

<packt>

SwiftUI

For Beginners

Build your first iOS app with
SwiftUI, Swift Playground, and iPad



Ahmad Sahar

SwiftUI for Beginners

Build your first iOS app with SwiftUI, Swift Playground,
and iPad

Ahmad Sahar



SwiftUI for Beginners

Copyright © 2026 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Portfolio Director: Ashwin Nair

Relationship Lead: Sohini Ghosh

Project Manager: Vishnu Priya R

Content Engineer: Nithya Sadanandan

Technical Editor: Rohit Singh

Copy Editor: Nithya Sadanandan

Indexer: Manju Arasan

Proofreader: Nithya Sadanandan

Creative Designer: Ajay Patule

Growth Lead: Sayantani Saha

First published: September 2026

Production reference: 1080926

Published by Packt Publishing Ltd.

Grosvenor House

11 St Paul's Square

Birmingham

B3 1RB, UK.

ISBN 978-1-80778-557-4

www.packtpub.com

*To my mother, Sharifah; my father, Sahar; my two sisters, Ainol Shareha and Ainol Shahrina;
and my beloved wife, Oni – thank you for your love and support.*

– Ahmad Sahar

Contributors

About the author

Ahmad Sahar is a trainer, presenter, and consultant at Tomafuwi Productions. He is an Apple Certified Trainer for app development in Swift and has conducted training courses for over 10 years on iOS mobile application development, Flutter mobile application development, and the use of Apple software and hardware. He also currently holds the App Development with Swift Associate and Certified User certifications. He is a member of the DevCon iOS online community in Malaysia and has given presentations and talks to the group. In his spare time, he enjoys building and programming LEGO MINDSTORMS® robots.

About the reviewers

Tim Oliver is a principal iOS engineer at Third Intelligence and an independent developer based in Tokyo, Japan. He has been building for Apple platforms for over 15 years, shipping apps to the App Store since the earliest days of the iPhone SDK, with previous stints at Instagram and Realm. Tim is also active in the iOS developer community, presenting at Apple developer conferences globally. He co-runs Tokyo iOS, a local meetup that helps bridge the gap between English- and Japanese-speaking developers. Outside of work, Tim has published over 50 open source libraries on GitHub. His image-editing library, TOCropViewController, has been adopted by 100,000+ apps. When he isn't writing code, he enjoys singing karaoke and traveling around Japan.

My deepest thanks to my parents, Ronald and Helen Oliver, the two biggest champions in my life, whose love and support have gotten me this far. And a huge shoutout to my friends and colleagues in Japan: to Mike Omoto, for his engineering mentorship and guidance; to Jetha Chan, for talking me up when I wouldn't; and to Dean Herbert, for always holding me to the highest technical standards.

Manvindu Pathak is an iOS software engineer with experience building and scaling consumer-facing applications at product-based technology companies. He specializes in Swift, SwiftUI, UIKit, and modern mobile application architecture, with a strong focus on performance, usability, and developer experience. Passionate about Apple platform technologies, software design, and emerging AI-driven experiences, he enjoys solving complex engineering challenges and creating intuitive mobile products. Beyond development, he actively shares technical insights through writing and mentoring and engages in continuous learning within the iOS development community.

Artem Mirzabekian holds a Doctor of Medicine degree and transitioned from medicine to software engineering. His career in iOS spans more than a decade, with a focus on architecture, performance, and the evolution of complex mobile applications. He also writes about software development, shares practical insights through his articles, and creates open-source projects focused on iOS and AI-assisted engineering.

Table of Contents

Preface	xi
Free benefits with your book.....	xvi
Part 1: Design	1
Chapter 1: Hello, Swift Playground	3
Technical requirements.....	4
Download and install Swift Playground from the App Store	4
Explore the Swift Playground User Interface	7
Preview your app in App Preview	8
Run your app on iPad	12
Summary	14
Chapter 2: Set Up the User Interface	15
Technical requirements.....	16
Useful SwiftUI terms	16
A tour of the CatLog app.....	22
The Cats screen • 22	
The Add New Cat screen • 23	
The Cat Detail screen • 24	
The Map screen • 24	
Modify your app playground	26
Set up a tab bar • 26	
Embed views in navigation stacks • 27	
Configure your app playground.....	29
Summary	32
Chapter 3: Build the User Interface	33
Technical requirements.....	34

Create a custom view	34
Modify the tab view	39
Present a view modally	41
Add a button to the navigation bar • 41	
Add a new view • 42	
Add Cancel and Save buttons to the navigation bar • 46	
Summary	49
Chapter 4: Complete the User Interface	51
Technical requirements.....	51
Implement the Cat Detail screen.....	52
Implement the Map screen	54
Summary	56
Chapter 5: Modify App Screens	57
Technical requirements.....	58
Modify the Cats screen	58
Modify the Add New Cat screen.....	63
Modify the Cat Detail screen	68
Summary	73
Part 2: Code	75
Chapter 6: Display Data in a List	77
Technical requirements.....	77
State variables	78
Create a class to represent a cat	81
Create sample data.....	83
Display data in a view	84
Summary	88
Chapter 7: Pass Data Between Screens	89
Technical requirements.....	89
Pass data from the Add New Cat screen to the Cats screen	90

Remove a cat from the Cats screen	95
Pass data from the Cats screen to the Cat Detail screen	97
Summary	100
Chapter 8: Core Location and MapKit	101
Technical requirements.....	101
Get your device location using the Core Location framework	102
Modify the AddNewCatView view • 103	
Display Annotations on the Map Screen.....	110
Display a map snapshot on the Cat Detail screen	119
Summary	123
Chapter 9: SwiftData	125
Technical requirements.....	126
About SwiftData.....	126
Modify the Cat class to work with SwiftData	127
Create the Data Container	132
Modify Your App Playground to Use the Data Container	134
Summary	142
Chapter 10: Custom Views	143
Technical requirements.....	143
Create a Custom View	144
Add your custom view to the Add New Cat screen.....	148
Add your custom view to the Cat Detail screen	149
Summary	150
Chapter 11: Photo Library	151
Technical requirements.....	151
Implement a new PhotosPicker view	152
Summary	158
Chapter 12: Search	159
Technical requirements.....	159

Implement a search bar for the Cats screen	160
Implement a navigation split view	162
Summary	164
Part 3: App Store	165

Chapter 13: Test Your App with TestFlight	167
--	------------

Technical requirements.....	167
Get an Apple Developer account.....	168
Submit Your App to App Store Connect	170
Test your app with TestFlight	182
Test your app internally • 182	
Test your app externally • 191	
Summary	197

Chapter 14: Submit Your App to the App Store	199
---	------------

Technical requirements.....	199
Get screenshots of your app	200
Submit Your App for App Review	206
Summary	220

Chapter 15: Unlock the Code Bundle and the PDF Version	221
---	------------

Unlock this book's free benefits in three easy steps	222
--	-----

Other Books You May Enjoy	227
----------------------------------	------------

Index	231
--------------	------------

Preface

Welcome to *SwiftUI for Beginners*. This book is a guide to building applications for all Apple platforms using the SwiftUI framework.

In this book, you will build an app called *CatLog*. You will start by exploring Swift Playground, a fun and engaging way to learn how to code and build apps. Once you are familiar with Swift Playground, you will start creating the user interface for the CatLog app. During this process, you will create your app's screens and connect them using SwiftUI.

With your user interface complete, you will then add code to implement your app's functionality. To start, you'll learn how to display data using a list view. Next, you'll learn how to add data to your app and pass data between views. After that, you'll learn how to determine your device's location and display annotations on a map. You'll then learn how to persist app data using SwiftData, create custom views, and add photos from the photo library. Finally, you'll make your app work on devices with larger screens, such as an iPad or Mac, by implementing a navigation split view.

You'll also learn how to test your app with internal and external testers and get it into the App Store.

Who this book is for

This book is tailored for individuals with minimal coding experience who are new to Swift and iOS app development. A basic understanding of programming concepts is recommended.

What this book covers

Chapter 1, Hello, Swift Playground, takes you on a tour of Swift Playground and shows you how to create your first app.

Chapter 2, Set Up the User Interface, introduces commonly used SwiftUI terms, takes you on a tour of the *CatLog* app, and sets up the initial screen that users will see.

Chapter 3, Build the User Interface, covers creating a screen that shows a list of cats and a screen that lets you add a new cat to the app.

Chapter 4, Complete the User Interface, covers setting up the remaining screens for the app.

Chapter 5, Modify App Screens, covers adding user interface elements to each screen of the app.

Chapter 6, Display Data in a List, covers how to display app data on the screen in a list.

Chapter 7, Pass Data Between Screens, teaches you how to add a new cat to the app and display details for a cat in the list.

Chapter 8, Core Location and MapKit, covers working with Core Location and MapKit to determine your device's location and add annotations to a map.

Chapter 9, SwiftData, teaches you how to store and retrieve user data using the SwiftData framework.

Chapter 10, Custom Views, teaches you how to create and use a custom view that displays a star rating.

Chapter 11, Photo Library, covers how to add photos from your photo library to your app.

Chapter 12, Search, teaches you how to implement a search bar for your app and modify your app to work better on larger screens.

Chapter 13, Test Your App with TestFlight, covers how to submit your app to App Store Connect and test it using TestFlight.

Chapter 14, Submit Your App to the App Store, covers how to capture screenshots for your app and submit it to the App Store.

To get the most out of this book

To complete all the exercises in this book, you will need:

- An iPad running iPadOS 26 or later
- Swift Playground 4.7 or later

To check whether your iPad supports iPadOS 26, see this link: <https://www.apple.com/my/os/ipados/>. If your iPad is supported, you can update iPadOS using software update in the **Settings** app.

To get the latest version of Swift Playground, you can download it from the App Store. Most of the exercises can be completed without an Apple Developer account and use the iPad to run your app. If you wish to test the app you are developing on other Apple devices, you will need a paid Apple Developer account.

The following chapters require a paid Apple Developer account:

- *Chapter 13, Test Your App with TestFlight*
- *Chapter 14, Submit Your App to the App Store*

Instructions on how to get a paid Apple Developer account are included in *Chapter 13, Test Your App with TestFlight*.

Download the example code files

This book includes a complete downloadable code bundle containing all the example projects and files used throughout the chapters. We recommend downloading the bundle so you can follow along smoothly and experiment with the examples.

Use the bundle as a practical starting point. Modify it, extend it, and apply what you learn by creating your own variations as you progress through the chapters.

Get the code bundle

If you bought the book directly from Packt:

1. Go to **packtpub.com**
2. Click your **profile picture** and select **Your Orders**
3. Find this book and click **Download Code**

If you bought this book from Amazon or any other channel partner:

1. Go to packtpub.com/unlock or scan the following QR code:



2. Search for this book
3. Sign up or log in to your free Packt account
4. Upload your proof of purchase and download the code bundle locally

Usage note: You're free to use and modify this code for personal learning and non-commercial projects.

Code in Action

Visit the following link to check out videos of the code being run:

<https://packt.link/dFB2u>

Download the color images

Your purchase includes a color, DRM-free PDF copy of this book, ideal for viewing color images, screenshots, and diagrams. Refer to *Free benefits with your book section* at the end of the *Preface* to unlock your PDF copy.

Conventions used

There are several text conventions used throughout this book.

CodeInText: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "So, this is a very simple function, named `serviceCharge()`."

A block of code is set as follows:

```
class ClassName {  
    property1  
    property2  
    property3  
    method1() {  
        code  
    }  
    method2() {  
        code  
    }  
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
let cat = Animal()  
cat.name = "Cat"  
cat.sound = "Mew"  
cat.numberOfLegs = 4  
cat.breathesOxygen = true  
print(cat.name)
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Launch **Swift Playground** and tap **New Playground...**"

Note

Important notes appear like this.

Tip

Tips appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at customer care@packtpub.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/support/errata, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packt.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Free benefits with your book

This book includes free benefits designed to support your learning and help you apply what you learn effectively. Activate them now for instant access (see the *How to unlock* section for instructions).

Here's a quick overview of what you can instantly unlock with your purchase:



Complete Code Bundle

Download the full source code for this book's examples and projects.



DRM-Free PDF Version

Download DRM-free PDF and ePub copies of this book.



7-Day Packt Library Access

Get 7-day unlimited access to 8,000+ books and videos. No credit card required.

Available for first-time Packt+ trial users only.



Next-Gen Reader Access

Read this book on Packt Reader with progress sync, dark mode and note-taking.

How to unlock

Scan the QR code (or go to packtpub.com/unlock). Search for this book by name, confirm the edition, and then follow the steps on the page.



UNLOCK NOW

Note: Have your invoice handy. Purchases made directly from the Packt website don't require an invoice.

Share your thoughts

Once you've read *SwiftUI for Beginners*, we'd love to hear your thoughts! Scan the QR code below to go straight to the Amazon review page for this book and share your feedback.



<https://packt.link/r/1807785572>

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Part 1

Design

Welcome to *Part 1* of this book. In this part, you will begin by exploring Swift Playground, a fun and engaging way to learn how to write code and create apps. Next, you'll be introduced to the app that you will create, named *CatLog*. After that, you will use SwiftUI, Apple's framework for creating user interfaces for all Apple devices, to build the screens that your app will use, add elements such as buttons, text, and images to them, and navigate between them. As you will see, you can do this with relatively few lines of code.

This part comprises the following chapters:

- *Chapter 1, Hello, Swift Playground*
- *Chapter 2, Set up the User Interface*
- *Chapter 3, Build the User Interface*
- *Chapter 4, Complete the User Interface*
- *Chapter 5, Modify App Screens*

By the end of this part, you'll be able to navigate the various screens of your app on your iPad and you will know how to prototype the user interface of your own apps. Let's get started!

1

Hello, Swift Playground

Welcome to *SwiftUI for Beginners*. I hope this book serves as a useful introduction to creating and publishing SwiftUI apps on the App Store. Throughout the course of this book, you will build a simple app named *CatLog*, which lets you document cats that live in your area.

In this chapter, you'll download and install **Swift Playground** on your iPad. Then, you'll explore the Swift Playground user interface. After that, you'll create your first **SwiftUI** app and see what it looks like in **App Preview**. Finally, you'll run your app on your iPad. This app will be the starting point of the *CatLog* app.

By the end of this chapter, you will know how to create a SwiftUI app, how to preview it in App Preview, and how to run it on your iPad.

The following topics will be covered in this chapter:

- Download and install Swift Playground from the App Store
- Explore the Swift Playground user interface
- Preview your app in App Preview
- Run your app on your iPad

Note

Download the code bundle and the PDF version of this book

Your purchase includes a DRM-free PDF copy of this book, the code bundle, and a range of exclusive benefits. To unlock everything, follow the *Free benefits with your book* section in the *Preface*.

Technical requirements

To do the exercises for this chapter, you will need the following:

- An iPad running iPadOS 26 or iPadOS 27
- An Apple account (if you don't have one, you will create one in this chapter)

A Bluetooth keyboard and mouse are optional, but they do make typing in code much easier.

The Swift Playground project for this chapter is in the `Chapter01.zip` file of the code bundle for this book. To access the repository link, follow the steps in the *Download the example code files* section in the *Preface*.

Check out the following video to see the code in action:

<https://packt.link/rxDV1>

You'll start by downloading Swift Playground, an app which helps you learn how to code in an interactive environment, in the next section.

Download and install Swift Playground from the App Store

Swift Playground is an app for iPad and Mac that helps you learn how to code. It includes interactive lessons that teach coding in a fun and engaging way, and it also allows you to create your own apps, which you can then upload to the App Store.

Note

To learn more about Swift Playground, see this link: <https://developer.apple.com/swift-playground/>.

To watch a video demonstration of Swift Playground from WWDC 2022, see this link: <https://developer.apple.com/videos/play/wwdc2022/110348/>.

You will build apps using Apple's SwiftUI framework, a simple and innovative way to build user interfaces for all Apple platforms with a minimum of code. It uses a declarative syntax, which means you can simply state what your user interface should do, instead of having to specify in detail how it should work. When you type in SwiftUI code in Swift Playground, you can immediately preview the results, and you can also run your apps on your iPad itself.

Note

To learn more about SwiftUI, see this link:

<https://developer.apple.com/swiftui/>.

To watch a video explaining the latest updates to SwiftUI from WWDC 2026, see this link:

<https://developer.apple.com/videos/play/wwdc2026/269/>.

You'll need to download and install Swift Playground on your iPad prior to writing your first app. Follow these steps:

1. On your iPad, tap the **App Store** icon.
2. Tap the Search (magnifying glass) icon, type `swift playground` and tap the *Return* key.
3. You'll see **Swift Playground** in the search results. Tap **Get**.
4. Type your Apple Account email or phone number in the text field and enter your password when prompted.

Note

You can see more information on how to create an Apple account using this link: <https://support.apple.com/en-us/108647#appstore>.

5. Once Swift Playground has been installed, tap **Open**. Swift Playground will launch, and you'll see a welcome screen. Tap the **New Playground...** button:

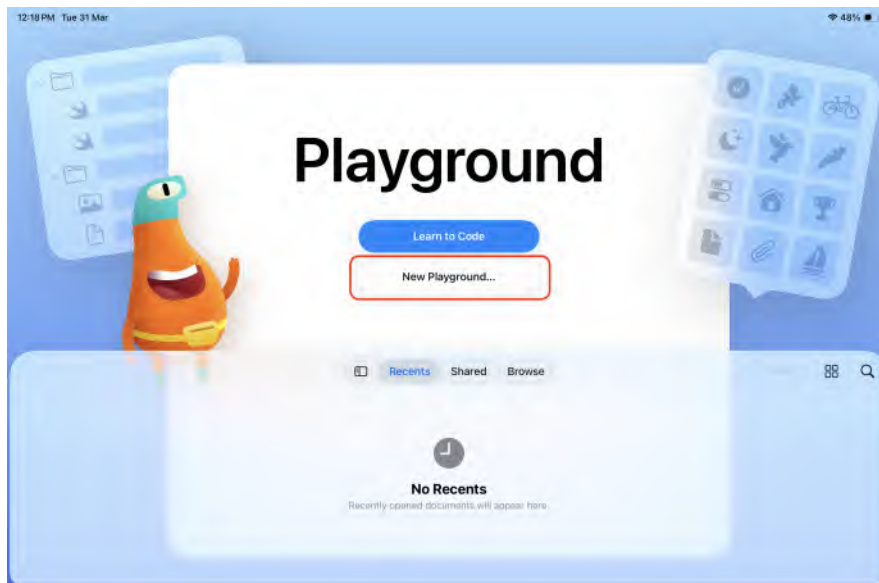


Figure 1.1: Welcome screen

6. You'll see the **New Playground...** dialog box. Tap the **App** button to create a new app playground:

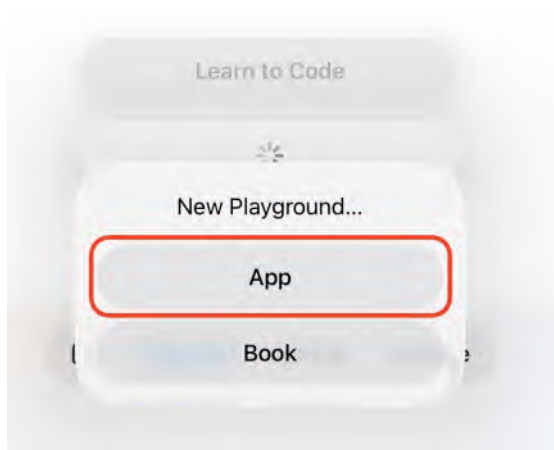


Figure 1.2: New Playground... dialog box

7. Your new app playground will appear.

Fantastic! You have now successfully downloaded and installed Swift Playground and created your first app playground. In the next section, you will learn about the Swift Playground user interface.

Explore the Swift Playground User Interface

You've just created your first Swift Playground app playground! As you can see, the app playground user interface is divided into several distinct parts, as shown here:

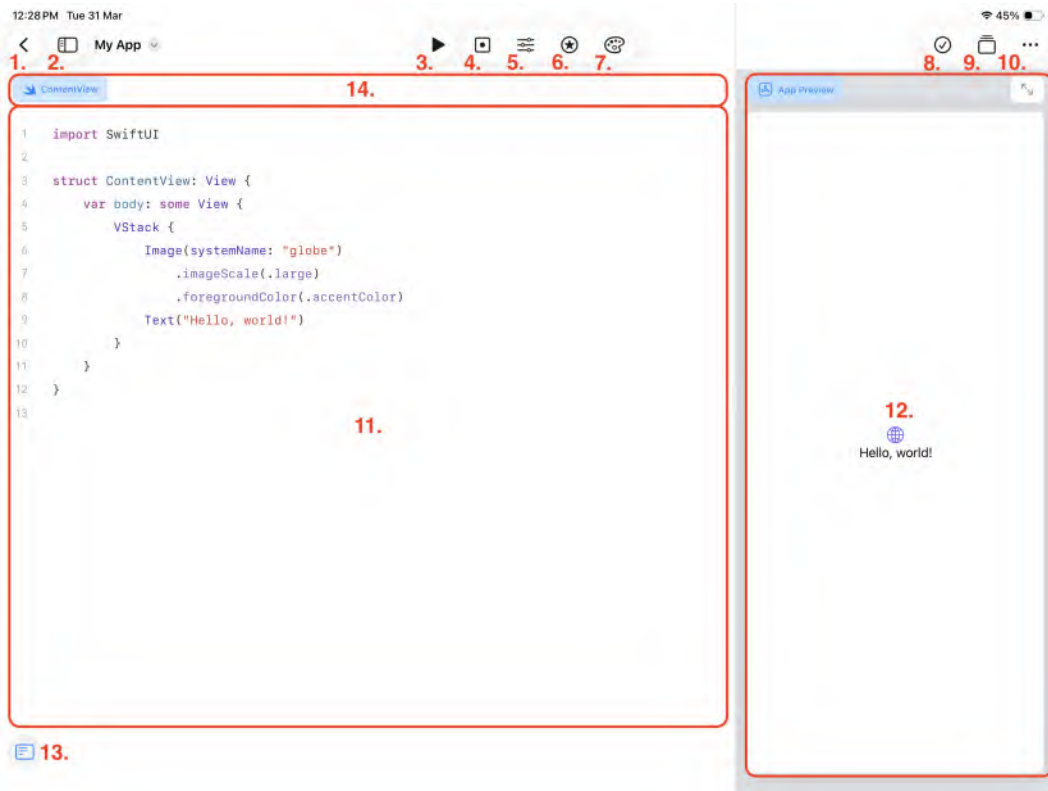


Figure 1.3: App playground user interface

Let's look at each part in more detail. The following descriptions correspond to the numbers shown in the preceding screenshot:

- **Back button (1):** Returns you to the welcome screen
- **Sidebar button (2):** Displays assets contained in your app playground, such as Swift files or images
- **Run App button (3):** Runs the app playground on your iPad in its own window

- **Views button (4):** Displays the Views library, which allows you to add a view or control to your app playground
- **Modifiers button (5):** Displays the Modifiers library, which allows you to add a control, an effect, a layout, text, an image, a list and more to your app playground
- **Symbols button (6):** Displays the Symbols library, which allows you to add a symbol to your app playground
- **Colors button (7):** Displays the Colors library, which allows you to add a color to your app playground
- **Issues button (8):** Displays warnings or errors in your app playground
- **App Preview button (9):** Hides or shows App Preview in your app playground
- **More button (10):** Allows you to change text size in the editor area, as well as accessing advanced settings for your app playground
- **Editor area (11):** Allows you to edit the code contained in the Swift files for your app playground
- **App Preview (12):** Displays the result of the code in the Editor area
- **Console button (13):** Shows or hides the Console area
- **Tab bar (14):** Allows you to switch between different Swift files in your project

Don't be overwhelmed by all the different parts, as you'll learn about them in more detail in the upcoming chapters. Now that you are familiar with the Swift Playground interface, you will make some changes in the Editor area and see the results in App Preview.

Preview your app in App Preview

App Preview automatically displays the results of the code that you type in the Editor area. You'll make some changes to your code and observe the results in App Preview. Follow these steps:

1. You may find the size of the text in the Editor area to be too small. Tap the **More** button and tap the right-side **A** button to make the text larger:

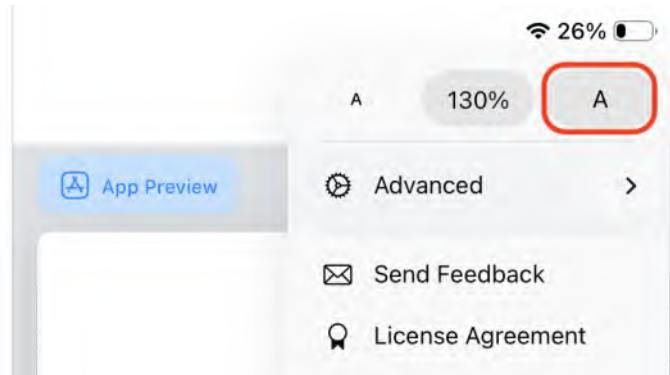


Figure 1.4: More menu showing A button

2. In the Editor area, select all the code between `VStack {` and `}` and tap *Delete* to remove it.



Figure 1.5: Editor area showing code to be removed

3. Swipe left with three fingers on the screen to undo what you just did. The text you removed should reappear.
4. Swipe right with three fingers on the screen to redo what you just did. The text will disappear again.
5. Tap twice on the screen with three fingers to undo your last action once more. The text will reappear.
6. Place the text cursor to the right of the `Image(systemName: "globe")` statement.

Tip

You can drag with two fingers on the on-screen keyboard to position the text cursor. You can also pair a Bluetooth keyboard and mouse with your iPad to make it easier to edit code.

7. Tap the Symbols button to display the Symbols library. Type Cat in the filter field, and tap `cat.fill` in the search results. This will add code to display a cat symbol to your app:

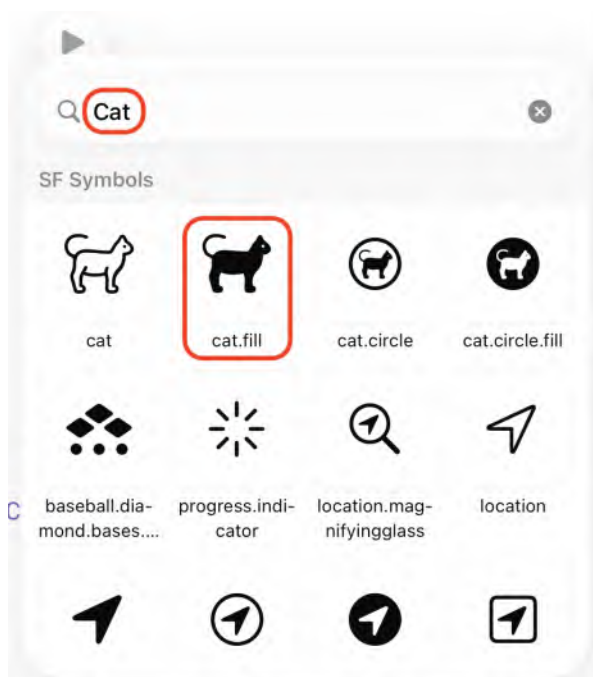


Figure 1.6: Symbols library showing the "cat.fill" symbol

8. An error icon will appear. Tapping it will display the message **Consecutive statements on a line must be separated by newline or ';' :**

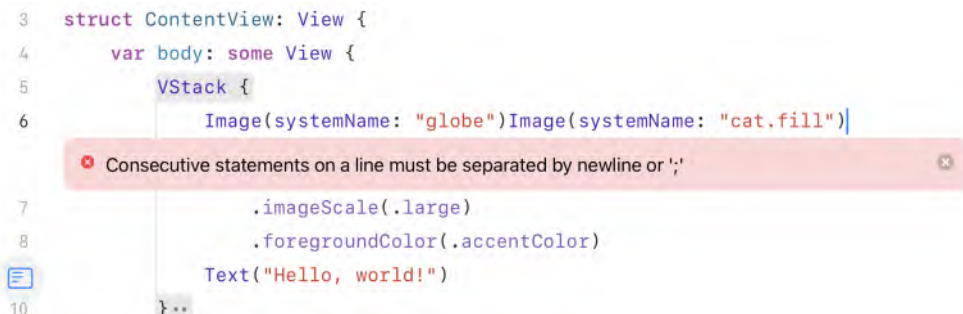


Figure 1.7: Editor area showing error message

9. To fix this error, place the text cursor to the right of the `Image(systemName: "globe")` statement and tap *Return*. The error will disappear and App Preview will display a cat symbol after the globe symbol:



Figure 1.8: App Preview displaying a cat symbol after the globe symbol

10. Remove the `Image(systemName: "globe")` statement, and change the "Hello, world!" text in the next statement to "CatLog". The App Preview updates after a few seconds to display the results:

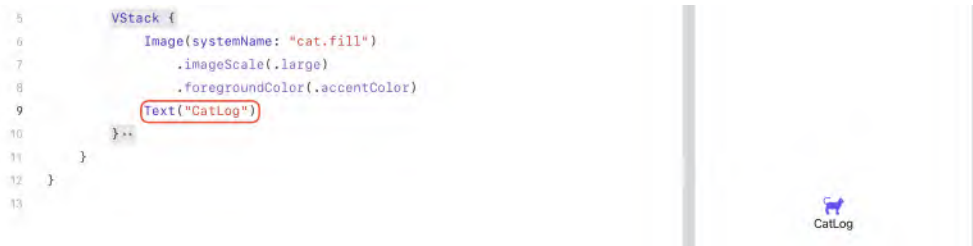


Figure 1.9: App Preview showing updated results

You have just created and previewed your SwiftUI app on App Preview! Great job! In the next section, you'll learn how to run your app in a new window on your iPad.

Run your app on iPad

Although App Preview already shows you a preview of your app, you may wish to run your app in its own window without showing the Swift Playground interface. This will give you a better idea of what your app looks like.

Note

You will learn how to run your app on other Apple devices in *Chapter 13, Test Your App with TestFlight*, and *Chapter 14, Submit Your App to the App Store*, of this book.

Follow these steps:

1. In your app playground, tap the Run button:



Figure 1.10: App playground showing Run App button

This will launch the app you created in its own window.

2. To return to Swift Playground, first tap the orange button in the upper right corner of the screen:

3:18 PM Tue 31 Mar



Figure 1.11: Orange button in the upper right corner

3. A dialog box will appear. Tap **Stop** to return to Swift Playground:

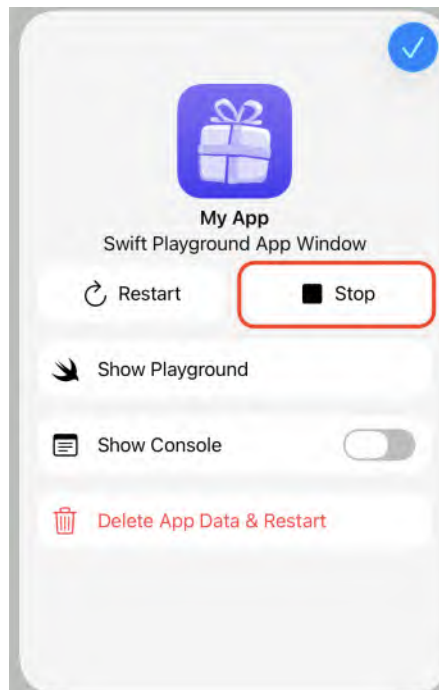


Figure 1.12: Dialog box showing Stop button

Congratulations! You have successfully run your app on your iPad!

Summary

In this chapter, you learned how to download and install Swift Playground on your iPad. Then, you familiarized yourself with the different parts of the Swift Playground user interface. After that, you created your first SwiftUI app and previewed it using App Preview. Finally, you ran your app in its own window on your iPad.

You now know how to create a SwiftUI app, how to preview it in App Preview, and how to run it on your iPad. Good job!

In the next chapter, you will learn more about the app that you will build and start creating the user interface for it.

2

Set Up the User Interface

In the previous chapter, you created a new app playground in Swift Playground, saw the result of the code in App Preview, and ran your app in a window on your iPad. At present, the app just displays a single screen containing a symbol and a line of text, and you can't interact with it. In this chapter, you will start building the user interface of an app, *CatLog*, that you will be working with throughout this book.

You'll start this chapter by learning useful SwiftUI terms that are used throughout this book. Next, you will tour the screens used in the *CatLog* app and learn how a user would use the app. After that, you will begin recreating the app's UI with SwiftUI, starting with the tab bar, which allows the user to select between the **Cats** and **Map** screens. Then, you'll add navigation bars to the top of both screens and configure the tab bar buttons. Finally, you'll change the name, accent color, and icon for your app playground.

By the end of this chapter, you'll have learned some common SwiftUI terms, what the flow of your app will be like, and how to use SwiftUI to add and configure UI elements.

The following topics will be covered in this chapter:

- Useful SwiftUI terms
- A tour of the *CatLog* app
- Modify your app playground
- Configure your app playground name and icon

Technical requirements

You will modify the My App app playground that you created in *Chapter 1, Hello, Swift Playground*.

The completed app playground for this chapter is in the `Chapter02.zip` file of the code bundle for this book. To access the repository link, follow the steps in the *Download the example code files* section in the *Preface*.

Check out the following video to see the code in action:

<https://packt.link/Nej9q>

Before you get started with the project, you'll learn some common SwiftUI terms used in app development.

Useful SwiftUI terms

As you begin your journey into app development using SwiftUI you will encounter special terms and definitions. Here are some of the most used terms and definitions. Just read through them for now. Even though you may not understand everything yet, things will become clearer as you go along:

- **View:** A view is a building block used to build your app's user interface. Some views display data such as text or symbols; others determine the position and size of other views. You can use views that have been defined by Apple such as `Text` or `Image` and also define your own custom views.
- **Stack view:** A stack view is used to group views together. Views can be positioned in a vertical line using `VStack`, a horizontal line using `HStack` and on top of each other using `ZStack`.
- **List:** A list is a scrollable container that presents other views arranged in a single column.

The *Settings* app displays your device settings in a list:

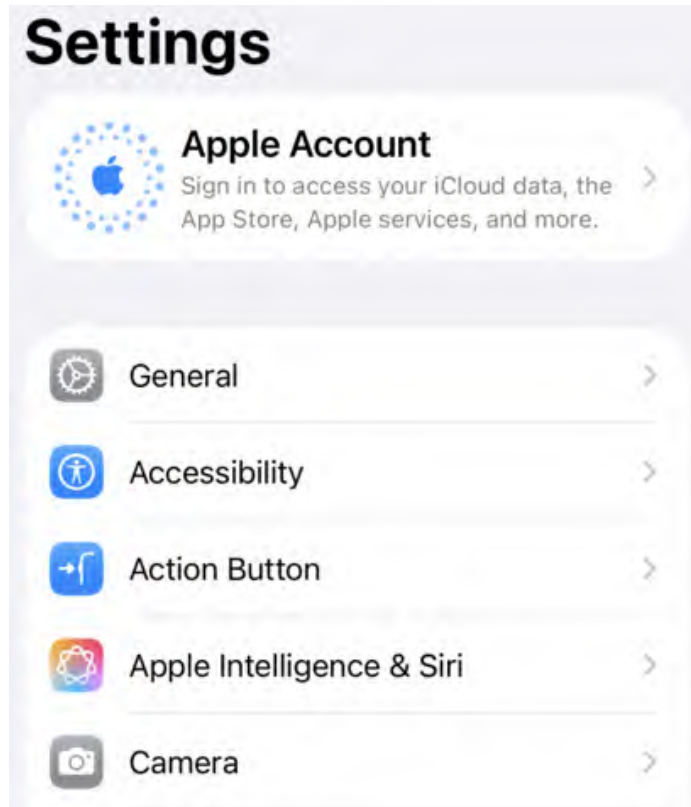


Figure 2.1: Settings window in Settings app

As you can see, all the different settings (**General**, **Accessibility**, **Action Button**, and so on) are views displayed in a single column inside the list.

- **Navigation stack:** A navigation stack is a container that holds a stack of views. The initial view in the stack is known as the root view. You can add more views to the stack, and the view at the top of the stack appears onscreen, along with a navigation bar at the top of the screen.

The list in the *Settings* app is embedded in a navigation stack, and you can see the navigation bar at the top of the screen:

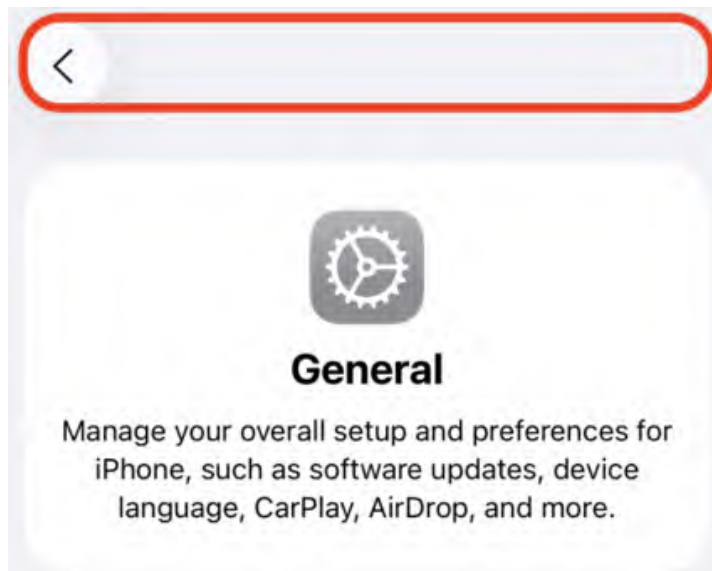


Figure 2.2: Navigation bar in the Settings app

When you tap a setting, the view for that setting is added to the stack. The user sees the view slide in from the right. Note the navigation bar at the top of the screen, which can hold a title and buttons. A < button appears on the top-left side of the navigation bar. Tapping this button returns you to the previous screen, and it removes the view for that setting from the stack.

- **Tab view:** A tab view displays a tab bar with buttons at the bottom of the screen. The button on the extreme left corresponds to a tab in the tab view and will already be selected. When you tap another button, the corresponding tab is loaded, and its view appears on the screen.

The *Fitness* app uses a tab view to navigate to different screens:

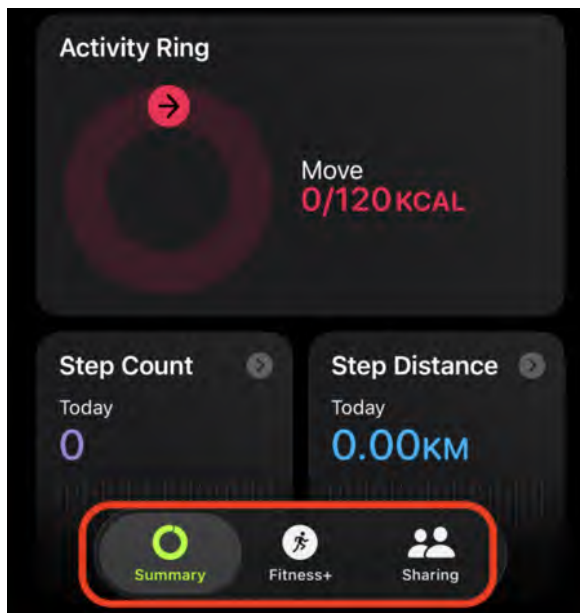


Figure 2.3: Tab bar in the *Fitness* app

As you can see, the different screens for this app (**Summary**, **Fitness+**, and **Sharing**) are accessed by tapping the corresponding tab bar button.

Now that you have had a brief introduction to SwiftUI, let's see how views are used in an app. Open the *My App* app playground that you created in *Chapter 1, Hello, Swift Playground*:

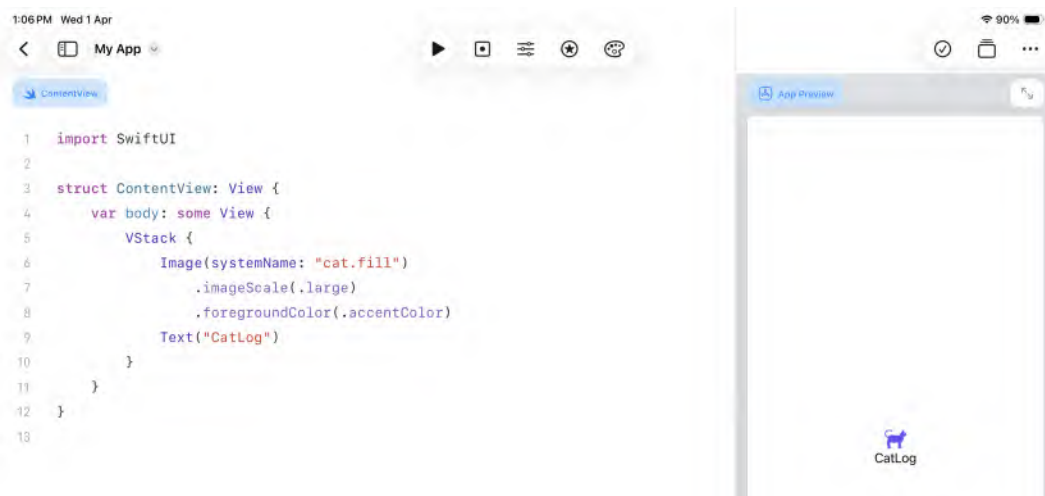


Figure 2.4: My App app playground showing the ContentView file

You'll see the code in the ContentView file on the left side, and the results of that code in **App Preview** on the right side. Read through the code that was automatically provided in the ContentView file. The coding language used here is named **Swift**, which was invented by Apple in 2014.

Note

You can learn more about the Swift language at this link: <https://docs.swift.org/swift-book/documentation/the-swift-programming-language>.

Swift uses many common English words, such as Image and Text. Note that some terms, such as `.imageScale`, start with a lowercase letter, and every subsequent word has an uppercase first letter. This is known as **camel case**. **Parentheses** usually contain information related to the preceding code. **Curly braces** are like containers. They group lines of code together. Note that the code between the curly braces is indented. This reflects the code's organization and makes it more readable. Different parts of your code have different colors based on their purpose; this is known as **syntax highlighting**.

Let's break down the code shown in the ContentView file:

```
import SwiftUI
```

This statement allows you to use the features of Apple's SwiftUI framework.

```
struct ContentView: View {
```

This statement declares a custom view named `ContentView`. This view will be the first view displayed when your app is launched.

```
    var body: some View {
```

This statement declares a container named `body` that determines what the view looks like and what it can do.

```
        VStack {
```

This statement declares a vertical stack view, which arranges the views inside it from top to bottom.

```
            Image(systemName: "cat.fill")
```

This statement displays an image on the screen. The image to be used is specified in parentheses, which in this case is an image of a cat. Note that the text is colored red and enclosed in double quotes. This is called a **string**, since it is a string, or sequence of characters.

```
                .imageScale(.large)  
                .foregroundColor(.accentColor)
```

These statements determine the size and color of the image to be displayed and are known as **modifiers**. The first modifier sets the size of the image in the image view to large. The second modifier sets the color of the image in the image view.

```
            Text("CatLog")
```

This statement displays text on the screen. The text to be used is specified by a string between the parentheses, which in this case is `CatLog`.

Note

For more information on the code provided in the `ContentView` file, you can do the tutorial at this link: <https://developer.apple.com/tutorials/develop-in-swift/hello-swiftui>.

Now that you are familiar with SwiftUI terms, let's take a tour of the app you will build.

A tour of the CatLog app

Let's take a quick tour of the app that you will build. *CatLog* is an app that lets users store a list of cats, with the option of storing a photo or a map location for each cat. Users can also view a map that shows the locations of cats that are close to a user's current location. You'll see all the screens used in the app and its overall flow in the next sections.

The Cats screen

When the app is launched, you will see the **Cats** screen:

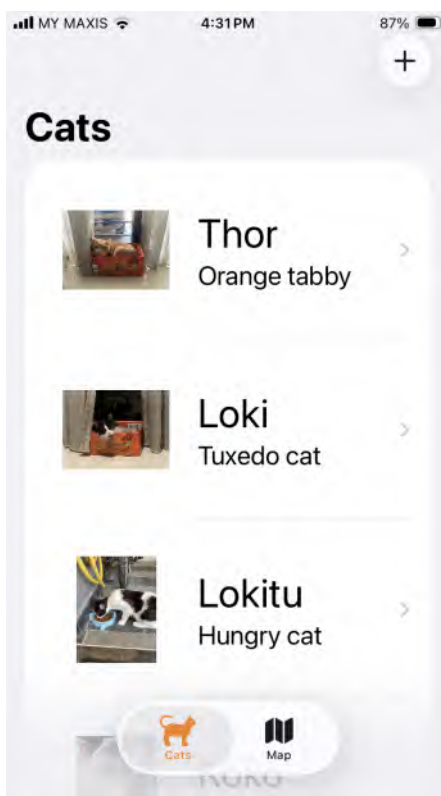


Figure 2.5: Cats screen

Let's study the different parts of this screen.

A tab bar at the bottom of the screen displays the **Cats** and **Map** buttons. The **Cats** button is selected, and you can see a list of cats. A search bar is located at the top of the screen. This allows you to search for a particular cat.

To add a new cat, tap the + button at the top of the screen. This displays the **Add New Cat** screen.

The Add New Cat screen

When you tap the + button at the top of the **Cats** screen, you will see the **Add New Cat** screen:

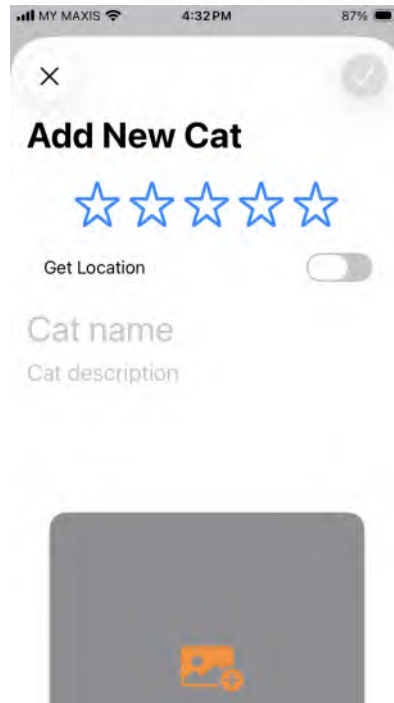


Figure 2.6: Add New Cat screen

Let's study the different parts of this screen.

A navigation bar at the top of the screen contains the Cancel button on the left side and the Save button on the right side. A stack view displays a custom rating view, a switch, a title text field, a body text view, and a placeholder photo. Tapping the rating view allows you to assign 0 to 5 stars for this cat. Turning the switch on will obtain your current location.

You can enter the cat's name and description in the text fields. You can also tap the placeholder photo to choose a picture from your Photo Library. Once you tap Save, you are returned to the **Cats** screen, and the new cat will be visible in the list. You can also tap cancel to return to the **Cats** screen without adding a new cat.

To see the details of a particular cat, tap the cat you want in the list, and then you will see the **Cat Detail** screen.

The Cat Detail screen

Tapping any one of the cats on the **Cats** screen will display the corresponding **Cat Detail** screen:



Figure 2.7: Cat Detail screen

Let's study the different parts of this screen.

A navigation bar at the top of the screen contains a < (Back) button. A list displays the date, rating, title text, body text, photo, and location map for the selected cat.

You can tap the < button to return to the **Cats** screen.

The Map screen

Tapping the **Map** button in the tab bar displays the **Map** screen:

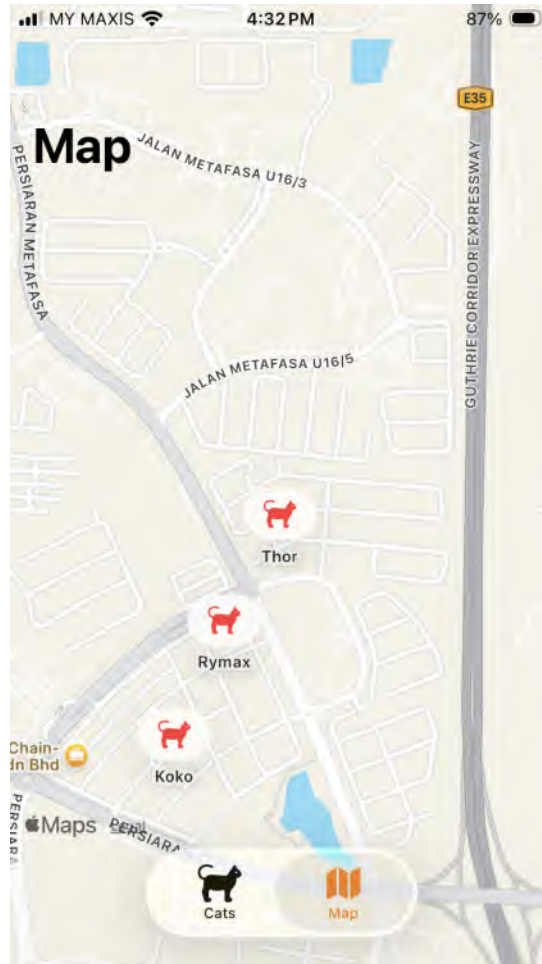


Figure 2.8: Map screen

Let's study the different parts of this screen.

A tab bar at the bottom of the screen displays the **Cats** and **Map** buttons. The **Map** button is selected, and you can see a map on the screen with markers for each cat that is nearby.

Tapping a marker will display the **Cat Detail** screen for that cat.

This completes the tour of the app. Now, it's time to start building the user interface for it!

Modify your app playground

Now that you know what the screens of the app are going to look like, you can start building it. If you have not yet done so, open the app playground you created in *Chapter 1, Hello, Swift Playground*:

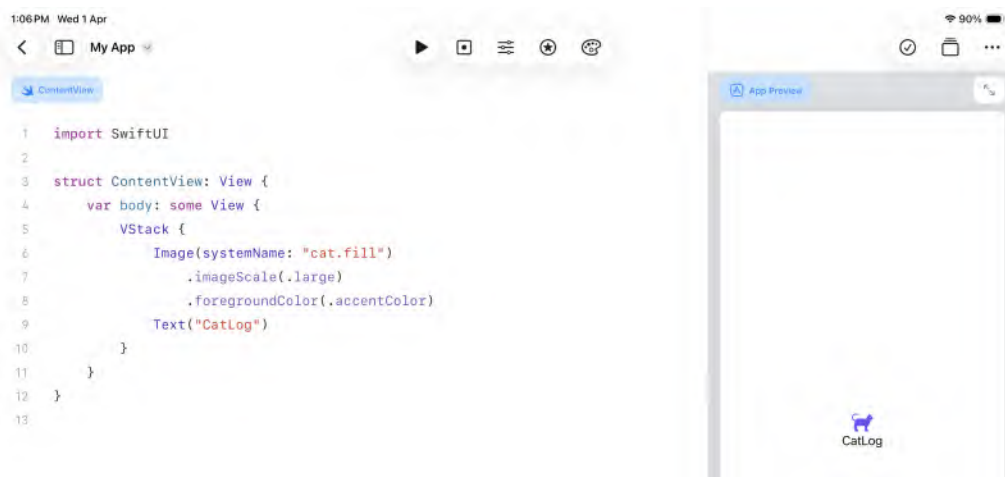


Figure 2.9: The My App playground

To configure the user interface, you will modify the ContentView file by adding a tab view to it. The tab view contains a tab bar that will display two buttons at the bottom of the screen. Tapping a button will display the screen associated with it. These screens correspond to the **Cats** and **Map** screens shown in the app tour. Let's see how to do this in the next section.

Set up a tab bar

As you saw in the app tour, the *CatLog* app has a tab bar with two buttons at the bottom of the screen, which are used to display the **Cats** and **Map** screens. You will modify the code in the ContentView file to make your app display a tab bar with two buttons. Follow these steps:

1. In the ContentView file, remove the VStack and its contents. Your code should look like this:

```
import SwiftUI

struct ContentView: View {
    var body: some View {
```

```
    }
}
```

Note that the App Preview no longer displays the cat symbol and text.

2. Add the following code between the curly braces of body:

```
TabView {
  Tab("Cats", systemImage: "cat.fill") {
    Text("Cats")
  }
  Tab("Map", systemImage: "map") {
    Text("Map")
  }
}
```

3. Verify that a tab bar with two buttons has appeared in App Preview, and you can tap them to display either **Cats** or **Map** on the screen:

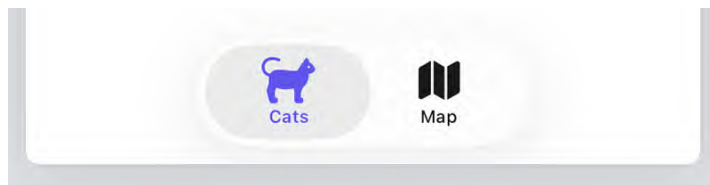


Figure 2.10: App Preview showing tab bar with two buttons

As you have seen in the app tour, some screens have titles and buttons in the navigation bar. In the next section, you will learn how to add navigation bars to your screens so that you can add buttons and titles to them later as required.

Embed views in navigation stacks

As you saw in the app tour, the **Cats** and **Map** screens both have a navigation bar at the top of the screen. To add the navigation bars for both screens, you will embed the text views in navigation stacks and use modifiers to display titles on each screen. Follow these steps:

1. In the ContentView file, modify your code as follows:

```
Tab("Cats", systemImage: "cat.fill") {
  NavigationStack {
    Text("Cats")
    .navigationTitle("Cats")
  }
}
```

```
}  
Tab("Map", systemImage: "map") {  
    NavigationStack {  
        Text("Map")  
        .navigationTitle("Map")  
    }  
}
```

2. Verify that both screens now have titles in App Preview:

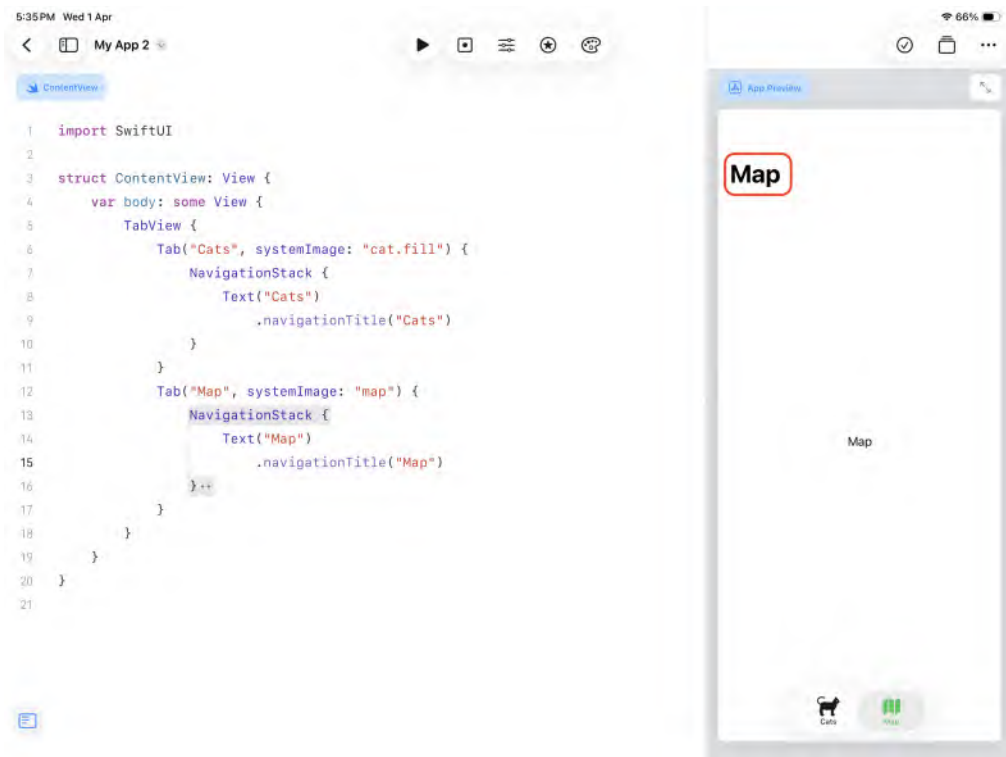


Figure 2.11: App Preview showing a navigation bar

Congratulations! You've just configured the tab bar and navigation stacks for your app!

At present, the name of your app is still `My App`, and the icon for your app was randomly assigned. You will learn how to change the name and icon of your app in the next section.

Configure your app playground

When you create a new app playground in Swift Playground, the default name used will be `My App`, and the app icon is randomly assigned. Let's configure the name and icon for your app playground. Follow these steps:

1. Tap the Sidebar button to display all the assets in your app playground:

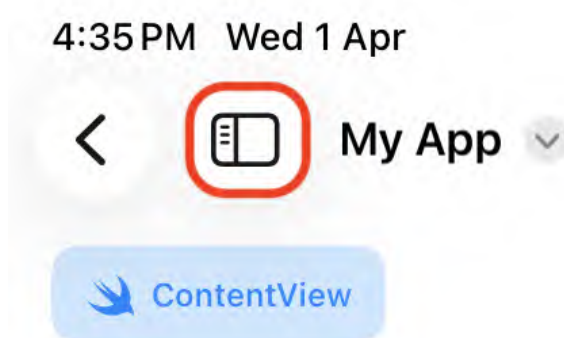


Figure 2.12: Assets button in the app playground

A column showing all the assets and settings in your app playground will appear.

2. Tap the **App Settings** button to display the **App Settings** window. You will now be able to change the name, accent color and icon for your app:

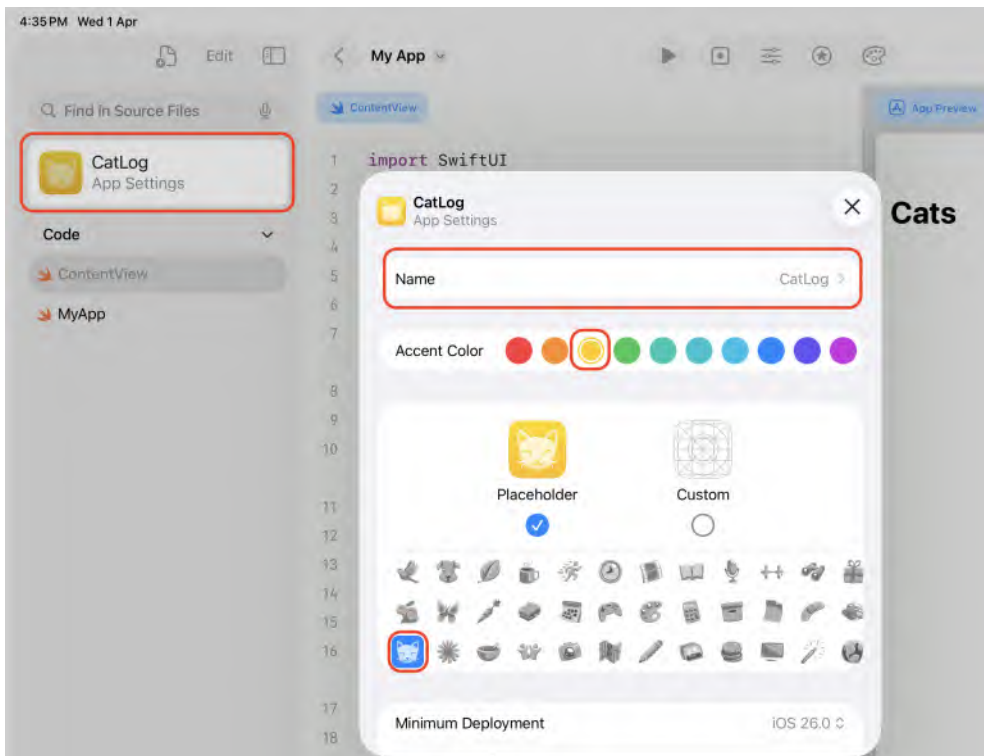


Figure 2.13: App Settings button and window

Tap the **X** button when you are done to dismiss the **App Settings** window.

3. Even though you have changed the name of your app, your app playground still has the original name, **My App**. To change it, tap the < button to return to the welcome screen:

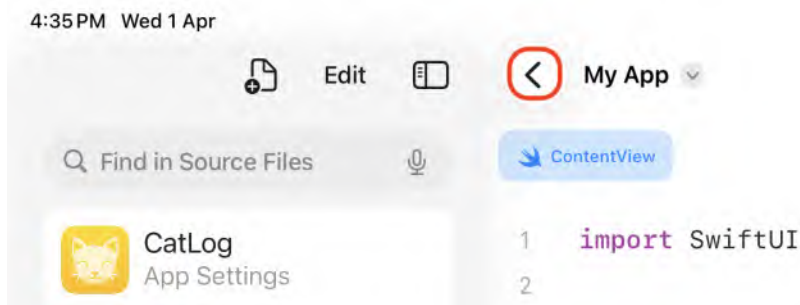


Figure 2.14: Button on the app playground

4. In the welcome screen, tap and hold on the **My App** icon, and choose **Rename** from the pop-up menu:

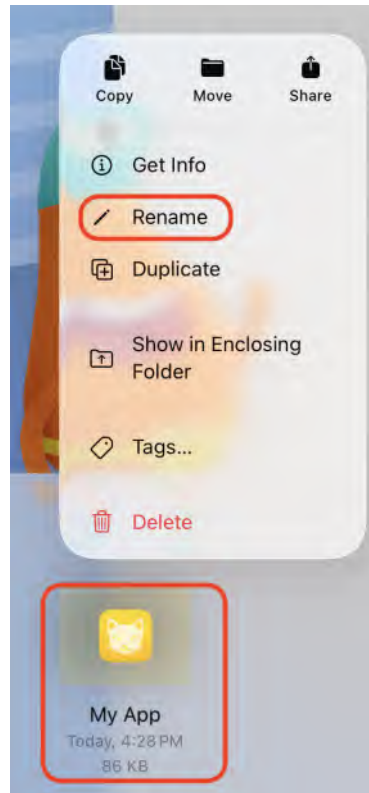


Figure 2.15: Welcome screen showing Rename selected from the pop-up menu

5. Change the name of the app playground to CatLog, and tap *Return*:

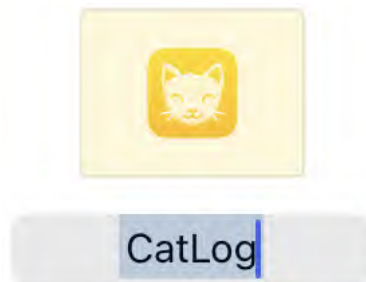


Figure 2.16: App Playground name

You have configured the name, icon, and accent color for your app playground! Well done!

Summary

In this chapter, you learned some useful SwiftUI terms. This will make it easier for you to understand the remainder of this book, as well as other books or online resources on the subject.

Then, you also learned about the different screens used in the *CatLog* app and how a user could use the app. As you recreate the app's UI from scratch, you're able to compare what you're doing to what the actual app looks like.

After that, you learned how to add a tab bar with buttons and icons to your app. Then, you added navigation stacks and titles for the **Cats** and **Map** screens. This will familiarize you with adding and configuring UI elements for your own apps.

Finally, you changed the name, accent color, and icon for your app playground.

In the next chapter, you will continue setting up your app's UI and become familiar with more UI elements. You will also add and configure the remaining screens for your app.

3

Build the User Interface

In the previous chapter, you modified an existing app playground, added a tab bar to your app that allowed the user to select between the **Cats** and **Map** screens, and configured the tab bar button titles and icons. When your app is launched, the **Cats** screen is displayed, but it is currently blank.

As you saw in the app tour in *Chapter 2, Set Up the User Interface*, the **Cats** screen should display a list of cats.

In this chapter, you will add a list containing three text views to the **Cats** screen, as well as a button that will display a view representing the **Add New Cat** screen when tapped. You'll also configure a Cancel button to dismiss this view and return you to the **Cats** screen.

By the end of this chapter, you'll have learned how to add custom SwiftUI views to your app playground, create and display a list, and present a view modally. This will be very useful when you design the user interface for your own apps.

The following topics will be covered in this chapter:

- Create a custom view
- Modify the tab view
- Present a view modally

Note

Download the code bundle and the PDF version of this book

Your purchase includes a DRM-free PDF copy of this book, the code bundle, and a range of exclusive benefits. To unlock everything, follow the *Free benefits with your book* section in the *Preface*.

Technical requirements

You will continue working on the CatLog app playground that you modified in the previous chapter.

The completed app playground for this chapter is in the `Chapter03.zip` file of the code bundle for this book. To access the repository link, follow the steps in the *Download the example code files* section in the *Preface*.

Check out the following video to see the code in action:

<https://packt.link/iBy2f>

Let's start by adding a new SwiftUI view to your project, which will eventually display a list of cats, in the next section.

Create a custom view

As you saw in the app tour, the *CatLog* app displays a list of cats when it is launched. A list in SwiftUI is a container that presents other SwiftUI views in a single column. Optionally, you can select one or more of the views in the list. The contents of the list can either be static or dynamic, and you can use either predefined views or custom views as list elements.

Note

For more information on SwiftUI lists, see this link: <https://developer.apple.com/documentation/SwiftUI/List>.

In this section, you'll start by creating a new SwiftUI view, then you'll add a list containing three text views to it. Open the CatLog app playground you created in the previous chapter and run the app to make sure everything still works as it should, then follow these steps:

1. Tap the Sidebar button to display all the assets in the app playground:

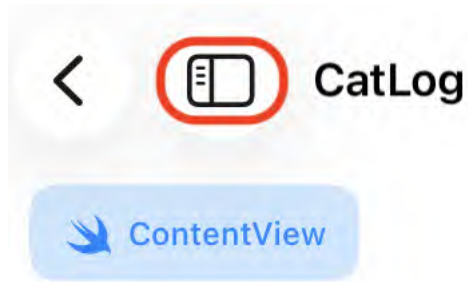


Figure 3.1: Assets button in the app playground

2. The list of assets in the playground will appear. Tap the New button to create a new asset:

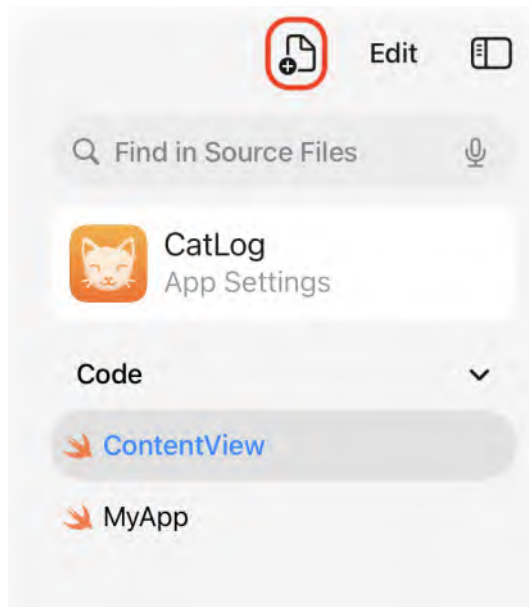


Figure 3.2: List of assets with New button shown

3. In the pop-up menu, choose **Swift File** to add a new Swift file to your app playground:

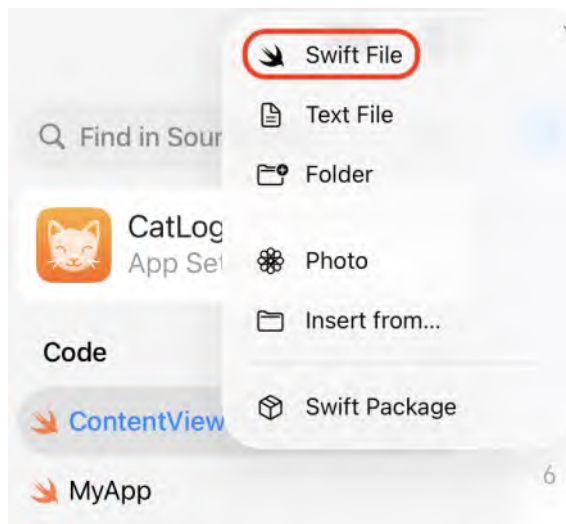


Figure 3.3: Pop-up menu with Swift File highlighted

4. Change the name of the new file to CatsView and tap *Return*. Verify that the contents of the newly added file are visible in the Editor area, and contain a statement to import the SwiftUI framework:

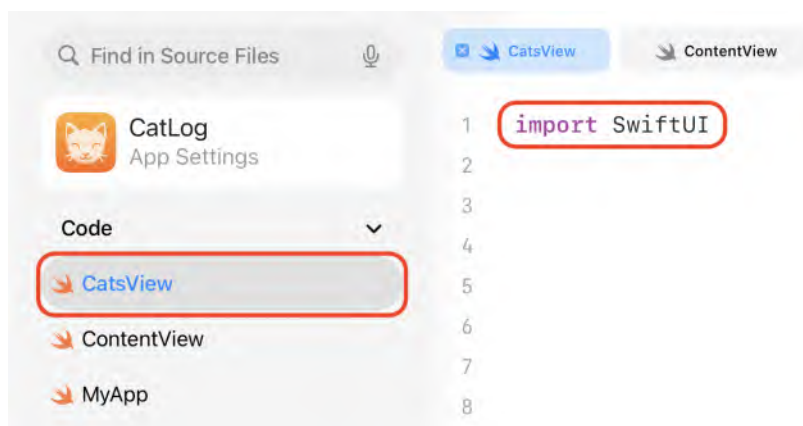


Figure 3.4: List of assets showing CatsView file

5. Add the following code to the `CatsView` file after the import statement:

```
struct CatsView: View {
    var body: some View {
        NavigationStack {
            List {
                Text("Thor")
                Text("Loki")
                Text("Herman")
            }
            .navigationTitle("Cats")
        }
    }
}
```

This code declares a new SwiftUI view named `CatsView`. It has a list containing three text views representing three cats. This list is embedded in a navigation stack, with the navigation title set to `Cats`. The navigation title will be displayed in the navigation bar.

6. To preview what this view will look like, add the following code to the `CatsView` file after all other code in it:

```
#Preview {
    CatsView()
}
```

This code will generate a preview of `CatsView` in App Preview.

7. Tap the **CatsView** tab in App Preview. A preview of CatsView will appear:

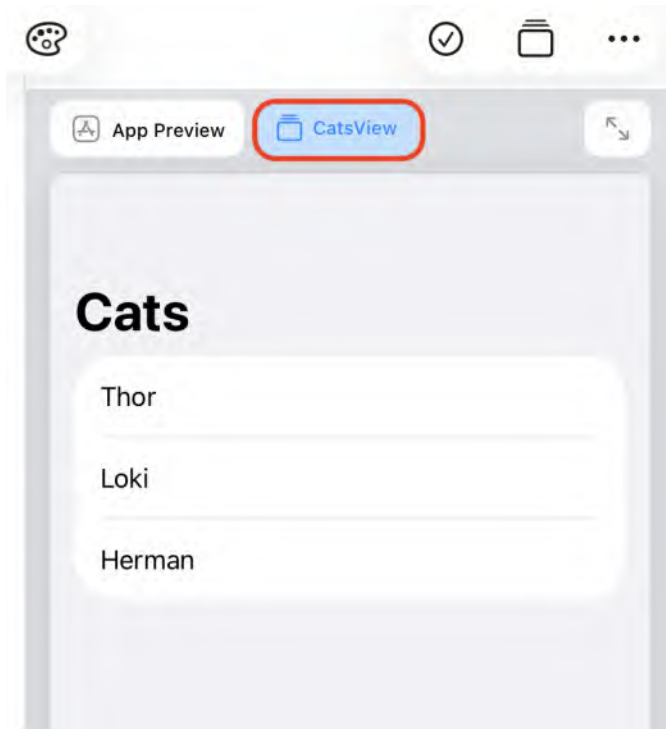


Figure 3.5: App preview showing a preview of CatsView

You have created a custom view containing a list of cats, and you have previewed it in App Preview. In the next section, you will replace the text view representing the **Cats** screen with your custom view.

Modify the tab view

You've created a new SwiftUI view named `CatsView`, which displays a list containing three text views. Initially, you used text views to represent the **Cats** screen and **Map** screen in your app playground. In this section, you'll replace the text view used to represent the **Cats** screen with your custom view. Follow these steps:

1. In your app playground, tap **ContentView** in the list of assets or tap the **ContentView** tab. The contents of the `ContentView` file will appear in the Editor area:

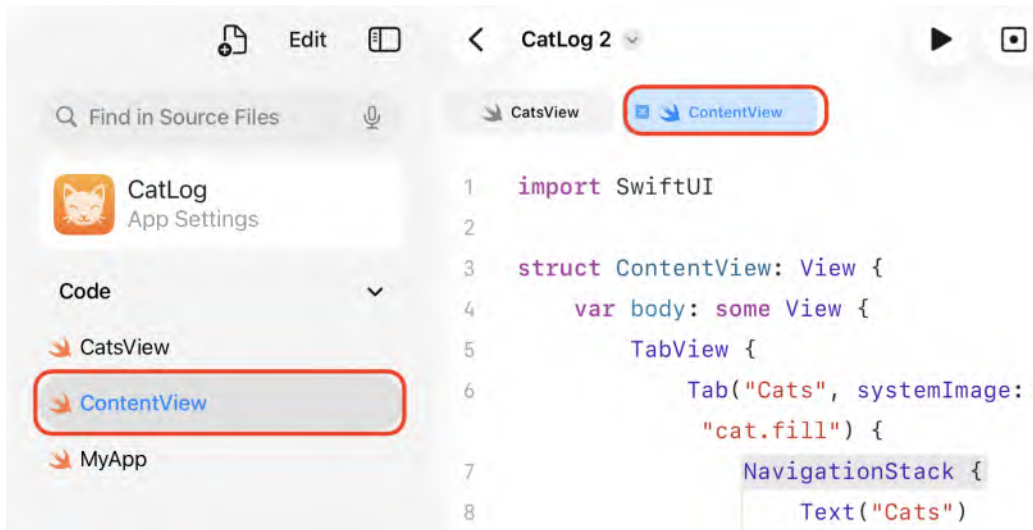


Figure 3.6: Editor area showing the contents of the `ContentView` file

2. Modify the code in the first tab as shown:

```
Tab("Cats", systemImage: "cat.fill") {  
    CatsView()  
}
```

3. Verify that the `CatsView` view is visible in App Preview when the **Cats** button is selected:

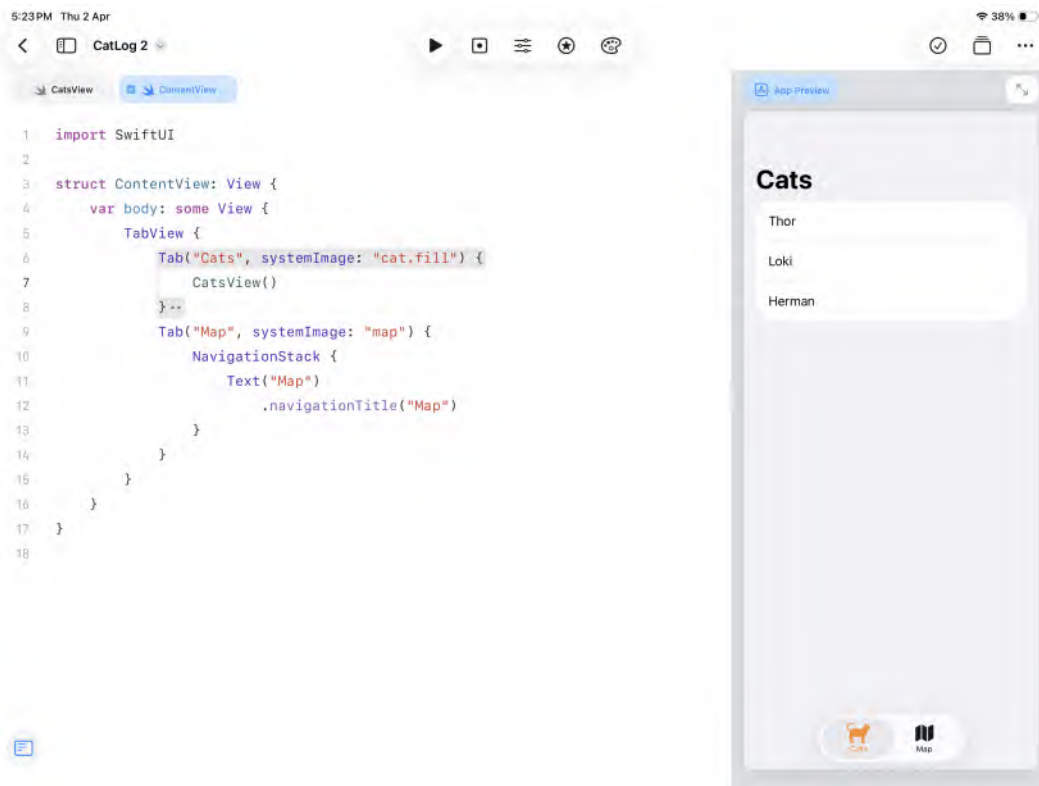


Figure 3.7: App Preview showing the `CatsView` view in the first tab

You have successfully created a new custom view representing the **Cats** screen and added it to the tab view in your app playground! Great job!

In the next section, you will learn how to add a button to the **Cats** screen to display another screen which will eventually be used to add new cats to your app playground.

Present a view modally

The navigation bar for the **Cats** screen can be configured to display a title and buttons. You have already configured the title in *Chapter 2, Set Up the User Interface*. Now, you will add a button to the navigation bar. When tapped, this button will display a view representing the **Add New Cat** screen. This view will be a new custom view which you will add to the app playground. The view will be presented modally, which means you won't be able to do anything else until it is dismissed.

To dismiss it, you'll add a Cancel button to the view's navigation bar. You'll also add a Save button, but you'll only implement its functionality in *Chapter 7, Pass Data Between Screens*. Let's start by adding a button to the navigation bar in the next section.

Add a button to the navigation bar

As shown in the app tour in *Chapter 2, Set Up the User Interface*, there is a + button in the top-right corner of the **Cats** screen. To implement this, you'll add code to the **CatsView** view to display a button to the navigation bar. Follow these steps:

1. In your app playground, tap the **CatsView** file in the list of assets or the **CatsView** tab. Add the following code after the `.navigationTitle("Cats")` modifier:

```
.navigationTitle("Cats")
.toolbar {
    ToolbarItem(placement: .primaryAction) {
        Button("Add", systemImage: "plus") {

        }
    }
}
```

This code adds a button displaying a + sign to the navigation bar. Setting the button's placement to `.primaryAction` will place the button on the right side of the navigation bar.

Note

For more information on SwiftUI buttons, see this link:

<https://developer.apple.com/documentation/swiftui/button>.

2. Verify that a button displaying a + sign is visible at the top right corner of the **Cats** screen:

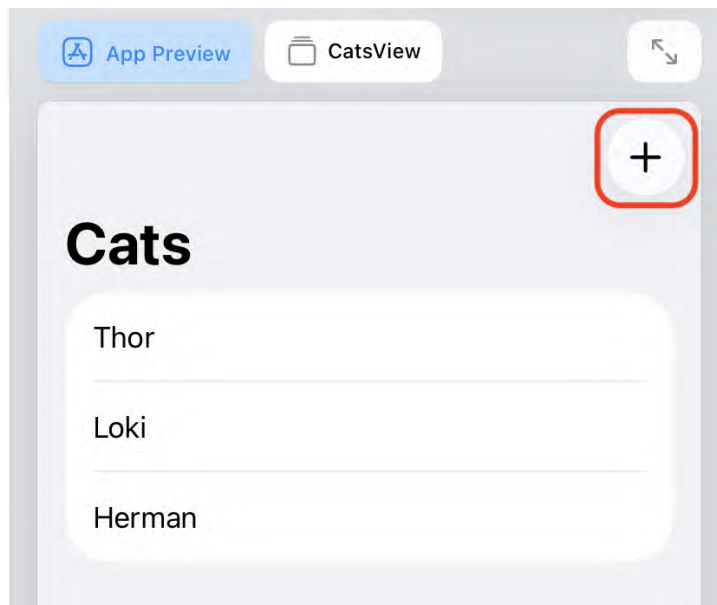


Figure 3.8: App preview showing button at the top right of the Cats screen

You now have a + button in your navigation bar. In the next section, you will add a new SwiftUI view to represent the **Add New Cat** screen that will appear when the button is tapped.

Add a new view

As shown in the app tour in *Chapter 2, Set Up the User Interface*, when you tap the + button in the navigation bar, the **Add New Cat** screen will be displayed. You'll add a new SwiftUI view to your app playground to represent this screen. Follow these steps:

1. Tap the Sidebar button and tap the New button. Choose **Swift File** from the pop-up menu and name the file `AddNewCatView`. It will be added to the app playground and its contents will appear in the Editor area:

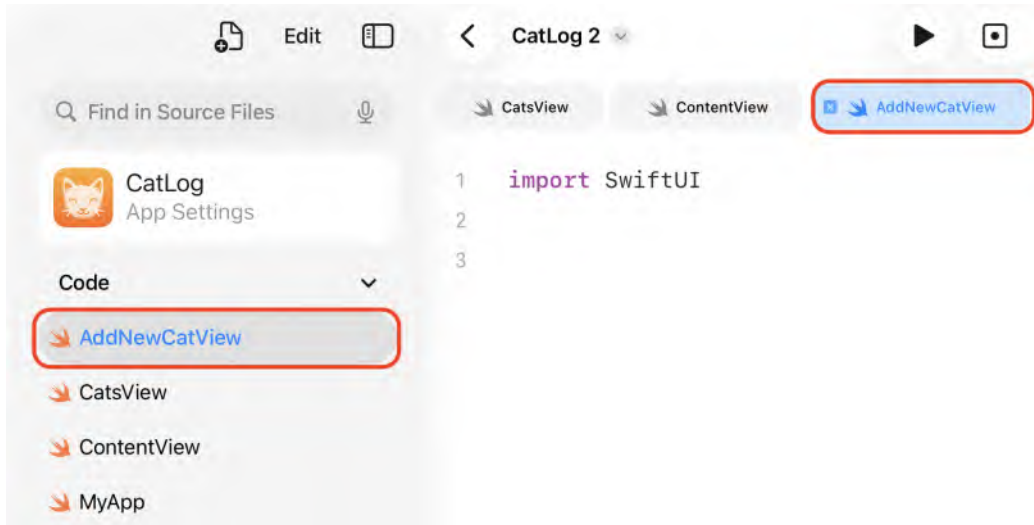


Figure 3.9: AddNewCatView file added to the app playground

2. Add the following code to this file after the `import` statement:

```
struct AddNewCatView: View {
    var body: some View {
        NavigationStack {
            Text("Add New Cat View")
                .navigationTitle("Add New Cat")
        }
    }
}

#Preview {
    AddNewCatView()
}
```

This code creates a new SwiftUI view named `AddNewCatView`, which displays **Add New Cat View** on the screen and **Add New Cat** in the navigation bar.

3. Switch to the `CatsView` file, and add the following statement after the `CatsView` declaration:

```
struct CatsView: View {
    @State private var showAddCatView = false
```

This statement creates a new variable. A variable is a container that has a name and can store a value. The name of this variable is `showAddCatView`, and the initial value assigned to it is `false`. The type of value that can be stored in a variable is inferred from the initial value. In this case, the `showAddCatView` can store either `false` or `true`. You will use this value to determine if the `AddNewCatView` view is visible. The `@State` keyword tells SwiftUI to keep track of this variable and update the user interface if its value is changed.

Note

To learn more about variables and state, see this link: <https://developer.apple.com/tutorials/develop-in-swift/update-the-ui-with-state>.

4. Modify the code for the button as follows:

```
.navigationTitle("Cats")
.toolbar {
    ToolbarItem(placement: .primaryAction) {
        Button("Add", systemImage: "plus") {
            showAddCatView = true
        }
        .sheet(isPresented: $showAddCatView) {
            AddNewCatView()
        }
    }
}
```

The statement you added between the button's curly braces will be executed when the button is tapped. In this case, `showAddCatView` will be assigned the value of `true`.

The `.sheet(isPresented:)` modifier determines the view to be presented when `showAddCatView` is set to `true`. In this case, the `AddNewCatView` view will be displayed. The use of the `$` here means that the value stored in `showAddCatView` is dependent on the visibility of the `AddNewCatView` view, and will be set to `false` when the user dismisses the presented view.

5. Tap the `+` button in App Preview:

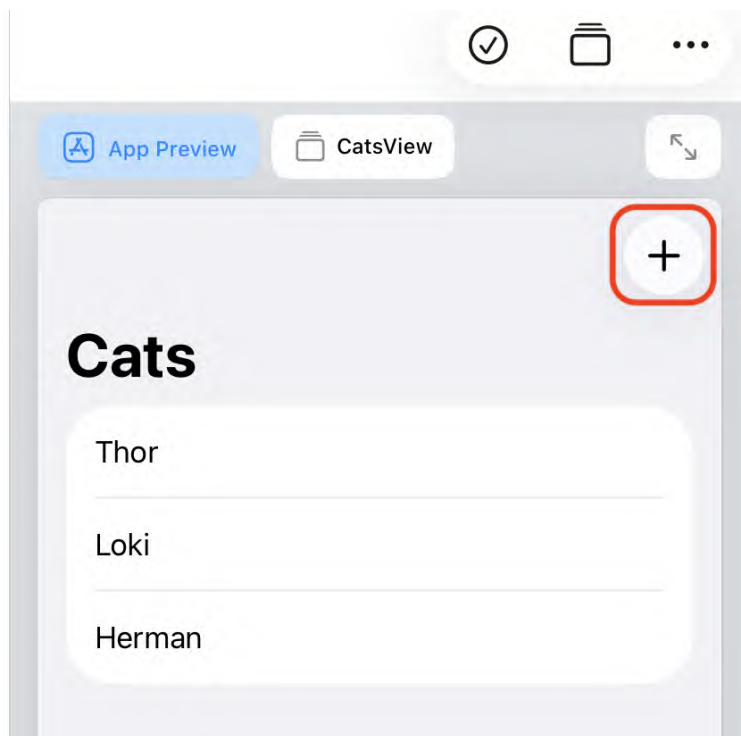


Figure 3.10: App Preview with `+` button highlighted

6. The AddNewCatView view will appear. Drag the top of the AddNewCatView view down to dismiss it:

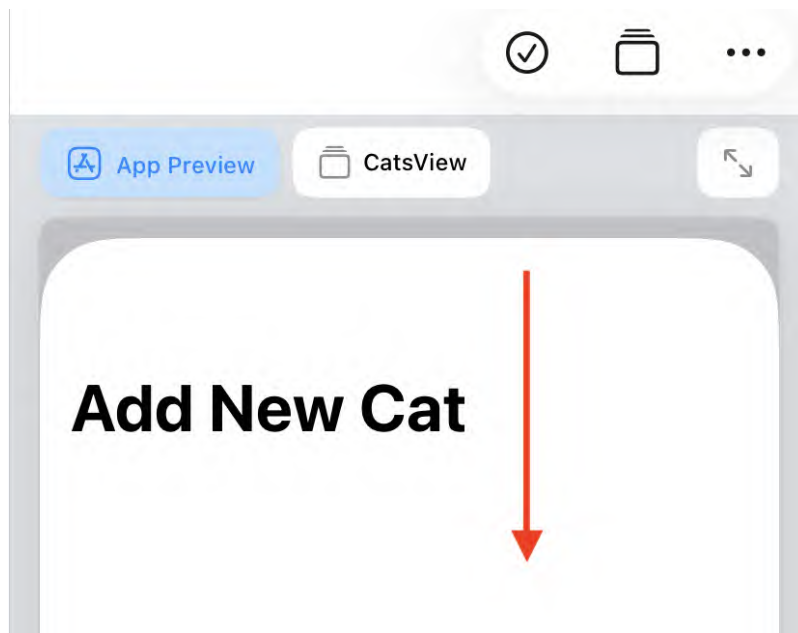


Figure 3.11: App Preview showing AddNewCatView

You can only dismiss this view by dragging it downward at present. In the next section, you will add a Cancel button to the navigation bar and program it to dismiss the view. You'll also add a Save button, but you won't program it yet.

Add Cancel and Save buttons to the navigation bar

In addition to a title, you can place buttons on both the left and right sides of a navigation bar. Follow these steps to add the Cancel and Save buttons to the navigation bar in the AddNewCatView view:

1. Switch to the AddNewCatView file, and add the following statement after the AddNewCatView declaration:

```
struct AddNewCatView: View {  
    @Environment(\.dismiss) private var dismiss
```

This statement allows you to assign the built-in way to dismiss a view to a variable, `dismiss`.

Note

To learn more about view environment, see this link: <https://developer.apple.com/documentation/swiftui/environment>.

To learn more about dismiss, see this link:

<https://developer.apple.com/documentation/swiftui/environmentvalues/dismiss>.

2. Add the following code after the `.navigationTitle("Add New Cat")` modifier:

```
.toolbar {
    ToolbarItem(placement: .cancellationAction) {
        Button("Cancel", systemImage: "xmark") {
            dismiss()
        }
    }
    ToolbarItem(placement: .confirmationAction) {
        Button("Save", systemImage: "checkmark") {

        }
    }
}
```

This code adds two buttons to the navigation bar:

- The Cancel button has an **X** symbol, and setting placement to `.cancellationAction` places it on the left side of the navigation bar. Tapping it will dismiss the `AddNewCatView` view.
- The Save button has a tick mark symbol, and setting placement to `.confirmationAction` places it on the right side of the navigation bar. Since there is nothing between the button's curly braces, tapping it does not do anything.

3. In App Preview, tap the + button. The **Add New Cat** screen will be displayed. Verify that it has two buttons in the navigation bar:

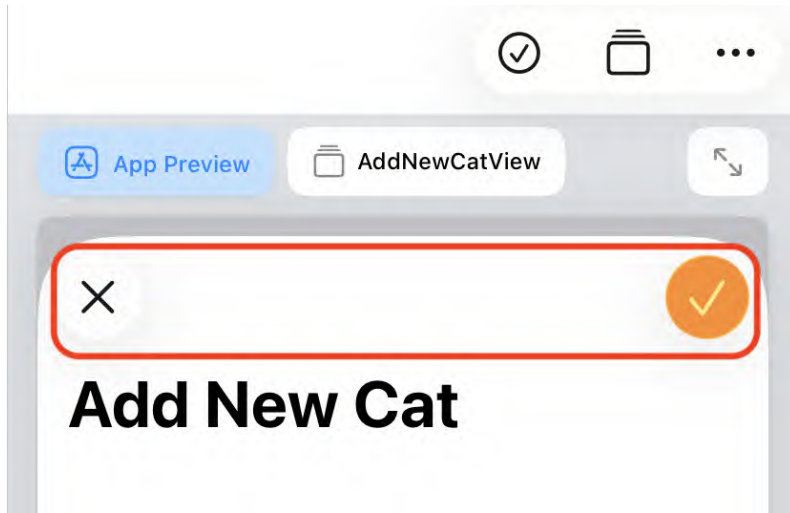


Figure 3.12: App Preview showing Add New Cat screen with two buttons

4. Tap the X button to dismiss the **Add New Cat** screen:

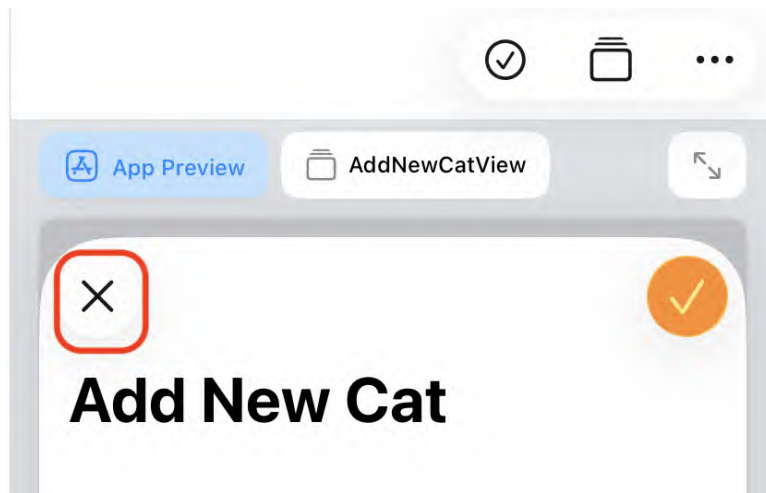


Figure 3.13: App Preview showing Cancel button

Congratulations! You've completed the basic structure for the **Cats** screen!

Summary

In this chapter, you added a list displaying three text views to the **Cats** screen, as well as a button that will display a view representing the **Add New Cat** screen when tapped. You also added cancel and save buttons to the **Add New Cat** screen and configured the Cancel button to dismiss it, returning you to the **Cats** screen.

You now know how to add custom SwiftUI views to your app playground, create and display a list, and present a view modally. This will be very useful when you design the UI for your own apps.

In the next chapter, you'll implement the **Cat Detail** screen of your app and a map view for the **Map** screen.

4

Complete the User Interface

In the previous chapter, you configured the **Cats** screen to display three text views, added a button to the navigation bar to present a view representing the **Add New Cat** screen modally, and added Cancel and Save buttons to it.

In this chapter, you'll add the remaining screens shown during the app tour in *Chapter 2, Set Up the User Interface*. You'll add the **Cat Detail** screen, which will be displayed when a text view in the **Cats** screen is tapped. You'll also add the **Map** screen, which displays a map.

By the end of this chapter, you'll have learned how to go to another screen when a view in the **Cats** screen is tapped, and how to create a view showing a map. The basic user interface of your app will be complete, and you will be able to walk through all the screens in App Preview. None of the screens will be displaying data, but you will finish their implementation in *Part 3* of this book.

The following topics will be covered in this chapter:

- Implement the **Cat Detail** screen
- Implement the **Map** screen

Technical requirements

You will continue working on the CatLog app playground that you modified in the previous chapter.

The completed Swift Playground app playground project for this chapter is in the `Chapter04.zip` file of the code bundle for this book. To access the repository link, follow the steps in the *Download the example code files* section in the *Preface*.

Check out the following video to see the code in action:

<https://packt.link/Q3TKu>

In the next section, you'll add a new view to represent the **Cat Detail** screen, which will be displayed when a text view in the **Cats** screen is tapped.

Implement the Cat Detail screen

As shown in the app tour in *Chapter 2, Set Up the User Interface*, when you tap a journal entry in the **Cats** screen, a **Cat Detail** screen containing the details of that cat will appear. In this section, you'll add a new view to represent the **Cat Detail** screen. Follow these steps:

1. In your app playground, tap the Sidebar button and tap the New button to create a new Swift file.
2. Name the file `CatDetailView`. Its contents will appear in the Editor area.
3. Add the following code to the file after the `import` statement to declare and define `CatDetailView`:

```
struct CatDetailView: View {
    var body: some View {
        Text("Cat Detail View")
            .navigationTitle("Cat Detail")
    }
}

#Preview {
    NavigationStack {
        CatDetailView()
    }
}
```

Note that this view is not embedded in a navigation stack. This is because it will be added to the existing navigation stack declared in `CatsView()`. Note that the `#Preview` block does embed `CatDetailView()` in a navigation stack, so that you are able to see the navigation title in App Preview.

4. Switch to the `CatsView` file and modify the code in the list as follows:

```
List {
    NavigationLink(destination: CatDetailView()) {
        Text("Thor")
    }
}
```

```
        NavigationLink(destination: CatDetailView()) {  
            Text("Loki")  
        }  
        NavigationLink(destination: CatDetailView()) {  
            Text("Herman")  
        }  
    }  
}
```

Each text view has been embedded in a navigation link, and the assigned destination is `CatDetailView()`. When the text view is tapped, `CatDetailView()` will be added to the list of views in the navigation stack, and it will appear on the screen. The view title will appear in the navigation bar.

Note

You can learn more about navigation links in SwiftUI at this link: <https://developer.apple.com/documentation/swiftui/navigationlink>.

5. In App Preview, note that each text view now has a > symbol on the right side. When you tap any text view, the **Cat Detail** screen will appear:

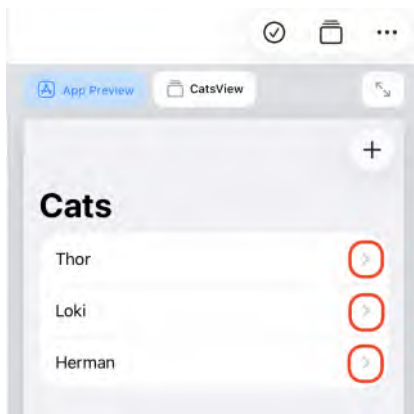


Figure 4.1: App Preview showing > symbols on the Cats screen

6. Tap the Back (<) button in the **Cat Detail** screen to return to the **Cats** screen:

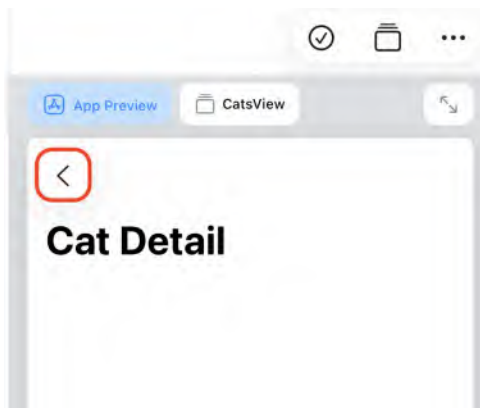


Figure 4.2: App Preview showing the < button on the Cat Detail screen

You have successfully implemented the **Cat Detail** screen! Great!

In the next section, you will create a view that represents the **Map** screen and make it display a map.

Implement the Map screen

When you launch the app, the **Cats** screen is displayed. Tapping the **Map** button in the tab bar currently displays a text view. To display a map, you'll create a new view and configure it to display a map, then replace the text view with this new view. Follow these steps:

1. In your app playground, tap the Sidebar button and tap the New button to create a new Swift file.
2. Name the file `MapView`. Its contents will appear in the Editor area.
3. Add the following code to the file after the `import` statement to declare and define `MapView`:

```
import MapKit
struct MapView: View {
    var body: some View {
        NavigationStack {
            Map()
                .navigationTitle("Map")
        }
    }
}
```

```
#Preview {  
    MapView()  
}
```

When you tap the **MapView** button in App Preview, you will see a map showing your approximate location.

Note

The MapKit framework has everything you need to configure and display maps in your app. You can learn more about it at this link:

<https://developer.apple.com/documentation/mapkit/mapkit-for-swiftui>.

4. Switch to the ContentView file and modify the code for the second tab as follows:

```
Tab("Map", systemImage: "map") {  
    MapView()  
}
```

MapView() will be displayed when that **Map** tab is selected.

5. In App Preview, tap the **Map** button. The **Map** screen will appear:

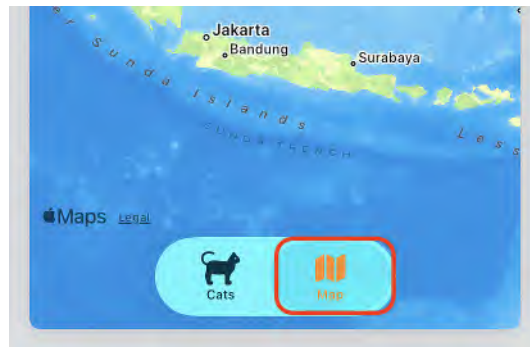


Figure 4.3: App Preview showing the Map screen

Wonderful! You've now completed the basic user interface for your app!

Summary

In this chapter, you completed the basic structure of your app. You added a new view representing the **Cat Detail** screen and configured the views on the **Cats** screen to display the **Cat Detail** screen when tapped. You also added a **Map** screen, which displays a map showing your approximate location.

You have successfully implemented all the screens required for your app, and you'll be able to test your app's flow in App Playground. You should also be more proficient in implementing navigation for a SwiftUI app.

In the next chapter, you'll add and configure more views to the **Cats** screen, **Add New Cat** screen and **Cat Detail** screen so that they match the screens shown in the app tour.

5

Modify App Screens

In *Chapter 3, Build the User Interface*, you added some of the screens required for your app to match what was shown during the app tour. In *Chapter 4, Complete the User Interface*, you added the remaining screens required for your app. You're now able to navigate through all the screens of your app in app playground, but the screens still lack the user interface elements required for data input and data display.

In this chapter, you'll add and configure the missing user interface elements to the **Cats**, **Add New Cat**, and **Cat Detail** screens, to match the design shown in the app tour.

For the **Cats** screen, you'll create a new view to replace the text views that you have been using previously. This view will be used to represent an individual cat in the list of cats and will contain an image view and two text views, so that it can display the cat's photo, name, and description.

For the **Add New Cat** screen, you'll modify it by adding a custom view, a switch, two text fields, and an image view, so that you can enter the details of a new cat. You'll also configure the image view to show a default image.

For the **Cat Detail** screen, you'll add text and image views to it and configure the image views to show default images, so that the screen can display the details of an existing cat. With all the user interface elements in place, your app will be ready for code, which you will implement in *Part 2* of this book.

By the end of this chapter, you will be more proficient in adding and positioning user interface elements and will have gained more experience in determining their position relative to one another. This will be useful to ensure compatibility with different screen sizes and orientations, enabling you to easily prototype the appearance and flow of your apps.

The following topics will be covered in this chapter:

- Modify the **Cats** screen
- Modify the **Add New Cat** screen
- Modify the **Cat Detail** screen

Technical requirements

You will continue working on the CatLog app playground that you modified in the previous chapter.

The completed app playground for this chapter is in the `Chapter05.zip` file of the code bundle for this book. To access the repository link, follow the steps in the *Download the example code files* section in the *Preface*.

Check out the following video to see the code in action:

<https://packt.link/c8WCq>

In the next section, you'll create a new view and add some user interface elements to make it match the **Cats** screen shown in the app tour.

Modify the Cats screen

Let's see what the **Cats** screen looked like in the tour of the app shown in *Chapter 2, Set Up the User Interface*:

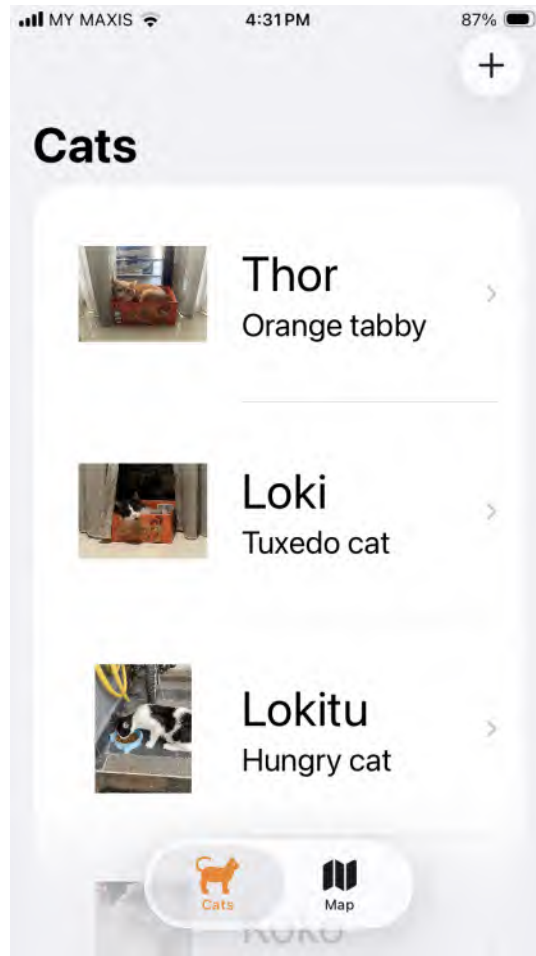


Figure 5.1: The Cats screen for the completed CatLog app playground

As you can see, the **Cats** screen has a list containing views for each individual cat. Each view contains the cat's photo, name, and description. In *Chapter 3, Build the User Interface*, you used three text views to represent three individual cats. You will create a new view, add user interface elements to it, and use it in place of the text views in the **Cats** screen, matching the design shown in the app tour. Follow these steps:

1. In your app playground, tap the Sidebar button and tap the New button to create a new Swift file.
2. Name the file `CatRowView`. Its contents will appear in the Editor area.

3. Add the following code after the import statement to declare and define `CatRowView`:

```
struct CatRowView: View {
    var body: some View {
        HStack {
            Image(systemName: "cat.fill")
            VStack {
                Text("Cat name")
                Text("Cat description")
            }
        }
    }
}

#Preview {
    CatRowView()
}
```

This code creates a view that contains a horizontal stack view containing an image view and a vertical stack view. The vertical stack view contains two text views.

4. In App Preview, tap the **CatRowView** button and verify the view appears as follows:

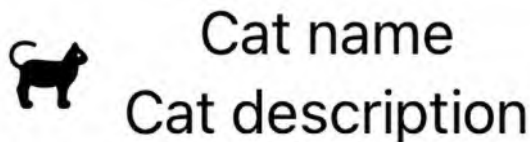


Figure 5.2: App Preview showing a preview of `CatRowView`

You will use modifiers to improve the appearance of this view.

5. Modify the code as shown to change the image view size, change the styles of the text views, and extend the view to the edges of app playground:

```
struct CatRowView: View {
    var body: some View {
        HStack {
            Image(systemName: "cat.fill")
                .resizable()
                .scaledToFit()
                .frame(width: 90, height: 90)
            VStack {
                Text("Cat name")
                Text("Cat description")
            }
        }
    }
}
```

```

VStack {
    Text("Cat name")
        .font(.largeTitle)
        .frame(maxWidth: .infinity,
            alignment: .leading)
    Text("Cat description")
        .lineLimit(2, reservesSpace: true)
        .font(.title2)
        .frame(maxWidth: .infinity,
            alignment: .leading)
}
}
}
}

```

The modifiers for the image view have set its size to 90 x 90 points and scaled the image to fit the size of the image view.

The modifiers for the first text view have set its font to the `.largeTitle` size, aligned the text to the left, and extended it to the full width of the screen.

The modifiers for the second text view have limited the number of lines it can display to 2, reserved the space required to display two lines, set its font to the `.title2` size, aligned the text to the left, and extended it to the full width of the screen.

6. In App Preview, tap the **CatRowView** button and verify the modified view appears as follows:



Figure 5.3: App Preview showing preview of *CatCellView*

7. Switch to the `CatsView` file and replace each text view in the list with `CatRowView`:

```

List {
    NavigationLink(destination: CatDetailView()) {
        CatRowView()
    }
    NavigationLink(destination: CatDetailView()) {

```

```
CatRowView()  
}  
NavigationLink(destination: CatDetailView()) {  
    CatRowView()  
}  
}
```

8. In App Preview, verify the **Cats** screen appears as follows:

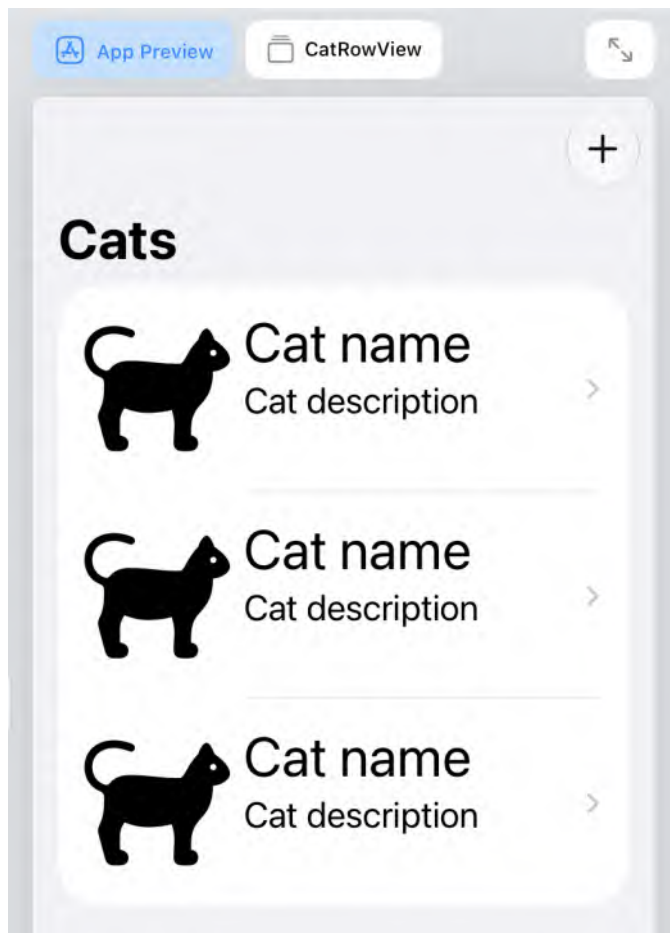


Figure 5.4: App Preview showing the Cats screen

You have successfully created a new view, `CatRowView`, added and configured an image view displaying a cat to it, and added and configured text views to display sample text. You also replaced each text view in the **Cats** screen with `CatRowView`. As you can see, the **Cats** screen

now has all the user interface elements required to display data as shown in the app tour. Fantastic!

In the next section, you'll add a vertical stack view containing user interface elements to the **Add New Cat** screen.

Modify the Add New Cat screen

Let's see what the **Add New Cat** screen looks like in the tour of the app shown in *Chapter 2, Set Up the User Interface*:

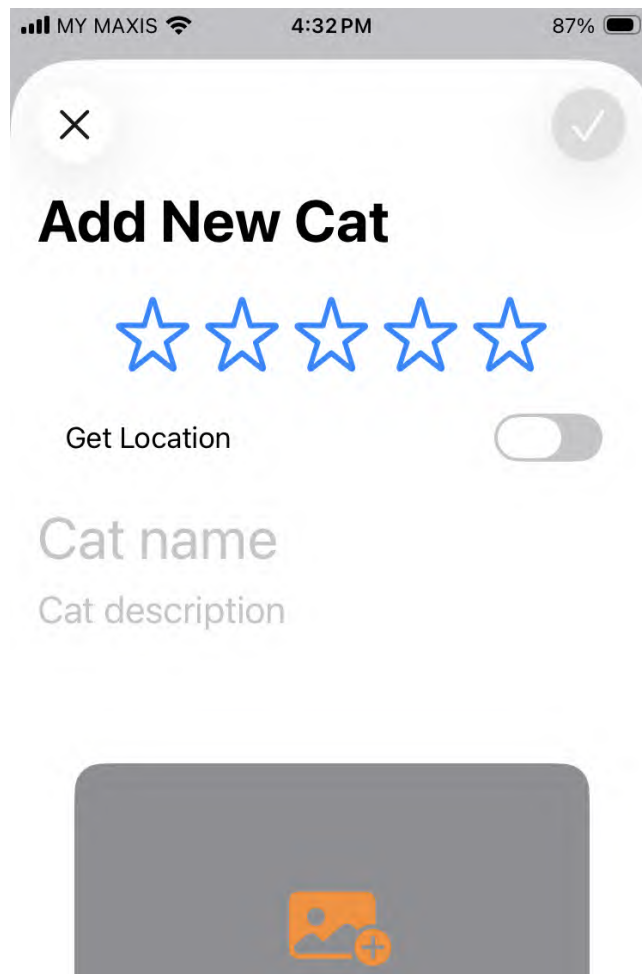


Figure 5.5: The Add New Cat Entry screen for the completed CatsLog app playground

Apple provides an extensive library of user interface elements that you can use in your own apps. This helps to give all iOS apps a consistent look and feel. As you can see, the **Add New Cat** screen has the following elements:

- A custom view showing star ratings
- A switch that allows you to get your current location
- A text field for the cat's name
- A text field for the cat's description
- An image view for a photo that you will select from your Photo Library

You will modify the **Add New Cat** screen to match the design shown in the app tour. Follow these steps:

1. In your app playground, switch to the `AddNewCatView` file. Add the following statements after the `AddNewCatView` declaration:

```
struct AddNewCatView: View {  
    @State private var catName = ""  
    @State private var catDescription = ""  
    @State private var toggleText = "Get Location"  
    @State private var toggleValue = false  
    @Environment(\.dismiss) private var dismiss
```

These statements declare the state variables that will be used by the text fields and switch that you will add in the next step.

2. In the navigation stack, replace the text view with the following code:

```
NavigationStack {  
    ScrollView {  
        VStack {  
            RoundedRectangle(cornerRadius: 8)  
            Toggle(toggleText, isOn: $toggleValue)  
            TextField("Cat name", text: $catName)  
            TextField("Cat description", text: $catDescription)  
            Image(systemName: "cat.fill")  
        }  
    }  
    .navigationTitle("Add New Cat")
```

`RoundedRectangle(cornerRadius: 8)` is a placeholder for the custom view showing star ratings, which will be added later in *Chapter 10, Custom Views*.

`Toggle(toggleText, isOn: $toggleValue)` displays a switch. The contents of `toggleText` appear on the left side of the switch and the value of the switch is stored in `toggleValue`.

The first text field will display **Cat name** and store the cat's name in `catName`.

The second text field will display **Cat description** and store the cat's description in `catDescription`.

The image view is configured to display a cat symbol.

3. In App Preview, tap the **AddNewCatView** button and verify the **Add New Cat** screen appears as follows:

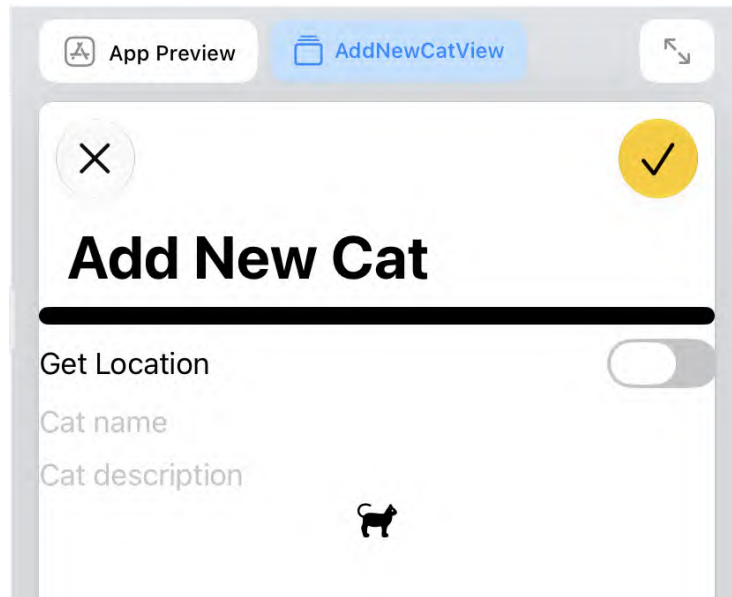


Figure 5.6: App Preview displaying the Add New Cat screen

All the user interface elements have been added, and you will use modifiers to improve their appearance.

4. Modify the contents of the `VStack` as follows:

```
VStack(alignment: .center) {
    RoundedRectangle(cornerRadius: 8)
```

```
        .frame(width: 252, height: 44)
        .foregroundColor(.blue)
        Toggle(toggleText, isOn: $toggleValue)
        TextField("Cat name", text: $catName)
        .font(.largeTitle)
        TextField("Cat description", text: $catDescription,
            axis: .vertical)
        .font(.title2)
        .multilineTextAlignment(.leading)
        .lineLimit(5...Int.max)
        Image(systemName: "cat.fill")
        .resizable()
        .scaledToFit()
        .frame(width: 300, height: 300)
        .foregroundColor(.orange)
    }
    .padding()
```

The rounded rectangle's dimensions are set to 252 points wide and 44 points high. Its color is set to blue.

The first text field's font size is set to `.largeTitle`, making it more prominent.

The second text field's axis is set to vertical. This, along with the `.lineLimit` modifier, makes it taller than a regular text field. The font size is increased slightly using the `.title2` modifier, and the text it contains is aligned to the left. The minimum height is set to five lines, and it can grow as necessary to accommodate the text entered by the user.

The image view's dimensions are set to 300 points wide and 300 points high and the color of the cat symbol is set to orange.

5. In App Preview, verify that the **Add New Cat** screen appears as follows:

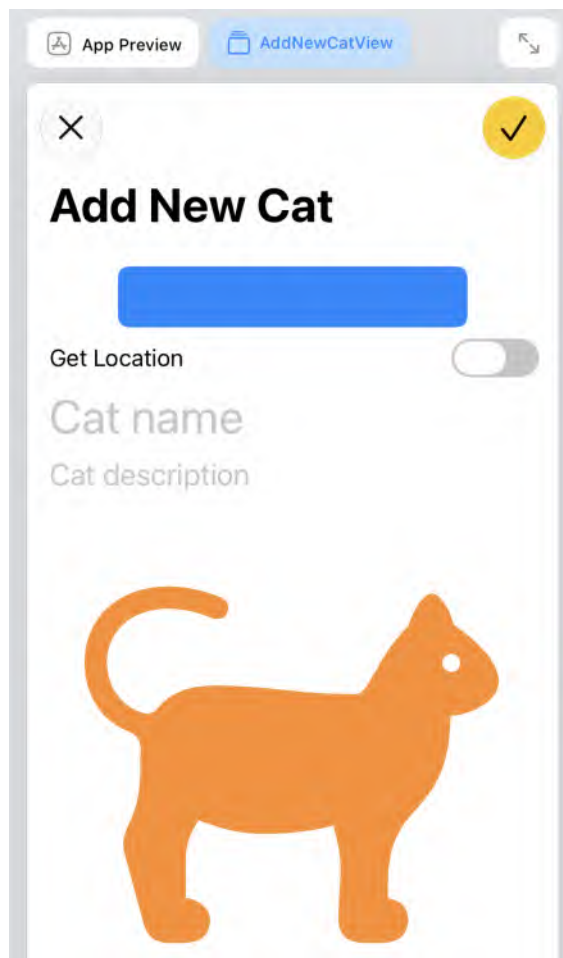


Figure 5.7: App Preview showing Add New Cat screen

Note that you can enter text in the text fields and interact with the switch, but you can't save a new cat to the list of cats just yet.

You have added all the required user interface elements to the **Add New Cat** screen. Great job! In the next section, you'll add user interface elements to the **Cat Detail** screen.

Modify the Cat Detail screen

Let's see what the **Cat Detail** screen looks like in the tour of the app shown in *Chapter 2, Set Up the User Interface*:

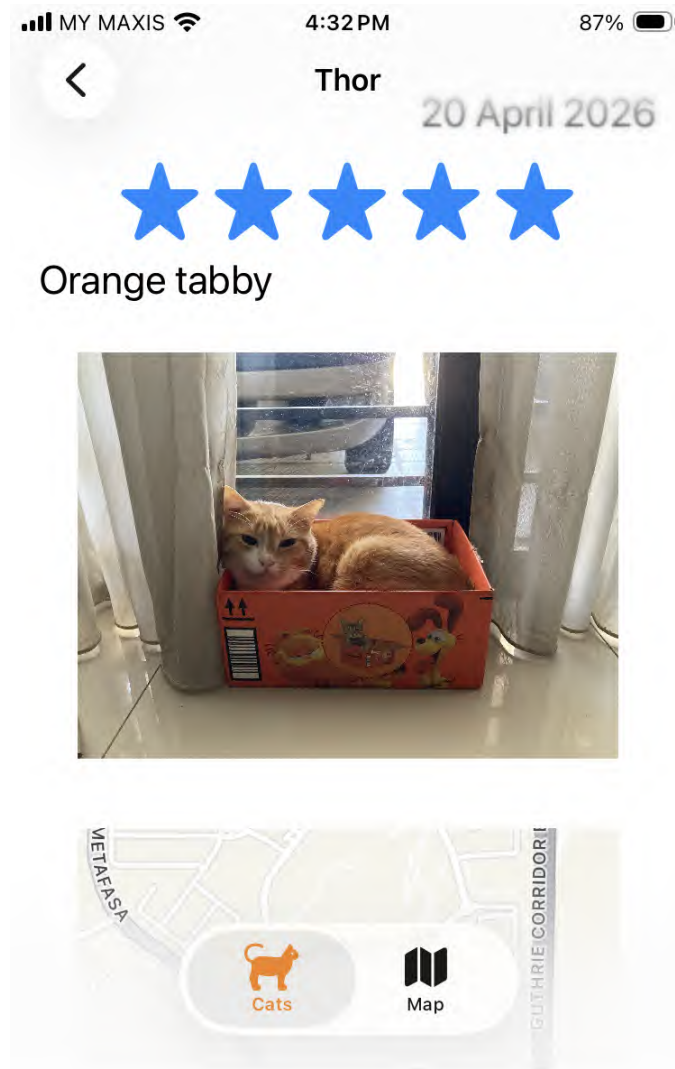


Figure 5.8: The Cat Detail screen for the completed CatsLog app playground

As you can see, the **Cat Detail** screen has the following elements:

- The title in the navigation bar showing the cat's name

- A text view showing the date
- A custom view showing star ratings
- A text view for the cat's description
- An image view for a photo that was previously selected from the photo library
- An image view showing a map location where the cat was found

Also, you need to scroll to view the entire screen. You will now modify it to match the design shown in the app tour. Follow these steps:

1. In the app playground, switch to the `CatDetailView` file and replace the text view with the following code:

```
var body: some View {  
    ScrollView {  
        VStack {  
            Text("8 April 2026")  
            RoundedRectangle(cornerRadius: 8)  
            Text("Cat description")  
            Image(systemName: "cat.fill")  
            Image(systemName: "map")  
        }  
    }  
    .navigationTitle("Cat name")  
}
```

The scroll view allows the contents of this screen to be scrolled.

The first text view will display a sample date.

`RoundedRectangle(cornerRadius: 8)` is a placeholder for the custom view showing star ratings, which will be added later in *Chapter 10, Custom Views*.

The second text view will display sample text representing the cat's description.

The first image view is configured to display a cat symbol.

The second image view is configured to display a map symbol.

The navigation title will display sample text representing the cat's name.

2. In App Preview, tap the **CatDetailView** button and verify the **Cat Detail** screen appears as follows:

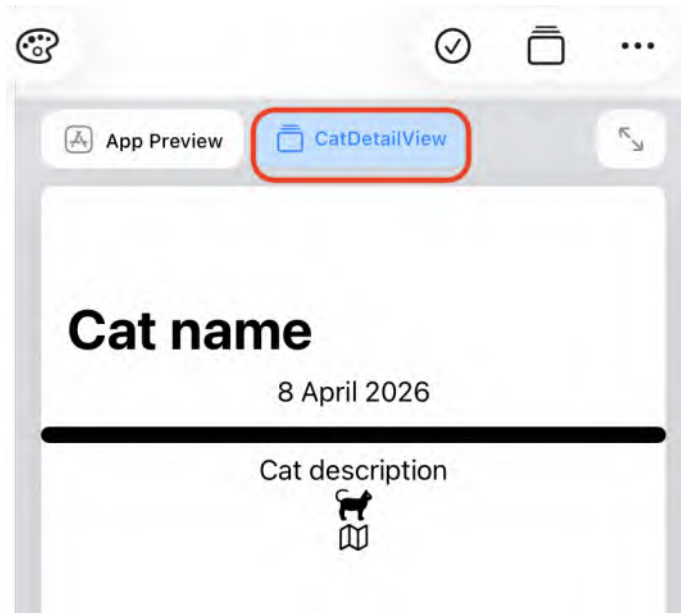


Figure 5.9: App Preview displaying the Cat Detail screen

All the user interface elements have been added, and you will use modifiers to improve their appearance.

3. Modify the contents of the scroll view as follows:

```
ScrollView {
  VStack {
    Text("8 April 2026")
      .font(.title2)
      .frame(maxWidth: .infinity, alignment: .trailing)
    RoundedRectangle(cornerRadius: 8)
      .frame(width: 252, height: 44)
      .foregroundColor(.blue)
    Text("Cat description")
      .font(.title2)
      .frame(maxWidth: .infinity, alignment: .leading)
    Image(systemName: "cat.fill")
      .resizable()
      .scaledToFit()
```

```
        .frame(width: 300, height: 300)
        .foregroundColor(.orange)
        Image(systemName: "map")
        .resizable()
        .scaledToFit()
        .frame(width: 300, height: 300)
        .foregroundColor(.green)
    }
    .padding()
}
.navigationTitle("Cat name")
```

The modifiers for the first text view have set its font to the `.title2` size, aligned the text to the right, and extended it to the full width of the screen.

The rounded rectangle's dimensions are set to 252 points wide and 44 points high. Its color is set to blue.

The modifiers for the second text view have set their font to the `.title2` size, aligned the text to the left, and extended it to the full width of the screen.

The first image view's dimensions are set to 300 points wide and 300 points high, the cat symbol it displays is scaled to fit within it, and the color of the cat symbol is set to orange.

The second image view's dimensions are set to 300 points wide and 300 points high, the map symbol it displays is scaled to fit within it, and the color of the map symbol is set to green.

The vertical stack also has some padding applied to it, and the navigation title has been set to sample text representing the cat's name.

4. In App Preview, verify that the **Cat Detail** screen appears as follows:

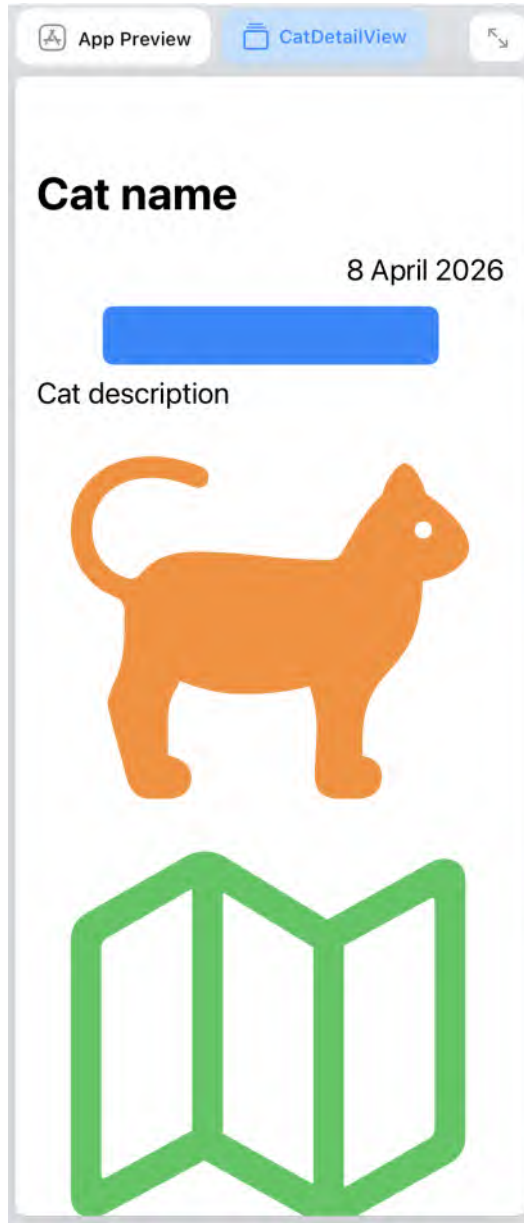


Figure 5.10: App Preview showing Cat Detail screen

Note that you can scroll down to see the entirety of this screen.

Excellent! You have modified all the screens for your app, and you're ready to add functionality to them in the next part of this book.

Summary

In this chapter, you modified the **Cats**, **Add New Cat**, and **Cat Detail** screens to match the design shown in the app tour. For the **Cats** screen, you created a new view representing an individual cat which contains an image view and two text views. You replaced the text views used previously in the **Cats** screen with this new view. You modified the **Add New Cat** screen by adding a custom view, a switch, two text fields, and an image view, so that you can enter the details of a new cat. The image view is configured to show a symbol of an orange cat. For the **Cat Detail** screen, you added a text view, labels, and image views to it, and configured the image views to show default images of an orange cat and a green map, respectively.

You now have more experience in how to add and configure multiple user interface elements, and how to determine their sizes and positions. This will be useful to ensure compatibility with different screen sizes and orientations. You should also be able to easily prototype the appearance and flow of your apps.

You're now finished with the design of your app. You can go through every screen that your app has and see what they look like, even though none of the screens have actual data in them. If this app were a house being built, it's as though you've built all the walls and floors, and the house is now ready to have the interior done. Great job!

This concludes *Part 1* of this book. In the next part, you'll begin to type in all the code required for your app to work. In the next chapter, you'll learn more about app state, and you'll also learn how to create sample data required to populate the **Cats** screen.

Part 2

Code

Welcome to *Part 2* of this book. With your user interface complete, you will now add code to implement your app's functionality. You will use a list view to display your data, and you will learn how to use an array as a data source. Next, you'll learn how to pass data from a view to the array used as a data source, and pass data from one view to another. You will also look at how to determine the device location and how to display a map containing annotations. After that, you will learn how to save and load app data using `SwiftData`. Then, you'll learn how to create custom views, use the photo library, and add search capability to your list. Finally, you'll use a navigation split view to make your app suitable for larger screens such as a Mac or iPad.

This part comprises the following chapters:

- *Chapter 6, Display Data in a List*
- *Chapter 7, Pass Data Between Screens*
- *Chapter 8, Core Location and MapKit*
- *Chapter 9, SwiftData*
- *Chapter 10, Custom Views*
- *Chapter 11, Photo Library*
- *Chapter 12, Search*

By the end of this part, you'll have completed the *CatLog* app. You'll have the experience of building a complete app from scratch, which will be useful as you build your own apps. Let's get started!

6

Display Data in a List

In the previous chapter, you created all the screens for your app and added all the required user interface elements, but your app does not display any data. As shown in the app tour in *Chapter 2, Set Up the User Interface*, it should display a list of cats.

In this chapter, you're going to implement a state variable to make the **Cats** screen display a list of cats. You'll start by learning about the requirements for the state variable that you will use. Then, you'll create a Swift class that can store cat instances. After that, you'll create sample cat data. This data will then be used to populate the list in the **Cats** screen.

By the end of this chapter, you'll have learned how to create state variables, how to create sample data, and how to populate SwiftUI lists.

The following topics will be covered in this chapter:

- State variables
- Create a class to represent a cat
- Create sample data
- Display data in a view

Technical requirements

You will continue working on the CatLog project that you modified in *Chapter 5, Modify App Screens*. The completed app playground for this chapter is in the `Chapter06.zip` file of the code bundle for this book. To access the repository link, follow the steps in the *Download the example code files* section in the *Preface*.

Check out the following video to see the code in action:

<https://packt.link/dbh08>

Let's start by examining the state variables required to store data on cats in the next section.

State variables

All apps have data, or **state**, that changes over time. For example, consider a fitness app that keeps track of the number of steps that you have walked. Before you start walking, the number of steps is 0. As you walk, the number of steps will increase. To store this value in your app, you create a container then can be assigned a value, and update that value as required. Such containers are called **variables**.

Note

For more information on variables, see this link: <https://docs.swift.org/swift-book/documentation/the-swift-programming-language/thebasics>.

Typically, a fitness app would be able to show you the total number of steps taken on the screen. Marking the variable used to store the number of steps with `@State` tells SwiftUI to monitor the variable and update the user interface when it changes.

Note

For more information on state, see this link: <https://developer.apple.com/tutorials/develop-in-swift/update-the-ui-with-state>.

An example of how to declare a variable used to keep track of the number of steps is as follows:

```
@State private var numberOfSteps = 0
```

This statement creates a variable named `numberOfSteps` that can store integer numbers and assigns 0 to it. The `private` keyword means that other SwiftUI views can't interfere with it.

Let's revisit the design of the **Cats** screen that you saw during the app tour in *Chapter 2, Set Up the User Interface*, which looks like this:

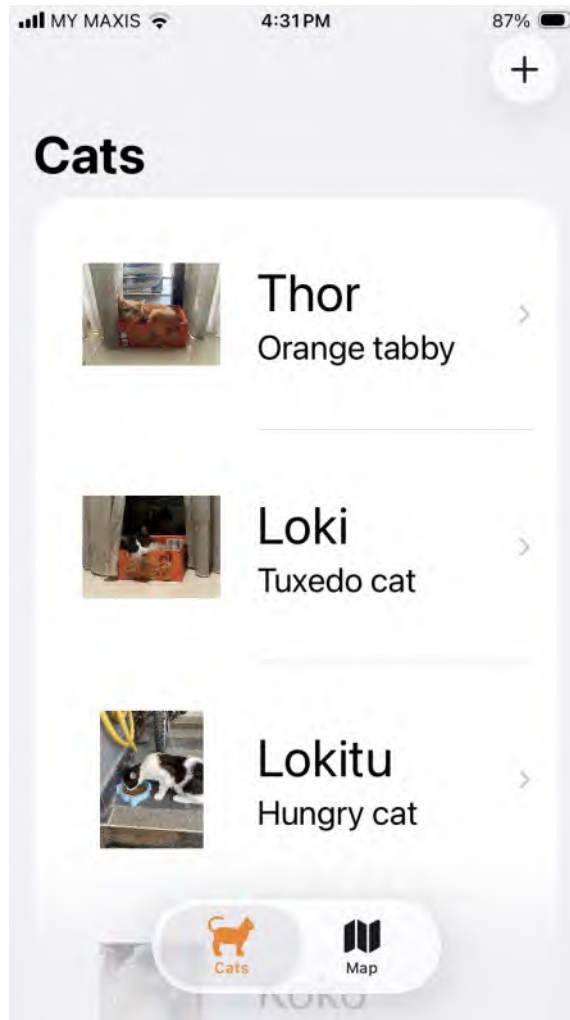


Figure 6.1: App Preview showing the Cats screen from the app tour

As you can see, each view in the list displays a picture, a name, and a description. At present, the App Preview for your app playground looks like this:

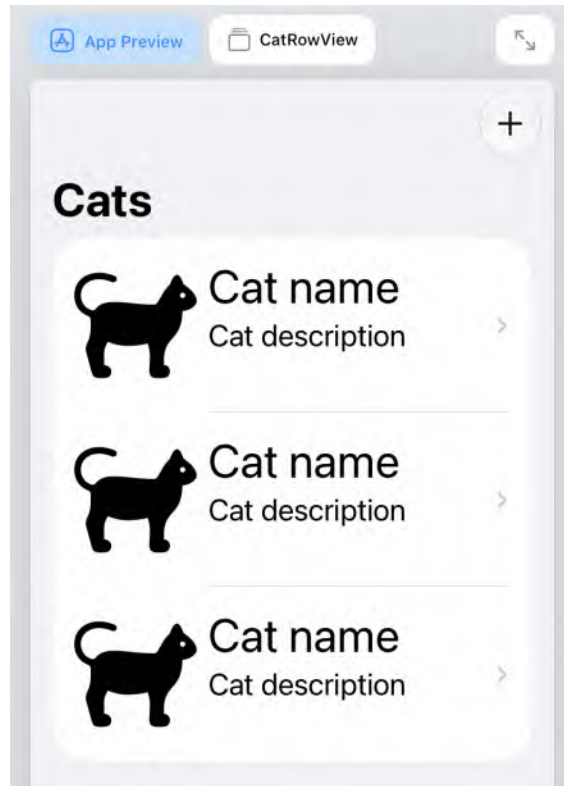


Figure 6.2: App Preview showing the Cats screen for your app playground

As you can see, all the views in the list currently display placeholders. You will need to create a state variable that will provide the data for your app.

For each individual cat in your app, you will need to store the following:

- The date when the cat's information was recorded
- A rating value
- The cat's name
- The cat's description
- An optional photo of the cat
- An optional geographical location

In *Chapter 3, Build the User Interface*, you used three text views to represent three individual cats. You used a sequence of characters enclosed in double quotes for each cat's name. These are known as **strings** in Swift.

Note

For more information on strings, see this link: <https://docs.swift.org/swift-book/documentation/the-swift-programming-language/thebasics>.

However, a `String` variable can only store strings, and you must be able to store data types other than `String`. To resolve this, you will create a **class** named `Cat` to store all the data required for an individual cat. Next, you will create a variable that returns sample data stored in an array of `Cat` instances. Finally, you will modify the `CatsView` class so that it can display the sample data.

Create a class to represent a cat

A class is a way to represent real-world objects in your app. It contains variables which describe its state, and functions that describe its behavior. Variables associated with a class are known as **properties**, and functions associated with a class are known as **methods**. **Instances** of a class will be used to represent individual cats in your app.

Note

For more information on classes, see this link: <https://docs.swift.org/swift-book/documentation/the-swift-programming-language/classesandstructures>.

To create a class that can represent a cat, you will add a new Swift file to your project, named `Cat`, and declare a `Cat` class that has the required properties for an individual cat. Follow these steps:

1. In your app playground, tap the Sidebar button and tap the New button to create a new Swift file.
2. Name the file `Cat`. Its contents will appear in the Editor area.
3. Add the following code after the `import` statement to declare and define the `Cat` class:

```
class Cat {  
    let date: Date
```

```
    let rating: Int
    let catName: String
    let catDescription: String
    let photo: Image?
    let latitude: Double?
    let longitude: Double?
}
```

Let's break this down:

- The date property is of type `Date` and will store the date the cat's information was recorded.
- The rating property is of type `Int` and will store the star rating of the cat.
- The `catName` property is of type `String` and will store the cat's name.
- The `catDescription` property is of type `String` and will store the cat's description.
- The photo property is of type `Image?` and will store a photo of the cat. The question mark after the type means that this is an **optional** property. Optional properties may or may not have a value. You use an optional property here because it is possible that you don't have a photo of the cat but still wish to record its details.
- The latitude and longitude properties are of type `Double?` and store the location where the cat was found. These are also optional properties because it is possible that you're not able to obtain the cat's location.

An error will appear because your class does not have an **initializer**, which ensures that instances of the class have valid values for all their properties before use.

4. Add the following code to implement an initializer after the longitude property:

```
init(rating: Int, catName: String, catDescription: String,
    photo: Image? = nil, latitude: Double? = nil,
    longitude: Double? = nil) {
    self.date = Date()
    self.rating = rating
    self.catName = catName
    self.catDescription = catDescription
    self.photo = photo
    self.latitude = latitude
```

```
        self.longitude = longitude
    }
```

Let's break this down:

```
    init(rating: Int, catName: String, catDescription: String,
        photo: Image? = nil, latitude: Double? = nil,
        longitude: Double? = nil) {
```

The initializer for the `Cat` class has arguments for an `Int` value, two `String` values, an optional `Image` value, and two optional `Double` values. The default value for all the optional values is `nil`, meaning it does not have a value.

```
        self.date = Date()
```

This statement creates a `Date` instance, which by default contains the current date, and assigns it to the `date` property when the `Cat` instance is created.

```
        self.rating = rating
        self.catName = catName
        self.catDescription = catDescription
        self.photo = photo
        self.latitude = latitude
        self.longitude = longitude
```

These statements assign the argument values to the corresponding properties of the `Cat` instance. Note the use of `self` to differentiate properties from arguments having the same name.

At this point, you have a class, `Cat`, that can store all the details of an individual cat. In the next section, you'll create a static property that returns an array containing sample `Cat` instances.

Create sample data

You can use an **extension** to provide additional capabilities to an existing class, and an **array** to store a list of `Cat` instances. An array stores values in an ordered list. You can access any value in an array by using the **array index**, which starts at 0.

You will now create an extension containing a static property that will return an array containing three sample cats.

In your app playground, switch to the Cat file, and type in the following after all other code in the file:

```
// Sample data
extension Cat {
    static let sampleCat = sampleCatData[0]
    static let sampleCatData = [
        Cat(rating: 5, catName: "Thor", catDescription: "Orange tabby",
            photo: Image(systemName: "cat")),
        Cat(rating: 4, catName: "Loki", catDescription: "Tuxedo cat",
            photo: Image(systemName: "cat.fill")),
        Cat(rating: 3, catName: "Lokitu", catDescription: "Hungry cat",
            photo: Image(systemName: "cat.fill"))
    ]
}
```

This extension contains a `sampleCatData` property that contains three `Cat` instances stored in an array, and a `sampleCat` property that returns the item at index 0 (the first item) from that array. The `let` keyword means that the contents of these properties may not be changed after they have been set. The `static` keyword means that a property is for the `Cat` type instead of its instances.

You have now completed the implementation of the `Cat` class. You have also added a static property named `sampleCatData` that contains three sample cats. In the next section, you'll modify the `CatsView` view to display the sample data from this property.

Display data in a view

So far in this chapter, you've implemented a static property containing three `Cat` instances. You will now modify the `CatsView` view to use that array to populate the list shown on the **Cats** screen. To do so, you will do the following:

- Modify the `CatRowView` view to display information contained in a `Cat` instance
- Modify the `CatsView` view to get sample data from the `sampleCatData` static property and assign it to a `cats` array
- Modify the list in the `CatsView` view to populate the list using data from the `cats` array

Follow these steps:

1. Switch to the `CatRowView` file, and declare a property after the `CatRowView` declaration to store a `Cat` instance:

```
struct CatRowView: View {  
    let cat: Cat
```

Don't worry about the error in the `#Preview` block; it will be addressed in the next step.

2. Modify the code in the `#Preview` block as shown to assign a `Cat` instance to the `CatRowView` view:

```
#Preview {  
    CatRowView(cat: Cat.sampleCat)  
}
```

3. Modify the code in the `HStack` view to display the photo, name and description of the `Cat` instance:

```
HStack {  
    if let catPhoto = cat.photo {  
        catPhoto  
        .resizable()  
        .scaledToFit()  
        .frame(width: 90, height: 90)  
    }  
    VStack {  
        Text(cat.catName)  
        .font(.largeTitle)  
        .frame(maxWidth: .infinity, alignment: .leading)  
        Text(cat.catDescription)  
        .lineLimit(2)
```

Since there is a possibility that the `photo` property of the `cat` instance does not have a value, the `if let catPhoto = cat.photo` statement will check to see if `photo` has a value, and if so, assigns that value to `catPhoto`, which then will be displayed on screen. Otherwise, no photo will be displayed.

App Preview will still not work properly as there are errors in the `CatsView` file. You will address these errors in the next step.

4. Switch to the `CatsView` file and assign `Cat.sampleCat` to all three `CatRowView` views:

```
NavigationLink(destination: CatDetailView()) {  
    CatRowView(cat: Cat.sampleCat)  
}  
NavigationLink(destination: CatDetailView()) {  
    CatRowView(cat: Cat.sampleCat)  
}  
NavigationLink(destination: CatDetailView()) {  
    CatRowView(cat: Cat.sampleCat)  
}
```

There should be no more errors at this point, and App Preview will be working once more.

5. In App Preview, verify that the photo, name and description from the `sampleCat` static property are displayed in the **Cats** screen:

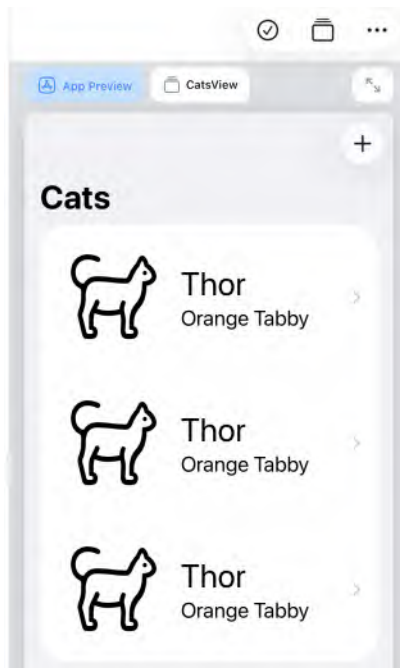


Figure 6.3: App Preview showing the *Cats* screen

6. Declare a state variable after the `showAddCatView` declaration to store an array of `Cat` instances and assign the contents of `Cat.sampleCatData` to it:

```
@State private var showAddCatView = false
@State private var cats = Cat.sampleCatData
```

`@State` is used here as you will need to add and remove cats to this array, which you will learn how to do in later chapters.

7. Modify the contents of the list view as follows:

```
List {
    ForEach(cats, id: \.catName) { cat in
        NavigationLink(destination: CatDetailView()) {
            CatRowView(cat: cat)
        }
    }
}
```

The `ForEach` statement will access each element of the `sampleCatData` array in turn. The accessed element will then be assigned to `cat`, which is then passed as an argument to `CatRowView` for display in the list.

Note

The code `id: \.catName` means that the cat's name is used to uniquely identify each cat, so no two cats in this array can have the same name. This will be addressed in *Chapter 9, SwiftData*.

8. In App Preview, verify that the details of each cat in the `sampleCatData` array are visible in the **Cats** screen:

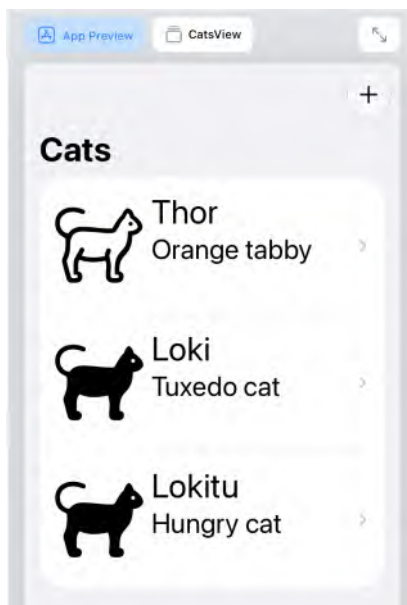


Figure 6.4: App Preview showing Cats screen populated by sampleCatData

Tapping a view in the list displays the **Cat Detail** screen, but this screen does not display data of the selected cat yet, and you can't add or remove cats from the list in the **Cats** screen. This will be covered in the next chapter.

Summary

In this chapter, you implemented a state variable for the **Cats** screen and used it to display a list of cats. You learned about state variables, created a `Cat` class that can be used to store individual cat details, and created a static property returning sample data. You then modified the `CatRowView` view to display data from a cat instance, then used the static property to assign sample data to a `cats` array, which in turn is used to populate the list in the **Cats** screen.

You now know how to create state variables, how to create sample data, and how to display it in a SwiftUI list view. This will be useful should you wish to create your own apps that use SwiftUI list views.

In the next chapter, you'll learn how to add and remove cats from the **Cats** screen. You'll also learn how to pass data between different screens.

7

Pass Data Between Screens

In the previous chapter, you modified the `CatsView` view to display cats from an array containing sample data in a list.

In this chapter, you'll learn how to pass data from one view to another. You'll start by adding code to pass data from the **Add New Cat** screen to the **Cats** screen. Next, you'll learn how to remove a cat while you're on the **Cats** screen. Finally, you'll pass data from the **Cats** screen to the **Cat Detail** screen.

By the end of this chapter, you'll have learned how to pass data between views using state and bindings. This will enable you to easily pass data between views in your own apps.

The following topics will be covered in this chapter:

- Pass data from the **Add New Cat** screen to the **Cats** screen
- Remove a cat from the **Cats** screen
- Pass data from the **Cats** screen to the **Cat Detail** screen

Technical requirements

You will continue working on the CatLog app playground that you modified in the previous chapter.

The completed app playground for this chapter is in the `Chapter07.zip` file of the code bundle for this book. To access the repository link, follow the steps in the *Download the example code files* section in the *Preface*.

Check out the following video to see the code in action:

<https://packt.link/YJivs>

Let's begin by learning how data is passed between the **Add New Cat** screen and the **Cats** screen in the next section.

Pass data from the Add New Cat screen to the Cats screen

As shown in the app tour in *Chapter 2, Set Up the User Interface*, the **Add New Cat** screen allows the user to add a new cat to the app. To do so, the user will tap the + button at the top-right corner of the **Cats** screen to display the **Add New Cat** screen. The user will then enter the details for the new cat. Tapping the Save button will dismiss the **Add New Cat** screen, and the photo, name and description of the newly added cat will appear in the **Cats** screen.

For this to work, you'll create a new `Cat` instance using the information the user entered and assign it to a variable. Then, you will add this variable to the `cats` array in the `CatsView` view, and it will appear in the **Cats** screen once the **Add New Cat** screen is dismissed. Follow these steps:

1. Open your app playground and switch to the `AddNewCatView` file.
2. Add a statement after the `AddNewCatView` structure declaration to create a new `cats` array that will be linked to the `cats` array in the `CatsView` view:

```
struct AddNewCatView: View {  
    @Binding var cats: [Cat]
```

The `@Binding` keyword tells SwiftUI that the `cats` array in `AddNewCatView` will be connected to another array in your app playground. Note that errors will appear; it will be resolved at the final step of this section.

Note

`@Binding` creates a two-way connection between a property that stores data, and a view that displays and changes the data. For more information, see this link:

<https://developer.apple.com/documentation/swiftui/binding>.

3. Modify the #Preview block as shown to pass a constant of the sample data to the AddNewCatView view:

```
#Preview {
  AddNewCatView(cats: .constant(Cat.sampleCatData))
}
```

Even though the sample data is not needed to display the AddNewCatView view in App Preview, omitting this will result in an error because the AddNewCatView view now requires a value to be assigned to the cats array:

4. Modify the code in the Save button to create a new Cat instance, insert it as the first element in the cats array, and dismiss the **Add New Cats** screen:

```
Button("Save", systemImage: "checkmark") {
  let newCat = Cat(rating: 3, catName: catName,
    catDescription: catDescription, photo:
    Image(systemName: "cat.fill"))
  cats.insert(newCat, at: 0)
  dismiss()
}
```

The first statement creates a new Cat instance. The rating value is set to 3, both catName and catDescription will be set to the values the user has typed into the corresponding text fields, and the default image will be set to cat.fill.

The second statement inserts the newCat instance as the first item in the cats array, making it appear at the top of the list in the **Cats** screen.

The final statement dismisses the **Add New Cat** screen.

5. Add a modifier to disable this button if catName or catDescription is empty:

```
Button("Save", systemImage: "checkmark") {
  let newCat = Cat(rating: 3, catName: catName,
    catDescription: catDescription, photo:
    Image(systemName: "cat.fill"))
  cats.insert(newCat, at: 0)
  dismiss()
}
.disabled(catName.isEmpty || catDescription.isEmpty)
```

Note

The `||` operator is the logical OR operator. To learn more about operators in Swift, see this link: <https://docs.swift.org/swift-book/documentation/the-swift-programming-language/basicoperators>.

6. Switch to the `CatsView` file. Modify the code in the `.sheet(isPresented:)` modifier to pass the `cats` array as a binding to the `AddNewCatView` view:

```
.sheet(isPresented: $showAddCatView) {  
    AddNewCatView(cats: $cats)  
}
```

This links the `cats` array in the `AddNewCatView` view to the `cats` array in the `CatsView` view.

7. In the App Preview, tap the `+` button. The **Add New Cat** screen will appear:

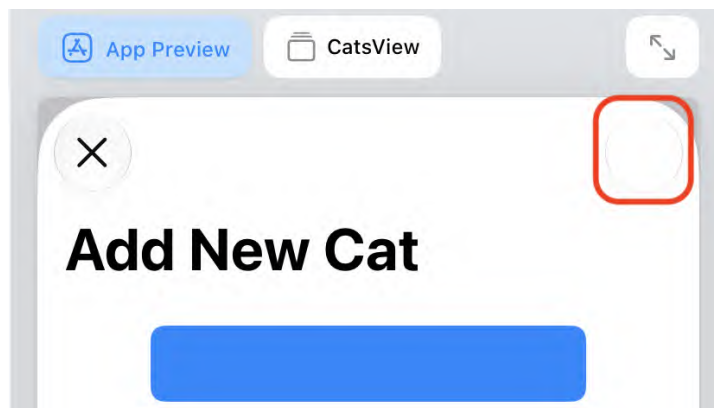


Figure 7.1: App Preview showing the Add New Cat screen with the Save button greyed out

Note that the Save button is greyed out.

8. Enter a cat's name (must be different from other cat names in the app) and description in the provided text fields. The Save button is now enabled. Tap it to save the new cat's details and return to the **Cats** screen:

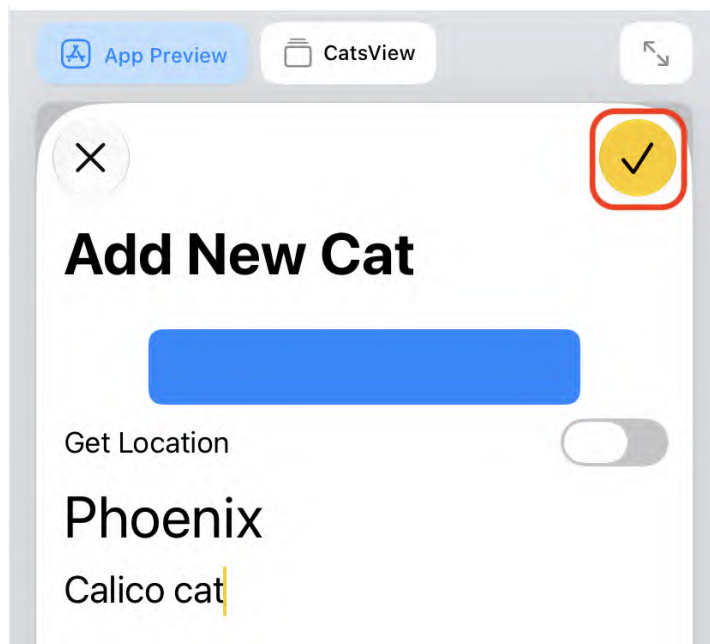


Figure 7.2: App Preview showing the Add New Cat screen with the Save button enabled

Note

At present, the `catName` property uniquely identifies each cat in the app. This means that the user can't add two cats with the same name to the app. This will be fixed in *Chapter 9, SwiftData*.

9. Verify that the new cat's details appear in the **Cats** screen:

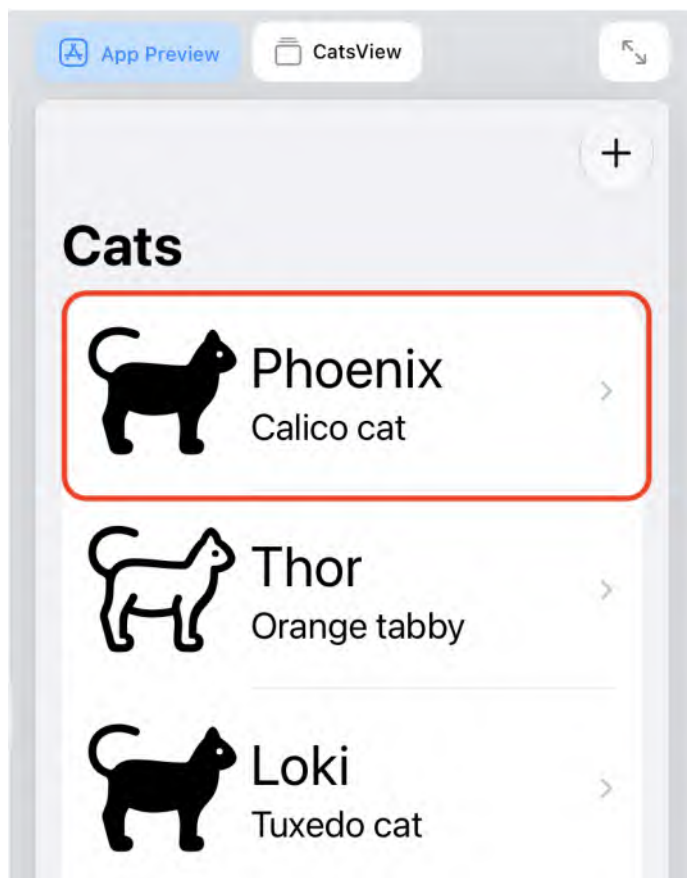


Figure 7.3: App Preview showing the Cats screen with a new cat added

Excellent! You have successfully implemented the **Add New Cat** screen and can now add new cats to the app, which will then appear on the **Cats** screen.

Note

Note that the changes are not permanent and will be reset as you navigate between different files in your app playground. You will learn how to store app data in *Chapter 9*, *SwiftData*.

In the next section, you'll implement the code that will let you remove a cat from the **Cats** screen.

Remove a cat from the Cats screen

To remove a view from a list of views, use the `.onDelete` modifier. This modifier adds a swipe-to-delete action to the list. You will add code to this modifier to remove the specified cat from the `cats` array. Follow these steps:

1. In the app playground, switch to the `CatsView` file, and add this modifier to the list:

```
List {
  ForEach(cats, id: \.catName) { cat in
    NavigationLink(destination: CatDetailView()) {
      CatRowView(cat: cat)
    }
  }
  .onDelete { indexSet in
    cats.remove(atOffsets: indexSet)
  }
}
```

All the views in a list have an index value. The first view in the list is at index 0, the second is at index 1, and so on. This corresponds to the index of the items in the `cats` array. The first item in the `cats` array is at index 0, the second is at index 1, and so on. When you swipe-to-delete on a view in the list, the index of that view is stored in `indexSet`. For every index in `indexSet`, the corresponding item at the same index in the `cats` array will be removed.

Imagine the user swiped on the second view in the list shown on the **Cats** screen. This view's index value is 1. As a result, the item at index 1 in the `cats` array will be removed.

2. In the app playground, swipe left on any view to reveal a **Delete** button:

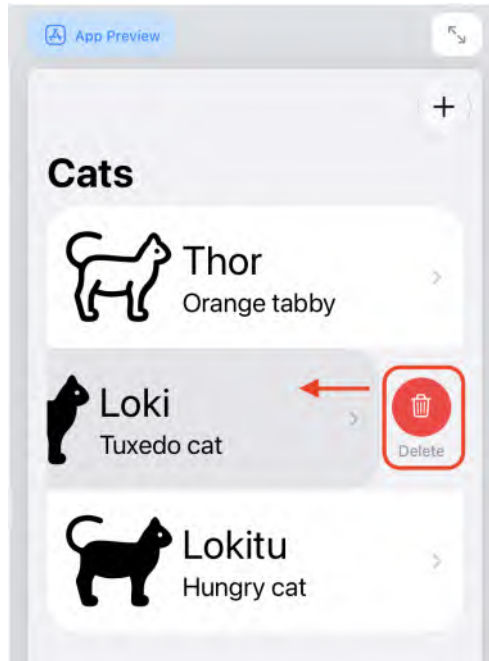


Figure 7.4: App Preview showing the Delete button on the Cats screen

3. Tap the **Delete** button, and the view will be removed from the list:

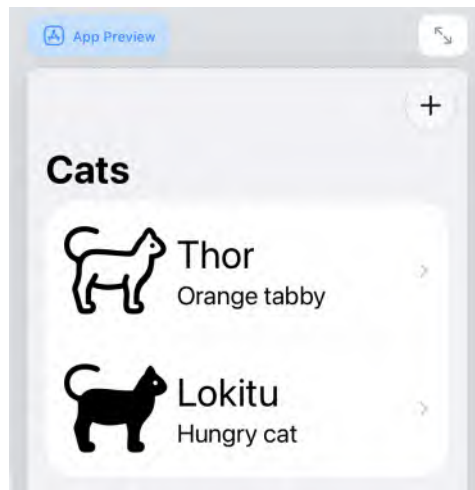


Figure 7.5: App Preview showing the updated list

You have successfully implemented a way to remove a view from a list! Awesome! In the next section, you'll learn how to pass data from the **Cats** screen to the **Cat Detail** screen when you tap a view in the list.

Pass data from the Cats screen to the Cat Detail screen

As shown in the app tour in *Chapter 2, Set Up the User Interface*, the **Cat Detail** screen allows the user to view the details of a cat when a view on the **Cat List** screen is tapped. For this to work, you'll add a property to the `CatDetailView` view that will be used to store a `Cat` instance. This property will then provide data to be displayed in the navigation bar and text views. After that, you will add code to the `CatsView` view to get the `Cat` instance corresponding to the view that was tapped and pass this instance to the `CatsDetailView` view. Follow these steps:

1. In the app playground, switch to the `CatDetailView` file and declare a new property named `selectedCat` after the `CatDetailView` structure declaration as shown:

```
struct CatDetailView: View {  
    var selectedCat: Cat
```

Note that errors will appear; they will be resolved at the final step of this section.

2. Modify the `#Preview` block as shown to pass a single cat's sample data to the `CatDetailView` view:

```
#Preview {  
    NavigationStack {  
        CatDetailView(selectedCat: Cat.sampleCat)  
    }  
}
```

3. Modify the first text view in the `VStack` to display the date when the cat was added to the app:

```
VStack {  
    Text(selectedCat.date, style: .date)  
        .font(.title2)  
        .frame(maxWidth: .infinity, alignment: .trailing)
```

4. Modify the second text view in the `VStack` to display the cat's description:

```
        .foregroundColor(.blue)
Text(selectedCat.catDescription)
        .font(.title2)
        .frame(maxWidth: .infinity, alignment: .leading)
```

5. Configure the `.navigationTitle` modifier to display the cat's name:

```
        .padding()
    }
    .navigationTitle(selectedCat.catName)
}
```

6. Switch to the `CatsView` file. Modify the navigation link to pass the cat corresponding to the index of the tapped view to the `CatDetailView` view:

```
NavigationLink(destination: CatDetailView(selectedCat: cat)) {
    CatCellView(cat: cat)
}
```

7. In App Preview, tap a cat in the **Cats** screen. The name, date and description for that cat will appear in the **Cat Detail** screen:

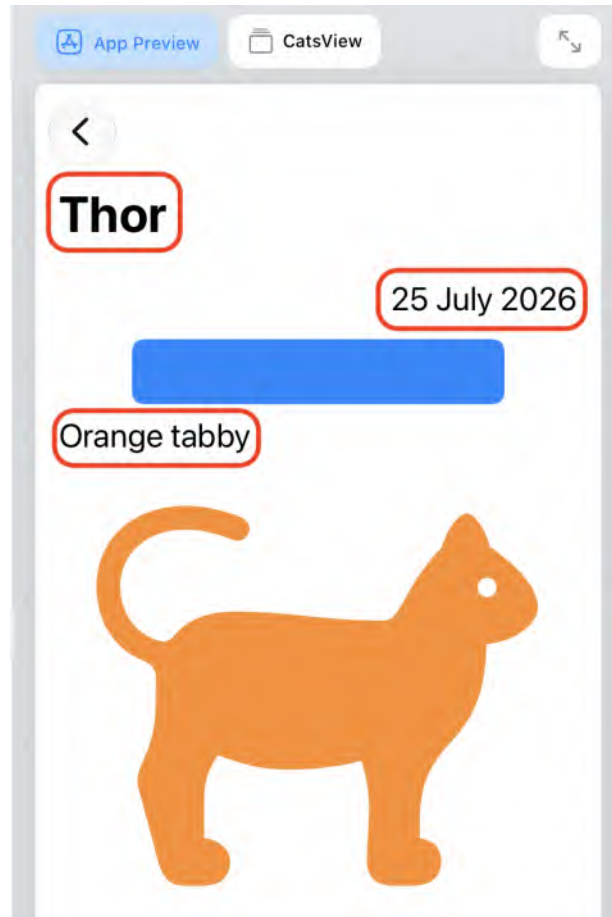


Figure 7.6: App preview showing Cat Detail screen

You have successfully implemented the **Cat Detail** screen. Great job! Now when the user taps a cat on the **Cats** screen, the details of that cat will be displayed in the **Cat Detail** screen.

Summary

In this chapter, you learned how to pass data from one view to another. You added code to pass data from the **Add New Cat** screen to the **Cats** screen. Next, you learned how to remove cats from the list in the **Cats** screen. After that, you learned how to pass data from the **Cats** screen to the **Cat Detail** screen.

You now know how to pass data between views using state and bindings, and this will enable you to easily pass data between views in your own apps. Cool!

In the next chapter, you will modify the **Map** screen to display cat locations using marker views. You'll also configure the markers to display the **Cat Detail** screen when they are tapped.

8

Core Location and MapKit

In the previous chapter, you learned how to pass data from the **Add New Cat** screen to the **Cats** screen, and from the **Cats** screen to the **Cat Detail** screen.

In this chapter, you'll learn how to get your device location using Apple's **Core Location** framework, and how to set map regions, display map annotations, and create map snapshots using Apple's **MapKit** framework. This will come in handy if you're planning to build apps that use maps, like *Apple Maps* or *Waze*.

First, you'll modify the **Add New Cat** screen so that the user can add the location where a new cat was found. Next, you'll configure the **Map** screen to display a map region centered on your location. After that, you'll modify the **Map** screen to display a map annotation for each cat within the map region that you set earlier. You'll configure the map annotation to display the **Cat Detail** screen when tapped. Finally, you'll modify the **Cat Detail** screen to display a map snapshot showing the location where the cat was found.

By the end of this chapter, you'll have learned how to use Core Location to get your device location and how to use MapKit to specify a map region, add annotations to a map, and create map snapshots.

The following topics will be covered in this chapter:

- Get your device location using the Core Location framework
- Display annotations on the **Map** screen
- Display a map snapshot on the Journal Entry Detail screen

Technical requirements

You will continue working on the CatLog app playground that you modified in the previous chapter.

The completed app playground for this chapter can be found in the `Chapter08.zip` file of the code bundle for this book. To access the repository link, follow the steps in the *Download the example code files* section in the *Preface*.

Check out the following video to see the code in action:

<https://packt.link/pyt6x>

In the next section, you'll learn how the Core Location framework is used to get your device location.

Get your device location using the Core Location framework

Every Apple device has multiple means of determining its location, including Wi-Fi, GPS, Bluetooth, magnetometer, barometer, and cellular hardware. Apple created the Core Location framework to gather location data using all available components on an iOS device.

Note

To learn more about Core Location, see <https://developer.apple.com/documentation/corelocation>.

To configure your app to use Core Location, you will need to create an instance of `CLLocationManager`, which is used to configure, start, and stop location services. Next, you will create an instance of `CLLocationUpdate`, a structure that contains location information delivered by the Core Location framework. Calling the `CLLocationUpdate` type's `liveUpdates` method tells Core Location to start delivering location updates containing the user's location, authorization status, and location availability.

Note

To learn more about the `CLLocationManager` class, see https://developer.apple.com/documentation/corelocation/configuring_your_app_to_use_location_services.

You can watch Apple's WWDC 2023 video on streamlined location updates here: <https://developer.apple.com/videos/play/wwdc2023/10180/>.

Since location information is considered sensitive user data, you'll also need to obtain authorization to use location services. You can also check the authorization status of location updates delivered by the `CLLocationUpdate` type's `liveUpdates` method.

Note

To learn more about requesting authorization to use location services, see https://developer.apple.com/documentation/corelocation/requesting_authorization_to_use_location_services.

You can watch Apple's WWDC 2024 video titled *What's new in location authorization* here: <https://developer.apple.com/videos/play/wwdc2024/10212/>.

In the next section, you'll modify the `AddNewCatView` view so that the user can assign a location when they add a new cat.

Modify the `AddNewCatView` view

At present, the **Add New Cat** screen has a **Get Location** switch, but it does not work yet. You'll modify it to add your location to the `Cat` instance when the switch is on. Follow these steps:

1. Open your app playground and open the `AddNewCatView` file. Add an `import` statement after the `import SwiftUI` statement to import the Core Location framework:

```
import SwiftUI
import CoreLocation
```

2. Add properties to store an instance of the `CLLocationManager` class, an asynchronous task that will manage location updates, and the current device location after all other property declarations:

```
@State private var toggleValue = false
private let locationManager = CLLocationManager()
@State private var locationTask: Task<Void, Error>?
@State private var currentLocation: CLLocation?
```

3. You will make the app display an alert if the device location cannot be determined. Add properties that will be used to display an alert showing the reason why the device location is not found:

```
@State private var toggleValue = false
private let locationManager = CLLocationManager()
@State private var locationTask: Task<Void, Error>?
@State private var currentLocation: CLLocation?
@State private var isShowingLocationNotFound = false
@State private var locationNotFoundReason = ""
```

4. Implement a method to determine the user's location before the closing curly brace:

```
private func fetchUserLocation() {
    locationManager.requestWhenInUseAuthorization()
    locationTask = Task {
        for try await update in
            CLLocationUpdate.liveUpdates() {
                if let location = update.location {
                    updateCurrentLocation(location)
                } else if update.authorizationDenied {
                    locationNotFoundReason = "Authorization Denied"
                    isShowingLocationNotFound = true
                } else if update.locationUnavailable {
                    locationNotFoundReason = "Location Unavailable"
                    isShowingLocationNotFound = true
                }
            }
        }
    }
}
```

Let's break this down:

```
locationManager.requestWhenInUseAuthorization()
```

This statement asks the user for permission to use their location information.

```
locationTask = Task {
```

This statement assigns a block of code that will automatically run in the background to `locationTask`. This block of code will continuously obtain location updates until it is canceled.

Note

For more information about Task, see this link:

<https://developer.apple.com/documentation/swift/task>

For more information about concurrency in Swift, see this link:

<https://docs.swift.org/swift-book/documentation/the-swift-programming-language/concurrency>

```
for try await update in  
    CLLocationUpdate.liveUpdates() {
```

This statement gets the updates provided by `CLLocationUpdate.liveUpdates()` and assigns each to an update instance.

```
    if let location = update.location {  
        updateCurrentLocation(location)  
    }
```

This statement gets the user location from the update instance and calls the `updateCurrentLocation` method.

```
    else if update.authorizationDenied {  
        locationNotFoundReason = "Authorization Denied"  
        isShowingLocationNotFound = true  
    }
```

These statements will set `locationNotFoundReason` to `Authorization Denied` and `isShowingLocationNotFound` to `true` if the user did not give authorization to use their private data.

```
else if update.locationUnavailable {
    locationNotFoundReason = "Location Unavailable"
    isShowingLocationNotFound = true
}
```

These statements will set `locationNotFoundReason` to `Location Unavailable` and `isShowingLocationNotFound` to `true` if the user's location is not available.

You will see error messages because the methods called by `fetchUserLocation()` have not been implemented yet.

5. Implement the `updateCurrentLocation(_ : )` method called by the `fetchUserLocation()` method before the closing curly brace:

```
private func updateCurrentLocation(_ location: CLLocation) {
    let interval = location.timestamp.timeIntervalSinceNow
    if abs(interval) < 30 {
        locationTask?.cancel()
        toggleText = "Done"
        let lat = location.coordinate.latitude
        let long = location.coordinate.longitude
        currentLocation = CLLocation(latitude: lat,
                                     longitude: long)
    }
}
```

This method first gets the timestamp of location. This location may be an old, cached location, and not actually the current location, so the timestamp is compared with the current date and time. If the duration is less than 30 seconds, this indicates that the user's location is current. In this case, the block of code assigned to `locationTask` will stop executing, `toggleText` is set to `Done` and the `currentLocation` property is set to this location.

6. In the VStack, add an `.onChange` modifier to the Toggle view:

```
Toggle(toggleText, isOn: $toggleValue)
    .onChange(of: toggleValue) {
        if toggleValue {
            toggleText = "Getting location..."
            fetchUserLocation()
        } else {
            currentLocation = nil
            toggleText = "Get Location"
            locationTask?.cancel()
        }
    }
```

If the switch is on, `toggleText` is set to `Getting location...` and the `fetchUserLocation()` method is called. If the switch is off, `currentLocation` will be set to `nil`, `toggleText` is reset to `Get location`, and the block of code assigned to `locationTask` will stop executing.

7. Add an `.alert` modifier after the `.onChange` modifier to the Toggle view:

```
Toggle(toggleText, isOn: $toggleValue)
    .onChange(of: toggleValue) {
        if toggleValue {
            toggleText = "Getting location..."
            fetchUserLocation()
        } else {
            currentLocation = nil
            toggleText = "Get Location"
            locationTask?.cancel()
        }
    }
    .alert(locationNotFoundReason,
        isPresented: $isShowingLocationNotFound) {
        Button("OK") {
            toggleValue = false
            isShowingLocationNotFound = false
        }
    }
```

This will display an alert with the appropriate message if the user has denied authorization to use location services or if the device location was not found. Tapping the **OK** button will dismiss it.

8. Modify the Save button's method to add location information to the Cat instance:

```
ToolBarItem(placement: .confirmationAction) {  
    Button("Save", systemImage: "checkmark") {  
        let newCat = Cat(rating: 3, catName: catName,  
            catDescription: catDescription, photo:  
            Image(systemName: "cat.fill"), latitude:  
                currentLocation?.coordinate.latitude, longitude:  
                currentLocation?.coordinate.longitude)  
        cats.insert(newCat, at: 0)  
        dismiss()  
    }  
    .disabled(catName.isEmpty || catDescription.isEmpty)  
}
```

9. In App Preview, tap the + button to display the **Add New Cat** screen. Enter a cat's name and description, and turn the **Get Location** switch on:

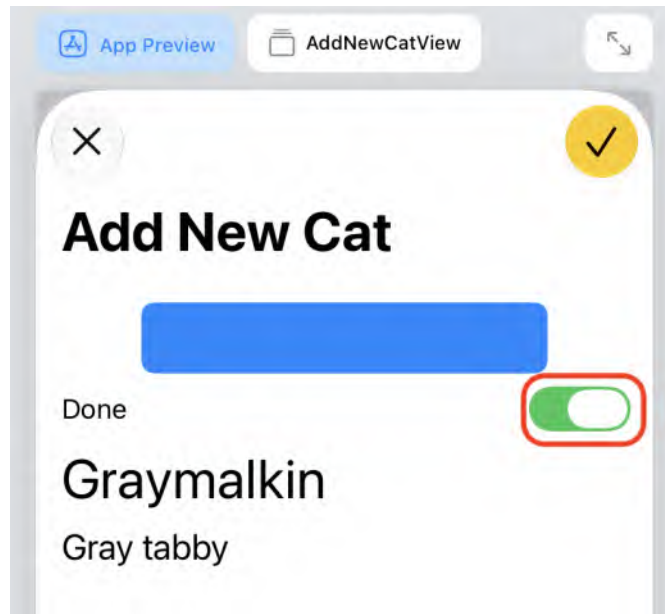


Figure 8.1: Add New Cat screen showing the Get Location switch

10. When prompted, tap the **Allow While Using App** button:

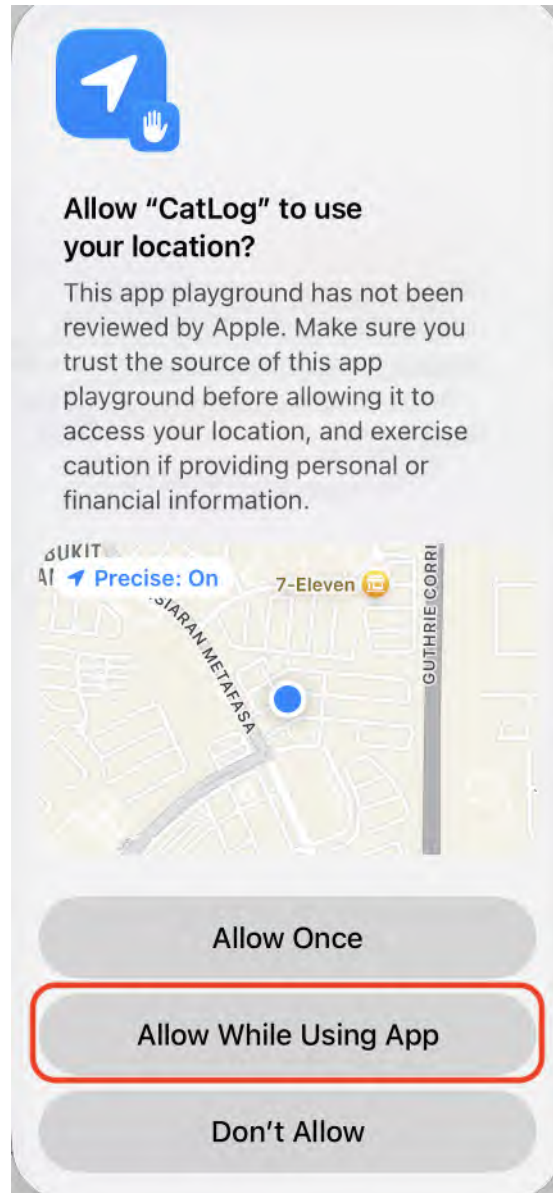


Figure 8.2: Alert showing Allow While Using App highlighted

Note that this alert will appear the first time you launch your app during this chapter, and the setting you picked will be used by default after that. Once the location has been determined, the label next to the **Get Location** switch will display **Done**.

11. Tap Save. You will be returned to the **Cats** screen.

At this point, the method for getting the device location has been assigned to the **Get Location** switch, and all the code required to add your location to the new Cat instance has been added. Great! In the next section, you'll configure the **Map** screen to display an annotation indicating a cat's location.

Display Annotations on the Map Screen

In *Chapter 4, Complete the User Interface*, you added a map view to the **Map** screen. A map view is one of the components of MapKit for SwiftUI.

Note

To learn more about MapKit for SwiftUI, see this link:<https://developer.apple.com/documentation/mapkit/mapkit-for-swiftui>.

When you build and run your app, you will see a map on the screen. Before you add annotations to it, you'll use Core Location to set the center point of the displayed map to your location. Follow these steps:

1. In the app playground, switch to the MapView file. Add a statement to import the Core Location framework after the existing import statement:

```
import SwiftUI
import MapKit
import CoreLocation
```

2. Add properties to store a map camera position and a string containing a value for the navigation bar after the MapView declaration:

```
struct MapView: View {
    @State private var position: MapCameraPosition =
        .region(MKCoordinateRegion())
    @State private var navTitle = "Map"
```

position will be used to set the center point of the displayed map to the device location later.

3. Add properties to store a `CLLocationManager` instance and an asynchronous task after the `navTitle` property:

```
@State private var position: MapCameraPosition =
    .region(MKCoordinateRegion())
@State private var navTitle = "Map"
private let locationManager = CLLocationManager()
@State private var locationTask: Task<Void, Error>?
```

4. You will make the app playground display an alert if the device location cannot be determined. Add properties that will be used to display an alert showing the reason why the device location is not found:

```
private let locationManager = CLLocationManager()
@State private var locationTask: Task<Void, Error>?
@State private var isShowingLocationNotFound = false
@State private var locationNotFoundReason = ""
```

5. Implement a method to determine the user's location before the closing curly brace:

```
private func fetchUserLocation() {
    locationManager.requestWhenInUseAuthorization()
    locationManager.desiredAccuracy = kCLLocationAccuracyKilometer
    navTitle = "Getting location..."
    locationTask = Task {
        for try await update in CLLocationUpdate.liveUpdates() {
            if let location = update.location {
                updateMapWithLocation(location)
            } else if update.authorizationDenied {
                locationNotFoundReason = "Authorization Denied"
                isShowingLocationNotFound = true
            } else if update.locationUnavailable {
                locationNotFoundReason = "Location Unavailable"
                isShowingLocationNotFound = true
            }
        }
    }
}
```

```
    }
}
```

This method is like the `fetchUserLocation()` method you implemented in the `AddNewCatView` view. First, it will ask for permission to use private user data and set the title of the **Map** screen to `Getting location...` Next, a block of code to continuously determine the user's location will be assigned to `locationTask`, and if the user's location is found, the map will be updated to show the user's location. Otherwise, an alert with an appropriate error message will be displayed.

Note that since the method called by `fetchUserLocation()` has not been implemented yet, you will see an error message.

6. Implement the missing method before the closing curly brace:

```
private func updateMapWithLocation(_ location: CLLocation) {
    let interval = location.timestamp.timeIntervalSinceNow
    if abs(interval) < 30 {
        locationTask?.cancel()
        let lat = location.coordinate.latitude
        let long = location.coordinate.longitude
        navTitle = "Map"
        let currentUserLocationRegion = MKCoordinateRegion(
            center: CLLocationCoordinate2D(latitude: lat,
            longitude: long), span: MKCoordinateSpan(
            latitudeDelta: 0.01, longitudeDelta: 0.01))
        position = .region(currentUserLocationRegion)
    }
}
```

This method is like the `updateCurrentLocation(location:)` method you implemented earlier in the `AddNewCatView` view. It will set the title of the **Map** screen to `Map`, create a map region centered on the user's location, and assign it as the region for the `position` property.

7. Modify the map view in the navigation stack as shown:

```
NavigationStack {
    Map(position: $position)
        .navigationTitle(navTitle)
        .onAppear() {
```

```

        fetchUserLocation()
    }
}

```

When the **Map** screen appears, the `.onAppear()` modifier will call the `fetchUserLocation()` method, which will set the value for `position`. `position` is then used to set the center of the map being displayed to the user location.

8. Add an `.alert` modifier after the `.onAppear` modifier to the map view:

```

NavigationStack {
    Map(position: $position)
        .navigationTitle(navTitle)
        .onAppear() {
            fetchUserLocation()
        }
        .alert(locationNotFoundReason, isPresented:
            $isShowingLocationNotFound) {
            Button("OK") {
                navTitle = "Map"
                isShowingLocationNotFound = false
            }
        }
}

```

This will display an alert with the appropriate message if the user has denied authorization to use location services or if the device location was not found. Tapping the **OK** button will dismiss it.

9. In App Preview, tap the **Map** tab button to display a map region centered on your location you selected:



Figure 8.3: App Preview showing a map centered on your location

The **Map** screen at present displays a map centered on your device location. Now that the map region has been set, you can determine which **Cat** instances are in this region based on their latitude and longitude properties. You will now modify the `MapView` view to get an array of **Cat** instances from the `Cat.sampleCatData` property and add it as annotations to the map view. Follow these steps:

1. Declare a state variable named `cats` after the `MapView` declaration and assign the contents of `Cat.sampleCatData` to it:

```
struct MapView: View {  
    @State private var cats = Cat.sampleCatData
```

This array will serve as the source of annotations which will be added to the map view.

Note

There is currently no connection between the **Cats** and **Map** screens. This means that any cats that you add using the **Add New Cat** screen will not appear on the **Map** screen. You will create a shared instance that will be used by both screens in the next chapter.

2. Modify the `updateMapWithLocation(_:)` method to add a new **Cat** instance to the `cats` array that has longitude and latitude set to the current device location:

```
let lat = location.coordinate.latitude  
let long = location.coordinate.longitude  
cats.append(Cat(rating: 4, catName: "Graymalkin",  
    catDescription:  
    "Gray tabby", photo: Image(systemName: "cat.fill"), latitude: lat,  
    longitude: long))  
navTitle = "Map"
```

This is necessary as the **Cat** instances in the sample data do not have latitude and longitude values set.

Note

At present, the `catName` property uniquely identifies each cat in the app. This means that the `Cat` instance added to the `cats` array here can't have the same name as any other instance already in the array. This will be fixed in *Chapter 9*, *SwiftData*.

3. Modify the map view as shown:

```
Map(position: $position) {  
    ForEach(cats, id: \.catName) { cat in  
        if let lat = cat.latitude, let long = cat.longitude {  
            Annotation(cat.catName, coordinate:  
                CLLocationCoordinate2D(latitude: lat, longitude:  
                    long)) {  
                Image(systemName: "cat.fill")  
            }  
        }  
    }  
}
```

Let's break this down:

```
ForEach(cats, id: \.catName) { cat in
```

This loop assigns each `Cat` instance in the `cats` array to `cat` in turn.

```
    if let lat = cat.latitude, let long = cat.longitude {
```

This `if` statement checks to see if the `Cat` instance has values in its `latitude` and `longitude` properties. If it does, the values are assigned to `lat` and `long` respectively, and the statements between the curly braces are executed.

```
        Annotation(cat.catName, coordinate: CLLocationCoordinate2D(  
            latitude: lat, longitude: long)) {  
            Image(systemName: "cat.fill")  
        }  
    }
```

This statement creates a map annotation containing a cat symbol and the cat's name. This annotation will appear at the center of the map view.

4. In the App Preview, tap the **Map** button. The newly created annotation will be displayed at the center of the map view:



Figure 8.4: App Preview showing the first cat's name in the Map screen

5. The **Map** screen can now display an annotation but tapping it doesn't do anything. To present the **Cat Detail** screen when the annotation is tapped, embed the image view in a navigation link as shown:

```
Annotation("cat.catName", coordinate:
CLLocationCoordinate2D(latitude: lat, longitude: long)) {
    NavigationLink(destination: CatDetailView(selectedCat:cat)) {
        Image(systemName: "cat.fill")
    }
    .buttonStyle(.glass)
    .foregroundColor(.red)
}
```

The `.buttonStyle(.glass)` modifier adds a nice effect around the image, and the `.foregroundColor(.red)` modifier sets the image color to red.

6. In the App Preview, tap the Map button, and tap the annotation:

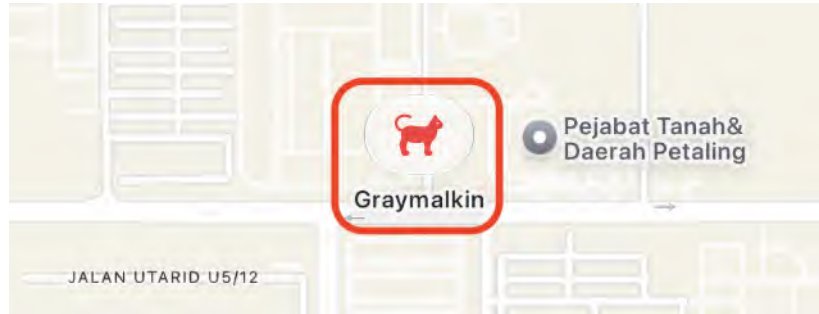


Figure 8.5: App Preview showing the annotation in the Map screen

7. The **Cat Detail** screen will appear showing the cat's details:



Figure 8.6: App Preview showing the Cat Detail screen

You have successfully displayed an annotation showing a cat's location in the **Map** screen, and passed data from the **Map** screen to the **Cat Detail** screen. Fantastic! In the next section, you'll configure the **Cat Detail** screen to display a map snapshot of a cat's location.

Display a map snapshot on the Cat Detail screen

The **Map** screen currently displays a single annotation representing a cat's location. When you tap the annotation on the **Map** screen, the cat's details are displayed on the **Cat Detail** screen, but the second image view on the **Cat Detail** screen currently displays a placeholder map image. You can capture a map region and convert it into an image using the `MKMapSnapshotter` class.

Note

For more information on the `MKMapSnapshotter` class, see <https://developer.apple.com/documentation/mapkit/mkmapsnapshotter>.

To configure the region and appearance of the map that is captured in the snapshot, an `MKMapSnapshotter.Options` object is used.

Note

For more information on the `MKMapSnapshotter.Options` object, see <https://developer.apple.com/documentation/mapkit/mkmapsnapshotter/options>.

You will replace the placeholder image with a map snapshot showing the cat's location. Follow these steps:

1. In the app playground, tap the `CatDetailView` file. Type the following after the `import` statement to import the MapKit framework:

```
import SwiftUI
import MapKit
```

2. Add the following property after the `selectedCat` property:

```
@State private var mapSnapshot: UIImage? = nil
```

The generated map snapshot image will be assigned to this property.

3. Add a method to generate the map snapshot before the closing curly brace:

```
private func getMapSnapshot() {
    guard let lat = selectedCat.latitude, let
```

```

        long = selectedCat.longitude else {
            return
        }
        let options = MKMapSnapshotter.Options()
        options.region = MKCoordinateRegion(center:
            CLLocationCoordinate2D(latitude: lat, longitude: long),
            span: MKCoordinateSpan(latitudeDelta: 0.01,
            longitudeDelta: 0.01))
        options.size = CGSize(width: 300, height: 300)
        options.preferredConfiguration = MKStandardMapConfiguration()
        let snapshotter = MKMapSnapshotter(options: options)
        snapshotter.start { snapshot, error in
            if let snapshot {
                mapSnapshot = snapshot.image
            } else if let error {
                print("snapshot error: \(error.localizedDescription)")
            }
        }
    }
}

```

This method checks to see whether `selectedCat` has values in its `latitude` and `longitude` properties. If it does, then an `MKMapSnapshotter.Options` object is created, configured, and assigned to an `MKMapSnapshotter` object. The `MKMapSnapshotter` object is then used to generate the map snapshot, which will be assigned to the `mapSnapshot` property.

4. Replace the second image view in the `VStack` with the code shown:

```

Image(systemName: "cat.fill")
    .resizable()
    .frame(width: 300, height: 300)
    .foregroundColor(.orange)
    .padding()
if let mapSnapshot {
    VStack {
        Image(uiImage: mapSnapshot)
            .resizable()
            .frame(width: 300, height: 300)
            .padding()
        Image(systemName: "cat.fill")
            .foregroundColor(.red)
    }
}

```

```
    }  
  }  
}
```

If `mapSnapshot` has a value, it will be converted into an image view containing the map snapshot. This image view will have a second image view containing a cat symbol overlaid on top of it using a `ZStack`.

5. Call the `getMapSnapshot()` method using an `.onAppear` modifier for the `VStack`:

```
    }  
    .padding()  
    .onAppear {  
        getMapSnapshot()  
    }  
    }  
    .navigationTitle(selectedCat.catName)  
}
```

This code will call the `getMapSnapshot()` method when the **Cat Detail** screen is displayed.

6. In App Preview, tap the **Map** button and tap the annotation. The **Cat Detail** screen will appear. Scroll down and you will see the map snapshot after the first image view:

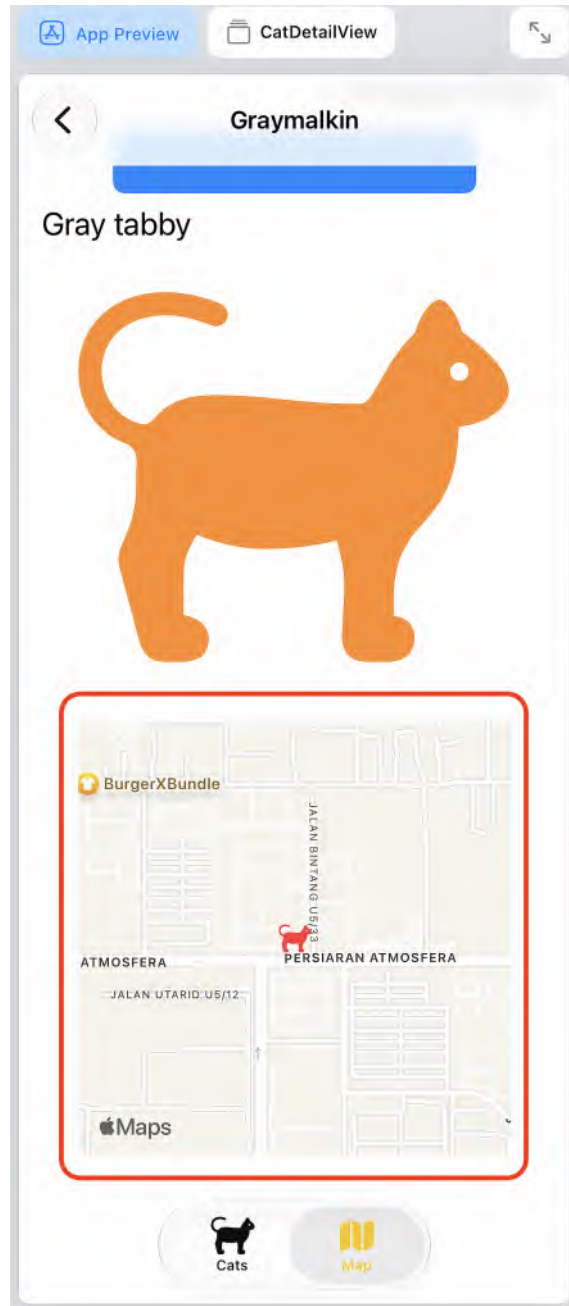


Figure 8.7: App Preview showing the Cat Display screen displaying the map snapshot

The **Cat Detail** screen can now display the map snapshot showing a cat's location. Cool!

Summary

In this chapter, you modified the **Add New Cat** screen so that the user can add the location where a cat was found. Next, you configured the **Map** screen to display a map region centered on your location. After that, you modified the **Map** screen to display a map annotation for each cat within the map region that you set earlier. You configured the map annotation to display the **Cat Detail** screen when tapped. Finally, you modified the **Cat Detail** screen to display a map snapshot showing the location where the cat was found.

You now know how to get your device location using Apple's Core Location framework, how to create custom map regions and display annotations using Apple's MapKit for SwiftUI framework, and how to create map snapshots, which will come in handy if you're planning to build apps that use maps, like *Apple Maps* or *Waze*.

In the next chapter, you'll learn how to create a shared data instance, and how to load and save data using SwiftData.

9

SwiftData

In the previous chapter, you modified the **Add New Cat** screen so that the user can include a location when they add a new cat and configured the **Map** screen to display annotations showing the locations where each cat was found. However, since the `MapView` view does not use the same `cats` array as the `CatsView` view, newly added cats do not appear on the **Map** screen as annotations. Also, all newly added cats are lost when you quit the app playground.

In this chapter, you will use **SwiftData** to create a persistent data container that will provide cat data to both the **Cats** and **Map** screens. This container will also be used to load cat data from your device when the app is launched and save cat data to your device when you add or delete cats.

First, you'll learn about SwiftData and its components. Next, you'll modify the `Cat` class to make it compatible with SwiftData, then modify the rest of the app playground to work with the modified `Cat` class. After that, you'll implement the data container for your app. Finally, you'll modify your app playground to load cat data when it is starting up, and to save cat data when you add or delete cats.

By the end of this chapter, you'll know how to use SwiftData to create a data container that can store, load, and save data in your own apps.

The following topics will be covered in this chapter:

- About SwiftData
- Modify the `Cat` class to work with SwiftData
- Create the data container
- Modify your app playground to use the data container

Technical requirements

You will continue working on the CatLog app playground that you modified in the previous chapter.

The completed app playground for this chapter is in the `Chapter09.zip` file of the code bundle for this book. To access the repository link, follow the steps in the *Download the example code files* section in the *Preface*.

Check out the following video to see the code in action:

<https://packt.link/zPX9Z>

Let's start by learning more about SwiftData in the next section.

About SwiftData

SwiftData is Apple's framework for saving app data to your device. With SwiftData, you can turn an existing class into a data model with the `@Model` macro. Primitive types such as `Bool`, `Int` and `String` are supported, as well as complex types such as structures and enumerations. Next, you specify which models to persist and create a `ModelContainer` instance, which manages an app's schema and model storage configuration. After that, you use a `ModelContext` instance to fetch, insert, and delete model instances, and save any changes to device storage while your app is running. Finally, to display model instances in a SwiftUI view, you use a `@Query` macro and specify a predicate or fetch descriptor.

Note

To watch Apple's WWDC23 video introducing SwiftData, see this link:

<https://developer.apple.com/videos/play/wwdc2023/10187/>.

To learn about the latest SwiftData updates announced by Apple during WWDC26, see this link:

<https://developer.apple.com/videos/play/wwdc2026/274/>.

To read Apple's developer documentation for SwiftData, see this link:

<https://developer.apple.com/documentation/swiftdata>.

To help you visualize how SwiftData works, imagine you're saving a cat's information using Microsoft Word. You first create a new Word document template with the relevant fields. You then create new Word documents based on the templates and fill in the data for each cat. You

make whatever changes are necessary, perhaps changing a cat's description or changing the photo. When you are happy with your document, you save it to the hard disk of your computer. The next time you want to see a particular cat's information, you search your hard disk for the relevant document and double-click it to open it in Word so you can see it once more.

Now that you have an idea how SwiftData works, let's review the steps required to implement SwiftData for your app playground.

First, you turn the existing `Cat` class into a model, which is like a Word template. You do this by annotating the `Cat` class with a `@Model` macro. The properties of the `Cat` class are like fields in a Word template.

Next, you will create a `ModelContainer` instance, which will be used to store `Cat` model instances on your device storage, and create a `ModelContext` instance, which will be used to store `Cat` model instances in memory. This is like the way Word files created from Word templates can be stored on your computer's hard disk or kept in memory as they are being edited.

After that, you will modify your app playground to allow `Cat` model instances to be added to or removed from the `ModelContext` instance, which then coordinates with `ModelContainer` instance to save changes to device storage. This is like manipulating Word documents on your hard disk.

You will also use the `@Query` macro to load `Cat` model instances when your app is launched. This is like using file search on your computer to find specific Word documents on your hard disk.

Now that you have a basic understanding of how SwiftData works, you'll turn the `Cat` class into a model in the next section.

Modify the `Cat` class to work with SwiftData

At present, when you add a cat with location information to your app playground, it will appear on the **Cats** screen, but when you switch to the **Map** screen, the newly added cat is not present. This is because the `CatsView` and `MapView` views are not using the same source of data. To solve this issue, you'll create a new data container using SwiftData which is referenced throughout your app. This means that the `CatsView` and `MapView` views will be getting their data from a single source.

Before you can implement the data container, you will need to modify the `Cat` class so that it can be turned into a `SwiftData` model. Follow these steps:

1. In the app playground, select the `Cat` file. Add a statement to import `SwiftData` after the existing `import` statement:

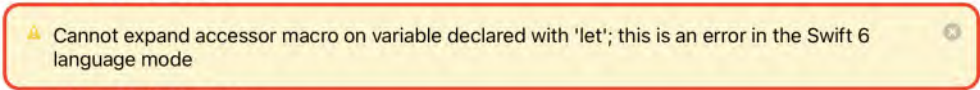
```
import SwiftUI
import SwiftData
```

2. Annotate the `Cat` class with `@Model` to turn it into a `SwiftData` model:

```
@Model
class Cat {
```

3. A warning will appear in your app playground. When you tap the warning notification, you will see the following:

```
4 @Model
5 class Cat {
6     let date: Date
7     let rating: Int
8     let catName: String
```

An expanded warning notification box with a yellow background and a red border. It contains a warning icon (a triangle with an exclamation mark) and the text: "Cannot expand accessor macro on variable declared with 'let'; this is an error in the Swift 6 language mode". There is a close button (an 'x' in a circle) in the top right corner of the box.

⚠ Cannot expand accessor macro on variable declared with 'let'; this is an error in the Swift 6 language mode

Figure 9.1 Expanded warning notification

4. To resolve this issue, change all the `let` property declarations in the `Cat` class to `var` declarations as shown:

```
class Cat {
    var date: Date
    var rating: Int
    var catName: String
    var catDescription: String
    var photo: Image?
    var latitude: Double?
    var longitude: Double?
```

5. An error will appear in your app playground. When you tap the Error button at the top right side of the screen, you will see the following:

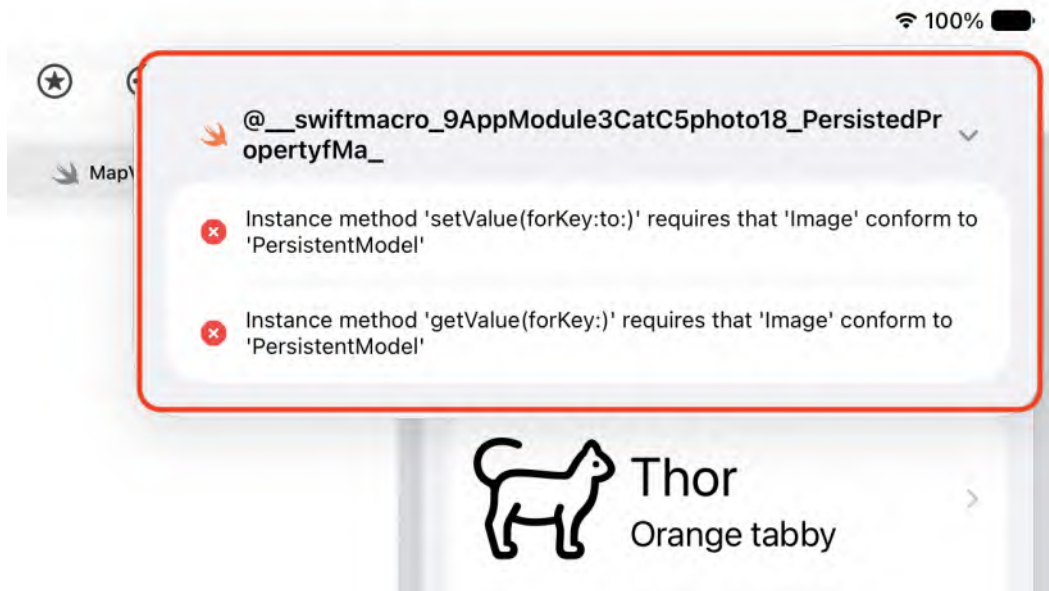


Figure 9.2: Error message showing issue with the Image type

This is because the Image type does not work with SwiftData, as it is a view and not intended for persistence. To resolve this, you will use the Data type to store image data and reconstruct the image using this data when required.

6. Replace the photo property in the Cat class with a photoData property as shown:

```
class Cat {
    var date: Date
    var rating: Int
    var catName: String
    var catDescription: String
    var photoData: Data?
    var latitude: Double?
    var longitude: Double?
}
```

7. More errors will appear. Modify the code as shown to fix the error in the initializer:

```
init(rating: Int, catName: String, catDescription: String,
    photoData: Data? = nil, latitude: Double? = nil,
    longitude: Double? = nil) {
    self.date = Date()
    self.rating = rating
    self.catName = catName
    self.catDescription = catDescription
    self.photoData = photoData
    self.latitude = latitude
    self.longitude = longitude
}
```

8. Add a computed variable after the longitude property to generate a photo from the data in photoData:

```
var longitude: Double?
var photo: UIImage? {
    photoData.flatMap {
        UIImage(data: $0)
    }
}
```

9. Modify the code in the extension as shown to fix the errors in the sample cat data:

```
static let sampleCatData = [
    Cat(rating: 5, catName: "Thor", catDescription: "Orange Tabby",
        photoData: UIImage(systemName: "cat")?.pngData()),
    Cat(rating: 4, catName: "Loki", catDescription: "Tuxedo cat",
        photoData: UIImage(systemName: "cat.fill")?.pngData()),
    Cat(rating: 3, catName: "Lokitu", catDescription: "Hungry cat",
        photoData: UIImage(systemName: "cat.fill")?.pngData()),
]
```

10. Switch to the AddNewCatView file. Modify the Save button code as shown to fix the error:

```
Button("Save", systemImage: "checkmark") {
    let newCat = Cat(rating: 3, catName: catName,
```

```

        catDescription: catDescription, photoData:
        UIImage(systemName: "cat.fill")?.pngData(),
        latitude:
        currentLocation?.coordinate.latitude,
        longitude:
        currentLocation?.coordinate.longitude)

```

11. Switch to the CatRowView file. Modify the code in the HStack as shown to fix the error:

```

HStack {
    if let catPhoto = cat.photo {
        Image(uiImage: catPhoto)
            .resizable()
            .scaledToFit()
            .frame(width: 90, height: 90)
    }
}

```

12. Switch to the MapView file. Modify the code in the updateMapWithLocation(_:) method as shown to fix the error:

```

private func updateMapWithLocation(_ location: CLLocation) {
    let interval = location.timestamp.timeIntervalSinceNow
    if abs(interval) < 30 {
        locationTask?.cancel()
        let lat = location.coordinate.latitude
        let long = location.coordinate.longitude
        cats.append(Cat(rating: 4, catName: "Graymalkin",
        catDescription: "Gray tabby", photoData:
        UIImage(systemName: "cat.fill")?.pngData(),
        latitude: lat, longitude: long))
        navTitle = "Map"
    }
}

```

13. Verify that all the errors have been resolved, and the app playground works like it did before.

You have successfully modified the Cat class to work with SwiftData, and resolved all the errors in your app playground. Excellent!

In the next section, you'll create a data container that can store Cat model objects for your app playground.

Create the Data Container

Now that you have modified the `Cat` class to work with `SwiftData`, you create a `SwiftData` data container and use the `Cat` class to specify the model objects that it will store. Follow these steps:

1. In your app playground, create a new Swift file named `DataContainer`. Add the following code to the file after the `import` statement:

```
import SwiftUI
import SwiftData

@Observable
@MainActor
class DataContainer {
    let modelContainer: ModelContainer
    var context: ModelContext {
        modelContainer.mainContext
    }
    init() {
        let schema = Schema([Cat.self])
        let modelConfiguration = ModelConfiguration(schema: schema)
        do {
            modelContainer = try ModelContainer(for: schema,
            configurations: [modelConfiguration])
        } catch {
            fatalError("Could not create ModelContainer: \(error)")
        }
    }
}
```

Let's break this down:

```
@Observable
```

This macro tells SwiftUI to watch this class for changes.

```
@MainActor
```

This class is marked with `@MainActor` to ensure that it should only be accessed from the main queue.

Note

For more information, watch Apple's WWDC 2022 video titled *Eliminate data races using Swift Concurrency* here: <https://developer.apple.com/videos/play/wwdc2022/110351/>.

```
class DataContainer {  
    let modelContainer: ModelContainer  
    var context: ModelContext {  
        modelContainer.mainContext  
    }  
}
```

These statements declare the `DataContainer` class. The `modelContainer` property sets a space where data will be stored on your device. The `context` property allocates a `ModelContext` object, where operations can be performed on data.

```
init() {  
    let schema = Schema([Cat.self])  
    let modelConfiguration = ModelConfiguration(schema: schema)
```

The initializer will be called when the app is launched. The schema determines the type of instances that will be stored.

```
do {  
    modelContainer = try ModelContainer(for: schema,  
        configurations: [modelConfiguration])  
} catch {  
    fatalError("Could not create ModelContainer: \(error)")  
}
```

These statements will attempt to create the model container. If creation fails, the app will crash and display an error message.

2. Switch to the `MyApp` file. Modify the code as shown to create the data container for your app playground to use:

```
import SwiftUI
import SwiftData

struct MyApp: App {
    let dataContainer = DataContainer()
    var body: some Scene {
        WindowGroup {
            ContentView()
                .environment(dataContainer)
        }
        .modelContainer(dataContainer.modelContainer)
    }
}
```

You have declared the model container, created it and assigned it to your app playground. In the next section, you will modify your app playground to use it.

Modify Your App Playground to Use the Data Container

Now that you have created the data container, you'll modify your app playground to use it. You'll start by making changes to the `CatsView` view to read cat data from the data container when the app playground is started, and to remove cats that are deleted from the data container. Follow these steps:

1. In your app playground, select the `CatsView` file. Add a statement to import `SwiftData` after the `import` statement:

```
import SwiftUI
import SwiftData
```

2. After the `showAddCatView` property declaration, add a statement to query the data container for any available cat instances for use in the app:

```
struct CatsView: View {  
    @State private var showAddCatView = false  
    @Query(sort: \Cat.date, order: .reverse)  
    @State private var cats = Cat.sampleData
```

The instances obtained from the data container will be sorted by date, with the latest entry appearing at the top of the list.

3. Modify the line declaring the `cats` array as shown:

```
struct CatsView: View {  
    @State private var showAddCatView = false  
    @Query(sort: \Ccat.date, order: .reverse)  
    private var cats: [Cat]
```

You no longer need the `@State` annotation and will no longer be using the sample data. The `cats` array will automatically be populated from the data container. The extension in the `Cats` file containing the sample cat data is no longer needed and may be removed.

4. Add a statement after the `cats` property declaration to declare an environment variable referencing the data container:

```
private var cats: [Cat]  
@Environment(DataContainer.self) private var dataContainer
```

5. Modify the `ForEach` statement as shown. `id` is no longer required since `SwiftData` can uniquely identify `Cat` instances without using it:

```
List {  
    ForEach(cats) { cat in
```

6. Modify the `onDelete` modifier as shown to remove the specified `Cat` instance from the data container:

```
.onDelete { indexSet in  
    for index in indexSet {
```

```
dataContainer.context.delete(cats[index])
do {
    try dataContainer.context.save()
} catch {
    print("Error: \(error.localizedDescription)")
}
}
```

7. Modify the `.sheet(isPresented:)` modifier for the toolbar button as shown. You no longer need to pass a binding to the `AddNewCatView` view, as new `Cat` instances will be inserted into the data container:

```
Button("Add", systemImage: "plus") {
    showAddCatView = true
}
.sheet(isPresented: $showAddCatView) {
    AddNewCatView()
}
```

An error will appear. Don't worry about it as it will be resolved when you modify the `AddVewCatView` file.

8. Remove the `#Preview` block from the `CatsView` file:

```
// remove this block
#Preview {
    CatsView()
}
```

Note

The `#Preview` block is removed because in order for it to work with `SwiftData`, you will need to configure previews with a sample model container that is different from the one used by the app, which can be confusing for new developers. The following tutorial will guide you on how to do this:

<https://developer.apple.com/tutorials/develop-in-swift/save-data>

You have made all the required changes to the CatsView view. Next, you will modify the AddNewCatView view. Follow these steps:

1. Switch to the AddNewCatView file. Import SwiftData after all other import statements:

```
import SwiftUI
import CoreLocation
import SwiftData
```

2. Remove the cats property:

```
struct AddNewCatView: View {
    @Binding var cats: [Cat] // remove this
    @State private var catName = ""
```

3. Add a statement after the dismiss variable declaration to declare an environment variable referencing the data container:

```
@Environment(\.dismiss) private var dismiss
@Environment(DataContainer.self) private var dataContainer
```

4. Modify the Save button method as shown to add a new cat to the data container:

```
Button("Save", systemImage: "checkmark") {
    let newCat = Cat(rating: 3, catName: catName,
catDescription: catDescription, photoData:
UIImage(systemName: "cat.fill")?.pngData(),
latitude: currentLocation?.coordinate.latitude,
longitude: currentLocation?.coordinate.longitude)
    dataContainer.context.insert(newCat)
    do {
        try dataContainer.context.save()
    } catch {
        print("Error: \(error.localizedDescription)")
    }
    dismiss()
}
.disabled(catName.isEmpty || catDescription.isEmpty)
```

5. Remove the #Preview block from the AddNewCatView file.

You have made all the required changes to the `AddNewCatsView` view. Next, you will modify the `MapView` view. Follow these steps:

1. Switch to the `MapView` file. Import `SwiftData` after all other import statements:

```
import SwiftUI
import MapKit
import CoreLocation
import SwiftData
```

2. After the `MapView` declaration, add a statement to query the data container for any available cat instances for use in the app:

```
struct MapView: View {
    @Query(sort: \Cat.date)
    @State private var cats = Cat.sampleData
```

The instances obtained from the data container will be sorted by date.

3. Modify the line declaring the `cats` array as shown:

```
struct MapView: View {
    @Query(sort: \Cat.date)
    private var cats: [Cat]
```

You no longer need the `@State` annotation and will no longer be using the sample data. The `cats` array will automatically be populated from the data container.

4. Modify the `ForEach` statement as shown. `id` is no longer required since `SwiftData` can uniquely identify `Cat` instances without using it:

```
Map(position: $position) {
    ForEach(cats) { cat in
```

5. In the `updateMapWithLocation(_:)` method, remove the statement to append a new cat to the `cats` array:

```
let long = location.coordinate.longitude
// code removed
navTitle = "Map"
```

6. Remove the #Preview block from the MapView file.
7. Remove the #Preview block from the CatDetailView and CatRowView files.

You will now test your app on your iPad. Follow these steps:

1. Tap the Run button to launch the app. Tap the + button and add a new cat. Make sure the **Get Location** switch is on:

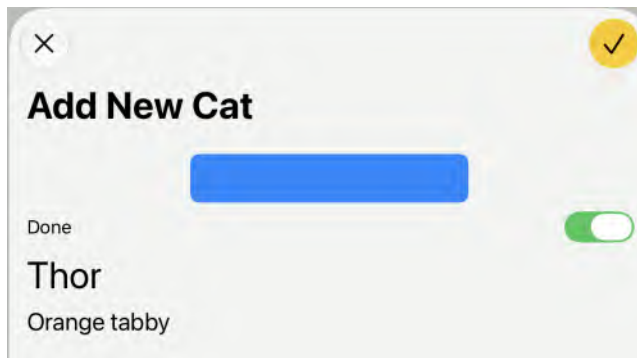


Figure 9.3: iPad showing Add New Cat screen

2. The cat will appear on the **Cats** screen. Tap the **Map** button to go to the **Map** screen. Note that the cat you added earlier is visible as an annotation on the **Map** screen:

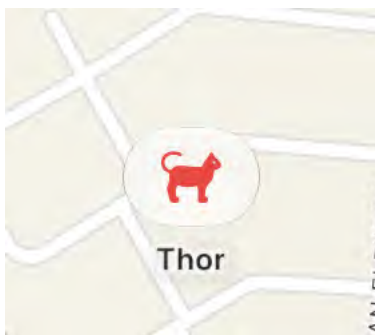


Figure 9.4: iPad showing Map screen

3. Tap the small red button in the top right-hand side of the screen to display the **Swift Playground App Window** and tap **Stop** to stop the running app:

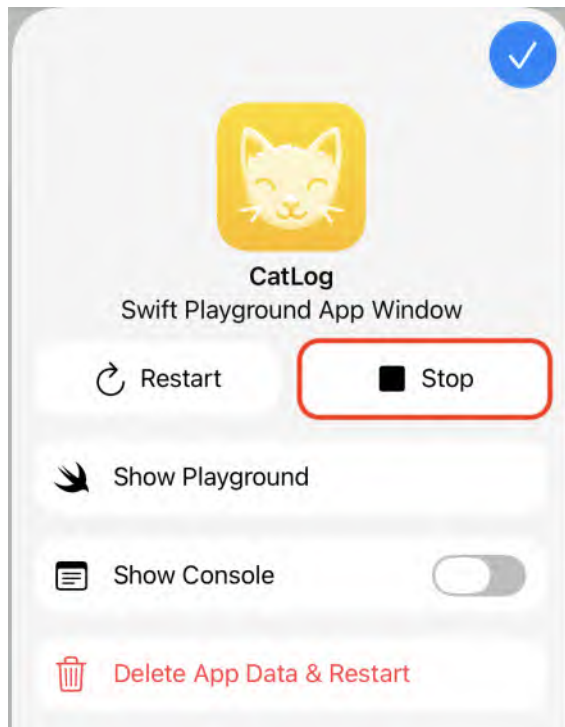


Figure 9.5: Swift Playground App Window

4. Force-quit the Swift Playgrounds app using the app switcher:

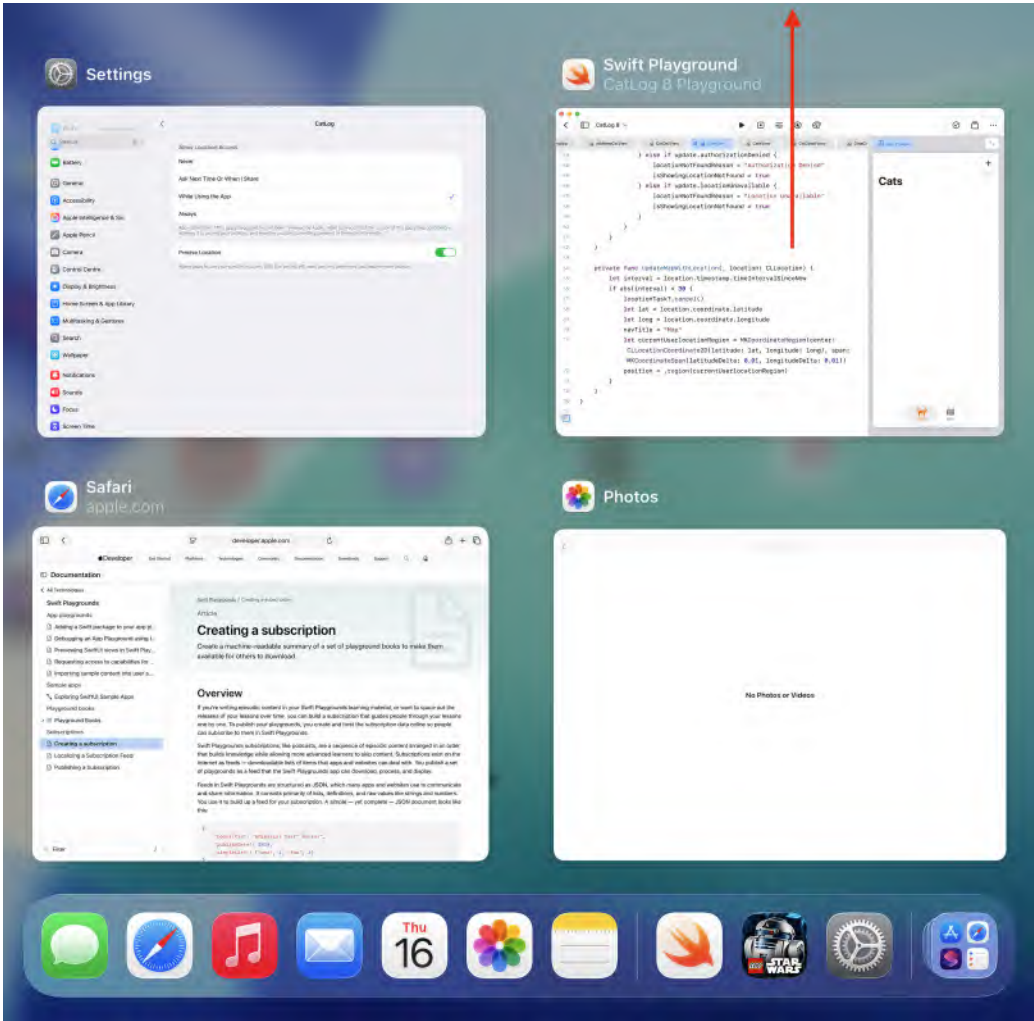


Figure 9.6: iPad showing app switcher

5. Relaunch Swift Playground. The cats you entered earlier will still be there when you relaunch the app:

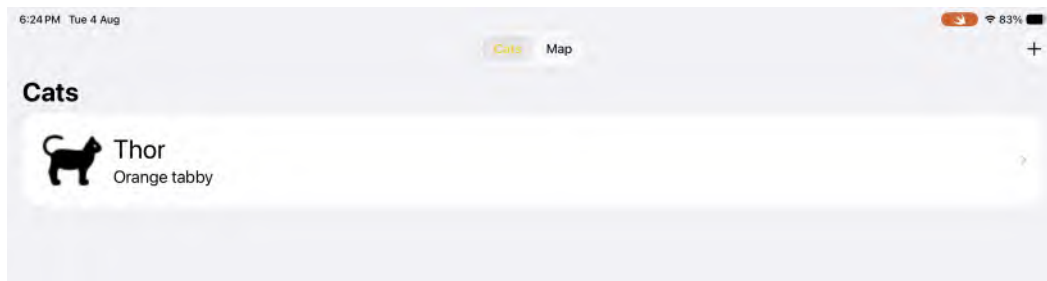


Figure 9.7: iPad showing relaunched app

You have successfully configured your app playground to use the data container for loading and saving cat data! Fantastic job!

Summary

In this chapter, you started by learning about `SwiftData` and its components. Next, you modified the `Cat` class to work with `SwiftData`. After that, you created the data container for your app. Finally, you modified your app playground to load cat data when it is starting up, and to save cat data when you add or delete cats.

You now know how to use `SwiftData` to create a data container that can store, load, and save data in your own apps.

In the next chapter, you'll implement a custom user interface element that allows you to set star ratings for cats.

10

Custom Views

At this point, your CatLog app playground is mostly functional. All the screens are working, but as the rating user interface element is missing, you can't set a star rating for a cat as shown in the app tour. You also can't set a custom picture, but that will be addressed in *Chapter 11, Photo Library*.

You have been using Apple's standard UI elements so far. In this chapter, you'll create a custom view that displays a cat's rating in the form of stars, and you'll modify it so users can rate a cat by tapping it. After that, you'll implement it on the **Add New Cat** screen. Finally, you'll implement it on the **Cat Detail** screen.

By the end of this chapter, you'll have learned how to create custom views for your own apps.

The following topics will be covered in this chapter:

- Create a custom view
- Add your custom view to the **Add New Cat** screen
- Add your custom view to the **Cat Detail** screen

Technical requirements

You will continue working on the CatLog app playground that you modified in the previous chapter.

The completed app playground for this chapter is in the `Chapter10.zip` file of the code bundle for this book. To access the repository link, follow the steps in the *Download the example code files* section in the *Preface*.

Check out the following video to see the code in action:

<https://packt.link/ciT32>

Let's start by learning how to create a custom stack view that will display a star rating on the screen.

Create a Custom View

You've only used Apple's predefined UI elements so far, such as text and image views. However, there will be cases where you require views that are not provided by Apple. In such cases, you will need to create your own. Let's review the **Add New Cat** screen that you saw in the app tour:

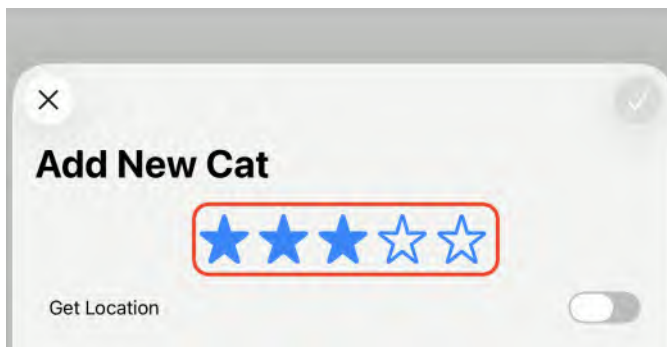


Figure 10.1: Add New Cat screen showing the star rating

You can see a group of five stars just above the **Get Location** switch. Currently, the `AddNewCatView` and `CatDetailView` views have placeholder view objects where the stars should be. You will create the `RatingView` view, a custom view that will display ratings as stars.

For the rest of this chapter, the `RatingView` view may also be referred to as a rating view (the same way an `Image` view is referred to as an image view).

Let's begin by creating a custom view. Follow these steps:

1. In your app playground, create a new Swift file named `RatingView`, and add the following code to it after the `import` statement:

```
struct RatingView: View {
    @Binding var rating: Int
    let maxRating = 5
    let emptyStar = Image(systemName: "star")
    let filledStar = Image(systemName: "star.fill")
    var body: some View {
        HStack(spacing: 8) {
            ForEach(1..
```

```

        number in
        Button {
            rating = number
        } label: {
            image(for: number)
                .resizable()
                .frame(width: 44, height: 44)
                .foregroundColor(.blue)
        }
    }
}
}
func image(for number: Int) -> Image {
    if number > rating {
        emptyStar
    } else {
        filledStar
    }
}
}

#Preview {
    RatingView(rating: .constant(3))
}

```

Let's break this down:

```
@Binding var rating: Int
```

This property holds the rating value. The `@Binding` keyword tells SwiftUI that `rating` has a two-way connection with another property in your app playground.

```
let maxRating = 5
```

This property determines how many stars will be displayed in the view.

```
let emptyStar = Image(systemName: "star")
let filledStar = Image(systemName: "star.fill")
```

These statements create two `Image` views from symbols in Apple's SF Symbols library. `emptyStar` will store an image of a star outline and `filledStar` will store an image of a filled star.

Note

For more information on Apple's SF Symbols library, see this link:

<https://developer.apple.com/design/human-interface-guidelines/sf-symbols>.

```
HStack(spacing: 8) {
```

This statement declares a horizontal stack view with the elements inside it separated by 8 points.

```
  ForEach(1..<maxRating + 1, id: \.self) {  
    number in
```

Since `maxRating` is set to 5, the body of this `ForEach` loop will repeat five times, with `number` initially assigned 1, and incremented each time the loop repeats.

```
    Button {  
      rating = number  
    } label: {  
      image(for: number)  
        .resizable()  
        .frame(width: 44, height: 44)  
        .foregroundColor(.blue)  
    }  
  }
```

This statement creates a button view. The button view's action assigns `number` to `rating`. The button's appearance is determined by the `image(for:)` method and will either be a filled star or an empty star.

Note

For more information on Button, see this link:

<https://developer.apple.com/documentation/SwiftUI/Button>.

```
func image(for number: Int) -> Image {
    if number > rating {
        emptyStar
    } else {
        filledStar
    }
}
```

This method returns the image that the button will display. If `number` is larger than `rating`, the button will display a star outline; otherwise, it will display a filled star.

For example, let's say `rating` is initially set to 0. When you tap the third button in the rating view, the `number` value for that button is 3, and `rating` is set to 3.

The `number` value for the first button is 1, so `number > rating` is `1 > 3`, which returns `false`. This button's image will be set to a filled star, and same is true for the next two buttons.

The `number` value for the fourth button is 4, so `number > rating` is `4 > 3`, which returns `true`. This button's image will be set to a star outline, and the same is true for the fifth button.

In short, when the `rating` property is set to 3, the rating view displays the first three buttons with filled stars, and the remaining two buttons with star outlines.

You have completed the implementation of the rating view. In the next section, you'll add it to the **Add New Cat** screen.

Add your custom view to the Add New Cat screen

So far, you have created a new `RatingView` view in your app playground and configured it to set its rating property when a star in it is tapped. In this section, you will replace the placeholder view in the **Add New Cat** screen with the rating view, and use the rating property's value to set a cat's rating. Follow these steps:

1. In the app playground, select the `AddNewCatView` file. Add a statement declaring a state variable that will store the rating value after the `catDescription` property declaration:

```
@State private var catName = ""
@State private var catDescription = ""
@State private var rating = 0
@State private var toggleText = "Get Location"
@State private var toggleValue = false
```

2. Replace the placeholder view in the `VStack` with a rating view as shown:

```
VStack(alignment: .center) {
    RatingView(rating: $rating)
    Toggle(toggleText, isOn: $toggleValue)
```

Since the rating state variable's initial value is 0, all the buttons in the rating view will display star outlines.

3. Modify the Save button action as shown to get the value from the rating state variable when creating a new cat instance:

```
Button("Save", systemImage: "checkmark") {
    let newCat = Cat(rating: rating, catName:
    catName, catDescription: catDescription,
```

4. Tap the Run button. Tap the + button to go to the **Add New Cat** screen, and you will see the rating view displayed above the **Get Location** switch. Add a cat name, description and rating, then tap save:

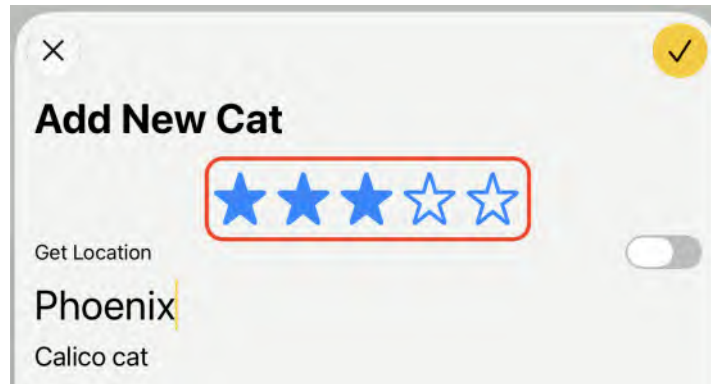


Figure 10.2: iPad showing the rating view on the Add New Cat screen

A new cat is now visible on the **Cats** screen. In the next section, you'll modify the **Cat Detail** screen to display the rating for this cat.

Add your custom view to the Cat Detail screen

At this point, you're able to set a rating for a new cat using the **Add New Cat** screen, but the rating you set is not visible on the **Cat Detail** screen. You'll replace the placeholder in the `CatDetailView` view with a rating view and modify the code to make it display a cat's rating. Follow these steps:

1. In the app playground, switch to the `CatDetailView` file. Replace the placeholder view with a rating view as shown:

```
Text(selectedCat.date, style: .date)
    .font(.title2)
    .frame(maxWidth: .infinity, alignment: .trailing)
RatingView(rating: .constant(selectedCat.rating))
    .disabled(true)
Text(selectedCat.catDescription)
    .font(.title2)
    .frame(maxWidth: .infinity, alignment: .leading)
```

Note that the rating view is provided with a `.constant` value, and has a `.disabled()` modifier. This is so that the rating value can't be modified by the user.

2. Tap the Run button. Tap the cat you added in the previous section, and you'll see the rating displayed in a rating view on the **Cat Detail** screen:



Figure 10.3: iPad showing the rating view on the Cat Detail screen

You've successfully added and configured a rating view on the **Cat Detail** screen! Good job!

Summary

In this chapter, you created a custom rating view that displays a rating in the form of stars, and you modified this view so users can set a rating for cat by tapping it. After that, you added it to the **Add New Cat** screen. Finally, you implemented it on the **Cat Detail** screen.

You now know how to create custom views for your own apps. In the next chapter, you'll learn how to work with photos from the photo library.

11

Photo Library

In the previous chapter, you created the `RatingView` view and added it to the **Add New Cat** and **Cat Detail** screens.

In this chapter, you will complete the implementation of the **Add New Cat** screen by adding a way to get a cat photo from the **photo library**. You'll start by implementing a `PhotosPicker` view, which allows you to get a photo from the photo library. After that, you'll modify your app playground to save the photo in the **Add New Cat** screen and to display the saved photo in the **Cat Detail** screen.

By the end of this chapter, you'll have learned how to enable access to the photo library in your own apps.

The following topic will be covered in this chapter:

- Implement a new `PhotosPicker` view

Technical requirements

You will continue working on the `CatLog` app playground that you modified in the previous chapter.

The completed app playground for this chapter can be found in the `Chapter11.zip` file of the code bundle for this book. To access the repository link, follow the steps in the *Download the example code files* section in the *Preface*.

Check out the following video to see the code in action:

<https://packt.link/YhFPr>

Let's start by implementing a `PhotosPicker` view in the **Add New Cat** screen, which allows you to select a photo from the user's photo library.

Implement a new PhotosPicker view

To make it easy for a user to use a photo from the photo library, Apple implemented the `PhotosPicker` view. This view manages the interface for choosing items from the user's photo library and it can display a photo picker on the screen.

If you have ever added a photo to a social media post, you will have seen what the photo picker looks like. It displays a grid of photos from your photo library, and you can then choose a photo to be added to your post:

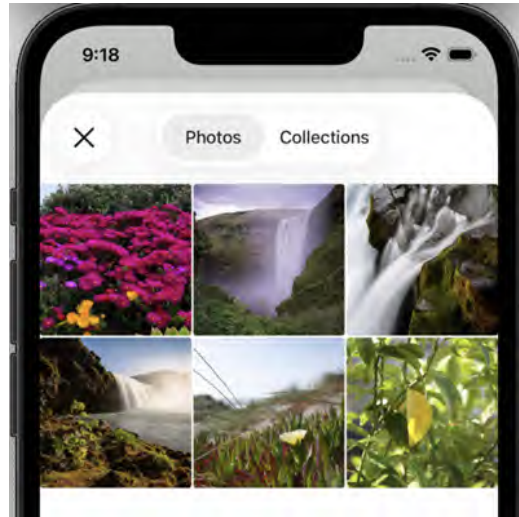


Figure 11.1: Simulator showing the photo picker

Note

To learn more about the `PhotosPicker` view, see this link: <https://developer.apple.com/documentation/PhotosUI/PhotosPicker>.

To display the photo picker on the **Add New Cat** screen, you'll add a `PhotosPicker` view to the **Add New Cat** screen and configure the Save button to save the selected photo when a new cat

is added. Photos can be large, so you'll also modify the `Cat` class to efficiently store photo data. Finally, you'll modify the **Cat Detail** screen to display the saved photo. Follow these steps:

1. In the app playground, select the `Cat` file and modify the `Cat` class as shown:

```
class Cat {  
    var date: Date  
    var rating: Int  
    var catName: String  
    var catDescription: String  
    @Attribute(.externalStorage) var photoData: Data?  
    var latitude: Double?  
    var longitude: Double?
```

The `@Attribute(.externalStorage)` annotation will store the data for the photo in a binary file adjacent to the model data, which makes it more efficient.

2. Select the `AddNewCatView` file. Import the `PhotosUI` framework after all other import statements:

```
import SwiftUI  
import CoreLocation  
import SwiftData  
import PhotosUI
```

3. Declare state variables to store the new photo and its data after all other state variables:

```
@State private var isShowingLocationNotFound = false  
@State private var locationNotFoundReason = ""  
@State private var photoData: Data?  
@State private var newPhoto: PhotosPickerItem?
```

4. Modify the `Save` button code as shown to save the new photo's data to the `Cat` instance when a new cat is added:

```
Button("Save", systemImage: "checkmark") {  
    let newCat = Cat(rating: rating, catName:  
catName, catDescription: catDescription,  
photoData: photoData, latitude:  
currentLocation?.coordinate.latitude,
```

```
longitude:
    currentLocation?.coordinate.longitude)
```

5. Implement the `PhotosPicker` view as shown before the closing curly brace:

```
private var photosPicker: some View {
    PhotosPicker(selection: $newPhoto, matching: .images) {
        Group {
            if let photoData, let uiImage = UIImage(data:
                photoData) {
                Image(uiImage: uiImage)
                    .resizable()
                    .scaledToFit()
                    .frame(width: 300, height: 300)
            } else {
                Image(systemName: "photo.badge.plus.fill")
                    .font(.largeTitle)
                    .frame(width: 300, height: 300)
                    .background(Color.gray)
            }
        }
        .clipShape(RoundedRectangle(cornerRadius: 16))
    }
    .onChange(of: newPhoto) {
        guard let newPhoto else { return }
        Task {
            guard let newPhotoData = try await
                newPhoto.loadTransferable(type: Data.self), let
                fullsizePhoto = UIImage(data: newPhotoData), let
                smallerPhoto = fullsizePhoto.preparingThumbnail(of:
                    CGSize(width: 300, height: 300)) else { return }
            photoData = smallerPhoto.pngData()
        }
    }
}
```

When the **Add New Cat** screen first appears, both `newPhoto` and `photoData` do not have values. The `else` block is executed, and an image view containing the `photo.badge.plus.fill` SF Symbol is displayed in the **Add New Cat** screen. Tapping this image view will display a photo picker showing the contents of the photo library.

The user may dismiss the photo picker or pick a photo from the photo library. If the user picks a photo, `.onChange(of:)` is triggered and the user selected photo is assigned to `newPhoto`. In the Task block, the data from `newPhoto` is reconverted into the photo at its original resolution and resized to 300 by 300 points, reducing the amount of storage required. It is then assigned to `photoData`.

Since `photoData` now has a value, the `if` block is executed, which converts the data in `photoData` to a `UIImage` instance, and passes it as an argument to `Image(uiImage:)`. This will generate an image view which is then displayed in the **Add New Cat** screen.

6. In the `VStack`, replace the image view after the text field with `photosPicker` as shown:

```
TextField("Cat description", text: $catDescription, axis: .vertical)
    .font(.title2)
    .multilineTextAlignment(.leading)
    .lineLimit(5...Int.max)
photosPicker
}
```

7. Switch to the `CatDetailView` file. In the `VStack`, modify the image view as shown:

```
Text(selectedCat.catDescription)
    .font(.title2)
    .frame(maxWidth: .infinity, alignment: .leading)
if let catPhoto = selectedCat.photo {
    Image(uiImage: catPhoto)
        .resizable()
        .scaledToFit()
        .frame(width: 300, height: 300)
        .padding()
}
if let mapSnapshot {
```

The `if` statement checks to see if `selectedCat.photo` has a value. If it does, that value is assigned to `catPhoto`, which is then passed as an argument to `Image(uiImage:)`. This will generate an image view containing the photo in the Cat Detail screen.

All the code required to choose a photo in the **Add New Cat** screen and display that photo in the **Cat Detail** screen has been implemented.

8. Run your app playground and tap the + button. In the **Add New Cat** screen, tap the photo picker view:

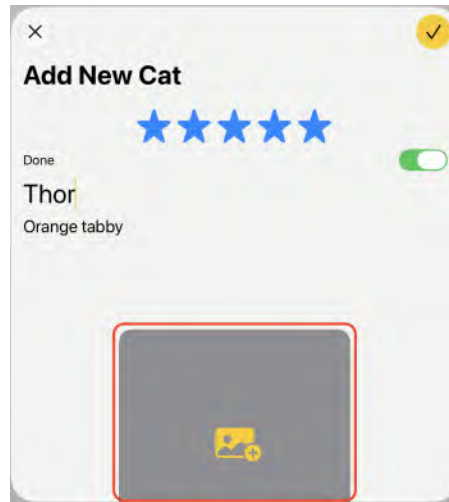


Figure 11.2: Add New Cat screen showing the photo picker view

9. The contents of your photo library will appear. Tap one of the photos to select it:

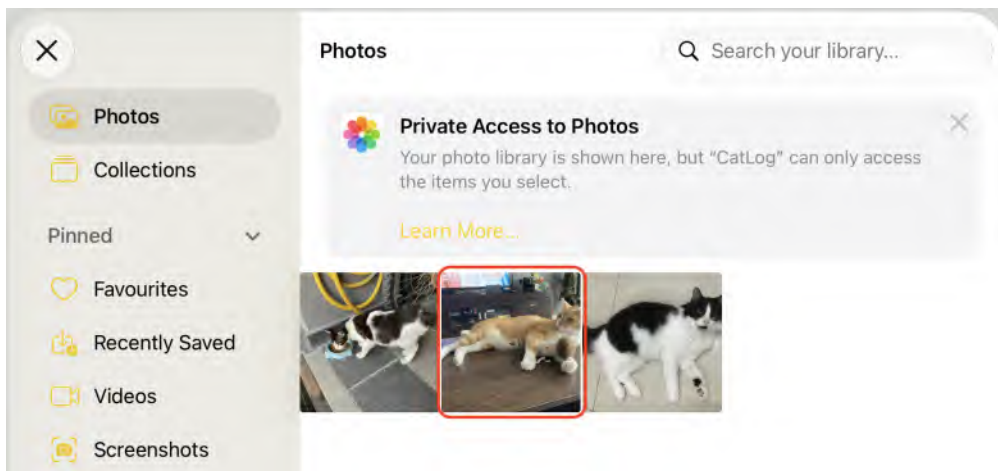


Figure 11.3: Photo library showing the photo to be added

Note that the app can only access the items you select and does not have access to the entire photo library.

10. The selected photo will appear in the **Add New Cat** screen. Enter sample details for the cat and tap the Save button:

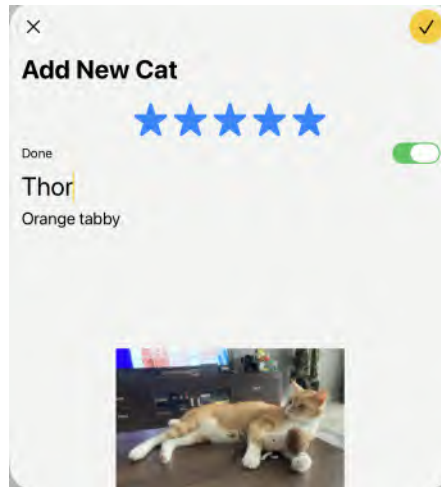


Figure 11.4: Add New Cat screen showing the selected photo

11. You will be returned to the **Cats** screen. Tap the newly added cat. You will see that the photo is displayed on the **Cat Detail** screen:

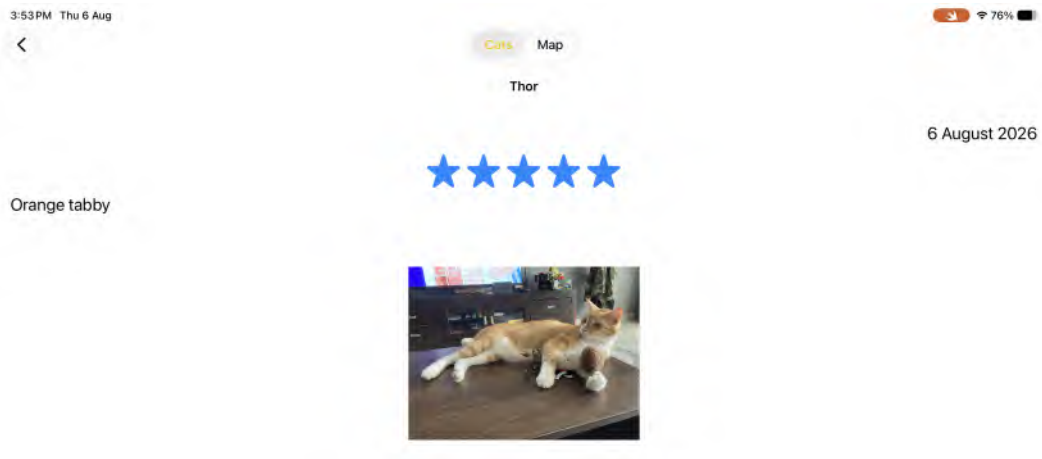


Figure 11.5: Cat Detail screen showing the selected photo

You can now add the photo of a cat on the **Add New Cat** screen and display it on the **Cats** and **Cat Detail** screens. Fantastic!

Summary

In this chapter, you completed the implementation of the **Add New Cat** screen by adding a way to get a photo from the photo library, which then can be added to a new cat entry. First, you implemented a `PhotoPicker` view, which allows you to get a photo from the photo library. After that, you modified your app playground to save the photo in the **Add New Cat** screen and to display the saved photo in the **Cat Detail** screen.

You are now able to write your own apps that import photos from your photo library. Fantastic!

In the next chapter, you'll implement a way for the user to search for a cat on the **Cats** screen.

12

Search

In the previous chapter, you added a way for the user to get a photo from the photo library, which can be added to a new cat entry.

In this chapter, you will implement a **search bar** for the **Cats** screen. You'll modify the `CatsView` view to display a search bar on the **Cats** screen and modify the code to display cats that match the search term typed in by the user. You'll also implement a navigation split view to make your app look better on the iPad.

By the end of this chapter, you'll have learned how to implement a search bar for your own apps. To name one example, if you were creating a contacts app, you could use a search bar to search for a particular contact.

The following topics will be covered in this chapter:

- Implement a search bar for the **Cats** screen
- Implement a navigation split view

Technical requirements

You will continue working on the CatLog app playground that you modified in the previous chapter.

The completed app playground for this chapter is in the `Chapter12.zip` file of the code bundle for this book. To access the repository link, follow the steps in the *Download the example code files* section in the *Preface*.

Check out the following video to see the code in action:

<https://packt.link/FAFNM>

Let's start by modifying the `CatsView` view to display a search bar on the **Cats** screen.

Implement a search bar for the Cats screen

At present, you only have a few cats on the **Cats** screen. But the longer you use the app, the more cats you'll have, and it's going to be hard to find a specific cat. To make it easier to look for a cat, you'll add a search bar to the navigation bar of the **Cats** screen. You will use the `searchable(text:placement:prompt:)` modifier to do this. To perform the search, you will filter the contents of the cats array using the search term that the user has typed in.

Note

To learn more about adding a search interface to your app, see this link:

<https://developer.apple.com/documentation/swiftui/adding-a-search-interface-to-your-app>.

You will now declare a state variable to store the search term typed in by the user, add the `searchable(text:placement:prompt:)` modifier to the list view in the `CatsView` view, and filter the cats in the `cats` array using the search term. The filtered results will be displayed in the **Cats** screen when the search bar is active. Follow these steps:

1. In the app playground, tap the `CatsView` file. Declare a state variable to store the search term after the `showAddCatView` declaration:

```
@State private var showAddCatView = false
@State private var searchText = ""
@Query(sort: \Cat.date)
```

2. Declare an array to store the filtered search results after the `cats` array declaration:

```
private var cats: [Cat]
private var filteredCats: [Cat] {
    if searchText.isEmpty {
        cats
    } else {
        cats.filter { $0.catName.localizedStandardContains(
            searchText)}
    }
}

@Environment(DataContainer.self) private var dataContainer
```

If `searchText` is empty, the entire `cats` array is assigned to `filteredCats`, otherwise, only cats with names containing the search term (including partial matches) are assigned to `filteredCats`. Since `filteredCats` is a computed property, its value will automatically be updated when the search term is changed.

The `.localizedStandardContains()` method automatically ignores case and accents on letters.

3. Modify the `ForEach` statement in the list view to use `filteredCats` in place of `cats`:

```
List {  
    ForEach(filteredCats) { cat in  
        NavigationLink(destination:
```

4. Add the `searchable(text:placement:prompt:)` modifier to the list view:

```
}  
.searchable(text: $searchText, prompt: "Find a cat")  
.navigationTitle("Cats")  
.toolbar {
```

Any text you type into the search bar will be assigned to `searchText`.

5. Modify the `.onDelete` modifier to use the `filteredCats` array instead of the `cats` array:

```
.onDelete { indexSet in  
    for index in indexSet {  
        dataContainer.context.delete(filteredCats[index])  
    }  
    try dataContainer.context.save()
```

6. Run your app playground, and you'll see a search bar on the **Cats** screen. Type some text into the search bar, and the cats whose names contain the search term will be displayed:

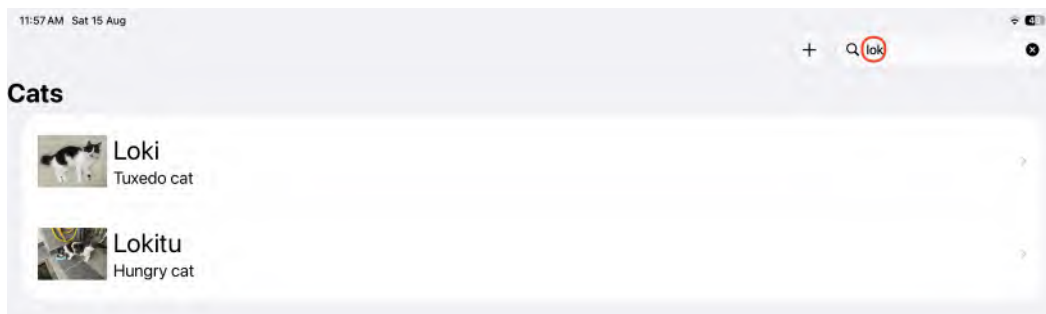


Figure 12.1: iPad showing search results in the Cats screen

You can now search for a cat on the **Cats** screen. Great! In the next section, you'll replace the navigation stack view with a navigation split view that will make better use of the iPad's large screen.

Implement a navigation split view

A navigation split view presents views in two or three columns, depending on screen size. Tapping a view in the first column will determine which views are presented in the second column. For devices that have a smaller screen, such as an iPhone, the navigation split view will behave like a navigation stack view.

Note

For more details about the navigation split view, see this link: <https://developer.apple.com/documentation/SwiftUI/NavigationSplitView>.

You will replace the navigation stack view in the `CatsView` view with a navigation split view. The left column will show the list of cats, and tapping on a cat will display the details in the right column. Follow these steps:

1. In the app playground, tap the `CatsView` file. Replace the navigation stack view with a navigation split view as shown:

```
var body: some View {  
    NavigationSplitView {  
        List {
```

An error will appear because a navigation split view requires a detail view which will be displayed initially in the right column.

2. Modify the code as shown to add the missing detail parameter, and add a modifier that will reduce the width of the **Cat Detail** screen to make room when showing the **Cats** screen in the left column:

```

        .sheet(isPresented: $showAddCatView) {
            AddNewCatView()
        }
    }
}
} detail: {
    Text("Choose a cat")
    .font(.largeTitle)
}
.navigationSplitViewStyle(.balanced)
}
}

```

3. Run your app playground, and you will see the list of cats in the left column, and the initial detail view in the right column:



Figure 12.2: iPad showing list of cats in the left column

4. Tapping a cat in the left column displays its details in the right column:

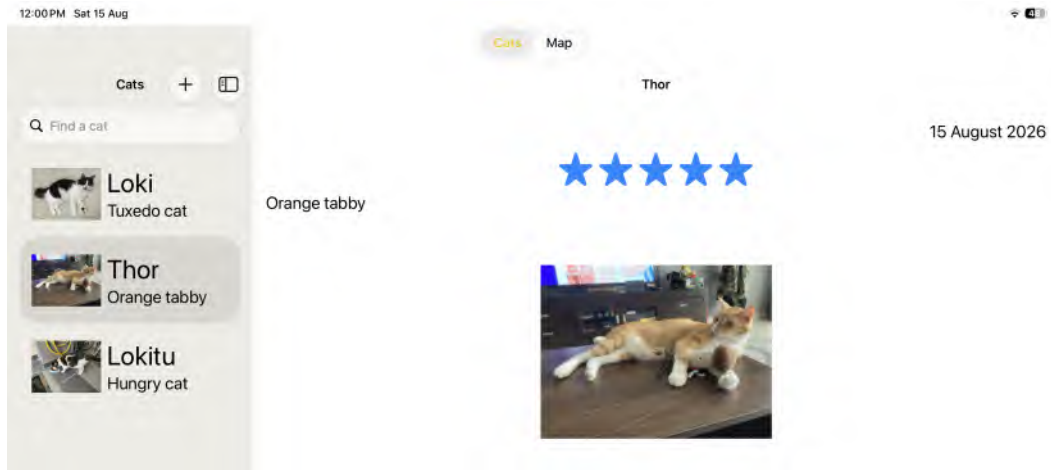


Figure 12.3: iPad showing cat details in the right column

You have successfully implemented the search bar for your app, and modified it to make better use of the screen space on the iPad! Excellent!

Summary

In this chapter, you implemented a search bar for the **Cats** screen. First, you modified the **CatsView** view to display a search bar on the **Cats** screen. Then, you modified the code to display cats that match the search term typed in by the user. You also implemented a navigation split view to make your app look better on the iPad.

You have now learned how to implement a search bar for your own apps, and you have also completed the *CatLog* app. Fantastic job!

In the next chapter, you'll learn how to test and submit your app to the App Store.

Part 3

App Store

Welcome to *Part 3* of this book. In this part, you will learn how to test and submit your app to the App Store. First, you will learn how to get a paid Apple developer account. Next, you'll learn how to submit your app to App Store Connect and how to use TestFlight for testing. After that, you'll learn how to get screenshots of your app. Finally, you'll learn how to submit your app to the App Store.

This part comprises the following chapters:

- *Chapter 13, Test Your App with TestFlight*
- *Chapter 14, Submit Your App to the App Store*

By the end of this part, you'll be able to test and publish your own apps to the App Store. Let's get started!

Note

Chapter 13, Test Your App with Testflight, and *Chapter 14, Submit Your App to the App Store*, both require a paid Apple Developer Account, which costs USD99 per year. Registering for a paid Apple Developer Account is a serious commercial contract that can include a lot of complexity, and you will enter into a binding legal agreement with Apple. Therefore, you should only sign up if your intention is to eventually publish apps on the App Store. Otherwise, feel free to skip both these chapters.

13

Test Your App with TestFlight

Over the course of this book, you have learned about SwiftUI and how to build an app using Swift Playground. However, so far, you've only been running your app on your iPad.

In this chapter, you will learn how to use **TestFlight** to test your app on other Apple devices. You'll start by obtaining a **paid Apple Developer account**. Next, you'll learn how to prepare and submit your app to **App Store Connect**. After that you'll learn how to use TestFlight to distribute your app to internal and external testers.

By the end of this chapter, you'll know how to prepare and submit your app for testing using TestFlight.

The following topics will be covered:

- Get an Apple Developer account
- Submit your app to App Store Connect
- Test your app with TestFlight

Technical requirements

You will need an Apple Account and a paid Apple Developer account to complete this chapter.

There are no project files for this chapter, as it should be used as a reference on how to submit apps and is not specific to any particular app. Enrollment flows, screens and button names are correct at the time of writing but may change over time.

Let's start by learning how to get a paid Apple Developer account, which is required for App Store Connect submission, in the next section.

Get an Apple Developer account

As you saw in earlier chapters, you can test your app on your own iPad. You can also share your app playground to other iPads and Macs that have Swift Playground installed. However, you're not able to test your app on other Apple devices which do not have Swift Playgrounds installed. To do so, you will need to upload your app to App Store Connect, which is Apple's portal for managing your apps. You can test apps submitted to App Store Connect using Testflight, but you will need a paid Apple Developer account.

Note

For an overview of Apple Developer Program membership, see this link:

<https://developer.apple.com/help/account/membership/programs-overview>.

Follow these steps to purchase an individual/sole proprietorship Apple Developer account:

1. Go to <https://developer.apple.com/programs/> and click on the **Enroll** button.
2. Click **Start your enrollment**.
3. Enter your Apple Account and password when prompted.
4. On the **Trust this browser?** screen, click **Trust** only if you are the only person using this browser; otherwise, click **Not Now**. This is to safeguard your account information.
5. Click **Continue enrollment on the web >**.
6. On the **Confirm your personal information** screen, enter your personal information and click **Continue**.
7. On the **Select your entity type** screen, choose **Individual/Sole Proprietor**. Click **Continue**.
8. On the **Review and Accept** screen, check the checkbox at the bottom of the page and click **Continue**.
9. On the **Complete your purchase** screen, click **Purchase**.

10. Follow the onscreen directions to complete your purchase. Once you have purchased your account, go to <https://developer.apple.com/account/> and sign in with the same Apple Account that you used to purchase your Developer account. You should see something like the following:

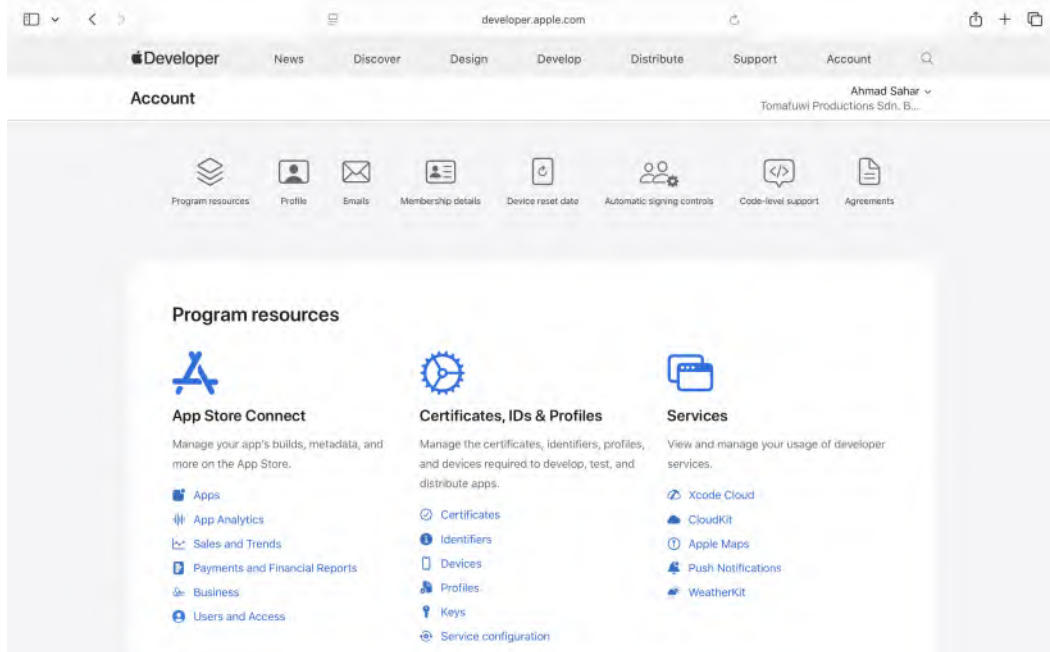


Figure 13.1: Apple Developer website with a paid Apple Developer account logged in

Now that you have a paid Apple Developer account, let's learn how to prepare and submit your app to App Store Connect in the next section.

Submit Your App to App Store Connect

You are now ready to submit your app to App Store Connect! In this section, the *CatLog* app will be used as an example.

Note

For more details on how to use Swift Playground to submit your app to App Store Connect, see this link: <https://support.apple.com/en-gb/guide/playgrounds-ipad/itc65b2d9a15/ipados>.

Before you submit your app, make sure that it runs successfully on your iPad. Then, you can use Swift Playground to submit your app. Follow these steps:

1. You must create an icon for your app, as a placeholder icon is not allowed. Create an icon for your app that is 1,024 x 1,024 pixels and save it to the **Photos** app on your iPad.

Note

For more details on how to create icons for your app, you can refer to Apple's Human Interface Guidelines here:
<https://developer.apple.com/design/human-interface-guidelines/app-icons>.

Tip

You can use *Image Playground* on your iPad to generate an icon for your app.

2. In your app playground, tap the **App Settings** button. Tap **Custom** and tap **Import from Photos...** to import your app icon:

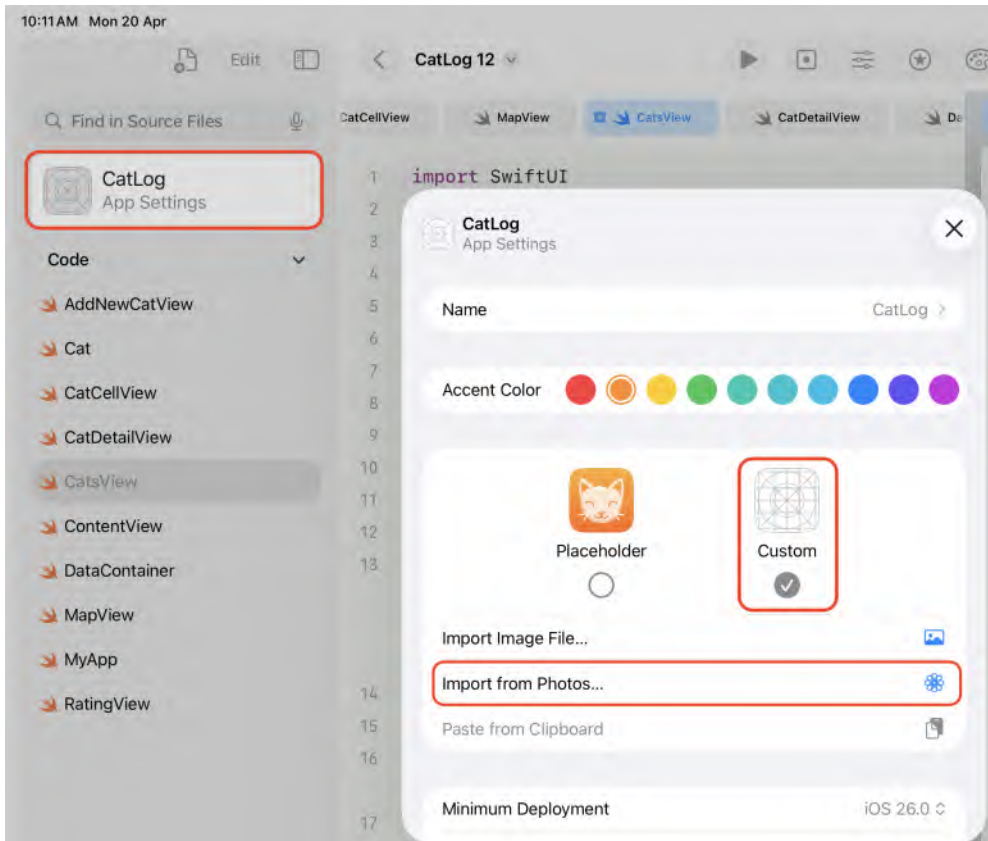


Figure 13.2: App Settings window showing Custom icon radio button and Import from Photos... button

3. Choose the icon that you saved to your Photos app earlier:

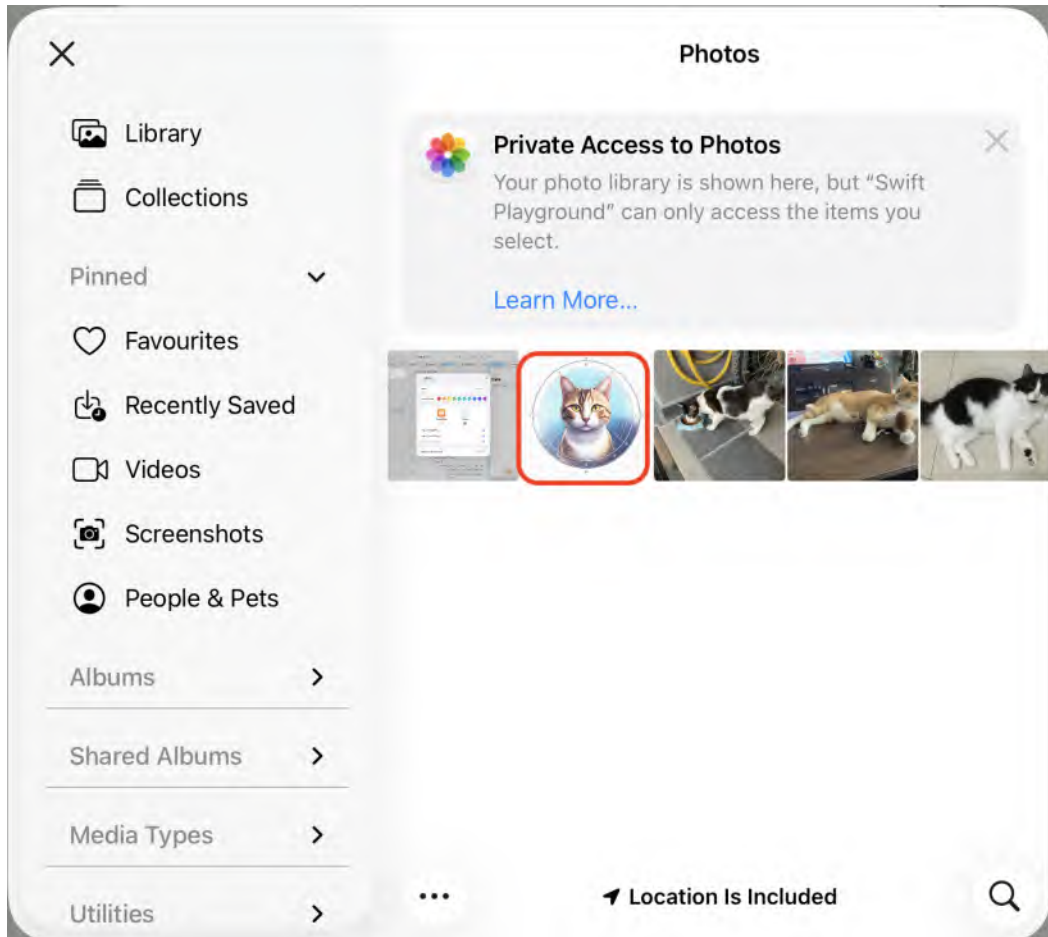


Figure 13.3: The photos picker window showing your app icon

4. Since your app requires access to your device location, you must add the Location capability to your app. Tap the **Capabilities** button:

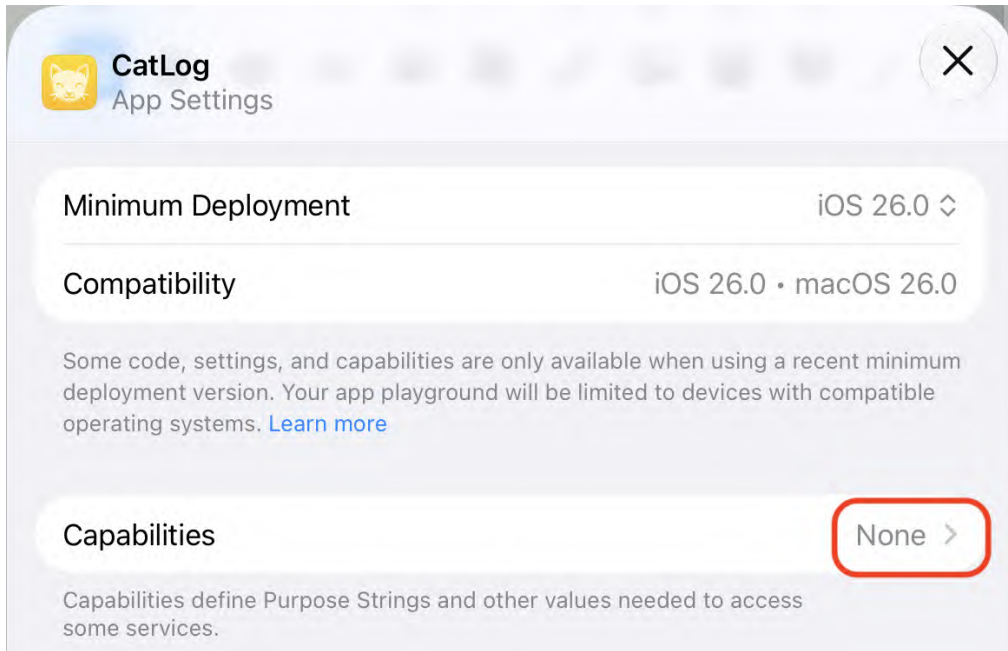


Figure 13.4: Capabilities button in the App Settings window

5. Tap the + button in the **Capabilities** window to add a capability:

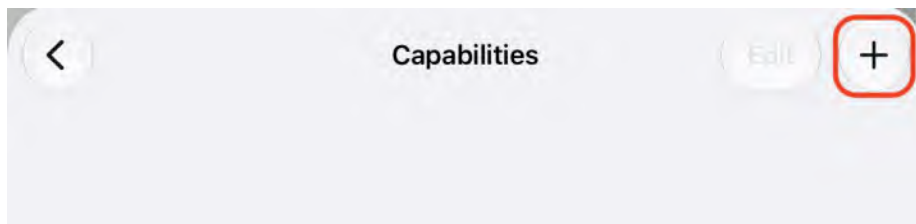


Figure 13.5: Capabilities window showing the + button

6. Choose **Core Location When in Use** from the pop-up menu:

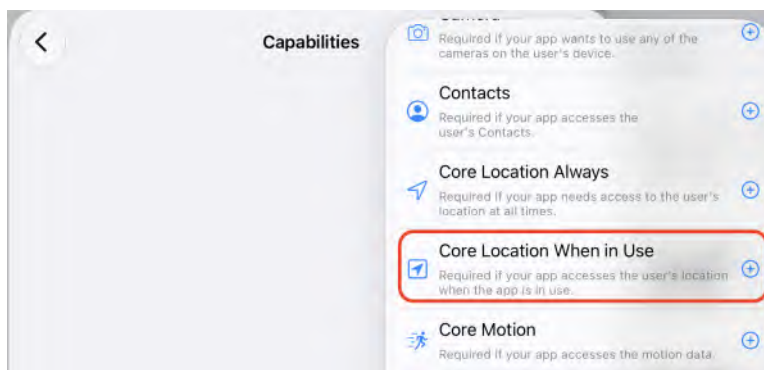


Figure 13.6: Capabilities window showing a pop-up menu

7. Enter a **Purpose String** in the provided field and tap **Add**:

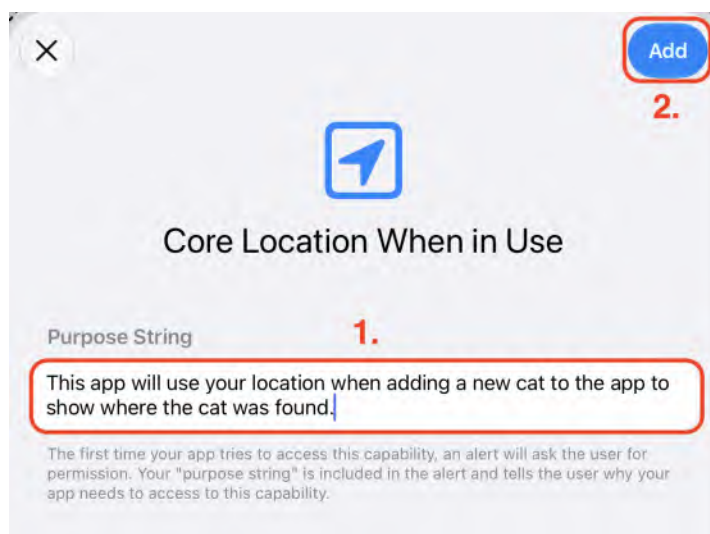


Figure 13.7 Core Location When in Use window showing field for Purpose String

When you turn on the **Get Location** switch in the **Add New Cat** screen or go to the **Map** screen for the first time, an alert displaying the purpose string will appear asking for permission to access the device location.

8. Tap the Back button to exit the **Capabilities** window:



Figure 13.8 Capabilities window showing Back button

9. Tap the **Sign In** button at the bottom of the **App Settings** window:

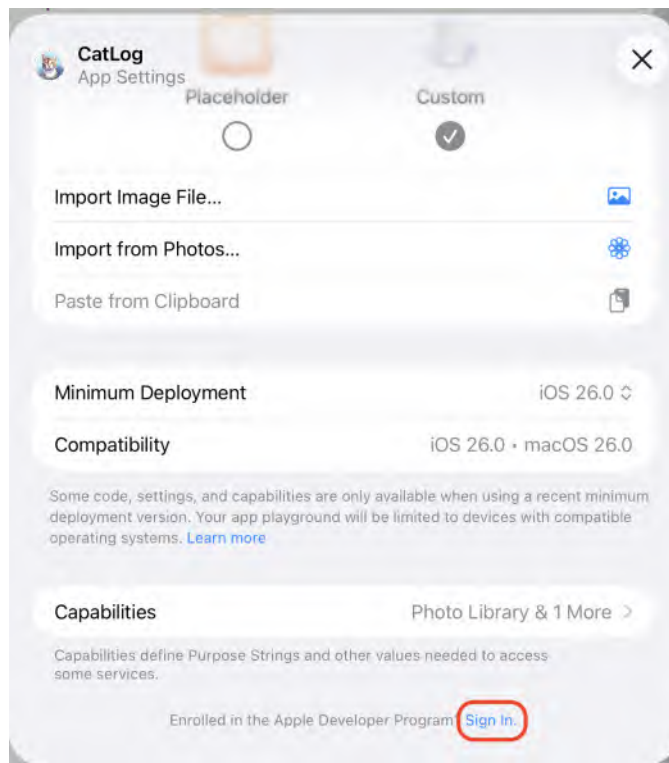


Figure 13.9: App Settings window showing Sign In button

10. Enter the Apple Account username and password that you used to create your paid Apple developer account in the text fields provided and tap the **Sign In** button:

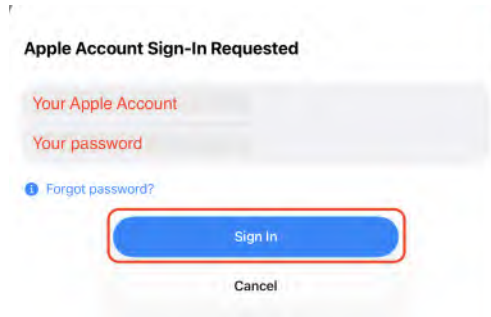


Figure 13.10: Sign In screen showing text fields for your Apple Account username and password

11. In the **App Settings** windows, tap the **Team & Bundle Identifier** button:

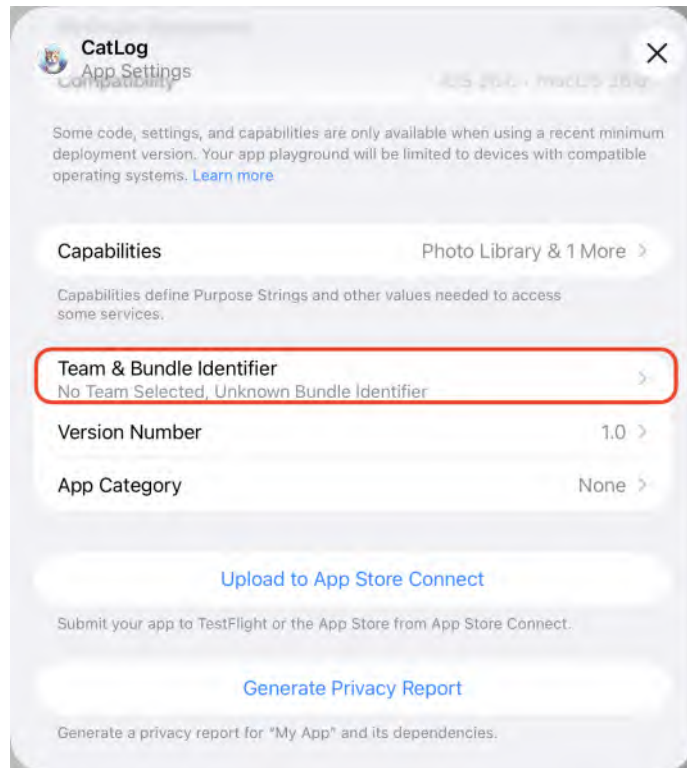


Figure 13.11: App Settings window showing Team & Bundle Identifier button

12. In the **Team & Bundle Identifier** window, select your paid developer account from the **Team** dropdown and enter the Bundle ID for your app, which consists of a string of characters in reverse-DNS format. A typical Bundle ID will look like `com.companyname.appname` as shown in the screenshot. Note that Bundle IDs must be globally unique within a developer account and cannot be changed after an app has been distributed. Tap **Save** when done:

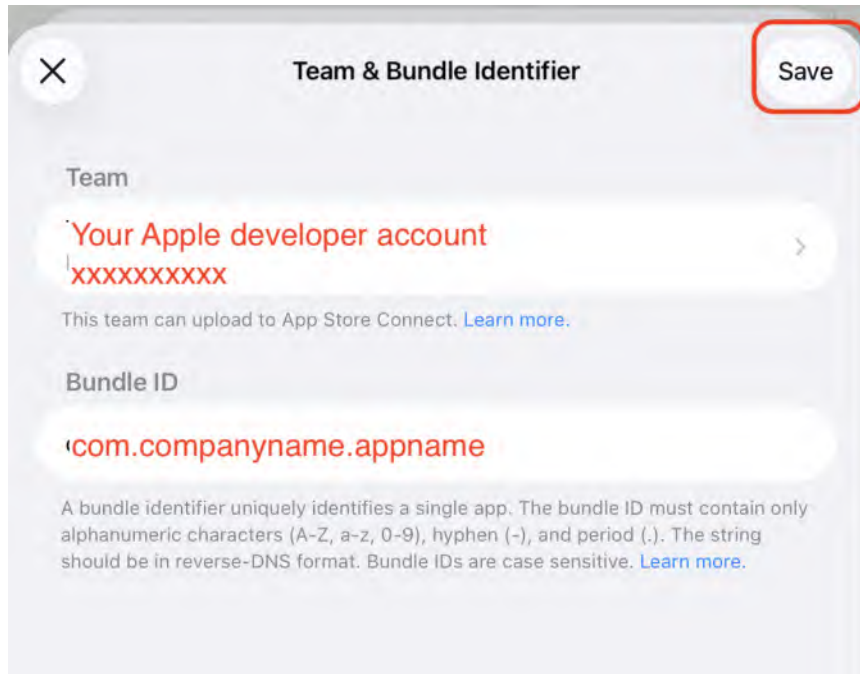


Figure 13.12: Team & Bundle Identifier window showing Save button

13. In the **App Settings** window, choose an app category from the pop-up menu:

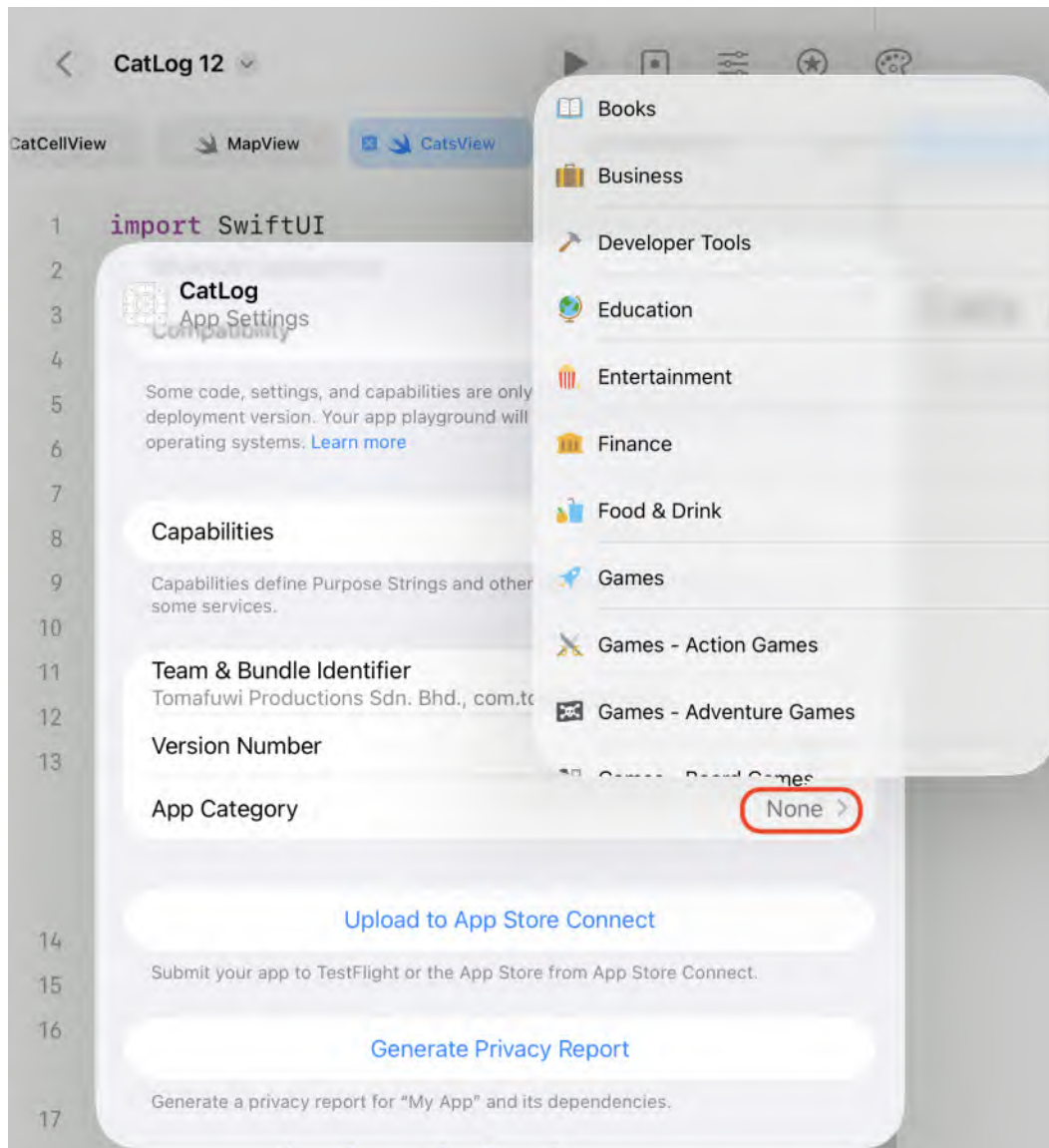


Figure 13.13: App Settings windows showing App Category pop-up menu

14. Tap the **Upload to App Store Connect** button to start the app submission process:

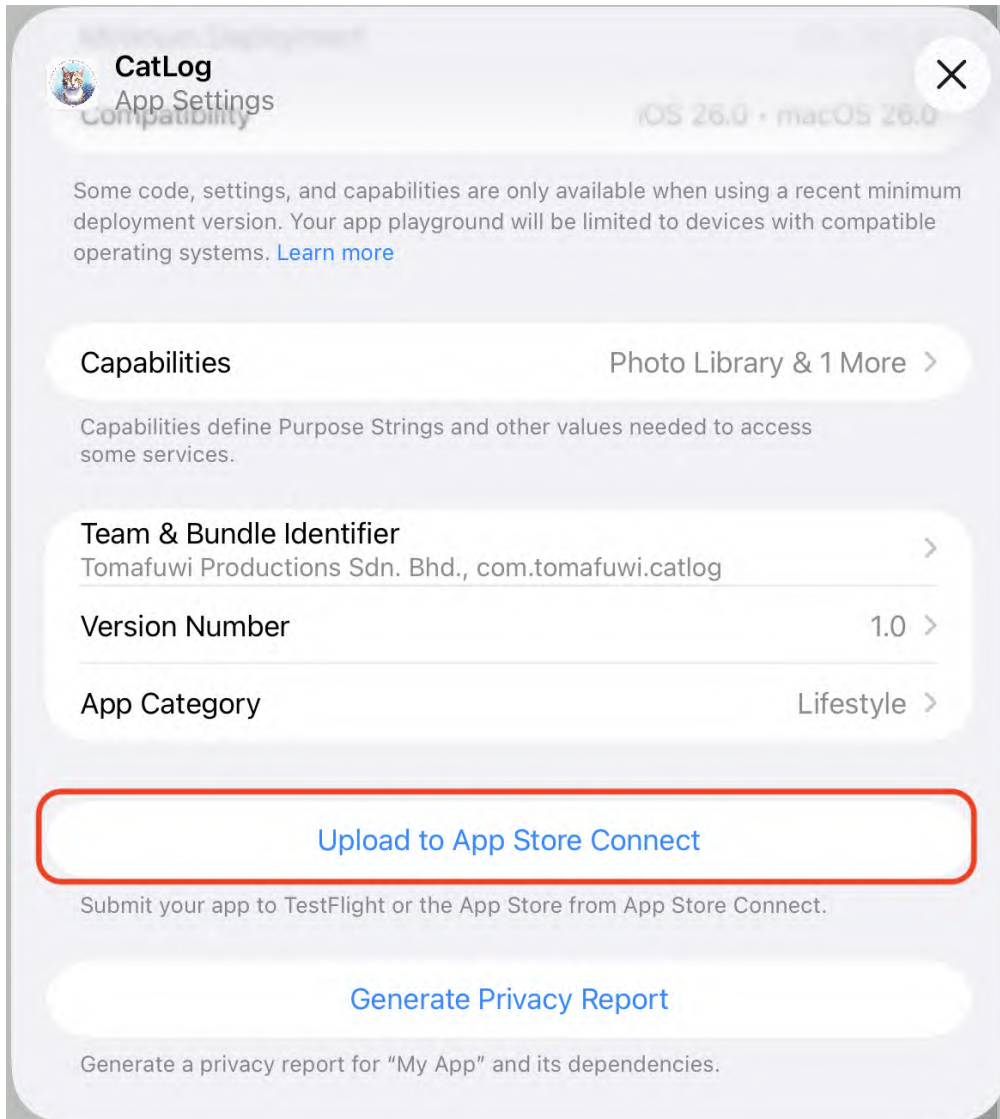
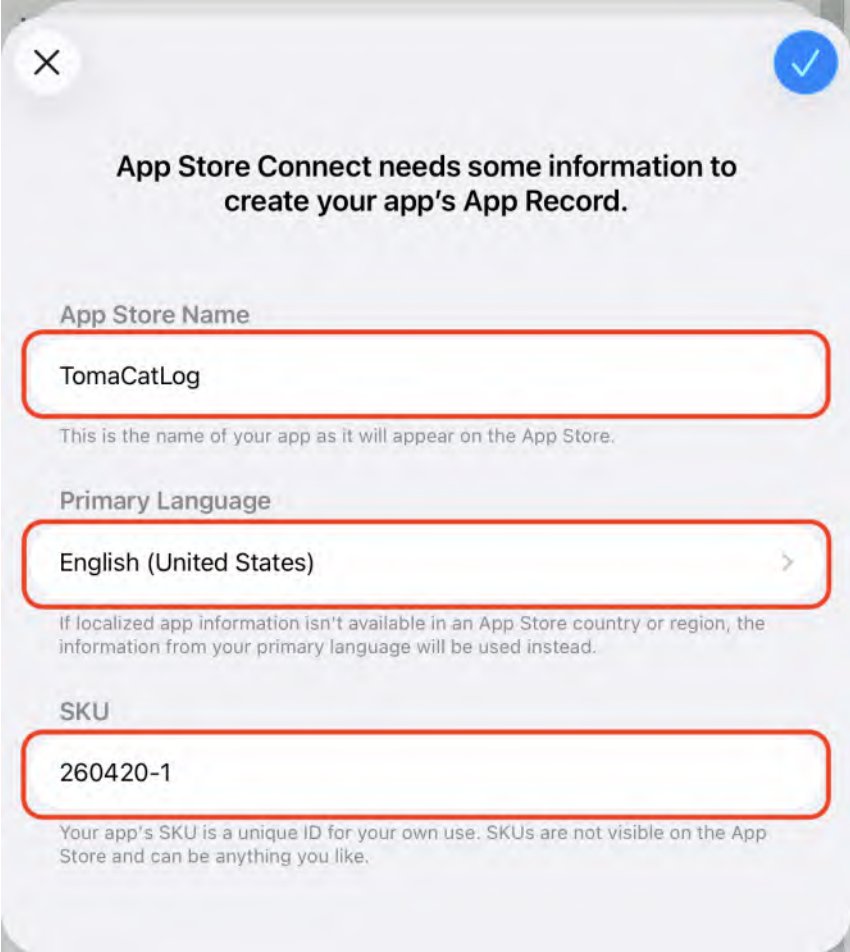


Figure 13.14: App Settings window showing the Upload to App Store Connect button

15. App Store requires your app's name, primary language, and SKU. The SKU value can be any string value, such as the date the app is submitted combined with the version number. Enter the required information and tap the Save button:



App Store Connect needs some information to create your app's App Record.

App Store Name

TomaCatLog

This is the name of your app as it will appear on the App Store.

Primary Language

English (United States) >

If localized app information isn't available in an App Store country or region, the information from your primary language will be used instead.

SKU

260420-1

Your app's SKU is a unique ID for your own use. SKUs are not visible on the App Store and can be anything you like.

Figure 13.15: App Store Connect window

The upload process will begin automatically. Wait for it to finish.

16. Once the upload has been completed, you can tap **Done** or tap **Show in App Store Connect**. Tap **Done** for now:

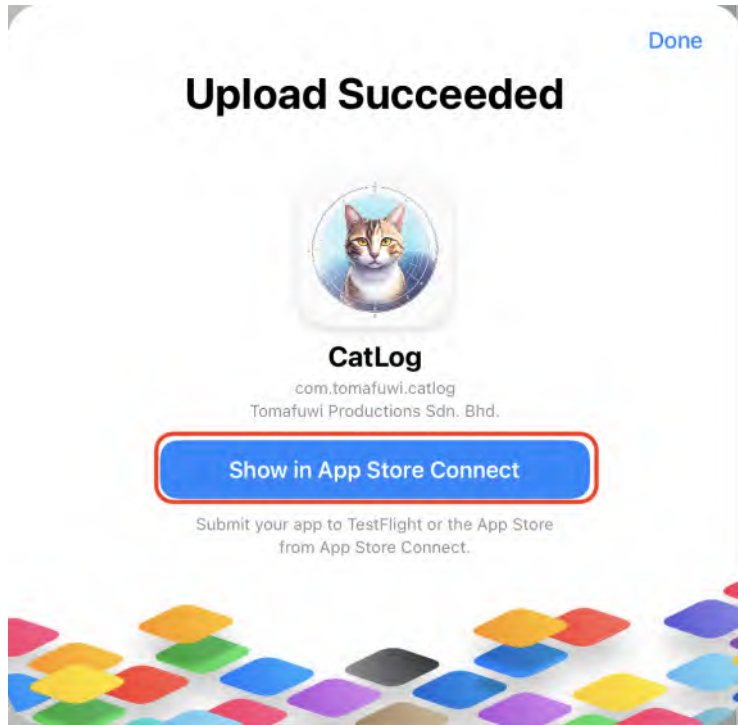


Figure 13.16: The Organizer window with the Distribute App button selected

If you tap **Show in App Store Connect**, your iPad will either show you the App Store Connect web interface or launch the *App Store Connect* app. The app is recommended as it presents a more streamlined interface for TestFlight.

Note

You can download App Store Connect app from the App Store at this link: <https://apps.apple.com/my/app/app-store-connect/id1234793120>.

You now know how to submit your app to App Store Connect! Awesome!

In the next section, you'll learn how to conduct internal and external testing for your app, which is important in ensuring that the app is high-quality and bug-free.

Test your app with TestFlight

By default, apps created with Swift Playground are compatible with iOS and macOS devices. Apple has a service named **TestFlight** that allows you to distribute your apps to testers prior to releasing them to the App Store. You'll need to download the *TestFlight* app, available from <https://developer.apple.com/testflight/> to test your app. Your testers can be both members of your internal team (internal testers) or the public (external testers).

Note

To learn more about how to use TestFlight, watch this video:

<https://developer.apple.com/videos/play/tech-talks/110343/>.

First, let's see how to allow internal team members to test your app in the next section.

Test your app internally

Internal testing should be performed when the app is in an early stage of development. It only involves members of your internal team; Apple does not review apps for internal testers. You can send builds to up to 100 testers for internal testing. Internal testers must be App Store Connect users who have access to the app and an eligible role. If you have just purchased your Apple Developer account, you will be the only one eligible. To create a new internal team to test your app, follow these steps:

1. Launch the App Store Connect app on your iPad and choose your app in the left column. Your app details will appear in the right column. Tap the **Missing Compliance** button:

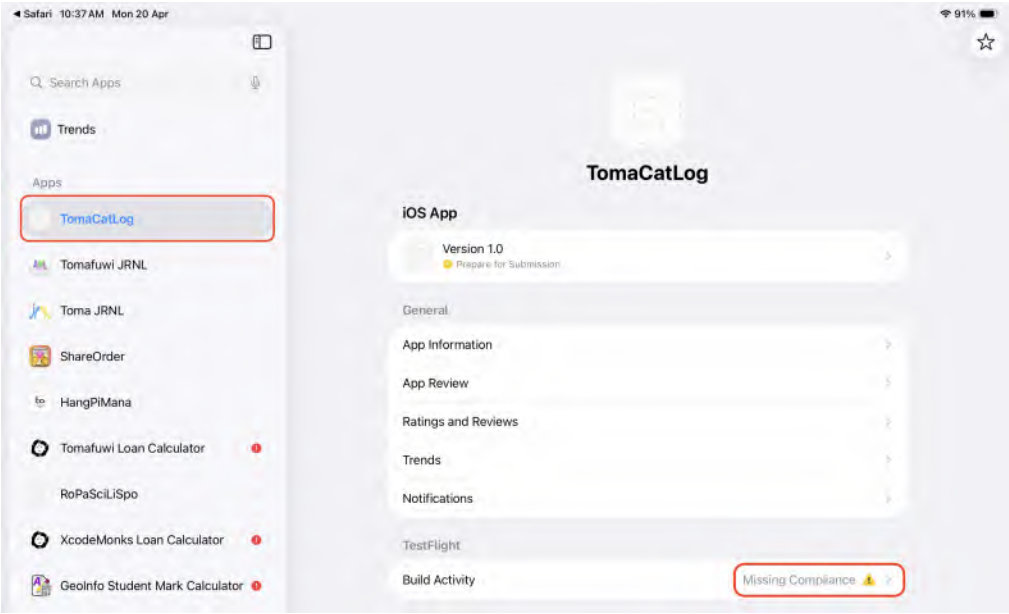


Figure 13.17: App Store Connect app showing your app details

2. In the **Build Activity** screen, tap the warning icon:

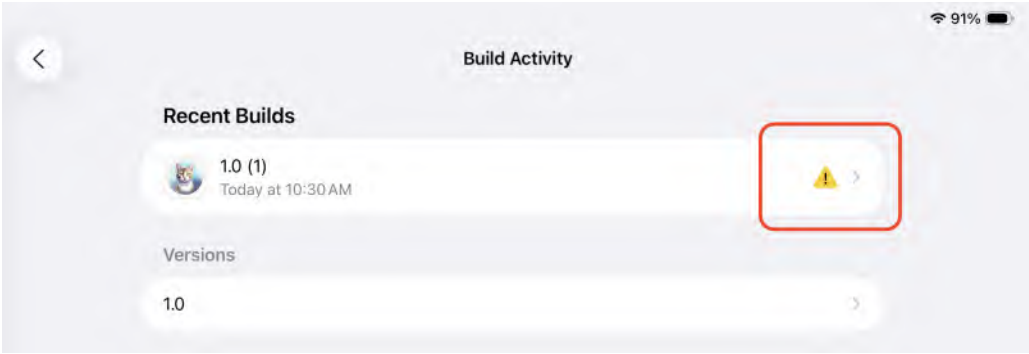


Figure 13.18: Build activity screen showing warning icon

3. The reason you see the warning icon is because you must declare that your app does not use encryption. Tap the **Get Started** button:

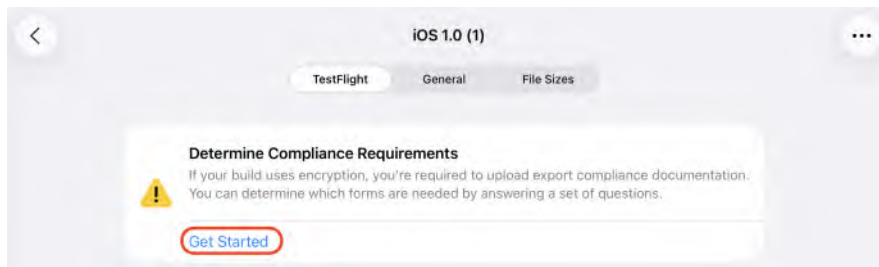


Figure 13.19: Screen showing Get Started button

4. Since the *CatLog* app does not use encryption, in the **Export Compliance Information** window, choose **None of the algorithms mentioned above** and tap the **Save** button:

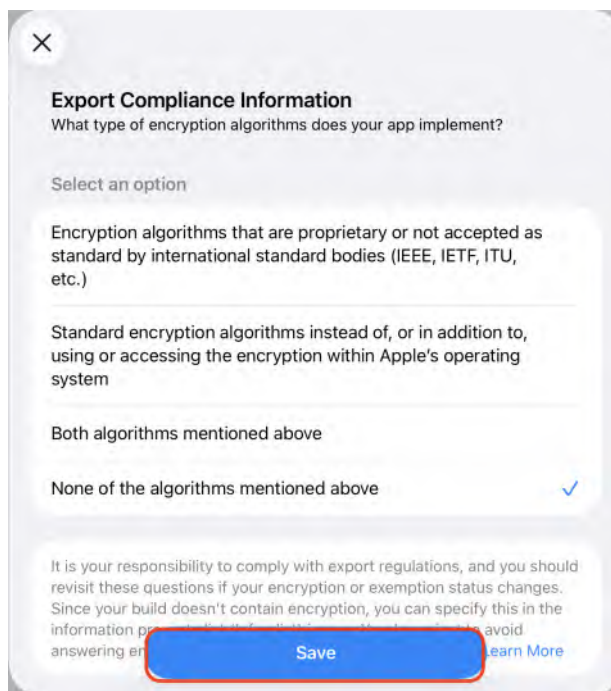


Figure 13.20: Export Compliance Information window showing Save button

5. In the app details screen for your app, tap the **Create Internal Group** button to create a new internal test group:

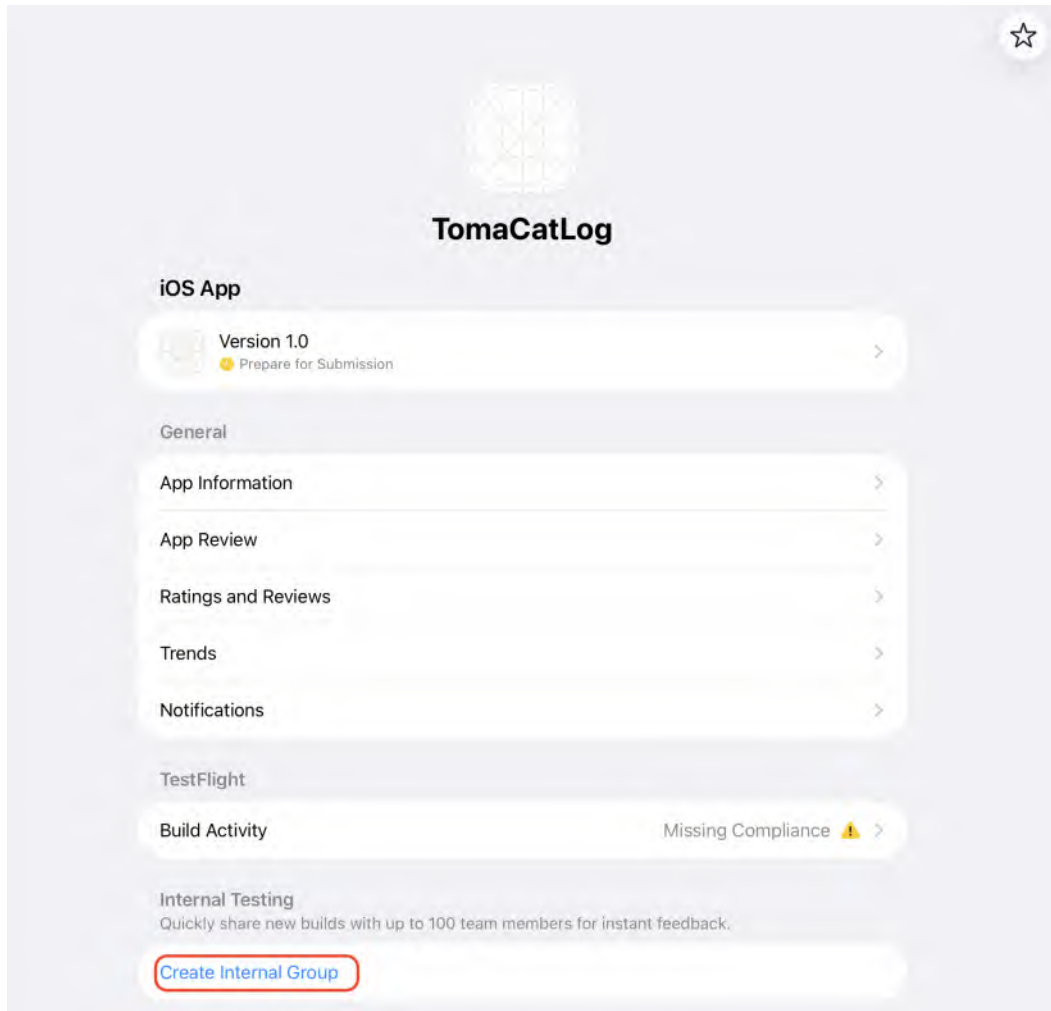


Figure 13.21: App details screen for your app showing the Create Internal Group button

6. The **New Internal Group** dialog box will appear. Name your internal group and tap the Save button:

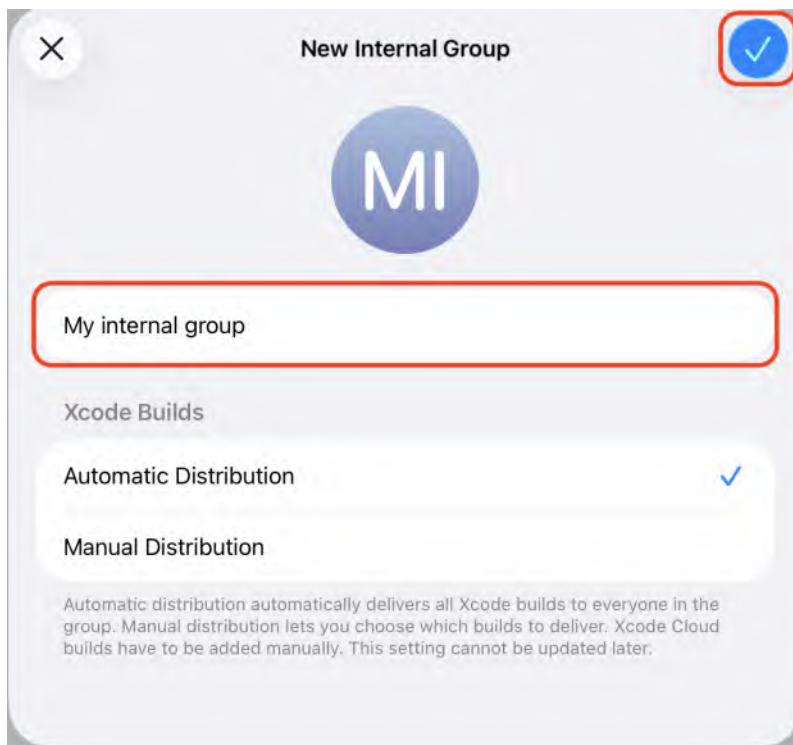


Figure 13.22: The New Internal Group dialog box

7. After your internal group has been created, tap the **Add Testers** button:

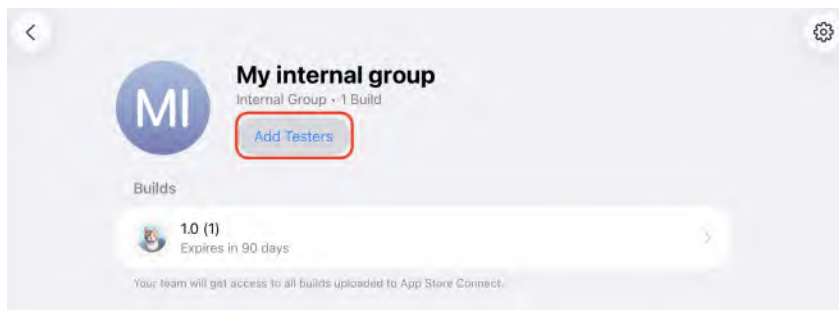


Figure 13.23: Internal group screen showing Add Testers button

If you have just purchased your Apple Developer account, you will be the only user available.

8. Select the pane containing your name and email address, and tap the Save button:

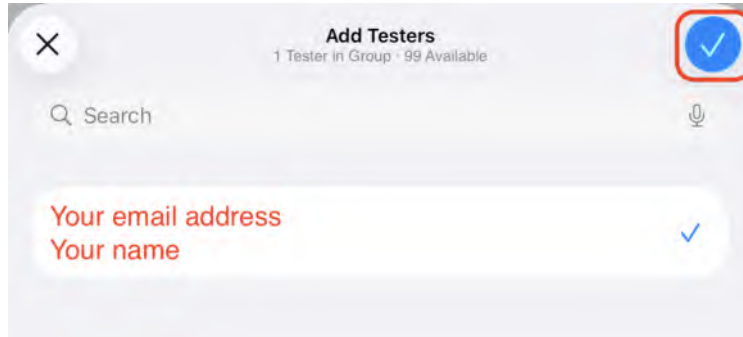


Figure 13.24: Add Testers screen showing your name and email address

9. You will be returned to the internal group screen, which shows that you have been added as a tester. Tapping the icon on the right side of the pane containing your email displays a pop-up menu. Note that the invite to test the app has already been sent automatically:

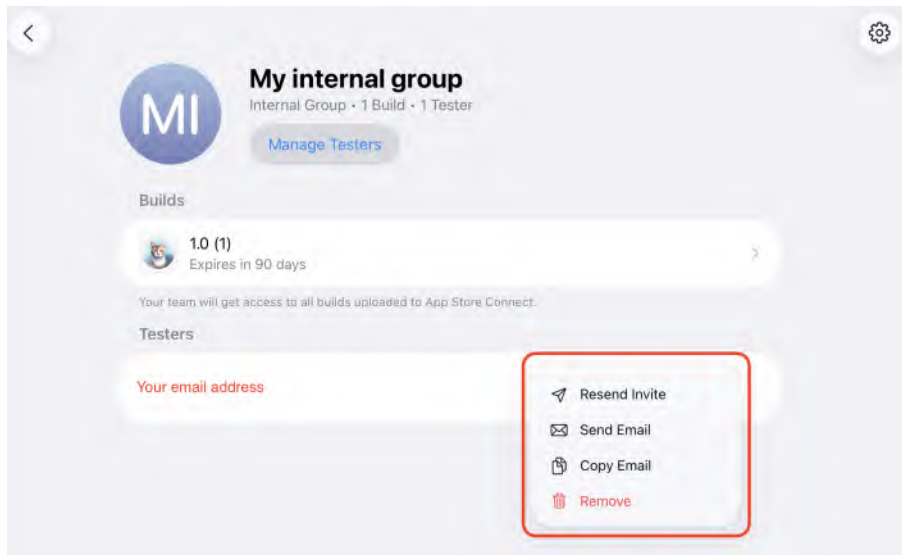


Figure 13.25: The internal group screen showing you have been added as a tester

10. You will receive an email inviting you to test the *CatLog* app. Tap the **View in Testflight** button:

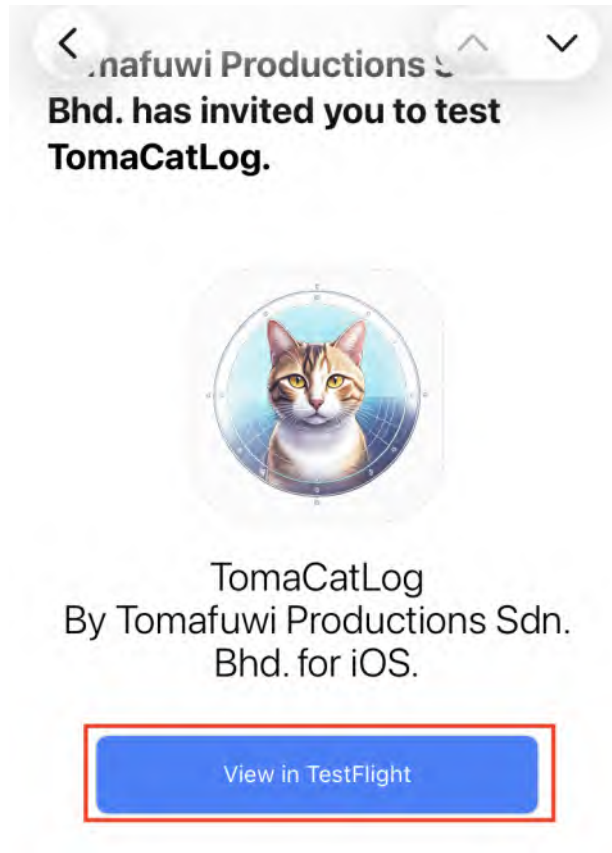


Figure 13.26: Email invite to test the CatLog app

You will need to download and install the *TestFlight* app to test the *CatLog* app. If you have not yet done so, you will be prompted to download it.

11. If you have the TestFlight app already installed, it will launch and prompt you to install the *CatLog* app as shown:

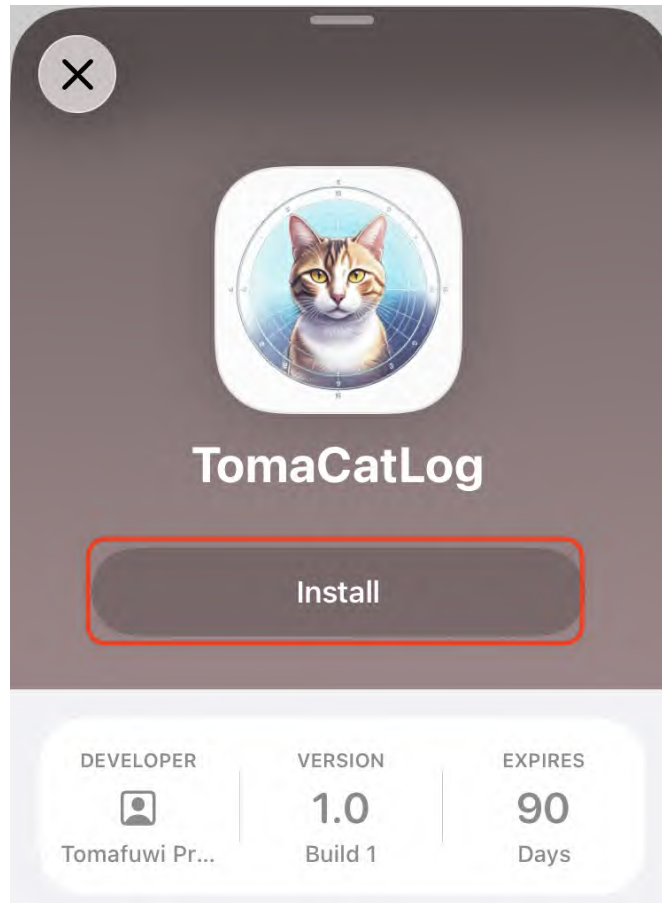


Figure 13.27: TestFlight app showing the Install button

12. The *CatLog* app will be installed and launched. You can now test your app:

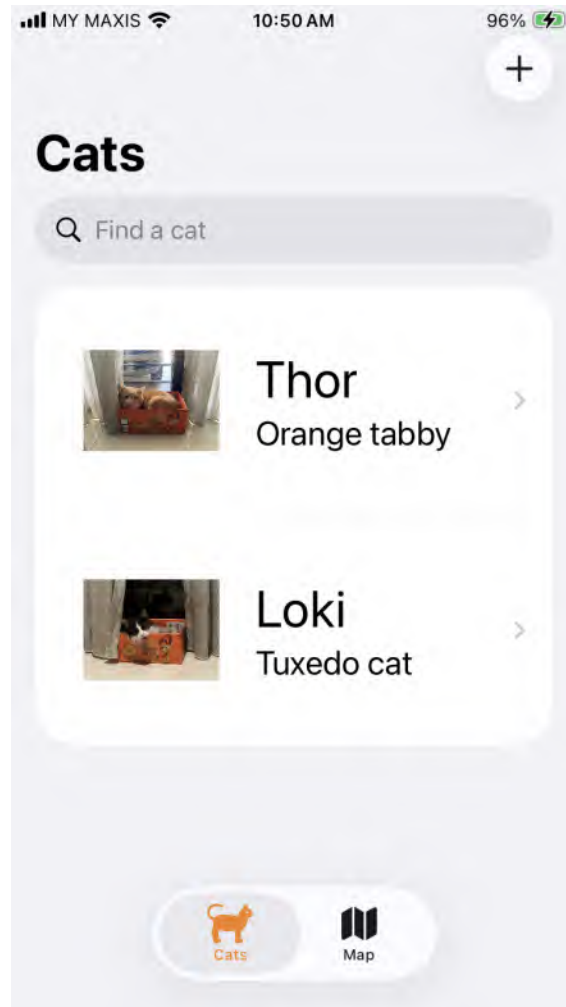


Figure 13.28: Your app running on your device

Remember that internal testing will only involve members of your team. If you want to conduct testing with more than 100 testers, you will need to do external testing, which is described in the next section.

Test your app externally

External testing should be performed when the app is in the final stages of development. You can select anyone to be an external tester, and you can send builds to up to 10,000 testers. Apps for external testers will need to be submitted for review by Apple. Note that this review is not the same as the final app review that Apple will do once your app has been submitted to the App Store. Follow these steps:

1. Launch the *App Store Connect* app, select your app in the left pane, and tap the **Add Test Information** button:

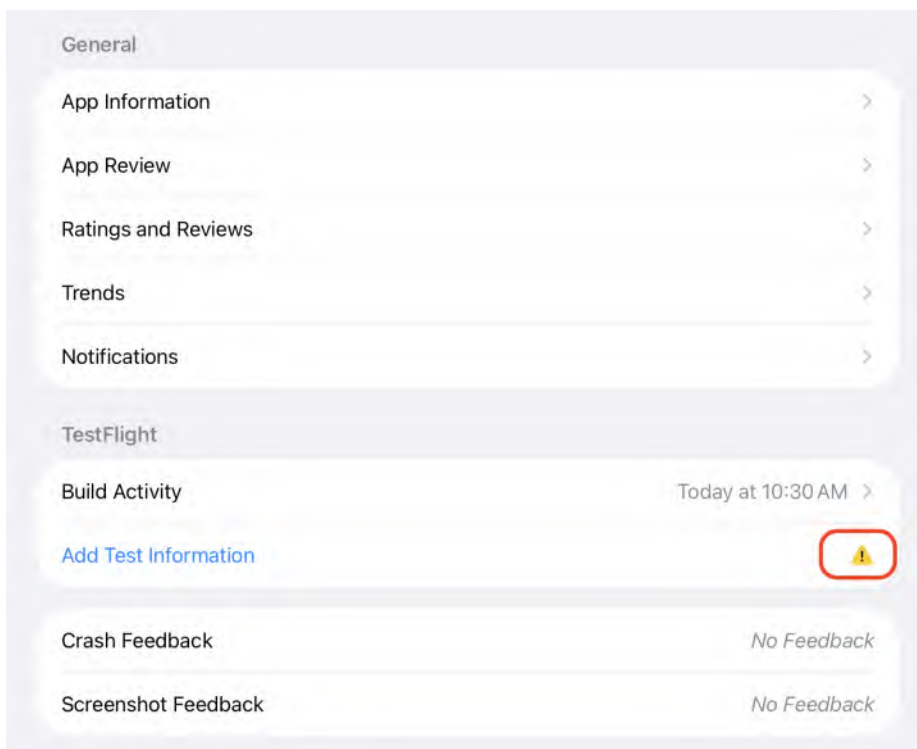
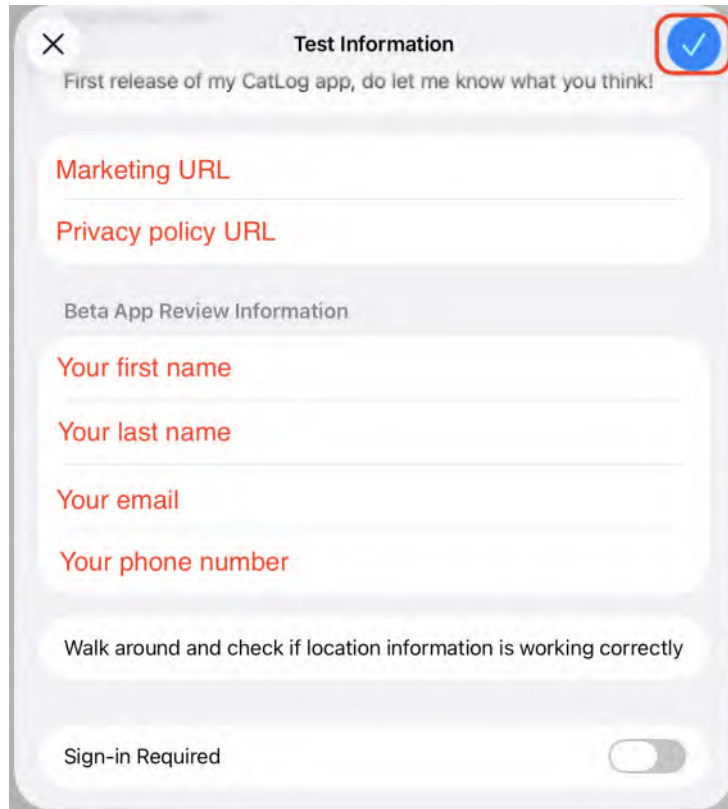


Figure 13.29: The TestFlight screen showing the Add Test Information button

This is required before sending the app to external testers.

2. In the **Test Information** dialog box, enter all the required information as shown, and tap the Save button:



Test Information

First release of my CatLog app, do let me know what you think!

Marketing URL

Privacy policy URL

Beta App Review Information

Your first name

Your last name

Your email

Your phone number

Walk around and check if location information is working correctly

Sign-in Required

Figure 13.30: Test Information dialog box

3. You will be returned to the app details screen for your app. Tap the **Create External Group** button:

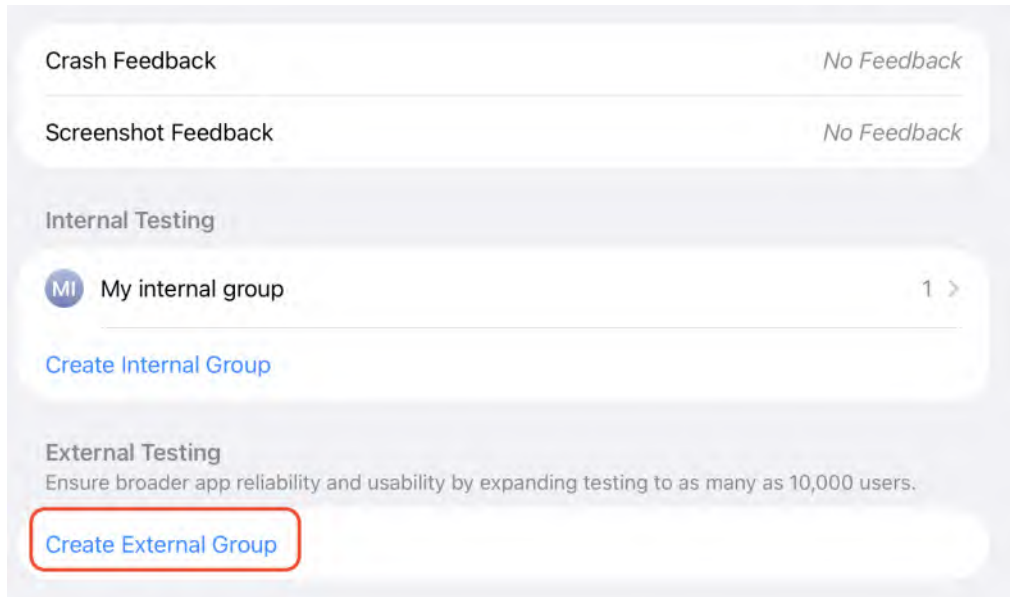


Figure 13.31: App details screen showing Create External Group button

4. Type in a name for the external group and tap the Save button:

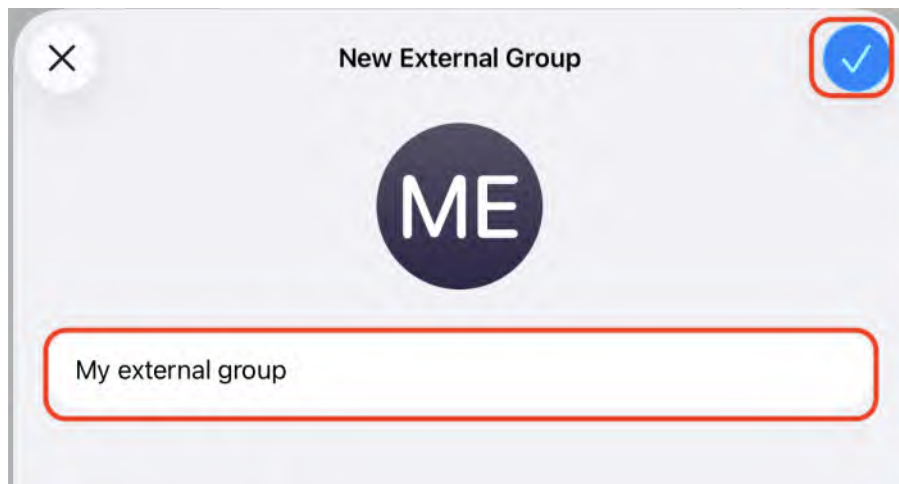


Figure 13.32: The New External Group screen

5. Click the **Add Build** button to choose a build that you want your testers to test:

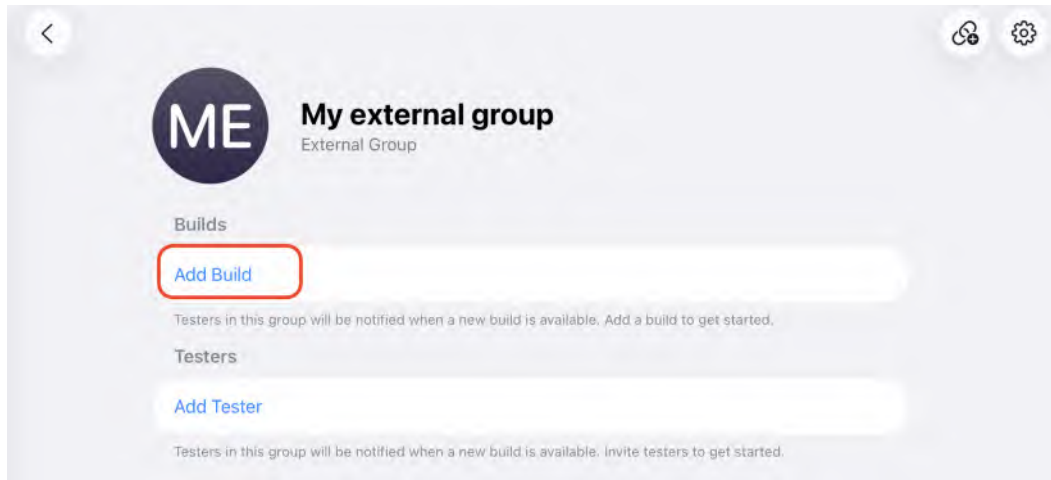


Figure 13.33: External group screen showing the Add Build button

6. Choose the app you just uploaded and tap the **Next** button:

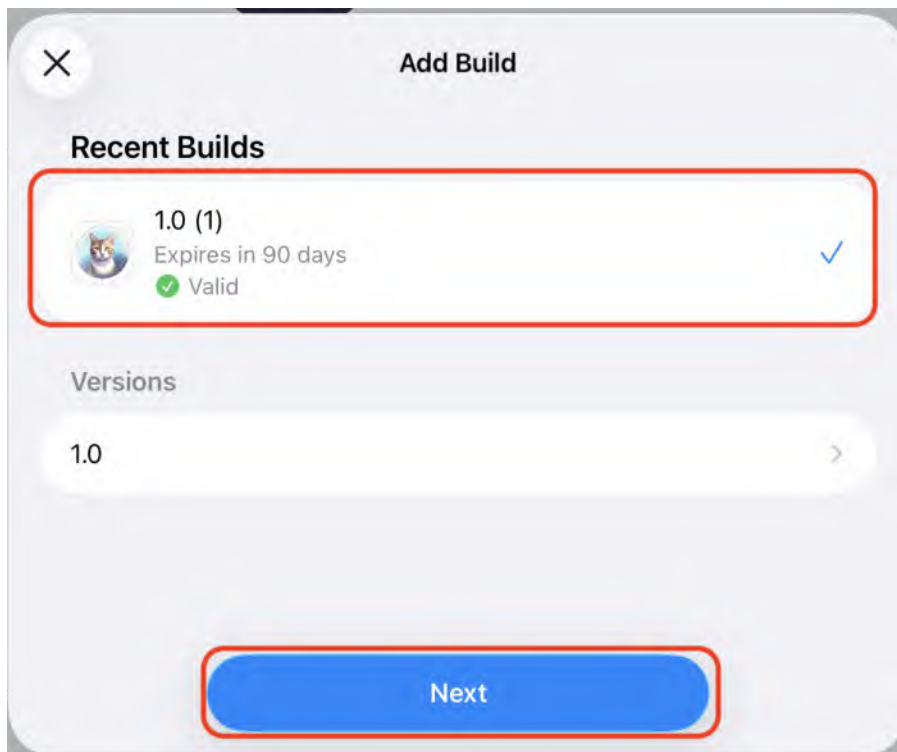


Figure 13.34: The Add Build screen

7. Type in what you would like your testers to test in the text field provided and tap **Submit**:

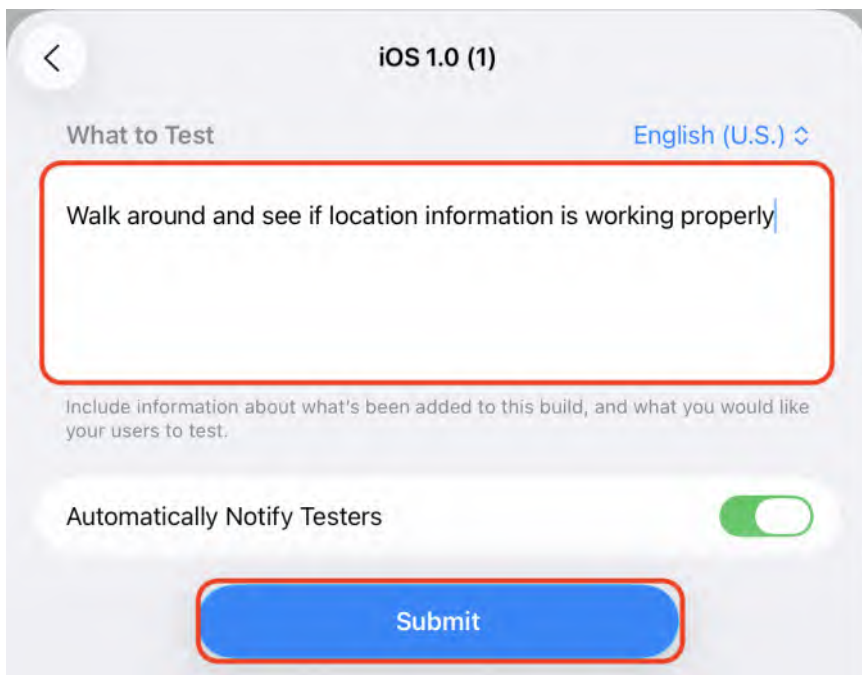


Figure 13.35: The What to Test screen

Apple may review the test build prior to making it available to your testers. If your app is rejected, you will need to fix the issues and resubmit your app.

8. You will be returned to your external group screen. Tap the **Add Tester** button:

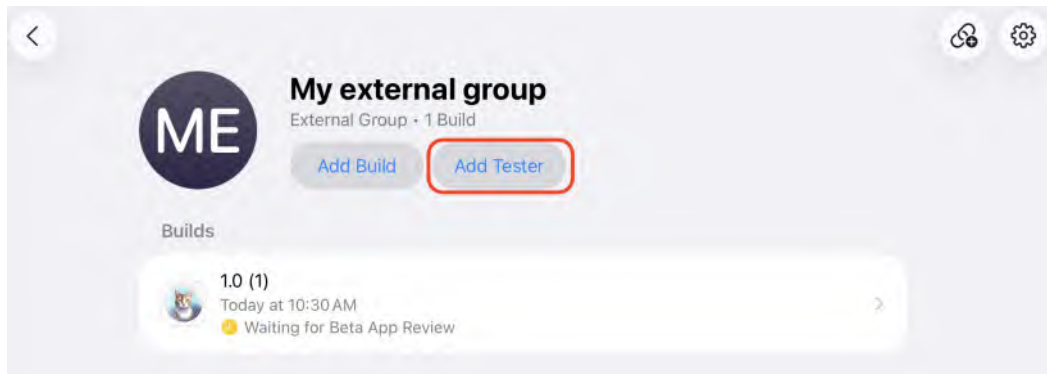


Figure 13.36: The external group screen showing the Add Tester button

9. Enter the name and email address of your tester and tap the Save button:

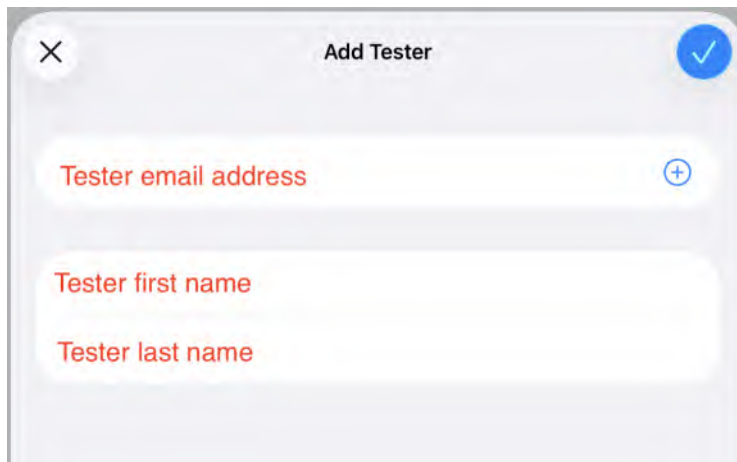


Figure 13.37: The Add Tester screen

10. Apple will notify your external testers automatically when a build has been reviewed and ready to be tested:

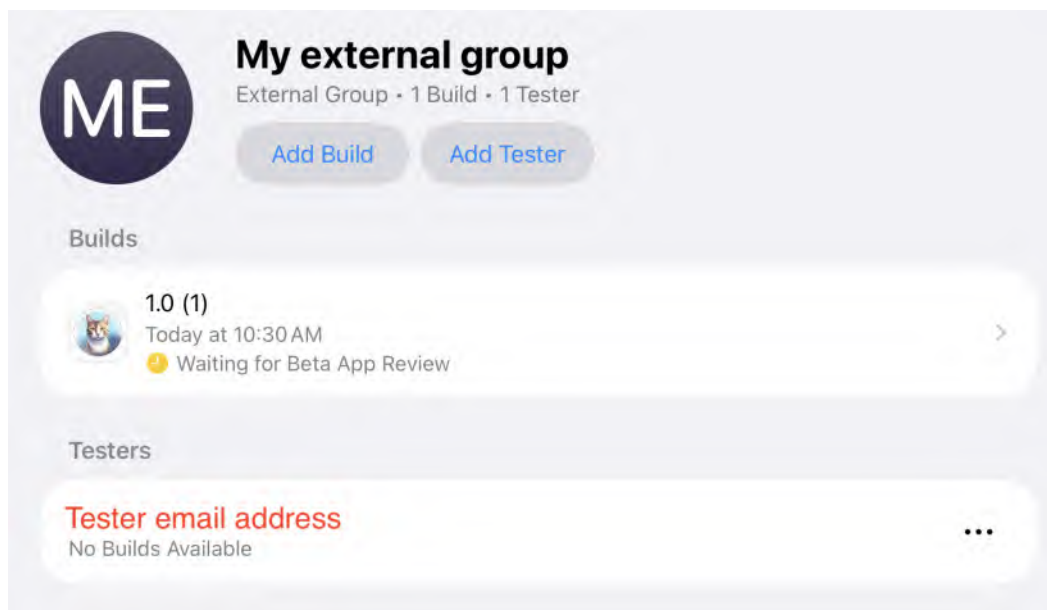


Figure 13.38: External test group screen

Great! You now know how to test your apps internally and externally.

Summary

In this chapter, you learned how to use **TestFlight** to test your app on other Apple devices. You started by obtaining a **paid Apple Developer account**. Next, you learned how to prepare and submit your app to **App Store Connect**. After that you learn how to use TestFlight to distribute your app to internal and external testers.

You now know how to prepare and submit your app for testing using TestFlight.

In the next chapter, you'll learn how to submit your app to the App Store.

14

Submit Your App to the App Store

Congratulations, you have reached the final chapter of this book!

In the previous chapter, you learned how to submit your app to App Store Connect for testing using TestFlight.

In this chapter, you will learn how to publish your app to the App Store. You'll start by learning how to get screenshots of your app. After that, you'll learn how to submit your app for App Review. If your app is approved by Apple, it will automatically be available on the App Store.

By the end of this chapter, you'll learn how to submit your own apps to the App Store.

The following topics will be covered:

- Get screenshots of your app
- Submit your app for App Review

Technical requirements

You will need an Apple Account and a paid Apple Developer account to complete this chapter.

There are no project files for this chapter, as it should be used as a reference on how to submit apps and is not specific to any particular app. Enrollment flows, screens, and button names are correct at the time of writing but may change over time.

Let's start by learning the requirements for your app submission in the next section.

Get screenshots of your app

You are now ready to submit your app to the App Store! In this section, the *CatLog* app will be used as an example. Before you submit your app, you must get screenshots of your app. Screenshots of your app are required to show your potential customers what your app looks like and how it works. Your screenshots should highlight your app's features and how your app looks on different screen sizes.

At present, Apple requires screenshots of your app running on an iPhone with a 6.5" display (1284 x 2778 pixels or 1242 x 2688 pixels). If your app runs on iPad as well, Apple requires screenshots of your app running on an iPad with a 13" display (2064 x 2752 pixels or 2048 x 2732 pixels). Note that this may change in future and you should verify the required dimensions in App Store Connect before submission.

Note

For more details on screenshot requirements, see this link:

<https://developer.apple.com/help/app-store-connect/reference/app-information/screenshot-specifications>.

If you don't have actual iOS or iPadOS devices, one way to get screenshots is to run your app on Device Hub. You will need a Mac running *Xcode*, or you could use a service such as Rent a Mac (<https://rentamac.io/>). If you have a Mac with *Xcode* installed, follow these steps:

1. Launch the *Swift Playground* app on the iPad you were using for development. Tap and hold the *CatLog* app playground. Choose **Share** from the pop-up menu:



Figure 14.1: Swift Playground showing pop-up menu with Share highlighted

2. Choose **AirDrop** from the sharing pop-up window:

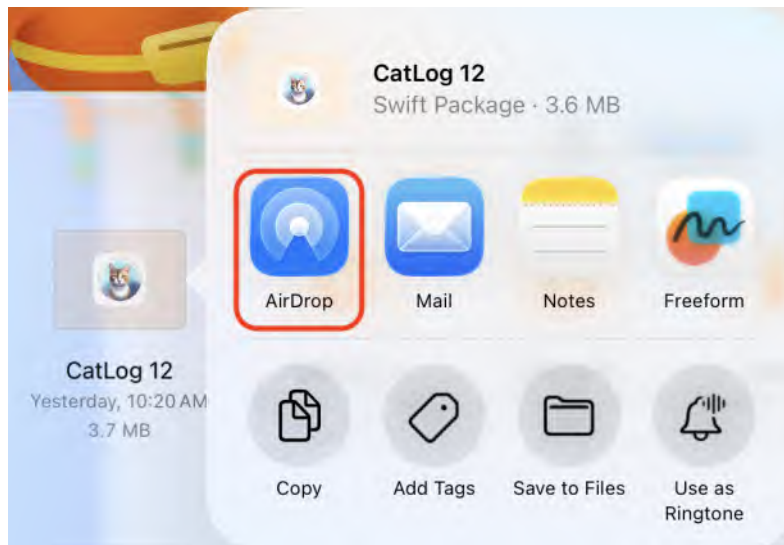


Figure 14.2: Sharing pop-up window with AirDrop highlighted

3. Choose your Mac from the **Other Devices** section:

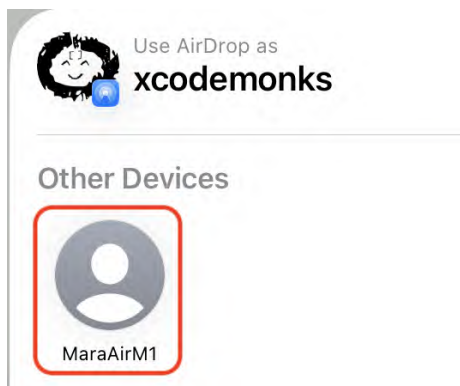


Figure 14.3: Other devices section showing a Mac

4. A pop-up window will appear on your Mac. Click **Accept**:

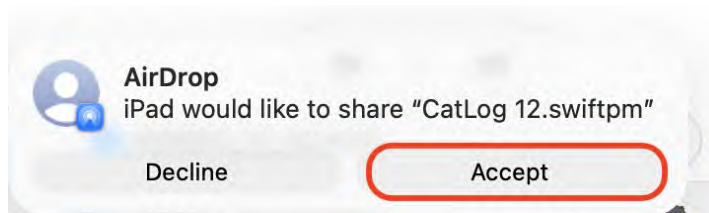


Figure 14.4: Pop-up window on your Mac with Accept button highlighted

5. Verify that the **CatLog** app playground is in your Mac's Download folder:

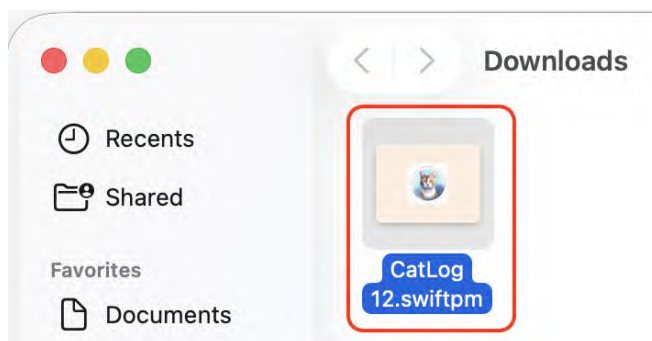


Figure 14.5: Download folder showing the CatLog app playground

6. Right-click on the app playground and choose **Open With > Xcode.app** from the pop-up menu:

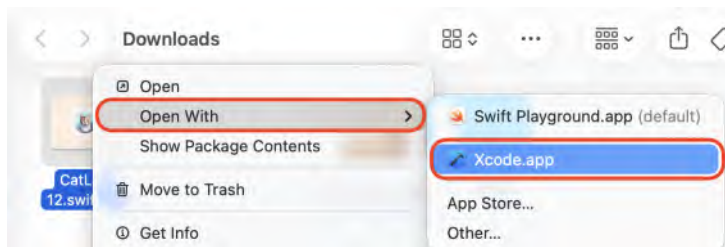


Figure 14.6: Pop-up menu showing Open With > Xcode.app

7. An alert window will appear. Click **Trust and Open**:



Figure 14.7: Alert showing Trust and Open button highlighted

8. *Xcode* will launch and display your app playground. Choose a simulated iOS device with the largest screen from the **Destination** menu:

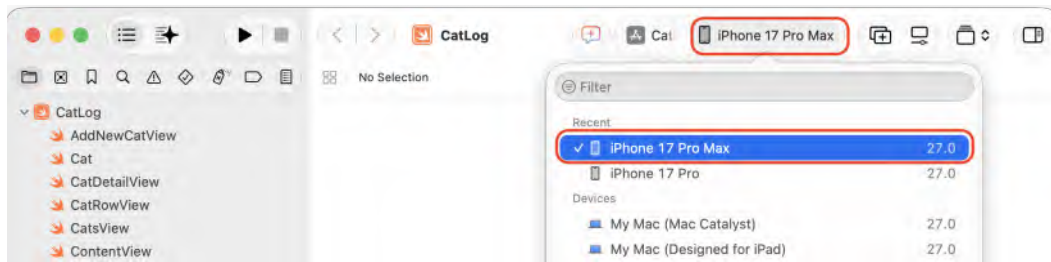


Figure 14.8: Destination menu showing simulated Apple devices

9. Click the Run button to run your app playground on your simulated device:



Figure 14.9: Xcode showing Run button highlighted

10. In device hub, click the Camera button to take app screenshots. It will be saved to your Mac's desktop:

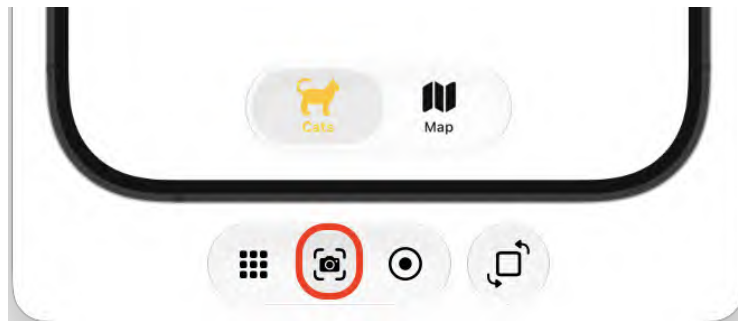


Figure 14.10: Device Hub showing Camera button highlighted

11. After you have taken your screenshots, click the Stop button in *Xcode* to stop running your app playground:

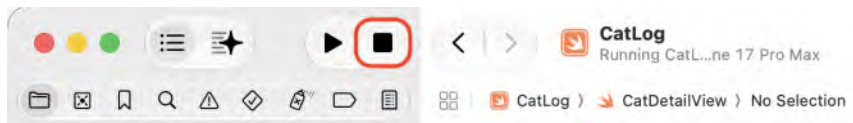


Figure 14.11: Xcode showing Stop button highlighted

12. Open the screenshots in the *Preview* app on your Mac and choose **Adjust Size...** from the **Tools** menu, and change the size of the screenshots if necessary:

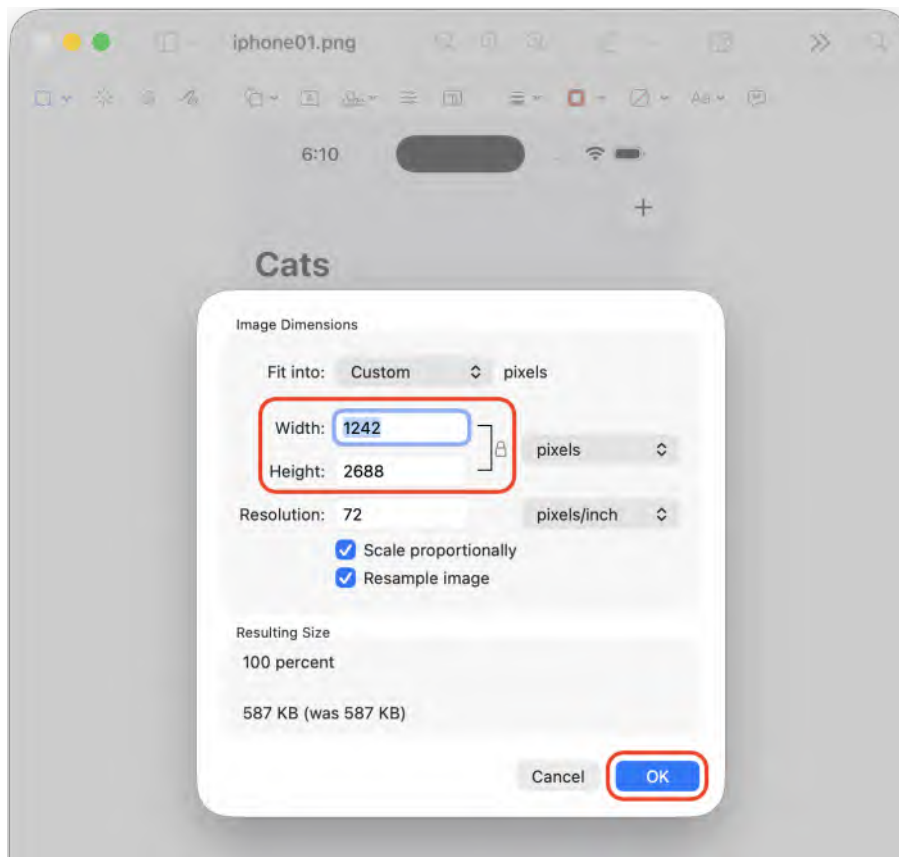


Figure 14.12: Preview app showing Adjust Size window

How to submit screenshots of your app will be discussed in more detail in the next section, where you will learn how to modify your App Store listing prior to submitting your app for App Review. The App Store listing contains all the information about your app that will be displayed in the App Store, so customers can make an informed decision about downloading or purchasing your app.

Submit Your App for App Review

Now that you have screenshots of your app, you'll modify the App Store listing for your app. This allows customers to see information about your app before they download it.

Note

To learn more about App Store Connect workflow, see this link:

<https://developer.apple.com/help/app-store-connect/get-started/app-store-connect-workflow>.

To see more information on how to submit your apps, see this link:

<https://developer.apple.com/app-store/submitting/>.

To see what's new in App Store Connect, watch this video:

<https://developer.apple.com/videos/play/wwdc2025/328/>.

Follow these steps:

1. Using a web browser, go to <http://appstoreconnect.apple.com> and select **Apps**:

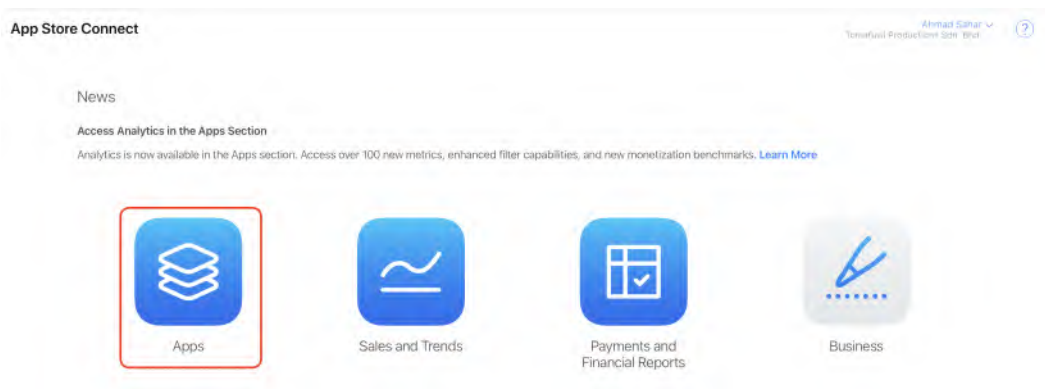


Figure 14.13: The App Store Connect website

2. Select the **CatLog** app:

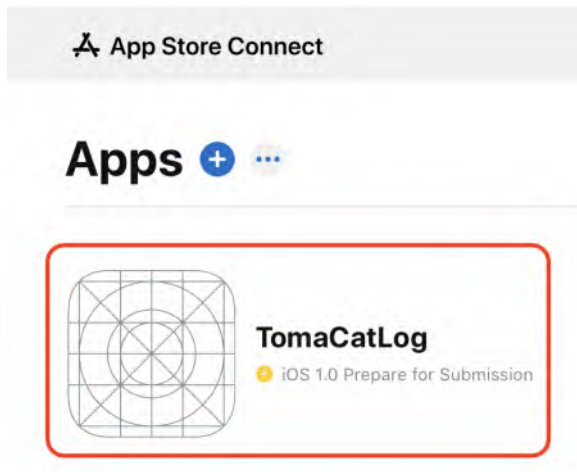


Figure 14.14: Apps screen with your app selected

3. Click **App Information** on the left side of the screen, and make sure the information in the **Name**, **Subtitle**, and **Category** fields are correct:

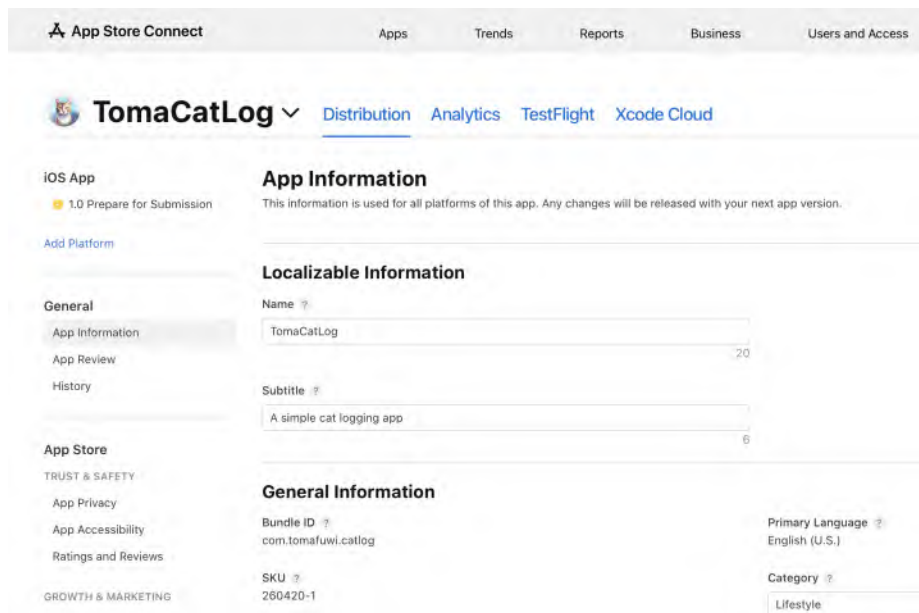


Figure 14.15: The App Information screen

4. Scroll to the **Content Rights** section and click **Edit**. Configure the content rights for your app, and click **Done**:

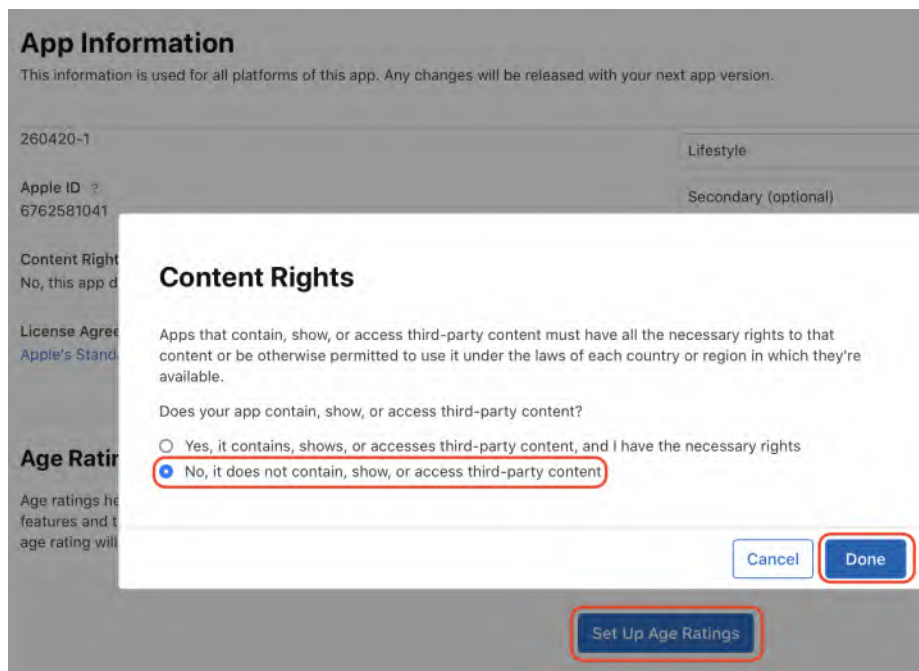


Figure 14.16: The Content Rights configuration window in the App Information screen

5. In the **Age Ratings** section, click **Edit**. Configure the age ratings for your app:

Age Ratings

Step 1: FeaturesStep 2Step 3Step 4Step 5Step 6Step 7

Select whether your app has certain in-app controls, which can be used to limit the content in your app, and capabilities.

 In-App Controls	NO	YES
Parental Controls ? Settings or tools that allow parents/guardians to monitor, manage or restrict a child's access to in-app content or features that may not be suitable.	☒	☐
Age Assurance ? Mechanism to confirm an individual's age meets the age requirement for accessing specific content or services.	☒	☐
 Capabilities	NO	YES
Unrestricted Web Access ? Users can navigate to any webpage within the app or freely browse the web.	☒	☐
User-Generated Content ? Includes the broad distribution of content created by users as a component of the app's intended user experience.	☐	☒
Social Media ? Redistribution, amplification, or interaction with user-generated content through a social feed or similar discovery method that visibly spreads content to many users.	☒	☐

CancelNext

Figure 14.17: The Age Ratings configuration window in the App Information screen

6. Verify that the **Content Rights** and **Age Ratings** have been set, and click **App Privacy** on the left side of the screen:

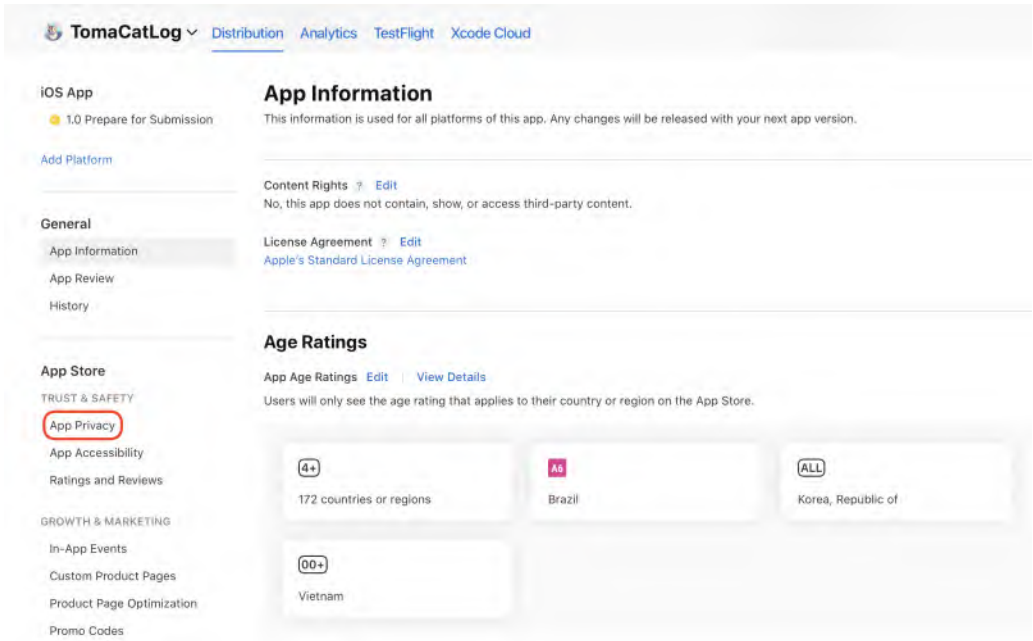


Figure 14.18: The App Privacy link

- You must state what data your app collects and usage practices, as inaccurate privacy disclosures can result in your app being rejected.
7. In the **Privacy Policy** section, click **Edit**. Enter a link to your privacy policy in the field provided, and click **Save**:

Privacy Policy

Any changes will be released with your next app version.

English (U.S.) ?

Privacy Policy URL ?

<https://codershah.tumblr.com/privacy>

219

User Privacy Choices URL (Optional) ?

<http://example.com>

255

Cancel

Save

Figure 14.19: Privacy Policy URL field

8. In the **Data Collection** section, click **Get Started**. Configure the data collection practices for your app, and click **Save**:

Data Collection

Thanks for helping users understand your app's privacy practices. Remember that you're responsible for any third-party code that is added to your app, so if your third-party partners collect data from your app, you must represent that in your responses.

- "Collect" refers to transmitting data off the device in a way that allows you and/or your third-party partners to access it for a period longer than necessary to service the transmitted request in real time.
- "Third-party partners" include analytics tools, advertising networks, third-party SDKs, or other external vendors whose code you have added to the app.

You can [view the full list of questions](#) at any time.

Do you or your third-party partners collect data from this app?

☐ Yes, we collect data from this app

☒ No, we do not collect data from this app

Cancel

Save

Figure 14.20: Data Collection practices configuration screen

9. Verify that the **Privacy Policy** and **Data Collection** have been set and click **Publish** on the top right corner. After that, click **Pricing and Availability** on the left side of the screen:

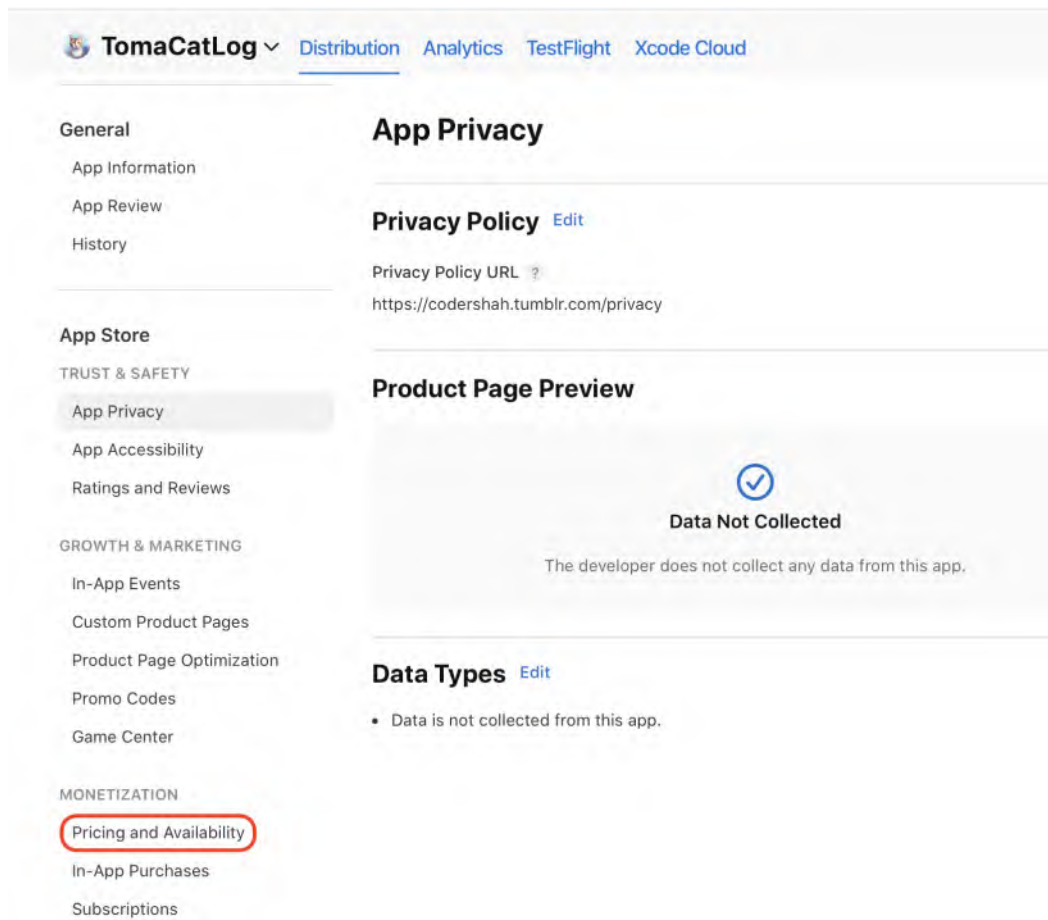


Figure 14.21: The Pricing and Availability link

10. In the **Price Schedule** section, click **Add Pricing**. Set the base country and price for your app, then set the price for all other regions:

App Pricing

Choose your base country or region. Apple may automatically adjust prices from time to time in every country or region except your base to account for changes in tax or foreign exchange rates. You can choose a different price for an individual country or region to prevent it from being automatically adjusted.

Base Country or Region

United States (USD) ▼

Choose a new price for your base country or region. Apple will provide comparable prices for all the other countries or regions based on tax or foreign exchange rates. You can edit prices for individual countries or regions in the next step.

Price ?

\$0.00 ▼

Cancel Next

Figure 14.22: Add Pricing initial configuration screen

11. In the **App Availability** section, click **Set Up Availability**. Configure the regions where your app will be made available for download:

App Availability

Select where your app will be available to purchase or download. [Learn More](#)

☒ **All Countries or Regions**
Make your app available in all 175 countries or regions.

☐ **Specific Countries or Regions**
Make your app available in the countries or regions you select.

☐ **Publish as Pre-Order**
Make your app available as pre-order in the countries or regions you select.

[Cancel](#) [Next](#)

Figure 14.23: App Availability initial configuration screen

12. Click **Prepare for Submission** on the left side of the screen. In the **App Preview** and **Screenshots** section, drag in the screenshots that you took earlier:

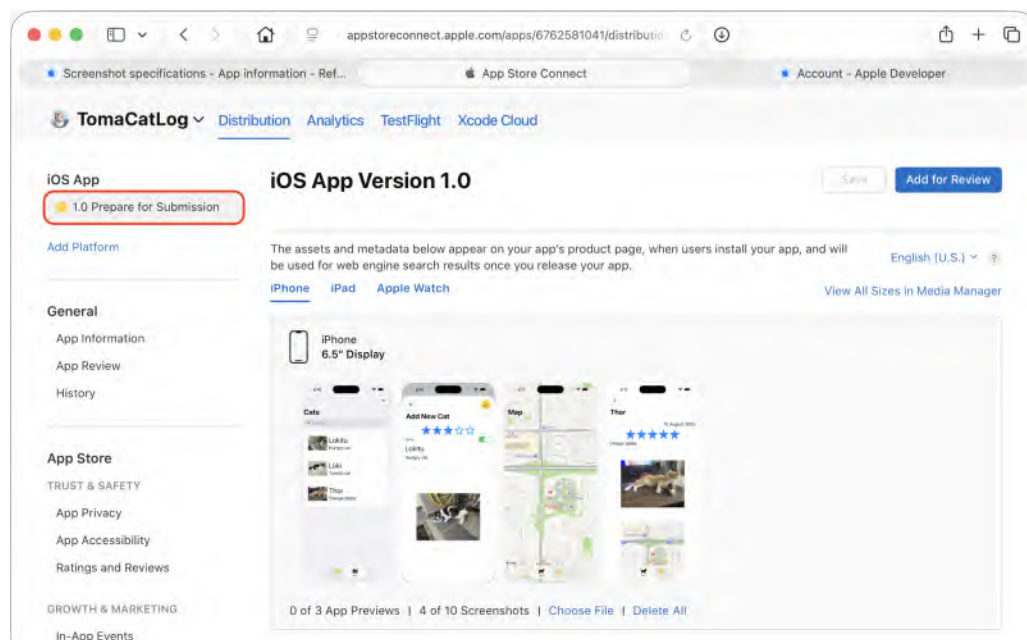


Figure 14.24: Prepare for Submission screen showing App Preview and Screenshots section

- 13. Scroll down and fill in the **Promotional Text, Description, Keywords, Support URL** (containing support information for your app), and **Marketing URL** (containing marketing information for your app) fields:

iOS App Version 1.0

Save

Add for Review

The assets and metadata below appear on your app's product page, when users install your app, and will be used for web engine search results once you release your app.

English (U.S.)

Description

A simple app that lets you record all the cats that live in your neighbourhood. You can add a cat's name, description, photo and location every time you see a new cat, and the app stores it in a searchable database. You can also see which cats are in your vicinity.

3,735

Keywords

cat, log, map

87

Support URL

https://tomafuwi.tumblr.com

Marketing URL

https://tomafuwi.tumblr.com

Figure 14.25: Prepare for submission screen showing required text fields

- 14. Scroll down and fill in the version number and copyright information text fields:

iOS App Version 1.0

Save

Add for Review

The assets and metadata below appear on your app's product page, when users install your app, and will be used for web engine search results once you release your app.

English (U.S.)

Version

1.3

Copyright

Copyright Tomafuwi Productions Sdn. Bhd. 2026

155

Figure 14.26: Prepare for submission screen showing version number and copyright text fields

- 15. Scroll down to the **Build** section and you'll see the archive build you uploaded earlier. If you don't see it, click the + or **Add Build** button, select a build, and click **Done**:

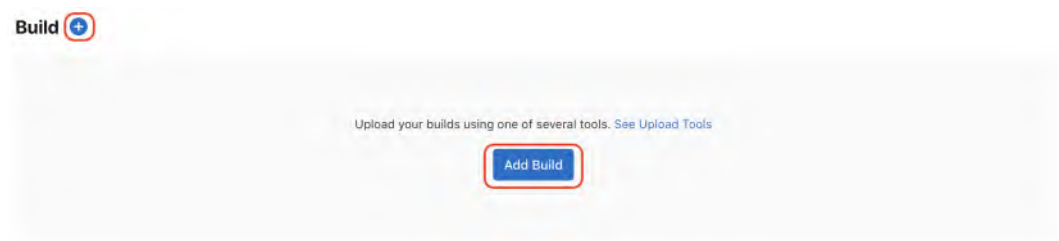


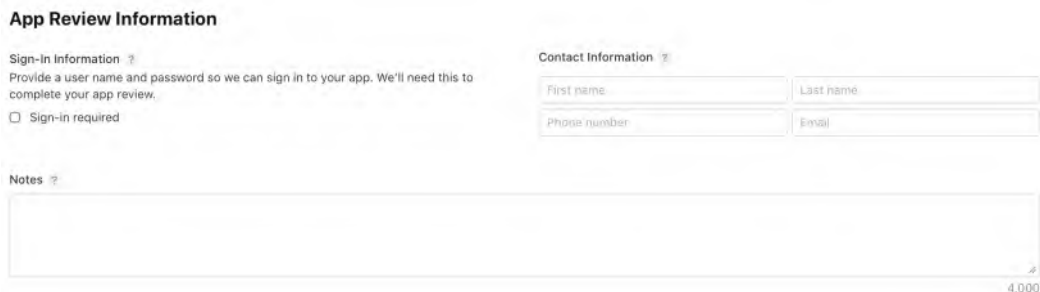
Figure 14.27: The Build selection screen

- 16. Verify that your build is in the **Build** section:



Figure 14.28: The Build section showing your build

17. Scroll down to the **App Review Information** section. If you would like to provide any additional information to the app reviewer, put it here:



App Review Information

Sign-In Information ⓘ
Provide a user name and password so we can sign in to your app. We'll need this to complete your app review.

☐ Sign-in required

Contact Information ⓘ

First name	Last name
Phone number	Email

Notes ⓘ

4,000

Figure 14.29: The App Review Information section

18. Scroll down to the **App Store Version Release** section and maintain the default settings, so your app will be automatically released right after it has been approved by Apple:



App Store Version Release

To make your app available on the App Store, you can automatically release it after it's been approved by App Review. You can also manually release it on the App Store at a later date.

☐ Manually release this version

☒ Automatically release this version

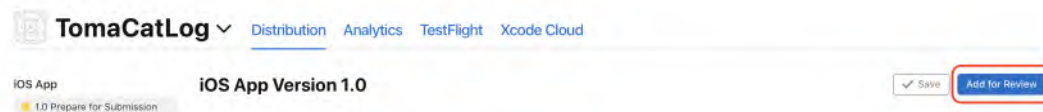
☐ Automatically release this version after App Review, no earlier than

Your local date and time.

Apr 21, 2026 17:00 GMT+8

Figure 14.30: The App Store Version Release section

19. Scroll back up to the top of the screen and click the **Add for Review** button:



TomaCatLog ▾ [Distribution](#) [Analytics](#) [TestFlight](#) [Xcode Cloud](#)

iOS App **iOS App Version 1.0**

1.0 Prepare for Submission

✓ Save **Add for Review**

Figure 14.31: The Add for Review button

The app status will change to **Ready for Review**.

20. A pop-up window will appear showing the details of your submission. Click the **Submit for Review** button to submit your app for review by Apple:

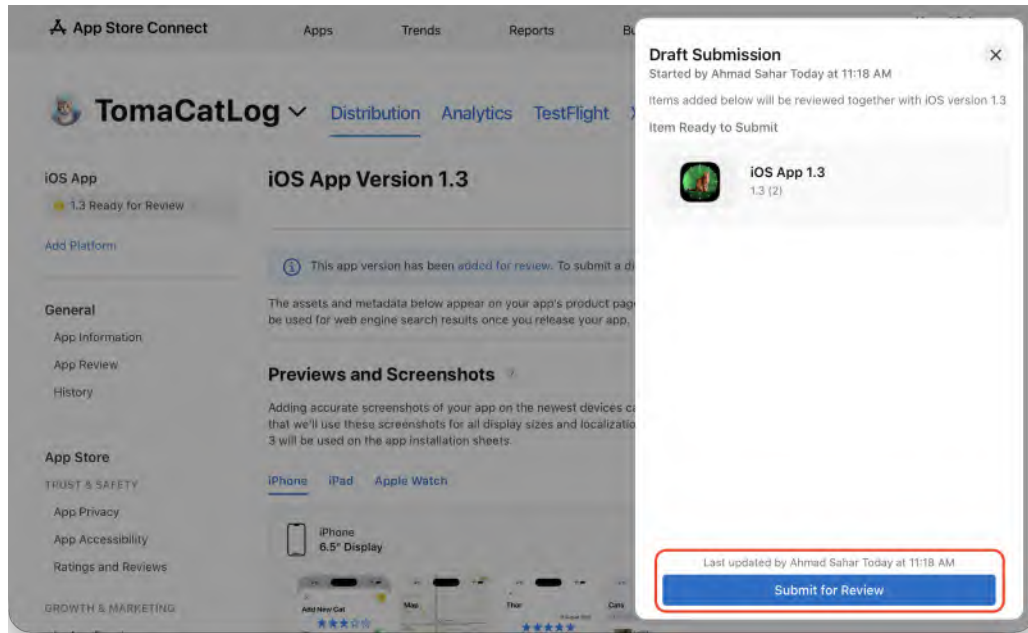


Figure 14.32: Pop-up window showing the Submit for Review button

The app status will change to **Waiting for Review**.

21. Verify that your app has been submitted for review, and click the **x** to close the pop-up window:

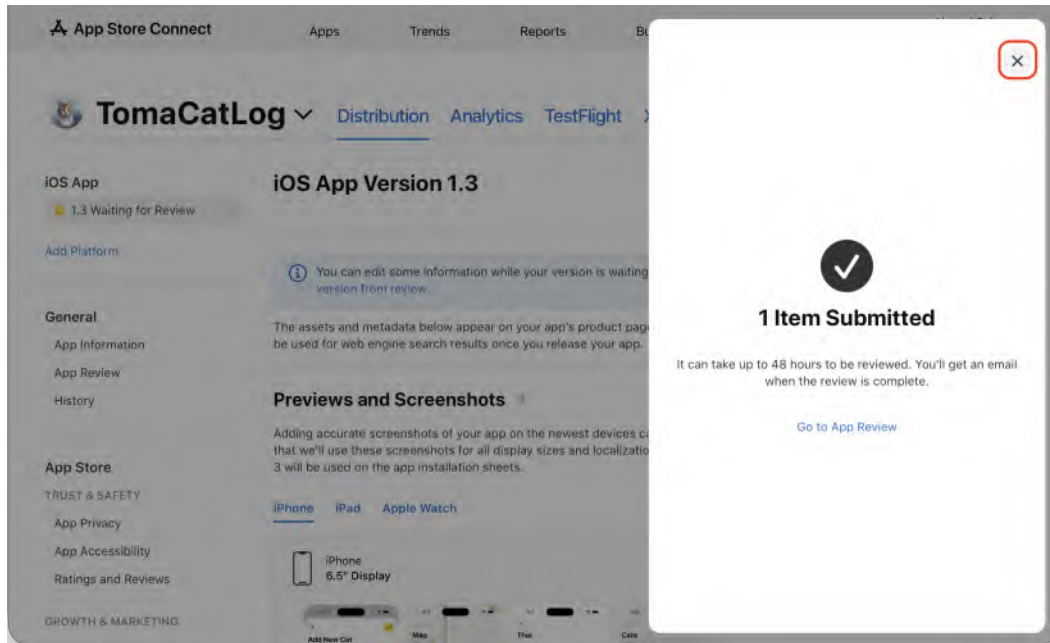


Figure 14.33: Pop-up window showing that app has been submitted for review

You will need to wait for Apple to review the app, and you will receive an email saying whether your app is approved or rejected. If your app is rejected, there will be a link that takes you to Apple's *Resolution Center* page, which describes why your app was rejected. Some common issues are submitting apps that frequently crash, submitting apps that do not match their description, and using Apple's internal APIs in the app. After you have fixed the issues, you can then resubmit your app for review.

You now know how to submit your app to the App Store, and you have reached the end of this book! Awesome!

Note

Want to see the completed app in action? The *CatLog* app that you build step by step throughout this book is available on the App Store:

<https://apps.apple.com/us/app/tomacatlog/id6762581041>.

Explore the finished app, compare it with your own implementation, and see how the concepts covered in the book come together in a real iOS application.

Summary

You have now completed the entire process of building an app and submitting it to the App Store. Congratulations!

In this chapter, you learned how to take screenshots for your app using Xcode and Device Hub. Next, you learned how to modify an App Store listing and submit your release build to the App Store.

You now know how to submit your apps to the App Store.

Once an app has been submitted for review, all you can do is wait for Apple to review your app. Don't worry if the app gets rejected—it happens to all developers. Work with Apple to resolve issues via the *Resolution Center* and do your research to know what is and what is not acceptable to Apple.

After your apps are on the App Store, feel free to reach out to me (@shah_apple) on Twitter to let me know—I would love to see what you have built.

15

Unlock the Code Bundle and the PDF Version

Your copy of this book includes the following exclusive benefits:



Complete Code Bundle

Download the full source code for this book's examples and projects.



DRM-Free PDF Version

Download DRM-free PDF and ePub copies of this book.



7-Day Packt Library Access

Get 7-day unlimited access to 8,000+ books and videos. No credit card required.

Available for first-time Packt+ trial users only.



Next-Gen Reader Access

Read this book on Packt Reader with progress sync, dark mode and note-taking.

Follow this guide to unlock them. The process takes only a few minutes and only needs to be completed once.

Unlock this book's free benefits in three easy steps

Step 1

Have your purchase invoice ready for *step 3*. If you have a physical copy, scan it using your phone and save it as a PDF, JPG, or PNG.

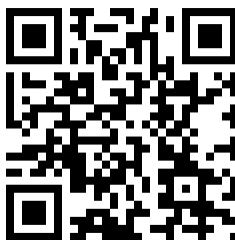
For more help on finding your invoice, visit <https://www.packtpub.com/unlock-benefits/help>.

Note

Note: If you bought this book directly from the Packt website, no invoice is required. After *step 2*, you can access your exclusive content right away.

Step 2

Scan the QR code or go to packtpub.com/unlock.



On the page that opens (similar to *Figure 14.1* on desktop), search for this book by name and select the correct edition.

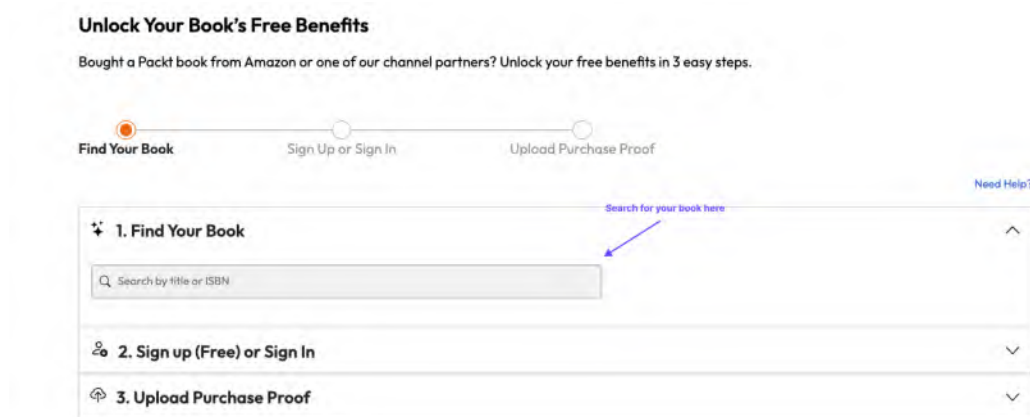


Figure 14.1: Packt unlock landing page on desktop

Step 3

After selecting the book, sign in to your Packt account or create one for free. Then, upload your invoice (PDF, PNG, or JPG, up to 10 MB). Follow the on-screen instructions to finish the process.

Need help?

If you get stuck and need help, visit <https://www.packtpub.com/unlock-benefits/help> for a detailed FAQ on how to find your invoice and more. This QR code will take you to the help page.



Note

Note: If you are still facing issues, reach out to customer@packt.com.



packtpub.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

At www.packtpub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

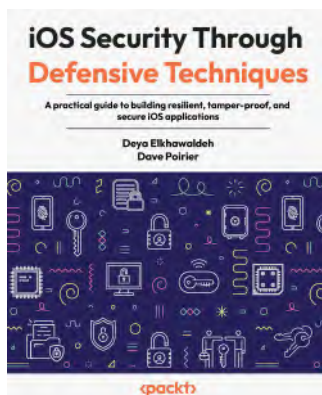


AI Driven Swift Architecture

Walid SASSI, Dave Poirier

ISBN: 9781835886540

- Configure Xcode workflows for AI assisted Swift workflows
- Apply Swift concurrency patterns with AI guided reasoning
- Design SwiftUI features using Clean Architecture boundaries
- Learn to use Claude Code to drive TDD, refactors, and code reviews
- Modernize UIKit codebases with safe, incremental migrations
- Integrate Apple Foundation Models for private on device AI
- Build MCP servers in Swift to power feature focused agents
- Write RFCs and ADRs to govern AI adoption in iOS teams



iOS Security Through Defensive Techniques

Deya Elkhawaldeh, Dave Poirier

ISBN: 9781806383177

- Understand core iOS security, privacy, and threat modeling concepts
- Learn to classify sensitive data and set clear handling and retention rules
- Apply platform safe coding patterns to reduce common app flaws
- Protect sensitive data using Keychain, file protection, and hardware backed keys.
- Implement secure storage using Keychain, file protection, and Secure Enclave
- Use modern cryptography with correct key management and safe defaults
- Harden apps with anti-debugging, RASP, and runtime integrity checks

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packt.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share your thoughts

Now you've finished *SwiftUI for Beginners*, we'd love to hear your thoughts! Scan the QR code below to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.



<https://packt.link/r/1807785572>

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Index

A

Add New Cat screen	
custom view, adding to	148, 149
data, passing to Cats screen	90 – 94
from	
modifying	63 – 67
AddNewCatView view	
modifying	103 – 110
App Preview	
app, running on iPad	12, 13
results, previewing	8 – 11
App Store Connect	167
app, submitting to	170 – 182
Apple Developer account	168, 169
annotations	
displaying, on Map screen	110 – 118
app	
submitting, to App Store	170 – 182
Connect	
testing, externally with	191 – 196
TestFlight	
testing, internally with	182 – 190
TestFlight	
testing, with TestFlight	182
app playground	
building	26
configuring	29 – 31
modifying, to use data	134 – 142
container	
tab bar, setting up	26, 27

C

CLLocationManager class	
reference link	102
Cat Detail screen	
custom view, adding to	149, 150
implementing	52 – 54
map snapshot, displaying	119 – 122
on	

modifying	68 – 73
-----------	---------

Cat class	
modifying, to work with	127 – 131
SwiftData	

Cat instances	
class, creating to represent	81 – 83
data, displaying in view	84 – 88
sample data, creating	83, 84

CatLog app	22
Add New Cat screen button	23
Cats Detail screen	24
Cats screen	22
screenshots, getting	200 – 206
submitting, for app review	206 – 219

Cats screen	
cat, removing from	95 – 97
data, passing from Add	90 – 94
New Cat screen to	
data, passing to Cat Detail	97 – 99
screen from	
modifying	58 – 62
search bar, implementing	160 – 162
for	

CatsView view	
data, displaying in	84 – 88

Core Location framework	101
AddNewCatView view,	103 – 110
modifying	
reference link	102
used, for obtaining device	102, 103
location	

camel case	20
-------------------	-----------

class	81
creating, to represent Cat	81 – 83
instances	

custom view	
adding, to Add New Cat	148, 149
screen	
adding, to Cat Detail screen	149, 150
creating	34 – 38, 144 – 147

D**data container**

app playground, modifying 134 – 142
to use

creating 132 – 134

device location

obtaining, with Core 102, 103
Location framework

I**iPad**

app, running on 12, 13

instances 81

M**Map screen**

annotations, displaying on 110 – 118

MapKit framework 101

MapKit, for SwiftUI

reference link 110

Maps screen

implementing 54, 55

map snapshot

displaying, on Cat Detail 119 – 122
screen

methods 81

N**navigation bar**

button, adding 41, 42

cancel and save buttons, 46 – 48
adding

view, adding 42 – 46

navigation split view

implementing 162 – 164

navigation stack 18 – 20

views, embedding 27, 28

O

optional property 82

P**PhotosPicker view**

implementing 152 – 157

properties 81

R**Rent a Mac**

URL 200

S**Swift Playground**

3, 7

downloading, from App 4, 5
Store

installing, from App Store 4 – 6

reference link 4

terms 16 – 21

user interface, exploring 7, 8

Swift language

reference link 20

SwiftData 126, 127

Cat class, modifying to 127 – 131
work with

SwiftUI app 3

reference link 5

screenshots

getting, of CatLog app 200 – 206

search bar

implementing, for Cats 160 – 162
screen

state 78

state variables 78 – 81

strings 81

syntax highlighting 20

T

TestFlight 167

reference link 182

used, for testing app 182

used, for testing app 191 – 196

externally

used, for testing app 182 – 190

internally

tab view 19

modifying 39, 40

V

variables 78

view	presenting, modally	41
-------------	---------------------	----

