it with our custom cell. If you look into the existing implementation of the `configureDataSource` method, you should find this line of code:

```
let cell = tableView.dequeueReusableCell(withIdentifier: cellIdentifier, f
or: indexPath)
```

I've explained the meaning of the `dequeueReusableCell` method in the previous chapter. It is flexible enough to return any cell type from the queue. By default, it returns a generic cell of a `UITableViewCell` type. How do I know that? I read the

documentation. Xcode provides you a convenient way to read the documentation of the APIs. Hold *option* key, place the mouse cursor over `dequeueReusableCell` and then click on it. Xcode will show you the description of the method.

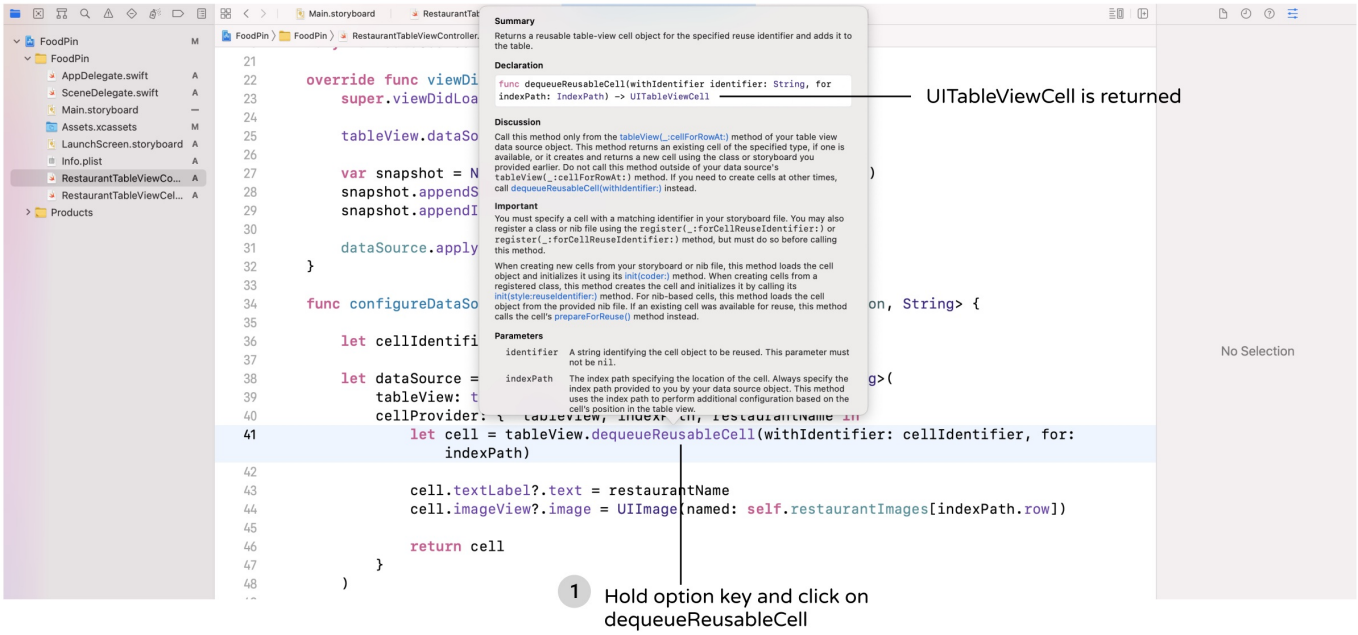


Figure 9-26. A gray circle indicates the outlet is connected properly

In order to use the `RestaurantTableViewCell` class, it's our responsibility to "convert" the returned object of `dequeueReusableCell` to `RestaurantTableViewCell`. This conversion process is known as *downcasting*. In Swift, we use the `as!` keyword to perform a forced conversion. Therefore, change the line of code in `configureDataSource()` to the following:

```
let cell = tableView.dequeueReusableCell(withIdentifier: cellIdentifier, f
or: indexPath) as! RestaurantTableViewCell
```

as! and as?

Downcasting allows you to convert a value of a class to its derived class. For example, `RestaurantTableViewCell` is a child class of `UITableViewCell`. The `dequeueReusableCellWithIdentifier` method always returns a `UITableViewCell` object. If a custom cell is used, this object can be converted to the specific cell type (e.g. RestaurantTableViewCell). Prior to Swift 1.2, you can just use the `as` operator for downcasting. However, sometimes the object may not be converted to a specified type. Therefore, from Swift 1.2 and onwards, Apple introduced two more operators: `as!` and `as?`. If you're quite sure that the downcasting can perform correctly, use `as!` to perform the conversion. In case you're not sure if the value of one type can be converted to another, use `as?` to perform an optional downcasting. You're required to perform additional checking to see if the downcasting is successful or not.

I know you can't wait to test the app, but we need to change a few more lines of code. The lines of code below set the values of restaurant name and image:

```
cell.textLabel?.text = restaurantName
cell.imageView?.image = UIImage(named: self.restaurantImages[indexPath.row
])
```

Both `textLabel` and `imageView` are properties of the default `UITableViewCell` class. Because we're now using our own `RestaurantTableViewCell`, we need to use the properties of the custom class. Change the above lines to the following:

```
cell.nameLabel.text = restaurantName
cell.thumbnailImageView.image = UIImage(named: self.restaurantImages[index
Path.row])
```

Now you're ready to go. Hit the Run button and test the app. Your app should look like the one shown in figure 9-27. Try to rotate the simulator. The app also works in landscape orientation. It is a huge improvement from the previous version of the app. We're going to make it even better by rounding the corners of the images.

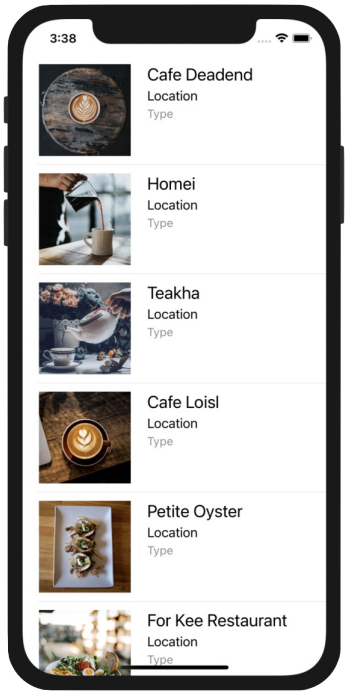


Figure 9-27. FoodPin app with custom table view cells

Rounding Image Corners

With the release of iOS 14 and macOS Big Sur, it makes me feel that Apple favors to round the edges of UI elements. For this demo app, wouldn't it be great to make all the restaurant images rounded corners? You do not need Photoshop to tweak

the images. What you need is to write a couple lines of code.

Every view in the UIKit (e.g. `UIView`, `UIImageView`) is backed by an instance of the `CALayer` class (i.e. layer object). The layer object is designed to manage the backing store for the view and handles view-related animations.

The layer object provides various attributes that can be set to control the visual content of the view such as:

- Background color
- Border and border width
- Shadow color, width, etc
- Opacity
- Corner radius

The corner radius is the attribute, which we use to draw rounded corners. Xcode provides two ways to edit the layer properties. One of the approaches is make the change through Interface Builder. First, select the image view in the stack view. Go to the Identity inspector, click the Add button (+) in the lower left of the *User Defined Runtime Attributes* editor. A new runtime attribute appears in the editor. Double click on the *Key Path* field of the new attribute to edit the key path for the attribute. Set the value to `cornerRadius` and hit Return to confirm. Click on the *Type* attribute and choose `Number`. Lastly, set the value to `20`.

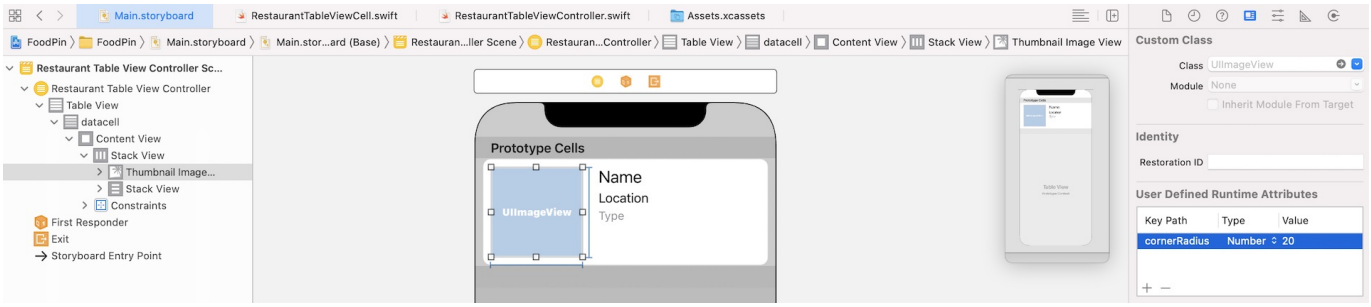


Figure 9-28. Set the runtime attribute to make the corner round

When the image view is initialized, this runtime attribute will be automatically loaded to round the corners. There is one more thing you have to configure before the circular image works properly. Select the image view and go to the Attributes inspector. In the Drawing section, please make sure to enable the *Clip to Bounds* option. This causes the content to be clipped to the rounded corners.

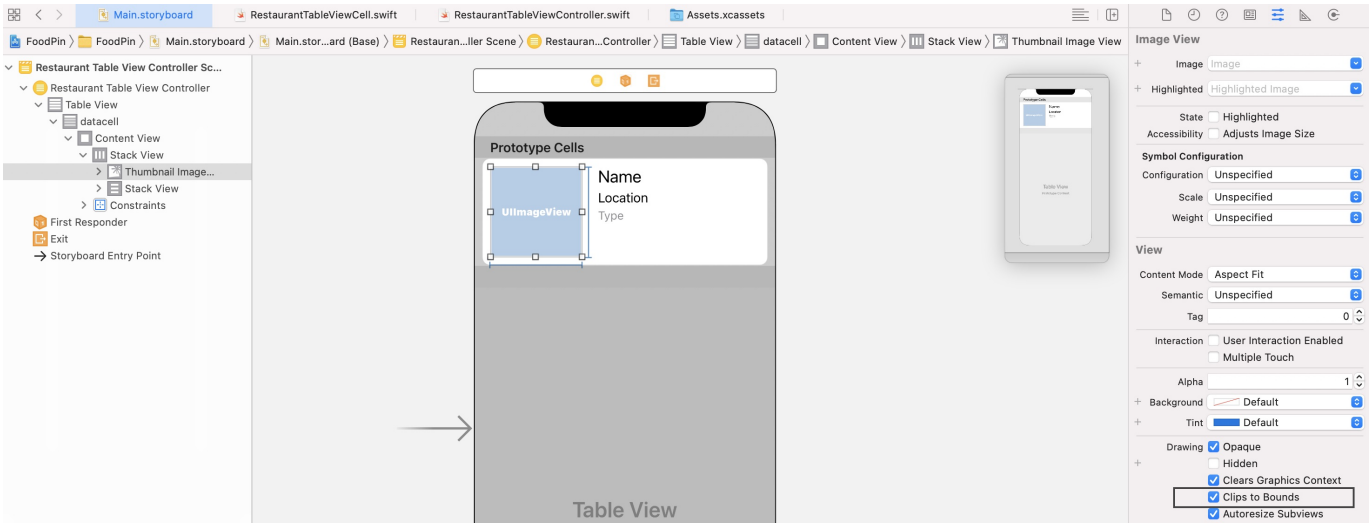


Figure 9-29. FoodPin app with circular images

Now compile and run the app. The UI looks even better, right? Without writing a line of code, we round the edge of the images. You're free to alter the value of corner radius. Try to change the corner radius to 60 points and see what you get.

Other than using Interface Builder, you can also change the corner radius programatically. In `RestaurantTableViewCell.swift`, you can add a `didSet` code block for `thumbnailImageView` like this :

```
@IBOutlet var thumbnailImageView: UIImageView! {
    didSet {
        thumbnailImageView.layer.cornerRadius = 20.0
        thumbnailImageView.clipsToBounds = true
    }
}
```

`didSet` is known as a property observer in Swift. As its name suggests, the code block specified in `didSet` will be called every time a property's value is set. In the code above, when an image view is assigned, the `didSet` code block will be called to change the corner radius and set the `clipsToBounds` value to `true`.

To create a circular image, you can change the code snippet like this, which set the radius to half the width of the image view.

```
@IBOutlet var thumbnailImageView: UIImageView! {
    didSet {
        thumbnailImageView.layer.cornerRadius = thumbnailImageView.bounds.
width / 2
        thumbnailImageView.clipsToBounds = true
    }
}
```

Testing the App Using Dark Mode

Since the release of iOS 13, Apple allows users to choose between light and dark system-wide appearance. When the user chooses to adopt Dark Mode, both the system and apps use a darker color palette for all screens and views. As an app developer, you should make sure your apps also comply with Dark Mode. There are multiple ways to test your app in Dark Mode.

While you are running the app on a simulator, you can click the *Environment Overrides* button to switch the appearance from light to dark. Once you enable the switch, the simulator is set to adopt Dark Mode. Our FoodPin app already looks great without any additional changes.

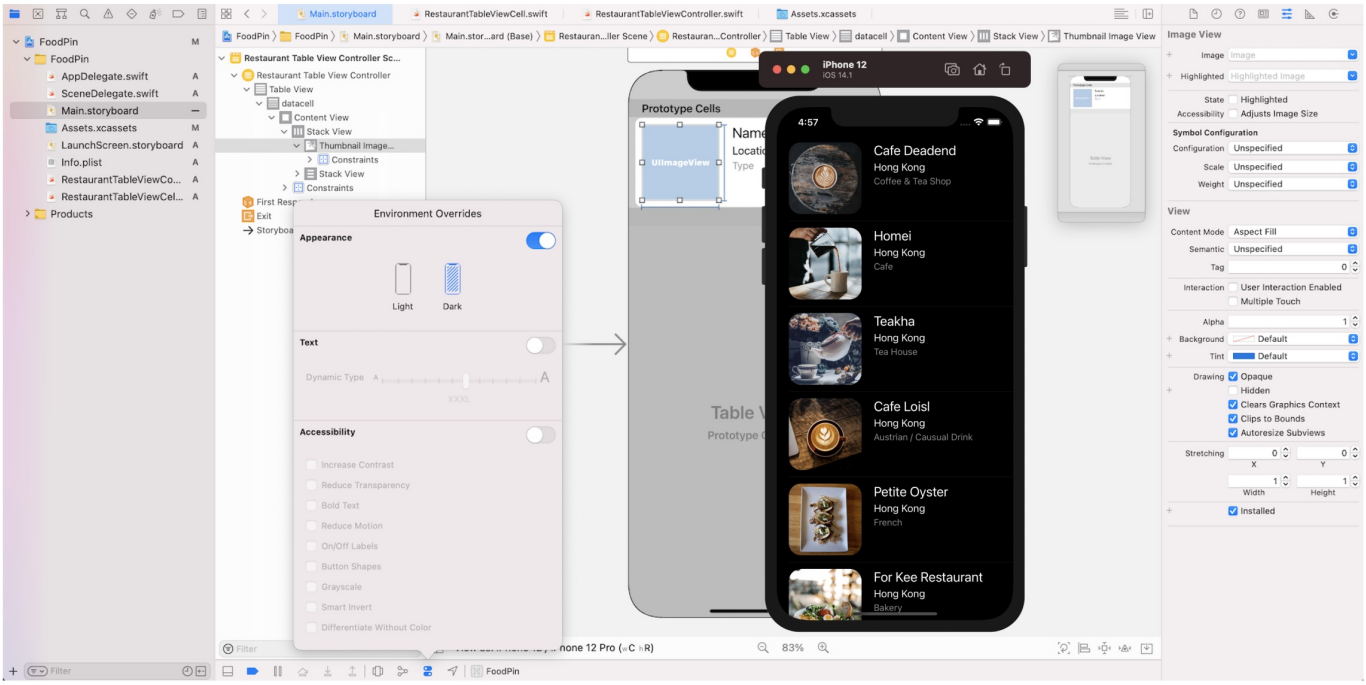


Figure 9-30. Redesigned Custom Table

Alternatively, you can directly change the setting of your simulator. On any of the simulators, go to *Settings* > *Developer*. Turn the switch of the *Dark Appearance* option to ON to enable Dark Mode.

Exercise #1

As of now, the app simply displays "Location" and "Type" for all rows. As an exercise, I'll leave it to you to fix the issue. Try to edit the source code to update the location and type labels. Below are the two arrays you need:

```
var restaurantLocations = ["Hong Kong", "Hong Kong", "Hong Kong", "Hong Ko
ng", "Hong Kong", "Hong Kong", "Hong Kong", "Sydney", "Sydney", "Sydney",
"New York", "New York", "New York", "New York", "New York", "New York", "N
ew York", "London", "London", "London", "London"]

var restaurantTypes = ["Coffee & Tea Shop", "Cafe", "Tea House", "Austrian
 / Causual Drink", "French", "Bakery", "Bakery", "Chocolate", "Cafe", "Ame
rican / Seafood", "American", "American", "Breakfast & Brunch", "Coffee &
Tea", "Coffee & Tea", "Latin American", "Spanish", "Spanish", "Spanish", "
British", "Thai"]
```

Exercise #2

The previous exercise may be too easy for you. Here is another challenge. Try to redesign the prototype cell and see if you create an app like this (see figure 9-31).

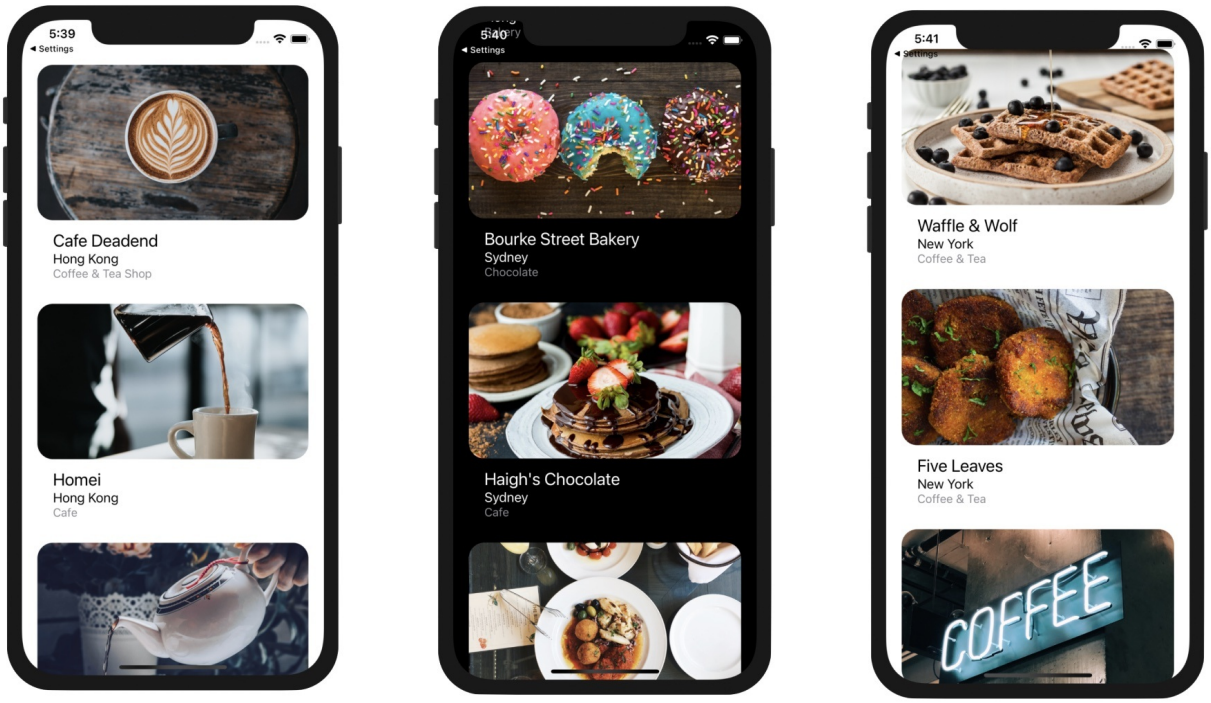


Figure 9-30. Redesigned Custom Table

As a hint, you can add the following line of code in `viewDidLoad()` to remove the cell separator:

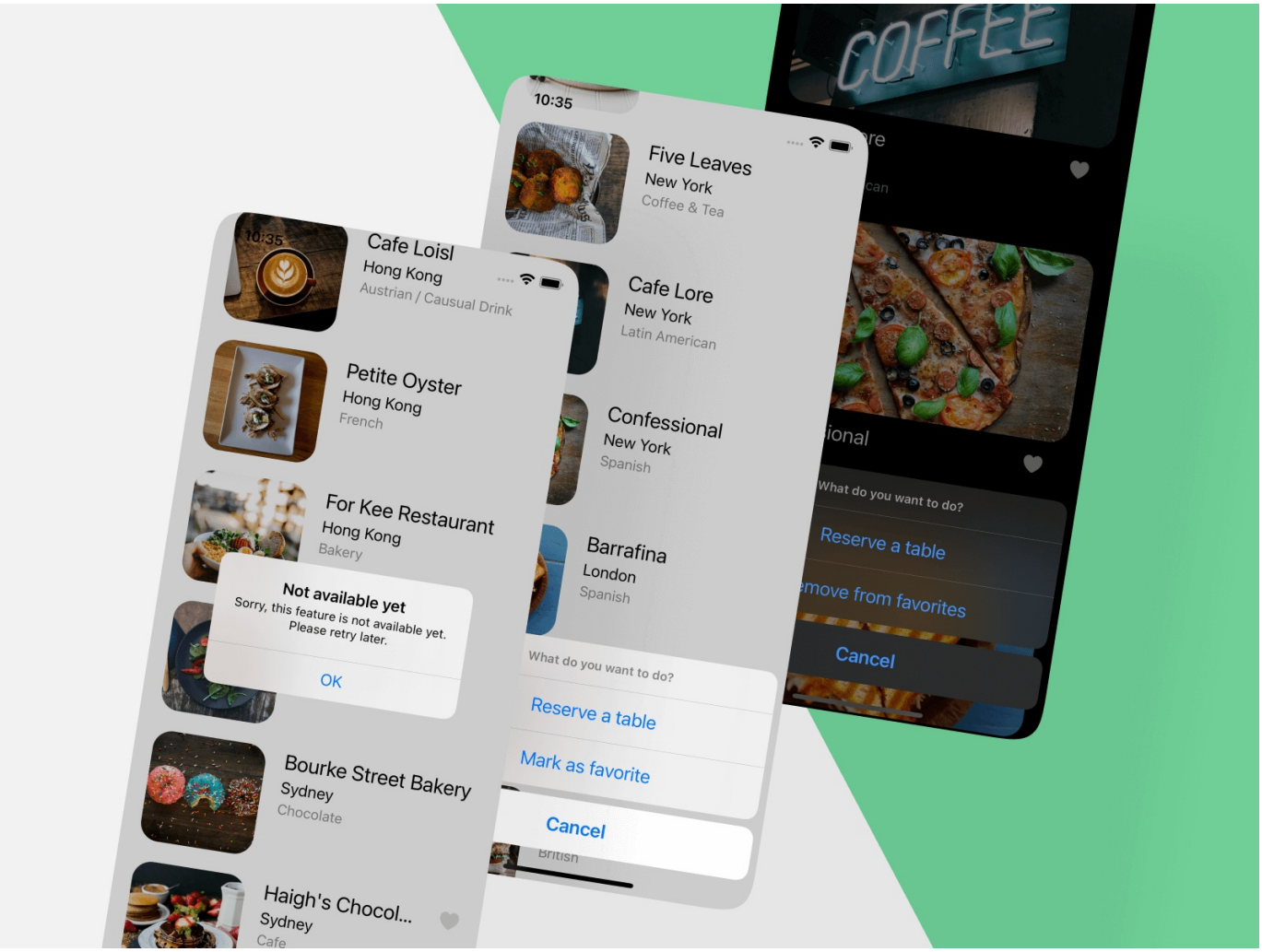
```
tableView.separatorStyle = .none
```

Summary

Congratulations! You've made a huge progress. If you understand the ins and outs of cell customization, you're ready to make some awesome UIs. Table view is the backbone for most iOS apps. Unless you're building a game, you'll probably implement a table view in one way or another when building your own apps. Table view customization may be a bit complex for some of you. So take some time to work on the exercises and play around with the code. Remember "learn by doing" is the best way to learn coding.

For reference, you can download the complete Xcode project from http://www.appcoda.com/resources/swift53/FoodPinCustomTable.zip. If you can't figure out how to complete the exercise, you can download the solution from http://www.appcoda.com/resources/swift53/FoodPinCustomTableExercise.zip.

Chapter 10
Displaying Alerts with UIAlertController and Handling Table View Selection



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

Are you able to complete the previous exercise and create the custom table view? No worries if you can't. I will go through the solution with you in this chapter and introduce some new layout techniques. Up till now, we only focus on displaying data in a table view. I guess you may wonder how we can interact with the table view and detect row selections. This is what we will also discuss in this chapter.

To begin, first download the complete project we built in the previous chapter (http://www.appcoda.com/resources/swift53/FoodPinCustomTable.zip). We will continue to enhance the app and make it even better. In brief, here are what we are going to implement:

- Add another prototype cell to the table view
- Bring up a menu when a user taps a cell. The menu offers two options: *Reserve a table* and *Mark as favorite*.
- Display a heart icon when a user selects *Mark as favorite*.

Through implementing these new features, you will also learn how to use two prototype cells and use `UIAlertController` to display alerts in iOS.

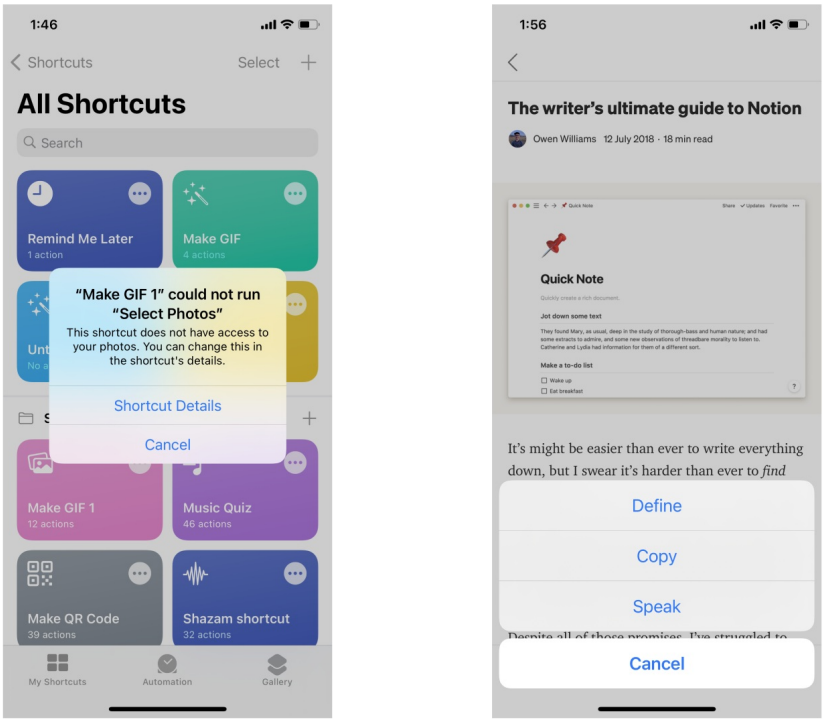


Figure 10-1. Sample alerts in Shortcut and Medium apps

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

Building a more Elegant Cell Layout

Previously, I had an exercise for you and asked you to redesign the prototype cell to make the cell layout look like that shown in figure 10-2. I hope you already tried to figure out the solution. Even if you couldn't find a way to design the cell, I appreciate your effort. It is not an easy exercise for beginners.

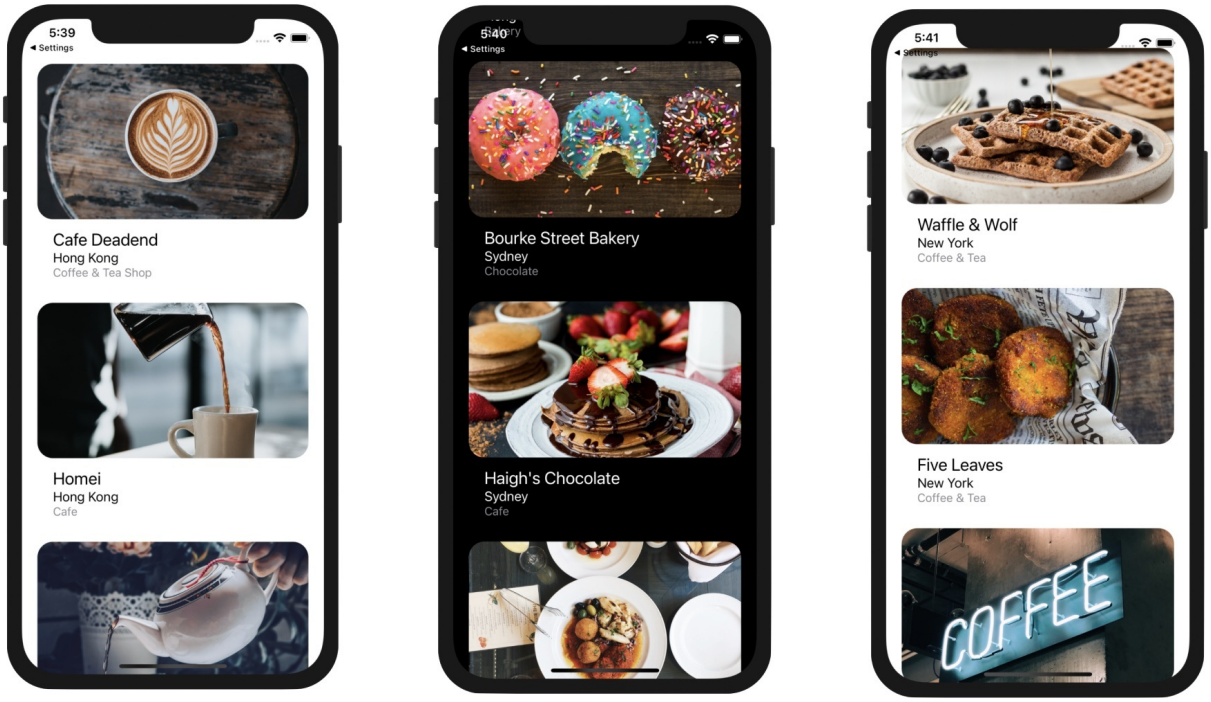


Figure 10-2. Redesigning the table view cell

Now let's see how to design the cell layout. Assuming you have already downloaded the project and open it in Xcode, select `Main.storyboard` and switch over to Interface Builder. Instead of deleting the existing cell layout, we will create another prototype cell for the new design. That's right! You can create multiple prototype cells in the same table view. The only requirement is that each of the prototype cell should have a distinct identifier.

In the document outline, select *Table View* and then open the Attributes inspector. Here, change the value of the *Prototype Cells* option from `1` to `2`. You will immediately see a new prototype cell added to the table view with the same cell layout.

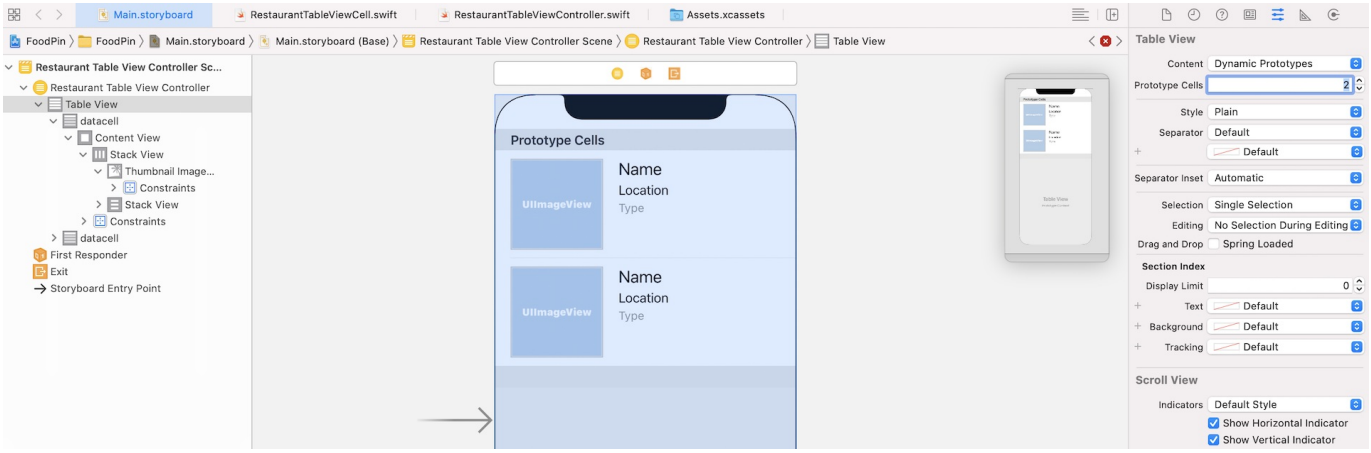


Figure 10-3. Adding a new prototype cell

At the same time, Xcode displays an error indicator in Interface Builder. If you click the error indicator, Xcode complains that the two prototype cells have the same identifier which is not allowed. Now select the new prototype cell and change its identifier in the Attributes inspector. Let's name it `favoritecell`.

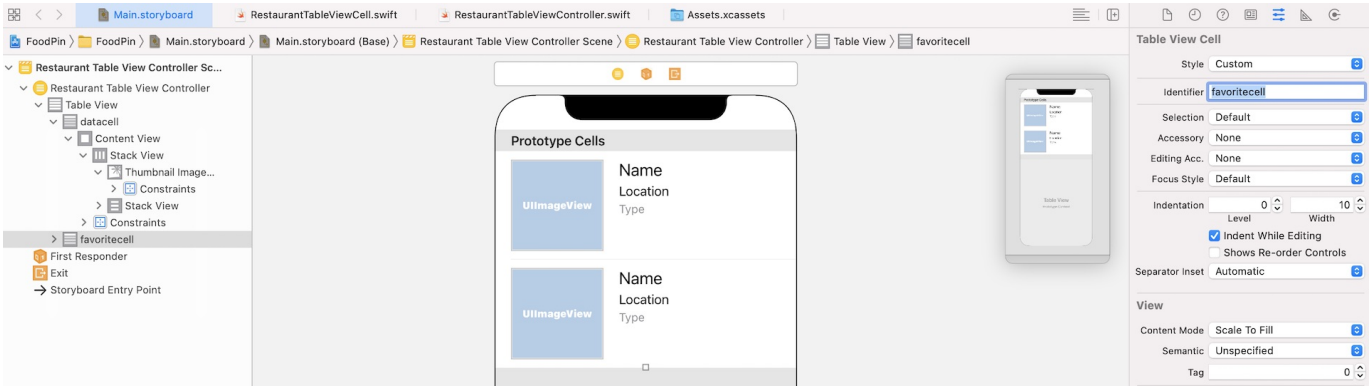


Figure 10-4. Changing the identifier for the new prototype cell

Once you made the change, the Xcode error disappears. Now it's time to redesign the new cell. You do not delete all the elements and create the new cell from scratch because we still need both the image view and the labels. Instead, we are going to rearrange the cell layout.

First, select the `favoritecell` and go to the Size inspector. Check the *automatic* checkbox of Row Height to let Xcode adjust the row height automatically.

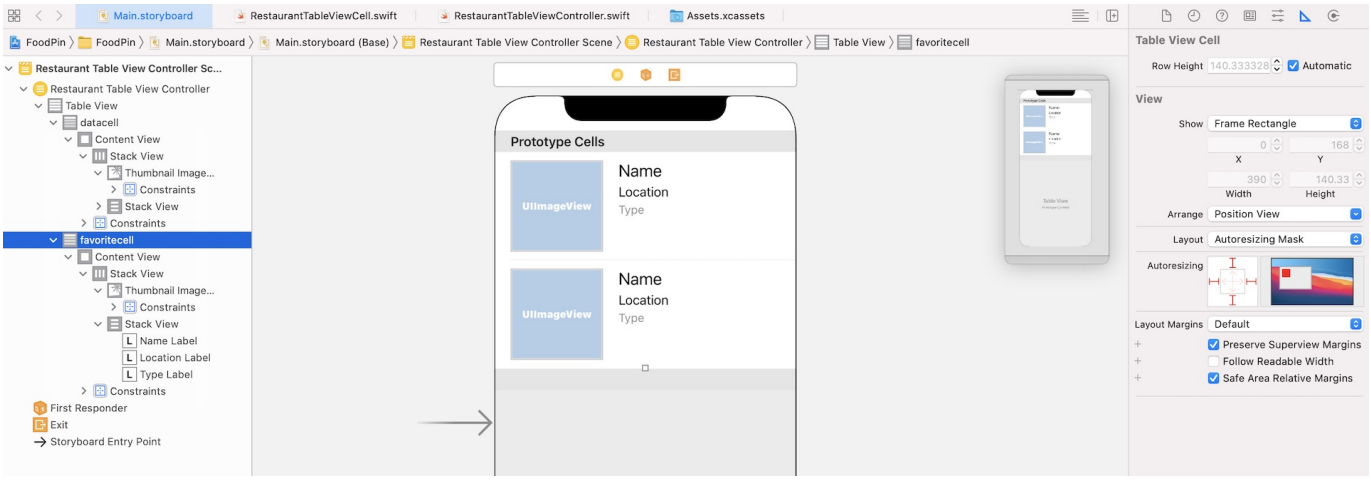


Figure 10-5. Setting the row height to automatic

Next, select the root Stack View of the `favoritecell` . In the Attributes inspector, change the *Axis* option from *Horizontal* to *Vertical* and you will see a new cell layout. This is one of the benefits of Stack View. It makes it so easy to rearrange the layout of the embedded components.

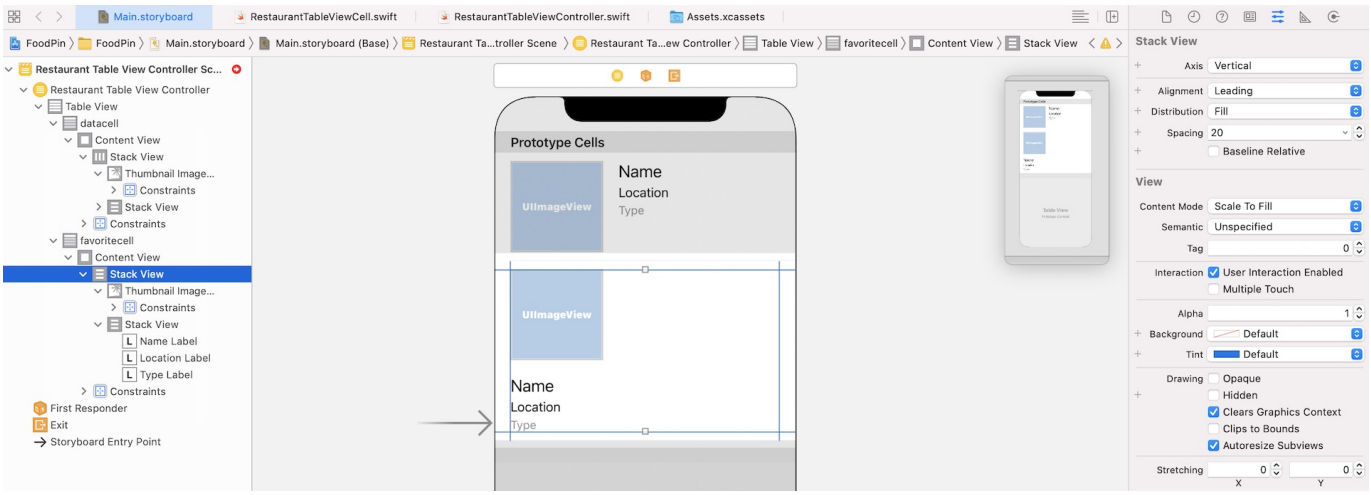


Figure 10-6. Changing the axis of the stack view

With the new design, we no longer fix the width of the image view. Therefore, select the width constraint of the image view and hit delete to remove it.

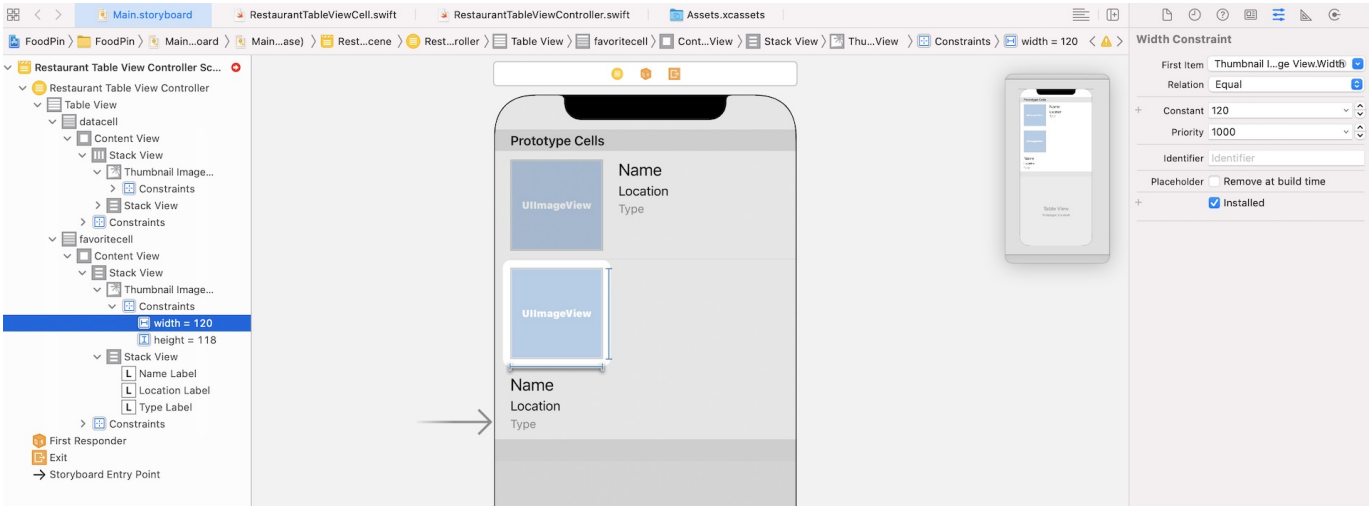


Figure 10-7. Delete the width constraint of the image view

We also want to make the image a little bit bigger. So, select the height constraint of the image view and change its constant in the Attributes inspector to `200`.

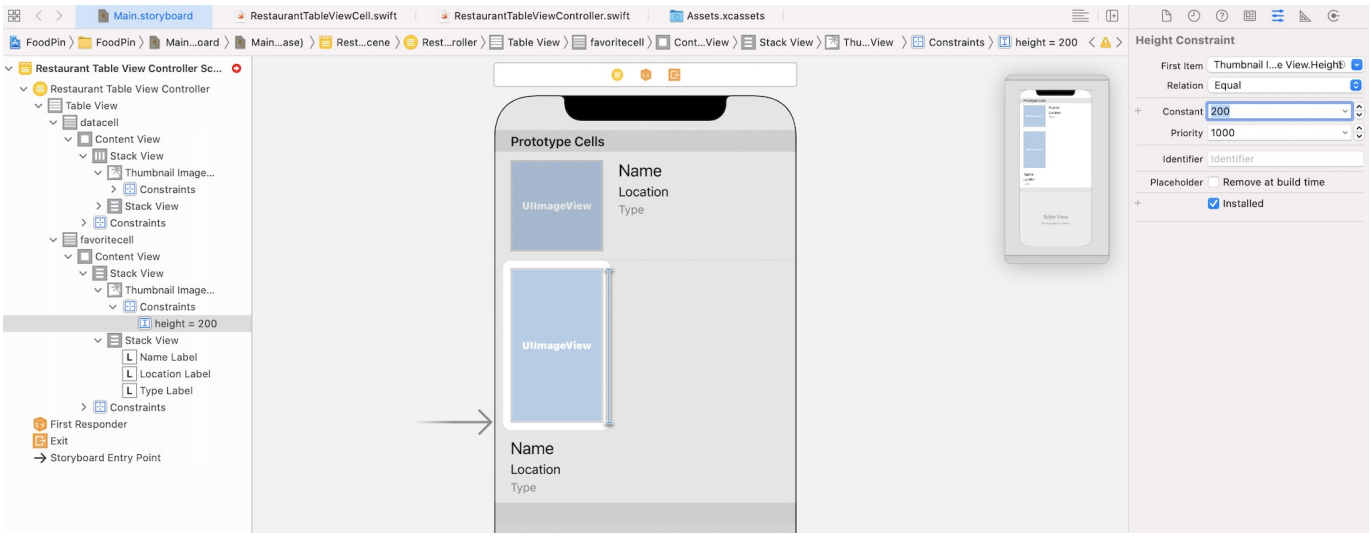


Figure 10-8. Changing the height constraint of the image view

Meanwhile, the image view doesn't scale to take up the width of the stack view. This is due to a setting in the root Stack view. Now select the stack view again and change its alignment from *Leading* to *Fill*. The image view should automatically expand. To make the cell look a little bit better, you can also change the spacing value from `20` to `5`. To fix the layout error (did you notice the red indicator in the document outline?), change the *Distribution* option from *Fill* to *Equal Spacing*. This tells the stack view to maintain an equal spacing between the subviews without resizing the subviews themselves.

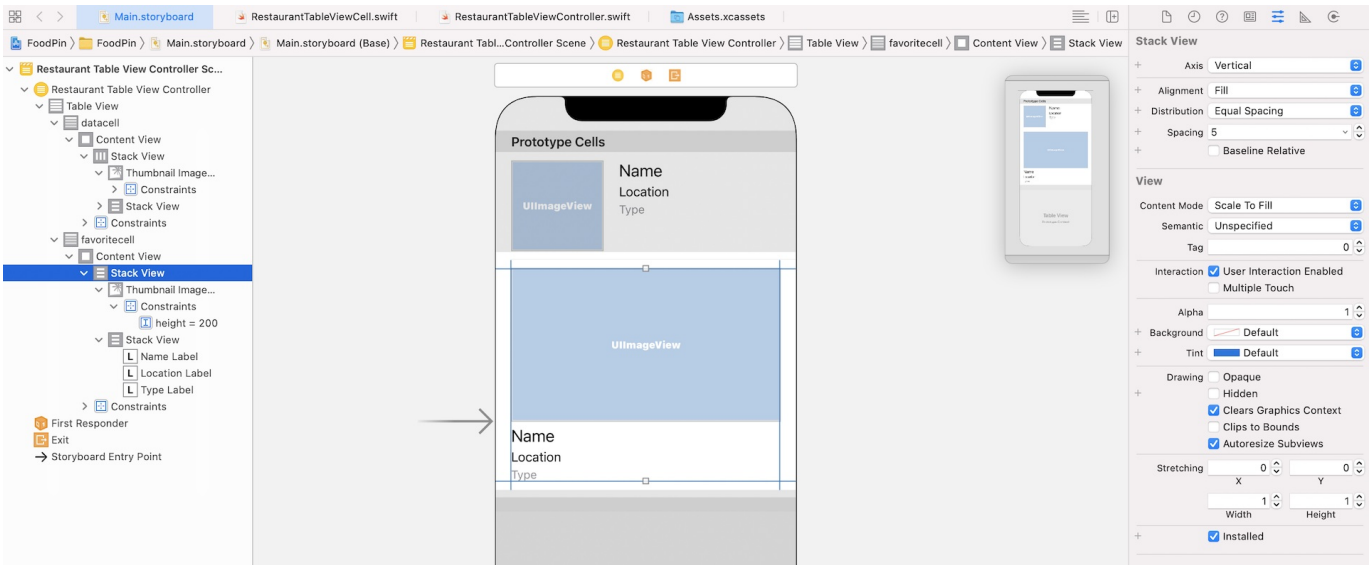


Figure 10-9. Changing the settings of the stack view

The cell layout now looks nearly the same as the one shown in figure 10.2. However, if you take a closer at the cell in that figure, the labels are indented slightly. How can you cater for this kind of indentation in a stack view?

Now select the stack view that holds the labels and go to the Size inspector. You can indent the stack view by setting the Layout Margins. Change the value of *Layout Margins* to *Language Directional* and set both leading & trailing to `20` points. This should indent the label stack view.

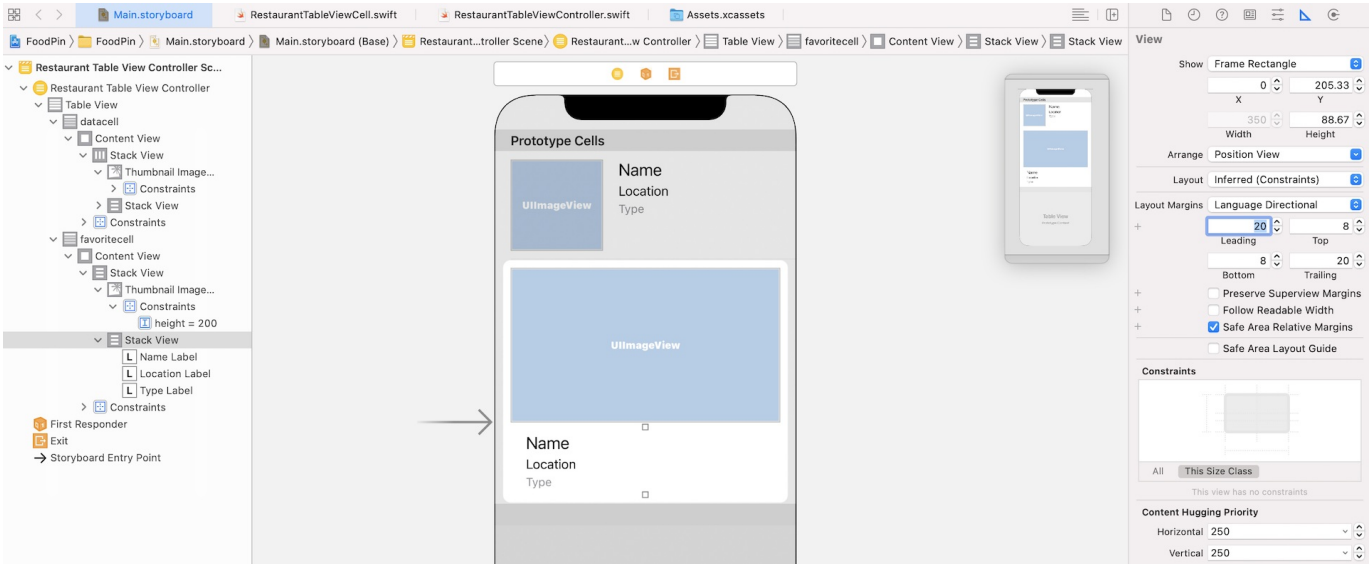


Figure 10-10. Changing the settings of the stack view

If you run the app, it would still show the old design. This is we are still referring to the `datacell` in the code. To use the new cell layout, open `RestaurantTableViewController.swift` and change the `cellIdentifier` constant in `configureDataSource()`:

```
let cellIdentifier = "favoritecell"
```

Instead of using `datacell`, we tell the table view to use the new `favoritecell`. Now you are ready to test the app. Choose any of the iPhone simulators and hit the Play button. Your app should show you a brand new look. If you want to remove the separator between the cells, insert this line of code in the `viewDidLoad()` method:

```
tableView.separatorStyle = .none
```

What happens if you want to switch back to the old design? Yes! You just need to change the `cellIdentifier` back to `datacell`. This is how you can create multiple prototype cells and apply different design. In later chapters, I will teach you how to show two different cell layouts in the same table view.

Understanding the UITableViewDelegate Protocol

Now let's move onto the next topic and discuss how you can detect a touch on the table view. When we first built the SimpleTable app in Chapter 8, we adopted two delegates, `UITableViewDelegate` and `UITableViewDataSource`, in the `RestaurantTableViewController` class. While we have a lengthy discussion on replacing the `UITableViewDataSource` protocol with `UITableViewDiffableDataSource`, I 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 `UITableViewDelegate` protocol is designed to take care the configuration of the section headings and footers of the table view, as well as, handling cell selections and cell reordering. Therefore, to manage the row selection, we will need to adopt the `UITableViewDelegate` protocol and implement its methods.

Looking Up the Documentation

Before implementing the methods, you may wonder:

How do we know which methods in UITableViewDelegate to implement?

The answer is "Read the documentation". You're granted free access to the Apple's official iOS developer reference (https://developer.apple.com/documentation/). As an iOS developer, you need to get used to reading the API documentation. There is no single book on earth to cover everything about the iOS SDK. Most of the time when we want to learn more about a class or a protocol, we have to look up to the API document.

Apple provides a simple way to access the documentation in Xcode. You can press and hold the *option* key and click on the class name to bring up the documentation. Alternatively, you can use the keyboard shortcut `control-command-?` and then place the cursor over a class or a protocol (e.g. `UITableViewController`). This brings up a popover showing the details of the class.

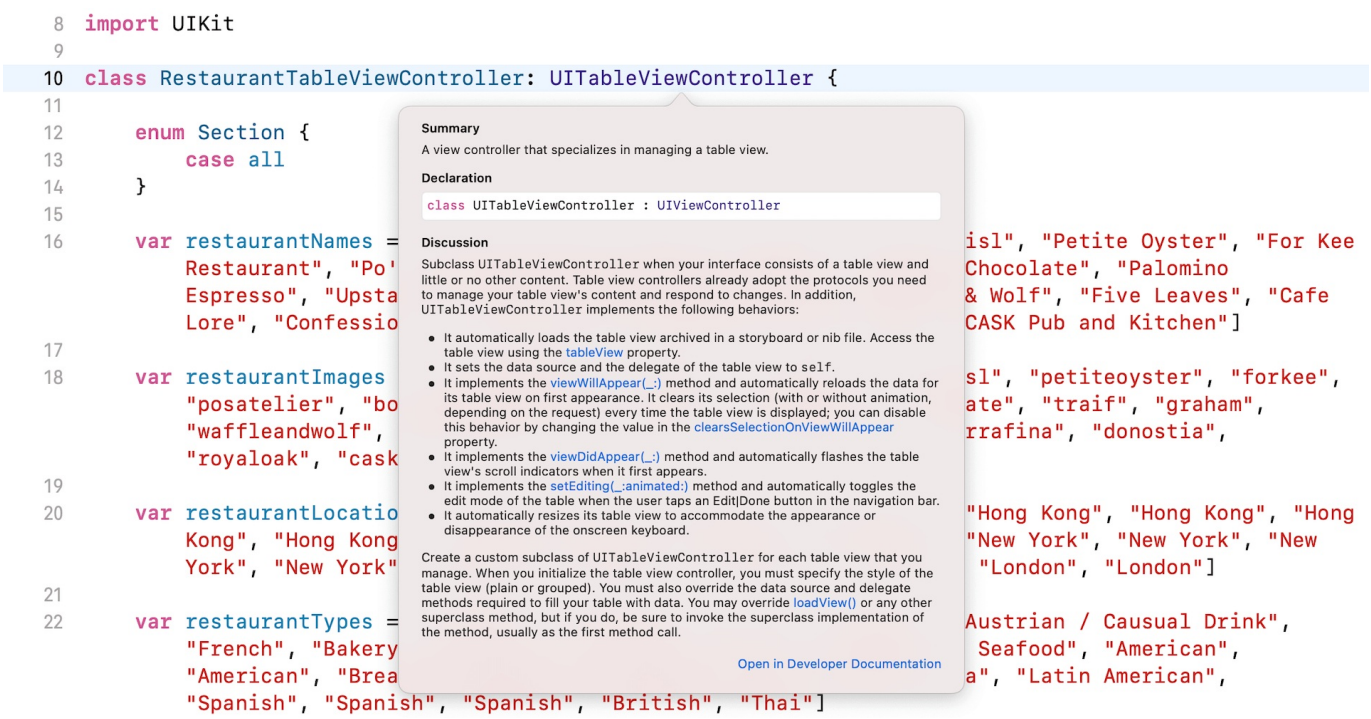


Figure 10-11. Accessing the API documentation by using a shortcut key

Scroll down and you will see the "Open in Developer Documentation" link. Click it to bring up the documentation browser. Here, you'll find all the usage of the class. If you scroll down to the end of the document, you should see a section named "Conforms To" that lists out all the protocols that this class adopts. One of them is the `UITableViewDelegate` protocol. Click it to access the documentation.

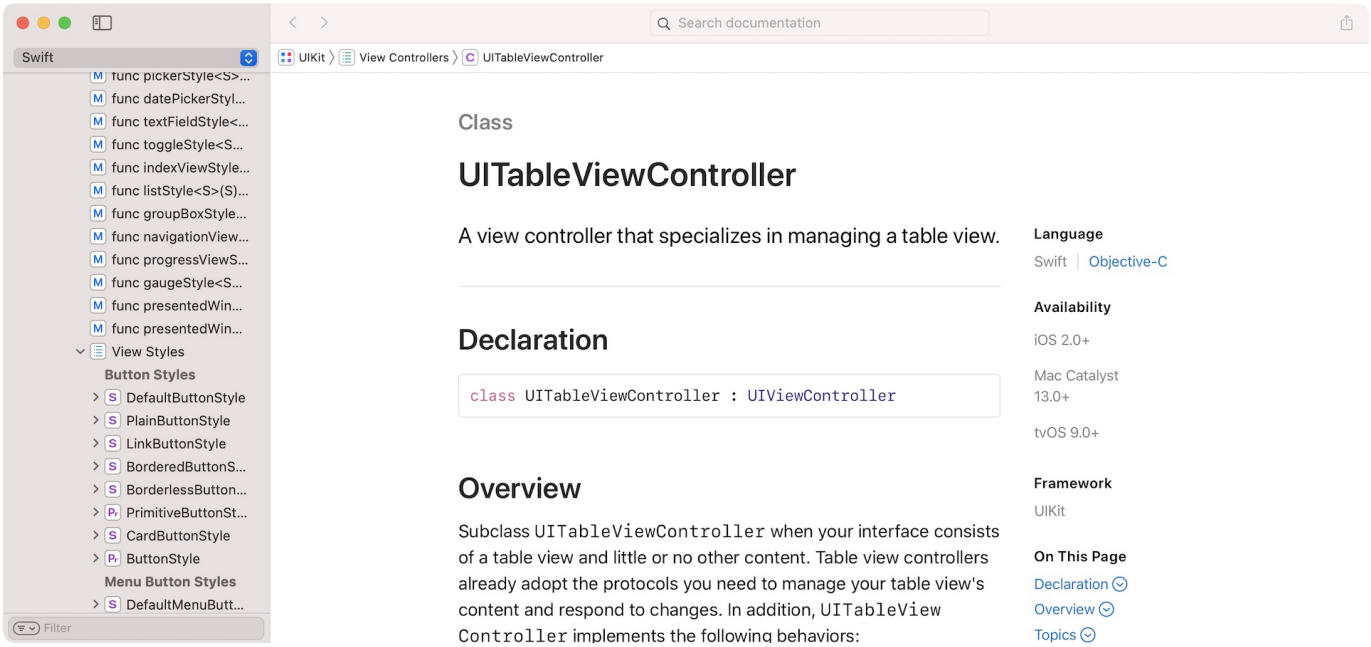


Figure 10-12. UITableViewController Documentation

By glancing through the document, you will find the methods below for managing row selections:

- `func tableView(UITableView, willSelectRowAt: IndexPath) -> IndexPath?`
- `func tableView(UITableView, didSelectRowAt: IndexPath)`

Both methods are designed for row selections. The only difference is that the first method is called when a specified row is about to be selected. You may use this method to prevent the selection of a particular cell from taking place. Typically,

you use the `tableView(_:didSelectRowAt:)` method. It is called after the user selects a row, to handle the row selection. We will implement this method to perform additional tasks (e.g. bringing up a menu) after a row is selected.

Managing Row Selections by Implementing the Protocol

Okay, that's enough for the explanation. Let's move onto the interesting part and write some code. In the FoodPin project, open the `RestaurantTableViewController.swift` file and implement the `tableView(_:didSelectRowAt:)` method in the `RestaurantTableViewController` class:

```
override func tableView(_ tableView: UITableView, didSelectRowAt indexPath
: IndexPath) {

    // Create an option menu as an action sheet
    let optionMenu = UIAlertController(title: nil, message: "What do you w
ant to do?", preferredStyle: .actionSheet)

    // Add actions to the menu
    let cancelAction = UIAlertAction(title: "Cancel", style: .cancel, hand
ler: nil)
    optionMenu.addAction(cancelAction)

    // Display the menu
    present(optionMenu, animated: true, completion: nil)
}
```

First, what's the `override` keyword? The documentation already tells us that the `UITableViewController` class adopted the `UITableViewDelegate` protocol and provided a default implementation for the method. Now we are *overriding* this method with our own code in `RestaurantTableViewController`. To override a method

of a superclass, we add the `override` keyword at the very beginning of the method. When you do this, it's just like saying, "Hey, don't use the default method implementation. Use mine."

The code above creates an option menu by instantiating a `UIAlertController` object. When a user taps any rows in the table view, this method will be called automatically to bring up an action sheet showing a *What do you want to do* message and a *Cancel* button. Try to run the project to have a quick test. The app should now be able to detect a touch.

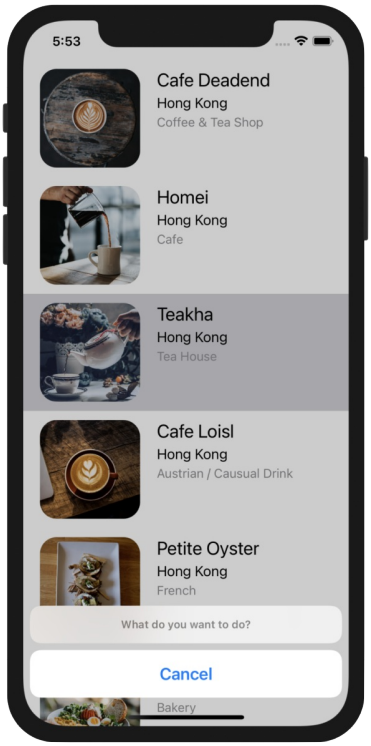


Figure 10-13. Displaying an action sheet

Understanding UIAlertController

Before we move on, let me talk more about the `UIAlertController` class. The `UIAlertController` class was first introduced in iOS 8 to replace both `UIAlertView` and `UIActionSheet` classes from older versions of iOS SDK. It is designed for displaying alert messages to a user.

Referring to the code snippet in the previous section, you can specify the style of the `UIAlertController` object through the `preferredStyle` parameter. The value we used is `.actionSheet`. Alternatively, you can set its value to `.alert` which gives you another alert style. Figure 10-14 displays the sample alert styles.

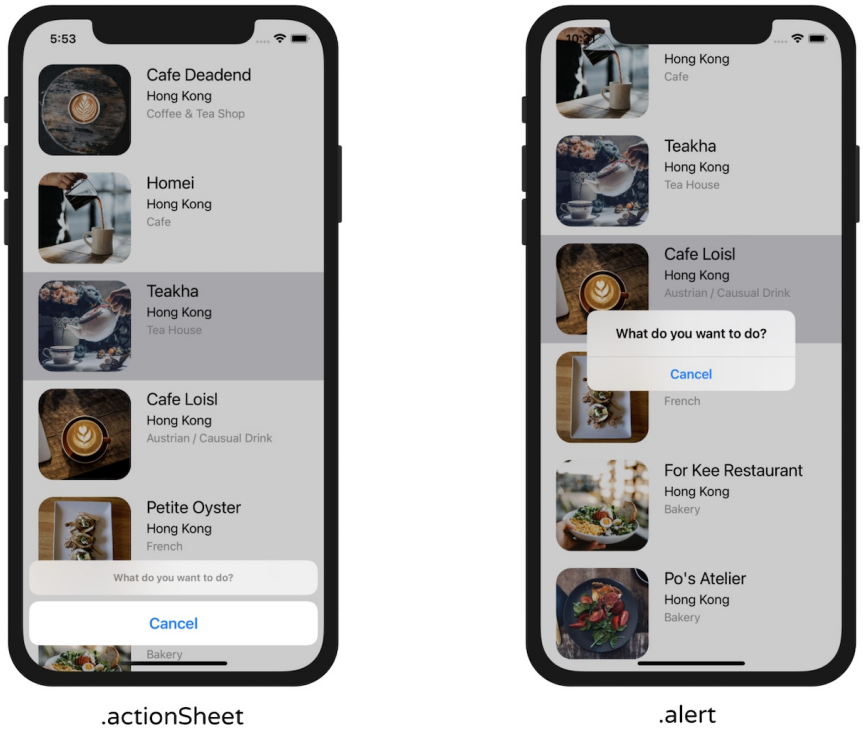


Figure 10-14. ActionSheet (left) and Alert (right)

Other than displaying a message to users, the alert controller usually associates with an action to give users a way to respond. In the code, we created a `UIAlertAction` object with a preferred title, style, and the block of code to execute for the action. The `cancelAction` object has the title `Cancel` and is set to use the

`.cancel` style. There is nothing to perform when a user selects the cancel action. Thus, the handler is set to `nil`. After the `UIAlertAction` object is created, you can assign it to the alert controller by using the `addAction` method.

When the alert controller is configured properly, you present it modally using the `present` method:

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

This is how you use the `UIAlertController` class to present an alert on screen. As a beginner, you may have a couple of questions in mind:

- How do I know the available values of the `preferredStyle` parameter when creating a `UIAlertController` object?
- The dot syntax looks new to me. Shouldn't it be written as `UIAlertControllerStyle.actionSheet`?

Both are good questions.

For the first question, again the answer is "Refer to the documentation". In Xcode, you can place the cursor over the `preferredStyle` parameter and press `control-command-?`. Xcode will show the method declaration. You can further click `UIAlertController.Style` to read the API reference. `UIAlertController.Style` is an enumeration, which defines two possible values: `actionSheet` and `alert`. That said, the most convenient way to find out the available values is by typing a `.` after `preferredStyle:`. Xcode automatically shows you all the possible values.

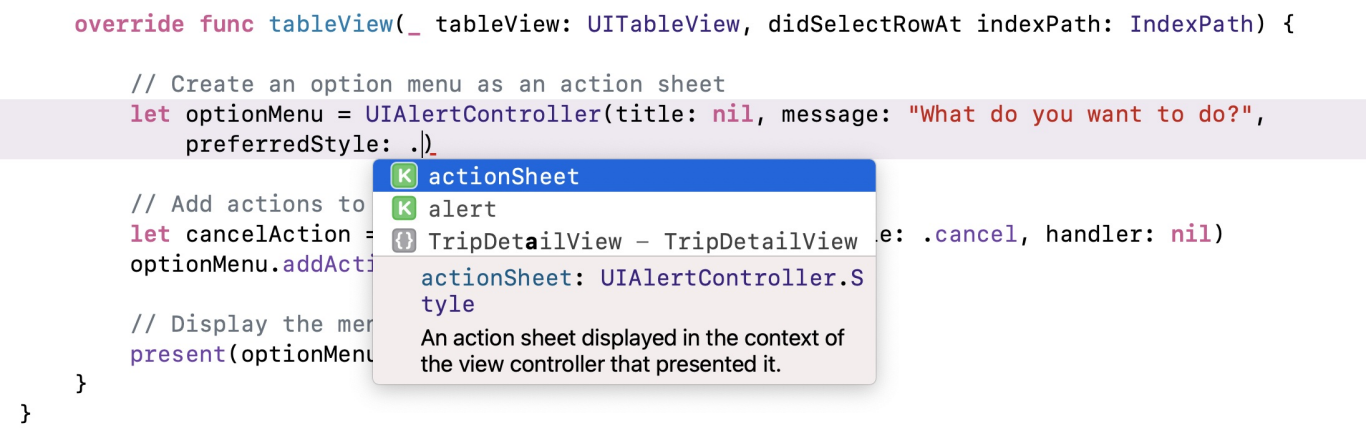


Figure 10-15. The available UIAlertControllerStyle values

> **Quick note:** An enumeration is a common type in Swift that defines a list of possible values for that type. The UIAlertController.Style is a good example.

Now let me answer the second question.

Normally we can refer to the values using `UIAlertController.Style.actionSheet` or `UIAlertController.Style.alert`. You can write the code like this when creating a UIAlertController:

```
let optionMenu = UIAlertController(title: nil, message: "What do you want
to do?", preferredStyle: UIAlertController.Style.actionSheet)
```

There is nothing wrong with the code above. Swift gives developers a shorthand and helps us type less code. Because the type of the `preferredStyle` parameter is already known (i.e. `UIAlertController.Style`), Swift lets you use a shorter dot syntax by omitting `UIAlertController.Style`. This is why we instantiate the `UIAlertController` object like this:

```
let optionMenu = UIAlertController(title: nil, message: "What do you want
to do?", preferredStyle: .actionSheet)
```

The same applies to `UIAlertAction.Style`. The `UIAlertAction.Style` is an enumeration with three possible values: *default*, *cancel* and *destructive*. When creating the `cancelAction` object, we also use the shorthand syntax:

```
let cancelAction = UIAlertAction(title: "Cancel", style: .cancel, handler:
nil)
```

Adding Actions to the Alert Controller

Now let's add two more actions to the alert controller:

- **Reserve a table action** - A fake feature to demo how to present an alert message. We will not implement this feature but simply display an alert prompt on screen.
- **Mark as Favorite action** - When selected, this option marks the selected restaurant as a favorite restaurant.

In the `tableView(_:didSelectRowAt:)` method, add the following code for the "Call" action. You can insert the code after the initialization of `cancelAction`:

```
// Add "Reserve a table" action
let reserveActionHandler = { (action:UIAlertAction!) -> Void in

    let alertMessage = UIAlertController(title: "Not available yet", messa
ge: "Sorry, this feature is not available yet. Please retry later.", prefe
rredStyle: .alert)
    alertMessage.addAction(UIAlertAction(title: "OK", style: .default, han
dler: nil))
    self.present(alertMessage, animated: true, completion: nil)

}

let reserveAction = UIAlertAction(title: "Reserve a table", style: .default
, handler: reserveActionHandler)
optionMenu.addAction(reserveAction)
```

In the code above, the `reserveActionHandler` object is new to you. As mentioned before, you can specify a block of code as the handler when creating a `UIAlertAction` object. The block of code will be executed when the user selects the action. Previously, we specify `nil` in the `cancelAction` object. That means we do not have any follow-up action for the *Cancel* button.

For the `reserveAction` object, it is a bit different. We assign it with `reserveActionHandler`. The code block displays an alert, telling the user that the feature is not yet available.

Technically, this block of code is known as *Closure* in Swift. Closures are self-contained blocks of functionality that can be passed around in your code. It is very similar to *blocks* in Objective-C. Like the example above, one way to provide the action closure is to declare it as a constant or a variable with the block of code as the value. The first part of the code block is identical to the definition of the

handler parameter of `UIAlertAction`. The `in` keyword indicates that the definition of the closure's parameters and return type has finished, and the body of the closure will begin. Figure 10-16 illustrates the syntax of a closure.

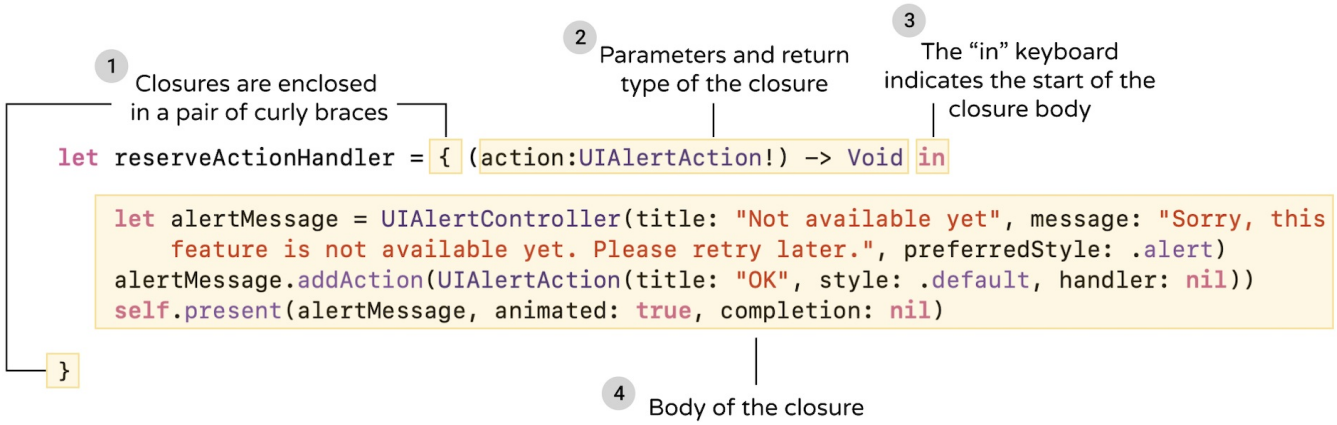


Figure 10-16. How you write a closure

The code specified in the body of the closure shouldn't be new to you. We create a `UIAlertController` to display the warning. However, instead of using the `.actionSheet` style, we use `.alert`. Later when you test the app, you will understand how an alert looks like.

Once we prepared the action handler, we create the `UIAlertAction` object and set its title to "Reserve a table". Most importantly, we assign the `reserveActionHandler` to the action object, followed by adding the action object to the option menu.

```
let reserveAction = UIAlertAction(title: "Reserve a table", style: .default
, handler: reserveActionHandler)
optionMenu.addAction(reserveAction)
```

Next, insert the following lines of code for the *Mark as favorite* action:

```
// Mark as favorite action
let favoriteAction = UIAlertAction(title: "Mark as favorite", style: .defa
ult, handler: {
    (action:UIAlertAction!) -> Void in

    let cell = tableView.cellForRow(at: indexPath)
    cell?.accessoryType = .checkmark
})
optionMenu.addAction(favoriteAction)
```

The above code shows you another way to use closure. You can write the closure inline as a parameter of the handler.

Optionals in Swift

Do you still remember what the question mark is for? If you've studied chapter 2, I hope you can answer the question. The cell is known as an optional in Swift. As mentioned before, Optional is a new type introduced in Swift. It simply means 'there is a value' or 'there isn't a value at all'. The cell returned by `tableView.cellForRow(at:)` is an optional. To access the `accessoryType` property of the cell, you use the question mark. In this case, Swift will check if the cell has a value and allow you to set the value of `accessoryType` if the cell exists. In most cases, the autocomplete feature of Xcode automatically adds the question mark for you when accessing a property of an optional.

When a user selects the "Mark as favorite" option, we add a checkmark to the

selected cell. For a table view cell, the right part is reserved for an accessory view. There are four types of built-in accessory views including *disclosure indicator*, *detail disclosure button*, *checkmark* and *detail*. In this case, we use *checkmark* as the indicator.

The first line of the code block retrieves the selected table cell using `indexPath`, which contains the index of the selected cell. The second line updates the `accessoryType` property of the cell with a check mark.

By default, the checkmark will be displayed in blue. Optionally, you can change its color by setting the `tintColor` property of a cell. In the `awakeFromNib()` method of `RestaurantTableViewCell`, you can add a line of code to set the cell's tint color:

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

    self.tintColor = .systemYellow
}
```

Compile and run the app. Tap a restaurant and choose one of the actions, it'll either show you a check mark or an alert.

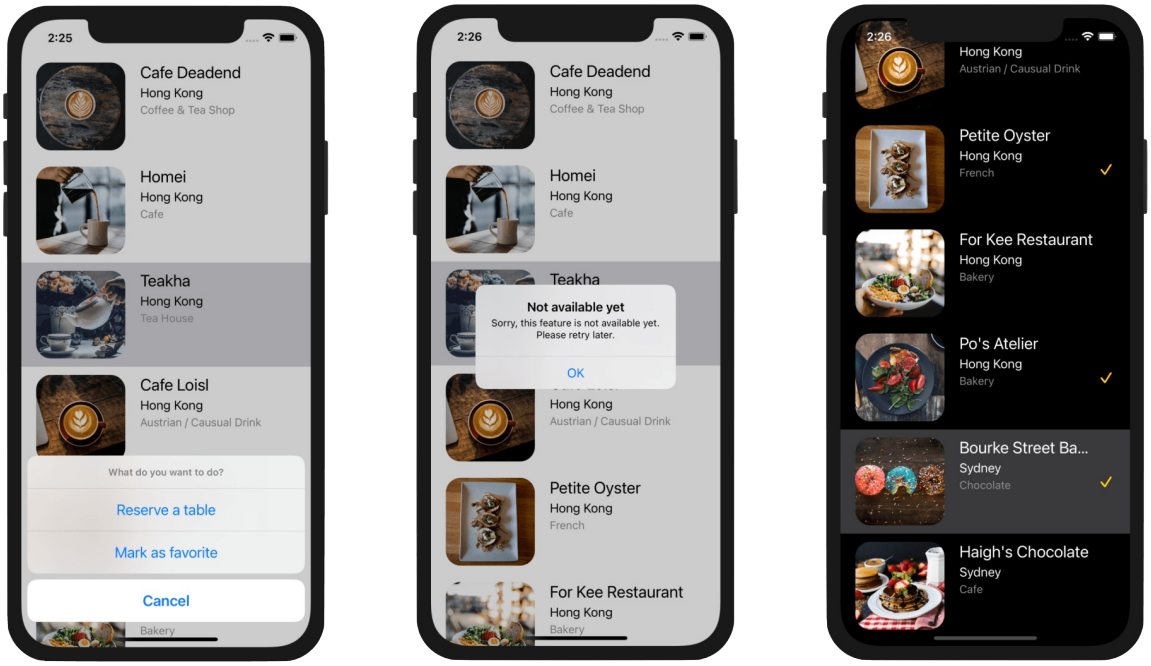


Figure 10-17. Reserve a table and Mark as favorite actions

For now, when you select a row, the row is highlighted in gray and stayed as selected. Add the following code at the end of the `tableView(_:didSelectRowAt:)` method to deselect the row.

```
tableView.deselectRow(at: indexPath, animated: false)
```

Your final code snippet of the `tableView(_:didSelectRowAt:)` method should like this:

```
override func tableView(_ tableView: UITableView, didSelectRowAt indexPath
: IndexPath) {

    // Create an option menu as an action sheet
    let optionMenu = UIAlertController(title: nil, message: "What do you w
ant to do?", preferredStyle: .actionSheet)

    // Add actions to the menu
```

```
    let cancelAction = UIAlertAction(title: "Cancel", style: .cancel, hand
ler: nil)
    optionMenu.addAction(cancelAction)

    // Add "Reserve a table" action

    let reserveActionHandler = { (action:UIAlertAction!) -> Void in

        let alertMessage = UIAlertController(title: "Not available yet", m
essage: "Sorry, this feature is not available yet. Please retry later.", p
referredStyle: .alert)
        alertMessage.addAction(UIAlertAction(title: "OK", style: .default,
 handler: nil))
        self.present(alertMessage, animated: true, completion: nil)

    }

    let reserveAction = UIAlertAction(title: "Reserve a table", style: .de
fault, handler: reserveActionHandler)
    optionMenu.addAction(reserveAction)

    // Mark as favorite action

    let favoriteAction = UIAlertAction(title: "Mark as favorite", style: .
default, handler: {
        (action:UIAlertAction!) -> Void in

        let cell = tableView.cellForRow(at: indexPath)
        cell?.accessoryType = .checkmark

    })
    optionMenu.addAction(favoriteAction)

    // Display the menu
    present(optionMenu, animated: true, completion: nil)

    // Deselect the row
    tableView.deselectRow(at: indexPath, animated: false)
}
```

We Hit a Bug

The app looks great. If you look at it closely, however, there is a bug in the app. Say, you mark the "Cafe Deadend" restaurant using the *Mark as favorite* action. If you scroll down the table, you'll find that another restaurant (e.g. Palomino Espresso) also contains a checkmark. What's the problem? Why did the app add an extra checkmark?

> Like every programmer, I hate bugs especially when facing a project deadline. However it's always the bugs that help me improve my programming skills. You'll hit a lot of bugs too as you continue to learn. Just get used to it.

The problem is due to cell reuse, that we have discussed in the earlier chapter. For example, the table view has 30 cells. For performance reasons, instead of creating 30 table cells, `UITableView` may just create 10 cells and reuses them as you scroll through the table. In this case, `UITableView` reuses the 1st cell (originally used for Cafe Deadend with a checkmark) for displaying another restaurant. In our code, we only update the image view and labels when the table view reuses the same cell. The accessory view is not updated. Thus, the next restaurant reusing the same cell shares the same accessory view. If the accessory view contains a checkmark, that restaurant will also carry a checkmark.

So how can we resolve the bug?

We have to find another way to keep track of the checked items. How about creating another array to save the checked restaurants? In the `RestaurantTableViewController.swift` file, declare a Boolean array:

```
var restaurantIsFavorites = Array(repeating: false, count: 21)
```

Bool is a data type in Swift that holds a Boolean value. Swift provides two Boolean values: *true* and *false*. We declare the `restaurantIsVisited` array to hold a collection of `Bool` values. Each value in the array indicates whether the corresponding restaurant is marked as "Check-in". For example, we can look into the value of `restaurantIsVisited[0]` to see if Cafe Deadend is checked or not.

The values in the array are initialized to `false`. In other words, the items are unchecked by default. The above line of code shows you a way to initialize an array in Swift with repeated values. The initialization is the same as the following:

```
var restaurantIsVisited = [false, false, false, false, false, false, false
, false, false, false, false, false, false, false, false, false, false, fa
lse, false, false, false]
```

We have to make a couple of changes in order to fix the bug. First, we need to update the value of the Bool array when a restaurant is marked as favorite. Add a line of code in the handler of the `reserveAction` object:

```
let favoriteAction = UIAlertAction(title: "Mark as favorite", style: .defa
ult, handler: {
    (action:UIAlertAction!) -> Void in

    let cell = tableView.cellForRow(at: indexPath)
    cell?.accessoryType = .checkmark
    cell?.tintColor = .systemYellow
    self.restaurantIsFavorites[indexPath.row] = true
})
```

The code is very straightforward. We update the value of the selected item from `false` to `true`.

Lastly, update the `configureDataSource` method and insert the following line of code in the closure of `cellProvider`:

```
cell?.accessoryType = self.restaurantIsFavorites[indexPath.row] ? .checkma
rk : .none
```

This is a simplified way to write a `if` condition. The `?` is known as the ternary conditional operator, an efficient shorthand for evaluating simple conditions. The purpose of this single code of code is exactly the same as the following:

```
if self.restaurantIsFavorites[indexPath.row] {
    cell.accessoryType = .checkmark
} else {
    cell.accessoryType = .none
}
```

Every time when the cell is rendered, we check if the restaurant to be displayed is marked as favorite. If the condition is `true`, we display a checkmark in the cell. Otherwise, just display nothing. Since we refer to the `restaurantIsFavorites` array to see if the restaurant should be marked, we can now display the accessory view correctly, even if the cell is reused.

Now, compile and run the app again. Your bug should now be resolved.

Working with iPad

Have you tried to run the app on iPad? It looks pretty well. However, since the cell stretches across the width of the device, you may find the cells are a bit too long especially when you turn the iPad device sideway.

From iOS 9 and onwards, you can use a property called `cellLayoutMarginsFollowReadableWidth` to adjust the cell width automatically. If you update the `viewDidLoad` method like this to enable the option:

```
override func viewDidLoad() {
    super.viewDidLoad()
    tableView.cellLayoutMarginsFollowReadableWidth = true
}
```

After the change, run the app on the iPad simulator again. You will immediately find the difference. It is up to you if you want to keep this option enabled or not.

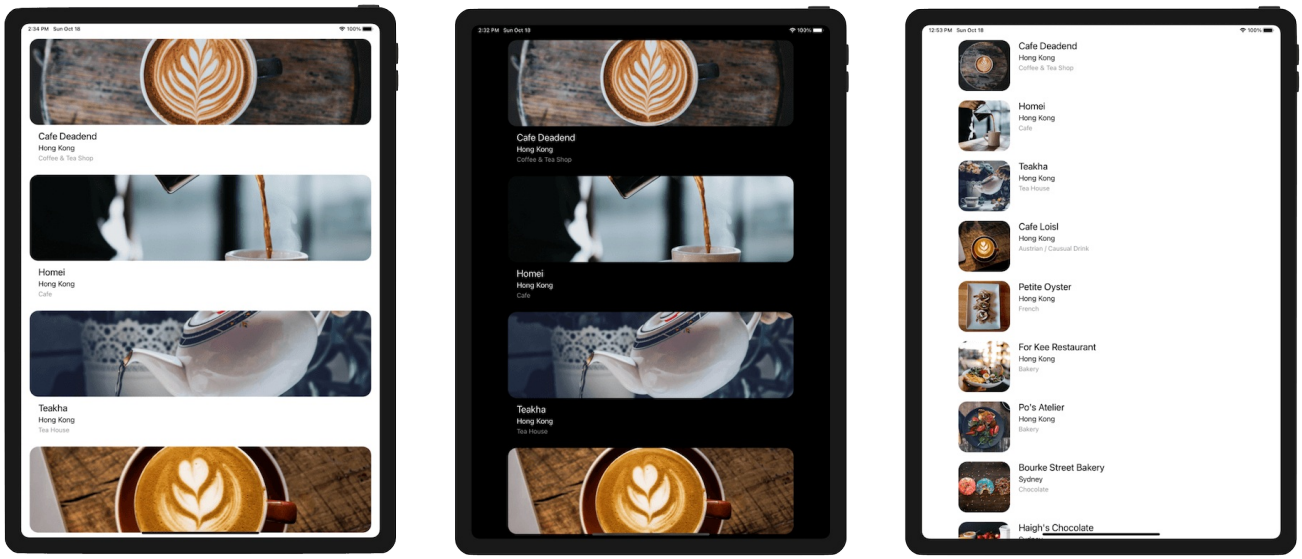


Figure 10-18. Table view with cellLayoutMarginsFollowReadableWidth disabled (left), Table view with cellLayoutMarginsFollowReadableWidth enabled (middle and right)

We Hit Another Bug on iPad

When you tap any of the table view cells on iPad, the app crashes and ends up with an error in console:

```
Terminating app due to uncaught exception 'NSGenericException', reason: 'Y
our application has presented a UIAlertController (<UIAlertController: 0x7
ff068897400>) of style UIAlertControllerStyleActionSheet from FoodPin.Rest
aurantTableViewController (<FoodPin.RestaurantTableViewController: 0x7ff06
9704d20>). The modalPresentationStyle of a UIAlertController with this sty
le is UIModalPresentationPopover. You must provide location information fo
r this popover through the alert controller's popoverPresentationControlle
r. You must provide either a sourceView and sourceRect or a barButtonItem.
  If this information is not known when you present the alert controller,
you may provide it in the UIPopoverPresentationControllerDelegate method -
prepareForPopoverPresentation.'
terminating with uncaught exception of type NSException
CoreSimulator 732.17 - Device: iPad Pro (11-inch) (2nd generation) (02AA48
3A-0CAA-4645-8B8F-90D3397769A3) - Runtime: iOS 14.1 (18A8394) - DeviceType
: iPad Pro (11-inch) (2nd generation)
```

In case that you can't see the console message, go up to the Xcode menu and choose *View* > *Debug Area* > *Activate Console*. Take a look at figure 10-19 if you don't know where the console is. The console message is very useful in helping you find out the root cause of the error. Whenever you experience an exception in your app, look into the console and dig out the error message.

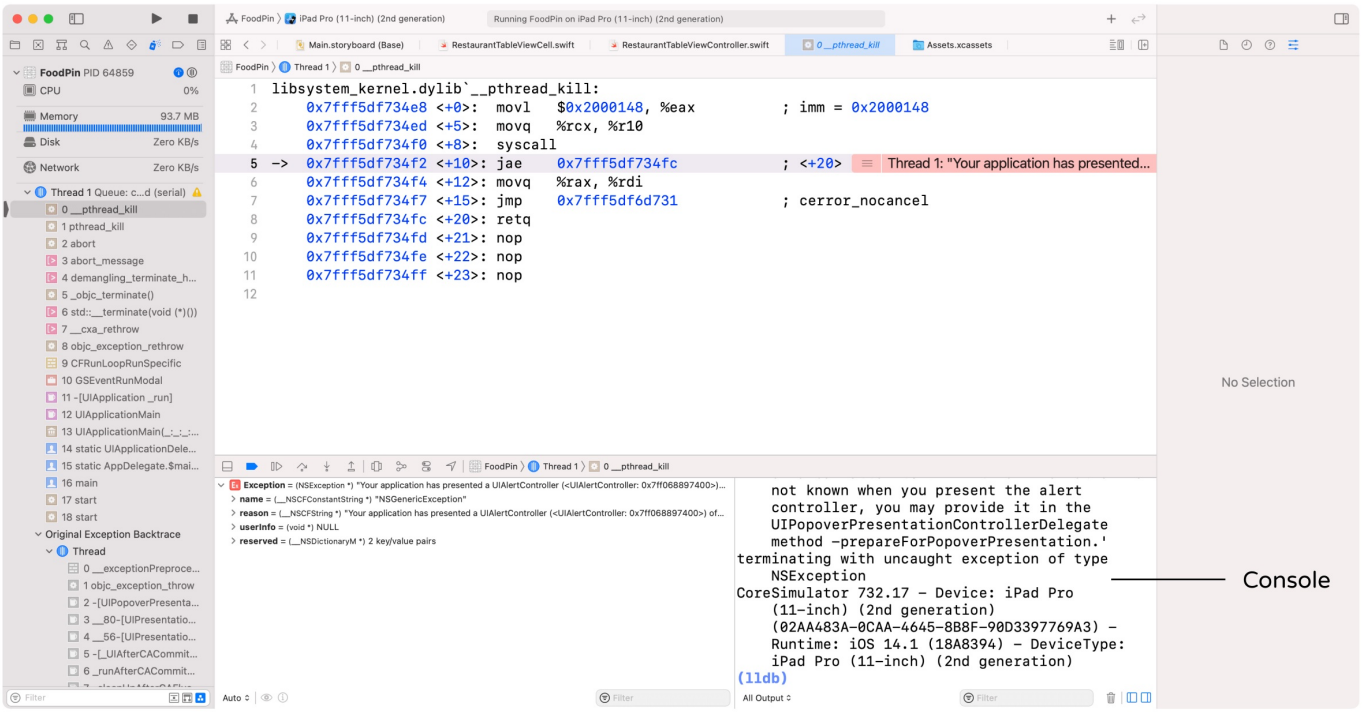


Figure 10-19. Console message

According to the error message, a UIActionController of style `.actionSheet` will behave differently on iPad. Instead of presenting the alert controller modally, iOS will present the controller using the popover style.

The presentation process is automatically handled by iOS. When the following line of the code is called, iOS will check if the device is an iPhone or iPad.

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

If it is an iPhone, the alert controller will be presented modally. On iPad, the alert controller will be presented in the popover style using the `UIPopoverPresentationController` object stored in its `popoverPresentationController` property.

Now try to look into the error message again. Can you find out the root cause of the error?

> You must provide location information for this popover through the alert controller's popoverPresentationController. You must provide either a sourceView and sourceRect or a barButtonItem.

When presenting the activity view controller using the popover style, you have to configure the source view of its `popoverPresentationController`. The source view indicates the view containing the anchor rectangle for the popover. You will understand what it means after implementing the popover.

Now, insert the following code in the `tableView(_:didSelectRowAt:)` method and place the code after the instantiation of `optionMenu`:

```
if let popoverController = optionMenu.popoverPresentationController {
    if let cell = tableView.cellForRow(at: indexPath) {
        popoverController.sourceView = cell
        popoverController.sourceRect = cell.bounds
    }
}
```

When the app is run on iPhone, the `popoverPresentationController` property of the alert controller (i.e `optionMenu`) is set to `nil`. Conversely, it will store the popover presentation controller when the app is run on iPad. Therefore, we use `if let` to check if the `popoverPresentationController` property has a value. If yes, we configure its `sourceView` to the cell that triggers the action. Optionally, we set the `sourceRect` property to the cell's bounds, so that the popover will be originated from the center of the cell.

Test the app on the iPad simulator again. It should work now.

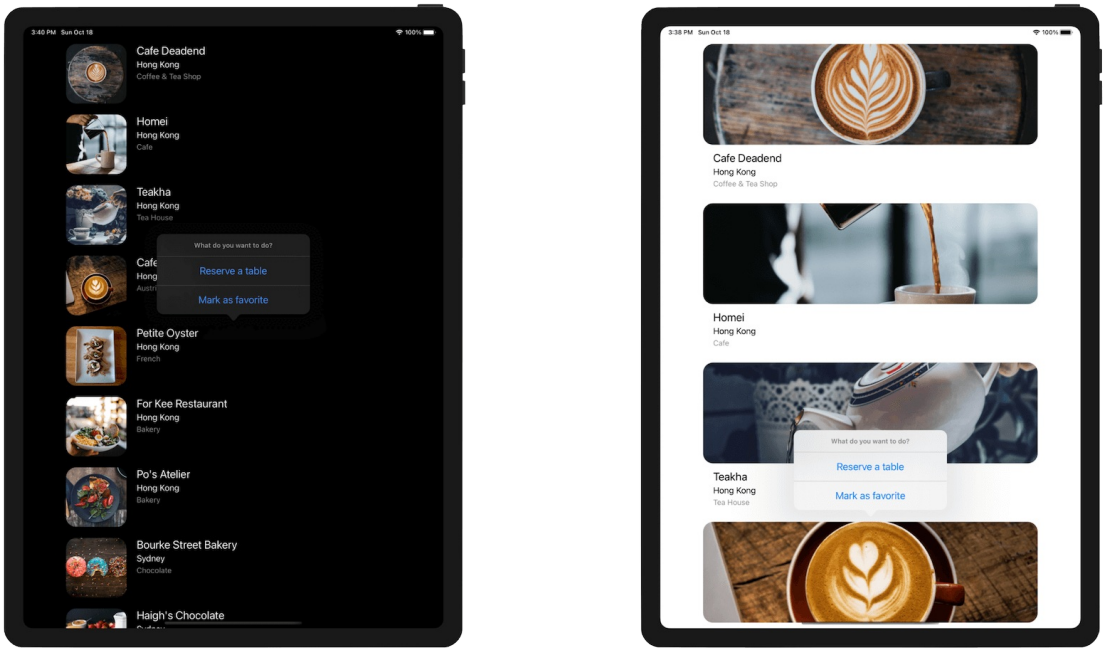


Figure 11-20. Popover menu on iPad

Exercise #1

Presently the app doesn't allow users to uncheck the checkmark. Think about how you can alter the code such that the app can toggle the checkmark. You'll also need to show a different title for the *Remove from favorites* button if the selected cell is marked. It's not too hard to make the changes. Take some time to work on the exercise. I'm sure you'll learn a lot.

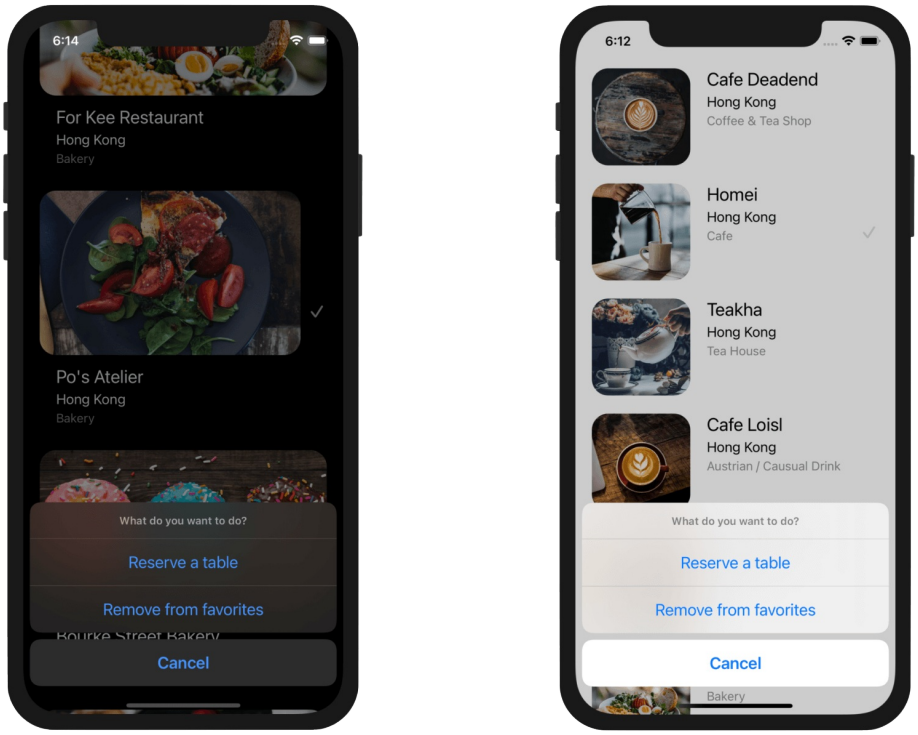


Figure 10-21. Removing a restaurant from favorites

Exercise #2

Instead of showing the default checkmark, replace it with a heart icon. Xcode has some built-in system images. After adding an image view, you can set the image to `heart.fill` to use the heart image built into the system. Figure 10-22 shows you the final layout.

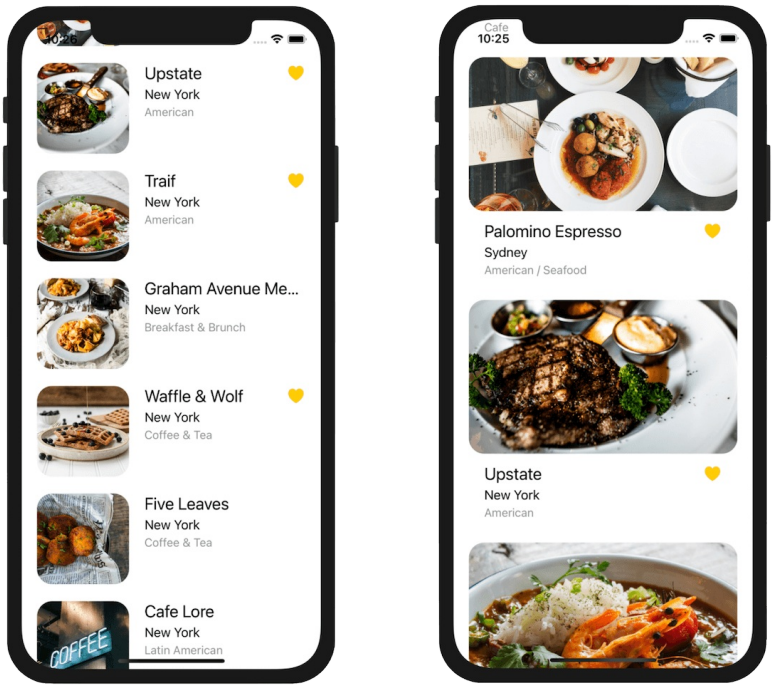


Figure 10-22. Using a heart icon to replace the default checkmark

You are expected to modify both prototype cells (i.e. `datacell` and `favoritecell`) to add the heart icon. Another hint is that you can use the `isHidden` property of an image view to control its visibility. For example, you can set the `isHidden` property to `true` to hide the image view like this:

```
cell.heartImageView.isHidden = true
```

I encourage you to take some time to work on this exercise too because it will help you revise everything you learned so far.

Summary

At this point, you should have a solid understanding of how to create table view, customize table cells and handle table row selection. You're ready to build a simple table view app on your own (e.g. a simple ToDo app). I always recommend you to create your own project. I don't mean you have to start a big one. If you love to travel, create a simple app that displays a list of your favorite destinations. If you love music, create your own app that shows a list of your favorite albums. Just play around with Xcode, make mistake and learn along the way.

For reference, you can download the complete Xcode project from http://www.appcoda.com/resources/swift53/FoodPinTableSelection.zip. For the solution of the exercise, you can download it from http://www.appcoda.com/resources/swift53/FoodPinTableSelectionExercise.zip.

In the next chapter, we'll continue to explore table view and see how you can delete a table row.

Chapter 11
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 a table-based iOS app, which works on both iPhone and iPad, using Interface Builder. We'll further enhance the FoodPin app and add more features. However, we will dive deeper into iOS app development and learn other APIs. I want to introduce you the basics of Object Oriented Programming and teach you how to write better code.

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 intentionally left it out when I began writing this book. I want to keep things interesting 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. After going through 10 chapters and you're still reading the book, I believe you're determined to learn iOS programming. And, I believe you really want to take your programming skills to the next level to become a professional developer.

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 `RestaurantTableViewController` 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. Take the Hello World app, that we've built at the very beginning, as an example. The `UIViewController` object is responsible for controlling the view of the app and its view object is used for holding the Hello World button. The `UIButton` (i.e. Hello World button) object implements a standard iOS button on the touch screen and listens to any touch events. On the other hand, the `UIAlertController` object displays an alert message to a user. After all, these objects work together to create the Hello World app.

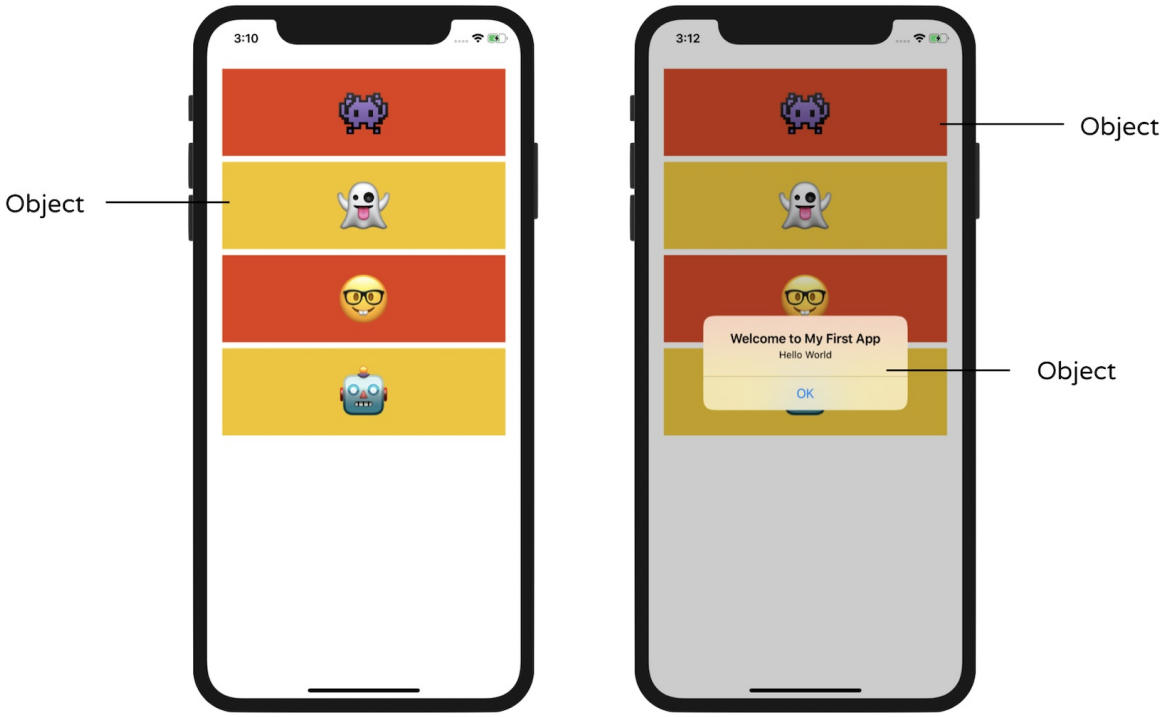


Figure 11-1. Sample Objects in Hello World App

In Object Oriented Programming, an object shares two characteristics: *properties* and *functionalities*. Let's consider a real world object – *Car*. A car has its own color, model, top speed, manufacturer, etc. These are the properties of a car. In terms of functionalities, a car should provide basic functions, such as accelerate, brake, steering, etc.

Software objects are conceptually similar to real-world objects. If we go back to the iOS world, let's take a look at the properties and functionalities of the `UIButton` object in the Hello World app:

- **Properties** – Background, size, color, and font are examples of the `UIButton` 's properties
- **Functionalities** – When the button is tapped, it recognizes the tap event. The ability to detect a touch is one of the many functions of `UIButton` .

In earlier chapters, you always come across the term *method*. In Swift, we create methods to provide the functionalities of an object. Usually, a method corresponds to a particular function of an object.

```
Note: You may wonder what is the difference between
functions and methods? In fact, they are functionally the
same. Functions defined in a class are known as methods.
```

Classes, Objects and Instances

Other than methods and objects, you have come across terms like *instance* and *class* . All these are the common terms in OOP. Let me give you a brief explanation.

A class is a blueprint or prototype from which objects are created. Basically, a class consists of properties and methods. Let's say, we want to define a `Course` class. A Course class contains properties, such as *name*, *course code*, and the *total number*

of students.

This class represents the blueprint of a course. We can use it to create different courses like iOS Programming course with course code *IPC101*, Cooking course with course code *CC101*, etc. Here, the "iOS Programming course" and "Cooking course" are known as the *objects* of the `Course` class. We typically refer a single course as an *instance* of the `Course` class. For the sake of simplicity, the term *instance* and *object* are sometimes interchangeable.

> A blueprint for a house design is like a class description. All the houses built from that blueprint are objects of that class. A given house is an instance.
>
> Source: http://stackoverflow.com/questions/3323330/difference-between-object-and-instance

Structures

> *Structures* and *classes* are general-purpose, flexible constructs that become the building blocks of your program's code. You define properties and methods to add functionality to your structures and classes using the same syntax you use to define constants, variables, and functions.
>
> - Apple Documentation (https://docs.swift.org/swift-book/LanguageGuide/ClassesAndStructures.html)

Other than classes, you can use use structures (or structs) in Swift to create your own type with properties and methods. *Structs* in Swift are very similar to *Classes* and have a lot in common. Both can define properties to store values and methods to provide functionalities. Both can create their own initializers to configure the initial state of the object.

Inheritance is one of the important features of Object Oriented Programming. Recalled that we have created the `RestaurantTableViewController` class, which is a subclass of the `UITableViewController`, this is a sample usage of class inheritance. The `RestaurantTableViewController` class inherits all the properties and methods of `UITableViewController`. Whenever we need to provide specialization for a particular function, we override the original function of the super class.

Structs, however, do not allow inheritance. You can't inherit one struct from another. This is one of main differences between Classes and Structs in Swift.

Types in Swift fall into two categories: *value types* and *reference types*. All structures in Swift are value types, while classes are reference types. This is another core difference between Classes and Structs. For structs, each instance has a unique copy of its data. On the other hand, reference types (classes) share a single copy of the data. When you assign an instance of a class to another variable, rather than copy the data of that instance, a reference to that instance is used.

Let me demonstrate the difference between value types (struct) and reference types (class) using an example. In the code snippet below, we have a class named `Car` with a property named `brand`. We create an instance of `Car` and assign it to a `car1` variable. Then we assign `car1` to another variable named `car2`. Lastly, we modify the `brand` value of `car1`.

```
class Car {
    var brand = "Tesla"
}

var car1 = Car()
var car2 = car1

car1.brand = "Audi"
print(car2.brand)
```

Can you guess the `brand` value of `car2`? Should it be Tesla or Audi? The answer is *Audi*. This is the nature of reference types. Both `car1` and `car2` are referring to the same instance, sharing the same copy of the data.

Contrarily, if you rewrite the same piece of code with struct (i.e. value types), you will see a different result.

```
struct Car {
    var brand = "Tesla"
}

var car1 = Car()
var car2 = car1

car1.brand = "Audi"
print(car2.brand)
```

In this case, only the `brand` value of `car1` is updated to Audi. For `car2`, its brand is still Tesla because each variable of value types has its own copy of data. Figure 11-2 illustrates the difference between classes and structs visually.

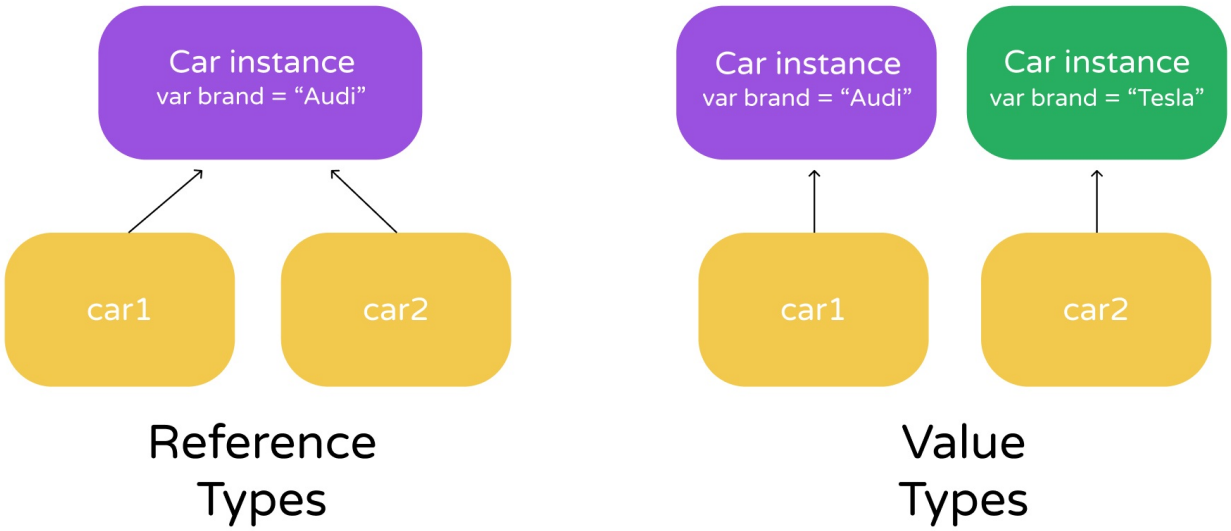


Figure 11-2. Illustrates the difference between value types and reference types

Since both classes and strutures provides similar features, which one should you use? In general, use structures to create your own type by default. This is a recommendation from Apple (https://developer.apple.com/documentation/swift/choosing_between_structures_and_classes). In case if you need additional functionalities like inheritance, choose classes over structs.

Revisit the FoodPin Project

So, why do we cover OOP in this chapter? There is no better way to explain the concept than showing you an example. Take a look at the FoodPin project (http://www.appcoda.com/resources/swift53/FoodPinTableSelectionExercise.zip) again.

In the `RestaurantTableViewController` class, we created multiple arrays to store the names, types, locations, and images of the restaurants.

```
var restaurantNames = ["Cafe Deadend", "Homei", "Teakha", "Cafe Loisl", "P
etite Oyster", "For Kee Restaurant", "Po's Atelier", "Bourke Street Bakery"
, "Haigh's Chocolate", "Palomino Espresso", "Upstate", "Traif", "Graham Av
enue Meats", "Waffle & Wolf", "Five Leaves", "Cafe Lore", "Confessional",
"Barrafina", "Donostia", "Royal Oak", "CASK Pub and Kitchen"]

var restaurantImages = ["cafedeadend", "homei", "teakha", "cafeloisl", "pe
titeoyster", "forkeerestaurant", "posatelier", "bourkestreetbakery", "haig
hschocolate", "palominoespresso", "upstate", "traif", "grahamavenuemeats",
"wafflewolf", "fiveleaves", "cafelore", "confessional", "barrafina", "dono
stia", "royaloak", "caskpubkitchen"]

var restaurantLocations = ["Hong Kong", "Hong Kong", "Hong Kong", "Hong Ko
ng", "Hong Kong", "Hong Kong", "Hong Kong", "Sydney", "Sydney", "Sydney",
"New York", "New York", "New York", "New York", "New York", "New York", "N
ew York", "London", "London", "London", "London"]

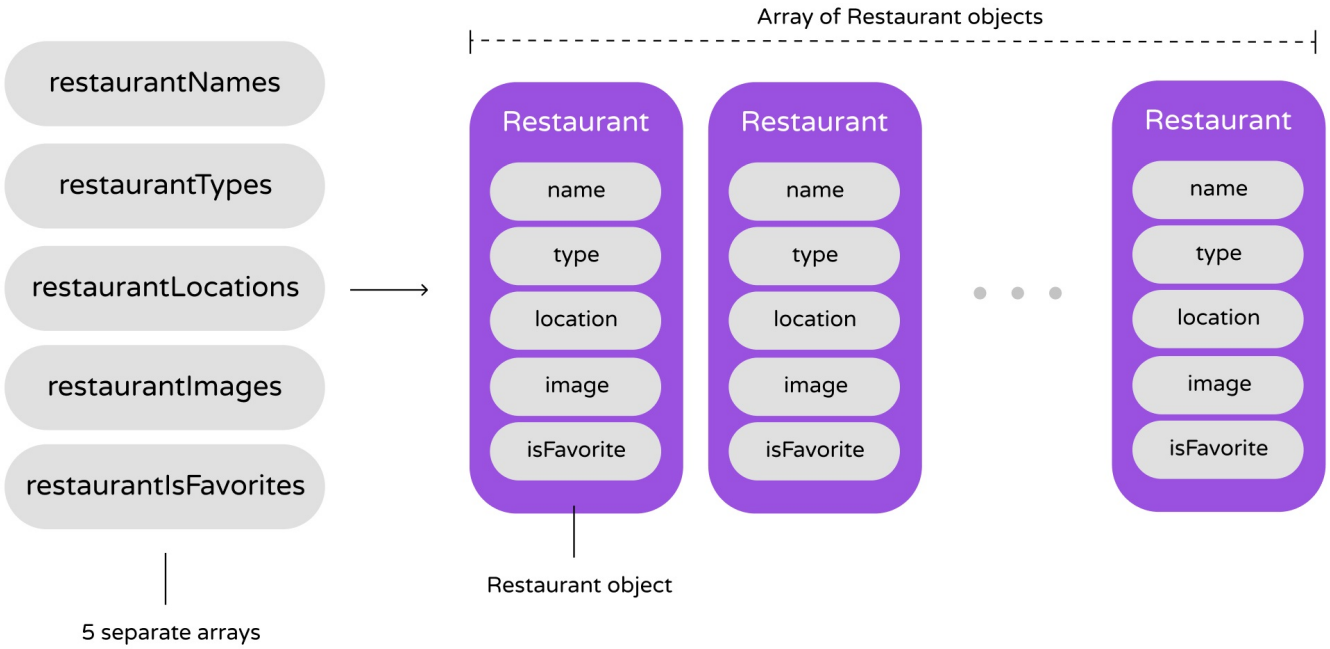
var restaurantTypes = ["Coffee & Tea Shop", "Cafe", "Tea House", "Austrian
 / Causual Drink", "French", "Bakery", "Bakery", "Chocolate", "Cafe", "Ame
rican / Seafood", "American", "American", "Breakfast & Brunch", "Coffee &
Tea", "Coffee & Tea", "Latin American", "Spanish", "Spanish", "Spanish", "
British", "Thai"]

var restaurantIsFavorites = Array(repeating: false, count: 21)
```

All these data are actually related to a list of restaurants. Why do we need to separate them into multiple arrays? Have you ever wondered if we can group these data together?

In Object Oriented Programming, these data can be characterized as the properties of a restaurant. Instead of storing these data in separate arrays, we can create a `Restaurant` structure to model a restaurant and store multiple restaurants in an array of `Restaurant` objects.

In Object Oriented Programming, these data can be characterized as the properties of a restaurant. Instead of storing these data in separate arrays, we can create a `Restaurant` structure to model a restaurant and store multiple restaurants in an array of `Restaurant` objects.



Now let's tweak the FoodPin project, create the `Restaurant` struct and convert the code to use a list of `Restaurant` objects.

Creating a Restaurant Struct

First, we'll start with the `Restaurant` struct. In the project navigator, right click on the `FoodPin` folder and select "New File...". This time we are not extending the UI objects provided by the iOS SDK. Instead, we are going to create a brand new struct. So, choose the "Swift File" template under Source and click "Next". Name the file `Restaurant.swift` and save it in the project folder.

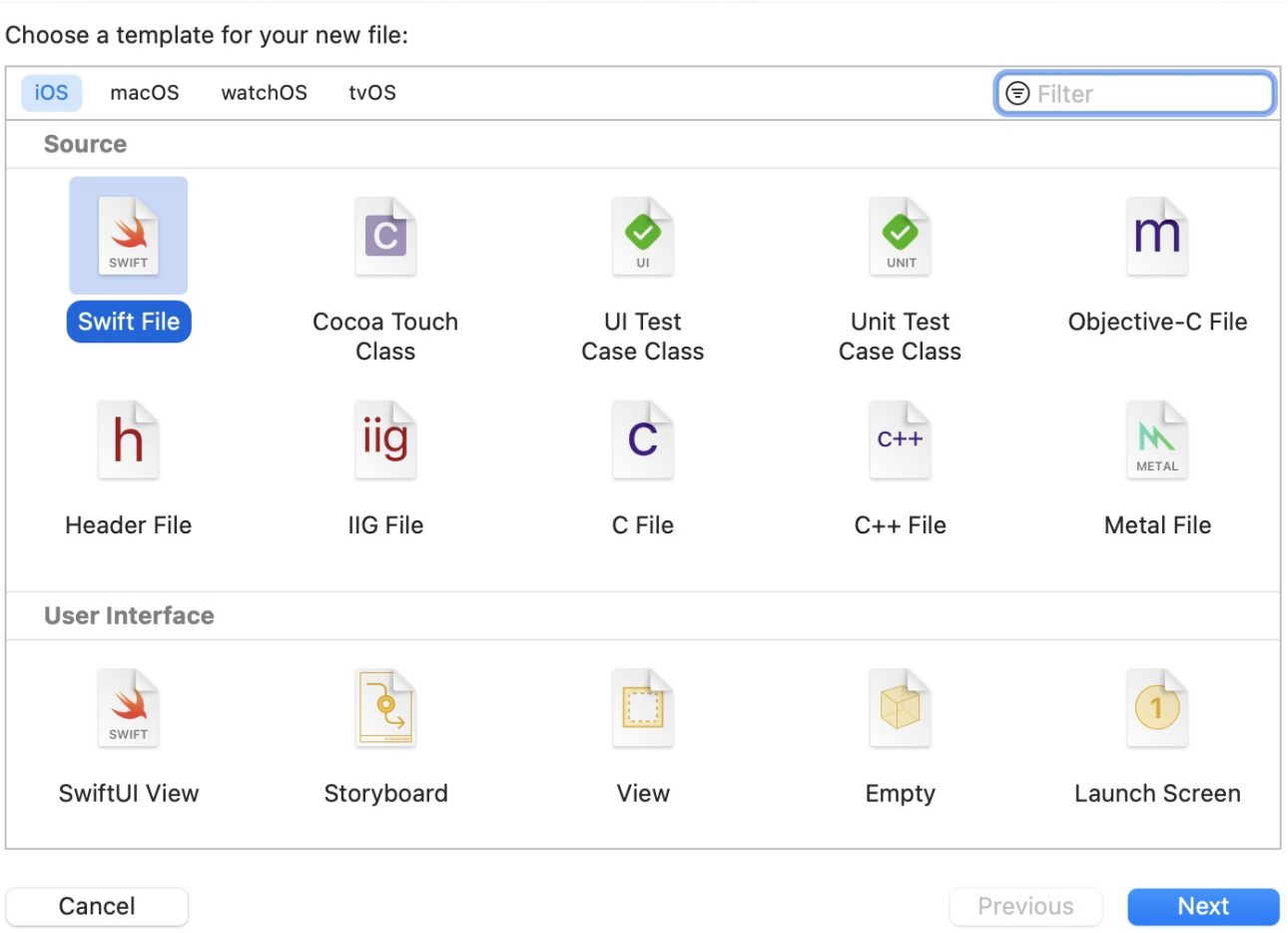


Figure 11-4. Creating a new class using the Swift File template

Once completed, declare the `Restaurant` struct in the `Restaurant.swift` file using the below code:

```
struct Restaurant {
    var name: String
    var type: String
    var location: String
    var image: String
    var isFavorite: Bool

    init(name: String, type: String, location: String, image: String, isFa
vorite: Bool) {
        self.name = name
        self.type = type
        self.location = location
        self.image = image
        self.isFavorite = isFavorite
    }

    init() {
        self.init(name: "", type: "", location: "", image: "", isFavorite:
false)
    }
}
```

You use the `struct` keyword to define a structure. The code above defines a `Restaurant` struct with five properties including `name`, `type`, `location`, `image` and `isFavorite`. Except for `isFavorite`, which is a boolean (i.e. `Bool`), the rest of the properties are of the type `String`. You can either set a default value or explicitly specify the type for each property. Here we demonstrate the latter option.

Initializers Explained

Initialization is the process of preparing an instance of a structure (or class). When you create an object, the initializer is called for setting an initial value for each stored property on that instance and performing any extra setup, before the

instance is ready to use. You use the `init` keyword to define an initializer. In its simplest form, it looks like this:

```
init() {

}
```

You can also customize an initializer to take input parameters, just like the one we have defined in the `Restaurant` struct. Our initializer has five parameters. Each of them is given a name and explicitly specified with a type. In the initializer, it initializes the values of the property with the given values.

To create an instance of the `Restaurant` struct, the syntax is like this:

```
Restaurant(name: "Thai Cafe", type: "Thai", location: "London", image: "th
aicafe", isFavorite: false)
```

You are allowed to define multiple initializers that accepts different parameters. In the code, we created another initializer for convenience purpose.

```
init() {
    self.init(name: "", type: "", location: "", image: "", isFavorite: fal
se)
}
```

Without this initializer, you can initialize an empty `Restaurant` object like this:

```
Restaurant(name: "", type: "", location: "", image: "", isFavorite: false)
```

Now with the convenience initializer, you can initialize the same object like this:

```
Restaurant()
```

It saves you time from typing all the initialization parameters every time you need to initialize an empty `Restaurant` object.

The self Keyword

What's the `self` keyword in the initializer? In Swift, you use `self` to differentiate between property names and arguments in initializers. Because the arguments have the same name as the properties, we use `self` to refer to the property of the class.

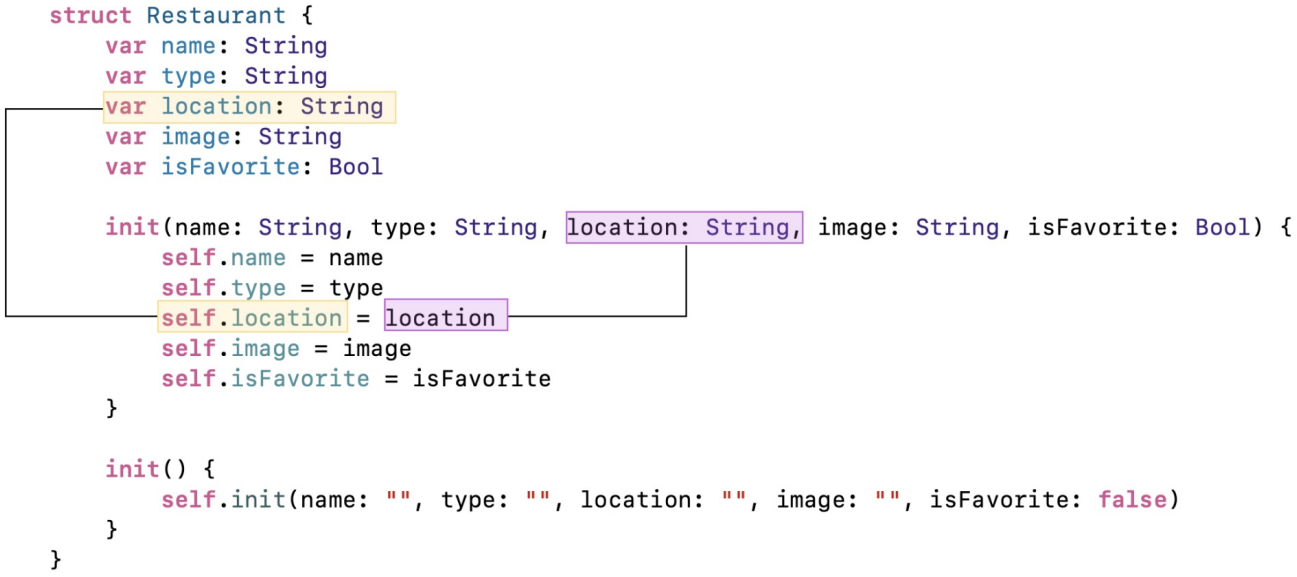


Figure 11-5. The use of self keyword

Default Initializers

You can actually assign each property with a default value and omit the initializator. Swift will generate the default initalizer behind the scene. Therefore, a short version of the `Restaurant` struct can be written like this:

```
struct Restaurant {
    var name: String = ""
    var type: String = ""
    var location: String = ""
    var image: String = ""
    var isFavorite: Bool = false
}
```

Using the Array of Restaurant Objects

With a basic understanding of classes, structs, and object initializations, let's go back to the FoodPin project and combine the existing arrays into an array of `Restaurant` objects. First, delete the restaurant-related arrays from the `RestaurantTableViewController` class:

```
var restaurantNames = ["Cafe Deadend", "Homei", "Teakha", "Cafe Loisl", "P
etite Oyster", "For Kee Restaurant", "Po's Atelier", "Bourke Street Bakery"
, "Haigh's Chocolate", "Palomino Espresso", "Upstate", "Traif", "Graham Av
enue Meats", "Waffle & Wolf", "Five Leaves", "Cafe Lore", "Confessional",
"Barrafina", "Donostia", "Royal Oak", "CASK Pub and Kitchen"]

var restaurantImages = ["cafedeadend", "homei", "teakha", "cafeloisl", "pe
titeoyster", "forkee", "posatelier", "bourkestreetbakery", "haigh", "palom
ino", "upstate", "traif", "graham", "waffleandwolf", "fiveleaves", "cafelo
re", "confessional", "barrafina", "donostia", "royaloak", "cask"]

var restaurantLocations = ["Hong Kong", "Hong Kong", "Hong Kong", "Hong Ko
ng", "Hong Kong", "Hong Kong", "Hong Kong", "Sydney", "Sydney", "Sydney",
"New York", "New York", "New York", "New York", "New York", "New York", "N
ew York", "London", "London", "London", "London"]

var restaurantTypes = ["Coffee & Tea Shop", "Cafe", "Tea House", "Austrian
 / Causual Drink", "French", "Bakery", "Bakery", "Chocolate", "Cafe", "Ame
rican / Seafood", "American", "American", "Breakfast & Brunch", "Coffee &
Tea", "Coffee & Tea", "Latin American", "Spanish", "Spanish", "Spanish", "
British", "Thai"]

var restaurantIsFavorites = Array(repeating: false, count: 21)
```

Instead of using the above arrays, replace them with a new array of `Restaurant` objects:

```
var restaurants:[Restaurant] = [
    Restaurant(name: "Cafe Deadend", type: "Coffee & Tea Shop", location:
"Hong Kong", image: "cafedeadend", isFavorite: false),
    Restaurant(name: "Homei", type: "Cafe", location: "Hong Kong", image:
"homei", isFavorite: false),
    Restaurant(name: "Teakha", type: "Tea House", location: "Hong Kong", i
mage: "teakha", isFavorite: false),
    Restaurant(name: "Cafe loisl", type: "Austrian / Causual Drink", locat
ion: "Hong Kong", image: "cafeloisl", isFavorite: false),
```

```
    Restaurant(name: "Petite Oyster", type: "French", location: "Hong Kong"
, image: "petiteoyster", isFavorite: false),
    Restaurant(name: "For Kee Restaurant", type: "Bakery", location: "Hong
 Kong", image: "forkee", isFavorite: false),
    Restaurant(name: "Po's Atelier", type: "Bakery", location: "Hong Kong"
, image: "posatelier", isFavorite: false),
    Restaurant(name: "Bourke Street Backery", type: "Chocolate", location:
"Sydney", image: "bourkestreetbakery", isFavorite: false),
    Restaurant(name: "Haigh's Chocolate", type: "Cafe", location: "Sydney"
, image: "haigh", isFavorite: false),
    Restaurant(name: "Palomino Espresso", type: "American / Seafood", loca
tion: "Sydney", image: "palomino", isFavorite: false),
    Restaurant(name: "Upstate", type: "American", location: "New York", im
age: "upstate", isFavorite: false),
    Restaurant(name: "Traif", type: "American", location: "New York", imag
e: "traif", isFavorite: false),
    Restaurant(name: "Graham Avenue Meats", type: "Breakfast & Brunch", lo
cation: "New York", image: "graham", isFavorite: false),
    Restaurant(name: "Waffle & Wolf", type: "Coffee & Tea", location: "New
 York", image: "waffleandwolf", isFavorite: false),
    Restaurant(name: "Five Leaves", type: "Coffee & Tea", location: "New Y
ork", image: "fiveleaves", isFavorite: false),
    Restaurant(name: "Cafe Lore", type: "Latin American", location: "New Y
ork", image: "cafelore", isFavorite: false),
    Restaurant(name: "Confessional", type: "Spanish", location: "New York"
, image: "confessional", isFavorite: false),
    Restaurant(name: "Barrafina", type: "Spanish", location: "London", ima
ge: "barrafina", isFavorite: false),
    Restaurant(name: "Donostia", type: "Spanish", location: "London", imag
e: "donostia", isFavorite: false),
    Restaurant(name: "Royal Oak", type: "British", location: "London", ima
ge: "royaloak", isFavorite: false),
    Restaurant(name: "CASK Pub and Kitchen", type: "Thai", location: "Lond
on", image: "cask", isFavorite: false)
]
```

Once you replaced the original arrays with the `restaurants` array, you'll end up with a few errors in Xcode because some of the code still refer to the old arrays.

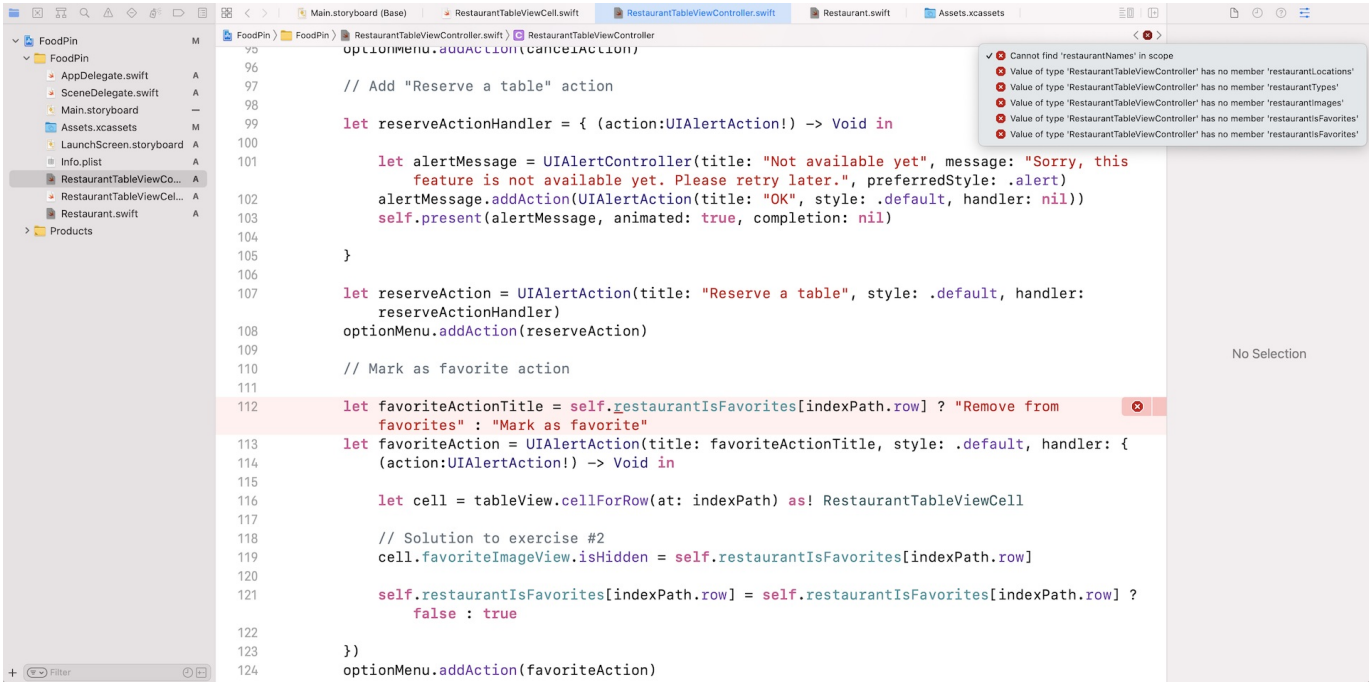


Figure 11-6. You can click the cross indicator to reveal the errors

To fix the errors, we have to modify the code to use the new `restaurants` array. First, update the `configureDataSource()` method like this:

```
func configureDataSource() -> UITableViewDiffableDataSource<Section, Resta
urant> {

    let cellIdentifier = "favoritecell"

    let dataSource = UITableViewDiffableDataSource<Section, Restaurant>(
        tableView: tableView,
        cellProvider: {  tableView, indexPath, restaurant in
            let cell = tableView.dequeueReusableCell(withIdentifier: cellI
dentifier, for: indexPath) as! RestaurantTableViewCell

            cell.nameLabel.text = restaurant.name
            cell.locationLabel.text = restaurant.location
            cell.typeLabel.text = restaurant.type
            cell.thumbnailImageView.image = UIImage(named: restaurant.imag
e)
            cell.favoriteImageView.isHidden = restaurant.isFavorite ? false
 : true

            return cell
        }
    )

    return dataSource
}
```

Since we now use an array of `Restaurant` object instead of an array of restaurant names to model the restaurants, the item type of `UITableViewDiffableDataSource` is now changed to `Restaurant`. For the `cellProvider` closure, it now passes us a `Restaurant` object instead of a `String`. So, we can retrieve the restaurant data by accessing the properties of the `Restaurant` object.

Next, replace these lines of code in `viewDidLoad()`:

```
var snapshot = NSDiffableDataSourceSnapshot<Section, String>()
snapshot.appendSections([.all])
snapshot.appendItems(restaurantNames, toSection: .all)
```

With:

```
var snapshot = NSDiffableDataSourceSnapshot<Section, Restaurant>()
snapshot.appendSections([.all])
snapshot.appendItems(restaurants, toSection: .all)
```

Again, the item type of the diffable data source is now changed from `String` to `Restaurant`. Instead of appending the `restaurantNames` to the section, we add the items in the `restaurants` array.

Other than that, you should see a number of errors in the `tableView(_:didSelectRowAt:)` method since the `restaurantIsFavorites` array is no longer available. You can replace the code of `favoriteActionTitle` and `favoriteAction` like below:

```
let favoriteActionTitle = self.restaurants[indexPath.row].isFavorite ? "Re
move from favorites" : "Mark as favorite"
let favoriteAction = UIAlertAction(title: favoriteActionTitle, style: .def
ault, handler: {
    (action:UIAlertAction!) -> Void in

    let cell = tableView.cellForRow(at: indexPath) as! RestaurantTableView
Cell

    cell.favoriteImageView.isHidden = self.restaurants[indexPath.row].isFa
vorite

    self.restaurants[indexPath.row].isFavorite = self.restaurants[indexPat
h.row].isFavorite ? false : true

})
```

We now access the `isFavorite` property of the selected restaurant objects to check its value.

Do we fix all the errors? Not yet. If you scroll back to the `configureDataSource()` method and the `viewDidLoad()` method, you should see a bunch of errors, complaining the type `Restaurant`.

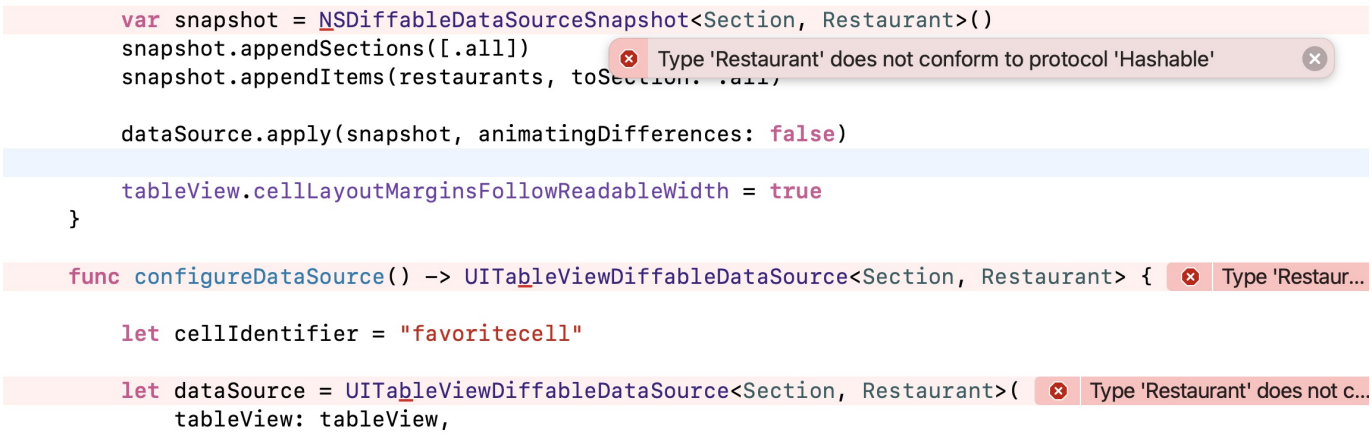


Figure 11-7. Error message for the Restaurant type

What's the problem of the `Restaurant` struct? What does the error message mean? The `UITableViewDiffableDataSource` class has two generic types, one for the section identifier and one for the item. For both types, it is required to conform to the `Hashable` protocol. Since the `Restaurant` type doesn't conform to `Hashable`, Xcode shows the error.

You may wonder why this error only shows up for this newly created `Restaurant` type. We didn't see this kind of error when using the `String` type. The reason is that the Swift standard types, such as `String` and `enum`, already conform to `Hashable`. For the `Restaurant` struct, however, we have to adopt the `Hashable` manually.

You do not need to write extra code to implement the protocol. All you need to do is declare the conformance like this:

```
struct Restaurant: Hashable {
    var name: String = ""
    var type: String = ""
    var location: String = ""
    var image: String = ""
    var isFavorite: Bool = false
}
```

This is a built-in feature of Swift. Since all the properties of the `Restaurant` struct are `Hashable`, the compiler will automatically generate the required code for adopting the `Hashable` protocol.

This should fix all the errors. You can now run your app. The look & feel of the app is exactly the same as before. However, we have refactored the code to use the new `Restaurant` struct. By combining multiple arrays into one, the code is now cleaner and more readable.

Organizing Your Xcode Project Files

As we continue to build our app, we will create even more files in the project folder. Therefore, I want to take this chance to show you a technique to better organize our projects.

Let's first look at the project navigator. All files you have created are placed at the top level of the `FoodPin` folder. As you add more files, it will be getting more difficult to locate your desired file. To better organize your project files, Xcode provides a *group* feature that lets you organize the files by group/folder.

There are a number of ways to group your files. You may group them by features or functions. One good practice is to group them by their responsibilities in MVC (Model-View-Controller). Say, for view controllers, they will be grouped under

`Controller`. Model classes like `Restaurant` will be grouped under `Model`. For custom views like `RestaurantTableViewCells`, they will be put under a group named `View`.

To create a group in the project navigator, right click the `FoodPin` folder and select *New Group* to create a new group. Name it `Controller`.

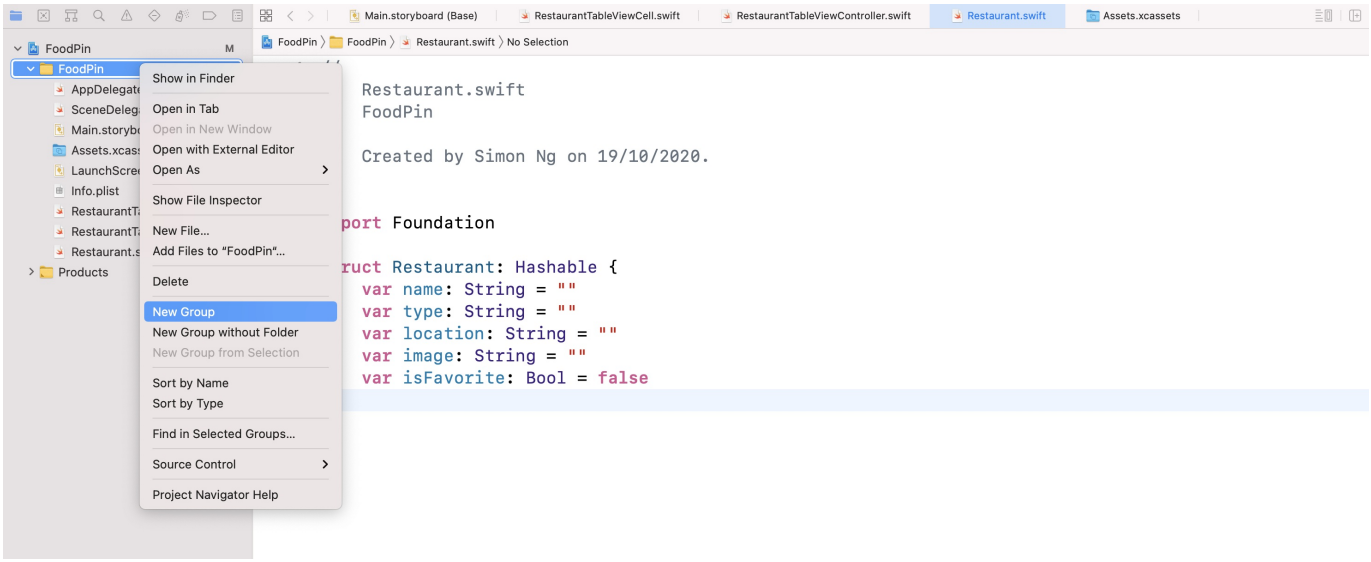


Figure 11-8. Right click FoodPin to create a new group

Next, select both `RestaurantTableViewController.swift` and drag them to the `Controller` group.

Repeat the same procedure to group other files like that shown in figure 11-9 (left). If you open your project folder in Finder, you will find that all files are nicely organized into folders (see figure 11-9 (right)). Each of which corresponds to the specific group in your Xcode project.

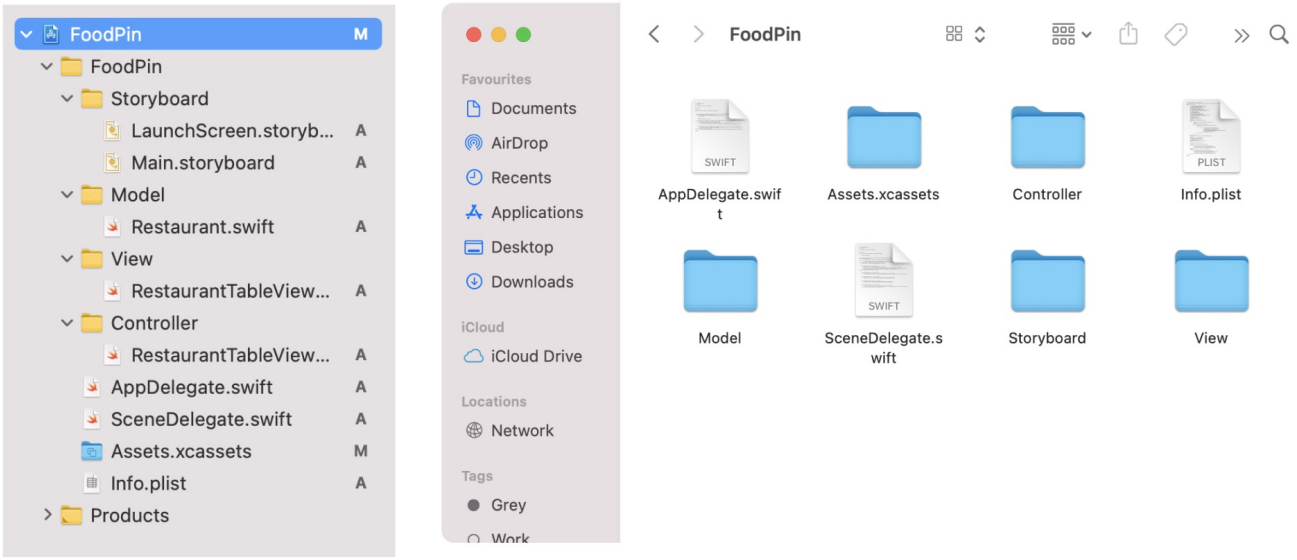


Figure 11-9. Organizing project files into groups (left), Each group is a subfolder of the FoodPin folder (right)

Even if you have moved around your files, you can still run the projects without any changes. Hit the Run button and try it out.

Documenting and Organizing Swift Code with MARK

> "Indeed, the ratio of time spent reading versus writing is well over 10 to 1. We are constantly reading old code as part of the effort to write new code. ... [Therefore,] making it easy to read makes it easier to write." – Robert C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship

Other than project files, there are some best practices to organizing the source code better. Here I am going to show you a powerful technique to organize your Swift code into useful and easy to read sections.

As you know, those lines of code which begins with `//` are comments. Comments serve as a note to yourself or other developers (if you are working in a team), providing extra information (e.g. intent) about the code. The primary purpose is just to make your code more easy to understand.

```
// Add "Reserve a table" action
```

You may have noticed that some comments have the `// MARK:` marker. Here is an example:

```
// MARK: - UITableViewDelegate Protocol
```

`MARK` is a special comment marker in Swift for organizing your code into easy-to-navigate sections. Say, for the `RestaurantTableViewController` class, some of the methods are related to the `UITableViewDelegate` protocol. And, some are related to view controller life cycle or the data source. We can then add the `MARK` comment to separate them into sections.

```
// MARK: - View controller life cycle

override func viewDidLoad() {

    .
    .
    .
}

// MARK: - UITableView Diffable Data Source

func configureDataSource() -> UITableViewDiffableDataSource<Section, Resta
urant> {

    .
    .
    .
}

// MARK: - UITableViewDelegate Protocol

override func tableView(_ tableView: UITableView, didSelectRowAt indexPath
: IndexPath) {

    .
    .
    .
}
```

Now when you click the jump bar on top of your editor window, you can see the methods are organized into different meaningful sections.

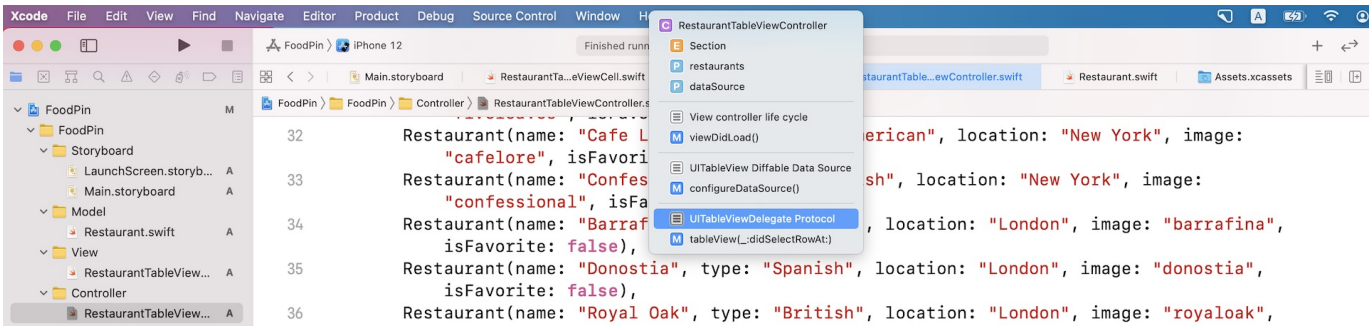


Figure 11-10. Organizing your Swift code with MARK

Summary

Congratulations if you made it this far. I hope you're not bored by the chapter. What I have covered is the basics of Object Oriented Programming. I also showed you some techniques for organizing your code and project files.

There are a lot more about the OOP concepts, such as polymorphism. However, we do not have time to discuss in-depth in this book. If you want to become a professional iOS developer, check out the references to learn more. It'll take you a lot of practices to truly pick up OOP. Anyhow, if you manage to finish this chapter, this is a good start.

For reference, you can download the complete Xcode project from http://www.appcoda.com/resources/swift53/FoodPinOOP.zip. In the next chapter, base on what we've learned, you'll continue to tweak the detail view screen of the FoodPin app. It's going to be fun!

Further References

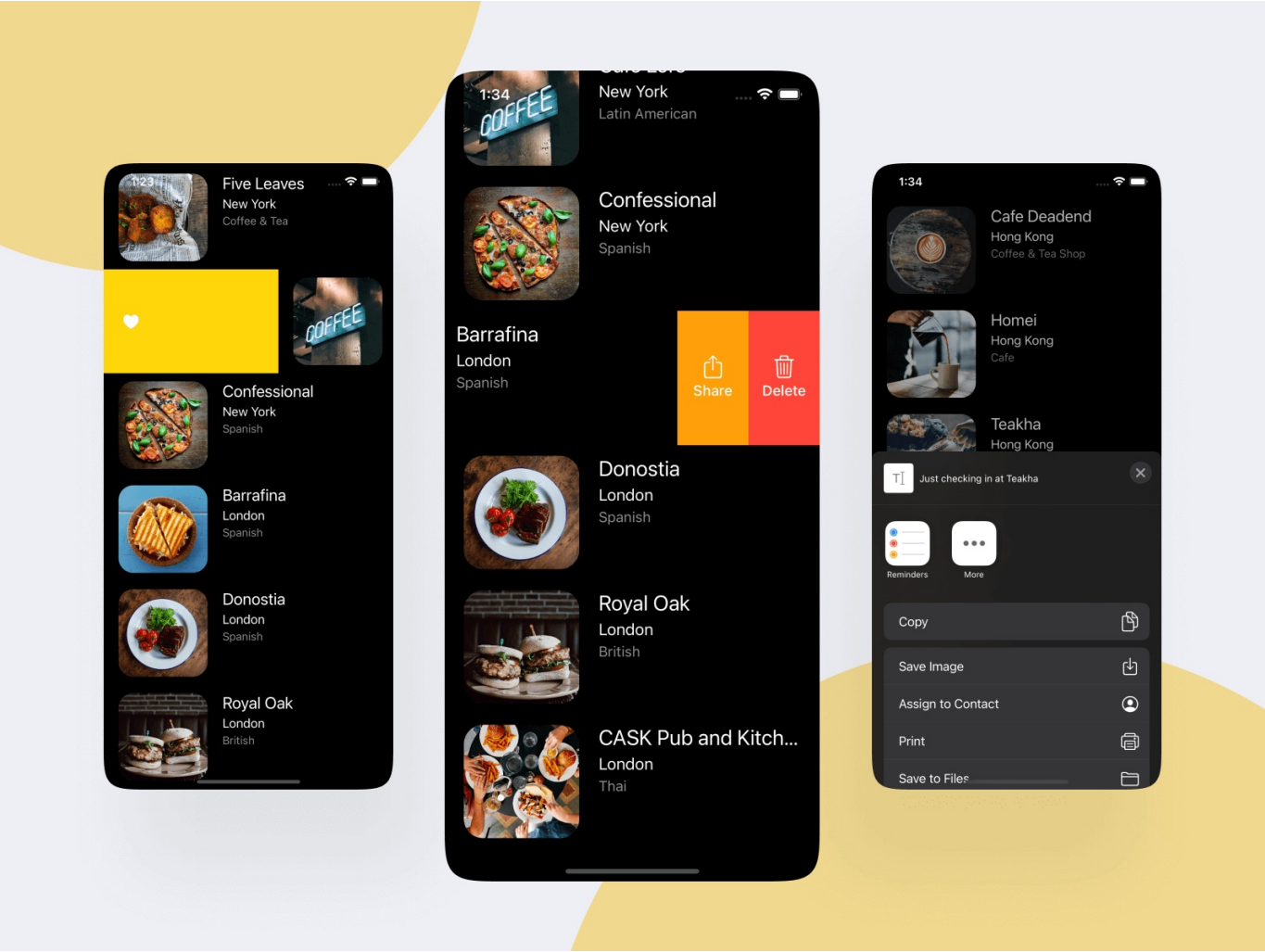
Swift Programming Language - Classes and Structures
https://docs.swift.org/swift-book/LanguageGuide/ClassesAndStructures.html

Swift Programming Language - Initialization https://docs.swift.org/swift-book/LanguageGuide/Initialization.html

Swift Programming Language - Inheritance https://docs.swift.org/swift-book/LanguageGuide/Inheritance.html

Object Oriented Programming from MIT Open Courseware https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-01sc-introduction-to-electrical-engineering-and-computer-science-i-spring-2011/unit-1-software-engineering/object-oriented-programming/

Chapter 12
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

In the earlier chapter, you learned 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 easily share the restaurants.

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

We briefly mentioned the term MVC in the previous chapter when organizing the source files. 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. Similar to OOP, if your ultimate goal is to become a professional developer, you can't avoid learning Model-View-Controller. 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.

Understanding Model-View-Controller

> At the heart of MVC, and the idea that was the most influential to later frameworks, is what I call Separated Presentation. The idea behind Separated Presentation is to make a clear division between domain objects that model our perception of the real world, and presentation objects that are the GUI elements we see on the screen. Domain objects should be completely self contained and work without reference to the presentation, they should also be able to support multiple presentations, possibly simultaneously. This approach was also an important part of the Unix culture, and continues today allowing many applications to be manipulated through both a graphical and command-line interface.
>
> By Martin Fowler

No matter which programming language you learn, one important concept that you need to know is *Separation of Concerns (SoC)*. The concept is pretty simple. Here, the *Concerns* are different aspects of software functionality. This concept encourages developers to break a complicated feature or program into several areas of concern such that each area has its own responsibility. The delegate pattern, that we explained in the earlier chapters, is one of the examples of SoC.

The *model-view-controller (MVC)* concept is another example of SoC. The core idea behind MVC is to separate a user interface into three areas (or groups of objects) that each area is responsible for a particular functionality. As the name suggests, MVC breaks a user interface into three parts:

- **Model** – a model is responsible for holding the data or any operations on the data. The model can be as simple as an array object that stores the table data. Add, update and delete are examples of the operations. In the business world, these operations are usually known as business rules.
- **View** – a view manages the visual display of information. For example, UITableView displays data in a list format.
- **Controller** – a controller is a bridge between the model and the view. It

translates the user interaction from the view (e.g. tap) into the appropriate action to be performed in the model. For example, a user taps the delete button in the view. Consequently, the controller triggers a delete operation in the model. Once finished, the model requests the view to refresh itself so as to reflect the update of the data model.

To help you better understand MVC, let's use the FoodPin app as an example. The app displays a list of restaurants in the table view. If you turn the implementation into a visual illustration, here is how the table data is displayed:

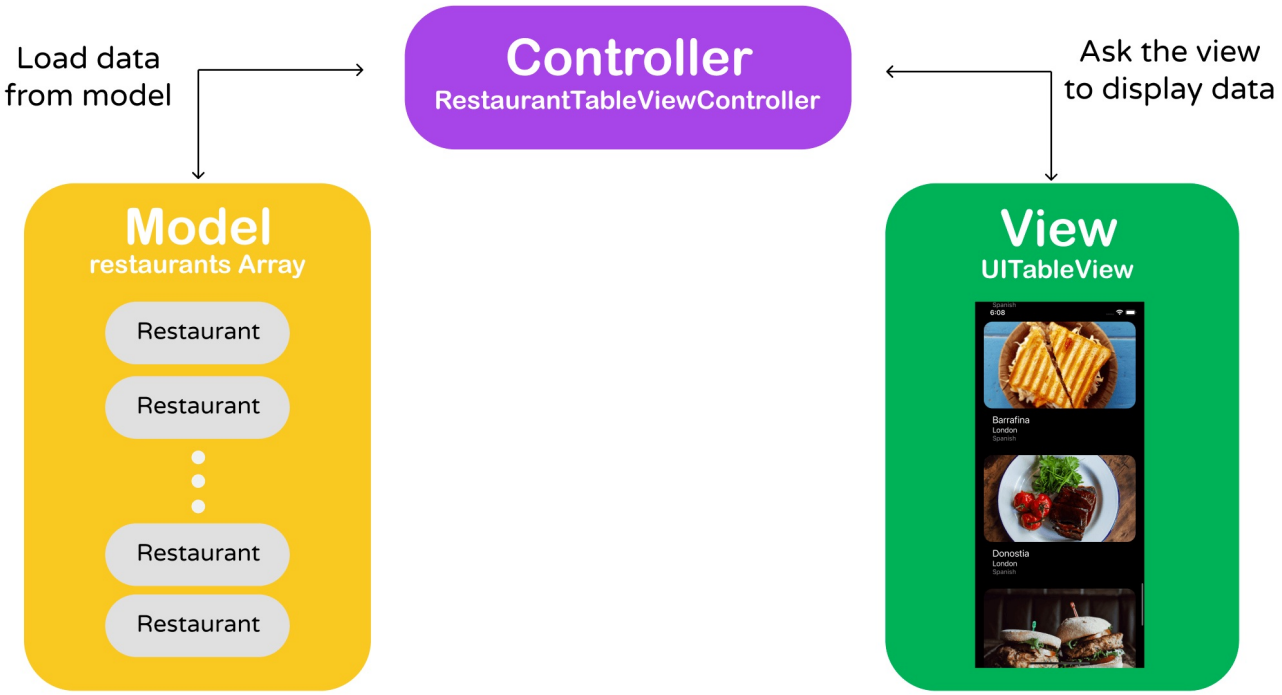


Figure 12-1. MVC Illustrated

The *Model* is the `restaurants` array which holds the list of restaurants. Each table row maps to an item of the `restaurants` array. The `UITableView` object is the actual *View* to be seen by a user. It's responsible for all the visuals (e.g. color of the table rows, style of the tableview, separator style, etc). The `UITableViewController`

object, which is the *Controller*, acts as the bridge between the table view and the data model. It manages the table view and is responsible to load the data from the model.

Deleting a Row from UITableView

I hope you now have a better understanding of Model-View-Controller. Let's move onto the coding part and see how we can delete a row from a table view. We'll continue to develop the FoodPin app (if you haven't completed the previous exercise, you can start with by downloading the project from (http://www.appcoda.com/resources/swift53/FoodPinOOP.zip) and add the "delete" feature.

If you understand the MVC model, you probably have some ideas on the implementation of row deletion. There are three main tasks we have to do:

1. Enable the swipe-to-delete feature of the table view so that the user can select the Delete option
2. Delete the corresponding table data from the data model
3. Update the snapshot and apply the change of table data

Enable the Swipe-to-delete feature

In iOS apps, users normally swipe horizontally across a table row to reveal the Delete button. If you've read the documentation of the `UITableViewDataSource` protocol, you will find a method called `tableView(_:commit:forRowAt:)`. This is the method that handles the deletion (or insertion) of a specific row.

The question is how can we implement the method? We are not using `UITableViewDataSource` but `UITableViewDiffableDataSource` to manage our data. If you dig deeper into the API documentation of `UITableViewDiffableDataSource`, this

protocol actually conforms to `UITableViewDataSource`! In this case, we can override `tableView(_:commit:forRowAt:)` and create the swipe-to-delete feature.

But wait! Referring to the `configureDataSource()` method, we are using the `UITableViewDiffableDataSource` class directly. How can we override the method and provide our own implementation?

Things get a little bit tricky here. In order to override the default implementation of the method, we have to subclass `UITableViewDiffableDataSource` and create our own version of diffable data source class.

Now go to the project navigator. Right click the *Model* folder and choose *New file...*. Select the *Cocoa Touch Class* template and click *Next*. Name the class `RestaurantDiffableDataSource` and set its subclass to `UITableViewDiffableDataSource`.

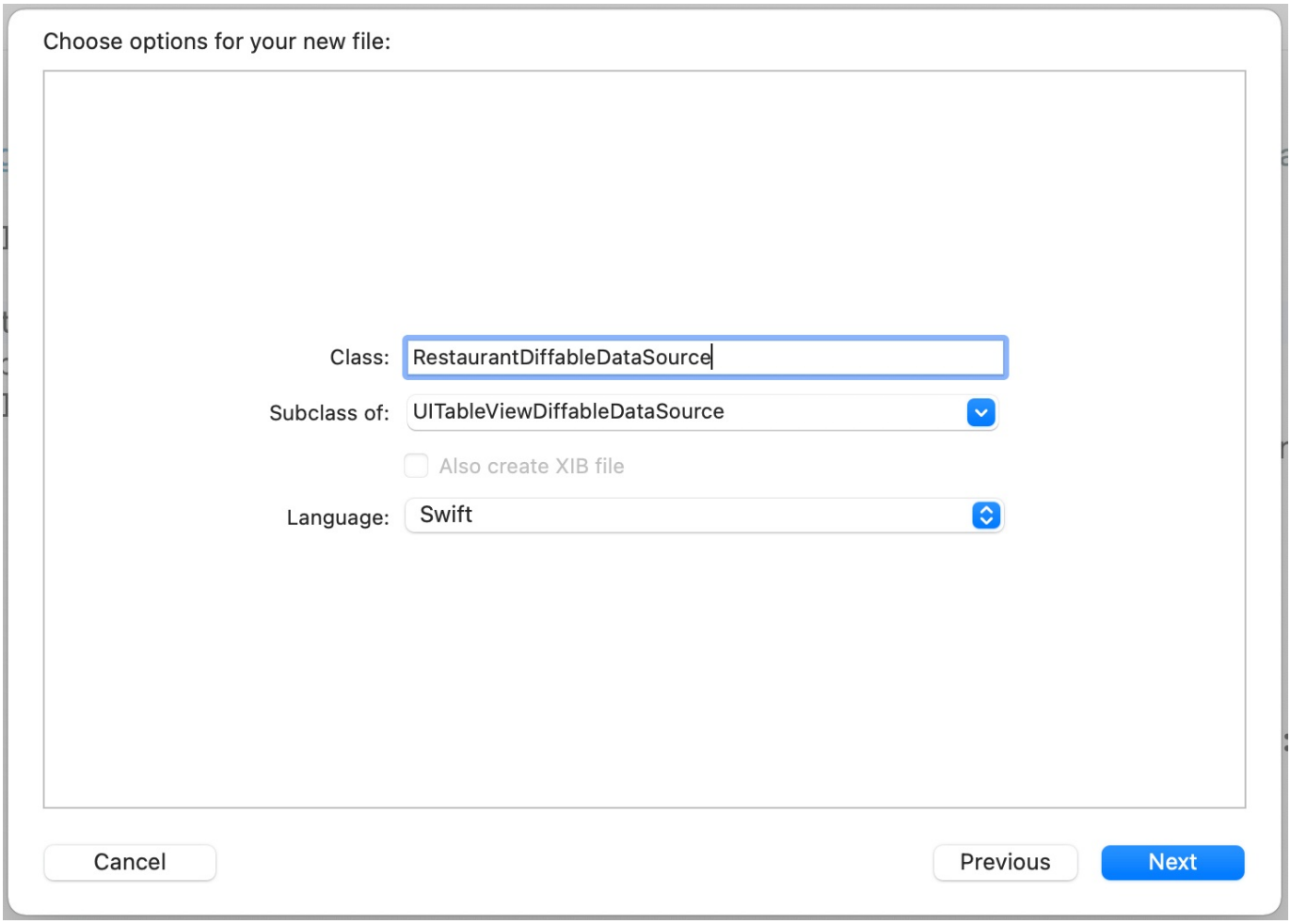


Figure 12-2. Creating a new class

Once the file is created, update the content like this:

```
import UIKit

enum Section {
    case all
}

class RestaurantDiffableDataSource: UITableViewDiffableDataSource<Section,
Restaurant> {

    override func tableView(_ tableView: UITableView, canEditRowAt indexPa
th: IndexPath) -> Bool {
        return true
    }

    override func tableView(_ tableView: UITableView, commit editingStyle:
 UITableViewCell.EditingStyle, forRowAt indexPath: IndexPath) {

    }
}
```

We created our custom version of diffable data source with the section type `Section` and the item type `Restaurant`. In the body, we override the `tableView(_:commit:forRowAt:)` method and the `tableView(_:canEditRowAt:)` method. The reason why we need to override the default implementation of `tableView(_:canEditRowAt:)` is that the cell is non-editable by default. This is why we override the method and return the value `true` to make the cells editable. The swipe-to-delete feature will not work if the cell is non-editable.

While we override the `tableView(_:commit:forRowAt:)` method, we haven't provided any implementation. This is good enough to enable the swipe-to-delete feature of a table view. If the method exists, the table view will automatically display a *Delete* button when the user swipes across a row.

Before we can test the app, there are a couple of changes we have to make. Since we have our custom version of `UITableViewDiffableDataSource`, switch over to `RestaurantTableViewController.swift` and change the `UITableViewDiffableDataSource` in the `configureDataSource()` method to `RestaurantDiffableDataSource`. The method should be like this after change:

```
func configureDataSource() -> UITableViewDiffableDataSource<Section, Resta
urant> {

    let cellIdentifier = "datacell"

    let dataSource = RestaurantDiffableDataSource(
        tableView: tableView,
        cellProvider: {  tableView, indexPath, restaurant in
            let cell = tableView.dequeueReusableCell(withIdentifier: cellI
dentifier, for: indexPath) as! RestaurantTableViewCell

            cell.nameLabel.text = restaurant.name
            cell.locationLabel.text = restaurant.location
            cell.typeLabel.text = restaurant.type
            cell.thumbnailImageView.image = UIImage(named: restaurant.imag
e)
            cell.favoriteImageView.isHidden = restaurant.isFavorite ? false
 : true

            return cell
        }
    )

    return dataSource
}
```

Please also note that I changed the value of `cellIdentifier` to `datacell`. You don't have to follow me. If you like, you can keep it to `favoritecell`.

There is one more change we have to make. Since we moved the `Section` enum to `RestaurantDiffableDataSource.swift`. We can delete the one declared in `RestaurantTableViewController`:

```
enum Section {
    case all
}
```

Now let's have a quick test. Run the app on an iPhone simulator. Though the method is currently without any implementation, you should see the *Delete* button when swiping across a row. If the delete button is functional, you can either tap the *Delete* button or continue to swipe to the left edge to remove the item.

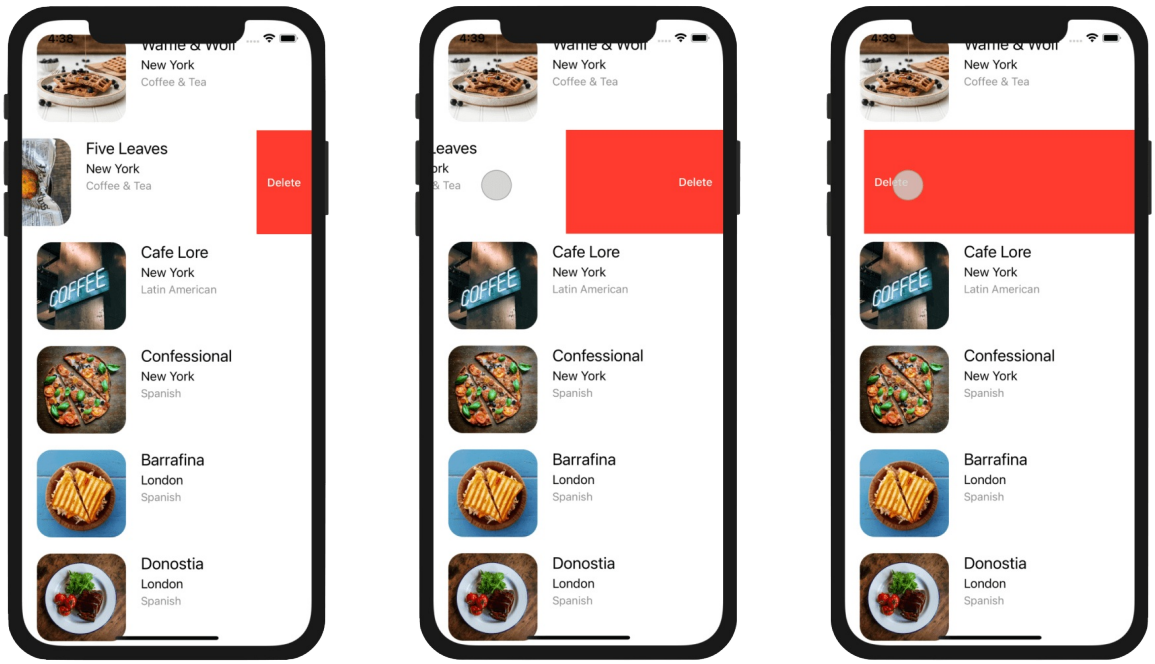


Figure 12-3. Swipe-to-delete feature

Delete Row Data from the Table View Using Snapshot

The next thing is to implement the method and write the code for removing the actual table data from the table view. From the method declaration, the `indexPath` parameter contains the row number of the cell, which is about to delete. You can make use of this information to retrieve the restaurant that we are going to delete.

Update the `tableView(_:commit:forRowAt:)` method of `RestaurantDiffableDataSource` like this:

```
override func tableView(_ tableView: UITableView, commit editingStyle: UIT
ableViewCell.EditingStyle, forRowAt indexPath: IndexPath) {

    if editingStyle == .delete {
        if let restaurant = self.itemIdentifier(for: indexPath) {
            var snapshot = self.snapshot()
            snapshot.deleteItems([restaurant])
            self.apply(snapshot, animatingDifferences: true)
        }
    }
}
```

The method supports two types of editing styles: *insert* and *delete*. Because we are only interested in the delete operation, we check the value of `editingStyle` before executing the code block. To find out the selected restaurant, we call the `itemIdentifier` method with the current `indexPath` .

The method returns us an optional, which means it can return a `Restaurant` object or nil. This is why we use the `if let` statement to perform a validation. Once we have the selected restaurant, we call the `snapshot()` method to get a copy of the data source's current snapshot. Then we delete that restaurant by calling `deleteItems` . Lastly, we call `apply` with the updated snapshot to ask the table view to update its UI. Note that we also request the table view to animate the change by setting `animatingDifferences` to `true` .

Now run and test your app. Swipe any of the restaurants and delete the item. The table view should animate the deletion and remove the restaurant from the table.

If you've implemented the same "Delete" feature without diffable data sources, you should find that `UITableViewDiffableDataSource` make it very simple to keep the model data and view data in-sync. You no longer need to call `reloadData` to reload the table view or call `deleteRows()` to delete a specific table row. All you need is to grab a copy of the snapshot data, delete the item from the snapshot, and apply the changes to the table view.

Swipe for More Actions Using UIContextualAction

When you swipe left a table cell in the stock Mail app, you will see a Trash button, a Flag, and a More button. The More button will bring up an action sheet providing a list of options such as Reply, Flag, etc. If you swipe right, you will find the Archive button.

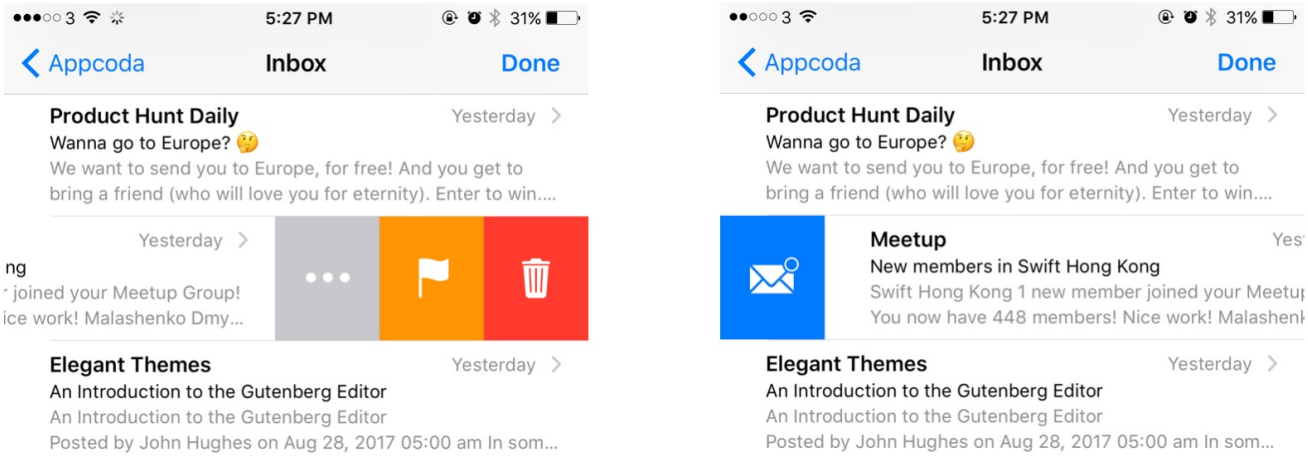


Figure 12-4. Swipe for more in the Mail app

This *swipe-for-more* feature was first introduced in the stock iPhone apps in iOS 7. From iOS 8 to iOS 10, Apple made the "swipe-left" feature available to developers via a class named `UITableViewRowAction`. In iOS 11 (or up), Apple even lets developers implement the "swipe-right" feature in our own apps through new classes like `UISwipeActionsConfiguration` and `UIContextualAction`. You use the `UIContextualAction` class to create custom actions for table rows of any table view.

If we just need to implement the swipe to delete function in our app, the code that we implemented in the previous section is good enough. However, if we want to have a swipe-for-more feature, we need something more than that.

Before moving on, remove the whole method of `tableView(_:commit:forRowAt:)` in `RestaurantDiffableDataSource`:

```
override func tableView(_ tableView: UITableView, commit editingStyle: UIT
ableViewCell.EditingStyle, forRowAt indexPath: IndexPath) {
    .
    .
    .
}
```

We no longer need the method. Instead, we will implement something new to replace it. iOS provided two methods for handling swipe actions in table view cells:

- tableView(_:leadingSwipeActionsConfigurationForRowAt:)
- tableView(_:trailingSwipeActionsConfigurationForRowAt:)

Both methods are defined in `UITableViewDelegate`. The first method is designed for handling swipe-to-right actions, while the second one is for swipe-to-left actions.

Now let's implement the second method in order to re-create the Delete function, plus adding a Share button. Insert the following method in `RestaurantTableViewController`:

```
override func tableView(_ tableView: UITableView, trailingSwipeActionsConf
igurationForRowAt indexPath: IndexPath) -> UISwipeActionsConfiguration? {

    // Get the selected restaurant
    guard let restaurant = self.dataSource.itemIdentifier(for: indexPath)
else {
        return UISwipeActionsConfiguration()
    }

    // Delete action
    let deleteAction = UIContextualAction(style: .destructive, title: "Del
ete") { (action, sourceView, completionHandler) in

        var snapshot = self.dataSource.snapshot()
        snapshot.deleteItems([restaurant])
        self.dataSource.apply(snapshot, animatingDifferences: true)

        // Call completion handler to dismiss the action button
        completionHandler(true)
    }

    // Share action
    let shareAction = UIContextualAction(style: .normal, title: "Share") {
 (action, sourceView, completionHandler) in
        let defaultText = "Just checking in at " + restaurant.name
        let activityController = UIActivityViewController(activityItems: [
defaultText], applicationActivities: nil)

        self.present(activityController, animated: true, completion: nil)
        completionHandler(true)

    }

    // Configure both actions as swipe action
    let swipeConfiguration = UISwipeActionsConfiguration(actions: [deleteA
ction, shareAction])

    return swipeConfiguration
}
```

The usage of `UIContextualAction` is very similar to that of `UIAlertAction`. You specify the title, the style and the block of code (i.e. the completion handler) to execute when a user taps the button. In this example, we have two contextual actions: one for *Delete* and another for *Share*.

The first line of code may be new to you. We use `guard let` instead of `if let` to check if we can get a valid restaurant. The `guard` statement works like this:

- If the returned value of `itemIdentifier(for: indexPath)` is not nil, save the `Restaurant` object to the `restaurant` constant.
- Otherwise, return an empty swipe action configuration.

Can you use `if let` instead of `guard let` to perform the checking? Absolutely! However, the code looks cleaner when using `guard let` in this scenario.

To create the delete action, we instantiate a `UIContextualAction` object with the title "Delete" and `.destructive` style, indicating that the button should be in red. We also provide the code block to run after the delete action is selected. The code in the code block is similar to the one we wrote earlier, except that it has an additional line of code that calls `completionHandler(true)`. After you successfully complete the action, you have to call the method to dismiss the action button. `true` indicates that the delete action has been performed.

Next, we create the share action using `UIContextualAction` with style `.normal` and title "Share". If the style is set to `.destructive`, the button's background color will be set to gray by default. When a user taps the button, it brings up an activity controller for social sharing.

The `UIActivityViewController` class is a standard view controller that provides several standard services, such as copying items to the clipboard, sharing content on social media sites, sending items via Messages, etc. The class is very simple to

use. Let's say you have a message for sharing. All you need to do is create an instance of `UIActivityViewController` with the message object, and then present the controller on the screen. That is what we have done in the above code snippet.

The last two lines of code is the most important part. It returns a `UISwipeActionsConfiguration` object with an array of `UIContextualAction` objects (i.e. `deleteAction` and `shareAction`), telling the table view to create the swipe-for-more buttons.

Compile and run the app. Swipe across a table row and it'll show you both Share and Delete buttons. Tapping the Share button will bring up a share menu like the one shown in figure 11-5.

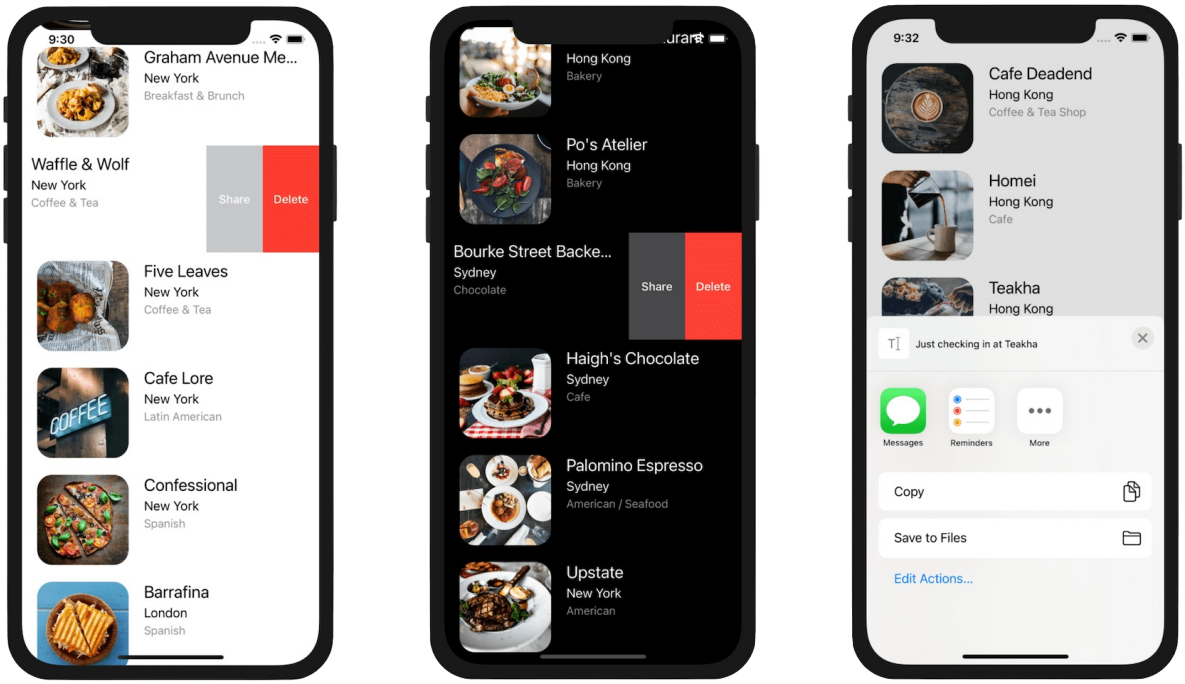


Figure 12-5. Swipe to Share button on the simulator

In the simulator, you can only find a few options such as copying items to the pasteboard and save to files. If the app is run on a real device, the share menu will offer more applications. Apple allows developers to define custom services that appear in the share menu. The available items vary from device to device, depending on the apps installed.

The `UIActivityViewController` class does not limit you from sharing content in text format. If you pass it a `UIImage` object during initialization, your app will allow users to copy images or share them to other applications. Modify the `shareAction` constant to the following:

```
let shareAction = UIContextualAction(style: .normal, title: "Share") { (ac
tion, sourceView, completionHandler) in
    let defaultText = "Just checking in at " + restaurant.name

    let activityController: UIActivityViewController

    if let imageToShare = UIImage(named: restaurant.image) {
        activityController = UIActivityViewController(activityItems: [defa
ultText, imageToShare], applicationActivities: nil)
    } else  {
        activityController = UIActivityViewController(activityItems: [defa
ultText], applicationActivities: nil)
    }

    self.present(activityController, animated: true, completion: nil)
    completionHandler(true)
}
```

In the code above, we added an `imageToShare` object for image sharing. When loading an image, it is possible that the image is failed to load. This is why the `UIImage` class returns an optional during initialization. Again, we use `if let` to

verify if an optional contains a value or not. If the image can be loaded properly, we pass both `defaultText` and `imageToShare` to `UIActivityViewController` during initialization.

```
activityController = UIActivityViewController(activityItems: [defaultText,
 imageToShare], applicationActivities: nil)
```

`UIActivityViewController` will automatically embed the image of the selected restaurant when the user chooses to copy the content to the pasteboard. You can try it out by selecting *Copy* and then paste it into the built-in Messages app.

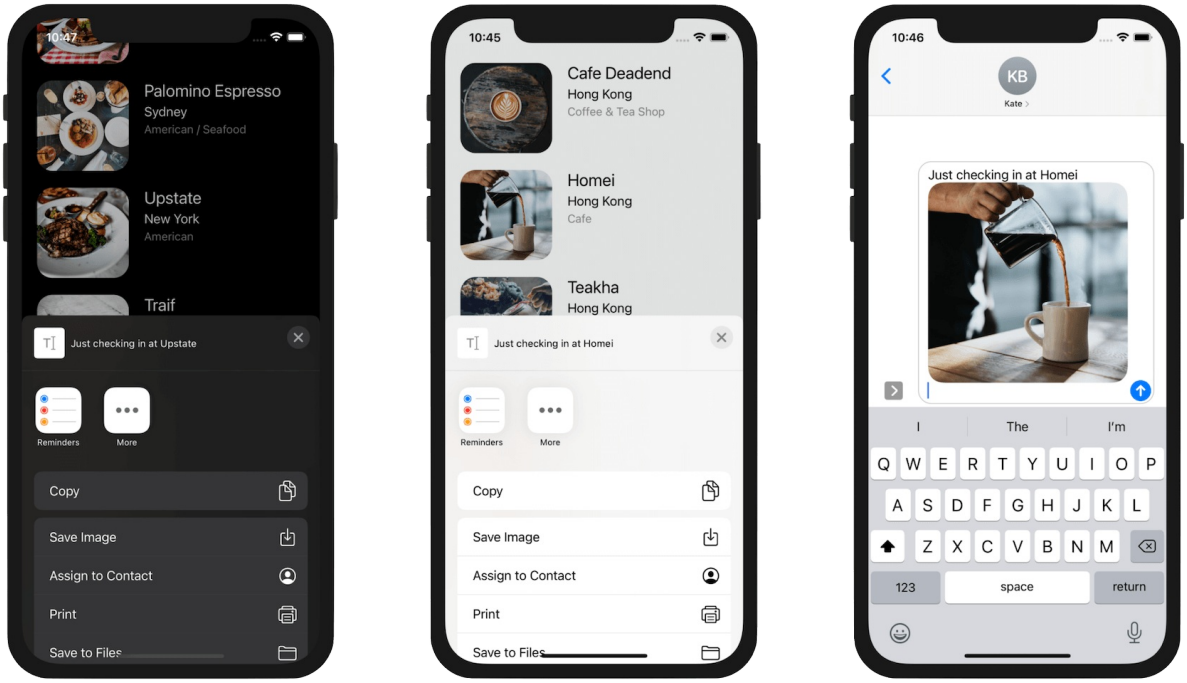


Figure 12-6. Copy the selected restaurant and paste it into the Messages app

Understanding SF Symbols

> With over 2,400 configurable symbols, SF Symbols is designed to integrate seamlessly with San Francisco, the system font for Apple platforms. Each symbol comes in a wide range of weights and scales that automatically align with text labels, and supports Dynamic Type and the Bold Text accessibility feature. You can also export symbols and edit them in vector graphics editing tools to create custom symbols with shared design characteristics and accessibility features.

Before I showed you how to customize the contextual action with icons, let's first talk about where the icons come from. Needless to say, you can provide your own images to use in the app. In iOS 13, Apple introduced a large set of system images called SF Symbols that allows developers to use them in any apps. Along with the release of Xcode 12, Apple released SF Symbols 2 with more configurable symbols and multicolor support.

These images are referred as symbols since it's integrated with the built-in San Francisco font. To use these symbols, no extra installation is required. As long as your app is deployed to a device running iOS 13 (or later), you can access these symbols directly.

To use the symbols, all you need to prepare is find out the name of the symbol. With over 2,400 symbols available for your use, Apple has released an app called SF Symbols 2 (https://devimages-cdn.apple.com/design/resources/download/SF-Symbols-2.dmg), so that you can easily explore the symbols and locate the one that fits your need. I highly recommend you to install the app before proceeding to the next section.

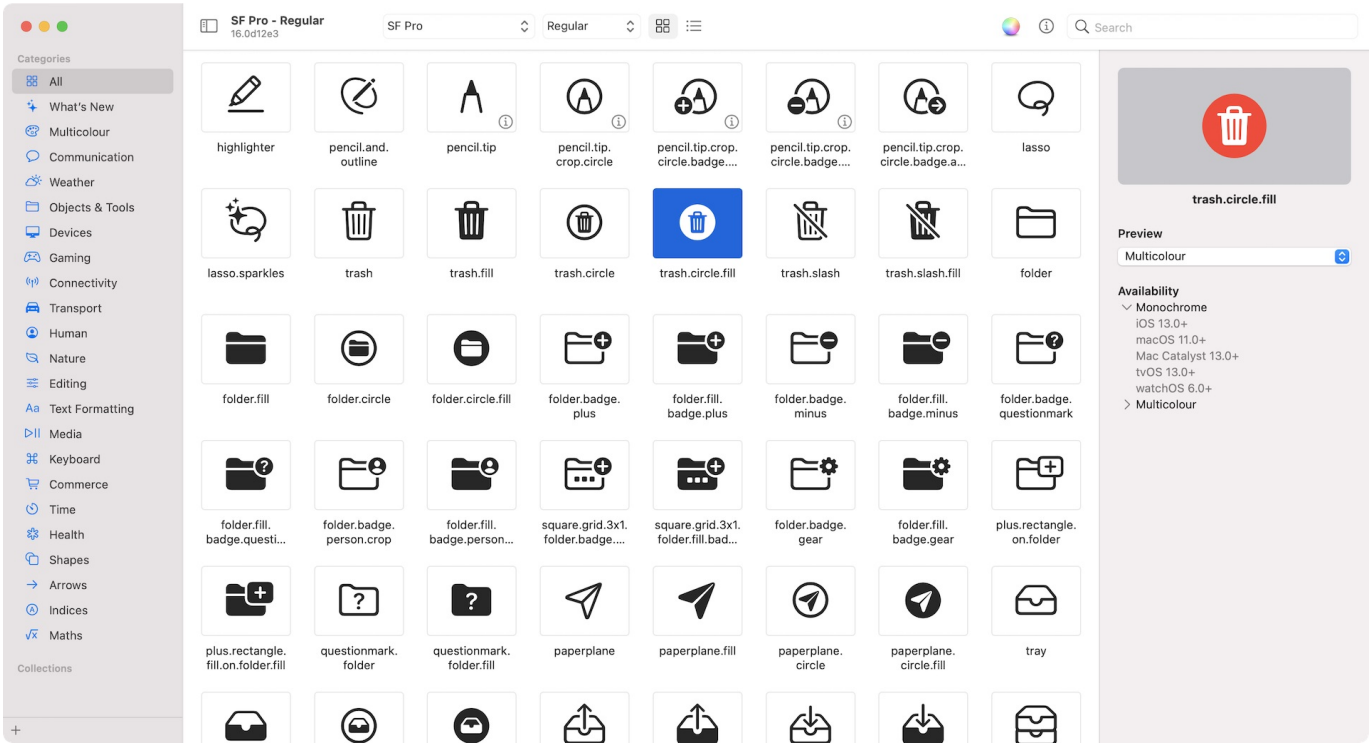


Figure 12-7. SF Symbols App

Customizing UIContextualAction

Okay, let's see how to customize the action buttons. By default, the action buttons are either in red or gray. The `UIContextualAction` class provides a couple of the properties for developers to customize its style including background color and icon. The code looks like this:

```
shareAction.backgroundColor = UIColor.systemOrange
shareAction.image = UIImage(systemName: "square.and.arrow.up")
```

You set the `backgroundColor` property to change its background color and the `image` property to add an icon. To use the built-in images provided by SF Symbols, you need to instantiate an `UIImage` and provide the image name in the `systemName` parameter.

This is the first time we work with color programmatically. The UIKit framework provides a `UIColor` class to represent color. Many methods in UIKit require you to provide color using a `UIColor` object. iOS offers a wide range of system colors that you can use directly.

Alternatively, if you want to use your own color, you can create a `UIColor` object using RGB component values. The component values should be within 0 and 1. If you're a web designer or have some experience with graphics design, you know that the RGB values are on a scale of 0 to 255. To conform with the requirement of `UIColor`, you have to divide each of the component values by 255 when creating a `UIColor` object. Here is an example:

```
shareAction.backgroundColor = UIColor(red: 254.0/255.0, green: 149.0/255.0
, blue: 38.0/255.0, alpha: 1.0)
```

Apple encourages us to use the built-in system color because it supports both light and dark modes. And, the colors automatically adapt to vibrancy and changes in accessibility settings like Increase Contrast and Reduce Transparency. Therefore, we will use the system color in our code change. You can refer to this link (https://developer.apple.com/design/human-interface-guidelines/ios/visual-design/color/) to check out all the supported system colors.

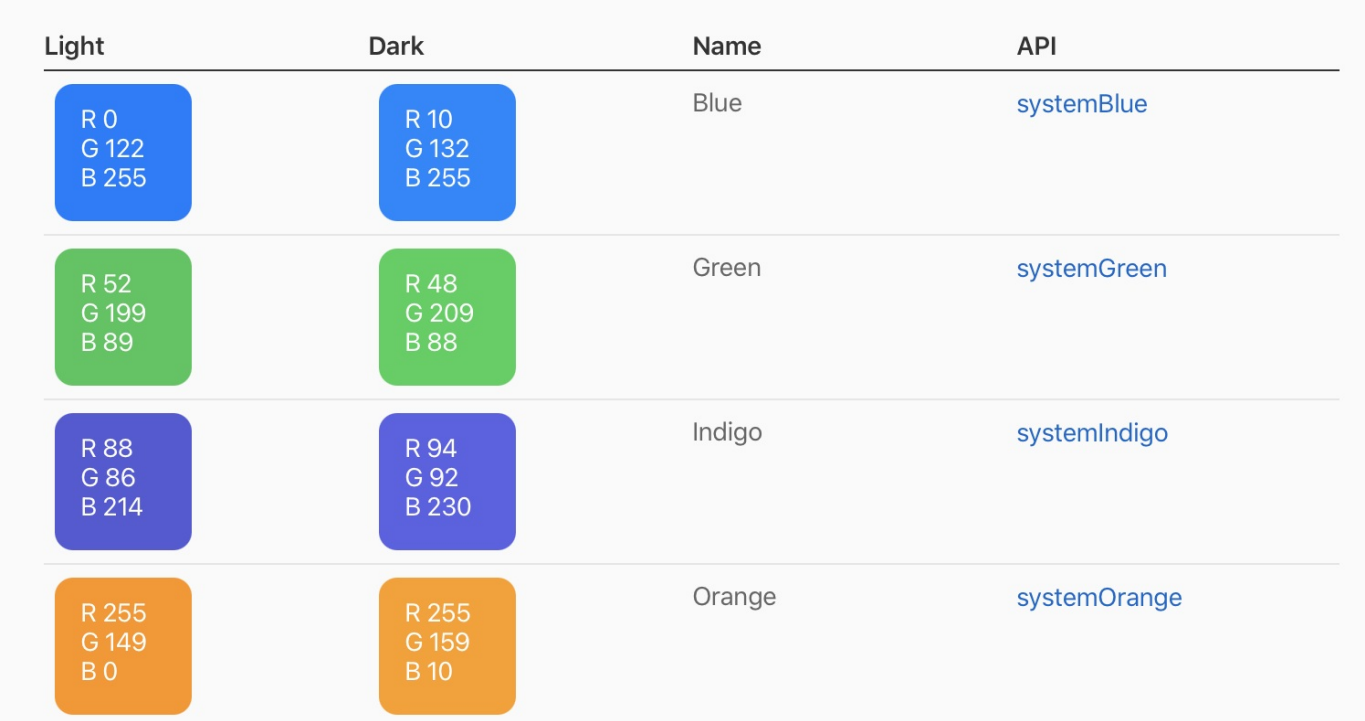


Figure 12-8. Some system colors for both light and dark mode

Now, insert the following lines of code in the `tableView(_:trailingSwipeActionsConfigurationForRowAt:)` method and place it before the instantiation of `swipeConfiguration`:

```
deleteAction.backgroundColor = UIColor.systemRed
deleteAction.image = UIImage(systemName: "trash")

shareAction.backgroundColor = UIColor.systemOrange
shareAction.image = UIImage(systemName: "square.and.arrow.up")
```

After you made the changes, test the app again and see if you like the new color. Otherwise, modify the system color and change to your preferred one.

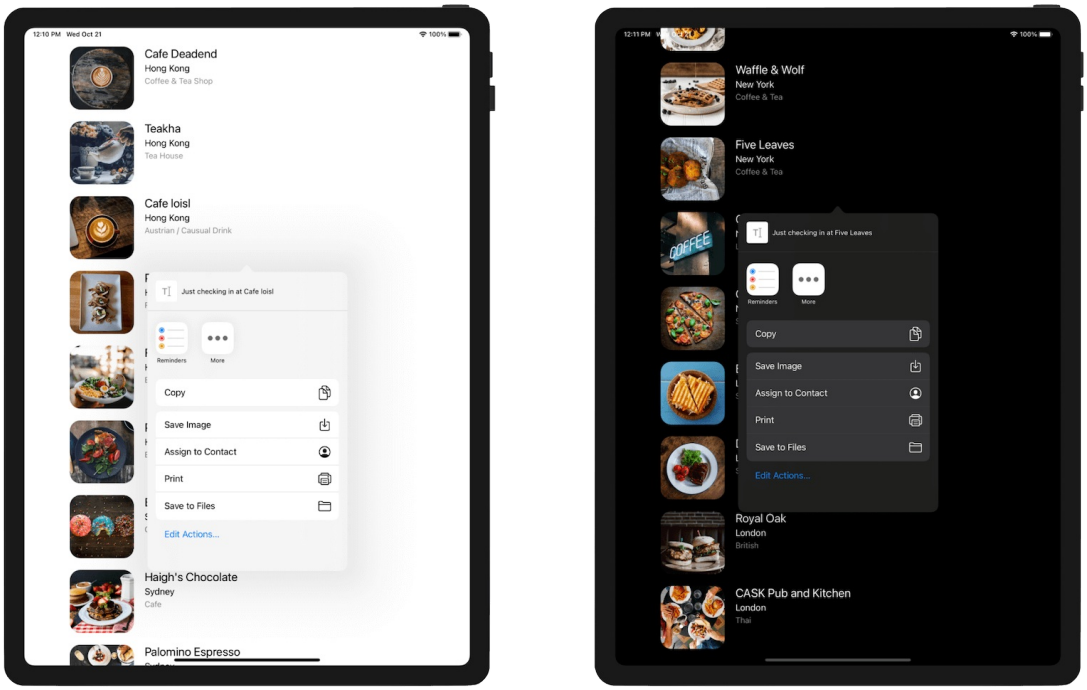


Figure 12-9. Customizing the action buttons with new background color and icons

Quick tip: Like me, if you are not a designer, you may need some color inspiration. You can check out Adobe Color CC (color.adobe.com) and Flat UI Color Picker (flatuicolorpicker.com). You'll find a lot of color combinations for designing your app.

Testing on iPad

Did you try to run the app on iPad? The app crashes when you tap the *Share* button and the following error shows up in console:

```
*** Terminating app due to uncaught exception 'NSGenericException', reason
: 'UIPopoverPresentationController (<UIPopoverPresentationController: 0x7f
adaa51f7e0>) should have a non-nil sourceView or barButtonItem set before
the presentation occurs.'
terminating with uncaught exception of type NSException
CoreSimulator 732.17 - Device: iPad Pro (11-inch) (2nd generation) (02AA48
3A-0CAA-4645-8B8F-90D3397769A3) - Runtime: iOS 14.1 (18A8394) - DeviceType
: iPad Pro (11-inch) (2nd generation)
```

> When presenting the view controller, you must do so using the appropriate means for the current device. On iPad, you must present the view controller in a popover. On iPhone and iPod touch, you must present it modally.
>
> https://developer.apple.com/documentation/uikit/uiactivityviewcontroller

Similar to what we have discussed in the previous chapter, `UIActivityViewController` will be presented in the popover style on large devices. I believe you know how to fix the bug.

> You must provide location information for this popover through the view controller's popoverPresentationController. You must provide either a sourceView and sourceRect or a barButtonItem.

When presenting the activity view controller using the popover style, you have to configure the source view of its `popoverPresentationController`. The source view indicates the view containing the anchor rectangle for the popover. You will understand what it means after implementing the popover.

Now, insert the following code before calling the `self.present(activityController, animated: true, completion: nil)` method in `shareAction`:

```
if let popoverController = activityController.popoverPresentationControlle
r {
    if let cell = tableView.cellForRow(at: indexPath) {
        popoverController.sourceView = cell
        popoverController.sourceRect = cell.bounds
    }
}
```

Test the app on the iPad simulator again. It should work now.

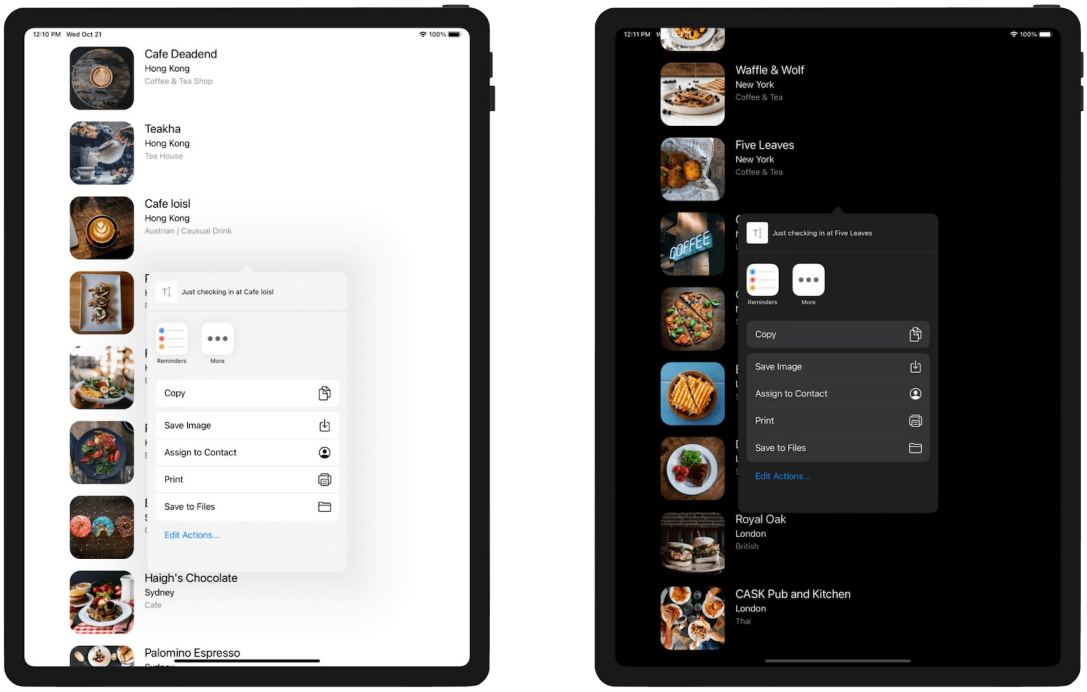


Figure 11-9. The share action shows a popover on iPad

Exercise

Up till now, we only implement the swipe left action. How about swipe right? I will leave you to implement the feature as an exercise. As mentioned before, all you need is to provide the implementation of the following method:

```
override func tableView(_ tableView: UITableView, leadingSwipeActionsConfi
gurationForRowAt indexPath: IndexPath) -> UISwipeActionsConfiguration? {

}
```

What I need you to do is add a check-in action when the user swipes right the cell. In the SF Symbols app, you should find the images named `heart.fill` and `heart.slash.fill`. You can use these images for the "Mark as favorite" action. After you finish the exercise, your app should allow users to swipe right to perform the "Mark as favorite" action that we covered before. When you swipe right on a cell with the heart icon, the tick icon will be replaced by the undo icon. Figure 12-10 illustrates the swipe-to-favorite action.

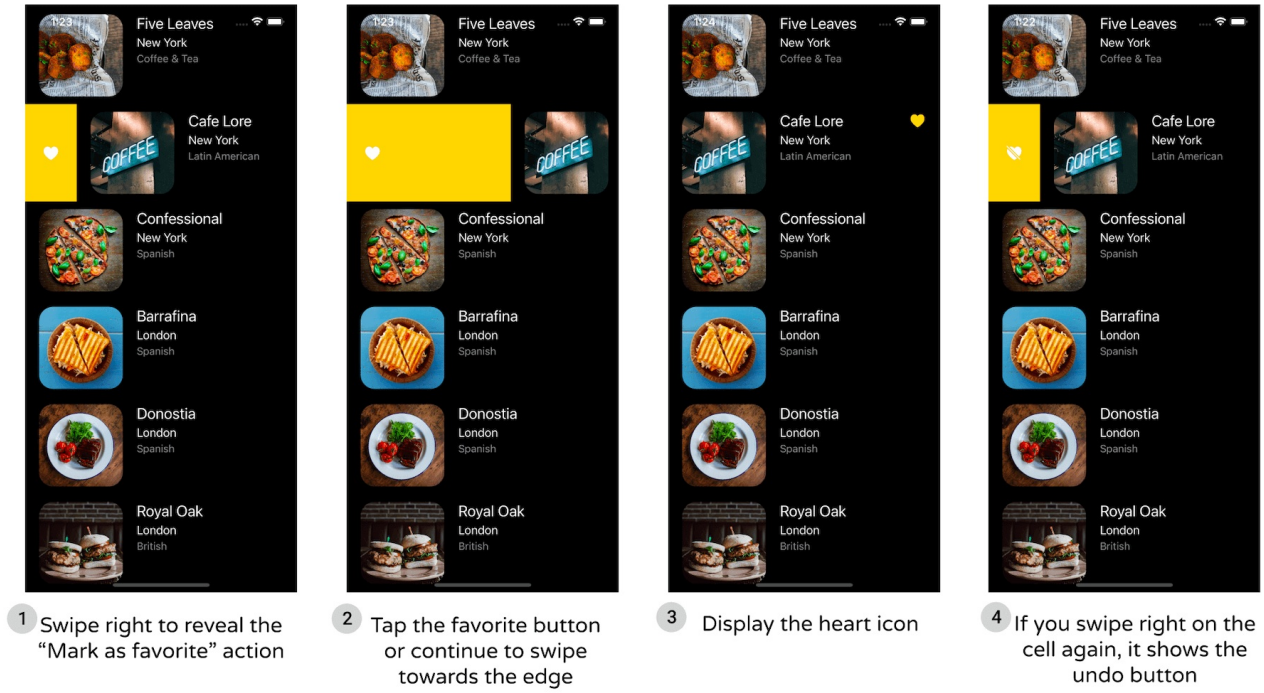


Figure 12-10. How the Swipe-to-favorite button works

This exercise is more challenging than those you worked on before. It helps you revise what you have learned and explore something new. Don't miss this opportunity and take some time to figure out the solution. If you have any questions about the exercise, discuss it in our private Facebook group.

Summary

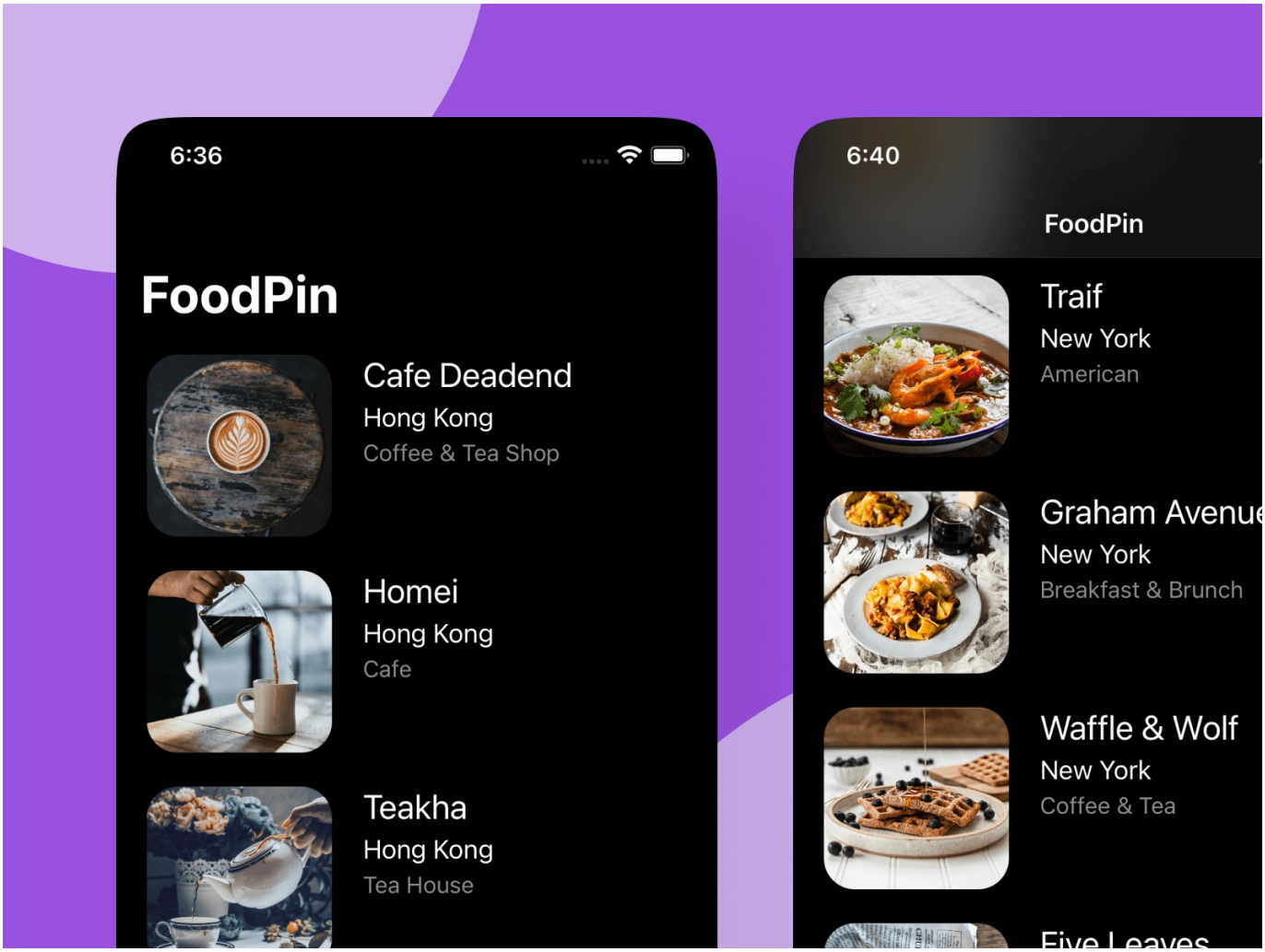
In this chapter, I gave you a brief overview of MVC, showed you how to manage deletion in a table view, and taught you how to create swipe action in table view cells using `UIContextualAction`. You also learned how to use `UIActivityViewController` for implementing a sharing feature. The FoodPin app is getting better and better. You should be proud of your achievement so far.

The MVC concept is important. If you're a programming newbie, it may take you some time to fully understand the material. Feeling confused? Just take a break and grab yourself a coffee. After your break, go over the chapter again. Probably you'll find it easier to digest.

For reference, you can download the complete Xcode project (including the solution to the exercise) from http://www.appcoda.com/resources/swift53/FoodPinTableDeletion.zip.

In the next chapter, we'll take a look at something new and create a navigation controller.

Chapter 13
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.

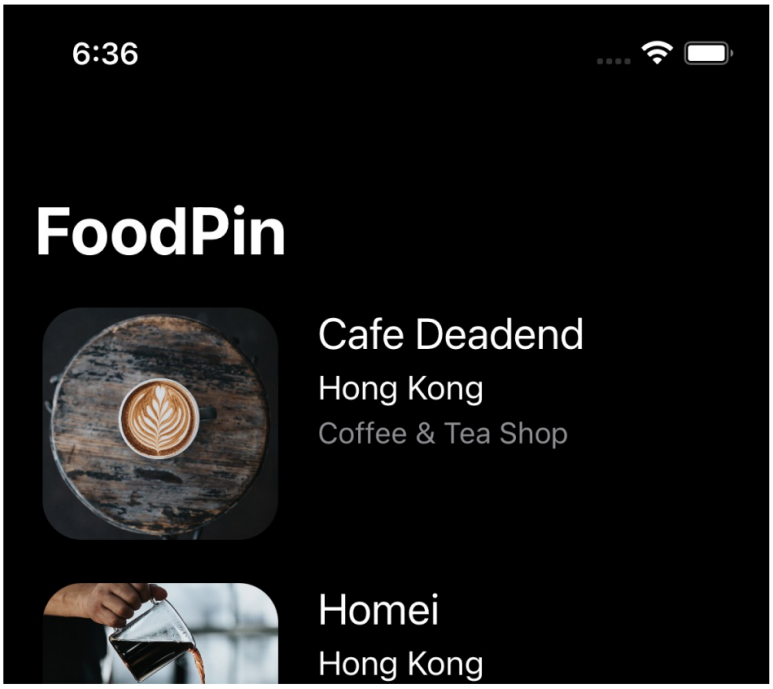


Figure 13-1. FoodPin app with a navigation bar in dark mode

If you've closed the FoodPin project (http://www.appcoda.com/resources/swift53/FoodPinTableDeletion.zip), it is time to fire up Xcode and open the project again. Select `Main.storyboard` to switch to the Interface Builder editor. Xcode provides an *embed* feature that makes it easy to embed any view controller in a navigation controller. Select the table view controller and click *Editor* in the menu. Choose *Embed in > Navigation Controller*.

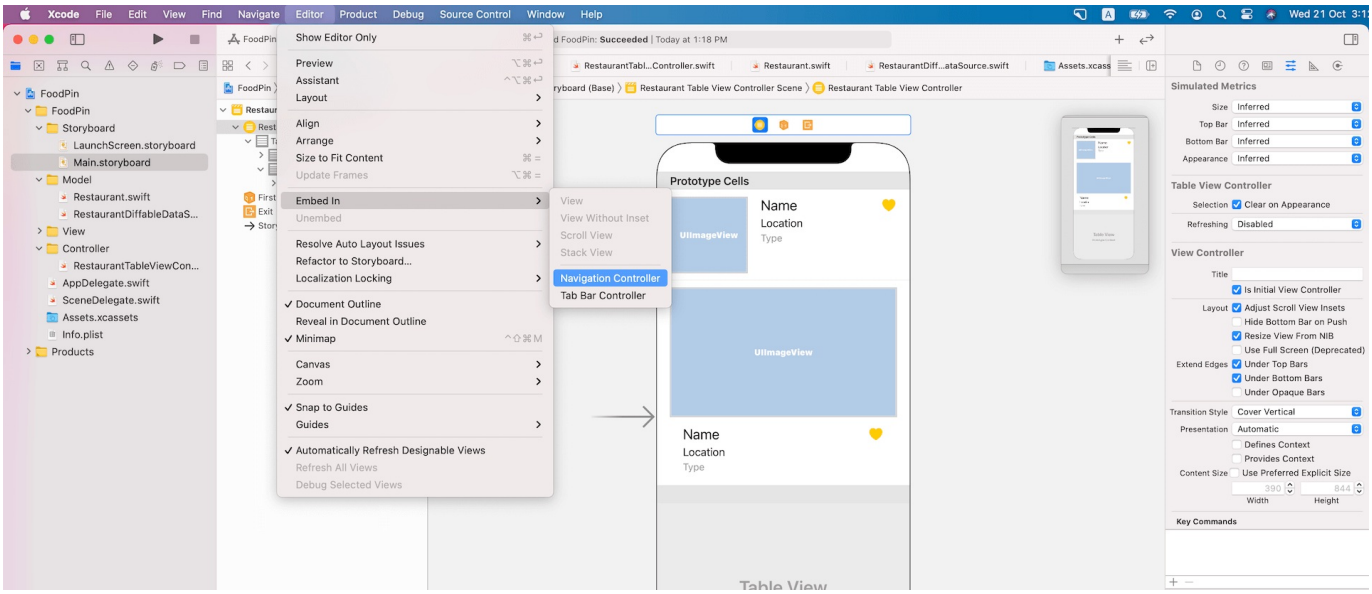


Figure 13-2. Embed in option in Xcode menu

Xcode automatically embeds the controller in a navigation controller. Let's set the title of the navigation bar. Select the navigation bar of the table view controller. Under the Attributes inspector, change the value of the title to *FoodPin*.

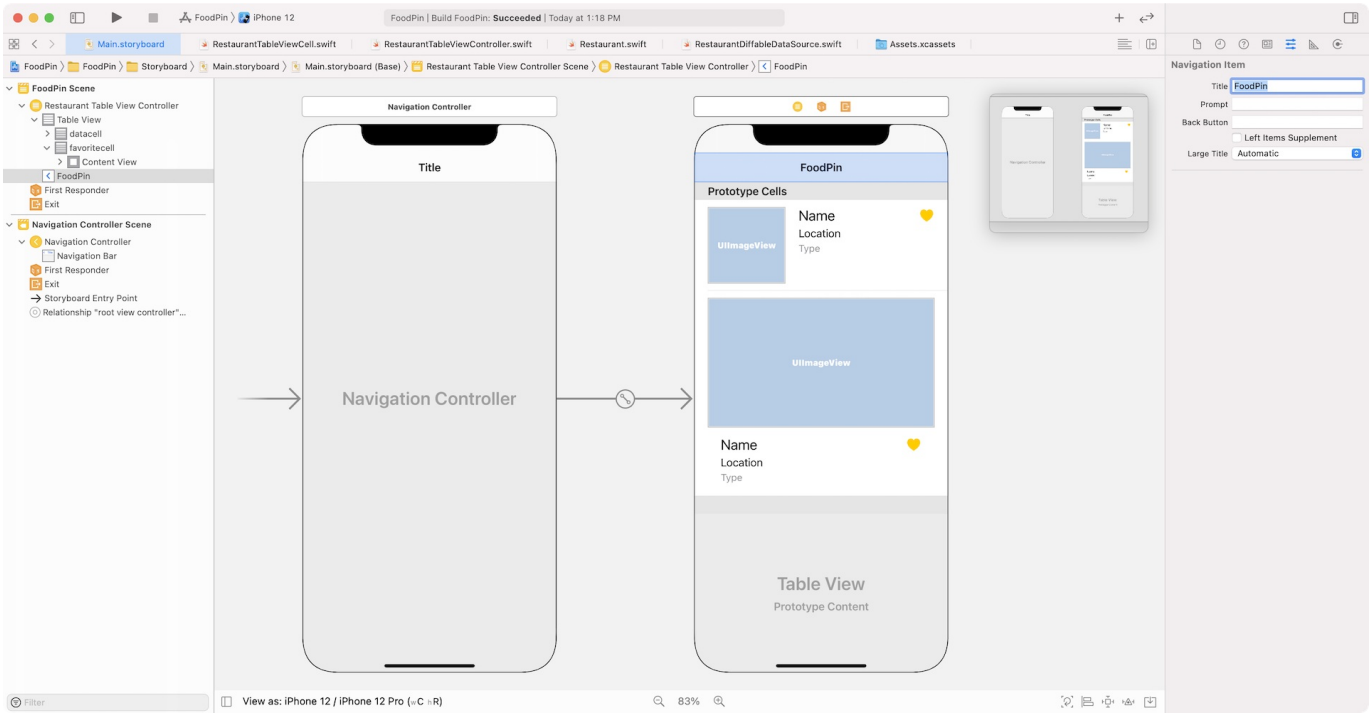


Figure 13-3. Embedding the table view controller in a navigation controller

Now run the app and see how it looks. The app is the same as before, but added with a navigation bar.

Large Navigation Bar Titles

Starting from version 11, iOS comes with large navigation bar titles. The stock apps, such as Mail and Settings, use large titles in the navigation bar. The size of the bar is not fixed, however. When you scroll the content, the navigation bar shrinks back down its size to that appears in iOS 10.

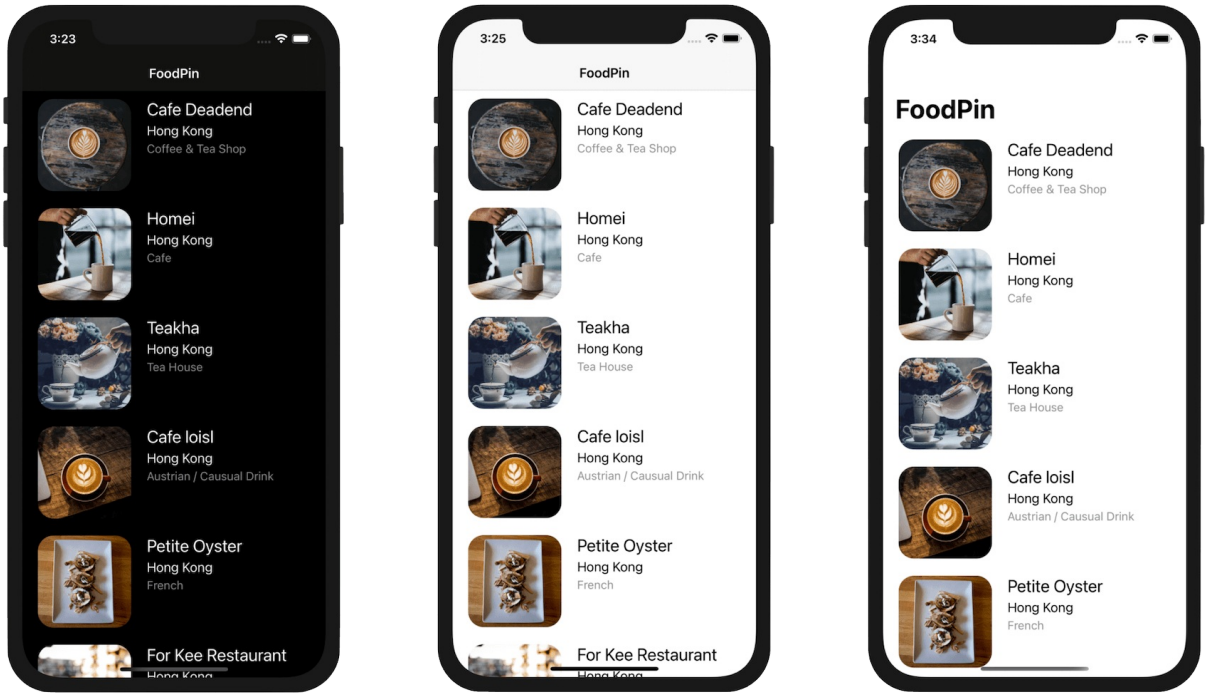


Figure 13-4. Default navigation bar title in iOS (left & middle), Large title enabled (right)

By default, large title is not enabled in Xcode. If you want to use large title in the navigation bar, you can insert this line of code in the `viewDidLoad()` method of the `RestaurantTableViewController` class and place it after `super.viewDidLoad()`:

```
navigationController?.navigationBar.prefersLargeTitles = true
```

Test the app again, and you will see the result as shown in figure 13-4.

Adding a Detail View Controller

With just a few clicks and a line of code, you've added a navigation bar to your app with large titles. What's missing is another view controller that displays the restaurant details. When a user selects a restaurant, the app transits to the detail

view controller and displays the restaurant details. This is what we are going to do.

In Interface Builder, drag a view controller from the Object library to create the detail view controller. The primary purpose of this chapter is to show you how to implement navigation controller. We'll keep the detail view as simple as possible. Let's just display the restaurant photo in the detail view. Drag an Image View from the Object library to the view controller. Resize it to fit the view, and add spacing constraints for each side of the image view. To ensure the image is scaled properly, go to the Attributes inspector and change the *mode* from *Scale to Fill* to *Aspect Fill*.

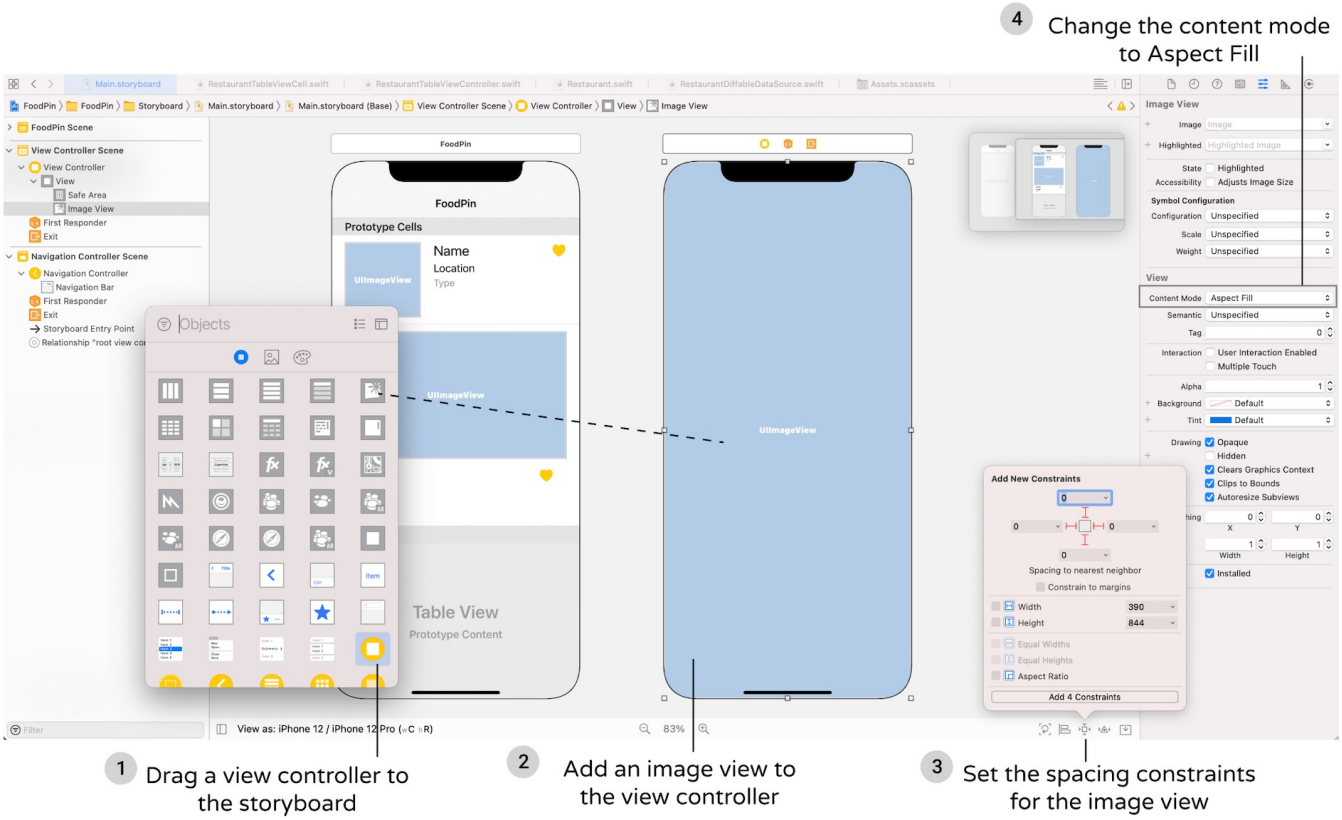


Figure 13-5. Adding an image view to the detail view controller

Now you have a table view controller and a detail view controller configured in the storyboard. The question is how can you connect them together? In storyboards, we need to connect them through a segue. In music, a segue is a seamless transition between one piece of music and another. In storyboards, the transition between two scenes is called *segue*.

The table view controller will transit to the detail scene when a user taps a cell. So, we will add a segue to connect the prototype cell and the detail scene. It's very straightforward to add a segue object. Press and hold the control key, click on the prototype cell (i.e. `datacell`) and drag to the Detail View Controller.

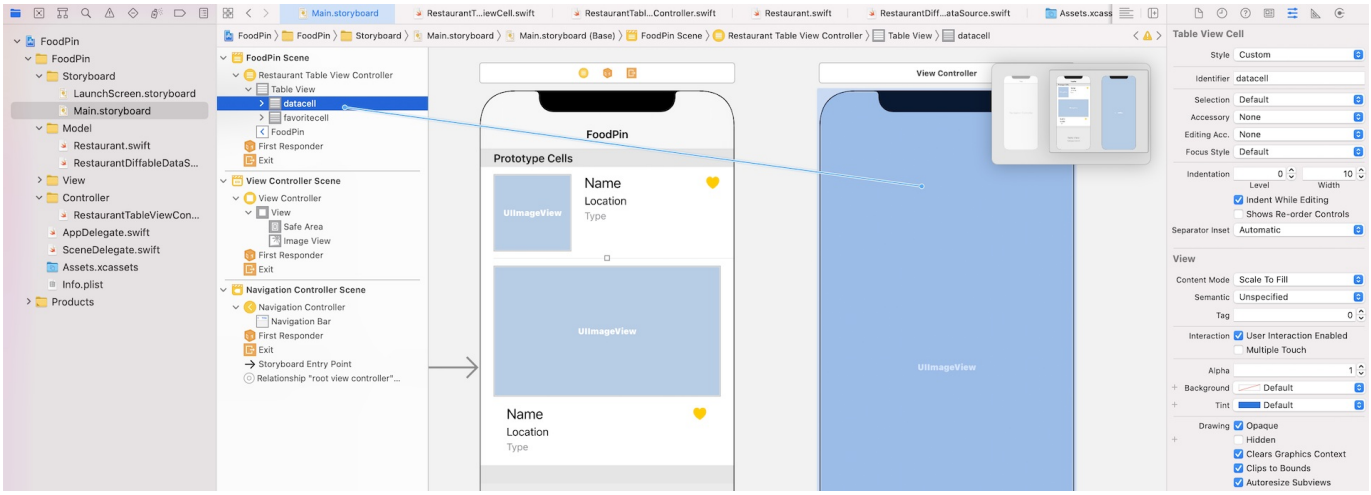


Figure 13-6. Connecting scenes with segue

Sometimes it is pretty difficult to select the prototype cell in the table view controller, I suggest you open the document outline view, and drag from the `datacell` to the view controller. When you release the buttons, a pop-up menu appears for you to choose the style for the segue. Select "Show" for the style.

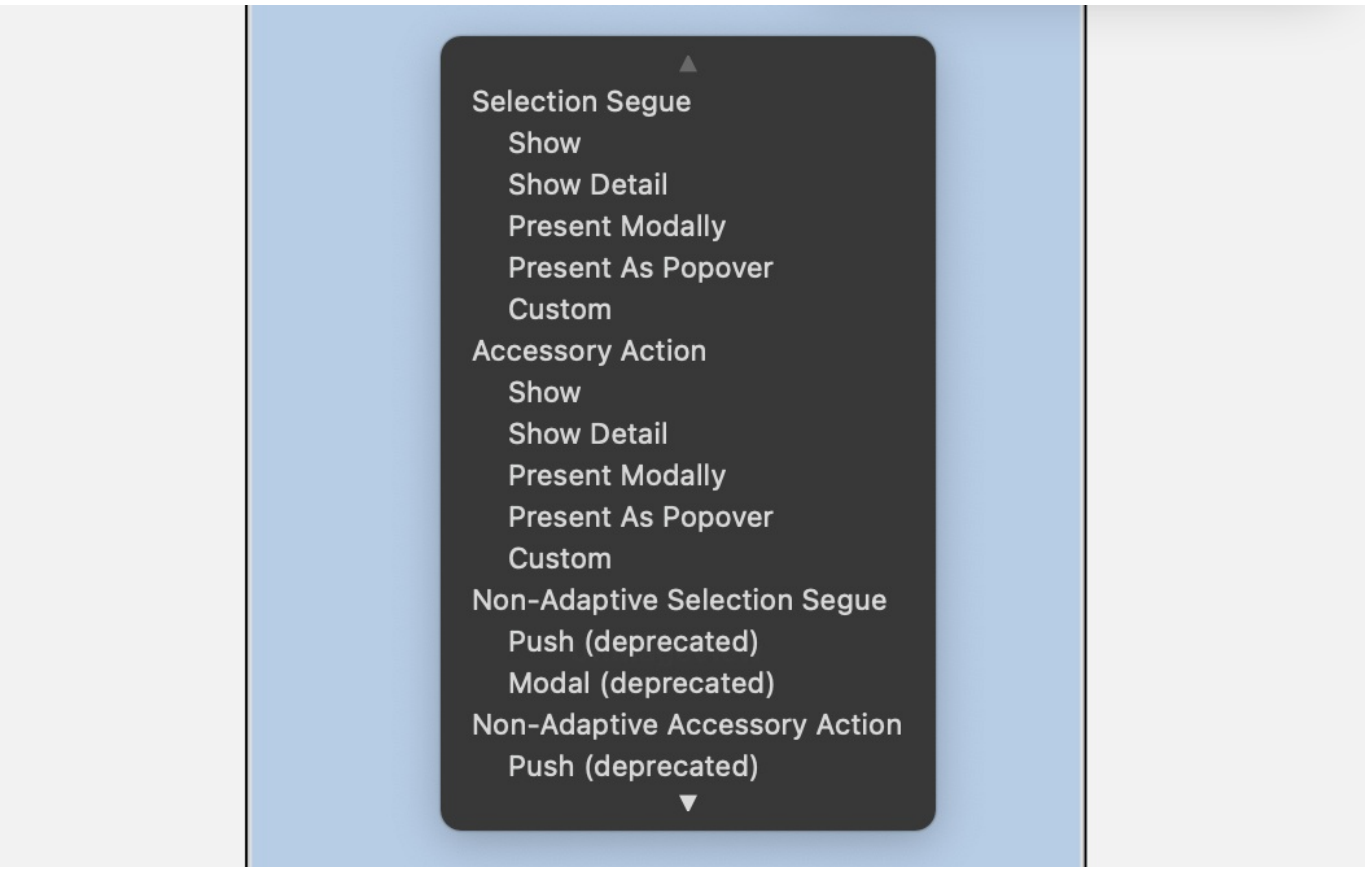


Figure 13-7. Choosing a segue style

Once chosen, Xcode automatically connects the `datacell` and the detail view controller with a *show* segue.

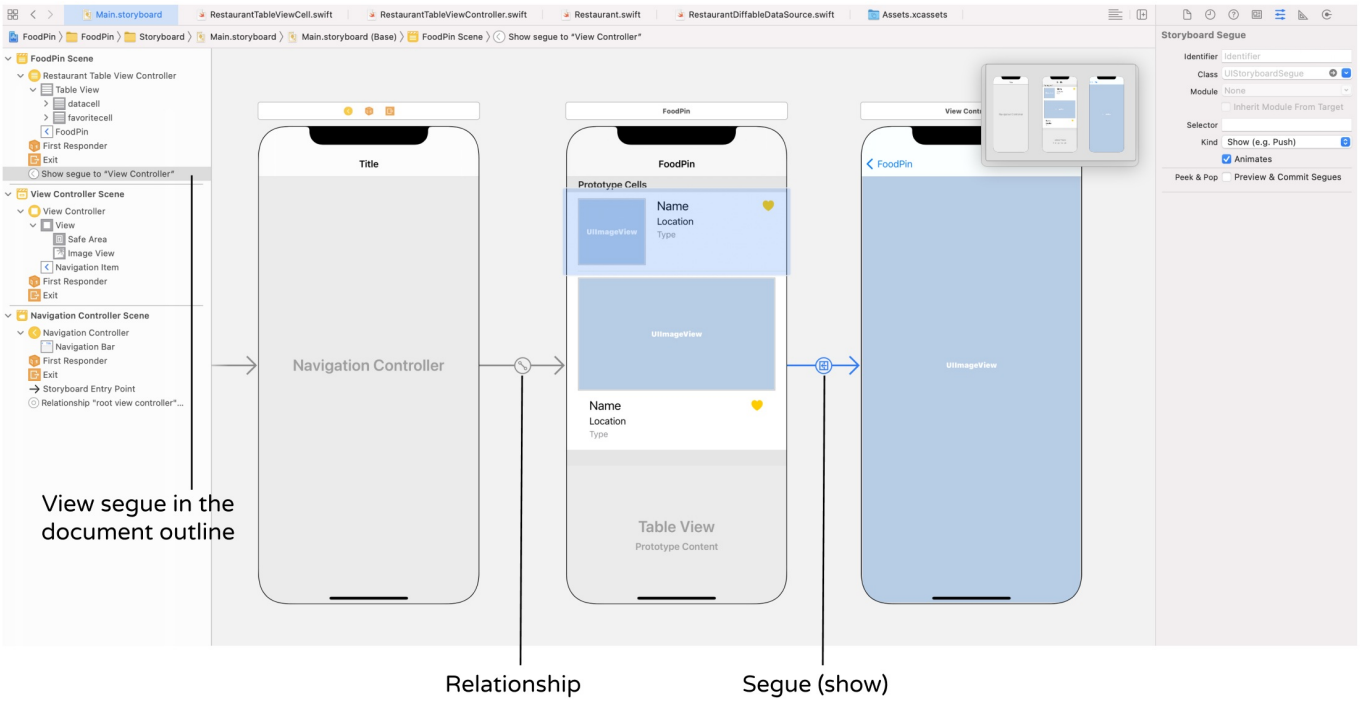


Figure 13-8. Viewing a segue

Repeat the same procedures to connect the `favoritecell` to the view controller. You will then in another segue between the table view controller and the view controller.

In iOS, it supports the following types of segue:

- **Show** - when the *show* style is used, the content is pushed on top of the current view controller stack. A back button will be displayed in the navigation bar for navigating back to the original view controller. Typically, we use this segue type for navigation controllers.
- **Show detail** - similar to the *show* style, but the content in the detail (or destination) view controller replaces the top of the current view controller stack. For example, if you select *show detail* instead of *show* in the FoodPin app, there will be no navigation bar and back button in the detail view.
- **Present modally** - presents the content modally. When used, the detail view controller will be animated up from the bottom and cover the entire screen on

iPhone. A good example of *present modally* segue is the "Add" feature of the built-in Calendar app. When you click the + button in the app, it brings up a "New Event" screen from the bottom.

- **Present as popover** - Present the content as a popover anchored to an existing view. Popover is commonly found in iPad apps, and you have already implemented popover in earlier chapters.

```
Note: These segue types are deprecated since the release of iOS 8: Push, Modal, Popover, Replace. When you see the word "deprecated", it means those functions are not recommended to use. They will probably be removed in the future version of iOS SDK.
```

Now, you're ready to run the app. Once launched, select a restaurant and the app should navigate to the detail view controller. Meanwhile, the detail view controller just shows a blank screen. The good news is that you already created a navigation interface.

Without writing a single line of code, you've added a navigation controller in your app. However, I guess you got a couple of questions in your mind:

- How can you pass the restaurant information from `RestaurantTableViewController` to the detail view controller?
- How can you display the photo of the selected restaurant in the detail view controller?

We'll look into them one by one in a minute.

Before moving on to the next section, let's make a little tweak. When you run the app and select a cell, it now navigates to the blank screen and shows an action sheet that we have implemented earlier. We no longer need that action sheet. Later

we'll add the same function in the detail view controller. Therefore, remove the `tableView(_:didSelectRowAt:)` method from `RestaurantTableViewController.swift`.

Quick tip: Sometimes you may just want to comment out a block of code instead of deleting them. Xcode provides a shortcut key to comment multiple lines of code. First, select the block of code and press `command+slash`. Xcode automatically marks the code block with comment indicators. To remove the comment, do the same procedures again.

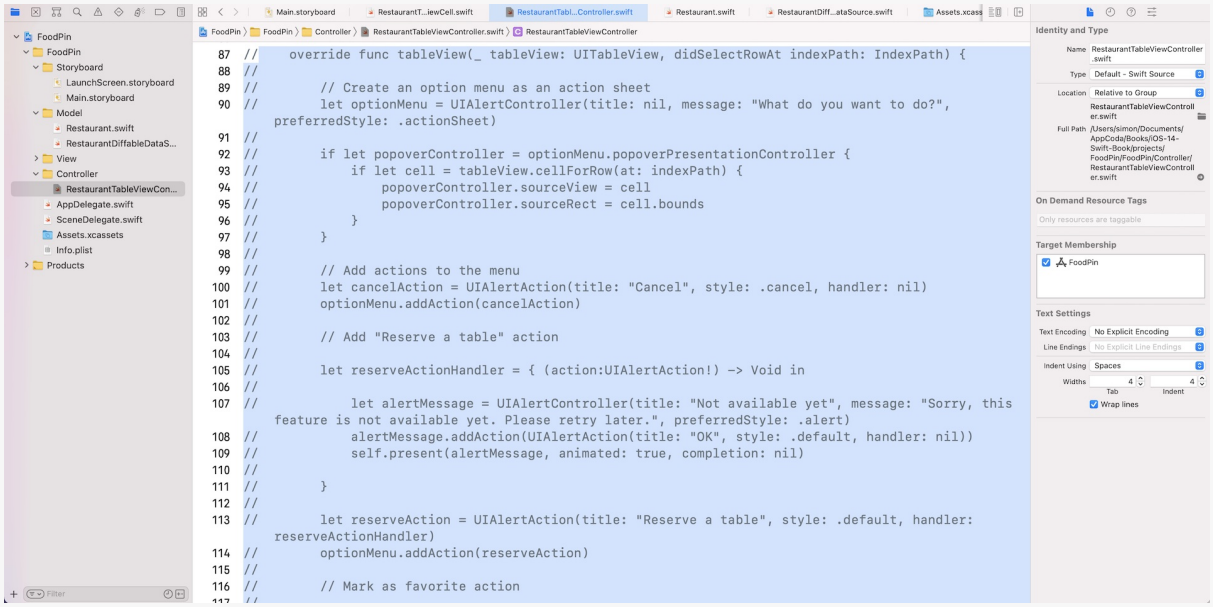


Figure 13-9. Use a shortcut key to comment a block of code quickly

Creating a New Class for the Detail View Controller

Okay, let's go back to the detail view controller. Our goal is to update the image view in the view controller with the selected restaurant. The view controller is now associated with the `UIViewController` class by default. As explained before, the `UIViewController` class only provides the fundamental view management model. There is no variable for storing the restaurant image. We have to extend `UIViewController` to create our own class so that we can add a new variable for the image view.

In the Project Navigator, right-click the *Controller* folder and select *New File...*. Choose *Cocoa Touch Class* as the class template. Name the class `RestaurantDetailViewController` and set it as a subclass of `UIViewController`. Click *Next* and save the file in your project folder.

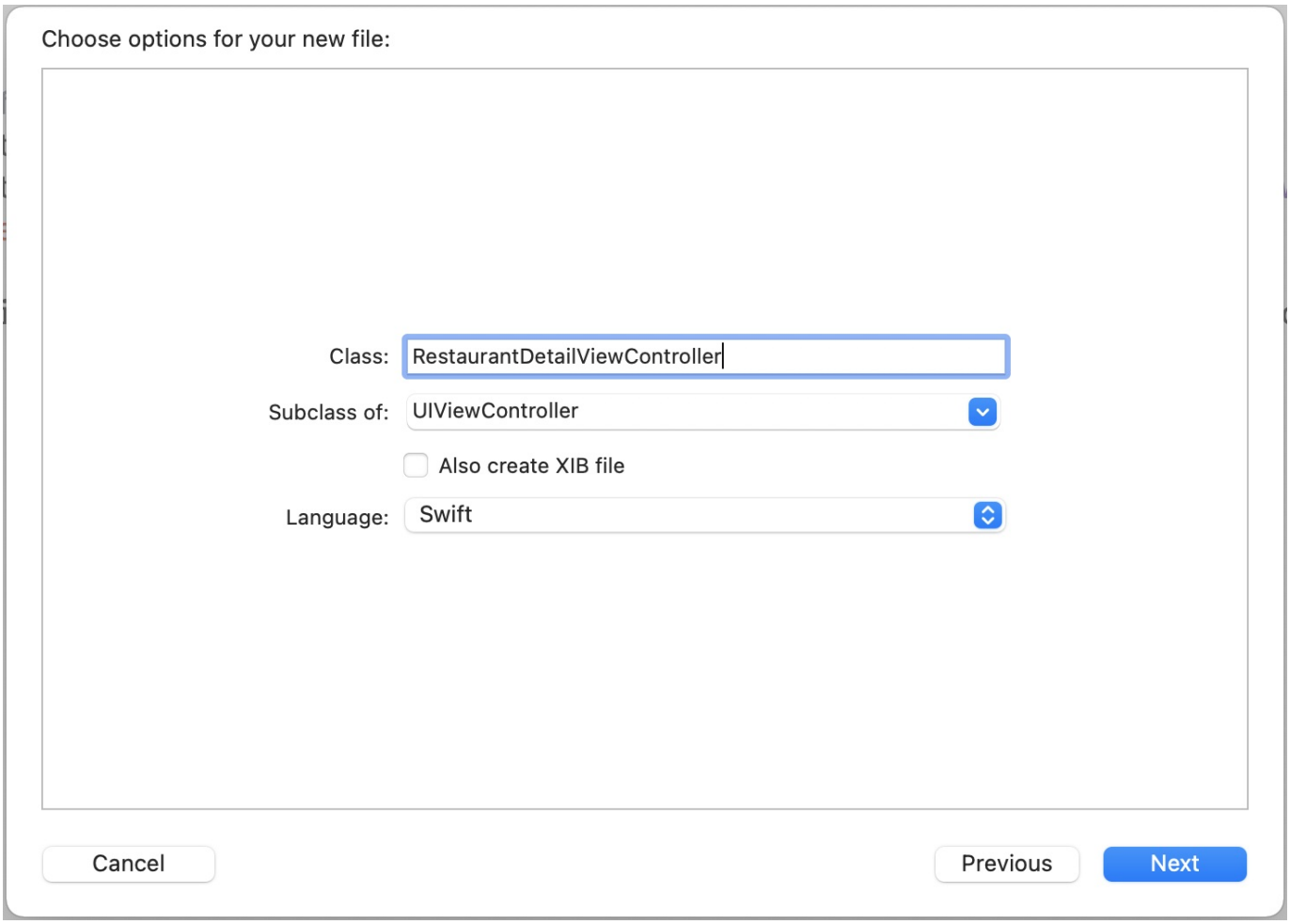


Figure 13-10. Create a DetailViewController class

First, we have to establish a relationship between the view controller in Interface Builder and the new class. Go to `Main.storyboard` and assign the `RestaurantDetailViewController` class to the detail view controller. In Interface Builder, select the view controller and open the Identity inspector. Change the custom class to `RestaurantDetailViewController`.

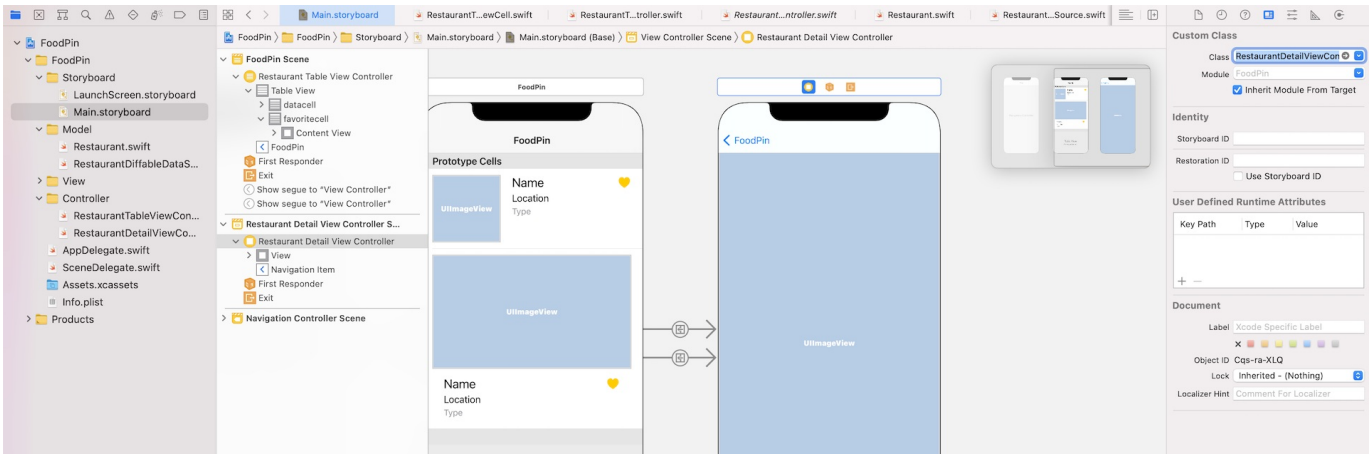


Figure 13-11. Assign a custom class for the detail view controller

Adding Variables to the Custom Class

There are a couple of things we have to add in the custom class:

- Create a variable named `restaurantImageName` – when a user selects a restaurant in the table view controller, there must be a way to pass the image name to the detail view. This variable will be used for data passing.
- Create an outlet called `restaurantImageView` for the image view – we need a reference to update the image view of the detail view controller, so we have to create an outlet.

Okay, insert the following code to the `RestaurantDetailViewController` class:

```
@IBOutlet var restaurantImageView: UIImageView!

var restaurantImageName = ""
```

Next, establish a connection between the `restaurantImageView` variable and the image view of the detail view controller. Go back to `Main.storyboard`. Right click the Restaurant Detail View Controller object in the document outline. In the

popover menu, connect the `restaurantImageView` outlet with the image view of the controller.

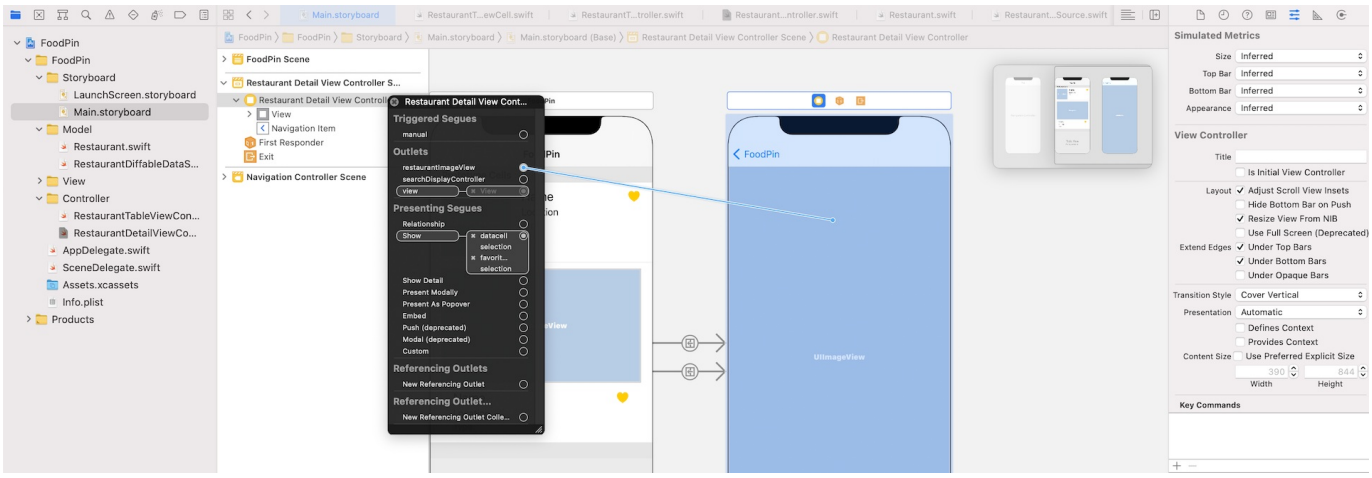


Figure 13-12. Establish the connection between the image view and the outlet

Now that you've linked up the outlet variable with the image view object in the storyboard, but there is still one thing left. You haven't provided the image yet. The image view should display the selected restaurant image. In the `viewDidLoad` method of the `RestaurantDetailViewController` class, insert a line of code. Your method should look like this:

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

    restaurantImageView.image = UIImage(named: restaurantImageName)
}
```

The `viewDidLoad` method is called when the view is loaded into memory. You can provide additional customization of the view in this method. In the above, we simply set the image view's image to the selected restaurant image.

Try to compile and run your app. Oops! The detail view is still blank after selecting a restaurant. We still miss one thing. We haven't passed the image name of the selected restaurant from the table view controller to the detail view controller. This is why the `restaurantImageName` variable is not assigned with any value.

Passing Data Using Segues

This comes to the core part of this chapter about data passing with segues. A segue manages the transition between view controllers, and contains the view controllers involved in the transition. When a segue is triggered, before the visual transition occurs, the storyboard runtime notifies the source view controller (i.e. `RestaurantTableViewController`) by calling the `prepare(for:sender:)` method:

```
func prepare(for segue: UIStoryboardSegue, sender: Any?) {

}
```

The default implementation of the `prepare(for:sender:)` method does nothing. By overriding the method, you can pass any relevant data to the new controller, which is `RestaurantDetailViewController` in our project.

Segues can be triggered by multiple sources. As your storyboard becomes more complex, it is very likely that you'll have more than one segue between view controllers. Therefore, the best practice is to give each segue a unique identifier. This identifier is a string to distinguish one segue from another. To assign an identifier for a segue, select the segue in the storyboard editor, and then go to the Attributes inspector. Set the value of the identifier to `showRestaurantDetail`.

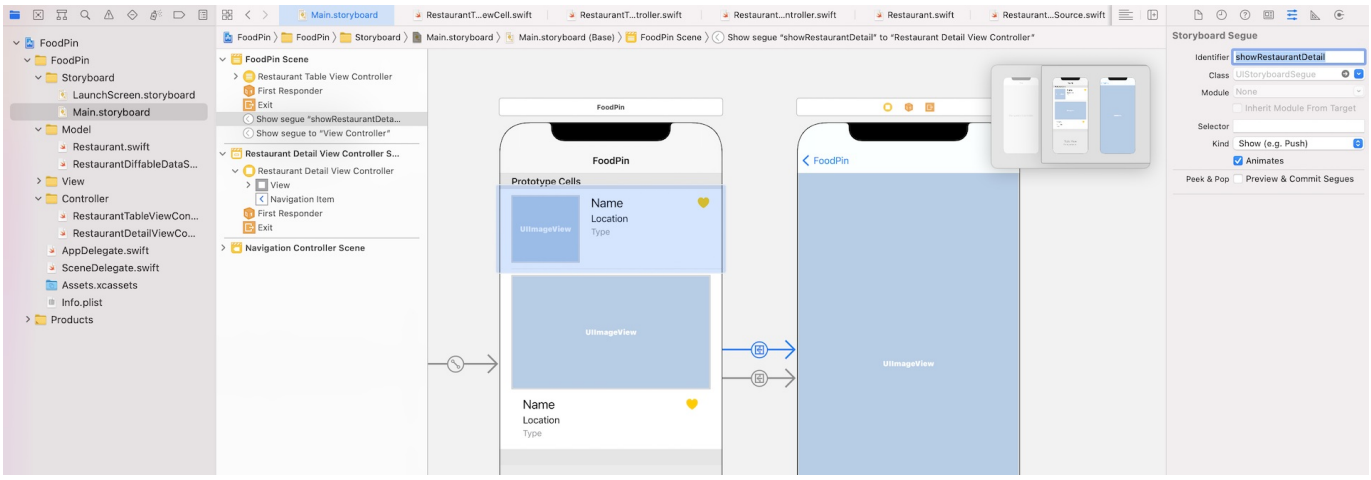


Figure 13-13. Adding a segue identifier

You have to set both segue for `datacell` and `favoritecell` to the same identifier (i.e. `showRestaurantDetail`) because your app can switch between these two types of cells.

With the segue configured, insert the following code in the `RestaurantTableViewController` class to override the default implementation of the `prepare(for:sender:)` method:

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    if segue.identifier == "showRestaurantDetail" {
        if let indexPath = tableView.indexPathForSelectedRow {
            let destinationController = segue.destination as! RestaurantDe
tailViewController
            destinationController.restaurantImageName = self.restaurants[i
ndexPath.row].image
        }
    }
}
```

The first line of code is used to check the segue's identifier. The block of code is only executed for the `showRestaurantDetail` segue. In the code block, we first retrieve the selected row by accessing `tableView.indexPathForSelectedRow`. The `indexPath` object should contain the selected cell.

A segue object contains both the source and destination view controllers involved in the transition. You use `segue.destination` to retrieve the destination controller. In this case, the destination controller is the `RestaurantDetailViewController` object. This is why we have to downcast it by using the `as!` operator. Lastly, we pass the image name of the selected restaurant to the destination controller.

Now, it's ready to test the app. Hit the Run button to compile and run the app. This time, your app should work as expected. Select a restaurant in the table view, the detail view should display the image of the selected item.

Disabling Large Titles

Other than the home screen of the app, you may not want to use large titles to catch users' attention. Apple provides developers an option to disable large titles in a particular view controller. For example, if you want to disable large titles in the detail view controller, you can insert the following line of code in the `viewDidLoad()` method of `RestaurantDetailViewController`:

```
navigationItem.largeTitleDisplayMode = .never
```

The code above changes the `largeTitleDisplayMode` of the navigation item to `.never`, indicating that it does not want to have the large title. The display mode can also be `.always` or `.automatic`. As the name suggests, `.always` means large title should always be used. If `.automatic` is set, it inherits the large title mode in the previous view controller.

Your Exercise

Wouldn't it be better to show more restaurant information in the detail view? In this exercise, you're required to add a few labels for displaying the name, type and location of the selected restaurant. Your resulting screen should look figure 13-14. If you understand how data passing works, it shouldn't be difficult for you to make these changes. A hint is that instead of passing the name of the restaurant image, you need to pass the restaurant object to the detail view controller.

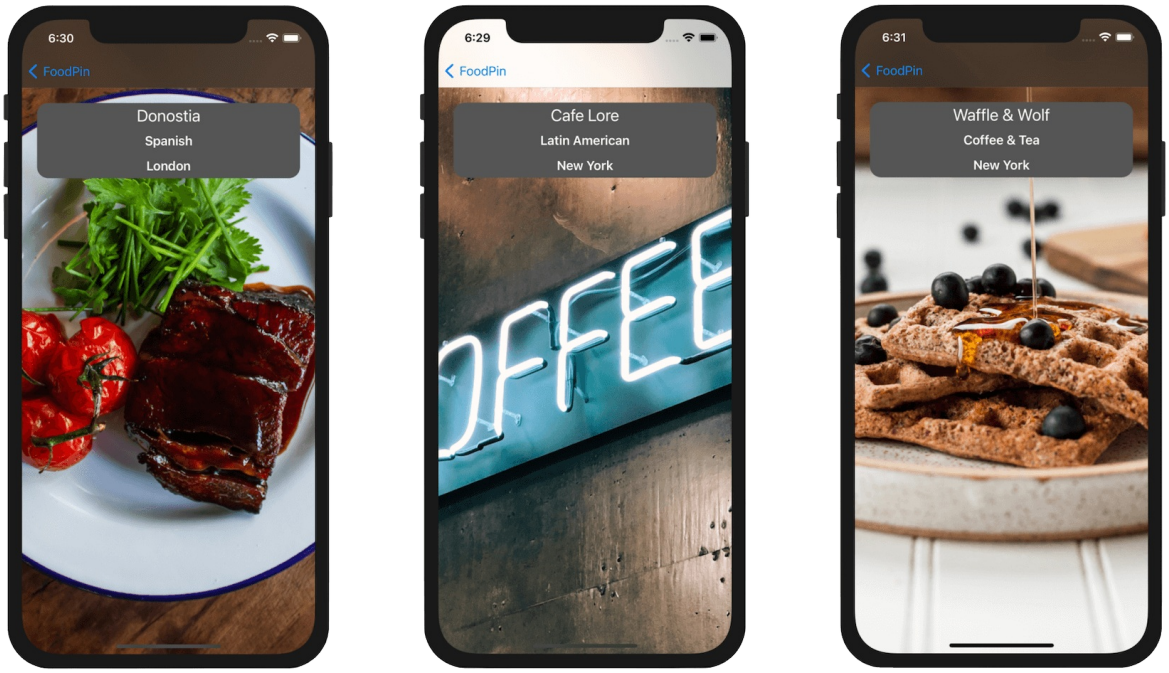


Figure 13-14. The detail view with more restaurant information

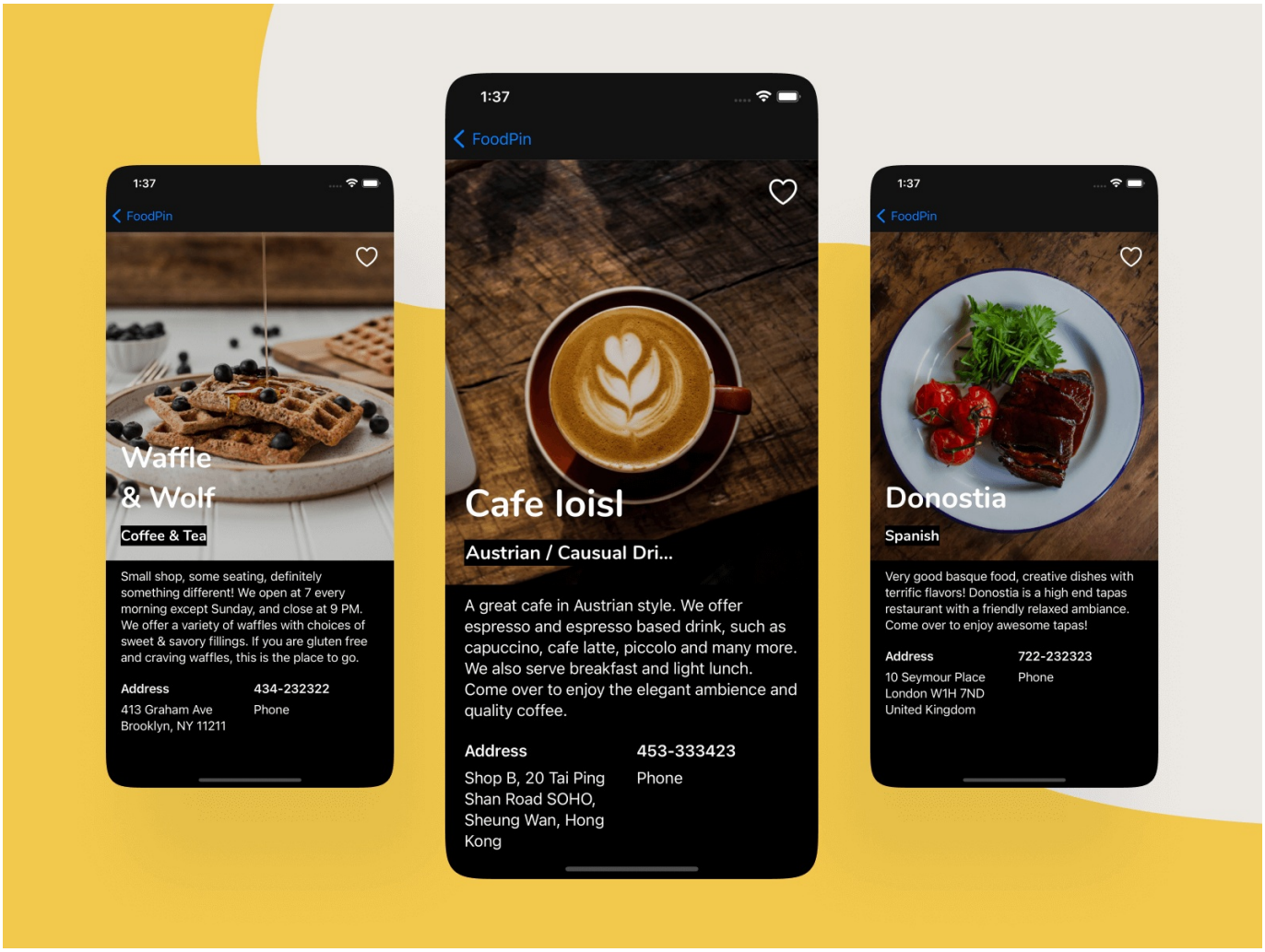
I highly recommend you to complete this exercise before moving onto the next chapter. Not only it helps you better understand the concept of segue and data passing, it will revise your knowledge of stack views and auto layout.

Summary

In this chapter, I have walked you through the basics of navigation controllers and segues. What we have built is really simple. We only pass the restaurant image from one view controller to another. But, by now you should know how to handle data passing between view controllers through segues.

For reference, you can download the complete Xcode project from http://www.appcoda.com/resources/swift53/FoodPinNavigationController.zip.

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.

Familiarize with the Starter Project

To begin, first download the FoodPin project from http://www.appcoda.com/resources/swift53/FoodPinDetailViewStarter.zip. The project was based on what we have completed in the previous chapter but I have modified the `Restaurant` struct to include two additional properties:

- phone
- description

The initialization of the `restaurants` array has been updated. If you look into the code in `RestaurantTableViewController.swift`, you should find that the values of the `location` property are updated with the full address. Also, it includes some dummy phone numbers and description like this:

```
var restaurants:[Restaurant] = [
    Restaurant(name: "Cafe Deadend", type: "Coffee & Tea Shop", location:
"G/F, 72 Po Hing Fong, Sheung Wan, Hong Kong", phone: "232-923423", descri
ption: "Searching for great breakfast eateries and coffee? This place is f
or you. We open at 6:30 every morning, and close at 9 PM. We offer espress
o and espresso based drink, such as capuccino, cafe latte, piccolo and man
y more. Come over and enjoy a great meal.", image: "cafedeadend", isFavori
te: false),
    Restaurant(name: "Homei", type: "Cafe", location: "Shop B, G/F, 22-24A
 Tai Ping San Street SOHO, Sheung Wan, Hong Kong", phone: "348-233423", de
scription: "A little gem hidden at the corner of the street is nothing but
 fantastic! This place is warm and cozy. We open at 7 every morning except
 Sunday, and close at 9 PM. We offer a variety of coffee drinks and specia
lties including lattes, cappuccinos, teas, and more. We serve breakfast, l
unch, and dinner in an airy open setting. Come over, have a coffee and enj
oy a chit-chat with our baristas.", image: "homei", isFavorite: false),

      ...

]
```

In the storyboard of the starter project, all the subviews of the detail view controller have been removed. We no longer use those views because we are going to redesign it.

These are the changes that I have prepared for you in the starter project. After you download the project, I encourage you to run it once and familiarize with the changes.

Using Custom Fonts

The San Francisco font was introduced in Nov 2014 and has been used as the default font in iOS apps. What if you find an open source font on Google Fonts (https://fonts.google.com) and want to use it in your app? How can you do it?

Xcode has made it very easy for developers to use custom fonts. All you need is just add the custom font files into your Xcode project. Let's say, you prefer to use Nunito in your app. You can go up to https://fonts.google.com/specimen/Nunito. Click *Download Family* to download the font files.

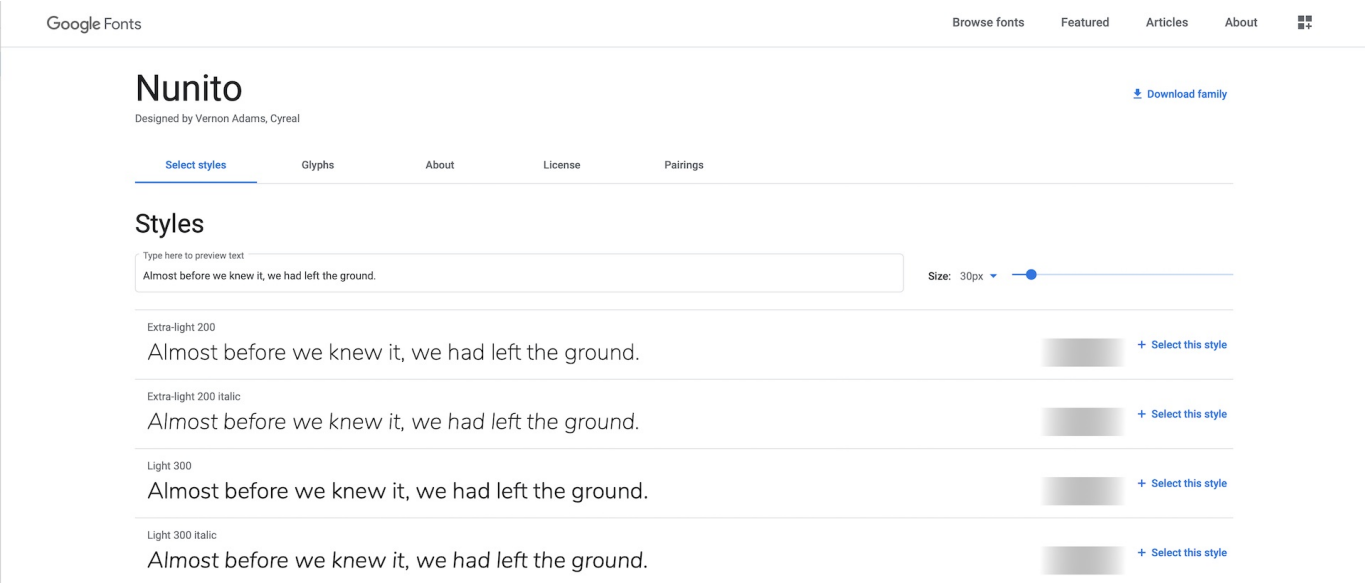


Figure 14-1. Downloading your preferred Google fonts

Now go back to Xcode and open the starter project (if you haven't). In the project navigator, right click the *FoodPin* folder and select *New Group*. Name the group *Resources*. Next, right click the *Resources* folder and select *New Group* to add a sub group. Name the sub group *Fonts*. Select *Nunito-Regular.ttf* and *Nunito-Bold.ttf*, and add them to the *Fonts* group. In case if you want to use all the font styles, you can add all the font files to the group.

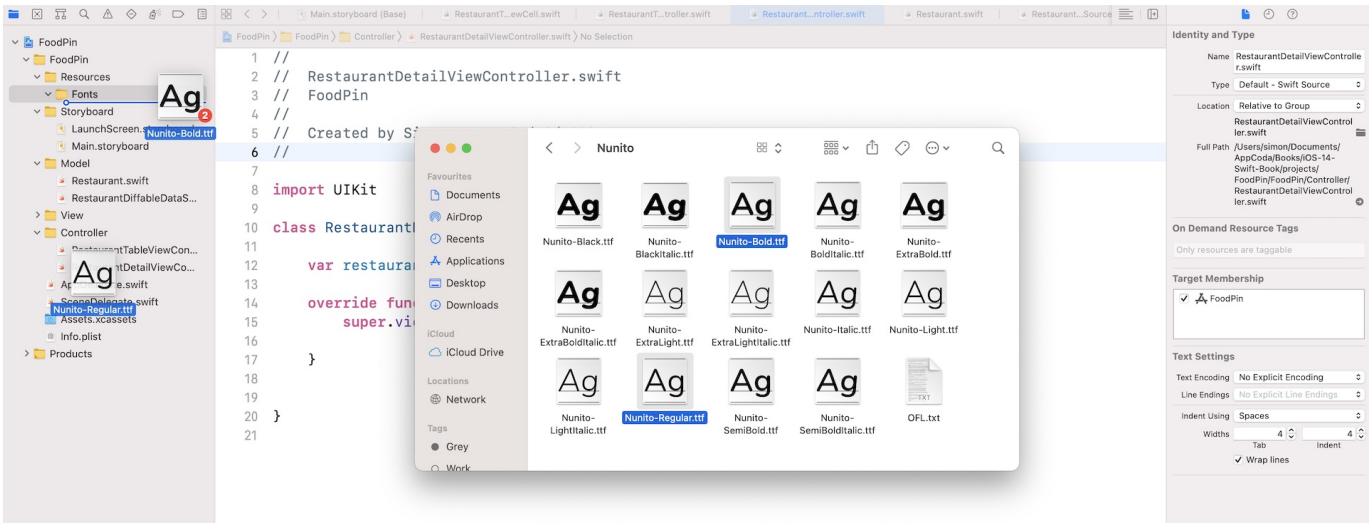


Figure 14-2. Adding the fonts file to the Xcode project

> **Note:** Is it a mandate to create a sub group? No, it is my practice to organize resource files like that.

Once you dropped the files into the group, you will be prompted with the dialog as shown in figure 14-3. Make sure you check the *Copy items if needed* option and the *FoodPin* target.

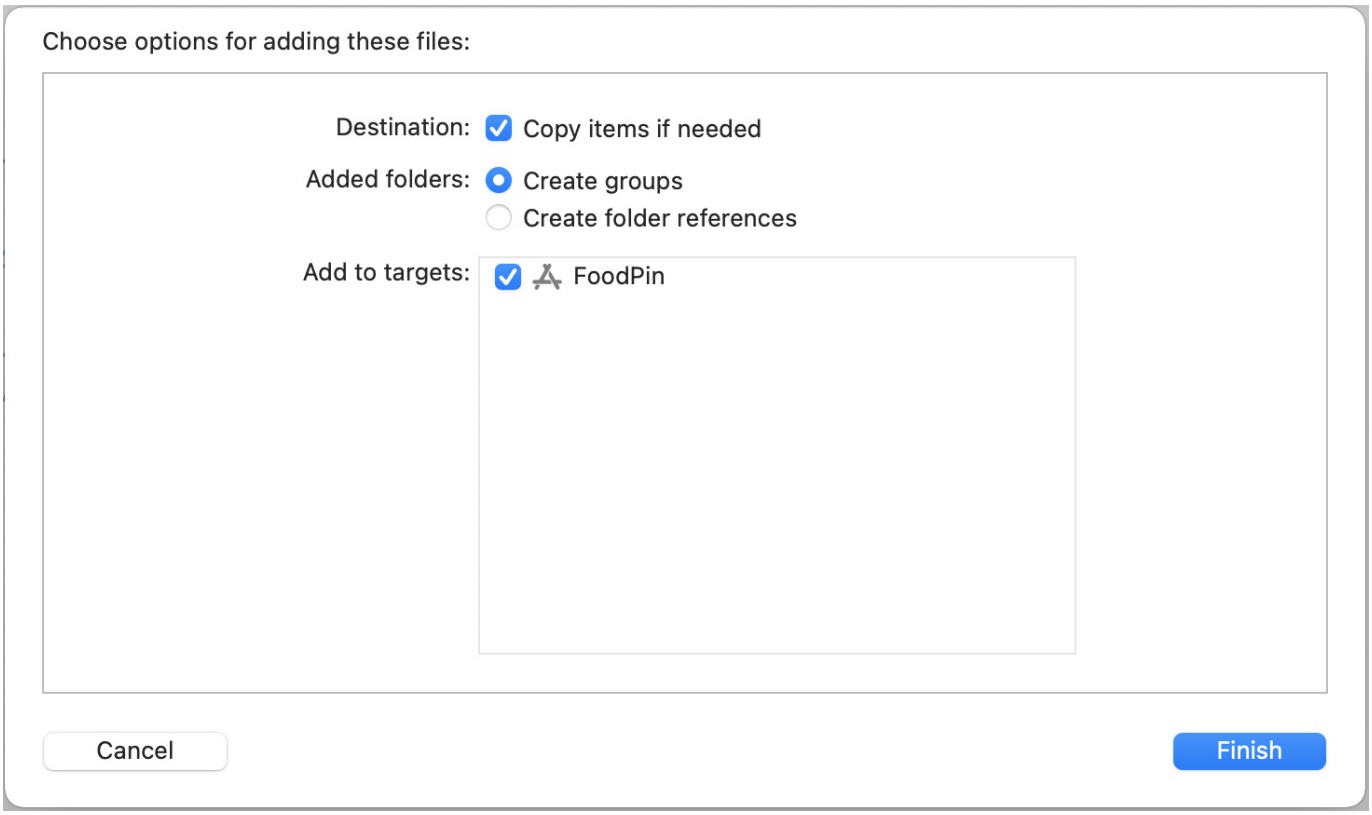


Figure 14-3. Select copy item if needed and the target

Once you hit *Finish*, the font files will appear in the project navigator. To ensure your app can use the font files, hold command key to select all the fonts files. In the File inspector, make sure the *FoodPin* option under target membership is enabled. If not, tick the option.

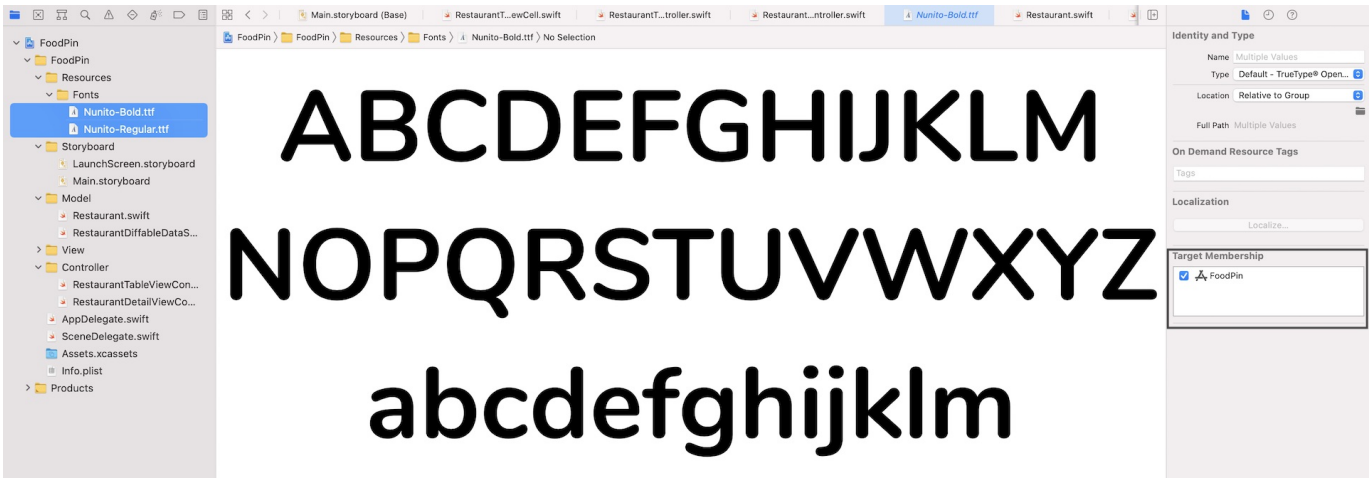


Figure 14-4. Select the FoodPin option under Target Membership

That's the procedures to install a custom font. Later you will be able to use the *Nunito* font when we design the detailed view.

Designing the Table View Header

Now let's start to design the detail view. Figure 14-5 shows the expected design of the detail screen. There are multiple ways to implement this kind of screen. One way is to use a dynamic table view to realize it. You have learned what a dynamic table view is. In earlier chapters, you created a custom table view with prototype cells. The prototype cell is similar to a design template. Though the cell's data differs, the look & feel of each cell is exactly the same. That's the idea of prototype cells.

As you know, it is not a mandate to design a table view using one prototype cell. We have already created two prototype cells for the restaurant table view. To create the detailed screen, we will create a couple of prototype cells. The other thing I haven't mentioned so far is that each table view has its own header view. This header view resides on top of the table cells. The header view is empty by default, but you are allowed to add your own view to that.

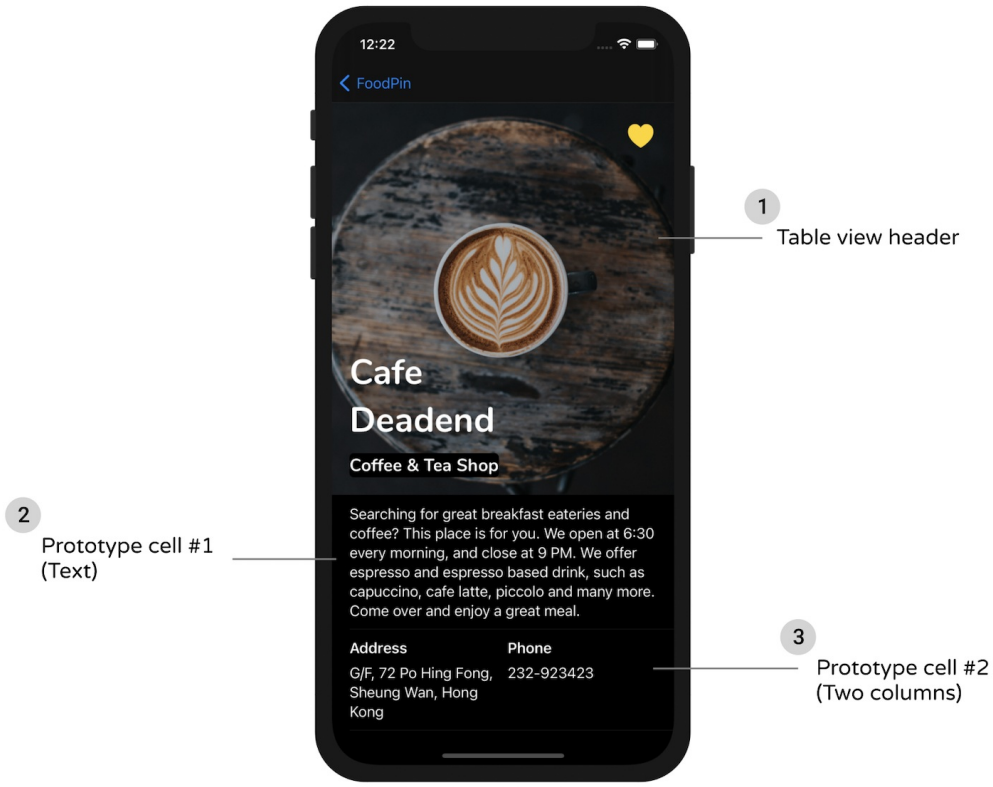


Figure 14-5. Breaking down the detail view into different cells

So, here is what we are going to do to realize the detail view design:

1. Create a table header view that displays the restaurant image, name, type, and the heart image.
2. Create two prototype cells for the restaurant information:
    - **Prototype cell #1** - for displaying a paragraph of text.
    - **Prototype cell #2** - for displaying two columns of information. In the detailed view, we use this prototype cell to display the restaurant's address and phone number.

Now that you should have some ideas about how we structure the table view, let's dive into Interface Builder and design the view.

Go to `Main.storyboard` of the starter project. Open the Object library and select a table view object (*note: it's a table view instead of a table view controller*). Drag it to the detail view controller. Resize it to fit the whole view and then click "Add new constraints". Set the value of the *top*, *left*, *right*, and *bottom* sides to `0`. And, click Add 4 constraints. This ensures the table always fits the whole screen regardless of the screen size. Your screen should look like that in figure 14-6.

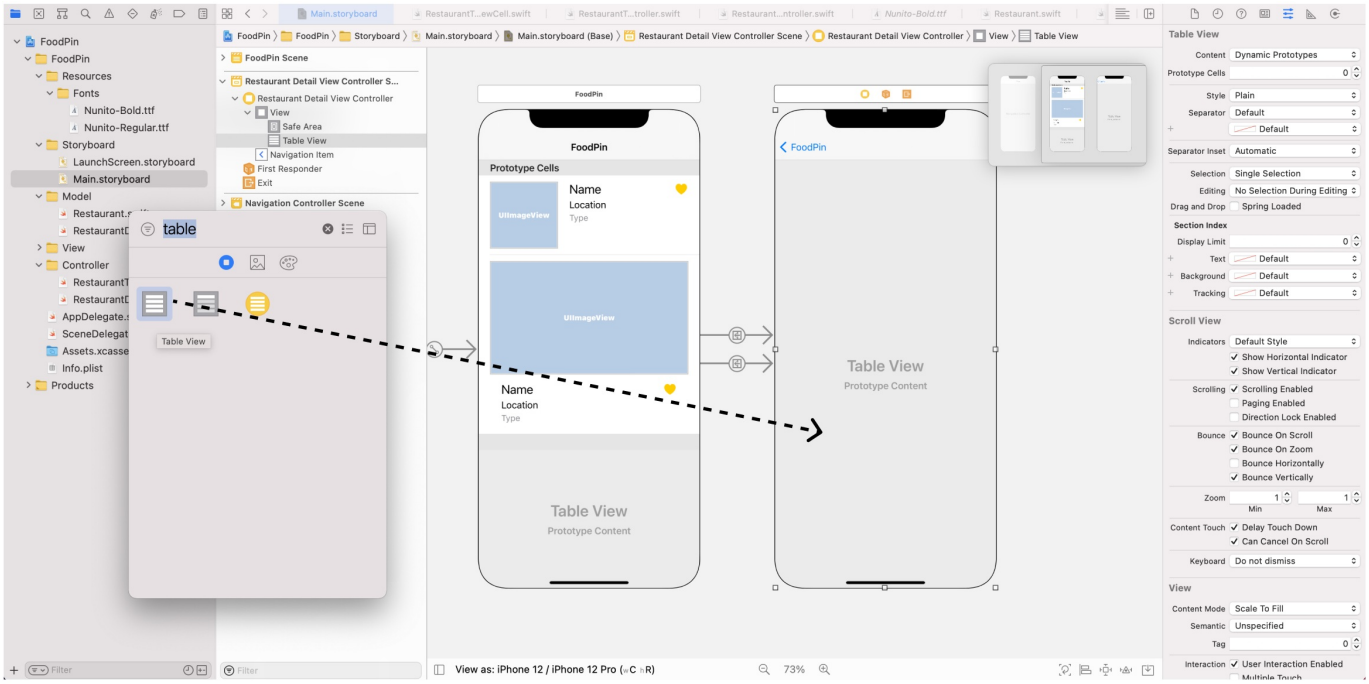


Figure 14-6. Adding a table view to the detail view controller

Select the table view and set the prototype cells to `2` in the Attributes inspector. This adds two prototype cells in the table view. We will deal with the prototype cells later.

Now, add a view from the object library to the table view's header. In the document outline view, drag the view to table view and place it right above the first table view cell. Change the height of the view to `445` points in the Size inspector. This view

serves as the container view holding other elements (e.g. image view) which we will add later. Once done, the design of your detail view should look like that in figure 14-7.

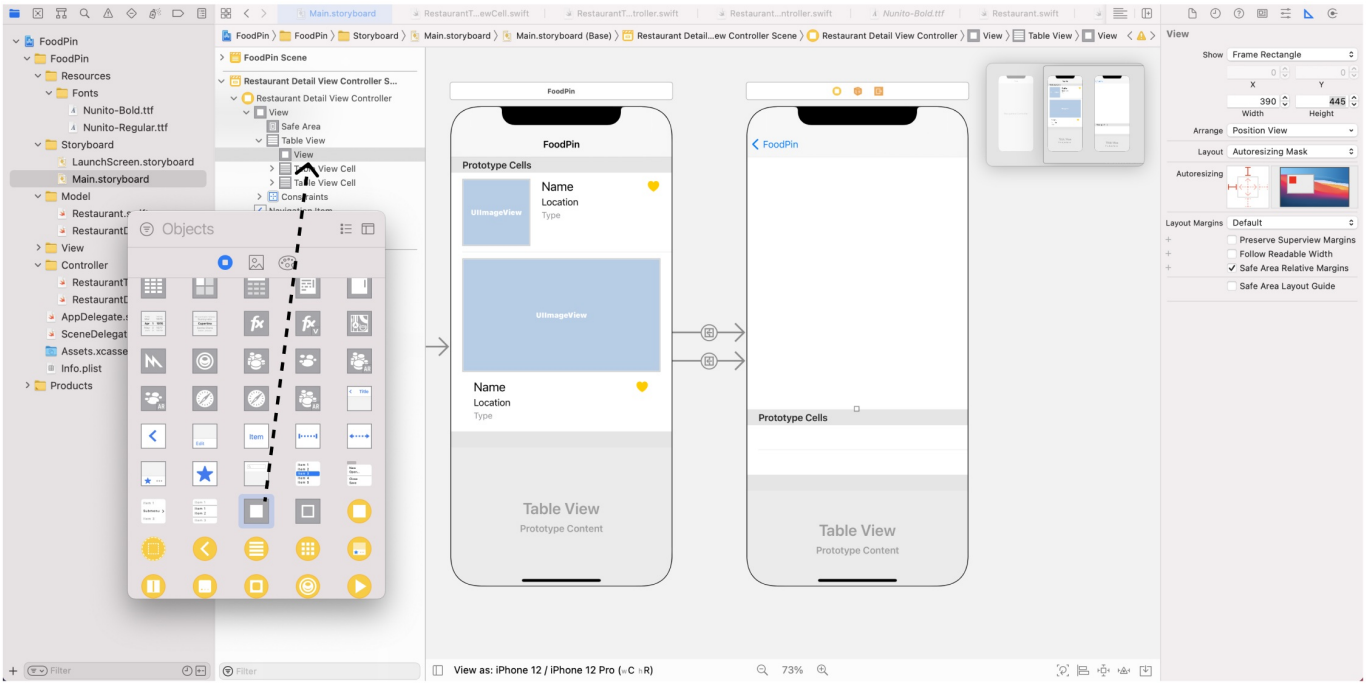


Figure 14-7. A view has been added to the table view's header

You can rename the view from `View` to `Header View` in the document outline. Just click on `View` and change it accordingly. This step is completely optional. However, by giving each view a meaningful name, it will make you easier to identify a specific view later.

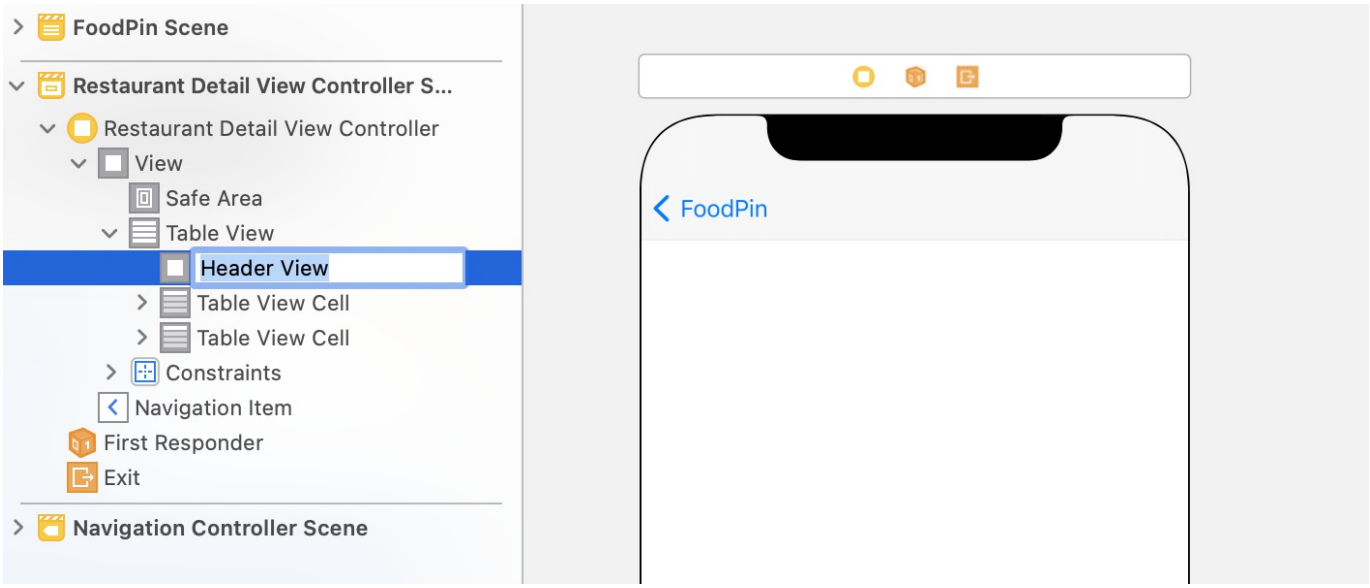


Figure 14-8. Changing the name of the view

We haven't designed the header view yet. Now drag an image view from the Object library and place it in header view. In the Attributes inspector, change the *Content Mode* to `Aspect Fill`. Again, to organize our UI components better, give the image view a proper name in document outline. Set it to `Header Image View`.

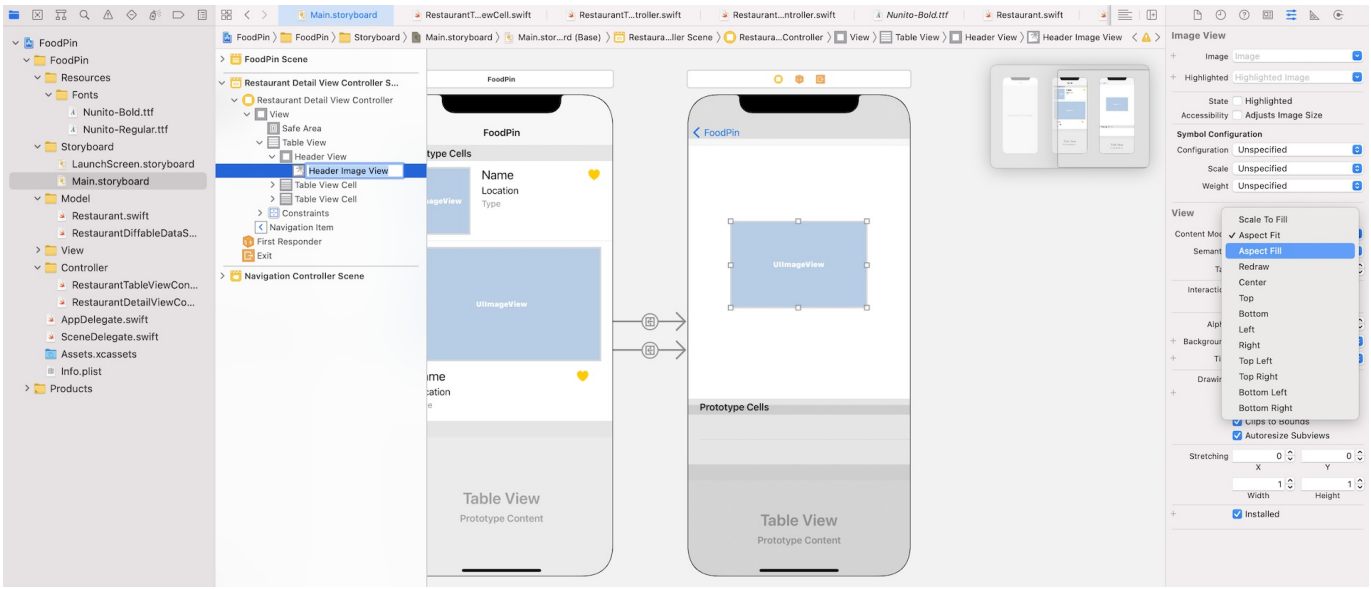


Figure 14-9. Adding an image view to the header view

You do not need to add auto layout constraints for the header view because it is added as the table view header. However, we have to define some constraints for the header image view. The image view should always fit the size of the header view. To make this possible, we will define four spacing constraints for the top, bottom, left and right sides of the image view. Select the image view and click the *Add new constraints* button of the layout bar. Set the spacing of the top, left, right and bottom side to `0`. Make sure the bars are in solid red and the *Constrain to margins* option is unchecked. Then hit the *Add 4 Constraints* button to add the constraints.

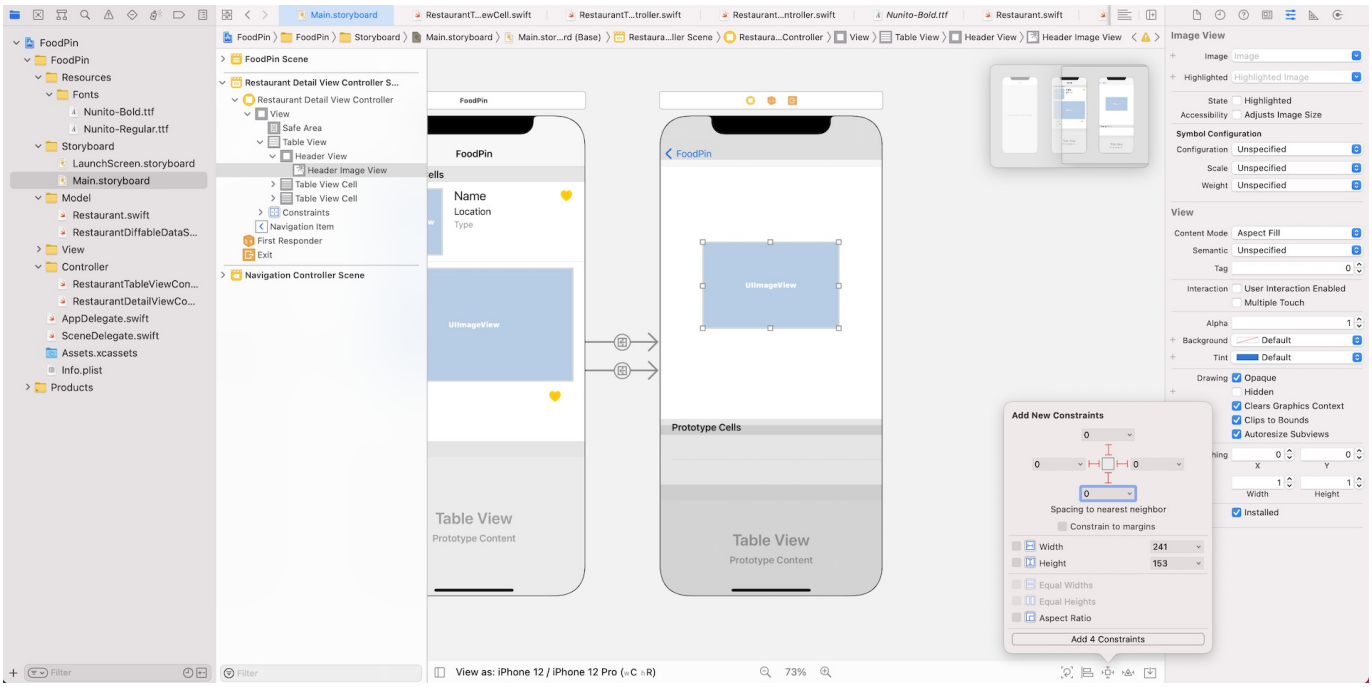


Figure 14-10. Adding layout constraints for the image view

Next, add a label and place it near the lower left corner of the image view. This label is used to display the restaurant name. Make sure the label is selected and go to the Attributes inspector. Change the color to white and set the *Font* to *Title 1* text style. Also, change its text to `Name`.

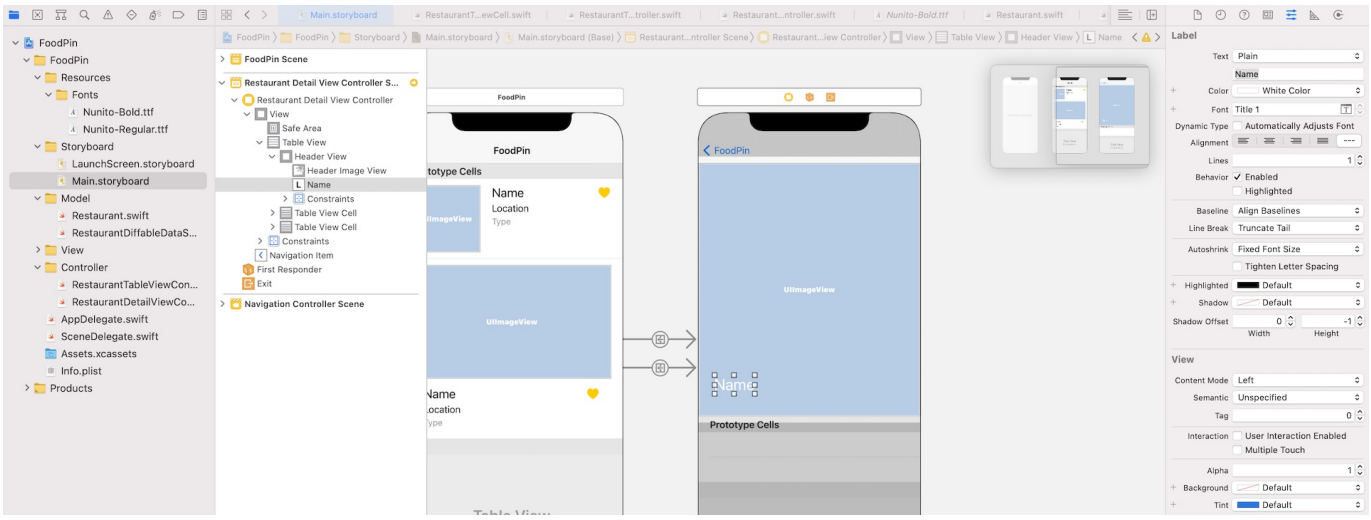


Figure 14-11. Add a new label to the header view

Now drag another label and place it under the previous label object. This new label is used for presenting the restaurant type. In the Attributes inspector, set the background color to *black*. For the font color, change it to *white*. Change the font style to `headline`. Also, change the alignment option to *center*. Lastly, change its text to `Type`.

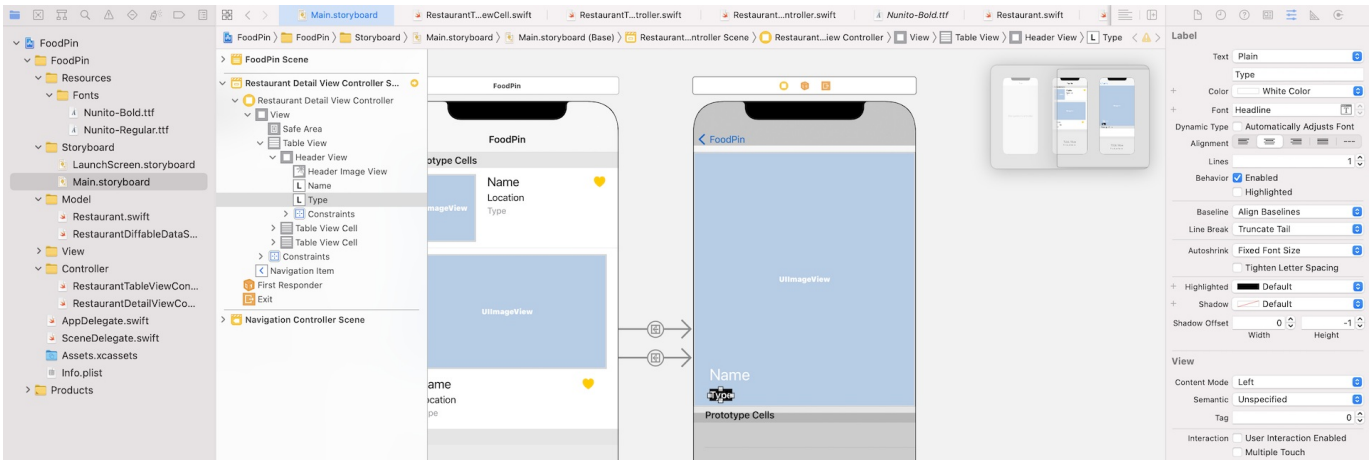


Figure 14-12. Adding the type label

Next, drag a new button from the Object library and put it at the top right corner of

the header view. This is an image button that allows users to mark the restaurant as favorite. In the Attributes inspector, set its title to *blank* and image to `heart`, which is a SF symbol. To resize the *heart* symbol, change the *configuration* option in the Default Symbol Configuration section and set it to `Point Size`. Increase the *point size* to `30` points to make the heart icon larger. As a final touch, change the button's tint color to *white*.

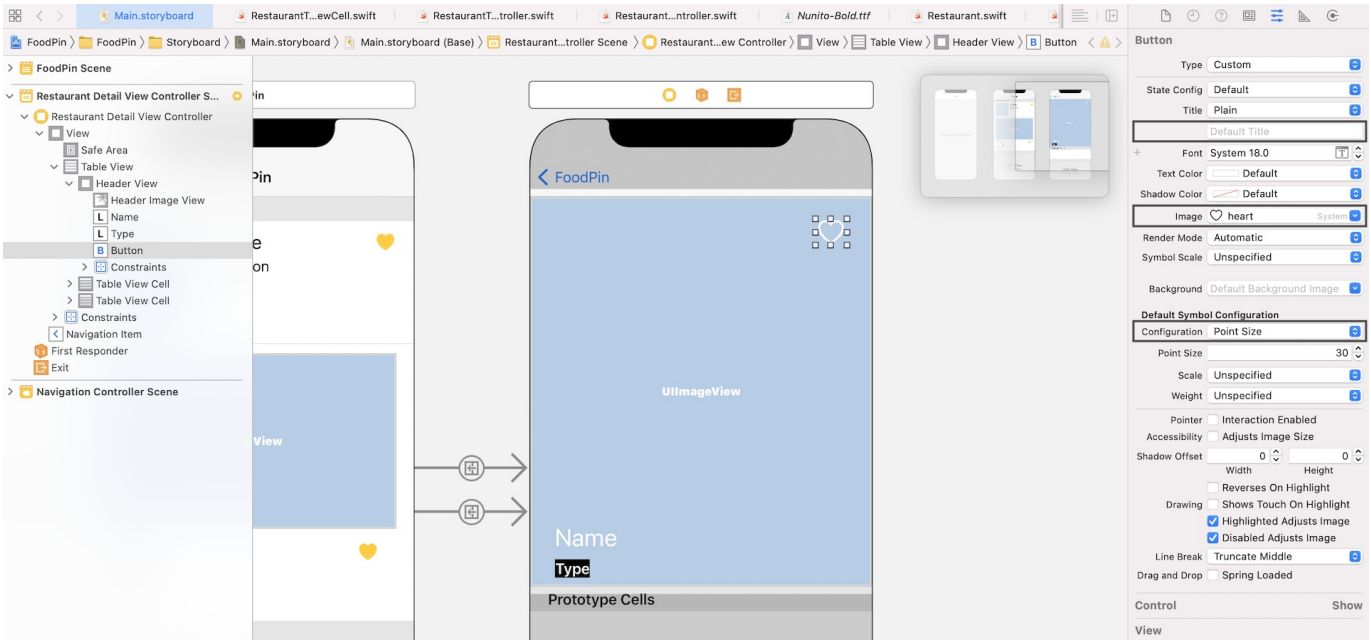


Figure 14-13. Adding a button for the heart icon

Now it's time to add some layout constraints for these components. Let's first start with the heart button. I want to define the following layout requirement for it:

- The size of the button should be fixed
- The top side of the label should be 10 points away from margin of the header view.
- The right (or trailing) side of the label should be 10 points away from margin of the header view.

Select the heart button and then click the *Add new constraints* button. Set the values of both top and right sides to `10` points. Please make sure the *Constrain to margins* option is checked. Next, check both width and height options to enable the size constraints. Lastly, click the *Add 4 Constraints* button to proceed.

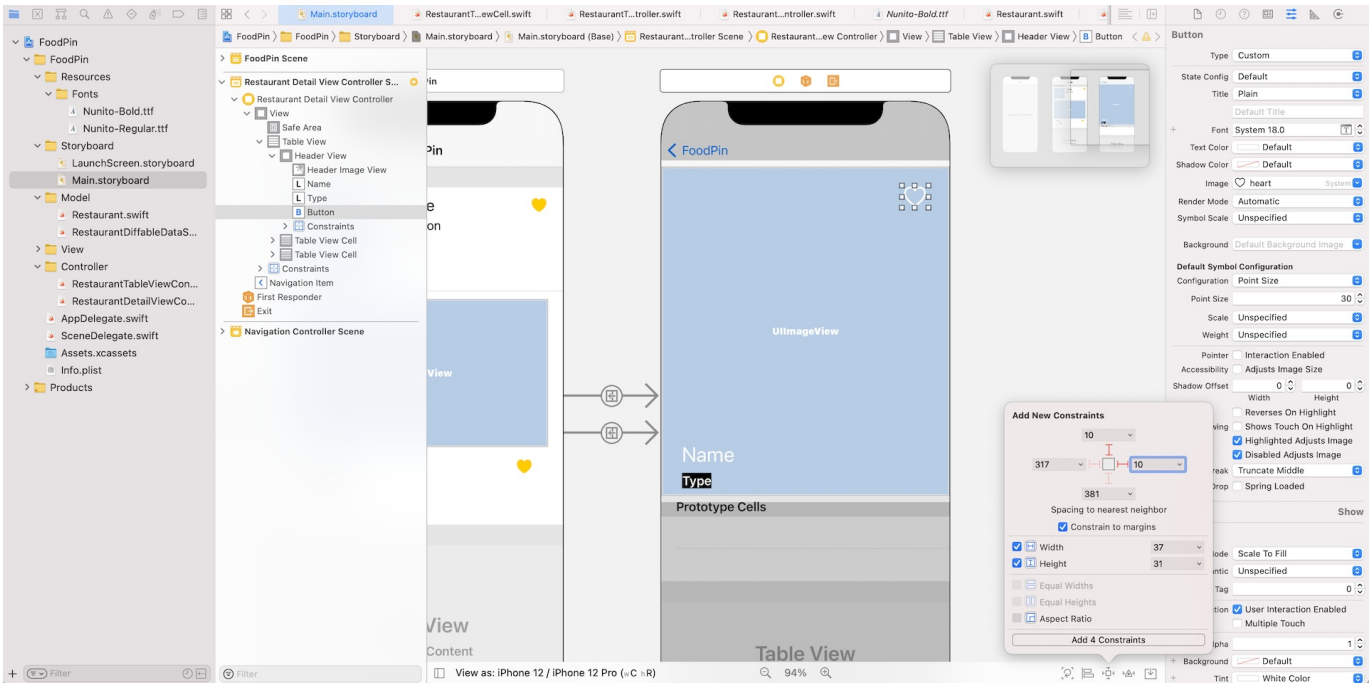


Figure 14-14. Adding layout constraints for the heart button

Next, let's add the layout constraints for the name and type labels. We will first embed both labels in a stack view. Select both labels in the document outline. Click the *Embed in* button and choose *Stack view* to embed both labels in a stack view. Once you created the stack view, go to the Attributes inspector and set the spacing value to `10` points.

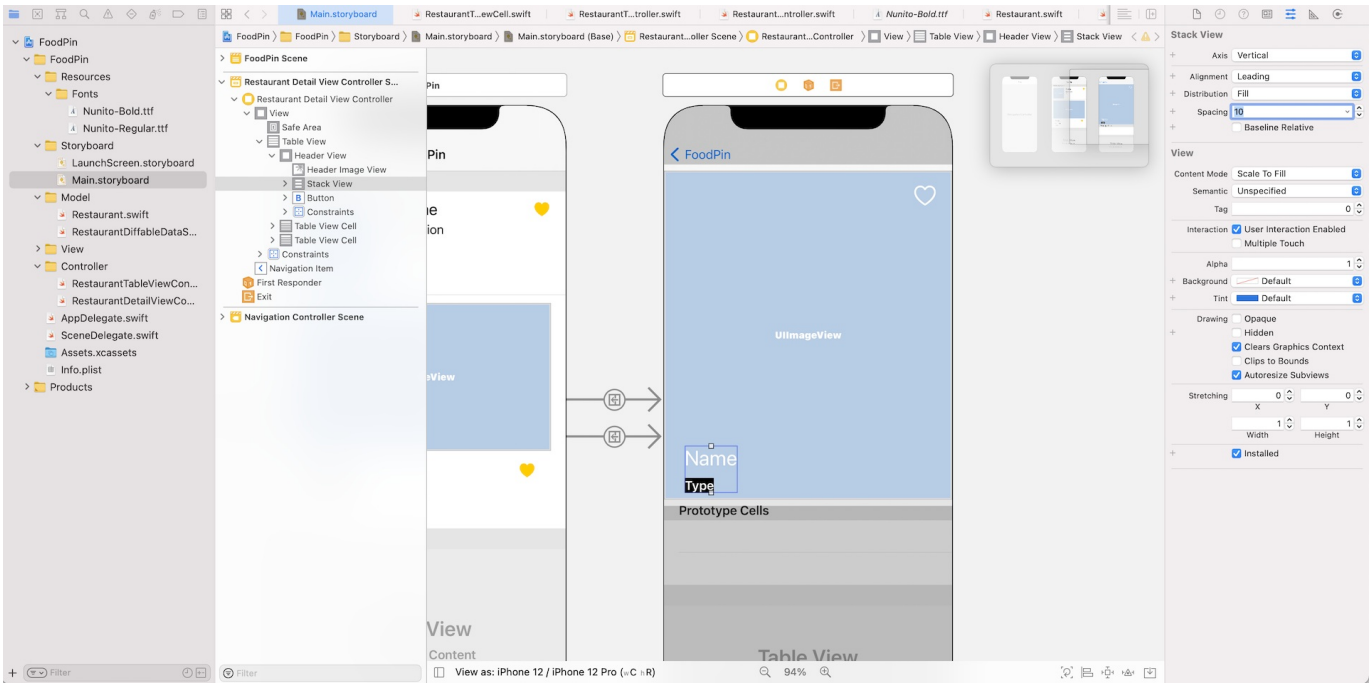


Figure 14-15. Embedding both labels in a stack view

Now it's for the layout requirement. I want to keep the stack view at the lower left corner of the header view, so we will add a few spacing constraints. Select the stack view and click the *Add new constraints* button. Set the left, right, and bottom value to `20`, `20`, and `150` points respectively. You are free to adjust these values if you want to change the position of the stack view.

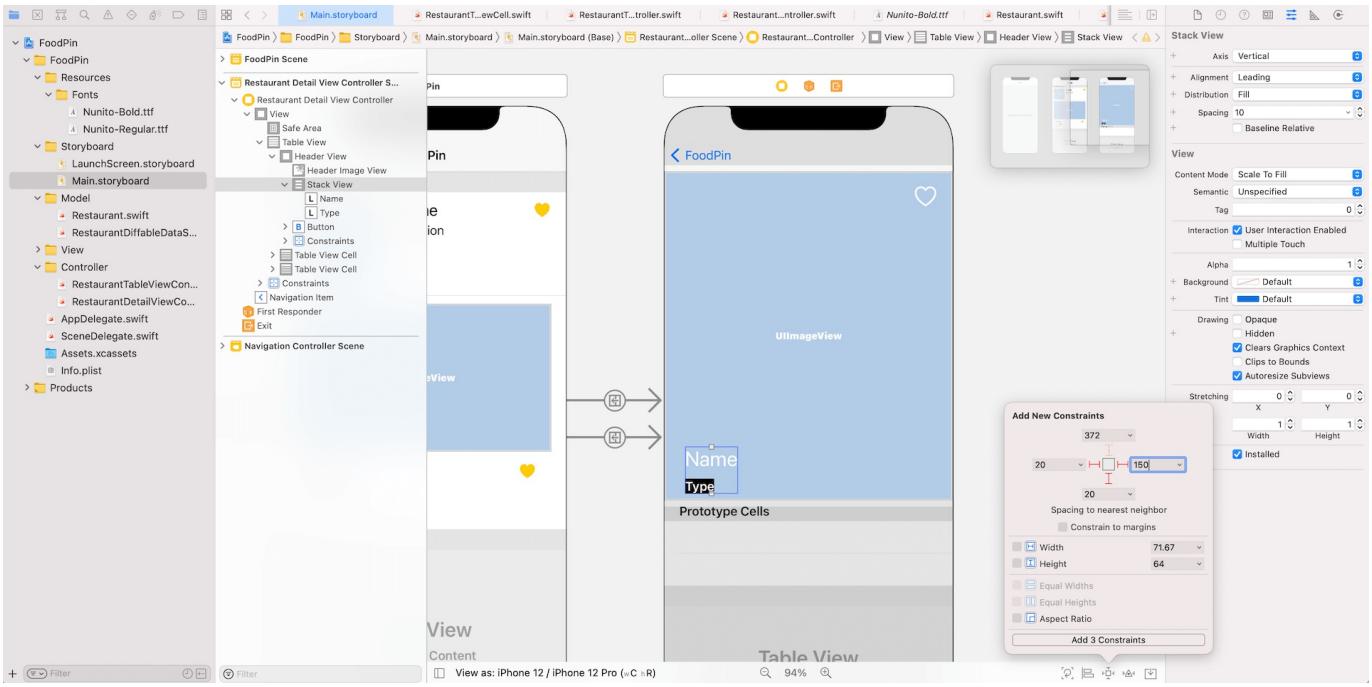


Figure 14-16. Adding spacing constraints for the stack view

That's it! We've completed the design of the header view. Now let's move onto the coding part. We will create a class for the header view. In project navigator, right-click the *View* folder and select *New File...*. Choose the *Cocoa Touch Class* template and hit *Next*. Set the class name to `RestaurantDetailHeaderView` and its subclass to `UIView`. Click *Next* to create the file.

Edit the `RestaurantDetailHeaderView.swift` file like below to add the required outlet variables:

```
import UIKit

class RestaurantDetailHeaderView: UIView {

    @IBOutlet var headerImageView: UIImageView!
    @IBOutlet var nameLabel: UILabel!
    @IBOutlet var typeLabel: UILabel!
    @IBOutlet var heartButton: UIButton!

}
```

Go back to Interface Builder and select the *Header View* in document outline. In Identity inspector, change the custom class to `RestaurantDetailHeaderView`. Next, establish a connection with each of the corresponding outlet variables.

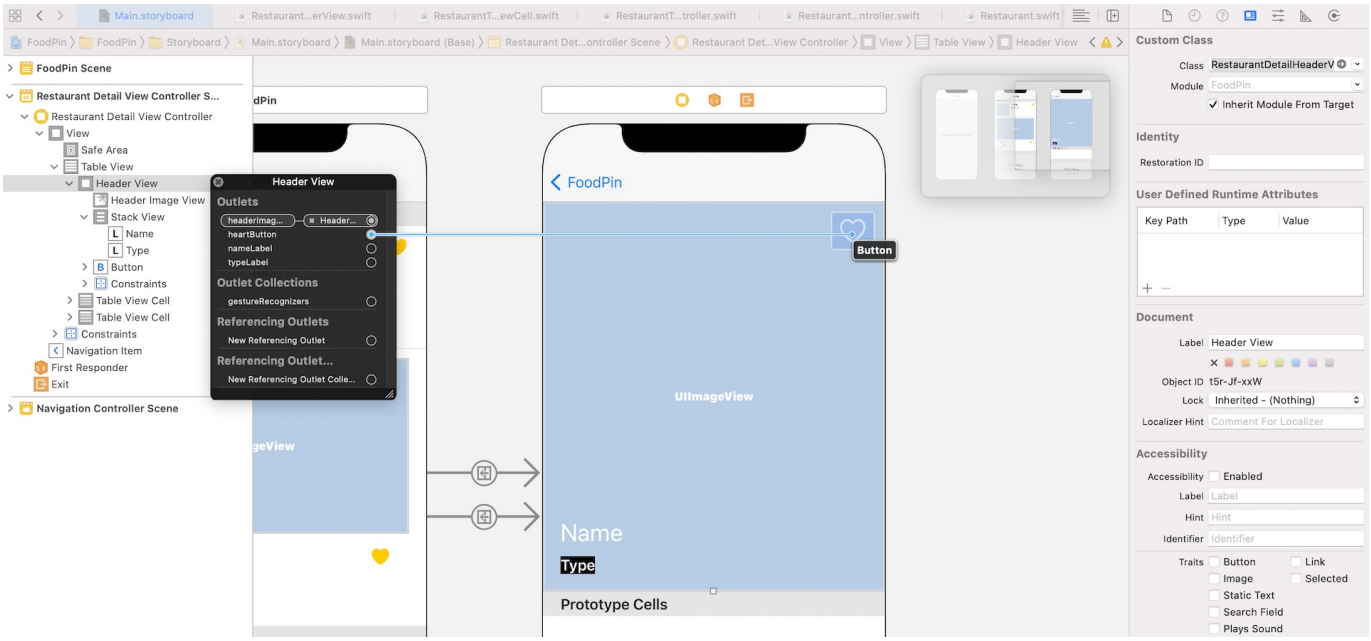


Figure 14-17. Establishing the connections

Now that the header view is ready for us to populate the restaurant data, it is time to modify `RestaurantDetailViewController.swift`. In the `RestaurantDetailViewController` class, insert two new outlet variables for the table view and the header view. Later, we need to modify their content in code. This is why we need to create these outlets:

```
@IBOutlet var tableView: UITableView!
@IBOutlet var headerView: RestaurantDetailHeaderView!
```

Next, update the `viewDidLoad` method with the following code:

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

    navigationController?.navigationBar.prefersLargeTitles = false

    // Configure header view
    headerView.nameLabel.text = restaurant.name
    headerView.typeLabel.text = restaurant.type
    headerView.headerImageView.image = UIImage(named: restaurant.image)

    let heartImage = restaurant.isFavorite ? "heart.fill" : "heart"
    headerView.heartButton.tintColor = restaurant.isFavorite ? .systemYell
ow : .white
    headerView.heartButton.setImage(UIImage(systemName: heartImage), for:
.normal)

}
```

The code is very straightforward. When the detail view is loaded, we set the restaurant data to the corresponding label or image view. We also check the `isFavorite` property of the selected restaurant and change the color of the heart icon. To make the code work, don't forget to connect the outlets with the

corresponding UI components. The `tableView` outlet should be connected with the *Table View* of the detail view co ntroller and the `headerView` outlet should be connected with *Header View*.

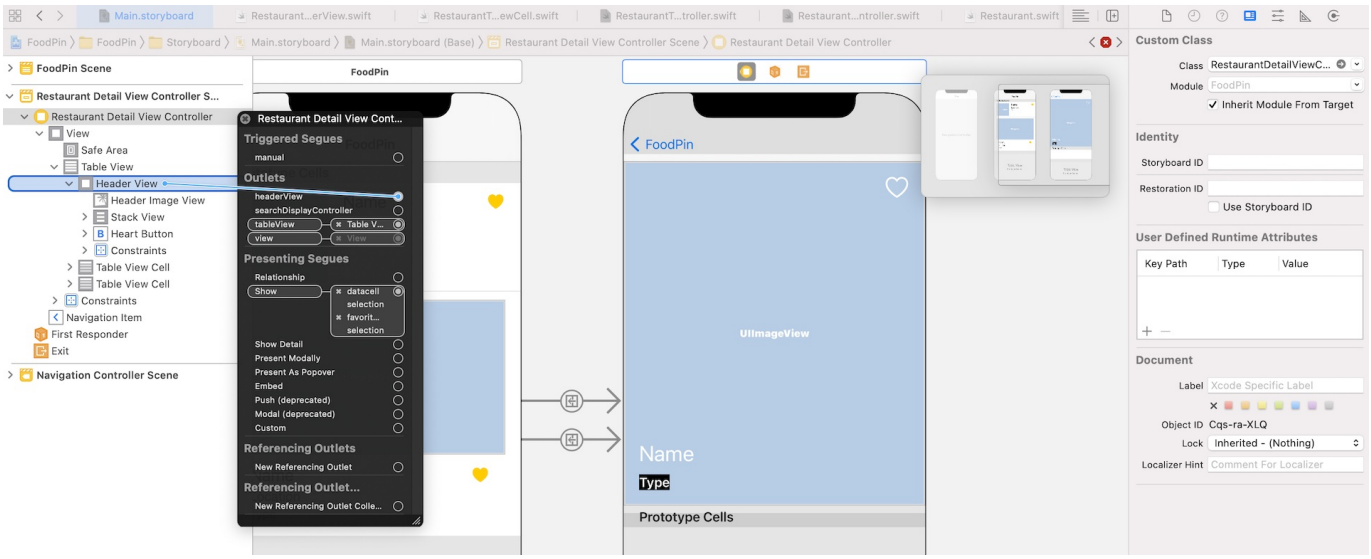


Figure 14-18. Establishing the connections for both header view and table view

After all the tedious procedures, it is now ready to test your app! Run the project and have a quick look. The image is now displayed in the table view's header (right above the rest of the table rows).

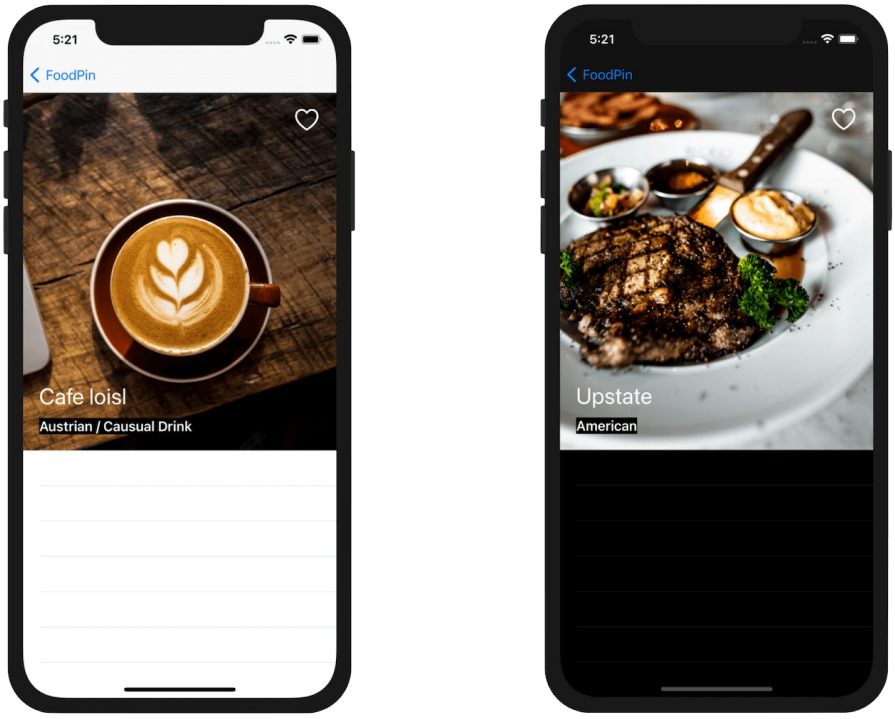


Figure 14-19. Displaying the restaurant in the header view

Understanding Image View Scaling

Earlier, I asked you to set the content mode of the image view to *Aspect Fill*. We haven't discussed how this content mode option affects the presentation of the image. Before we continue to build the detailed view, let me explain the usage of this option. Every view in iOS has a content mode. The `UIImageView` class is a subclass of `UIView`, so it also has a content mode property to specify how a view adjusts its content. Here are the three common scaling modes you usually use:

- Scale to Fill (default)
- Aspect Fit
- Aspect Fill

By default, the `UIImageView` object is set to *Scale To Fill* . In this mode, the image view scales its image to fit the view's size. The aspect ratio of the image may be changed due to the scaling. If you change the mode to *Aspect Fit*, the aspect ratio of the image will be maintained. However, it may leave some extra spaces on both sides of the image if the image view's aspect ratio is different from that of the image.



Figure 14-20. Image View Scaling Mode

The *Aspect Fill* mode is the best fit for our design, which is why we set the header image view to this mode. In this mode, the image view scales the image to fill the size of the view. The aspect ratio is maintained, though some parts of the image may extend over the edges of the image view. In this case, you will need to enable an option called *Clip to Bounds* to cut off the extra parts (see figure 14-20).

In Xcode 12, the *Clip to Bounds* option is enabled by default. This is why the *Aspect Fill* mode works out of the box. You can select the image view and check the *Clips to Bounds* setting in the Attributes inspector.

Fixing a Couple of Issues

The header view looks great, right? However, there are a couple issues we need to fix:

1. For some lengthy restaurant names, they cannot be fully displayed.
2. For some restaurant images with a lot of white backgrounds, the restaurant name may be clearly shown.

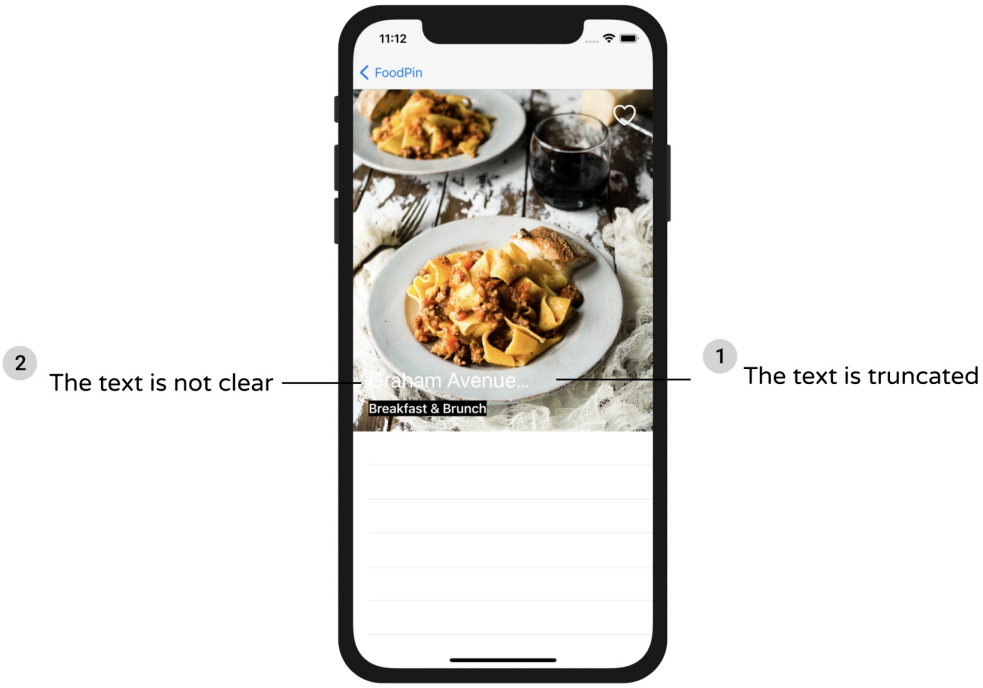


Figure 14-21. A couple of the issues with the current header view

Truncated Restaurant Names

The first issue is related to the restaurant name. If the name is too long, the label will truncate the text. By default, the label is set to display text in a single line. You can select the label and then reveal the *Lines* option in the Attributes inspector. The current value should be set to `1`.

Labels in iOS are capable to display multiple lines. All you need is to change the value to `0`. In this case, the label will automatically determine the total number of lines to fit its content. While you can change the value in Interface Builder, I prefer to customize the property of the label in code. Open `RestaurantDetailHeaderView.swift` and edit the `nameLabel` variable to update the `numberOfLines` property using `didSet`:

```
@IBOutlet var nameLabel: UILabel! {
    didSet {
        nameLabel.numberOfLines = 0
    }
}
```

Dimming the Image Using UIView

Now let's fix the second issue. Since the restaurant name label is in white, it may not be shown clearly if the restaurant image has a lot of white in it. How can we solve this issue?

One way is to add an overlay to dim the image. To do that, go to `Main.storyboard`. Open the Object library, drag a view and place it right below the header image view in document outline. Name it *Dim View*.

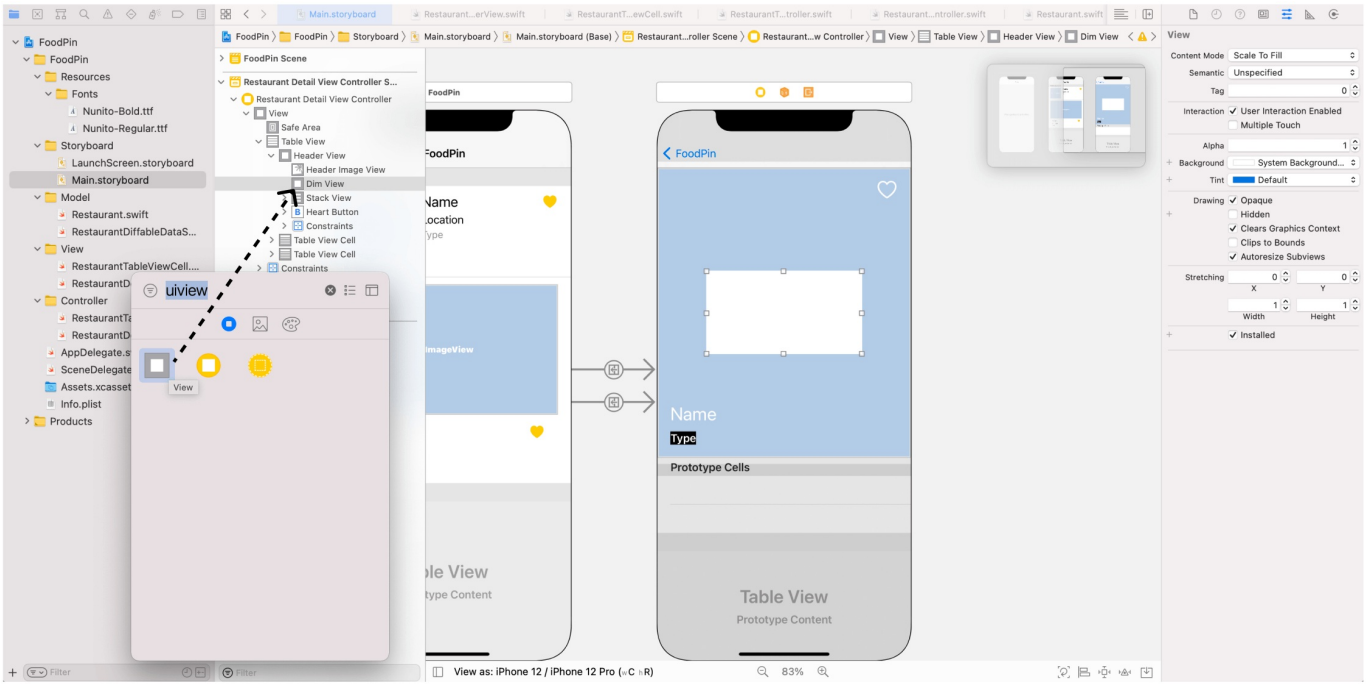


Figure 14-22. Adding a new view

Resize the view to make it the same size as the header view. Then go to the Attributes inspector. Set the background color to `black` and alpha value to `0.2`. The alpha value controls the transparency of the color. It goes from 0 to 1 where 0 is completely transparent and 1 fully opaque. We set the value to 0.2 so that it is partially transparent, and achieve a dimming effect. If you don't know how to change the opacity value, you can choose *Custom* in the *Background* option. Then you will see a Colors dialog. Choose the *Color Sliders* tab and set to *RGB Sliders*. You can then choose the opacity.

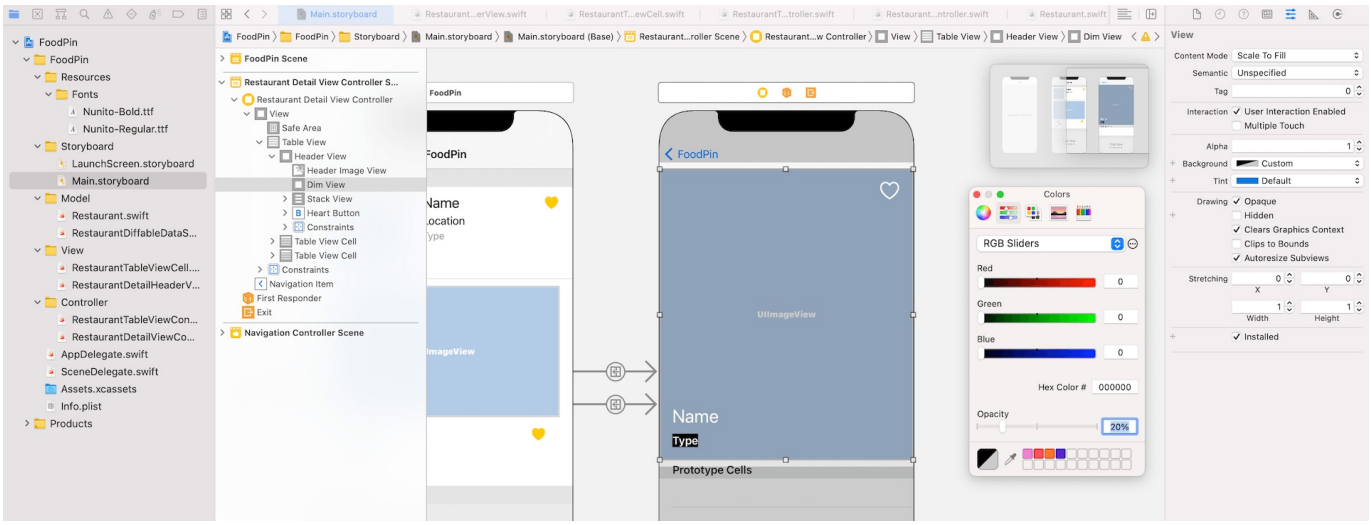


Figure 14-23. Changing the background color and setting the opacity

Lastly, don't forget to define the layout constraint for the view. Select the Dim View and click the *Add new constraints* button. Set the spacing of the top, left, right and bottom side to `0`. Make sure the bars are in solid red and the *Constrain to margins* option is unchecked. Then hit the *Add 4 Constraints* button to add the constraints.

Now it is time to test the header view again. If you got everything correct, the header view should have fixed the mentioned issues. With the *Dim View*, we increase the color contrast of the image and the text. The restaurant name is now clearly shown even in light background. And, the name label automatically wraps the text onto a new line.

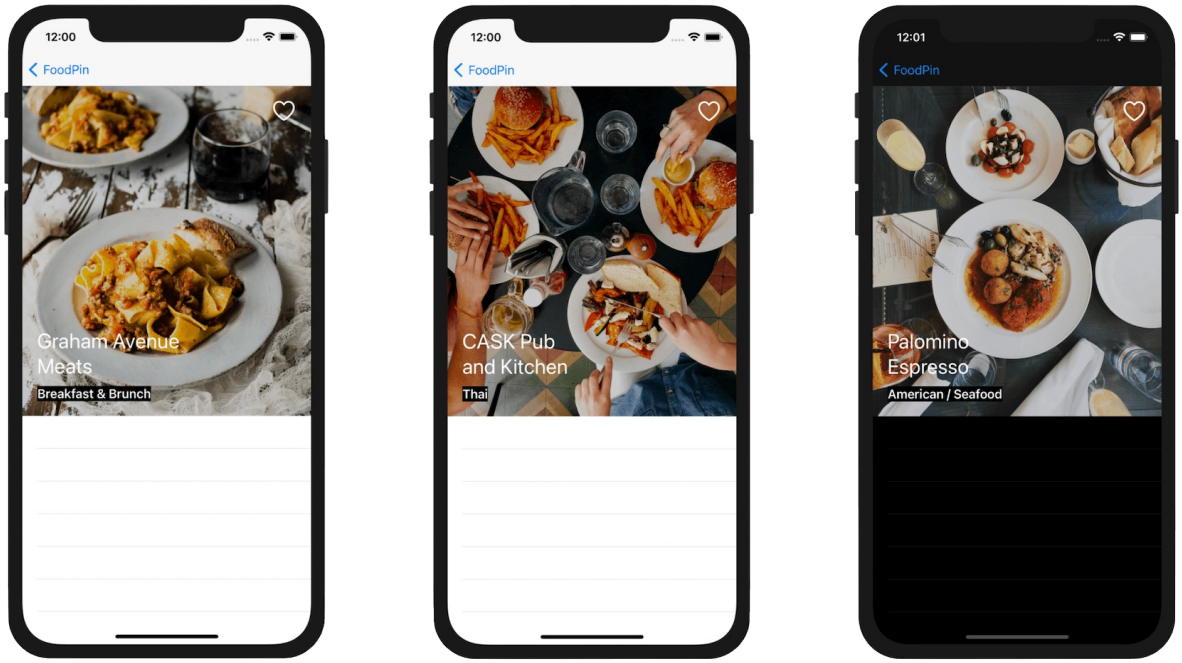


Figure 14-24. The image is now dimmed

Using Custom Fonts for Dynamic Types

Do you still remember the fonts files we imported at the very beginning of the chapter? We haven't used them yet. Right now, all the labels are configured to use text styles. For example, the name label uses the *Title 1* text style. Without any customizations, iOS uses San Francisco as the default font.

To use a custom font for text style (or Dynamic Types), we first have to configure those font files in `Info.plist`, which is the configuration of the Xcode project. In the project navigator, select `Info.plist` and right-click any blank area to open the context menu. Select *Add row* to create a new row in the file.

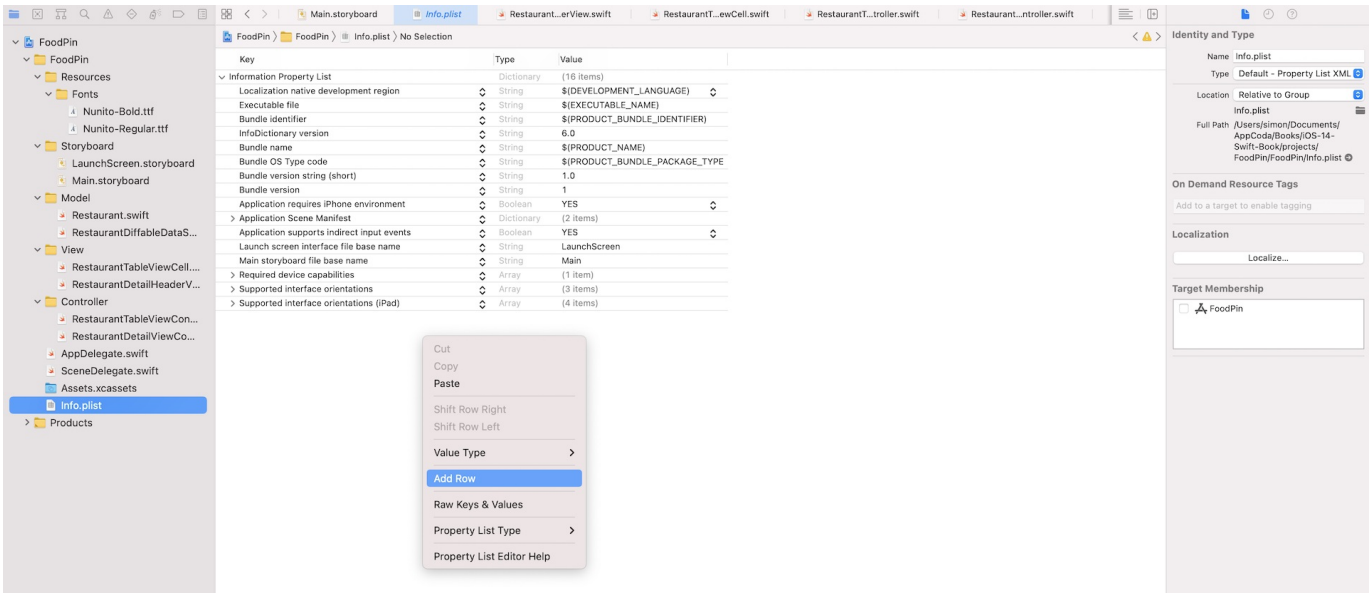


Figure 14-25. Editing the Info.plist file

Next, type *Fonts provided by application* in the Key field. In order to register the font files, we need to add the key and specify all the font files. Once you add the key, click the disclosure button to expand it. For item 0, set its value to `Nunito-Regular.ttf`. You can click the + button of item 0 and add another item. For item 1, set the value to `Nunito-Bold.ttf`.

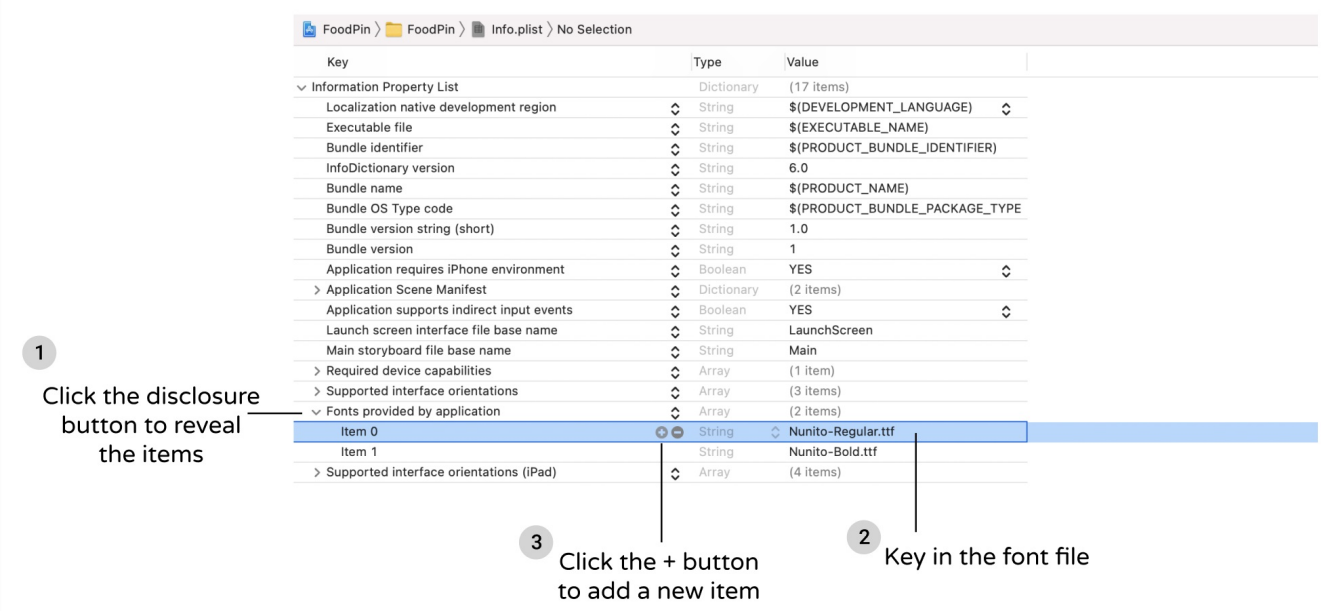


Figure 14-26. Specifying the font files in Info.plist

Now you are ready to use the fonts in Interface Builder or using code. To change the font of a specified text style, you can write the code like this:

```
if let customFont = UIFont(name: "Nunito-Bold", size: 40.0) {
    nameLabel.font = UIFontMetrics(forTextStyle: .title1).scaledFont(for:
customFont)
}
```

The `UIFont` class lets you load the font file and provides you with access to the font's characteristics. You use a `UIFontMetrics` object to support scalable custom fonts in your app. In the code above, we apply the `Nunito-Bold` font to the *Title1* text style. If you want to set the custom font for another text style, just pass the `forTextStyle` parameter with another value.

Now modify both `nameLabel` and `typeLabel` in the `RestaurantDetailHeaderView` class like this:

```
@IBOutlet var nameLabel: UILabel! {
    didSet {
        nameLabel.numberOfLines = 0

        if let customFont = UIFont(name: "Nunito-Bold", size: 40.0) {
            nameLabel.font = UIFontMetrics(forTextStyle: .title1).scaledFo
nt(for: customFont)
        }
    }
}
@IBOutlet var typeLabel: UILabel! {
    didSet {
        if let customFont = UIFont(name: "Nunito-Bold", size: 20.0) {
            typeLabel.font = UIFontMetrics(forTextStyle: .headline).scaled
Font(for: customFont)
        }
    }
}
```

We tell the system to use our own font for both name and type labels. Once you made the change, you can run the app again. In the detailed view, the app should now use the custom font.



Figure 14-27. Using the custom font for both labels

Designing the Prototype Cells

Now that we've completed the implementation of the header view, let's move to the implementation of the table view cells. If you read the book from the very beginning, you should know how to customize the prototype cell. To display the restaurant information, we will create two prototype cells to support two different types of cell layout. One prototype cell is designed to display the multiline text. The other cell is for displaying two column data.

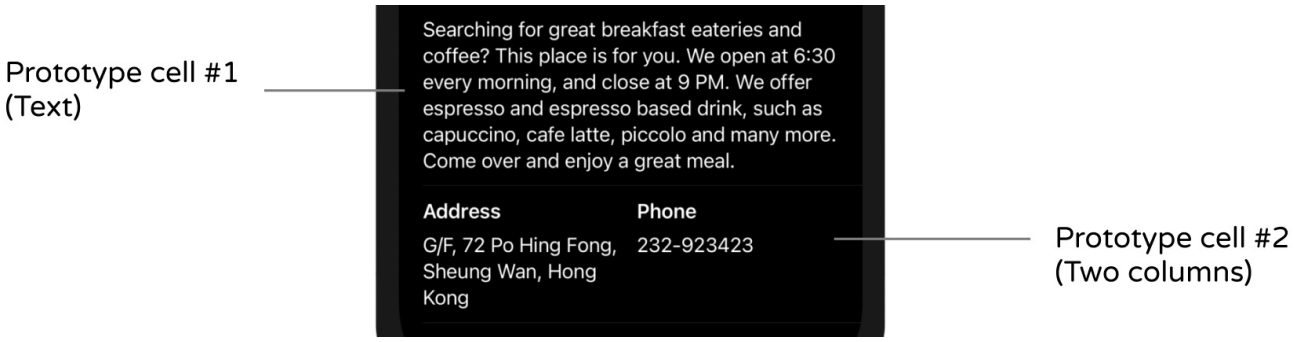


Figure 14-28. Creating two prototype cells

RestaurantDetailTextCell

Now open `Main.storyboard` and drag a label object to the first prototype cell. Double click the label and change its title to *Description*. Go to the Attributes inspector and change the font to the *Body* text style. Then click the *Add new constraints* button to add four spacing constraints. Set the spacing value to `0` for all sides and make sure you check the *Constrain to margins* option.

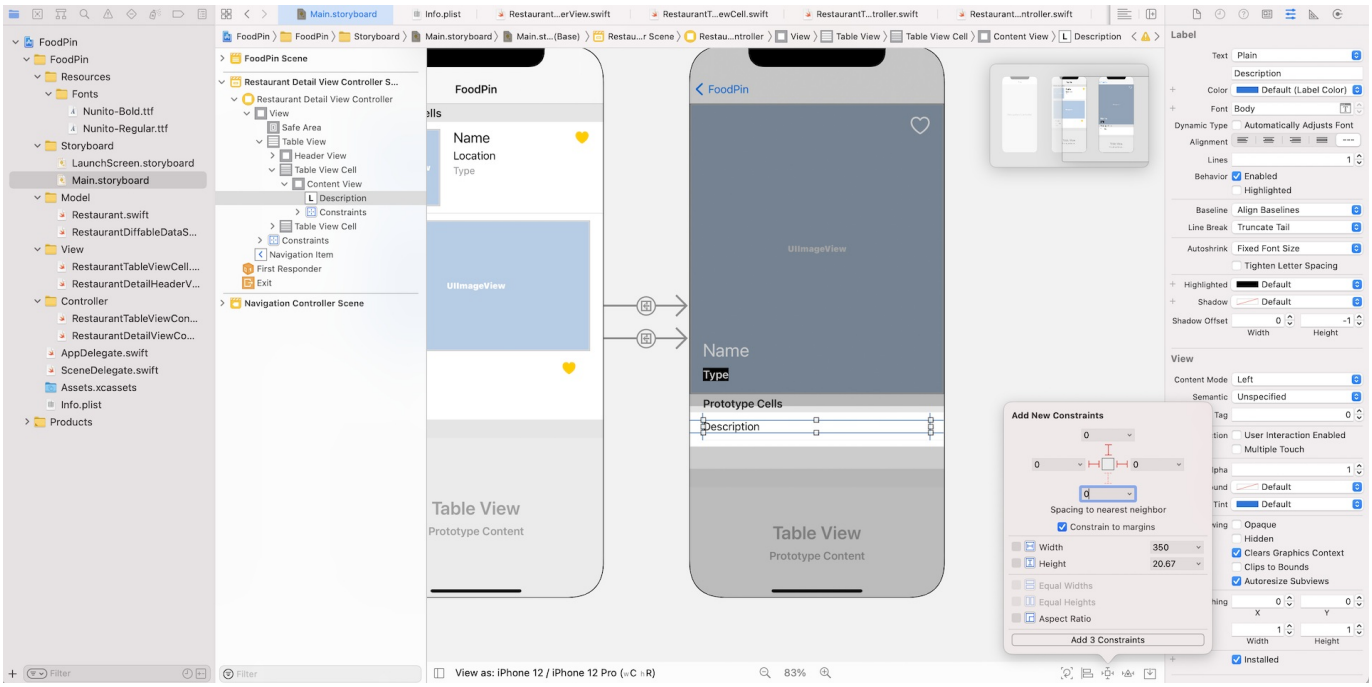


Figure 14-29. Designing the first prototype cell with a label

For each prototype cell, we will create a corresponding Swift file to connect with it. In project navigator, right click *View* folder and choose *New file*.... Select the *Cocoa Touch Class* template. In the next screen, name the class `RestaurantDetailTextCell` and set it as a subclass of `UITableViewCell`. Click *Next* to save the file.

Similarly to what we have done before, we will create an outlet variables for the connections of the label. Declare the following variables in the `RestaurantDetailTextCell.swift` file:

```
@IBOutlet var descriptionLabel: UILabel! {
    didSet {
        descriptionLabel.numberOfLines = 0
    }
}
```

You should understand the code above. We just create an outlet for the text label. Since the text is going to extend over multiple lines, we set the label's `numberOfLines` property to `0`.

Now go back to Interface Builder and select the prototype cell we just designed. We haven't set the cell identifier and custom class. So, go to the Attributes inspector and set the identifier to `RestaurantDetailTextCell`.

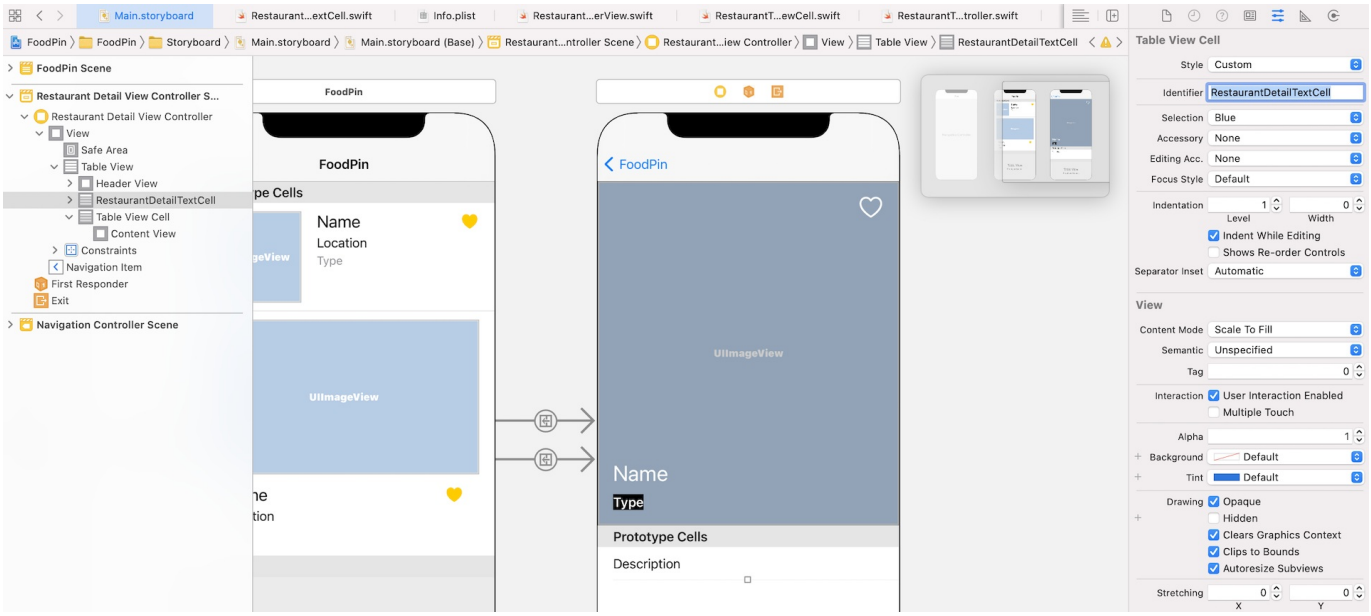


Figure 14-30. Setting the cell's identifier

To set the custom class, go to the Identity inspector and change the custom class to `RestaurantDetailTextCell`.

Did you notice that the identifier and the class name are the same? I will explain why I did that in a while. Now, right click `RestaurantDetailTextCell` in the document outline. Establish the connections between the description label and the outlet variable we defined earlier.

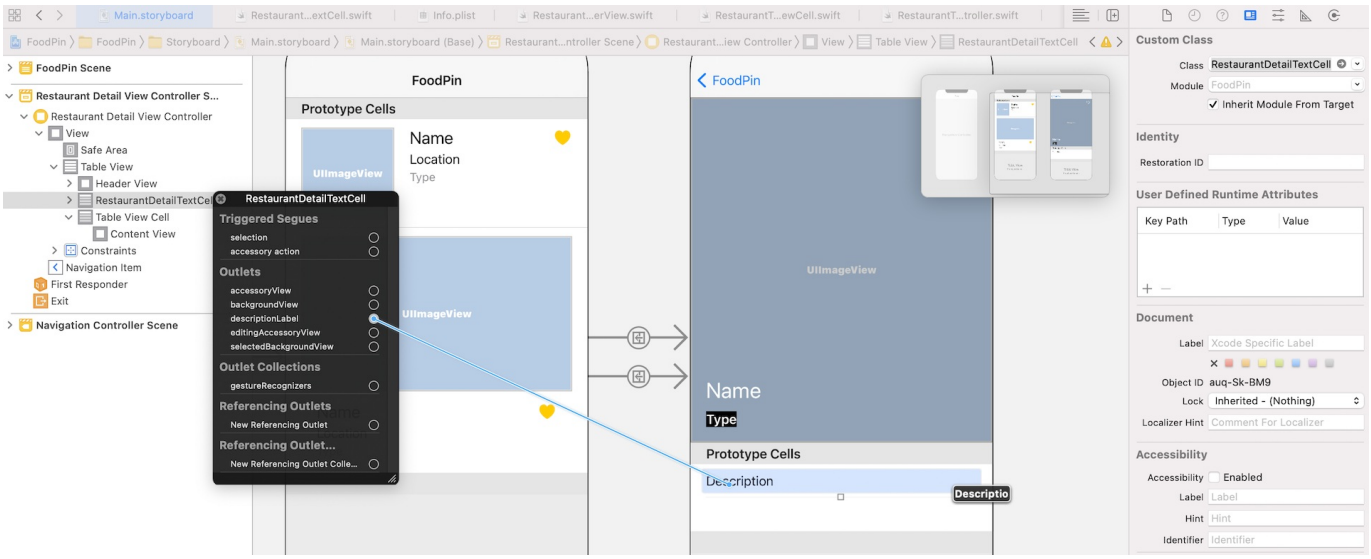


Figure 14-31. Setting the cell's identifier

RestaurantDetailTwoColumnCell

Now we will move onto the implementation of the second prototype cell. This cell is designed to display two columns of data. Obviously, we will make use of stack views to create the layout.

First, make the second prototype cell a little bit taller (say, 140 points). This would let us easier fill the UI components in the cell. Now drag a label to the cell and change the text to `Address`. In the Attributes inspector, set its font style to `Headline`. Next, drag another label to the cell and place it right below the *Address* label. Name the label `Full Address` and set its font style to `Body`.

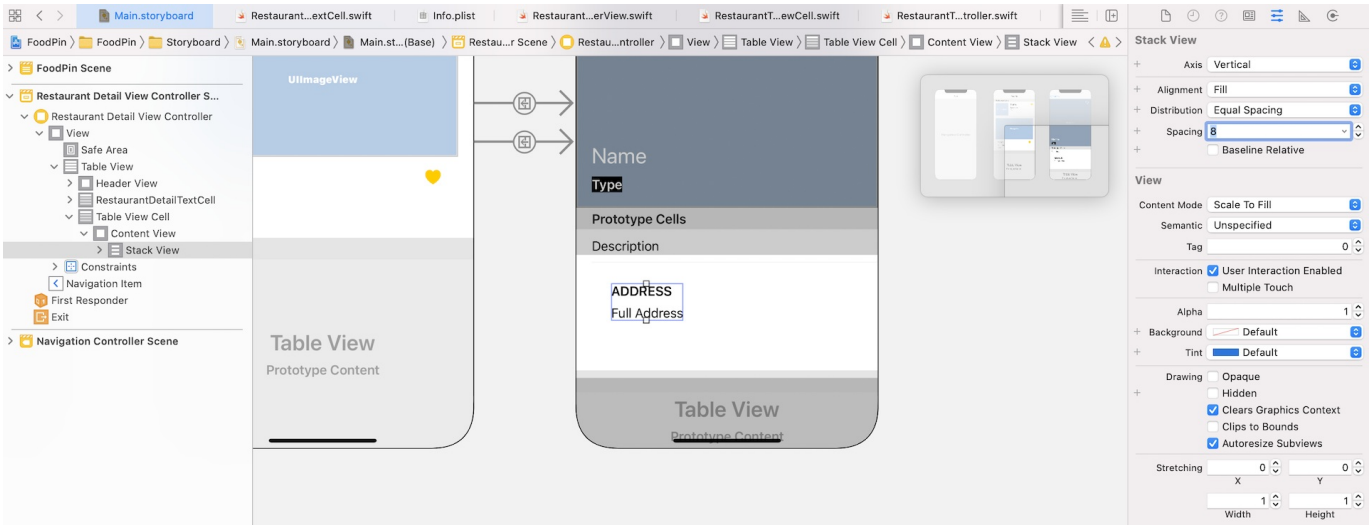


Figure 14-32. Adding two labels to the second prototype cell

Once you configured the two labels, hold command key and select both items. Click *Embed in* and choose *Stack view* to embed both labels in a vertical stack view (see figure 14-32). In the Attributes inspector, change the stack view's distribution option to `Equal Spacing` and its spacing value to `8`.

We will duplicate the stack view to create the second column. You can use copy-and-paste to duplicate the view. Alternatively, you can hold the *Option* key and drag the stack view to duplicate one. For the new stack view, change both labels to `Phone` and `Phone Number` respectively.

To arrange both stack views side by side, select both stack views and click *Embed in*. Choose stack view to embed them in a horizontal stack view. For the new stack view, set its alignment to `Top` because we want to align the text to the top of the view. Also, change the distribution option to `Fill Equally` and set its spacing to `10`. Your stack view should look similar to the one shown in figure 14-33.

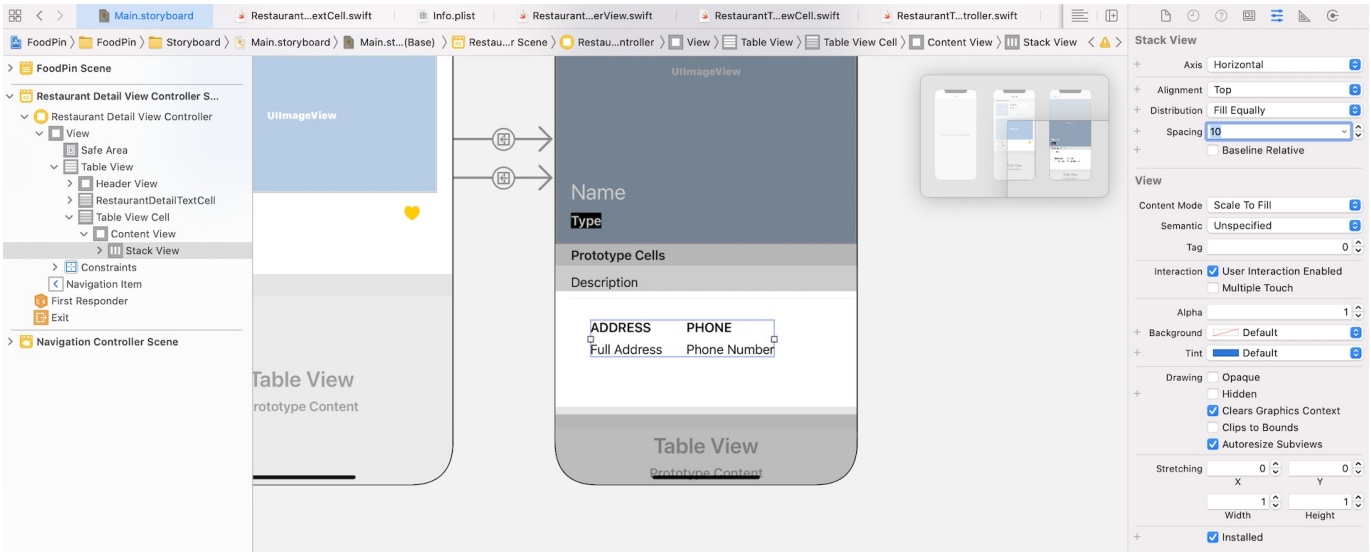


Figure 14-33. Embed the two stack views into a horizontal stack view

Now, we will add a few spacing constraints for the stack view. Select the new stack view and click *Add new constraints*. Set the spacing value for all sides to `0` and tick the *Constrain to margins* option.

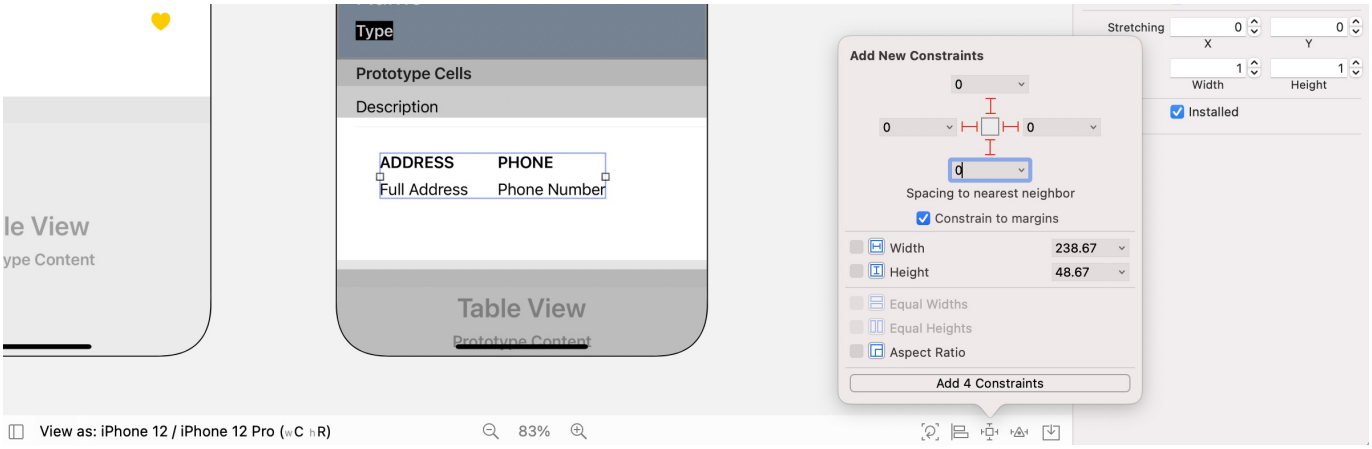


Figure 14-34. Adding spacing constraints

Currently, the height of the cell is fixed. To make the cell adaptable to variable height, select the table view cell in document outline. Go to the Size inspector and tick the *Automatic* checkbox.

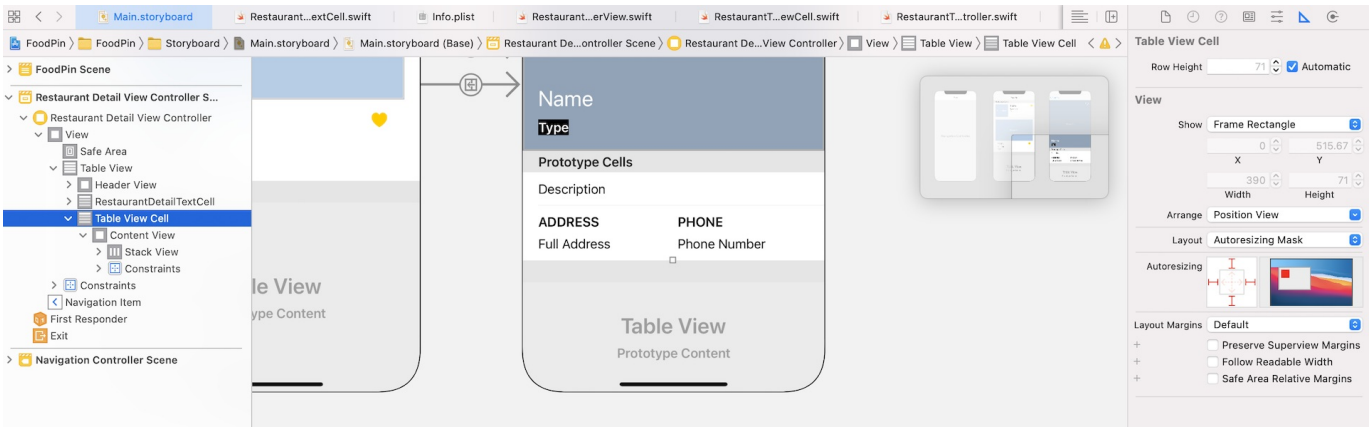


Figure 14-35. Set the cell's size to automatic

Now that we have completed the design of the second prototype cell, the next step is to create a Swift file to pair with this cell. In project navigator, right click the *View* folder and choose *New File...*. Choose the *Cocoa Touch Class* template and click the *Next* button. Name the class `RestaurantDetailTwoColumnCell` and set its subclass to `UITableViewCell`.

Once created, open the `RestaurantDetailTwoColumnCell.swift` file. Declare the outlet variables for the labels:

```
@IBOutlet var column1TitleLabel: UILabel! {
    didSet {
        column1TitleLabel.text = column1TitleLabel.text?.uppercased()
        column1TitleLabel.numberOfLines = 0
    }
}
@IBOutlet var column1TextLabel: UILabel! {
    didSet {
        column1TextLabel.numberOfLines = 0
    }
}
@IBOutlet var column2TitleLabel: UILabel! {
    didSet {
        column2TitleLabel.text = column2TitleLabel.text?.uppercased()
        column2TitleLabel.numberOfLines = 0
    }
}
@IBOutlet var column2TextLabel: UILabel! {
    didSet {
        column2TextLabel.numberOfLines = 0
    }
}
```

I tried to make this prototype cell as flexible as possible, so all the four labels can be configurable using code. They are not limited to display address and phone number. It is actually suitable to display any text in two column layout.

Again, we set the `numberOfLines` property to `0` in order to display multiple lines. For the title labels, we have an additional line of code to convert the text to uppercase.

Go back to `Main.storyboard` and select the second prototype cell. Go to the Attribute inspector and set the cell identifier to `RestaurantDetailTwoColumnCell`. In the Identity inspector, set the custom class to `RestaurantDetailTwoColumnCell`.

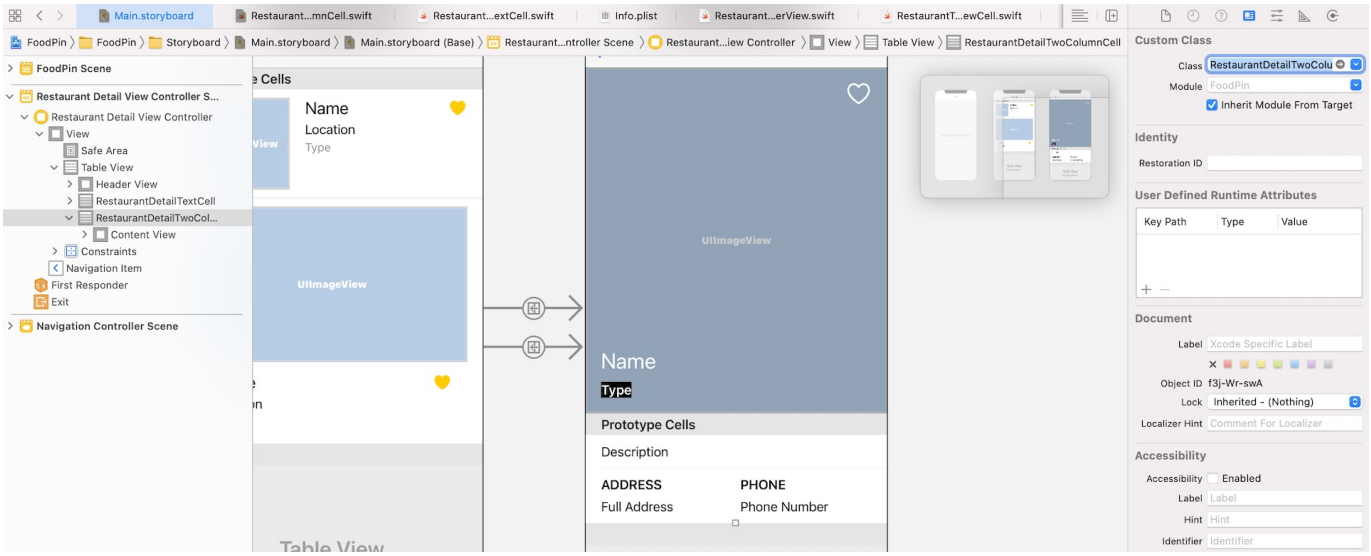


Figure 14-36. Setting the custom class of the 2-column prototype cell

Finally, establish a connection between the labels and the outlet variables. The `column1TitleLabel` should be connected to the *Address* label and the `column1TextLabel` should be connected to *Full Address*.

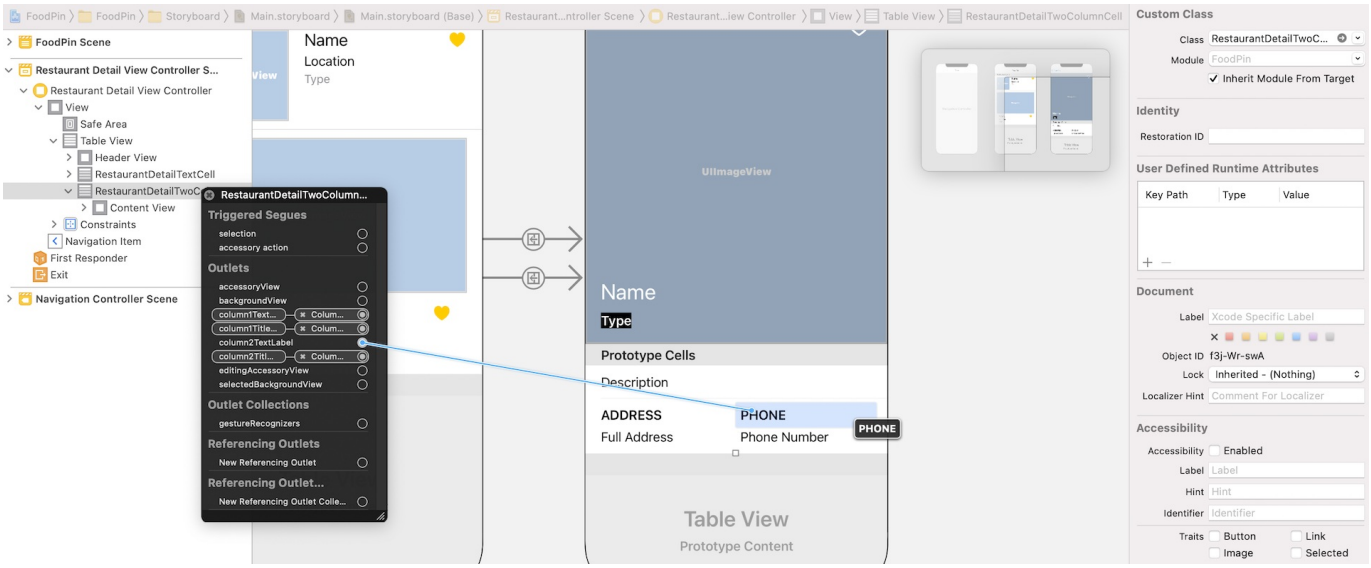


Figure 14-37. Setting the custom class of the 2-column prototype cell

Updating the RestaurantDetailViewController Class

Great! You will finished the design of the prototype cells. Let's update the `RestaurantDetailViewController` class to populate the restaurant information. The `RestaurantDetailViewController` class is a subclass of `UIViewController`. As you've learned in chapter 8, we have to adopt both the `UITableViewDataSource` and `UITableViewDelegate` protocols for displaying content in the table view.

In the earlier chapter, we learned to adopt the protocols with the following syntax:

```
class RestaurantDetailViewController: UIViewController, UITableViewDataSou
rce, UITableViewDelegate {
```

You append the required protocols after the super class. This is one of the ways to adopt protocols in Swift. Let me take this chance to show you another approach by using a Swift feature known as Extensions. Extensions allow you to extend the functionalities of an existing class and structure. To create an extension, you start with the keyword `extension` followed by the class you want to extend. For example, to adopt the `UITableViewDataSource` and `UITableViewDelegate` protocols in the `RestaurantDetailViewController` class, you can create an extension like this:

```
extension RestaurantDetailViewController: UITableViewDataSource, UITableVi
ewDelegate {
    .
    .
    .
}
```

Now let's open the `RestaurantDetailViewController.swift` file and write the extension. Insert the following code at the end of the file:

```
extension RestaurantDetailViewController: UITableViewDataSource, UITableVi
ewDelegate {

    func tableView(_ tableView: UITableView, numberOfRowsInSection section
: Int) -> Int {
        return 2
    }

    func tableView(_ tableView: UITableView, cellForRowAt indexPath: Index
Path) -> UITableViewCell {
        switch indexPath.row {
        case 0:
            let cell = tableView.dequeueReusableCell(withIdentifier: String
(describing: RestaurantDetailTextCell.self), for: indexPath) as! Restauran
tDetailTextCell

            cell.descriptionLabel.text = restaurant.description

            return cell

        case 1:
            let cell = tableView.dequeueReusableCell(withIdentifier: String
(describing: RestaurantDetailTwoColumnCell.self), for: indexPath) as! Rest
aurantDetailTwoColumnCell

            cell.column1TitleLabel.text = "Address"
            cell.column1TextLabel.text = restaurant.location
            cell.column2TitleLabel.text = "Phone"
            cell.column2TextLabel.text = restaurant.phone

            return cell

        default:
            fatalError("Failed to instantiate the table view cell for deta
il view controller")

        }
    }
```

```
}
```

We created an extension for `RestaurantDetailViewController`, adopted the protocols, and implemented the required method for populating the table view's data. The detail view has two rows of data. This is why we return `2` in the `tableView(_:numberOfRowsInSection:)` method.

The implementation of the `tableView(_:cellForRowAt:)` method is a bit different from what you learned before because we now have to cater for two different types of prototype cells. The first cell uses `RestaurantDetailTextCell`, while the second one uses `RestaurantDetailTwoColumnCell`.

In the code above, we use `switch` to control the program flow and execute different branches of code depending on the value of `indexPath.row`. For the first cell (i.e. `indexPath.row` is 0), we dequeue the reusable cell with the identifier `RestaurantDetailTextCell` and then populate the restaurant description.

Recall that we set the cell's identifier to its custom class name, you can explicity specify the identifier and dequeue the cell like this:

```
let cell = tableView.dequeueReusableCell(withIdentifier: "RestaurantDetail
TextCell", for: indexPath) as! RestaurantDetailIconTextCell
```

The result of the code above is exactly the same as the one we wrote earlier. The code `String(describing: RestaurantDetailTextCell.self)` will return you the name of the class (i.e. `RestaurantDetailTextCell`).

```
let cell = tableView.dequeueReusableCell(withIdentifier: String(describing
: RestaurantDetailTextCell.self), for: indexPath) as! RestaurantDetailText
Cell
```

So, why do we use the latter version? Hardcoding the cell's identifier is prone to typos. Xcode will not show you any errors even if you type the identifier wrongly. You can only reveal the error when running the app.

By using the latter version, if we wrongly type the class name (say, `RestaurantDetailTxtCell.self`), Xcode will immediately prompt you the error.

For case `1`, the code is very similar but we use the `RestaurantDetailTwoColumnCell` for displaying the address and phone number.

The case `default` will be executed only if the value of `indexPath.row` doesn't match 0 or 1. In this case, we give a fatal error.

Okay, the code of the `RestaurantDetailViewController` class is almost ready. There is only one thing left. We haven't established the connection with the table view in the storyboard.

In earlier chapters, I showed you how to do it via Interface Builder (refer to chapter 8). This time let's do it using code. In the `viewDidLoad` method of `RestaurantDetailViewController`, insert the following lines of code:

```
tableView.delegate = self
tableView.dataSource = self
```

That's it. We set the table view's delegate and dataSource to `self`.

Ready to Test

Hit the Run button and test your app. The detail view now displays more information about the selected restaurant.

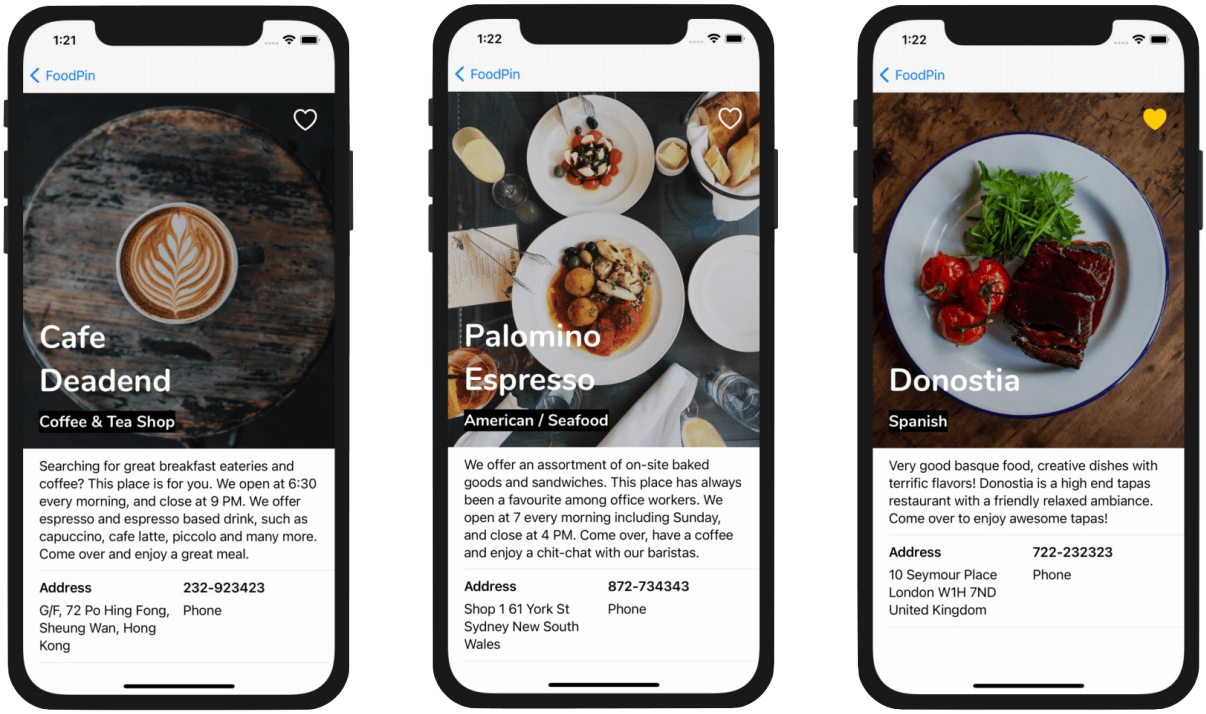


Figure 14-38. Detail view of some sample restaurants

Customizing the Table View Separator

The table view is now pretty nice. If you want to remove the separator of the table, you can insert the following line in the `viewDidLoad` method:

```
tableView.separatorStyle = .none
```

By setting the style of the separator to `.none`, you can remove all the separators.

Understanding Self Sizing Cells

Before we end this chapter, let me talk about the concept of self sizing cells. Have you ever wondered why the cell's height is automatically adjusted? When the description / location has more text, the height of the cell will be increased accordingly.

Prior to iOS 8, if you want to display dynamic content in a table view with multiple rows, you have to calculate the row height of a cell manually. Later in iOS 8, Apple introduced a new feature for `UITableView` known as *Self Sizing Cells*.

In brief, to enable self sizing of a table view cell, you have to implement a few things:

1. Define auto layout constraints for the prototype cell.
2. Specify the `estimatedRowHeight` property of the table view.
3. Set the `rowHeight` property of the table view to `UITableViewAutomaticDimension`.

Before iOS 11, you have to manually handle point 2 and 3. Now Xcode sets both the estimated row height and row height to automatic by default (see figure 14-35). Therefore, to use self sizing cells, all you need to do is just to define the layout constraints.

As long as you define the proper layout constraints, the cell will be resized automatically to fit its content.

Summary

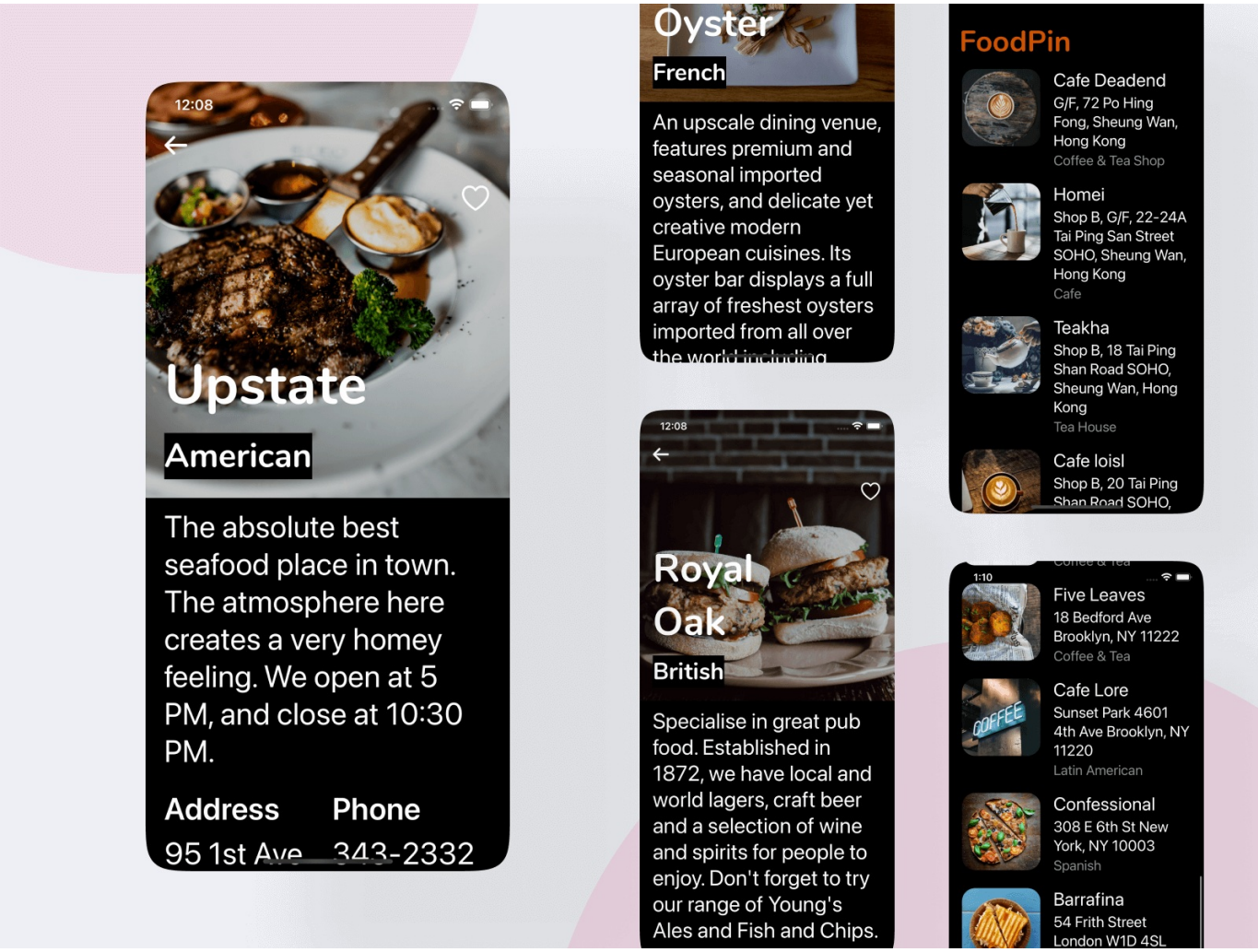
Great! You've made it. I hope you love it and the app you created. You've already built a quite polished app. It's not a complex app but you've managed to use some common components such as table view and navigation controller in iOS. What's more is that you've learned how to work with custom fonts and create a detail screen using a custom table view.

I also introduced a popular feature in Swift called Extensions. This is a very powerful feature that you will utilize it in your app development. To learn more about Swift Extensions, you can check out this document (https://docs.swift.org/swift-book/LanguageGuide/Extensions.html).

This is a huge chapter. Even if you can't wait to proceed to the next chapter, I suggest taking a break here. It needs time to let yourself digest all the materials we discussed so far. Grab a coffee or whatever drinks you like and relax.

For reference, you can download the complete project from http://www.appcoda.com/resources/swift53/FoodPinDetailView.zip.

Chapter 15
Navigation Bar Customization, Dark Mode, 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.

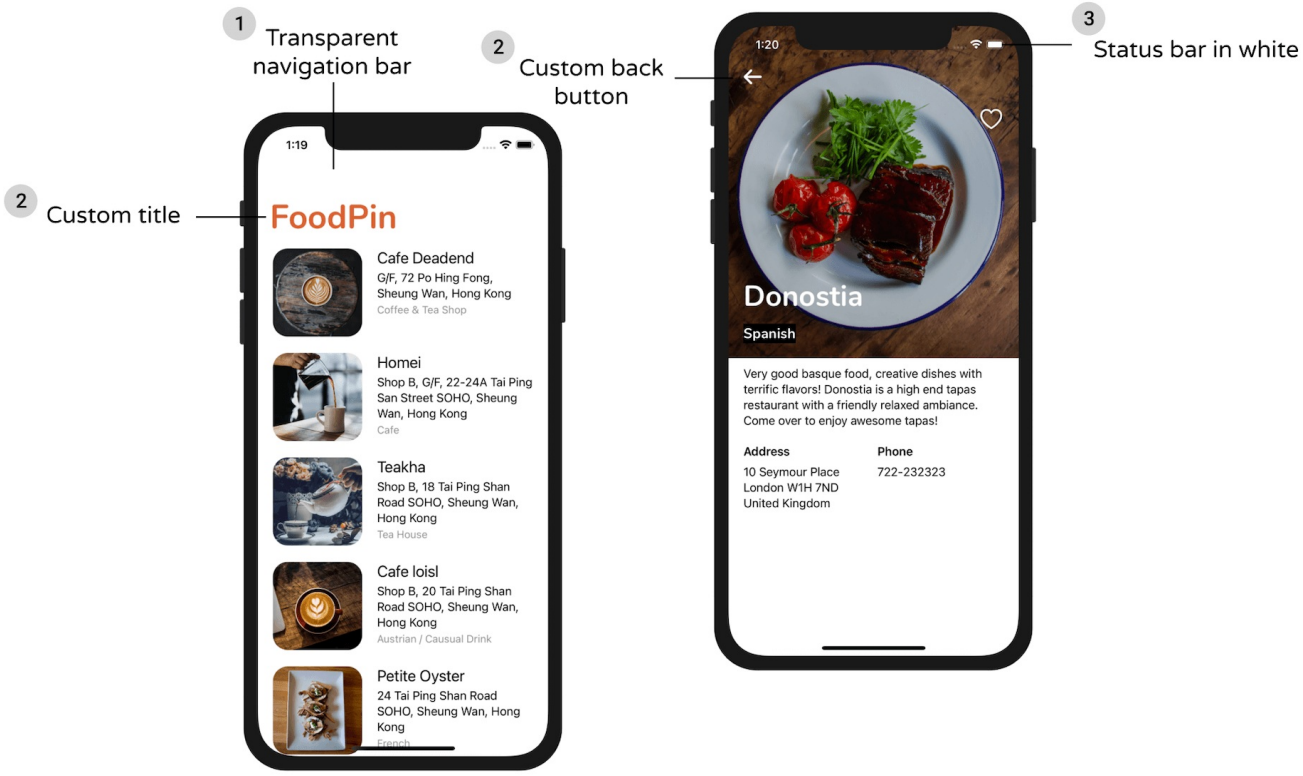


Figure 15-1. Customizing the navigation bar and status bar

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 and make its color adaptable to different interface style (light/dark mode).
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.

Customizing the Navigation Bar

We will start with the customization of the home screen. What we are going to do is to make its background transparent and change the color of the title.

Starting from iOS 13, Apple introduced a new class called `UINavigationBarAppearance` for customizing the navigation bar. You can apply the customization to the navigation bar of a particular view or the whole application. I will demonstrate both cases in this chapter.

Let's open the `RestaurantTableViewController.swift` file. Insert the following lines of code in the `viewDidLoad()` method:

```
if let appearance = navigationController?.navigationBar.standardAppearance
 {

    appearance.configureWithTransparentBackground()

    if let customFont = UIFont(name: "Nunito-Bold", size: 45.0) {

        appearance.titleTextAttributes = [.foregroundColor: UIColor(red: 2
18/255, green: 96/255, blue: 51/255, alpha: 1.0)]
        appearance.largeTitleTextAttributes = [.foregroundColor: UIColor(r
ed: 218/255, green: 96/255, blue: 51/255, alpha: 1.0), .font: customFont]
    }

    navigationController?.navigationBar.standardAppearance = appearance
    navigationController?.navigationBar.compactAppearance = appearance
    navigationController?.navigationBar.scrollEdgeAppearance = appearance
}
```

To customize the navigation bar, you first need to retrieve the current `UINavigationBarAppearance` object. The `standardAppearance` property contains the current appearance settings for the standard size navigation bar. The next step is to modify the properties of the `appearance` object and apply our customizations. The `configureWithTransparentBackground()` function configures the navigation bar with a transparent background and no shadow.

To change the bar title's color and font, you modify the `titleTextAttributes` and `largeTitleTextAttributes` properties. Both properties accept a set of attributes in key/value format. The `titleTextAttributes` is designed for the standard-size title, while the `largeTitleTextAttributes` property is used when displaying large-size title. In the code above, we modify the font color and apply our custom font.

Even though we have completed the modification of the `appearance` object, the customization will not be applied to the navigation bar immediately. You have to assign the object to the `standardAppearance`, `compactAppearance`, and

`scrollEdgeAppearance` properties. Each of these properties is responsible for the appearance of the navigation bar at different state. The `standardAppearance` property stores the appearance settings of the standard size navigation bar, while the setting of `compactAppearance` controls the appearance of the compact size navigation bar. `scrollEdgeAppearance` is the appearance settings to use when the edge of any scrollable content reaches the matching edge of the navigation bar.

Now run the app to have a quick test. You should see the result displayed in figure 15-2.



Figure 15-2. The custom navigation bar in light and dark modes

Now let's move on to the detailed view. What we want to do is to shift the table view upwards towards the top edge of the screen. Insert the following code in the `viewDidLoad()` method of `RestaurantDetailViewController` :

```
tableView.contentInsetAdjustmentBehavior = .never
```

The value of the `contentInsetAdjustmentBehavior` property controls the behavior for determining the adjusted content offsets of the table view. By default, it is set to `.always` . In this case, iOS automatically adjusts the content offset of the table view

such that the content area will not be blocked by the navigation bar (see figure 15-3 (left)). Now we set `contentInsetAdjustmentBehavior` to `.never`, telling iOS not to adjust the content area.

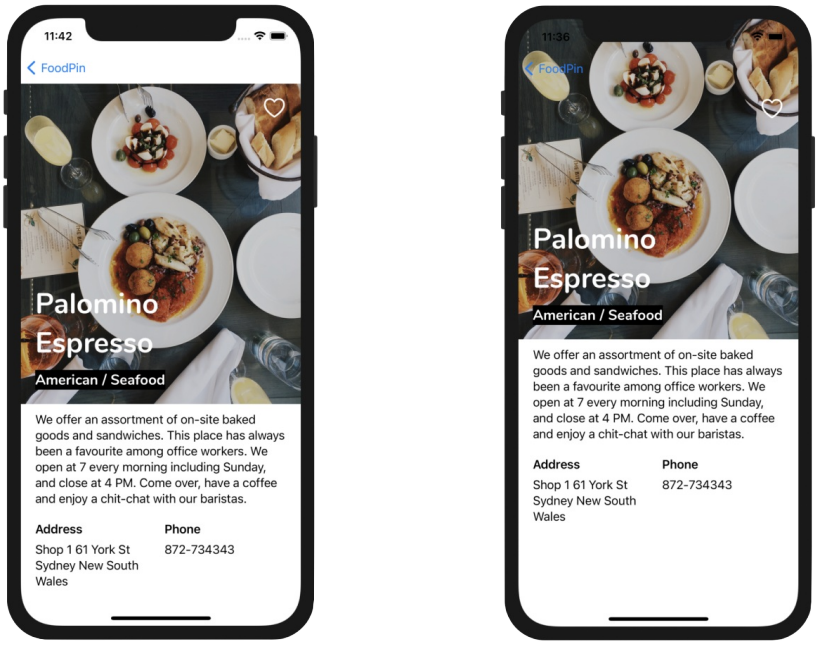


Figure 15-3. contentInsetAdjustmentBehavior sets to .alway (left), contentInsetAdjustmentBehavior sets to .never (right)

As mentioned at the beginning of this chapter, we also want to customize the back button icon of the navigation controller. Instead of using the default image, we want to display an arrow image (i.e. arrow.backward) from SF Symbols.

The `UINavigationBarAppearance` object also provides you with a function called `setBackIndicatorImage` for changing the back button image like this:

```
var backButtonImage = UIImage(systemName: "arrow.backward", withConfigurat
ion: UIImage.SymbolConfiguration(pointSize: 20.0, weight: .bold))

backButtonImage = backButtonImage?.withAlignmentRectInsets(UIEdgeInsets(to
p: 0, left: -10, bottom: 0, right: 0))
appearance.setBackIndicatorImage(backButtonImage, transitionMaskImage: bac
kButtonImage)
```

In the sample code above, we load a system image and enlarge it by passing our symbol configuration. To change the back button image, you can call `setBackIndicatorImage` of the `UINavigationBarAppearance` object and pass it our own image. Optionally, we adjust the back button image's position by using `withAlignmentRectInsets`.

This is a global change that applies to all screens with a back button, not just a single view. Therefore, we will not modify the appearance object in `RestaurantTableViewController` to apply the change.

The question is: *where should we put the code above?*

The `AppDelegate` class is sort of the entry point of the application. The class is generated by Xcode when your project is created from a project template. Typically we put this customization code in the `application(_:didFinishLaunchingWithOptions:)` method of the `AppDelegate` class. This method will be called when the application loads up and is suitable for adding customization code that affects the entire application.

Now, update the `application(_:didFinishLaunchingWithOptions:)` method of `AppDelegate` like this:

```
func application(_ application: UIApplication, didFinishLaunchingWithOptio
ns launchOptions: [UIApplication.LaunchOptionsKey: Any]?) -> Bool {

    let navBarAppearance = UINavigationBarAppearance()

    var backButtonImage = UIImage(systemName: "arrow.backward", withConfig
uration: UIImage.SymbolConfiguration(pointSize: 20.0, weight: .bold))

    backButtonImage = backButtonImage?.withAlignmentRectInsets(UIEdgeInsets
(top: 0, left: -10, bottom: 0, right: 0))
    navBarAppearance.setBackIndicatorImage(backButtonImage, transitionMask
Image: backButtonImage)

    UINavigationBar.appearance().tintColor = .white
    UINavigationBar.appearance().standardAppearance = navBarAppearance
    UINavigationBar.appearance().compactAppearance = navBarAppearance
    UINavigationBar.appearance().scrollEdgeAppearance = navBarAppearance

    return true

}
```

In the code above, we first create the `UINavigationBarAppearance` object and change the back button image. The modified `appearance` object is then assigned to the `standardAppearance`, `compactAppearance`, and `scrollEdgeAppearance` properties.

That's it! You can now run your app and have a look. Since we also change the `tintColor` property of the navigation bar, the back button icon should be changed to a white arrow.



Figure 15-4. Changing the back button image

As you can see, the app still shows the back button title. If you want to remove the title, open `RestaurantTableViewController.swift` and add the following line of code in the `viewDidLoad()` method:

```
navigationItem.backButtonTitle = ""
```

Test the app again, the navigation bar should no longer display the back button title.

Hiding a Navigation Bar on Swipe

Since iOS 8, Apple has introduced a nifty feature which allows you to hide a navigation bar on swipe or tap. This feature shouldn't be new to you. When scrolling down a web page in mobile Safari, the address bar is condensed and the toolbar goes away. If you want to build this feature in your app, you will need a click or a line of code to enable this feature.

In `Main.storyboard`, select the navigation controller in the navigation controller scene. In the Attribute inspector, there is an option named `On Swipe` under the navigation controller section. When enabled, your app will hide the navigation bar, as well as, toolbar (if any) when a user scrolls down the content view. The bars appear again when the user swipes back up.

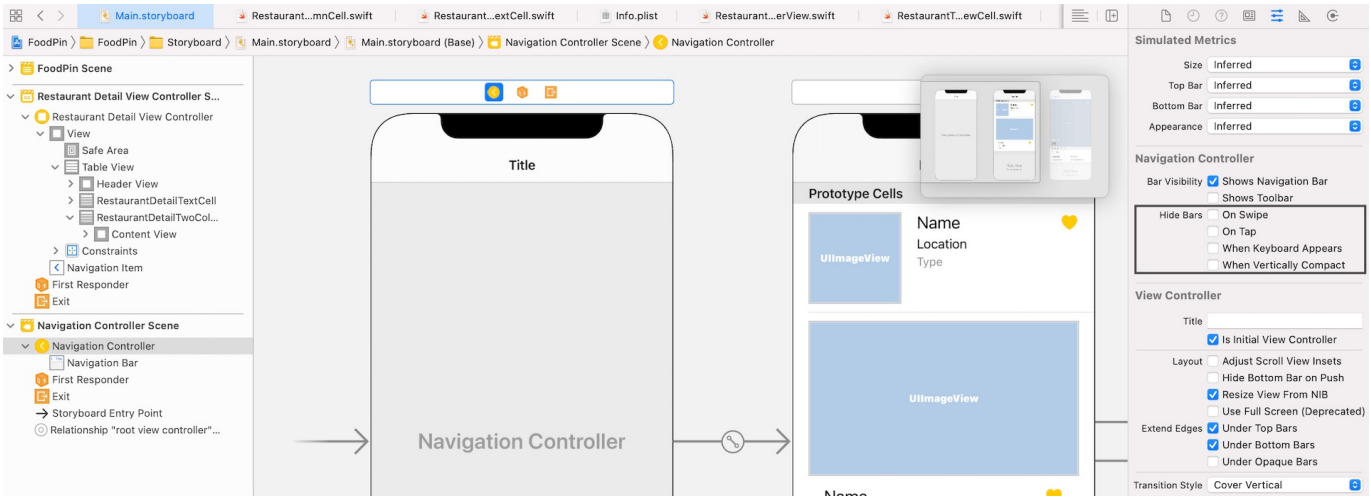


Figure 15-5. On Swipe option for hiding the bars

This change applies to all navigation bars of the entire app. That means, both the restaurant table view controller and the detail view controller are automatically enabled with the feature. What if we just want to hide the navigation bar for the restaurant table view controller? It would be better for us to do this in code.

> **Note:** If you've enabled the On Swipe option in Interface Builder, please disable it before moving on because we will set the option programmatically.

To hide the bars on swipe, you set the `hidesBarsOnSwipe` property to `true` using the code below:

```
navigationController?.hidesBarsOnSwipe = true
```

Let's add the line above in the `viewDidLoad` method of `RestaurantTableViewController`. After that, run the app to have a quick test. Swipe up the table view and the navigator bar will be hidden. It works pretty well.

However, if you select a restaurant to navigate to the detailed view, this swipe-to-hide feature is enabled too. Okay, we want to disable the swipe-to-hide feature in the detail view. Let's add the following code in the `viewDidLoad` method of `RestaurantDetailViewController` to disable it:

```
navigationController?.hidesBarsOnSwipe = false
```

Run the app again to test it. Does it fix the issue?

At first, it looks like we have fixed the navigation bar issue. But if you test more thoroughly, you will find other issues as illustrated in figure 15-6.



Figure 15-6. Possible issues when setting the hidesBarsOnSwipe property in the viewDidLoad method

We briefly discussed the view controller life cycle. iOS calls the `viewDidLoad` method when the view controller's content view is first created and loaded from a storyboard. This is why we use to set up our outlet variables or perform other initializations in `viewDidLoad`.

However, keep in mind that the `viewDidLoad` method is called only once when the view is first loaded. After that, it won't be called again. In other words, if you navigate to the detail view, the `hidesBarsOnSwipe` property is set to `false`. And the value will keep intact even when you navigate back to the table view controller. This is the reason why the navigation bar won't hide (Problem #1).

For Problem #2, the hidden navigation bar is carried over to the detail view. Even if we manage to set the `hidesBarsOnSwipe` property back to false in the `viewDidLoad` method, it won't display the navigation bar. We have to explicitly tell the app to re-display the navigation bar.

Fortunately, the `UIViewController` class provides several methods that respond to different view states. Unlike the `viewDidLoad` method, these methods are called every time when a view is displayed or removed. When a view is displayed, both `viewWillAppear` and `viewDidAppear` methods are invoked. The `viewWillAppear` method is called when the view is about to display, whereas the `viewDidAppear` method is triggered as soon as the view is presented on screen.

Apparently, the `viewWillAppear` method fits our need. Insert the following code in the `RestaurantTableViewController` class:

```
override func viewWillAppear(_ animated: Bool) {
    super.viewWillAppear(animated)

    navigationController?.hidesBarsOnSwipe = true
}
```

For the `RestaurantDetailViewController` class, add the following code snippet:

```
override func viewWillAppear(_ animated: Bool) {
    super.viewWillAppear(animated)

    navigationController?.hidesBarsOnSwipe = false
    navigationController?.setNavigationBarHidden(false, animated: true)
}
```

As the `viewWillAppear` method is called every time when a view is displayed, we can toggle the `hidesBarsOnSwipe` property perfectly. The extra line of code in the `RestaurantDetailViewController` class explicitly tells the app to unhide the navigation bar. This resolves problem #2. One thing to take note is that you must call the `super` method at some point of the implementation. After the changes, run the app and have fun!

Exercise #1

Before we continue to work on the new features of the app, let's fix a bug that you may notice for some time. In the main view, we use the large title for the navigation bar. Did you notice that the navigation bar becomes a standard size bar when you navigate from the detailed view back to the main view?

Your exercise is to fix this bug such that the navigation bar of the main view (i.e. `RestaurantTableViewController`) always shows large title.

Swift Extensions

Before I go on to discuss with you about other UI customizations, I want to take a detour here and talk about a feature in Swift known as *Extensions*. We have discussed extensions before but I want to go a little deeper and show you how to utilize this powerful Swift feature.

Extensions in Swift is a great feature that lets you add new functionality to an existing class or other types including Struct and Enum . What does it mean to you? How can you use this feature to write better code? Let me show an example.

If you look into the `RestaurantTableViewController` class, you will find multiple lines of code related to the initialization of `UIColor` like this:

```
UIColor(red: 218/255, green: 96/255, blue: 51/255, alpha: 1.0)
```

Here we instantiate a `UIColor` object with custom RGB values. Because `UIColor` only accepts the RGB value in the scale of 0 to 1, we have to divide each of the RGB components by 255 during initialization.

It is a bit troublesome. Can we simplify the code above to this?

```
UIColor(red: 218, green: 96, blue: 51)
```

Here we can apply extensions to extend the functionality of `UIColor`. Even though `UIColor` is a built-in class provided by the iOS SDK, we can use Swift extensions to add more features to it.

To better organize our projects, let's first create a group for storing the extension file. In project navigator, right click the `FoodPin` folder and choose *New Group*. Name the group *Extensions*.

Next, right click `Extensions` and select *New File...*. Choose the *Swift File* template and name the file `UIColor+Ext.swift`. Once the file is created, update the code like this:

```
import UIKit

extension UIColor {
    convenience init(red: Int, green: Int, blue: Int) {
        let redValue = CGFloat(red) / 255.0
        let greenValue = CGFloat(green) / 255.0
        let blueValue = CGFloat(blue) / 255.0
        self.init(red: redValue, green: greenValue, blue: blueValue, alpha
: 1.0)
    }
}
```

To declare an extension for an existing class, you start with the keyword `extension` followed by the class you want to extend. Here it is the `UIColor` class.

We implement an additional convenience initializer that accepts three parameters: *red*, *green* and *blue*. In the body of the initializer, we perform the conversion by dividing the given RGB value by 255. Finally, we call the original `init` method with the converted RGB components.

That's how you use Swift extensions to add another initializer to a built-in class. Now the new initializer is ready for use. For example, you can change the following line of code in `RestaurantTableViewController` from:

```
appearance.titleTextAttributes = [.foregroundColor: UIColor(red: 218/255,
green: 96/255, blue: 51/255, alpha: 1.0)]
```

To:

```
appearance.titleTextAttributes = [.foregroundColor: UIColor(red: 218, gree
n: 96, blue: 51, alpha: 1.0)]
```

Can you still use the original initializer? Absolutely. The new initializer just simplifies the redundant conversion and lets you type less code.

Adapting Colors for Dark Mode

With the introduction of Dark Mode, your app should cater for both light and dark appearance. Up till now, the FoodPin app works pretty well in Dark mode. One main reason is that we mostly use the built-in adaptive colors provided by Apple.

Along the debut of Dark Mode in iOS 13, Apple introduced two types of built-in color: *system colors* and *semantic colors*. All these colors are designed to adapt to different system environment. Whether you choose the system color or the semantic color, the color looks good on both light and dark appearance. As mentioned before, the same system color (say, System Red Color) has different color values for different interface styles.

Like System colors, semantic colors are also designed to be adaptive and return different color values for different interface styles. The color name describes the intention of the color. For example, *Label Color* is created for label and *System Background Color* is used for painting the view's background.

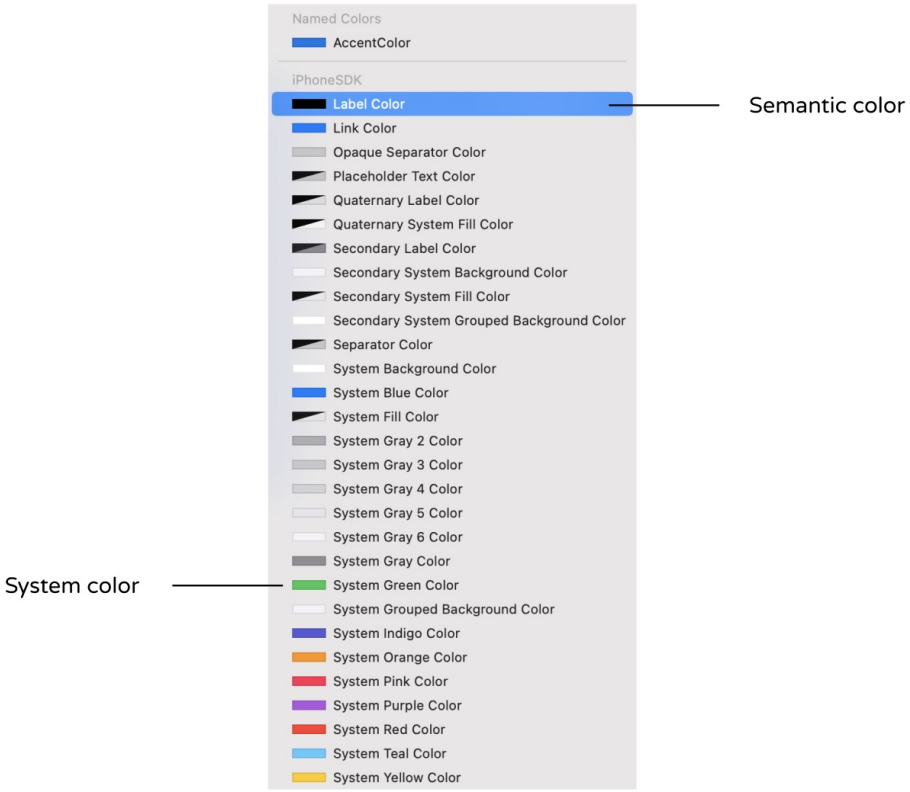


Figure 15-7. Adaptive and semantic colors

According to Apple, developers are encouraged to use system and semantic colors because it makes you a lot easier to support dark mode. You should avoid creating a `UIColor` object with hardcoded color values. That said, what if we want to use our own color instead of the built-in ones like the one we used for the navigation bar title?

In the asset catalog, you can create your custom color that has different color values for light and dark appearance. Right any blank area in the asset catalog to bring up the context menu. Select *Color Set* to add a new color set.

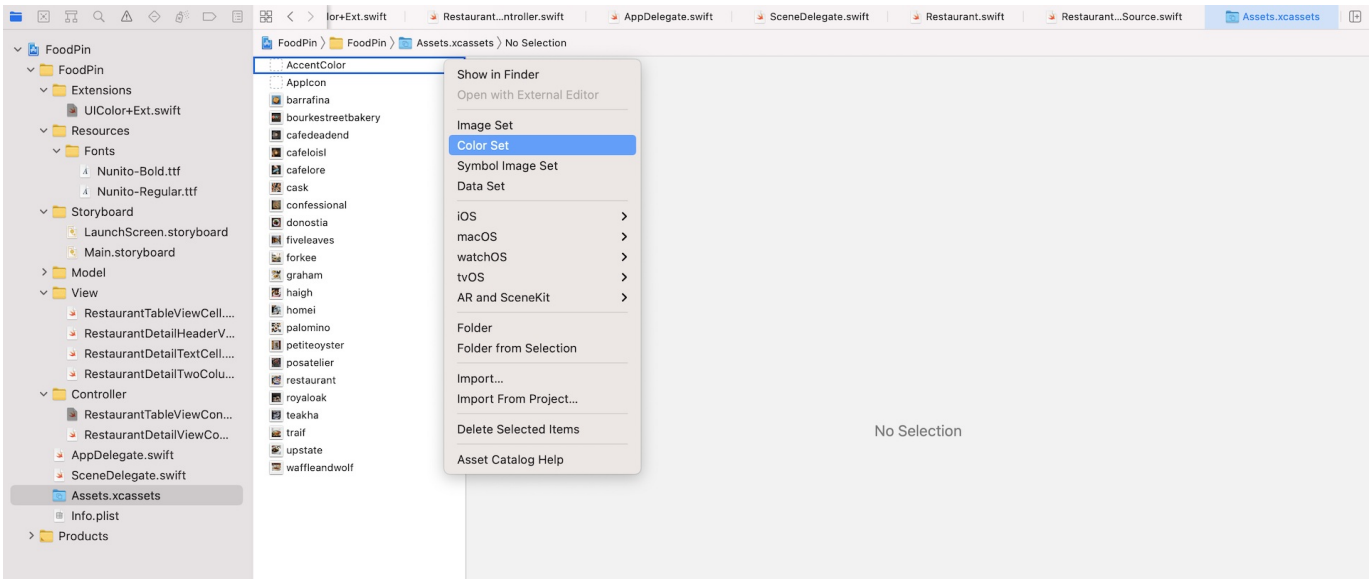


Figure 15-8. Adding a new color set in asset catalog

Name the color set `NavigationBarTitle` and then you will need to fill in two different color for *Any Appearance* and *Dark Appearance*. First, select *Any Appearance* and go to the Attributes inspector. Change the input method to *8-bit Hexidecimal* and set the Hex value to `#DA6033` .

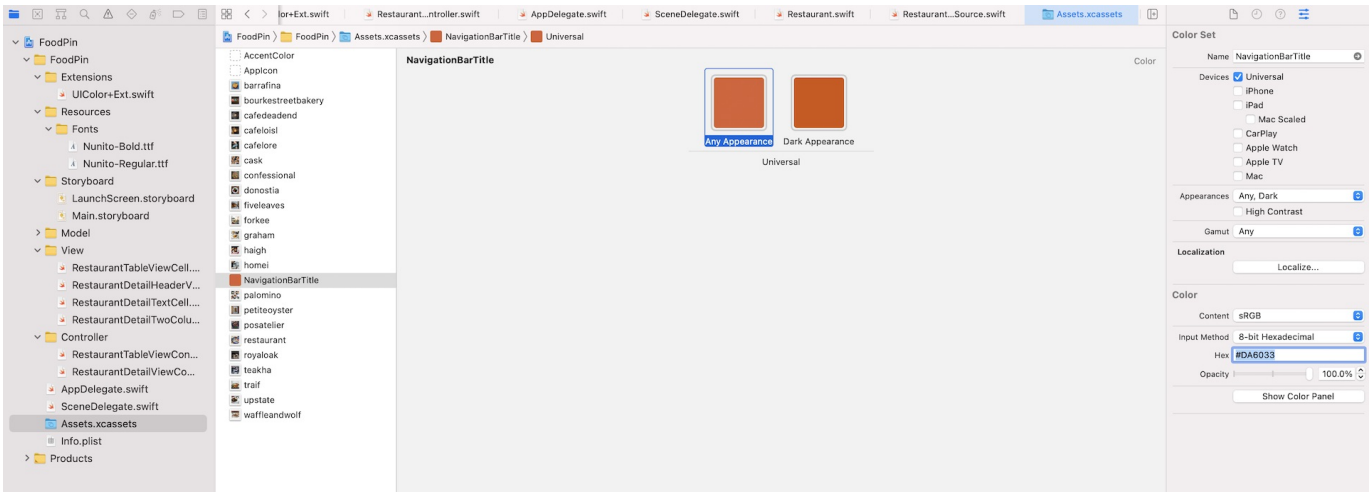


Figure 15-9. Setting the color for Any Appearance

Repeat the same procedures but set the color for Dark Appearance. We use a different color value for this interface style, so set the *Hex* value to `#D35400`. This is how you define a custom semantic color.

To use this custom color, you can create the `UIColor` instance with the color name like this:

```
let color = UIColor(named: "NavigationBarTitle")!
```

Now you can replace the following lines of code in the `viewDidLoad()` method of `RestaurantTableViewController`:

```
appearance.titleTextAttributes = [.foregroundColor: UIColor(red: 218, gree
n: 96, blue: 51, alpha: 1.0)]
appearance.largeTitleTextAttributes = [.foregroundColor: UIColor(red: 218,
 green: 96, blue: 51, alpha: 1.0), .font: customFont]
```

With:

```
appearance.titleTextAttributes = [.foregroundColor: UIColor(named: "Naviga
tionBarTitle")!]
appearance.largeTitleTextAttributes = [.foregroundColor: UIColor(named: "N
avigationBarTitle")!, .font: customFont]
```

Run the app to have a quick test. The navigation bar title now has different colors in light and dark mode.

Changing the Style of Status Bar

Now let's head back to the UI customization. The style of the status bar looks great on the home screen but doesn't go well with the restaurant image in the detail view. Wouldn't it be great if we can change the color of the status bar to white for

the detail view?

The iOS SDK allows developers to use a `UIStatusBarStyle` constant to specify whether the status bar content should be dark or light. By default, the status bar displays dark content. In other words, items such as time, battery indicator and Wi-Fi signal are displayed in dark color.

You may want to change the style of status bar from dark to light to make the app look better. There are two ways to do this. You can change the status bar style for the entire application or just change the style of individual controllers.

Apparently, the status bar style varies from view controller to view controller for the FoodPin app. For the home screen, we want to keep the default style, while the status of the detail view should change to the *light* style.

Normally, to control the style of the status bar in any view controller, you can add the following code in your class to override the `preferredStatusBarStyle` property and return your preferred style:

```
override var preferredStatusBarStyle: UIStatusBarStyle {
    return .lightContent
}
```

You can insert the code above in the `RestaurantDetailViewController` class to have a try.

Unfortunately, the above code does not work in this case.

The reason is that the detail view controller is embedded in a navigation controller. iOS uses the default style specified in the navigation controller instead.

So how can we tell iOS to change the status bar style?

It is a bit tricky. You need to create a custom `UINavigationController` class and override its `preferredStatusBarStyle` property. In the project navigator, right click the `Controller` group and choose *New File*... Select the *Cocoa Touch Class* template and name the class `NavigationController`. Set the subclass to `UINavigationController`. Update the code like this:

```
class NavigationController: UINavigationController {

    override var preferredStatusBarStyle: UIStatusBarStyle {
        return topViewController?.preferredStatusBarStyle ?? .default
    }

}
```

We override the `preferredStatusBarStyle` property such that it returns the value of `preferredStatusBarStyle` of the top view controller. Here, the `topViewController` is the view controller at the top of the navigation stack. In the case of the detailed view, the `RestaurantDetailViewController` is the top view controller.

Lastly, switch over to `Main.storyboard`. We need to update the navigation controller to use our custom navigation controller. Select the *Navigation Controller* in the document view and then change the custom class in the Identity inspector.

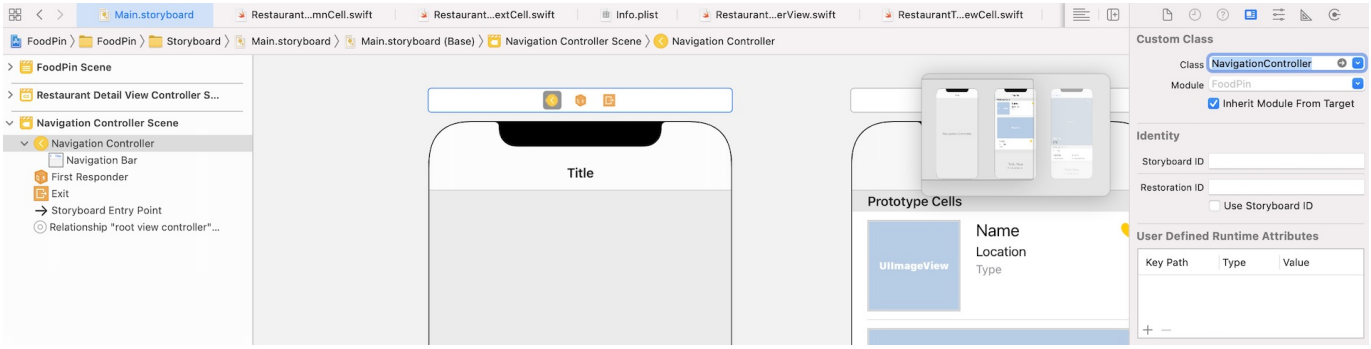


Figure 15-10. Changing the custom class to NavigationController

Now run the app again to see what you get. The status bar should be changed to white in the detailed view.

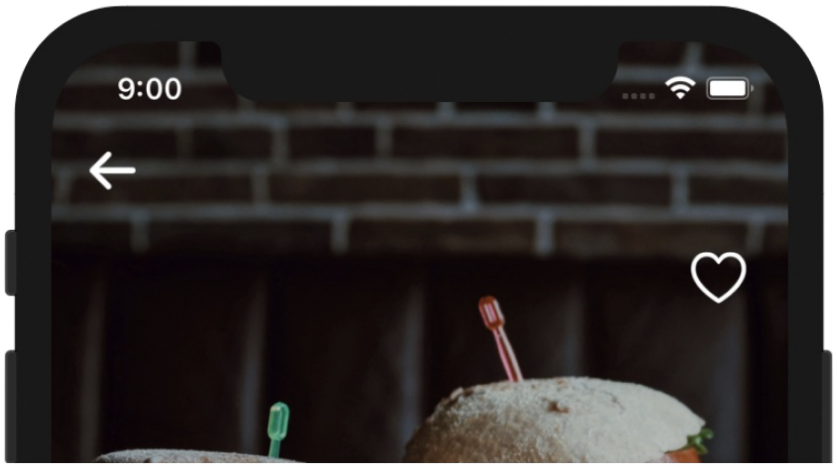


Figure 15-11. The detailed view shows the status bar in white

This is how you manage the status bar style for individual controllers. Though this is good enough for our app, I want to take a few minutes to show you how to change the style of the status bar across the entire application.

To do that, you can select the *FoodPin* project in the project navigator. In the *General* tab, you can change the status bar style to *Light Content*. This will change the status bar style for the entire application.

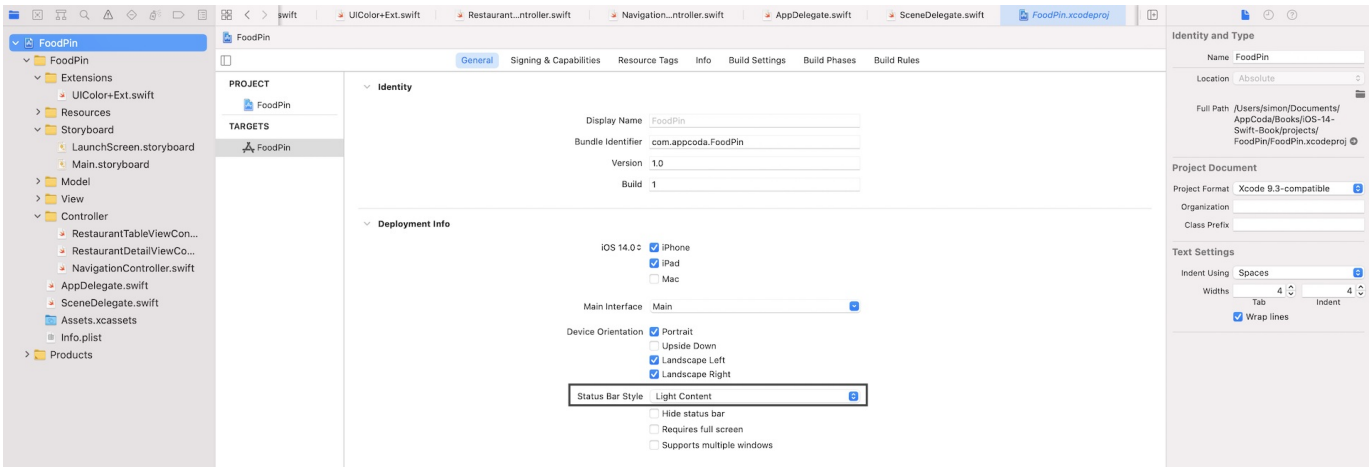


Figure 15-12. Changing the status bar style

However, there is one thing you need to configure before this modification works.

By default, the Xcode project is enabled to use "View controller-based status bar appearance". This means you can control the appearance of the status per view controller. If you want to change the style of the status bar globally, you need to opt out the *View controller-based status bar appearance* by modifying the `Info.plist` file.

Select the file to open it. Right click *Information Property List* and select *Add Row*. Insert a new key named `View controller-based status bar appearance` and set the value to `NO`. This instructs Xcode to use the status bar setting we set in the *General* tab earlier. Now the entire app will use the new status style.

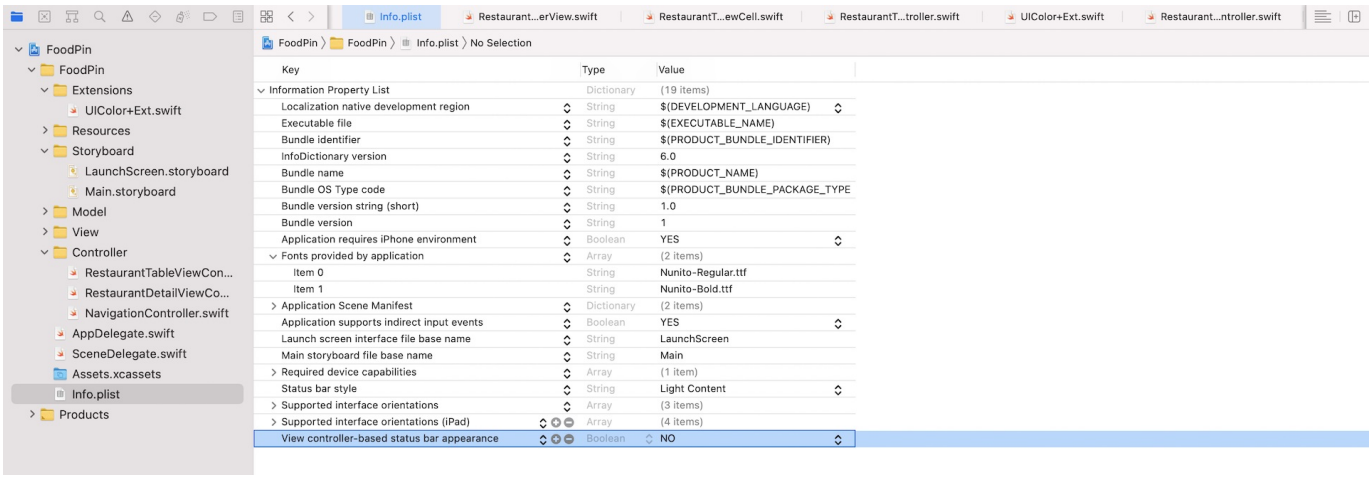


Figure 15-13. Editing the Info.plist to add a new key

Dynamic Type

What is Dynamic Type? You may not have heard of Dynamic Type but you should have seen the setting screen (*Settings* > *Accessibility* > *Display & Text Size* > *Large Text* or *Settings* > *Display & Brightness* > *Text Size*) shown in figure 15-14.

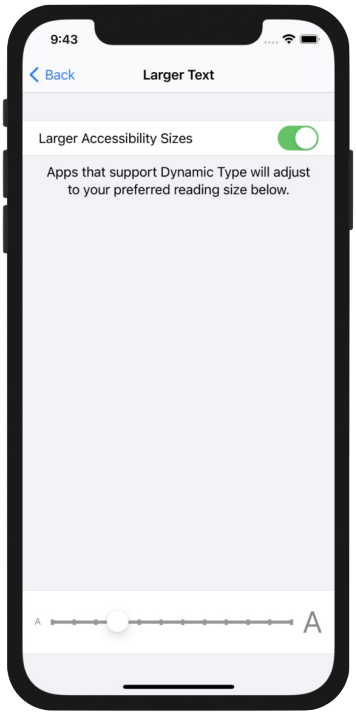


Figure 15-14. Larger Text Setting

Dynamic Type is not new and has been introduced since iOS 7. With this feature, users are able to customize the text size of an app to fit their own needs. However, there is a catch - only apps that adopt Dynamic Type respond to the text change.

All of the stock apps have adopted Dynamic Type. For third party apps, it is up to developers' decision. Though Apple doesn't mandate developers to support Dynamic Type, it is always recommended so users can choose their own text size.

So how can you adopt Dynamic Type? As we have designed our views using auto layout, adopting Dynamic Type is just a piece of cake. Recalled that we configured all the labels to use text style, this is what you need to adopt Dynamic Type: *use a text style instead of a fixed font type*.

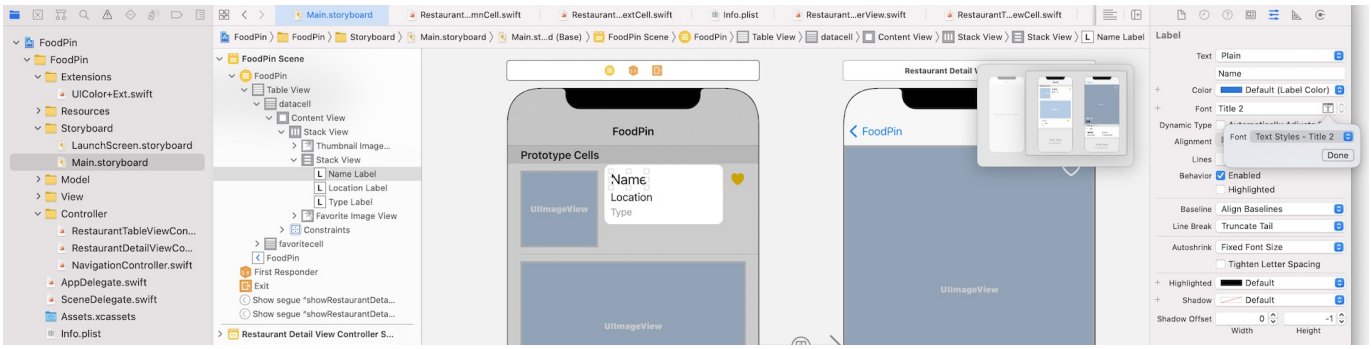


Figure 15-15. The name label is configured with a text style

So, how can you test dynamic types using the built-in simulator? One way is to change the setting in the simulator by going to *Settings* > *Accessibility* > *Display & Text Size* > *Large Text*. Enable *Larger Accessibility Sizes* and drag the slider to your right to enlarge the font.

A more convenience approach is use the *Environment Overrides* option in Xcode. While running the app in simulators, click the *Environment Overrides* button and switch on the Text option. You can use the Dynamic Type slider to adjust the font size. In the simulator, the app should respond to the size change.

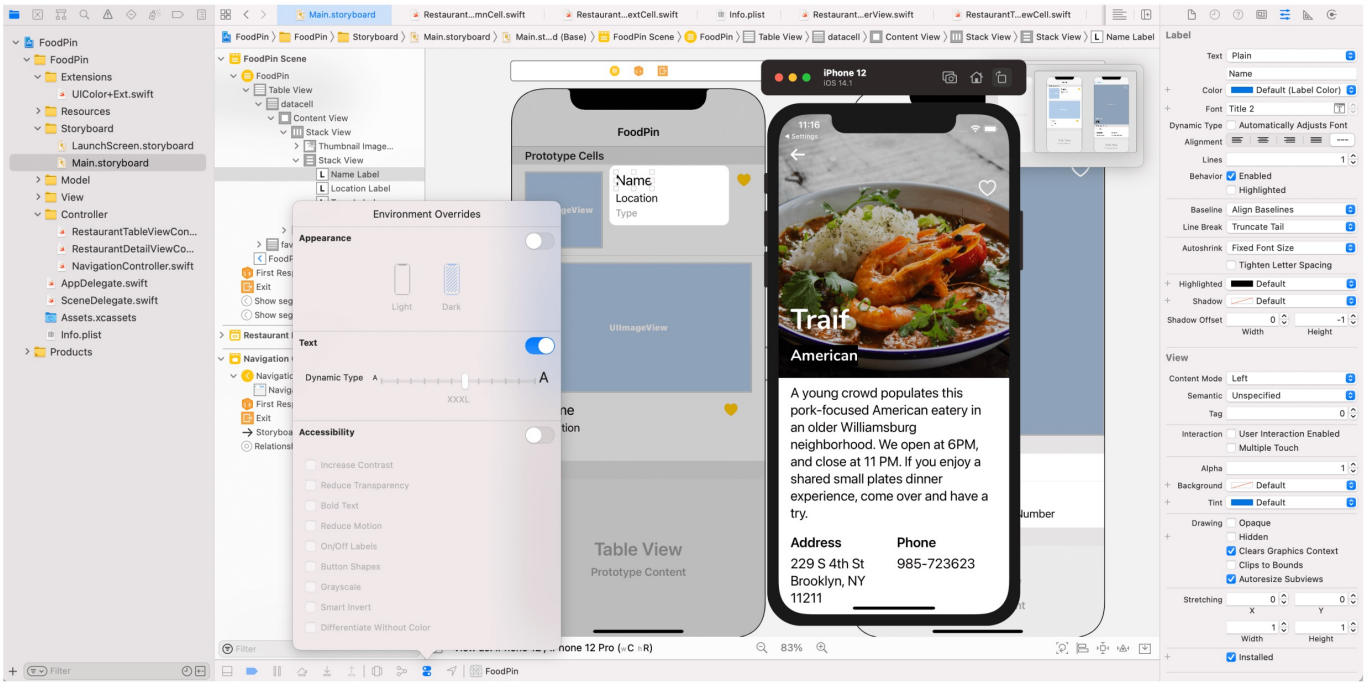


Figure 15-16. Using Environment Overrides to test dynamic types

There is an issue with the current implementation. In the simulator, go to *Settings* > *Accessibility* > *Display & Text Size* > *Large Text*. Enable *Larger Accessibility Sizes* and increase the text size. Switch back to the simulator. You should find that the size of the labels remains unchanged. However, if you rerun the app again, the font size will change accordingly.

How can we fix that?

There is an option in text labels called *Automatically Adjusts Font*, which is disabled by default. When this option is enabled, it will listen to the change of the text size in the system settings and update to the new size automatically.

Go to `Main.storyboard` and select the name label of the restaurant table view controller. In the Attributes inspector, enable the *Automatically Adjusts Font* option. Repeat the same procedures for other labels. By turning on the option, the

label will be automatically resized whenever the user changes the preferred font size in Settings.

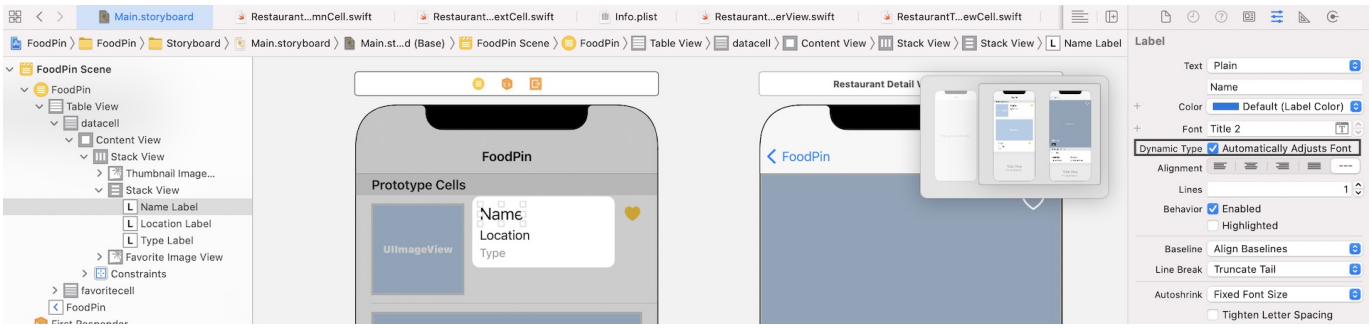


Figure 15-17. Enabling the Automatically Adjusts Font option

Alternatively, you can also apply the change by setting the `adjustsFontForContentSizeCategory` property of `UILabel` to `true`. Here is an example:

```
descriptionLabel.adjustsFontForContentSizeCategory = true
```

Exercise #2

Similar to exercise #1, this exercise is about bug fixing. Assuming you have tested the app for Dynamic Type, you should notice that the app doesn't work great for all font sizes. Here are the issues when you set the font size to maximum:

1. In the main screen, the address is truncated when the font is enlarged.
2. If the text is too long, the type label in the detailed view is truncated
3. For iPad, the restaurant name in the detailed view cannot be fully displayed if the name is too long.

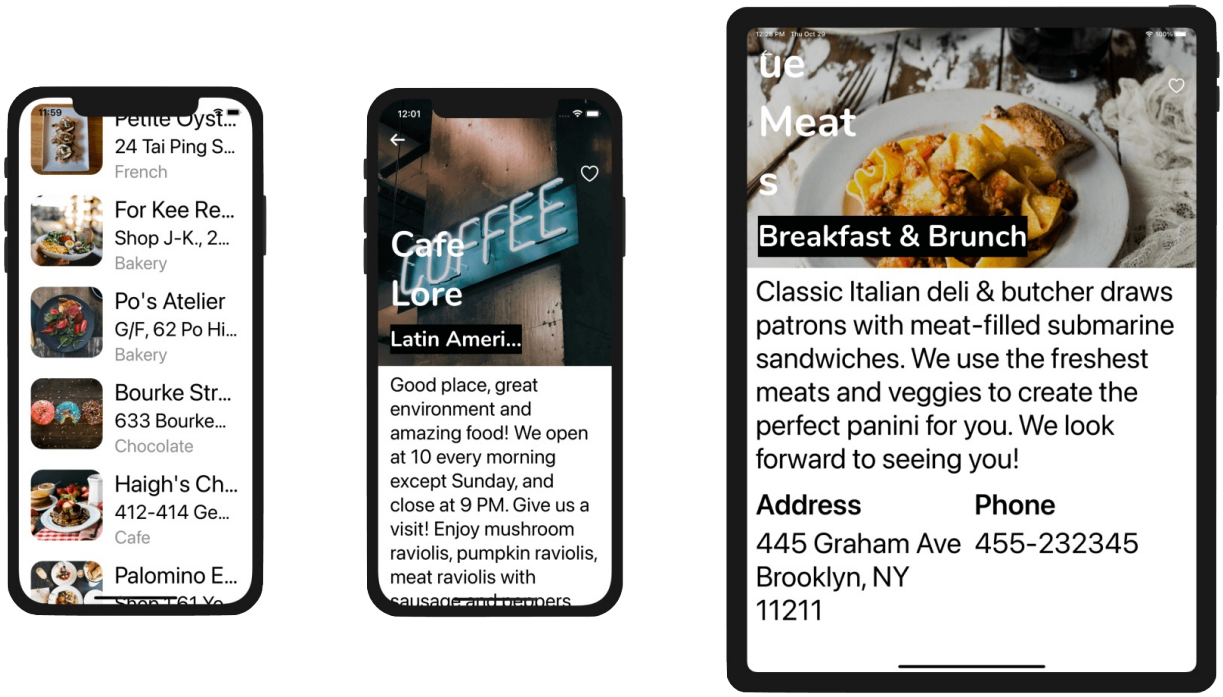


Figure 15-18. A few issues when the font becomes larger

Based on what you have learned, try to fix all these issues. One of the issues is related to auto layout. You will need to alter a constraint specifically for the iPad devices.

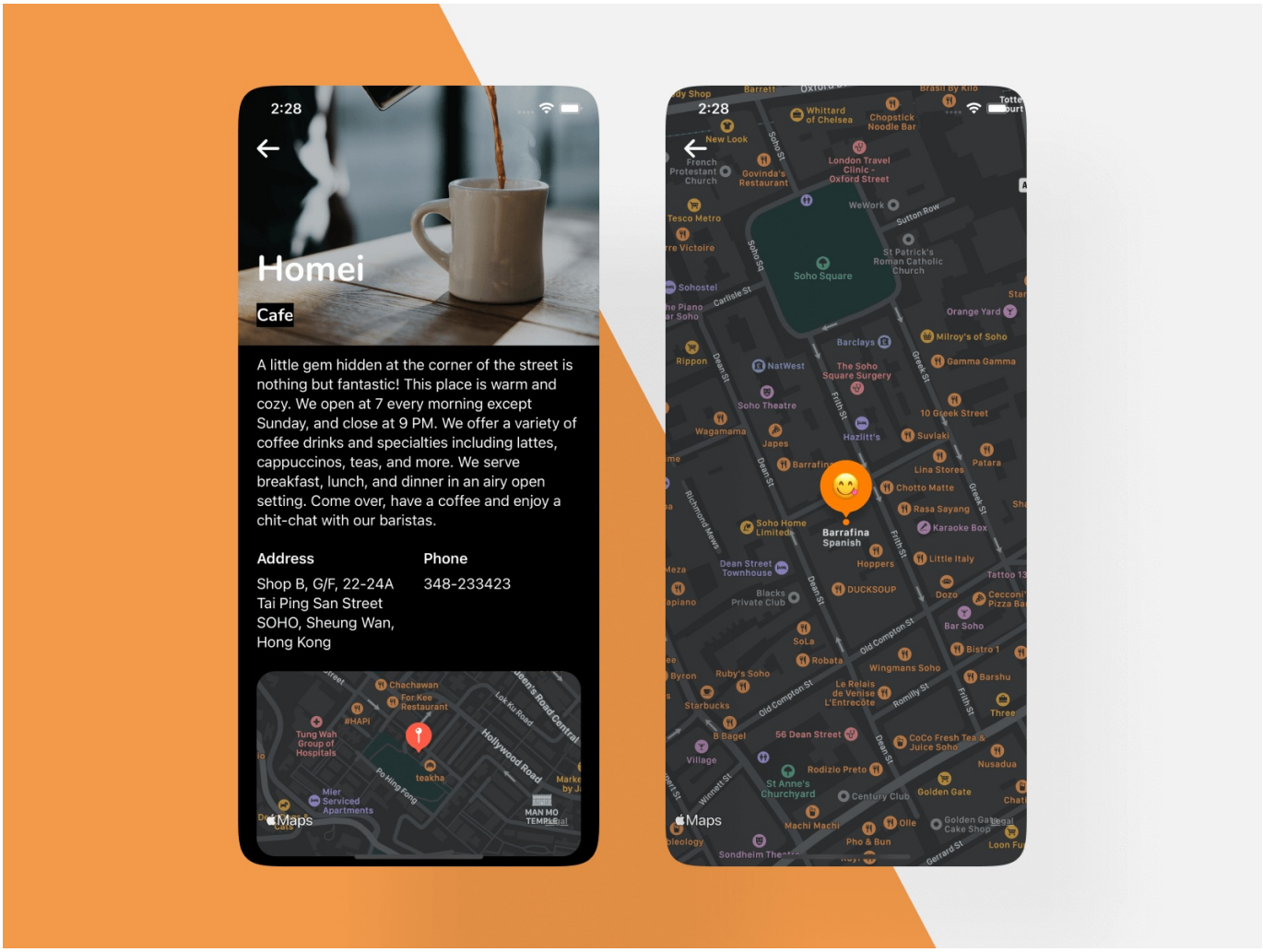
Summary

In this chapter, you've learned how to customize the navigation bar and status bar. I have also covered Dynamic Color and Dynamic Type, which allows your users to choose their preferred font size. Apple encourages all iOS developers to adopt this technology because this gives your users a choice. For people with less acute vision, they probably prefer a larger text size. For some people, they may prefer a smaller text size. Whenever possible, try your best to support Dynamic Type in your apps.

For your reference, you can download the Xcode project (including the solution to exercise) from http://www.appcoda.com/resources/swift53/FoodPinNavCustomization.zip.

Reference: Apple's iOS Human Interface Guidelines (https://developer.apple.com/ios/human-interface-guidelines/visual-design/typography/)

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.

The latest version of the framework also allows developers to provide pin customization, transit, and flyover support. With the built-in API, developers are given with the option to customize the annotation. 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.

Using MapKit Framework

By default, the MapKit framework is not bundled in the Xcode project. To use it, you have to first add the framework and bundle it in your project. But you don't need to do it manually. Xcode has a capability section that lets you configure frameworks for various Apple technologies such as Maps and iCloud.

In the project navigator, select the *FoodPin* project and choose the *FoodPin* target. You can then enable the `Maps` feature by clicking the `+ Capabilities` button. Just look for Maps and double click it. Xcode will automatically configure the project to use the MapKit framework.

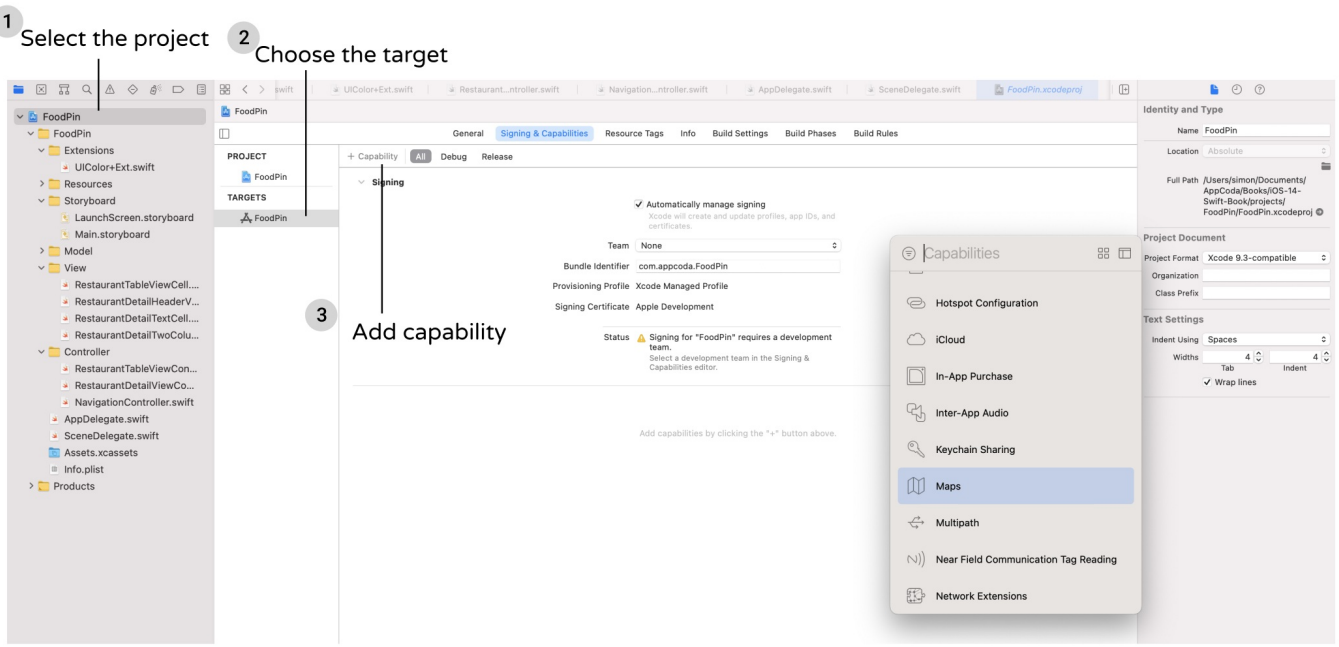


Figure 16-1. Enabling Maps in your Xcode project

Adding a Map Interface to Your App

What we're going to do is to add a non-interactive map to the footer of the restaurant detail view. When a user taps the map, the app navigates to a map view controller showing a full-screen map of the restaurant location. Figure 16-2 displays the resulting UI of the app.

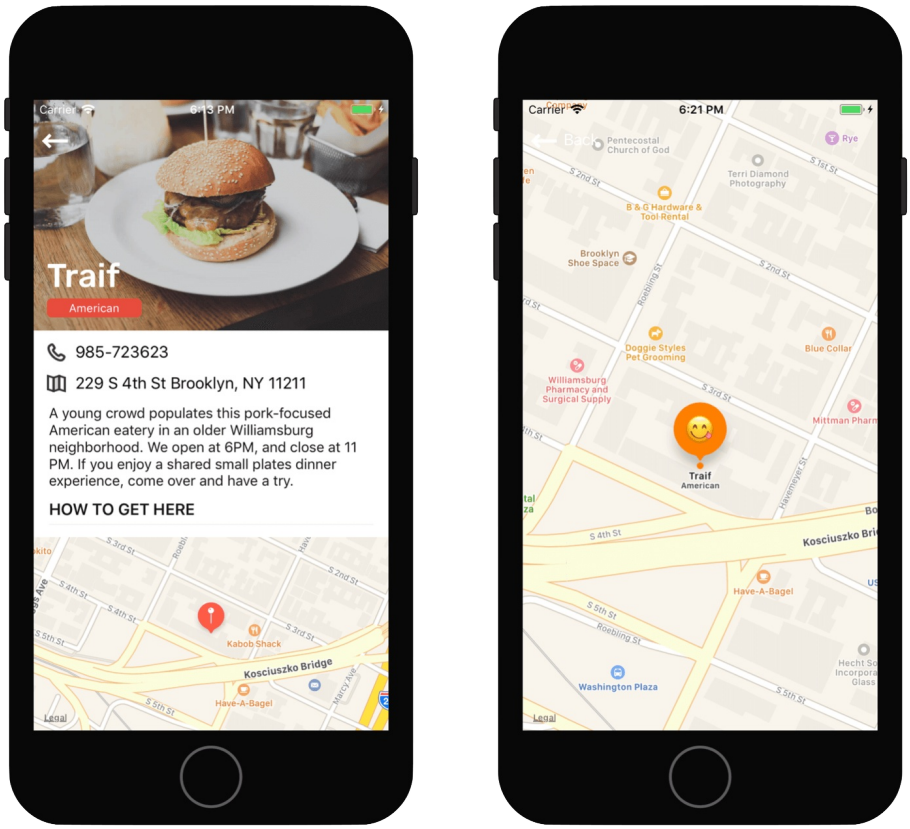


Figure 16-2. An embedded map in the table view footer (left), A full screen map in a view controller (right)

We first start to modify the detail view to support the map function. As you may know, we need to implement a new prototype cell to display the map view. Now open `Main.storyboard` and select the table view of the detail view controller. In the Attributes inspector, change the value of Prototype cell from `2` to `3`. Xcode will automatically duplicate the 2 column cell in the table view. Then delete the stack view to clear its content.

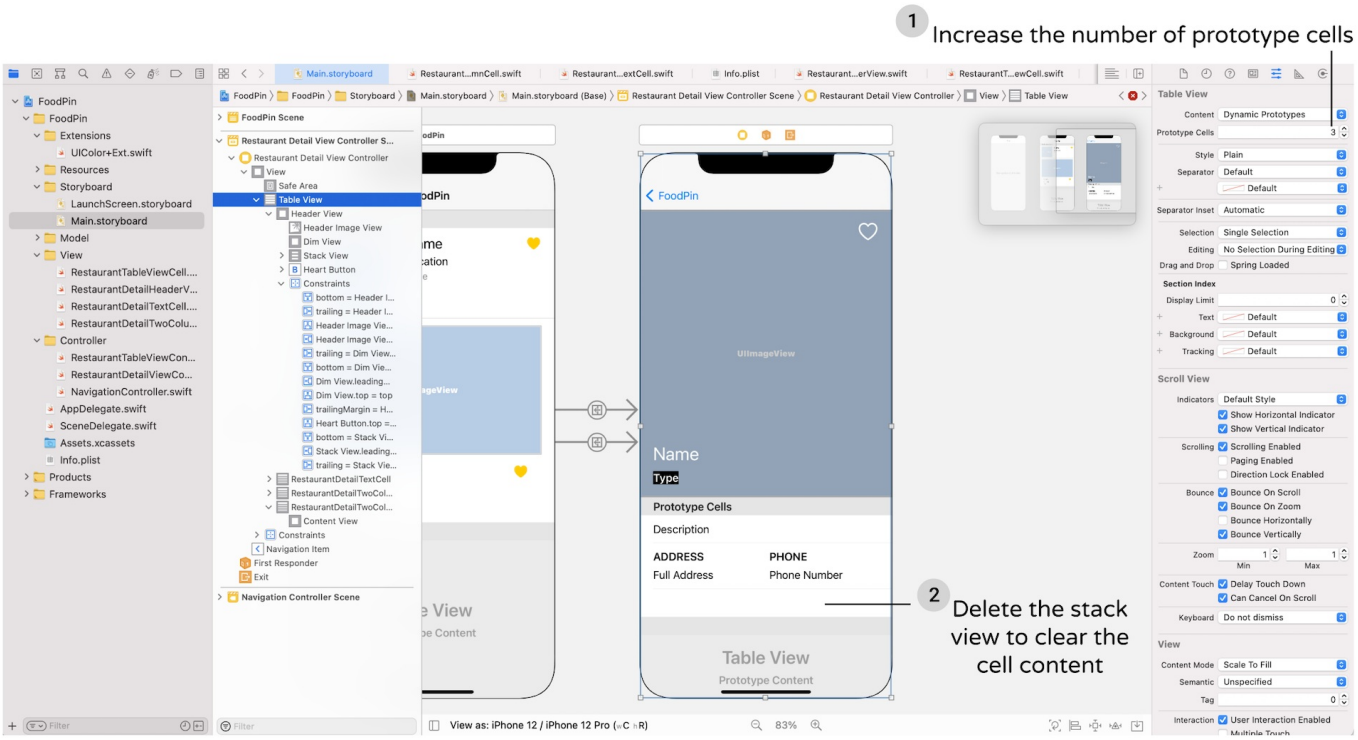


Figure 16-3. Adding two more prototype cells

Open the Object library, look for the *Map Kit View*. This is the object for embedding a map interface.

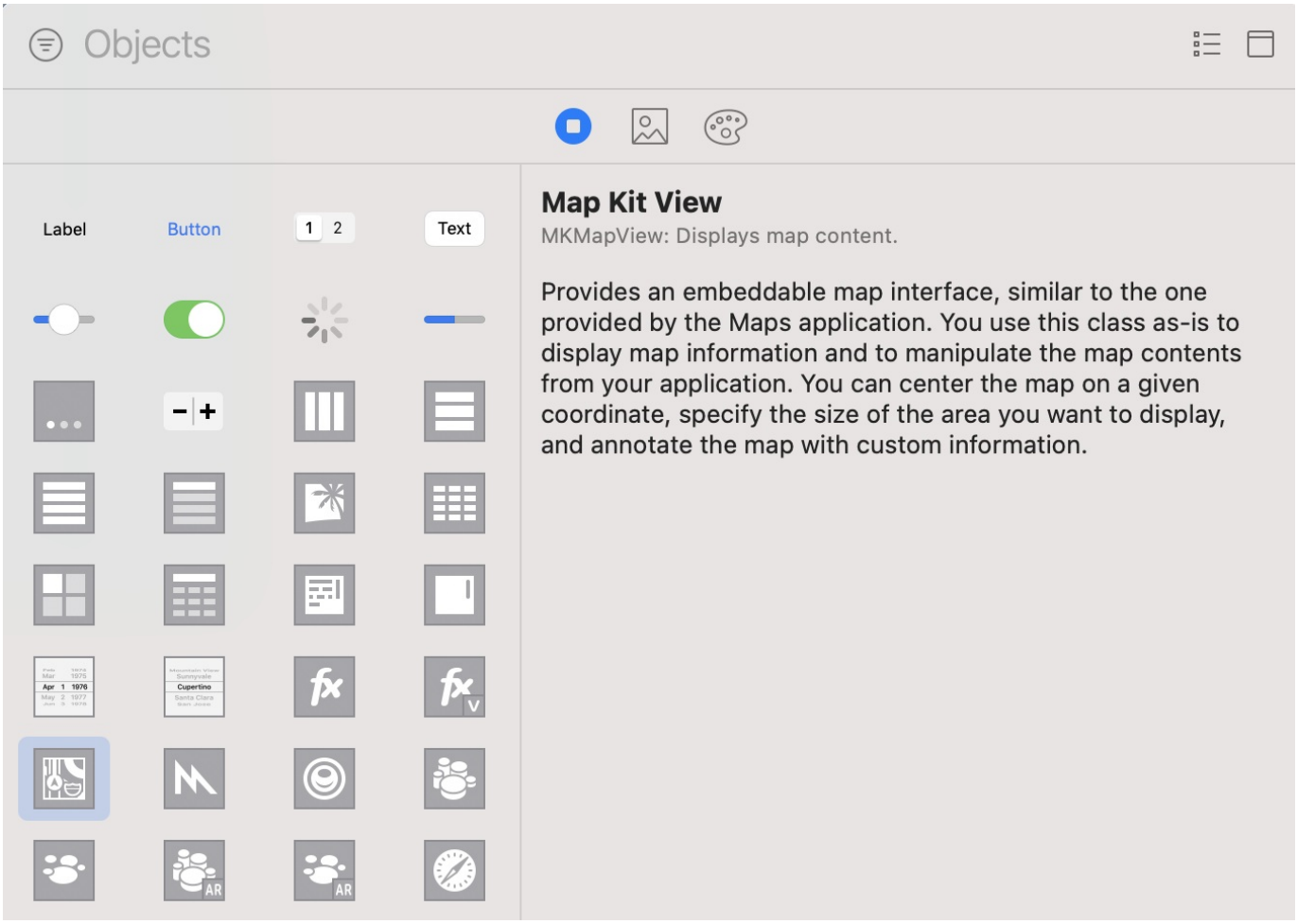


Figure 16-4. Map Kit View

Drag the Map Kit View to the new cell. Then click the *Add new constraints* button and add the spacing constraints for the map view. Set the value of all sides to `0` and height to `200` points. Also, make sure the *Constrain to margins* option is unchecked. Then click *Add 5 constraints* to make the map view fit the whole cell.

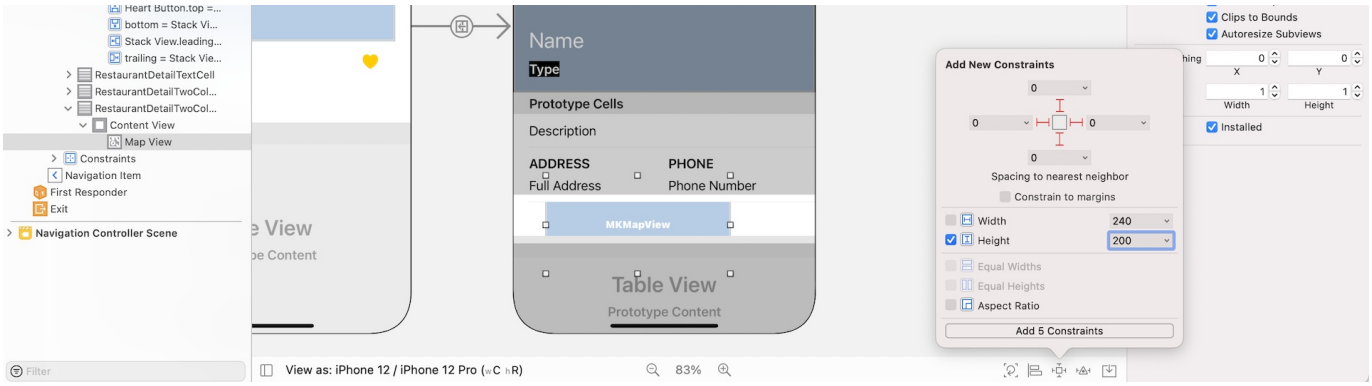


Figure 16-5. Adding the map view to the new prototype cell and defining the spacing constraints

As you will notice, the cell goes beyond the boundary of the table view. For the ease of editing, you can make the view controller a bit larger. Select the *Restaurant Detail View Controller* in Document Outline. In the Size inspector, change its simulated size from `Fixed` to `Freeform`. Set the height to `1000`. This stretches the detail view controller to 1000 points. Now you can easily edit the map cell.

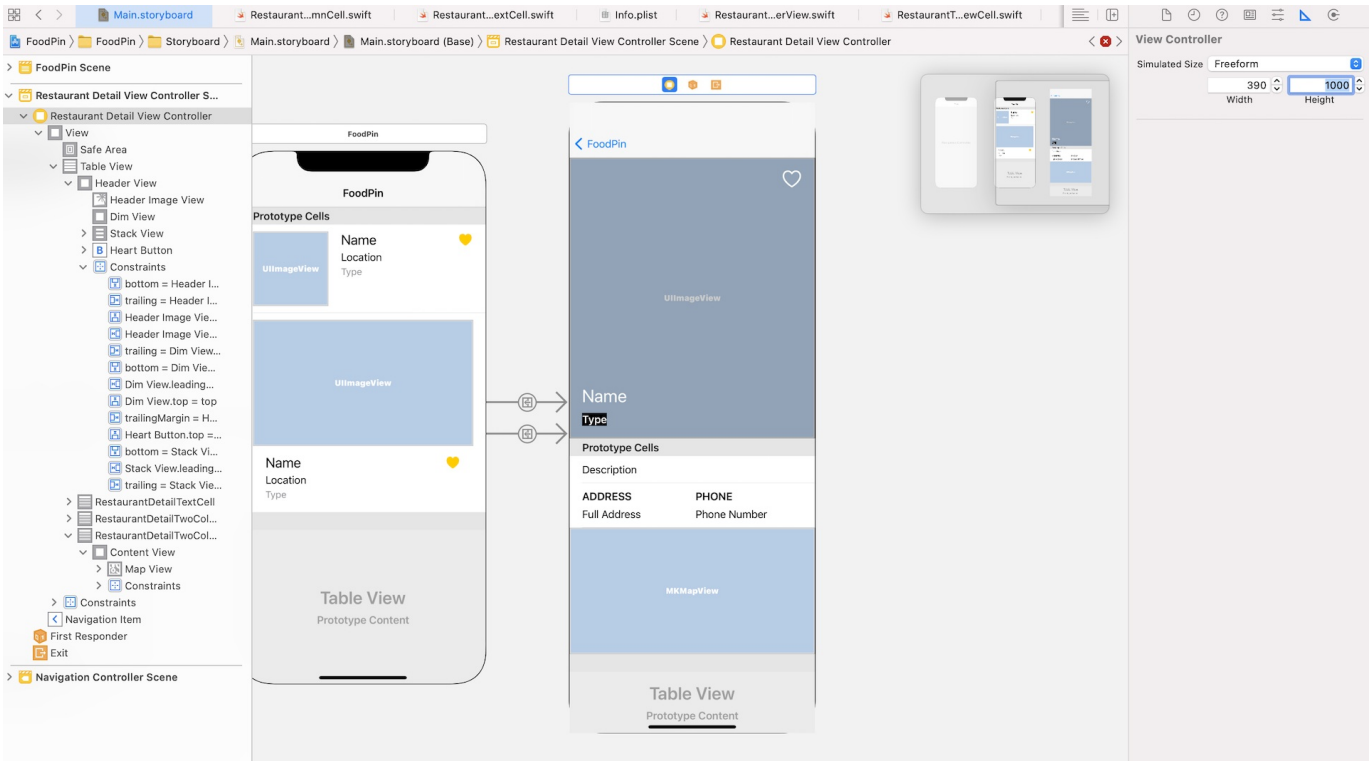


Figure 16-6. Adjusting the simulated size of the detail view controller

If you select the map view and go to the Attributes inspector, you will find several options such as zooming and scrolling for configuring its behavior. As this map view is just displayed in a table cell, I want to keep it less interactive. Uncheck the Scrolling, Rotating and 3D View options.

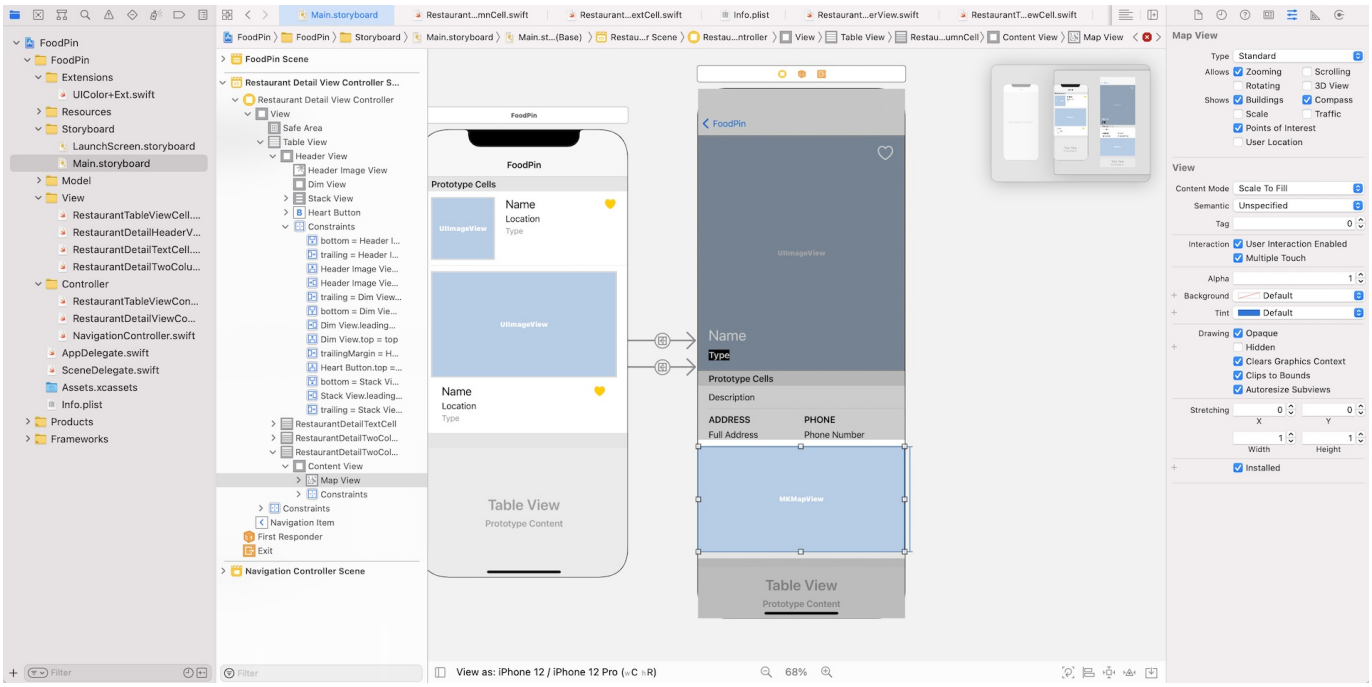


Figure 16-7. Changing the option of the map view

I know you can't wait to test the change. However, there are a couple things we need to set up. We haven't created the Swift files for the map cell yet.

In project navigator, right click the `View` folder and choose *New File*.... Use the *Cocoa Touch Class* template. In the next screen, name the class `RestaurantDetailMapCell` and set its subclass to `UITableViewCell`. Once the Swift file is saved, update its content like this:

```
import UIKit
import MapKit

class RestaurantDetailMapCell: UITableViewCell {

    @IBOutlet var mapView: MKMapView!

    override func awakeFromNib() {
        super.awakeFromNib()
        // Initialization code
    }

    override func setSelected(_ selected: Bool, animated: Bool) {
        super.setSelected(selected, animated: animated)

        // Configure the view for the selected state
    }

}
```

Here we've made a couple of changes. First, we add an import statement at the beginner of the code. The other thing is the `mapView` outlet. Because the class `MKMapView` is made available through the `MapKit` package, we have to add the import statement.

Now go back to Interface Builder to establish the outlet connection. Select the prototype cell with the map view. Go to the Attributes inspector and change its identifier to `RestaurantDetailMapCell`.

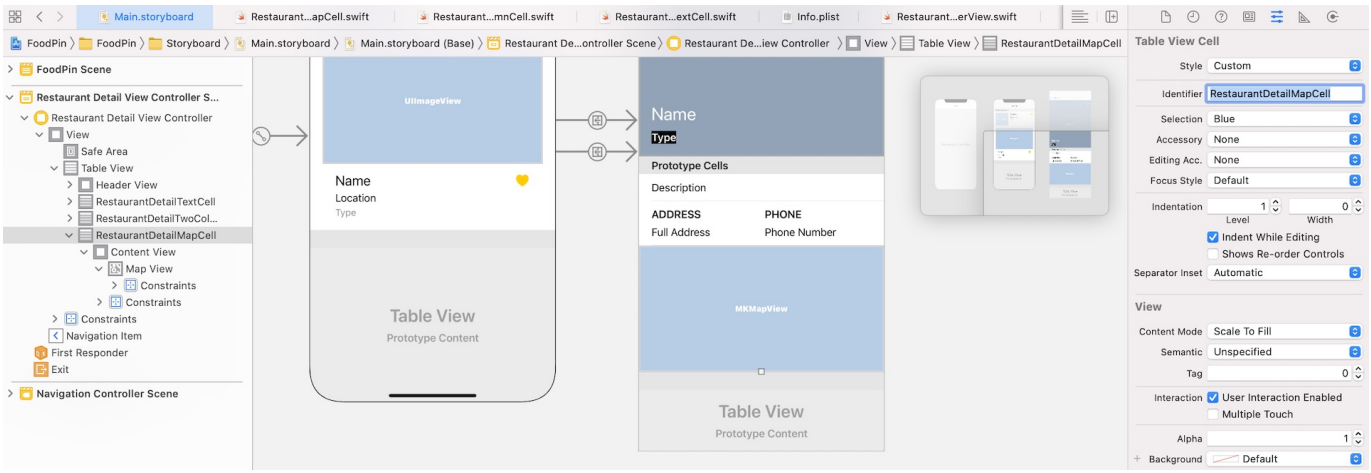


Figure 16-8. Changing the identifier of the prototype cell

In the Identity inspector, set the custom class to `RestaurantDetailMapCell`. Next, establish the connection between the `mapView` variable and the map view.

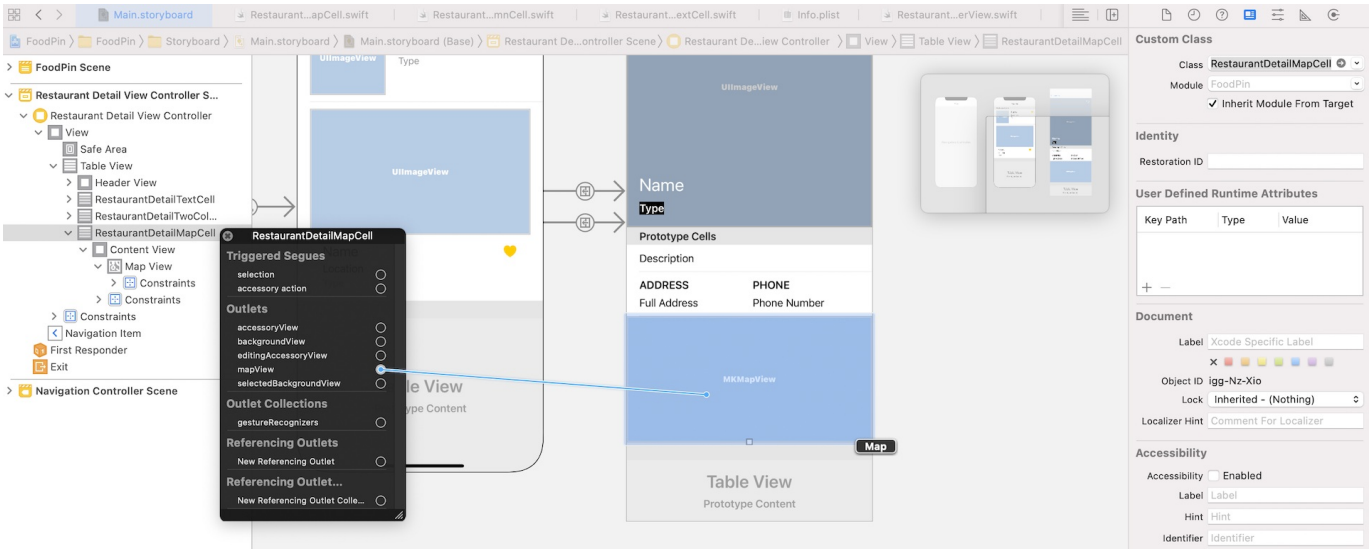


Figure 16-9. Establishing the connection between the outlet variable and the map view

Cool! You are almost done. With this new prototype cells, you have to make some changes to the `RestaurantDetailViewController.swift` file:

1. The total number of cells in the detail view is now changed from 2 to 3.
2. The `tableView(_:cellForRowAt:)` method should be updated because we have to return a different cell type for row #2.

Now open `RestaurantDetailViewController.swift` and update the total number of cells:

```
func tableView(_ tableView: UITableView, numberOfRowsInSection section: In
t) -> Int {
    return 3
}
```

Also, update the `tableView(_:cellForRowAt:)` method like this:

```
func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath)
 -> UITableViewCell {
    switch indexPath.row {
    case 0:
        let cell = tableView.dequeueReusableCell(withIdentifier: String(de
scribing: RestaurantDetailTextCell.self), for: indexPath) as! RestaurantDe
tailTextCell

        cell.descriptionLabel.text = restaurant.description

        return cell

    case 1:
        let cell = tableView.dequeueReusableCell(withIdentifier: String(de
scribing: RestaurantDetailTwoColumnCell.self), for: indexPath) as! Restaur
antDetailTwoColumnCell

        cell.column1TitleLabel.text = "Address"
        cell.column1TextLabel.text = restaurant.location
        cell.column2TitleLabel.text = "Phone"
        cell.column2TextLabel.text = restaurant.phone

        return cell

    case 2:
        let cell = tableView.dequeueReusableCell(withIdentifier: String(de
scribing: RestaurantDetailMapCell.self), for: indexPath) as! RestaurantDet
ailMapCell

        return cell

    default:
        fatalError("Failed to instantiate the table view cell for detail v
iew controller")

    }
}
```

In the code above, we just added a new case to handle the map cell. Let's hit the Run button to have a quick test. When you select a restaurant to navigate to the detail view, it shows a map. This is the power of MapKit. Without writing a line of code for the map view, you already embed a map within your app. Right now, it only displays a default map based on your current location. But it's really cool, right?

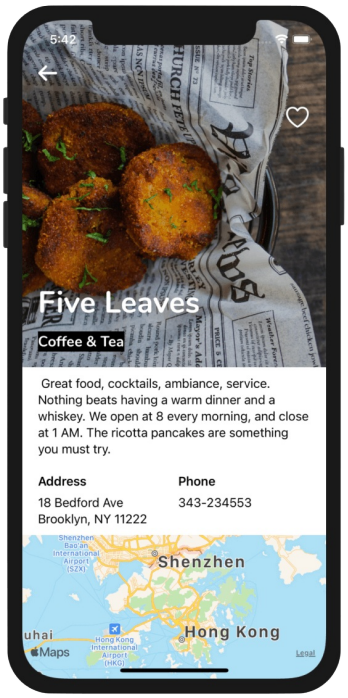


Figure 16-10. Now the detail view shows a map

Exercise #1

We haven't completed the implementation of the map view yet. Before we move on, let's have a simple exercise. Modify the layout constraints and corner radius of the map view to make it like the one shown in figure 16-11.

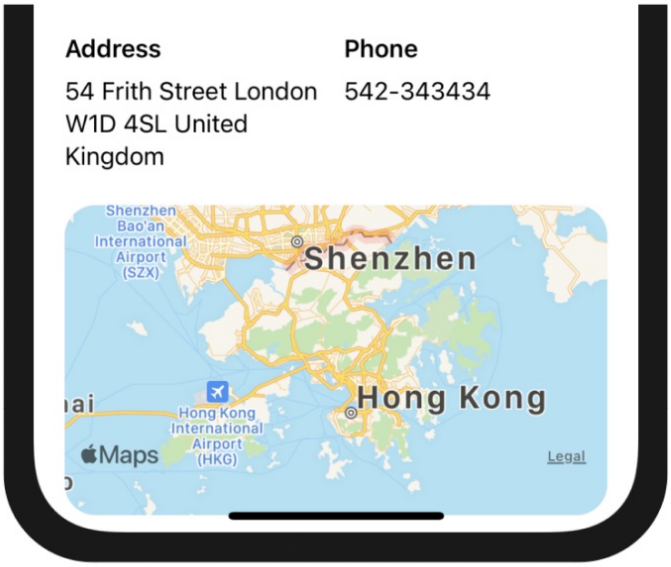


Figure 16-11. Now the detail view shows a map

If you can't figure out the solution, you can download the complete project at http://www.appcoda.com/resources/swift53/FoodPinRoundedMap.zip.

Displaying a Full Screen Map

We haven't completed the UI yet. When the user taps the map, it will navigate to another screen which shows a full-screen map. Now let's implement the UI of the full-screen map. For this map, we will implement it as a separate view controller. Open the Object library and drag a view controller to the storyboard.

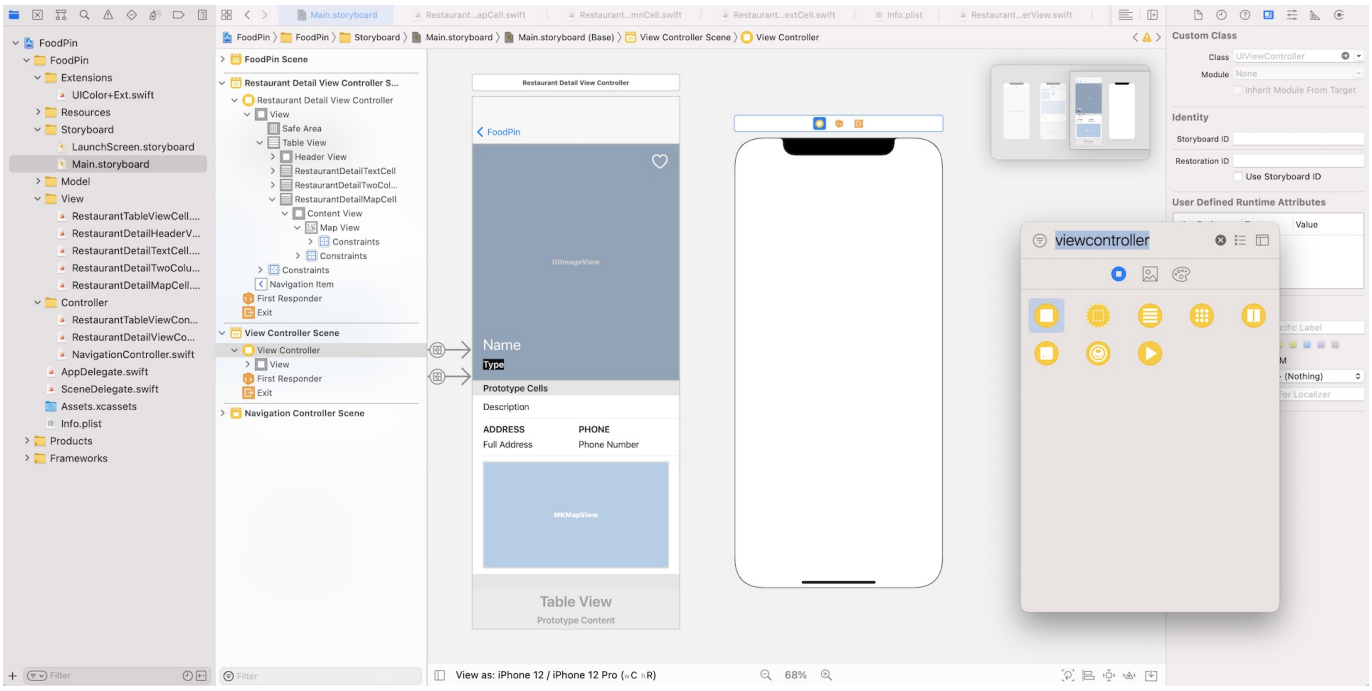


Figure 16-12. Adding a new view controller

Then drag a map view to the view controller and resize it to fit the view. Next, click the *Add New Constraints* button to add the required layout constraints. Set the value of top, left, bottom and right side to `0` and click *Add 4 Constraints* to confirm.

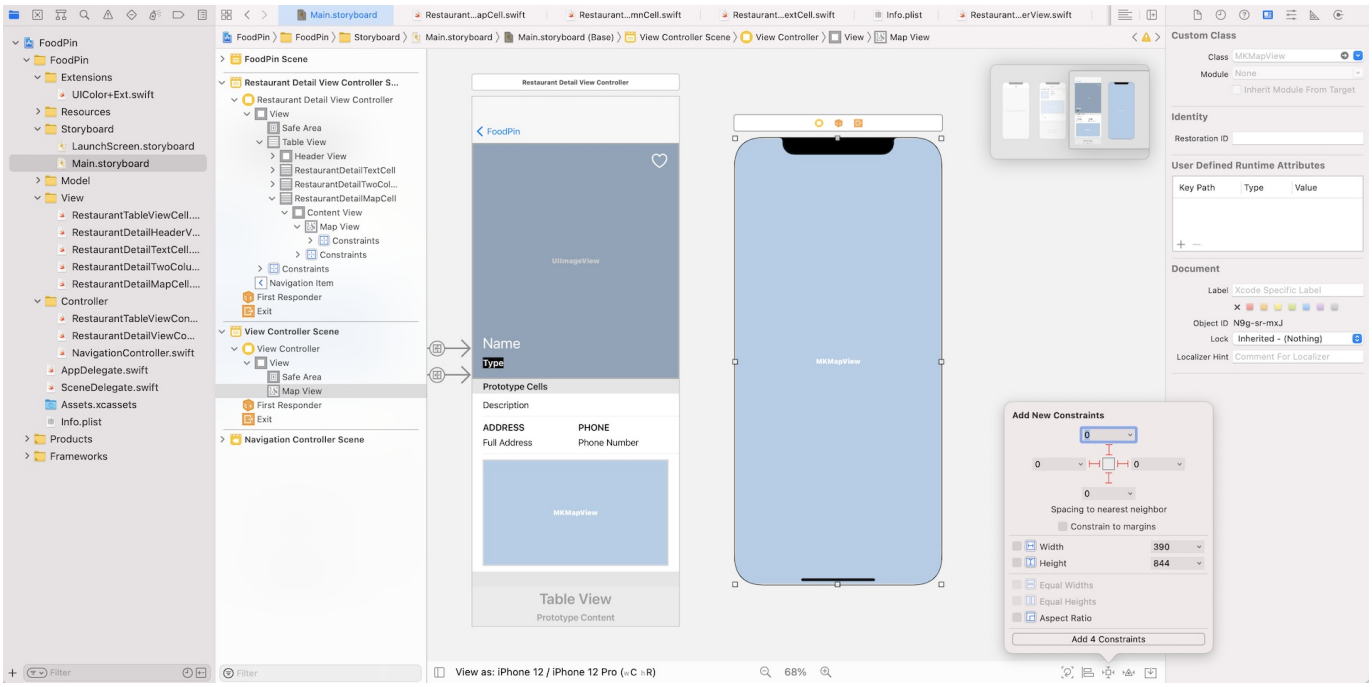


Figure 16-13. Adding spacing constraints for the map view

To bring up a full-screen map when a user taps the map cell, we have to set up a segue between the view controller and the cell. Now hold the *control* key and drag from `RestaurantDetailMapCell` to the new map view controller to create a segue. Select *Show* as the segue type. Once the segue is created, select it and set its identifier to `showMap` in the Attribute inspector.

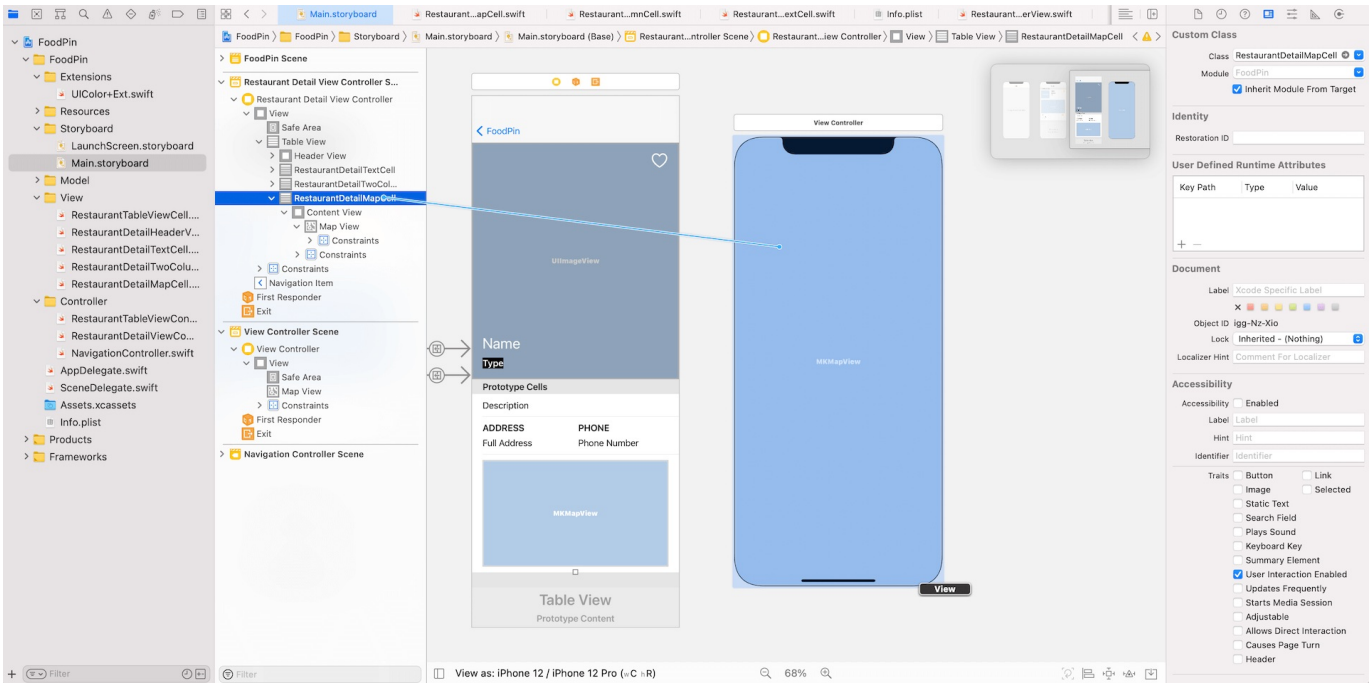


Figure 16-14. Creating a segue for the map view cell

If you compile and run the app, tap the map in the detail view. The app should navigate to a fully functional map.

Exercise #2

Here comes to another adhoc exercise. Again, it's about bug fixing. In the detailed view, did you try to tap the description or the address cell? When you tap any of the cells, the cell is highlighted in gray and it never switches back to white. We have discussed this problem before, your task is to figure out a solution for this issue.



Figure 16-15. Creating a segue for the map view cell

Converting an Address into Coordinates Using Geocoder

Now that you should know how to embed a map in your app, but how can you pin a location on maps? Before we pin the restaurant's location on the map, let's understand how to work with a location on maps.

To highlight a location on the map, you cannot just use a physical address. The MapKit framework doesn't work like that. Instead, the map has to know the geographical coordinates expressed in terms of the latitude and longitude of the corresponding point on the globe.

The framework provides a `Geocoder` class for developers to convert a textual address, known as *placemark*, into global coordinates. This process is usually referred to *forward geocoding*. Conversely, you can use `Geocoder` to convert

latitude and longitude values back to a *placemark*. This process is known as *reverse geocoding*.

To initiate a forward-geocoding request using the `CLGeocoder` class, all you need do is to create an instance of `CLGeocoder`, followed by calling the `geocodeAddressString` method with the address parameter. Here is an example:

```
let geoCoder = CLGeocoder()
geoCoder.geocodeAddressString("524 Ct St, Brooklyn, NY 11231", completionH
andler: { placemarks, error in

// Process the placemark

})
```

There is no designated format of an address string. The method submits the specified location data to the geocoding server asynchronously. The server then parses the address and returns you an array of placemark objects. The number of placemark objects returned greatly depends on the address you provide. The more specific the address information you have given, the better the result. If your address is not specific enough, you may end up with multiple placemark objects.

With the `placemark` object, which is an instance of `CLPlacemark` class, you can easily get the geographical coordinates of the address using the code below:

```
let coordinate = placemark.location?.coordinate
```

The completion handler is the code block to be executed after the forward-geocoding request completes. Operations like annotating the placemark will be done in the code block.

A Quick Overview of Map Annotations

Now that you have a basic idea of `Geocoder` and understand how to get the global coordinates of an address, we will look at how you can pin a location on maps. To do that, the MapKit framework provides an annotation feature for you to pinpoint a specific location.

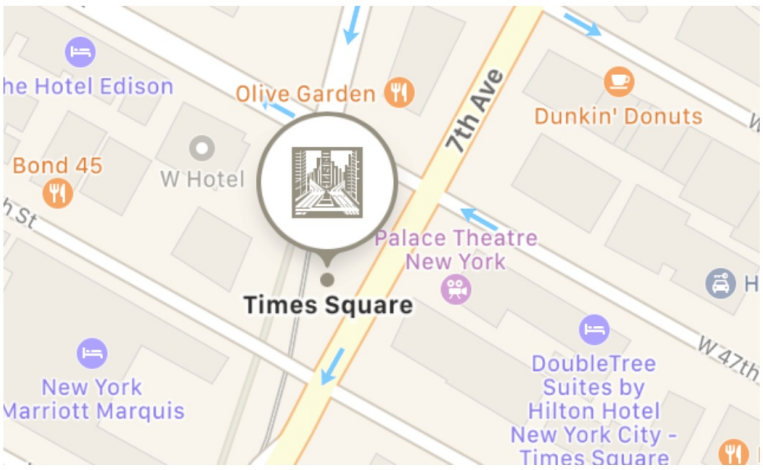


Figure 16-16. An annotation in iOS

An annotation can appear in many forms. Prior to iOS 11, the annotation appears in the form of a pin. In the latest version of the MapKit framework, Apple has redesigned the default annotation to make it look more modern. Typically an annotation consists of an icon (e.g. pin or custom image) and a title.

From a developer's point of view, an annotation actually consists of two different objects:

- **an annotation object** - which stores the data of an annotation such as the name of the placemark. The object should conform to the `MKAnnotation` protocol as defined in the Map Kit.
- **an annotation view** - which is the actual object for the visual representation of the annotation. The pin image is an example. If you want to display the annotation in your own form (say, use pencil instead of pin), you'll need to

create your own annotation view.

The MapKit framework comes with a standard annotation object and an annotation view (known as `MarkerAnnotationView`). Therefore, you do not need to create your own, unless you want to customize the annotation.

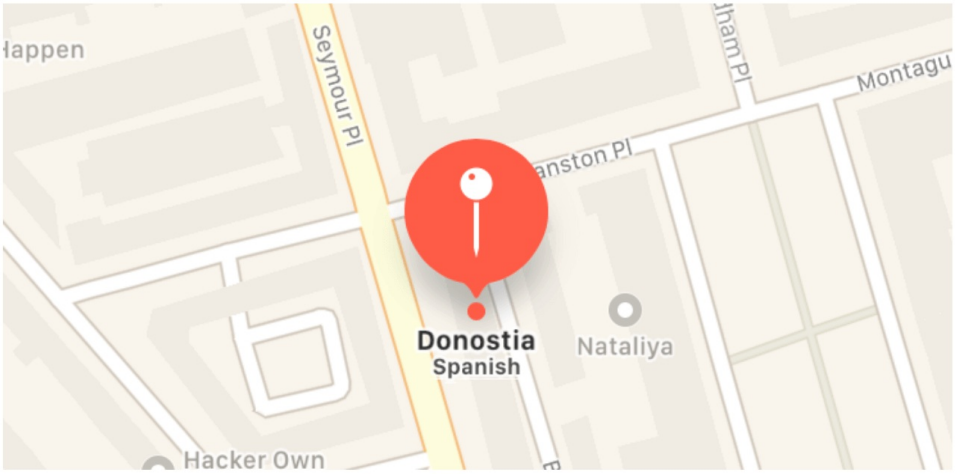


Figure 16-17. The default annotation

In the simplest form, a standard annotation appears like this one shown in figure 16-17. To add a pin on a map, you just need a few lines of code:

```
let annotation = MKPointAnnotation()
if let location = placemark.location {
    annotation.coordinate = location.coordinate
    mapView.addAnnotation(annotation)
}
```

The `MKPointAnnotation` class is a standard class, which adopts the `MKAnnotation` protocol. By specifying the coordinates in the annotation object, you can call the `addAnnotation` method of the `mapView` object to put a pin on the map.

Adding an Annotation to the Map

After introducing the basics of annotations and geocoding, let's get back to the FoodPin project. We first add a pin annotation to the map in the table cell.

In `RestaurantDetailMapCell.swift`, insert the following code method:

```
func configure(location: String) {
    // Get location
    let geoCoder = CLGeocoder()

    print(location)
    geoCoder.geocodeAddressString(location, completionHandler: { placemark
s, error in
        if let error = error {
            print(error.localizedDescription)
            return
        }

        if let placemarks = placemarks {
            // Get the first placemark
            let placemark = placemarks[0]

            // Add annotation
            let annotation = MKPointAnnotation()

            if let location = placemark.location {
                // Display the annotation
                annotation.coordinate = location.coordinate
                self.mapView.addAnnotation(annotation)

                // Set the zoom level
                let region = MKCoordinateRegion(center: annotation.coordin
ate, latitudinalMeters: 250, longitudinalMeters: 250)
                self.mapView.setRegion(region, animated: false)
            }
        }

    })
}
```

This map cell is responsible to handle the map view and display an annotation on the map. So, we add a new method called `configure` that accepts the restaurant's address as the parameter. I will not go into the code above line by line as we've

discussed the usage of `Geocoder` and annotation earlier. In brief, we first convert the address of the selected restaurant (i.e. `location`) into coordinates using `Geocoder`. In most cases, the placemarks array should contain a single entry. So we just pick the first element from the array and then display the annotation on the map view.

The last two lines of code are new to you. Here we use the `MKCoordinateRegion` function to adjust the initial zoom level of the map to 250m radius.

> **Note:** You can still annotate the location without these two lines of code. Try to remove these lines of code and see how the map differs.

Now edit the `RestaurantDetailViewController.swift` file, update `case 2` of the `tableView(_:cellForRowAt:)` method like this:

```
case 2:
    let cell = tableView.dequeueReusableCell(withIdentifier: String(descri
bing: RestaurantDetailMapCell.self), for: indexPath) as! RestaurantDetailM
apCell
    cell.configure(location: restaurant.location)
    cell.selectionStyle = .none

    return cell
```

We just add a line of code to call the `configure` method with the restaurant's location. If you run the app now, it should correctly display the restaurant's location on the map.

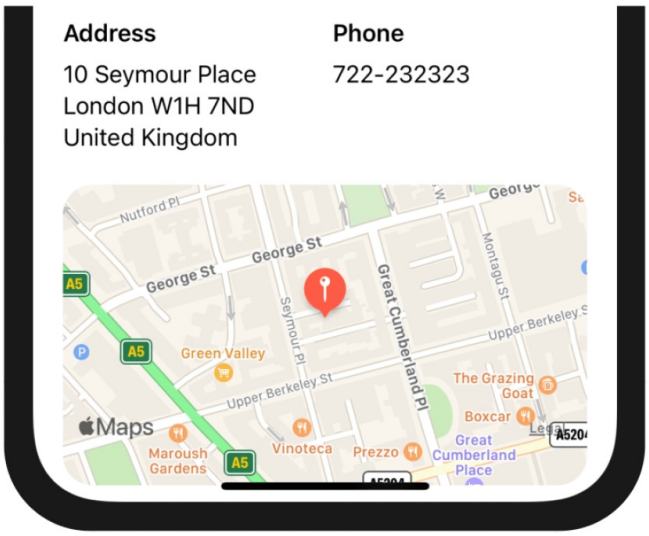


Figure 16-18. Pin the restaurant location on a map

Adding an Annotation to the Full Screen Map

Okay, let's move onto the implementation of the full-screen map. As usual, we first create a custom class for the map view controller. In the project navigator, right click the `Controller` folder and select *New File*.... Create the new class using the *Cocoa Touch class* template and name the class `MapViewController`. Make sure you set it as a subclass of `UIViewController` and save the file.

Again, we first import the MapKit framework. Insert the following line of code at the very beginning of the `MapViewController.swift` file:

```
import MapKit
```

Next, declare the following outlet variable for the map view and another variable for the selected restaurant:

```
@IBOutlet var mapView: MKMapView!

var restaurant = Restaurant()
```

The outlet variable is used for establishing a connection with the map view in the storyboard. Go to Interface Builder and select the map view controller. Under the Identity inspector, set the custom class to `MapViewController`. Then establish a connection between the outlet variable and the map view.

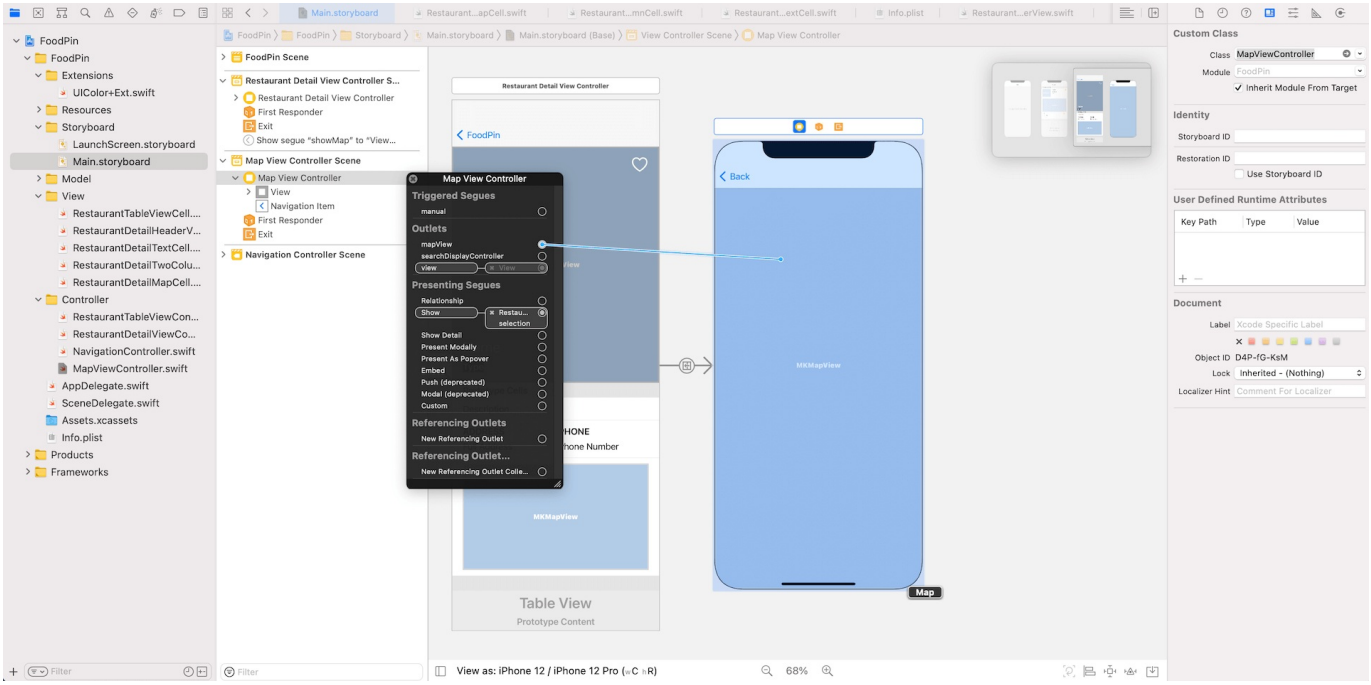


Figure 16-19. Establish a connection between MKMapView and the outlet variable

To add an annotation on the map, update the `viewDidLoad` method of the `MapViewController` class to the following code:

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

    // Convert address to coordinate and annotate it on map
    let geoCoder = CLGeocoder()
    geoCoder.geocodeAddressString(restaurant.location, completionHandler:
{ placemarks, error in
        if let error = error {
            print(error)
            return
        }

        if let placemarks = placemarks {
            // Get the first placemark
            let placemark = placemarks[0]

            // Add annotation
            let annotation = MKPointAnnotation()
            annotation.title = self.restaurant.name
            annotation.subtitle = self.restaurant.type

            if let location = placemark.location {
                annotation.coordinate = location.coordinate

                // Display the annotation
                self.mapView.showAnnotations([annotation], animated: true)
                self.mapView.selectAnnotation(annotation, animated: true)
            }
        }

    })

}
```

The above code is very similar to the one we have just discussed, so I will not go into the code above line by line. We convert the address into a coordinate for annotation. Once again, we use `MKPointAnnotation` to add a pin to the map. But this

time we assign it with a title and a subtitle and use the `showAnnotations` method of the `mapView` object to put a pin on the map. The method is smart enough to determine the best-fit region for the annotation.

The annotation marker has three states. By default, when the annotation marker is not selected, it is in a normal state that appears smaller. In the code above, we invoke the `selectAnnotation` method to select the annotation marker to turn it into the selected state. Its icon will grow larger in this state. You will better understand what I mean about the size when you test the app.

There is still one thing left before testing the app. We haven't passed the selected restaurant to the map view controller. In the `RestaurantDetailViewController` class, insert the `prepare(for:sender:)` method like this:

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    if segue.identifier == "showMap" {
        let destinationController = segue.destination as! MapViewController

        destinationController.restaurant = restaurant
    }
}
```

We simply get the selected restaurant and pass it to the destination view controller. In this case, it's the `MapViewController` class. Okay, let's compile and run the app. Tap the Map button in the detail view and you'll see a pin on the map.

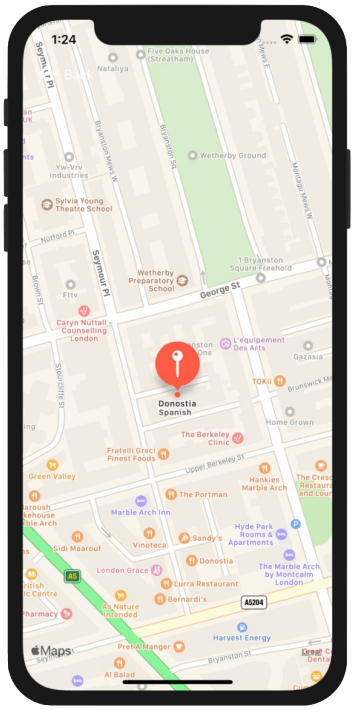


Figure 16-20. Tapping the map button now shows a map with the restaurant's location

Customizing the Annotation

You are allowed to change the look and feel of the Marker Annotation View (i.e. `MKMarkerAnnotationView`) like customizing its color and icon. If you go up to read the API documentation of `MKMarkerAnnotationView` (https://developer.apple.com/documentation/mapkit/mkmarkerannotationview), you will find several properties for customization. Here are some examples:

- `markerTintColor` - the background color of the marker balloon
- `glyphText` - the text to display in the marker balloon
- `glyphImage` - the image displayed in the marker balloon

As mentioned at the beginning of this chapter, an annotation view controls the visual part of an annotation. In order to modify the appearance of the annotation, we have to modify the annotation view. To do that, we have to adopt the `MKMapViewDelegate` protocol, which defines a set of optional methods that you can use to receive map-related update messages. The map view uses one of these methods to request annotation. Every time when the map view needs an annotation view, it calls the `mapView(_:viewFor:)` method:

```
optional func mapView(_ mapView: MKMapView, viewFor annotation: MKAnnotati
on) -> MKAnnotationView?
```

So far we haven't adopted the protocol and provided our own implementation for the method. This is why a default annotation view is displayed. We're going to implement the method and customize the color & icon of annotation view:

Go back to the `MapViewController.swift` file and adopt the `MKMapViewDelegate` protocol using an extension like this:

```
extension MapViewController: MKMapViewDelegate {

    func mapView(_ mapView: MKMapView, viewFor annotation: MKAnnotation) -
> MKAnnotationView? {
        let identifier = "MyMarker"

        if annotation.isKind(of: MKUserLocation.self) {
            return nil
        }

        // Reuse the annotation if possible
        var annotationView: MKMarkerAnnotationView? = mapView.dequeueReusa
bleAnnotationView(withIdentifier: identifier) as? MKMarkerAnnotationView

        if annotationView == nil {
            annotationView = MKMarkerAnnotationView(annotation: annotation
, reuseIdentifier: identifier)
        }

        annotationView?.glyphText = "   "
        annotationView?.markerTintColor = UIColor.orange

        return annotationView
    }
}
```

Let's go through the above code line by line. Every time when the map view needs to display an annotation, the method above will be called. Earlier, we haven't implemented this method. In that case, the map view will use the default annotation. Basically, there are two types of annotations:

- Placemark
- Current location

The user's current location is also a kind of annotation. The map view will also call this method when annotating the user's location. As you may know, the current location is displayed as a blue dot in maps. Even though we haven't enabled the app to display the current location, we don't want to change its annotation view. This is why we verify if the annotation object is a kind of `MKUserLocation`. If yes, we simply return `nil` and then the map view will just display the location using a blue dot.

For performance reasons, it is preferred to reuse an existing annotation view instead of creating a new one. The map view is intelligent enough to cache unused annotation views that it isn't using. Similar to `UITableView`, we can call up the `dequeueReusableAnnotationView(withIdentifier:)` method to see if any unused views are available. If yes, we then downcast it to `MKMarkerAnnotationView`.

If there are no unused views available, we create a new one by instantiating a `MKMarkerAnnotationView` object. Once we get hold of the annotation view, we modify its `glphyText` and `markerTintColor` property to customize the look and feel.

In the `viewDidLoad` method, add the following line of code to define the delegate of `mapView`:

```
mapView.delegate = self
```

Here we define MapViewController as the delegate object of mapView.

Cool! We're done. Press the Run button and launch the app. Pick a restaurant and tap the Map button to see the modified annotation.



Figure 16-21. An annotation with an emoji icon

If you want to replace the annotation with an image, you replace `glyphText` with `glypImage` like this:

```
annotationView?.glyphImage = UIImage(systemName: "arrowtriangle.down.circle")
```

Map Customizations

Apple also lets developers control what goes on the map view. Here are the three new properties for you to control the content of a map view:

- **showTraffic** - shows any high traffic on your map view
- **showScale** - shows a scale on the top-left corner of your map view.
- **showCompass** - displays a compass control on the top-right corner of your map view. Please note that the compass will only appear when the map is rotated a little bit away from pure north.

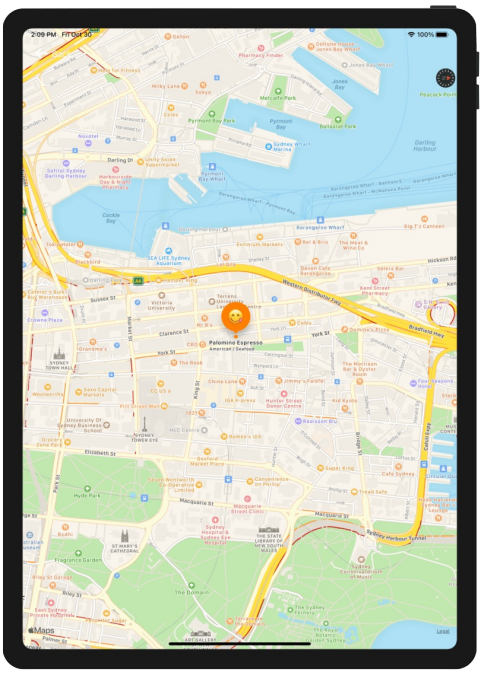


Figure 16-22. Show traffic, scale and compass in the map

As a demo, you can insert the following lines of code in the `viewDidLoad` method to give it a try:

```
mapView.showsCompass = true
mapView.showsScale = true
mapView.showsTraffic = true
```

If you run the app on the simulator, you may not see the compass icon when the map is first loaded up. You can hold the Option key and drag the map to reveal it.

Exercise #3

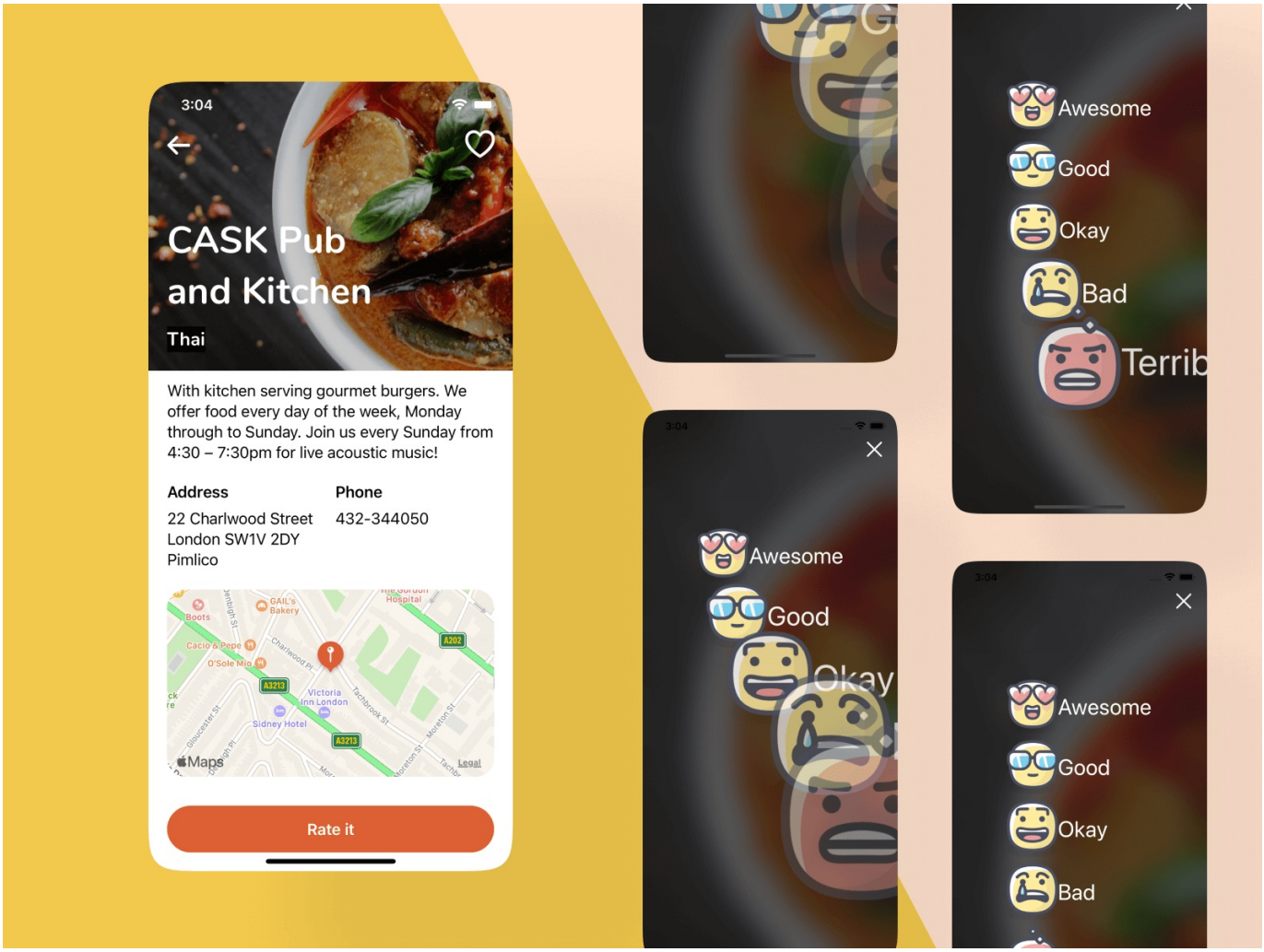
Here comes to the final exercise. In the map view controller, the back button in the navigation bar shows the title "Back". Your task is to modify the code to remove the title.

Summary

In this chapter, I've walked you through the basics of the MapKit framework. By now, you should know how to embed a map in your app and add an annotation. But this is just a beginning. There is a lot more you can explore from here. One thing you can do is to further explore MKDirection . It provides you with route-based directions data from Apple servers to get travel-time information or driving or walking directions. You can take the app further by showing directions.

For your reference, you can download the complete Xcode project from http://www.appcoda.com/resources/swift53/FoodPinMaps.zip.

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.
>
> – Walt 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.

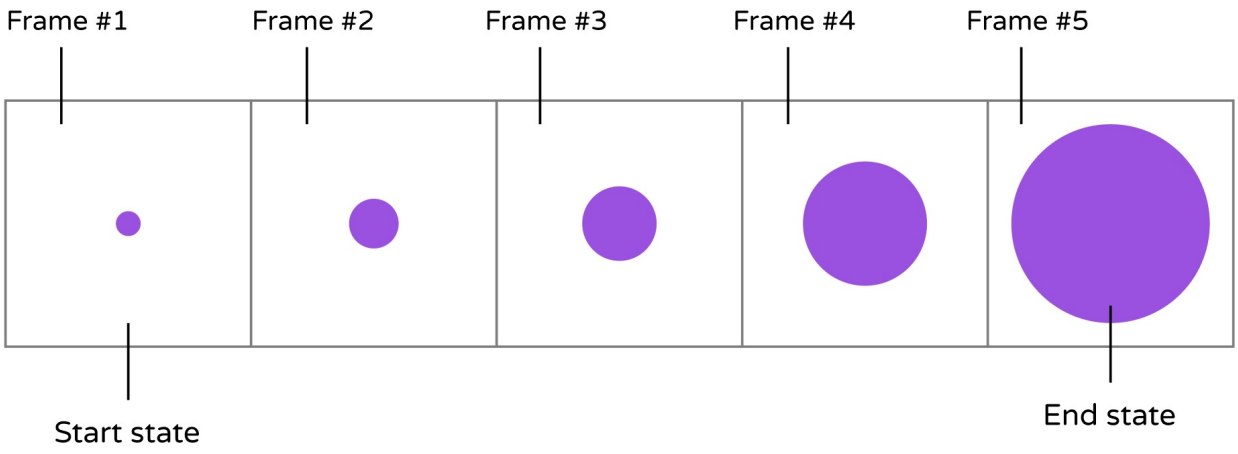


Figure 17-1. Sequence of frames for creating an animation

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.

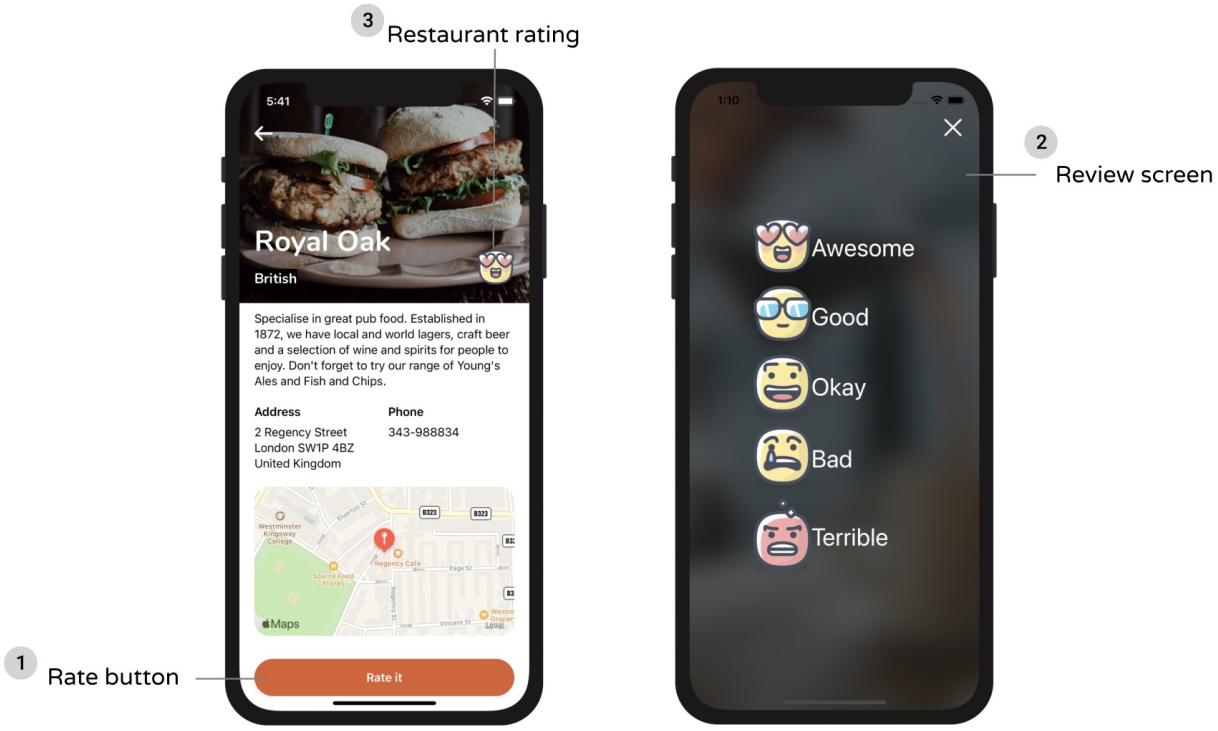


Figure 17-2. Adding a rating feature in the detail screen

Adding a Rating button

Before creating the animated views, we'll add a rating button to the detail view controller. First, download this image pack (http://www.appcoda.com/resources/swift4/FoodPinRatingButtons.zip) and add the icons to `Assets.xcassets`. Optionally, you can create a folder in the asset catalog to better organize the images.

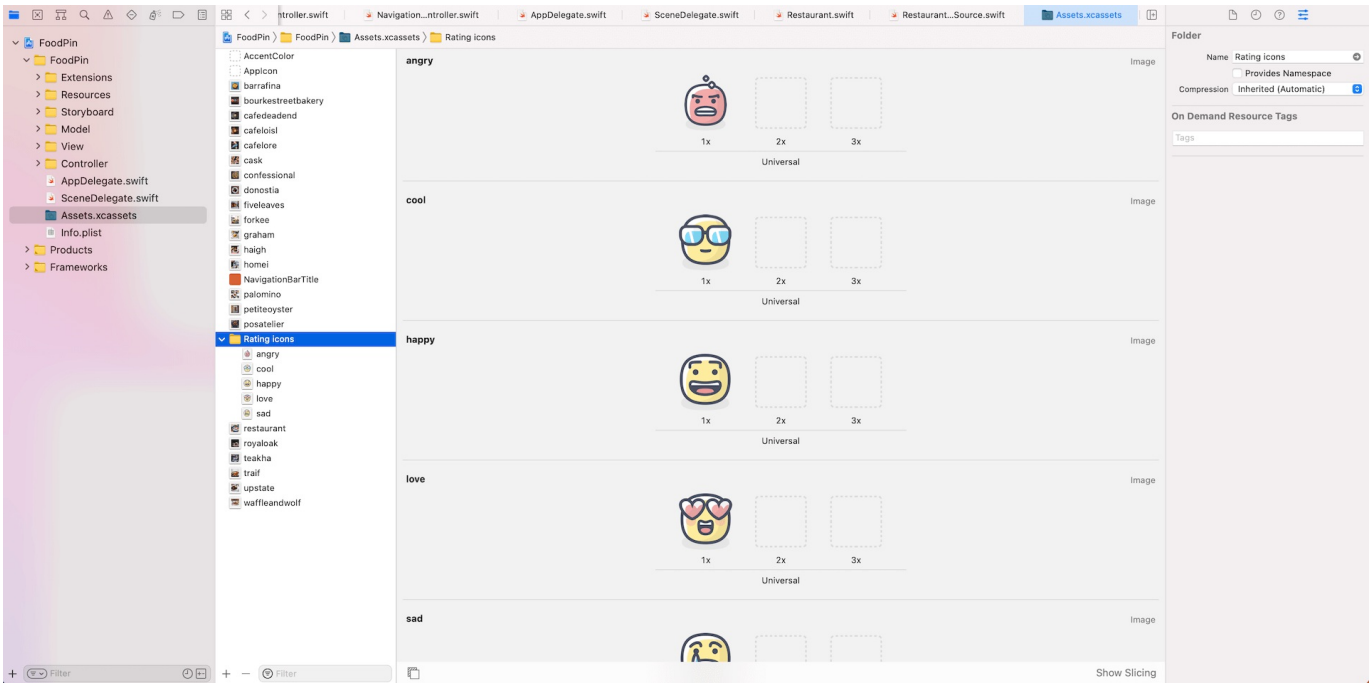


Figure 17-3. Adding icons to the asset catalog

> **Credit:** The icons are made by Pat Johnson.

You've learned how to use the table view's header. This time, we will add the *Rate it* button to the footer of the table view. Go to `Main.storyboard` and select the *Restaurant Detail View Controller* in document outline. For convenience sake, before we add the rate button, let's first make the controller a little bit longer. In the Size inspector, set the simulated size to *Freeform* and height to `1200`.

Next, drag a View object from the Object library to the footer of the table view. Place the view right under `RestaurantDetailMapCell` in document outline. This view object serves as a container holding the rate button. If you've done it correctly, you should see a white area below the map view. The default size of the view is a bit large. Go to the Size inspector and set the height to `90`.

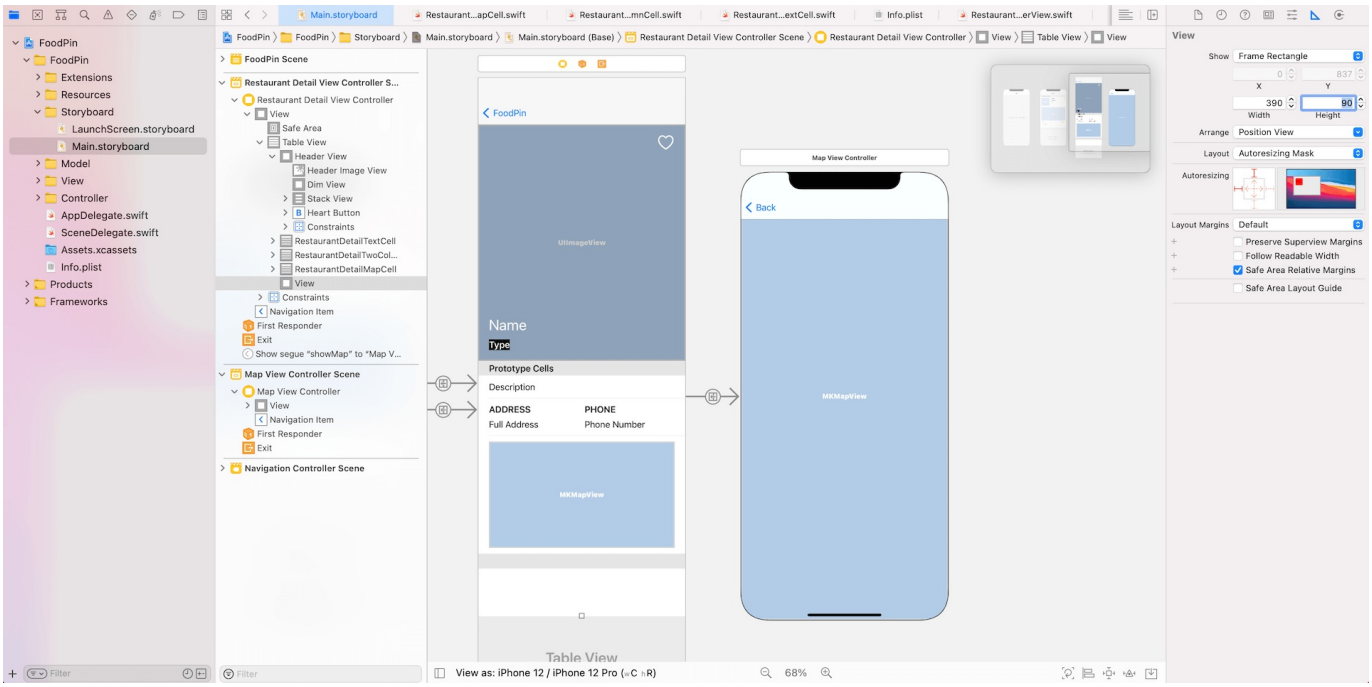


Figure 17-4. Adding a review button in the detail view

Next, drag a button from the Object library to the view we just added. In the Attributes inspector, set the title to `Rate it`. Change the font to *Headline - Text Style* and color to *white*. For the background color, change it to `NavigationBarTitle` color.

As usual, we need to add some layout constraints for the button. Select the button and click the *Add New Constraints* button of the layout bar. Set the value of all sides to `20` and deselect the *Constrain to margins* option. Click *Add 4 Constraints* to confirm.

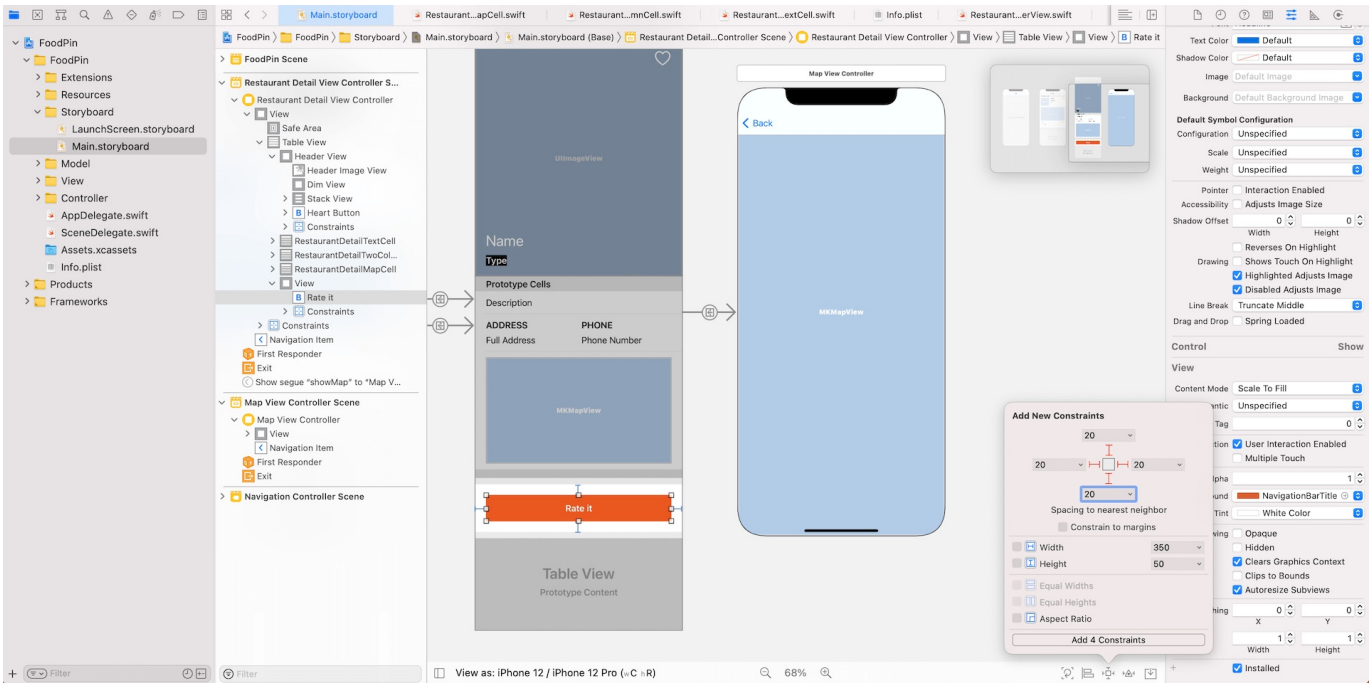


Figure 17-5. Adding layout constraints for the button

As you may notice, the *Rate it* button has rounded corners. You may implement it by writing code. Now I'd like to show another way to modify the corner radius of the layer. First, make sure you select the *Rate it* button. Go to the Identity inspector and click the `+` button under the *User Defined Runtime Attributes* section. Then enter `cornerRadius` in the Key Path column. Change the *type* to `Number` and set its *value* to `25`.

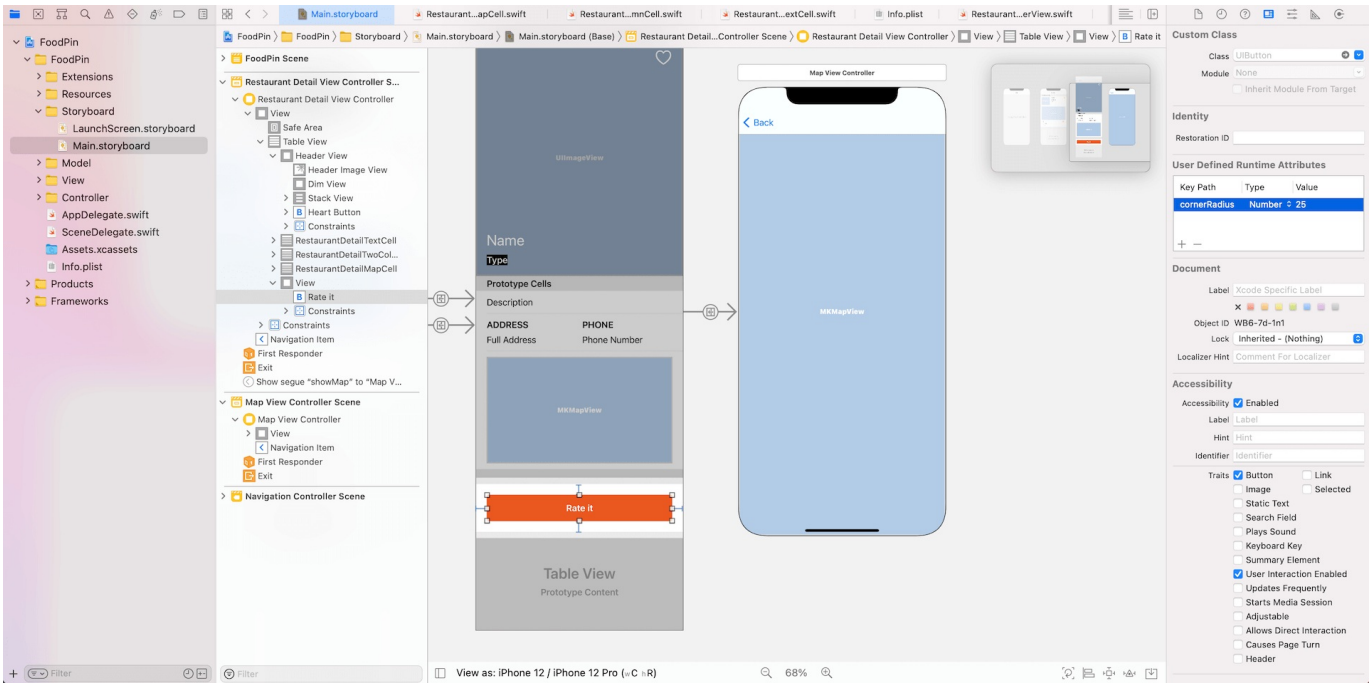


Figure 17-6. Adding a runtime attribute

Once done, run the app to have a quick test. Your detail view should look very similar to that in figure 17-7.

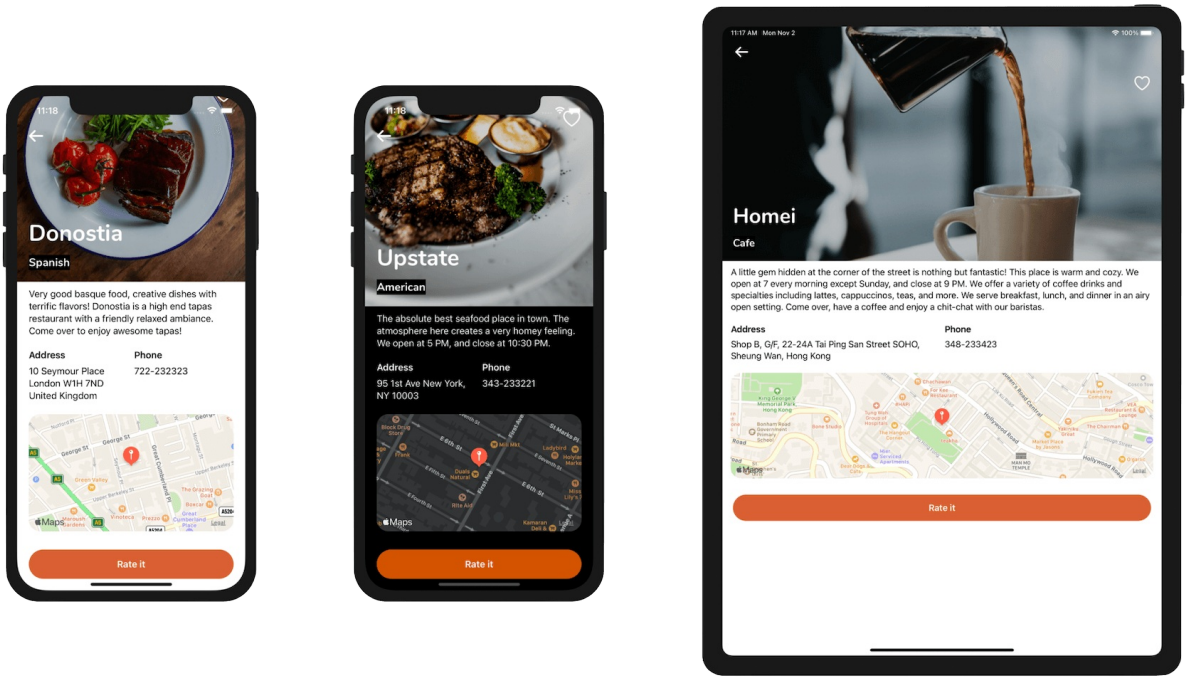


Figure 17-7. The Rate button in the detail view

Create a View Controller for Restaurant Review

When a user taps the *Rate it* button, we want to bring up a modal view for the user to give a rating for the restaurant. If you forgot how the review screen looks like, refer to figure 17-2.

In Interface Builder, drag a new view controller from the Object library to the storyboard. Add an image view to the view and set its content mode to *Aspect fill*. The image view is used for displaying a background image. Later you will learn how to blur the image to create a blurred background.

As always, we need to define some layout constraints for the UI component. Click the *Add New Constraints* button. Set the value of the top, left, bottom and right sides to `0`. Press *Add 4 Constraints* to confirm.

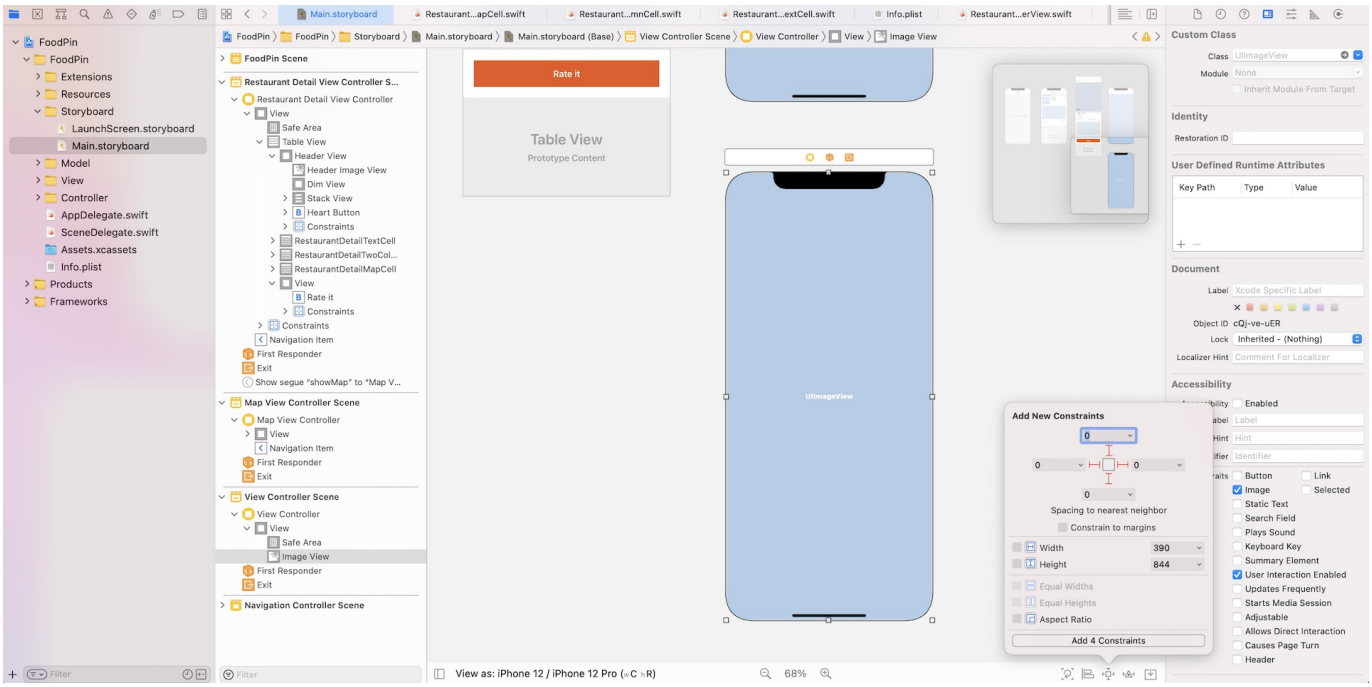


Figure 17-8. Adding missing constraints

Next, drag a button from the Object library and place it near the top-right corner. This button is a close button. In the Attributes inspector, set its title to blank and image to `xmark`. You can directly use the SF Symbols in Interface Builder. `xmark` is the name of the `X` symbol. To make it a little larger, you can change the configuration option to `Point Size` and set it to `30` points. Also, scroll down to the View section and change the tint color to `white`.

Again, we need to add a couple of spacing constraints for the top and right side. You can refer to figure 17-9 for details.

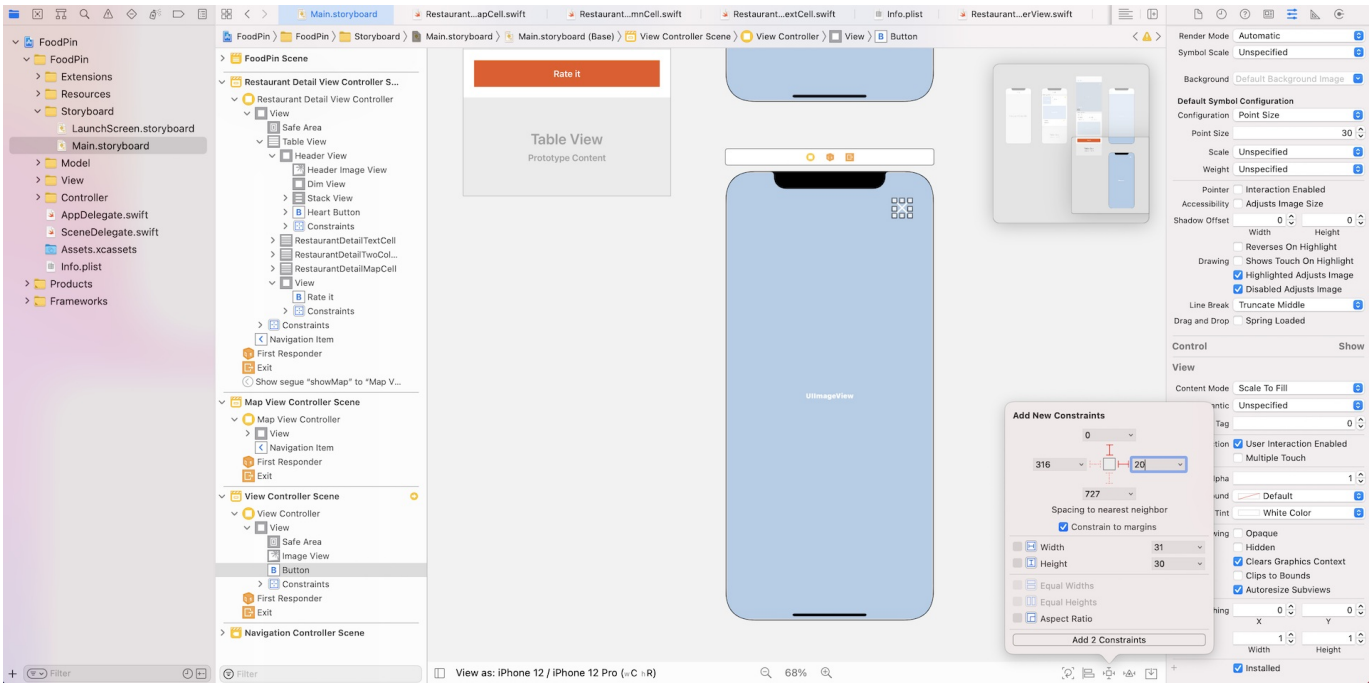


Figure 17-9. UI design of the review view

Now we are going to lay out the rating buttons. Drag a button from the Object library to the view controller. In earlier chapters, we either worked with an image button or a text button. This time, it is a little bit different. The button has both text and image. In the Attributes inspector, set the title to `Love` and image to `Awesome`. Change the font to the text style `Large Title`.



Figure 17-10. Adding a view to the view controller

As the other 4 buttons are similar to the one we created, we can just do "copy & paste" and change each of the title and image accordingly:

- For the second button, set the title `Cool` and image to `Good`.
- For the third button, set the title `Happy` and image to `Okay`.
- For the fourth button, set the title `Sad` and image to `Bad`.
- For the fifth button, set the title `Angry` and image to `Terrible`.

Your buttons should be similar to that in figure 17-11.

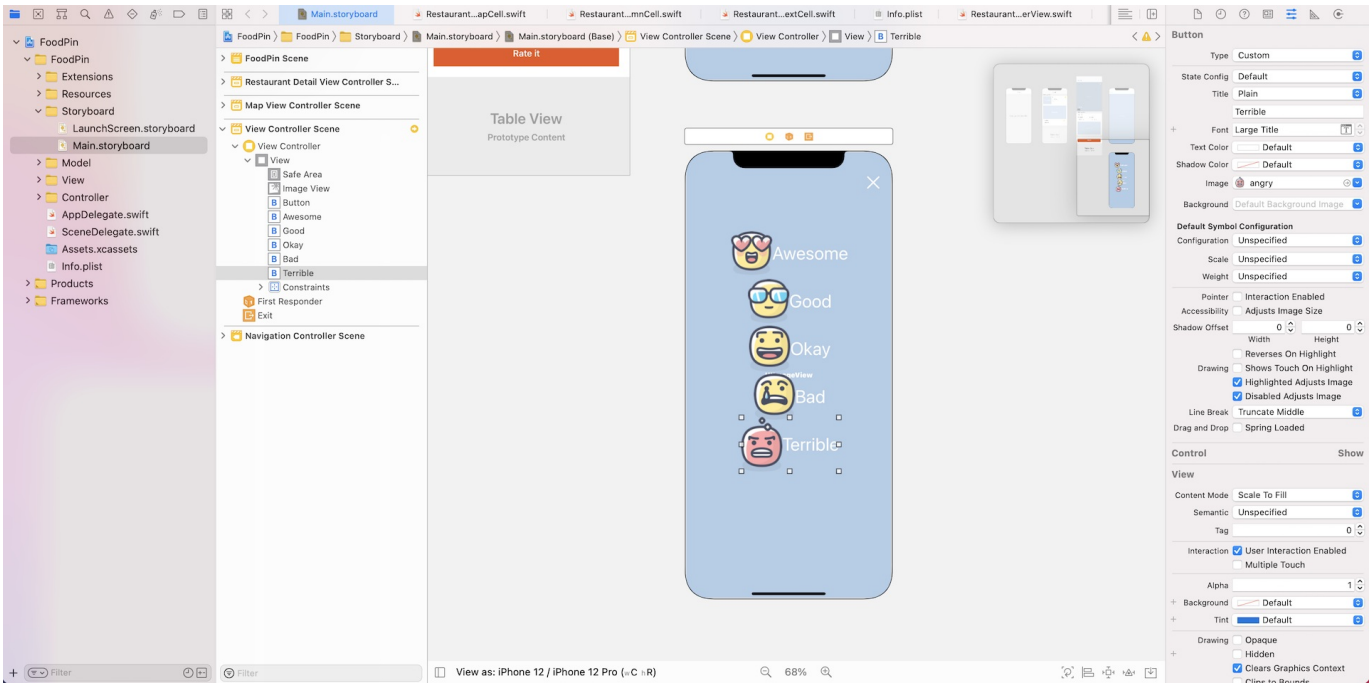


Figure 17-11. Rating buttons

Again, it's time to define the layout constraints for the UI objects. First, let's embed the buttons in a stack view, and then define the necessary constraints. Hold the *command* key and select all the five buttons. Click the *Embed in* button in the layout bar and choose *Stack view* to embed them in a vertical stack view. In the Attributes inspector, change the Alignment option to *Leading* and spacing to `10` points.

We want to center the stack view both vertically and horizontally. Make sure you select the stack view. Click the *Align* button, and check both *Horizontally in Container* and *Vertically in Container* checkboxes. Once you confirm to add the constraints, the stack view is now always centered.

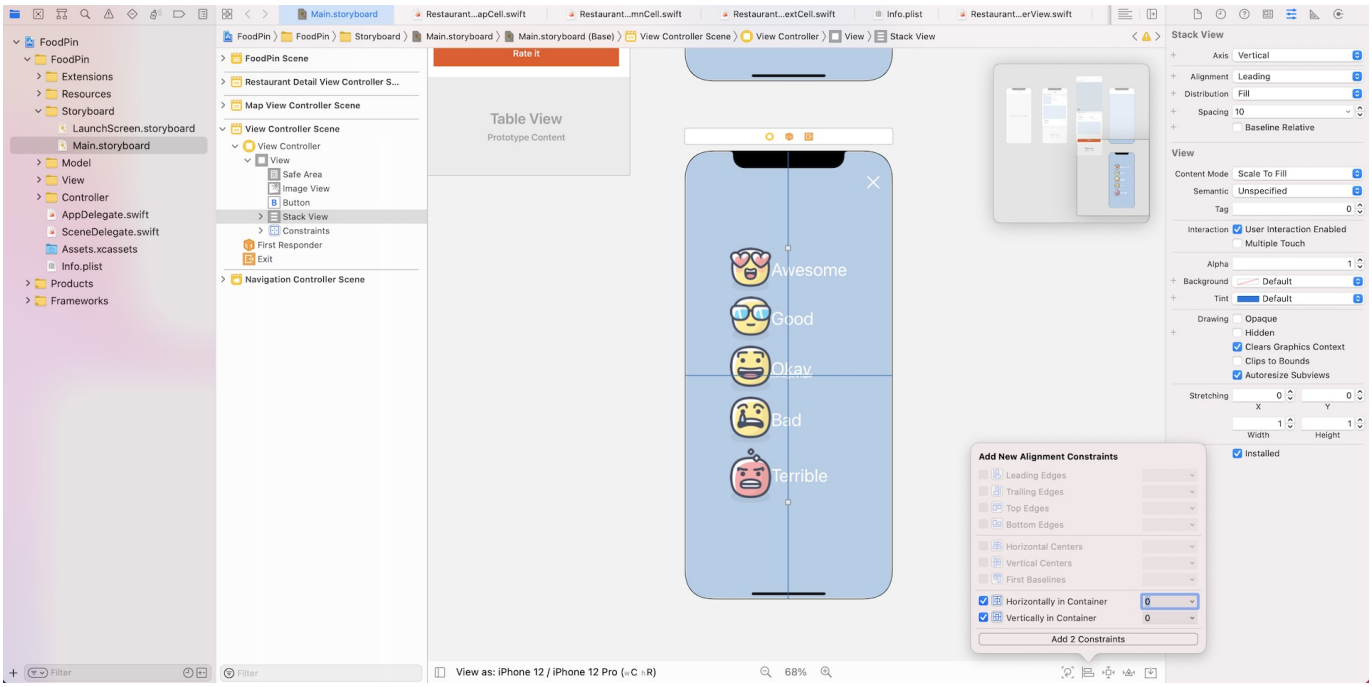


Figure 17-12. Adding layout constraints for the container view

Now that we've completed the design of the view controller, let's create a class to pair with it. In the project navigator, right-click the `Controller` folder and choose *New File...*. Choose the Cocoa Touch Class template. Set the class name `ReviewViewController` and its subclass to `UIViewController`.

Once the file is created, declare an outlet variable for the background image view in the `ReviewViewController` class. Also, declare a variable for holding the current restaurant:

```
@IBOutlet var backgroundImageView: UIImageView!

var restaurant = Restaurant()
```

In the `viewDidLoad()` method, insert the following line of code to set the image view:

```
backgroundImageView.image = UIImage(named: restaurant.image)
```

This line of code loads the restaurant's image as the background image. Switch over to `Main.storyboard` and select the view controller with the rating buttons. In the Identify inspector, change the custom class to `ReviewViewController`. Then right-click the Review View Controller in document outline and connect the `backgroundImageView` outlet variable.

Create a Segue for the Modal View

To bring up the review view modally, we have to connect the *Rate it* button with the review view controller with a segue. Hold the control key and drag from the *Rate it* button to the review view controller. Release the buttons and select *Present modally* as the segue type. Once the segue is created, select it and set the identifier of the segue to `showReview` under the Attribute inspector.

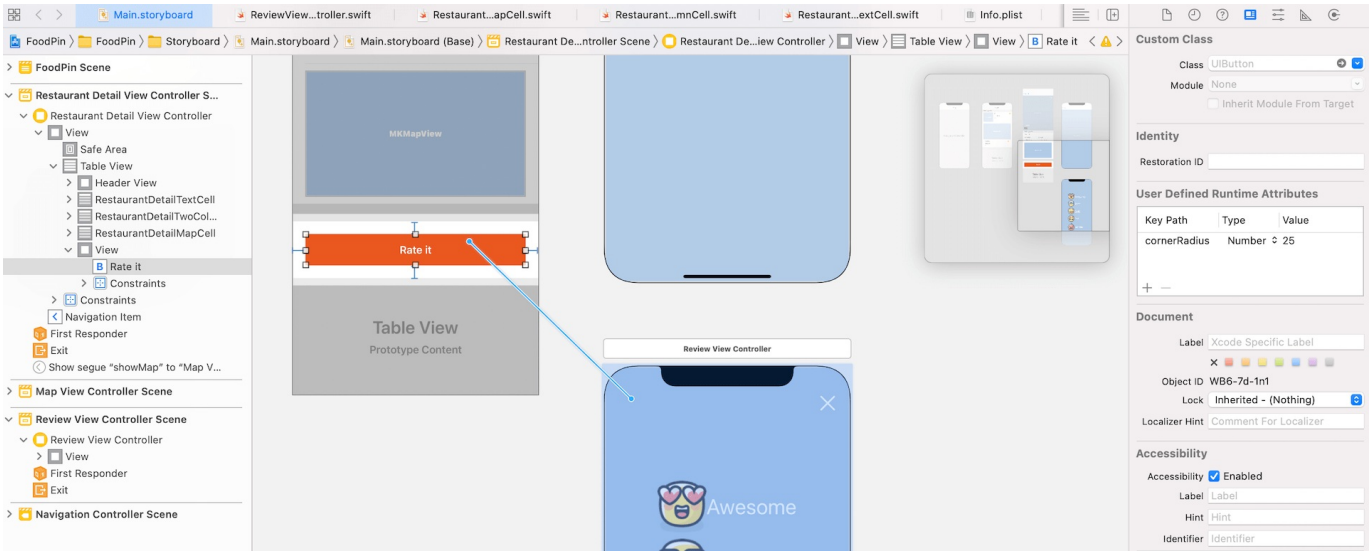


Figure 17-13. Create a segue to present the review view controller modally

As we need to pass the restaurant object from the detail view to the review view, open the `RestaurantDetailViewController.swift` file and edit the following method like this:

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    switch segue.identifier {
    case "showMap":
        let destinationController = segue.destination as! MapViewController

        destinationController.restaurant = restaurant

    case "showReview":
        let destinationController = segue.destination as! ReviewViewContro
ller
        destinationController.restaurant = restaurant

    default: break
    }
}
```

The change is very straightforward. For segue with the identifier `showReview`, we retrieve the `ReviewViewController` object and then pass it the current restaurant. Now let's have a quick test. Run the app and go to the detail view. Tapping the *Rate it* button should now bring up the review view.



Figure 17-14. The review view

Starting from iOS 13, the presentation of a modal view has changed. iOS doesn't show you a full screen modal view. Instead, when a view is presented modally, the system displays a card like modal view with the original view sits behind the modal view. What's more is that it has gesture support built-in. Swiping down from the top of the screen will close the modal view.

In case you don't like this default behaviour and want to change it back to full screen mode, you can select the "showReview" segue and set the *Presentation* option to `Full Screen` in the Attributes inspector.

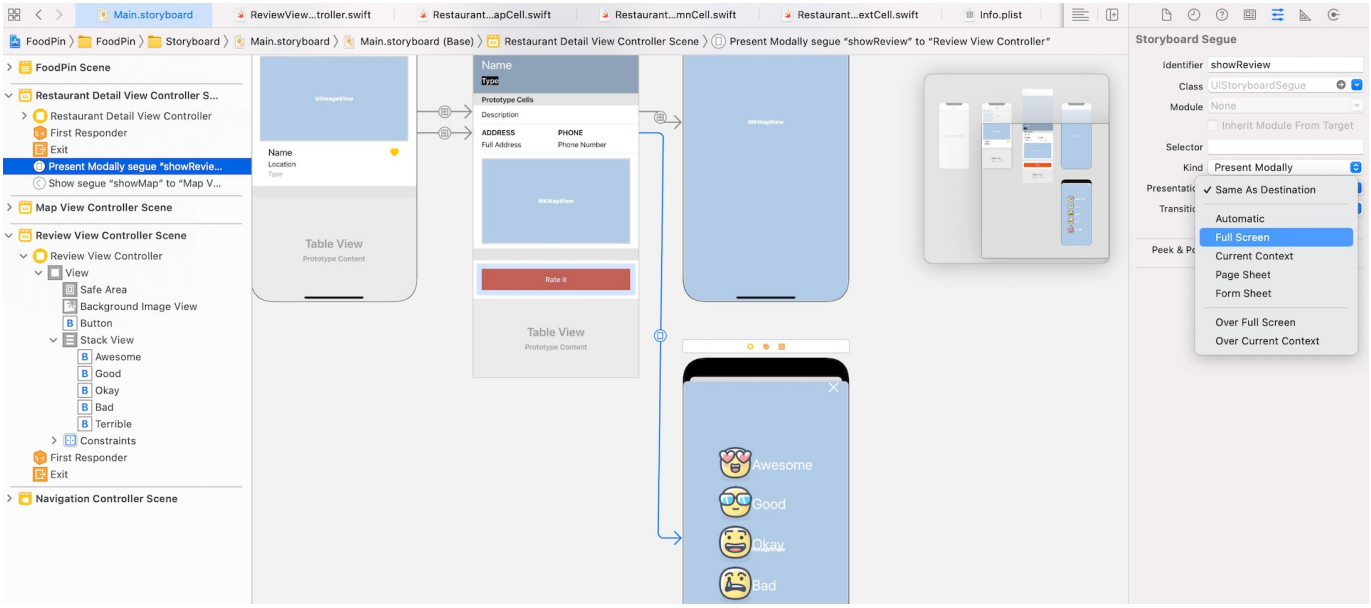


Figure 17-15. Changing the presentation mode

iOS has some built-in transition animation at your disposal. By default, the review view is brought up from the bottom of the screen. This transition animation is known as *Cover Vertical*, which is the default transition for modal presentation. If you select the segue and go to the Attributes inspector, you can find other transition options such as Cross Dissolve and Flip Horizontal. Try to change the transition option from *Default* to *Cross Dissolve* and see what animation you get. In this demo, I will continue to use *Cross Dissolve* as the transition animation, but you are free to choose your own.

Defining an Exit for the Review View Controller

After you change the presentation mode to *Full Screen*, you lose the built-in gesture support for dismissing the view. There are multiple ways to handle view dismissal. What we are going to implement is something called *unwind segue*. An unwind segue can be used to navigate back through a modal or push segue. In this example, we can use it to dismiss the modal view.

To use an unwind segue, you need to do two things. First, declare a method in the destination view controller. In this case, it is the `RestaurantDetailViewController` class. Add the following method in `RestaurantDetailViewController.swift`:

```
@IBAction func close(segue: UIStoryboardSegue) {
    dismiss(animated: true, completion: nil)
}
```

Before you can add unwind segues in Interface Builder, you must define at least one unwind action in the destination controller. This action method serves a sign that this view controller can be the destination controller of an unwind segue. Here we just call the `dismiss(animated: true, completion: nil)` method to dismiss the current view controller. Optionally, you can implement additional logic in the method. Go back to Interface Builder. Press and hold control key and drag from the *Close* button to the Exit icon of the scene dock (see figure 17-16). When prompted, select `closeWithSegue:` for the action segue.

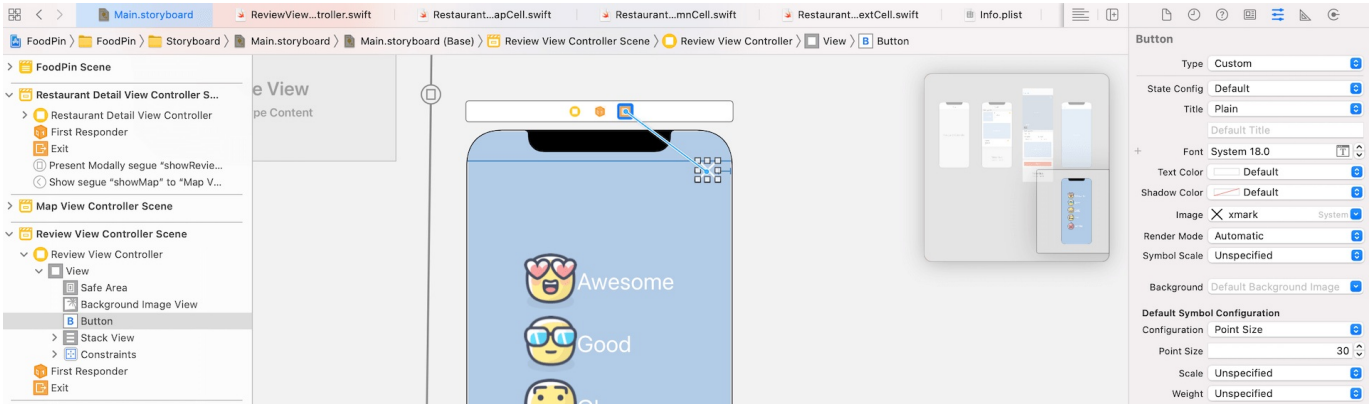


Figure 17-16. Adding an unwind segue for the close button

You can run the project again. Now when the user taps the close button, the modal view will be dismissed.

Applying a Blurring Effect to the Background Image

Now that the background image shows the original photo of a restaurant, this doesn't look as expected. It should be blurred instead. The iOS SDK has a class called `UIVisualEffectView` that lets developers apply visual effects to a view. Combining with the `UIBlurEffect` class, you can easily apply a blurring effect to an image view.

Open the `ReviewViewController.swift` file and update the `viewDidLoad` method like this:

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

    backgroundImageView.image = UIImage(named: restaurant.image)

    // Applying the blur effect
    let blurEffect = UIBlurEffect(style: .dark)
    let blurEffectView = UIVisualEffectView(effect: blurEffect)
    blurEffectView.frame = view.bounds
    backgroundImageView.addSubview(blurEffectView)
}
```

To apply a blurring effect to the background image view, all you need to do is create a `UIVisualEffectView` object with the blurring effect, followed by adding the visual effect view to the background image view. The `UIBlurEffect` class offers three different styles: `.dark`, `.light`, and `.extraLight`. I like the dark style but it is up to you to choose your own style. The above lines of code are all you need to blur the background image.

You're ready to go. Figure 17-17 displays the resulting screen, depending on your choice of blur style.



Figure 17-17. Extralight blurring style (left), Light blurring style (middle), and Dark blurring style (right)

Understanding Outlet Collections

We will soon discuss how to create UI animations. Since we want to animate the buttons, we have to connect them to our code. Usually, we create an outlet for each of the buttons. This time, let me show you something different called *Outlet Collections*.

The function of an outlet collection is exactly the same as an outlet. It is still a property that references the UI objects in Interface Builder. However, the difference is that an outlet collection allows you to reference multiple UI objects with a single outlet variable. You will see what I mean in a while.

Let's say, we want to reference the group of the rate buttons with an outlet collection. We can declare an outlet variable in the `ReviewViewController` class:

```
@IBOutlet var rateButtons: [UIButton]!
```

As you can see, the variable of an outlet collection has a type `[UIButton]`. It holds an array of `UIButton` instead of a single button.

Now head over to `Main.storyboard`. We will connect each of the rate buttons to this outlet variable. Right-click the Review View Controller. Click the circle button of `rateButtons` and drag it to the *Love* button.

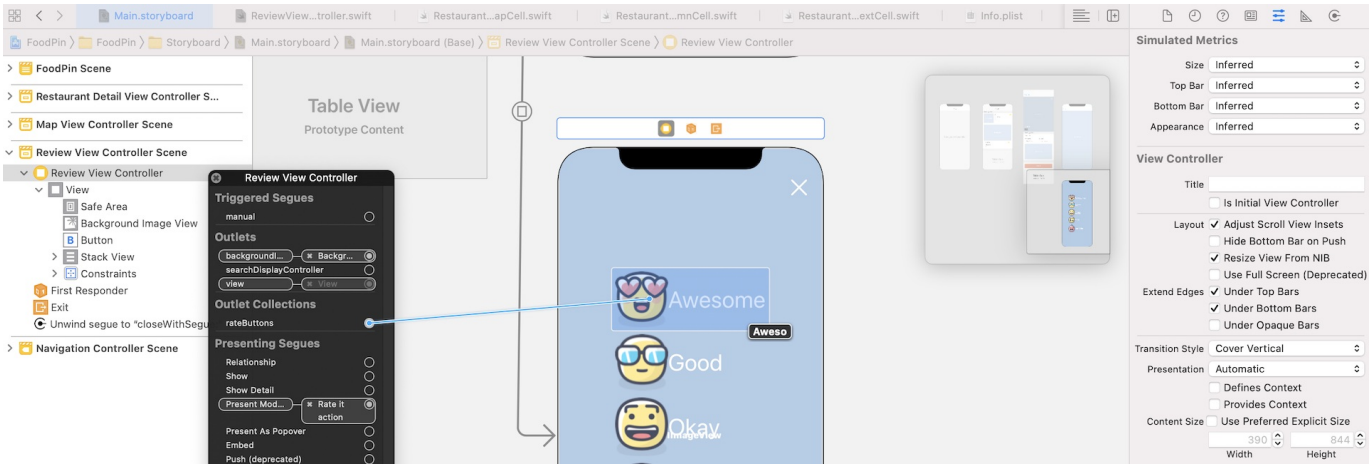


Figure 17-18. Establishing a connection with the outlet collection

You should be very familiar with this procedure. What's interesting about outlet collections is that you can keep establishing connections with other buttons. Once the *Love* button is connected, click the circle button and drag to the *Cool* button to establish a connection.

Repeat the procedures for the rest of the buttons. The result of the `rateButton` outlet collection should appear like the one shown in figure 17-19.

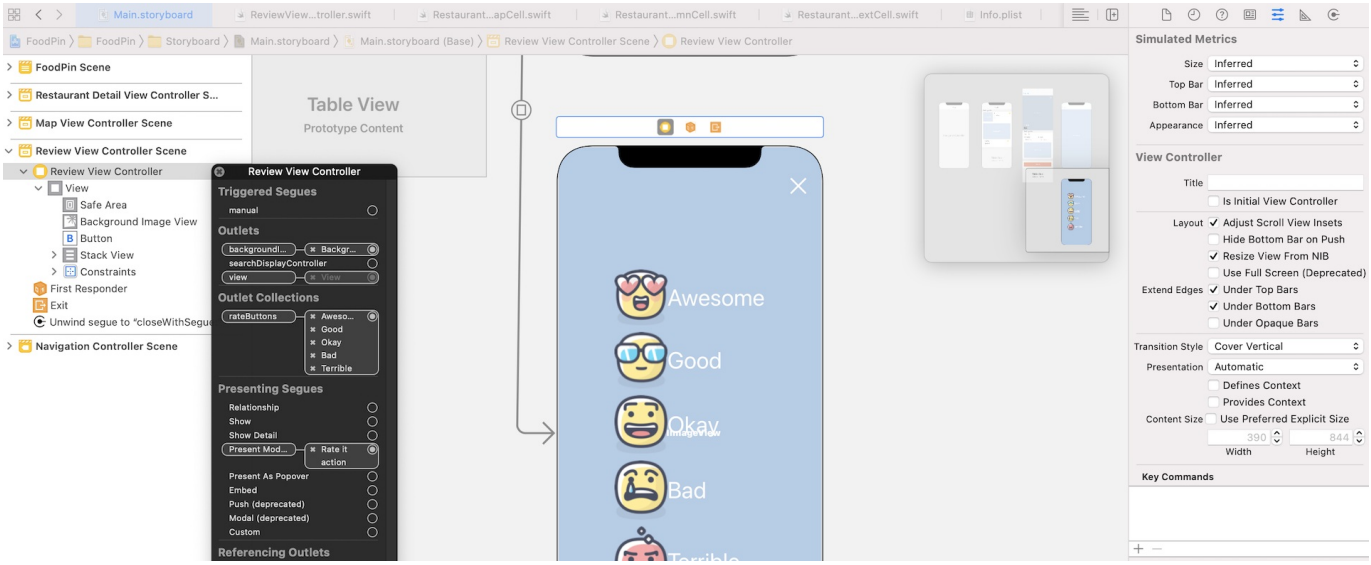


Figure 17-19. All buttons are connected with the outlet collection

You may wonder how you can retrieve each of the buttons from the `rateButtons` outlet. The buttons are added to the array based on the connection order. So you can access the first button using `rateButtons[0]`, and vice versa.

```
rateButtons[0] // Awesome button
rateButtons[1] // Good button
rateButtons[2] // Okay button
rateButtons[3] // Bad button
rateButtons[4] // Terrible button
```

Animating Dialog View Using UIView Animation

After all the preparation, we finally come to the core part of the chapter: *UIView animation*. To begin, we're going to create a simple fade-in animation for the rate buttons. Figure 17-20 will give you a better idea of the animation.

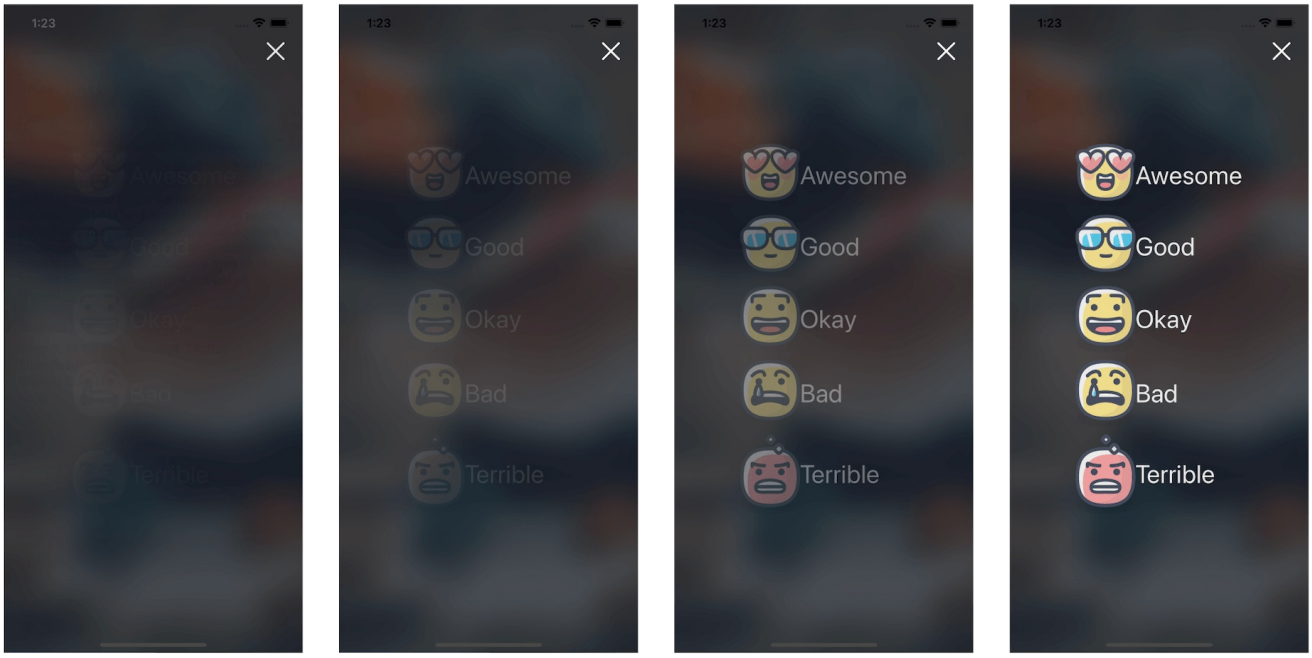


Figure 17-20. Fade-in animation

As explained before, you just need to provide two states (start and end state) of an animation. `UIView` will then generate the animation for you. For the fade-in animation, these are the start and end states:

- **Invisible state** - this is the buttons' start state. This can be done by setting its alpha value to `0`.
- **Visible state** - this is the end state. The alpha value is set to `1`.

Let's begin with the start state. We will make all the rating buttons invisible in the `viewDidLoad()` method of the `ReviewViewController` class. Insert the following code at the end of `viewDidLoad()`:

```
// Make the button invisible
for rateButton in rateButtons {
    rateButton.alpha = 0
}
```

When setting the value of the `alpha` property to `0`, we can make the buttons invisible.

Now it comes to the end state. To create the fade-in effect, insert the following code in the `ReviewViewController` class:

```
override func viewWillAppear(_ animated: Bool) {
    UIView.animate(withDuration: 2.0) {
        self.rateButtons[0].alpha = 1.0
        self.rateButtons[1].alpha = 1.0
        self.rateButtons[2].alpha = 1.0
        self.rateButtons[3].alpha = 1.0
        self.rateButtons[4].alpha = 1.0
    }
}
```

The code is very straightforward. We call the `UIView.animate(withDuration:animations:)` method to animate the change of the alpha value. In the body of the closure, you just need to specify the end state (i.e. alpha = 1.0). The API will automatically compute the animation for you. The duration parameter specifies the duration of the animation. In this example, the animation will complete in 2 seconds.

You may have a question about `viewWillAppear`. Why do you implement the animation in this method instead of `viewDidLoad`? I have briefly explained the view controller life cycle in earlier chapters. The `viewDidLoad` method is called when a view is first loaded, but the view hasn't displayed on the screen. If we implement the animation in this method, the animation may start too early and probably finish even before the view appears on the screen. The `viewWillAppear` method, on the other hand, is automatically called after `viewDidLoad` by iOS when the view is about to appear on the screen. It is more suitable for us to render the animation. This is why we write the code in `viewWillAppear` rather than `viewDidLoad`.

That's it. Run the app for a quick test and enjoy the animation.

> **Note:** If you can't see the animation clearly, you can change the duration parameter from 2 seconds to other value (e.g. 10.0) to slow down the animation. Alternatively, you can enable slow animations in the iOS simulator. Go up to the simulator menu, select Debug > Slow animations.

The `UIView.animate(withDuration:animations:)` method is not the only API for rendering animations. It has other variations with additional parameters to control an animation. Try to replace the `viewWillAppear` method with the following code:

```
override func viewWillAppear(_ animated: Bool) {

    UIView.animate(withDuration: 0.4, delay: 0.1, options: [], animations:
 {
        self.rateButtons[0].alpha = 1.0
    }, completion: nil)

    UIView.animate(withDuration: 0.4, delay: 0.15, options: [], animations
: {
        self.rateButtons[1].alpha = 1.0
    }, completion: nil)

    UIView.animate(withDuration: 0.4, delay: 0.2, options: [], animations:
 {
        self.rateButtons[2].alpha = 1.0
    }, completion: nil)

    UIView.animate(withDuration: 0.4, delay: 0.25, options: [], animations
: {
        self.rateButtons[3].alpha = 1.0
    }, completion: nil)

    UIView.animate(withDuration: 0.4, delay: 0.3, options: [], animations:
 {
        self.rateButtons[4].alpha = 1.0
    }, completion: nil)

}
```

The `UIView.animate` method has other parameters such as delay for developers to further customize an animation. In the code above, we set a distinct `delay` value for each of the buttons. Run the app again and you will experience a fancier fade-in animation.

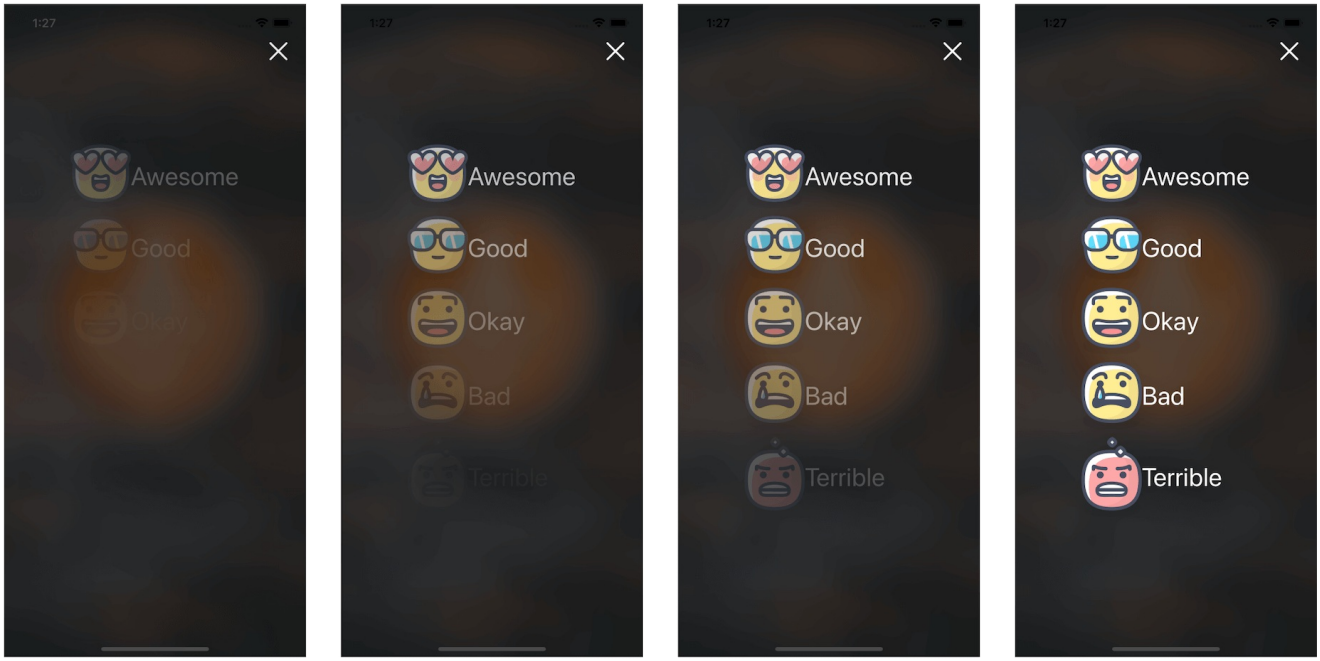


Figure 17-21. Fade-in animation with delay

Slide-in Animation

A slide-in animation is another common type of animations. If you don't know what a slide-in animation looks like, take a look at figure 17-22. The object slides from the far right of the screen until it reaches a certain position.

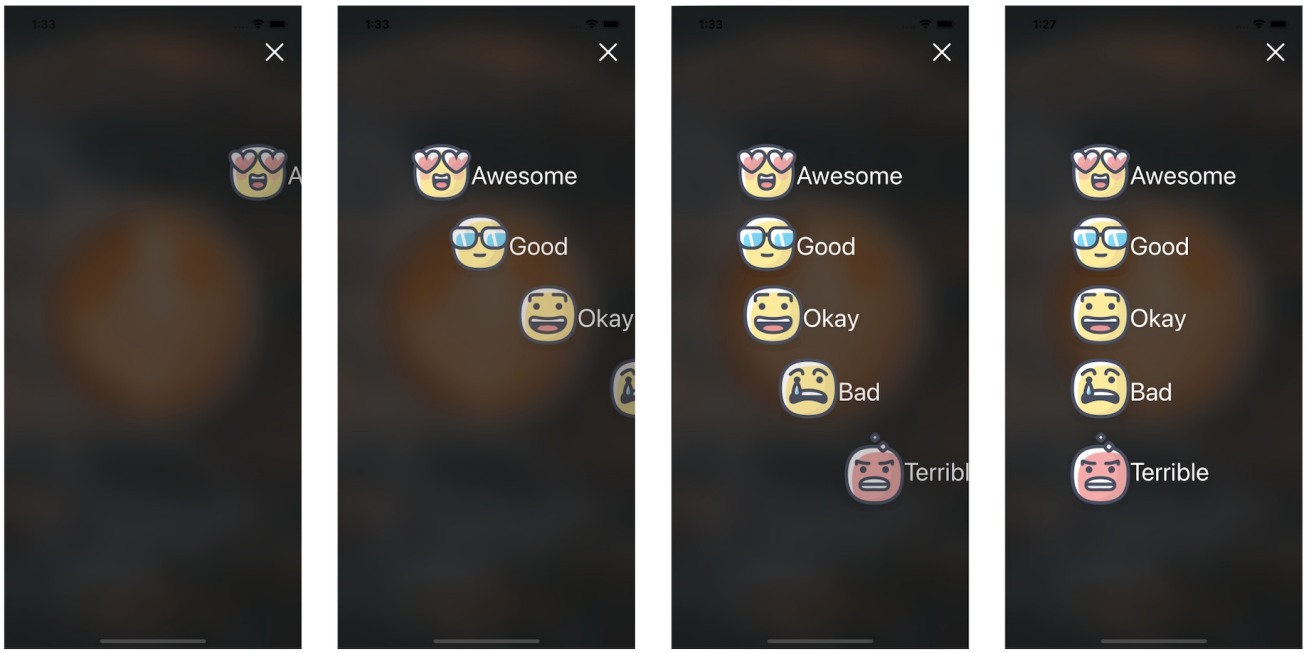


Figure 17-22. How the slide-in animation works

Let's see how to implement this type of animation. Again, consider the start and end state of the animation. To create a slide-in-from-right animation, we will move all the rate buttons off the screen to the right. This is the start state. The end state is the original position of the buttons that we define in Interface Builder.

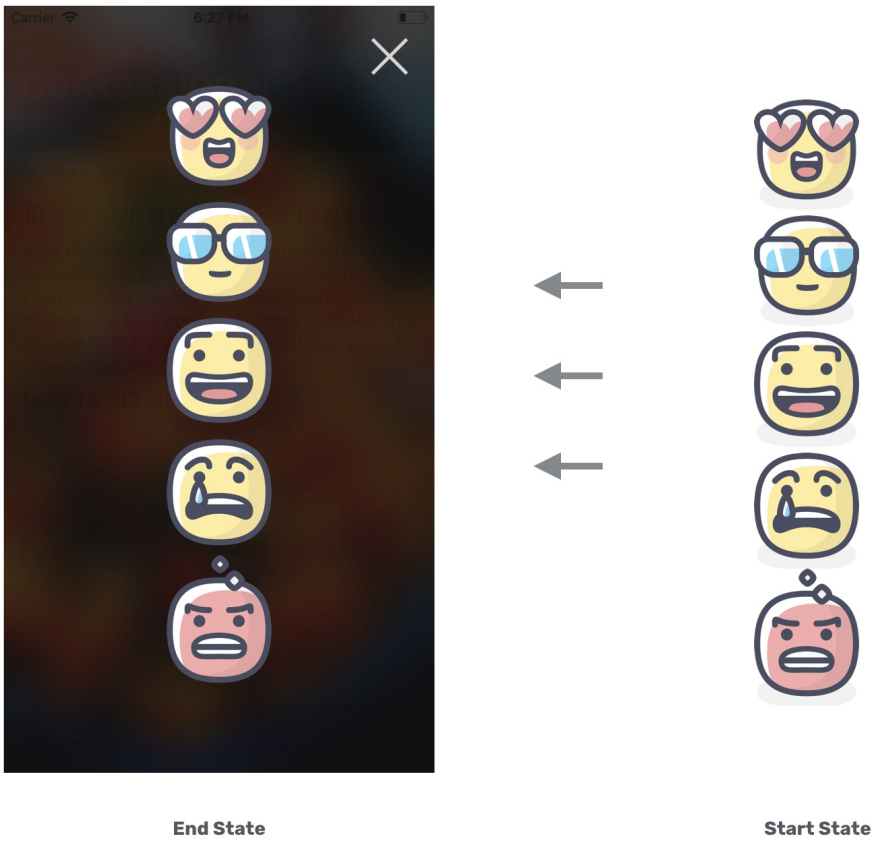


Figure 17-23. The start and end state of the slide-in animation

At this point, you might be trying to figure out how to move a view (or button) off the screen. iOS provides a structure called `CGAffineTransform` for you to move, scale, and rotate a view. To move a view, all you need to do is create an affine transformation and set it to the `transform` property of a `UIView` object. Here is the line of code for creating a translation transform that moves a view 600 points to the right of the screen:

```
CGAffineTransform.init(translationX: 600, y: 0)
```

If you want to move a view to the left of the screen, use a negative number. The parameter `y` lets you move a view along the y-axis. In other words, you can set this value to move a view vertically.

Now let's try out the code in action. Update the `viewDidLoad` method of the `ReviewViewController` class like this:

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

    backgroundImageView.image = UIImage(named: restaurant.image)

    // Applying the blur effect
    let blurEffect = UIBlurEffect(style: .dark)
    let blurEffectView = UIVisualEffectView(effect: blurEffect)
    blurEffectView.frame = view.bounds
    backgroundImageView.addSubview(blurEffectView)

    let moveRightTransform = CGAffineTransform.init(translationX: 600, y: 0
)

    // Make the button invisible
    for rateButton in rateButtons {
        rateButton.transform = moveRightTransform
        rateButton.alpha = 0
    }
}
```

We have made a couple of minor changes here. First, we declare a constant called `moveRightTransform` that holds the translation transform for moving a view to the right. Secondly, we add a line of code in the `for-in` loop:

```
rateButton.transform = moveRightTransform
```

The `transform` property of each button is assigned with `moveRightTransform`. This will move the buttons off the screen to the right.

Okay, that's the implementation for the start state. For the end state, the question is how can we reset the position of rate buttons to its original position?

That's very simple. There is a built-in transform called *identity transform* that can be used to clear any pre-defined transform. So update the `viewWillAppear` method like this:

```
override func viewWillAppear(_ animated: Bool) {

    UIView.animate(withDuration: 0.4, delay: 0.1, options: [], animations:
 {
        self.rateButtons[0].alpha = 1.0
        self.rateButtons[0].transform = .identity
    }, completion: nil)

    UIView.animate(withDuration: 0.4, delay: 0.15, options: [], animations
: {
        self.rateButtons[1].alpha = 1.0
        self.rateButtons[1].transform = .identity
    }, completion: nil)

    UIView.animate(withDuration: 0.4, delay: 0.2, options: [], animations:
 {
        self.rateButtons[2].alpha = 1.0
        self.rateButtons[2].transform = .identity
    }, completion: nil)

    UIView.animate(withDuration: 0.4, delay: 0.25, options: [], animations
: {
        self.rateButtons[3].alpha = 1.0
        self.rateButtons[3].transform = .identity
    }, completion: nil)

    UIView.animate(withDuration: 0.4, delay: 0.3, options: [], animations:
 {
        self.rateButtons[4].alpha = 1.0
        self.rateButtons[4].transform = .identity
    }, completion: nil)

}
```

For each of the buttons, we add a line of code to set the button's transform to `.identity` that will reset the button to its original position.

That's it! Now it's time to test the app and see the animation in action.

Spring Animation

The animation is cool, right? Let me introduce a variation of the `UIView` animation known as *spring animation*. The spring animation API lets you easily create some bouncing animations. To take advantage of spring animation in your app, here is the single method call you need:

```
UIView.animate(withDuration: 0.8, delay: 0.1, usingSpringWithDamping: 0.2,
 initialSpringVelocity: 0.3, options: [], animations: {
    self.rateButtons[0].alpha = 1.0
    self.rateButtons[0].transform = .identity
}, completion: nil)
```

The method should look familiar to you, but it adds the `damping` and `initialSpringVelocity` parameters. Damping takes a value from 0 to 1, and controls how much resistance the spring has when it approaches the end state of an animation. If you want to increase oscillation, set to a lower value. The `initialSpringVelocity` property specifies the initial spring velocity. Try to replace the existing animation code of `rateButtons[0]` with the above code and see how the spring animation works.

Combining Two Transforms

What's interesting about affine transform is that you can concatenate one transform with another. This is the function you need to remember:

```
transform1.concatenating(transform2)
```

Now that we have implemented the *translate* transform for the rate buttons, wouldn't it be great if we can add a scale transform to them to create some amazing animations? In the `viewDidLoad` method, replace the following code:

```
let moveRightTransform = CGAffineTransform.init(translationX: 600, y: 0)

// Make the button invisible
for rateButton in rateButtons {
    rateButton.transform = moveRightTransform
    rateButton.alpha = 0
}
```

with:

```
let moveRightTransform = CGAffineTransform.init(translationX: 600, y: 0)
let scaleUpTransform = CGAffineTransform.init(scaleX: 5.0, y: 5.0)
let moveScaleTransform = scaleUpTransform.concatenating(moveRightTransform
)

// Make the button invisible and move off the screen
for rateButton in rateButtons {
    rateButton.transform = moveScaleTransform
    rateButton.alpha = 0
}
```

We create a scale-up transform that will make a view 5 times larger. And then we combine it with the translation transform by calling the `concatenating` function. Each of the rate buttons is assigned with this combine transform.

For the end state, we can keep the `viewWillAppear` method unchanged. That's it! You're ready to go and see how the combined animation works.

Unwind Segues and Data Passing

Earlier, we used an unwind segue to "go back" to the detail view when a user taps the *close* button. How about the *rating* buttons? How can we pass the selected rating from the review view controller to the detail view controller?

We will make use of unwind segues to facilitate the data passing. Here are what we are going to implement:

- We will add another unwind action method in the `ReviewViewController` class - when any of the buttons (awesome/good/okay/bad/terrible) is tapped, the method will be called.
- In the method, we determine which button is tapped, and save the corresponding rating.
- Then we display the selected rating in the detail view. You can refer to figure 17-2 to see how the rating icon is displayed.

It sounds complicated, but the implementation is quite similar to the one we just implemented for the close button. That said, there are some extra works we have to do before we can store the rating of a restaurant.

First, the `Restaurant` class doesn't have any property for holding the value of the selected rating. So we need to edit the class to add a `rating` property. The question is what type should be used for this `rating` property? Should we use a `String` type to store the image file name? Or do we need to create a new type?

In this case, creating a new type using an enum is a better approach. Open the `Restaurant.swift` file and update the `Restaurant` structure like this:

```
struct Restaurant: Hashable {
    enum Rating: String {
        case awesome
        case good
        case okay
        case bad
        case terrible

        var image: String {
            switch self {
            case .awesome: return "love"
            case .good: return "cool"
            case .okay: return "happy"
            case .bad: return "sad"
            case .terrible: return "angry"
            }
        }
    }

    var name: String = ""
    var type: String = ""
    var location: String = ""
    var phone: String = ""
    var description: String = ""
    var image: String = ""
    var isFavorite: Bool = false
    var rating: Rating?
}
```

In the code above, we have added a new `rating` property of the type `Rating`. This variable is an optional meaning that it is not a must to store a value. The `Rating` type is an enumeration that stores all the available ratings including awesome, good, okay, bad, and terrible. In the enum, we have a computed property named `image` that returns the corresponding file name of the rating image.

You may wonder why we used an enum instead of a string to represent the rating. String is a good choice for handling freeform text but not a perfect choice for storing this kind of data. We only have 5 different ratings here. If we use `String` type to hold the rating, this means it can store any text data. You may store "awesome", "Awesome", or "AWESOME". Since Swift is a case sensitive language, all these values are not the same. To avoid programming errors, it is more appropriate to use enum to creat a new type, defining the possible values it can hold.

Now switch over to `Main.storyboard`. We will add another image view to the header view. This image view will be used to display the icon of the selected rating. Drag an image view from the Object library to the Restaurant Detail View Controller. Place it right under the Dim View in document outline.

Next, click the *Add New Constraints* button to add a few layout constraints. Set the value of right and bottom sides to `20`. To limit the size of the image view, check both height and width options. Set the height to `50` and width to `52`. Click *Add 4 Constraints* to confirm.

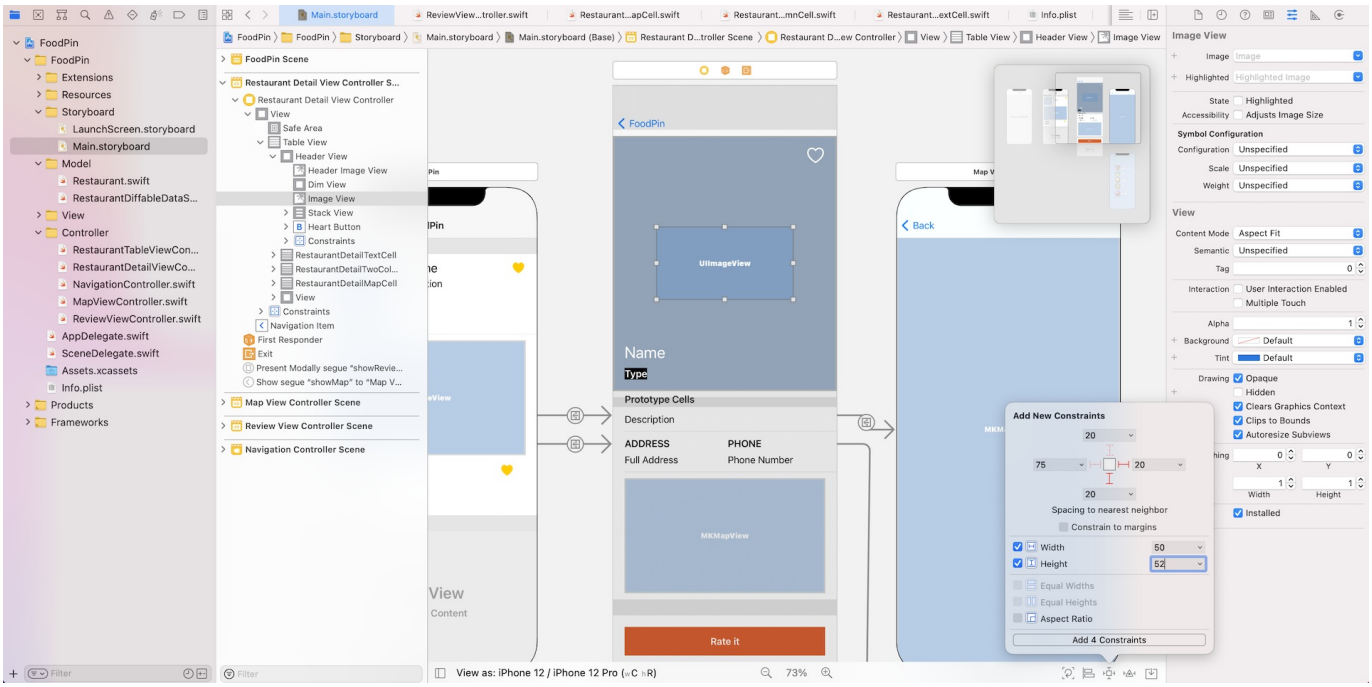


Figure 17-24. Adding an image view to the header view

After adding the constraints, the image view is resized and moved. However, it is not placed at our preferred location. We expect it resides near the lower-right corner of the header view. There is an issue with the bottom constraint of the image view. If you select it, you will find that this spacing constraint is relative to the Name label, which is incorrect. It should be relative to the bottom of the header view.

In the Attributes inspector, change the First Item from `Stack View.Top` to `Stack View.Bottom`. You have to click the `Stack View.Top` option and select `Bottom`. Also, change the constant to `0`.

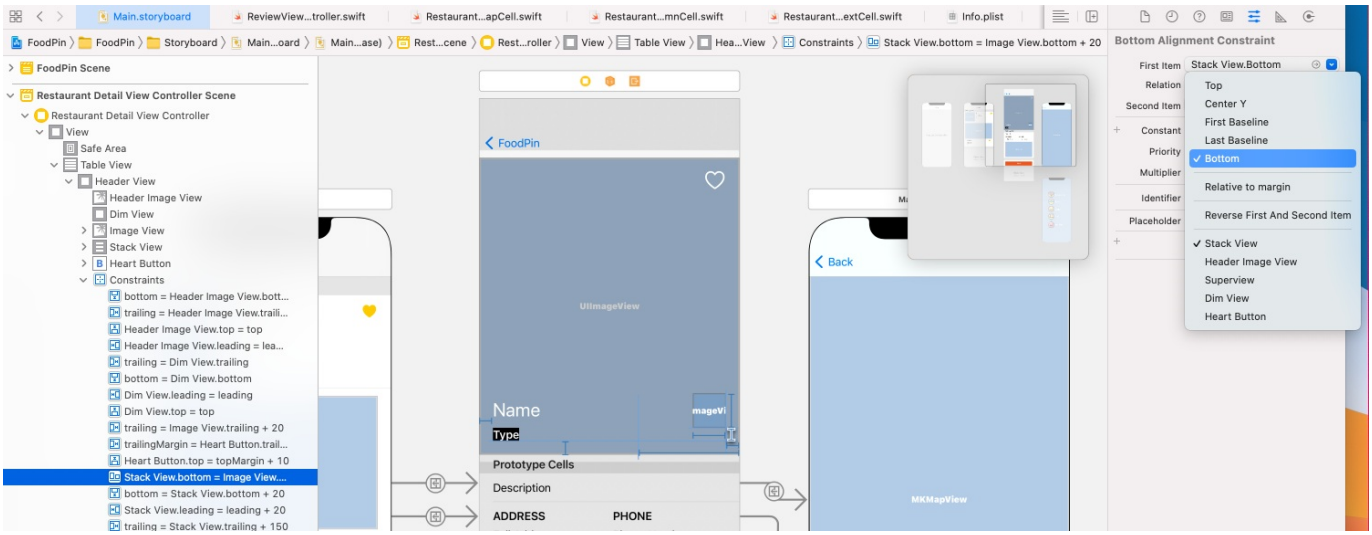


Figure 17-25. Change the first item of the spacing constraint

If you've made the change correctly, the bottom of the image view should be perfectly aligned with the bottom of the type label.

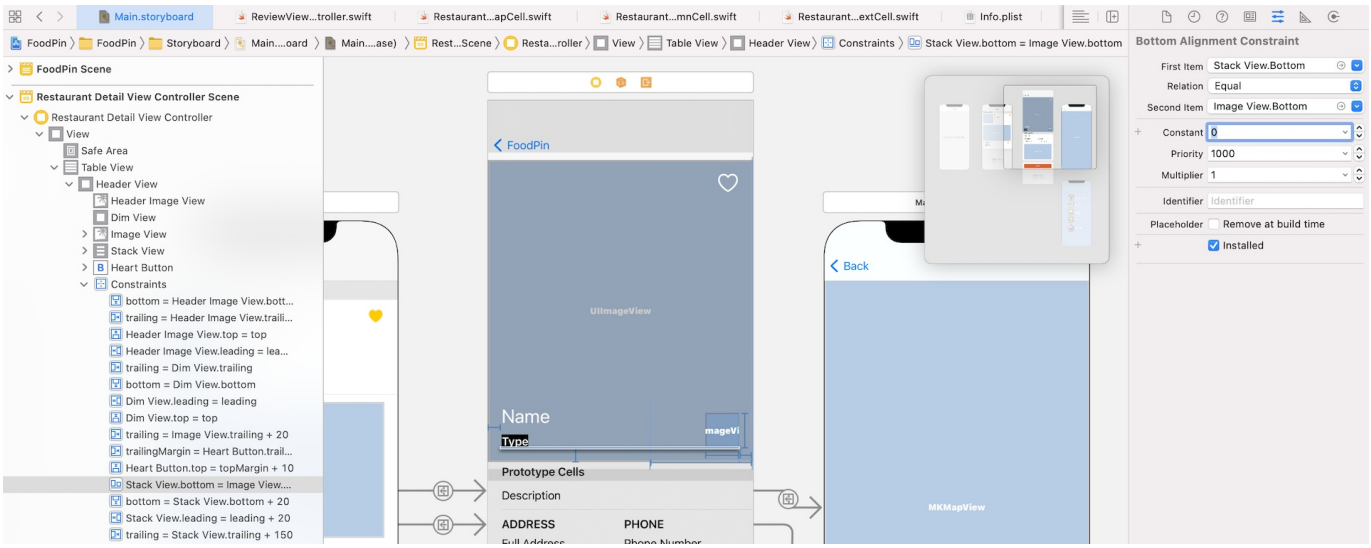


Figure 17-26. Change the first item of the spacing constraint

Now we've completed the UI design of the detail view. Open the `RestaurantDetailHeaderView.swift` file and add an outlet variable for this image view:

```
@IBOutlet var ratingImageView: UIImageView!
```

Go back to the storyboard. Right-click Header View in document outline to establish the connection between the outlet variable and the new image view.

Finally, we've completed all the UI preparations. It's time to create the unwind segue to facilitate the data passing. Declare the following method in the `RestaurantDetailViewController` class:

```
@IBAction func rateRestaurant(segue: UIStoryboardSegue) {

}
```

Now go to `Main.storyboard` to connect each of the rating buttons with the unwind action method. Control-drag from the *Love* button to the exit icon. Release the button and select `rateRestaurantWithSegue:`. Once the unwind segue is created, select it in the document outline. Set its identifier to `awesome` in the Attributes inspector.

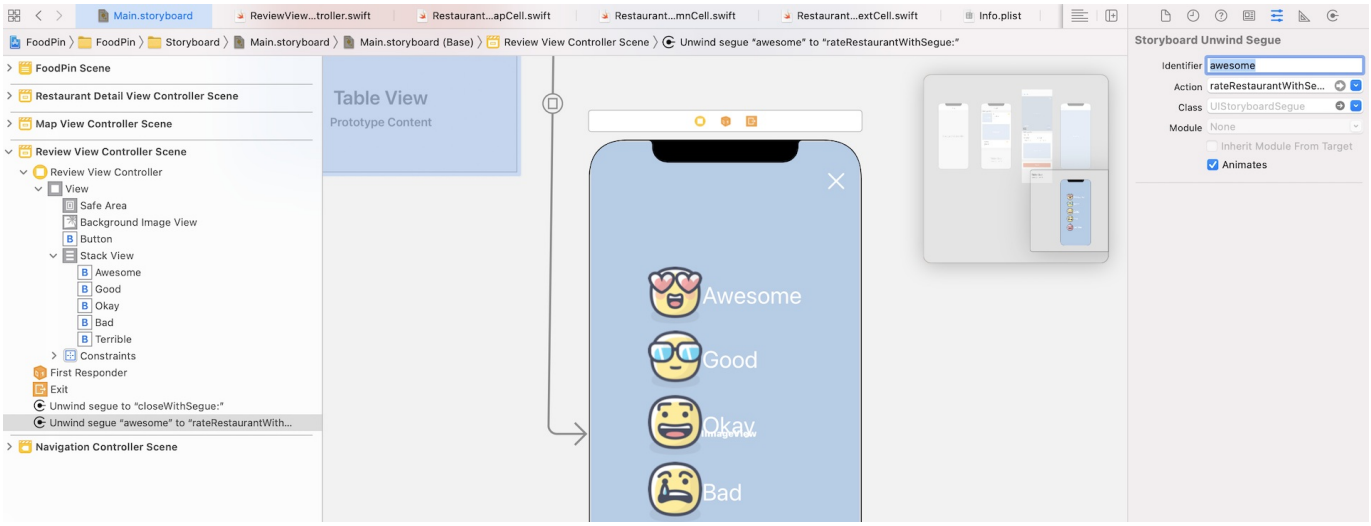


Figure 17-27. Adding an identifier for the unwind segue

Repeat the same step to connect the other 4 buttons with the `ratingButtonTappedWithSegue:` method:

- For the *Good* button, set the unwind segue's identifier to `good`.
- For the *Okay* button, set the unwind segue's identifier to `okay`.
- For the *Bad* button, set the unwind segue's identifier to `bad`.
- For the *Terrible* button, set the unwind segue's identifier to `terrible`.

Do you know why we set these identifiers? Each of the rating buttons is connected to the same unwind action method. We need to have a way to find out which button is tapped. This is the reason why we set these segue identifiers. And, the value of the identifier is the same as that we defined in the `Rating` enum.

Now, go back to `RestaurantDetailViewController.swift` to implement the unwind action method. Update the `rateRestaurant(segue:)` method like this:

```
@IBAction func rateRestaurant(segue: UIStoryboardSegue) {

    guard let identifier = segue.identifier else {
        return
    }

    if let rating = Restaurant.Rating(rawValue: identifier) {
        self.restaurant.rating = rating
        self.headerView.ratingImageView.image = UIImage(named: rating.imag
e)
    }

    dismiss(animated: true, completion: nil)
}
```

The code above is pretty straightforward. We first retrieve the unwind segue's identifier. Depending on the segue's identifier (awesome/good/okay/bad/terrible), we set the rating of the restaurant accordingly. And then we retrieve the rating image by accessing the `image` property and assign it to the rating image view.

If you run the app now and rate a restaurant, you will notice the image of your selected rating appeared on the detail view.

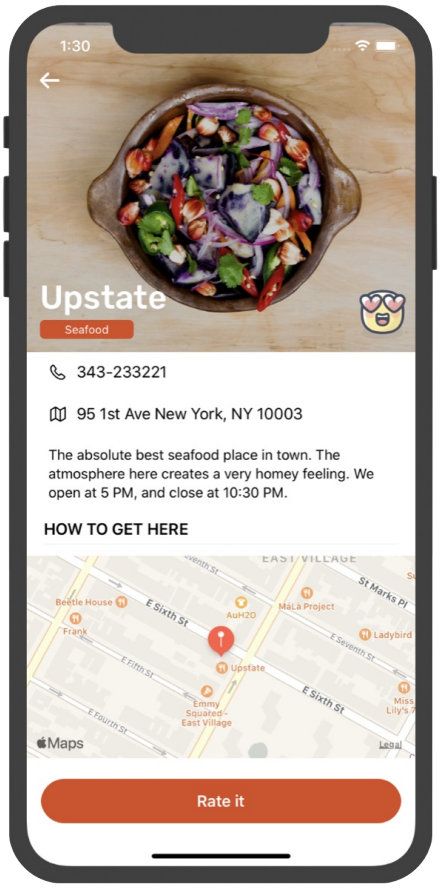


Figure 17-28. Rating icon is now displayed on the detail view

The rating feature works great! But we can further improve it by adding a subtle animation. Try to update the `rateRestaurant` method to the following:

```
@IBAction func rateRestaurant(segue: UIStoryboardSegue) {

    guard let identifier = segue.identifier else {
        return
    }

    dismiss(animated: true, completion: {

        if let rating = Restaurant.Rating(rawValue: identifier) {
            self.restaurant.rating = rating
            self.headerView.ratingImageView.image = UIImage(named: rating.
image)
        }

        let scaleTransform = CGAffineTransform.init(scaleX: 0.1, y: 0.1)
        self.headerView.ratingImageView.transform = scaleTransform
        self.headerView.ratingImageView.alpha = 0

        UIView.animate(withDuration: 0.4, delay: 0, usingSpringWithDamping
: 0.3, initialSpringVelocity: 0.7, options: [], animations: {
            self.headerView.ratingImageView.transform = .identity
            self.headerView.ratingImageView.alpha = 1
        }, completion: nil)

    })

}
```

The `dismiss` method accepts two parameters: `animated` and `completion`. The `animated` parameter indicates whether the view dismissal is animated. The `completion` parameter lets you specify a block of code to be executed after the view controller is dismissed. In the code block, we first scale down the rating image and then reset it back to its original size to create a subtle animation.

Run the app to try it out. You will understand what I mean.

Exercise #1

This time, I have two exercises for you. The first one is designed to test your knowledge of view animations. The close button of the review view controller is not animated. Your task is to implement a slide-in-from-top animation for the close button.

```
Hint: You will need to create an outlet for the close
button and create a translate transform for the button.
```

Exercise #2

The next exercise is related to code refactoring. If you look into the `viewWillAppear` method of `ReviewViewController`, you will find a lot of duplicate code:

```
override func viewWillAppear(_ animated: Bool) {

    UIView.animate(withDuration: 0.4, delay: 0.1, options: [], animations:
 {
        self.rateButtons[0].alpha = 1.0
        self.rateButtons[0].transform = .identity
    }, completion: nil)

    UIView.animate(withDuration: 0.4, delay: 0.15, options: [], animations
: {
        self.rateButtons[1].alpha = 1.0
        self.rateButtons[1].transform = .identity
    }, completion: nil)

    UIView.animate(withDuration: 0.4, delay: 0.2, options: [], animations:
 {
        self.rateButtons[2].alpha = 1.0
        self.rateButtons[2].transform = .identity
    }, completion: nil)

    UIView.animate(withDuration: 0.4, delay: 0.25, options: [], animations
: {
        self.rateButtons[3].alpha = 1.0
        self.rateButtons[3].transform = .identity
    }, completion: nil)

    UIView.animate(withDuration: 0.4, delay: 0.3, options: [], animations:
 {
        self.rateButtons[4].alpha = 1.0
        self.rateButtons[4].transform = .identity
    }, completion: nil)

}
```

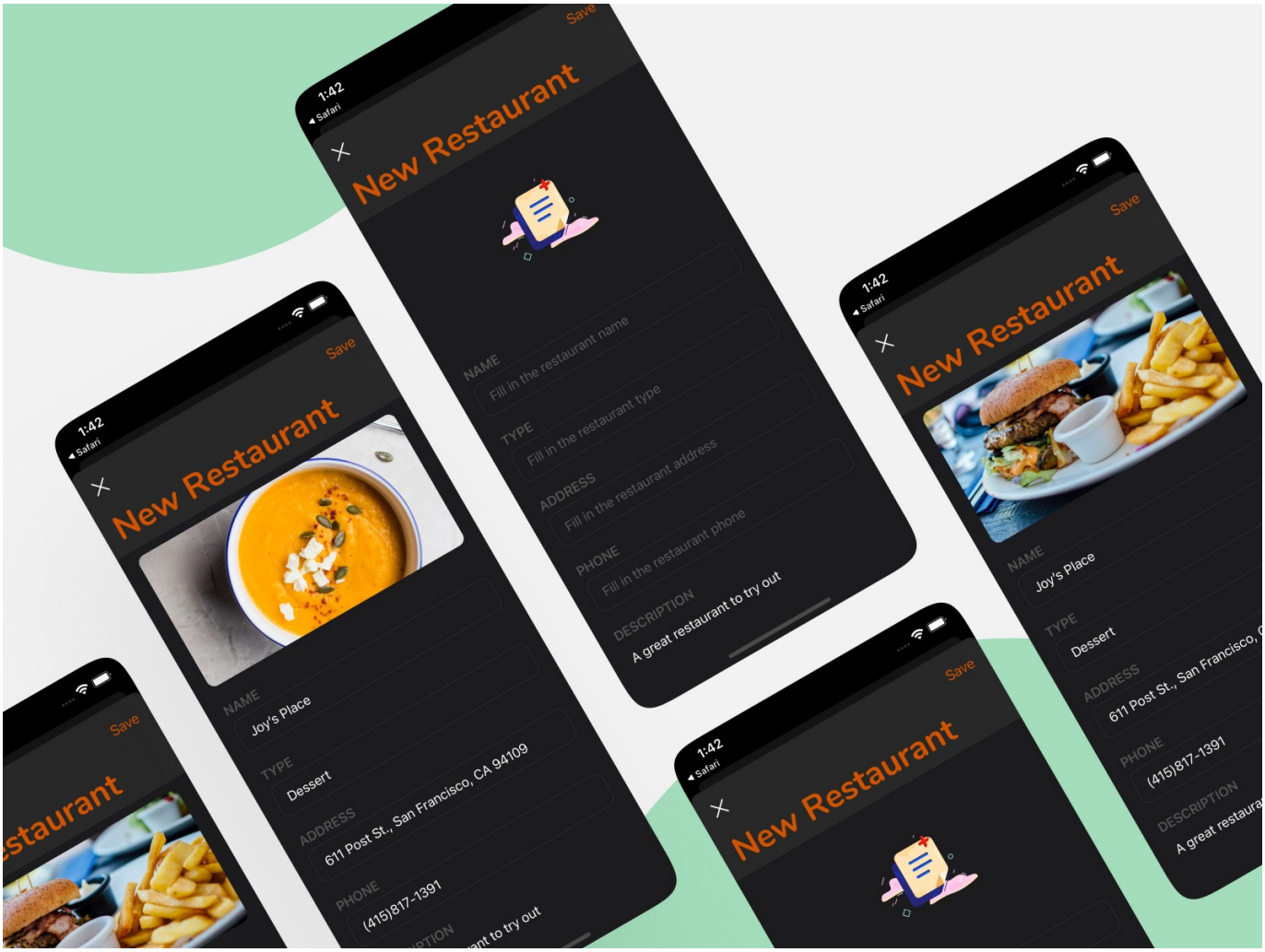
For each of the buttons, we set the value of `alpha` to 1.0 and `transform` to .identity. The code above can be rewritten using a `for-in` loop. Give it some thought and try to rewrite it.

Summary

This is another huge chapter that covers UIView animation, visual effects, and unwind segues. I hope you love all the techniques you've learned about UIView animations and visual effects. As you can see, it is very easy to animate a view, whether it's a button or a view. I would encourage you to play around with the parameters (like damping, initial spring velocity, and delay) and see what animations you can create. And, don't forget to take some time to complete the exercises.

For your reference, you can download the complete Xcode project from http://www.appcoda.com/resources/swift53/FoodPinAnimation.zip. The solution to the exercises is included in the project.

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 `NSLayoutAnchor`

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.

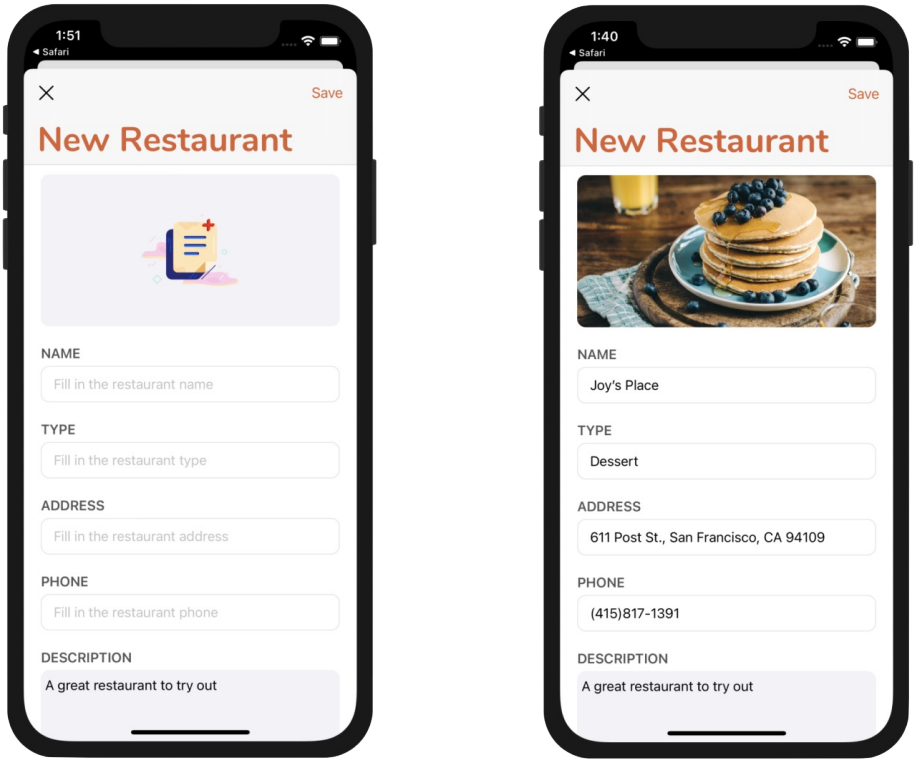


Figure 18-1. Creating a New Restaurant screen for adding a new restaurant

Designing the New Restaurant View Controller

Before you begin, download this image file http://www.appcoda.com/resources/swift53/newphotoicon.zip and add it to the asset catalog. Next, go to `Main.storyboard` and drag a table view controller from the Object library to the storyboard. Select the table view in the document outline. In the Attribute inspector, change the *Content* option from `Dynamic Prototypes` to `Static Cells`. Once changed, you will have a table view with three empty static cells.

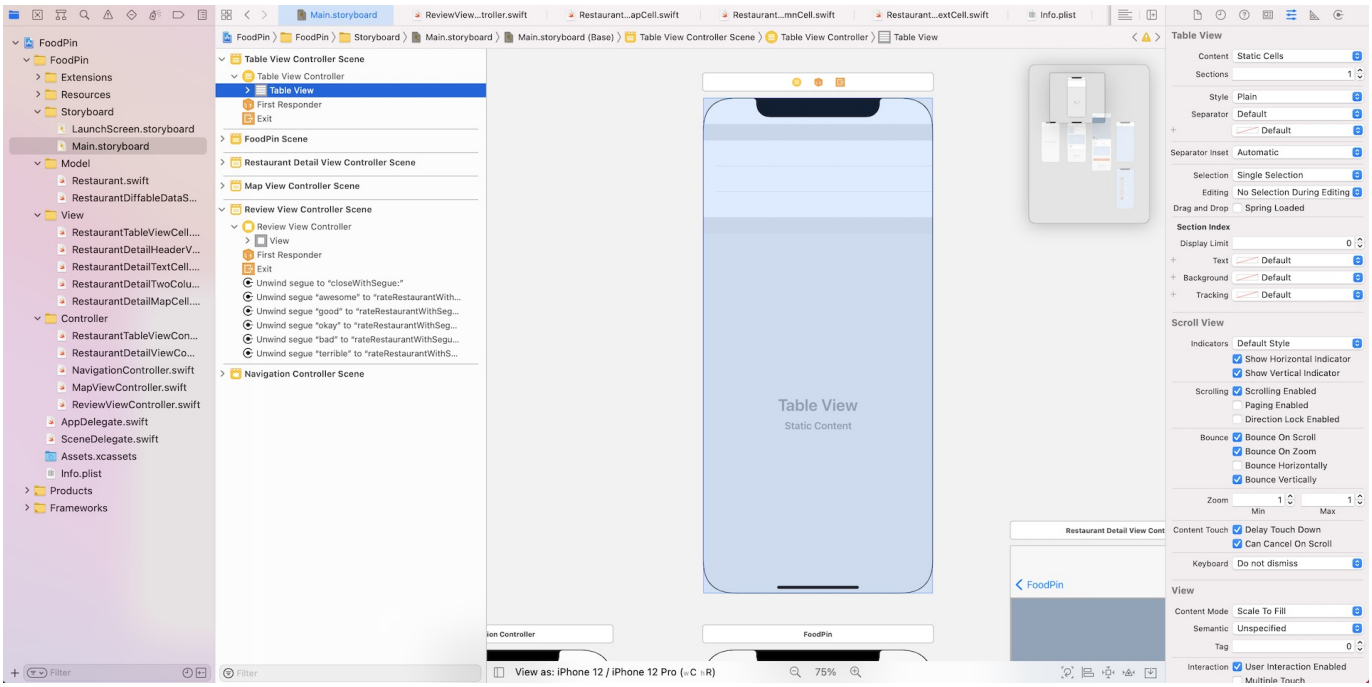


Figure 18-2. A static table view

If you refer to figure 18-1, you will find that we need a total of 6 table view cells:

- Cell #1 - an image view for displaying the restaurant image
- Cell #2 - a *Name* label and a text field
- Cell #3 - a *Type* label and a text field
- Cell #4 - an *Address* label and a text field
- Cell #5 - a *Phone* label and a text field
- Cell #6 - a *Description* label and a multiline text field

You can easily increase the number of cells by selecting the *Table View Section* in the document outline. In the Attribute inspector, you can change the *Rows* option from `3` to `6`.

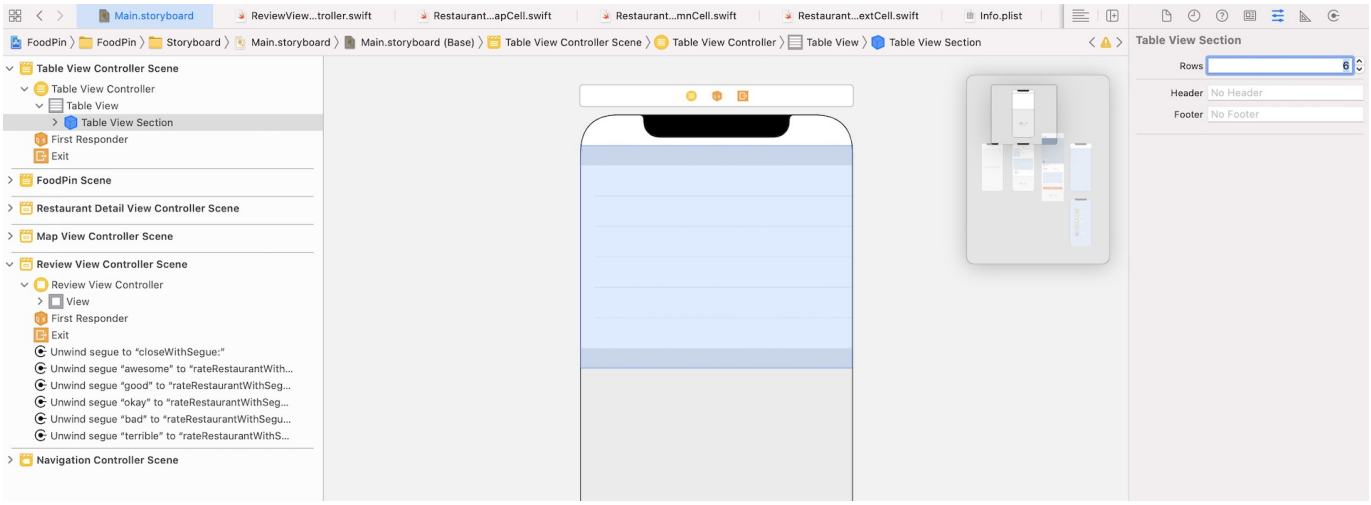


Figure 18-3. Changing the number of rows from 3 to 6

Now we'll layout each of the table view cells. For the first cell, change its height to `200` (or whatever value you prefer) in the Size inspector. And, set the Selection option to *None* in the Attributes inspector.

Open the Object library and drag an image view to the first cell. In the Attribute inspector, set the image to `newphoto` and change the background color to `System Gray 6 Color`. This is the image you imported to the asset catalog. If you don't like the image, please feel free to use other system images like `photo`. Next, click *Add new constraints* to add a few spacing constraints. Check the *Constrain to margin* option and set the value of all sides to `0`.

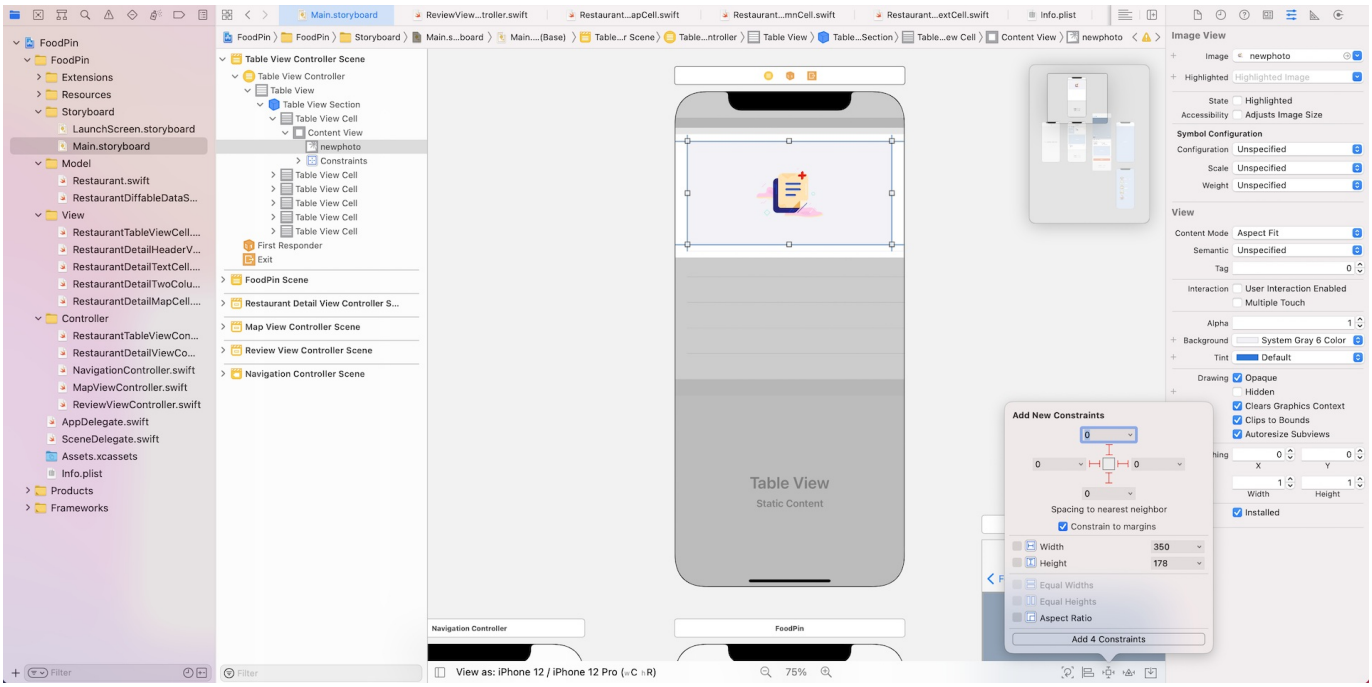


Figure 18-4. Add an image view to the first cell

For the second cell, adjust its height to make a bit taller. You will find it hard to place both the label and the text field in the cell. In the Attributes inspector, set the *Selection* option to *None*. This ensures the cell doesn't change its color even if it is touched.

Open the Object library, drag a label to the cell and change its title to `NAME`. Optionally, change its font to *Headline* style and font color to *dark gray*. Then, drag a text field to the cell and place it under the name label. The text field is a control for capturing the user input and displaying the editable text. Typically you use it to gather small amounts of text from users. In the Attribute inspector, set the placeholder value to `Fill in the restaurant name` and change the font type to `Body`. A placeholder is displayed when there is no other text in the text field.

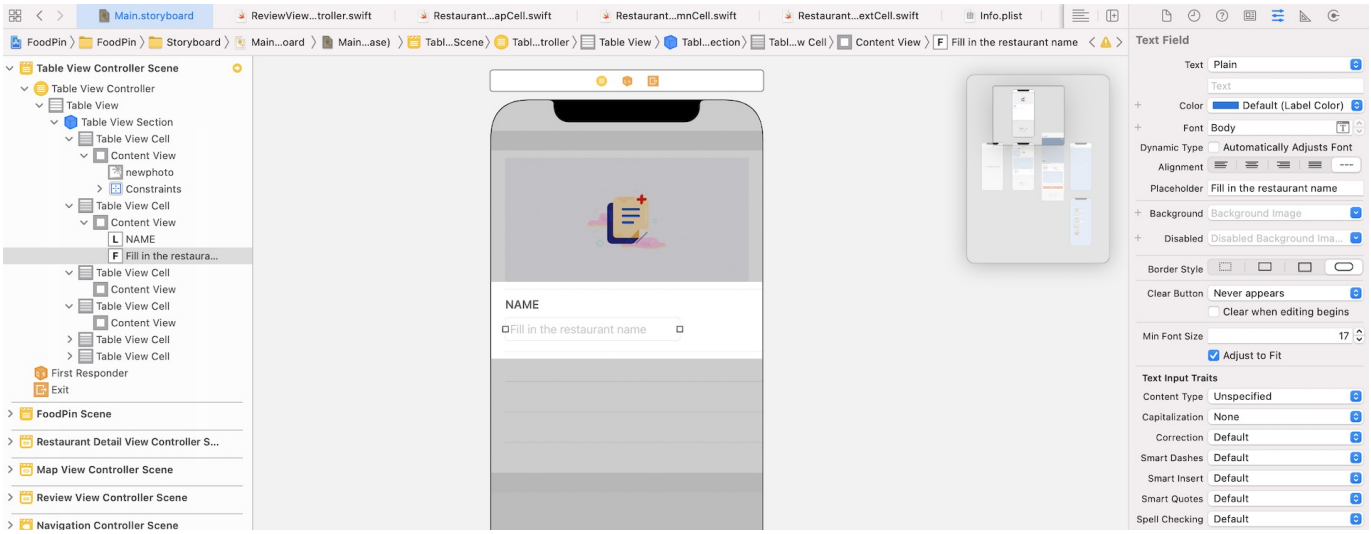


Figure 18-5. The cell layout for capturing user's input

Next, select both name label and the text field. Click the *Embed in* button and then select *Stack view* to embed them in a stack view. Then click the *Add new constraints* button to add the spacing constraints for the top, left, bottom and right sides. Set the spacing values to `0` and check the *Constrain to margins* option.

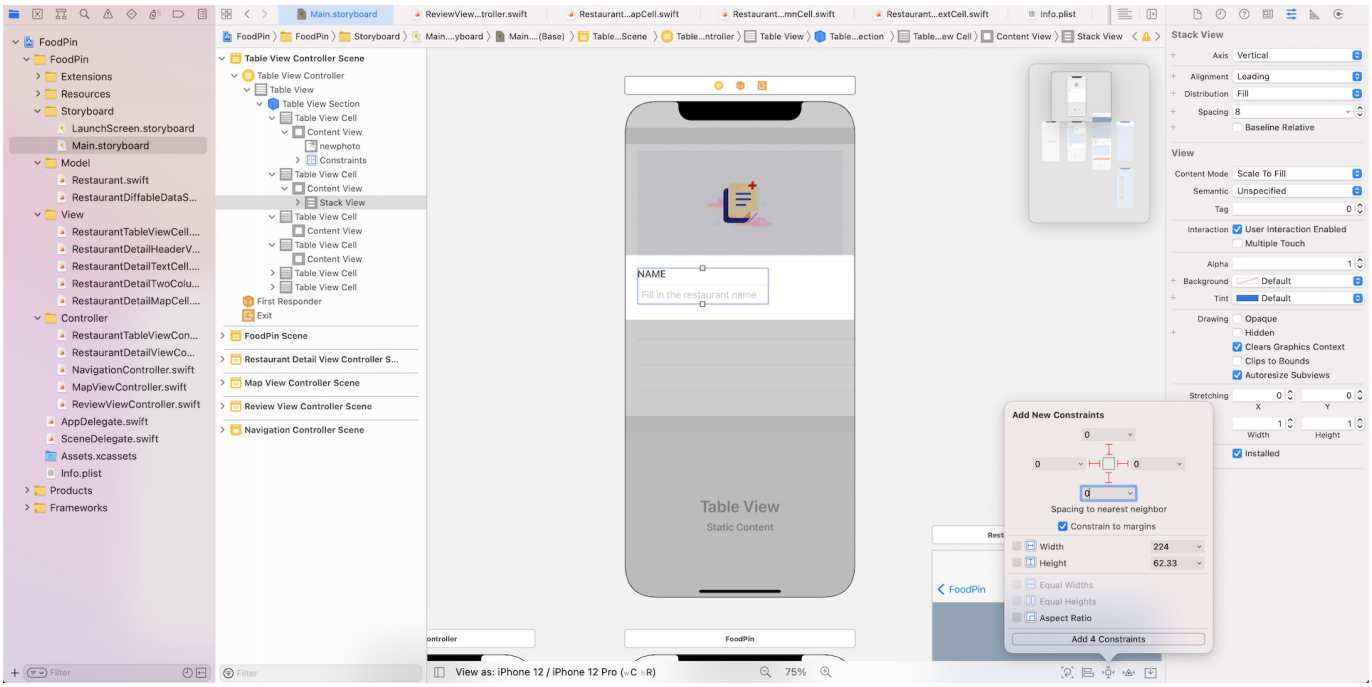


Figure 18-6. Embed both items in a stack view and define the layout constraints

The text field doesn't look as expected. Make sure the stack view is selected. Go to the Attributes inspector and change the *Alignment* option from `Leading` to `Fill`. Also, change the spacing value to `5`.

For the third, fourth, and fifth cells, repeat the same procedures. But set the label to *TYPE*, *ADDRESS*, and *PHONE* respectively.

For the last cell, which is for users to input the restaurant description, it needs a bigger text area. Resize the cell to make it taller. Then drag a label to the cell and name it *DESCRIPTION*. Again, change its font to *Headline* and font color to *dark gray*.

Note: If you find it hard to edit the last cell, you can select the view controller and go to the Attributes inspector to change the simulated size to *Freeform*. Set the height to *900* to make the view controller bigger.

Next, drag a Text View from the Object library to the cell. A text view is similar to a text field but allows you to enter multiple lines of text. In the Attributes inspector, replace the initial text with *A great restaurant to try out*. Also, change its background color to `System Gray 6 Color`. For the font, change it to *Body* text style.

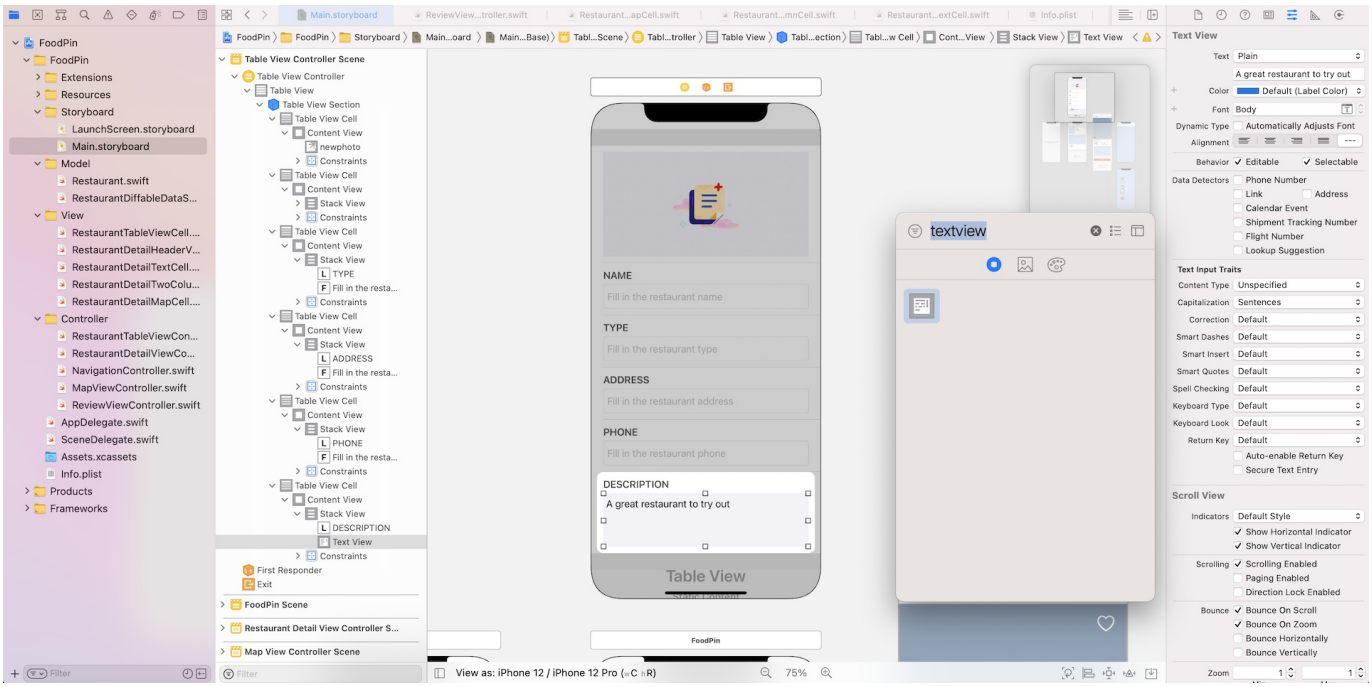


Figure 18-7. Adding a text view

Embed both the label and the text view in a stack view. In the Attributes inspector, change the *Alignment* option to `Fill` and set the *Spacing* value to `5` points. Then add a number of spacing constraints for the stack view. Please refer to figure 18-8 for details.

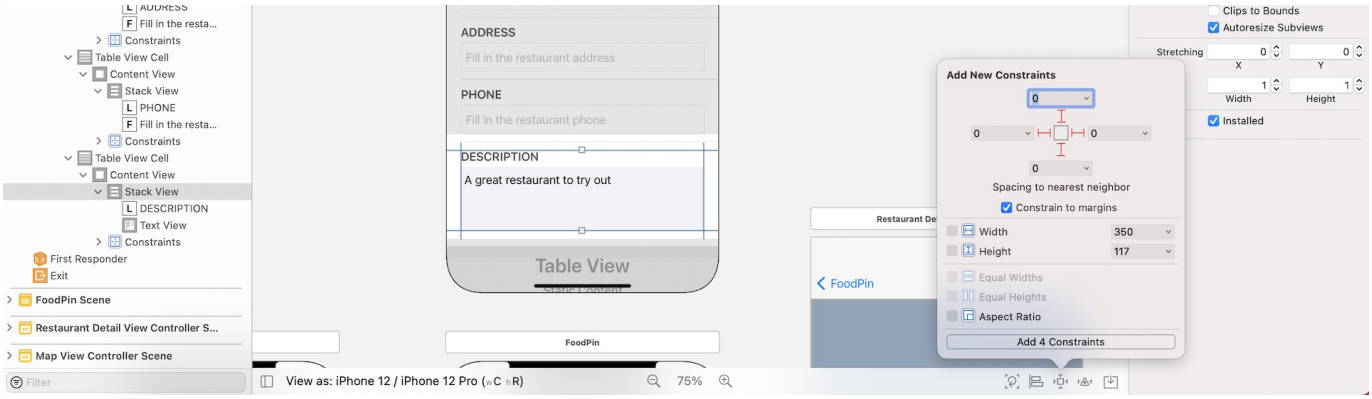


Figure 18-8. Defining the layout constraints for the stack view

To complete the layout of the screen, embed the table view controller in a navigation controller. Select the table view controller and go up to the Xcode menu. Select Editor > Embed in > Navigation Controller. Set the title of the navigation bar to `New Restaurant`. I prefer to use large titles for the navigation bar. So, select the navigation bar of the navigation controller. In the Attributes inspector, tick the *Prefers Large Titles* option.

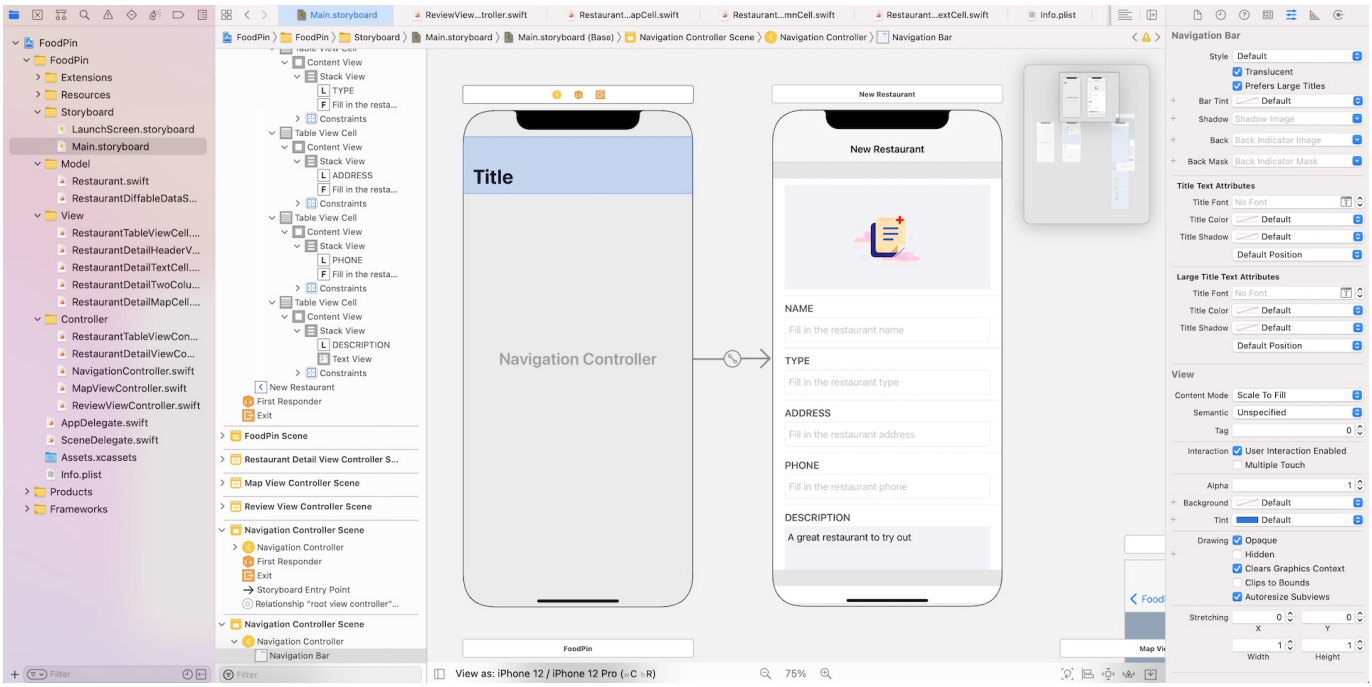


Figure 18-9. Setting the navigation bar title

Without writing a single line of code, you create a form by using a static table view. You're not limited to create forms with static table views. You can apply the same technique to create other screens like Settings.

> **Note:** If you find it hard to create the UI design, you can download the starter project from http://www.appcoda.com/resources/swift53/FoodPinStaticTableStarter.zip for reference.

Linking Up the New Restaurant Controller

So far we just created a standalone table view controller. We expect to bring up this controller when a user taps a `+` button in the top-right corner of the main controller (i.e. Restaurant Table View Controller). Obviously, we need to connect the button with the "New Restaurant" controller using a segue.

In the Interface Builder editor, first drag a *bar button* item from the Object library to the navigation bar of the Food Pin controller. In the Attribute inspector, change the *Title* option to blank, and set the *Image* option to `plus`. Also, change the tint to *Label Color* so that iOS will automatically change its color in dark mode.

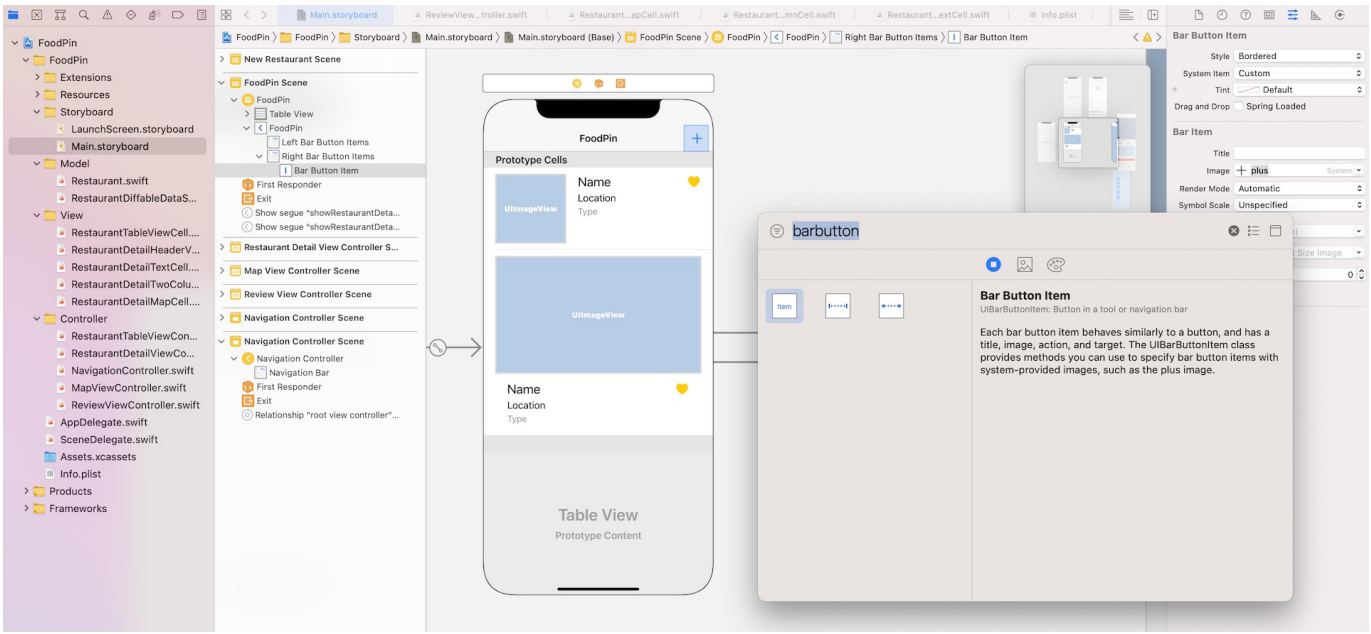


Figure 18-10. Add a bar button item in the navigation bar of the main screen

> **Note:** A bar button (UIBarButtonItem) is very similar to a standard button (UIButton). However, a bar button is specifically designed for navigation bars and toolbars.

Next, hold the control key, and drag the `+` icon to the navigation controller that embeds the *New Restaurant* view controller. Release the buttons and select *present modally* for the segue type. Set the identifier of the segue to `addRestaurant` in the Attribute inspector.

Quick tip: You can right-click any empty area on the storyboard, and select the zoom out option to have an overview of the storyboard.

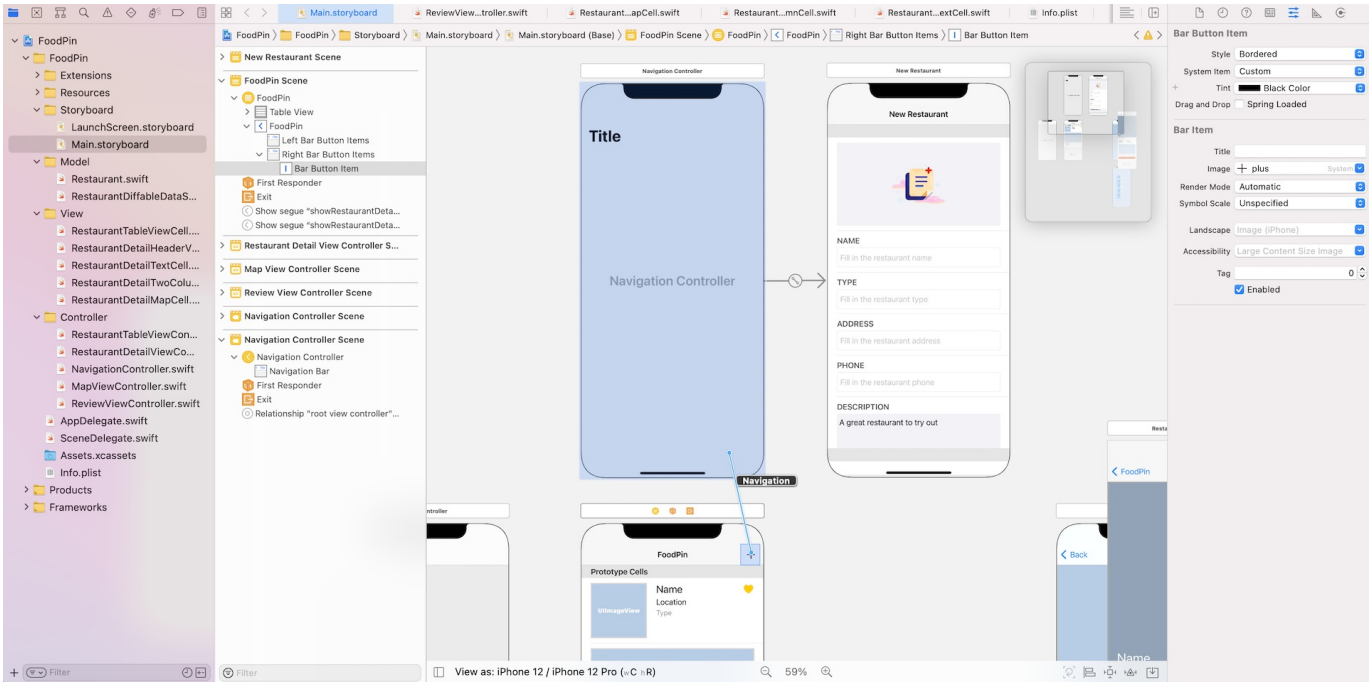


Figure 18-11. Adding a segue for the + icon

Similar to the modal view controller we built before, we need to provide a way for users to dismiss it. In the navigation bar of the *New Restaurant* view controller, add a bar button item to the top-left corner. In the Attribute inspector, set the title to *blank* and image to `xmark`. Again, change the tint to `Label Color`.

When a user taps the button, the modal view will be dismissed. In order to do that, we'll define an unwind segue. Select the `RestaurantTableViewController.swift` file and define the following unwind action:

```
@IBAction func unwindToHome(segue: UIStoryboardSegue) {
    dismiss(animated: true, completion: nil)
}
```

Once you add this method, Interface Builder can recognize the unwind action. Now, go back to `Main.storyboard`. Hold control key and drag from the *Close* button to the Exit icon of the scene dock (see figure 18-12). In the popover menu, select the `unwindToHomeWithSegue:` option for the action segue.

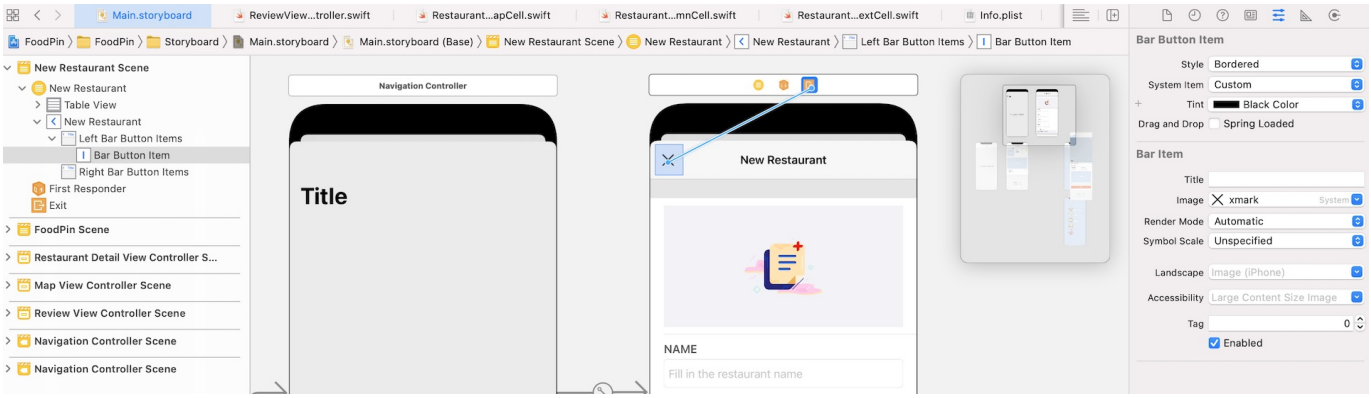


Figure 18-12. Adding an unwind segue for the Cancel button

Now let's have a quick test. Compile and run the project. After launching the app, tap the + icon. You should see the New Restaurant screen. You can either tap the `x` button or simply swipe down the view to dismiss it.

Building a Rounded Corner Text Field

Do you aware that the text field of the app doesn't have rounded corners? Also, the text of the text fields shown in the final design (refer to figure 18-1) is indented. However, the ones we are now having are so close to the edge.

The default text field doesn't come with text indentation. It's our responsibility to build a custom text field with these features. To do that, we will create a custom version of `UITextField`.

In the project navigator, right-click the `View` folder and select *New file...*. Choose the *Cocoa Touch Class* template. In the next screen, name the class `RoundedTextField` and set its subclass to `UITextField`. Confirm and save the file.

Once the file is created, replace the content with the following code:

```
import UIKit

class RoundedBorderTextField: UITextField {

    let padding = UIEdgeInsets(top: 0, left: 15, bottom: 0, right: 15);

    override func textRect(forBounds bounds: CGRect) -> CGRect {
        return bounds.inset(by: padding)
    }

    override func placeholderRect(forBounds bounds: CGRect) -> CGRect {
        return bounds.inset(by: padding)
    }

    override func editingRect(forBounds bounds: CGRect) -> CGRect {
        return bounds.inset(by: padding)
    }

    override func layoutSubviews() {
        super.layoutSubviews()

        self.layer.borderWidth = 1.0
        self.layer.borderColor = UIColor.systemGray5.cgColor
        self.layer.cornerRadius = 10.0
        self.layer.masksToBounds = true
    }

}
```

The first three methods are for text indentation, while the last method is for creating the rounded corners. Let me first explain how text indentation is done. Each of the three methods is designed for drawing the rectangle for the text field:

- `textRect(forBounds:)` - this method returns the drawing rectangle for the text field's text.
- `placeholderRect(forBounds:)` - this method returns the drawing rectangle for

the text field's placeholder text.

- `editingRect(forBounds:)` - this method returns the rectangle in which editable text can be displayed.

By default, these three methods return a rectangle without indentation. In our case, we want to indent the text in the text field. Therefore, we override these methods and return a rectangle with indentation. We use `UIEdgeInsets` to create a padding area for both left and right side of the text field. And then we call `bounds.inset(by: padding)` to apply that edge insets to shrink the rectangle.



Figure 18-13. Use UIEdgeInsets and UIEdgeInsetsInsetRect together to shrink the rectangle

The default style of the text field does have rounded corners. But I want to change the corner radius to make the corners more rounded. You should know how to create rounded corners for a text field, as we have discussed the technique in earlier chapters. What you need to do is to assign a value (greater than zero) to the `cornerRadius` property. Here we set it to `10` points. On top of that, we change the border's width and color.

So, what the heck is `layoutSubviews()` ? It is called every time when the text field is laid out. Thus, we put these lines of code in the method:

```
self.layer.borderWidth = 1.0
self.layer.borderColor = UIColor.systemGray5.cgColor
self.layer.cornerRadius = 10.0
self.layer.masksToBounds = true
```

Now that we have created a rounded-corner version of the text field, but we haven't applied it to the text field in the controller. Let's go back to `Main.storyboard`. Select the text field for the restaurant name. Go to the Identity inspector and change its custom class to `RoundedTextField`.

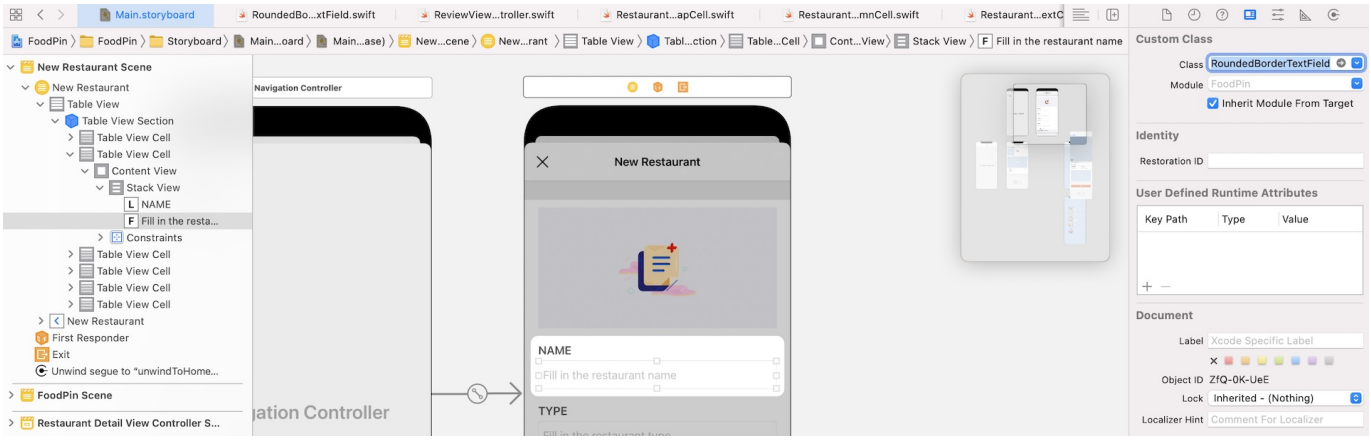


Figure 18-14. Setting the custom class of the text field

Repeat the procedures for the rest of the text fields (except the text view for the restaurant description). Once you finish, you can run the app to try it out. The text fields should now have text indentation.

There is one more thing that you can tweak to make the form look better. You can select the table view in document outline and change the *Separator* option to `None`. This will remove the separators between cells.

Moving onto the Next Text Field

Have you tried to key in something in the text field? If you haven't tried that, please run the app again and give it a shot. By default, the simulator doesn't show the software keyboard. You can press command+k to bring it up.

Do you find something you can improve? For now, when you hit the return key, the cursor doesn't move to the next text field. It just stays where it is.

Let's see how to move the cursor to the next text field when the user taps the return key. The general idea is that we assign each of the text fields a tag. Say, for the *Name* text field, we set its tag to `1`. The *Type* text field will have a tag value of `2`. When a user hits the return key, the text field recognizes the event. Next, it looks up the next text field by increasing the tag value by 1 and then tell the cursor to move to that text field.

In order to implement this solution, we have to first create a custom class for the New Restaurant controller. In the project navigator, right-click the `Controller` folder and select *New file...*. Choose *Cocoa Touch Class* as the template. Set the class name to `NewRestaurantController` and its subclass to `UITableViewController`.

Once the `NewRestaurantController.swift` file is created, remove the following generated methods because we do not need them for the static table view:

```
override func numberOfSections(in tableView: UITableView) -> Int {
    // #warning Incomplete implementation, return the number of sections
    return 0
}

override func tableView(_ tableView: UITableView, numberOfRowsInSection se
ction: Int) -> Int {
    // #warning Incomplete implementation, return the number of rows
    return 0
}
```

Next, create an outlet variable for each of the text fields and the text view:

```
@IBOutlet var nameTextField: RoundedBorderTextField! {
    didSet {
        nameTextField.tag = 1
        nameTextField.becomeFirstResponder()
        nameTextField.delegate = self
    }
}

@IBOutlet var typeTextField: RoundedBorderTextField! {
    didSet {
        typeTextField.tag = 2
        typeTextField.delegate = self
    }
}

@IBOutlet var addressTextField: RoundedBorderTextField! {
    didSet {
        addressTextField.tag = 3
        addressTextField.delegate = self
    }
}

@IBOutlet var phoneTextField: RoundedBorderTextField! {
    didSet {
        phoneTextField.tag = 4
        phoneTextField.delegate = self
    }
}

@IBOutlet var descriptionTextView: UITextView! {
    didSet {
        descriptionTextView.tag = 5
        descriptionTextView.layer.cornerRadius = 10.0
        descriptionTextView.layer.masksToBounds = true
    }
}
```

Whether it's a button or a text field (or any other view objects), you can make use of the `tag` property to store a tag value. In the code above, we assign each of the text fields and the text view with a different tag value. The name text field has the smallest tag value and it is the first responder.

The next question is how we can detect the pressing of the return button?

`UITextFieldDelegate` is the protocol we need to adopt in order to process the event. The protocol provides developers a set of optional methods to manage the editing of a text field. The method that is particularly useful for this scenario is called `textFieldShouldReturn(_:)`. When a user presses the return key in a text field, the method will be invoked. We can implement the method to move the cursor to the next text field.

We will create an extension to adopt the `UITextFieldDelegate` protocol like this:

```
extension NewRestaurantController: UITextFieldDelegate {
    func textFieldShouldReturn(_ textField: UITextField) -> Bool {
        if let nextTextField = view.viewWithTag(textField.tag + 1) {
            textField.resignFirstResponder()
            nextTextField.becomeFirstResponder()
        }

        return true
    }
}
```

In the code above, we increase the current tag value by 1 and call `view.viewWithTag` to retrieve the next text field. Then we remove the focus of the current text field by calling `resignFirstResponder()` and let the next text field become the first responder.

It's not ready to test the app yet. We have to go back to `Main.storyboard` to connect the outlets.

In Interface Builder, select the New Restaurant controller. Go to the Identity inspector and set its custom class to `NewRestaurantController`. Then right-click *New Restaurant Controller* to establish the connections with the outlet variables.

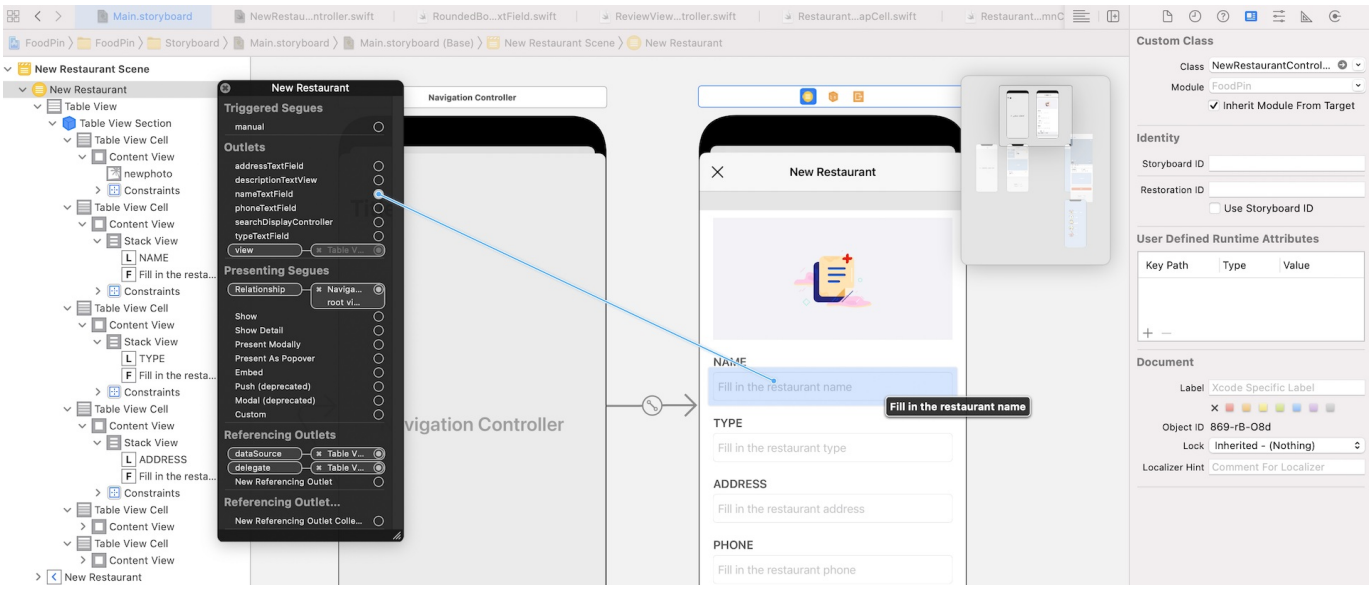


Figure 18-15. Connecting the outlets

Now test the app again! If you tap the return key while editing a text field, the cursor will automatically move to the next field.

Customizing the Navigation Bar

Before we move onto the usage of `UIImagePickerController`, let's customize the navigation bar. This is completely optional, but I want to make the UI consistent.

Update the `viewDidLoad()` method of the `NewRestaurantController` class like this:

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

    // Customize the navigation bar appearance
    if let appearance = navigationController?.navigationBar.standardAppear
ance {

        if let customFont = UIFont(name: "Nunito-Bold", size: 40.0) {

            appearance.titleTextAttributes = [.foregroundColor: UIColor(na
med: "NavigationBarTitle")!]
            appearance.largeTitleTextAttributes = [.foregroundColor: UICol
or(named: "NavigationBarTitle")!, .font: customFont]
        }

        navigationController?.navigationBar.standardAppearance = appearanc
e
        navigationController?.navigationBar.compactAppearance = appearance
        navigationController?.navigationBar.scrollEdgeAppearance = appeara
nce
    }
}
```

I have discussed the customization of navigation bars in chapter 15, so I believe you understand what the code means. If not, go back to read chapter 15.

Displaying the Photo Library Using UIImagePickerController

When the first table view cell (i.e. the one with the camera icon) is tapped, the app will let users choose the photo source: *Camera* or *Photo library*. The UIKit framework provides a convenient API called `UIImagePickerController` for loading photos from the photo library. What's great is that the same API can be used to bring up a camera interface for taking a picture.

The simulator doesn't support the camera feature. If you want to test an app that utilizes the built-in camera, you'll need a real iOS device.

> **Note:** Starting from Xcode 7, you no longer need to enroll in the Apple Developer Program before you can test your app on a real iOS device. I will show you how to deploy an app to the device for testing in chapter 25.

In order to allow users to choose the photo source, we will use `UIAlertController` to bring up an action sheet and ask the user to pick an option. Insert the following method in the `NewRestaurantController` class to detect the touch:

```
override func tableView(_ tableView: UITableView, didSelectRowAt indexPath
: IndexPath) {
    if indexPath.row == 0 {

        let photoSourceRequestController = UIAlertController(title: "", me
ssage: "Choose your photo source", preferredStyle: .actionSheet)

        let cameraAction = UIAlertAction(title: "Camera", style: .default,
 handler: { (action) in
            if UIImagePickerController.isSourceTypeAvailable(.camera) {
                let imagePicker = UIImagePickerController()
                imagePicker.allowsEditing = false
                imagePicker.sourceType = .camera

                self.present(imagePicker, animated: true, completion: nil)
            }
        })

        let photoLibraryAction = UIAlertAction(title: "Photo library", sty
le: .default, handler: { (action) in
            if UIImagePickerController.isSourceTypeAvailable(.photoLibrary
) {
                let imagePicker = UIImagePickerController()
```

```
                imagePicker.allowsEditing = false
                imagePicker.sourceType = .photoLibrary

                self.present(imagePicker, animated: true, completion: nil)
            }
        })

        photoSourceRequestController.addAction(cameraAction)
        photoSourceRequestController.addAction(photoLibraryAction)

        // For iPad
        if let popoverController = photoSourceRequestController.popoverPre
sentationController {
            if let cell = tableView.cellForRow(at: indexPath) {
                popoverController.sourceView = cell
                popoverController.sourceRect = cell.bounds
            }
        }

        present(photoSourceRequestController, animated: true, completion:
nil)

    }
}
```

As we've discussed before, the `tableView(_:didSelectRowAt:)` method is called when a cell is selected. In this case, we only want to bring up an alert controller when the first cell is selected. Therefore, we have a conditional check at the very beginning.

We created two alert actions in the code above: one is for using the camera and the other is for loading the photo library. If you look into the code of both actions, the code snippets are basically the same, except the `sourceType` is different.

To load the photo library, all you need to do is create an instance of `UIImagePickerController` and set its `sourceType` to `.photoLibary`. If you want to bring the camera for taking photos, you just need to set the `sourceType` to

`.camera`. You then call up the `present(_:animated:completion:)` method to bring up the photo library/camera.

That's it. Easy, right? Sometimes, the user may not allow you to access the photo library. As a good practice, you should always use the class method `isSourceTypeAvailable` to verify if a particular media source is available.

In iOS 10 or later, for privacy reasons, you have to explicitly describe the reason why your app accesses the user's photo library or camera. If you fail to do so, you may end up with an error. You will need to add two keys (`NSPhotoLibraryUsageDescription` and `NSCameraUsageDescription`) in the `Info.plist` file and provide your reason.

You can treat the `Info.plist` file as a configuration file of your app project. It contains the meta information of your app such as bundle ID, supported orientation, and its content is structured using XML.

Now select `Info.plist` in the project navigator. Right-click *Information Property List* in the editor and select *Add Row*. Choose "Privacy - Photo Library Usage Description" for the key and set the value to:

You need to grant the app access to your photo library so you can pick your favorite restaurant photo.

Repeat the same procedure and add another row. Set the key to "Privacy - Camera Usage Description" and the value to:

You need to grant the app access to your camera in order to take photos.

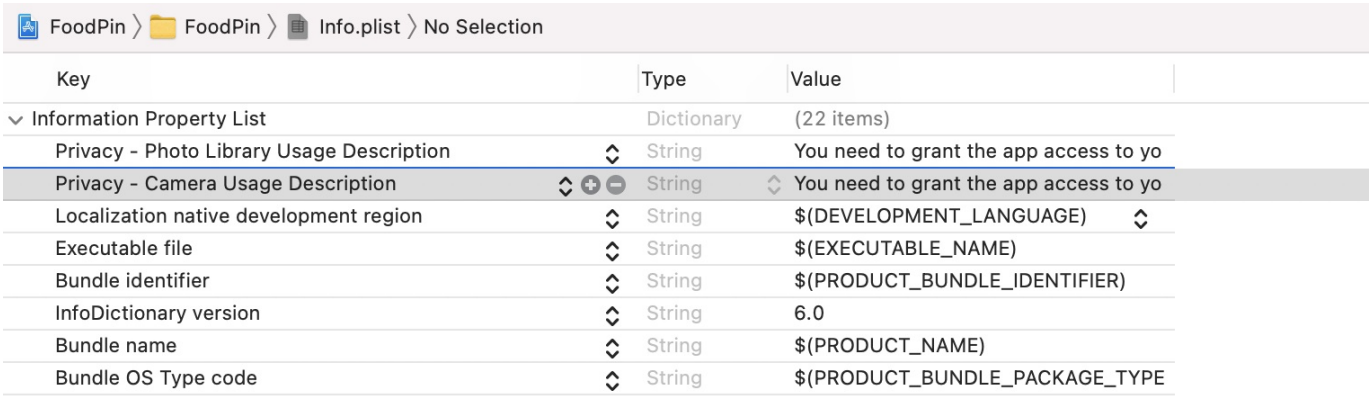


Figure 18-16. Add a key to specify the reason

Now compile and run the app. Tapping the photo icon should bring up the action sheet. Choose Photo library. Your app should be able to access the photo library. The simulator already comes with several sample photos. If you want to add your own photos, drag them from Finder to the simulator. This will automatically add the photos to the Photos app of the simulator.

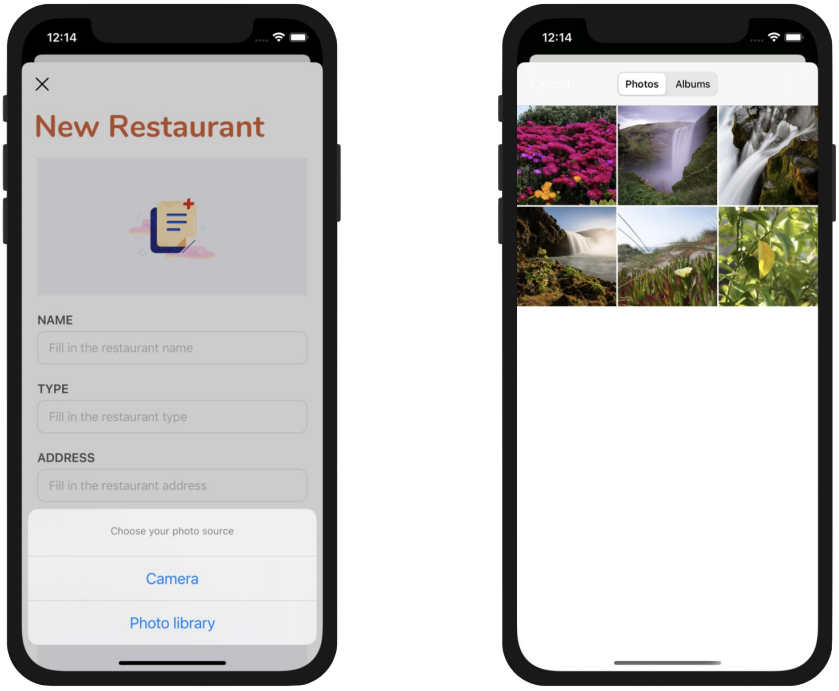


Figure 18-17. Loading the built-in photo library in FoodPin app

Adopting the UIImagePickerControllerDelegate Protocol

If you select a photo from the photo library, it wouldn't show up in the image view. How do you know which photo the user selects? To interact with the image picker interface, the `NewRestaurantController` class has to adopt two delegates: `UIImagePickerControllerDelegate` and `UINavigationControllerDelegate`.

When a user chooses a photo from the photo library, the `imagePickerController(_:didFinishPickingMediaWithInfo:)` method of the delegate is called.

```
func imagePickerController(_ picker: UIImagePickerController, didFinishPic
kingMediaWithInfo info: [UIImagePickerController.InfoKey : Any]) {

}
```

By implementing the method, we can get the selected photo from the method's parameter. Before implementing the method, let's declare an outlet variable for the image view in `NewRestaurantController`. Later we can set the image view with the chosen image.

```
@IBOutlet var photoImageView: UIImageView! {
    didSet {
        photoImageView.layer.cornerRadius = 10.0
        photoImageView.layer.masksToBounds = true
    }
}
```

Go to Interface Builder and connect the image view of the cell with the `photoImageView` outlet variable.

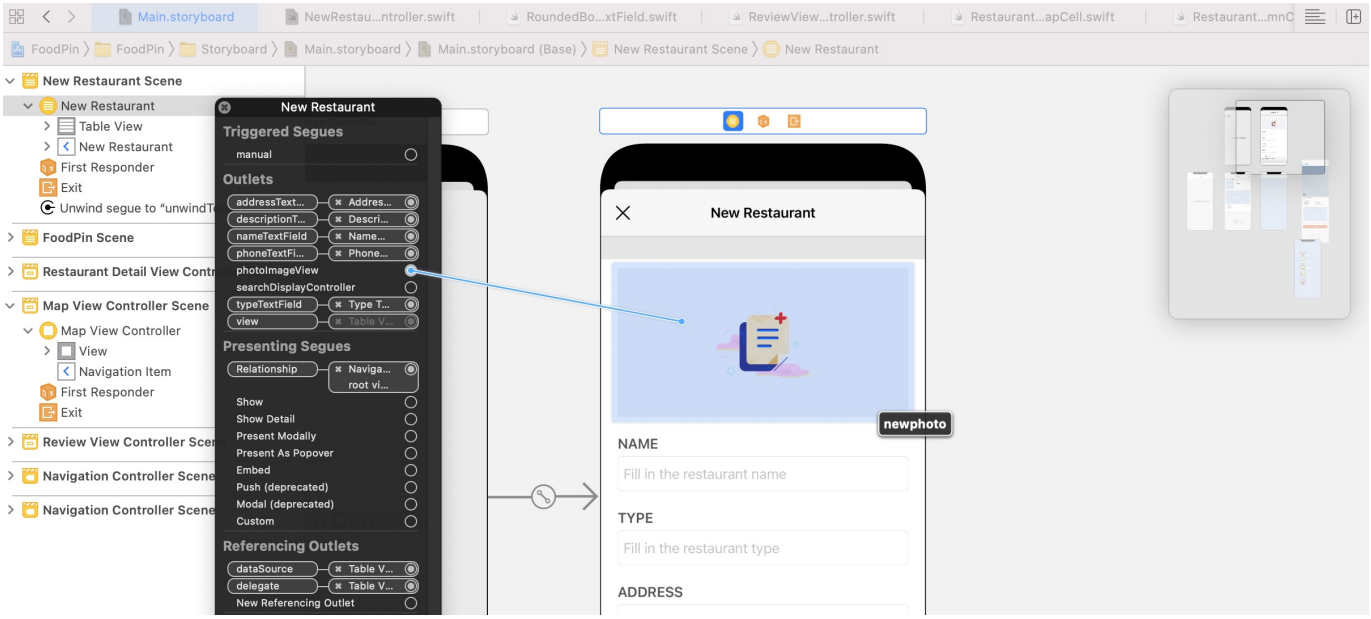


Figure 18-18. Establishing a connection between the image view and the outlet variable

Next, create an extension to adopt the required protocols and implement the delegate method like this:

```
extension NewRestaurantController: UIImagePickerControllerDelegate, UINavi
gationControllerDelegate {

    func imagePickerController(_ picker: UIImagePickerController, didFinis
hPickingMediaWithInfo info: [UIImagePickerController.InfoKey : Any]) {

        if let selectedImage = info[UIImagePickerController.InfoKey.origin
alImage] as? UIImage {
            photoImageView.image = selectedImage
            photoImageView.contentMode = .scaleAspectFill
            photoImageView.clipsToBounds = true
        }

        dismiss(animated: true, completion: nil)
    }

}
```

When the method is called, the system passes you an `info` dictionary object that contains the selected image. `UIImagePickerControllerOriginalImage` is the key of the image selected by the user.

The above code assigns the image view with the selected image. We also change the content mode so that the image is displayed in *aspect fill* mode. Lastly, we call the `dismiss` method to dismiss the image picker.

Don't forget to add this line of code in the `tableView(_:didSelectRowAt:)` method, immediately after the declaration of `imagePicker` for both alert actions:

```
imagePicker.delegate = self
```

Now you're ready to test the app. Try to select a photo from the album and it should be displayed right in the image view.

Defining Auto Layout Constraints Programmatically

The New Restaurant view works pretty good. Before we close this chapter, I want to show you how to define auto layout constraints using code. You should know how to work with layout constraints in Interface Builder. As you continue to develop your programming skill, you may prefer to define these layout constraints programmatically by using `NSLayoutAnchor` .

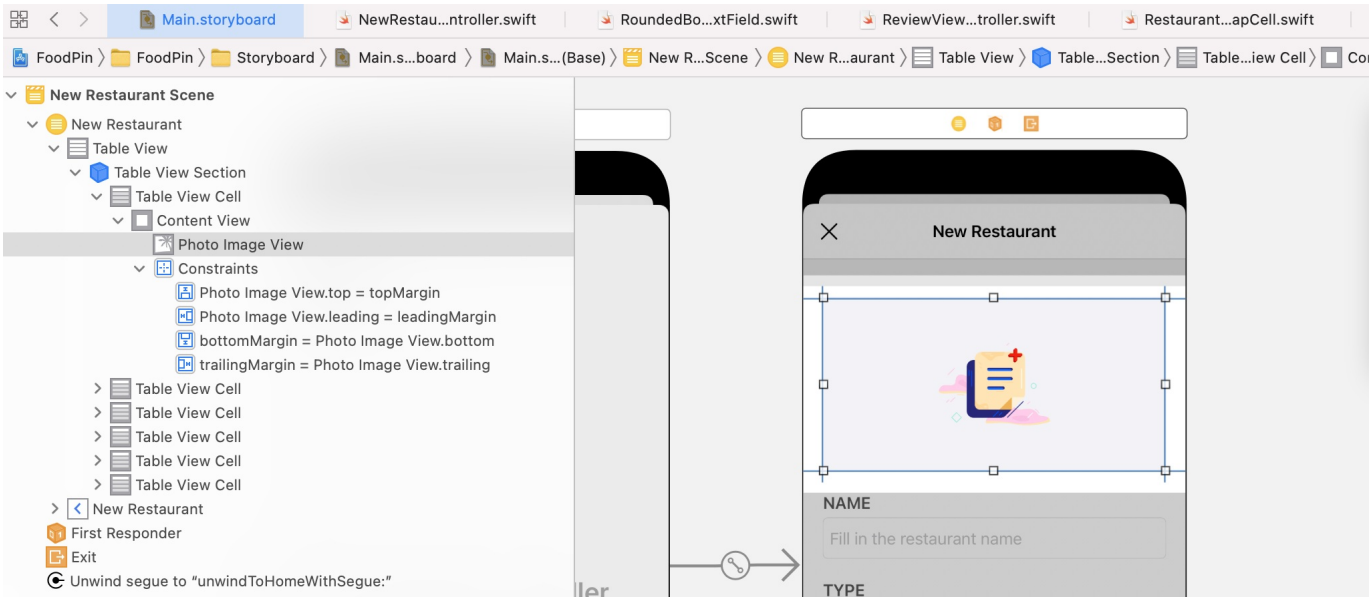


Figure 18-19. Sample spacing constraints for the photo view

Let's use the photo view of the *NewRestaurantController* as an example. Earlier, we already define a set of spacing constraints to arrange the layout of the image view. All sides of the image view should be 0 points from the margin.

You should know how to define the spacing constraints in Interface Builder. The question is how can you define these layout constraints programmatically?

When writing code for auto layout, you can define constraints by using an API called `NSLayoutAnchor`. Take the leading constraint as an example. We specify that there is no space between the leading side of the image view and the leading margin of its superview. Here is the constraint you can find in the document outline:

```
Photo Image View.leading = leadingMargin
```

The same constraint can be written using `NSLayoutAnchor` like this:

```
let margins = photoImageView.superview!.layoutMarginsGuide
photoImageView.leadingAnchor.constraint(equalTo: margins.leadingAnchor).is
Active = true
```

A layout constraint actually defines a relationship between two user interface objects. If you compare the constraint's properties defined in the Attribute inspector with those defined in the code, you should reveal their relationship. The `superView` of our `photoImageView` refers to the content view of the cell. The `layoutMarginsGuide` represents the view's layout margin. What the code does is to pin the leading anchor of the image view to the leading anchor of the content view's margin.

By default, the constraint is not activated after instantiation. You have to set its `isActive` property to `true` in order to activate it.

Before you write the code, please switch back to `Main.storyboard` and delete the constraints of the `Photo Image View`, as shown in figure 18-19.

Now head back to `NewRestaurantController.swift` and write the code for the layout constraints. We have to add four layout constraints for the top, bottom, leading (i.e. left), and trailing (i.e. right) sides of the image view. Insert these lines of code in the `viewDidLoad()` method:

```
// Get the superview's layout
let margins = photoImageView.superview!.layoutMarginsGuide

// Disable auto resizing mask to use auto layout programmatically
photoImageView.translatesAutoresizingMaskIntoConstraints = false

// Pin the leading edge of the image view to the margin's leading edge
photoImageView.leadingAnchor.constraint(equalTo: margins.leadingAnchor).is
Active = true

// Pin the trailing edge of the image view to the margin's leading edge
photoImageView.trailingAnchor.constraint(equalTo: margins.trailingAnchor).
isActive = true

// Pin the top edge of the image view to the margin's leading edge
photoImageView.topAnchor.constraint(equalTo: margins.topAnchor).isActive =
true

// Pin the bottom edge of the image view to the margin's leading edge
photoImageView.bottomAnchor.constraint(equalTo: margins.bottomAnchor).isAc
tive = true
```

Here we define and activate four layout constraints for the image view so that the image is bound within the margins of the cell. To create the layout constraints programmatically, you have to set the `translatesAutoresizingMaskIntoConstraints` property of the view to `false`. Now let's run the app again. The layout of the *New Restaurant* form is the same as before but the constraints of the image view is set programmatically.

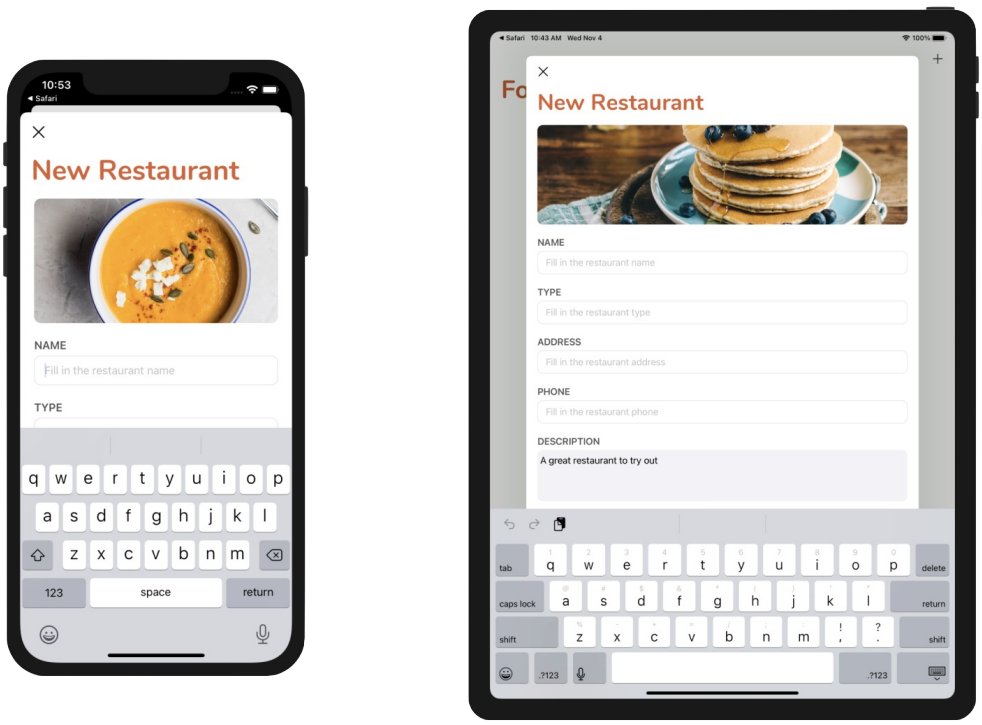


Figure 18-20. The image is scaled and displayed perfectly

Hiding Keyboard

Right now, you can't dismiss the keyboard on iPhone. We want to let users dismiss the keyboard when tapping any blank area of the form. To do that, you can insert the following code in the `viewDidLoad()` method:

```
let tap = UITapGestureRecognizer(target: view, action: #selector(UIView.en
dEditing))
tap.cancelsTouchesInView = false
view.addGestureRecognizer(tap)
```

In the code, we create a tap gesture recognizer to the view. Whenever the user taps the form view (excluding those text fields), we call the `endEditing` method to dismiss the software keyboard.

Exercise #1

Your next task is to add a *Save* button in the top-right corner of the navigation bar. When the user taps the button, you get the values of the text fields, validate them and print them out to the console.

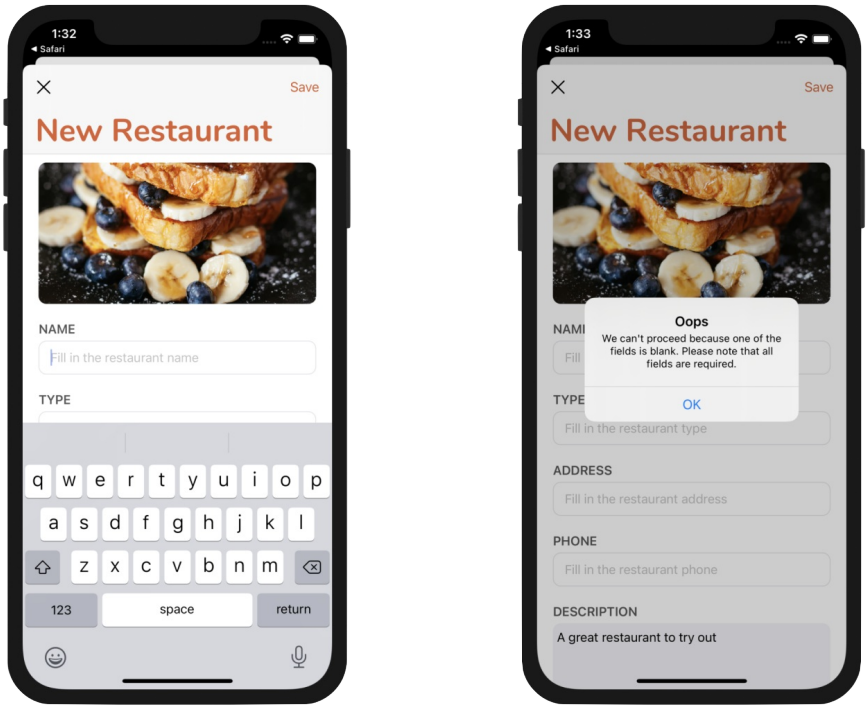


Figure 18-21. Sample alert for the form validation

This exercise is more challenging than the previous one, so I will give you some hints. The first hint is that you will need to refer to the documentation of `UITextField` (https://developer.apple.com/documentation/uikit/uitextfield). By reading the API document, you will find out which attribute is used for storing the text in the text field.

Secondly, you have to declare a `saveButtonTapped` action method in the `NewRestaurant` class. When the *Save* button is tapped, this action method will be called. You will have to connect the button to the action method in Interface Builder. In the body of the method, you will write code to perform a couple of things:

1. **Form validation** - check if any of the input text is blank. Otherwise, use `UIAlertController` to present an alert prompt.
2. **Console output** - print out the form values to console. Here is a sample output:

```
Name: Izakaya Sozai
Type: Japanese
Location: 1500 Irving St San Francisco, CA 94122
Phone: (415) 742-5122
Description: A great restaurant to try out
```

I have included the solution for you. You can find the download link at the end of the chapter. However, I highly recommend you to spend some time on the exercise. It's gonna be fun to figure out your own implementation.

Summary

In this chapter, we demonstrate how to create a static table view using Interface Builder. While you can use a table view to display dynamic data from data source, a static table view provides a great way to display a finite quantity of data that is already known beforehand. You also learned how to access the built-in photo library. `UIImagePickerController` makes it so simple for developers to access the built-in camera and photo library.

Furthermore, we walked through `NSLayoutAnchor` together. You should now have some ideas about how to create layout constraints programmatically. You commonly use Interface Builder to define the layout constraints. That's good enough for most layouts. However, when you need to build a dynamic UI that changes within the application, you will need to modify the layout constraints at runtime. I only give you a brief introduction to `NSLayoutAnchor`. You can refer to the official documentation (https://developer.apple.com/documentation/uikit/nslayoutanchor) for details.

For reference, you can download the complete Xcode project from http://www.appcoda.com/resources/swift53/FoodPinStaticTableView.zip.

Next up, we will talk about Core Data and see how to save the restaurant data in a database.

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.

If you haven't done the previous exercise, you can download this project's templates (http://www.appcoda.com/resources/swift53/FoodPinStaticTableView.zip) to start with.

What is Core Data?

When we talk about persistent data, you probably think of databases. If you are familiar with Oracle or MySQL, you know that a relational database stores data in the form of tables, rows, and columns; your app talks to the database by using a SQL (Structured Query Language) query. However, don't mix up Core Data with databases. Though the SQLite database is the default persistent store for Core Data on iOS, Core Data is not exactly a relational database - it is actually a framework that lets developers interact with databases (or other persistent storage) in an object-oriented way.

> **Note:** If you have no idea of SQL and want to understand what it is, check out this simple tutorial (https://www.w3schools.com/sql/sql_intro.asp).

Take the FoodPin app as an example. If you want to save the data to a database, you are responsible for writing the code to connect to the database and retrieve or update the data using SQL. This would be a burden for developers, especially for those who do not know SQL.

Core Data provides a simpler way to save data to a persistent store of your choice. You can map the objects in your apps to the table in the database. Simply put, it allows you to manage records (select/insert/update/delete) in the database without even knowing any SQL.

Core Data Stack

Before we start working on the project, you need to first have a basic understanding of the Core Data Stack (see figure 19-1):

- **Managed Object Context** – Think of it as a *scratch pad* or temporary memory area containing objects that interact with data in the persistent store. Its job is to manage objects created and returned using Core Data framework.

Among the components in the Core Data stack, the managed object context, an instance of the `NSManagedObjectContext` class, is the one you'll work directly with most of the time. In general, whenever you need to fetch and save objects in the persistent store, the context is the first component you'll interact with.

- **Managed Object Model** – This describes the schema that you use in the app. If you have some background in databases, think of this as the database schema. However, the schema is represented by a collection of objects (also known as entities). For example, a collection of model objects can be used to represent the collection of restaurants in the FoodPin app. In Xcode, the managed object model is defined in a file with the extension `.xcdatamodeld`. You can use the visual editor to define the entities and their attributes and relationships.
- **Persistent Store Coordinator** – As its name suggests, it is the coordinator of the Core Data stack. It sits between the managed object context and the persistent store. While the figure shows a single store, a Core Data application can have multiple stores. The persistent store coordinator, an instance of `NSPersistentStoreCoordinator`, is the party responsible for managing different persistent object stores and saving the objects to the stores. You seldom interact with the persistent store coordinator directly when using Core Data.
- **Persistent Store** - This is the repository in which your data is actually stored. Usually, it's a database, and SQLite is the default database. But it can also be a binary or XML file.

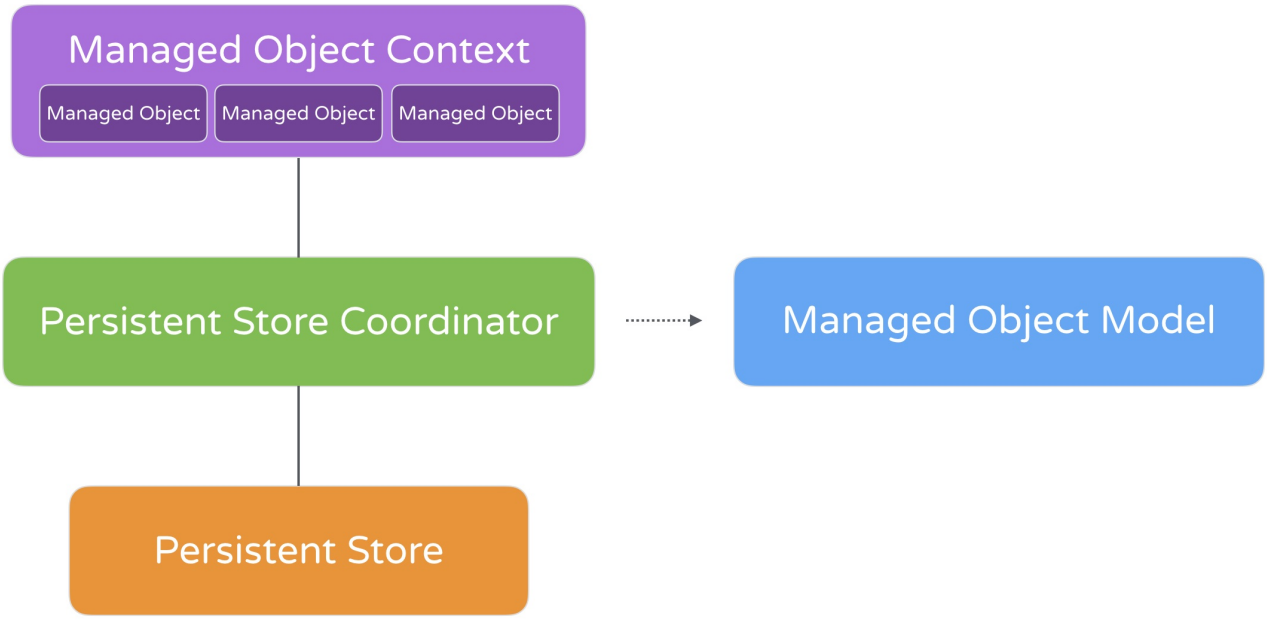


Figure 19-1. Core Data Stack

That looks complicated, right? Definitely. Apple's engineers were also aware of the issue. Starting from iOS 10, the team introduced a class called `NSPersistentContainer` to simplify the management of Core Data stack in your apps. `NSPersistentContainer` is the class you will deal with for saving and retrieving data.

Feeling confused? No worries. You will understand what I mean as we convert the FoodPin app from arrays to Core Data.

Using Core Data Template

The simplest way to use Core Data is to enable the Core Data option during project creation. Xcode will generate the required code in `AppDelegate.swift` and create the data model file for Core Data.

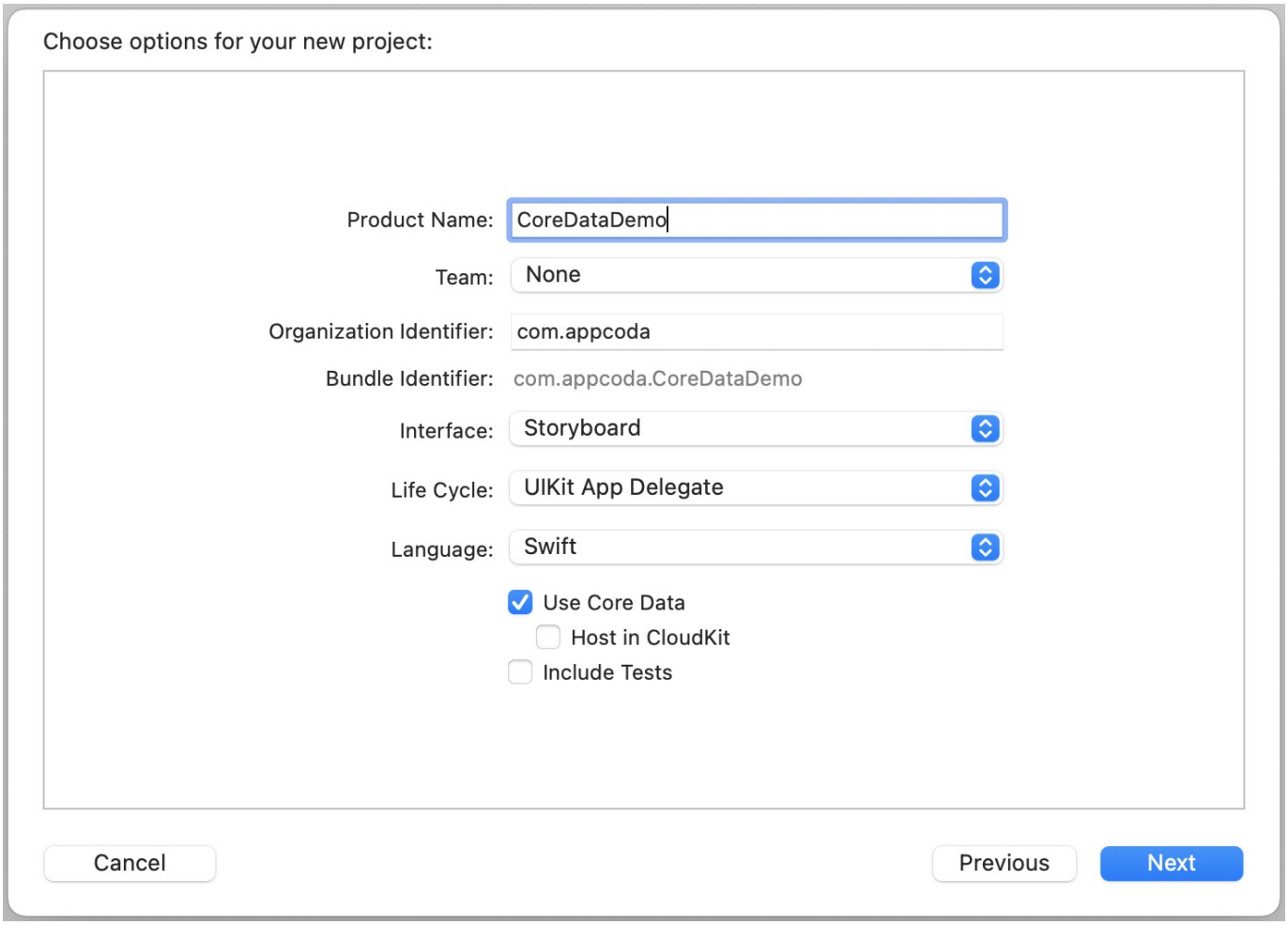


Figure 19-2. Enable Core Data option

If you create a *CoreDataDemo* project with Core Data option enabled, you will see the following variables and method generated in the `AppDelegate` class:

```
// MARK: - Core Data stack

lazy var persistentContainer: NSPersistentContainer = {
    /*
     The persistent container for the application. This implementation
     creates and returns a container, having loaded the store for the
     application to it. This property is optional since there are legitima
te
     error conditions that could cause the creation of the store to fail.
```

```
        */
        let container = NSPersistentContainer(name: "CoreDataDemo")
        container.loadPersistentStores(completionHandler: { (storeDescription,
 error) in
            if let error = error as NSError? {
                // Replace this implementation with code to handle the error a
ppropriately.
                // fatalError() causes the application to generate a crash log
 and terminate. You should not use this function in a shipping application
, although it may be useful during development.

                /*
                 Typical reasons for an error here include:
                 * The parent directory does not exist, cannot be created, or
disallows writing.
                 * The persistent store is not accessible, due to permissions
or data protection when the device is locked.
                 * The device is out of space.
                 * The store could not be migrated to the current model versio
n.
                 Check the error message to determine what the actual problem
was.
                 */
                fatalError("Unresolved error \(error), \(error.userInfo)")
            }
        })
        return container
    }()

    // MARK: - Core Data Saving support

    func saveContext () {
        let context = persistentContainer.viewContext
        if context.hasChanges {
            do {
                try context.save()
            } catch {
                // Replace this implementation with code to handle the error a
ppropriately.
```

```
            // fatalError() causes the application to generate a crash log
 and terminate. You should not use this function in a shipping application
, although it may be useful during development.
            let nserror = error as NSError
            fatalError("Unresolved error \(nserror), \(nserror.userInfo)")
        }
    }
}
```

The generated code provides a variable and a method:

- The variable `persistentContainer` is an instance of `NSPersistentContainer`, which was initialized with a persistent store named "CoreDataDemo". This container encapsulates the Core Data stack in your application. Before iOS 10, you will need to create and manage the creation of the managed object model (`NSManagedObjectModel`), persistent store coordinator (`NSPersistentStoreCoordinator`), and the managed object context (`NSManagedObjectContext`). The introduction of `NSPersistentContainer` simplifies all that. Later, you just use this variable to interact with the Core Data stack.
- The `saveContext()` method is a helper that provides data saving. When you need to insert/update/delete data in the persistent store, you will call this method.

The question is how we can use this code template in our existing Xcode project. You can simply copy and paste the code into the `AppDelegate.swift` file of your Food Pin project, but you will need to make a minor change.

```
let container = NSPersistentContainer(name: "CoreDataDemo")
```

The original code template was generated for the `CoreDataDemo` project. Xcode names the SQLite and data model file using the project name. For the FoodPin project, we use the name *FoodPin* instead. So change the above line of code to the

following:

```
let container = NSPersistentContainer(name: "FoodPin")
```

Finally, add the `import` statement at the beginning of the `AppDelegate` class to import the Core Data framework:

```
import CoreData
```

Quick note: For reference, you can also download this project template (http://www.appcoda.com/resources/swift53/FoodPinCoreDataStarter.zip)

Creating the Data Model

Now that you've prepared the code for accessing the Core Data stack, let's move on to the creation of the data model. Every Core Data application has a data model that describes the structure of the stored data. In the project navigator, right-click the `FoodPin` folder and select `New File...`. Scroll down to the Core Data section and select *Data Model*.

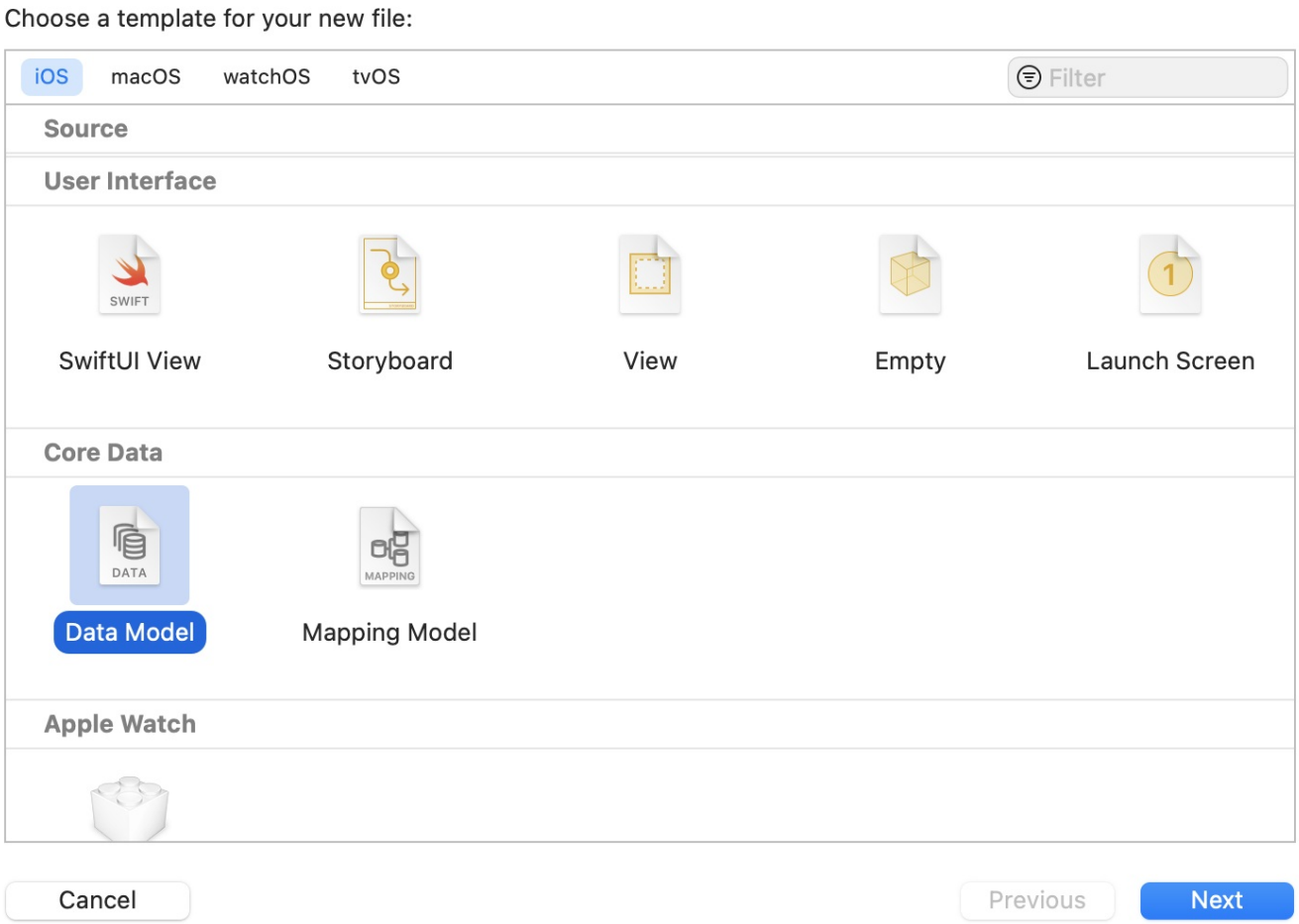


Figure 19-3. Creating the data model using Data Model template

Name the model `FoodPin` and click `Create` to create the data model. Once created, you should find a file named `FoodPin.xcdatamodeld` in the project navigator. Select it to open the data model editor. From here, you can create entities for your data model.

Since we would like to store the `Restaurant` object in the database, we will create a `Restaurant` entity that matches the `Restaurant` struct in our code. To create an entity, click the *Add Entity* button at the bottom of the editor pane and name the entity `Restaurant`.

In order to save the data of the `Restaurant` object to the database, we have to add several attributes for the entity that align with the attributes of the object. Simply click the `+` button under the attributes section to create a new attribute. Add eight attributes for the `Restaurant` entity including *name*, *type*, *location*, *phone*, *summary*, *image*, *isFavorite*, and *ratingText*. Refer to figure 19-4 for details.

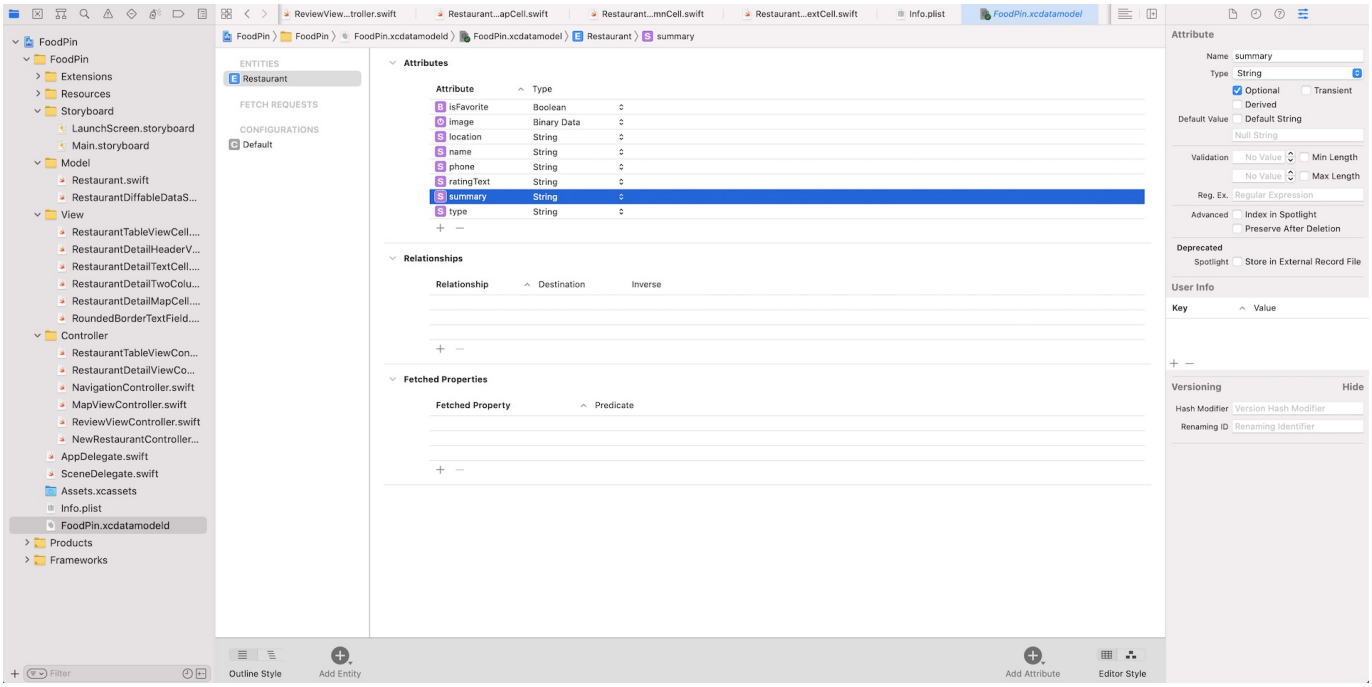


Figure 19-4. Adding attributes to the Restaurant entity

You may wonder why we use the attribute name *summary* instead of *description*. If you've tried to use the name *description*, you will encounter an error saying that the name *description* is reserved. This is why we couldn't use it and had to replace it with *summary*.

The attribute types of *name*, *type*, *location*, *phone*, *isFavorite*, *summary*, and *ratingText* are trivial, but why do we set the attribute type of *image* to Binary Data?

Presently, the restaurant images are bundled in the app and managed by the asset catalog. This is why we can load an image by passing `UIImage` with the image name. When a user creates a new restaurant, the image is loaded from an external source, whether it's from the built-in photo library or taken from a camera. In this case, we can't just store the file name. Instead, we save the actual data of the image into the database. Binary data type is used for this purpose.

You may have another question about the `ratingText` attribute. Why do we name it `ratingText` instead of `rating`? If you refer to the `Restaurant.swift` file again, the `rating` variable is of the type `Rating`, which is an enum. Each case has a default text value (e.g. awesome). This is why we name the attribute `ratingText` to store the text value.

If you select a particular attribute, you can further configure its properties in the Data Model inspector. For example, the *name* attribute is a required attribute. You can uncheck the *Optional* checkbox to make it mandatory. For the FoodPin project, you can configure all attributes (except `rating`) as required.

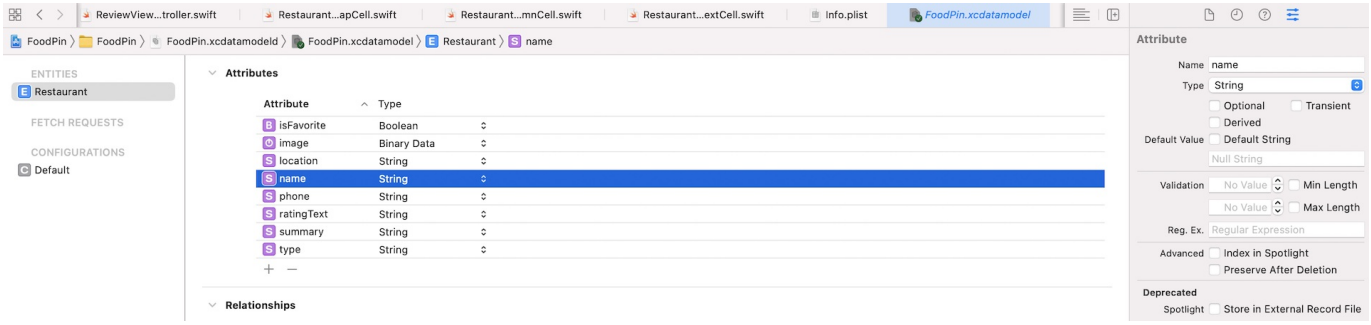


Figure 19-5. Editing attribute properties in the Data Model inspector

Creating Managed Objects

To use Core Data, the very first step is to create the data model, which you've already done. The next step is to create the managed objects. Model objects that tie into in the Core Data framework are known as *managed objects,* which are at the heart of any Core Data application.

For the FoodPin project, you can manually convert the `Restaurant` struct to a managed object class. From Xcode 8 and onwards, however, you can let the development tool do the work for you. By referencing the entity model, Xcode can automatically generate the managed object class for you.

Note: You're probably wondering why we have to create a managed object class. Do you remember the relationship between outlet variables and UI objects in Interface Builder? By updating the value of outlets, we can modify the content of UI objects. Here, the managed objects are very similar to outlet variables. You can modify the entity's content by updating the managed object.

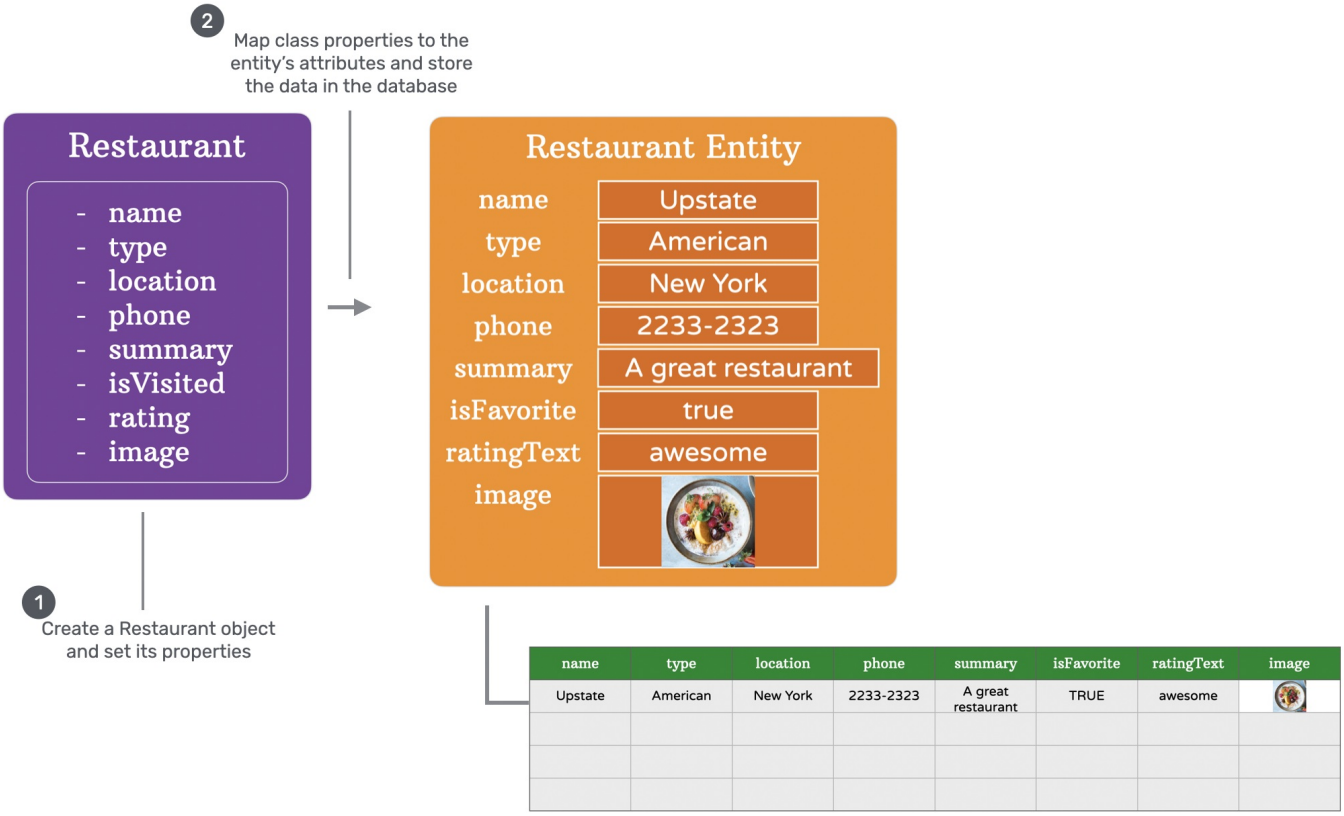


Figure 19-6. Sample relationship between model object and entity

Now select the `Restaurant` entity and go to the Data Model inspector. You should see the Class section. The default *Codegen* option is set to Class Definition. This means Xcode will generate the model class for you when you build the project. If you choose this option, you won't be able to find or modify the class file. It is suitable for simple projects that the class doesn't need further modifications



Figure 19-7. Assign the entity with the Restaurant class

For this project, we will use the manual option because we need to customize the model class to handle the restaurant rating. In the *Class* section, set the class name to `Restaurant`, module to `Current Product Module`, and the Codegen option to `Manual/None`.

The name of the class is the same as the `Restaurant` struct. We will convert it to a model class for Core Data. Open `Restaurant.swift` and replace the file content like this:

```
import CoreData

public class Restaurant: NSManagedObject {
    @nonobjc public class func fetchRequest() -> NSFetchRequest<Restaurant
> {
        return NSFetchRequest<Restaurant>(entityName: "Restaurant")
    }

    @NSManaged public var name: String
    @NSManaged public var type: String
    @NSManaged public var location: String
    @NSManaged public var phone: String
    @NSManaged public var summary: String
    @NSManaged public var image: Data
    @NSManaged public var ratingText: String?
    @NSManaged public var isFavorite: Bool

}
```

The model class of Core Data should be inherited from `NSManagedObject`. Each property is annotated with `@NSManaged` and corresponds to the attribute of the Core Data model we created earlier. By using `@NSManaged`, this tells the compiler that the property is taken care by Core Data. In the code, we also create a method called `fetchRequest()`. This will be used to retrieve the restaurant data from database.

If you used the *Class Definition* option of Codegen, this code above will be generated by Xcode. But it is not hard to create it manually.

We haven't finished the implementation yet. In the original `Restaurant` struct, it had a property called `rating` and an enum called `Rating` for holding the restaurant rating. The property should be migrated to the new `Restaurant` class. To do that, we will create an extension like this:

```
extension Restaurant {

    enum Rating: String {
        case awesome
        case good
        case okay
        case bad
        case terrible

        var image: String {
            switch self {
            case .awesome: return "love"
            case .good: return "cool"
            case .okay: return "happy"
            case .bad: return "sad"
            case .terrible: return "angry"
            }
        }
    }

    var rating: Rating? {
        get {
            guard let ratingText = ratingText else {
                return nil
            }

            return Rating(rawValue: ratingText)
        }

        set {
            self.ratingText = newValue?.rawValue
        }
    }
}
```

The `Rating` enum is exactly the same as before. For the `rating` property, it is now a computed property that handles the conversion of the rating text. Up till now, all our code uses the `Rating` enum to process the restaurant rating. Now, due to Core Data, we store the rating as text. We need to have a way to bridge between `rating` and `ratingText`. This is why we created the `rating` property as a computed property. For the getter, we convert the `ratingText` back to an enum. For the setter, we retrieve the raw value of the enum and assign it to `ratingText`.

Using the Managed Object

Now that we've prepared the data model and model class, let's start to convert the existing code to support Core Data. To begin, open the `RestaurantTableViewController.swift` file. We no longer need to initialize the restaurants array with default values as we will pull the data from the database. Therefore, declare the `restaurants` object like this:

```
var restaurants: [Restaurant] = []
```

Once you make the change, you will see quite a couple of errors in `RestaurantTableViewController`. You can review the issues of your project in the Issue navigator.

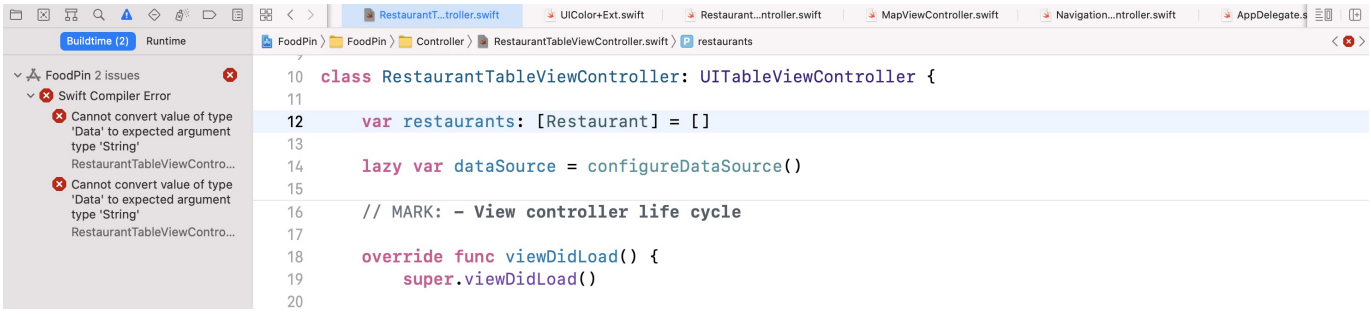


Figure 19-8. Issue Navigation

Basically, both errors are related to the `image` property of the `Restaurant` class. Before the change, the `image` property is of the type `String`. With Core Data, the `image` is now stored as a `Data` object. This is why Xcode issues those errors.

Let's fix them one by one. First, it is the `configureDataSource` method of `RestaurantTableViewController`. Xcode issues an error for the following line of code:

```
cell.thumbnailImageView.image = UIImage(named: restaurant.image)
```

To load the image, instead of passing the image through the `named` parameter, we now initialize the `UIImage` object using the `data` parameter. Replace the line of code with the following:

```
cell.thumbnailImageView.image = UIImage(data: restaurant.image)
```

This will fix the error. The same applies to `imageToShare` in `tableView(_:trailingSwipeActionsConfigurationForRowAt:)`. Replace that line of code with the following:

```
if let imageToShare = UIImage(data: restaurant.image) {
```

Next, let's move onto `RestaurantDetailViewController.swift`. Similarly, you will see some errors related to the change of the `image` property of `Restaurant`. In the `viewDidLoad()` method, initialize the restaurant image using data instead of name:

```
headerView.headerImageView.image = UIImage(data: restaurant.image)
```

Next, it's the `ReviewViewController` class. In the `viewDidLoad()` method, change the `UIImage` initialization using data:

```
backgroundImageView.image = UIImage(data: restaurant.image)
```

After all the modifications, you should be able to run the app and see a blank table. I encourage you to fix the errors on your own. But for reference, you can download the Xcode project from http://www.appcoda.com/resources/swift53/FoodPinCoreDataStarter2.zip.

Handling Empty Table Views

Before we continue to discuss how to work with Core Data, let me sidetrack a bit to talk about empty table views. Now that you should see a blank table when running the app, wouldn't it be great if it can show users some instructions like the screen shown in figure 19-9?



Figure 19-9. A fancy empty table view

The general idea of an empty view is to provide users with more information so that they know where to begin. It is not very difficult to implement an empty view. `UITableView` has a `backgroundView` property that allows developers to assign a view to it. We will make use of the property to display our empty view. Of course, we need to perform some checking and display the empty view only if there is no restaurant data.

I have already designed an image for the empty view. You can download it from http://www.appcoda.com/resources/swift53/emptydata.zip. Unzip the file and add the image to the asset catalog (`Assets.xcasset`). Make sure you enable the *Preserve Vector Data* option for the image.

Now open `Main.storyboard`. Interface Builder has a feature that allows you to embed an extra view in the existing view controller. First, select the Restaurant Table View Controller to reveal the scene dock. Open the Object library, drag a View object to the scene dock, and put it right next to the exit icon.

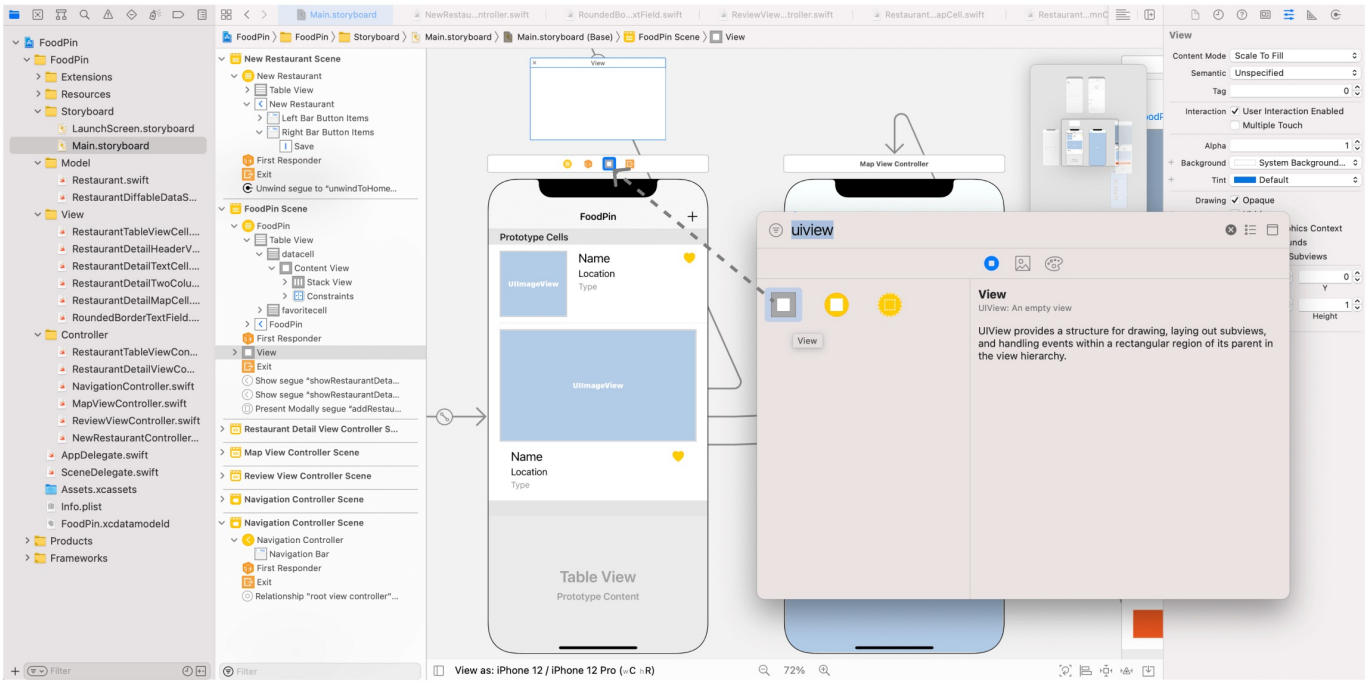


Figure 19-10. Add a view object to the scene dock of the Restaurant Table View Controller

Once you added the view, you will see a view right above the controller. You can now design the view just like any other views. Drag an image view to the view. In the Attributes inspector, set the *Image* option to `emptydata`.

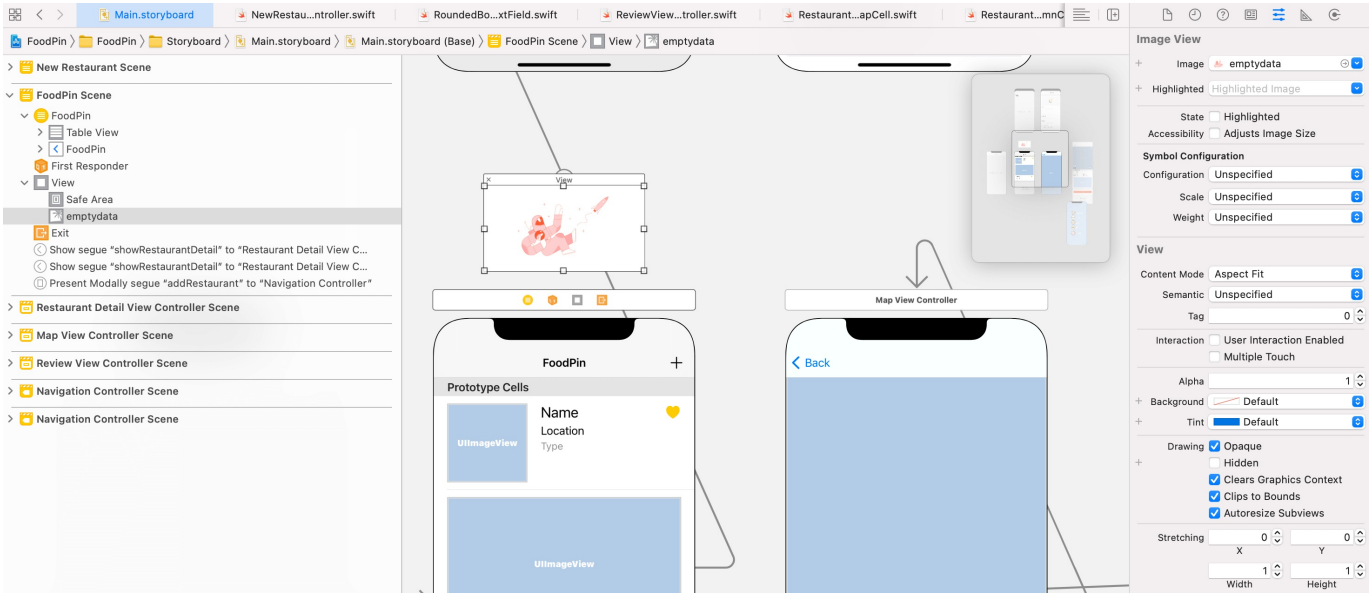


Figure 19-11. Adding an image view to the extra view

This view is an extra view of the table view controller. In order to access it from code, we have to define an outlet to connect with it. Go to `RestaurantTableViewController.swift` and declare an outlet called `emptyRestaurantView` in the class:

```
@IBOutlet var emptyRestaurantView: UIView!
```

Now go back to `Main.storyboard`. Right-click Food Pin controller and connect `emptyRestaurantView` with the extra view.

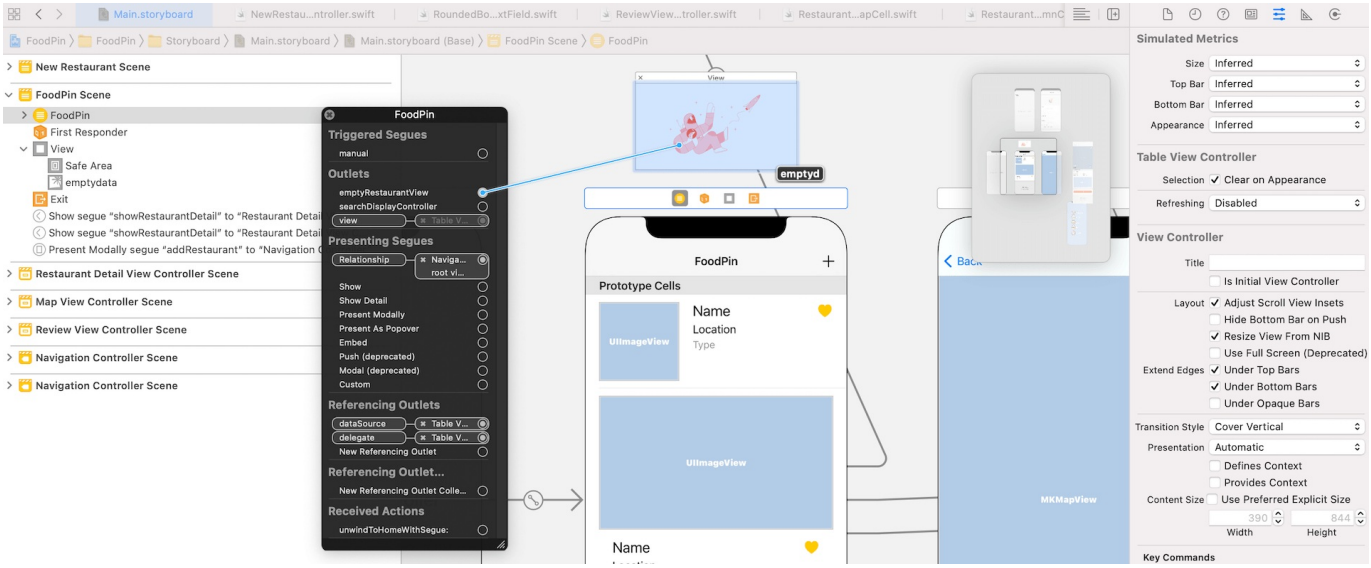


Figure 19-12. Defining the layout constraints

You've already prepared the design of the empty view. It is time to write some code to display it. Our implementation will be like this:

- Add the empty view to the background view of the table view.
- By default, we set to hide the background view.
- When the table view is about to display, we check to see if there are any restaurant records. If there is no record, we set to unhide the background view so as to display the empty view.

In the `viewDidLoad()` method of the `RestaurantTableViewController` class, insert the following code:

```
// Prepare the empty view
tableView.backgroundView = emptyRestaurantView
tableView.backgroundView?.isHidden = restaurants.count == 0 ? false : true
```

Now run the project again. Your app should look like the image shown in figure 19-9.

Working with Managed Objects

Now back to Core Data. Earlier, we converted our FoodPin project to use managed objects, the next question is how can we use the objects to save data into the database?

With the introduction of `NSPersistentContainer`, the complexity involved in setting up the Core Data stack had been dramatically reduced. Common operations, such as inserting a record to a database, are a lot simpler. To save a restaurant, you basically need to handle a couple of things:

1. Create a `Restaurant` object with the view context of the persistent container and then set its properties as usual.

```
restaurant = Restaurant(context: appDelegate.persistentContainer.viewConte
xt)
restaurant.name = "Upstate"
restaurant.type = "Cafe"
restaurant.location = "New York"
```

2. Next, you call the `saveContext()` method in `AppDelegate` to save the data to the database:

```
appDelegate.saveContext()
```

That's it. As you can see, Core Data shields you away from the underlying logic of database management. You do not need to understand how to insert a record into the database using SQL. All is done by using the Core Data APIs.

Saving a New Restaurant to the Database

With some basic understanding of managed objects, let's update the `NewRestaurantController` class to save a new restaurant to the database. First, add the following `import` statement at the very beginning of `NewRestaurantController.swift` so that the class can utilize the Core Data framework:

```
import CoreData
```

Declare a restaurant variable in the `NewRestaurantController` class:

```
var restaurant: Restaurant!
```

In the `saveButtonTapped(sender: AnyObject)` method, apply what you've just learned to save the `restaurant` object into the persistent store. Insert the following code before calling the `dismiss` method:

```
if let appDelegate = (UIApplication.shared.delegate as? AppDelegate) {
    restaurant = Restaurant(context: appDelegate.persistentContainer.viewC
ontext)
    restaurant.name = nameTextField.text!
    restaurant.type = typeTextField.text!
    restaurant.location = addressTextField.text!
    restaurant.phone = phoneTextField.text!
    restaurant.summary = descriptionTextView.text
    restaurant.isFavorite = false

    if let imageData = photoImageView.image?.pngData() {
        restaurant.image = imageData
    }

    print("Saving data to context...")
    appDelegate.saveContext()
}
```

The code above is pretty much the same as what we discussed in the previous section. However, there are a few lines of code that are new to you. First, it's the following line of code:

```
if let appDelegate = (UIApplication.shared.delegate as? AppDelegate) {
```

The `persistentContainer` variable is declared in `AppDelegate.swift`. To access the variable, we have to first get a reference to `AppDelegate`. In iOS SDK, you can use `UIApplication.shared.delegate as? AppDelegate` to get the `AppDelegate` object.

Next, it's related to the `image` property.

```
if let imageData = photoImageView.image?.pngData() {
    restaurant.image = imageData
}
```

Recalled that the image type of the Restaurant Entity is set to Binary Data, in this case, the image property of the generated `RestaurantMO` has a type of `Data`.

Therefore, when we set the value of the `image` property, we have to retrieve the data of the selected image and convert it to a `Data` object. The UIKit framework provides a set of built-in functions for graphics operations. The `pngData()` function allows us to get the data of a specified image in PNG format. We then create the `Data` object using the image data.

This is the piece of code you need to add a new restaurant to the database. If you run the app now and save a new restaurant, the app should be able to save the record into the database without any errors. You should see the message "Saving data to context ..." in the console. However, your app is not ready to display the restaurant just added. That's what we're going to do next.

Fetching Data Using Core Data

The `Restaurant` class has the `fetchRequest()` method. When called, it returns you an `NSFetchRequest` object that specifies the search criteria and which entity to search (here, it is the `Restaurant` entity). Together with another API called `NSFetchedResultsController`, you will be able to fetch records from the database efficiently.

`NSFetchedResultsController` is specially designed for managing the results returned from a Core Data fetch request and providing data for a table view. It monitors changes to objects in the managed object context and reports changes in the result set to its delegate.

Let's see how to use `NSFetchedResultsController` and the diffable data source to retrieve the restaurants and update the data in the table view.

In `RestaurantTableViewController.swift`, import the CoreData framework:

```
import CoreData
```

Next, declare an instance variable for the fetched results controller:

```
var fetchResultController: NSFetchedResultsController<Restaurant>!
```

To better organize our code, we will handle the retrieval of data item in a separate method. Create a new method called `fetchRestaurantData()` like this:

```
func fetchRestaurantData() {
    // Fetch data from data store
    let fetchRequest: NSFetchRequest<Restaurant> = Restaurant.fetchRequest
()
    let sortDescriptor = NSSortDescriptor(key: "name", ascending: true)
    fetchRequest.sortDescriptors = [sortDescriptor]

    if let appDelegate = (UIApplication.shared.delegate as? AppDelegate) {
        let context = appDelegate.persistentContainer.viewContext
        fetchResultController = NSFetchedResultsController(fetchRequest: f
etchRequest, managedObjectContext: context, sectionNameKeyPath: nil, cache
Name: nil)
        fetchResultController.delegate = self

        do {
            try fetchResultController.performFetch()
            updateSnapshot()
        } catch {
            print(error)
        }
    }
}
```

We first get the `NSFetchRequest` object from `Restaurant` and specify the sort order using an `NSSortDescriptor` object. `NSSortDescriptor` lets you describe how the fetched objects are sorted. Here we specify that the `Restaurant` objects should be sorted in ascending order using the *name* key.

After creating the fetch request, we initialize `fetchResultController` and specify its delegate for monitoring data changes. Lastly, we call the `performFetch()` method to execute the fetch request. When complete, we call the `updateSnapshot()` method to update the restaurants in the table view.

We haven't created the `updateSnapshot()` method yet. Insert the following code to implement the method:

```
func updateSnapshot(animatingChange: Bool = false) {

    if let fetchedObjects = fetchResultController.fetchedObjects {
        restaurants = fetchedObjects
    }

    // Create a snapshot and populate the data
    var snapshot = NSDiffableDataSourceSnapshot<Section, Restaurant>()
    snapshot.appendSections([.all])
    snapshot.appendItems(restaurants, toSection: .all)

    dataSource.apply(snapshot, animatingDifferences: animatingChange)

    tableView.backgroundView?.isHidden = restaurants.count == 0 ? false :
true
}
```

The method takes in a boolean parameter called `animatingChanges`. By default, its value is set to `false`. This parameter gives the caller a flexibilty to specify if it needs animations. If it's set to true, the diffable data source will animate the data update.

After executing the fetch request, all the fetched objects are stored in the `fetchedObjects` property of the fetch result controller. In the method, the very first thing we check is whether the controller contains any fetched objects. If the result is positive, we create a snapshot and update the UI accordingly. The code is exactly the same as we discussed before when implementing the table view.

The delegate of `fetchResultController` is set to `self`, that is `RestaurantTableViewController`. In this case, the class should adopt the `NSFetchedResultsControllerDelegate` protocol.

The protocol provides methods to notify its delegate whenever there is any change in the controller's fetch results. Later, we'll implement the methods. For now, simply create an extension to adopt the protocol like this:

```
extension RestaurantTableViewController: NSFetchedResultsControllerDelegate
 {

}
```

With the `fetchRestaurantData()` method ready, invoke it in the `viewDidLoad()` method:

```
fetchData()
```

And, since we created the `updateSnap()` method, we can delete the following lines of code:

```
var snapshot = NSDiffableDataSourceSnapshot<Section, Restaurant>()
snapshot.appendSections([.all])
snapshot.appendItems(restaurants, toSection: .all)

dataSource.apply(snapshot, animatingDifferences: false)
```

If you compile and run the app now, it should display the restaurants that you previously added. However, if you try to add another new restaurant, the table does not refresh with the new record.

There is still something left.

When there is any content change due to add/remove/move/update, the following methods of the `NSFetchedResultsControllerDelegate` protocol will be called:

- controllerWillChangeContent(_:)
- controllerDidChangeContent(_:)

We will implement the `controllerDidChangeContent(_:)` method and refresh the UI accordingly. Update the extension like this:

```
extension RestaurantTableViewController: NSFetchedResultsControllerDelegate
 {

    func controllerDidChangeContent(_ controller: NSFetchedResultsControll
er<NSFetchRequestResult>) {
        updateSnapshot()
    }

}
```

When there is any content change in the managed object context (e.g. a new restaurant is saved), the method is automatically called. Here we call the `updateSnapshot()` method to refresh the UI with the newly-added (or deleted) item.

That's it. Now run the app again and create a few restaurants. The app should respond to the change instantaneously.



Figure 19-13. Adding a new restaurant (left), The newly-added restaurant now displays in the table view

Deleting Data Using Core Data

To delete a record from the persistent data store, you just need to call a method named `delete` with the managed object to delete. Then, you call the `saveContext` method to apply the changes. Here is a sample code snippet:

```
let context = appDelegate.persistentContainer.viewContext
context.delete(objectToDelete)
appDelegate.saveContext()
```

To remove the selected restaurant from database, you have to update the `deleteAction` variable in the `tableView(_:trailingSwipeActionsConfigurationForRowAt:_)` method like this:

```
let deleteAction = UIContextualAction(style: .destructive, title: "Delete"
) { (action, sourceView, completionHandler) in

    if let appDelegate = (UIApplication.shared.delegate as? AppDelegate) {
        let context = appDelegate.persistentContainer.viewContext

        // Delete the item
        context.delete(restaurant)
        appDelegate.saveContext()

        // Update the view
        self.updateSnapshot(animatingChange: true)
    }

    // Call completion handler to dismiss the action button
    completionHandler(true)
}
```

In the above code, we call the `delete` method to delete the item. Finally, we call the `saveContext()` method to apply the changes.

Now compile and run the app again. At this point, if you delete a record it should remove completely from the database.

Exercise #1

We haven't finished this chapter yet. But I have an exercise for you. Did you notice that there is a bug for the description field of the restaurant detail screen? It now shows a weird message.

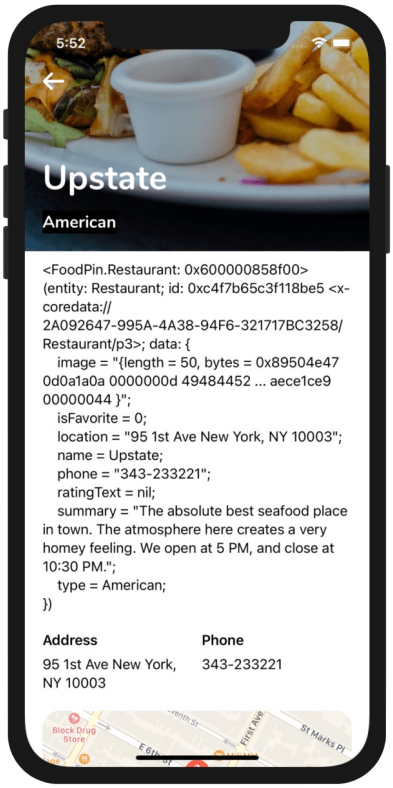


Figure 19-14. The description label now shows a weird message

Try to find out the root cause and fix the error. I can give you a hint. Look into the `RestaurantDetailViewController` class.

Updating a Managed Object

What if we need to update the rating of an existing restaurant? How can we update the record in the database?

Similar to creating a new restaurant, you can update a restaurant record in the persistent store by updating the corresponding managed object and then call `saveContext()` to apply the changes.

For example, to save the rating of a restaurant, you can update the `rateRestaurant` method of the `RestaurantDetailViewController` class like this:

```
@IBAction func rateRestaurant(segue: UIStoryboardSegue) {

    dismiss(animated: true, completion: {
        if let rating = segue.identifier {
            self.restaurant.rating = rating
            self.headerView.ratingImageView.image = UIImage(named: rating)

            if let appDelegate = (UIApplication.shared.delegate as? AppDel
egate) {
                appDelegate.saveContext()
            }

            let scaleTransform = CGAffineTransform.init(scaleX: 0.1, y: 0.1
)
            self.headerView.ratingImageView.transform = scaleTransform
            self.headerView.ratingImageView.alpha = 0

            UIView.animate(withDuration: 0.4, delay: 0, usingSpringWithDam
ping: 0.3, initialSpringVelocity: 0.7, options: [], animations: {
                self.headerView.ratingImageView.transform = .identity
                self.headerView.ratingImageView.alpha = 1
            }, completion: nil)
        }
    })
}
```

The code is exactly the same as before, except that we add a couple lines of code to call up `appDelegate.saveContext()` to save the changes to the database.

The current version of restaurant detail view controller doesn't display the restaurant's rating when it is first loaded. Insert the following code in the `viewDidLoad()` method of `RestaurantDetailViewController` to fix that:

```
if let rating = restaurant.rating {
    headerView.ratingImageView.image = UIImage(named: rating.image)
}
```

Cool! You are now ready to test. Choose a restaurant and rate it. Now the rating is saved permanently in the database.

Exercise #2

The "Save as favorite" button in the detailed view doesn't work yet. In this exercise, your task is to make it functional and save its state in the database using Core Data. On top of that, move the heart icon in the detailed view to the navigation bar. Figure 19-15 illustrates the layout changes.

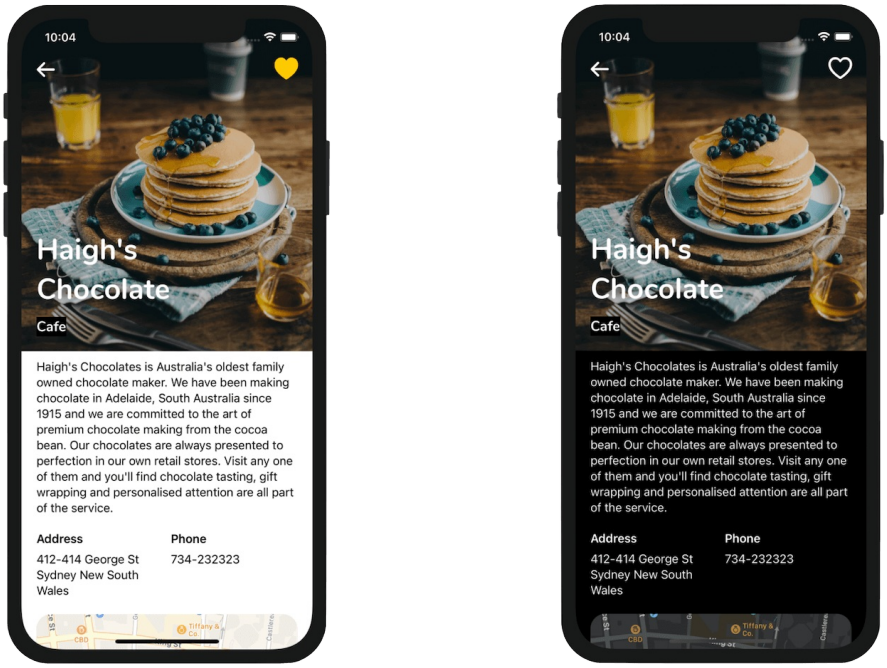


Figure 19-15. Change the layout of the detailed view by moving the favorite button to the navigation bar

Summary

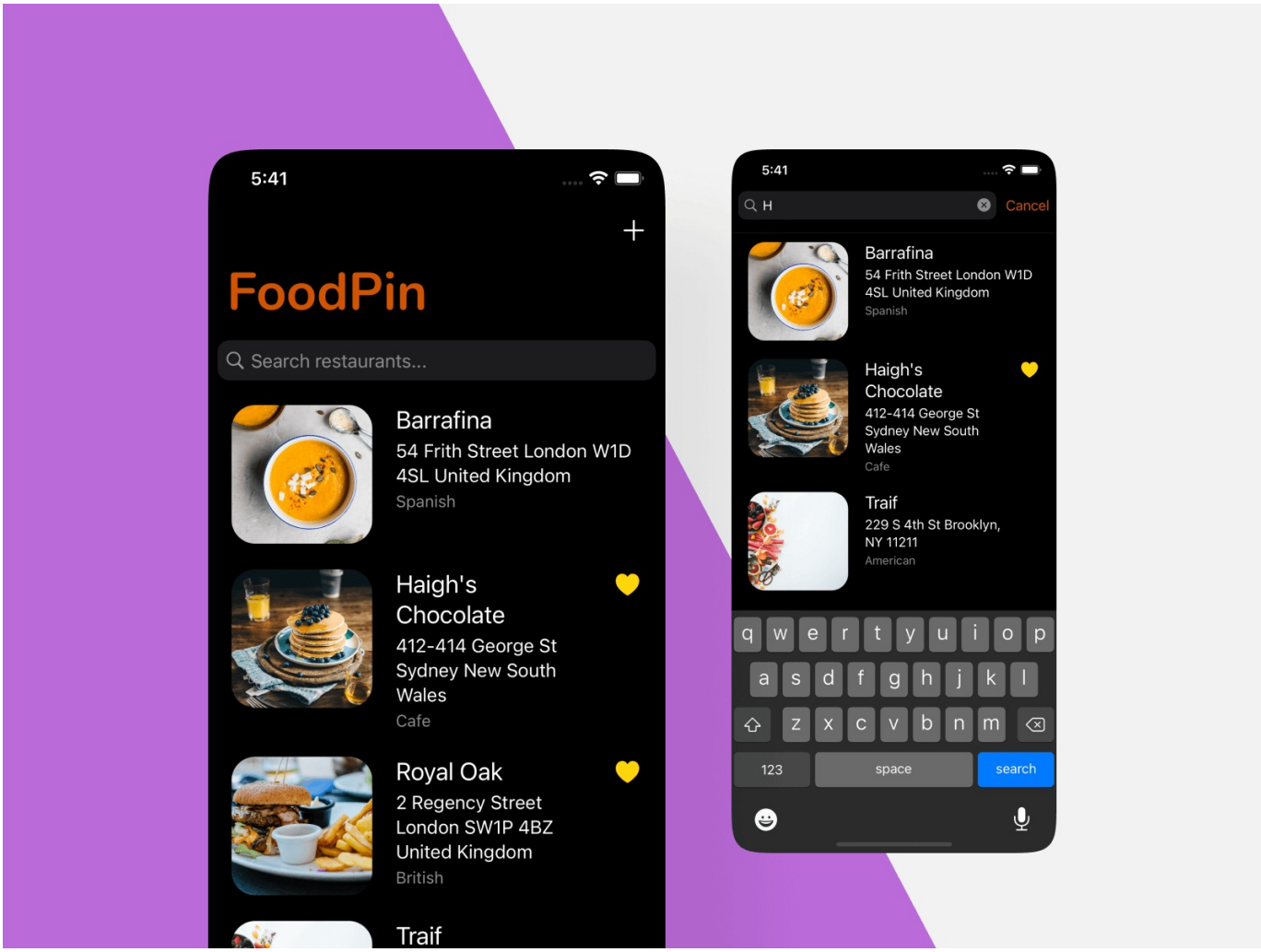
Congratulate yourself on making an app using Core Data. At this point, you should know how to retrieve and manage data in a persistent data store. Core Data is a powerful framework for working with persistent data especially for those who do not have any database knowledge. The `NSPersistentContainer` class encapsulates the whole Core Data stack and makes working with Core Data a breeze, especially for beginners.

This chapter gives you a brief overview of Core Data. I hope you understand the basics of Core Data, and how to use it to store persistent data. Though we will not further discuss Core Data in this book, don't stop learning and exploring. You can check out Apple's official reference (https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/CoreData/index.html) to learn more.

For reference, you can download the complete Xcode from http://www.appcoda.com/resources/swift53/FoodPinCoreDataFinal.zip.

Are you ready to further improve the app? I hope you're still with me. Let's move on and take a look at how to add a search bar.

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 `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.

Starting from iOS 11, Apple further simplified the implementation of the search bar. It introduced 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.

Using UISearchController

In general, to add a search bar in a navigation bar, it essentially comes down to the following lines of code:

```
let searchController = UISearchController(searchResultsController: nil)
searchController.searchResultsUpdater = self
self.navigationItem.searchController = searchController
```

The first line of code creates an instance of `UISearchController`. If you pass a `nil` value, that means the search results would be displayed in the same view that you're searching. Optionally, you can specify another view controller for displaying the search result.

You may wonder when you need to define another view controller. Take the FoodPin app as an example. If `nil` is used, the search results will be displayed in the table view. Figure 20-1 shows the format of search results. As you can see, the display style is exactly the same as that of the table view. If you want to display the search results in a different format, you will need to create another view controller and specify it during the initialization of `UISearchController`.

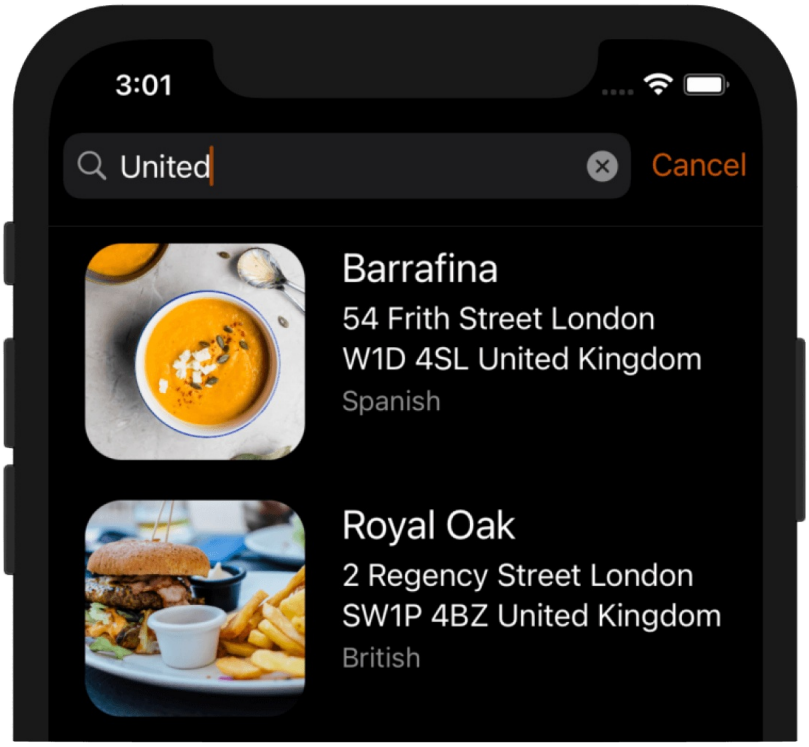


Figure 20-1. A sample search result

The second line of code tells the search controller which object is responsible for updating the search result. It can be any object in your application or simply the current one.

The last line of code adds the search bar to the navigation bar.

Adding a Search Bar

Now let's try to add a search bar in the FoodPin app. Open `RestaurantTableViewController.swift`, declare the `searchController` variable:

```
var searchController: UISearchController!
```

Then add the following lines of code in the `viewDidLoad` method:

```
searchController = UISearchController(searchResultsController: nil)
self.navigationItem.searchController = searchController
```

As I have explained the code before, I will not go over it again. But as you can see, you can add a default search bar with just two lines of code. If you compile and run the app now, you should find a search bar below the navigation bar by dragging down the table view. However, it doesn't work yet because we haven't implemented the search logic.

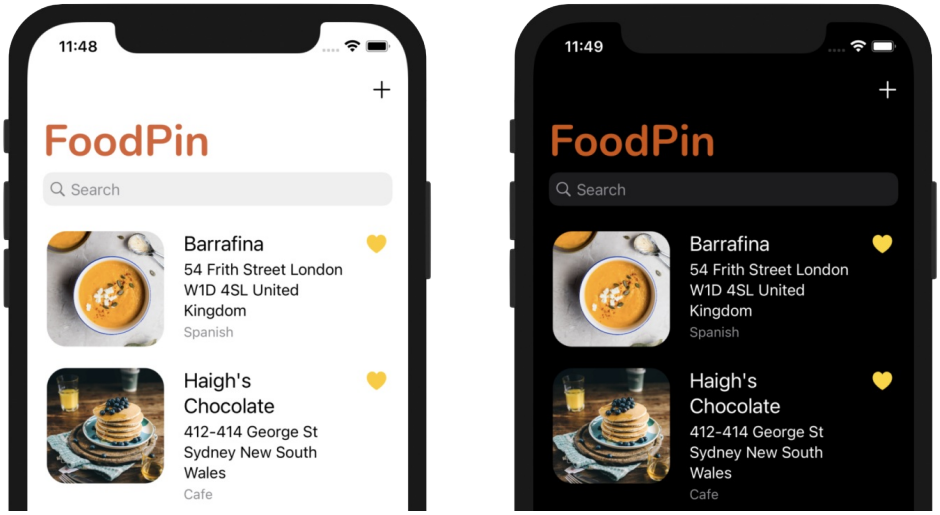


Figure 20-2. Search bar added in the navigation bar

Filtering Content

The search controller doesn't provide any default functions to filter your data. It's your responsibility to develop your own implementation to filter the content. For the FoodPin app, it allows users to do a search against the name of a restaurant. You have two ways to filter the content and search for the results. First, you may compute the search result by filtering the existing `restaurants` array. In Swift, there is a built-in method called `filter` for filtering an existing array. You use `filter` to loop over a collection and return a new array containing those items that match the specified condition. For example, the new array can only contain restaurants with the name starts with "up".

The `filter` method takes in a block of code in which you provide the filtering rule. For those elements to be included, you indicate with a return value of `true`. Otherwise, `false` is returned and that element will be excluded. Here is a sample

```
let searchResults = restaurants.filter({ (restaurant) -> Bool in
    let isMatch = restaurant.name.localizedCaseInsensitiveContains(searchT
ext)
    return isMatch
})
```

In the code above, we use the `localizedCaseInsensitiveContains` method to see if the restaurant name contains the search text, regardless of the case of the string (i.e. case-insensitive). If the search text is found, the method returns `true`, indicating the restaurant name should be included in the new array. Otherwise, a `false` is returned to exclude the item.

An alternate approach is to execute the search query using Core Data. The `NSFetchRequest` class allows developers to specify a predicate, which specifies which properties to filter by and the constraints on the selection of data. For example, if you want to fetch those restaurants with the name containing the term "cafe", you can assign the fetch request object a predicate like this:

```
fetchRequest.predicate = NSPredicate(format: "name CONTAINS[c] %@", "cafe"
)
```

The operator CONTAINS[c] will match all the restaurants that its name contains the word "cafe". The [c] part means it's a case insensitive search.

For this demo, we will use the second approach to implement the search feature. Therefore, update the `fetchRestaurantData()` method like this:

```
func fetchRestaurantData(searchText: String = "") {
    // Fetch data from data store
    let fetchRequest: NSFetchRequest<Restaurant> = Restaurant.fetchRequest
()

    if !searchText.isEmpty {
        fetchRequest.predicate = NSPredicate(format: "name CONTAINS[c] %@"
, searchText)
    }

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

    if let appDelegate = (UIApplication.shared.delegate as? AppDelegate) {
        let context = appDelegate.persistentContainer.viewContext
        fetchResultController = NSFetchedResultsController(fetchRequest: f
etchRequest, managedObjectContext: context, sectionNameKeyPath: nil, cache
Name: nil)
        fetchResultController.delegate = self

        do {
            try fetchResultController.performFetch()
            updateSnapshot(animatingChange: searchText.isEmpty ? false : t
rue)
        } catch {
            print(error)
        }
    }
}
```

We modify the method to accept a parameter called `searchText` with the default value set to blank. In this case, if the caller of the method doesn't specify any search term. This method will fetch all restaurants from database. The code is nearly the same as before but with a couple of lines of modifications. When the `searchText` is not empty, we assign a `NSPredicate` object to the fetch request. This is how we only filter those restaurants that match the search criteria.

Using Predicate for Searching the Results

Now that we have implemented the search logic, the question is how can we know the user is entering a search text? The trick is to adopt the `UISearchResultsUpdating` protocol. This protocol defines a method called `updateSearchResults(for:)`. When a user selects the search bar or key in a search keyword, the method will be called.

By implementing the method, we can call the `fetchRestaurantData(searchText:)` method with the corresponding search term. Again, we will use an extension to adopt the protocol. Insert the following code snippet in the `RestaurantTableViewController.swift` file:

```
extension RestaurantTableViewController: UISearchResultsUpdating {
    func updateSearchResults(for searchController: UISearchController) {

        guard let searchText = searchController.searchBar.text else {
            return
        }

        fetchRestaurantData(searchText: searchText)
    }
}
```

The code is very straightforward - we get the search text as entered by the user and pass it to the `fetchRestaurantData(searchText:)` method. The method will trigger another fetch request and display the filtered content accordingly.

You're almost done. The very last thing is to add the following lines of code in the `viewDidLoad` method:

```
searchController.searchResultsUpdater = self
searchController.obscuresBackgroundDuringPresentation = false
```

The first line assigns the current class as the search results updater. The `obscuresBackgroundDuringPresentation` property controls whether the underlying content is dimmed during a search. Because we are presenting the search results in the same view, the property should be set to `false`.

Cool! You're ready to fire up your app and test out the search function. What's great is that you can navigate to the restaurant details by tapping the search results. Everything in the original table view controller is reused.

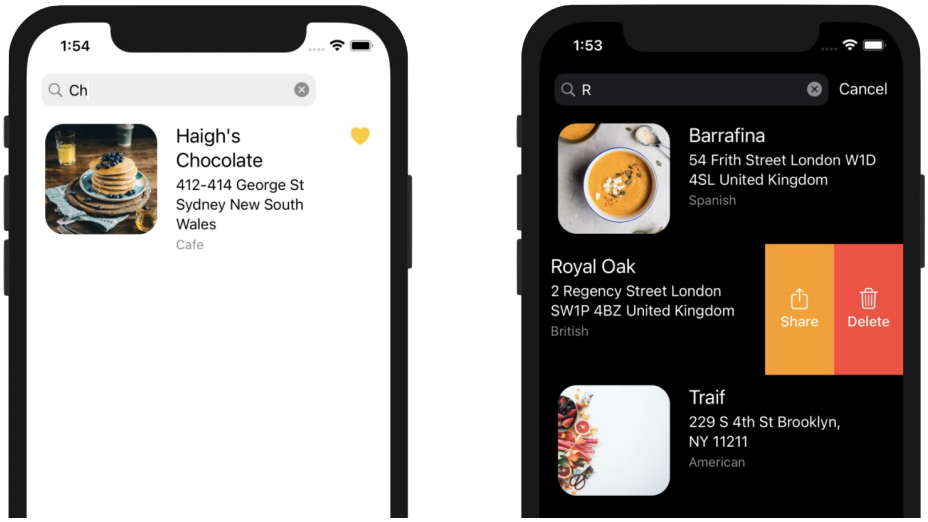


Figure 20-3. Search results

As you may aware, the table cell inherits the share and delete buttons. This is something that you probably don't want to show the buttons in the search results. To disable them, you can insert the following lines of code in `tableView(trailingSwipeActionsConfigurationForRowAt:)` method:

```
override func tableView(_ tableView: UITableView, trailingSwipeActionsConf
igurationForRowAt indexPath: IndexPath) -> UISwipeActionsConfiguration? {

    if searchController.isActive {
        return UISwipeActionsConfiguration()
    }

    .
    .
    .

}
```

The `searchController` object has a property called `isActive` that indicates whether the user is using the search bar. We simply return an empty configuration if the search controller is active.

Search Bar in the Table Header View

It is not a mandate to place the search bar in the navigation bar. Alternatively, you can place it in the table view header. To do that, you can change the following line of code in the `viewDidLoad()` method of `RestaurantTableViewController`:

```
self.navigationItem.searchController = searchController
```

To:

```
tableView.tableHeaderView = searchController.searchBar
```

This will place the search bar in the table's header view. Run the app again and you will see the search bar.

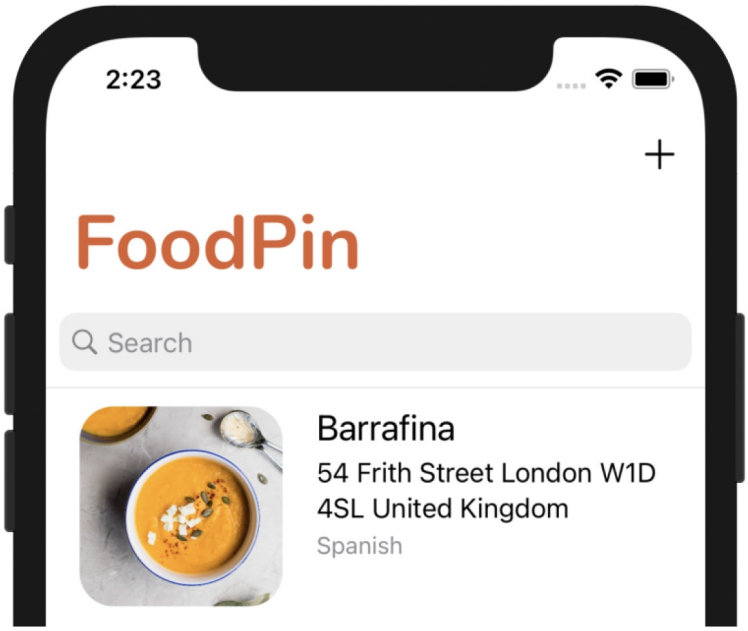


Figure 20-4. A search bar in the table header view

Customizing the Appearance of the Search Bar

`UISearchBar` provides several options for customizing the appearance of a search bar. You can access its properties by using the following line of code:

```
searchController.searchBar.tintColor
```

Here are some of the common properties for the customization.

- **placeholder** - you can use the `placeholder` property to set the default text when there is no other text in the text field.
- **prompt** - the `prompt` property allows you to display a single line of text at the top of the search bar.
- **barTintColor** - set the background color of the search bar.
- **tintColor** - set the tint color of the key elements in the search bar. For

example, you can use the property to change the color of the Cancel button in the search bar.

- **searchBarStyle** - specify the search bar's style. By default, it is set to `.prominent`. When this style is set, the search bar has a translucent background, and the search field is opaque. Alternatively, you can change it to `.minimal` to remove the background and make the search field translucent.

As an example, you can insert the following lines of code at the end of the `viewDidLoad` method of the `RestaurantTableViewController` class:

```
searchController.searchBar.placeholder = "Search restaurants..."
searchController.searchBar.backgroundImage = UIImage()
searchController.searchBar.tintColor = UIColor(named: "NavigationBarTitle"
)
```

Figure 20-5 shows the custom search bar after customization. You can further refer to the official documentation (https://developer.apple.com/documentation/uikit/uisearchbar) for the full set of customizable properties.

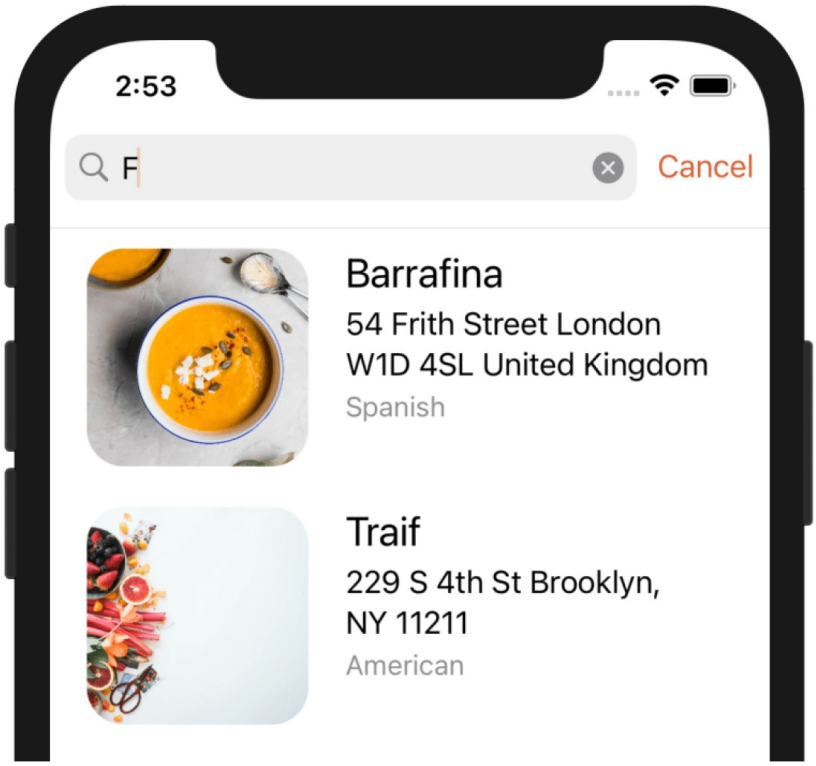


Figure 20-5. A sample search bar after customization

Exercise #1

Now the app only allows users to search restaurants by name. Your exercise is to enhance the search feature so that it supports location search too. For example, if your user keys in *Sydney* in the search field, the app searches through the restaurant list and shows you the restaurants that are either located in Sydney or have "Sydney" in the restaurant name.

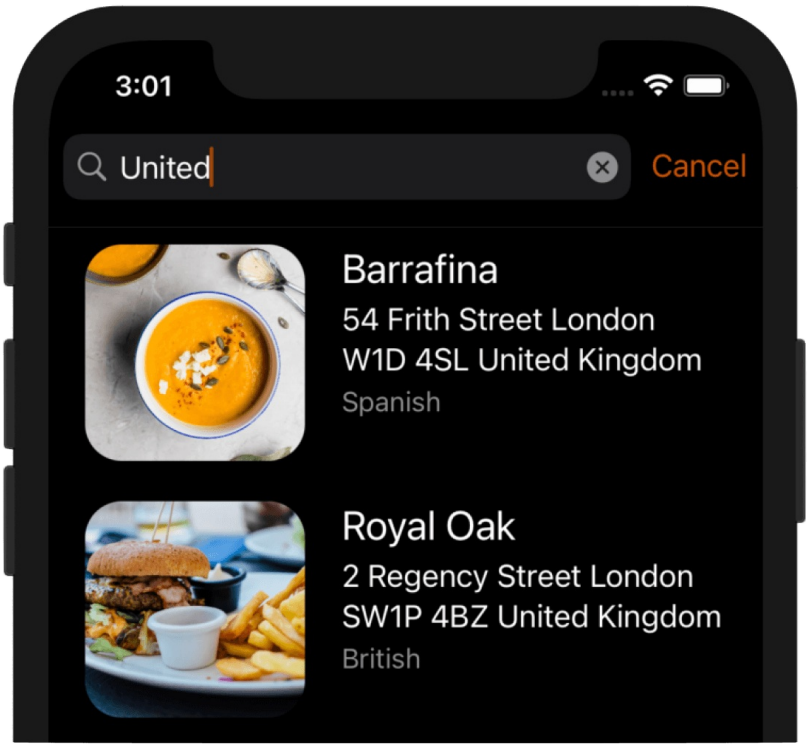


Figure 20-6. The app now supports location search

Summary

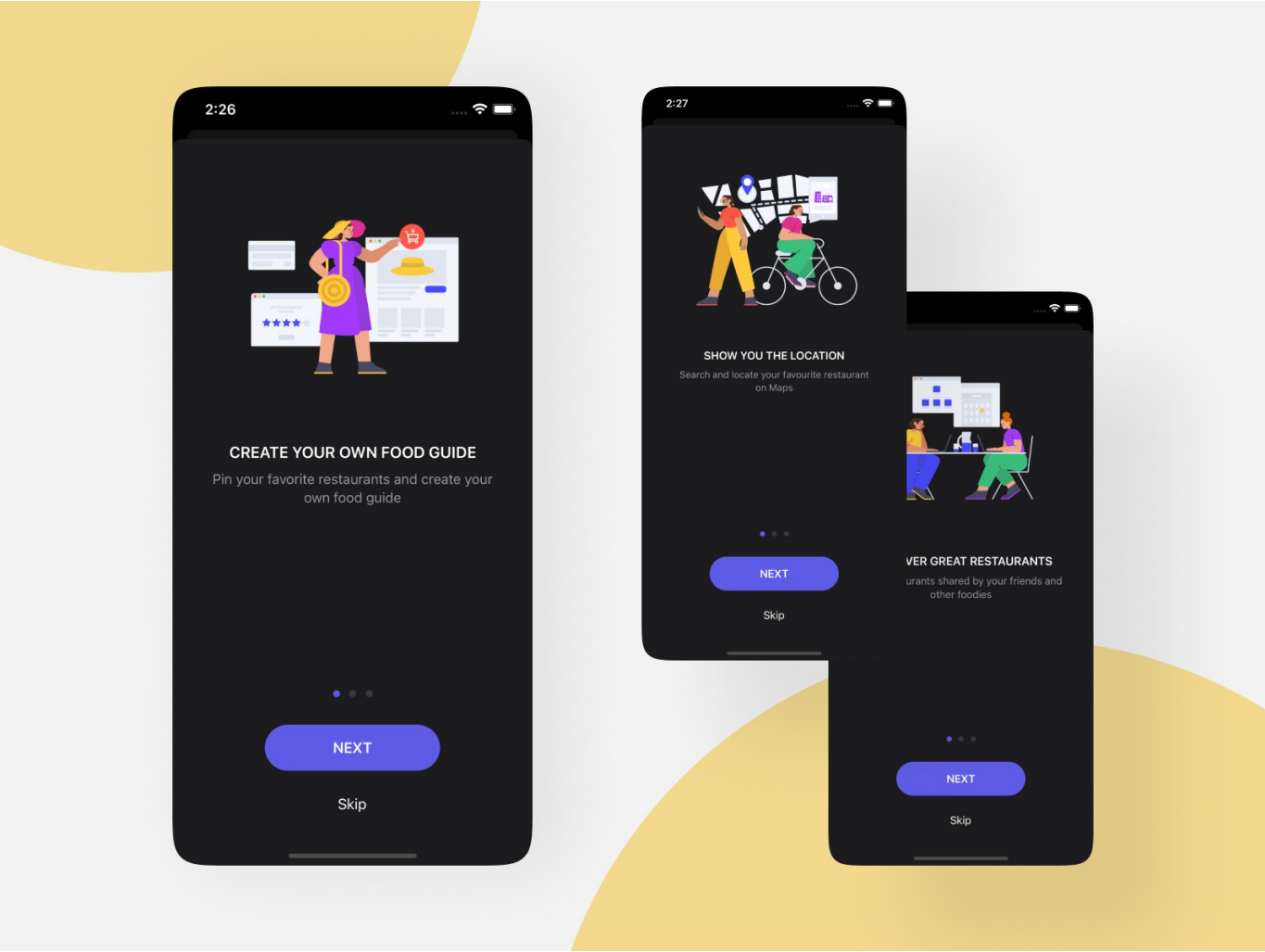
By now, you should know how to implement a search bar in an iOS app. We've made the FoodPin app even better by enhancing it with a search feature; this search feature is particularly important when you have a large amount of information to display. If you still don't fully understand the search bar feature, go back and work through this chapter the second time before moving onto the next one.

In this chapter, we only covered the basics of `UISearchController`. However, there are numerous other ways to use a search bar in your app. For instance, you may want to display the search bar only when the user taps a search button. To explore new ways of search bar integration, I encourage you to check out the UICatalog

demo from Apple (https://developer.apple.com/library/prerelease/ios/samplecode/UICatalog/Introduction/Intro.html).

For reference, you can download the complete Xcode project from http://www.appcoda.com/resources/swift53/FoodPinSearch.zip. The solution of the exercise is included.

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*.

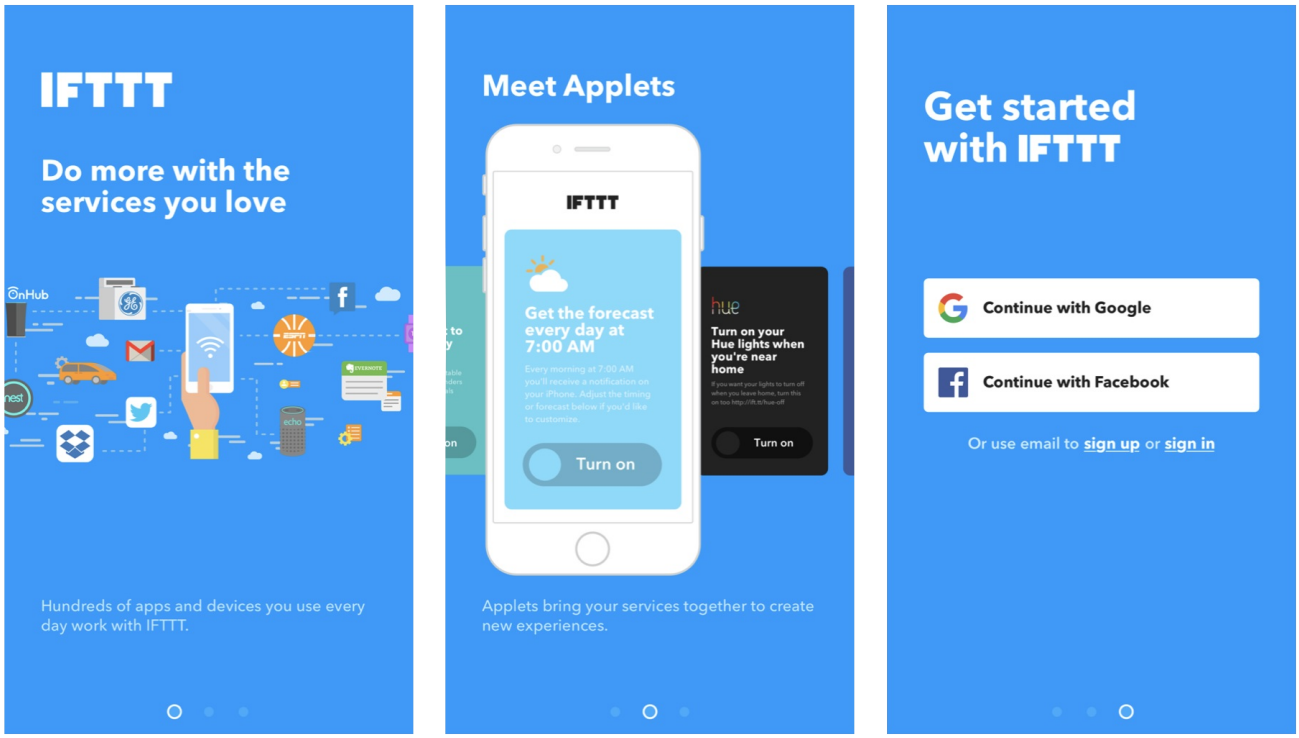


Figure 21-1. Sample Walkthrough Screens of IFTTT

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.

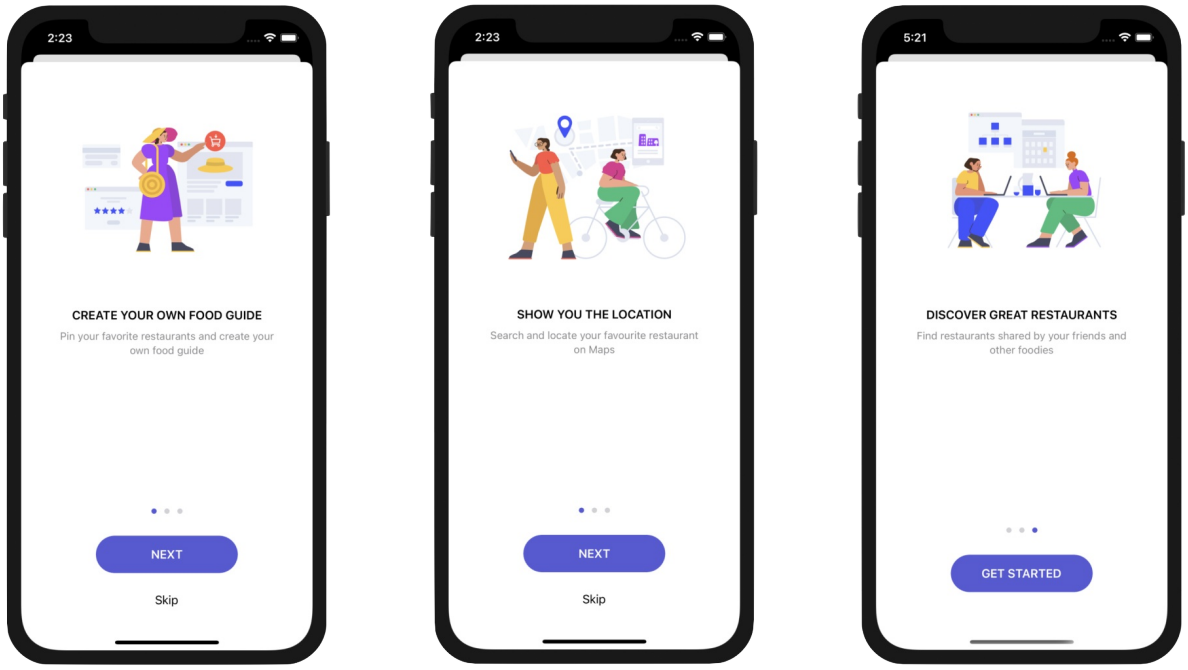


Figure 21-2. Walkthrough screens for the FoodPin app

Creating a New Storyboard for UIPageViewController

So far we only use `Main.storyboard` to design the app UI. Xcode doesn't limit you to have a single storyboard. You are free to create separate storyboards to lay out your app components. As your app becomes more complex, chances are that you will create more than one storyboard to better organize your app UI.

If you haven't completed the previous exercise, you can download the current version of the FoodPin project from http://www.appcoda.com/resources/swift53/FoodPinSearch.zip.

In the project navigator, right-click the `Storyboard` folder and choose *New File...*. Choose the *Storyboard* template and click *Next*. Name the file `Onboarding`.

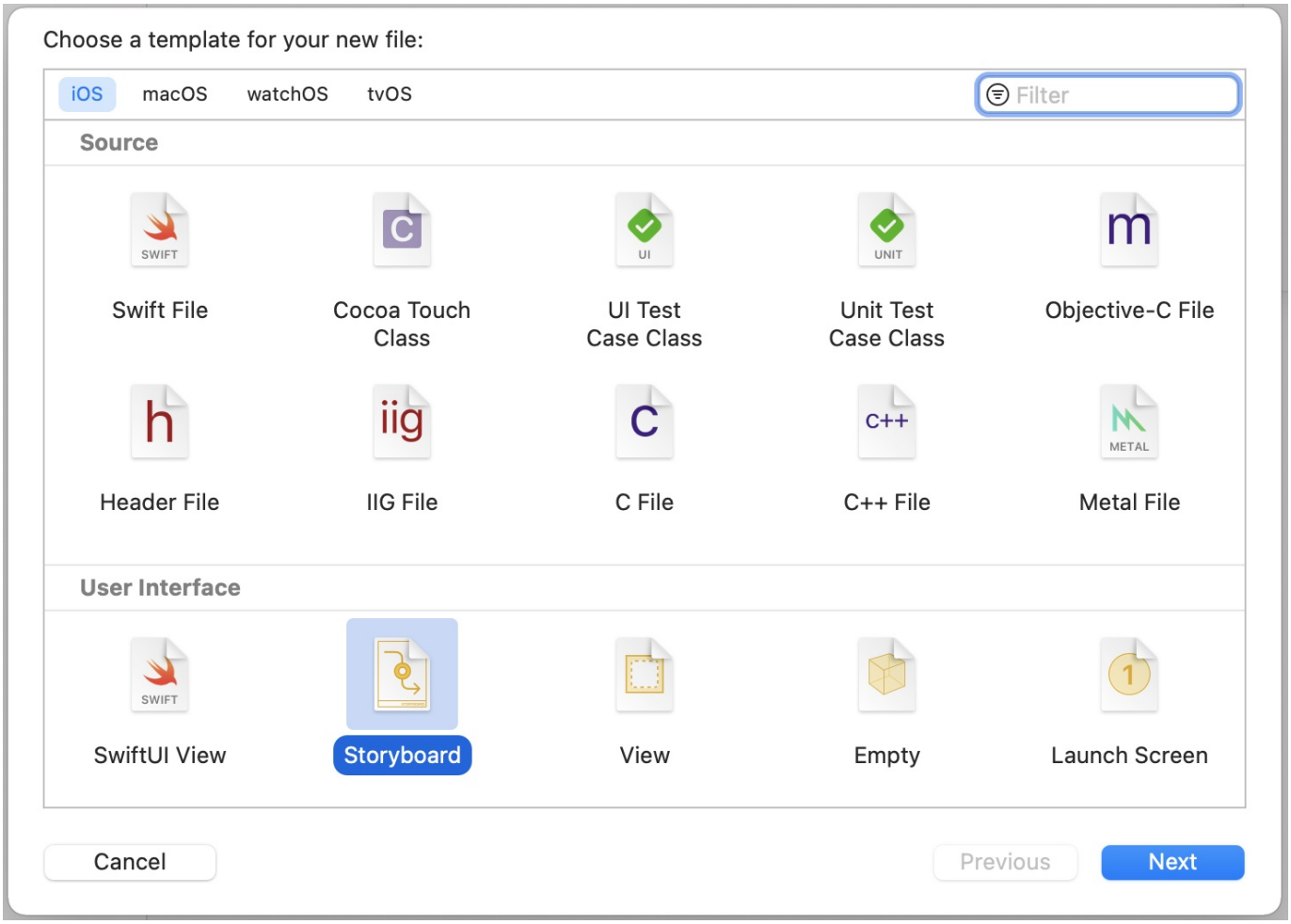


Figure 21-3. Choose the Storyboard template

Now jump to `Onboarding.storyboard`. Open the Object library and drag a page view controller to the storyboard. As you can see, there are multiple options for configuring the behavior of the controller in the Attributes inspector. You can

customize the navigation style (horizontal/vertical), transition style, page spacing, and spine location.

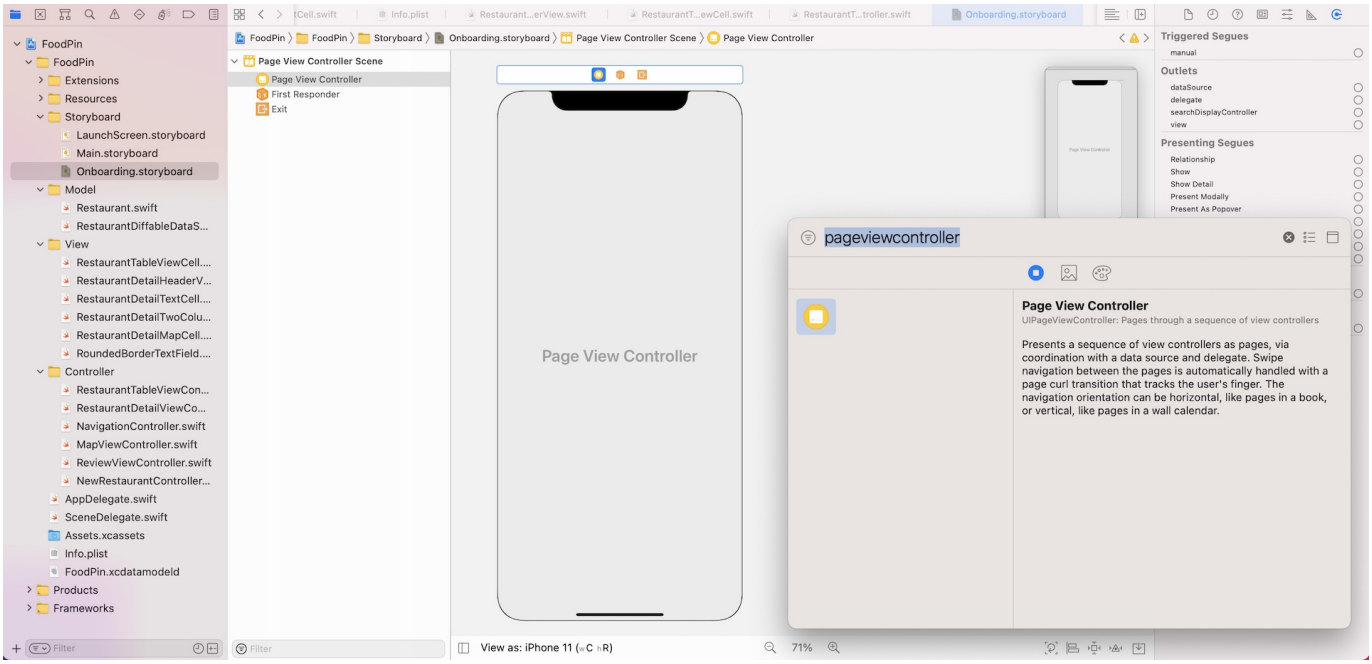


Figure 21-4. Adding a UIPageViewController to the storyboard

By default, the transition style of the page view controller is set as `Page Curl`. This style is perfect for book apps. For walkthrough screens, we prefer to use the scrolling style. In the Attributes inspector, change the transition style to `Scroll`.

Next, assign a storyboard ID to the page view controller. In the Identity inspector, set the storyboard ID of the page view controller to `WalkthroughController`. Later, we will use the ID for creating the controller programmatically. This storyboard ID is optional. But if you need to use it in the code, you have to give the controller a storyboard ID.

Understanding Page View Controllers and Container Views

For now, you can't layout the walkthrough screens directly on the page view controller. Before I walk you through the procedures, let's see how the page view controller works and how we are going to implement the walkthrough screens.

If you refer to figure 21-2 again, you may find that the walkthrough screen can actually be broken down into two main parts. There are two ways to navigate through the tutorial screens. The user can either tap the *Next* button or swipe left the upper part of the screen. Say, you tap the *Next* button, what happens is that the current view of the upper part of the screen starts to move off the screen. At the same time, the next walkthrough screen comes in and eventually replaces the current view.

The lower part of the screen is almost static, that only changes its page indicator to reflect the current page number.

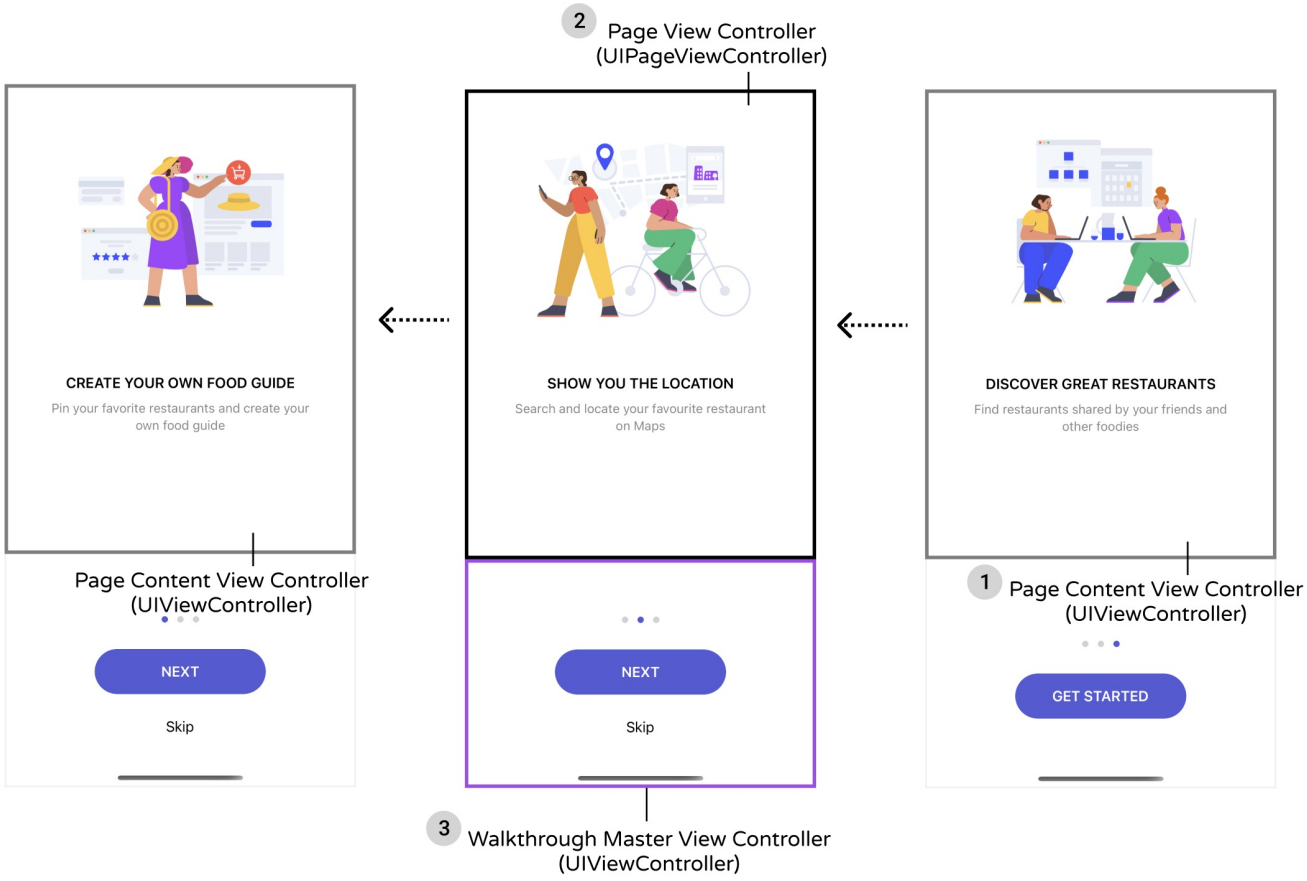


Figure 21-5. Relationship between the page view controller and the page content view controller

So how can we implement this type of walkthrough screens? Here is what we are going to do:

1. We will have a page content view controller to hold the content of the walkthrough screens. It is a plain view controller that shows an image and a couple of the description labels. You may be curious why we just need a single view controller for three pages of content. Shouldn't we have three different view controllers? If you look into figure 21-5 carefully, you should find that all walkthrough screens look very similar. Therefore, instead of building a view controller for each screen, it's better to reuse the same view controller and fill in different content.

2. Secondly, we will create a page view controller using the `UIPageViewController` class. This is the controller that handles the navigation of page content view controller I mentioned in point 1. Similar to `UINavigationController`, the `UIPageViewController` class is classified as a container controller. The container controller is designed to hold and manage multiple view controllers shown in an app. It also controls how one view controller switches to another. Figure 21-5 depicts the relationship between the page view controller and the page content view controller.
3. Lastly, we need to master view controller to hold both the page view controller (i.e. the upper part of the screen) and the buttons (i.e. the lower part of the screen).

Feeling confused? This is not a problem. Just keep reading. You will better understand how `UIPageViewController` works after implementing the walkthrough screens.

Designing the Master View Controller

Let's work it backward and begin to design the master view controller first. The master view controller, as explained before, is used to hold the page view controller, the buttons, and the page indicator.

Now open `Onboarding.storyboard` and make sure you use *iPhone 12 / iPhone 12 Pro* to design the user interface. Drag a View Controller to the storyboard. The master view controller is simply a view controller, but in order to hold the page view controller, drag a Container View to the view controller. Set the size of the container view to 414x500 points.

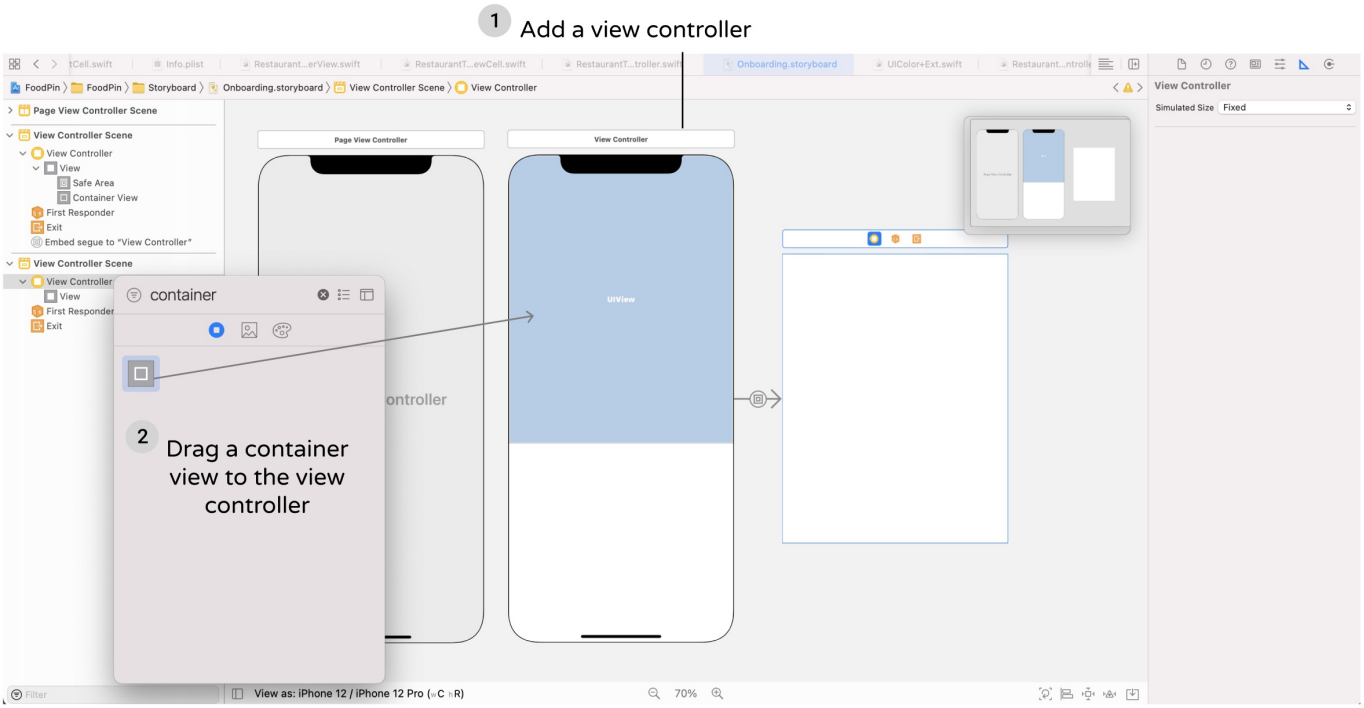


Figure 21-6. Adding a container view

A container view is a placeholder object that represents the contents of a child view controller. As soon as you add the container view to the view controller, the view is automatically associated with another view controller. If you resize the container view, the size of that view controller will be adjusted automatically.

Since the container view is designed to display the content of the page view controller, the view controller generated by Xcode is not appropriate in this case. Select the segue between the generated view controller and the container view. Click the Delete button to remove it.

Next, control-drag from the container view to the page view controller. Release the button and then choose the *Embed* option. The page view controller is now automatically resized and connected with the container view using an *Embed* segue.

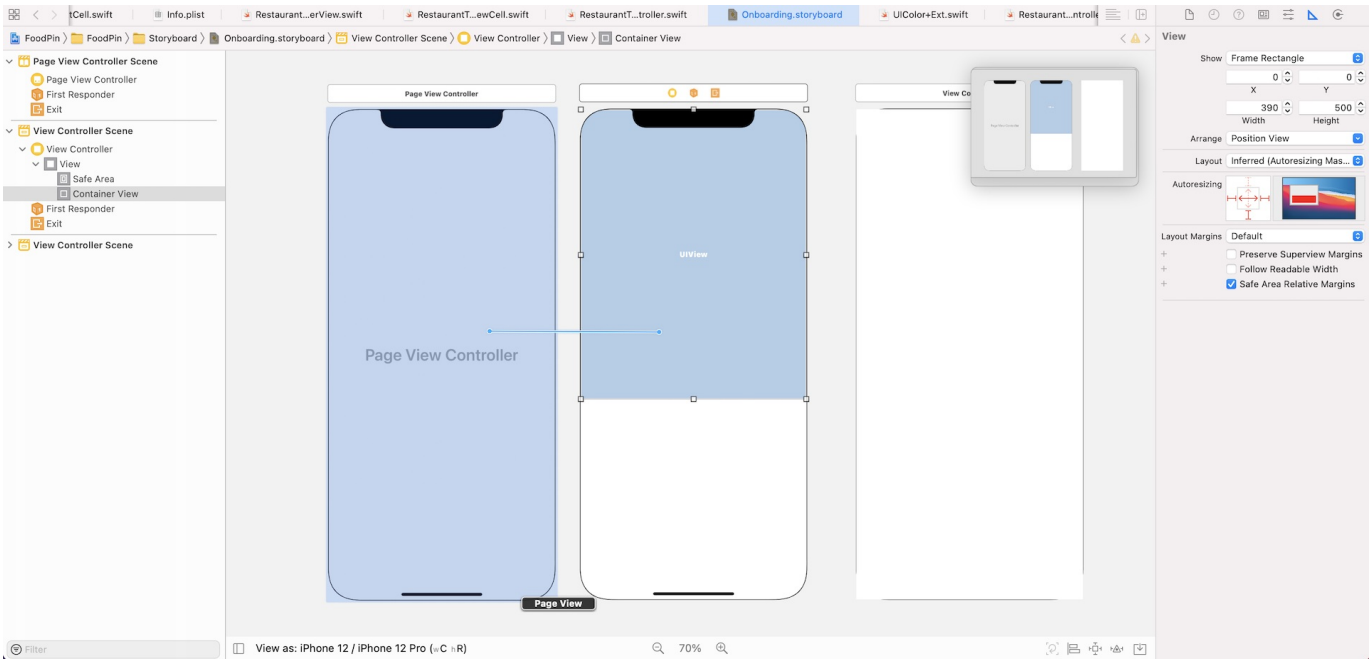


Figure 21-7. Connect the container view with the page view controller

Select the container view and click the *Add New Constraints* button. Set the spacing value of the top, left and right sides to `0`. Click *Add 3 Constraints* to confirm.

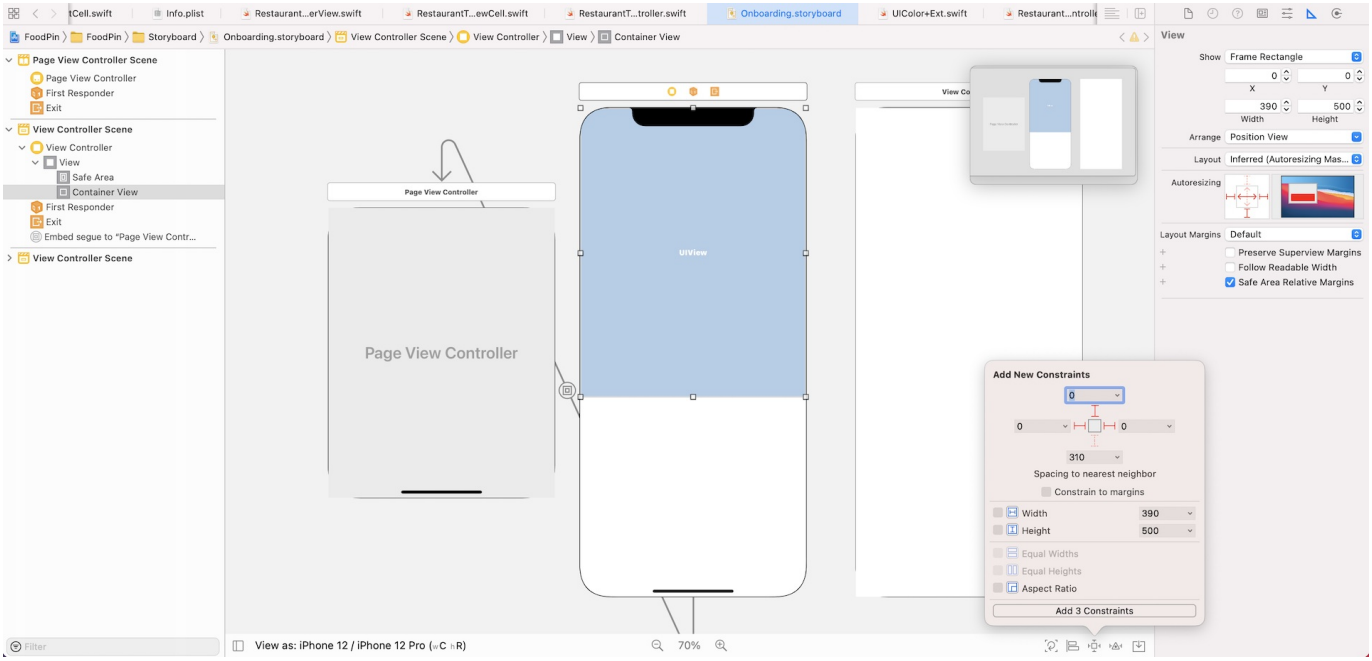


Figure 21-8. Adding a view to the lower part of the view controller

We are now going to design the lower part of the view. First, drag a View object to the view controller. In the Size inspector, set X to `0`, Y to `500`, width to `390`, and height to `344`. Again, we need to add a few layout constraints. Refer to figure 21-9 to add 4 spacing constraints.

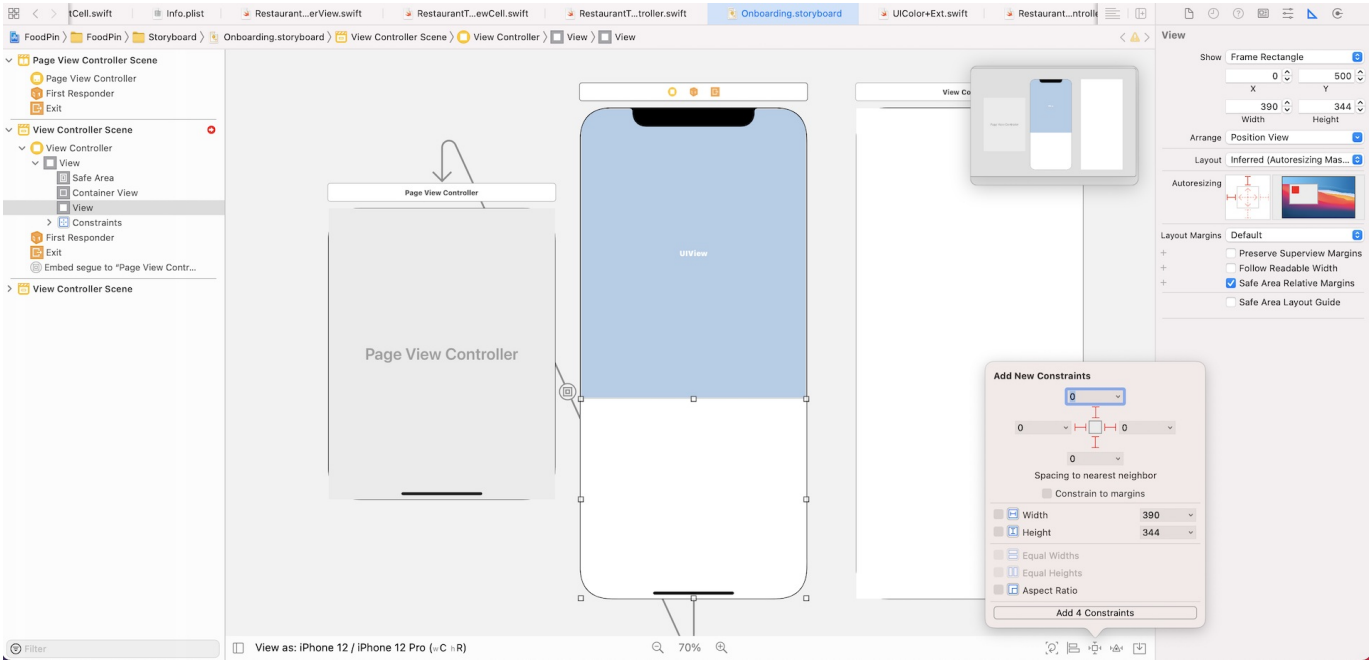


Figure 21-9. Adding a view to the lower part of the view controller

After defining the spacing constraints, Xcode still indicates that there are some layout errors. The reason is that we haven't defined a height constraint for the views. Here, I want the container view to take up two-thirds of the view controller. In other words, the empty view will take up one-third of the view.

How can we define a layout constraint for this requirement? Let me show you a very useful trick of auto layout.

In the document outline, select the `View` object. Control-drag from it to Safe Area. When the popover appears, select *Equal Heights*. This defines a height constraint for the view, saying that its height should equal the height of the safe area.

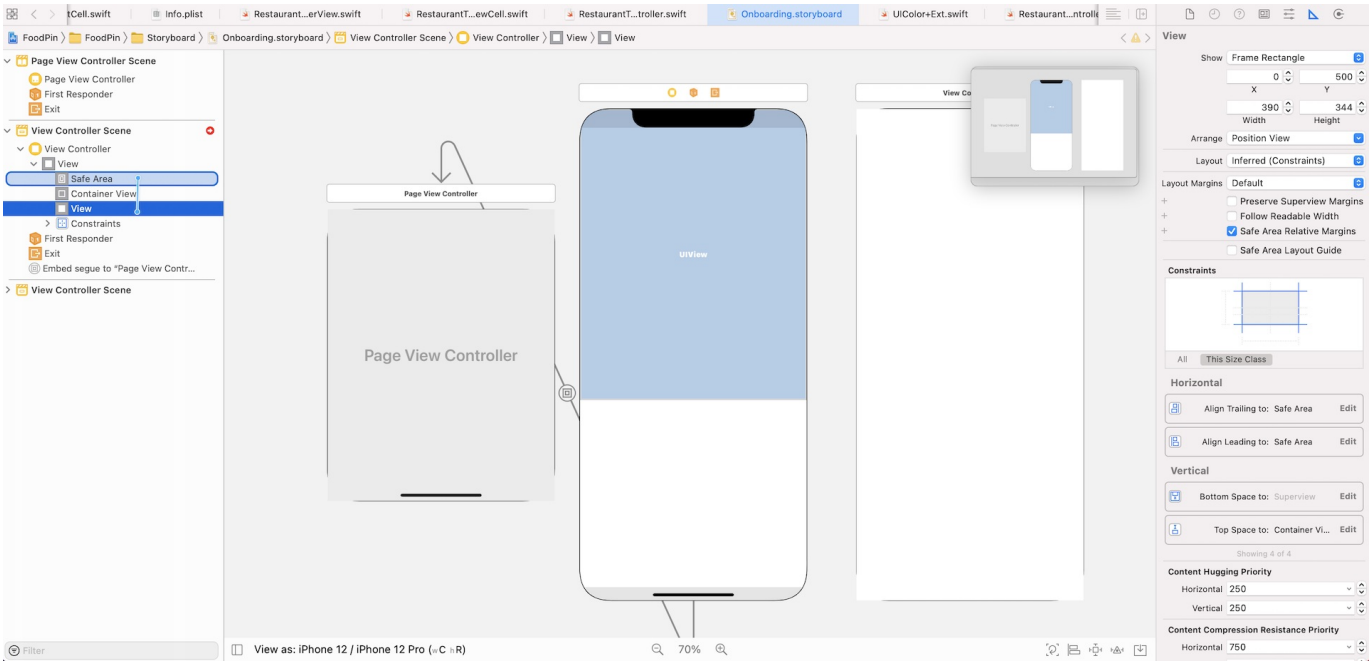


Figure 21-10. Control-drag from View to Safe Area

Now the container view is squeezed into a little rectangle. This is not what we want. So select the height constraint we just defined. Go to the Size inspector and change its multiplier from `1` to `1:3`. This specifies that the empty view takes up one-third of the safe area.

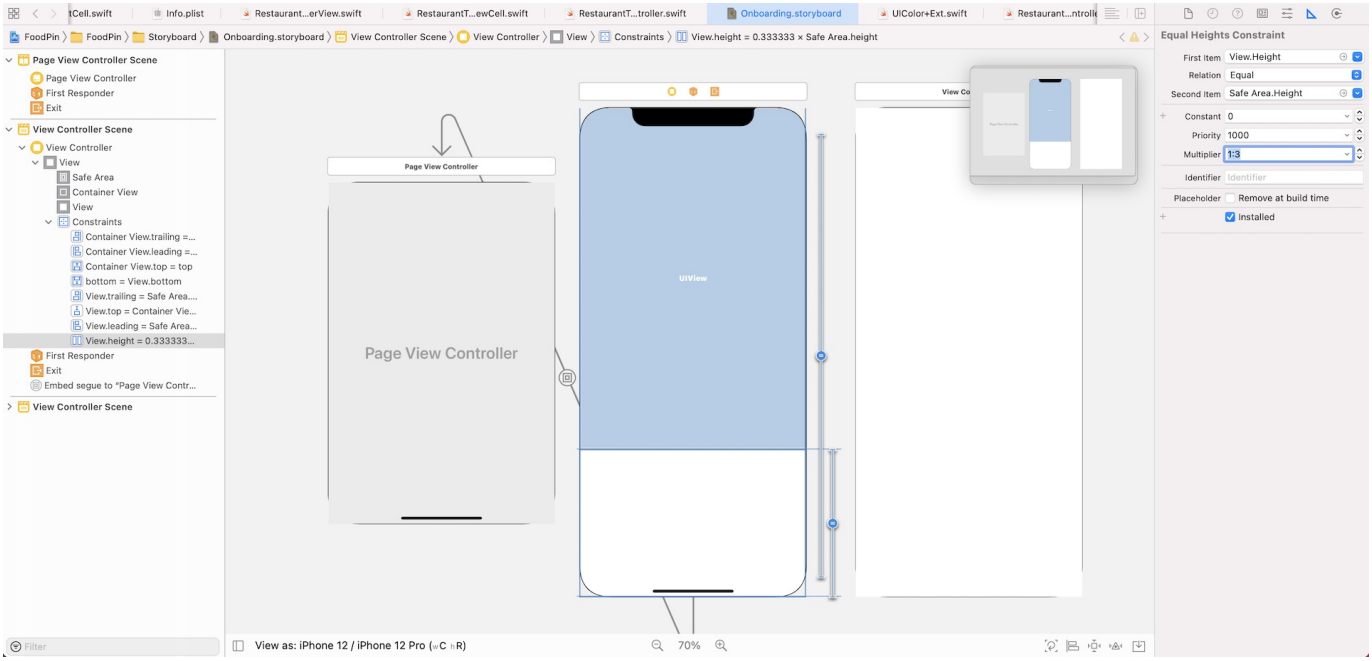


Figure 21-11. Changing the multiplier to 1:3

Next, we need to add the buttons and page indicator to the empty view. Drag a button from the object library to the empty view. In the Size inspector, set its width to `190` and height to `50`. In the Attributes inspector, change the title to `NEXT` and set its font to `Headline` text style. Also, change the font color to *white* and background color to *System Indigo Color*.

Drag another button and place it under the *NEXT* button. In the Attributes inspector, set the button's title to `Skip`. Change its font to `Body` text style and font color to *Label Color*.

In the object library, look up the page control object. Drag it to the empty view and place it right above the *NEXT* button. You can't see the dots because the color of the dot is set to transparent by default. Go to the Attributes inspector and change its tint color to *System Gray 4 Color*. Also, set its current page color to *System Indigo Color*.

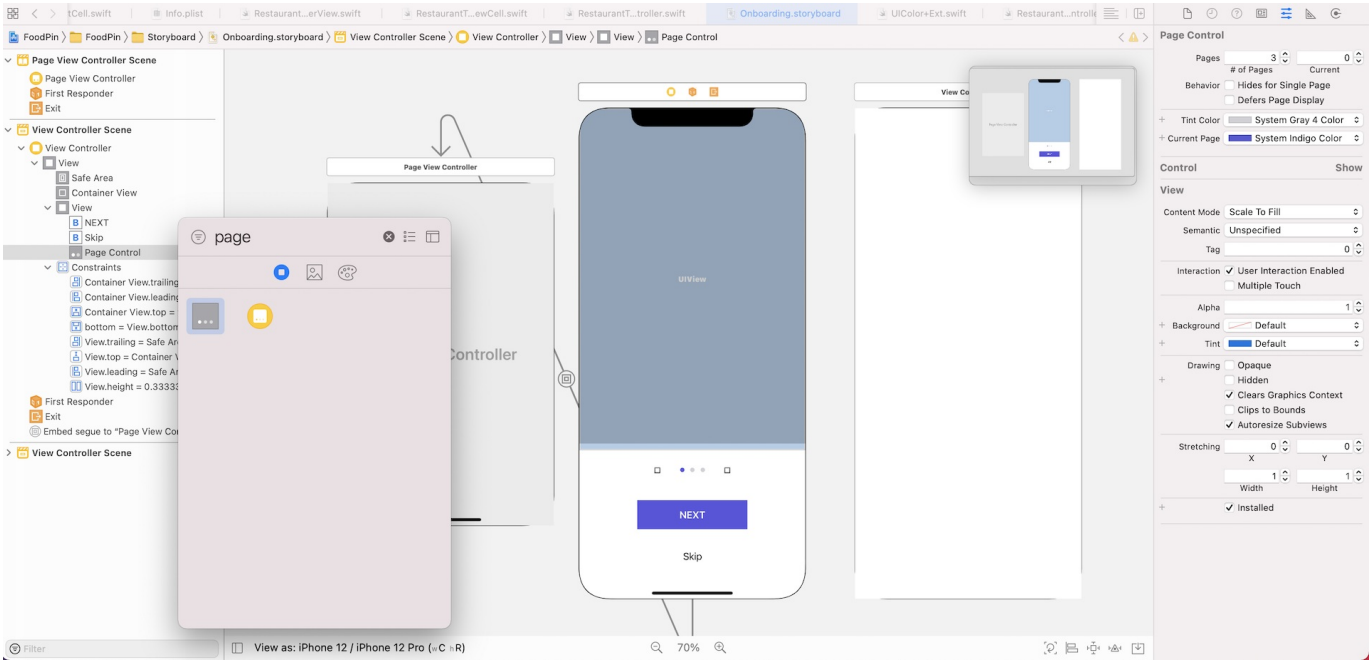


Figure 21-12. Adding the page control object

We are almost done. Now it's time to define the layout constraint for the buttons and page control. Select the *NEXT* button and click the *Add New Constraints* button. Check both width and height checkboxes to limit its width to `190` and height to `50`.

Now hold the command key and select the page control, the *NEXT* button and the *Skip* button. Click the *Embed in* button and then select *Stack view* to embed them in a vertical stack view. In the Attributes inspector, change the *spacing* to `20` to set the components apart. Lastly, click the *Align* button of the auto layout bar and add two constraints to center the stack view.

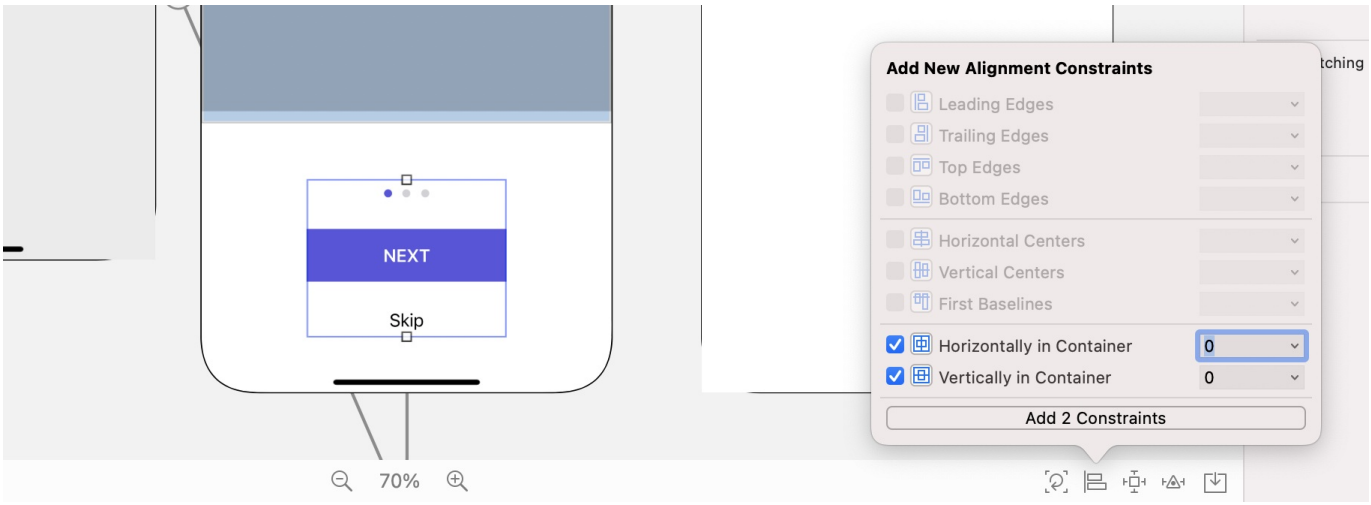


Figure 21-13. Adding the page control object

Designing the Page Content View Controller

Now that we've prepared the master view controller and the page view controller, let's move on to designing the page content view controller.

First, download the image pack from (http://www.appcoda.com/resources/swift53/onboarding.zip) and add the images into `Assets.xcasset`. Make sure you enable the *Preserve Vector Data* option.

Open `Onboarding.storyboard`. We are going to design the view controller. I suggest you change its simulated size such that its size is similar to that of the container view. It is not a must but it will help you design the page content view controller. In the Size inspector, set the size option to `Freeform` and change the height to `623`.

Now follow these procedures to lay out the UI:

- Drag an image view to the view controller. Set its size to 333x229 points.
- Add a label object and name it `HEADING`. Change to the font to `Headline` text style. Also, set the alignment to `center`. You are free to choose your own font

color. For me, I prefer *Label Color*.

- Next, drag another label object to the view controller and name it `Subheading`. Set its font to `Subhead` text style. Change the alignment to `center`. Again, choose your preferred font color. I use *Secondary Label Color* in this case.

You'll end up with a screen similar to that shown in figure 21-14.

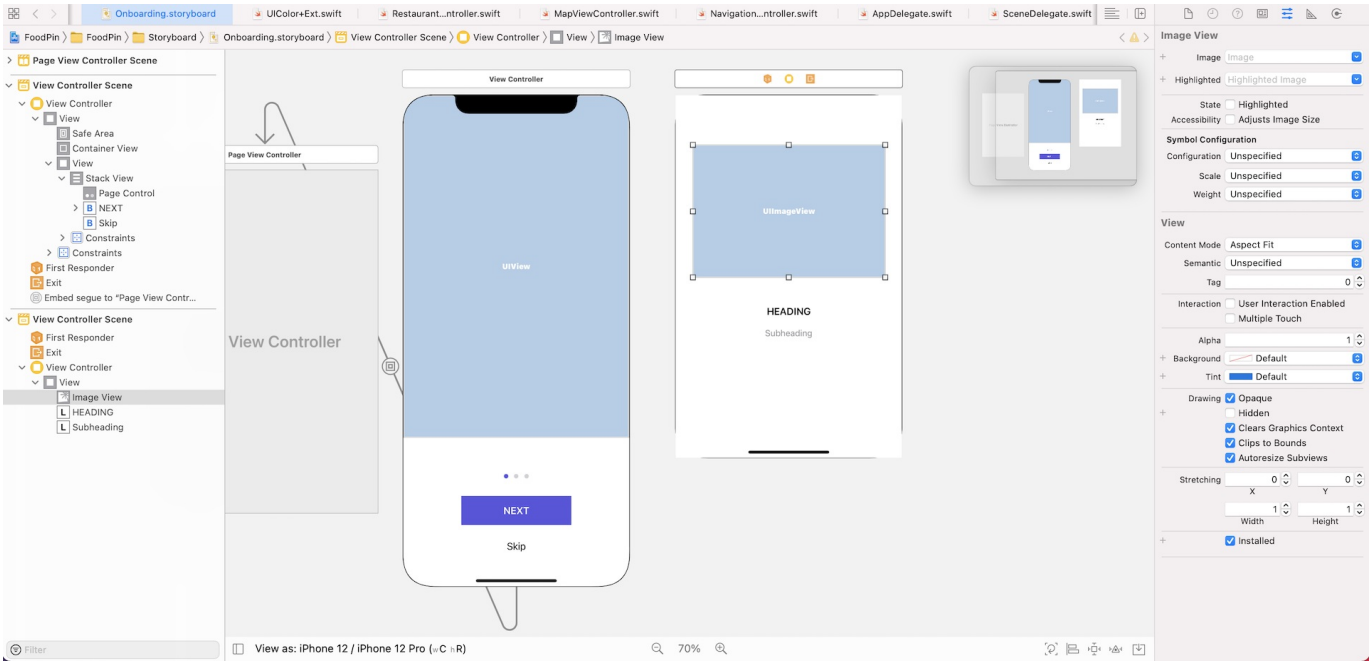


Figure 21-14. Designing the View Controller for walkthrough screens

As usual, you have to define the layout constraints for the components. First, hold the command key and select both labels. Click *Embed in* and then select *Stack view* to embed them in a vertical stack view. Change the stack view's option in the Attributes inspector. Set the *Alignment* option to `Center` and the *Spacing* option to `10`.

Next, select the stack view and the image view. Click *Embed in* and then select *Stack view* to embed both view objects in another stack view. In the Attributes inspector, set the *Alignment* option to `Center` and the *Spacing* option to `50`.

Select the stack view you created in the previous step. Click the *Add New Constraints* button and set the value of top, left, bottom and right sides to `50`, `24`, `24`, and `10`.

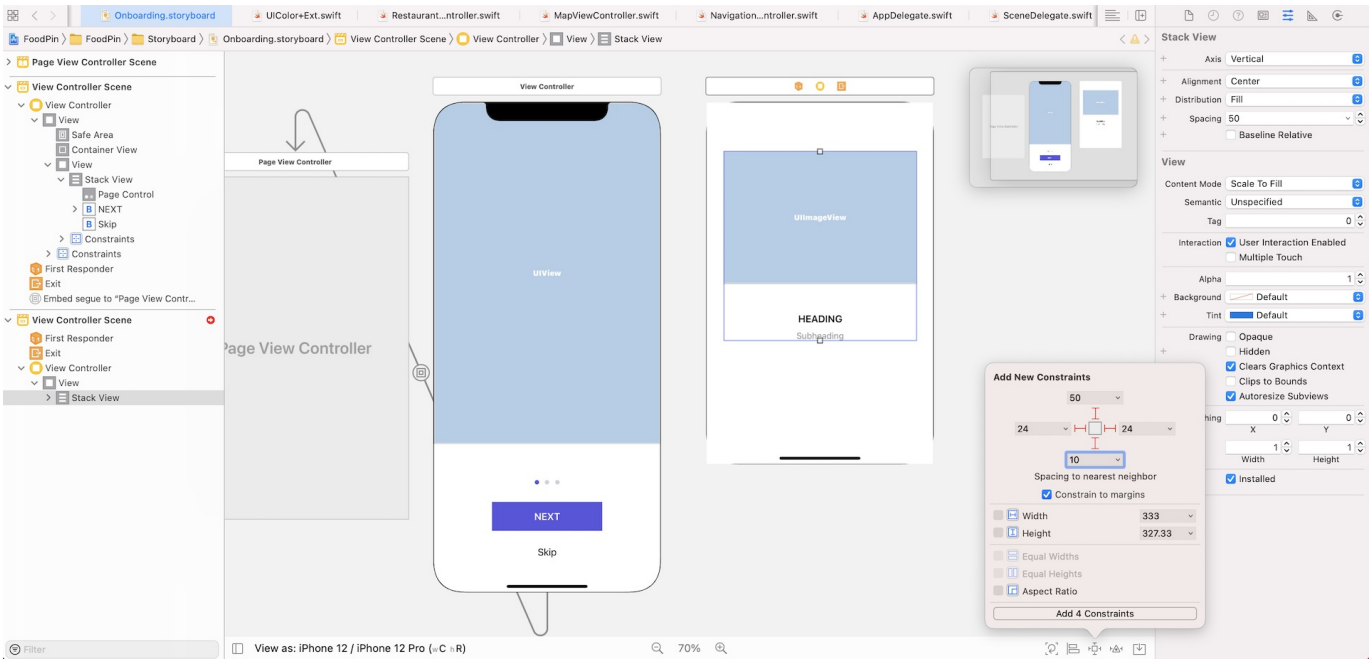


Figure 21-15. Adding layout constraints for the stack view

Next, select the image view and click the *Add New Constraints* button. Tick the *Aspect Ratio* checkbox to add an aspect ratio constraint.

Finally, choose the bottom constraint and change its *Relation* option from `Equal` to `Great than or Equal`.

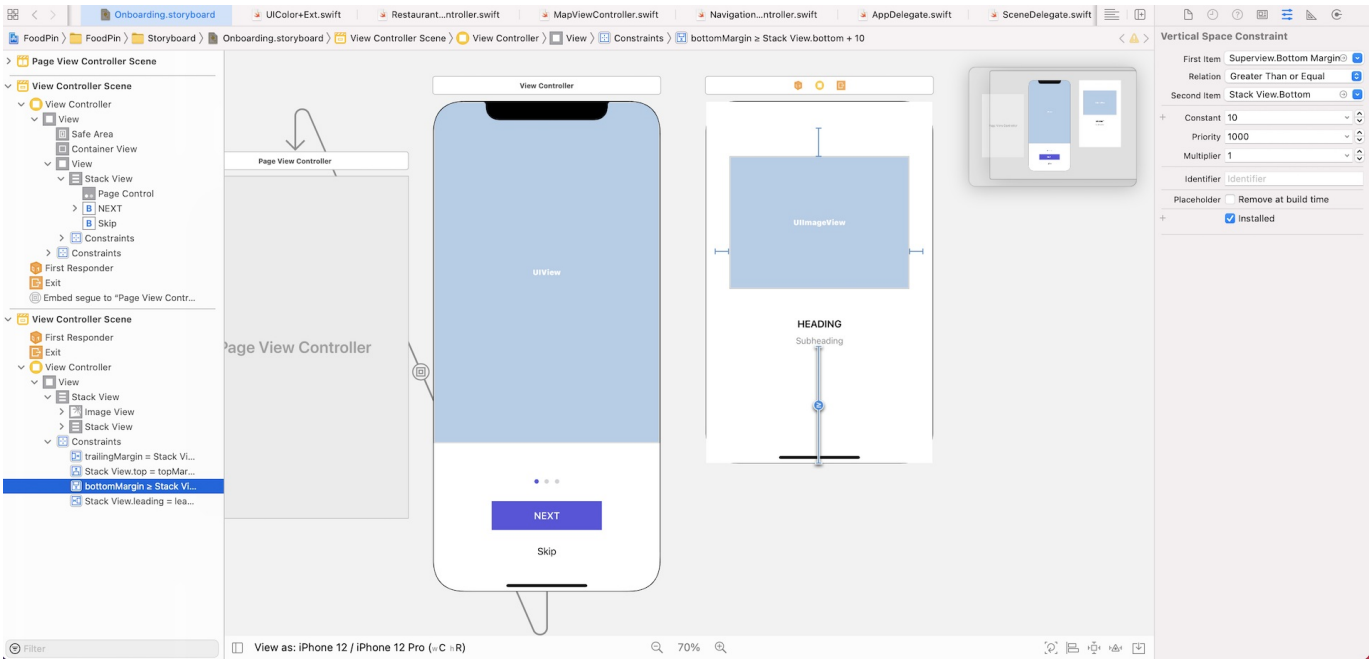


Figure 21-16. Adjusting the bottom constraint

If you have any difficulties creating the UI, you can refer to this Xcode project template (http://www.appcoda.com/resources/swift53/FoodPinOnboarding1.zip).

Creating WalkthroughContentViewController Class

Okay, we have completed the UI design of the walkthrough views. The next step, as you should know, is to create the class to pair with the view controllers.

We will start with the page content view controller that we just created in the previous section. From now on, I will refer this controller to `WalkthroughContentViewController`.

Now go the project navigator and right-click the `Controller` folder. Select *New file...* and choose the *Cocoa Touch Class* template. Name the class `WalkthroughContentViewController` and set it as a subclass of `UIViewController`.

Once the file is created, declare the following outlets and instance variables in the class:

```
@IBOutlet var headingLabel: UILabel! {
    didSet {
        headingLabel.numberOfLines = 0
    }
}
@IBOutlet var subHeadingLabel: UILabel! {
    didSet {
        subHeadingLabel.numberOfLines = 0
    }
}
@IBOutlet var contentImageView: UIImageView!

var index = 0
var heading = ""
var subHeading = ""
var imageFile = ""
```

We will use this class to support multiple walkthrough screens. The index variable is used to store the current page index. For instance, the first walkthrough screen will have the index value of `0`. The view controller is designed to display an image, heading, and subheading. So we create three variables for data passing.

For both the heading and subheading labels, I add a `didSet` observer to set the `numberOfLines` property to `0`. This will allow the label to support multiple lines.

Next, change the `viewDidLoad()` method to the following:

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

    headingLabel.text = heading
    subHeadingLabel.text = subHeading
    contentImageView.image = UIImage(named: imageFile)
}
```

Here, we initialize the labels and image view. As you should know, the next thing we have to do is to establish a connection between the UI components and the outlet variables.

Go to `Onboarding.storyboard` and select the page content view controller you've created. In the Identity inspector, set the custom class to `WalkthroughContentViewController`. Also, set the storyboard ID field to `WalkthroughContentViewController`. Later we use this storyboard ID to instantiate the view controller. I will further explain it in the next section.

Next, right-click the page content view controller in the document outline and establish the following connections:

- Connect the `headingLabel` outlet with the *HEADING* label.
- Connect the `subHeadingLabel` outlet with the *Subheading* label.
- Connect the `contentImageView` outlet with the Image View object.

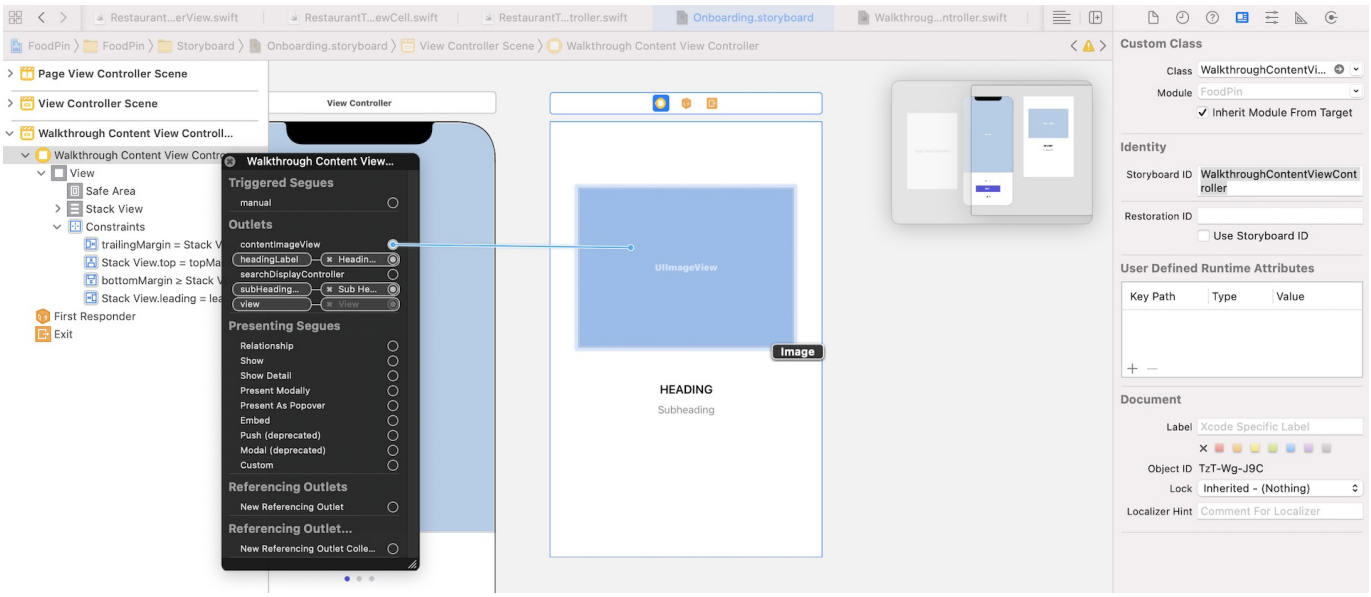


Figure 21-17. Establishing connections with the outlet variables

Implementing the Page View Controller

Now that we've prepared the content view controller, the next step is to create each of the content view controller and add it into the page view controller (i.e. `UIPageViewController`) so that the user can navigate between them.

You have two ways to tell `UIPageViewController` what to display; you either provide the content view controllers one at a time or on-demand using a data source. If you just want to display a certain view controller in `UIPageViewController`, you can call `setViewControllers(_:direction:animated:completion:)` method with the view controller to be displayed.

As our app supports gesture-based navigation for the walkthrough, it's required to use the on-demand approach. For this approach, you assign a data source object serving as the provider of the content view controllers. Every time when a user navigates from one page to another, `UIPageViewController` asks its data source, "*Hey, what content view controller should be displayed? Please pass it to me.*" The data source object then returns the corresponding content view controller.

The data source object should conform to the `UIPageViewControllerDataSource` protocol and implement the following required methods:

- pageViewController(_:viewControllerBefore:)
- pageViewController(_:viewControllerAfter:)

Does UIPageViewControllerDataSource sound familiar to you? It is very similar to UITableViewDataSource. You implement the required methods of UITableViewDataSource to provide the data of a table view. UIPageViewControllerDataSource requires you to implement the methods to provide the page content view controller.

Both methods may be called when a user navigates between pages. When called, the method is passed with a specific view controller. Your job is to determine and return a content view controller for display before/after the given controller. Let's say, if you have a content view controller with the index value of `1`. The `UIPageViewController` object will ask:

- Hey, what is the *next* view controller? In this case, you should return a content view controller with index #2.
- Hey, what is the *before* view controller? In this case, you should return a content view controller with index #0.

Now that you have some ideas about how `UIPageViewController` works, let's proceed to create a new class for the page view controller. Right-click the `Controller` folder and select *New file...*. Choose the *Cocoa Touch Class* template. Name the class `WalkthroughPageViewController` and set it as a subclass of `UIPageViewController`.

Once created, open `WalkthroughPageViewController.swift`. Declare and initialize the heading, subheading, and image variables, which are used when creating the content view controllers in the class. Also, declare a `currentIndex` variable to store the current index of the page view:

```
var pageHeadings = ["CREATE YOUR OWN FOOD GUIDE", "SHOW YOU THE LOCATION",
"DISCOVER GREAT RESTAURANTS"]
var pageImages = ["onboarding-1", "onboarding-2", "onboarding-3"]
var pageSubHeadings = ["Pin your favorite restaurants and create your own
food guide",
                       "Search and locate your favourite restaurant on Maps",
                       "Find restaurants shared by your friends and other food
ies"]

var currentIndex = 0
```

Next, create an extension and implement the two required methods of the `UIPageViewControllerDataSource` protocol:

```
extension WalkthroughPageViewController: UIPageViewControllerDataSource {

    func pageViewController(_ pageViewController: UIPageViewController, vi
ewControllerBefore viewController: UIViewController) -> UIViewController?
{

        var index = (viewController as! WalkthroughContentViewController).
index
        index -= 1

        return contentViewController(at: index)
    }

    func pageViewController(_ pageViewController: UIPageViewController, vi
ewControllerAfter viewController: UIViewController) -> UIViewController? {

        var index = (viewController as! WalkthroughContentViewController).
index
        index += 1

        return contentViewController(at: index)
    }

}
```

The above methods are very straightforward. First, we get the current page index of the given view controller. Depending on the method, we simply increase/decrease the index number and return the view controller to display.

As you may notice, we haven't created the `contentViewController(at:)` method. This is a helper method that is designed to create the page content view controller on demand. It takes in the index parameter and creates the corresponding page content controller.

Now, add the helper method in the extension:

```
func contentViewController(at index: Int) -> WalkthroughContentViewControl
ler? {
    if index < 0 || index >= pageHeadings.count {
        return nil
    }

    // Create a new view controller and pass suitable data.
    let storyboard = UIStoryboard(name: "Onboarding", bundle: nil)
    if let pageContentViewController = storyboard.instantiateViewControlle
r(withIdentifier: "WalkthroughContentViewController") as? WalkthroughConte
ntViewController {

        pageContentViewController.imageFile = pageImages[index]
        pageContentViewController.heading = pageHeadings[index]
        pageContentViewController.subHeading = pageSubHeadings[index]
        pageContentViewController.index = index

        return pageContentViewController
    }

    return nil
}
```

There is a lot going on here. Let's look into the method line by line.

This method is designed to accept a page index parameter. Say, a value of `0` is passed, this method will create the first walkthrough screen. At the beginning of the method, we perform some validation to make sure that the given index is valid.

Recalled that we have set a storyboard ID for the walkthrough content view controller in Interface Builder, the ID is used as a reference for creating the view controller instance. To instantiate a view controller in a storyboard, you first create an instance of the specific storyboard. Here, it is the `Onboarding` storyboard. Then

you call the `instantiateViewController(withIdentifier:)` method with a specific storyboard ID. The method returns you a generic view controller. This is why we use `as?` to downcast the object to `WalkthroughContentViewController`.

Following the instantiation, we assign the content view controller with the specific image, heading, subheading, and index.

Lastly, update the `viewDidLoad` method to the following:

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

    // Set the data source to itself
    dataSource = self

    // Create the first walkthrough screen
    if let startingViewController = contentViewController(at: 0) {
        setViewControllers([startingViewController], direction: .forward,
animated: true, completion: nil)
    }
}
```

In the code above, we set the data source of `UIPageViewController` to itself. We also create the first content view controller when the page view controller is first loaded.

With the class configured, go to storyboard and select the page view controller. In the Identity inspector, change the custom class from `UIPageViewController` to `WalkthroughPageViewController`. Also, set the storyboard ID to `WalkthroughPageViewController`.

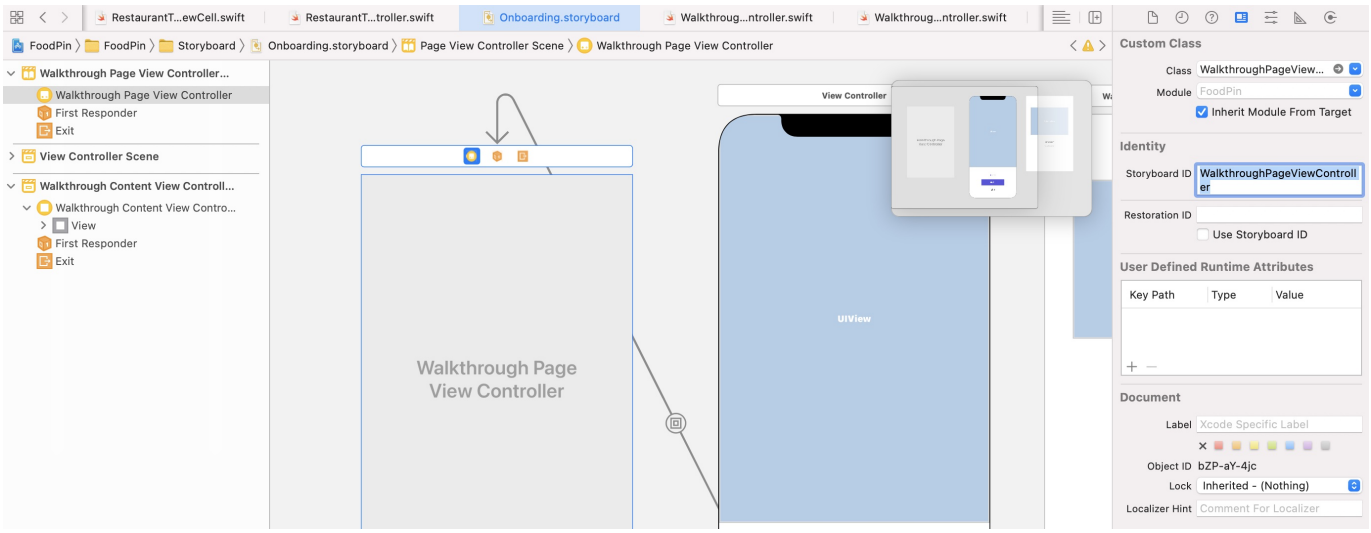


Figure 21-18. Setting the custom class and storyboard ID

Implementing the Walkthrough View Controller

Finally, let's create a class for the walkthrough view controller. In the project navigator, right-click the `Controller` folder and select *New file*.... Choose the *Cocoa Touch Class* template to proceed. Name the class `WalkthroughViewController` and set its subclass to `UIViewController`.

Once created, declare the outlet variables in the class:

```
@IBOutlet var pageControl: UIPageControl!
@IBOutlet var nextButton: UIButton! {
    didSet {
        nextButton.layer.cornerRadius = 25.0
        nextButton.layer.masksToBounds = true
    }
}
@IBOutlet var skipButton: UIButton!
```

Now switch over to `Onboarding.storyboard`. Select the walkthrough view controller, which is the master view that holds the page view controller. In the Identity inspector, set the custom class to `WalkthroughViewController` and the storyboard ID to `WalkthroughViewController`.

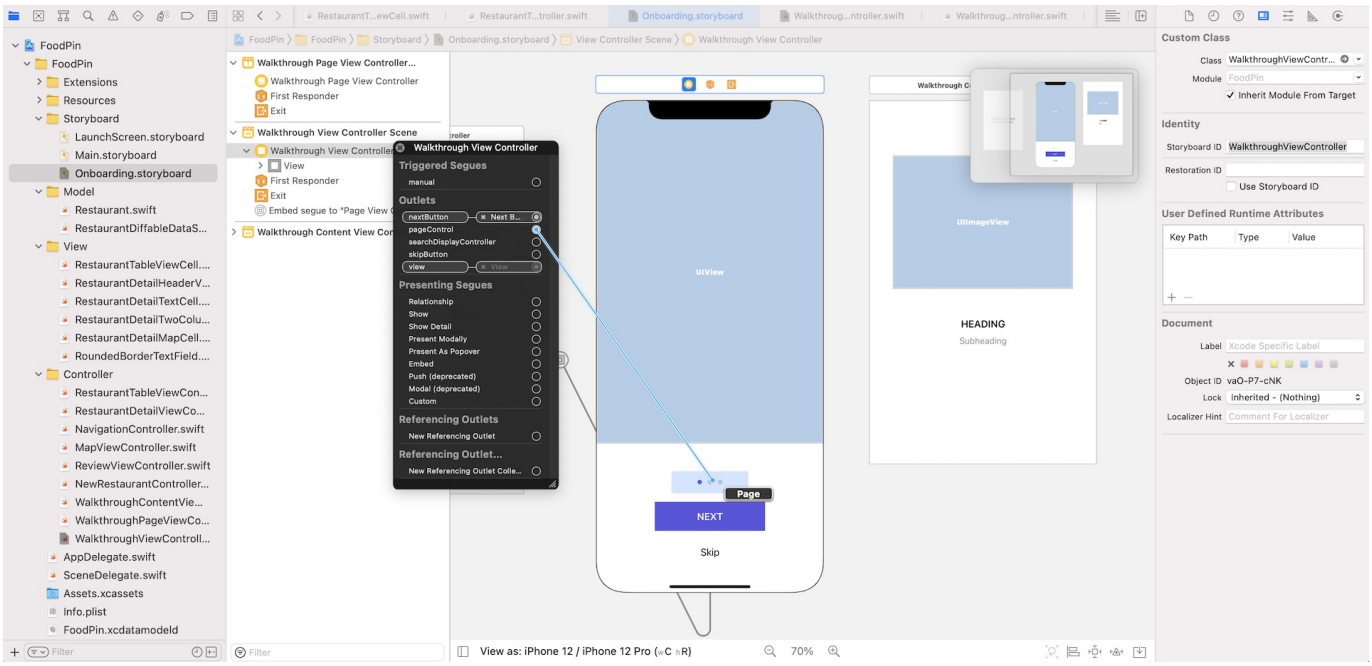


Figure 21-19. Setting the custom class and storyboard ID

Next, connect the outlet variables with the UI component:

- Connect the `pageControl` outlet to the page control
- Connect the `nextButton` outlet to the *NEXT* button
- Connect the `skipButton` outlet to the *Skip* button

Displaying the Walkthrough Screens

You're almost ready to test out the walkthrough screens. Because we want to bring up the walkthrough view controller when a user first launches the app. We instantiate the controller in the `RestaurantTableViewController` class. Insert the

following method:

```
override func viewDidAppear(_ animated: Bool) {
    let storyboard = UIStoryboard(name: "Onboarding", bundle: nil)
    if let walkthroughViewController = storyboard.instantiateViewControlle
r(withIdentifier: "WalkthroughViewController") as? WalkthroughViewControll
er {

        present(walkthroughViewController, animated: true, completion: nil
)
    }
}
```

The `viewDidAppear` method will be automatically called by iOS. We make use of this method to bring up the walkthrough view controller. The code is self-readable. We just instantiate the `WalkthroughViewController` object and present it in a modal way.

Now you're ready to hit the Run button and test the app. As soon as you launch the app, it brings up the walkthrough screens and you can navigate through the screens using gestures. Swipe right the image will bring you the next screen.

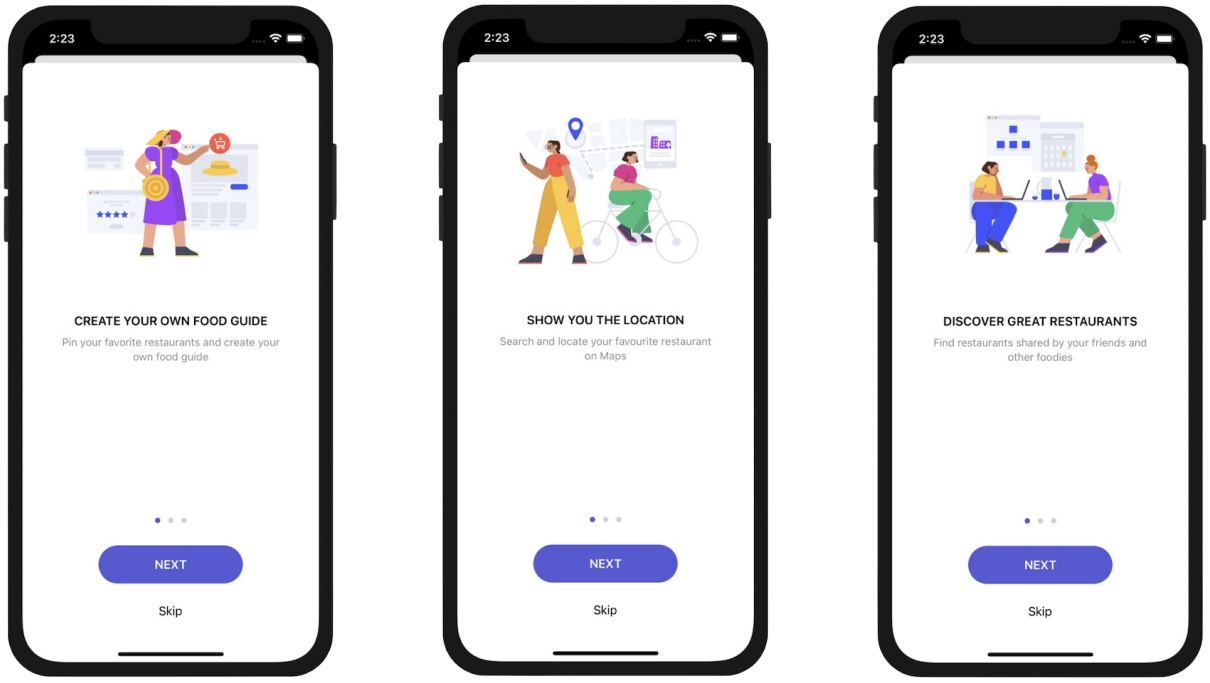


Figure 21-20. Walkthrough screens without page indicator

Handling the Page Indicator and the Next/Skip Buttons

The walkthrough screens work great, right? However, you should find two problems:

- The next and skip buttons are not working yet. That is obvious because we haven't provided the implementation.
- In the last screen, the skip button should be hidden and the next button should be changed to *GET STARTED* button.
- The page indicator doesn't work yet. The red dot doesn't change its position to reflect the current index of the walkthrough screen.

Now let's tackle both issues. First, we are going to implement the methods for the *Next* and *Skip* button.

The function of the *Skip* button is simple. When the user taps the button, the app dismisses the walkthrough view controller. Therefore, create an action method in the `WalkthroughViewController` class:

```
@IBAction func skipButtonTapped(sender: UIButton) {
    dismiss(animated: true, completion: nil)
}
```

In the method, we just call the `dismiss` method to dismiss the view controller.

For the next button, it is more complicated. When a user taps the button, it automatically shows the next walkthrough screen. How can we programmatically show the next screen of the page view controller?

In the `WalkthroughPageViewController` class, insert a new method named `forwardPage()`:

```
func forwardPage() {
    currentIndex += 1
    if let nextViewController = contentViewController(at: currentIndex) {
        setViewControllers([nextViewController], direction: .forward, anim
ated: true, completion: nil)
    }
}
```

When the method is called, it automatically creates the next content view controller. If the controller can be created, we call the built-in `setViewControllers` method and navigate to the next view controller.

Now go back to `WalkthroughViewController` and add a property:

```
var walkthroughPageViewController: WalkthroughPageViewController?
```

This is the property that stores a reference to the `WalkthroughPageViewController` object. Later, we will use it to find out the current index of the walkthrough screen.

Recall that the container view connects with the Walkthrough Page View Controller through an embed segue (see figure 21-8), we can add the `prepare` method to get the reference of the Walkthrough Page View Controller. In the `WalkthroughViewController` class, insert the following method:

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    let destination = segue.destination
    if let pageViewController = destination as? WalkthroughPageViewControl
ler {
        walkthroughPageViewController = pageViewController
    }
}
```

Now, it is time to create the `nextButtonTapped` action method in the `WalkthroughViewController` class. The method will be called when the next button is tapped:

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

    if let index = walkthroughPageViewController?.currentIndex {
        switch index {
        case 0...1:
            walkthroughPageViewController?.forwardPage()

        case 2:
            dismiss(animated: true, completion: nil)

        default: break

        }
    }

    updateUI()
}
```

The code is self-explanatory. We perform a different action, depending on the current page index. For the first two pages, we call the `forwardPage()` method to display the next walkthrough screen. For the last walkthrough page, we call the `dismiss` method to dismiss the view controller.

Okay, we already implemented the action methods for both Next and Skip buttons. However, we still haven't managed the page control. This is one of the purposes of the `updateUI` method. In the code above, we call `updateUI()` at the end of the method. Add the following code to create the method:

```
func updateUI() {

    if let index = walkthroughPageViewController?.currentIndex {
        switch index {
        case 0...1:
            nextButton.setTitle("NEXT", for: .normal)
            skipButton.isHidden = false
        case 2:
            nextButton.setTitle("GET STARTED", for: .normal)
            skipButton.isHidden = true
        default: break
        }

        pageControl.currentPage = index
    }

}
```

The `updateUI()` method handles two things. First, it controls the title of the next button and determines whether the skip button should be hidden. Secondly, it changes the indicator of the page control by setting the `currentPage` property.

Now that we have prepared the action methods, let's switch over to `Onboarding.storyboard` to connect the methods. Control-drag from the *NEXT* button to the Walkthrough View Controller. Select `nextButtonTappedWithSender:` when prompted.

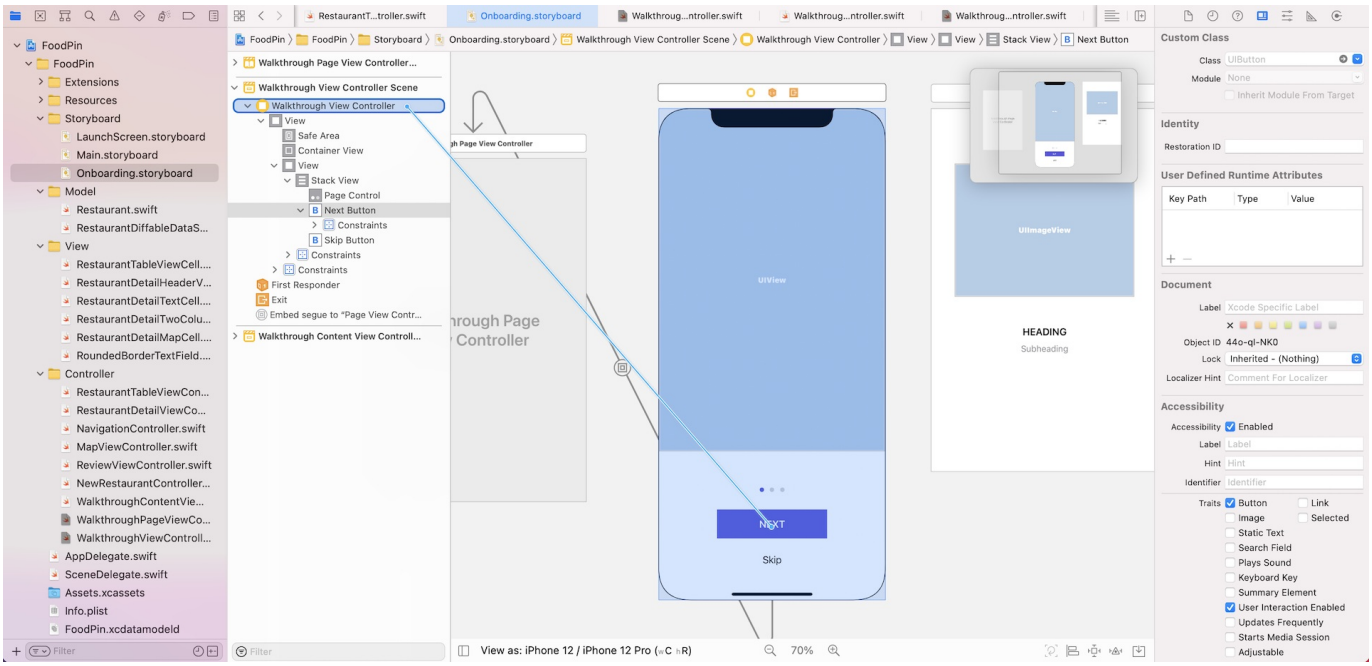


Figure 21-21. Connecting the button with its action method

Repeat the same procedures for the *Skip* button but choose `skipButtonTappedWithSender:` instead.

Now run the app to test it out! Tapping the next button will show the next walkthrough screen. Most importantly, the page indicator works as expected.

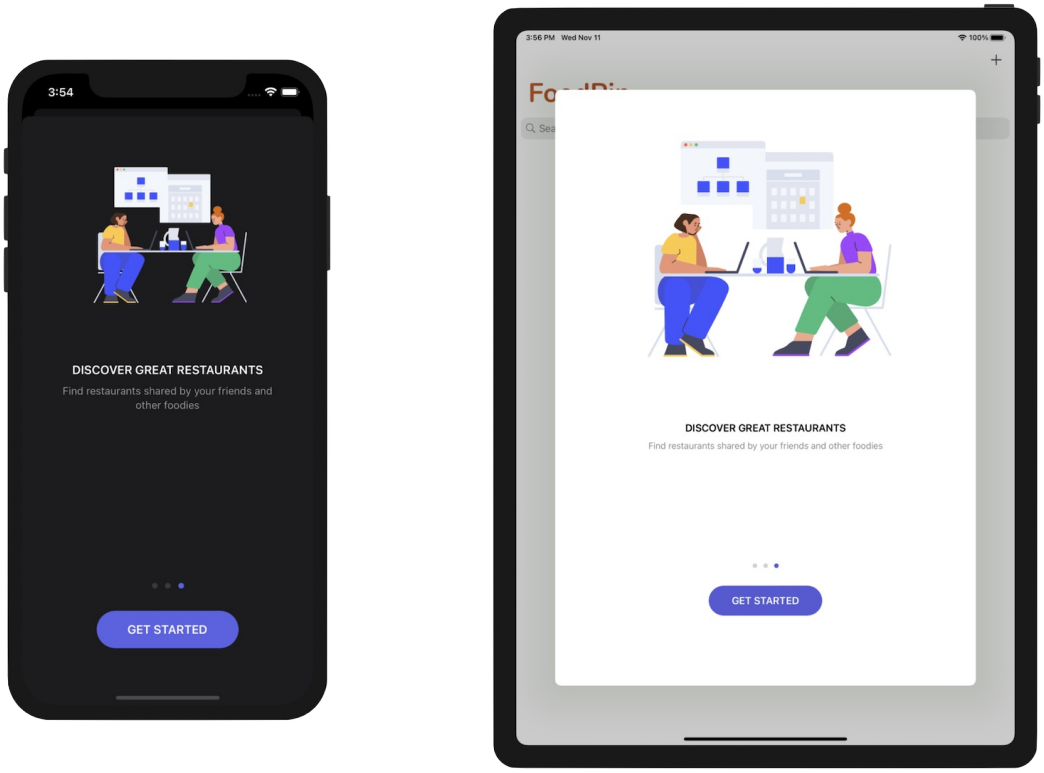


Figure 21-22. The last screen now hides the Skip button

Updating the Page Indicator for Gesture-based Navigation

Do you find another issue with the page indicator?

The walkthrough view controller supports both gesture-based and button-based navigation. When you tap the *NEXT* button, it navigates to the next screen with an animated transition and the page indicator updates accordingly. That's perfect! However, if you navigate to the next screen using gestures, the page control doesn't update itself.

What's the problem here? When the user swipes the walkthrough screen, the page view controller didn't notify the walkthrough view controller to update the page indicator.

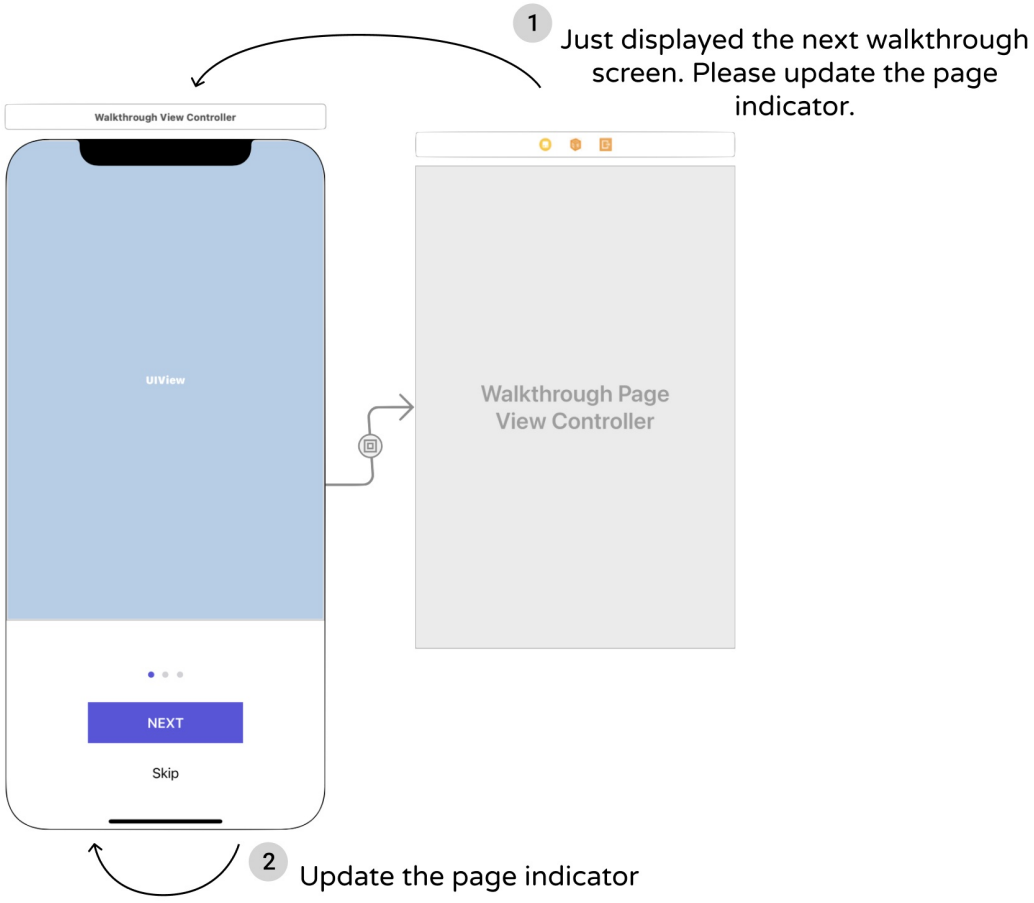


Figure 21-23. The page view controller should find a way to communicate with the walkthrough view controller

Okay, how can the page view controller inform the walkthrough view controller? In iOS programming, one common approach to performing this kind of notification is by using a delegate. The general idea is that the Walkthrough Page View Controller defines a delegate protocol with a required method.

For the Walkthrough View Controller (or any view controller) that wants to be informed, it has to adopt that delegate protocol and tell the Walkthrough Page View Controller who the delegate is. In this example, the delegate is the Walkthrough View Controller.

At an appropriate time, the Walkthrough Page View Controller will invoke that method to tell its delegate the current index. The delegate can then update the page indicator.

Okay, let's get to the implementation.

In the `WalkthroughPageViewController.swift` file, insert the following code and place it right below `import UIKit` :

```
protocol WalkthroughPageViewControllerDelegate: class {
    func didUpdatePageIndex(currentIndex: Int)
}
```

You have adopted some protocols before like `UITableViewDelegate` . But this is the very first time you define your own protocol. A protocol in Swift lets you define a blueprint of methods for a particular task. It starts with a `protocol` keyword followed by the protocol name. In the body, you declare the method definition. Here, we declare a `didUpdatePageIndex` method that tells its delegate the current page index.

In the `WalkthroughPageViewController` class, declare a `walkthroughDelegate` variable to hold the delegate:

```
weak var walkthroughDelegate: WalkthroughPageViewControllerDelegate?
```

In most cases, we use the `weak` keyword for delegate to prevent memory leak. I will not go into memory management in details here. To learn more about that, refer to the Automatic Reference Counting section of the Swift Programming Language guide (https://docs.swift.org/swift-book/LanguageGuide/AutomaticReferenceCounting.html).

Now that we have defined our protocol, the next question is when we should call the delegate's method?

Apparently, we should inform the delegate when the next walkthrough screen completely shows. There is a built-in protocol called `UIPageViewControllerDelegate` that comes with the following method:

```
optional func pageViewController(_ pageViewController: UIPageViewControlle
r, didFinishAnimating finished: Bool, previousViewControllers: [UIViewCont
roller], transitionCompleted completed: Bool)
```

If you adopt the `UIPageViewControllerDelegate` protocol and implement the method, the method will be automatically called after a gesture-driven transition completes. This is exactly what we are looking for.

Again, we will create an extension to adopt the `UIPageViewControllerDelegate` protocol and implement the method like this:

```
extension WalkthroughPageViewController: UIPageViewControllerDelegate {

    func pageViewController(_ pageViewController: UIPageViewController, di
dFinishAnimating finished: Bool, previousViewControllers: [UIViewControlle
r], transitionCompleted completed: Bool) {

        if completed {
            if let contentViewController = pageViewController.viewControll
ers?.first as? WalkthroughContentViewController {

                currentIndex = contentViewController.index

                walkthroughDelegate?.didUpdatePageIndex(currentIndex: cont
entViewController.index)
            }

        }
    }

}
```

We first check if the transition is completed and find out the current page index. Then we call the `didUpdatePageIndex` to inform the delegate.

Finally, insert the following line of code in the `viewDidLoad()` method to assign the delegate of the `UIPageViewControllerDelegate` protocol:

```
delegate = self
```

Okay, it's time to complete the puzzle. Now open the `WalkthroughViewController.swift` file and create an extension to adopt the `WalkthroughPageViewControllerDelegate` protocol:

```
extension WalkthroughViewController: WalkthroughPageViewControllerDelegate
{

    func didUpdatePageIndex(currentIndex: Int) {
        updateUI()
    }

}
```

Then update the `prepare` method like this:

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    let destination = segue.destination
    if let pageViewController = destination as? WalkthroughPageViewControl
ler {
        walkthroughPageViewController = pageViewController
        walkthroughPageViewController?.walkthroughDelegate = self
    }
}
```

The method is nearly the same as before, except that we assign a value for the `walkthroughDelegate` property.

Cool! We've made the change. Will it work? Just compile and run the project to test it out. If you got everything right, the page indicator should now work properly.

Wait! Another Bug!

Now the walkthrough is up and running. However, it's displayed over and over again, every time when you launch the app again or navigate back from the detailed view. The method `viewDidAppear` is called every time when the main

screen appears. We haven't implemented any logic to control the reappearance of the walkthrough screens.

In general, walkthrough or tutorial screens are displayed only when a user launches the app for the very first time. In order to do so, we need to find a way to store a status that indicates whether the user has viewed the walkthrough.

Where should we keep the status?

You've learned Core Data, so you probably want to keep the status in the local database. While this is an option, there is a simpler way to store the application and user settings.

Introducing UserDefaults

The iOS SDK comes with the `UserDefaults` class for managing application and user-related settings. For example, you can save a status to indicate whether the user has gone through the walkthrough. Or if you've enhanced the app to display the average price of a basic meal and want to allow users to configure the default currency, you can also use `UserDefaults` to store the user's preference.

The `UserDefaults` class provides a programmatic interface for interacting with the defaults system. The defaults system is created automatically and available for all code across your app. Data stored in the defaults system is persistent. In other words, you can still access the data even if the user quits the app or the app crashes.

To use `UserDefaults`, all you need is to get the shared defaults object:

```
UserDefaults.standard
```

Depending on the type of objects to retrieve, you can use one of the following methods to retrieve the setting:

- array(forKey:)
- bool(forKey:)
- data(forKey:)
- dictionary(forKey:)
- float(forKey:)
- integer(forKey:)
- object(forKey:)
- stringArray(forKey:)
- string(forKey:)
- double(forKey:)
- url(forKey:)

To save a setting in the defaults system, you set the value with a specific key. Here is an example:

```
UserDefaults.standard.set(true, forKey: "hasViewedWalkthrough")
```

Working with UserDefaults

If you understand how `UserDefaults` works, you may know how to use it to save the walkthrough status. After a user taps the *GET STARTED* button, we store a status in the user defaults to indicate that the user has gone through the tutorial. Open `WalkthroughViewController.swift` and update the `nextButtonTapped` method to the following:

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

    if let index = walkthroughPageViewController?.currentIndex {
        switch index {
        case 0...1:
            walkthroughPageViewController?.forwardPage()

        case 2:
            UserDefaults.standard.set(true, forKey: "hasViewedWalkthrough"
)
            dismiss(animated: true, completion: nil)

        default: break

        }
    }

    updateUI()
}
```

When the *GET STARTED* button is tapped, we add a `hasViewedWalkthrough` key to the user defaults and set it to `true` .

The same logic should be applied to `skipButtonTapped` :

```
@IBAction func skipButtonTapped(sender: UIButton) {
    UserDefaults.standard.set(true, forKey: "hasViewedWalkthrough")
    dismiss(animated: true, completion: nil)
}
```

Now open `RestaurantTableViewController.swift` and add a simple logic in the `viewDidAppear` method to determine if it should present the walkthrough view controller. Update the method like this:

```
override func viewDidAppear(_ animated: Bool) {

    if UserDefaults.standard.bool(forKey: "hasViewedWalkthrough") {
        return
    }

    let storyboard = UIStoryboard(name: "Onboarding", bundle: nil)
    if let walkthroughViewController = storyboard.instantiateViewControlle
r(withIdentifier: "WalkthroughViewController") as? WalkthroughViewControll
er {

        present(walkthroughViewController, animated: true, completion: nil
)
    }
}
```

The change should be very straightforward. We retrieve the `hasViewedWalkthrough` key from the user defaults and check its value. The walkthrough view controller will only be presented when the value is set to `false`.

That's it. Run the project to have a quick test. Now once you tap the *GET STARTED* button or the *Skip* button, the app will no longer display the walkthrough screens.

Summary

In this chapter, we covered the basics of `UIPageViewController`, container view, and `UserDefaults`. You should understand how to create walkthrough or tutorial screens when your users first use the app.

`UIPageViewController` is a very handy class for implementing walkthrough or tutorial screens. That said, the usage of `UIPageViewController` is unlimited. You can use it to display whatever information you like such as pages of web views.

So far we just use the *scroll* transition style. Don't you know that you can easily use `UIPageViewController` to build a simple book app? Simply change the transition style from *scroll* to *page curl* and see what you'll get.

For reference, you can download the complete Xcode project from http://www.appcoda.com/resources/swift53/FoodPinOnboarding.zip.

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.

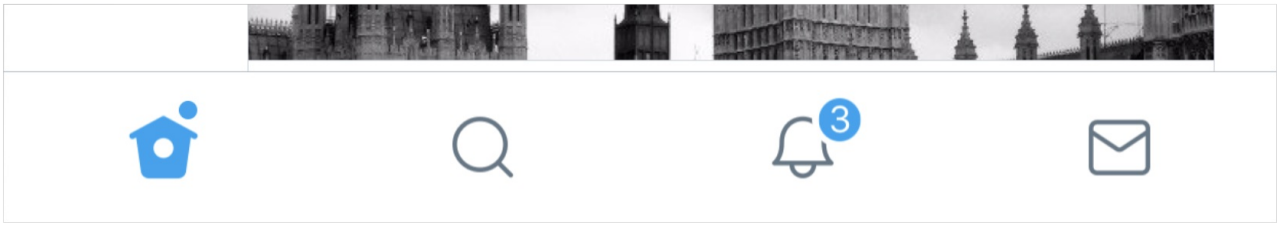


Figure 22-1. A tab bar in the Twitter app

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.

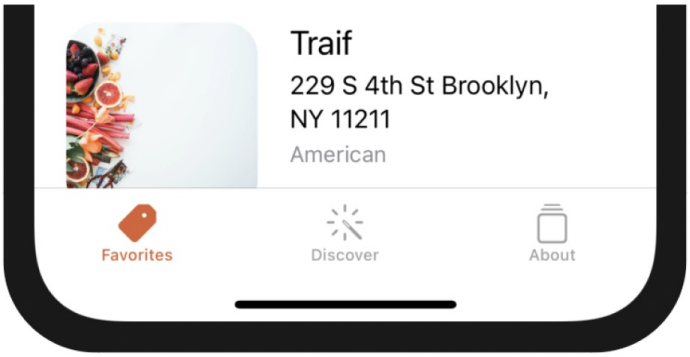
While navigation controllers let users navigate hierarchical content by managing a stack of view controllers, tab bars manage multiple view controllers that don't necessarily have a relation to one another. Normally a tab bar controller contains at least two tabs and you're allowed to add up to five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

Open the `Main.storyboard` file and select the Navigation Controller, which is the initial controller of the app. This is the controller we want to embed in a tab bar controller. Now go up to the Xcode menu, select Editor > Embed in > Tab Bar Controller.

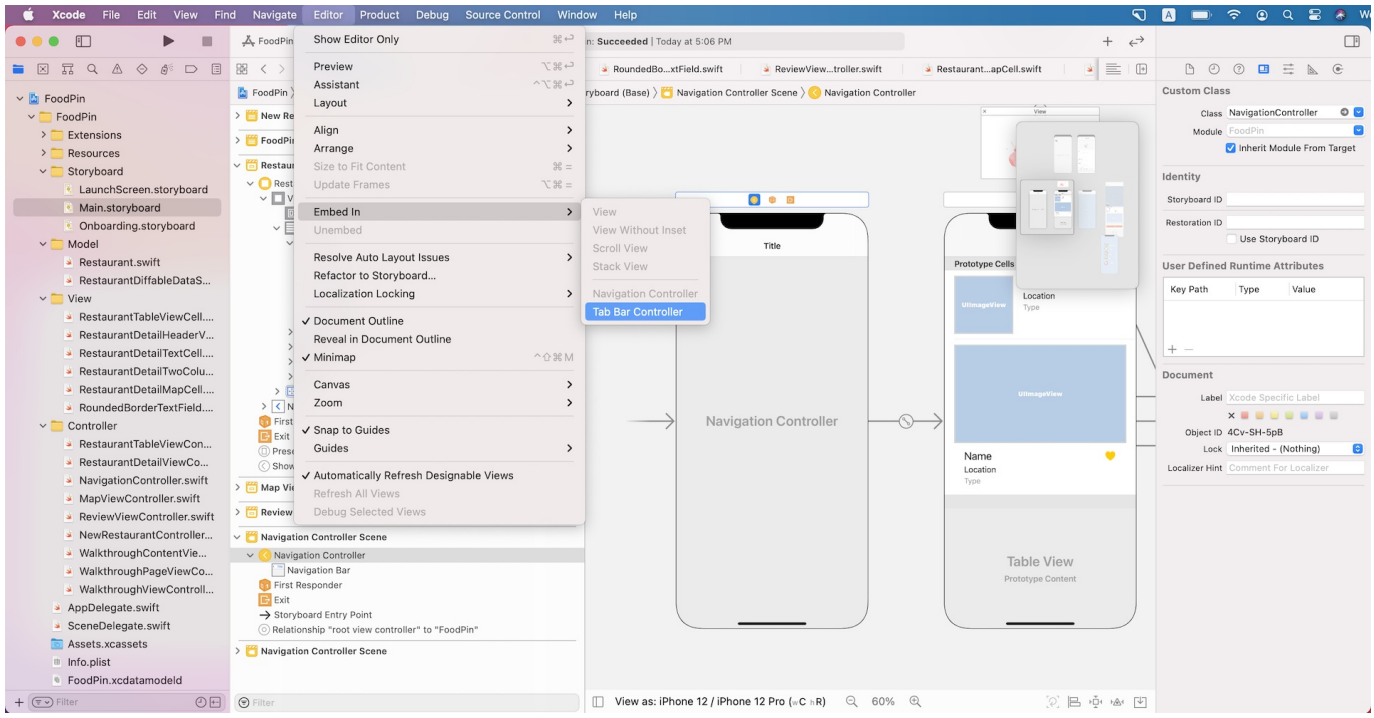


Figure 22-3. Embed the navigation controller in a tab bar controller

Interface Builder automatically embeds the navigation controller in a tab bar controller. Easy, right? You can then click the tab item of the navigation controller and change its properties in the Attribute inspector. You can use your own images to create a custom tab item. For simplicity, we just use the system items. Change the *System Item* option to *Favorites*. The tab bar item should be updated like the one shown in figure 22-4.

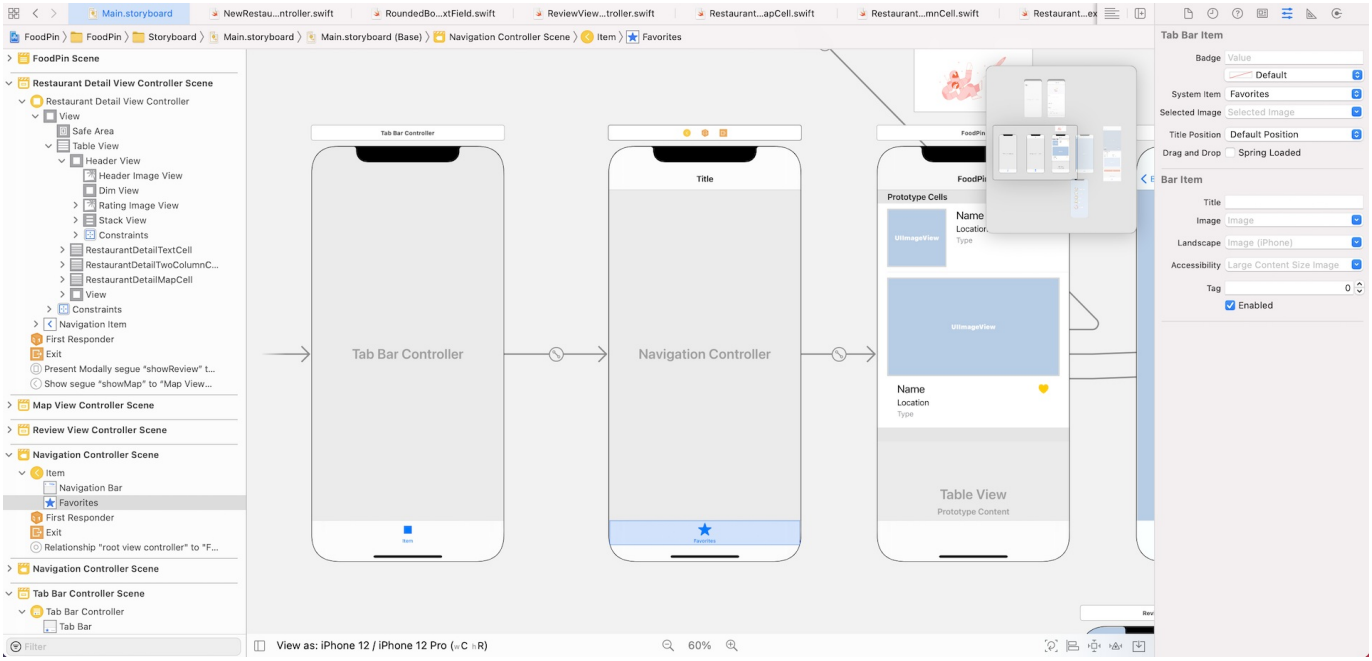


Figure 22-4. Setting the tab bar item

It's now ready to test the app. Hit the Run button and see how it looks. The app now has a tab bar with the *Favorites* tab. What's great is that you do not need to define any auto layout constraint for the tab bar. It works out of the box. Try to test the app on iPhone and iPad. The tab bar still works well.

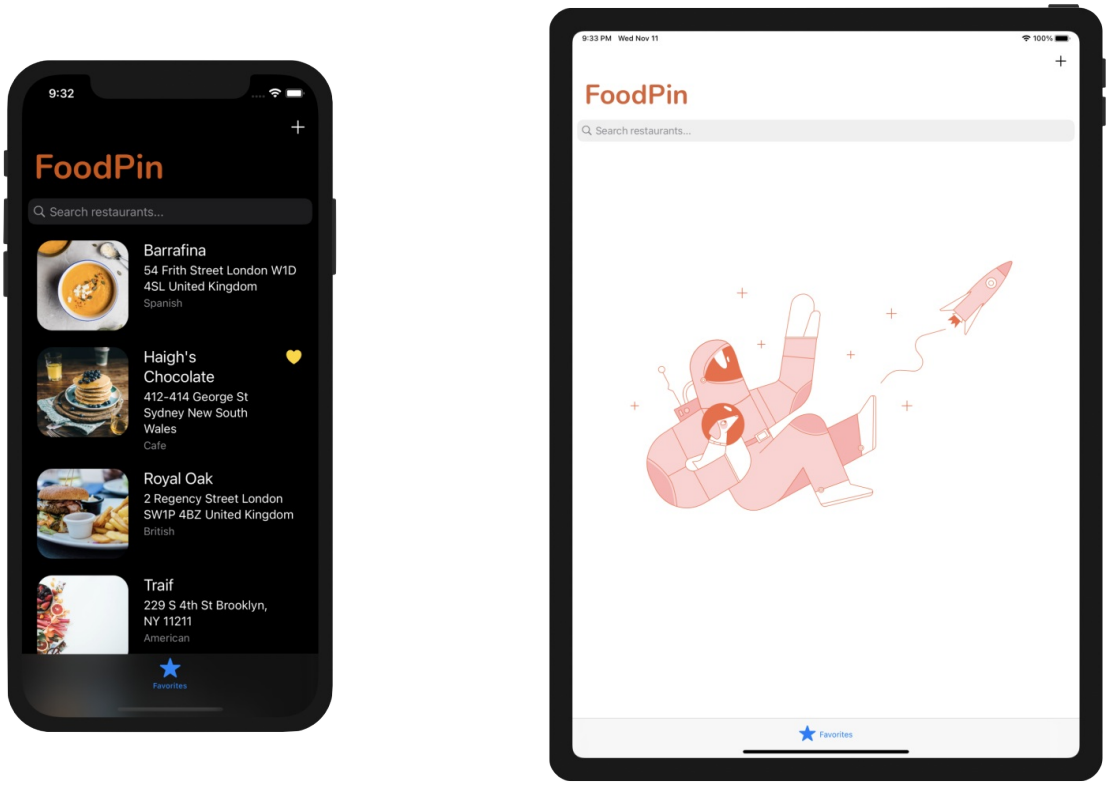


Figure 22-5. The FoodPin app now has a tab bar

Hide Tab Bar When Pushed

Everything we implemented in the earlier chapters is now put right inside a tab bar controller. If you select a restaurant, the app navigates to the detailed view and the tab bar is still there. You may want to hide the tab bar in the detailed view or any other views along the navigation hierarchy. iOS provides a simple way to hide a tab bar when a view controller is pushed on a navigation controller.

For example, to hide the tab bar in the detail view, you can select the detail view controller in storyboard and enable the *Hide Bottom Bar on Push* option in the Attribute inspector. The tab bar will be hidden when the detail view appears.

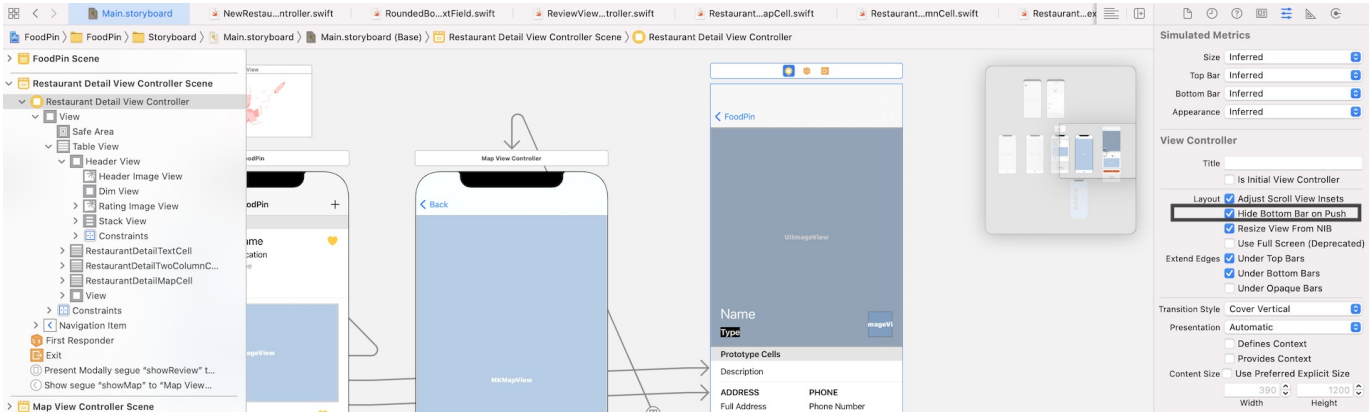


Figure 22-6. Enabling the Hide Bottom Bar on Push option for the detail view controller

Alternatively, if you want to do it in code, you can add the following line of code in the `prepare(for:)` method of the `RestaurantTableViewController` class:

```
destinationController.hidesBottomBarWhenPushed = true
```

Try to test the app again. The tab bar should now disappear when you navigate to the detailed view.

Adding New Tabs

There is no reason why an app needs a tab bar for displaying a single tab item. You use a tab bar interface to organize your app into distinct modes of operation. Each tab opens a specific feature. Apparently, there are at least two tab items in an app if you use a tab bar. In our FoodPin app, we're going to create two more tab items: *Discover* and *About*.

In the `Main.storyboard`, open the Object library and drag a Navigation Controller object into the storyboard. The default navigation controller is associated with a table view controller. Simply change the title of the navigation bar to *Discover*.

Meanwhile, keep everything unchanged. We will deal with the implementation of this tab in later chapters.

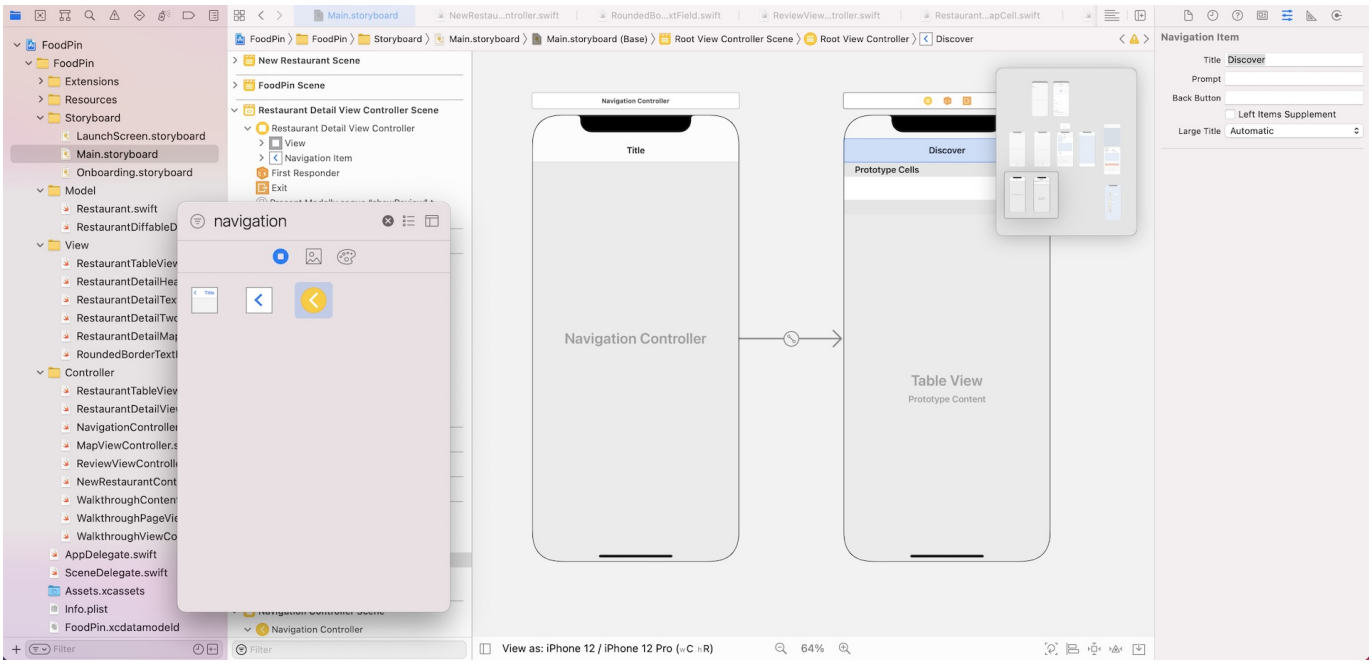


Figure 22-7. Navigation controller and table view controller in the Discover tab

Next, we want to make this set of controllers to be a part of the tab bar controller. All you have to do is establish a relationship between the new navigation controller and the existing tab bar controller. Press and hold the control key, drag from the tab bar controller to the new navigation controller. Release both buttons, and select the `View Controllers` option under Relationship Segue. This tells Xcode that the new navigation controller is a part of the tab bar controller.

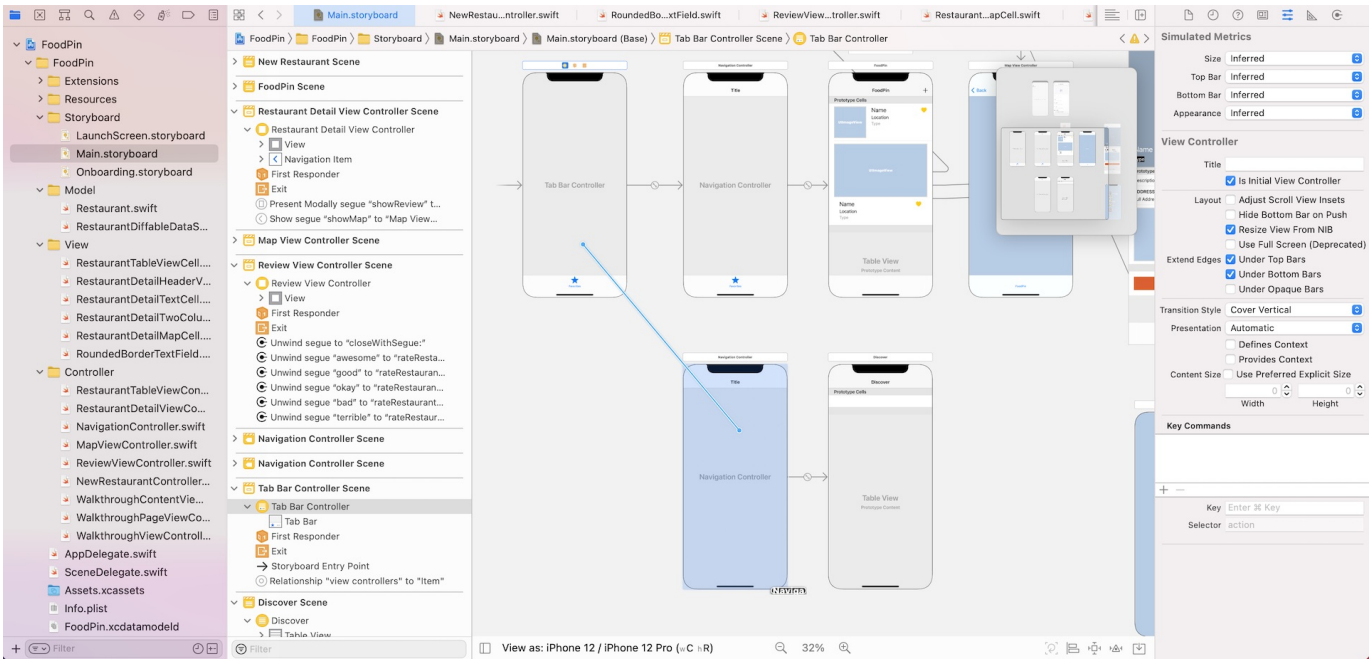


Figure 22-8. Link up the tab bar controller and the new navigation controller

As soon as the relationship is established, the tab bar controller automatically adds a new tab and associates it with the new navigation controller. Select the tab bar item of the navigation controller and change the tab bar item to *Recent* in the Attributes inspector. If you can't wait to test the app, run the project again. You should now see the new *Recents* tab.

> **Quick note:** For the sake of simplicity, we just use the system item. You may wonder how to use your own title and image for the tab item. No worries. I will show you in later sections of the chapter.

Repeat the same procedures to create another tab for *About*. Drag another navigation controller and name the title of the table view controller to *About*. Then establish a relationship with the tab bar controller. This will add another new tab in the tab bar controller. Change the tab bar item of the new navigation controller

to the system item *More*. The connections in your storyboard for the two new controllers should be similar to that in figure 22-9.

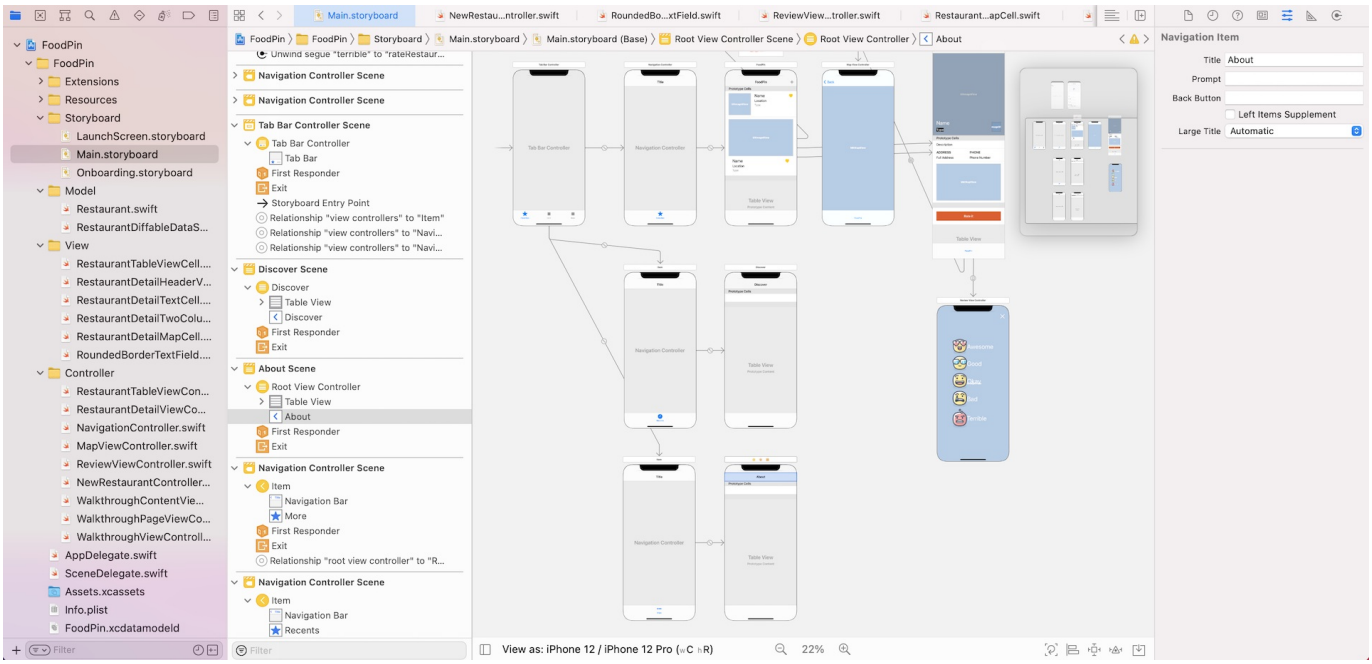


Figure 22-9. The tab bar controllers are now added with two new tab items

Now run the project to have a quick test. The FoodPin app should display a tab bar with three tab items - Favorites, Recent, and More. You're allowed to switch between tabs, though the Discover and About screens are blank.

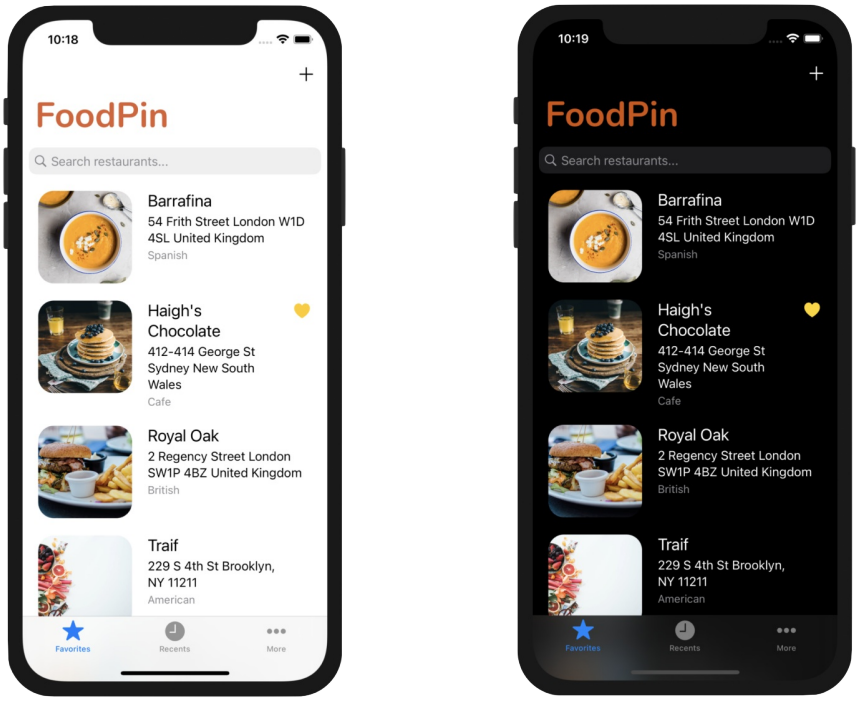


Figure 22-10. The FoodPin app now has three tabs

Customizing the Appearance of the Tab Bar

Up till now, we used some built-in tab items such as Recents and More. Similar to the navigator bar, you can customize the tab bar's appearance by using an appearance class. Starting from iOS 13, Apple introduced a new class named `UITabBarAppearance` for customizing the appearance of a tab bar. The usage of `UITabBarAppearance` is very similar to `UINavigationBarAppearance`. You first create an instance of `UITabBarAppearance` and configure its properties (e.g. background color). Then you assign the appearance object to the standard appearance of the tab bar. Here is the code:

```
let tabBarAppearance = UITabBarAppearance()
tabBarAppearance.configureWithDefaultBackground()
tabBarAppearance.backgroundColor = .darkGray
UITabBar.appearance().standardAppearance = tabBarAppearance
```

If you want to configure the tab bar with transparent background, you can replace the method call `.configureWithDefaultBackground` with `.configureWithTransparentBackground` like this:

```
tabBarAppearance.configureWithTransparentBackground()
```

To change the tint color of the tab bar item, you can set the `tintColor` property of the appearance API:

```
UITabBar.appearance().tintColor = UIColor.systemRed
```

For example, you can insert the following code snippet in the `application(_:didFinishLaunchingWithOptions:)` method of the `AppDelegate` class to customize the tab bar:

```
let tabBarAppearance = UITabBarAppearance()
tabBarAppearance.configureWithOpaqueBackground()

UITabBar.appearance().tintColor = UIColor(named: "NavigationBarTitle")
UITabBar.appearance().standardAppearance = tabBarAppearance
```

Run the project again. You should see a tab bar with a different tint color in both light and dark modes.

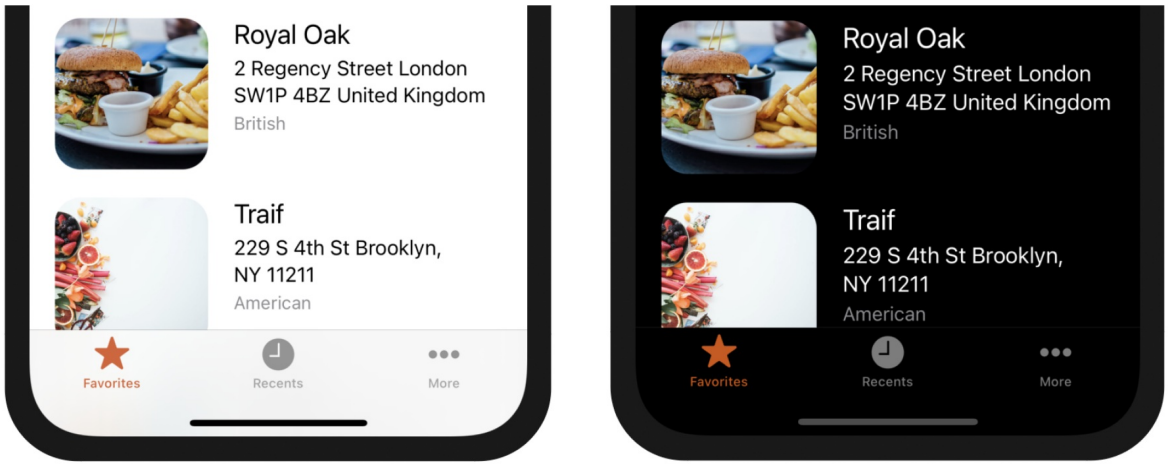


Figure 22-11. Sample tab bars after customization

You can refer to the official documentation of `UITabBarAppearance` to learn more about the customization properties (https://developer.apple.com/documentation/uikit/uitabbarappearance).

Changing the Tab Bar Item Image

Instead of using the system item, you're allowed to use your own image and title for the tab bar item. To change the tab bar item, select the item in the storyboard. In the Attribute inspector, change the System Item Option to `Custom` and set the Title/Image to your own values. For the *Favorite* tab, set its title to `Favorites` and its image to `tag.fill`, which is a built-in image from SF Symbols. For the *Recent* tab, set the title to `Discover` and image to `wand.and.rays`. For the *More* tab, change the title to `About` and image to `square.stack`.

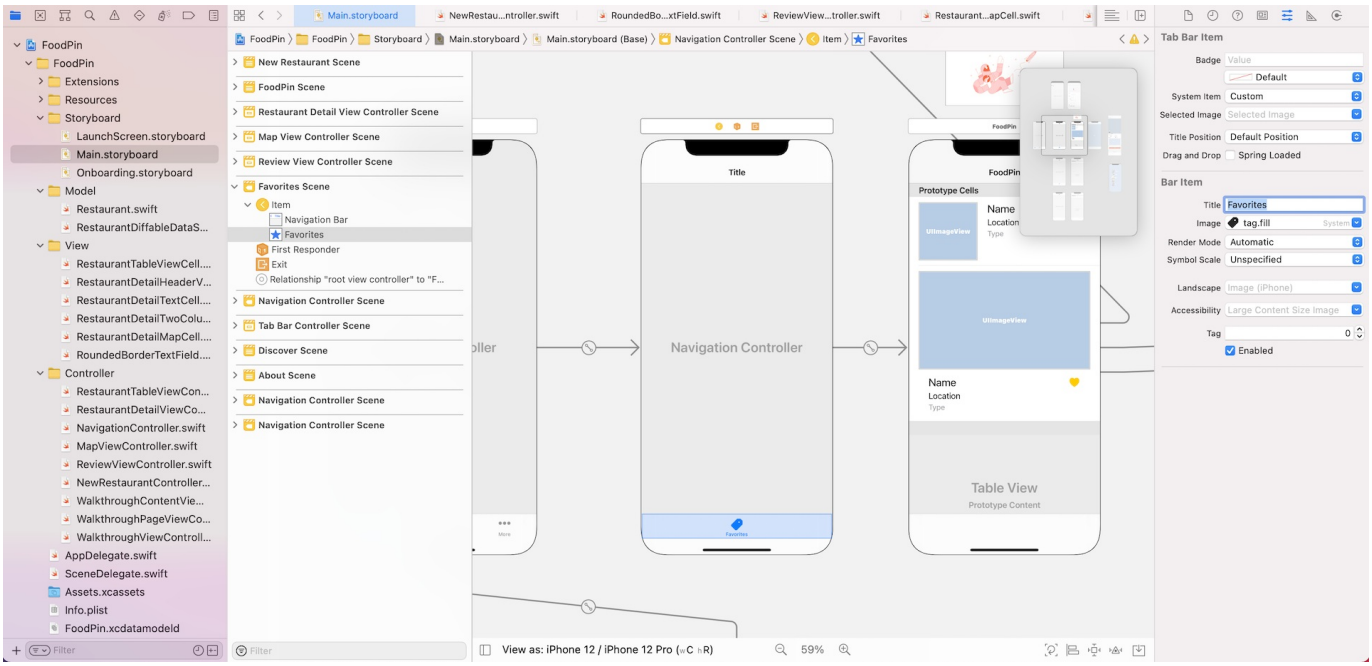


Figure 22-12. Setting the title and image of a tab bar item

If you test the app again, the tab bar should look like that in figure 22-13.

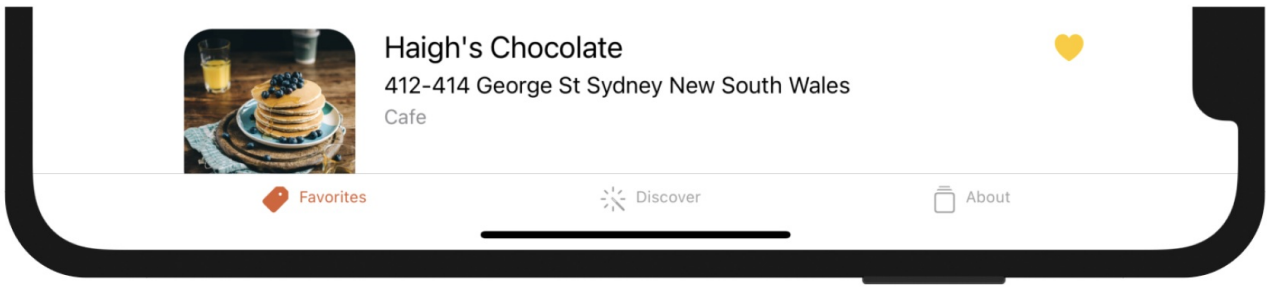


Figure 22-13. A customized tab bar in landscape orientation with your own images

Storyboard References

Storyboards is great for you to layout the app's UI visually. One complaint is that storyboards are less manageable as your project grows. In the previous chapter, you manually create a separate storyboard to organize the onboarding UI. Now I am going to show you another way to better manage your storyboards.

Since Xcode 7, Apple provided a feature called *storyboard references* to make storyboards more manageable. You can easily move certain UI components from one storyboard over to another.

Let's use the FoodPin project as an example to see how storyboard references work. The app now has three tabs: *Favorites*, *Discover* and *About*. Each of these tabs is implemented with its own UI navigation controller. The view controllers under each tab are mostly independent of the rest of the scenes in the storyboard.

Suppose that you're working on the project with two other members. Each team member is responsible for one of the tabs. It makes sense to pull each tab out to its own storyboard.

To do that, go to `Main.storyboard` and select those two controllers that belong the *About* tab. Then go up to the editor menu and choose *Refactor to Storyboard*....

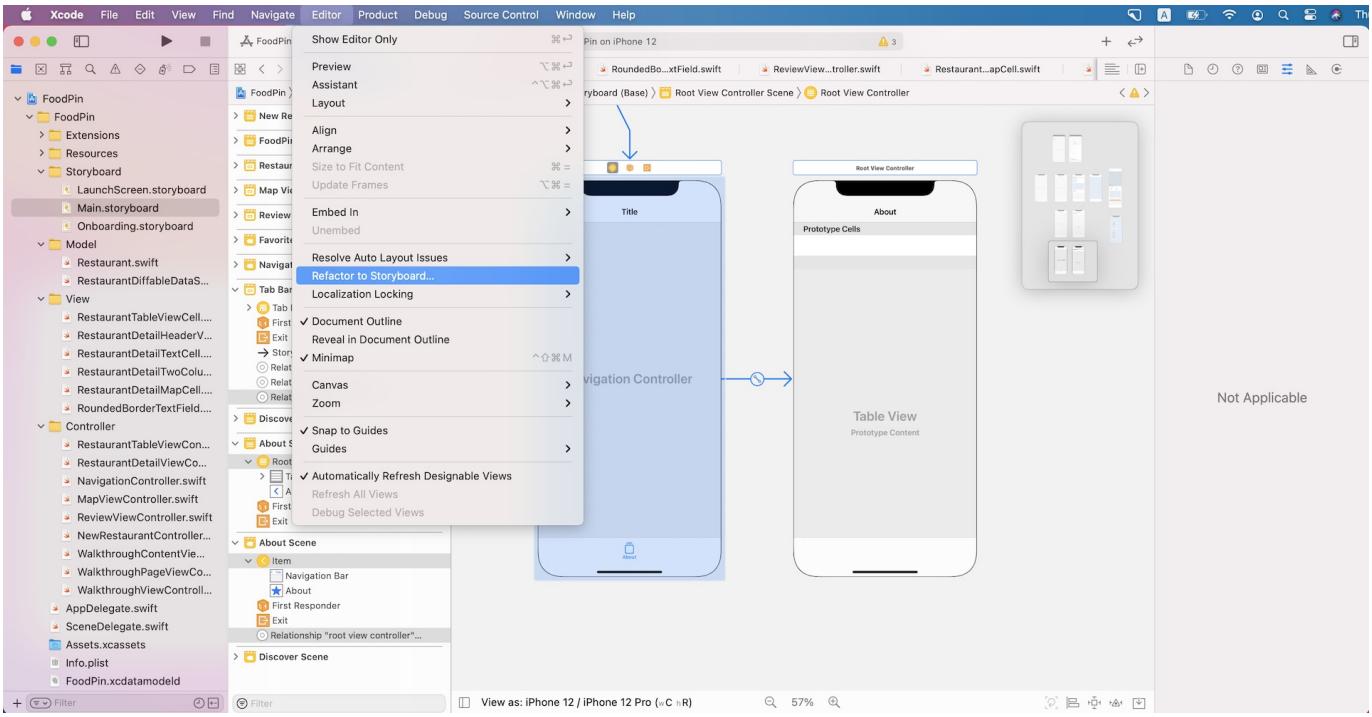


Figure 22-14. Refactor the About view controllers into a new storyboard

When prompted, give the new storyboard a name. Let's name it `About.storyboard`. Once you confirm the change, Xcode pulls out the selected view controllers, and place them in the `About.storyboard` file. In `Main.storyboard`, it shows a storyboard reference. If you double-click the storyboard reference, Xcode will take you to the corresponding storyboard and show you the view controllers that the main storyboard is referencing.

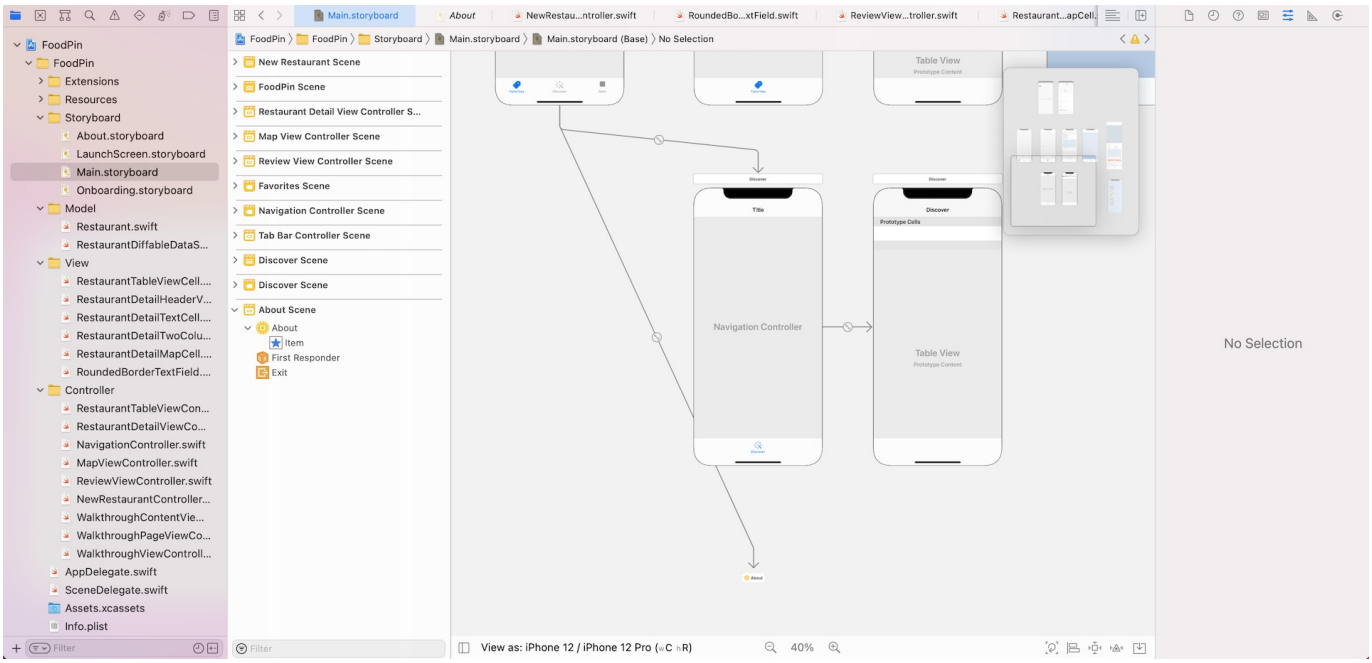


Figure 22-15. A storyboard reference

Repeat the same procedures to refactor the *Discover* tab, and name the new storyboard `Discover.storyboard`.

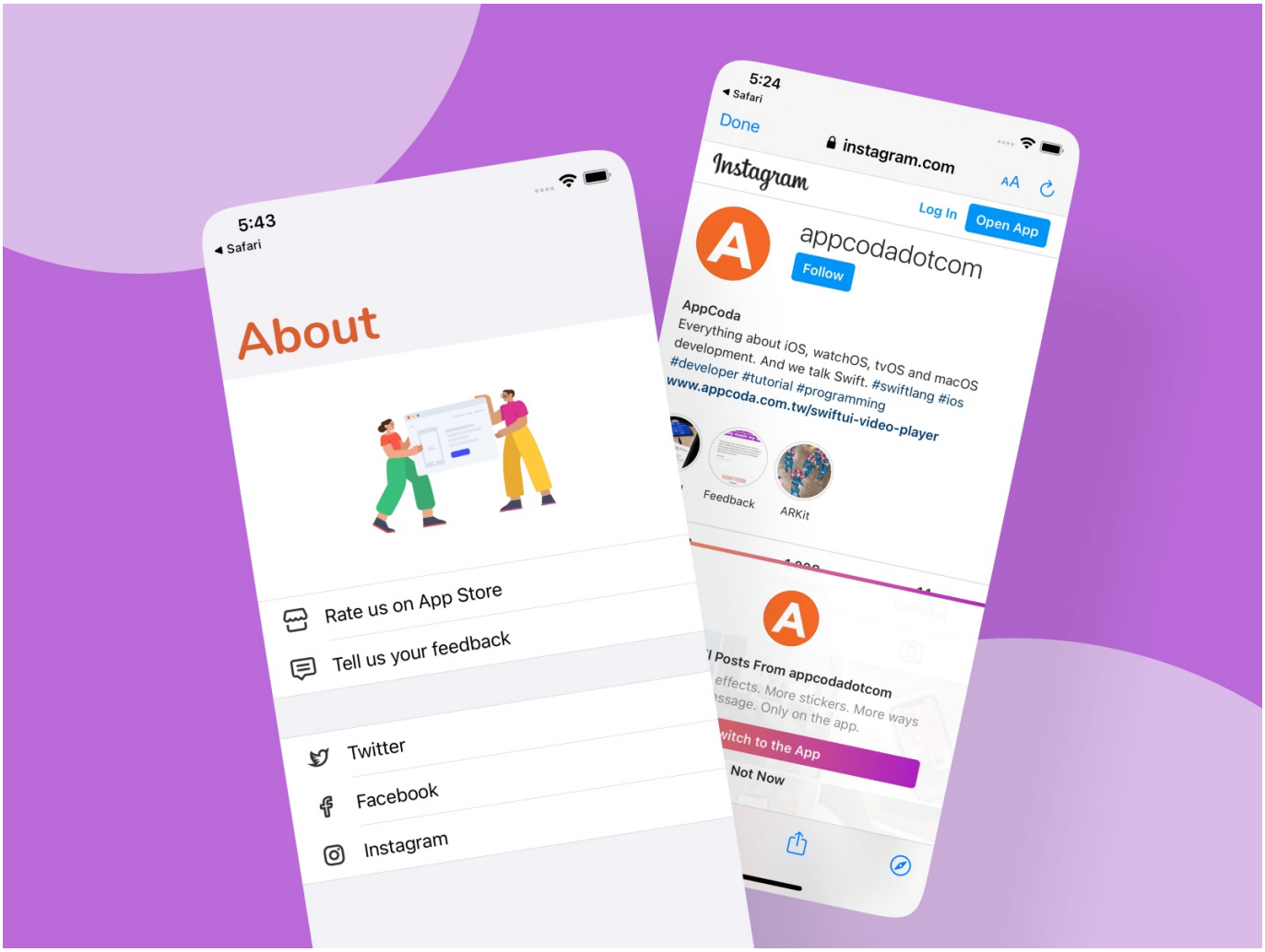
Summary

By now you should know how to create a tab bar controller and add new tab bar items. Interface Builder makes it very easy to embed any view controllers in a tab bar controller. Tab bars provide users a quick access to different features of your app.

We also went over a feature of Xcode called storyboard references. As your storyboard becomes more complex, you can break down a storyboard into multiple pieces to stay organized. This feature is especially useful for teams. If you're reluctant to use storyboards, it is time to think again and adopt storyboards in your app projects.

For reference, you can download the complete Xcode project from http://www.appcoda.com/resources/swift53/FoodPinTabbar.zip.

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.
- **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 `WKWebView` 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, `WKWebView` 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 `WKWebView`. 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.

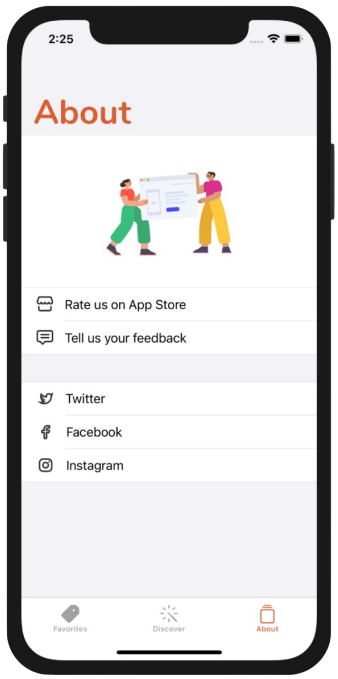


Figure 23-1. The About screen of the FoodPin app

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 and no longer available from iOS 13.

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.

Designing the About View

First, download the image pack from http://www.appcoda.com/resources/swift53/abouticons.zip and import the images into `Assets.xcasset`.

Open `About.storyboard` to switch to Interface Builder. Let's start to design the table view controller. Drag a View object from the object library to the header view of the table view (just like what we have done before). The View object should be placed right above the table view cell. You can reveal it in document outline. In the Size inspector, set the height to `200` points.

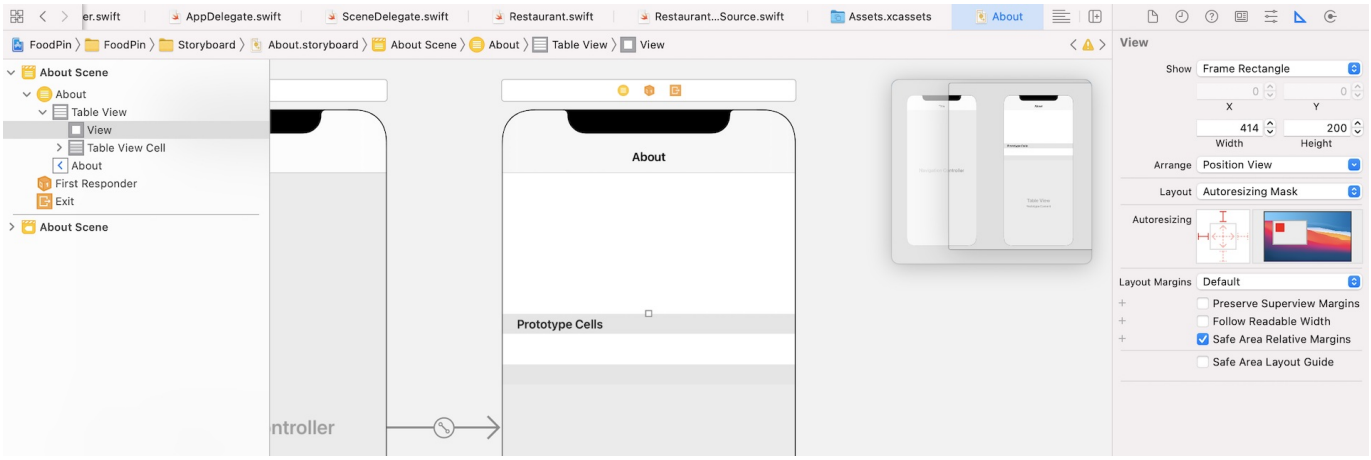


Figure 23-2. Adding a View object to the header view of the table view

Next, drag an image view and place it in the view. In the Attribute inspector, set the image to `about`. As always, we need to add a few layout constraints. Click the *Add New Constraints* button of the auto layout bar. Set the values of all sides to `0` and make sure you check the *Constrain to margins* option. Then click *Add 4 constraints* to confirm.

Now, select the table view cell. In the Attributes inspector, set the cell's identifier to `aboutcell` and style to `Basic`. Your design should look like figure 23-3.

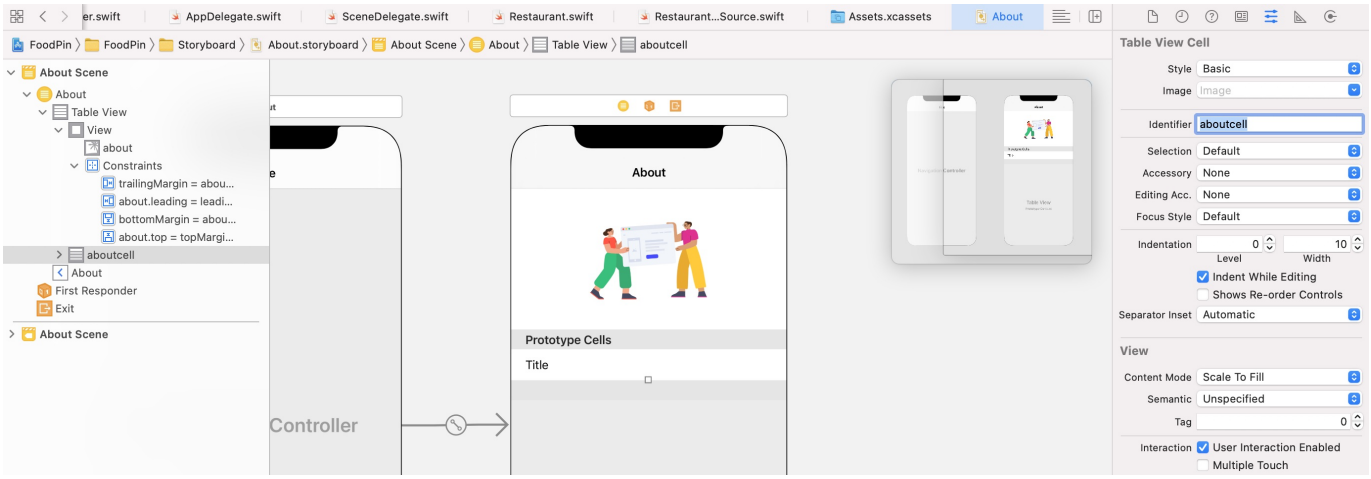


Figure 23-3. The layout of the About view

Creating a Custom Class for the About View Controller

As usual, we need a class to associate with the table view controller in `About.storyboard`. In the project navigator, right-click the `Controller` folder and select *New File*.... Choose the *Cocoa Touch Class* template. Name the class `AboutTableViewController` and set its subclass to `UITableViewController`.

In `About.storyboard`, select the table view controller and set the custom class to `AboutTableViewController` in the Identity inspector.

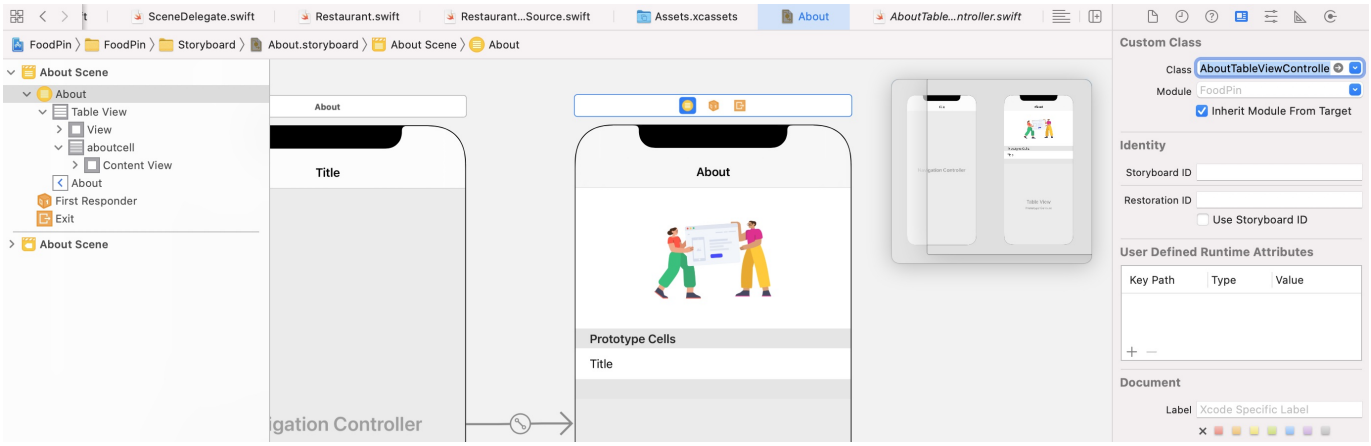


Figure 23-4. Setting the custom class

You should be very familiar with the implementation of table views using `UITableViewDiffableDataSource`. But this time, it is a little different. We are going to create two sections in the table view. Let's see how to do it.

First, declare an enum for the table section in `AboutTableViewController`. Since we have two sections, the enum has two cases:

```
enum Section {
    case feedback
    case followus
}
```

Next, define a structure for the link items. Insert the following code inside the `AboutTableViewController` class:

```
struct LinkItem: Hashable {
    var text: String
    var link: String
    var image: String
}
```

Next, declare an array variable to store the items for the two sections:

```
var sectionContent = [ [LinkItem(text: "Rate us on App Store", link: "http
s://www.apple.com/ios/app-store/", image: "store"),
                       LinkItem(text: "Tell us your feedback", link: "htt
p://www.appcoda.com/contact", image: "chat")
                       ],
                      [LinkItem(text: "Twitter", link: "https://twitter.c
om/appcodamobile", image: "twitter"),
                       LinkItem(text: "Facebook", link: "https://facebook
.com/appcodamobile", image: "facebook"),
                       LinkItem(text: "Instagram", link: "https://www.ins
tagram.com/appcodadotcom", image: "instagram")]
                       ]
```

If you go back to figure 23-1 to take a look at the About screen again, each row of data has an image and a description. When the user taps any of the rows, we will load the corresponding hyperlink of the selected item. Therefore, for each row of record, it actually has the following information:

- An image
- A description
- A link

How do you store these table data? There are many ways to do that. Try to think about the solution.

Probably you will store the data like this:

```
var images = [ ["store", "chat"], ["twitter", "facebook", "instagram"] ]
var text = [ ["Rate us on App Store", "Tell us your feedback"], ["Twitter"
, "Facebook", "Instagram"] ]
var links = [ ["https://www.apple.com/ios/app-store/", "http://www.appcoda
.com/contact"], ["https://twitter.com/appcodamobile", "https://facebook.co
m/appcodamobile", "https://www.instagram.com/appcodadotcom"] ]
```

There is nothing wrong with the code above. You can use three separate arrays to store the data. It will work. However, it is not easy to reveal the relationship of the data if you store the data like that.

A better way to do that is to group related values together by using struct. This is why we define the `LinkItem` structure.

The value of `sectionContent` may look a bit complex. But actually the code is exactly the same as follows:

```
let section0 = [LinkItem(text: "Rate us on App Store", link: "https://www.
apple.com/ios/app-store/", image: "store"), LinkItem(text: "Tell us your f
eedback", link: "http://www.appcoda.com/contact", image: "chat")]

let section1 = [LinkItem(text: "Twitter", link: "https://twitter.com/appco
damobile", image: "twitter"),
                        LinkItem(text: "Facebook", link: "https://facebook
.com/appcodamobile", image: "facebook"), LinkItem(text: "Instagram", link:
"https://www.instagram.com/appcodadotcom", image: "instagram")]

var sectionContent = [ section0, section1 ]
```

This is how I come up with this `sectionContent` variable. Let's continue to implement the table view.

First, delete the following methods generated by Xcode:

```
override func numberOfSections(in tableView: UITableView) -> Int {
    // #warning Incomplete implementation, return the number of sections
    return 0
}

override func tableView(_ tableView: UITableView, numberOfRowsInSection se
ction: Int) -> Int {
    // #warning Incomplete implementation, return the number of rows
    return 0
}
```

We do not need them because we are going to use the diffable data source to load the data. First, create a method named `configureDataSource()` like this:

```
func configureDataSource() -> UITableViewDiffableDataSource<Section, LinkI
tem> {

    let cellIdentifier = "aboutcell"

    let dataSource = UITableViewDiffableDataSource<Section, LinkItem>(tabl
eView: tableView) { (tableView, indexPath, linkItem) -> UITableViewCell? in


        let cell = tableView.dequeueReusableCell(withIdentifier: cellIdent
ifier, for: indexPath)

        cell.textLabel?.text = linkItem.text
        cell.imageView?.image = UIImage(named: linkItem.image)

        return cell
    }

    return dataSource
}
```

This method returns a diffable data source for the table view and describe how the cell data should be presented. Next, create another method for the snapshot:

```
func updateSnapshot() {

    // Create a snapshot and populate the data
    var snapshot = NSDiffableDataSourceSnapshot<Section, LinkItem>()
    snapshot.appendSections([.feedback, .followus])
    snapshot.appendItems(sectionContent[0], toSection: .feedback)
    snapshot.appendItems(sectionContent[1], toSection: .followus)

    dataSource.apply(snapshot, animatingDifferences: false)
}
```

Assuming you understand how the diffable data source works, the code above should be very familiar to you. To support a table view with multiple sections, all you need is to do is to append all the sections by calling `appendSections`. And, you specify the section items by using the `appendItems` method.

Next, declare a `dataSource` variable in `AboutTableViewController`:

```
lazy var dataSource = configureDataSource()
```

Lastly, insert the following code in the `viewDidLoad()` method:

```
tableView.dataSource = dataSource
updateSnapshot()
```

Now run the app and take a quick look. You have already created the user interface of the About screen.

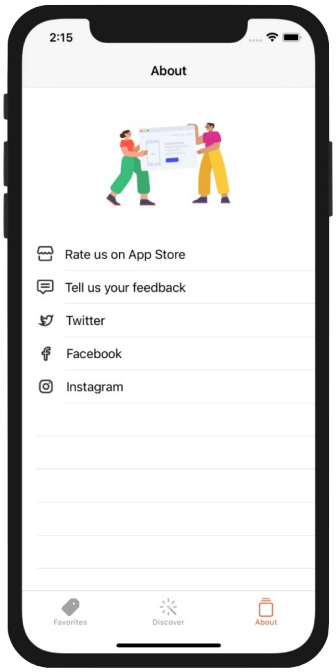


Figure 23-5. The About screen using the plain style

There are a couple of things I want to tweak:

1. I prefer to use a large title in the navigation bar.
2. I want to break the items into two sections.

For the first tweak, you can update the `viewDidLoad()` method like this:

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

    // Use large title for the navigation bar
    navigationController?.navigationBar.prefersLargeTitles = true

    // Customize the navigation bar appearance
    if let appearance = navigationController?.navigationBar.standardAppear
ance {

        appearance.configureWithTransparentBackground()

        if let customFont = UIFont(name: "Nunito-Bold", size: 45.0) {

            appearance.titleTextAttributes = [.foregroundColor: UIColor(na
med: "NavigationBarTitle")!]
            appearance.largeTitleTextAttributes = [.foregroundColor: UICol
or(named: "NavigationBarTitle")!, .font: customFont]
        }

        navigationController?.navigationBar.standardAppearance = appearanc
e
        navigationController?.navigationBar.compactAppearance = appearance
        navigationController?.navigationBar.scrollEdgeAppearance = appeara
nce
    }

    // Load table data
    tableView.dataSource = dataSource
    updateSnapshot()
}
```

You should be familiar with the code for navigation bar customization, so I would not go into details. For the second tweak, you can switch over to `About.storyboard`. Select the *Table View* of the *About* screen and go to the Attributes inspector. Change the *Style* option to `Grouped` or `Inset Grouped`.

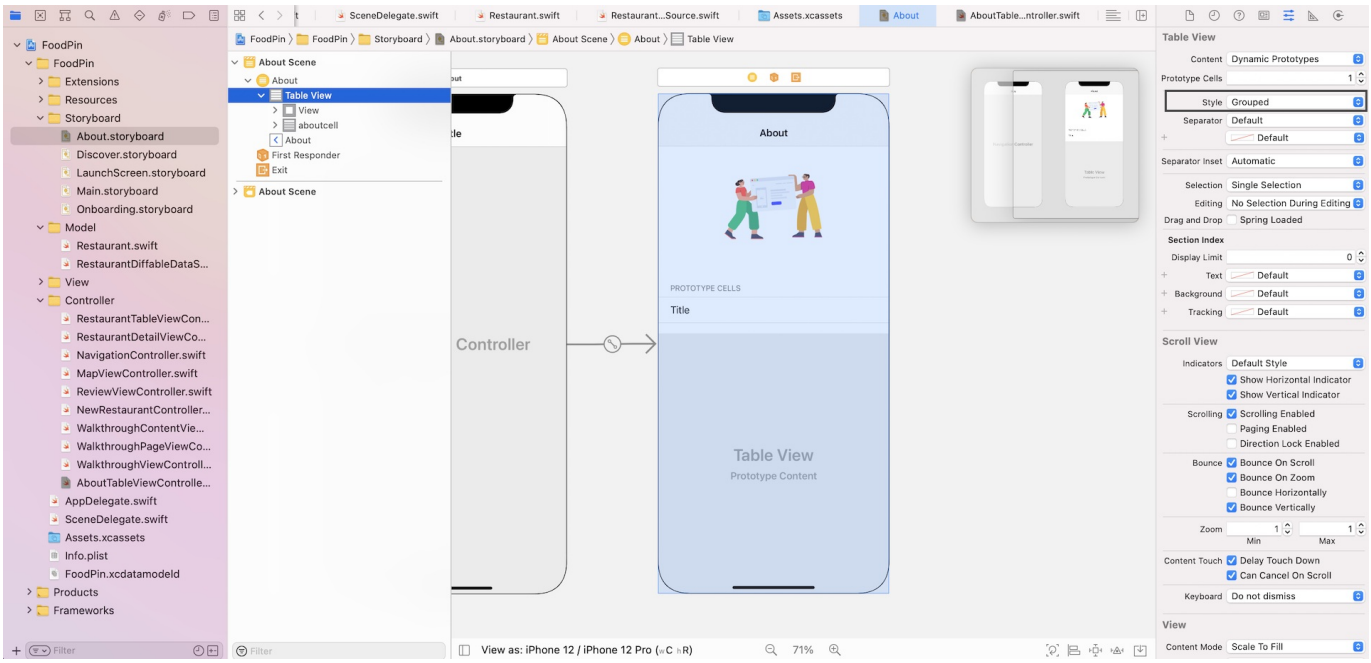


Figure 23-6. Editing the table view's style

Run the app again and you should see the large title in the About screen. And you will see two separate sections in the table view.

Opening Web Content in Mobile Safari

Now that we've prepared the user interface of the About view, let's start to explore the first option of loading web content. When the *Rate us on App Store* option is selected, the app will switch to mobile Safari and load the content.

To open a link in the Safari browser, you just need to call the `open(_:options:completionHandler:)` method of `UIApplication` with a specific URL:

When the method is invoked, your user will leave the current application and switch over to Safari to load the web content. To handle table cell selection, as you know, we need to override the `tableView(_didSelectRowAtIndexPath:)` method of `AboutTableViewController`. Insert the following code snippet:

```
override func tableView(_ tableView: UITableView, didSelectRowAt indexPath
: IndexPath) {

    // Get the selected link item
    guard let linkItem = self.dataSource.itemIdentifier(for: indexPath) el
se {
        return
    }

    if let url = URL(string: linkItem.link) {
        UIApplication.shared.open(url)
    }

    tableView.deselectRow(at: indexPath, animated: false)
}
```

We first find out the link of the selected row and then call `UIApplication.shared.open(url)` to open the Safari browser. Now, run the app in the simulator. In the About tab, tapping any of the items will switch over to Mobile Safari to display the web content.

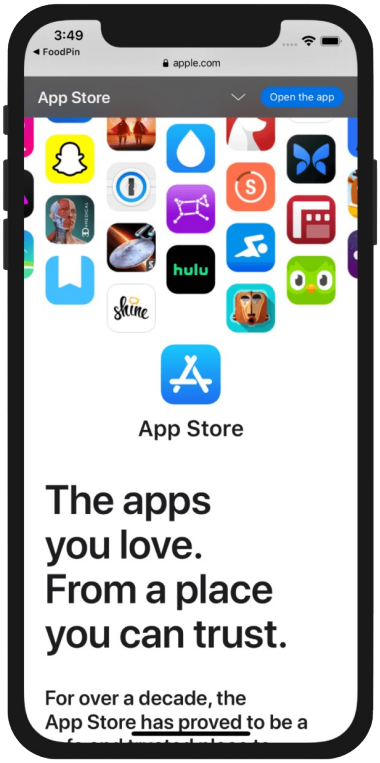


Figure 23-7. Displaying web content in Safari

Loading Web Content Using WKWebView

It's very easy to open a web page in Mobile Safari. To load web content using `WKWebView`, it will take you a few more steps. The `WKWebView` class is a successor of `UIWebView`, which is capable of loading remote web content. If you bundle an HTML file in your app, you can also use the class to load the web page locally.

You can no longer use `UIWebView` as Apple has deprecated the API and rejects any apps which utilize `UIWebView`. You now have to use `WKWebView`, which outperforms its predecessor in terms of page loading speed.

Practically, you need to create an `URL` object, followed by a `URLRequest` object, and then load the request by calling the `load` method of `WKWebView`. Here is a sample code snippet:

```
if let url = URL(string: "http://www.appcoda.com/contact") {
    let request = URLRequest(url: url)
    webView.load(request)
}
```

The above code instructs the web view (i.e. the instance of `WKWebView`) to load web content remotely. As mentioned before, you're allowed to load a local web page. Let's say, there is an HTML file bundled in your app. You can initialize an `URL` object using the `fileURLWithPath` parameter:

```
let url = URL(fileURLWithPath: "about.html")
let request = URLRequest(url: url)
webView.load(request)
```

`WKWebView` will then load the web page locally, which works even without an Internet connection.

With a basic understanding of the `WKWebView` class, let's start to implement it in the app. In `About.storyboard`, drag a View Controller object from the Object library to the storyboard. Then drag a WebKit View object (i.e. WKWebView) to the view controller. Resize it to make it take up the whole view. Next, click the *Add New Constraints* button to add some spacing constraints for the top, left, bottom and right sides. Your view controller should look like figure 23-8.

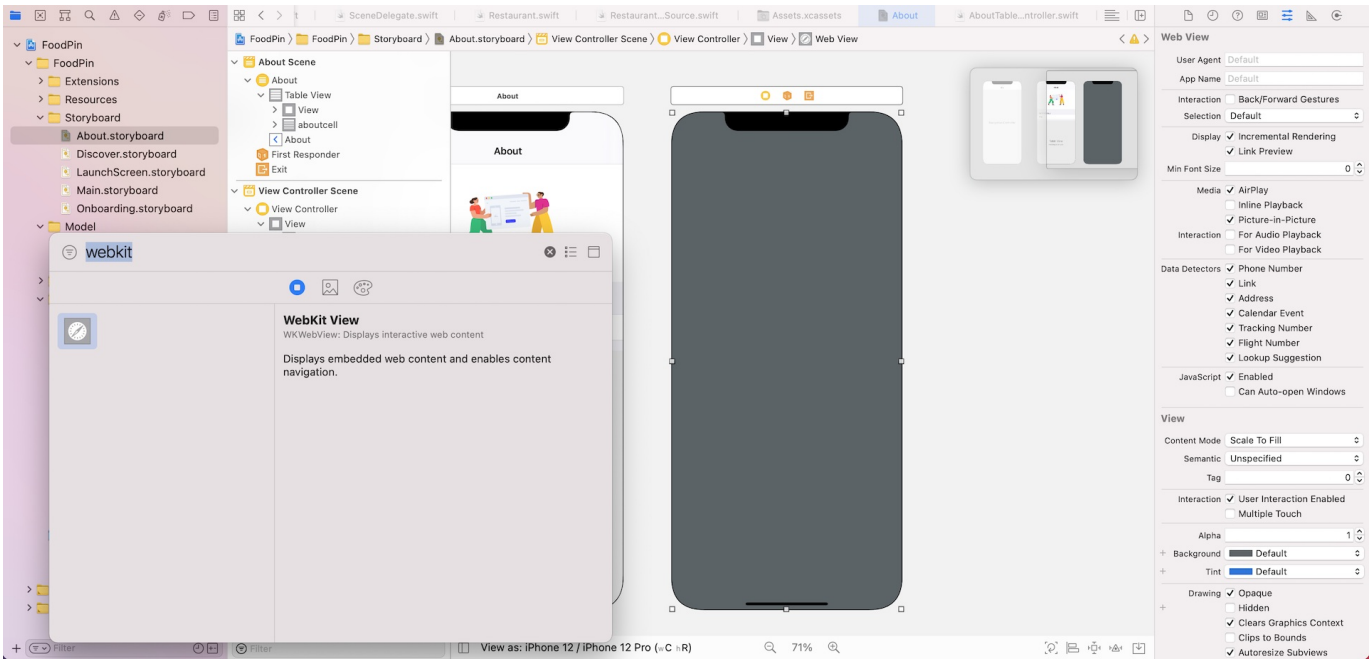


Figure 23-8. Embedding a WebKit View in a view controller

When any of the items is tapped, the app will transit to the new view controller to display the web content. To do that, we have to define a segue between the About view controller and the web view controller. Drag from the view controller button in the scene dock to the new view controller. Release the buttons and select *Present Modally* as the segue type.

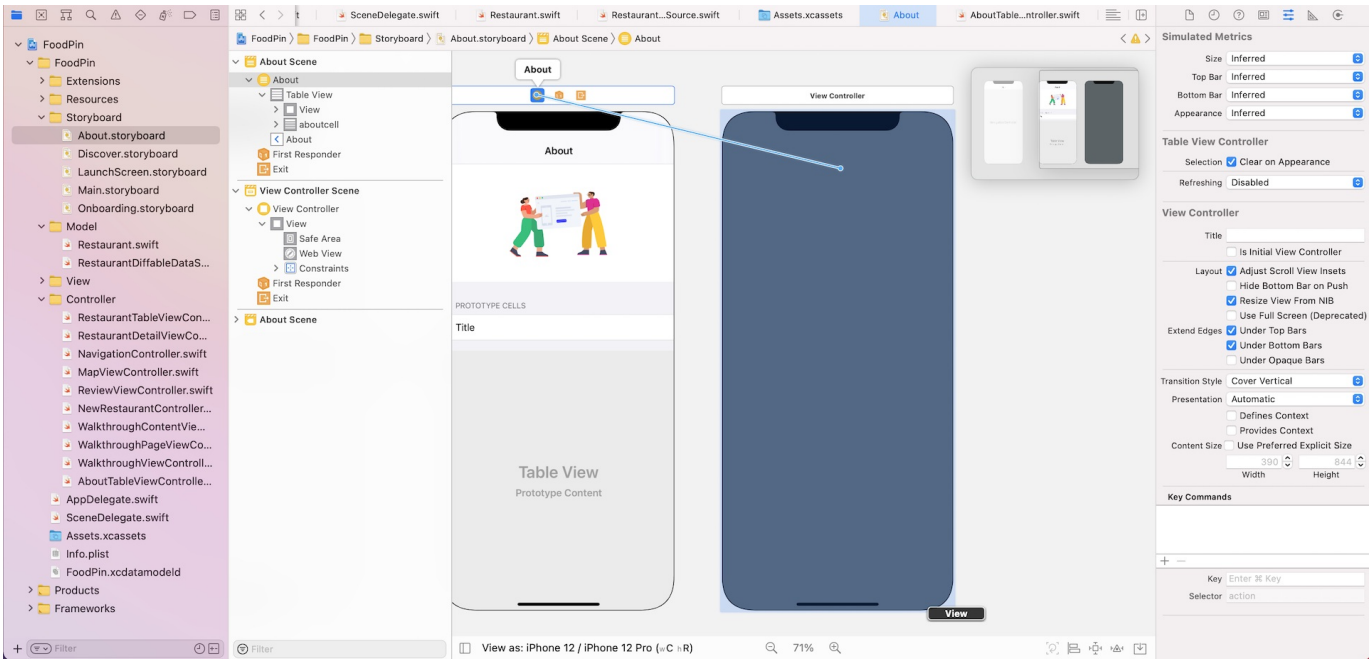


Figure 23-9. Creating a segue between the About controller and the view controller

This creates a segue between the two controllers. Select the segue and set the identifier to `showWebView` in the Attributes inspector.

As usual, the next step is to create a custom class for the web view controller. Right-click the `Controller` folder in the project navigator, and select *New File...*. Choose the *Cocoa Touch Class* template and name the class `WebViewController`. Set it as a subclass of `UIViewController`. Click `Next` and save the file.

In the `WebViewController.swift` file, declare an outlet variable for the web view and a property:

```
@IBOutlet var webView: WKWebView!

var targetURL = ""
```

The `targetURL` variable is used for storing the target URL, passed by `AboutTableViewController`. Since `WKWebView` is in the `WebKit` framework, you have to insert the following import statement at the very beginning of the file:

```
import WebKit
```

For the `viewDidLoad` method, update it with the following line of code to load a web page:

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

    if let url = URL(string: targetURL) {
        let request = URLRequest(url: url)
        webView.load(request)
    }
}
```

Now go back to the `AboutTableViewController` class and update the `tableView(_didSelectRowAtIndexPath:)` method like this:

```
override func tableView(_ tableView: UITableView, didSelectRowAt indexPath
: IndexPath) {

    performSegue(withIdentifier: "showWebView", sender: self)

    tableView.deselectRow(at: indexPath, animated: false)
}
```

When the cell is selected, we simply call `performSegue(withIdentifier:sender:)` method to trigger the `showWebView` segue and transit to the web view controller.

Additionally, create the `prepare(for:sender:)` method:

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {

    if segue.identifier == "showWebView" {
        if let destinationController = segue.destination as? WebViewContro
ller,
           let indexPath = tableView.indexPathForSelectedRow,
           let linkItem = self.dataSource.itemIdentifier(for: indexPath) {

            destinationController.targetURL = linkItem.link
        }
    }
}
```

The purpose of this method is to find out the link of the selected item and pass it to the web view controller by setting the `targetURL` property. The `if let` syntax of the code above may be new to you. In Swift, you can unwrap multiple optionals using a single `if let` block. It's `if` followed by a series of `let` statements, each separated by a comma.

Lastly, switch over to `About.storyboard`. Select the view controller that contains the web view. In the Identify inspector, set its custom class to `WebViewController`. Also, establish a connection between the `webView` outlet and the WebKit view object in Interface Builder.

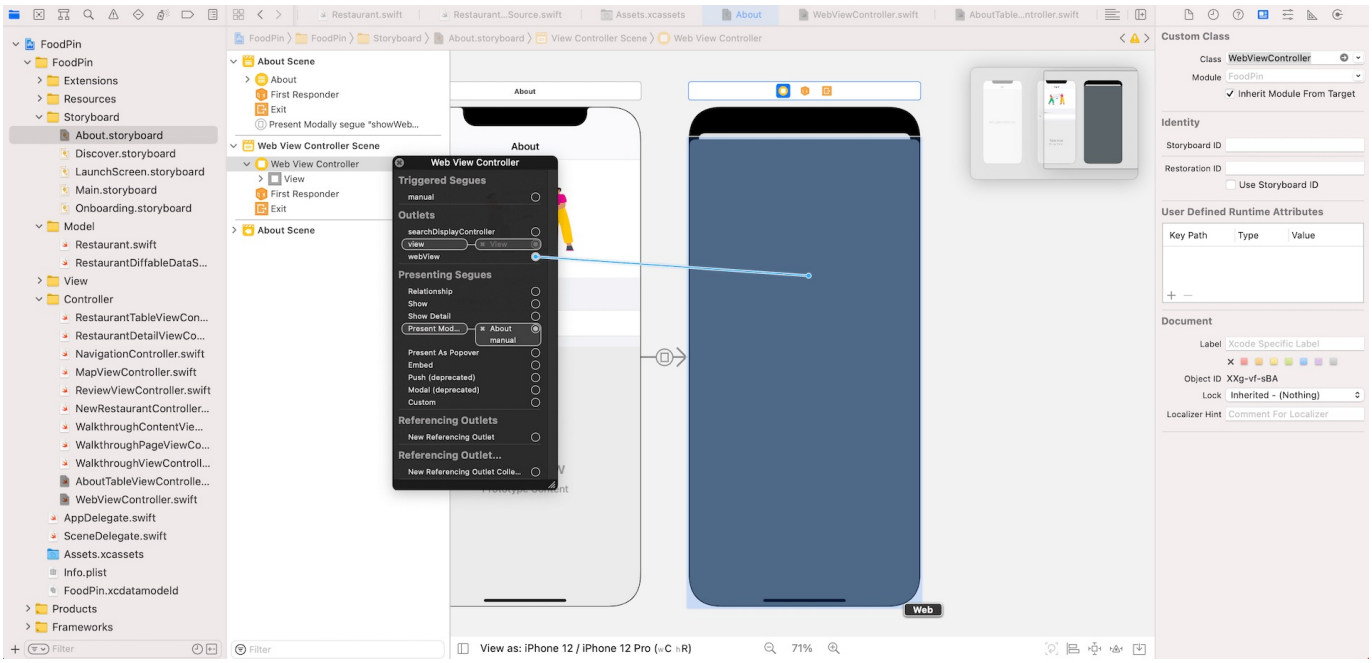


Figure 23-10. Connecting the outlet with the WebKit view object

Now run the project to test out the changes. You should be able to load the web view controller for most of the links. However, if you tap the *Tell us your feedback* item, it shows a blank view.

What's the problem? It's due to the non-HTTPS URL. Starting from iOS 9, Apple introduced a feature called App Transport Security or ATS for short. The purpose of this feature is to improve the security of connections between an app and web services by enforcing some of the best practices. One of them is the use of secure connections. With ATS, all network requests should now be sent over HTTPS. If you make a network connection using HTTP, ATS will block the request.

To resolve the issue, you should always load a network request over HTTPS instead of HTTP. That is what Apple wants you to do. And, this is why all our links work except the one for the *Tell us your feedback* item.

So, how can you work around this restriction if the websites that you're talking to is out of your control and that doesn't support HTTPS?

Apple provides an option for developers to disable ATS restrictions for content loaded inside of web views. To do so, you need to set the value of the specific key named `NSAllowsArbitraryLoadsInWebContent` in your app's `Info.plist`.

The `Info.plist` file contains essential configuration information for your app. To edit the file, select `Info.plist` in the project navigator to display the content in a property list editor. To add a new key, right-click *Information Property List* and select *Add Row*. For the *key* column, enter `App Transport Security Settings`. Then follow the procedures in figure 23-10 to add the `Allow Arbitrary Loads in Web Content` key.

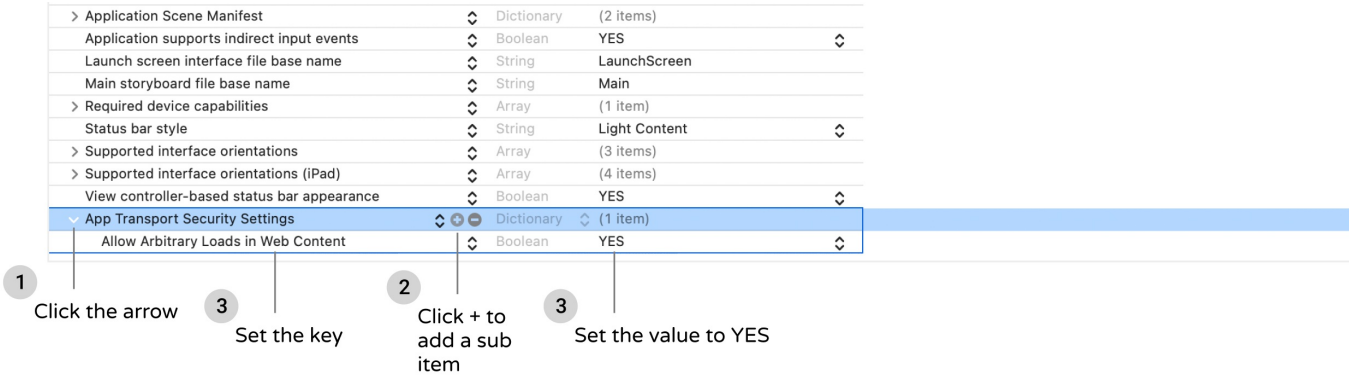


Figure 23-11. Adding the Allow Arbitrary Loads in Web Content key

By setting the key to `YES`, you explicitly disable App Transport Security. Now run the app again. It should be able to load the web page.

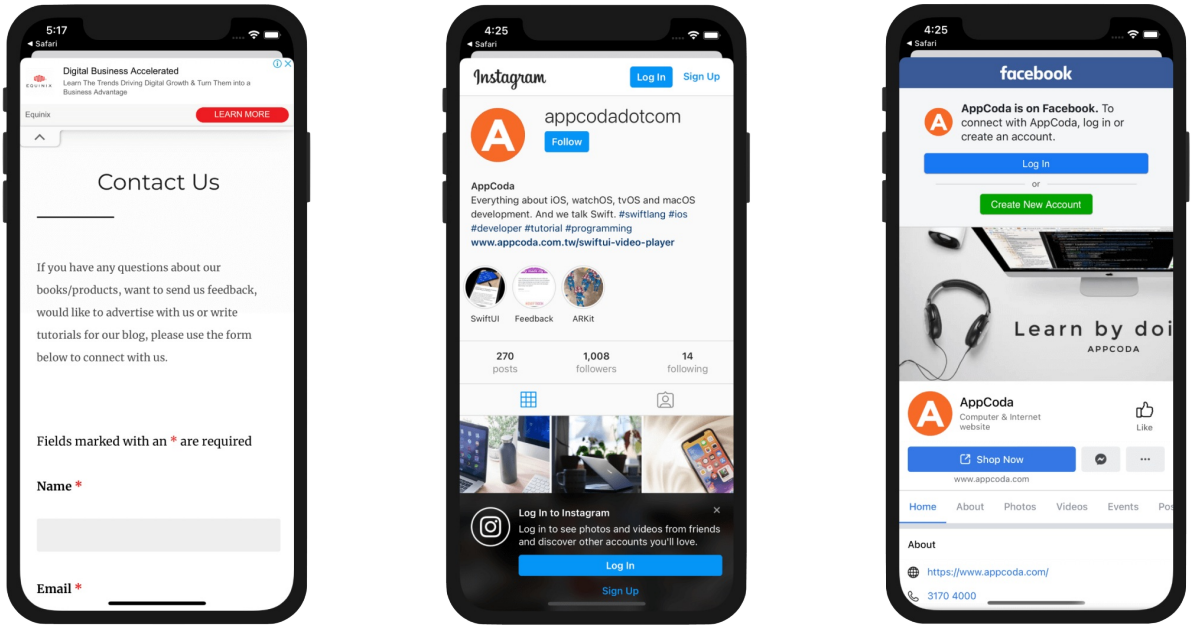


Figure 23-12. Loading a web page using WKWebView

Loading Web Content Using SFSafariViewController

Finally, let's talk about `SFSafariViewController`. As discussed earlier, this new controller allows developers to embed a Safari browser in your app. It shares many great features of Safari such as Safari AutoFill and Safari Reader. According to Apple, if your app displays web content in a standard browser, it is recommended to use `SFSafariViewController`.

To embed `SFSafariViewController` in your app, all you need is a couple line of code:

```
let safariController = SFSafariViewController(url: url)
present(safariController, animated: true, completion: nil)
```

You first create the `SFSafariViewController` object with a specified URL. Optionally, you can set `entersReaderIfAvailable` to `true` to enable Safari Reader. The next step is to present the controller by calling the `present(_:animated:completion:)` method.

Now open the `AboutTableViewController.swift` file. We'll update the code so that the app displays the social links (Twitter/Facebook/Instagram) using `SFSafariViewController`. First, insert the following import statement at the very beginning of the file:

```
import SafariServices
```

You have to import the `SafariServices` framework before using `SFSafariViewController`. Next, create a new method called `openWithSafariViewController` for loading the web content with `SFSafariViewController` like this:

```
func openWithSafariViewController(indexPath: IndexPath) {
    guard let linkItem = self.dataSource.itemIdentifier(for: indexPath) el
se {
        return
    }

    if let url = URL(string: linkItem.link) {
        let safariController = SFSafariViewController(url: url)
        present(safariController, animated: true, completion: nil)
    }
}
```

This function accepts an index path of the selected item. We then retrieve the link item from the index path and display the web content with `SFSafariViewController`.

Lastly, update `tableView(_:didSelectRowAtIndexPath:)` to the following:

```
override func tableView(_ tableView: UITableView, didSelectRowAt indexPath
: IndexPath) {

    switch indexPath.section {
    case 0: performSegue(withIdentifier: "showWebView", sender: self)

    case 1: openWithSafariViewController(indexPath: indexPath)

    default: break
    }

    tableView.deselectRow(at: indexPath, animated: false)
}
```

For demo purpose, the items in the first section are loaded using `WKWebView`. For the second section, we use the `SFSafariViewController` to display the web content. This will give you a better idea about the visual difference between `WKWebView` and `SFSafariViewController`.

It's ready to test the app! Once launched, tap an item in the second section. The app will open the link in a Safari-like browser.

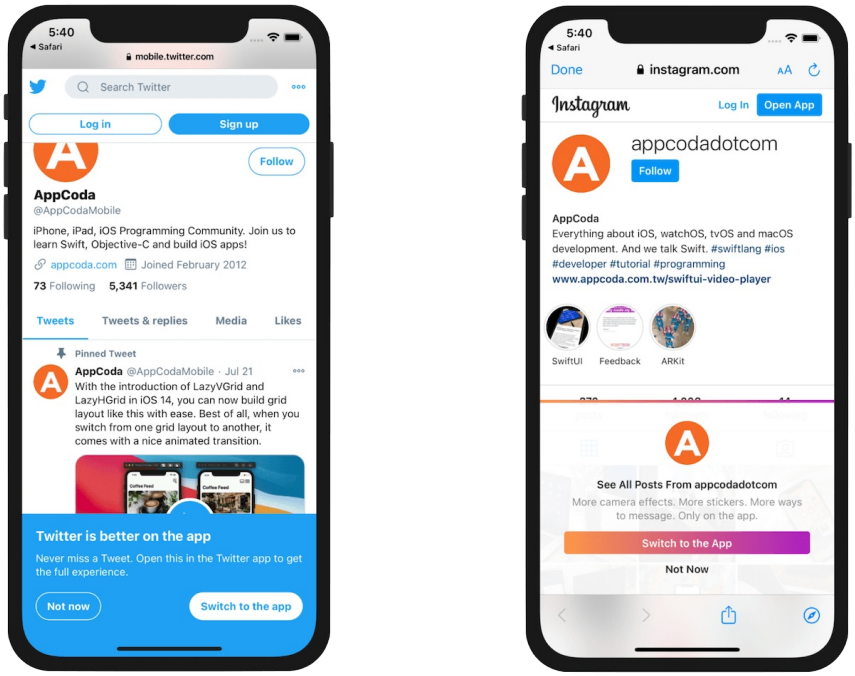


Figure 23-13. Loading a web page using SFSafariViewController

Summary

We have explored three options for displaying web content. You do not need to use all options in your app like we did. It is just for demo purpose.

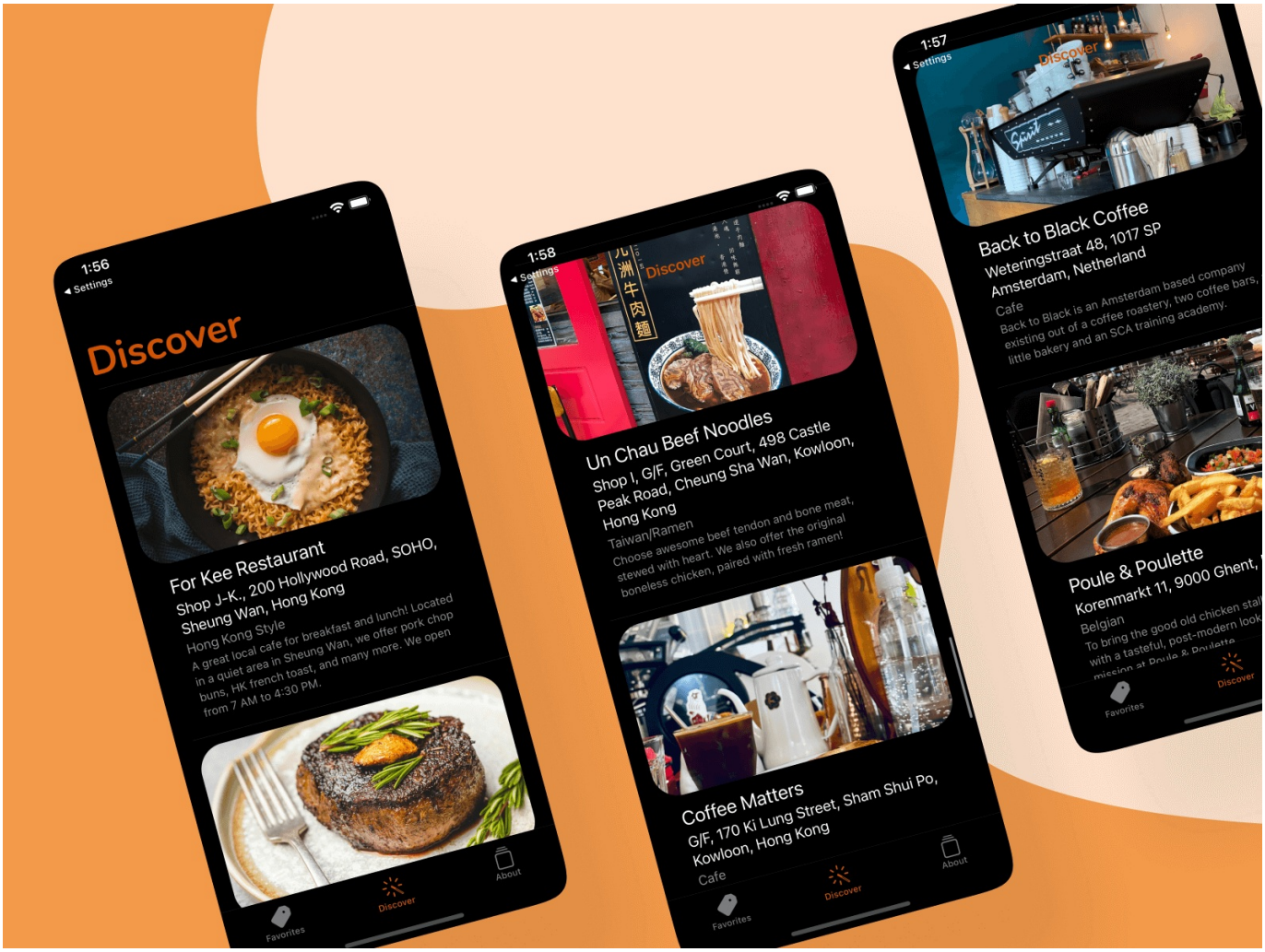
The `SFSafariViewController` class provides developers a simple way to embed a web browser in your app. If your app needs to provide a first class browsing experience for your user, the Safari view controller should save you a lot of time from creating your own version of web browser.

In some cases, you may just want a simple web view to display web content. `WKWebView` will be probably your choice.

Take some time to study all the web browsing options and pick the one that most fits your need.

For reference, you can download the complete Xcode project from http://www.appcoda.com/resources/swift53/FoodPinWebView.zip.

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.

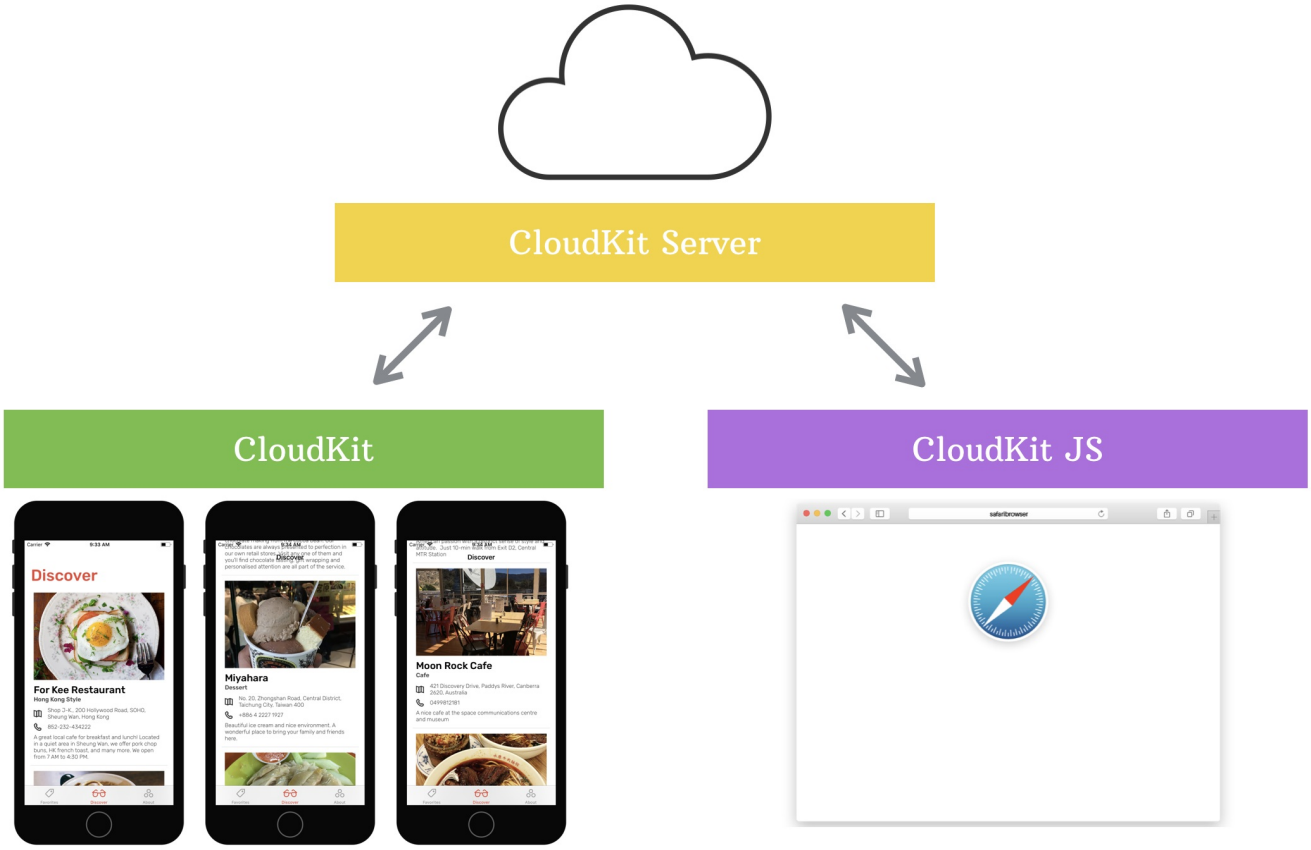


Figure 24-1. Storing your data to the cloud

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.

Understanding CloudKit Framework

CloudKit is not just about storage. Apple provides the CloudKit framework for developers to interact with iCloud. The CloudKit framework offers services for managing the data transfer to and from iCloud servers. It's a transfer mechanism

that takes your user's app data from the device and transfers it to the cloud.

Importantly, CloudKit doesn't provide any local persistence and it only provides minimal offline caching support. If you need caching to persist the data locally, you should develop your own solution.

Containers and databases are the fundamental elements of the CloudKit framework. Every app has its own container for managing its content. By default, one app talks to one container. The container is exposed as the `CKContainer` class.

Inside a container, it contains a public database, a shared database and a private database for storing data. As the name suggests, the public database is accessible by all users of the app and is designed to store shared data. Data stored in the private database is visible to a single user, while data stored in the shared database (introduced in iOS 10) can be shared among a group of users.

Apple lets you choose the type of database that best fits your app. For example, if you're developing an Instagram-like app, you can use the public database to save photos uploaded by users. Or if you're creating a To-do app, you probably want to use the private database to store the to-do items per user. The public database doesn't require users to have an active iCloud account unless you need to write data to the public database. Users need to log into the iCloud before accessing its private database. In the CloudKit framework, the database is exposed as the CKDatabase class.

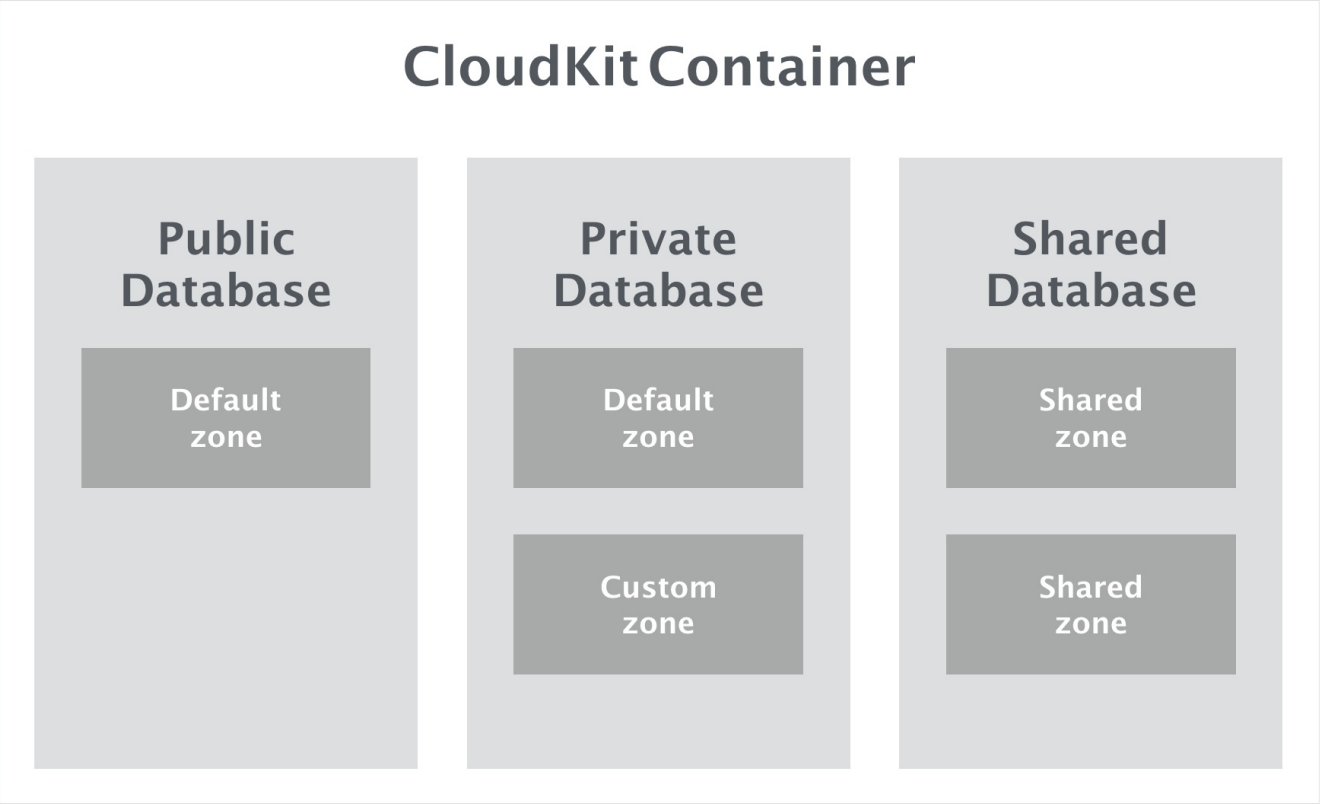


Figure 24-2. Visual Representation of Containers and Databases

Navigating further down the hierarchy is *Record Zone*. CloudKit does not store data loosely. Instead, records of data are partitioned into different *Record Zones*. Depending on the type of database, it supports different types of record zone. Both *private* and *public* databases have a default zone. It is good enough for most scenarios. That said, you're allowed to create custom zones if needed. Record Zone is exposed as the `CKRecordZone` class in the framework.

At the heart of the data, a transaction is a *Record*, represented by the `CKRecord` class. A *Record* is essentially a dictionary of key-value pairs. The key represents a record field. The associated value of a key is the value of a specific record field. Each record has a record type. The record type is defined by developers in the CloudKit dashboard. Meanwhile, you may be confused about all these terms. No worries. You will understand what they mean after going through a working demo.

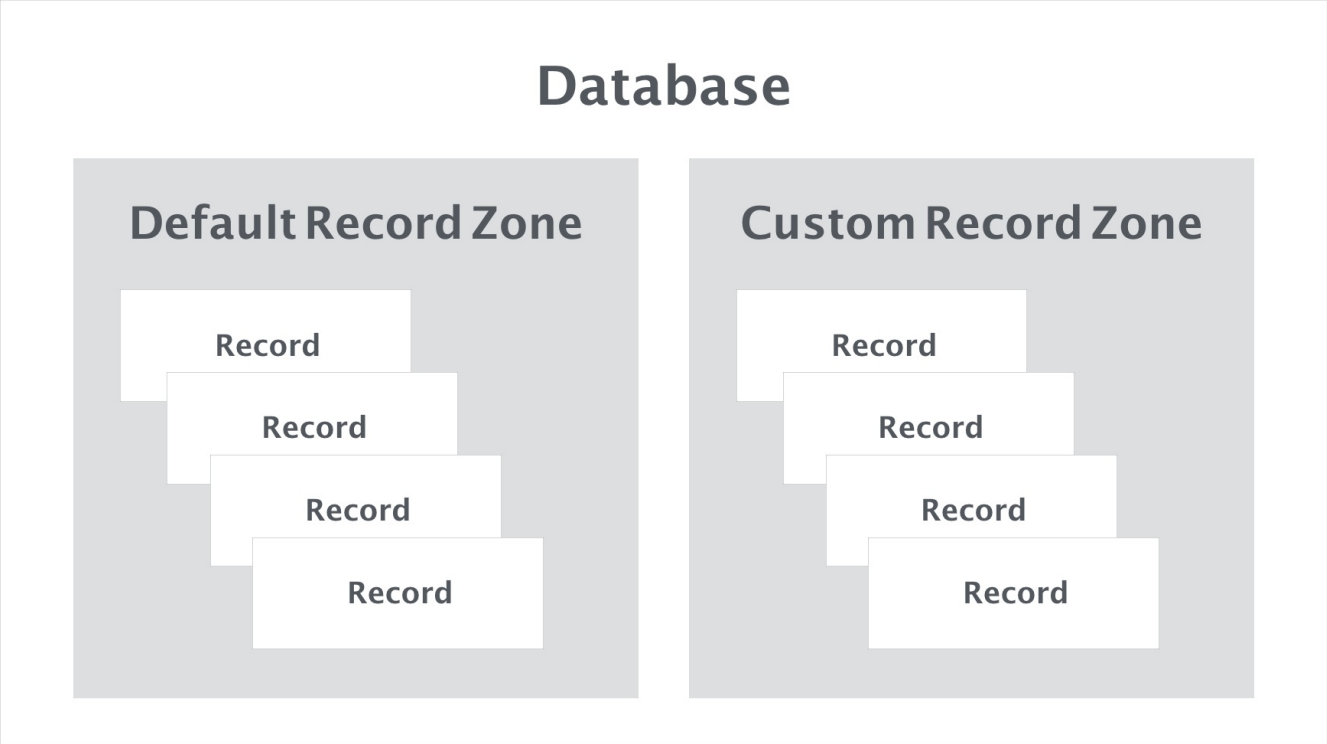


Figure 24-3. Record zones and Records

Now that you have some ideas about the CloudKit framework, let's get started and build the *Discover* tab. By integrating the app with CloudKit, you'll learn:

- How to enable CloudKit in your app
- How to use CloudKit Dashboard to create your records in the cloud
- How to fetch records asynchronously from iCloud using the convenience API
- How to fetch records from iCloud servers using the operational APIs
- How to improve app performance using lazy loading
- How to cache images using NSCache
- How to save data to iCloud servers

Enabling CloudKit in Your App

Assuming you have enrolled in the Apple Developer Program, the first task is to register your account in the Xcode project.

Note: In the project navigator, select the FoodPin project and then select FoodPin under targets. If you are using `com.appcoda.FoodPin` as the bundle identifier, you will need to change it to something else. Say, `[your domain].FoodPin`. If you don't own a domain, you may use `[your name].FoodPin`. Later, CloudKit will use the bundle identifier to generate the container. Because the name space of containers is global to all developers, you have to ensure the name is unique.

Under the *Signing & Capabilities* tab, if you haven't assigned a developer account in the *Signing* section, simply click the dropdown box of the *Team* option. Select *Add an account*, you'll be prompted to log in with your developer account. Follow the procedures and your developer account will appear in the *Team* option.

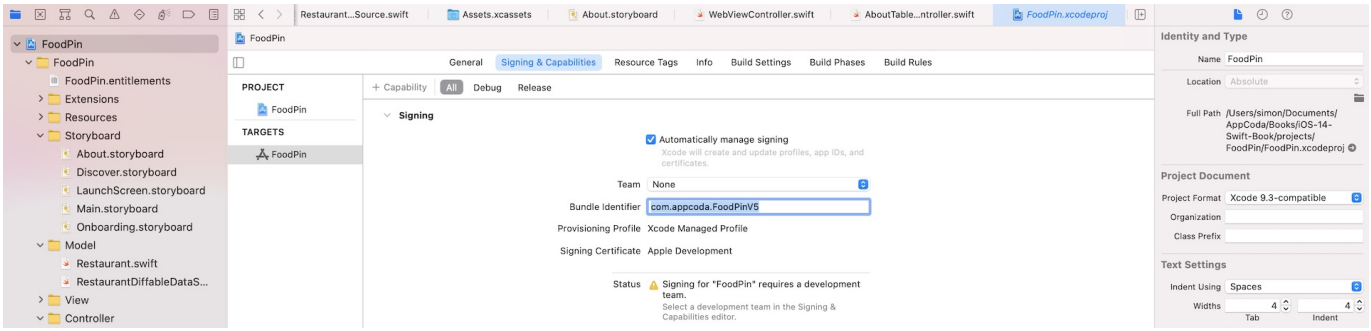


Figure 24-4. Assigning a developer account for your Xcode project

Quick note: For this demo, I will use `com.appcoda.FoodPinV5` as the bundle identifier.

Assuming you have the identity and bundle identifier configured, click the *+Capability* button. To enable CloudKit, all you need to do is add the *iCloud* module to your project. And then select `CloudKit` in the services option. For containers, click the `+` button in the *Containers* section to create a new container. The naming convention is like this:

iCloud.com.[bundle-ID]

As soon as you confirm, Xcode automatically creates the container on the CloudKit server and adds the necessary frameworks in the project. It may take a few minutes for Xcode to create the container on the cloud for you. If the container is not ready, it's displayed in red. You can click the reload button until the container changes to black.

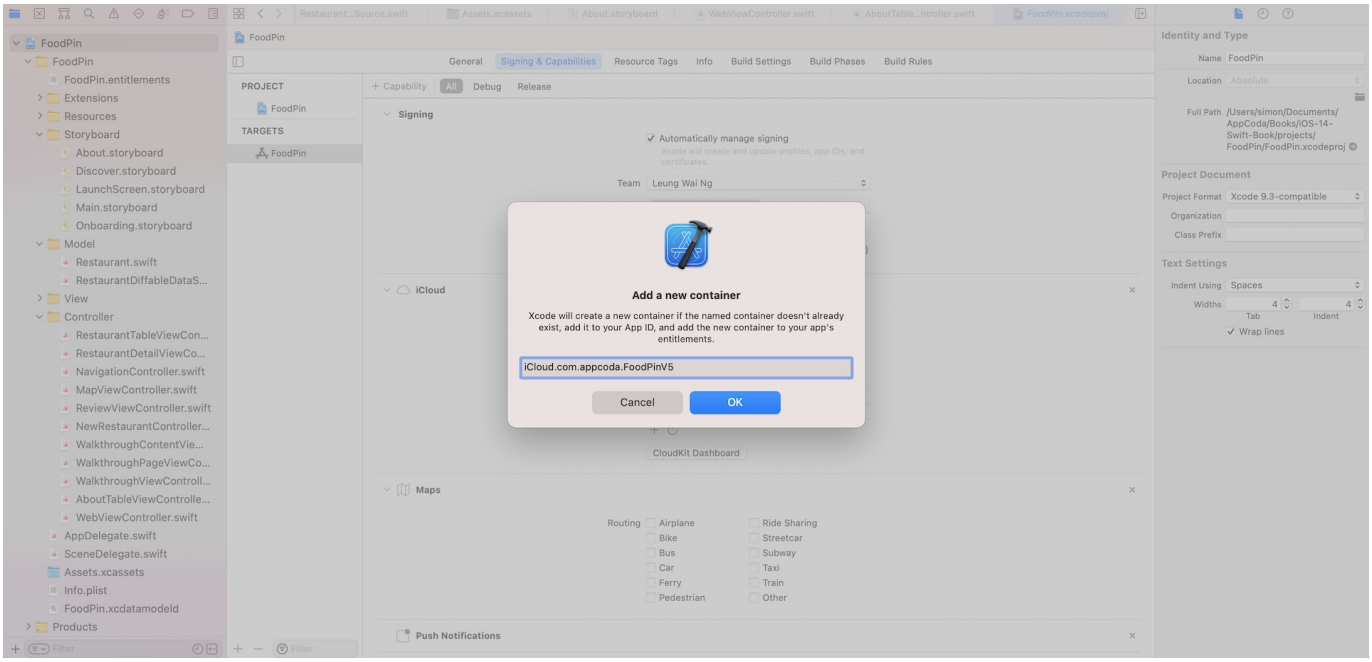


Figure 24-5. Enabling CloudKit for your app and add a new container

Quick tip: If you experience the error "An App ID with identifier is not available. Please enter a different string.", you may need to choose another bundle identifier.

Managing Your Record in CloudKit Dashboard

You can click the *CloudKit Dashboard* button to open a web-based dashboard (see figure 24-6). You should see the iCloud container with the name "iCloud.". For me, as my bundle ID is set to `com.appcoda.FoodPinV5`. The name of the iCloud container is `iCloud.com.appcoda.FoodPinV5`. Click on it to display the container's details. In case if you can't find the iCloud container, please try to restart your browser.

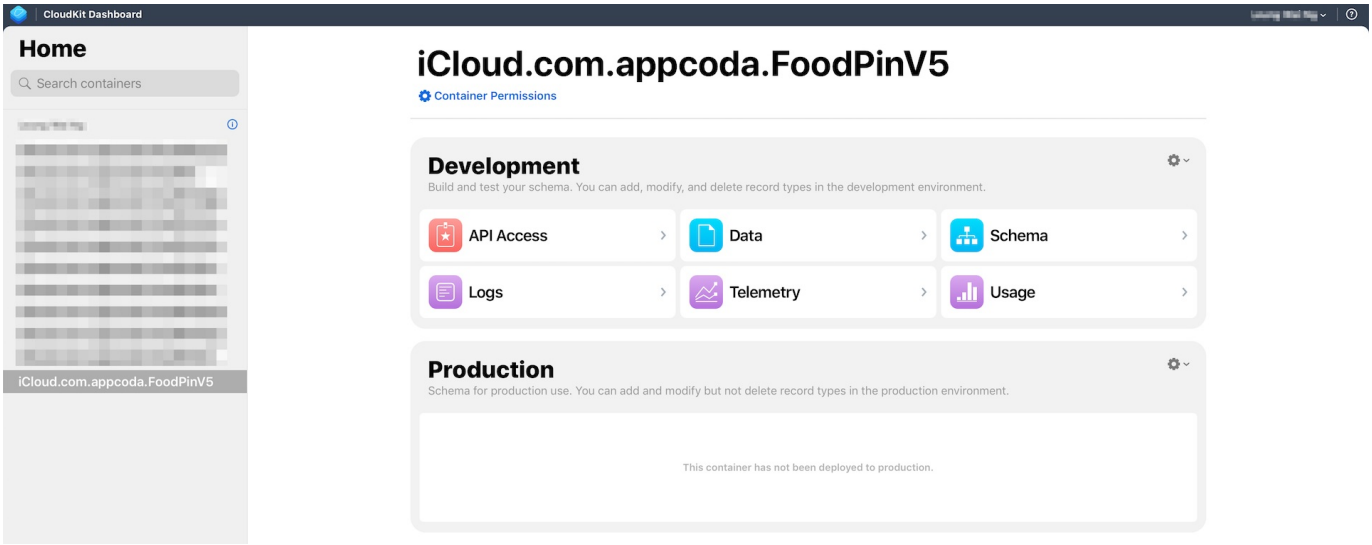


Figure 24-6. CloudKit Dashboard

Here you will be presented with two environments: Development and Production. *Production* is the live environment that is used when your app is released to public users. Development, as the name suggests, is used when you are developing the

app or for testing. Both environments offer the same options. The *Data* option allows you to manage your records in the cloud database. Now choose the *Data* option of the Development environment.

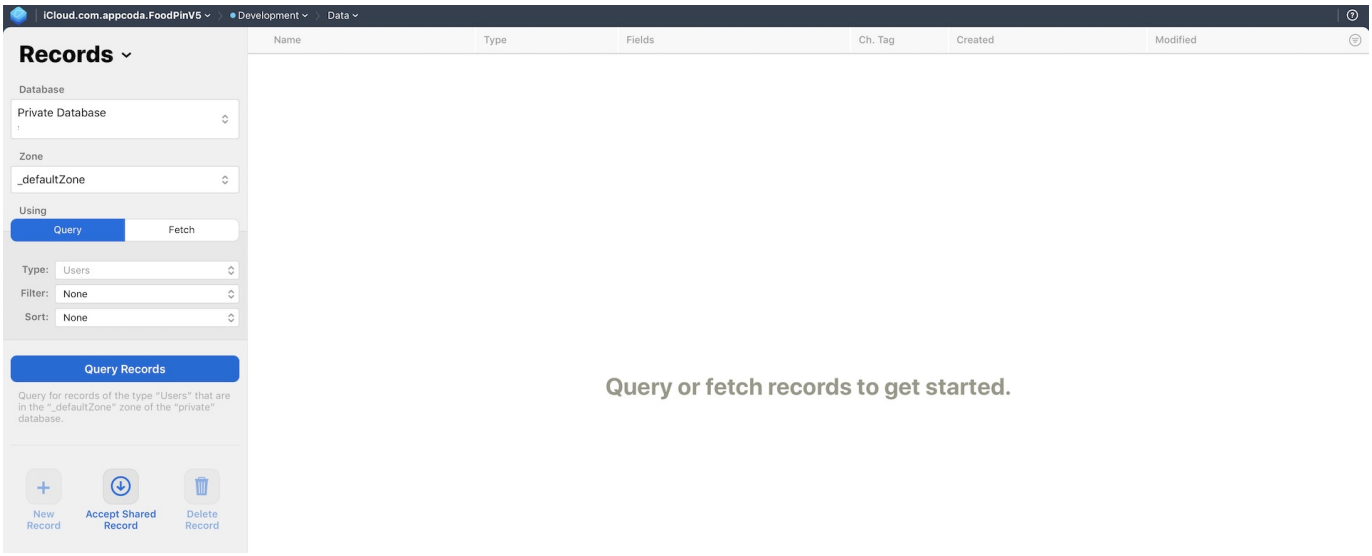


Figure 24-7. CloudKit Dashboard - Data

You will then see a dashboard that lets you manage your container and perform operations like adding record types and removing records.

Before your app can save restaurant records to the cloud, you first need to define a record type. Do you remember that we created a `Restaurant` entity when working with Core Data? A record type in CloudKit is equivalent to an entity in Core Data.

In the top menu bar of the dashboard, select *Schema* and then click `+ New Type` to create a new record type. Name the record type `Restaurant`. Once you created the record type, CloudKit dashboard will show you some system fields such as createdBy and createdAt.

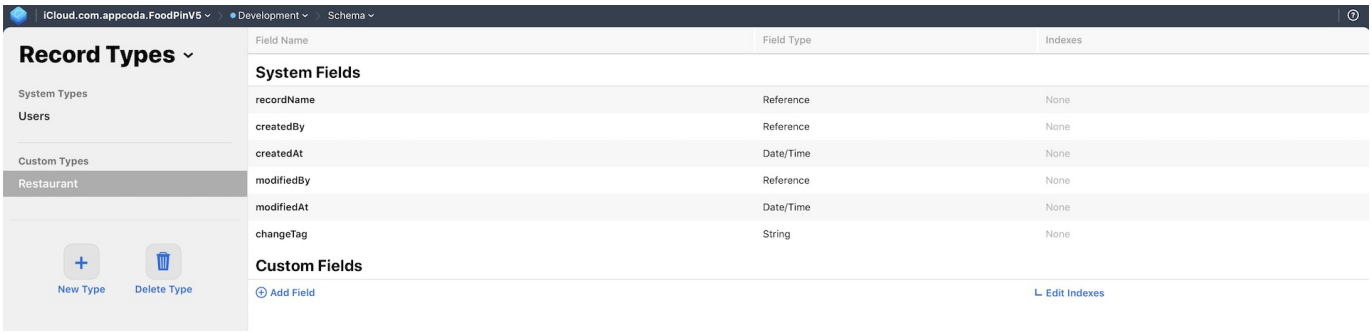


Figure 24-8. Adding a new record type

You can define your own field name and type for the *Restaurant* record type. CloudKit supports various attribute types such as `String`, `Date/Time`, `Double` and `Location`. If you need to store binary data like image, you use the `Asset` type.

Now click the *Add Field* button and add the following field names/types for the `Restaurant` record type:

Field Name	Field Type
name	String
type	String
location	String
phone	String
description	String
image	Asset

Once you finish adding your own fields, don't forget to click the *Save* button to confirm the changes.

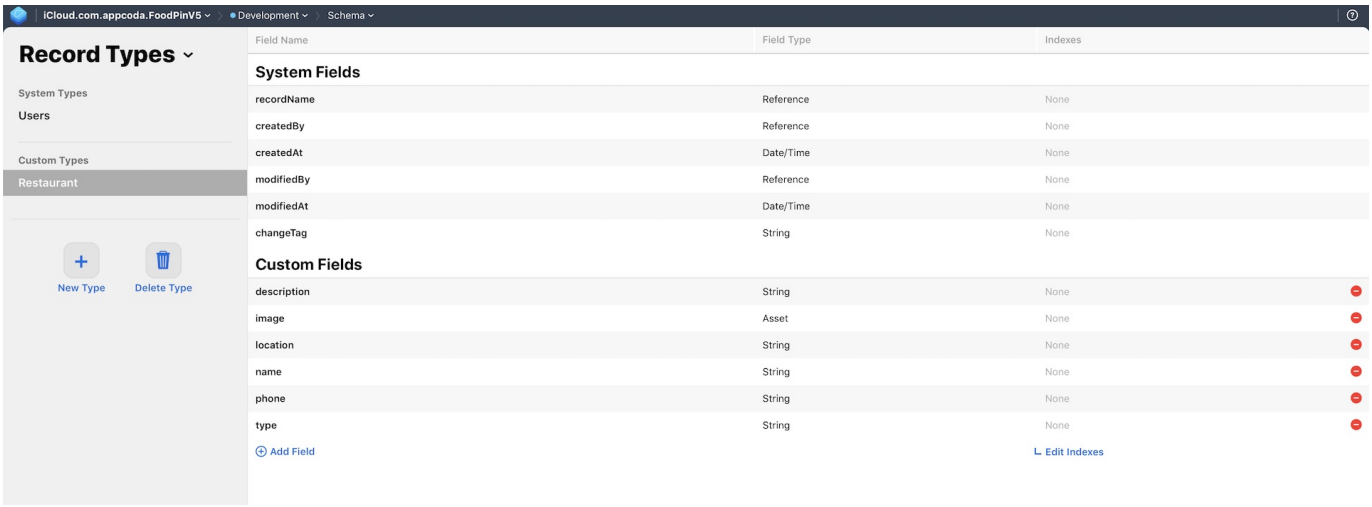


Figure 24-9. Adding custom fields

> **Note:** CloudKit uses asset objects to incorporate external files such as image, sound, video, text, and binary data files. An asset is exposed as the `CKAsset` class and associated with a record. When saving an asset, CloudKit only stores the asset data. It does not store the filename. Other than images, you can configure the sort, query and search options for the rest of the fields.

With the record type configured, it's ready for your app to upload the restaurant records to iCloud. You have two ways to add records to the database:

- You either create the records through the CloudKit APIs.
- Or you add the records via the CloudKit dashboard.

Let's try to populate some records using the dashboard. In the top menu bar, choose *Data* to go back to the records panel. By default, the dashboard is set to the private database. Click the dropdown box right below "Load Records From" and choose *Public Database* to switch over to the public database.

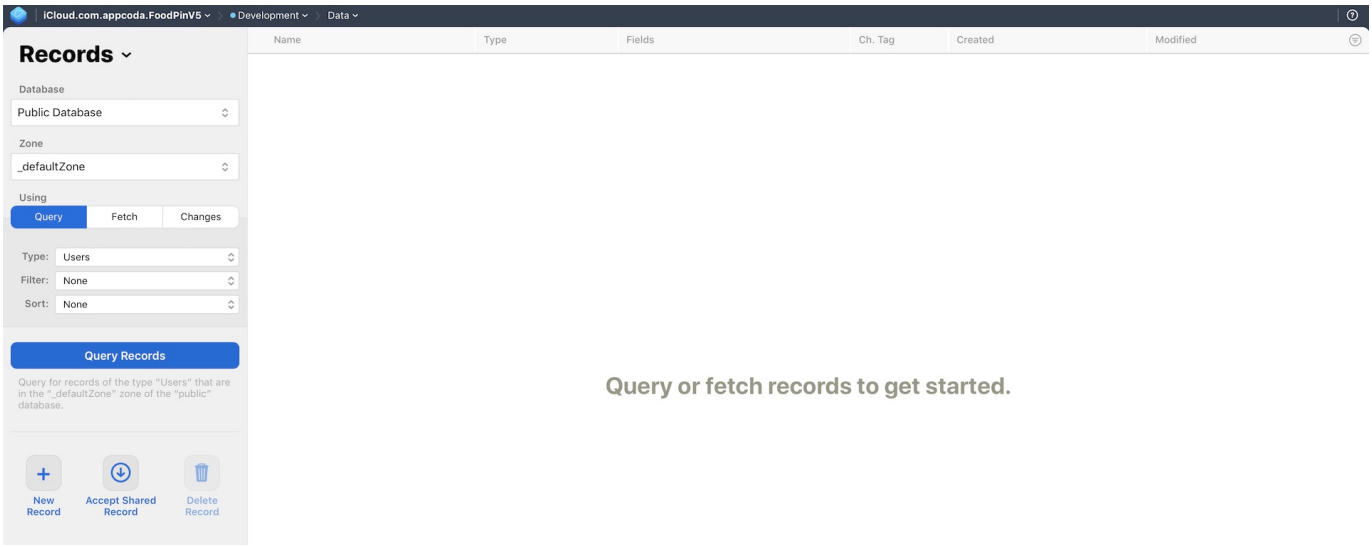


Figure 24-10. Choose the public database

In the left pane, make sure the *_defaultZone* is selected. This is the default record zone of your public database. For the type, set it to *Restaurant*. By default, the zone doesn't contain any records. You can click the + *New Record* button to create one. Simply key in the name, type, location, phone, description, and upload your image. Then click *Save* to save the record. Figure 24-11 shows a sample new record.

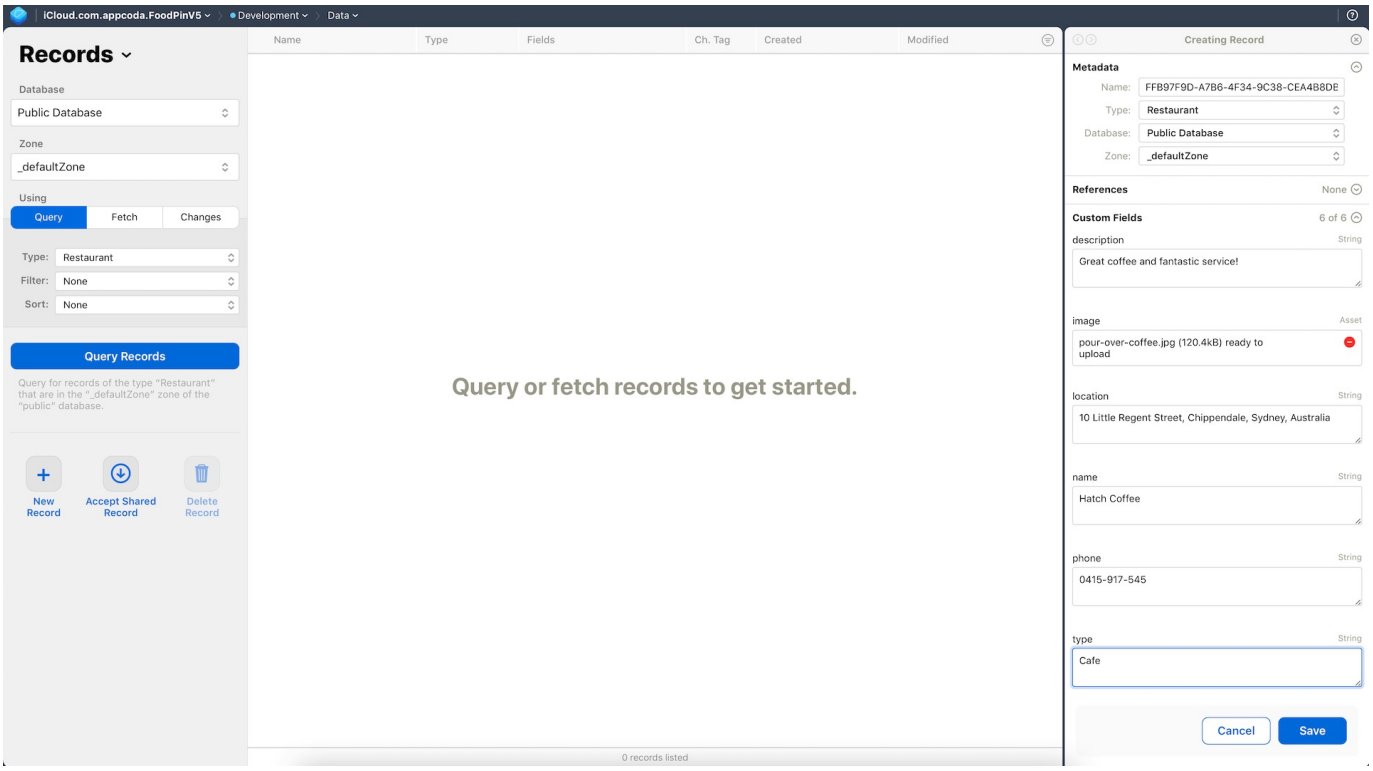


Figure 24-11. Adding a new record in CloudKit dashboard

Now you have created a Restaurant record in the cloud. Repeat the same procedures and create around 10 records; we'll use them later.

If you've tried to query the records, you will end up with an error "*There was a problem query the Restaurant type: Type is not marked indexable. Enable the queryable index on this field in the indexes section.*" All the metadata indexes for record types created are disabled by default. Therefore, before you are allowed to query the records, you will have to add an index to the database.

Click *Schema* in the top menu bar, choose *Restaurant*, and then click *Edit index*. A database index allows a query to efficiently retrieve data from a database. You can click the *Add Index* button to create an index. We are going to create two indexes on the `recordName` and `createdAt` fields. For the `recordName` field, the index type

is set to *Queryable* which means that the records can be queried. Later, we will retrieve the records in reverse chronological order. Thus, we set the index type of the `createAt` field to *Sortable*.

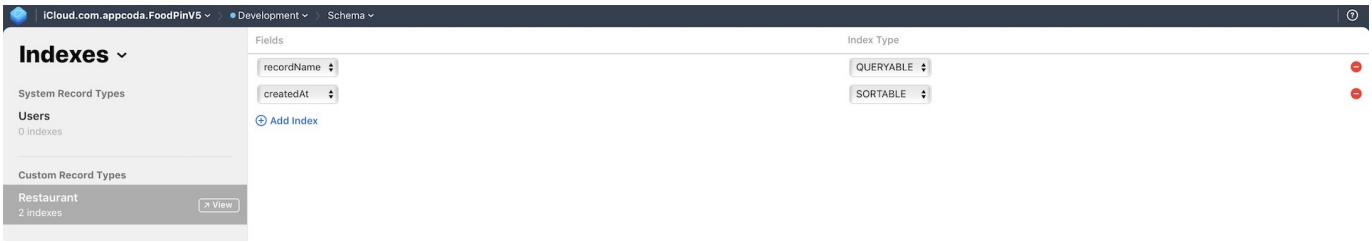


Figure 24-12. Adding indexes

After saving the changes, go back to the *Data* panel, choose *Public Database*, and click the *Query Records* button. You should now be able to retrieve the restaurant records.

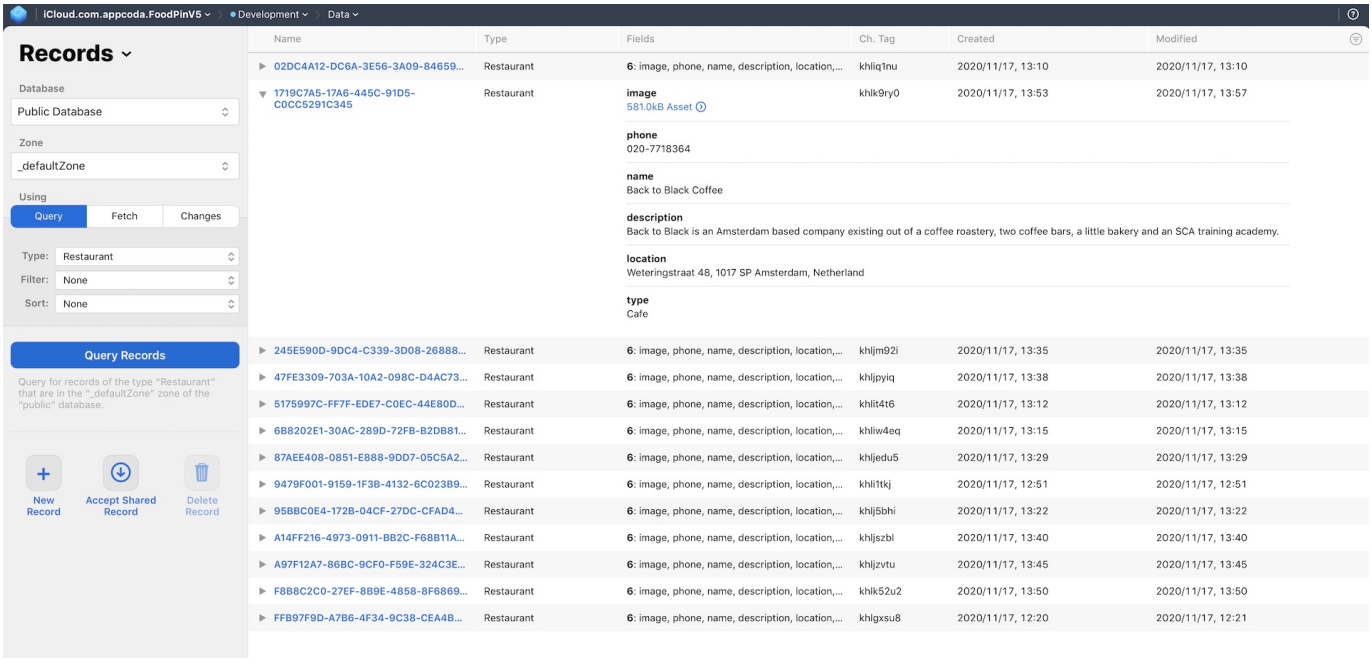


Figure 24-13. Querying records in the public database

Fetching Data from a Public Database Using Convenience API

The CloudKit framework provides two types of APIs for developers to interact with iCloud. They are known as the convenience API and the operational API. Both APIs let you save and fetch data from iCloud asynchronously. In other words, the data transfer is executed in the background. We will first go over the convenience API and use it to implement the *Discover* tab. After that, we will discuss the operational API.

As its name suggests, the convenience API allows you interact with iCloud with just a few lines of code. In general, you just need the following code to fetch the `Restaurant` records from the cloud:

```
let cloudContainer = CKContainer.default()
let publicDatabase = cloudContainer.publicCloudDatabase
let predicate = NSPredicate(value: true)
let query = CKQuery(recordType: "Restaurant", predicate: predicate)
publicDatabase.perform(query, inZoneWith: nil, completionHandler: {
            (results, error) -> Void in

// Process the records

})
```

The above code is fairly straightforward. We first get the default CloudKit container of the app, followed by obtaining the default public database. To retrieve the `Restaurant` records from the public database, we construct a `CKQuery` object with the `Restaurant` record type and the search criteria (i.e. `predicate`).

The predicate may be new to you. The iOS SDK provides a foundation class called `NSPredicate` for developers to specify how data should be filtered. If you have some database background, you may think of it as the `WHERE` clause in SQL. You

can't perform a `CKQuery` without a predicate. Even if you want to query the records without any filtering, you still need to specify a predicate. In this case, we initialize a predicate that always evaluates to `true`. This means we do not perform any sorting on the query result.

Lastly, we call the `performQuery` method of `CKDatabase` with the query. CloudKit then searches and returns the results as an array of `CKRecord`. The search and data transfer operations are executed in the background. When the records are ready, it will invoke the complete handler for further processing.

Simple, right? Now let's go back to our FoodPin project and implement the *Discover* tab.

First, create a `DiscoverTableViewController` class for the table view controller. In the project navigator, right-click the `Controller` folder and select *New Files...*. Choose the *Cocoa Touch Class* template. Name the class `DiscoverTableViewController` and set it as a subclass of `UITableViewController`.

Once created, go to `Discover.storyboard`. Select the table view controller of the *Discover* tab and set its custom class to `DiscoverTableViewController` in the Identity inspector. For the prototype cell, just use the `Basic` style and set the identifier to `discovercell`.

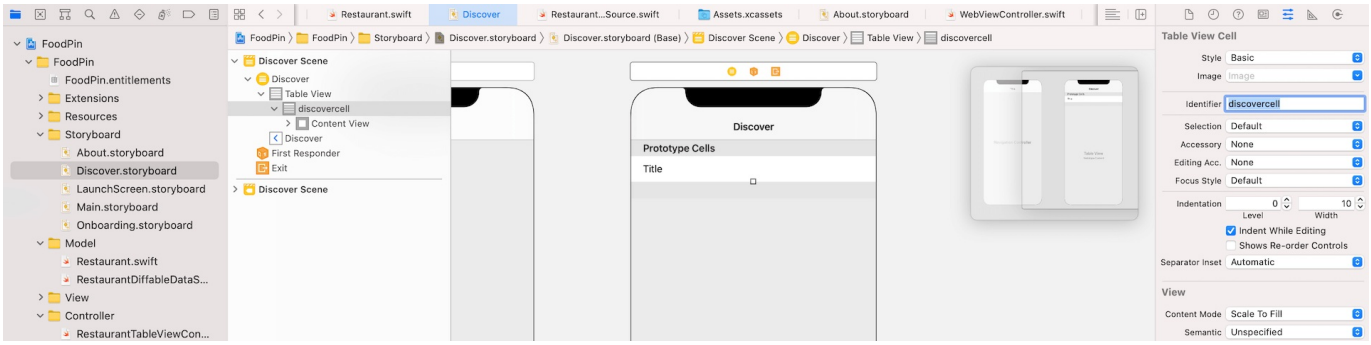


Figure 24-14. Set the cell identifier for the table view controller

Now you are ready to implement `DiscoverTableViewController` to fetch records from the cloud. To use CloudKit, you have to first import the CloudKit framework. Insert the following line of code at the very beginning of `DiscoverTableViewController.swift`:

```
import CloudKit
```

These two methods are generated by Xcode. However, we do not need them. Delete the following code:

```
override func numberOfSections(in tableView: UITableView) -> Int {
    // #warning Incomplete implementation, return the number of sections
    return 0
}

override func tableView(_ tableView: UITableView, numberOfRowsInSection se
ction: Int) -> Int {
    // #warning Incomplete implementation, return the number of rows
    return 0
}
```

Next, declare a restaurants variable that stores an array of `CKRecord` objects. Initially, the array is empty. Later, we use it to store the records fetched from the cloud.

```
var restaurants: [CKRecord] = []
```

Also, declare another variable for the data source. Again, we will use the diffable data source to populate the data in the table view:

```
lazy var dataSource = configureDataSource()
```

When the *Discover* tab is loaded, the app will start to fetch the records via CloudKit. Update the `viewDidLoad()` method like this:

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

    tableView.cellLayoutMarginsFollowReadableWidth = true
    navigationController?.navigationBar.prefersLargeTitles = true

    // Customize the navigation bar appearance
    if let appearance = navigationController?.navigationBar.standardAppear
ance {

        appearance.configureWithTransparentBackground()

        if let customFont = UIFont(name: "Nunito-Bold", size: 45.0) {

            appearance.titleTextAttributes = [.foregroundColor: UIColor(na
med: "NavigationBarTitle")!]
            appearance.largeTitleTextAttributes = [.foregroundColor: UICol
or(named: "NavigationBarTitle")!, .font: customFont]
        }

        navigationController?.navigationBar.standardAppearance = appearanc
e
        navigationController?.navigationBar.compactAppearance = appearance
        navigationController?.navigationBar.scrollEdgeAppearance = appeara
nce
    }

    // Fetch record from iCloud
    fetchRecordsFromCloud()

    // Set the data source of the table view for Diffable Data Source
    tableView.dataSource = dataSource
}
```

In the code above, we customize the navigationbar and call a new method `fetchRecordsFromCloud()` to fetch records from iCloud. We also set the table view's data source. We haven't implemented both `fetchRecordsFromCloud()` and `configureDataSource()` methods.

Let's start with the implementation of the `fetchRecordsFromCloud` method. Insert the following code in the `DiscoverTableViewController` class:

```
func fetchRecordsFromCloud() {

    // Fetch data using Convenience API
    let cloudContainer = CKContainer.default()
    let publicDatabase = cloudContainer.publicCloudDatabase
    let predicate = NSPredicate(value: true)
    let query = CKQuery(recordType: "Restaurant", predicate: predicate)
    publicDatabase.perform(query, inZoneWith: nil, completionHandler: {
        (results, error) -> Void in

        if let error = error {
            print(error)
            return
        }

        if let results = results {
            print("Completed the download of Restaurant data")
            self.restaurants = results

            self.updateSnapshot()
        }
    })
}
```

The block of code is nearly the same as the one we discussed before. In the complete handler, we check if there are any errors. If there is no error, we then get the `results` array and store it in the `restaurants` array. Finally, we call the `updateSnapshot()` method to refresh the table view with the updated restaurants.

We haven't implemented the `updateSnapshot()` method yet, so insert the following code to create the method:

```
func updateSnapshot(animatingChange: Bool = false) {

    // Create a snapshot and populate the data
    var snapshot = NSDiffableDataSourceSnapshot<Section, CKRecord>()
    snapshot.appendSections([.all])
    snapshot.appendItems(restaurants, toSection: .all)

    dataSource.apply(snapshot, animatingDifferences: false)
}
```

The code should be very familiar to you. We create a snapshot and add the restaurants to the table view.

Lastly, let's implement the `configureDataSource()` method:

```
func configureDataSource() -> UITableViewDiffableDataSource<Section, CKRec
ord> {

    let cellIdentifier = "discovercell"

    let dataSource = UITableViewDiffableDataSource<Section, CKRecord>(tabl
eView: tableView) { (tableView, indexPath, restaurant) -> UITableViewCell?
in

        let cell = tableView.dequeueReusableCell(withIdentifier: cellIdent
ifier, for: indexPath)

        cell.textLabel?.text = restaurant.object(forKey: "name") as? String


        if let image = restaurant.object(forKey: "image"), let imageAsset
= image as? CKAsset {

            if let imageData = try? Data.init(contentsOf: imageAsset.fileU
RL!) {
                cell.imageView?.image = UIImage(data: imageData)
            }
        }

        return cell
    }

    return dataSource
}
```

Again, you should be very familiar with the code. This is how we use the diffable data source to populate the table data. Note that the `restaurants` variable contains an array of `CKRecord` objects. As mentioned before, a `CKRecord` object is a dictionary of key-value pairs that you use to fetch and save the data of your app. It provides the `object(forKey:)` method to retrieve the value of a record field (e.g.

the name field of the restaurant). When creating the `Restaurant` record type in the CloudKit dashboard, we use `Asset` as the type of the image field. Thus, the object returned is a `CKAsset` object.

When downloading a record containing an asset, CloudKit stores the asset data in the local file system. You can then retrieve the asset data with the URL in the `fileURL` property. This is why we can load the image by accessing the `fileURL` property of the image asset.

```
if let imageData = try? Data.init(contentsOf: imageAsset.fileURL) {
    cell.imageView?.image = UIImage(data: imageData)
}
```

We've discussed how to load an image with binary data before. But what is the `try?` keyword? When you initialize a `Data` with the file URL, `Data` may fail to load the file and throws you an error. You can create a do-catch statement to handle the error. But in the above code, we use `try?` instead. In this case, if an error is thrown, the method returns an optional without a value. If the image file is successfully loaded, the data is bind to the `imageData` variable. `try?` is particularly useful if you do not care about the error message.

> In development, when you run your app through Xcode on a simulator or a device, you need to enter iCloud credentials to read records in the public database. In production, the default permissions allow non-authenticated users to read records in the public database but do not allow them to write records.
>
> - Apple's CloudKit Quick Starter Guide

I know you can't wait to test the Discover feature. Before testing the app in the simulator, however, you have to configure the iCloud setting. Otherwise, you will not be able to fetch the data from iCloud. In the Simulator, click the home button to go back to Home screen. Select *Settings* and sign in with your Apple ID.

Now you're ready to run the app. Select the *Discover* tab and the app should be able to fetch the restaurants through CloudKit. Figure 24-15 shows a sample screenshot of the *Discover* tab.

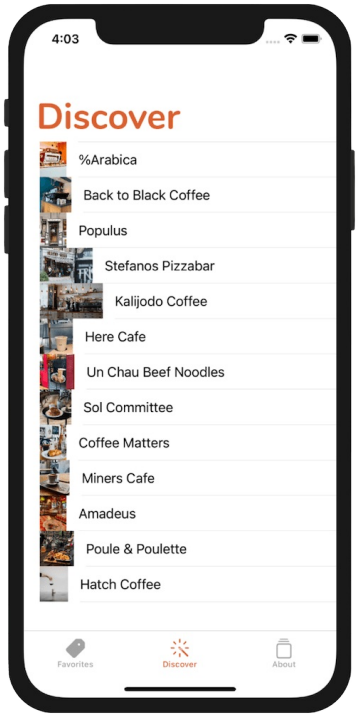


Figure 24-15. The Discover tab now downloads the restaurant from the public database

> **Note:** Occasionally you may end up with an error in the console. In this case, click the Stop button to quit the app and launch it again.

Fetching Data from Public Database Using Operational API

The convenience API is good for simple queries. However, it does have some limitations. The `perform(_:inZoneWith:completionHandler:)` method is only suitable for retrieving a small amount of data. If you have a few hundred records (or even more), the convenience API may not fit. You cannot perform additional tweaking with the convenience API. When you call the `perform` method, it retrieves all restaurant records. For each record, it downloads the whole records including the image and every other field. However, as you may notice, the app just displays the restaurant names and the images. We can actually leave out those fields like phone number, type, and location to save some bandwidth.

So how can you tell CloudKit to merely retrieve the name field of all restaurant records? You cannot do that via the convenience API. In this case, you'll need to explore the operational API.

The usage of operational API is similar to that of the convenience API but it offers more flexibility. Let's jump right into the code. Replace the `fetchRecordsFromCloud` method with the following code snippet:

```
func fetchRecordsFromCloud() {

    // Fetch data using Operational API
    let cloudContainer = CKContainer.default()
    let publicDatabase = cloudContainer.publicCloudDatabase
    let predicate = NSPredicate(value: true)
    let query = CKQuery(recordType: "Restaurant", predicate: predicate)

    // Create the query operation with the query
    let queryOperation = CKQueryOperation(query: query)
    queryOperation.desiredKeys = ["name", "image"]
    queryOperation.queuePriority = .veryHigh
    queryOperation.resultsLimit = 50
    queryOperation.recordFetchedBlock = { (record) -> Void in
        self.restaurants.append(record)
    }

    queryOperation.queryCompletionBlock = { [unowned self] (cursor, error)
 -> Void in
        if let error = error {
            print("Failed to get data from iCloud - \(error.localizedDescr
iption)")
            return
        }

        print("Successfully retrieve the data from iCloud")

        self.updateSnapshot()
    }

    // Execute the query
    publicDatabase.add(queryOperation)

}
```

The first few lines of code are exactly as before. We get the default container and the public database, followed by creating the query for retrieving the restaurant records.

Instead of calling the `perform` method to fetch the records, we create a `CKQueryOperation` object for the query. This is why Apple called it *operational API*. The query operation object provides several options for your configuration. The `desiredKeys` property lets you specify the fields to fetch. You use this property to retrieve only those fields that you need for the app. In the above code, we tell the query operation object that we only need the `name` and `image` fields of the records.

Other than the `desiredKeys` property, you can use the `queuePriority` property to specify the execution priority of the operation and `resultsLimit` property to set the maximum number of records at any one time.

The operation object will be executed in the background. It reports the status of the query operation through two callbacks. One is `recordFetchedBlock` and the other is `queryCompletionBlock`. The block of code within `recordFetchedBlock` will be executed every time a record returned. In the code snippet, we simply append each of the returned records to the `restaurants` array.

On the other hand, `queryCompletionBlock` allows you to specify the code block that executes after all records are fetched. In this case, we ask the table view to reload and display the restaurant records.

Let me talk a bit more about the `queryCompleteBlock`. It provides a cursor object as a parameter to indicate if there are more results to fetch. Recall that we use the `resultsLimit` property to control the number of the fetched records, the app may not be able to fetch all data in a single query. In this case, a `CKQueryCursor` object indicates there are more results to fetch. Additionally, it marks the stopping point of the query and the starting point for retrieving the remaining results. For example, let's say you have a total of 100 restaurant records. For each search

query, you can get a maximum of 50 records. After the first query, the cursor would indicate that you have fetched record 1-50. For your next query, you should start from the 51st record. The cursor is very useful if you need to get your data in multiple batches. This is one of the ways to retrieve a large set of data.

At the end of the `fetchRecordsFromCloud` method, we call the `add` method of the `CKDatabase` class to execute the query operation. You can now compile and run the app again. The result should be the same as before. Internally, however, you have built a custom query to fetch only those data you need.

Optimizing the Performance

Up until now, both approaches achieve the same performance. It is not too slow as the size of the data set is relatively small. However, as your data set grows, it will take even longer to download and display the records to users. Performance is critical in mobile app development. Your users expect your app to be fast and responsive. A sluggish performance of an app may mean losing a customer. So how can we improve the performance?

Optimizing the server performance? No, that's out of our control. We can only focus on the things that we can optimize.

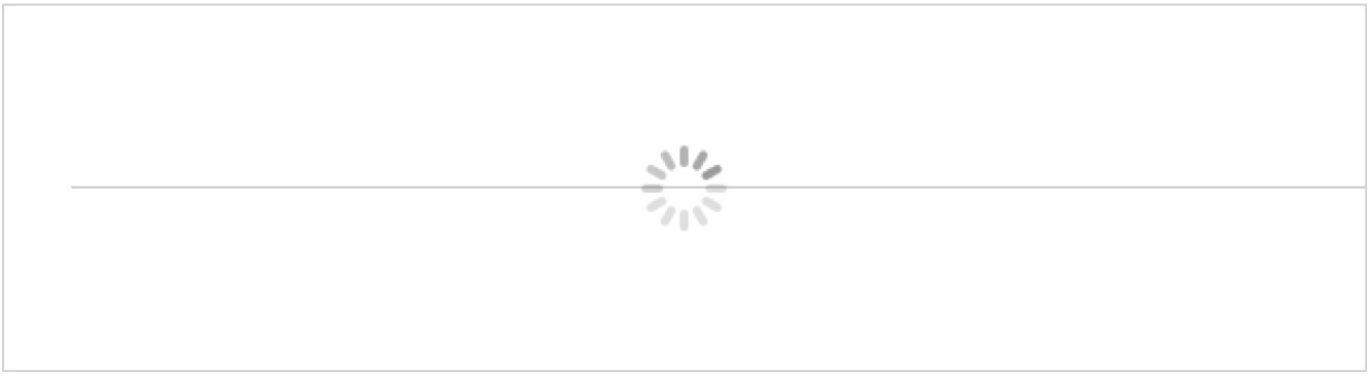
Reducing the size of the images? This is a good suggestion. You can further optimize the images to achieve a faster download. But from a user point of view, the improvement would not be very significant as the sizes of the original images were moderately optimized.

> **Tip:** tinypng.com is a great website for image optimization.

When we talk about performance optimization, sometimes we're not talking about optimizing the real performance but rather perceived performance. Perceived performance refers to how fast your user thinks your app is. Let me give you an example. Say, after a user taps the *Discover* tab, it takes 10 seconds to load the restaurant records. You then optimize the image sizes and reduce the loading time to 6 seconds. The real performance is improved by 40%. You think that's a huge improvement. But the perceived performance is still sluggish. To the user, your app is still slow because it can't respond instantaneously. When it comes to performance optimization, sometimes the technical statistics do not matter. Rather, it's about optimizing the perceived performance to make users feel that your app is speedy.

Activity Indicators

One way to improve the perceived performance is to make your app responsive. Currently, when a user selects the *Discover* tab, nothing shows up. The user expects an instantaneous response. This doesn't mean you have to show the remote content immediately. However, your app should at least display something even it's waiting for the data. Adding an animated spinner or an activity indicator will do the trick. The spinner gives a real-time feedback and tells the user that something is happening. Research has also shown that the feeling of waiting can be reduced by keeping a user's attention occupied.



The UIKit framework provides a class called `UIActivityIndicatorView` for showing a spinner to indicate a task is in progress. Now declare a property in the `DiscoverTableViewController` class:

```
var spinner = UIActivityIndicatorView()
```

In the `viewDidLoad()` method, insert the following code:

```
spinner.style = .medium
spinner.hidesWhenStopped = true
view.addSubview(spinner)

// Define layout constraints for the spinner
spinner.translatesAutoresizingMaskIntoConstraints = false
NSLayoutConstraint.activate([ spinner.topAnchor.constraint(equalTo: view.s
afeAreaLayoutGuide.topAnchor, constant: 150.0),
                              spinner.centerXAnchor.constraint(equalTo: vi
ew.centerXAnchor)])

// Activate the spinner
spinner.startAnimating()
```

When the spinner is first initialized, its position is set to the top-left of a view. Here we use the `center` property to place the indicator at the center of the view. The `hidesWhenStopped` property controls whether the indication is hidden when the animation is stopped. Once the indicator is configured, you can add it to the current view and call the `startAnimating` method to start the animation.

In the code above, we define the layout constraints programmatically. We first set the `translatesAutoresizingMaskIntoConstraints` to `false`. This tells iOS not to create any auto layout constraints for the spinner view. We will do it manually. Then we add a couple of the layout constraints to center the spinner horizontally and place it 150 points below the top anchor of the safe area.

That's it! If you run the app, the activity indicator will appear when you select the *Discovery* tab. However, it doesn't hide even when the restaurant records are fully loaded.

The indicator has no idea when it should be hidden. When the data download completes, you have to explicitly call the `stopAnimating` method to stop the animation, right after calling the `updateSnapshot()` method in `fetchRecordsFromCloud()`. Because the `hidesWhenStopped` property is set to true, the activity indicator will then be hidden automatically.

```
func fetchRecordsFromCloud() {

    .
    .
    .

    queryOperation.queryCompletionBlock = { [unowned self] (cursor, error)
 -> Void in
        .
        .
        .

        self.updateSnapshot()
        self.spinner.stopAnimating()
    }

    .
    .
    .
}
```

Once you made the change, you can run the app again and see if the spinner works. Oops... why didn't it work? The spinner was not stopped and Xcode ends up with the following error in the console:

```
Main Thread Checker: UI API called on a background thread: -[UIActivityInd
icatorView stopAnimating]
PID: 12011, TID: 946337, Thread name: (none), Queue name: com.apple.cloudk
it.operation-1F5E2A18564C3875.callback, QoS: 0
```

You should also notice a purple indicator at the top right corner of the code editor in Xcode. Click the indicator and you can find out the root cause of the error:

```
77
78          queryOperation.queryCompletionBlock = { [unowned self] (cursor, error) -> Void in
79              if let error = error {
80                  print("Failed to get data from iCloud - \(error.localizedDescription)")
81                  return
82              }
83
84              print("Successfully retrieve the data from iCloud")
85
86              self.updateSnapshot()
87              self.spinner.stopAnimating()   UIActivityIndicatorView.stopAnimating() must be used from main thr...
88          }
89
90          // Execute the query
91          publicDatabase.add(queryOperation)
92
93      }
```

Figure 24-17. Clicking the purple indicator to reveal the root cause of the error

What does the error message mean? How do you make the call in the main thread?

Basically, there are two types of operations in an app: *synchronous* and *asynchronous*. When the app executes a synchronous operation, it initiates the operation and waits until it finishes. During the execution of the operation, every other operations like UI updates and user interaction events will be blocked.

On the other hand, if the operation is asynchronous in nature, the app usually initiates the operation in a background thread and will not wait for its completion. This frees up itself to handle other operations such as UI updates. When the operation is done, it calls up the completion handler to perform further operations.

Like most network calls in iOS, the query operation is an asynchronous call. The block of code specified in the `queryCompletionBlock` are executed on the background thread.

There is one rule of iOS programming you have to remember. All UI updates should be run on the main thread to keep your UI responsive. In the code above, `self.spinner.stopAnimating()` is a UI update. This is why Xcode highlighted that line of code telling you that it must be used from the main thread.

So, how can you instruct iOS to run a certain line or block of code on the main thread?

You can wrap that line of code in a call to `DispatchQueue.main.async` like this to ensure the table view update is run on the main thread:

```
DispatchQueue.main.async {
    self.spinner.stopAnimating()
}
```

Now run the project again. The app should be able to stop the spinner properly.

Lazy Loading Images

Cool! The spinner made the app more responsive. Are there any other ways to optimize the performance? Let's take a look at how we fetch the data from iCloud. Currently, the app displays the records on screen only when all restaurant records are downloaded completely including the download of the restaurant images.

Obviously, this download operation is one of the bottlenecks. It takes time to download all images. Wouldn't it be great if we can retrieve the restaurant names and display them in the table view first? The size of the restaurant names is significantly smaller than the size of the images. It will take a fraction of the total time to download the data.

This sounds great! But what about the images? We will employ a well-known technique called *lazy loading*. Simply put, we defer the download of the images. At first, we just display a local image, bundled in the app, in the table view cell. Then we start another thread in the background to download the remote images. When the image is ready, the app updates the cell's image view. Figure 24-18 illustrates the lazy loading technique.

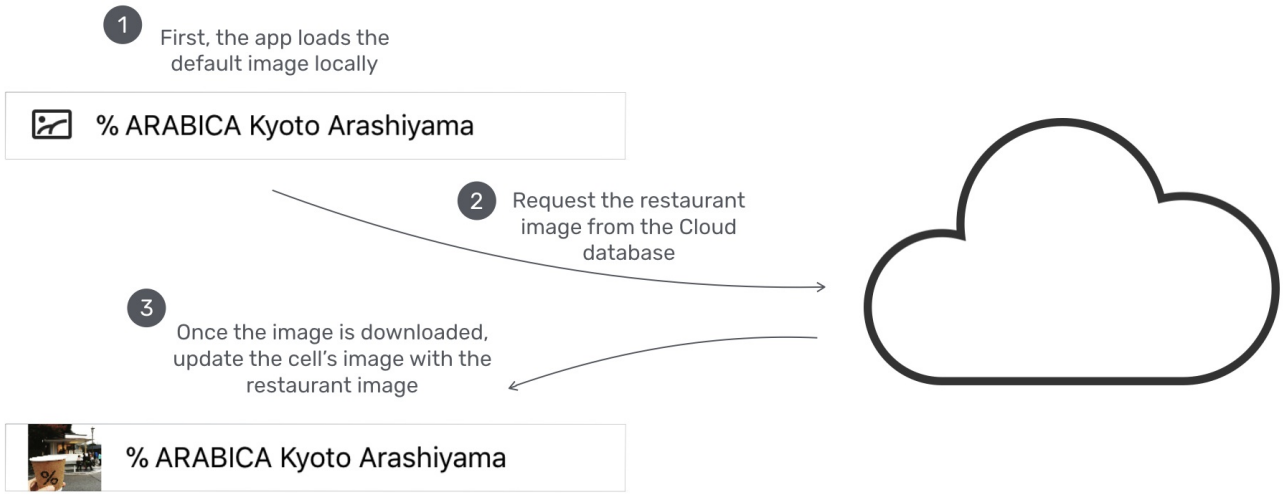


Figure 24-18. The lazy loading technique illustrated

With lazy loading in place, your user will be able to view the restaurant data almost instantly. Though the images are not ready when the data is first loaded, it improves the responsiveness of your app.

Let's see how we can implement it in the app. Because we defer the image download, we have to update the `desireKeys` property of the query operation in the `fetchRecordsFromCloud` method like this.

From:

```
queryOperation.desiredKeys = ["name", "image"]
```

To:

```
queryOperation.desiredKeys = ["name"]
```

Now we only retrieve the restaurant names and leave out the images. In the `configureDataSource()` method, we modify two things to implement lazy loading:

```
func configureDataSource() -> UITableViewDiffableDataSource<Section, CKRec
ord> {

    let cellIdentifier = "discovercell"

    let dataSource = UITableViewDiffableDataSource<Section, CKRecord>(tabl
eView: tableView) { (tableView, indexPath, restaurant) -> UITableViewCell?
in

        let cell = tableView.dequeueReusableCell(withIdentifier: cellIdent
ifier, for: indexPath)

        cell.textLabel?.text = restaurant.object(forKey: "name") as? String


        // Set the default image
        cell.imageView?.image = UIImage(systemName: "photo")
        cell.imageView?.tintColor = .black

        // Fetch Image from Cloud in background
        let publicDatabase = CKContainer.default().publicCloudDatabase
        let fetchRecordsImageOperation = CKFetchRecordsOperation(recordIDs
: [restaurant.recordID])
        fetchRecordsImageOperation.desiredKeys = ["image"]
        fetchRecordsImageOperation.queuePriority = .veryHigh

        fetchRecordsImageOperation.perRecordCompletionBlock = { (record, r
ecordID, error) -> Void in
            if let error = error {
                print("Failed to get restaurant image: \(error.localizedDe
```

```
scription)")
                return
            }

            if let restaurantRecord = record,
                let image = restaurantRecord.object(forKey: "image"),
                let imageAsset = image as? CKAsset {

                if let imageData = try? Data.init(contentsOf: imageAsset.f
ileURL!) {

                    // Replace the placeholder image with the restaurant i
mage
                    DispatchQueue.main.async {
                        cell.imageView?.image = UIImage(data: imageData)
                        cell.setNeedsLayout()
                    }
                }
            }
        }

        publicDatabase.add(fetchRecordsImageOperation)

        return cell
    }

    return dataSource
}
```

First, we set a default image (`photo`) for each cell. This image file is from the built-in SF Symbols. Since the image is loaded locally, it will be displayed almost instantly when a user taps the *Discover* tab. Secondly, we spawn a background thread to download the image asynchronously. Once the image is retrieved, the default image will be replaced by the image just downloaded. Because the size of the restaurant image is different from that of the placeholder image, we call `setNeedsLayout()` to ask the cell to lay out the view again.

The code is very similar to that of the `fetchRecordsFromCloud` method except that we use `CKFetchRecordsOperation` to fetch a specific record. Every record on the cloud has its own ID. To fetch the image of a specific restaurant record, we create a `CKFetchRecordsOperation` object with the ID of that particular restaurant record. Similar to `CKQueryOperation`, you can assign a code block (`perRecordCompletionBlock`) to execute when the record is available. In the code block, we just load the downloaded image in the cell's image view.

After the modification, you can test the app again. You'll see a great improvement in performance as the restaurant records should appear with just a little (or even no) delay. The thumbnails of the restaurants will get loaded in the background.

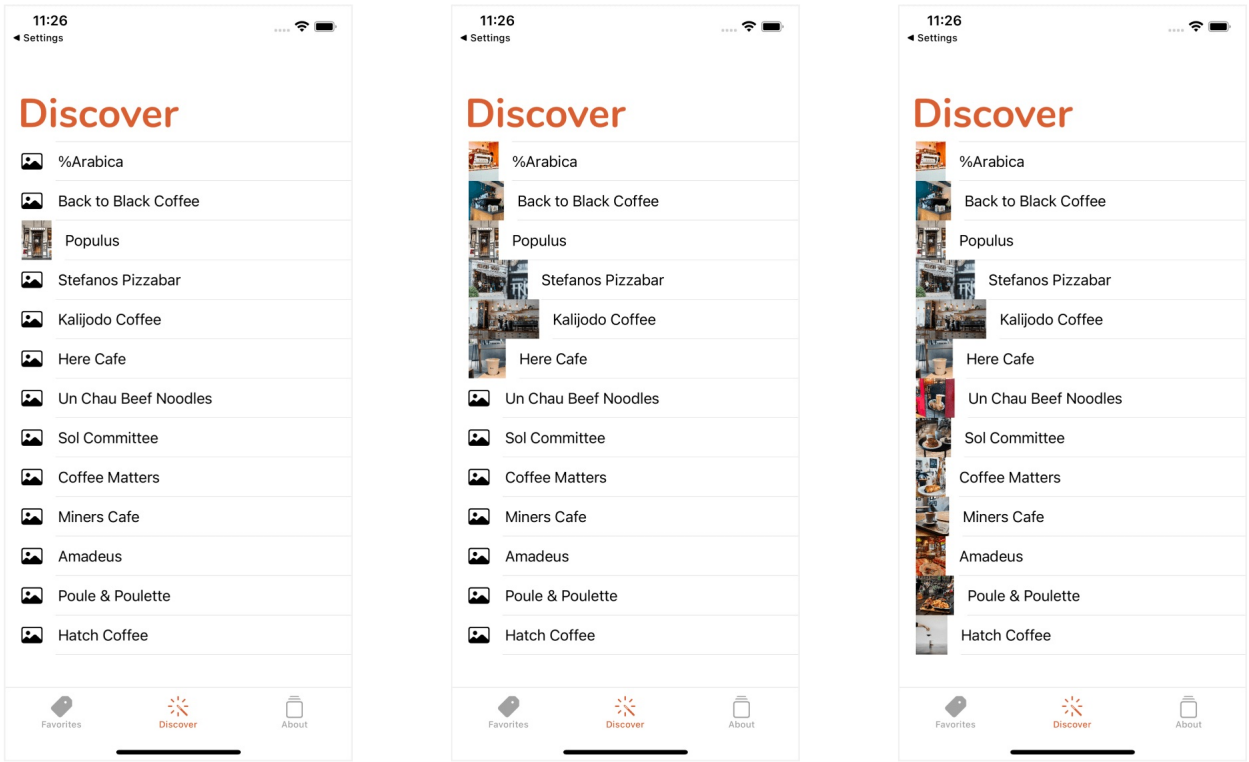


Figure 24-19. Lazy loading image in action

Caching Images Using NSCache

Have you tried to scroll through the table? Do you notice something unusual?

Every time a table view cell goes off and reappears on the screen, the cell's image will be reset back to the default image. The app then initiates the image download to retrieve the same restaurant image again. In terms of performance, this is extremely inefficient and a waste of Wi-Fi/cellular data.

Why don't we cache the image for later reuse?

Caching is one of the common approaches to improve app performance. iOS SDK provides an `NSCache` class for developers to implement simple caching. `NSCache` behaves like a dictionary and caches data in key-value pairs. You use `setObject(_:forKey:)` method to add an object to the cache using a specific key. To obtain an object of a given key from the cache, you use the `object(forKey:)` method. That's much the same as accessing items in a dictionary.

However, unlike a dictionary, it incorporates various auto-removal policies to clean up the cached objects. This ensures that it does not use too much of the system's memory.

So what do we need to do in order to cache the restaurant objects?

As mentioned earlier, CloudKit automatically downloads the images and saves them in the local file system. The images are then temporarily available for offline reading. You can use the `fileURL` property of the `CKAsset` class to retrieve the image's location. Thus, what we need to cache are the file URLs of the images.

To implement caching in the *Discover* tab, we'll make a couple of modifications:

- If the image is downloaded for the first time, we cache its file URL using `NSCache` with the record ID as the key.
- When a table view cell loads an image, we first check if the image's file URL is stored in the cache. If yes, we get the URL from the cache and load it directly. Otherwise, we'll retrieve it from iCloud.

To create a cache object using `NSCache`, insert the following line of code in the `DiscoverTableViewController` class:

```
private var imageCache = NSCache<CKRecord.ID, NSURL>()
```

`NSCache` is a generic which is capable of caching multiple types of objects. When initializing it, you have to provide the type of the key and value pair in angle bracket. In this case, the key is of type `CKRecord.ID` and the value is of type `NSURL`. In other words, this `imageCache` is designed for caching `NSURL` objects using `CKRecord.ID` as a key.

In order to handle the image caching, we need to add a conditional block to check if the image's URL is available in the cache in the `configureDataSource()` method. Update the method like this:

```
func configureDataSource() -> UITableViewDiffableDataSource<Section, CKRec
ord> {

    let cellIdentifier = "discovercell"

    let dataSource = UITableViewDiffableDataSource<Section, CKRecord>(tabl
eView: tableView) { (tableView, indexPath, restaurant) -> UITableViewCell?
in

        let cell = tableView.dequeueReusableCell(withIdentifier: cellIdent
ifier, for: indexPath)

        cell.textLabel?.text = restaurant.object(forKey: "name") as? String


        // Set the default image
        cell.imageView?.image = UIImage(systemName: "photo")
        cell.imageView?.tintColor = .black

        // Check if the image is stored in cache
```

```
        if let imageFileURL = self.imageCache.object(forKey: restaurant.re
cordID) {
            // Fetch image from cache
            print("Get image from cache")
            if let imageData = try? Data.init(contentsOf: imageFileURL as
URL) {
                cell.imageView?.image = UIImage(data: imageData)
            }

        } else {

            // Fetch Image from Cloud in background
            let publicDatabase = CKContainer.default().publicCloudDatabase
            let fetchRecordsImageOperation = CKFetchRecordsOperation(recor
dIDs: [restaurant.recordID])
            fetchRecordsImageOperation.desiredKeys = ["image"]
            fetchRecordsImageOperation.queuePriority = .veryHigh

            fetchRecordsImageOperation.perRecordCompletionBlock = { (recor
d, recordID, error) -> Void in
                if let error = error {
                    print("Failed to get restaurant image: \(error.localiz
edDescription)")
                    return
                }

                if let restaurantRecord = record,
                   let image = restaurantRecord.object(forKey: "image"),
                   let imageAsset = image as? CKAsset {

                    if let imageData = try? Data.init(contentsOf: imageAss
et.fileURL!) {

                        // Replace the placeholder image with the restaura
nt image
                        DispatchQueue.main.async {
                            cell.imageView?.image = UIImage(data: imageDat
a)
                            cell.setNeedsLayout()
```

```
                        }

                        // Add the image URL to cache
                        self.imageCache.setObject(imageAsset.fileURL! as N
SURL, forKey: restaurant.recordID)
                    }
                }
            }

            publicDatabase.add(fetchRecordsImageOperation)
        }

        return cell
    }

    return dataSource
}
```

Basically, we made a couple of changes:

1. If the URL can be found in the cache, we simply get the URL and load it right away.
2. For images that are downloaded for the first time, we need to cache the image's URL. Therefore, we add a line of code to call `setObject` of the image cache. We use the record ID of the restaurant record as key and add the URL to the cache.

That's it. If you run the app and scroll through the records in the *Discover* tab, the images should be loaded from the cache if it is available. You will see the message "Get image from cache" in the console.

Pull to Refresh

After all the tweaks, the *Discover* tab should be working much better. However, there is a limitation. Once the restaurant records are loaded, there is no way to get an update.

Most modern iOS apps allow users to refresh their content through a feature called *pull-to-refresh*. The pull-to-refresh interaction was originally created by Loren Brichter. Since its invention, an endless number of apps, including Apple's Mail app, have adopted the design for content updates.

Thanks to the popularity of the pull-to-refresh feature. Apple has made a standard pull-to-refresh control in the iOS SDK. With the built-in control, it is very simple to add the pull-to-refresh feature to your app.

`UIRefreshControl` is a standard control for implementing the pull-to-refresh feature. You can simply associate a refresh control with a table view controller, which will automatically add the control to the table view. Let's see how it works.

In the `DiscoverTableViewController` class, insert the following lines of code in the `viewDidLoad` method:

```
// Pull To Refresh Control
refreshControl = UIRefreshControl()
refreshControl?.backgroundColor = UIColor.white
refreshControl?.tintColor = UIColor.gray
refreshControl?.addTarget(self, action: #selector(fetchRecordsFromCloud),
for: UIControl.Event.valueChanged)
```

`UIRefreshControl` is very simple to use. You can instantiate it and assign it to the table view controller's `refreshControl` property. That's it. The table view controller handles the task of adding the control to the table's visual appearance. Similar to other view objects, you can configure the background color and tint color using the `backgroundColor` and `tintColor` properties.

The refresh control doesn't initiate the refresh operation when it is first created. Instead, when the table view is pulled down sufficiently, the refresh control triggers the `UIControlEvent.valueChanged` event. We have to assign an action method to this event and use it to fetch the restaurant records. To capture the event and specify the follow-up action, you can use the `addTarget` method of the refresh control object to register the `UIControlEvent.valueChanged` event. In the method call, you specify the target object (i.e. self) and the action method (i.e. `fetchRecordsFromCloud`) to handle the event. When the event is triggered, the refresh control will invoke the `fetchRecordsFromCloud` method to refresh the restaurant records.

If you've inserted the code snippet, Xcode immediately shows you an error and proposes a fix:

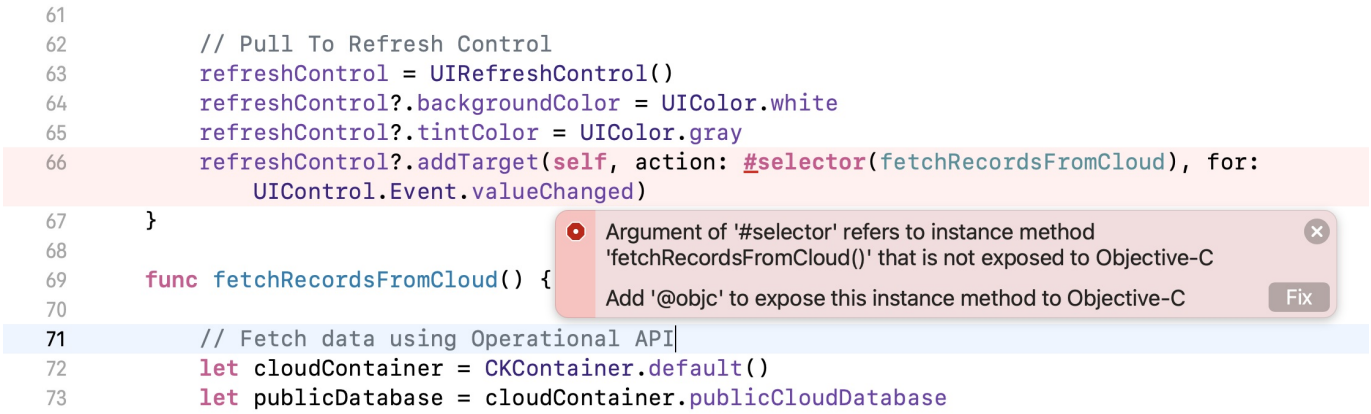


Figure 24-20. Xcode detects an error

You can click the *Fix* button and let Xcode add the `@objc` prefix for `fetchRecordsFromCloud()`.

```
@objc func fetchRecordsFromCloud() {
```

I know the error is a bit weird to you. Why Objective-C? We are now writing in Swift. While we are developing the app in Swift, not all features of the iOS SDK are written in Swift. Some are still in Objective-C. Selector is an example that isn't part of the default Swift runtime behavior. Therefore, in order to create a selector (`#selector(fetchRecordsFromCloud)`) for a method that can be called from Objective-C, we have to specify the @objc keyword in the method declaration.

If you compile and run the app, the pull-to-refresh feature should work. Try to pull down the table until it triggers the refresh. If you've added new restaurants using CloudKit dashboard, the table should show the update. However, there's still a problem; the refresh control appears even after the table content is loaded.

So how can we hide the refresh control? When the data is ready, you can simply call the `endRefreshing` method to hide the control. Apparently, we can update `queryCompletionBlock` in the `fetchRecordsFromCloud` method by adding a few lines of code:

```
if let refreshControl = self.refreshControl {
    if refreshControl.isRefreshing {
        refreshControl.endRefreshing()
    }
}
```

You can place the code after `self.spinner.stopAnimating()` . Here we check if the refresh control is still refreshing. If yes, we call `endRefreshing()` to stop the animation and hide the control.

Now run the app again. The pull-to-refresh control should disappear when the data transfer completes.

Currently, there is a bug when you refresh the data from the cloud. Some of the records are duplicated because we didn't perform any checking before appending the records to the `restaurants` array. A simple fix is to update the

`recordFetchedBlock` of the query operation in the `fetchRecordsFromCloud` method:

```
queryOperation.recordFetchedBlock = { (record) -> Void in
    if let _ = self.restaurants.first(where: { $0.recordID == record.recor
dID }) {
        return
    }

    self.restaurants.append(record)
}
```

Before appending the record to the `restaurants` array, we check if the record already appears in the array by using the record ID.

Saving Data Using CloudKit

Now that we have discussed data query, let's further explore the CloudKit framework and see how you can save data to the cloud. It all comes down to this convenience API provided by the `CKDatabase` class:

```
func save(_ record: CKRecord, completionHandler: @escaping (CKRecord?, Err
or?) -> Void)
```

The `save(_:completionHandler:)` method takes in a `CKRecord` object and uploads it to iCloud. When the operation completes, it reports the status by calling the completion handler. You can examine the error and see if the record is saved successfully.

To demonstrate the usage of the API, we'll tweak the Add Restaurant function of the FoodPin app. When a user adds a new restaurant, in addition to saving it to the local database, the record will also be uploaded to iCloud.

I'll go straight into the code and walk you through the logic along the way. In the `NewRestaurantController` class, import CloudKit:

```
import CloudKit
```

And add the following method:

```
func saveRecordToCloud(restaurant: Restaurant) {

    // Prepare the record to save
    let record = CKRecord(recordType: "Restaurant")
    record.setValue(restaurant.name, forKey: "name")
    record.setValue(restaurant.type, forKey: "type")
    record.setValue(restaurant.location, forKey: "location")
    record.setValue(restaurant.phone, forKey: "phone")
    record.setValue(restaurant.summary, forKey: "description")

    let imageData = restaurant.image as Data

    // Resize the image
    let originalImage = UIImage(data: imageData)!
    let scalingFactor = (originalImage.size.width > 1024) ? 1024 / origina
lImage.size.width : 1.0
    let scaledImage = UIImage(data: imageData, scale: scalingFactor)!

    // Write the image to local file for temporary use
    let imageFilePath = NSTemporaryDirectory() + restaurant.name
    let imageFileURL = URL(fileURLWithPath: imageFilePath)
    try? scaledImage.jpegData(compressionQuality: 0.8)?.write(to: imageFil
eURL)

    // Create image asset for upload
    let imageAsset = CKAsset(fileURL: imageFileURL)
    record.setValue(imageAsset, forKey: "image")

    // Get the Public iCloud Database
    let publicDatabase = CKContainer.default().publicCloudDatabase
```

```
    // Save the record to iCloud
    publicDatabase.save(record, completionHandler: { (record, error) -> Vo
id  in

        if error != nil {
            print(error.debugDescription)
        }

        // Remove temp file
        try? FileManager.default.removeItem(at: imageFileURL)
    })
}
```

The `saveRecordToCloud` method takes in a `Restaurant` object, which is the object to save. We first transform the `Restaurant` object into a `CKRecord` object by instantiating a `CKRecord` of the `Restaurant` record type and setting its name, type and location values.

The restaurant image requires a bit of work. First, we don't want to upload a super-high resolution photo. We would like to scale it down before uploading. The `UIImage` class allows us to create an object with a certain scaling factor. In this case, any photo with a width larger than 1024 pixels will be resized.

As you know, you use `CKAsset` object to represent an image on the cloud. To create the `CKAsset` object, we have to provide the file URL of the scaled image. So we save the image in the temporary folder. You can use the `NSTemporaryDirectory` function to get the path of the temporary directory. By combining the path with the restaurant name, we have the temporary file path of the image. We then use `jpegData(compressionQuality:)` function of `UIImage` to compress the image data and call the `write` method to save the compressed image data as a file.

With the scaled image ready for upload, we can create the `CKAsset` object using the file URL. Lastly, we get the default public database and save the record to the cloud using the `save` method of `CKDatabase`. In the complete handler, we clean up the temporary file just created.

The `saveRecordToCloud` method is now ready. We will call the method in the `saveButtonTapped(sender:)` method of the `NewRestaurantController` class so that we upload the record to the public database. Insert the following line of code in the `saveButtonTapped(sender:)` method and place it right before `dismiss(animated:completion:)`:

```
saveRecordToCloud(restaurant: restaurant)
```

You're ready to go! Hit the Run button and test the app. Click the + button to add a new restaurant. Once you save the restaurant, go to the *Discover* tab and you should find the new restaurant there. If it doesn't appear, wait for a few seconds, and pull-to-refresh the table again. Alternatively, you can go up to CloudKit Dashboard to reveal the new record.

In case if you ended up with the following error in the console, this means you do not have the *Write* permission to save the restaurant record.

```
Optional(<CKError 0x6000037a1f80: "Permission Failure" (10/2007); server m
essage = "Operation not permitted"; uuid = BD8A4F23-97EA-467F-8183-277D6F2
D63D0; container ID = "iCloud.com.appcoda.FoodPinV5">)
```

To fix the issue, you have to change the permission of the *Restaurant* type in your cloud container. Therefore, go up the CloudKit dashboard and select your container. In the top menu, choose *Schema* and then select *Security Role*.

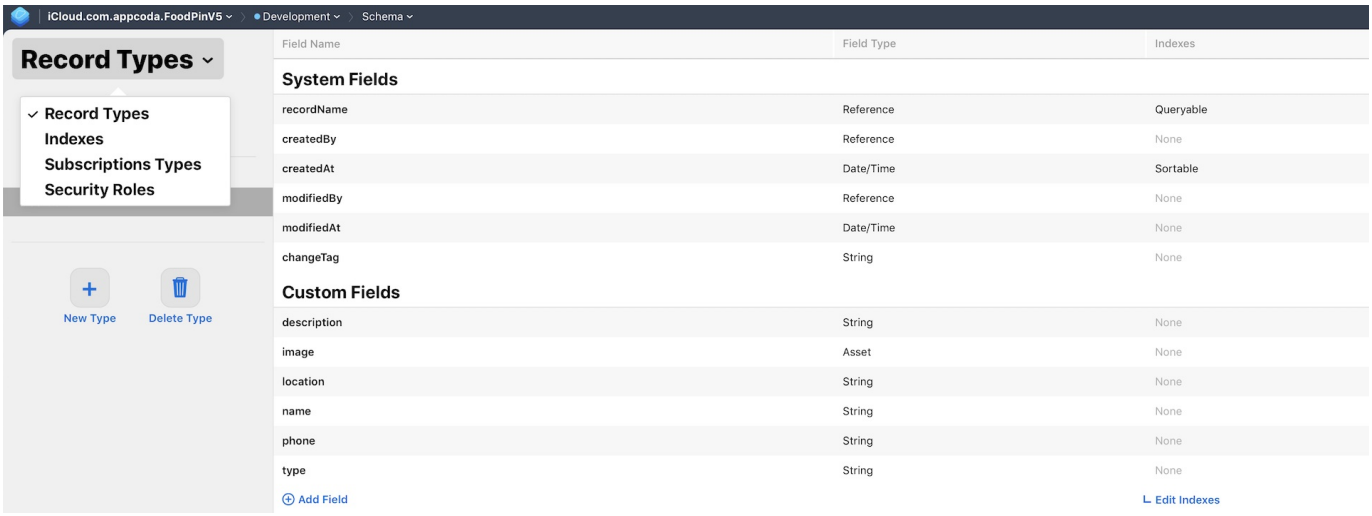


Figure 24-21. Setting the Security Roles

Next, choose *Authenticated* to set the permission of this role. By default, only the *Create* permission is enabled. To fix the issue, you have to enable both *Read* and *Write* permissions for the authenticated users.

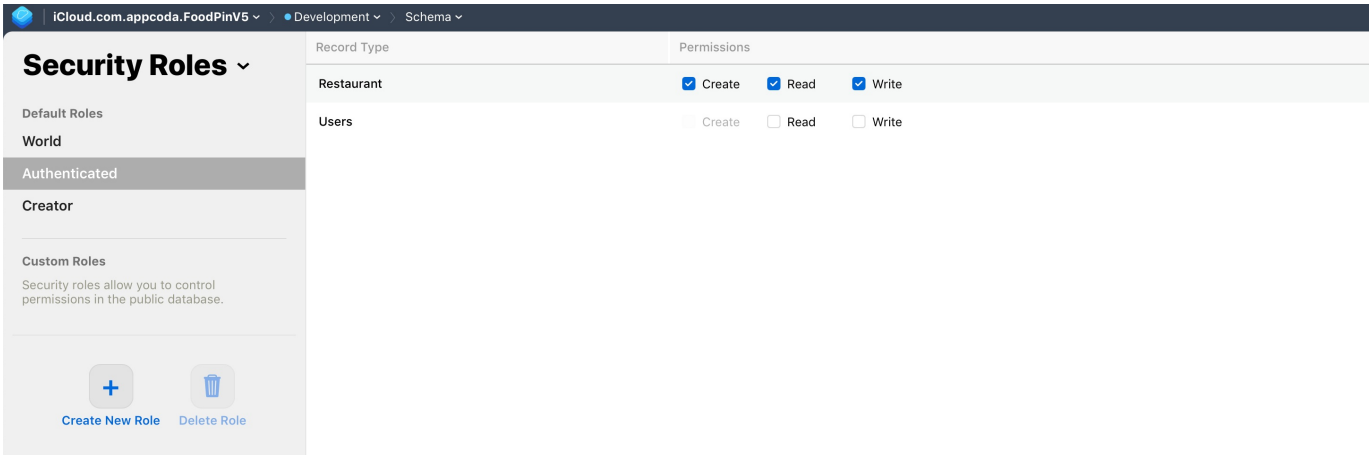


Figure 24-22. Setting the permission of the Authenticated roles

Once you save the role, you can test the app again. If you've already signed into iCloud on the simulator, you should be able to save the restaurant record to the cloud.

Sorting the Result by Creation Date

One problem with the *Discover* feature is that the restaurants are not in any order. As a user, you may want to view the new restaurants shared by other app users. That means, we need to arrange the results in reverse chronological order.

Sorting has been built into the `CKQuery` class. Here is the code snippet for creating the search query in the `fetchRecordsFromCloud` method of `DiscoverTableViewController`:

```
// Prepare the query
let predicate = NSPredicate(value: true)
let query = CKQuery(recordType: "Restaurant", predicate: predicate)
```

The `CKQuery` class actually provides a property named `sortDescriptor`. Earlier, we didn't set the sort descriptor. In this case, CloudKit returns an unordered collection of restaurant records. To request CloudKit to sort the records in reverse chronological order, you need to insert a line of code:

```
query.sortDescriptors = [NSSortDescriptor(key: "creationDate", ascending:
false)]
```

This creates an `NSSortDescriptor` object using the `creationDate` key (which is a property of `CKRecord`) and sets the order to *descending*. When CloudKit performs the search query, it will order the results by creation date. You can run now the app again and add a new restaurant. Once saved, go to the *Discover* tab and the restaurant just added should appear first.

Exercise

This exercise is designed to refresh your knowledge of table cell customization and self sizing cells. Presently, the cell in the *Discover* tab only displays the name and thumbnail of a restaurant. Modify the project so that the cell shows the restaurant's location and type. Figure 24-23 displays a sample screenshot.



Figure 24-23. Redesigning the discover UI

Summary

Wow! You've made a social network app for sharing restaurants. This is a huge chapter; by now you should understand the basics of CloudKit. With CloudKit, Apple has made it so easy for iOS developers to integrate their apps with the iCloud database. The service is completely free, as long as, you've enrolled in the Apple Developer Program ($99 per year).

Along with the introduction of CloudKit JS, you are able to build a web app for users to access the same containers as your iOS app. This is a huge deal for developers. That said, CloudKit isn't perfect. CloudKit is an Apple product. I don't see any possibilities that the company would open up the service to other platforms. If you want to create a cloud-based app for both iOS and Android, CloudKit may not be your first choice. You may want to explore Google's Firebase, the open-sourced Parse server, Microsoft's Azure, or Kinvey.

If your primary focus is on the iOS platform, CloudKit holds a lot of potential for both you and your users. I encourage you to adopt CloudKit in your next app.

For your reference, you can download the complete Xcode project from http://www.appcoda.com/resources/swift53/FoodPinCloudKit.zip.

For the solution to the exercise, you can download the project from http://www.appcoda.com/resources/swift53/FoodPinCloudKitExercise.zip.

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.

Internationalizing Your App

The first step in building a localized app is *internationalization*. In this section, we will modify the app so that it can be easily adapted to various languages.

First, let's talk about the user-facing text of the app. There is tons of user-facing text in the source code. For convenience, we just define the strings in the source code. Figure 25-1 displays a few user-facing texts in the `RestaurantTableViewController` class. When it comes to localization, these hardcoded strings are not localizable. We have to internationalize them first.

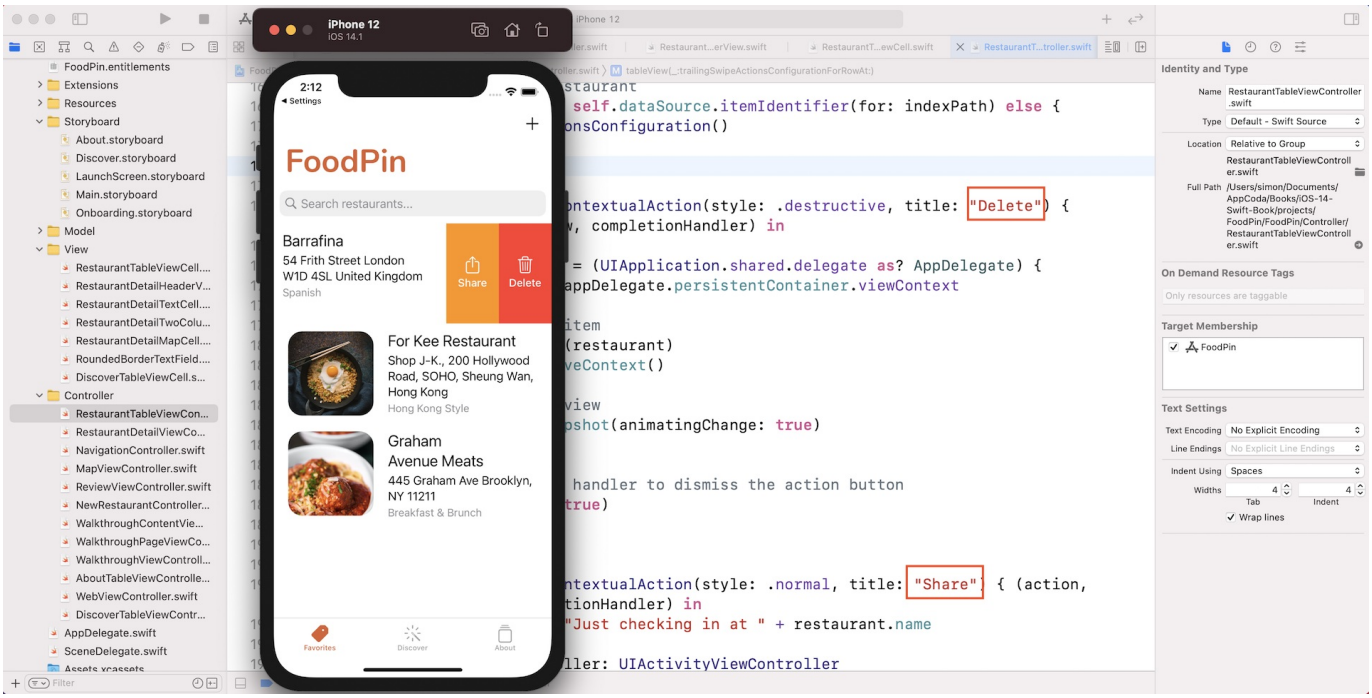


Figure 25-1. Sample user-facing text

At the heart of string internationalization is the `NSLocalizedString` macro. The macro allows you to internationalize the user-facing strings with ease. It has two arguments:

- `key` - the string to be localized
- `comment` - the string that is used to provide additional information for translation.

The macro returns a localized version of String. To internationalize the user-facing strings in your app, one thing that you need to do is wrap the existing strings with `NSLocalizedString`. Let's do some code changes so you can fully understand the macro. Take a look at the below code snippet from the `AboutTableViewController` class:

```
var sectionContent = [ [LinkItem(text: "Rate us on App Store", link: "http
s://www.apple.com/ios/app-store/", image: "store"),
                        LinkItem(text: "Tell us your feedback", link: "htt
p://www.appcoda.com/contact", image: "chat")
                        ],
                       [LinkItem(text: "Twitter", link: "https://twitter.c
om/appcodamobile", image: "twitter"),
                        LinkItem(text: "Facebook", link: "https://facebook
.com/appcodamobile", image: "facebook"),
                        LinkItem(text: "Instagram", link: "https://www.ins
tagram.com/appcodadotcom", image: "instagram")]
                        ]
```

The text is displayed in English and non-localizable. To make them language independent, you need to wrap the strings with `NSLocalizedString` like this:

```
var sectionContent = [ [LinkItem(text: NSLocalizedString("Rate us on App S
tore", comment: "Rate us on App Store"), link: "https://www.apple.com/ios/
app-store/", image: "store"),
                       LinkItem(text: NSLocalizedString("Tell us your fee
dback", comment: "Tell us your feedback"), link: "http://www.appcoda.com/c
ontact", image: "chat")
                       ],
                      [LinkItem(text: NSLocalizedString("Twitter", commen
t: "Twitter"), link: "https://twitter.com/appcodamobile", image: "twitter"
),
                       LinkItem(text: NSLocalizedString("Facebook", comme
nt: "Facebook"), link: "https://facebook.com/appcodamobile", image: "faceb
ook"),
                       LinkItem(text: NSLocalizedString("Instagram", comm
ent: "Instagram"), link: "https://www.instagram.com/appcodadotcom", image:
"instagram")]
                       ]
```

Xcode actually stores the localized strings in the `Localizable.strings` files. Each language has its own `Localizable.strings` file. Say, your user's device is using German as the default language, `NSLocalizedString` looks up the German version of the Localizable.strings file and returns the string in German.

Tip: If you don't know how an app picks a language, it actually refers to the language settings of iOS (General > International > Language). An app refers to that setting and accesses the corresponding localized resources.

That's all we need to do to internationalize the user-facing strings. In the coming section, we will discuss how to create the Localizable.strings file, and show you how to localize those labels in the storyboard.

Exercise #1

Now choose a couple of Swift files in your FoodPin project (say, `RestaurantTableViewController.swift` and `NewRestaurantController.swift`), modify all the user-facing text and wrap them with `NSLocalizedString`.

Adding Your Supported Languages

For now, your app only supports English. To support other languages, select the FoodPin project in the project navigator. Then select *Info* and choose *FoodPin* under the Project section. To add another language, click the + button under the *Localizations* section. Pick the language you want to support. For this demo, I will choose *Chinese (Traditional) (zh-Hant)*.

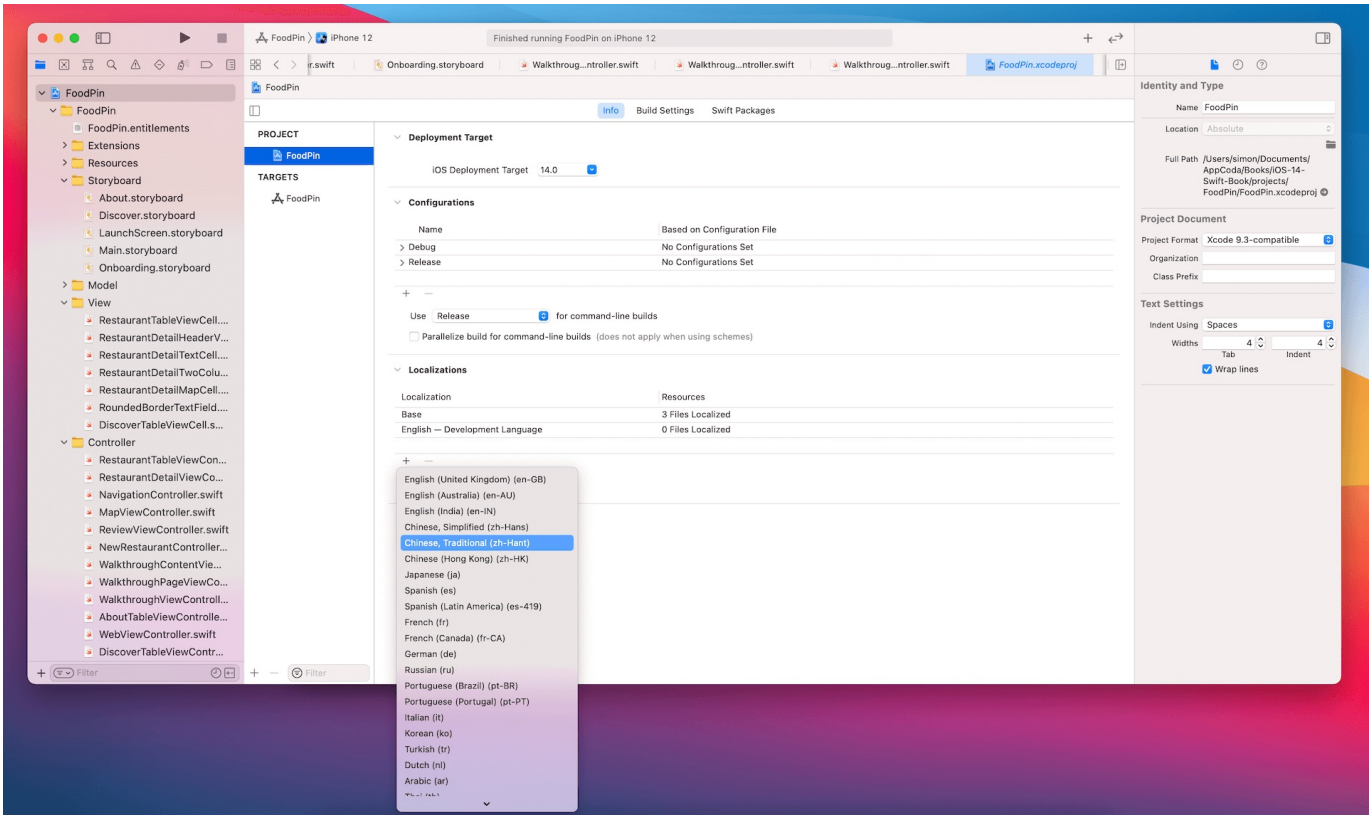


Figure 25-2. Adding a new language

Once you chose the language, Xcode will prompt you to pick the files for creating the localization. Check all the resource files because we want to localize all of them.

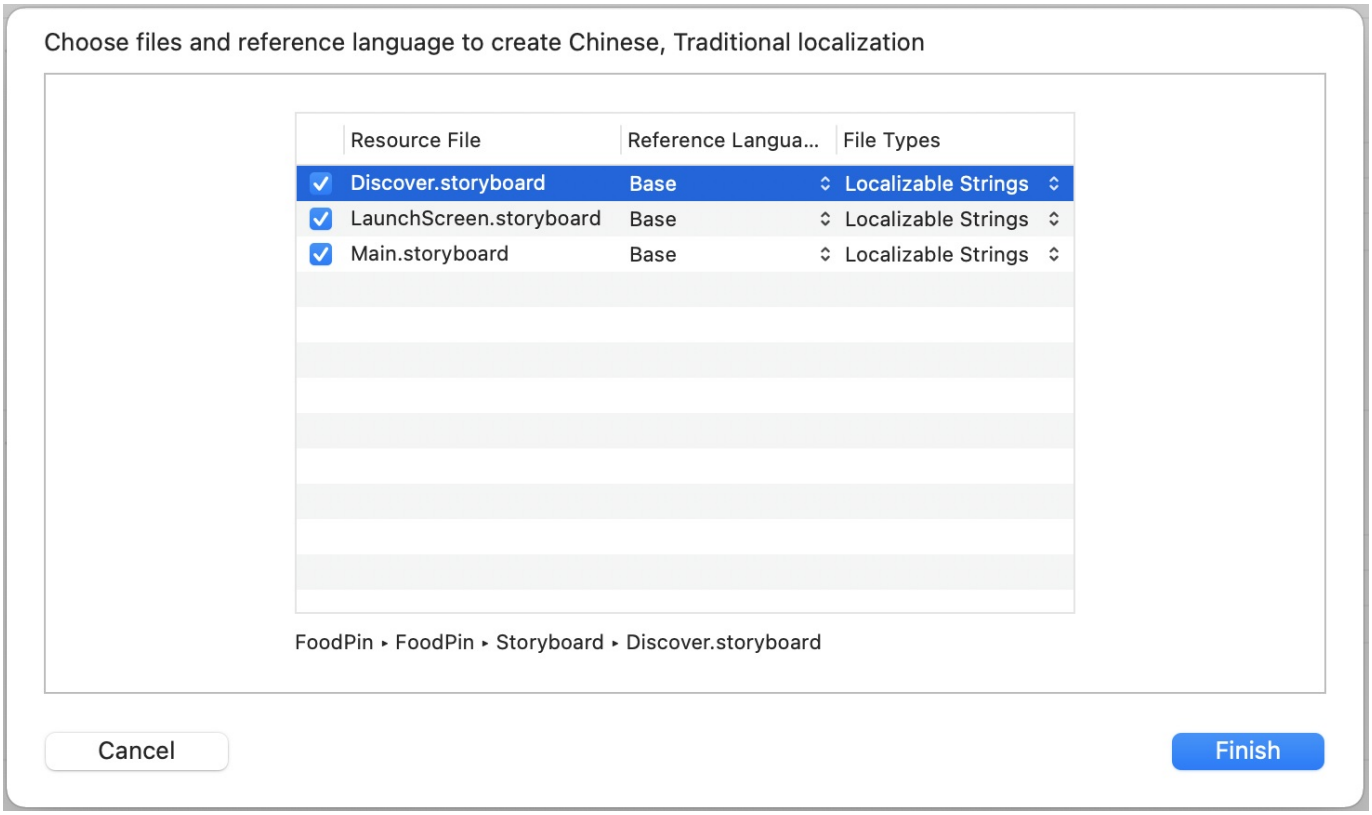


Figure 25-3. Choose the files for creating the localization

You will then see the *Chinese (Traditional)* language added to the *Localizations* section. You can continue to add more languages as long as you are going to support them. For this app, other than Chinese, we are going to support German. Therefore, repeat the procedures and add the *German* language.

Export for Localization

Xcode comes with an export feature to streamline the translation process. The export feature is intelligent enough to extract all localizable strings from your source code and Interface Builder/Storyboard files. The extracted strings will be

stored in an XLIFF file. If you haven't heard of the term, XLIFF stands for XML Localization Interchange File, which is a well-known and globally-recognized file format for localization.

To use the export feature, select the FoodPin project in the project navigator. Then head up to the Xcode menu and select *Editor* > *Export For Localization*. When prompted, choose a folder to save the localization files. Xcode then examines all your files and generates the what-so-called "Localization Catalog."

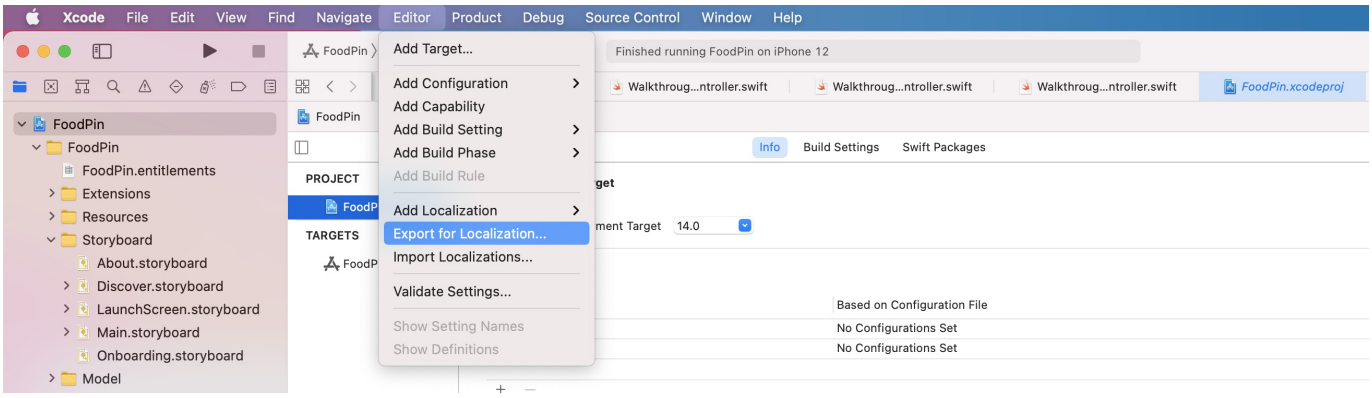


Figure 25-4. Export localization

The Localization Catalog is a new type of localization artifact with the `.xcloc` extension. Prior to version 10, Xcode only generated the XLIFF file for localization. Starting from Xcode 10, it introduced the localization catalog to provide additional contextual information and support asset localization.

Assuming you save the localization to a `FoodPin` folder, if you open Finder and go to `FoodPin`, you should find three sub-folders with the `.xcloc` extension:

- en.xcloc
- de.xcloc
- zh-Hant.xcloc

Each of the languages has its own localization catalog. Say, for Chinese (Traditional), the localization catalog is named `zh-Hant.xcloc`. Under the `.xcloc` folder, you will find a JSON file and three sub-folders:

- **contents.json** - it contains the meta data about the localization catalog like development region, target locale, tool info, and version.
- **Localized Contents** - this is the folder you would pass to your translator. It contains the .xliff file and other assets (e.g. images) for localization.
- **Source Contents** - this content of this folder is not for localization but just to provide additional contextual information to the localizers.
- **Note** - by default, this folder is empty. However, in case you want to provide additional information (e.g. screenshot) for the translators or localizers for reference, you can put the files here.

Now let's take a look at the `zh-Hant.xliff` file under *Localized Contents*. Open it with a text editor, you will find something like this:

```
<trans-unit id="Feedback" xml:space="preserve">
  <source>Feedback</source>
  <note>Feedback</note>
</trans-unit>
<trans-unit id="Find restaurants shared by your friends and other foodies"
xml:space="preserve">
  <source>Find restaurants shared by your friends and other foodies</source
>
  <note>Find restaurants shared by your friends and other foodies</note>
</trans-unit>
<trans-unit id="Follow Us" xml:space="preserve">
  <source>Follow Us</source>
  <note>Follow Us</note>
</trans-unit>
<trans-unit id="Instagram" xml:space="preserve">
  <source>Instagram</source>
  <note>Instagram</note>
</trans-unit>
```

Xcode has extracted the strings that we have wrapped with the `NSLocalizedString` macro in the previous section. Other than the above, the file should have something like this:

```
<file original="FoodPin/Storyboard/Base.lproj/Main.storyboard" source-lang
uage="en" target-language="zh-Hant" datatype="plaintext">
  <header>
    <tool tool-id="com.apple.dt.xcode" tool-name="Xcode" tool-version="11.
2" build-num="11B52"/>
  </header>
  <body>
    <trans-unit id="4gn-hN-6RK.placeholder" xml:space="preserve">
      <source>Fill in your restaurant name</source>
      <target>Fill in your restaurant name</target>
      <note>Class = "UITextField"; placeholder = "Fill in your restaurant
name"; ObjectID = "4gn-hN-6RK";</note>
    </trans-unit>
    <trans-unit id="40d-lO-w2k.text" xml:space="preserve">
      <source>ADDRESS</source>
      <target>ADDRESS</target>
      <note>Class = "UILabel"; text = "ADDRESS"; ObjectID = "40d-lO-w2k";</
note>
    </trans-unit>
```

The export feature automatically examines the storyboard and extracts all the localizable strings, including labels and button titles. Typically, you pass the file to a professional translator for translation. The translator then uses an XLIFF-enabled tool to add all the missing translations. Or, if you are like me, you can use Google Translate to do the translation and Xcode editor to edit the file. You only need to translate the text enclosed by the `<source>` tag. Content inside the `<note>` tag contains comments that we (or Xcode) added, so you do not need to translate them.

The translated text is placed inside the `<target>` tag. Below is a part of the Traditional Chinese translation file:

```
<file original="FoodPin/Storyboard/Base.lproj/Main.storyboard" source-lang
uage="en" target-language="zh-Hant" datatype="plaintext">
  <header>
    <tool tool-id="com.apple.dt.xcode" tool-name="Xcode" tool-version="11.
2" build-num="11B52"/>
  </header>
  <body>
    <trans-unit id="4gn-hN-6RK.placeholder" xml:space="preserve">
      <source>Fill in your restaurant name</source>
      <target>請填寫餐廳名字</target>
      <note>Class = "UITextField"; placeholder = "Fill in your restaurant
name"; ObjectID = "4gn-hN-6RK";</note>
    </trans-unit>
    <trans-unit id="40d-lO-w2k.text" xml:space="preserve">
      <source>ADDRESS</source>
      <target>地址</target>
      <note>Class = "UILabel"; text = "ADDRESS"; ObjectID = "40d-lO-w2k";</
note>
    </trans-unit>
```

The first line of the XML code specifies the original file and the target language of the translation. The `target-language` attribute specifies the language code of the translation. `zh-Hant` is the language code of Chinese. For German, the language code is `de`.

Import Localizations

Assuming your translator has completed the translations and passed you the localized files (download them from http://www.appcoda.com/resources/swift53/FoodPinTranslation.zip), you just need a few clicks to import the translations.

In the project navigation, select the FoodPin project. Head up to the Editor menu and select *Import Localizations*. When prompted, select the localization catalog of the translated language (e.g. `zh-Hant.xcloc`). Xcode then automatically compares the translation with the existing files and shows you the differences (see figure 25-5).

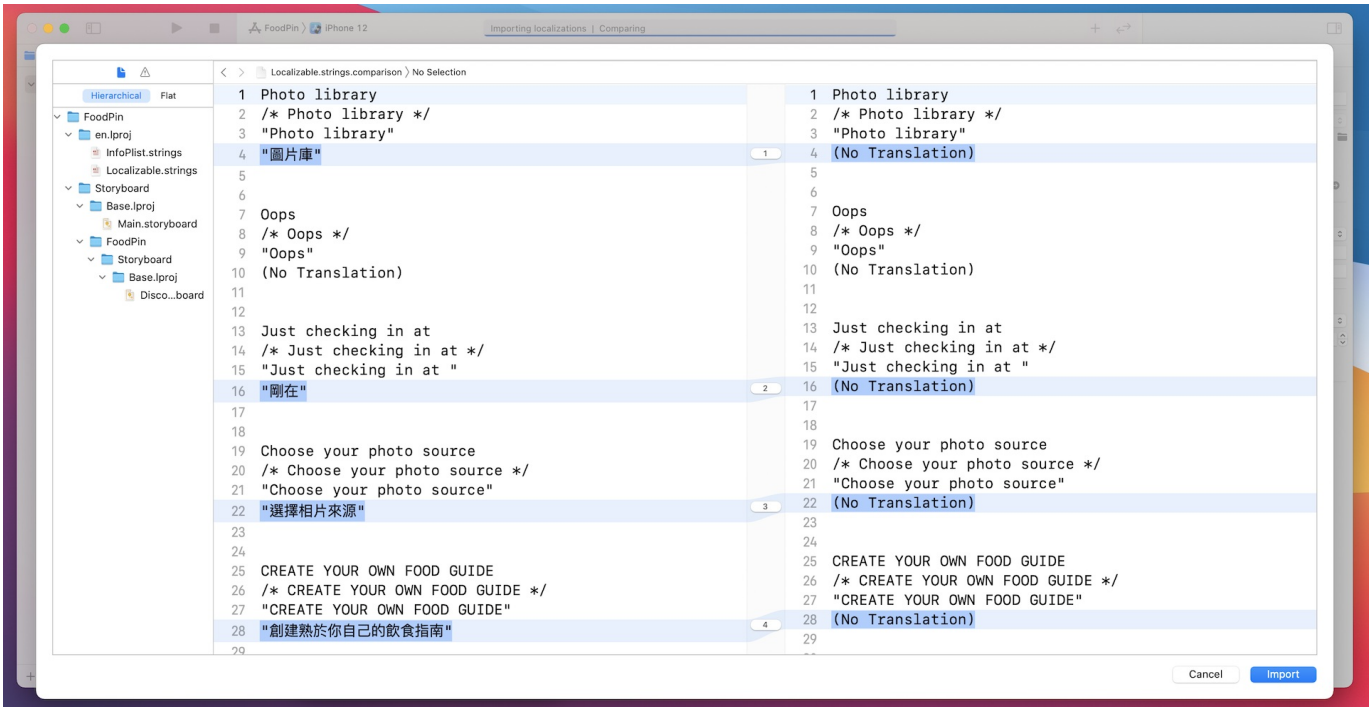


Figure 25-5. Import localizations

If you confirm the translation is correct, click the `Import` button to import it right away. To import other translation files (e.g. `de.xcloc`), repeat the same procedures to add them. When finished, Xcode displays the available localizations in the project screen. Say, if you expand `Main.storyboard`, you should find the two additional string files got translated.

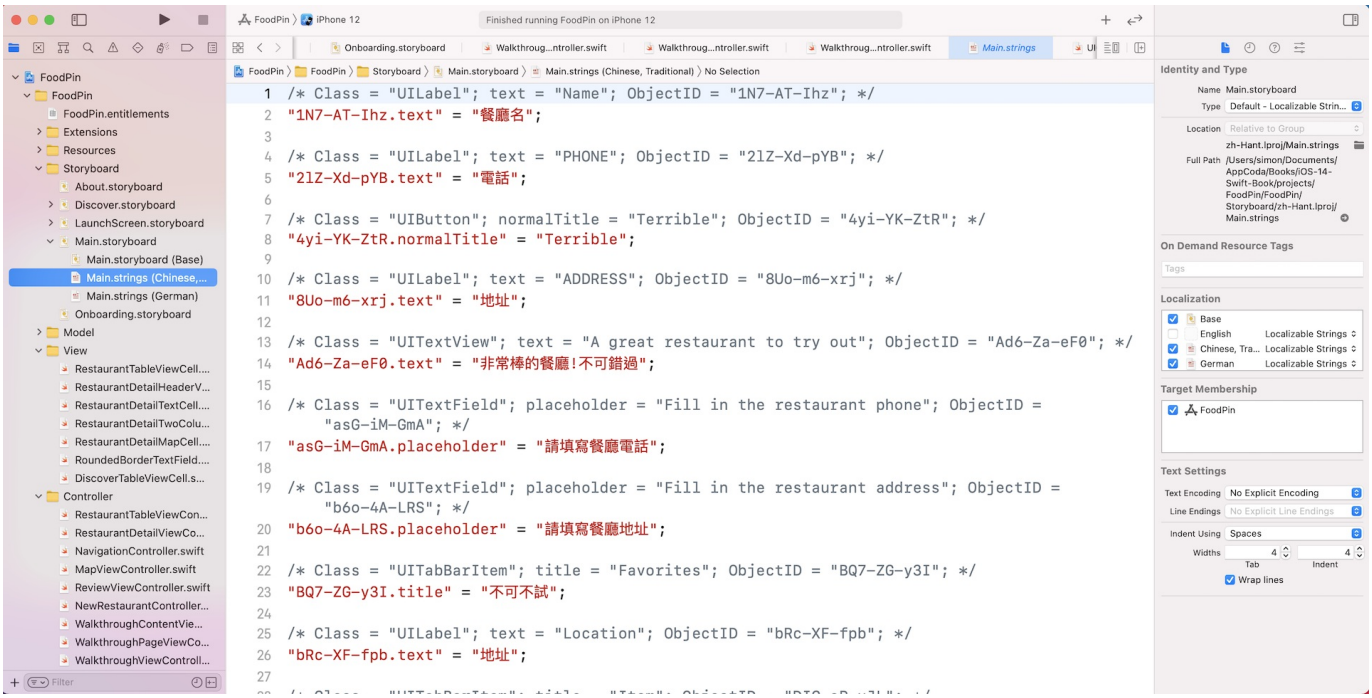


Figure 25-6. Localized strings for Main.storyboard

Before we move on to testing the localized app, let me give you some more information about base internationalization and the localized resource files. In the project navigator, you should also find the `Localizable.strings` files. These files are automatically generated when you added a new localization. The localized strings in the source code (i.e. those wrapped with `NSLocalizedString`) are stored in a language-dependent `Localizable.strings` file. As we have created the `zh-Hant` and `de` localizations, there are two versions of `Localizable.strings` files.

If you look into any of the `Localizable.strings` file (say, the Chinese version), you will find the translation of the localizable strings. At the beginning of each entry, it is the comment you put in the source code. On the left side of the equal sign, it is the key you specify when using the `NSLocalizedString` macro.

```
/* Address */
"Address" = "地址";

/* Camera */
"Camera" = "相機";

/* Choose your photo source */
"Choose your photo source" = "選擇相片來源";

/* CREATE YOUR OWN FOOD GUIDE */
"CREATE YOUR OWN FOOD GUIDE" = "創建熟於你自己的飲食指南";

/* Delete */
"Delete" = "刪除";

/* DISCOVER GREAT RESTAURANTS */
"DISCOVER GREAT RESTAURANTS" = "發掘更多值得拜訪的餐廳";

/* Facebook */
"Facebook" = "臉書";

/* Walkthrough descriptionn #3 */
"Find restaurants shared by your friends and other foodies" = "查找由您的朋友和其他美食家分享的餐館廳";
```

While you can edit the translations directly, I would recommend you to use the export feature as I have just demonstrated.

The second thing I want to talk about is *base internationalization*. As you can see from figure 25-6, `Main.storyboard` is now split into three files: the base file, the Chinese translation file, and the German translation file. But what is `Main.storyboard (Base)`?

The concept of base internationalization was first introduced in Xcode 4.5. Prior to Xcode 4.5, there was no concept of base internationalization. Xcode replicates the whole set of storyboards for each localization. For example, let's say if your app is

localized into 5 languages. Xcode generates 5 sets of storyboards for localization purpose. There is a major drawback of this design. When you need to add a new UI control in the storyboard, you'll need to add the same element in each localized storyboard. This is a tedious process, which was why Apple introduced base internationalization in Xcode 4.5.

With base internationalization, an Xcode project has only one set of storyboards that is set to the default language. This storyboard is known as the base for internationalization. Whenever the storyboard is localized into another language, Xcode only generates a `.strings` file (e.g. `Main.strings (German)`) containing all the translated text of the base storyboard. This efficiently separates the UI design from the translations.

Testing the Localized App

One way to test the localization is to change the language preference of the simulator and then run the localized app on it. Alternatively, you can utilize a preview feature of Xcode that lets you test your app in different languages and regions, both at runtime and in Interface Builder. I will go through them with you one by one.

Xcode supports preview at runtime. You can enable this feature by editing the scheme sheet and set your preferred language in the dialog box.



Figure 25-7. Edit scheme option

In the dialog box, select *Run* > *Options* and change the *application language* to your preferred language. For example, Chinese (Traditional). Click the Close button to save the setting.

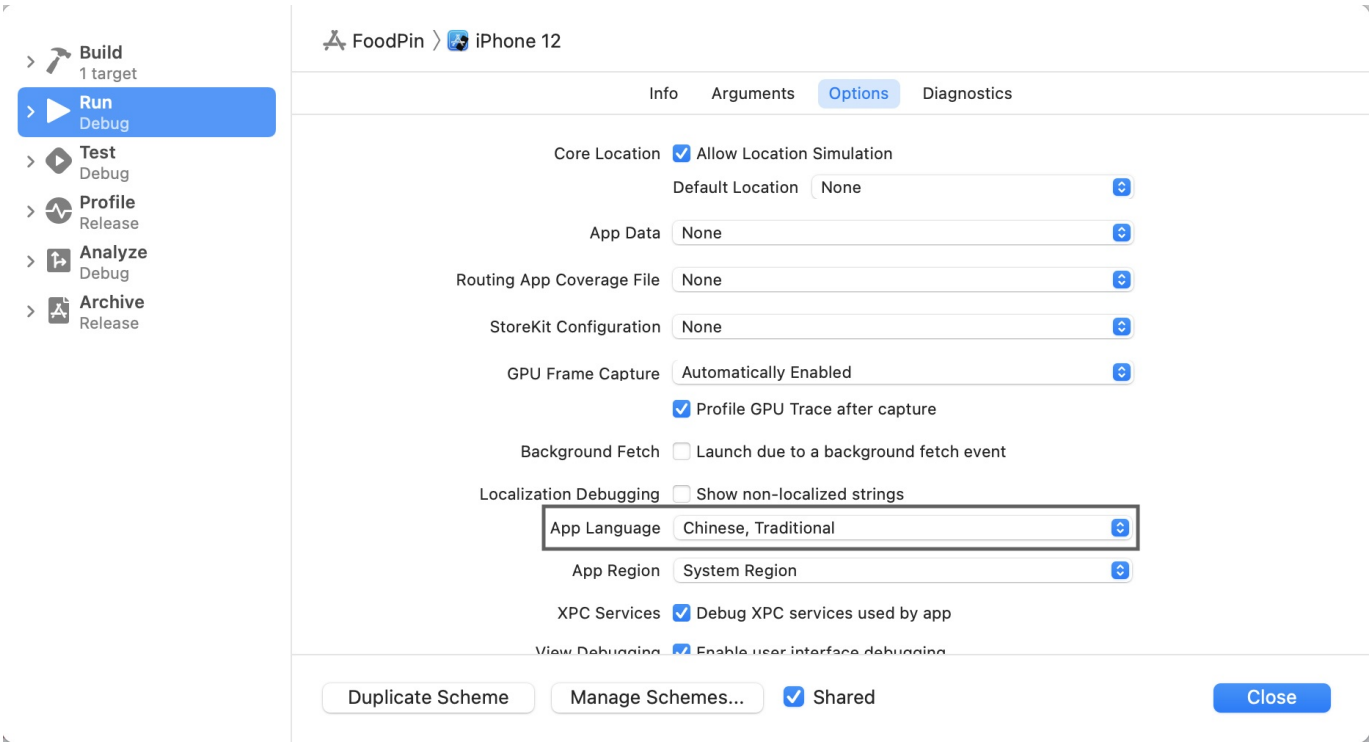


Figure 25-8. Changing the application language to your preferred language

Now click the Run button to launch the app; the language of the simulator should set to your preferred language. If you've set it to Chinese/German, your app should look like the screenshot shown in figure 25-9.

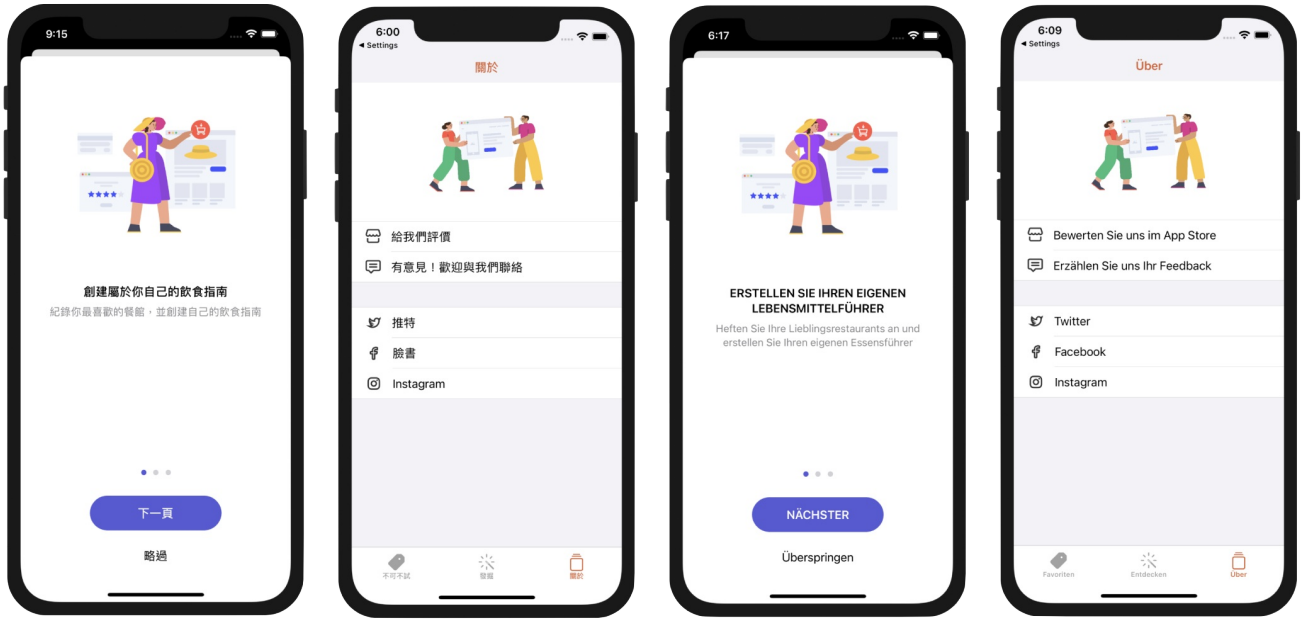


Figure 25-9. FoodPin App in Traditional Chinese and German

Alternatively, Xcode allows you to preview the localized app UI in the preview assistant. To launch the preview assistant, select `Main.storyboard` and select one of the view controllers (e.g. New Restaurant). Then click the *Adjust Editor Options* button at the top-right corner of the Interface Builder. Choose `Preview` to launch the preview assistant.

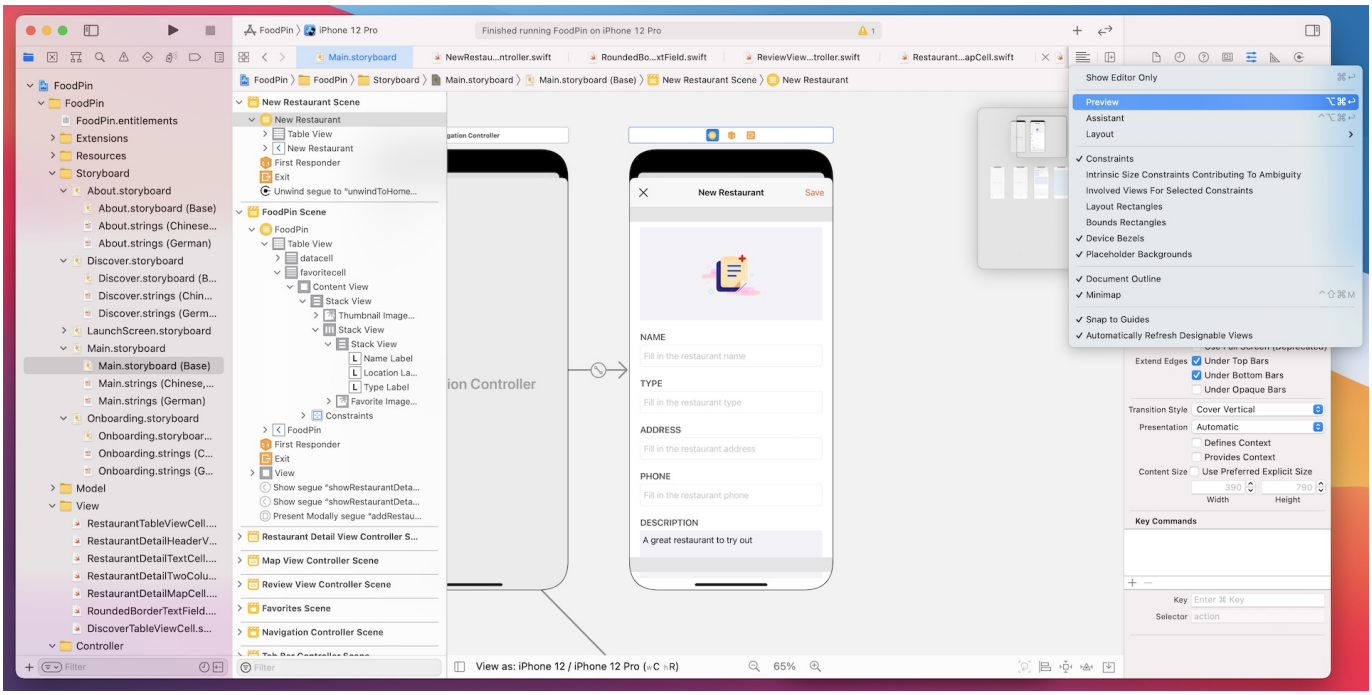


Figure 25-10. Open the preview assistant

You can then preview the localized UI by selecting the language from the language pop-up at the lower right corner of the preview window.

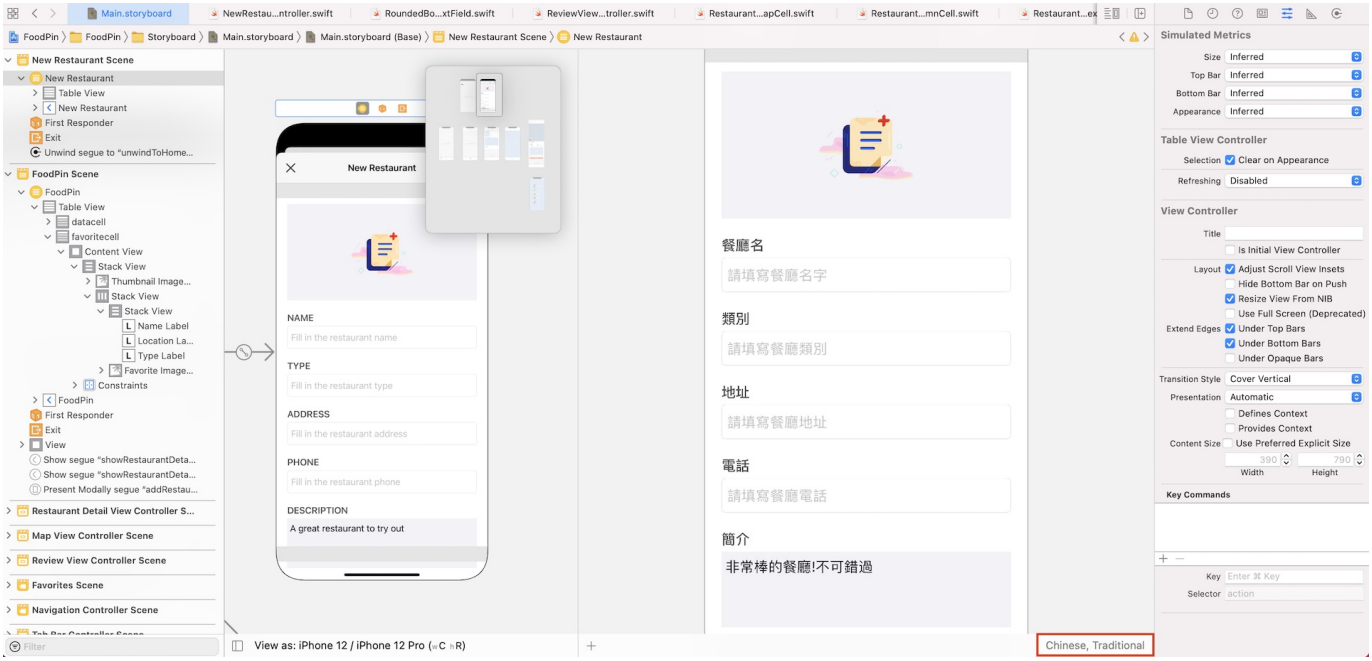


Figure 25-11. Previewing your localized app UI using Preview Assistant

Enable Localization Manually

For some storyboards (e.g. About.storyboard and Onboarding.storyboard), you will need to enable localization manually. Say, let's select the `About.storyboard` file and go to the File inspector. Scroll down to the Localization section and click the *Localize* button. Choose *Base* and click *Localize* to confirm the localization.

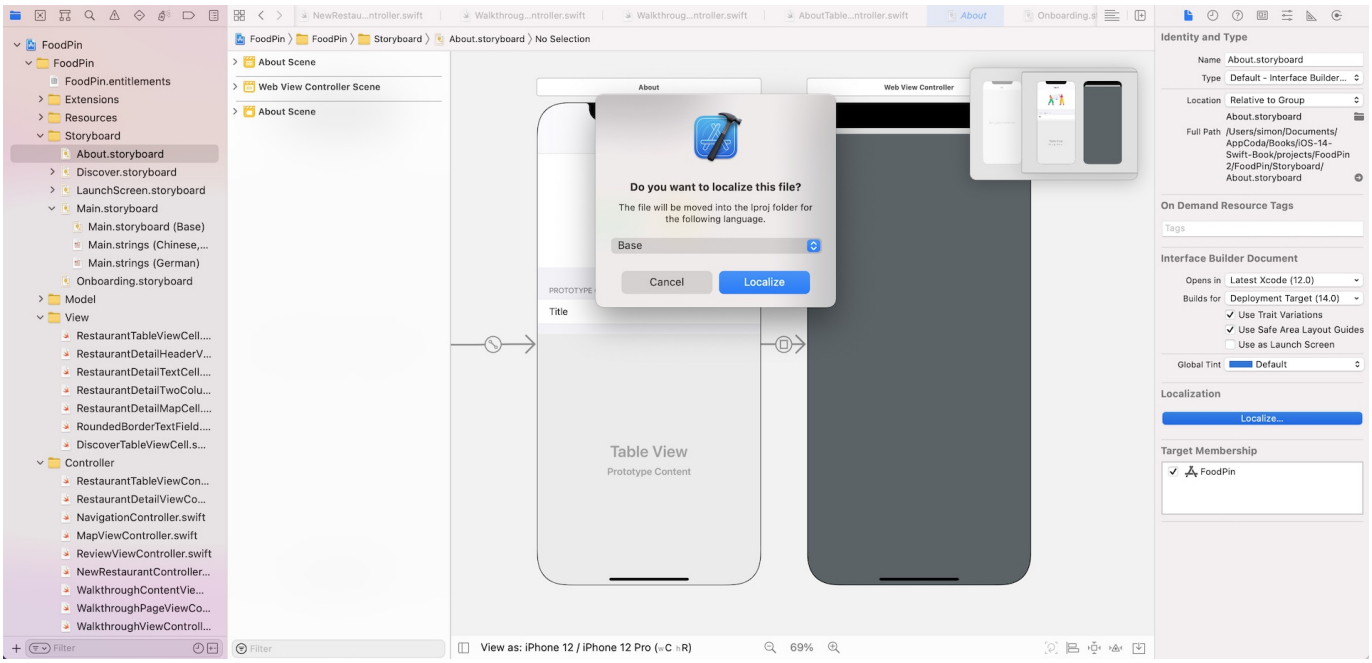


Figure 25-12. Enable localization for About.storyboard

Once you confirm to localize the storyboard, you should find the available languages for localization. Here, tick both Chinese and German checkboxes.

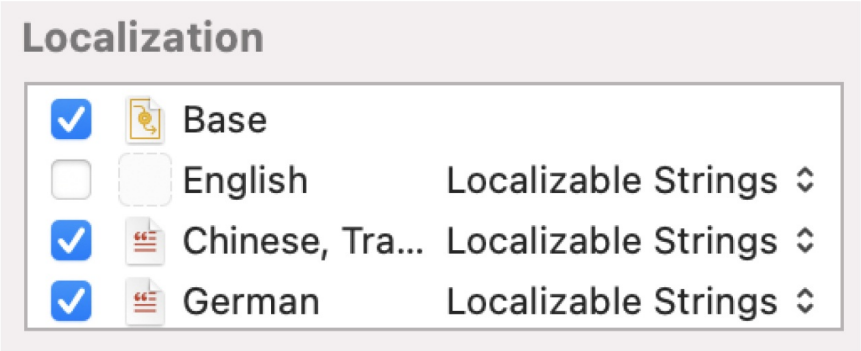


Figure 25-13. Tick the checkboxes of Chinese and German

That's it. You should now find the localized string files of `About.storyboard` in the project navigator. Now export them and pass the file to your translators for translation.

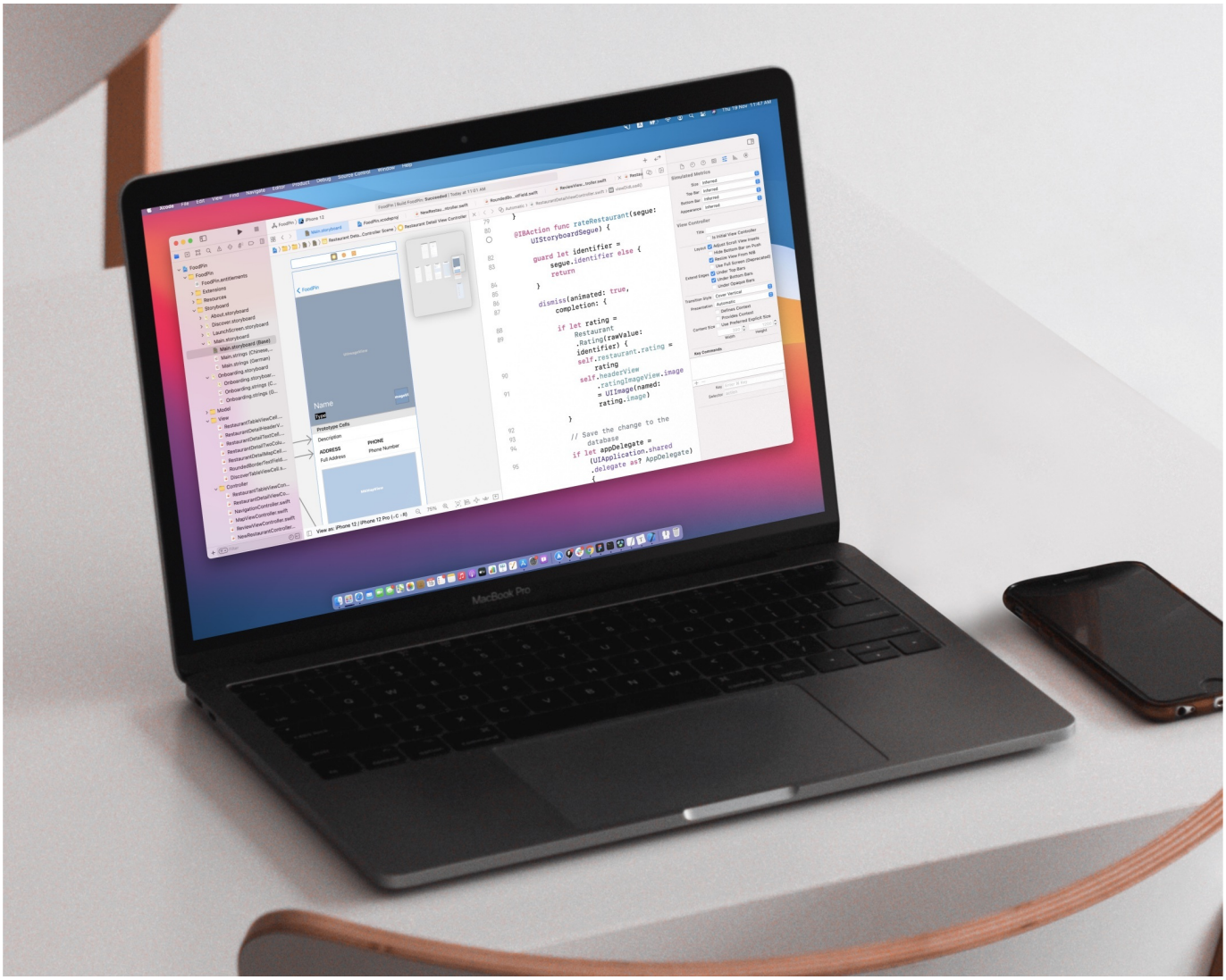
Summary

In this chapter, I have walked you through the localization process in Xcode. By using the export feature, it is pretty easy to localize an iOS app.

You've learned how to localize text and images in storyboards using the built-in localization support of Xcode. You should also know how to localize strings using the `NSLocalizedString` macro. Remember that your apps are marketed to a global audience - users prefer to have an app UI in their own language. By localizing your app, you'll be able to provide a better user experience and attract more downloads.

For reference, you can download the complete Xcode project from http://www.appcoda.com/resources/swift53/FoodPinLocalization.zip.

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 for some of you. 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.

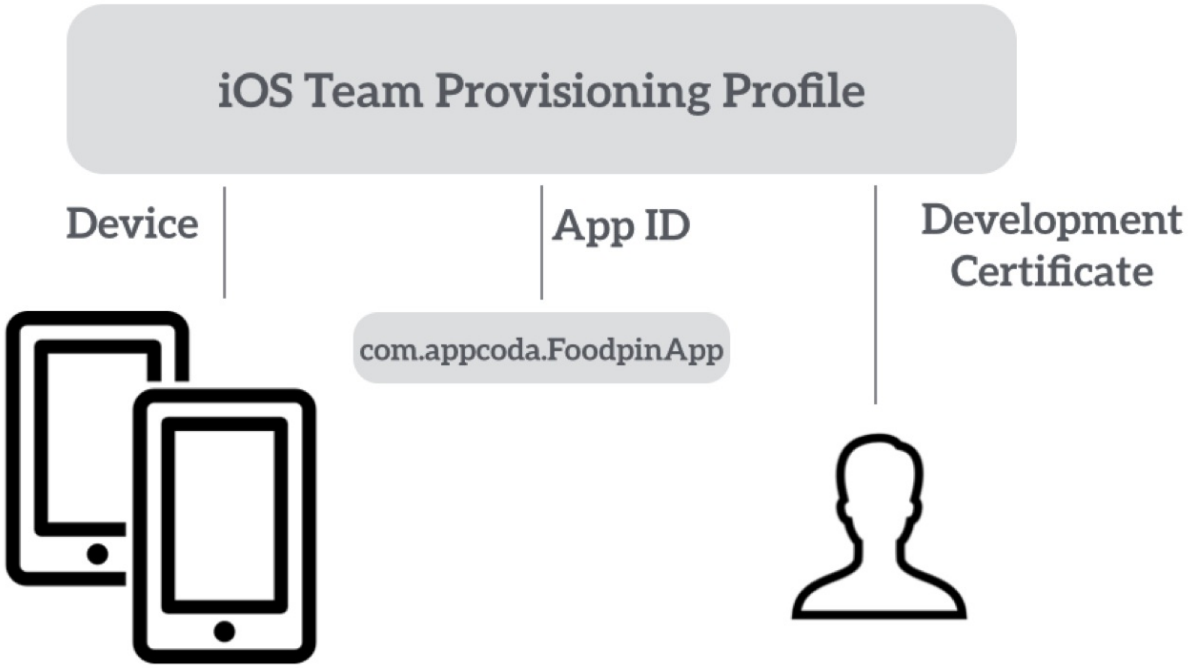
Understanding Code Signing and Provisioning Profiles

You may wonder why you need to request a certificate and go through all these complicated procedures just to put an app on a device. Not to say, it's your own app and device. The main reason is because of security. Every app deployed on an iOS device is code signed. Xcode uses both your private key and digital certificate to sign the code. By signing your code, Apple knows you're the rightful owner of the app. As you know, Apple only issues digital certificates to legitimate developers who have joined the developer program. This also prevents non-authorized developer to deploy apps on the app store.

Apple issues two types of digital certificates: *Development* and *Distribution*. To run an app on a real device, you just need the development certificate. Later, when you submit an app to App Store, you have to sign your app using a distribution certificate. Again, this is automatically taken care by Xcode. We will discuss app submission in chapter 28. For now, let's focus on deploying your app to a device.

Provisioning Profile is another term you come across in app deployment. Provisioning refers to the process of preparing and configuring an app to launch on devices. A team provisioning profile allows your app to be signed and run by all team members (if you're an indie developer, you're the only member of the team)

on their devices. Figure 26-1 shows an illustration of a sample provisioning profile.



Reviewing Your Bundle ID

Before moving on, I would like you to review your bundle ID. As discussed before, a bundle ID identifies a single app, so it must be unique. Assuming you have followed the CloudKit chapter, I believe you already created your own bundle ID. But let me remind you once again. You must not use `com.appcoda.FoodPin` (or `com.appcoda.FoodPinV5`) because it has been used by the demo app. Otherwise, you will not be able to deploy the app onto your physical device.

Therefore, make sure you change the bundle ID. It can be anything. However, if you already own a domain, use it and write the ID in reverse-DNS format (e.g. `edu.self.foodpinapp`).

> **Note:** After your app is published on the app store, you will not be able to change the bundle ID.

Automatic Signing in Xcode

Automatic signing is a new feature since the release of Xcode 8. As mentioned before, you will need to manage the following tasks before you can deploy and run your app on a real device:

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

With Automatic Signing, you no longer need to create your development certificate or provisioning profile manually. Xcode takes care of all these tasks for you.

When creating an Xcode project, automatic signing is enabled by default. If you select the FoodPin project in the project navigator and go to the FoodPin target, you should find the *Automatically manage signing* option is on.

Before we perform automatic signing, connect your iPhone (or iPad) to your Mac. In Xcode, click the simulator button, you should see your device in the simulator/device list. Please make sure you select the device because Xcode will register your iPhone as the valid test device during the signing process.

Now, go to the project option and choose the *Signing & Capabilities* section, the *Team* option is set to *None*. To use automatic signing, you have to add an account to your project. Click the *Team* option and choose *Add an account*. Fill in your Apple ID/password, and click `Add` to proceed. If you have already added your account before, you can skip this step.

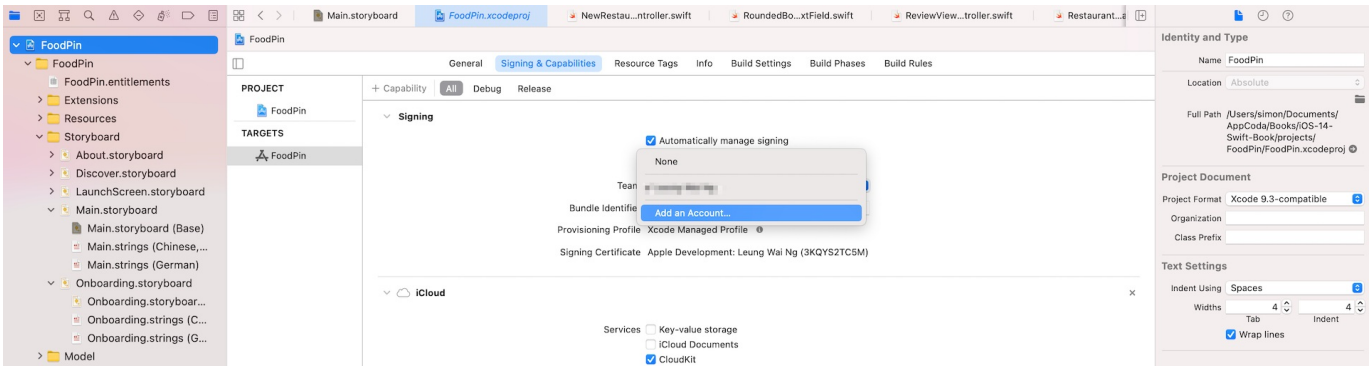


Figure 26-2. Adding your iOS developer account in Xcode

Even if you haven't enrolled in the Apple Developer Program, you can still sign in with your Apple ID.

Once you added your account, Xcode will generate the development certificate and create the provisioning profile automatically. In the signing section, you will find your signing certificate and provisioning profile. You can further click the *info* icon (next to the provisioning profile) to reveal the details.

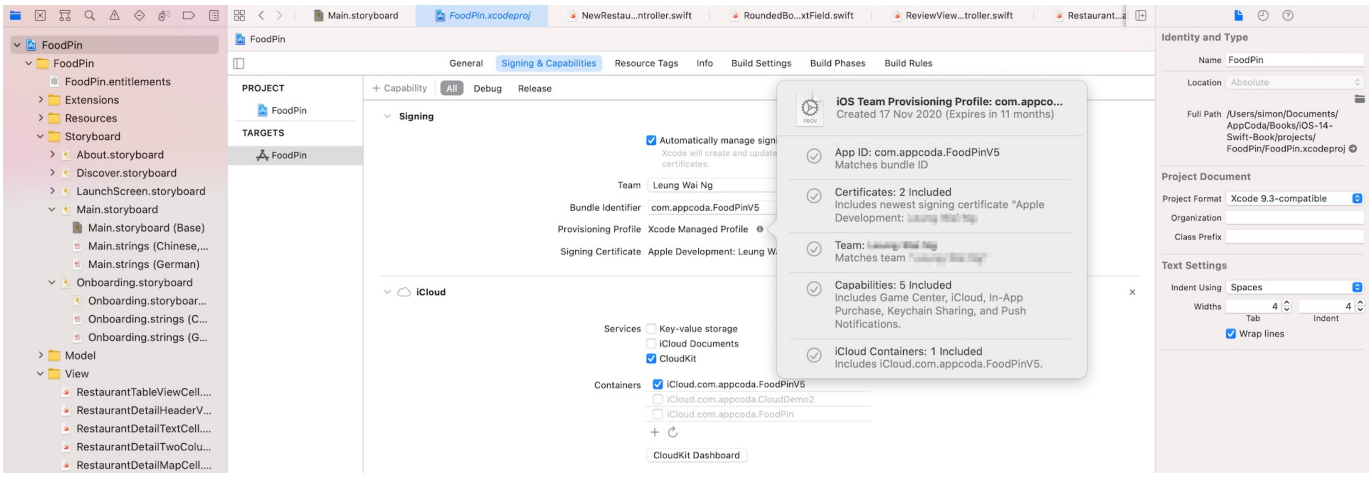


Figure 26-3. Xcode creates the provisioning profile and signing certificate for you

You can always review your App ID and provisioning profile in the Member Center. Open Safari and access https://developer.apple.com/membercenter/. Sign in with your developer account and you'll see the below screen.

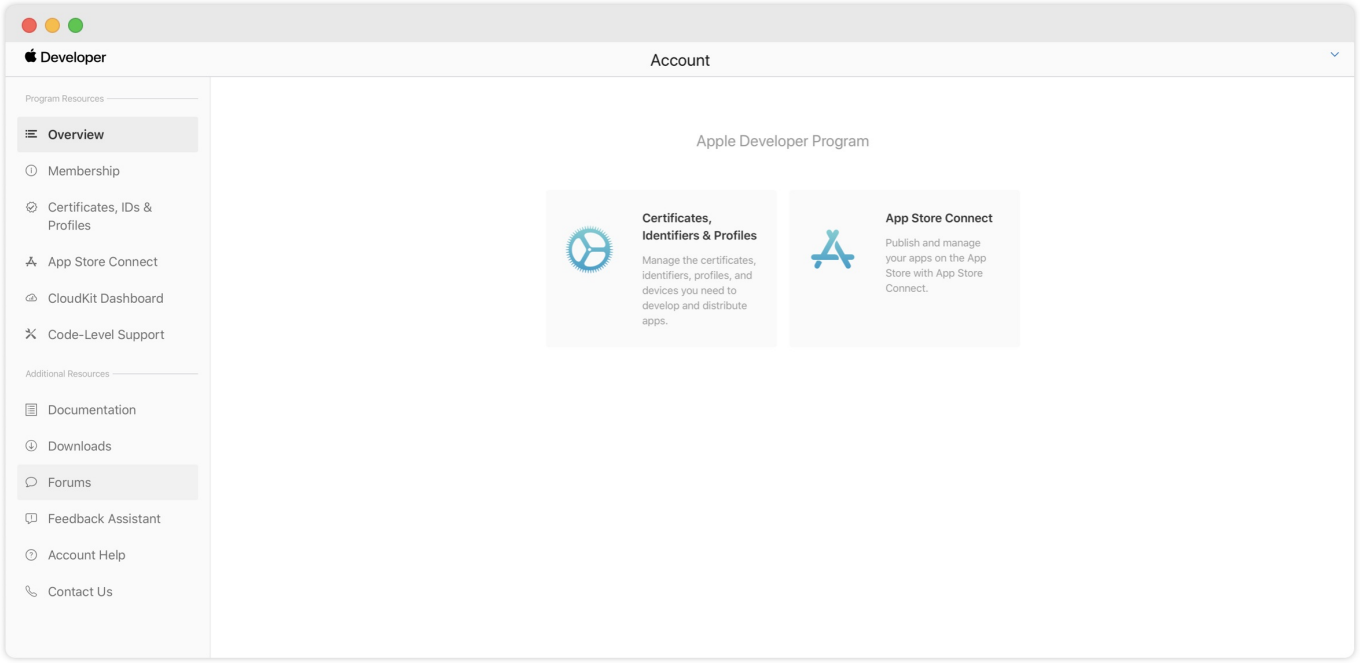


Figure 26-4. Member Center for iOS Developer

> **Note:** The "Certificates, identifiers & Profiles" will only appear if you have enrolled in the Apple Developer Program.

Select "Certificates, IDs & Profiles" and you'll navigate to another screen for managing certificates, identifiers, devices and provisioning profiles. Choose identifiers under iOS Apps and you will find the App ID (which is the same as your bundle ID) of your app. Figure 26-5 shows a sample App ID.

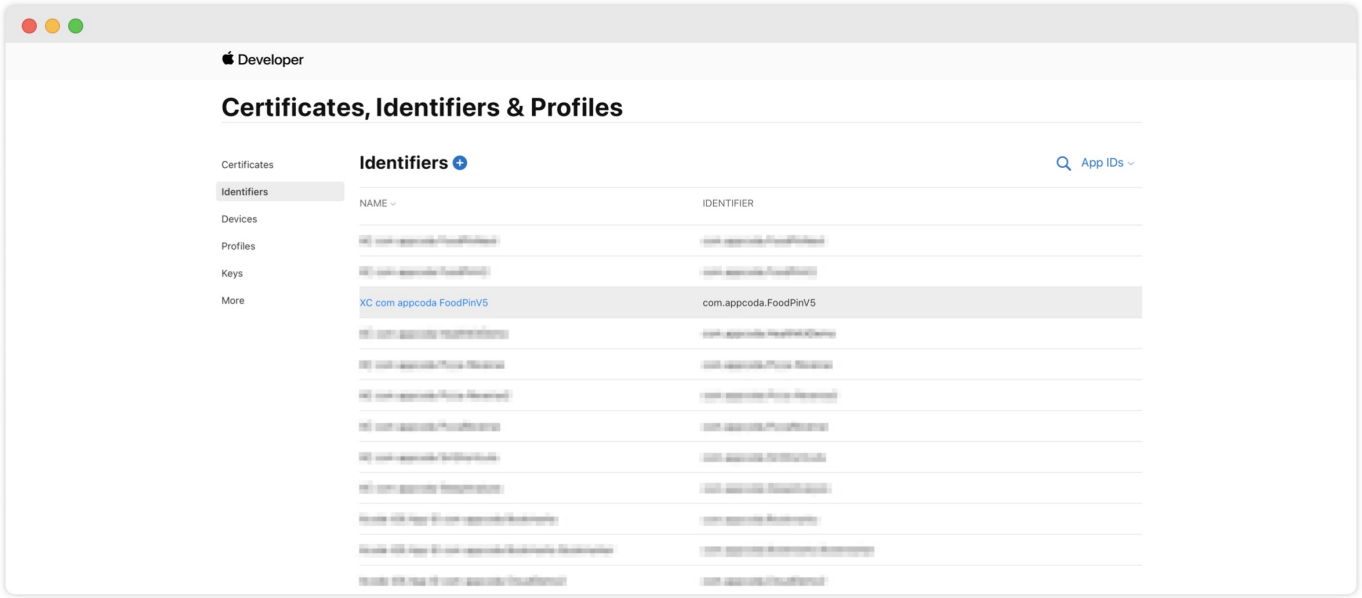


Figure 26-5. Sample App ID

Deploying the App on Your Device via USB

Assuming you've connected your iOS device to your Mac, you are ready to deploy your app and test it on your iPhone (or iPad). Click the simulator/device pop-up menu and scroll to the top of the list, choose your device.

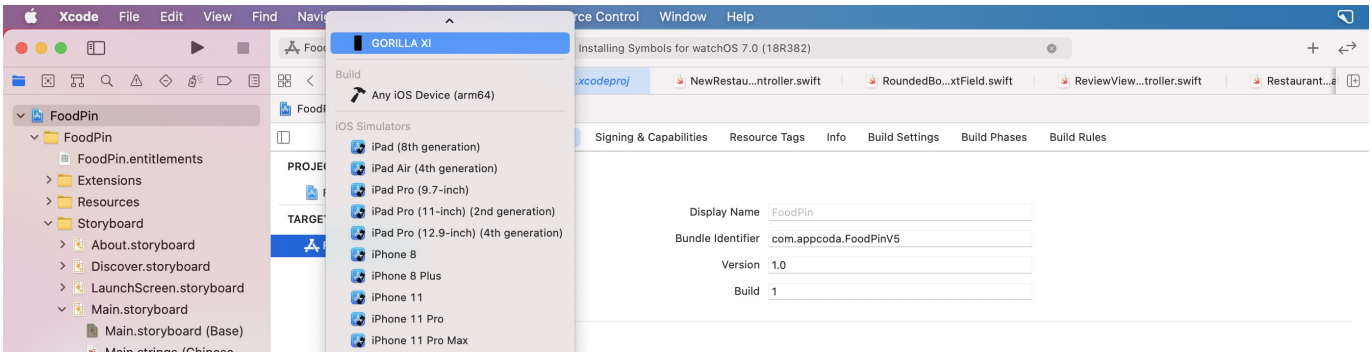


Figure 26-6. iOS device in the Scheme pop-up menu

Quick note: If your iOS device is ineligible, fix the issue before continuing. For example, if the device doesn't match the deployment target (e.g. iOS 14), upgrade the version of iOS on the device.

You're now ready to deploy and run the app on your device. Hit the Run button and start the deployment process! Remember to unlock your iPhone before deploying the app. Otherwise, you won't be able to launch it.

For the very first time running your app, Xcode may prompt you with a security error. Though your app was deployed on a registered device, due to a security restriction, Xcode cannot launch it without your authorization. You may end up with an error prompt (see figure 26-7). To fix the error, go to Settings > General > Device Management. Select the developer profile and click the `Trust <Apple ID>` option. It should prompt you for confirmation. Simply click `Trust` to proceed. Once you trust the developer, you will be able to launch the app.

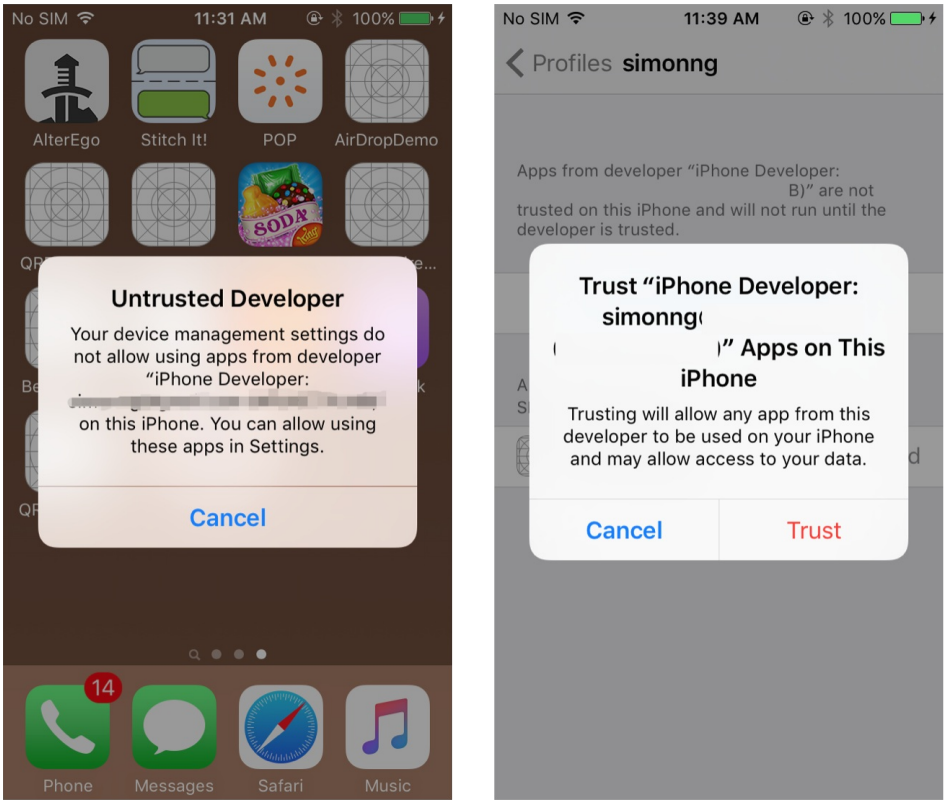


Figure 26-7. Prompt to trust the app developer (left), Trust the developer in Settings (right)

Deploying the App over Wi-Fi

From Xcode 9 and onwards, the development tool comes with one of the most anticipated features for app deployment. Xcode allows you to cut the USB cable and deploy your app to any iOS devices over Wi-Fi.

But, there is a catch. Before you can use this new feature, you have to first connect your iOS device using a USB cable and go through a simple configuration.

Assuming you have your device connected, go up to the Xcode menu and choose *Window > Devices and Simulators*.

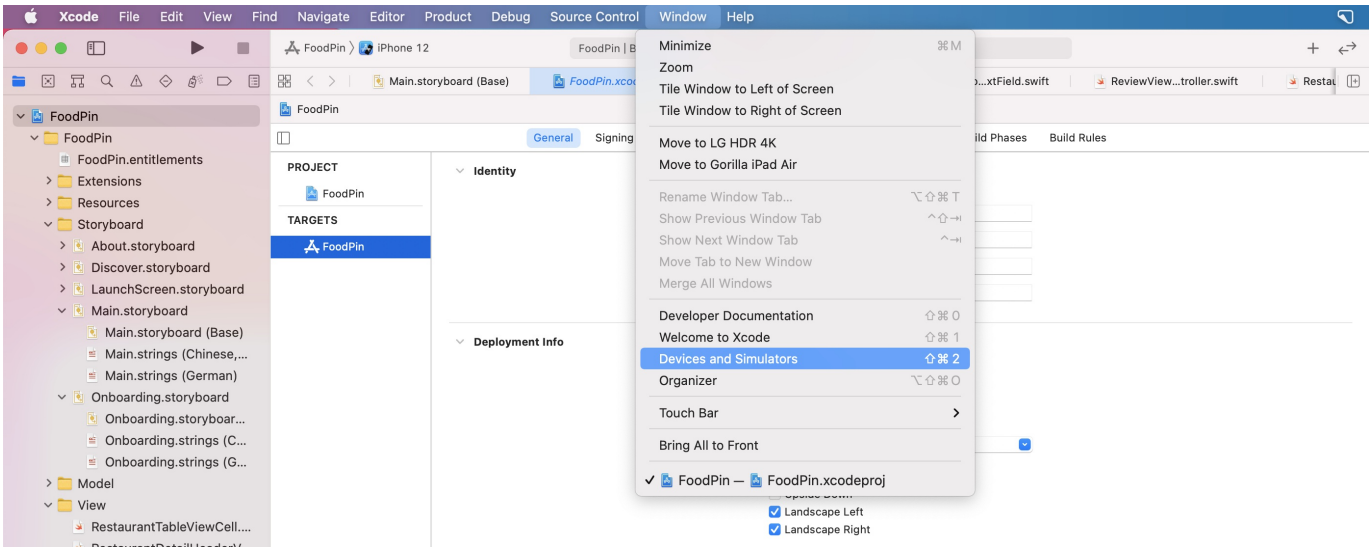


Figure 26-8. Choose the Devices and Simulators option

Choose your device under the *Connected* section. Then you should find a checkbox *Connect via network*. Tick it to enable Wi-Fi deployment.

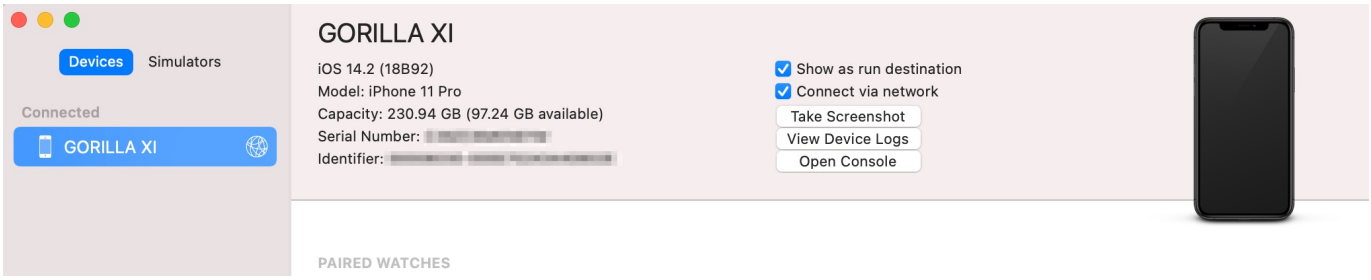


Figure 26-9. Enable Connect via network option

Now you can disconnect your iPhone (or iPad) from your Mac. Make sure your device and your Mac are connected to the same Wi-Fi network. You can then deploy your app over Wi-Fi.

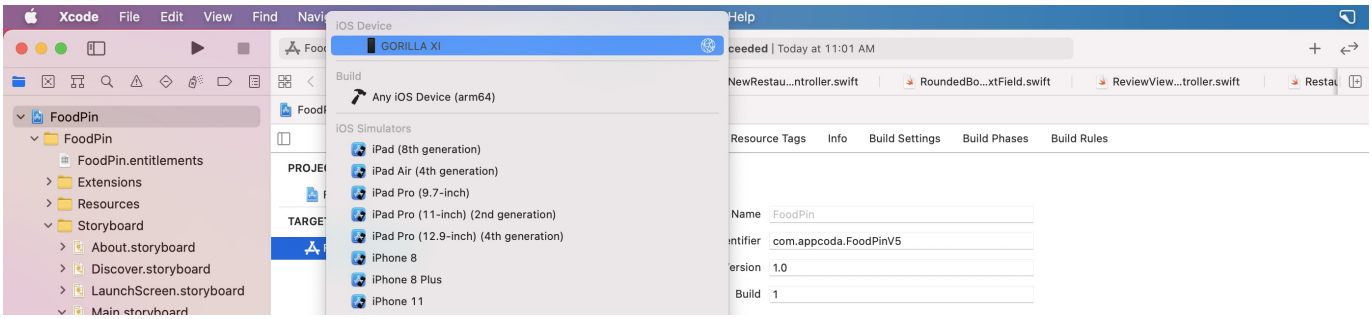


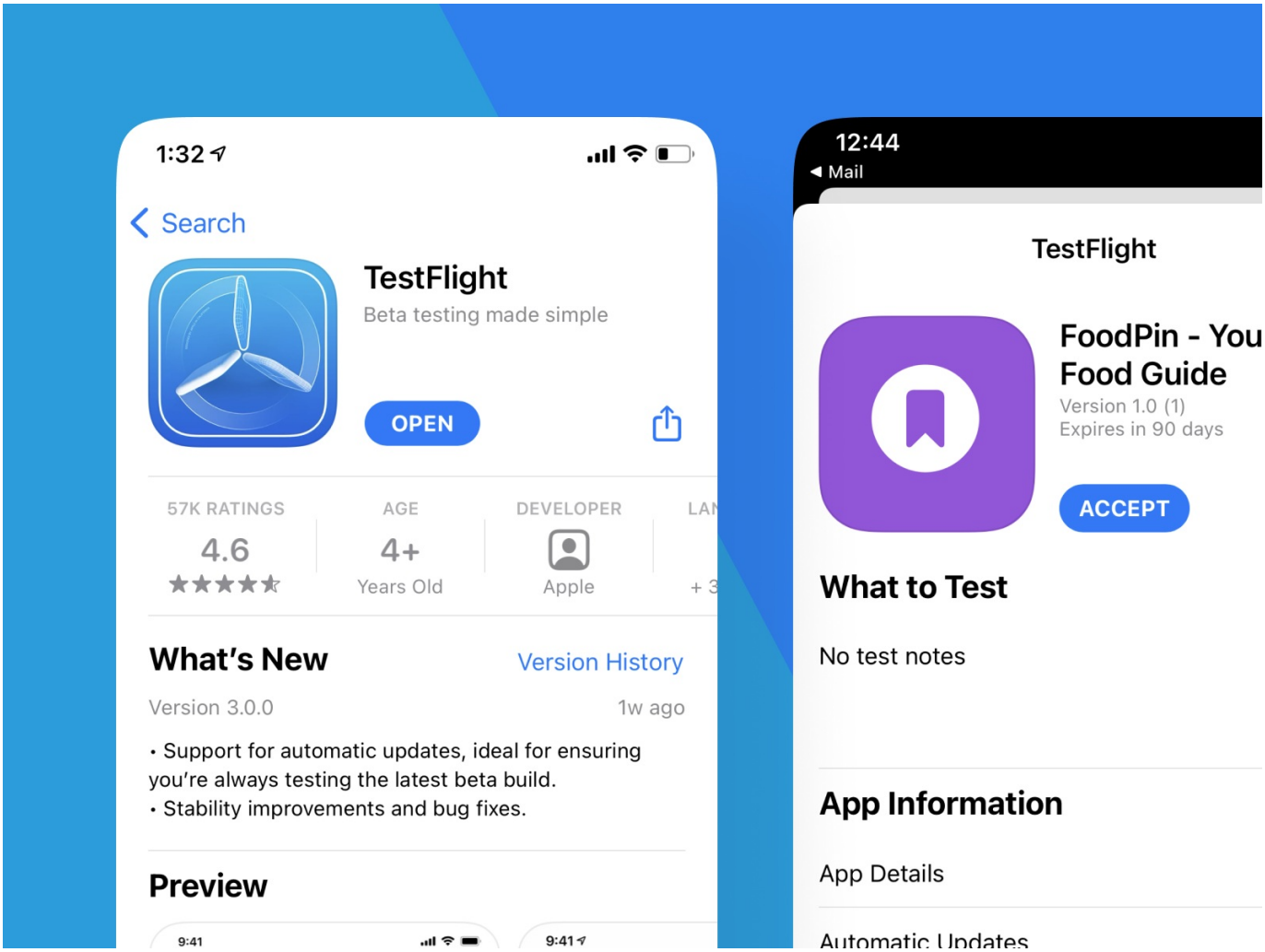
Figure 26-10. A globe icon will appear if your device is eligible for Wi-Fi deployment

Summary

Isn't it cool to run your app on a real device? You can now show off your app to your friends and family member. The latest version of Xcode makes your developer's life easier by simplifying the way to create a provisioning profile, app ID creation, and developer certificate generation. With a few clicks, you will be able to test your app on your devices. Needless to say, Wi-Fi deployment is a great add-on.

If you run into any issues of the app deployment, just post your question to our private Facebook group (https://facebook.com/groups/appcoda).

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.

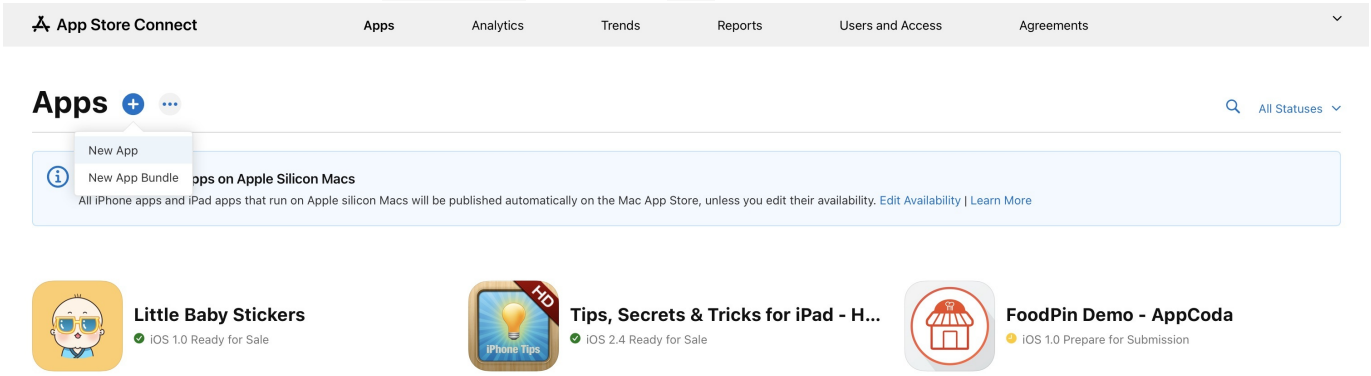
In this chapter, I will walk you through the beta test process using TestFlight. In general, we will go through the tasks below to distribute an app for beta testing:

- 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.

Creating an App Record on App Store Connect

Firstly, you need an app record on iTunes Connect before you can upload an app. iTunes Connect is a web-based application for iOS developers to manage their apps sold on App Store. If you have enrolled in the iOS Developer Program, you can

access App Store Connect at https://appstoreconnect.apple.com. Once signed into iTunes Connect, select `My Apps` and the `+` icon to create a new iOS app.



You'll be prompted to complete the following information:

- **Platform** - choose app platform, which is iOS.
- **App name** - your app name appears on App Store. It shouldn't exceed 30 characters.
- **Primary language** - the primary language of your app such as English.
- **Bundle ID** - the bundle ID that was created in the previous chapter.
- **SKU** - stands for Stock Keeping Unit. It can be anything you like. For example, your app name is "Awesome Food App". You can use "awesome_food_app" as the SKU. You can use letters, numbers, hyphens, periods and underscores except for space.
- **Full access** - choose *full access* unless you want to limit a particular users to see this app in App Store Connect.

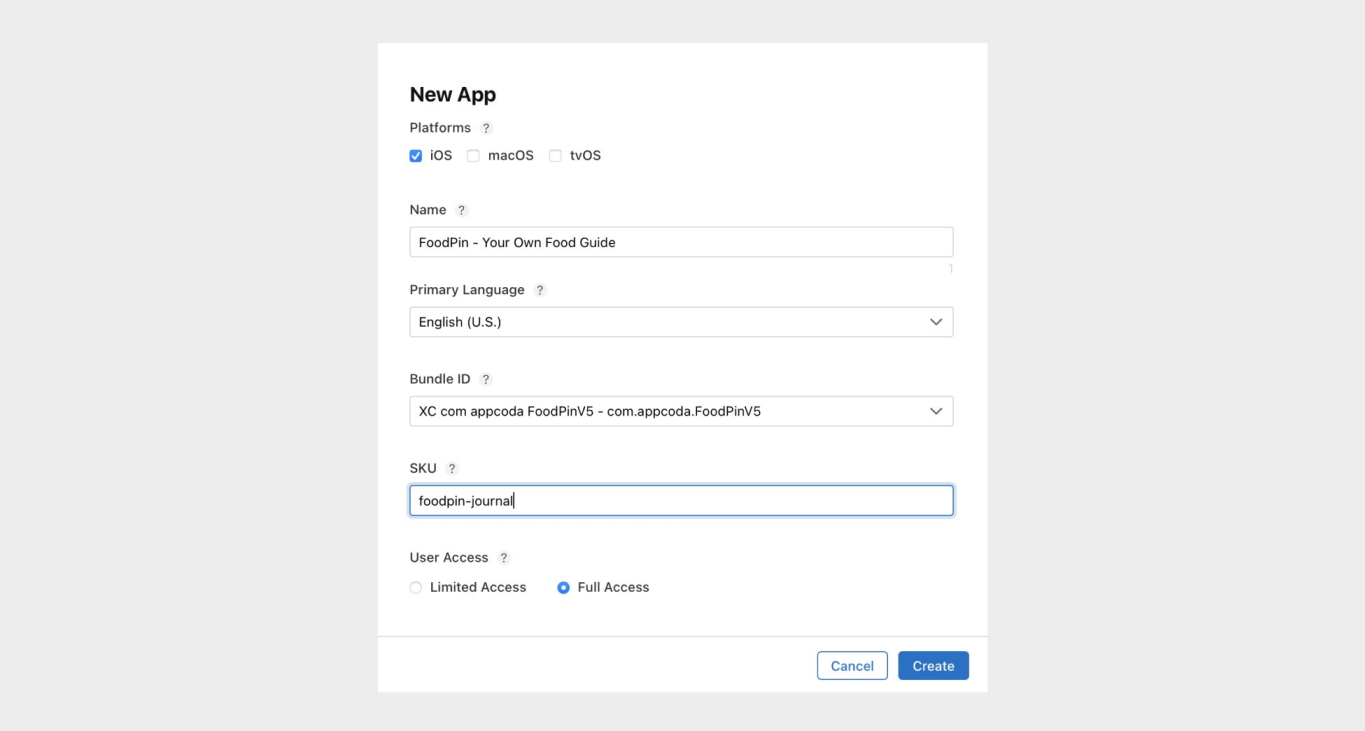


Figure 27-2. Fill in your app information

Once you click the Create button, you'll proceed to another screen to fill in the details of your app.

App Information

First, click the *App Information* in the side menu. For this section, it displays the app details that you have just filled in. Other than the name of your app, you can supply a subtitle which further describes your app.

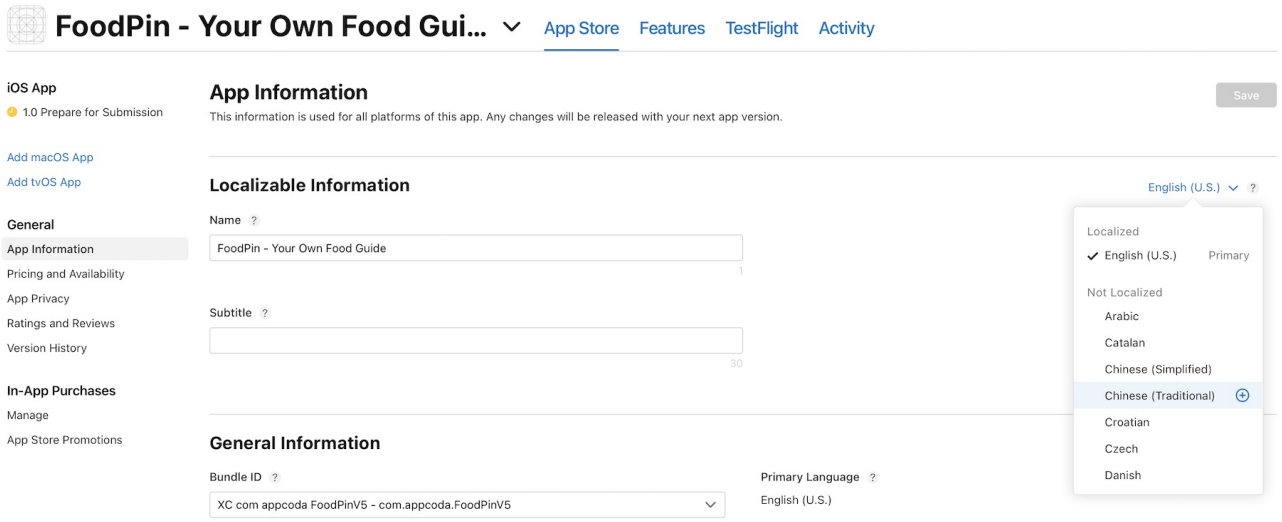


Figure 27-3. The App Information

Optionally, you can provide the app information in other languages. Simply click the *English (U.S.)* button to choose another language. There is one option you have to set - choose your app's primary category. This is the category your app will be listed on the app store. Pick the category that best fits the app. For example, you can choose the *Food & Drink* category for the FoodPin app.

Pricing and Availability

In the sidebar menu, you should also see the Pricing and Availability section. This is where you set the price of your app. By default, your app will be made available globally. In case you want to limit it to certain countries, you can click the *Edit* button under the *Availability* section, and choose your preferred countries. Remember to save the changes before you go to the next section.

Starting with macOS Big Sur, compatible iPhone and iPad apps can be made available on Apple silicon Macs. By default, your iOS apps will be made available to macOS users. If you do not want Apple silicon Mac users to download your app, uncheck the *Make this app available* checkbox.

App Privacy

Starting October 3, 2018, it is required to have a privacy policy for all apps and app updates in order to be submitted for distribution on the App Store or through TestFlight external testing. Choose *App Privacy* in the sidebar menu and fill in your privacy policy URL. Also, click the *Get Started* button to declare if your app will collect any data from users.

Prepare for Submission

Other than the basic information, you will need to provide additional information such as screenshots, app description, app icon, contact information, etc.

Choose the *Prepare for Submission* option in the sidebar menu to get started.

1. App Preview and Screenshots

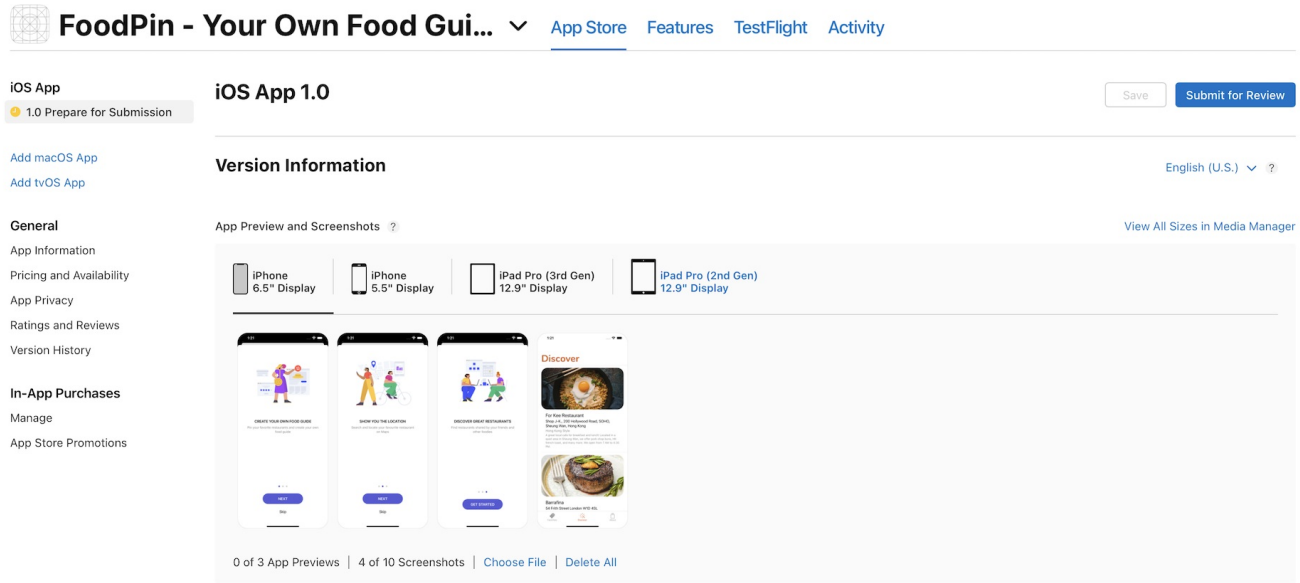


Figure 27-4. Prepare the app screenshots

These are the preview screens for your app. For the screenshots, you need to provide at least one screenshot for 6.5-inch devices. Optionally, you can upload the screenshots for iPhone 5.5" display. If your app supports iPad, you will also need to provide the screenshots for iPad.

To add the screenshots, you can click *Choose File* or simply drag the file to the box. Optionally, you can incorporate app preview videos.

> **Note:** You can further refer to Apple's App Connect Developer guide for details (https://developer.apple.com/support/app-store-connect/)

2. App Description and URL

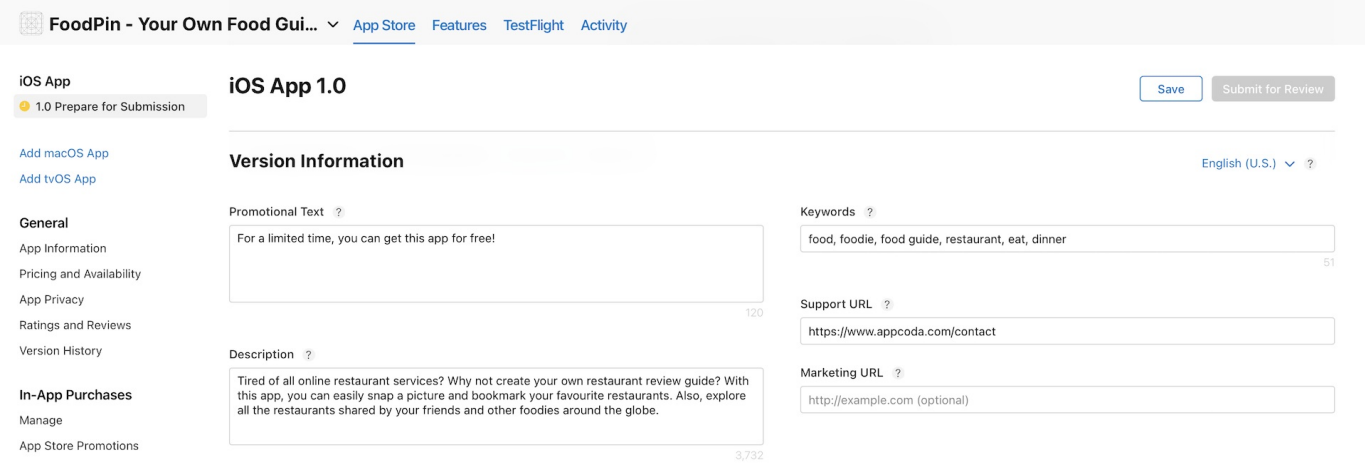


Figure 27-5. Your app description

Next, fill in your app description and promotional text. Promotional Text is a new feature of the revamped App Store. Prior to iOS 11, whenever you need to update the description of your app, you will need to go through Apple's review process. Now, you are free to modify the promotional text anytime you want. The

promotional text appears at the top of the description and is up to 170 characters long. You may use this field to share any news about your app. Say, your app is on sale. Use this field to inform your potential users.

Keywords are used to describe the nature of your app. You should enter at least one keyword in the Keywords field. If you have more than one keyword, separate them by commas. For example, "food,restaurant,recipe". Keywords are referenced by App Store for app search. It's one of the most important elements impacting your app download. You may have heard of App Store Optimization (ASO). Keyword optimization is a part of ASO. I will not go into keyword optimization here. If you want to learn more about keyword optimization, you can refer to this article (https://neilpatel.com/blog/app-store-optimization/) or just google ASO.

The support URL is mandatory. You can fill in the URL of your website or blog. If you don't have one, register a website at wordpress.com.

3. General App Information

You can skip the Build section and go straight to the General App Information section. This section is all about the general information of your app. You need to give a rating for your app. Just click the Edit button next to Rating and complete the form. App Store Connect generates a rating for your app based on your answers.

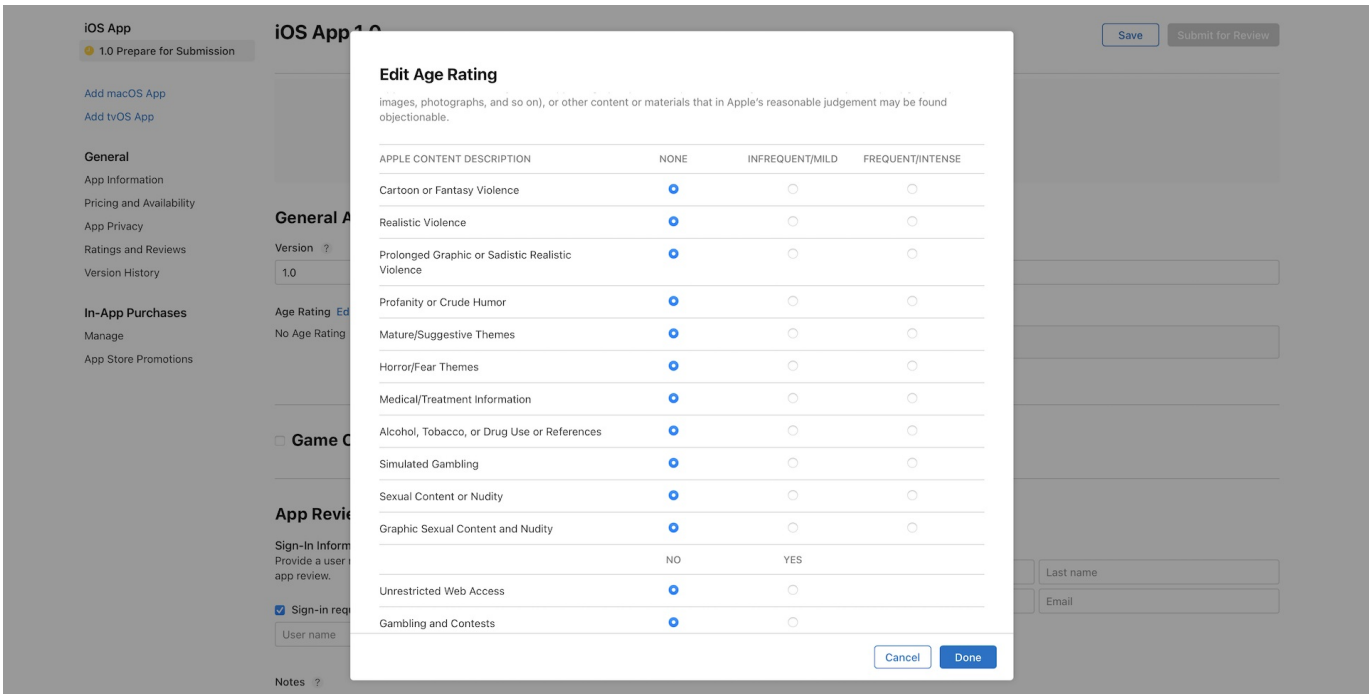


Figure 27-6. Edit age rating

For the copyright field, you can just fill in your name or company preceded by the year the rights were obtained (e.g. 2020 AppCoda Limited).

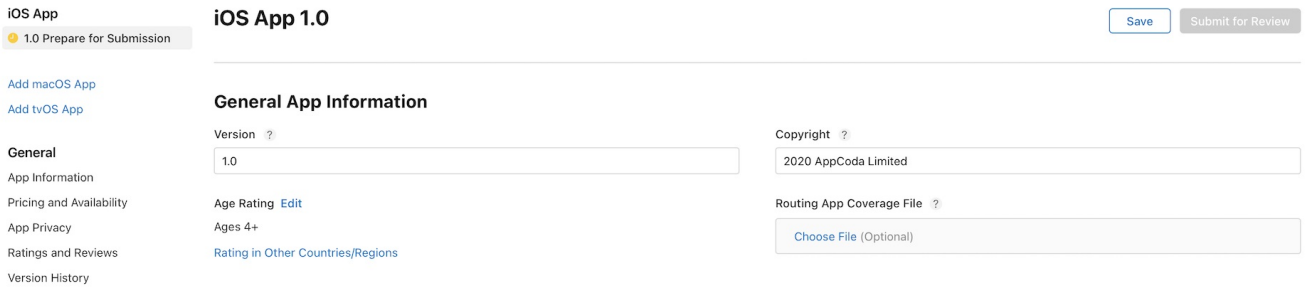


Figure 27-7. Sample app information

4. App Review Information

Simply fill in your contact information in this section. The demo account field is optional. It is for those apps that require a login. For apps that do not require sign-in, uncheck the *Sign-in required* checkbox.

5. Version Release

You're allowed to release your app automatically or manually right once it has been approved by App Review. Just set it to *Automatically release this version*. Finally, click the Save button at the top right corner to save the changes.

If you didn't leave out any required information, the `Submit for Review` button should be enabled. That means your app record was successfully created on App Store Connect.

Update Your Build String

Now go back to Xcode. We're going to build the app and upload to App Store Connect. Before that, review your project and make sure the version number matches what you entered in App Store Connect. In the project navigator, select the project and the target to display the project editor. Under the General tab, review the version field under the Identity section. As this is the first build, set the Build field to `1`.

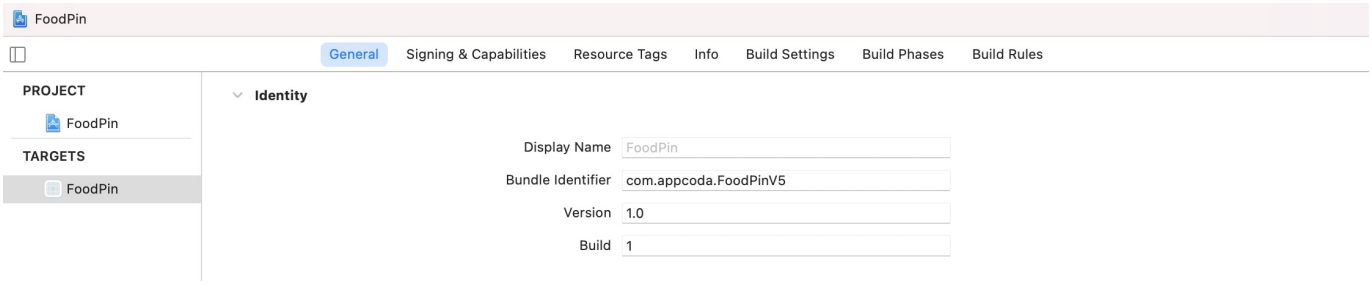


Figure 27-8. Review the version and build in project editor

Prepare Your App Icon and Launch Image

Before submitting your app, make sure you include the app icon and launch screen. The app icon is managed by the asset catalog. You should find the `AppIcon` set in `Assets.xcassets` .

Assuming you've created your app icon set, select an app icon in the Finder and drag it to the appropriate image well in the set viewer. You will need to provide various sizes of app icons to fit for different devices. Just follow the instructions accordingly. For example, your Spotlight icon should be in 80x80 pixels (40pt x 2). And, the app icon displayed on the App Store should be in 1024x1024 pixels.

Note that you may use design tools like Sketch and Figma to generate app icon for different sizes. Online tools like App Icon Generator (https://appicon.co) and MakeAppIcon (https://makeappicon.com) are also freely available to help you create app icons.

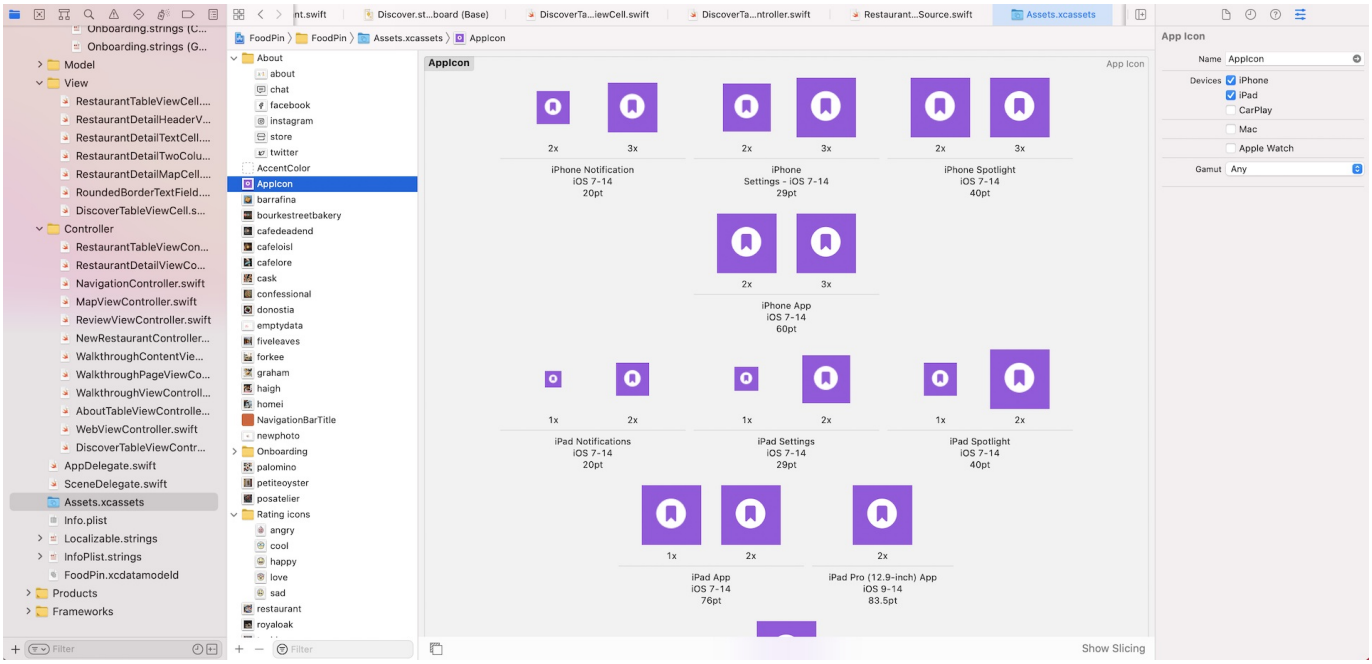


Figure 27-9. A sample AppIcon set

We have briefly mentioned the launch screen at the very beginning of the book. Xcode allows developers to use a storyboard or Interface Builder file to design a launch screen. In the FoodPin project, you should have a `LaunchScreen.storyboard` file, which is set as the default launch screen file in the project editor.

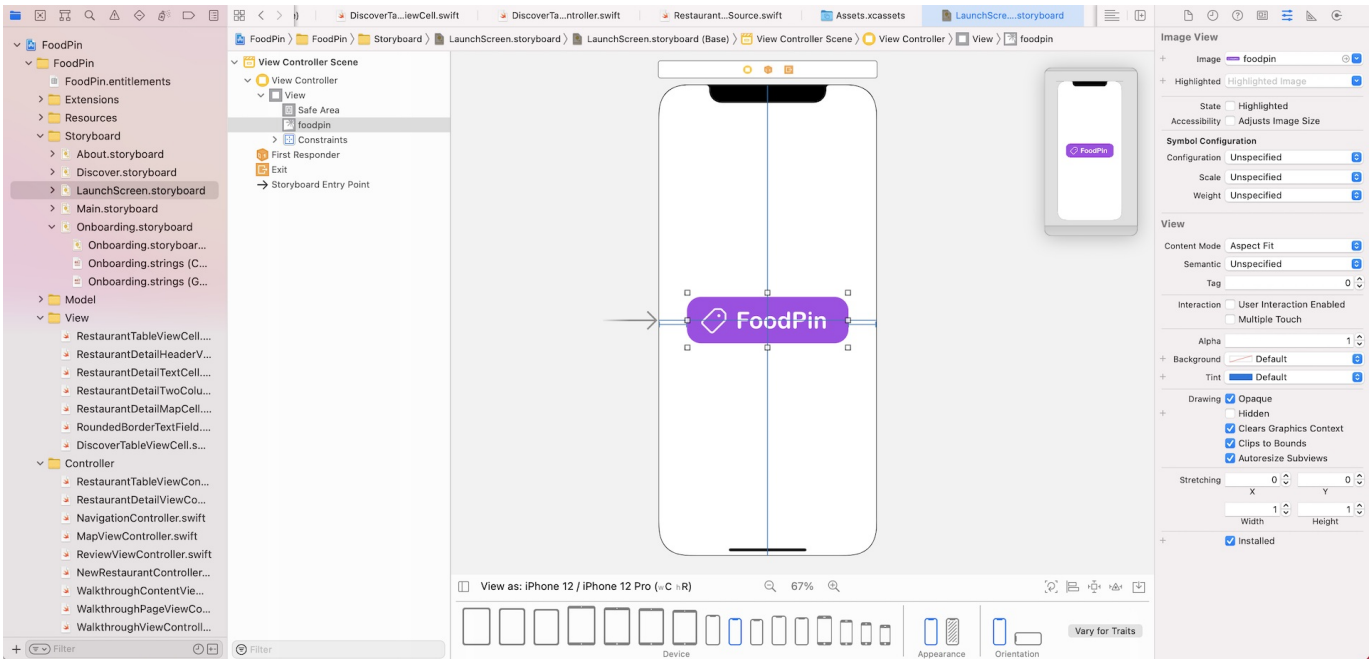


Figure 27-11. LaunchScreen is set as the default launch screen file

Quick tip: If you do not have a launch screen, you can right-click the FoodPin folder (or your project folder) and select "New File...". Just use the Launch Screen template under User Interface to create one.

In the `LaunchScreen.storyboard` file, you will find an empty view controller. When the app is launched, this view controller will be loaded as the startup screen. You're free to design its interface just like what you did for other view controllers. For reference, you can refer to Apple's Human Interface Guideline about the best practice of a launch screen. Figure 27-11 displays a sample launch screen design.

Archiving and Validating Your App

Before uploading an app to App Store Connect, you will need to create an app archive. First, review the *Archive* scheme settings and ensure the build configuration sets to Release (instead of Debug). Go up to the Xcode menu. Choose *Product* > *Scheme* > *Edit Scheme*. Select Archive scheme and review the Build Configuration setting. The option should be set to *Release*.

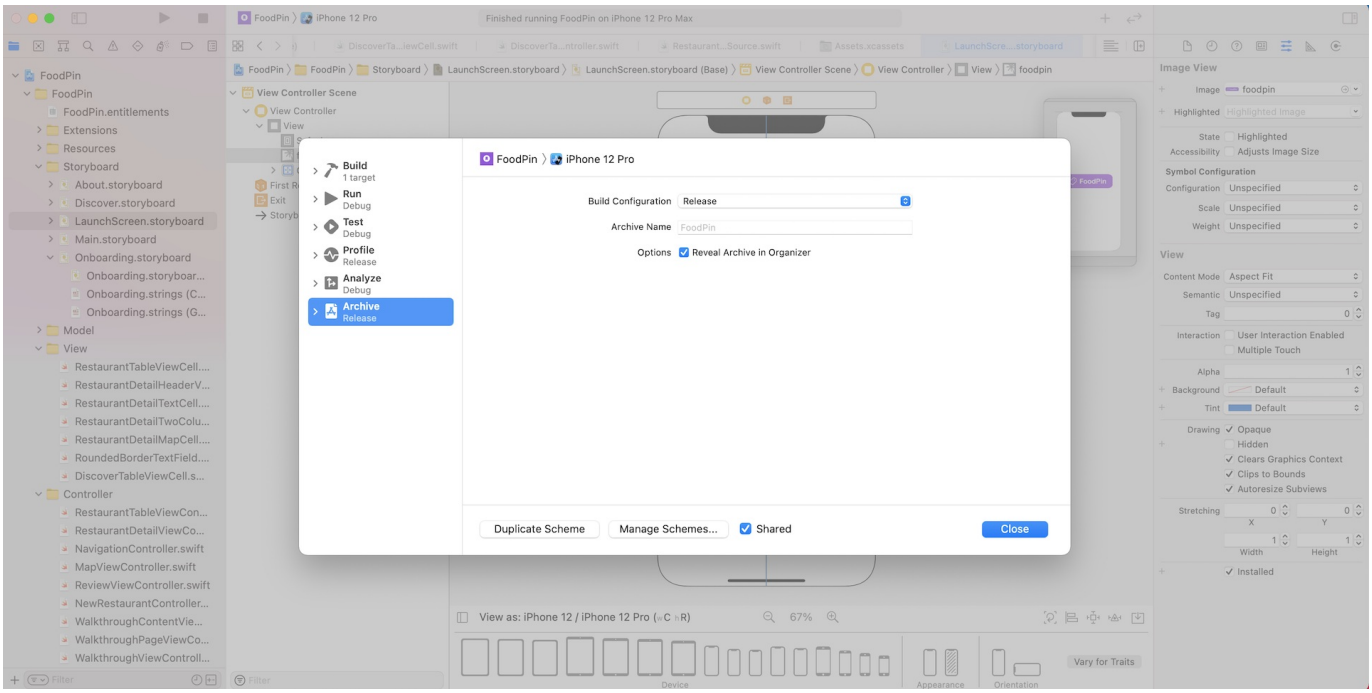


Figure 27-12. Review the Archive scheme setting

Now you're ready to archive your app. The Archive feature is disabled if you're using a simulator. Therefore, you need to select *Any iOS Device* or your device name (if you have connected a device to your Mac) from the Scheme pop-up menu. Then go up to the Xcode menu and choose *Product* > *Archive*.

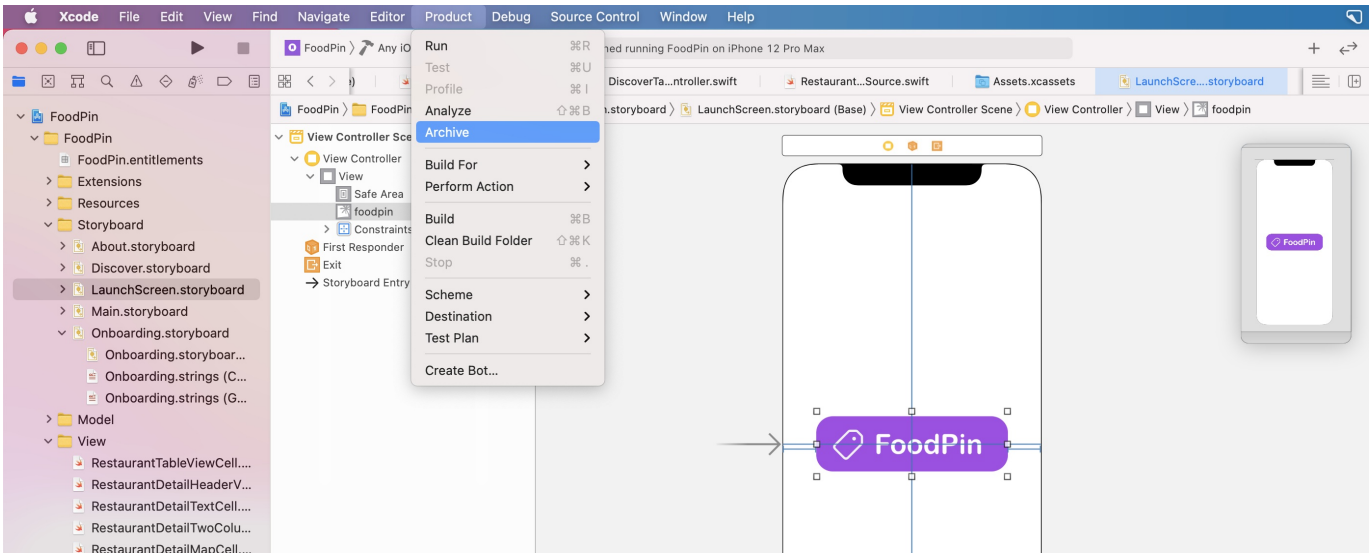


Figure 27-13. Archive your app

After archiving, your archive will appear in the Organizer. It's ready to upload to App Store Connect. But it's best to go through the validation process to see if there are any issues. Just click the `Validate App` button to begin the validation process.

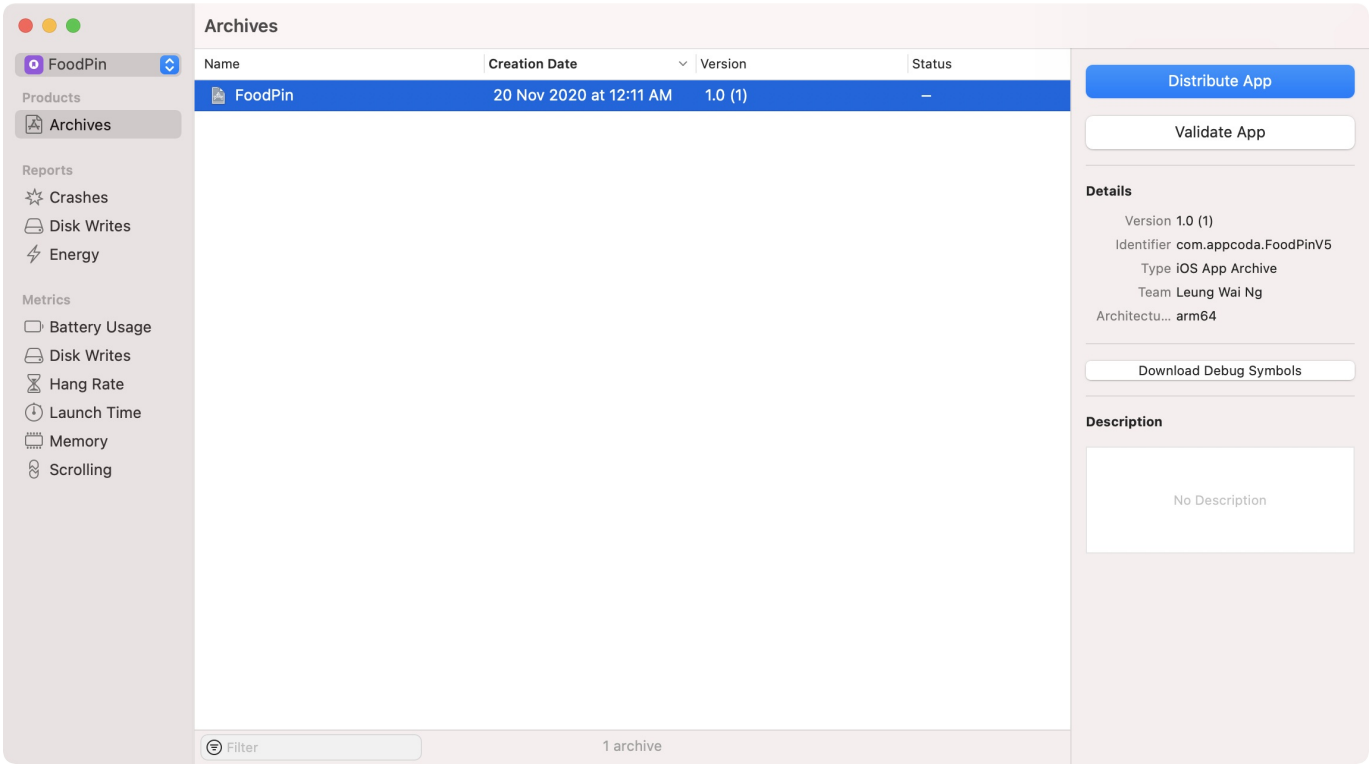


Figure 27-14. Your app archive appears in Organizer

You will be prompted to choose the App Store distribution option. Just accept the default setting and proceed. In the next screen, choose *Automatically manage signing* to let Xcode sign the distribution release for you. Now click *Validate* to proceed. In case you are asked to generate an Apple Distribution certificate, please tick the checkbox. It's a required procedure for distributing an app on the App Store.

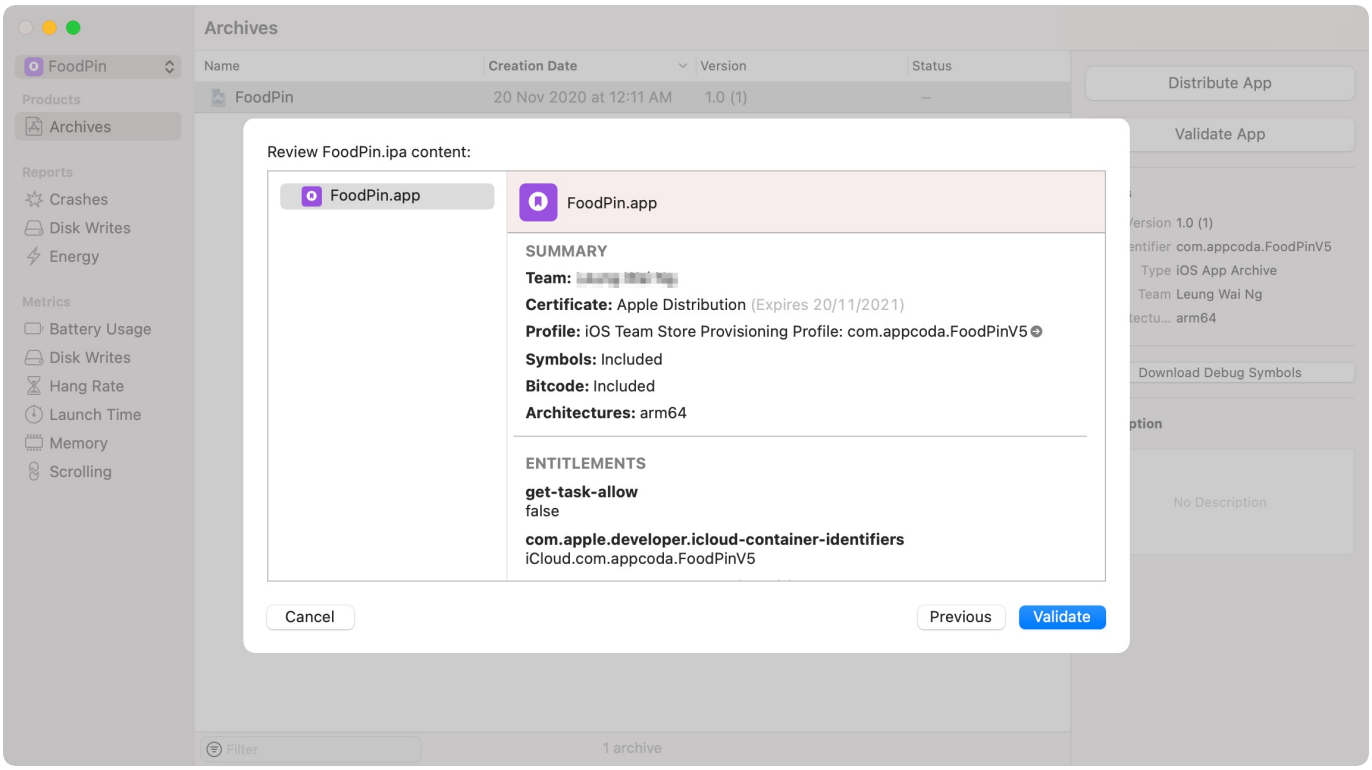


Figure 27-15. Xcode automatically creates the provisioning profile for your distribution release

If your app fulfills all the requirements, you should see the message "Your app successfully passed all the validation checks."

Upload Your App to App Store Connect

If the validation is successful, you can click the `Distribute App` button to upload the archive to App Store Connect. Since we want to upload to the App Store, choose *App Store Connect* when you are asked for the distribution method. Again, simply accept the default settings of the App Store Distribution options and then select the *Automatically manage signing* option.

Now click the *Upload* button to start uploading your app archive to App Store Connect. The whole process will take several minutes before you see the "Upload Success" message.

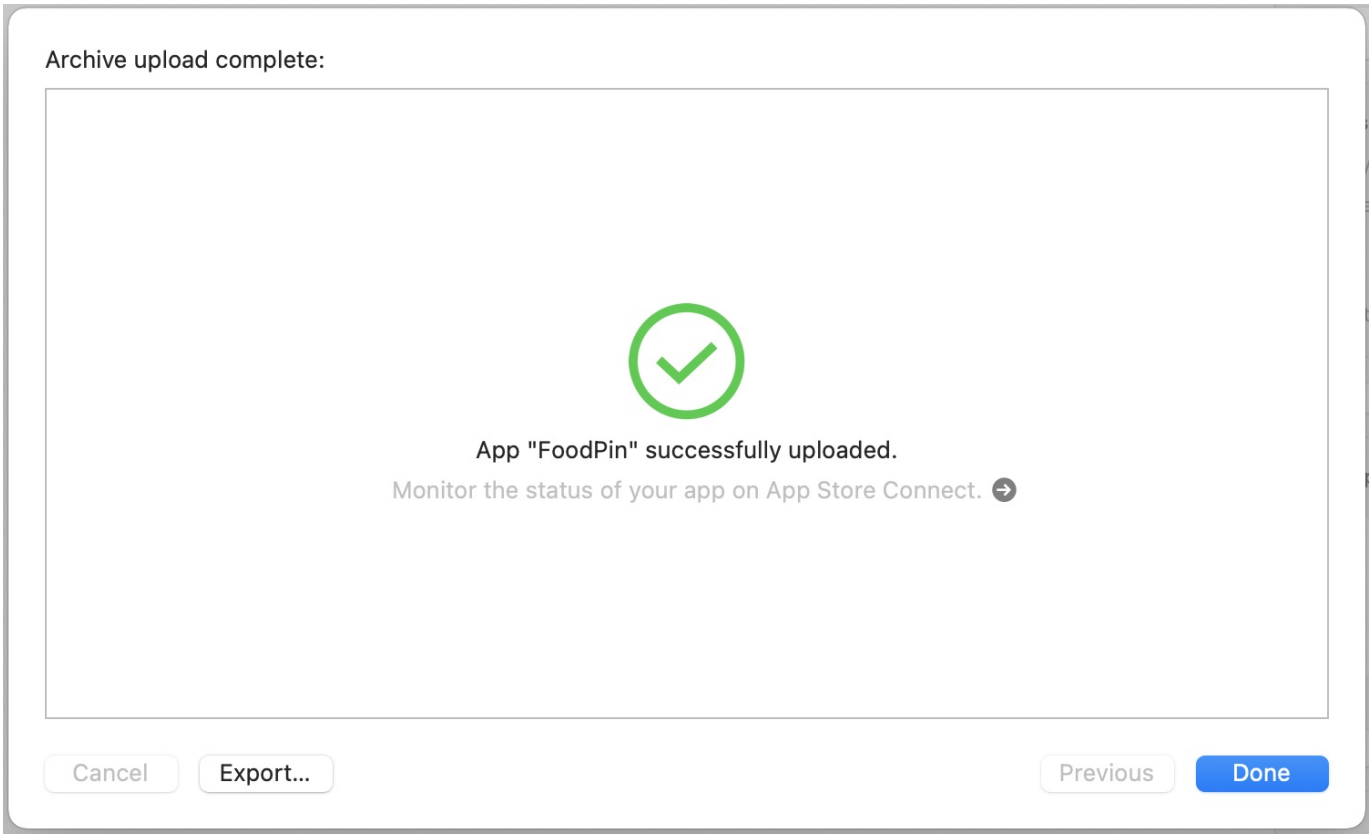


Figure 27-16. Upload your app archive to App Store

Manage Internal Testing

Now that you have uploaded your build to App Store Connect, let's see how you can roll out your app for internal testing.

Go back to https://appstoreconnect.apple.com. Select My Apps and then your app. In the menu, choose *TestFlight*. It will take App Store Connect some time to process the build you just uploaded. If you don't see anything in TestFlight, go to

Activity to check the status. It is very likely App Store Connect is still processing your build. Meanwhile, just wait and take a short break.

Assuming your build is ready for testing, it should appear in TestFlight. You first need to click the *Manage* button to provide the export compliance information.

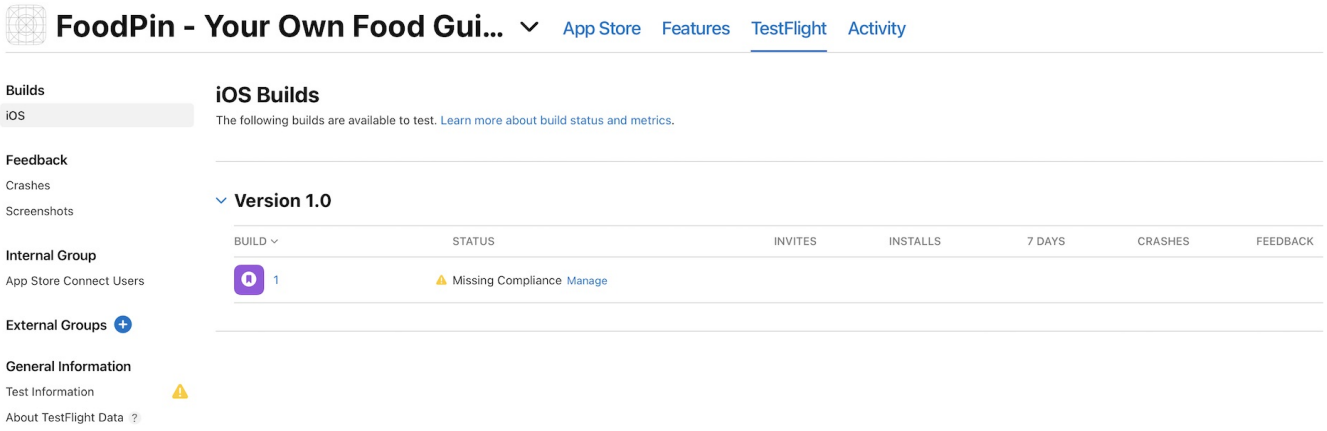


Figure 27-17. Your iOS Build in TestFlight

Before notifying your internal users to test the app, you have to fill in your test information including feedback email, privacy policy, marketing URL, and license agreement in TestFlight. Click *Test Information* and fill in the required information. Your testers can send feedback to the feedback email address. So make sure you enter the correct information.

Once you fill in the required information, select *App Store Connect Users* from the side menu. Click *Add Testers* and choose the testers. Then click *Add* to confirm.

That's it. TestFlight will automatically inform the testers by email.

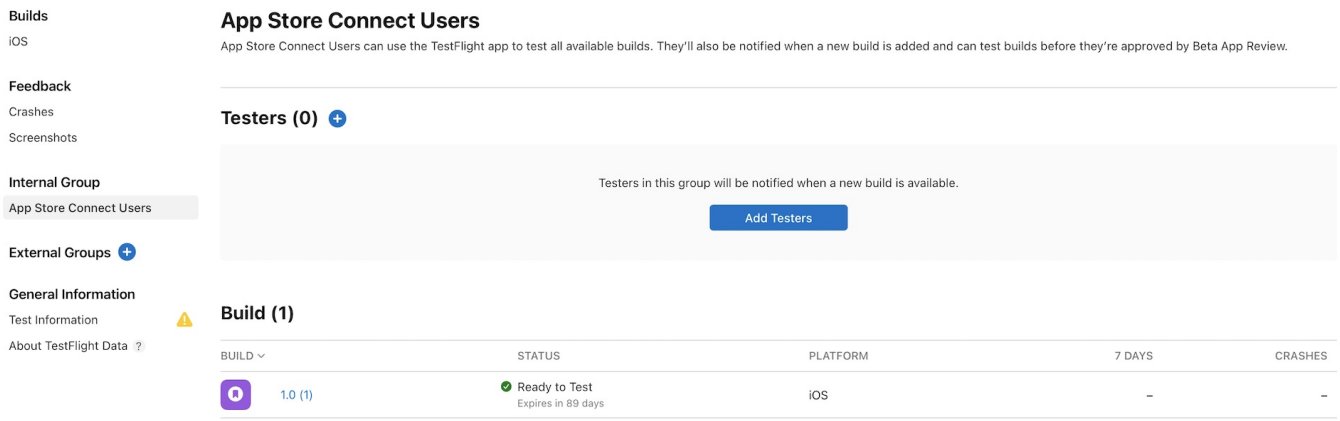


Figure 27-18. TestFlight automatically notifies your internal testers via email

> **Note:** Internal users refer to those who are part of your iTunes Connect team with the Admin, Legal, or Technical role. If you want to add more users or change their roles, select iTunes Connect > Users and Roles, and then click + to add the user. Alternatively, you can add individual testers directly.

When a tester receives the email notification, he/she needs to click the *View in TestFlight* button. iOS automatically opens the link in Safari. The tester just follows the instructions to install the TestFlight app, and then use the redeem code to download the app.

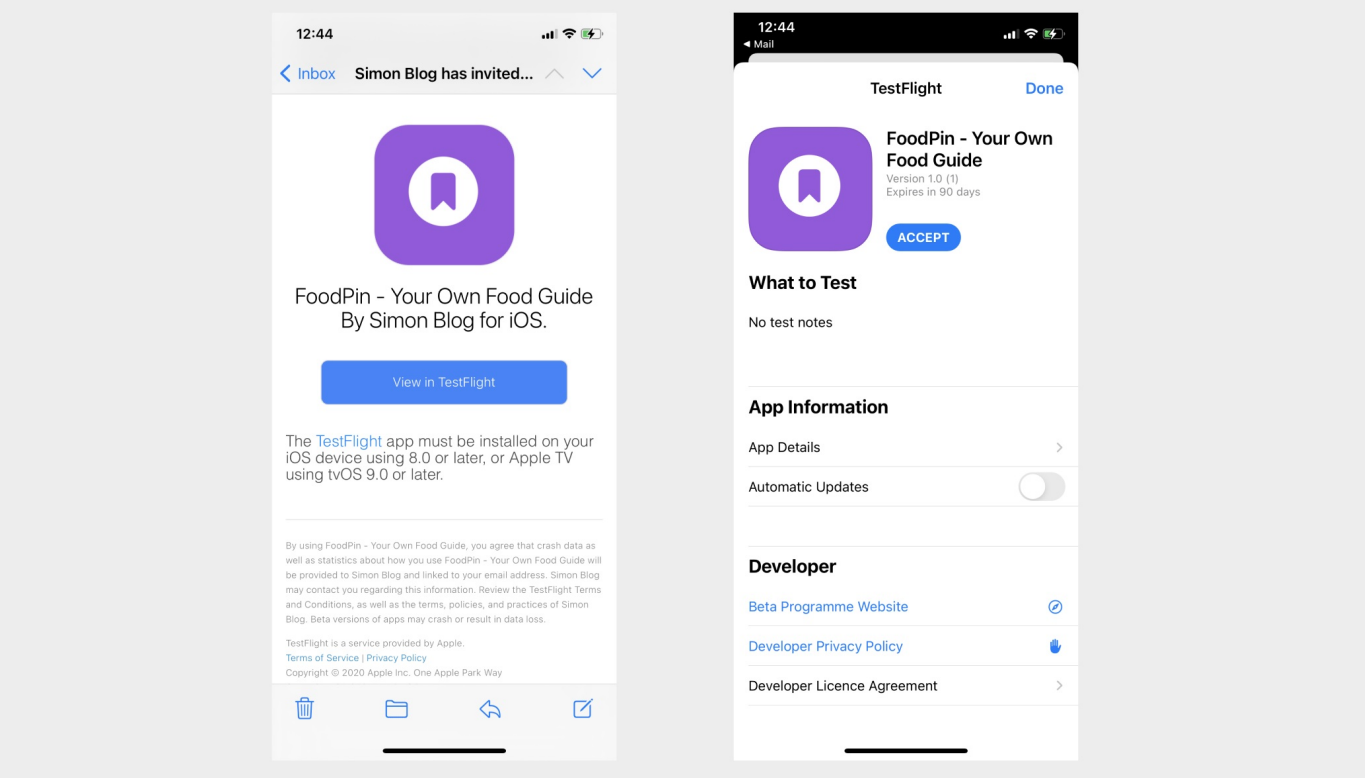


Figure 27-19. Send invitation to your testers

The beta version of your app expires in 90 days. For any future update of your beta app, your internal testers will always get the most recent build you uploaded.

Manage Beta Testing with External Users

You're allowed to invite up to 25 internal users for testing. Once your app lives up to the expectation of your users, you can invite more users to test out your app. TestFlight lets you invite up to 10,000 external users for beta testing. All you need is an email address of each of the testers so that you can send them an invite.

To add the external testers, click *Add External Testers* in the side menu. You are allowed to create multiple groups to organize your testers. When prompted, enter the name of your group and then hit *Create*.

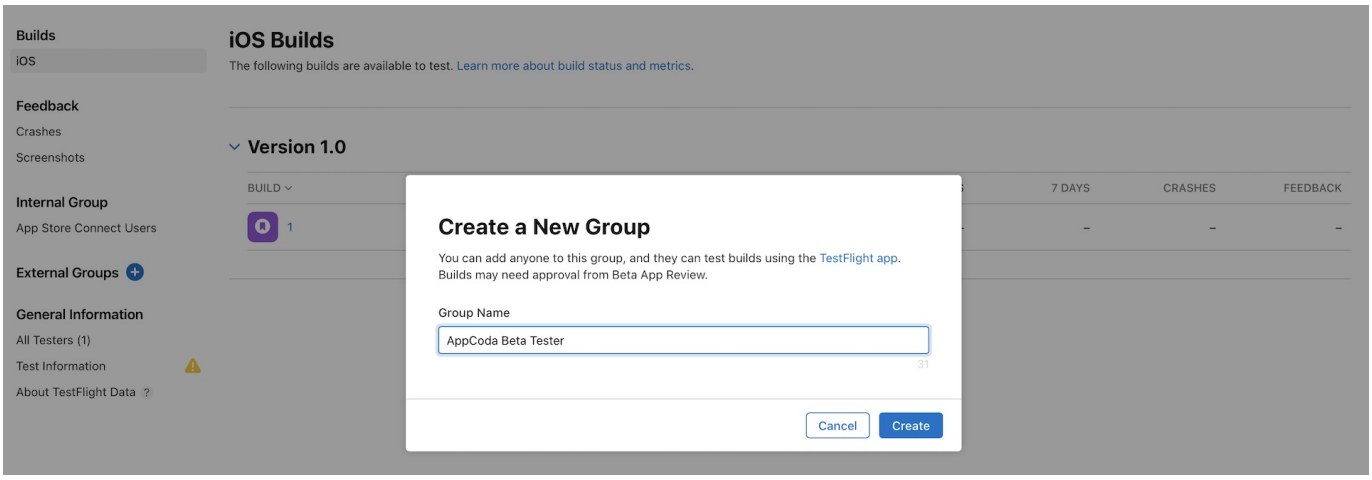


Figure 27-20. Submitting your app for beta app review

Follow the instructions to add the new testers. You can manually fill in their email address or import them by uploading a CSV file.

Once you created the group, choose *iOS* in the side menu and then click the app icon. Next, click the + icon next to *Group*. In the popover, select the group you just created and proceed. Follow the instructions to enter the test information for your beta testers.

As mentioned earlier, your app must be approved by Apple before you can send out your invitations. So the last step is to hit the *Submit for Review* button to submit your app for review.

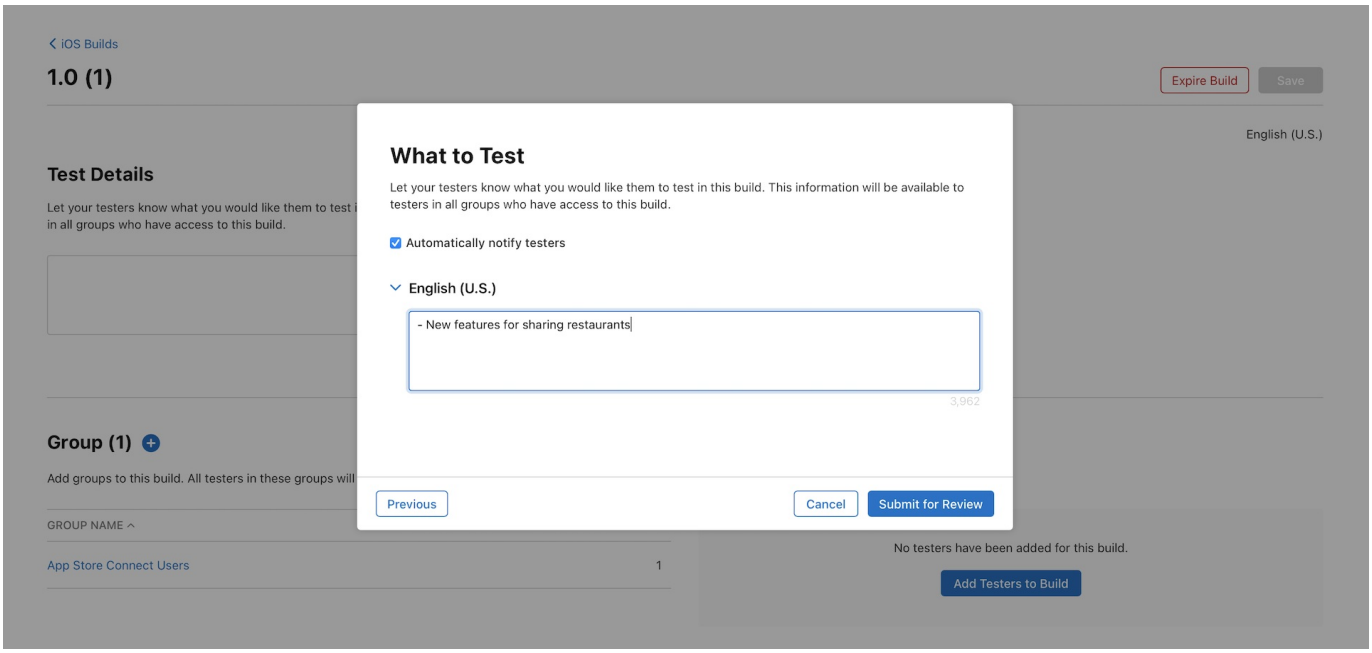


Figure 27-21. You have to submit your app for review before you can invite external users for testing

Once your app is submitted for review, the build status will change to Waiting for Beta App Review. Normally it'll take less than two days before your app is approved.

After Apple approves your beta app, the external users will be notified for beta testing.

CloudKit Production Deployment

The FoodPin app utilizes the iCloud database to store public records. Up till now, we only test our app in the development environment of CloudKit. For apps distributed using TestFlight, they are no longer allowed to use the development environment. Therefore, you have to deploy the database configuration of the development environment to production.

To do that, go back to the CloudKit dashboard and choose your container. In the development section, you should find the *Deploy Schema to Production* button. Click on it to set up the production environment.

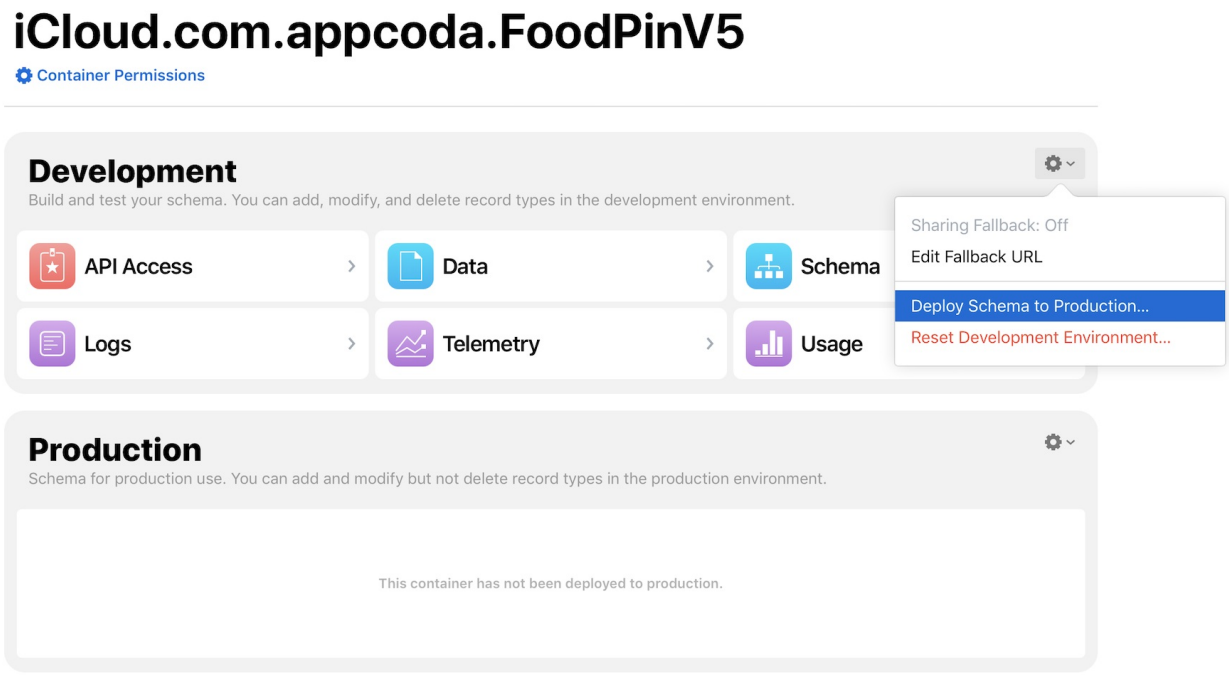


Figure 27-22. Deploy to production

You will be prompted to confirm the deployment. Click *Deploy Changes* to proceed. Deployment only promotes the schema (e.g. record types) to the production environment. However, it doesn't copy the records in the development environment to the production environment. So you have to create the production environment with records after the deployment.

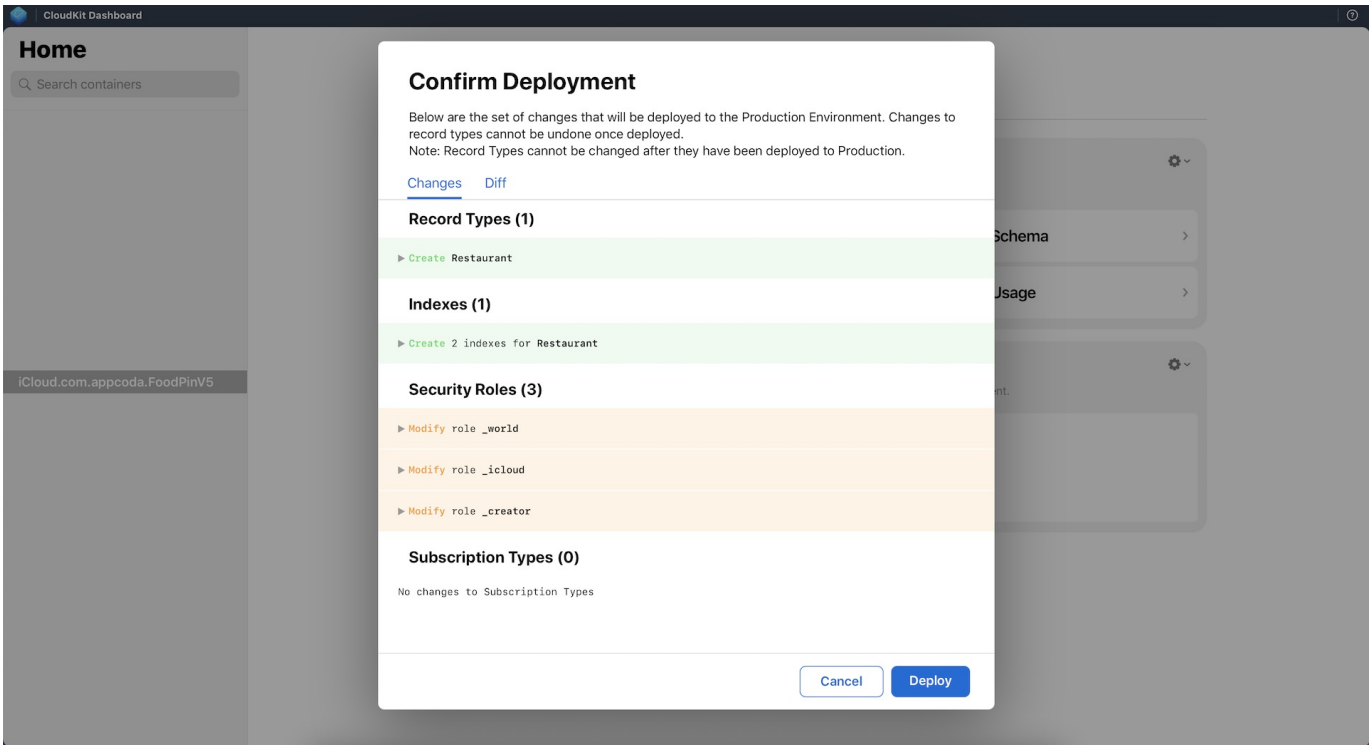
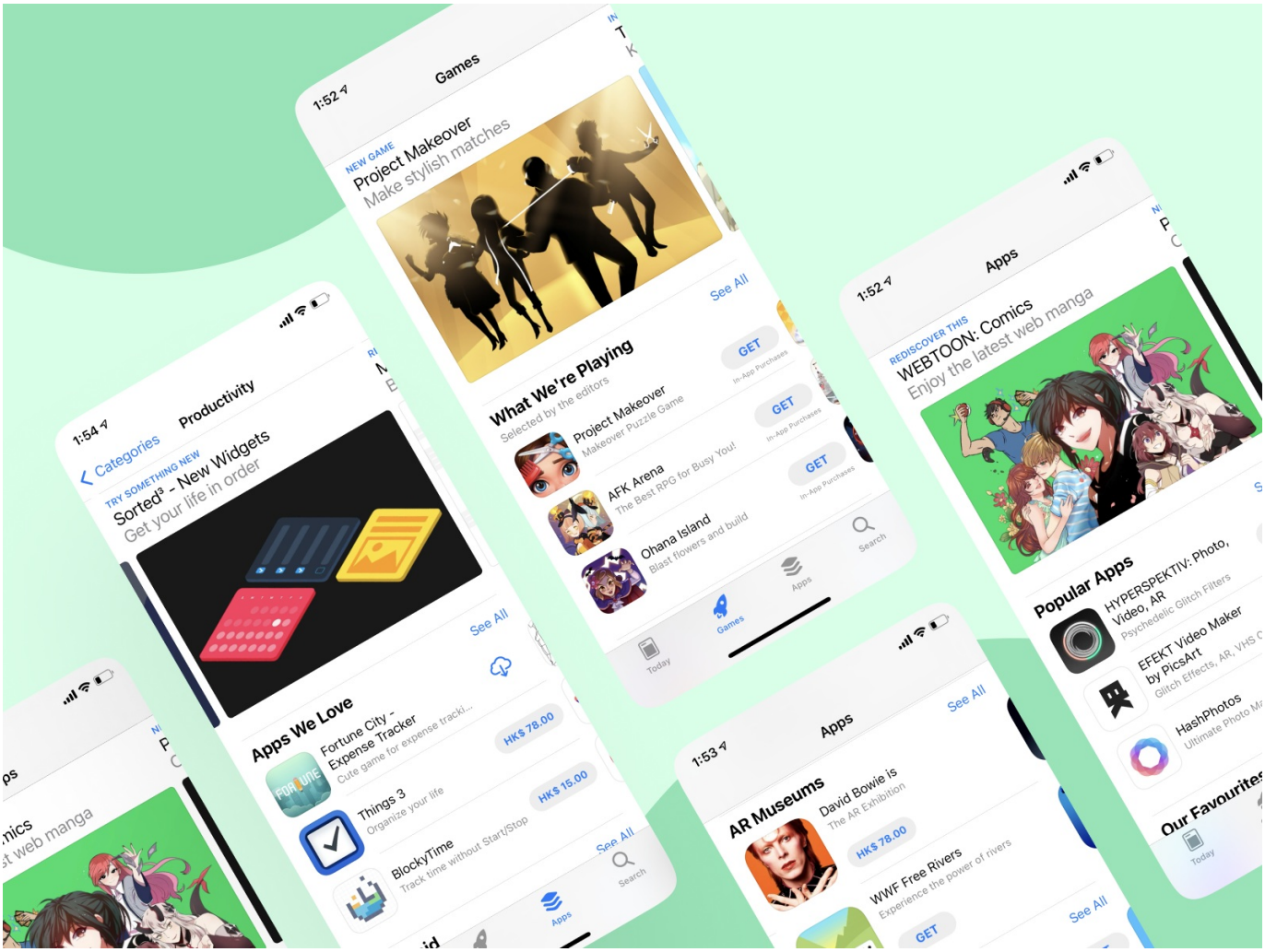


Figure 27-23. CloudKit dashboard shows you the changes to be applied on the production evironment

Summary

TestFlight provides us with a powerful tool to easily beta test our apps. In this chapter, I have walked you through the basics of TestFlight Beta Testing. If you're building your next app, use the tool to invite your friends and beta users to test out your app before the official release. This is an important step to build a high-quality app.

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.

Get Prepared and Well-Tested

From a developer's point of view, the app approval process is a black box. I still remembered how frustrated I felt when I submitted my first app to App Store and the app was rejected by Apple. Since then, I've learned that there are a few things you can do to minimize the likelihood of rejection.

1. **Test your app thoroughly** - Have you tested your app before submission? If you followed the previous two chapters, you should have tested your app on a real device and invited a group of beta testers to try it out. But let me emphasize again the importance of testing. You shouldn't just test your app using the built-in simulator - you have to test it at least on one real iOS device. If the app is a universal app, remember to test it on both iPhone and iPad. If your app integrates with other services like iCloud, make sure you test it on both cellular and Wi-Fi networks. Whenever possible, follow the instructions in the previous chapter to beta test your app. If you submit an app that can't run properly or crashes unexpectedly, Apple will definitely reject it. Also, remember to test your app under both normal and exceptional scenarios. Let's

say, your app requires an internet connection to work properly. What happens if the device has no signal? Will the app crash? Or it just displays a nice error message? This is something you have to take care of and make sure your app works in all situations.

2. **Follow the App Store review guidelines** - Even though we said the app store process is like a black box, Apple does provide a review guideline for developers' reference. You can access the guideline at https://developer.apple.com/app-store/review/guidelines/. Although I recommend you read it thoroughly, here are a few main points:
    - Apps that crash will be rejected
    - Apps that exhibit bugs will be rejected
    - Apps that do not perform as advertised by the developer will be rejected
    - Apps that include undocumented or hidden features inconsistent with the description of the App will be rejected
    - Apps that use non-public APIs will be rejected

You should also take note when your app title includes any of Apple's trademarks. I once created a book app about iOS and named it *iPhone Handbook*. The app was rejected immediately by Apple. Later, I changed the app title to *Handbook for iPhone* and the app was approved. Apple allows the use of Apple trademarked phrases and words when used in a referential phrase such as *for*.

For details, you can check out Apple's guideline at: https://www.apple.com/legal/intellectual-property/guidelinesfor3rdparties.html.

1. **Meet the User Interface requirement** - the app UI should be clean and user-friendly. Otherwise, Apple may reject your app due to substandard UI design. You can refer to UI Design Dos and Don'ts for details.
2. **Broken links** - All links embedded in your app must be functional. No broken links are allowed.
3. **Finalize all images and text** - All your images and text in your app should be finalized before submitting your app for review. Your app will be rejected if

it contains any placeholder content.

For more information about the common pitfalls that cause app rejections, you can refer to https://developer.apple.com/app-store/review/rejections/. Furthermore, don't forget to check out the App Store Review guidelines (https://developer.apple.com/app-store/review/guidelines/).

Submit Your App to App Store

If you haven't read chapter 27, please go back to read about the procedures of creating an app record on iTunes Connect and uploading an app archive.

Now go to https://appstoreconnect.apple.com and select My Apps. Then select the *FoodPin* app (or the name of your app). Assuming you have completed all of the required information in App Information, click *Pricing and Availability* in the side menu. If you have set your price before, this is a good chance to review your settings again. The latest version of App Store Connect allows developers to plan a price change. Let's say, you want to offer your app for free when it is first released. After a certain period of time, you change it to a paid app. The *Plan a Price Change* option can help you plan ahead for future price changes.

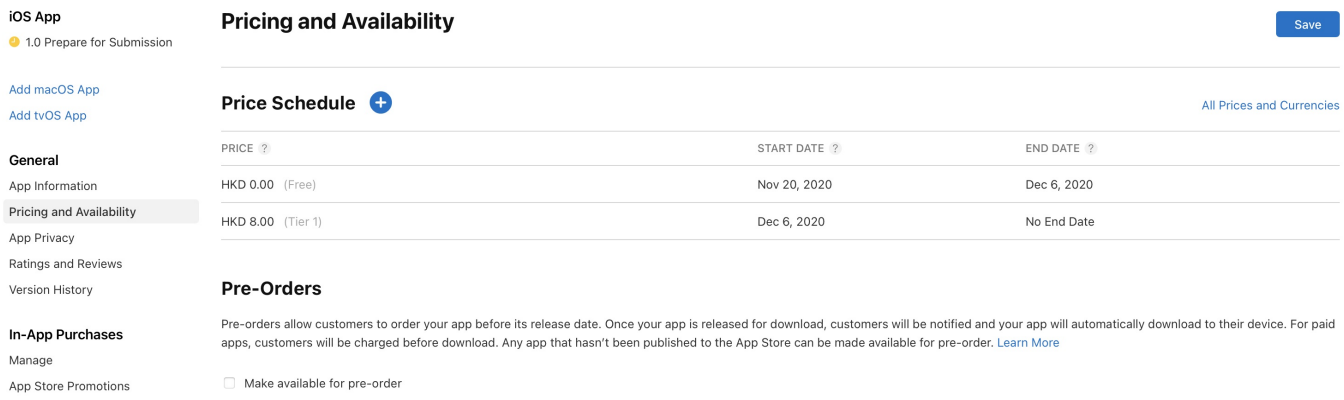


Figure 28-1. Fill in the availability date and price tier

Next, go to the *Prepare for Submission* option. Scroll down to the Build section. Click `Select a build before you submit your app` to add a build and select the one you want to submit.

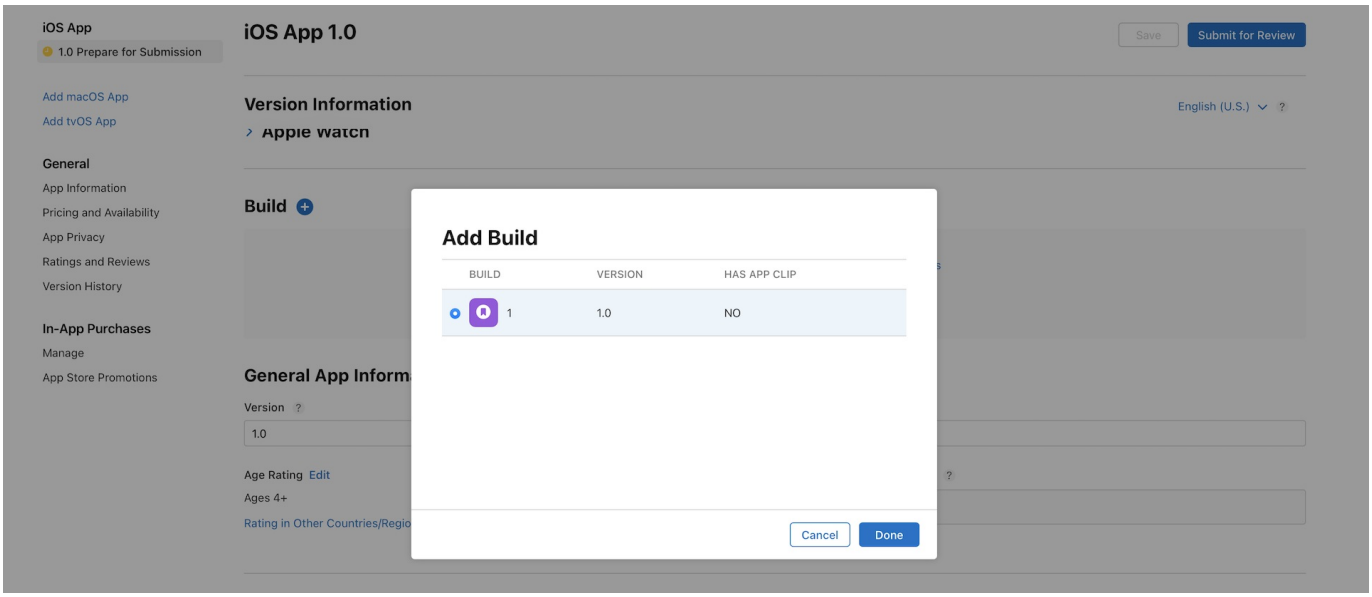


Figure 28-2. Pick a build for submission

Finally, save the changes and click the *Submit for Review* button to submit your app. Once you fill in the Export Compliance, Content Rights, and Advertising Identifier, your app is ready for submission.

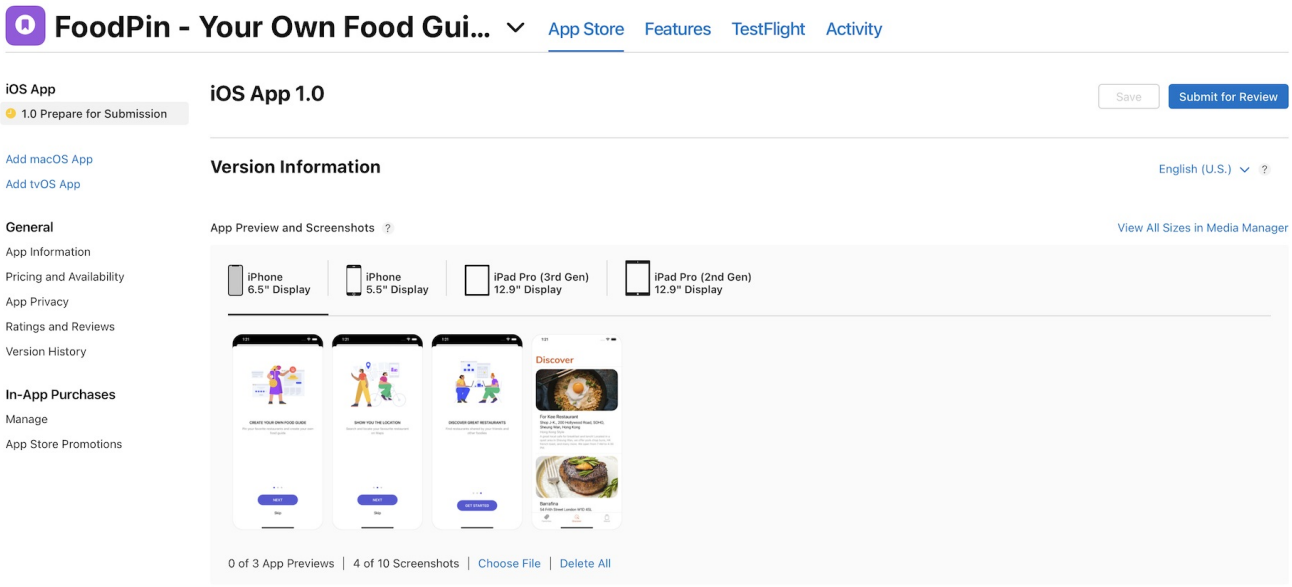


Figure 28-3. Click Submit for Review button to submit your app

After you successfully submit your app, the status will be changed to Waiting for Review.

Now, what's next? That's it. There is nothing you can do right now except wait. In the past, you would have to wait for around 7 days before your app was approved (or rejected). Starting from May 2016, Apple started to cut the approval time from more than a week to less than two days.

> On average, 50% of apps are reviewed in 24 hours and over 90% are reviewed in 48 hours.
>
> https://developer.apple.com/support/app-review/

So be patient and wait. You will receive an email notification when your app is approved or rejected.

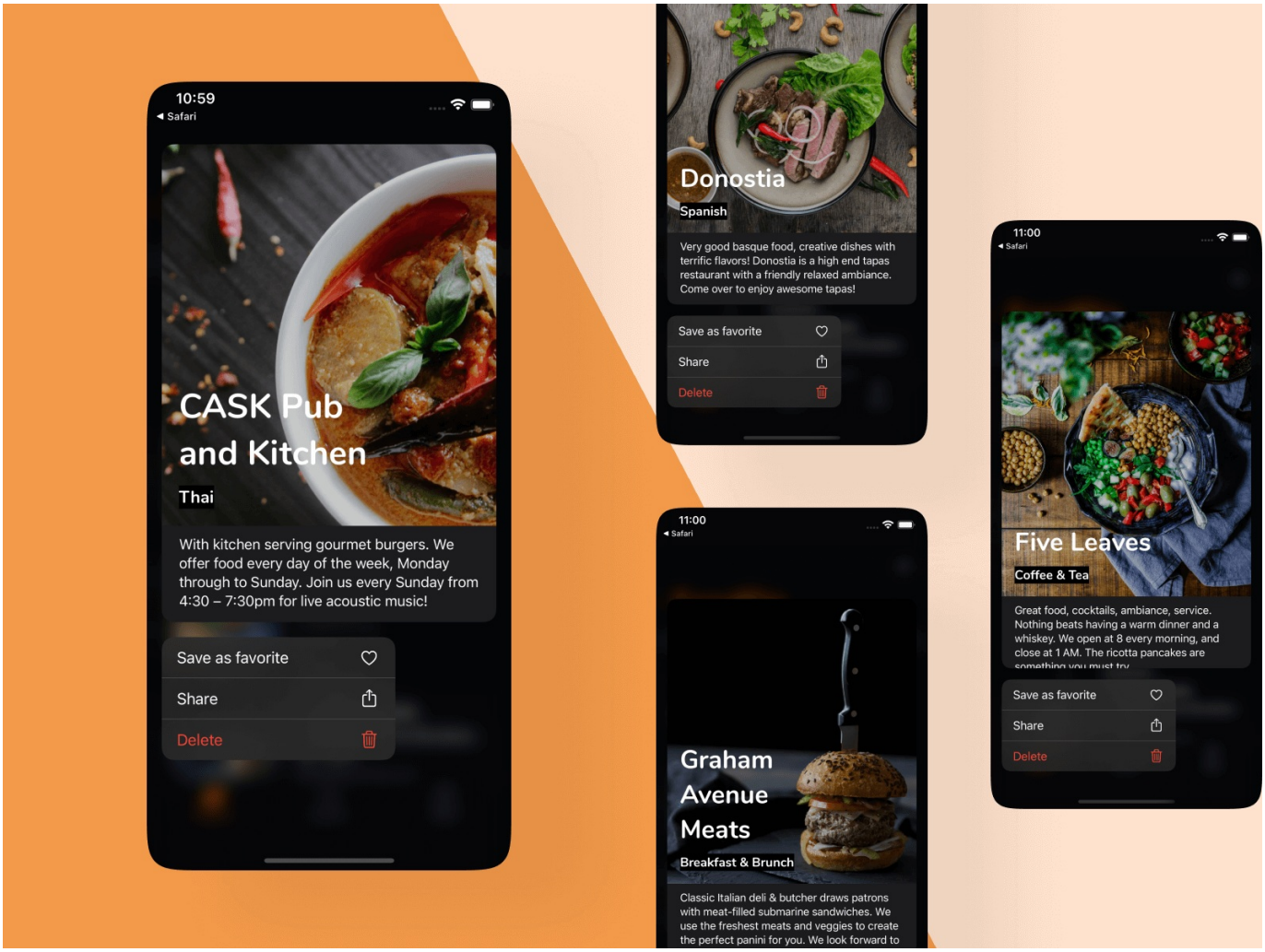
Summary

Once again, congratulations! You have built a real app and learned how to submit it to the App Store. I hope your app will be approved by Apple the first time you submit it. Even if it's rejected, don't get frustrated - many iOS developers share the same experience. Just fix the issue and re-submit your app again.

If your app got approved, don't forget to share your app with me and other readers on our private Facebook group (https://facebook.com/groups/appcoda).

This isn't the end. I still got a couple of things to share in the next chapter.

Chapter 29
Adopting Haptic Touch and Context Menus



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 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".

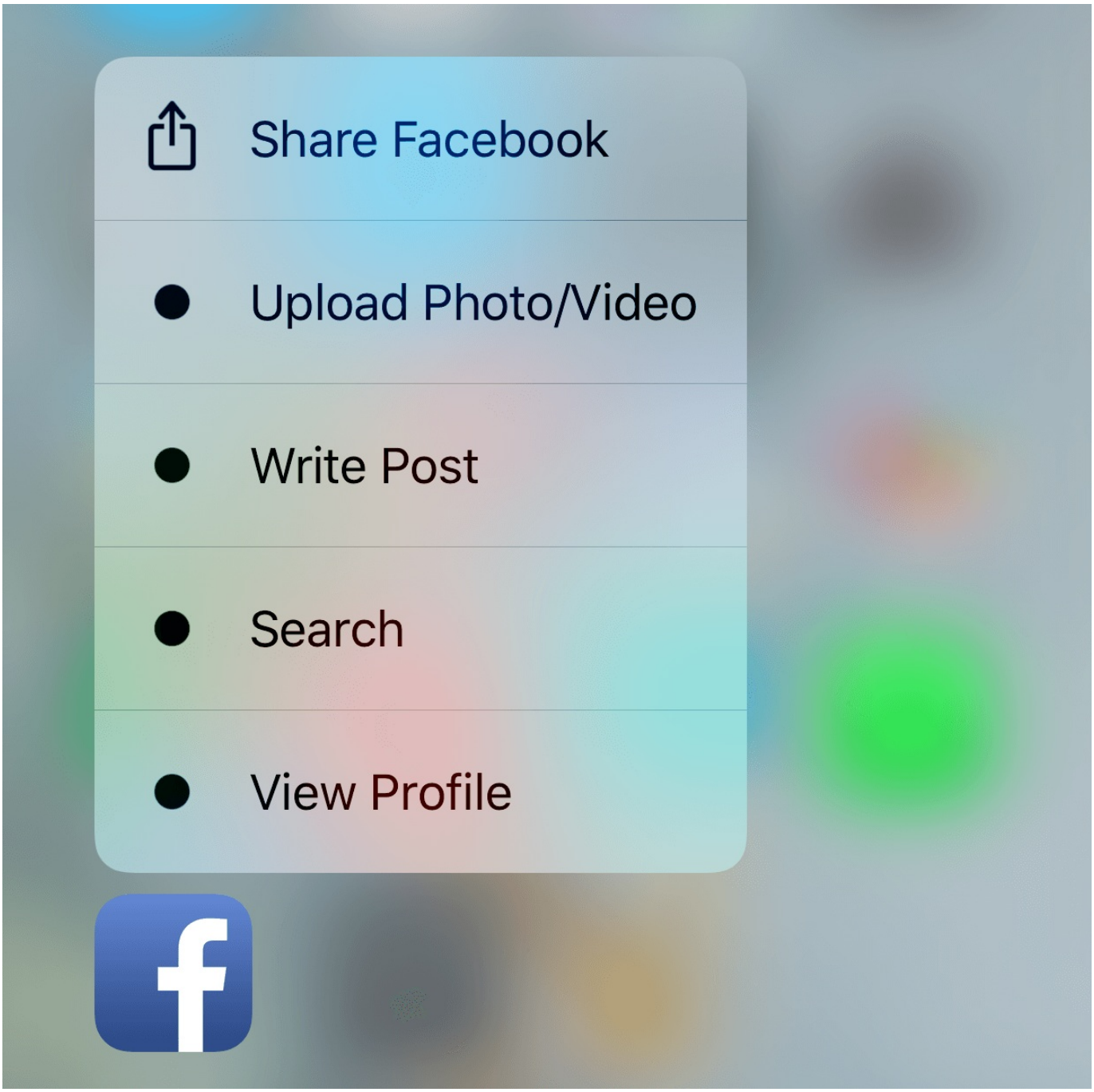


Figure 29-1. Sample Quick Actions

Since the release of iPhone 11, 11 Pro, and 11 Pro Max, Apple was replacing 3D Touch with Haptic Touch across its entire iPhone lineup. Haptic Touch is pretty similar to 3D Touch. While 3D Touch supports force touch, Haptic Touch is a

touch and hold gesture.

If you've used 3D Touch before, Peep and Pop is a very nice feature that gives users a quicker access to the app's contents. In iOS 13, this feature is replaced by *Context Menus*. It's very similar to Peep and Pop but with an instant access to a list of action items. What's more is that Context Menus work on all devices running on iOS 13 (or later).

So, you still have no ideas about Context Menus? Open the Photos app to try it out. When you touch and hold a thumbnail, you can bring the photo preview and a contextual menu that lets you have a quick access to some common functions. If you want more than a preview, just tap the photo preview to bring up a full view.

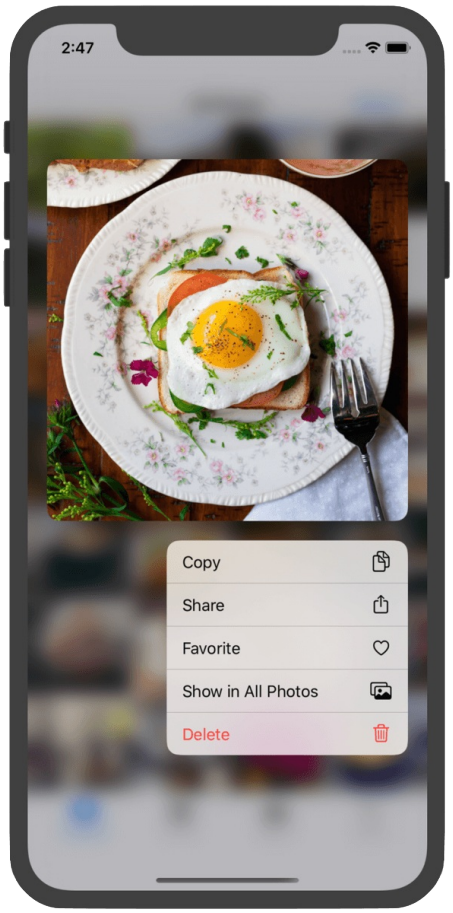


Figure 29-2. A sample context menu in Photos

In this chapter, we will see how to work with context menus in iOS 13 (or later). More specifically, we will add Quick Actions and Context Menus 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.

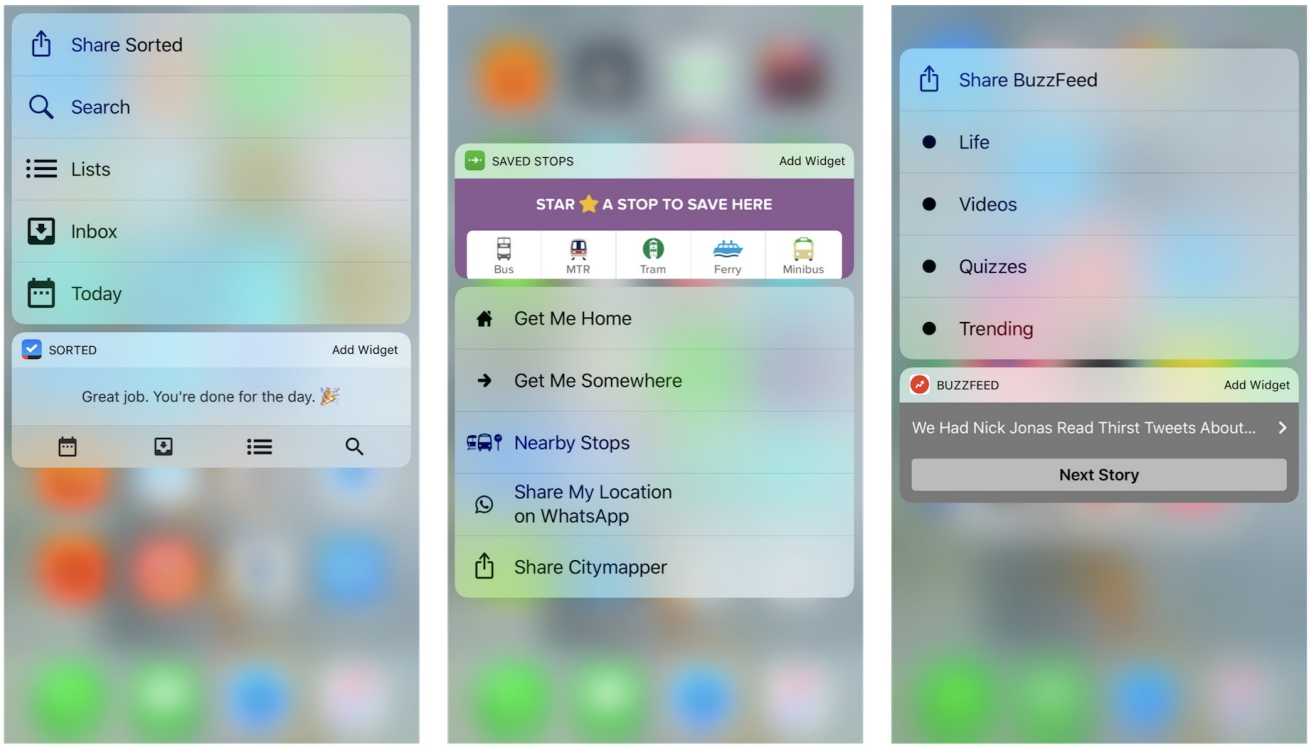


Figure 29-3. Quick actions Widgets

But one thing they have in common is that you can create at most 4 quick actions, no matter you're using static or dynamic quick actions.

It's pretty simple to create static quick actions. All you need to do is edit the `Info.plist` file and add a `UIApplicationShortcutItems` array. Each element of the array is a dictionary containing the following properties:

- `UIApplicationShortcutItemType` (required) - a unique identifier used to identify the quick action. It should be unique across all apps. So a good practice is to prefix the identifier with the app bundle ID (e.g. com.appcoda.).
- `UIApplicationShortcutItemTitle` (required) - the name of the quick action visible to the user.
- `UIApplicationShortcutItemSubtitle` (optional) - the subtitle of the quick action. It is an optional string displayed right below the title of the quick action.

- `UIApplicationShortcutItemIconType` (optional) - an optional string to specify the type of an icon from the system library. Refer to this document for the available icon type.
- `UIApplicationShortcutItemIconFile` (optional) - if you want to use your own icon, specify the icon image to use from the app's bundle. Alternatively, specify the name of the image in an asset catalog. These icons should be square, single colour with sizes 35x35 (1x), 70x70 (2x) and 105x105 (3x).
- `UIApplicationShortcutItemUserInfo` (optional) - an optional dictionary containing some extra information you want to pass. For example, one use for this dictionary is to pass the app version.

If you want to add some static quick actions, here is an example of the `UIApplicationShortcutItems` array, which creates a "New Restaurant" shortcut:

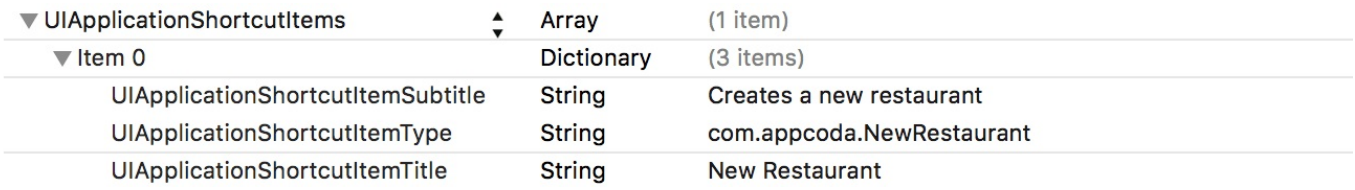


Figure 29-4. Sample Info.plist for static quick actions

Now that you should have some ideas about static quick actions, let's talk about the dynamic ones. As a demo, we will modify the FoodPin project, and add three quick actions to the app:

- **New Restaurant** - go to the New Restaurant screen directly
- **Discover restaurants** - jump right into the Discover tab
- **Show Favorites** - jump right into the Favorites tab

First things first, why do we use dynamic quick actions? A simple answer is that I want to show you how to work with dynamic quick actions. But the actual reason is that I want to enable these quick actions only after the user goes through the

walkthrough screens.

To create a quick action programmatically, you just need to instantiate a `UIApplicationShortcutItem` object with the required properties and then assign it to the `shortcutItems` property of `UIApplication`. Here is an example:

```
let shortcutItem = UIApplicationShortcutItem(type: "com.appcoda.NewRestaur
ant", localizedTitle: "New Restaurant", localizedSubtitle: nil, icon: UIAp
plicationShortcutIcon(type: .add), userInfo: nil)
UIApplication.shared.shortcutItems = [shortcutItem]
```

The first line of code defines a shortcut item with the quick action type `com.appcoda.NewRestaurant` and system icon `.add`. The title of the quick action is set to `New Restaurant`. The second line of code initializes an array with the shortcut item and set it to the `shortcutItems` property.

To create the quick actions, let's create a helper method in the `WalkthroughViewController` class:

```
func createQuickActions() {
    // Add Quick Actions
    if let bundleIdentifier = Bundle.main.bundleIdentifier {
        let shortcutItem1 = UIApplicationShortcutItem(type: "\(bundleIdent
ifier).OpenFavorites", localizedTitle: "Show Favorites", localizedSubtitle
: nil, icon: UIApplicationShortcutIcon(systemImageName: "tag"), userInfo:
nil)
        let shortcutItem2 = UIApplicationShortcutItem(type: "\(bundleIdent
ifier).OpenDiscover", localizedTitle: "Discover Restaurants", localizedSub
title: nil, icon: UIApplicationShortcutIcon(systemImageName: "eyes"), user
Info: nil)
        let shortcutItem3 = UIApplicationShortcutItem(type: "\(bundleIdent
ifier).NewRestaurant", localizedTitle: "New Restaurant", localizedSubtitle
: nil, icon: UIApplicationShortcutIcon(type: .add), userInfo: nil)
        UIApplication.shared.shortcutItems = [shortcutItem1, shortcutItem2
, shortcutItem3]
    }
}
```

The code is pretty much the same as what I have covered before. We create three quick action items. Each of which has its own identifier, title, and icon. For the *Discover restaurant* and *Show favorites* shortcuts, we use our own icons.

So where should we call this `createQuickActions()` method?

Do you still remember how we indicate a user has gone through the walkthrough screens? We set the key named `hasViewedWalkthrough` to `true` into the user defaults after the user completes the walkthrough. This line of code can be found in the `nextButtonTapped` method and the `skipButtonTapped` method of the `WalkthroughViewController` class:

```
UserDefaults.standard.set(true, forKey: "hasViewedWalkthrough")
```

Therefore, make the call `createQuickActions()` after the code above.

Now you're ready to go. Hit the Run button and deploy the app to your iPhone. After launching the app once and going through the walkthrough screens, you will be able to see the quick actions. You do not need to test 3D Touch/Haptic Touch on a real iOS device, just test the feature on a simulator.

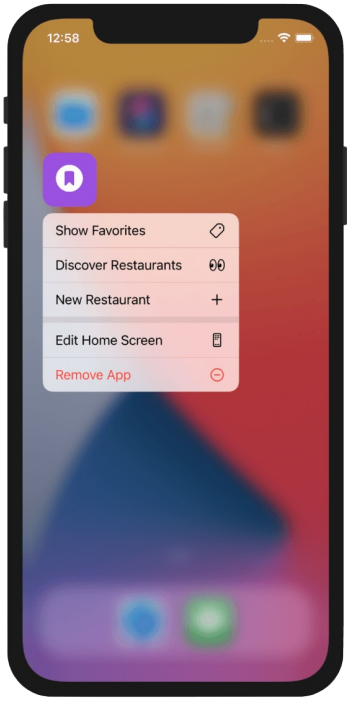


Figure 29-5. Quick actions in the FoodPin app

The quick actions are not ready to work yet because we haven't implemented the required methods to launch the quick actions. In the iOS 13, there is a method called `windowScene(_:performActionFor:completionHandler:)` defined in the `UIWindowSceneDelegate` protocol. When the user selects a quick action, this method is called. So we will implement the method in `SceneDelegate.swift`.

Let's first declare an enum for the quick actions in the delegate:

```
enum QuickAction: String {
    case OpenFavorites = "OpenFavorites"
    case OpenDiscover = "OpenDiscover"
    case NewRestaurant = "NewRestaurant"

    init?(fullIdentifier: String) {

        guard let shortcutIdentifier = fullIdentifier.components(separated
By: ".").last else {
            return nil
        }

        self.init(rawValue: shortcutIdentifier)
    }
}
```

The Enumeration type in Swift is particularly useful for defining a group of related values. In the `QuickAction` enum, we define the available quick actions in each case. We also create an initialization method that turns a full identifier (e.g. `com.appcoda.NewRestaurant`) into the corresponding enumeration case (e.g. `NewRestaurant`).

Now implement the `windowScene(_:performActionFor:completionHandler:)` method in the `AppDelegate` class like this:

```
func windowScene(_ windowScene: UIWindowScene, performActionFor shortcutIt
em: UIApplicationShortcutItem, completionHandler: @escaping (Bool) -> Void
) {

    completionHandler(handleQuickAction(shortcutItem: shortcutItem))
}

private func handleQuickAction(shortcutItem: UIApplicationShortcutItem) ->
Bool {

    let shortcutType = shortcutItem.type
```

```
    guard let shortcutIdentifier = QuickAction(fullIdentifier: shortcutTyp
e) else {
        return false
    }

    guard let tabBarController = window?.rootViewController as? UITabBarCo
ntroller else {
        return false
    }

    switch shortcutIdentifier {
    case .OpenFavorites:
        tabBarController.selectedIndex = 0
    case .OpenDiscover:
        tabBarController.selectedIndex = 1
    case .NewRestaurant:
        if let navController = tabBarController.viewControllers?[0] {
            let restaurantTableViewController = navController.children[0]
            restaurantTableViewController.performSegue(withIdentifier: "ad
dRestaurant", sender: restaurantTableViewController)
        } else {
            return false
        }
    }

    return true
}
```

When a quick action is activated, the `windowScene(_:performActionFor:completionHandler:)` method is called. The selected shortcut (quick action) is passed as a parameter. We created a separate method, which we will discuss later, for handling the quick action. When the quick action is complete, you are expected to call the completion handler with an appropriate boolean value, depending on the success/failure of the quick action.

Let's now talk about the `handleQuickAction` method. It takes in a shortcut item and has a `switch` statement to perform the action accordingly:

1. For `.OpenFavorites` and `.OpenDiscover`, we simply change the selected index of the tab bar controller, so the app jumps right into the particular screen.
2. It takes a little more work for handling the New Restaurant action. We need to first retrieve the restaurant table view controller from the tab bar controller, and then programmatically invoke the *addRestaurant* segue. This will open the New Restaurant screen directly.

You can now run the app and see the results. The quick actions should now work.

Context Menu and Preview

In iOS 13 (or later), you can implement context menus to display additional functionalities for onscreen items. On top of that, it allows your users to preview content without opening the actual screen. A context menu can be triggered by touch and hold gesture. Usually, the user taps and holds an item on screen (e.g. image), iOS then displays a preview of the item with additonal options. The user can further tap the preview item to pop into the full view of the content.

If you've worked with Peek and Pop in 3D Touch before, context menus are very similar to that, except that context menus work on all devices running iOS 13 (or later).

To create a context menu for items in a table view, you can adopt the following methods:

1. `tableView(_:contextMenuConfigurationForRowAt:point:)`
2. `tableView(_:willPerformPreviewActionForMenuWith:animator:)`

Both methods are from the `UITableViewDelegate` protocol and newly introduced in iOS 13 SDK. The first method requires you to return a `UIContextMenuConfiguration` object. This object contains the configuration details for the contextual menu including the action items in the menu and the view controller for previewing the content.

The second method controls what to be performed when the preview is tapped. If this method is not implemented, tapping the preview does nothing. But, say for example, if you want to display the full content of the preview in a view controller, you can provide the implementation in the method.

Now that you have some basic understanding of the implementation, we will modify the FoodPin app to build the context menu in the *Favorites* tab. This menu shows a preview of the restaurant and three menu items. Figure 29-6 illustrates how it works.

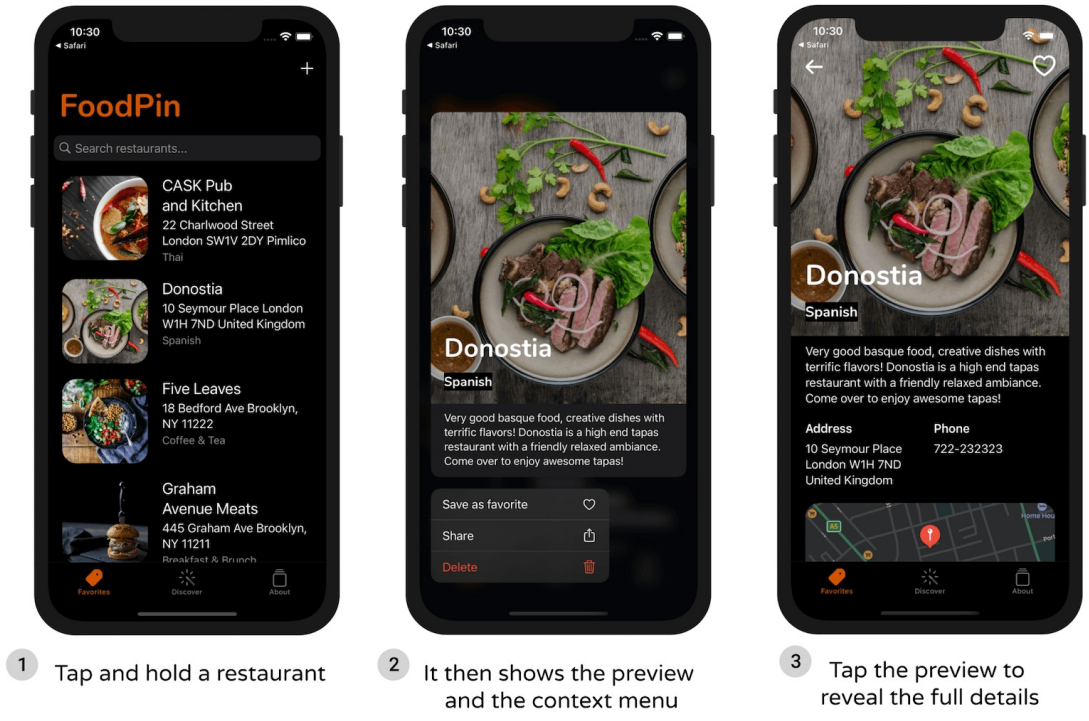


Figure 29-6. Activate the contextual menu in the FoodPin app

Building a Context Menu with Action Items

As mentioned earlier, to create a context menu in a table view, you need to implement the following method in `RestaurantTableViewController`:

```
func tableView(_ tableView: UITableView, contextMenuConfigurationForRowAt indexPath: IndexPath, point: CGPoint) -> UIContextMenuConfiguration? {

}
```

We have to create a `UIContextMenuConfiguration` object that contains the menu items and the preview provider. Here is the code:

```
override func tableView(_ tableView: UITableView, contextMenuConfiguration ForRowAt indexPath: IndexPath, point: CGPoint) -> UIContextMenuConfigurati on? {

    // Get the selected restaurant
    guard let restaurant = self.dataSource.itemIdentifier(for: indexPath) else {
        return nil
    }

    let configuration = UIContextMenuConfiguration(identifier: indexPath.r ow as NSCopying, previewProvider: {

        guard let restaurantDetailViewController = self.storyboard?.instan tiateViewController(withIdentifier: "RestaurantDetailViewController") as? RestaurantDetailViewController else {
            return nil
        }

        restaurantDetailViewController.restaurant = restaurant

        return restaurantDetailViewController

    }) { actions in
```

```
        let favoriteAction = UIAction(title: "Save as favorite", image: UI
Image(systemName: "heart")) { action in

            let cell = tableView.cellForRow(at: indexPath) as! RestaurantT
ableViewCell
            self.restaurants[indexPath.row].isFavorite.toggle()
            cell.favoriteImageView.isHidden = !self.restaurants[indexPath.
row].isFavorite
        }

        let shareAction = UIAction(title: "Share", image: UIImage(systemNa
me: "square.and.arrow.up")) { action in

            let defaultText = NSLocalizedString("Just checking in at ", co
mment: "Just checking in at") + self.restaurants[indexPath.row].name

            let activityController: UIActivityViewController

            if let imageToShare = UIImage(data: restaurant.image as Data)
{
                activityController = UIActivityViewController(activityItem
s: [defaultText, imageToShare], applicationActivities: nil)
            } else  {
                activityController = UIActivityViewController(activityItem
s: [defaultText], applicationActivities: nil)
            }

            self.present(activityController, animated: true, completion: n
il)
        }

        let deleteAction = UIAction(title: "Delete", image: UIImage(system
Name: "trash"), attributes: .destructive) { action in

            // Delete the row from the data store
            if let appDelegate = (UIApplication.shared.delegate as? AppDel
egate) {
                let context = appDelegate.persistentContainer.viewContext
```

```
                let restaurantToDelete = self.fetchResultController.object
(at: indexPath)
                context.delete(restaurantToDelete)

                appDelegate.saveContext()
            }
        }

        // Create and return a UIMenu with the share action
        return UIMenu(title: "", children: [favoriteAction, shareAction, d
eleteAction])
    }

    return configuration
}
```

This code snippet looks a bit complicated but actually you've come across most of the code. The `UIContextMenuConfiguration` class accepts three parameters for initialization:

- **identifier** - this is a unique identifier for the menu configuration object. You may use `nil` to let the system generate a unique identifier for you. Or you provide your own. Here, we use the row number of the selected restaurant as identifier.
- **previewProvider** - this is a block that returns the custom view controller for previewing the content. In the code above, we instantiate the `RestaurantDetailViewController` for previewing the restaurant.
- **actionProvider** - the last parameter accepts a block for creating the menu items of the contextual menu. In the menu, we list three action items including check-in, share, and delete. The action items are very similar to the swipe-for-action items that we created before. To create the menu, you return a `UIMenu` object with the preferred title and action items.

That's it! However, before testing the changes on your device, remember to set the identifier of the Restaurant Detail View Controller to `RestaurantDetailViewController`. Otherwise, you will not be able to instantiate the controller programmatically. Go to `Main.storyboard` and select Restaurant Detail View Controller. Under the Identity inspector, set the storyboard ID to `RestaurantDetailViewController`.

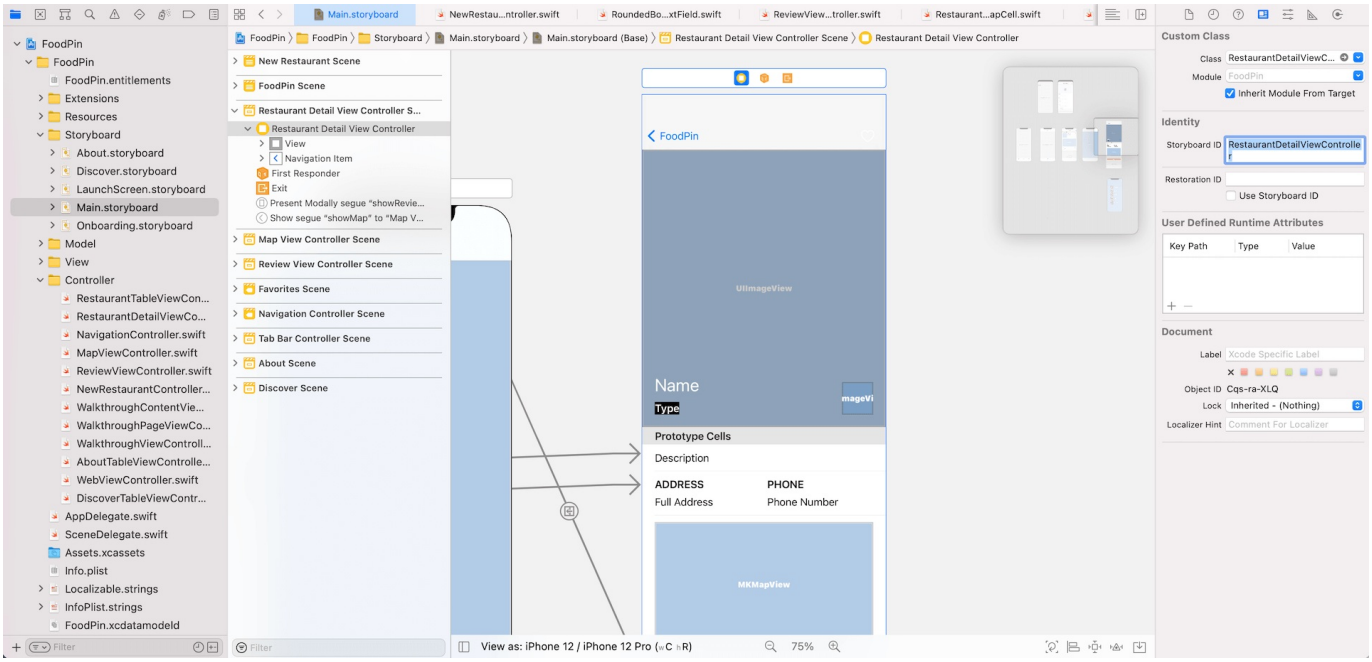


Figure 29-7. Assign an identifier for the Restaurant Detail View Controller

Once you have completed the changes, you can run the app to have a quick test. By tapping and holding a restaurant record, you can bring up a contextual menu similar to the one displayed in figure 29-6.

Showing the Full Content

For now, if you further tap the preview, the system simply dismisses the preview and does nothing. What we want to do is display the full content when the user taps the preview. To make this happen, we have to implement the following

method and tell the system the custom view controller to display:

```
func tableView(_ tableView: UITableView, willPerformPreviewActionForMenuWi
th configuration: UIContextMenuConfiguration, animator: UIContextMenuInter
actionCommitAnimating) {

}
```

Now insert the following code snippet in the `RestaurantTableViewController` class:

```
override func tableView(_ tableView: UITableView, willPerformPreviewAction
ForMenuWith configuration: UIContextMenuConfiguration, animator: UIContext
MenuInteractionCommitAnimating) {

    guard let selectedRow = configuration.identifier as? Int else {
        print("Failed to retrieve the row number")
        return
    }

    guard let restaurantDetailViewController = self.storyboard?.instantiat
eViewController(withIdentifier: "RestaurantDetailViewController") as? Rest
aurantDetailViewController else {

        return
    }

    restaurantDetailViewController.restaurant = self.restaurants[selectedR
ow]

    animator.preferredCommitStyle = .pop
    animator.addCompletion {
        self.show(restaurantDetailViewController, sender: self)
    }
}
```

Our goal is to show the restaurant detail view controller for displaying the full details of a restaurant. For the configuration's identifier, we can reveal the selected row number. By using that, we can create a `RestaurantDetailViewController` with the selected restaurant.

To bring up the view controller, we provide a completion block for the animator. This animator will animate the change and display the specified view controller. Optionally, you can specify your preferred commit style. In the code above, we use the `.pop` style. Later, you may change it to `.dismiss` and try out a different style if you want.

You're good to go. Run the app in the simulator or test it on a real device. The FoodPin app should now let you preview the restaurant and see the full content when you tap the preview.

Summary

In this chapter, I have walked you through some basic APIs of context menus. As you can see, the APIs are pretty easy to use. And you can add quick actions without writing a line of code.

Context menus, introduced in iOS 13, provides an entirely new way for users to interact with their phones. As an app developer, it is your responsibility to deliver a great user experience for your users. Now is the best time to start thinking about how to make use of this new technology to further improve your apps.

For reference, you can download the complete Xcode project from http://www.appcoda.com/resources/swift53/FoodPinContextMenu.zip.

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 and home screen

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.

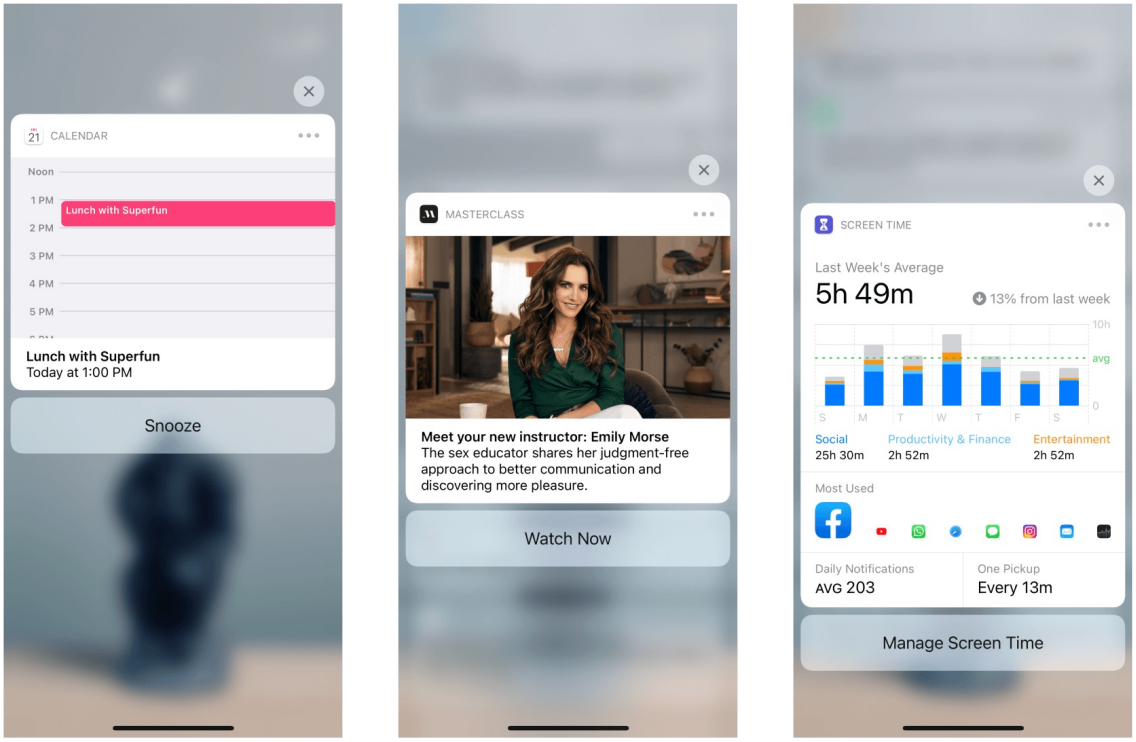


Figure 30-2. Sample user notifications in rich content

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.

Using User Notifications to Increase Customer Engagement

So what features are we going to add in the FoodPin app? Using local notifications is a great way to remind your users about your app. A recent study revealed that only less than 25% of people will use an app for more than one time. In other words, over 75% of users download an app, open it once, and then never return to it.

> More than 75% of App Downloads Open an App Once And Never Come Back
>
> by Erin Griffith
>
> http://fortune.com/2016/05/19/app-economy/

With over two million apps available in the App Store, it is hard to get people notice and download your app. Yet it is even harder to keep people using it. A smart use of user notifications can help you retain your users and improve the user experience of your app.

The User Notifications framework provides different triggers to initiate a local notification:

- **Time-based trigger** - triggers a local notification after a specific amount of time (say, after 10 minutes).
- **Calendar-based trigger** - triggers a local notification at a specified date and time.
- **Location-based trigger** - triggers a local notification when the user reaches

a specific location.

The FoodPin app is designed for food lovers to bookmark their favorite restaurants. Wouldn't it be great if the app suggests one's favorite restaurants when the user reaches a particular location? For example, you've saved a few restaurants in Tokyo. At the time you arrive Tokyo, the app triggers a notification showing a list of your favorite restaurants in the city.

Or you can use a calendar-based trigger to initiate notifications at festival times (say, 10 days before Christmas). The notification can be something like this:

"Hey, Christmas is around the corner. It's time to plan for the holidays and try some good food with your friends. Here are some of your favorite restaurants you can check out."

These are a couple of sample use cases. It's more likely users will return to the app after seeing the notifications.

To keep things simple in this beginner book, we would not implement the above triggers. Instead, I will show you how to use time-based triggers to trigger local notifications. That said, it doesn't mean that the notifications are useless or spammy. And, once you understand the basics of the User Notifications framework, it is not too difficult for you to implement other types of triggers.

What we are going to do is that we will notify the user and recommend him/her a restaurant after a certain period of time (say, 24 hours) since he/she last used the app. Furthermore, we will allow users to interact with the notification. When the user sees the notification, he/she will be given an option to reserve a table. If the user taps the button, it will directly make a call to the restaurant. Figure 30-3 illustrates a sample notification.

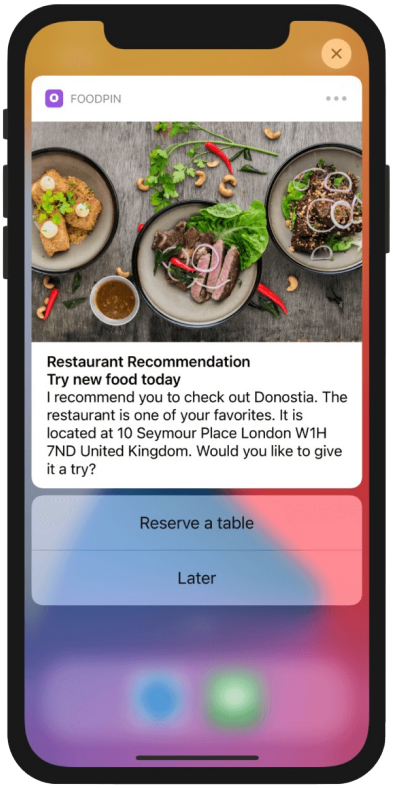


Figure 30-3. The FoodPin app recommends users a restaurant through local notifications

Does it look great? Let's get started and see how you can make the notifications happen in your app.

Using the User Notifications Framework

The User Notifications framework is a framework for managing and scheduling notifications. To implement user facing notifications, the very first thing to do is import the framework in your code, so you can access the APIs bundled in the framework.

Later, we will implement user notifications in both `AppDelegate.swift` and `RestaurantTableViewController.swift`. Insert the following line of code in both files:

```
import UserNotifications
```

Asking for User Permission

Regardless of the type of notifications, you have to ask for the user's authorization and permission before you can send notifications to the user's device.

If you have used iOS for some time, this kind of authorization request shouldn't be new to you. The request is usually prompted when you first launch an app.

So in code, we usually implement the authorization request in the `AppDelegate.swift` file. Insert the following code snippet in the `application(_:didFinishLaunchingWithOptions:)` method:

```
UNUserNotificationCenter.current().requestAuthorization(options: [.alert,
.sound, .badge]) { (granted, error) in

    if granted {
        print("User notifications are allowed.")
    } else {
        print("User notifications are not allowed.")
    }
}
```

It is very simple to prompt users with an authorization request. As you can see from the code above, we call `requestAuthorization` on the `UNUserNotificationCenter` object associated with the application. And we request the ability to display alerts, play sounds, and update the app's badge.

Now run the project to test it out. When the app is launched, you should see the authorization request. Once you accept it, the app is allowed to send notifications to the device. To verify the notification settings, you can go to Setting > FoodPin > Notifications.

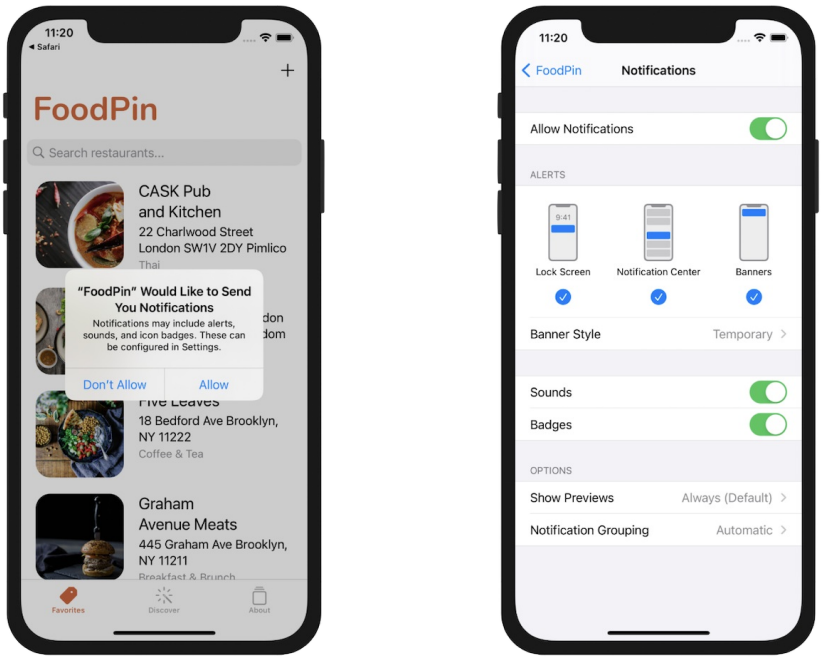


Figure 30-4. Ask for the user's permission and authorization

Creating and Scheduling Notifications

Now the FoodPin app is ready to send notifications to the users. Let's first check out the basic look & feel of a notification in iOS. At the top, it is the title of the notification. The next line is the subtitle, followed by the body of the message. Figure 30-5 is an example.

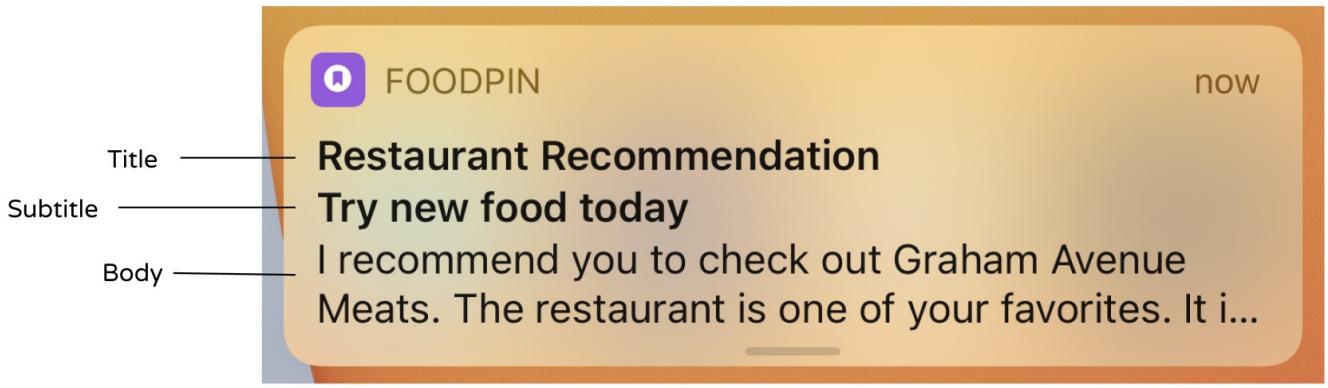


Figure 30-5. A standard notification in iOS 10/11

The content of a user notification is represented by `UNMutableNotificationContent`. To create the content, you instantiate an `UNMutableNotificationContent` object and set its properties to the appropriate data. Here is an example:

```
let content = UNMutableNotificationContent()
content.title = "Restaurant Recommendation"
content.subtitle = "Try new food today"
content.body = "I recommend you to check out Cafe Deadend."
```

If you want to play a sound when the notification is triggered, you can also set the `sound` property of the content:

```
content.sound = UNNotificationSound.default()
```

Scheduling the notification is as simple as creating the `UNNotificationRequest` object with your preferred trigger, and then adding the request to `UNUserNotificationCenter`. Take a look at the code snippet below. This is the code you need to schedule a notification.

```
let trigger = UNTimeIntervalNotificationTrigger(timeInterval: 10, repeats:
false)
let request = UNNotificationRequest(identifier: "foodpin.restaurantSuggest
ion", content: content, trigger: trigger)

// Schedule the notification
UNUserNotificationCenter.current().add(request, withCompletionHandler: nil
)
```

As said before, we want to trigger the notification after a certain period of time. We create a `UNTimeIntervalNotificationTrigger` object and set the time interval to a specific value (e.g. 10 seconds). Then we construct a `UNNotificationRequest` object with the notification content and the trigger. You have to assign the request with a unique identifier. Later, if you want to remove or update the notification, you use this identifier to identify the notification. Finally, you call the `add` method of `UNUserNotificationCenter` with the notification request to schedule the notification.

Now that you should have some ideas about how to create and schedule the notifications, let's implement the Restaurant Recommendation notification. Open `RestaurantTableViewController.swift` and insert the following method:

```
func prepareNotification() {
    // Make sure the restaurant array is not empty
    if restaurants.count <= 0 {
        return
    }

    // Pick a restaurant randomly
    let randomNum = Int.random(in: 0..<restaurants.count)
    let suggestedRestaurant = restaurants[randomNum]

    // Create the user notification
    let content = UNMutableNotificationContent()
    content.title = "Restaurant Recommendation"
    content.subtitle = "Try new food today"
    content.body = "I recommend you to check out \(suggestedRestaurant.nam
e). The restaurant is one of your favorites. It is located at \(suggestedR
estaurant.location). Would you like to give it a try?"
    content.sound = UNNotificationSound.default

    let trigger = UNTimeIntervalNotificationTrigger(timeInterval: 10, repe
ats: false)
    let request = UNNotificationRequest(identifier: "foodpin.restaurantSug
gestion", content: content, trigger: trigger)

    // Schedule the notification
    UNUserNotificationCenter.current().add(request, withCompletionHandler:
nil)

}
```

To better manage the code, we create the `prepareNotification()` method for handling user notifications. We have gone through most of the code before. But the first line of code is new to you. Here we want to randomly pick a restaurant from the favorites and recommend it to the user. This line of code is used to generate a random number:

```
let randomNum = Int.random(in: 0..<restaurants.count)
```

The `random(in:)` function is newly introduced in Swift 4.2 for generating a random number. It takes in a range of numbers and generates a random number within that range. For the code above, let's say you have 10 restaurants in the favorites. The function will generate a number between 0 and 9. With the random number, we can choose a suggested restaurant from the array, and create the notification content.

One thing you have to take note is that we now set the time interval to 10 seconds. This is for demo purpose and ease of testing. In reality, this is too short. You may want to trigger the notification after 24 hours (24 *60* 60 seconds) of use or even longer:

```
let trigger = UNTimeIntervalNotificationTrigger(timeInterval: 86400, repea
ts: false)
```

Now that the `prepareNotification` method is ready, you can call it in the `viewDidLoad` method. Insert this line of code before the end of the method:

```
prepareNotification()
```

Cool! Let's run the project and have a quick test. The notification will not be presented in-app. Therefore, once you launch your app, make sure you go back to home screen or lock screen. Wait for 10 seconds and you should see the notification.

Note: If you test the app using the iPhone simulator, press shift-command-H or the home button to switch to the home screen. To access the lock screen, press command-L.

If the notification appears on the lock screen, you can slide the notification to return to the FoodPin app. And when the device is unlocked, the notification rolls down from the top as a banner. You can swipe it further down to reveal the full content or simply tap it to jump back to the app.

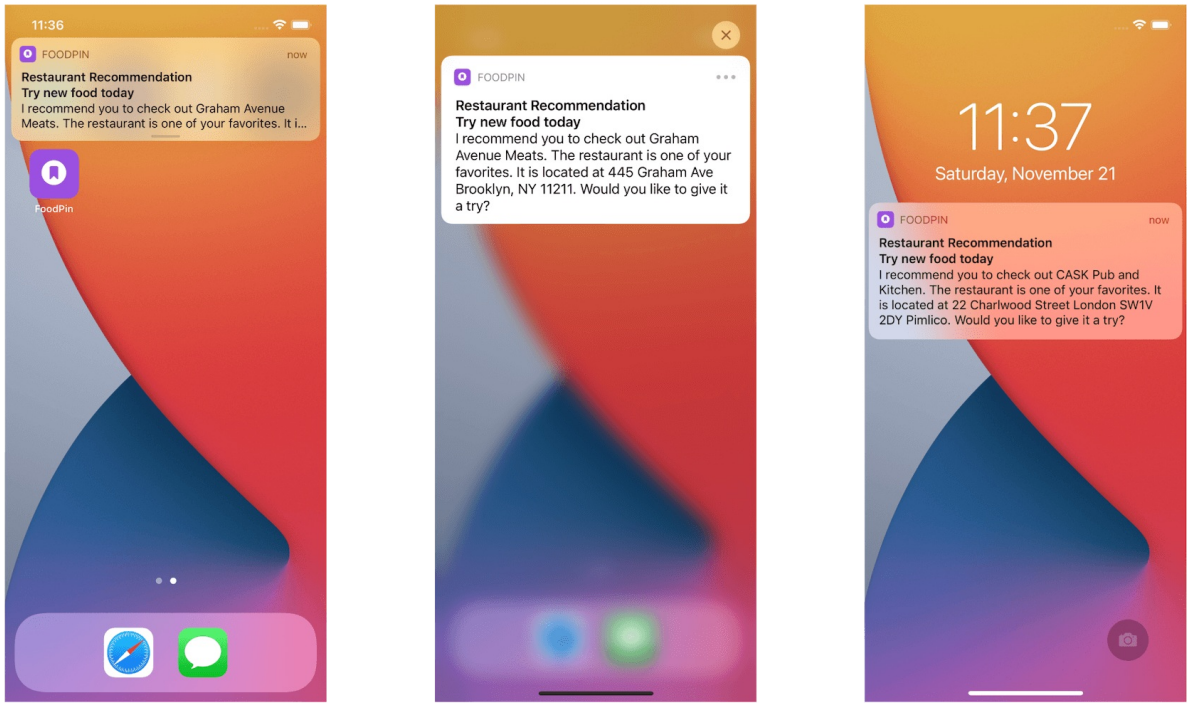


Figure 30-6. User notfications on home screen and lock screen

Adding Images to the Notification

We've talked about notifications with rich content from the beginning of the chapter. So far the notification we have created is in plain text. How can we bundle the image of the suggested restaurant in the notification?

It is as simple as setting the `attachment` property of the `UNMutableNotificationContent` object:

```
content.attachments = [attachment]
```

The `attachments` property accepts an array of `UNNotificationAttachment` objects to display with the notification. Attachment can be images, sounds, audio, and movie files.

Note that you should provide the file URL of the attachment. In our case, it's the image file of the suggested restaurant.

If you're not forgetful, you probably remember that the `image` property of `RestaurantMO` is of type `Data`. So how can we create the image file from the image data in order to create the attachment object?

Let's first check out the code for creating attachment. You can insert the following in the `prepareNotification` method (before the instantiation of the trigger):

```
let tempDirURL = URL(fileURLWithPath: NSTemporaryDirectory(), isDirectory:
true)
let tempFileURL = tempDirURL.appendingPathComponent("suggested-restaurant.
jpg")

if let image = UIImage(data: suggestedRestaurant.image as Data) {

    try? image.jpegData(compressionQuality: 1.0)?.write(to: tempFileURL)
    if let restaurantImage = try? UNNotificationAttachment(identifier: "re
staurantImage", url: tempFileURL, options: nil) {
        content.attachments = [restaurantImage]
    }
}
```

The iOS SDK provides a built-in function called `jpegData(compressionQuality:)` to convert image data into a JPEG image file. What's going on in the code is that we first find the temporary directory for saving the image. The `NSTemporaryDirectory()`

function returns you the directory for temporary files. And, we set the temporary file name as `suggested-restaurant.jpg`. If you print the file path to console, it will be something like this:

```
file:///Users/simon/Library/Developer/CoreSimulator/Devices/DC573158-103F-4D1B-8489-742E3C651D33/data/Containers/Data/Application/C2386E9A-48F7-411B-B485-95EC07CA0D8E/tmp/suggested-restaurant.jpg
```

The `jpegData(compressionQuality:)` function returns you the image's data in JPEG format and then we write the data to a JPEG file. With the image file created in the temporary directory, we can create the `UNNotificationAttachment` object and assign it to the `attachments` property of the notification.

Note: Both jpegData(compressionQuality:) and UNNotificationAttachment throw an exception when it encounters an error. You can refer to the appendix to learn about error handling in Swift and how to use do-try-catch.

It's time to test the app again. Run the project to open the app in the simulator. Remember to go back to home screen and wait for the notification to appear. This time you should see a small thumbnail in the notification. Swipe it down to view a large version of the image.

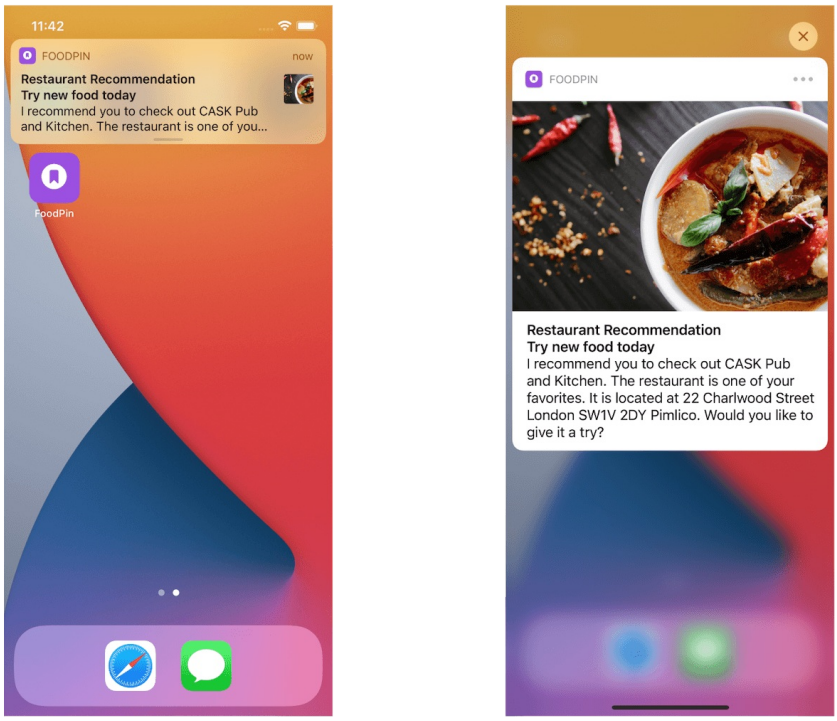


Figure 30-7. Adding an image to notifications

Interacting with User Notifications

Now users have only one way to interact with the notification: tap to launch the app. This is the default action if you do not provide any custom implementation.

Actionable notifications let users respond to the notification without having to switch to the app. One great example of actionable notifications is reminders. When you receive a notification from the Reminders app, you will have the option to manage the reminder directly from the notification. You can either mark it as completed or reschedule the reminder. All these can be done without the need of launching the app.

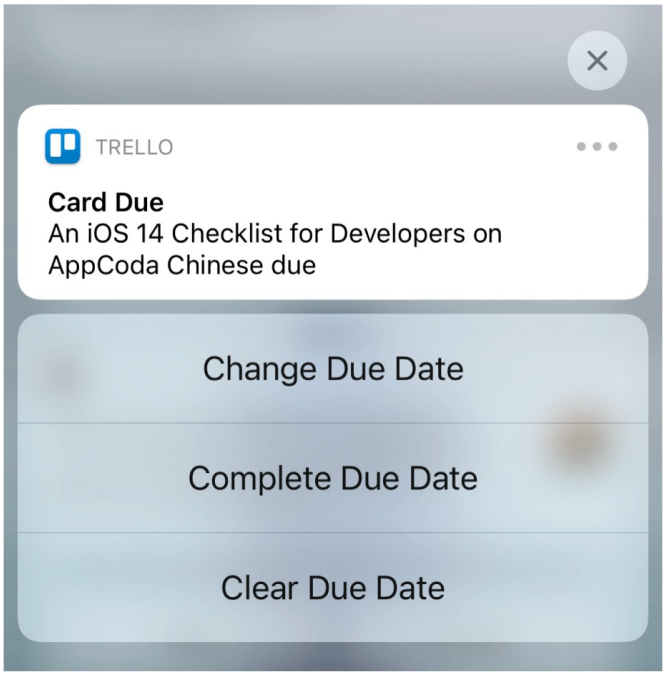


Figure 30-8. A sample notification from Trello

Creating and Registering Custom Actions

With the User Notifications framework, we can implement custom actions for notifications from the FoodPin app. When a notification appears on screen, it provides two options for users to choose:

1. **Reserve a table** - if the user selects this option, the app will make a call to the suggested restaurant so the user can reserve a table.
2. **Later** - for this option, we just dismiss the notification.

To implement custom actions, you need to create a `UNNotificationAction` object and associate it with a notification category. The action object has a unique identifier and a title (e.g. Reserve a table), which appears on the action button. Optionally, you can specify how the action should be performed. By default, action

will be a background action, which means it dismisses the notification and performs your custom task in background. For example, if we write the `Later` action in code, it looks like this:

```
let laterAction = UNNotificationAction(identifier: "foodpin.cancel", title
: "Later", options: [])
```

If the `Later` is selected, the action is to dismiss the notification. Thus, we do not provide additional options when creating the action object.

On the other hand, the *Reserve a table* action will be a foreground action because we have to bring the app to foreground in order to make a call. Thus, the action will be implemented like this:

```
let makeReservationAction = UNNotificationAction(identifier: "foodpin.make
Reservation", title: "Reserve a table", options: [.foreground])
```

Once you set up the action objects, you associate it with a category:

```
let category = UNNotificationCategory(identifier: "foodpin.restaurantactio
n", actions: [makeReservationAction, cancelAction], intentIdentifiers: [],
 options: [])
```

You give the category a unique identifier and pass the action objects to associate with the category. Once you have the category ready, register it to the `UNUserNotificationCenter` object like this:

```
UNUserNotificationCenter.current().setNotificationCategories(["foodpin.res
taurantaction"])
```

Now that we have created the actions and registered them to the notification center, but these actions are not yet associated with the notification. To do that, all you need to do is set the category identifier to the `categoryIdentifier` property of the notification content.

```
content.categoryIdentifier = "foodpin.restaurantaction"
```

That's the code you need to implement custom actions for user notifications. Insert the following code snippet to the `prepareNotification` method (before the trigger variable):

```
let categoryIdentifer = "foodpin.restaurantaction"
let makeReservationAction = UNNotificationAction(identifier: "foodpin.make
Reservation", title: "Reserve a table", options: [.foreground])
let cancelAction = UNNotificationAction(identifier: "foodpin.cancel", titl
e: "Later", options: [])
let category = UNNotificationCategory(identifier: categoryIdentifer, actio
ns: [makeReservationAction, cancelAction], intentIdentifiers: [], options:
 [])
UNUserNotificationCenter.current().setNotificationCategories([category])
content.categoryIdentifier = categoryIdentifer
```

The code above is exactly the same as what we have just discussed. It's time to run the app to try out the notification actions. When the notification banner appears, swipe it down and you should see the custom actions just implemented.

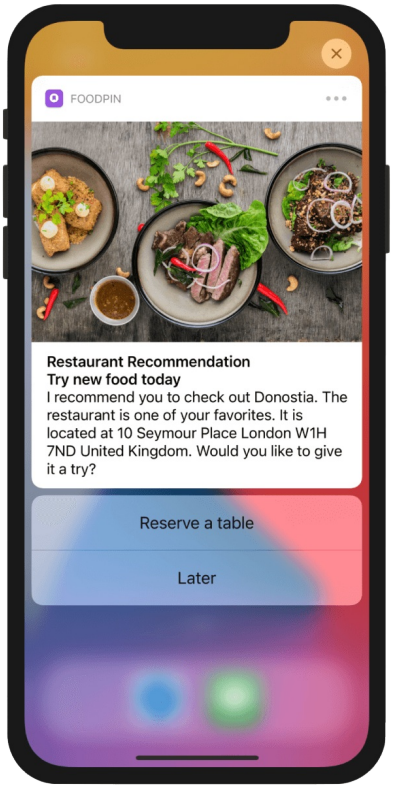


Figure 30-9. Custom actions for User Notifications

Handling the Actions

If you tap the *Reserve a table* button, it brings up the FoodPin app. However, it will not call the restaurant for you. As explained in the earlier section, the action object has an option that indicates how the action should be performed. For the *Later* button, we did not provide any additional options, so it defaults to dismiss the notification. For the *Reserve a table* button, we set the option to `.foreground`, that will bring the app to foreground.

So how can we handle the action when the app returns to foreground?

The `UNUserNotificationCenterDelegate` protocol in the User Notifications framework is designed for this purpose. The protocol defines a method for responding to the actionable notifications:

```
optional func userNotificationCenter(_ center: UNUserNotificationCenter,
                            didReceive response: UNNotificationResponse,
                  withCompletionHandler completionHandler: @escaping () -> Vo
id)
```

To handle the action and execute custom code, you need to implement the protocol in a delegate object and assign it to the notification center object. When the app is returned to foreground, the method will be called accordingly.

We will implement this protocol in `AppDelegate`. But before we do that, you probably have another question in mind. We're going to make a call to the suggested restaurant. How can we pass the restaurant's phone number from `RestaurantTableViewController` to `AppDelegate`?

The notification content has a property named `userInfo` for you to store custom information in the form of a dictionary. For example, you can store the phone number in the notification like this:

```
content.userInfo = ["phone": suggestedRestaurant.phone]
```

Put the line of code above in the `prepareNotification` method right below `content.sound`.

With the phone number associated with the notification, we now implement the `UNUserNotificationCenterDelegate` protocol.

I have introduced a Swift feature known as Extensions in earlier chapters. Extensions allow you to expand the functions of an existing type. You can also make use of it to adopt a protocol and make the code more organized. Let's see

how it can be done. At the end of the `AppDelegate.swift` file, insert the following code snippet to create the extension:

```
extension AppDelegate: UNUserNotificationCenterDelegate {
    func userNotificationCenter(_ center: UNUserNotificationCenter, didRec
eive response: UNNotificationResponse, withCompletionHandler completionHan
dler: @escaping () -> Void) {

        if response.actionIdentifier == "foodpin.makeReservation" {
            print("Make reservation...")
            if let phone = response.notification.request.content.userInfo[
"phone"] {
                let telURL = "tel://\(phone)"
                if let url = URL(string: telURL) {
                    if UIApplication.shared.canOpenURL(url) {
                        print("calling \(telURL)")
                        UIApplication.shared.open(url)
                    }
                }
            }
        }

        completionHandler()
    }
}
```

When an action of a notification is selected by a user, the `userNotificationCenter(_:didReceive:withCompletionHandler:)` is called. So we provide our own implementation for making a call to the suggested restaurant.

As we only need to handle the *Reserve a table* action, we first verify the identifier of the action. Then we retrieve the phone number of the restaurant from the `userInfo` property of the notification content.

In iOS, you can launch some of the system apps by using specific URLs. In this case, we want to open the Phone app to call up the number. You can use the `tel` scheme to initiate a call. Here is a sample URL:

```
tel://<phone-number>
```

So in the code above, we construct the `telURL` and then call the `open` method of `UIApplication` to launch the Phone app.

At the end of the method, it is required to call the `completionHandler` block to let the system know that you are done processing the notification.

The last thing you have to do is set the delegate of the notification center. In the `application(_:didFinishLaunchingWithOptions:)` of `AppDelegate`, add the following line of code:

```
UNUserNotificationCenter.current().delegate = self
```

That's it. Run the project and deploy the app on a real device for testing. You have to use a real device because you can't make a call using the simulator. If you select the *Reserve a table* button, the app will launch the FoodPin app and show you an option to make the call.

Summary

The User Notifications framework is a nice framework for developers to manage and schedule user notifications. In this chapter, I have given you an overview of the framework and demonstrate how to schedule a local notification.

With interactive and rich content notifications, you can improve the user experience and engagement of your app. This is a good way to increase app retention rate. When you're going to build your next app, try to make use of user notifications and see how it can add value to your app.

Lastly, thanks for reading this book. This has been a long journey for both of us. I wish you the best of luck and hope you will release your app very soon. If you've submitted an app that's been approved, I'd love to hear about your success story. Feel free to email me at simonng@appcoda.com and remember to join our developer group on Facebook (https://www.facebook.com/groups/appcoda).

For reference, you can download the complete Xcode project from http://www.appcoda.com/resources/swift53/FoodPinUserNotifications.zip.

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, seco
ndMessage];
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 Breakfas
t", @"Hamburger", @"Ham and Egg Sandwich"];
```

Swift:

```
var recipes = ["Egg Benedict", "Mushroom Risotto", "Full Breakfast", "Hamb
urger", "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 Breakf
ast", "Hamburger", "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 Panin
i"]
```

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:

```
var companies = ["AAPL" : "Apple Inc", "GOOG" : "Google Inc", "AMZN" : "Am
azon.com, Inc", "FB" : "Facebook Inc"]
```

The key and value in the key-value pairs are separated by a colon. The key-value pairs are surrounded by a pair of square brackets, each of which is separated by commas.

Like array and other variables, Swift automatically detects the type of the key and value. But, if you like, you can specify the type information by using the following syntax:

```
var companies: [String: String] = ["AAPL" : "Apple Inc", "GOOG" : "Google Inc", "AMZN" : "Amazon.com, Inc", "FB" : "Facebook Inc"]
```

To iterate through a dictionary, use the `for-in` loop.

```
for (stockCode, name) in companies {
    print("\(stockCode) = \(name)")
}

// You can also use the keys and values properties to
// retrieve the keys and values of the dictionary.
for stockCode in companies.keys {
    print("Stock code = \(stockCode)")
}
for name in companies.values {
    print("Company name = \(name)")
}
```

To access the value of a particular key, specify the key using the subscript syntax. If you want to add a new key-value pair to the dictionary, simply use the key as the subscript and assign it with a value like below:

```
companies["TWTR"] = "Twitter Inc"
```

Now the `companies` dictionary contains a total of 5 items. The "TWTR":"Twitter Inc" pair is automatically added to the `companies` dictionary.

Set

A set is very similar to an array. While an array is an ordered collection, a set is an unordered collection. Items in an array can be duplicated. A set stores no repeated values.

To declare a set, you can write like this:

```
var favoriteCuisines: Set = ["Greek", "Italian", "Thai", "Japanese"]
```

The syntax is very similar to creating an array, but you have to explicitly specify the type `Set`.

As mentioned, a set is an unordered collection of distinct items. If you declare a set of duplicated values, the set will not store the duplicates. Here is an example:

```
var favoriteCuisines: Set = ["Greek", "Italian", "Thai", "Japanese", "Thai", "Italian"]
```

```
{"Japanese", "Italian", "Thai", "Greek"}
```

Operations on sets are quite similar to an array. You can use `for-in` loop to iterate over a set. But to add a new item to a set, you can't use the `+=` operator. You have to call the `insert` method:

```
favoriteCuisines.insert("Indian")
```

With sets, you can easily determine the values two sets have in common, or vice versa. For example, you use two sets to represent the favorite cuisines of two persons:

```
var tomsFavoriteCuisines: Set = ["Greek", "Italian", "Thai", "Japanese"]
var petersFavoriteCuisines: Set = ["Greek", "Indian", "French", "Japanese"
]
```

You want to find out the common cuisines which they both love. You can call the `intersection` method like this:

```
tomsFavoriteCuisines.intersection(petersFavoriteCuisines)
```

This will return you the result: `{"Greek", "Japanese"}`.

Or if you want to determine which cuisines they don't have in common, you can use the `symmetricDifference` method:

```
tomsFavoriteCuisines.symmetricDifference(petersFavoriteCuisines)
// Result: {"French", "Italian", "Thai", "Indian"}
```

Classes

In Objective-C, you create separate interface (.h) and implementation (.m) files for classes. Swift no longer requires developers to do that. You can define classes in a single file (.swift) without separating the external interface and implementation.

To define a class, you use the class keyword. Here is a sample class in Swift:

```
class Recipe {
    var name: String = ""
    var duration: Int = 10
    var ingredients: [String] = ["egg"]
}
```

In the above example, we define a `Recipe` class with three properties including name duration and ingredients. Swift requires you to provide the default values of the properties. You'll end up with a compilation error if the initial values are missing.

What if you don't want to assign a default value? Swift allows you to write a question mark (`?`) after the type of a value that indicates the value is optional.

```
class Recipe {
    var name: String?
    var duration: Int = 10
    var ingredients: [String]?
}
```

In the above code, the `name` and `ingredients` properties are automatically assigned with a default value of `nil`. We will discuss optionals in details in later section. To create an instance of a class, just use the below syntax:

```
var recipeItem = Recipe()
// You use the dot notation to access or change the property of an instanc
e.
recipeItem.name = "Mushroom Risotto"
recipeItem.duration = 30
recipeItem.ingredients = ["1 tbsp dried porcini mushrooms", "2 tbsp olive
oil", "1 onion, chopped", "2 garlic cloves", "350g/12oz arborio rice", "1.
2 litres/2 pints hot vegetable stock", "salt and pepper", "25g/1oz butter"
]
```

Swift allows you to subclass classes and adopt protocols. For example, if you have a `SimpleTableViewController` class that extends from `UIViewController` and adopts both `UITableViewDelegate` and `UITableViewDataSource` protocols, you can write the class declaration like this:

```
class SimpleTableViewController : UIViewController, UITableViewDelegate, U
ITableViewDataSource
```

Methods

Similar to other object-oriented languages, Swift allows you to define functions in a class known as *methods*. You can use the `func` keyword to declare a method. Here is a sample method without return values and parameters:

```
class TodoManager {
    func printWelcomeMessage() {
        print("Welcome to My ToDo List")
    }
}
```

In Swift, you call a method by using dot syntax:

```
todoManager.printWelcomeMessage()
```

If you need to declare a method with parameters and return values, the method will look this:

```
class TodoManager {
    func printWelcomeMessage(name:String) -> Int {
        print("Welcome to \(name)'s ToDo List")

        return 10
    }
}
```

The syntax looks a bit awkward especially for the `->` operator. The above method takes a name parameter in String type as the input. The `->` operator is used as an indicator for methods with a return value. In the above code, you specify a return type of `Int` that returns the total number of todo items. Below demonstrates how you call the method:

```
var todoManager = TodoManager()
let numberOfTodoItem = todoManager.printWelcomeMessage(name: "Simon")
print(numberOfTodoItem)
```

Control Flow

Control flow and loops employ a very C-like syntax. As you can see in the previous section, Swift provides for-in loop to iterate through arrays and dictionaries.

for loops

In case you just want to iterate over a range of values, you can use the `...` or `..<` operators. These are the new operators introduced in Swift for expressing a range of values. Here is an example:

```
for i in 0..<5 {
    print("index = \(i)")
}
```

This would print out the following result in console:

```
index = 0
index = 1
index = 2
index = 3
index = 4
```

So what's the difference between `..<` and `...`? If we replace the `..<` operator with `...` in the above example, this defines a range that runs from 0 to 5 and 5 is included in the range. Here is the console result:

```
index = 0
index = 1
index = 2
index = 3
index = 4
index = 5
```

if-else statement

Just like Objective-C, you can use `if` statement to execute code based on a certain condition. The syntax of the `if-else` statement is pretty similar to that in Objective-C. Swift just makes the syntax even simpler in that you no longer need to enclose the condition within a pair of round brackets.

```
var bookPrice = 1000;
if bookPrice >= 999 {
    print("Hey, the book is expensive")
} else {
    print("Okay, I can affort it")
}
```

switch statement

I'd just like to highlight the switch statement in Swift that is a dramatic improvement compared to its Objective-C counterpart. Take a look at the following sample switch statement. Do you notice anything special?

```
switch recipeName {
    case "Egg Benedict":
        print("Let's cook!")
    case "Mushroom Risotto":
        print("Hmm... let me think about it")
    case "Hamburger":
        print("Love it!")
    default:
        print("Anything else")
}
```

First up, the `switch` statement can now handle strings. You can't do switching on `NSString` in Objective-C. You had to use several if statements to implement the above code. At last, Swift brings us this most sought after utilization of switch statement.

Another interesting feature that you may notice is that there is no break. Recalled that in Objective-C, you need to add a break in every switch case. Otherwise, it will fall through to the next case. In Swift, you do not need to add the break statement explicitly. Switch statements in Swift do not fall through the bottom of each case and into the next one. Instead, as soon as the first matching case completes, the entire switch statement completes its execution.

On top of that, the switch statement now supports range matching. Take a look at the below code:

```
var speed = 50
switch speed {
case 0:
    print("stop")
case 0...40:
    print("slow")
case 41...70:
    print("normal")
case 71..<101:
    print("fast")
default:
    print("not classified yet")
}

// as the speed falls within the range of 41 and 70, it'll print normal to
 console
```

The switch case lets you check values within a certain range by using two new operators: `...` and `..<`. Both operators serve as a shortcut for expressing a range of values.

Consider the sample range of "41...70", the `...` operator defines a range that runs from 41 to 70, including both 41 and 70. If we use the `..<` operator instead of `...` in the example, this defines a range that runs from 41 to 69. In other words, 70 is excluded from the range.

Tuples

Swift introduces an advanced type that is not available in Objective-C known as tuples. Tuples allow developers to create a group of values and pass it around. Consider that you're developing a method call to return multiple values, you can now use tuples as a return value instead of returning a custom object.

Tuples treat multiple values as a single compound value. Here is an example:

```
let company = ("AAPL", "Apple Inc", 93.5)
```

The above code creates a tuple that includes stock code, company name and stock price. As you may be aware, you're allowed to put any value of any type within a tuple. You can decompose the values of tuples and use it like this:

```
let (stockCode, companyName, stockPrice) = company
print("stock code = \(stockCode)")
print("company name = \(companyName)")
print("stock price = \(stockPrice)")
```

A better way to use tuple is to give each element in tuple a name and you can access the element value by using dot notation. Here is another example:

```
let product = (id: "AP234", name: "iPhone X", price: 599)
print("id = \(product.id)")
print("name = \(product.name)")
print("price = USD\(product.price)")
```

A common use of tuples is to serve as a return value. In some cases, you want to return multiple values in a method without using a custom class. You can use tuples as the return value like the following example:

```
class Store {
    func getProduct(number: Int) -> (id: String, name: String, price: Int)
 {
        var id = "IP435", name = "iMac", price = 1399
        switch number {
        case 1:
            id = "AP234"
            name = "iPhone X"
            price = 999
        case 2:
            id = "PE645"
            name = "iPad Pro"
            price = 599
        default:
            break
        }

        return (id, name, price)
    }
}
```

In the above code, we create a method call named `getProduct` that takes in a number and returns a product as tuple. You can call the method and store the return value like below:

```
let store = Store()
let product = store.getProduct(number: 2)
print("id = \(product.id)")
print("name = \(product.name)")
print("price = USD\(product.price)")
```

Optionals Overview

What is optional? When declaring variables in Swift, they are designated as non-optional by default. In other words, you have to assign a non-nil value to the variable. If you try to set a `nil` value to a non-optional, the compiler will say, "Nil cannot be assigned to type String!"

```
var message: String = "Swift is awesome!" // OK
message = nil // compile-time error
```

The same applies when declaring properties in a class. The properties are designated as non-optional by default.

```
class Messenger {
    var message1: String = "Swift is awesome!" // OK
    var message2: String // compile-time error
}
```

You will get a compile-time error for `message2` because it's not assigned with an initial value. If you have written in Objective-C before, you may be a bit surprised. In Objective-C or another programming language (e.g. Javascript), you won't get any compile-time error when assigning `nil` to a variable or declaring a property without initial value:

```
NSString *message = @"Objective-C will never die!";
message = nil;

class Messenger {
    NSString *message1 = @"Objective will never die!";
    NSString *message2;
}
```

However, it doesn't mean you can't declare a property without assigning an initial value in Swift. Swift introduces optional type to indicate the absence of a value. It is defined by adding a question mark `?` operator after the type declaration. Here is an example:

```
class Messenger {
    var message1: String = "Swift is awesome!" // OK
    var message2: String? // OK
}
```

You can still assign a value when the variable is defined as optional. However if the variable is not assigned with any value like the below code, its value automatically defaults to `nil`.

Why Optionals?

Swift is designed for safety. As Apple mentioned, optionals are an example of the fact that Swift is a type safe language. As you can see from the above examples, Swift's optionals provide compile-time checks that would prevent some common programming errors happening at run-time. Let's go through the below example and you will have a better understanding of the power of optionals.

Consider the following method in Objective-C:

```
- (NSString *)findStockCode:(NSString *)company {
    if ([company isEqualToString:@"Apple"]) {
        return @"AAPL";
    } else if ([company isEqualToString:@"Google"]) {
        return @"GOOG";
    }

    return nil;
}
```

You can use the `findStockCode` method to get the stock code for a particular listed company. For demo purposes, the method only returns you the stock code of Apple and Google. For other inputs, it returns `nil`.

Assuming the method is defined within the same class and we use it like this:

```
NSString *stockCode = [self findStockCode:@"Facebook"]; // nil is returned
NSString *text = @"Stock Code - ";
NSString *message = [text stringByAppendingString:stockCode]; // runtime e
rror
NSLog(@"%@", message);
```

The code can compile correctly but as the method returns `nil` for Facebook, a runtime exception is thrown when running the app. With Swift's optionals, instead of discovering the error at runtime, it reveals the error at compile time. If we rewrite the above example in Swift, it will look like this:

```
func findStockCode(company: String) -> String? {
    if (company == "Apple") {
        return "AAPL"
    } else if (company == "Google") {
        return "GOOG"
    }

    return nil
}

var stockCode: String? = findStockCode(company: "Facebook")
let text = "Stock Code - "
let message = text + stockCode  // compile-time error
print(message)
```

`stockCode` is defined as an optional. That means it can either contain a string or `nil`. You can't execute the above code because the compiler detects a potential error ("value of optional type String? is not unwrapped) and tells you to correct it.

As you can see from the example, Swift's optionals reinforce the nil-check and offer compile-time cues to developers. Thus, the use of optionals contributes to better code quality.

Unwrapping Optionals

So how can we make the code work? Apparently, we need to test if `stockCode` contains a nil value or not. We modify the code as follows:

```
var stockCode: String? = findStockCode(company: "Facebook")
let text = "Stock Code - "
if stockCode != nil {
    let message = text + stockCode!
    print(message)
}
```

Just like the Objective-C counterpart, we used `if` to perform the nil-check. Once we know the optional must contain a value, we unwrap it by placing an exclamation mark (`!`) to the end of the optional's name. In Swift, this is known as *forced unwrapping*. You use the `!` operator to unwrap an optional and reveal the underlying value.

Referring to the above example, we only unwrap the `stockCode` optional after the nil-check. We know the optional must contain a non-nil value before unwrapping it using the `!` operator. It is always recommended to ensure that an optional contains a value before unwrapping it.

But what if we forget the verification like below?

```
var stockCode:String? = findStockCode(company: "Facebook")
let text = "Stock Code - "
let message = text + stockCode!  // runtime error
```

There will be no compile-time error. The compiler assumes that the optional contains a value as forced unwrapping is used. When you run the app, a runtime error is shown in the console.

Optional Binding

Other than forced unwrapping, optional binding is a simpler and recommended way to unwrap an optional. You use optional binding to check if the optional contains a value or not. If it does contain a value, unwrap it and put it into a temporary constant or variable.

There is no better way to explain optional binding than using an example. We convert the sample code in the previous example into optional binding:

```
var stockCode: String? = findStockCode(company: "Facebook")
let text = "Stock Code - "
if let tempStockCode = stockCode {
    let message = text + tempStockCode
    print(message)
}
```

The `if let` (or `if var`) are the two keywords of optional binding. In plain English, the code says, "If `stockCode` contains a value, unwrap it, set its value to `tempStockCode` and execute the conditional block. Otherwise, just skip the block." As `tempStockCode` is a new constant, you no longer need to use the `!` suffix to access its value.

You can further simplify the code by evaluating the function in the `if` statement:

```
let text = "Stock Code - "
if var stockCode = findStockCode(company: "Apple") {
    let message = text + stockCode
    print(message)
}
```

Here `stockCode` is not an optional. There is no need to use the `!` suffix to access its value in the conditional block. If a `nil` value is returned from the function, the block will not be executed.

Optional Chaining

Before explaining optional chaining, let's tweak the original example. We create a new class named Stock with the code and price properties, which are optionals. The `findStockCode` function is modified to return a `Stock` object instead of String.

```
class Stock {
    var code: String?
    var price: Double?
}

func findStockCode(company: String) -> Stock? {
    if (company == "Apple") {
        let aapl: Stock = Stock()
        aapl.code = "AAPL"
        aapl.price = 90.32

        return aapl

    } else if (company == "Google") {
        let goog: Stock = Stock()
        goog.code = "GOOG"
        goog.price = 556.36

        return goog
    }

    return nil
}
```

We rewrite the original example as below. We first find the stock code/symbol by calling the `findStockCode` function. And then we calculate the total cost needed when buying 100 shares of the stock.

```
if let stock = findStockCode(company: "Apple") {
    if let sharePrice = stock.price {
        let totalCost = sharePrice * 100
        print(totalCost)
    }
}
```

As the return value of `findStockCode()` is an optional, we use optional binding to check if it contains an actual value. Apparently, the price property of the `Stock` class is an optional. Again we use the `if let` statement to test if `stock.price` contains a non-nil value.

The above code works without any error. Instead of writing nested `if let`, you can simplify the code by using *Optional Chaining*. The feature allows us to chain multiple optionals together with the `?.` operator. Here is the simplified version of the code:

```
if let sharePrice = findStockCode(company: "Apple")?.price {
    let totalCost = sharePrice * 100
    print(totalCost)
}
```

Optional chaining provides an alternative way to access the value of price. The code now looks a lot cleaner and simpler. Here I just cover the basics of optional chaining. You can find further information about optional chaining in Apple's Swift guide.

Failable Initializers

Swift has a feature called *Failable Initializers*. Initialization is the process of providing initial values to each of the stored properties of a class. In some cases, the initialization of an instance may fail. Now such failure can be reported using a failable initializer. The resulting value of a failable initializer either contains the object or `nil`. You will need to use `if let` to check if the initialization is successful or not. Let me give you an example:

```
let myFont = UIFont(name : "AvenirNextCondensed-DemiBold", size: 22.0)
```

The initialization of the `UIFont` object will fail if the font file doesn't exist or is unreadable. This initialization failure will report using a failable initializer. The returned object is an optional that can either be the object itself or nil. Thus, we need to use `if let` to handle the optional:

```
if let myFont = UIFont(name : "AvenirNextCondensed-DemiBold", size: 22.0)
{

   // Further processing

}
```

Generics

The concept of Generics is not new and has been around for a long time in other programming languages like Java. For iOS developers, however, you may be new to Generics.

Generic Functions

Generics are one of the most powerful features of Swift and allow you to write flexible functions. So what are Generics exactly? Well, let's take a look at an example. Suppose you're developing a `process` function:

```
func process(a: Int, b: Int) {
     // do something
}
```

The function accepts two integer values for further processing. What if you need to take in other types of values like `Double`? You probably write another function like this:

```
func process(a: Double, b: Double) {
    // do something
}
```

Both functions look very similar. Assuming the bodies of the functions are identical, the main difference is the types of inputs they take in. With Generics, you can simplify them into one generic function that handles multiple input types:

```
func process<T>(a: T, b: T) {
    // do something
}
```

Now it defines a placeholder type instead of an actual type name. The `<T>` after the function name indicates that this is a generic function. For the function arguments, the actual type name is replaced with a generic type `T`.

You can call the process function in the same way. The actual type to use in place of `T` will be determined each time the function is called.

```
process(a: 689, b: 167)
```

Generic Functions with Type Constraints

Let's take a look at another example. Suppose you're writing another function to compare if two integer values are equal.

```
func isEqual(a: Int, b: Int) -> Bool {
    return a == b
}
```

If you need to compare other types of value such as String, you'll write another function like this:

```
func isEqual(a: String, b: String) -> Bool {
    return a == b
}
```

With Generics, you'll combine the two functions into one:

```
func isEqual<T>(a: T, b: T) -> Bool {
    return a == b
}
```

Again, we use `T` as a placeholder of the value types. But if you test out the above code in Xcode, the function will not compile. The problem lies with the `a==b` equality check. Though the function accepts values with any types, not every type can support the equal to operator (`==`). This is why Xcode indicates an error. In this case, you need to apply a type constraint for the generic function.

```
func isEqual<T: Equatable>(a: T, b: T) -> Bool {
    return a == b
}
```

You write type constraint by placing a protocol constraint after a type parameter's name, separated by a colon. Here the Equatable is the protocol constraint. In other words, the function will only accept values that support the Equatable protocol.

In Swift, it comes with a standard protocol called *Equatable*. For any types conforming to the *Equatable* protocol, they support the equal to (`==`) operator. All standard types like `String` , `Int` , `Double` support the `Equatable` protocol.

So you can use the `isEqual` function like this:

```
isEqual(a: 3, b: 3)             // true
isEqual(a: "test", b: "test")   // true
isEqual(a: 20.3, b: 20.5)       // false
```

Generic Types

You are not limited to use Generics in functions. Swift allows you to define your own generic types. This can be custom classes or structure. The built-in Array and Dictionary are examples of generic types.

Let's take a look at the below example:

```
class IntStore {
    var items = [Int]()

    func addItem(item: Int) {
        items.append(item)
    }

    func findItemAtIndex(index: Int) -> Int {
        return items[index]
    }
}
```

The `IntStore` class is a simple class to store an array of `Int` items. It provides two methods for:

- Adding a new item to the store
- Returning a specific item from the store

Apparently, the `IntStore` class supports items in `Int` type. Wouldn't it be great if you can define a generic `ValueStore` class that manages any types of values? Here is the generic version of the class:

```
class ValueStore<T> {
    var items = [T]()

    func addItem(item: T) {
        items.append(item)
    }

    func findItemAtIndex(index: Int) -> T {
        return items[index]
    }
}
```

Like what you have learned in the Generic functions section, you use a placeholder type parameter (T) to indicate a generic type. The type parameter after the class name indicates the class is a generic type.

To instantiate the class, you write the type to be stored in the `ValueStore` within angle brackets.

```
var store = ValueStore<String>()
store.addItem(item: "This")
store.addItem(item: "is")
store.addItem(item: "generic")
store.addItem(item: "type")
let value = store.findItemAtIndex(index: 1)
```

You can call the method the same way as before.

Computed Properties

A computed property does not actually store a value. Instead, it provides its own getter and setter to compute the value. Here is an example:

```
class Hotel {
    var roomCount: Int
    var roomPrice: Int
    var totalPrice: Int {
        get {
            return roomCount * roomPrice
        }
    }

    init(roomCount: Int = 10, roomPrice: Int = 100) {
        self.roomCount = roomCount
        self.roomPrice = roomPrice
    }
}
```

The `Hotel` class has two stored properties: `roomPrice` and `roomCount`. To calculate the total price of a hotel, we can simply multiply `roomPrice` by `roomCount`. In the past, you might create a method that performs the calculation and returns the total price. With Swift, you can use computed properties instead. In the example, `totalPrice` is a computed property. Rather than storing a fixed value, it defines a custom getter that actually performs the calculation and returns the total price of the rooms. Just like stored properties, you can access the computed property through dotted syntax:

```
let hotel = Hotel(roomCount: 30, roomPrice: 100)
print("Total price: \(hotel.totalPrice)")
// Total price: 3000
```

Optionally, you can define a custom setter for the computed property. Consider the same example again:

```
class Hotel {
    var roomCount: Int
    var roomPrice: Int
    var totalPrice: Int {
        get {
            return roomCount * roomPrice
        }

        set {
            let newRoomPrice = Int(newValue / roomCount)
            roomPrice = newRoomPrice
        }
    }

    init(roomCount: Int = 10, roomPrice: Int = 100) {
        self.roomCount = roomCount
        self.roomPrice = roomPrice
    }
}
```

Here we define a custom setter to calculate the new room price when the value of total price is updated. When a new value of `totalPrice` is set, a default name of `newValue` can be used in the setter. Base on `newValue`, you can then perform the calculation and update the `roomPrice` accordingly.

Could you use methods instead of computed properties? Sure. To me, it is a matter of coding style. Computed properties are especially useful for performing simple conversions and calculations. As you can see from the above example, the implementation is much cleaner.

Property Observers

Property observers are one of my favorite features of Swift. Property observers observe and respond to changes in a property's value. The observers are called every time a property's value is set. You have the option to define two kinds of observers on a property:

- `willSet` is called just before the value is stored.
- `didSet` is called immediately after the new value is stored.

Consider the `Hotel` class again. For instance, we want to limit the room price to a thousand dollars. Whenever a caller sets the room price to a value larger than 1000, we will set it to 1000. You can then use a property observer to monitor the value change like this:

```
class Hotel {
    var roomCount: Int
    var roomPrice: Int {
        didSet {
            if roomPrice > 1000 {
                roomPrice = 1000
            }
        }
    }

    var totalPrice: Int {
        get {
            return roomCount * roomPrice
        }

        set {
            let newRoomPrice = Int(newValue / roomCount)
            roomPrice = newRoomPrice
        }
    }

    init(roomCount: Int = 10, roomPrice: Int = 100) {
        self.roomCount = roomCount
        self.roomPrice = roomPrice
    }
}
```

Say, you set the `roomPrice` property to `2000`. The `didSet` observer will be called and perform the validation. Because the value is larger than 1000, the room price is then set it to 1000. As you can see, property observers are particularly useful for value change notifications.

Failable Casts

`as!` (or `as?`) is known as a failable cast operator. You have to either use `as!` or `as?` to downcast an object to a subclass type. If you're quite sure that the downcasting will succeed, you can use `as!` to force the casting. Here is an example:

```
let cell = tableView.dequeueReusableCell(withIdentifier: cellIdentifier, f
or: indexPath) as! RestaurantTableViewCell
```

If you're not sure whether the casting will succeed, just use the `as?` operator. By using `as?`, it returns an optional value, but in case the downcasting fails, the value will be `nil`.

repeat-while

Starting from Swift 2, Apple introduces a new control flow operator called `repeat-while`, which is to replace the classic `do-while` loop. Here is an example:

```
var i = 0
repeat {
    i += 1
    print(i)
} while i < 10
```

`repeat-while` evaluates its condition at the end of each pass through the loop. If the condition is `true`, it repeats the block of code again. It exits the loop when the condition evaluates to `false`.

for-in where Clauses

Not only can you iterate over all items in an array using a `for-in` loop, you can define a condition to filter the items using the `where` clause. When you loop through an array, for example, only those items that meet the criteria will be processed.

```
let numbers = [20, 18, 39, 49, 68, 230, 499, 238, 239, 723, 332]
for number in numbers where number > 100 {
    print(number)
}
```

In the above example, it only prints out those numbers that are greater than 100.

Guard

The `guard` keyword was first introduced in Swift version 2. According to Apple's documentation, a guard is described like this:

> A guard statement, like an if statement, executes statements depending on the Boolean value of an expression. You use a guard statement to require that a condition must be true in order for the code after the guard statement to be executed.

Before I further explain the guard statement, let's go straight to the following example:

```
struct Article {
     var title: String?
     var description: String?
     var author: String?
     var totalWords: Int?
}

func printInfo(article: Article) {
    if let totalWords = article.totalWords, totalWords > 1000 {
        if let title = article.title {
            print("Title: \(title)")
        } else {
            print("Error: Couldn't print the title of the article!")
        }
    } else {
        print("Error: It only works for article with more than 1000 words."
)
    }
}

let sampleArticle = Article(title: "Swift Guide", description: "A beginner
's guide to Swift 2", author: "Simon Ng", totalWords: 1500)
printInfo(article: sampleArticle)
```

In the above code, we create a `printInfo` function to display the title of an article. However, we will only print the information for an article with more than a thousand words. As the variables are optionals, we use `if let` to verify if the optional contains a value or not. If the optional is `nil`, we display an error message. If you execute the code in Playgrounds, it should display the title of the article.

In general, the `if-else` statements follow this pattern:

```
if some conditions are met {
        // do something
        if some conditions are met {
                // do something
        } else {
                // show errors or performs other operations
        }
} else {
       // show errors or performs other operations
}
```

As you may notice, if you have to test more conditions, it will be nested with more conditions. There is nothing wrong with that programmatically. But in terms of readability, your code will get messy if there are a lot of nested conditions.

This is where the `guard` statement comes in. The syntax of `guard` looks like this:

```
guard else {
        // what to do if the condition is not met
}
// continue to perform normal actions
```

If the condition, defined in the `guard` statement is not met, the code inside the `else` branch is executed. On the other hand, if the condition is met, it skips the `else` clause and continues the code execution.

If you rewrite the sample code using `guard`, it is a lot cleaner:

```
func printInfo(article: Article) {
    guard let totalWords = article.totalWords, totalWords > 1000 else {
        print("Error: It only works for article with more than 1000 words."
)
        return
    }

    guard let title = article.title else {
        print("Error: Couldn't print the title of the article!")
        return
    }

    print("Title: \(title)")
}
```

With `guard`, you focus on handling the condition you don't want. Furthermore, it forces you to handle one case at a time, avoiding nested conditions. Thus, the code is cleaner and easier to read.

Error Handling

When developing an app or any programs, you'll need to handle every possible scenario, whether it's good or bad. Obviously, things may go wrong. Say, if you're developing an app that connects to a cloud server, your app has to deal with situations where the Internet connection is unavailable or the cloud server is failed to connect.

In the older version of Swift, it lacks a proper error handling model. As an example, you handle error conditions like this:

```
let request = NSURLRequest(URL: NSURL(string: "http://www.apple.com")!)
var response: NSURLResponse?
var error: NSError?
let data = NSURLConnection.sendSynchronousRequest(request, returningRespon
se: &response, error: &error)

if error == nil {
        print(response)
        // Parse the data
} else {
       // Handle error
}
```

When calling a method that may cause a failure, you normally pass it with an `NSError` object (as a pointer). If there is an error, the object will be assigned with the corresponding error. You then check if the error object is nil or not and respond to the error accordingly.

That's how you handle errors in the early version of Swift (i.e. Swift 1.2).

Note: NSURLConnection.sendSynchronousRequest() has been deprecated in iOS 9. As most readers are familiar with the usage, it is used in the example.

try / throw / catch

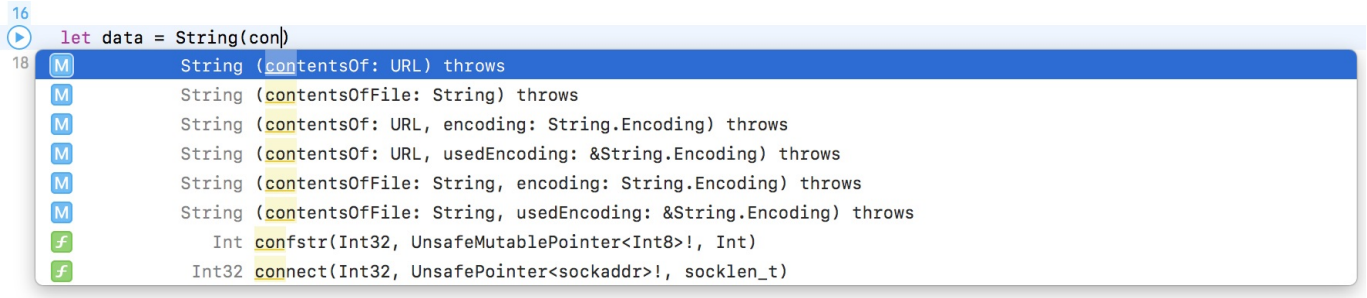
Beginning from Swift 2, it comes with an exception-like model using `try-throw-catch` keywords. The same code snippet will become this:

```
let request = URLRequest(url: URL(string: "https://www.apple.com")!)
var response:URLResponse?
do {
    let data = try NSURLConnection.sendSynchronousRequest(request, returni
ng: &response)
    print(response)

    // Parse the data
} catch {
    // handle error
    print(error)
}
```

Now you use a `do-catch` statement to catch errors and handle them accordingly. As you may notice, we put a `try` keyword in front of the method call. With the introduction of the error handling model, some methods can throw errors to indicate failures. When we invoke a throwing method, you will need to put a `try` keyword in front of it.

How do you know if a method throws an error? As you type the method in the built-in editor, the throwing methods are indicated by the `throws` keyword.



Now that you should understand how to call a throwing method and catch the errors, how do you indicate a method or function that can throw an error?

Imagine you're modelling a lite shopping cart. Customers can use the cart to temporarily store and check out their items, but this cart will throw an error for the following conditions:

- The shopping cart can only store a maximum of 5 items. Otherwise, it throws a `cartIsFull` error.
- There must be at least one item in the shopping cart during checkout. Otherwise, it throws a `cartIsEmpty` error.

In Swift, errors are represented by values of types conforming to the `Error` protocol.

Usually, you use an enumeration to model the error conditions. In this case, you can create an enumeration that adopts `Error` like this for the shopping cart errors:

```
enum ShoppingCartError: Error {
    case cartIsFull
    case emptyCart
}
```

For the shopping cart, we create a `LiteShoppingCart` class to model its functions. Here is a sample code snippet:

```
struct Item {
    var price:Double
    var name:String
}

class LiteShoppingCart {
    var items:[Item] = []

    func addItem(item: Item) throws {
        guard items.count < 5 else {
            throw ShoppingCartError.cartIsFull
        }

        items.append(item)
    }

    func checkout() throws {
        guard items.count > 0 else {
            throw ShoppingCartError.emptyCart
        }
        // Continue with the checkout
    }
}
```

If you take a closer look at the `addItem` method, you probably notice the `throws` keyword. We append the `throws` keyword in the method declaration to indicate that the method can throw an error. In the implementation, we use `guard` to ensure the total number of items is less than 5. Otherwise, we throw the `ShoppingCartError.cartIsFull` error.

To throw an error, you just write the `throw` keyword, followed by the actual error. For the checkout method, we have a similar implementation. If the cart does not contain any items, we throw the `ShoppingCartError.emptyCart` error.

Now, let's see what happens when performing a checkout on an empty cart. I recommend you to fire up Xcode and use Playgrounds to test out the code.

```
let shoppingCart = LiteShoppingCart()
do {
    try shoppingCart.checkout()
    print("Successfully checked out the items!")
} catch ShoppingCartError.cartIsFull {
    print("Couldn't add new items because the cart is full")
} catch ShoppingCartError.emptyCart {
    print("The shopping cart is empty!")
} catch {
    print(error)
}
```

As the `checkout` method can throw an error, we use the `do-catch` statement to catch the errors. If you execute the above code in Playgrounds, it will catch the `ShoppingCartError.emptyCart` error and print the error message accordingly, because we haven't added any items.

Now insert the following code in the `do` clause, right before calling the `checkout` method:

```
try shoppingCart.addItem(item: Item(price: 100.0, name: "Product #1"))
try shoppingCart.addItem(item: Item(price: 100.0, name: "Product #2"))
try shoppingCart.addItem(item: Item(price: 100.0, name: "Product #3"))
try shoppingCart.addItem(item: Item(price: 100.0, name: "Product #4"))
try shoppingCart.addItem(item: Item(price: 100.0, name: "Product #5"))
try shoppingCart.addItem(item: Item(price: 100.0, name: "Product #6"))
```

Here we try to add a total of 6 items to the `shoppingCart` object. Again, it will throw an error because the shopping cart cannot hold more than 5 items.

When catching errors, you can identify the exact error (e.g. `ShoppingCartError.cartIsFull`) to match, so you can provide very specific error handling.

If you do not specify a pattern in the `catch` clause, Swift will match any error and automatically bind the error to the error constant. As a best practice, you should try to catch the specific errors that are thrown by the throwing method. At the same time, you should write a `catch` clause that matches any errors. This ensures all possible errors are handled.

Availability Checking

If all users are forced to upgrade to the latest version of iOS, this would make our developers' life much easier. Apple has tried hard to promote users to upgrade their iOS devices. However, there are still some users who are reluctant to upgrade their devices. Thus, in order to reach more users, our apps have to cater for different versions of iOS (e.g. iOS 12, iOS 13 ,and iOS 14).

If you just use the latest version of APIs in your app, this may cause errors when the app runs on older versions of iOS. When using an API that is only available on the latest version of iOS, you will need to do some kinds of verification before using the class or calling the method.

Say, the property `safeAreaLayoutGuide` of `UIView` is only available on iOS 12 (or later). If you use the property on older versions of iOS, you'll end up with an error and that probably causes an app crash.

Swift has built-in support for checking API availability. You can easily define an availability condition so that the block of code will only be executed on certain iOS versions. Here is an example:

```
if #available(iOS 12.0, *) {
    // iOS 12 or up
    let view = UIView()
    let layoutGuide = view.safeAreaLayoutGuide
} else {
    // Fallback on earlier versions
}
```

You use the `#available` keyword in the `if` statement. In the availability condition, you specify the OS versions (e.g. iOS 14) you want to verify. The `*` is required and indicates that the `if` clause is executed on the minimum deployment target and any other versions of OS. For the above example, the body of the `if` will be executed on iOS 12 or up, and other platforms such as watchOS.

Similarly, you can use `guard` instead of `if` for checking API availability. Here is another example:

```
guard #available(iOS 12.0, *) else {
    // what to do if it doesn't meet the minimum OS requirement
    return
}

let view = UIView()
let layoutGuide = view.safeAreaLayoutGuide
```

What if you want to develop a class or method, which is available to certain versions of OS? Swift lets you apply the `@available` attribute on classes/methods/functions to specify the platform and OS version you want to target. Say, for example, you're developing a class named `SuperFancy` and it is only available for iOS 12 or later. You can apply `@available` like this:

```
@available(iOS 12.0, *)
class SuperFancy {
    // implementation
}
```

If you try to use the class on an Xcode project that supports multiple versions of iOS, Xcode will show you the errors below:



Note: You cannot test availability check in Playgrounds. If you want to give it a try, create a new Xcode project to test out the feature.