iOS 17 App Development Essentials

iOS 17 App Development Essentials

ISBN-13: 978-1-951442-80-4

Rev: 1.0

Contents

Table of Contents

Chapter 1

1. Start Here

This book aims to teach the skills necessary to build iOS 17 applications using SwiftUI, Xcode 15, and the Swift programming language.

Beginning with the basics, this book outlines the steps to set up an iOS development environment, together with an introduction to using Swift Playgrounds to learn and experiment with Swift.

The book also includes in-depth chapters introducing the Swift programming language, including data types, control flow, functions, object-oriented programming, property wrappers, structured concurrency, and error handling.

A guided tour of Xcode in SwiftUI development mode follows an introduction to the key concepts of SwiftUI and project architecture. The book also covers creating custom SwiftUI views and explains how these views are combined to create user interface layouts, including stacks, frames, and forms.

Other topics covered include data handling using state properties and observable, state, and environment objects, as are key user interface design concepts such as modifiers, lists, tabbed views, context menus, user interface navigation, and outline groups.

The book also includes chapters covering graphics and chart drawing, user interface animation, view transitions and gesture handling, WidgetKit, Live Activities, document-based apps, Core Data, SwiftData, and CloudKit.

Chapters also explain how to integrate SwiftUI views into existing UIKit-based projects and integrate UIKit code into SwiftUI.

Finally, the book explains how to package up a completed app and upload it to the App Store for publication.

Along the way, the topics covered in the book are put into practice through detailed tutorials, the source code for which is also available for download.

The aim of this book, therefore, is to teach you the skills to build your own apps for iOS 17 using SwiftUI. Assuming you are ready to download the iOS 17 SDK and Xcode 15 and have an Apple Mac system, you are ready to get started.

1.1 For Swift Programmers

This book has been designed to address the needs of both existing Swift programmers and those new to Swift and iOS app development. If you are familiar with the Swift programming language, you can probably skip the Swift-specific chapters. If you are not yet familiar with the SwiftUI-specific language features of Swift, however, we recommend that you at least read the sections covering implicit returns from single expressions, opaque return types, and property wrappers. These features are central to the implementation and understanding of SwiftUI.

1.2 For Non-Swift Programmers

If you are new to programming in Swift, then the entire book is appropriate for you. Just start at the beginning and keep going.

1.3 Source Code Download

The source code and Xcode project files for the examples contained in this book are available for download at:

https://www.ebookfrenzy.com/retail/ios17/

1.4 Feedback

We want you to be satisfied with your purchase of this book. Therefore, if you find any errors in the book or have any comments, questions, or concerns, please contact us at *feedback@ebookfrenzy.com*.

1.5 Errata

While we make every effort to ensure the accuracy of the content of this book, inevitably, a book covering a subject area of this size and complexity may include some errors and oversights. Any known issues with the book will be outlined, together with solutions, at the following URL:

https://www.ebookfrenzy.com/errata/ios17.html

In the event that you find an error not listed in the errata, please let us know by emailing our technical support team at *feedback@ebookfrenzy.com*.

Chapter 2

2. Joining the Apple Developer Program

The first step in the process of learning to develop iOS 17 based applications involves gaining an understanding of the advantages of enrolling in the Apple Developer Program and deciding the point at which it makes sense to pay to join. With these goals in mind, this chapter will outline the costs and benefits of joining the developer program and, finally, walk through the steps involved in enrolling.

2.1 Downloading Xcode 15 and the iOS 17 SDK

The latest versions of both the iOS SDK and Xcode can be downloaded free of charge from the macOS App Store. Since the tools are free, this raises the question of whether to enroll in the Apple Developer Program, or to wait until it becomes necessary later in your app development learning curve.

2.2 Apple Developer Program

Membership in the Apple Developer Program currently costs $99 per year to enroll as an individual developer. Organization level membership is also available.

Much can be achieved without the need to pay to join the Apple Developer program. There are, however, areas of app development which cannot be fully tested without program membership. Of particular significance is the fact that Siri integration, iCloud access, Apple Pay, Game Center and In-App Purchasing can only be enabled and tested with Apple Developer Program membership.

Of further significance is the fact that Apple Developer Program members have access to technical support from Apple's iOS support engineers (though the annual fee initially covers the submission of only two support incident reports, more can be purchased). Membership also includes access to the Apple Developer forums; an invaluable resource both for obtaining assistance and guidance from other iOS developers, and for finding solutions to problems that others have encountered and subsequently resolved.

Program membership also provides early access to the pre-release Beta versions of Xcode, macOS and iOS.

By far the most important aspect of the Apple Developer Program is that membership is a mandatory requirement in order to publish an application for sale or download in the App Store.

Clearly, program membership is going to be required at some point before your application reaches the App Store. The only question remaining is when exactly to sign up.

2.3 When to Enroll in the Apple Developer Program?

Clearly, there are many benefits to Apple Developer Program membership and, eventually, membership will be necessary to begin selling your apps. As to whether to pay the enrollment fee now or later will depend on individual circumstances. If you are still in the early stages of learning to develop iOS apps or have yet to come up with a compelling idea for an app to develop then much of what you need is provided without program membership. As your skill level increases and your ideas for apps to develop take shape you can, after all, always enroll in the developer program later.

If, on the other hand, you are confident that you will reach the stage of having an application ready to publish,

or know that you will need access to more advanced features such as Siri support, iCloud storage, In-App Purchasing and Apple Pay then it is worth joining the developer program sooner rather than later.

2.4 Enrolling in the Apple Developer Program

If your goal is to develop iOS apps for your employer, then it is first worth checking whether the company already has membership. That being the case, contact the program administrator in your company and ask them to send you an invitation from within the Apple Developer Program Member Center to join the team. Once they have done so, Apple will send you an email entitled *You Have Been Invited to Join an Apple Developer Program* containing a link to activate your membership. If you or your company is not already a program member, you can enroll online at:

https://developer.apple.com/programs/enroll/

Apple provides enrollment options for businesses and individuals. To enroll as an individual, you will need to provide credit card information in order to verify your identity. To enroll as a company, you must have legal signature authority (or access to someone who does) and be able to provide documentation such as a Dun & Bradstreet D-U-N-S number and documentation confirming legal entity status.

Acceptance into the developer program as an individual member typically takes less than 24 hours with notification arriving in the form of an activation email from Apple. Enrollment as a company can take considerably longer (sometimes weeks or even months) due to the burden of the additional verification requirements.

While awaiting activation you may log in to the Member Center with restricted access using your Apple ID and password at the following URL:

https://developer.apple.com/membercenter

Once logged in, clicking on the *Your Account* tab at the top of the page will display the prevailing status of your application to join the developer program as *Enrollment Pending*. Once the activation email has arrived, log in to the Member Center again and note that access is now available to a wide range of options and resources as illustrated in Figure 2-1:

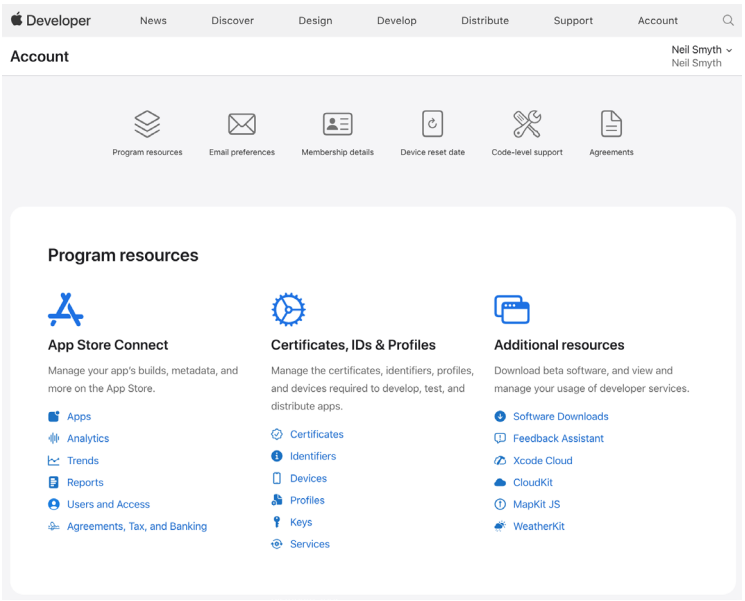


Figure 2-1

2.5 Summary

An important early step in the iOS 17 application development process involves identifying the best time to enroll in the Apple Developer Program. This chapter has outlined the benefits of joining the program, provided some guidance to keep in mind when considering developer program membership and walked briefly through the enrollment process. The next step is to download and install the iOS 17 SDK and Xcode 15 development environment.

Chapter 3

3. Installing Xcode 15 and the iOS 17 SDK

iOS apps are developed using the iOS SDK in conjunction with Apple's Xcode development environment. Xcode is an integrated development environment (IDE) within which you will code, compile, test and debug your iOS applications.

All of the examples in this book are based on Xcode version 15 and make use of features unavailable in earlier Xcode versions. In this chapter we will cover the steps involved in installing both Xcode 15 and the iOS 17 SDK on macOS.

3.1 Identifying Your macOS Version

When developing with SwiftUI, the Xcode 15 environment requires a system running macOS Ventura 13.5, or later. If you are unsure of the version of macOS on your Mac, you can find this information by clicking on the Apple menu in the top left-hand corner of the screen and selecting the *About This Mac* option from the menu. In the resulting dialog check the *macOS* line:

Figure 3-1

If the "About This Mac" dialog does not indicate that macOS 13.5 or later is running, click on the *Software Update…* button to download and install the appropriate operating system upgrades.

3.2 Installing Xcode 15 and the iOS 17 SDK

The best way to obtain the latest versions of Xcode and the iOS SDK is to download them from the Apple Mac App Store. Launch the App Store on your macOS system, enter Xcode into the search box and click on the *Get* button to initiate the installation. This will install both Xcode and the iOS SDK.

3.3 Starting Xcode

Having successfully installed the SDK and Xcode, the next step is to launch it so that we are ready to start development work. To start up Xcode, open the macOS Finder and search for *Xcode*. Since you will be making frequent use of this tool take this opportunity to drag and drop it onto your dock for easier access in the future. Click on the Xcode icon in the dock to launch the tool. The first time Xcode runs you may be prompted to install additional components. Follow these steps, entering your username and password when prompted to do so.

Once Xcode has loaded, and assuming this is the first time you have used Xcode on this system, you will be presented with the *Welcome* screen from which you are ready to proceed:

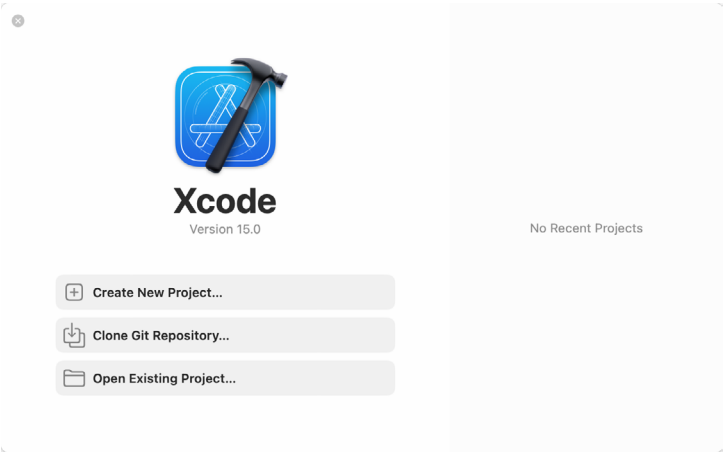


Figure 3-2

3.4 Adding Your Apple ID to the Xcode Preferences

Regardless of whether or not you choose to enroll in the Apple Developer Program it is worth adding your Apple ID to Xcode now that it is installed and running. Select the *Xcode -> Settings…* menu option followed by the *Accounts* tab. On the Accounts screen, click on the + button highlighted in Figure 3-3, select *Apple ID* from the resulting panel and click on the *Continue* button. When prompted, enter your Apple ID and password before clicking on the *Sign In* button to add the account to the preferences.

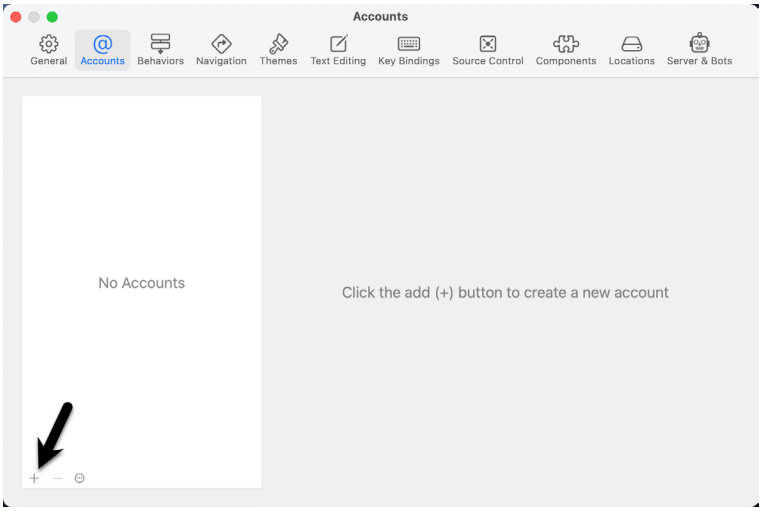


Figure 3-3

3.5 Developer and Distribution Signing Identities

Once the Apple ID has been entered the next step is to generate signing identities. To view the current signing identities, select the newly added Apple ID in the Accounts panel and click on the *Manage Certificates…* button to display a list of available signing identity types. To create a signing identity, simply click on the + button highlighted in Figure 3-4 and make the appropriate selection from the menu:

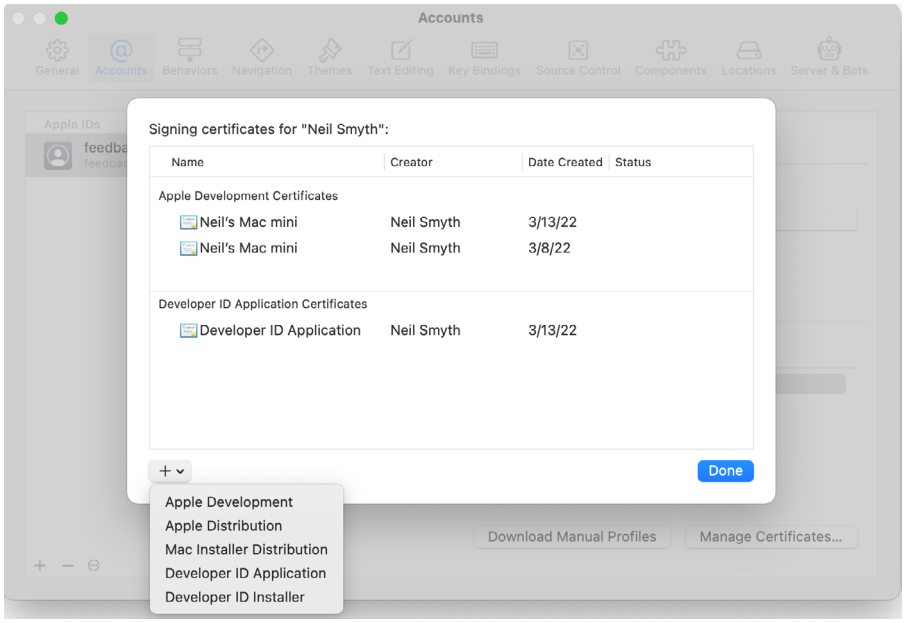


Figure 3-4

If the Apple ID has been used to enroll in the Apple Developer program, the option to create an *Apple Distribution* certificate will appear in the menu which will, when clicked, generate the signing identity required to submit the app to the Apple App Store. You will also need to create a *Developer ID Application* certificate if you plan to integrate features such as iCloud and Siri into your app projects. If you have not yet signed up for the Apple Developer program, select the *Apple Development* option to allow apps to be tested during development.

3.6 Summary

This book was written using Xcode 15 and the iOS 17 SDK running on macOS 13.5.2 (Ventura). Before beginning SwiftUI development, the first step is to install Xcode and configure it with your Apple ID via the accounts section of the Preferences screen. Once these steps have been performed, a development certificate must be generated which will be used to sign apps developed within Xcode. This will allow you to build and test your apps on physical iOS-based devices.

When you are ready to upload your finished app to the App Store, you will also need to generate a distribution certificate, a process requiring membership in the Apple Developer Program as outlined in the previous chapter.

Having installed the iOS SDK and successfully launched Xcode 15 we can now look at Xcode in more detail, starting with Playgrounds.

Chapter 14

14. Working with Array and Dictionary Collections in Swift

Arrays and dictionaries in Swift are objects that contain collections of other objects. This chapter will cover some of the basics of working with arrays and dictionaries in Swift.

14.1 Mutable and Immutable Collections

Collections in Swift come in mutable and immutable forms. The contents of immutable collection instances cannot be changed after the object has been initialized. To make a collection immutable, assign it to a *constant* when it is created. On the other hand, collections are mutable if assigned to a *variable*.

14.2 Swift Array Initialization

An array is a data type designed specifically to hold multiple values in a single ordered collection. An array, for example, could be created to store a list of String values. Strictly speaking, a single Swift based array is only able to store values that are of the same type. An array declared as containing String values, therefore, could not also contain an Int value. As will be demonstrated later in this chapter, however, it is also possible to create mixed type arrays. The type of an array can be specified specifically using type annotation or left to the compiler to identify using type inference.

An array may be initialized with a collection of values (referred to as an *array literal*) at creation time using the following syntax:

var variableName: [type] = [value 1, value2, value3, …….]

The following code creates a new array assigned to a variable (thereby making it mutable) that is initialized with three string values:

```
var treeArray = ["Pine", "Oak", "Yew"]
```

Alternatively, the same array could have been created immutably by assigning it to a constant:

```
let treeArray = ["Pine", "Oak", "Yew"]
```

In the above instance, the Swift compiler will use type inference to decide that the array contains values of String type and prevent values of other types being inserted into the array elsewhere within the application code.

Alternatively, the same array could have been declared using type annotation:

```
var treeArray: [String] = ["Pine", "Oak", "Yew"]
```

Arrays do not have to have values assigned at creation time. The following syntax can be used to create an empty array:

```
var variableName = [type]()
```

Consider, for example, the following code which creates an empty array designated to store floating point values and assigns it to a variable named priceArray:

```
var priceArray = [Float]()
```

Another useful initialization technique allows an array to be initialized to a certain size with each array element

pre-set with a specified default value:

```
var nameArray = [String](repeating: "My String", count: 10)
```

When compiled and executed, the above code will create a new 10 element array with each element initialized with a string that reads "My String".

Finally, a new array may be created by adding together two existing arrays (assuming both arrays contain values of the same type). For example:

```
let firstArray = ["Red", "Green", "Blue"]
let secondArray = ["Indigo", "Violet"]

let thirdArray = firstArray + secondArray
```

14.3 Working with Arrays in Swift

Once an array exists, a wide range of methods and properties are provided for working with and manipulating the array content from within Swift code, a subset of which is as follows:

14.3.1 Array Item Count

A count of the items in an array can be obtained by accessing the array's count property:

```
var treeArray = ["Pine", "Oak", "Yew"]
var itemCount = treeArray.count

print(itemCount)
```

Whether or not an array is empty can be identified using the array's Boolean *isEmpty* property as follows:

```
var treeArray = ["Pine", "Oak", "Yew"]

if treeArray.isEmpty {
    // Array is empty
}
```

14.3.2 Accessing Array Items

A specific item in an array may be accessed or modified by referencing the item's position in the array index (where the first item in the array has index position 0) using a technique referred to as *index subscripting*. In the following code fragment, the string value contained at index position 2 in the array (in this case the string value "Yew") is output by the print call:

```
var treeArray = ["Pine", "Oak", "Yew"]

print(treeArray[2])
```

This approach can also be used to replace the value at an index location:

```
treeArray[1] = "Redwood"
```

The above code replaces the current value at index position 1 with a new String value that reads "Redwood".

14.3.3 Random Items and Shuffling

A call to the *shuffled()* method of an array object will return a new version of the array with the item ordering randomly shuffled, for example:

```
let shuffledTrees = treeArray.shuffled()
```

To access an array item at random, simply make a call to the *randomElement()* method:

```
let randomTree = treeArray.randomElement()
```

14.3.4 Appending Items to an Array

Items may be added to an array using either the *append* method or + and += operators. The following, for example, are all valid techniques for appending items to an array:

```
treeArray.append("Redwood")
treeArray += ["Redwood"]
treeArray += ["Redwood", "Maple", "Birch"]
```

14.3.5 Inserting and Deleting Array Items

New items may be inserted into an array by specifying the index location of the new item in a call to the array's *insert(at:)* method. An insertion preserves all existing elements in the array, essentially moving them to the right to accommodate the newly inserted item:

```
treeArray.insert("Maple", at: 0)
```

Similarly, an item at a specific array index position may be removed using the *remove(at:)* method call:

```
treeArray.remove(at: 2)
```

To remove the last item in an array, simply make a call to the array's *removeLast* method as follows:

```
treeArray.removeLast()
```

14.3.6 Array Iteration

The easiest way to iterate through the items in an array is to make use of the for-in looping syntax. The following code, for example, iterates through all of the items in a String array and outputs each item to the console panel:

```
let treeArray = ["Pine", "Oak", "Yew", "Maple", "Birch", "Myrtle"]

for tree in treeArray {
    print(tree)
}
```

Upon execution, the following output will appear in the console:

```
Pine
Oak
Yew
Maple
Birch
Myrtle
```

The same result can be achieved by calling the *forEach()* array method. When this method is called on an array, it will iterate through each element and execute specified code. For example:

```
treeArray.forEach { tree in
    print(tree)
}
```

Note that since the task to be performed for each array element is declared in a closure expression, the above example may be modified as follows to take advantage of shorthand argument names:

```
treeArray.forEach {
    print($0)
```

```
}
```

14.4 Creating Mixed Type Arrays

A mixed type array is an array that can contain elements of different class types. Clearly, an array that is either declared or inferred as being of type String cannot subsequently be used to contain non-String class object instances. Interesting possibilities arise, however, when taking into consideration that Swift includes the *Any* type. Any is a special type in Swift that can be used to reference an object of a non-specific class type. It follows, therefore, that an array declared as containing Any object types can be used to store elements of mixed types. The following code, for example, declares and initializes an array containing a mixture of String, Int and Double elements:

```
let mixedArray: [Any] = ["A String", 432, 34.989]
```

The use of the Any type should be used with care since the use of Any masks from Swift the true type of the elements in such an array thereby leaving code prone to potential programmer error. It will often be necessary, for example, to manually cast the elements in an Any array to the correct type before working with them in code. Performing the incorrect cast for a specific element in the array will most likely cause the code to compile without error but crash at runtime. Consider, for the sake of an example, the following mixed type array:

```
let mixedArray: [Any] = [1, 2, 45, "Hello"]
```

Assume that, having initialized the array, we now need to iterate through the integer elements in the array and multiply them by 10. The code to achieve this might read as follows:

```
for object in mixedArray {
    print(object * 10)
}
```

When entered into Xcode, however, the above code will trigger a syntax error indicating that it is not possible to multiply operands of type Any and Int. In order to remove this error it will be necessary to downcast the array element to be of type Int:

```
for object in mixedArray {
    print(object as! Int * 10)
}
```

The above code will compile without error and work as expected until the final String element in the array is reached at which point the code will crash with the following error:

```
Could not cast value of type 'Swift.String' to 'Swift.Int'
```

The code will, therefore, need to be modified to be aware of the specific type of each element in the array. Clearly, there are both benefits and risks to using Any arrays in Swift.

14.5 Swift Dictionary Collections

String dictionaries allow data to be stored and managed in the form of key-value pairs. Dictionaries fulfill a similar purpose to arrays, except each item stored in the dictionary has associated with it a unique key (to be precise, the key is unique to the particular dictionary object) which can be used to reference and access the corresponding value. Currently only String, Int, Double and Bool data types are suitable for use as keys within a Swift dictionary.

14.6 Swift Dictionary Initialization

A dictionary is a data type designed specifically to hold multiple values in a single unordered collection. Each item in a dictionary consists of a key and an associated value. The data types of the key and value elements type may be specified specifically using type annotation, or left to the compiler to identify using type inference.

A new dictionary may be initialized with a collection of values (referred to as a *dictionary literal*) at creation time using the following syntax:

var *variableName*: [*key type*: *value type*] = [*key 1: value 1*, *key 2: value2* ….]

The following code creates a new dictionary assigned to a variable (thereby making it mutable) that is initialized with four key-value pairs in the form of ISBN numbers acting as keys for corresponding book titles:

```
var bookDict = ["100-432112" : "Wind in the Willows",
                "200-532874" : "Tale of Two Cities",
                "202-546549" : "Sense and Sensibility",
                "104-109834" : "Shutter Island"]
```

In the above instance, the Swift compiler will use type inference to decide that both the key and value elements of the dictionary are of String type and prevent values or keys of other types being inserted into the dictionary.

Alternatively, the same dictionary could have been declared using type annotation:

```
var bookDict: [String: String] =
               ["100-432112" : "Wind in the Willows",
                "200-532874" : "Tale of Two Cities",
                "202-546549" : "Sense and Sensibility",
                "104-109834" : "Shutter Island"]
```

As with arrays, it is also possible to create an empty dictionary, the syntax for which reads as follows:

var *variableName* = [*key type*: *value type*]()

The following code creates an empty dictionary designated to store integer keys and string values:

```
var myDictionary = [Int: String]()
```

14.7 Sequence-based Dictionary Initialization

Dictionaries may also be initialized using sequences to represent the keys and values. This is achieved using the Swift *zip()* function, passing through the keys and corresponding values. In the following example, a dictionary is created using two arrays:

```
let keys = ["100-432112", "200-532874", "202-546549", "104-109834"]
let values = ["Wind in the Willows", "Tale of Two Cities",
                "Sense and Sensibility", "Shutter Island"]

let bookDict = Dictionary(uniqueKeysWithValues: zip(keys, values))
```

This approach allows keys and values to be generated programmatically. In the following example, a number range starting at 1 is being specified for the keys instead of using an array of predefined keys:

```
let values = ["Wind in the Willows", "Tale of Two Cities",
               "Sense and Sensibility", "Shutter Island"]

var bookDict = Dictionary(uniqueKeysWithValues: zip(1..., values))
```

The above code is a much cleaner equivalent to the following dictionary declaration:

```
var bookDict = [1 : "Wind in the Willows",
                2 : "Tale of Two Cities",
                3 : "Sense and Sensibility",
```

```
                4 : "Shutter Island"]
```

14.8 Dictionary Item Count

A count of the items in a dictionary can be obtained by accessing the dictionary's count property:

```
print(bookDict.count)
```

14.9 Accessing and Updating Dictionary Items

A specific value may be accessed or modified using key subscript syntax to reference the corresponding value. The following code references a key known to be in the bookDict dictionary and outputs the associated value (in this case the book entitled "A Tale of Two Cities"):

```
print(bookDict["200-532874"])
```

When accessing dictionary entries in this way, it is also possible to declare a default value to be used in the event that the specified key does not return a value:

```
print(bookDict["999-546547", default: "Book not found"])
```

Since the dictionary does not contain an entry for the specified key, the above code will output text which reads "Book not found".

Indexing by key may also be used when updating the value associated with a specified key, for example, to change the title of the same book from "A Tale of Two Cities" to "Sense and Sensibility"):

```
bookDict["200-532874"] = "Sense and Sensibility"
```

The same result is also possible by making a call to the *updateValue(forKey:)* method, passing through the key corresponding to the value to be changed:

```
bookDict.updateValue("The Ruins", forKey: "200-532874")
```

14.10 Adding and Removing Dictionary Entries

Items may be added to a dictionary using the following key subscripting syntax:

dictionaryVariable[*key*] = *value*

For example, to add a new key-value pair entry to the books dictionary:

```
bookDict["300-898871"] = "The Overlook"
```

Removal of a key-value pair from a dictionary may be achieved either by assigning a *nil* value to the entry, or via a call to the *removeValueForKey* method of the dictionary instance. Both code lines below achieve the same result of removing the specified entry from the books dictionary:

```
bookDict["300-898871"] = nil
bookDict.removeValue(forKey: "300-898871")
```

14.11 Dictionary Iteration

As with arrays, it is possible to iterate through dictionary entries by making use of the for-in looping syntax. The following code, for example, iterates through all of the entries in the books dictionary, outputting both the key and value for each entry:

```
for (bookid, title) in bookDict {
  print("Book ID: \(bookid) Title: \(title)")
}
```

Upon execution, the following output will appear in the console:

```
Book ID: 100-432112 Title: Wind in the Willows
```

```
Book ID: 200-532874 Title: The Ruins
Book ID: 104-109834 Title: Shutter Island
Book ID: 202-546549 Title: Sense and Sensibility
```

14.12 Summary

Collections in Swift take the form of either dictionaries or arrays. Both provide a way to collect together multiple items within a single object. Arrays provide a way to store an ordered collection of items where those items are accessed by an index value corresponding to the item position in the array. Dictionaries provide a platform for storing key-value pairs, where the key is used to gain access to the stored value. Iteration through the elements of Swift collections can be achieved using the for-in loop construct.

Chapter 18

18. SwiftUI Architecture

A completed SwiftUI app is constructed from multiple components which are assembled in a hierarchical manner. Before embarking on the creation of even the most basic of SwiftUI projects, it is useful to first gain an understanding of how SwiftUI apps are structured. With this goal in mind, this chapter will introduce the key elements of SwiftUI app architecture, with an emphasis on App, Scene and View elements.

18.1 SwiftUI App Hierarchy

When considering the structure of a SwiftUI application, it helps to view a typical hierarchy visually. Figure 18-1, for example, illustrates the hierarchy of a simple SwiftUI app:

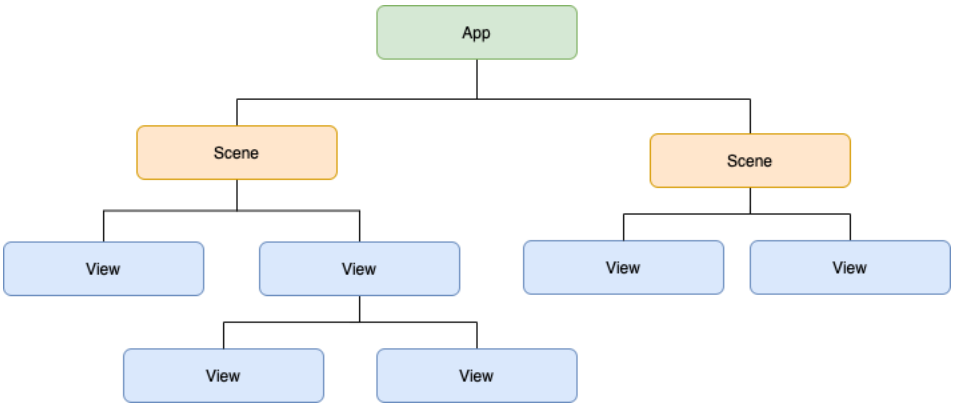


Figure 18-1

Before continuing, it is important to distinguish the difference between the term "app" and the "App" element outlined in the above figure. The software applications that we install and run on our mobile devices have come to be referred to as "apps". In this chapter reference will be made both to these apps and the App element in the above figure. To avoid confusion, we will use the term "application" to refer to the completed, installed and running app, while referring to the App element as "App". The remainder of the book will revert to using the more common "app" when talking about applications.

18.2 App

The App object is the top-level element within the structure of a SwiftUI application and is responsible for handling the launching and lifecycle of each running instance of the application.

The App element is also responsible for managing the various Scenes that make up the user interface of the application. An application will include only one App instance.

18.3 Scenes

Each SwiftUI application will contain one or more scenes. A scene represents a section or region of the application's user interface. On iOS and watchOS a scene will typically take the form of a window which takes up the entire device screen. SwiftUI applications running on macOS and iPadOS, on the other hand, will likely be comprised of multiple scenes. Different scenes might, for example, contain context specific layouts to be displayed when tabs are selected by the user within a dialog, or to design applications that consist of multiple

windows.

SwiftUI includes some pre-built primitive scene types that can be used when designing applications, the most common of which being WindowGroup and DocumentGroup. It is also possible to group scenes together to create your own custom scenes.

18.4 Views

Views are the basic building blocks that make up the visual elements of the user interface such as buttons, labels and text fields. Each scene will contain a hierarchy of the views that make up a section of the application's user interface. Views can either be individual visual elements such as text views or buttons, or take the form of containers that manage other views. The Vertical Stack view, for example, is designed to display child views in a vertical layout. In addition to the Views provided with SwiftUI, you will also create custom views when developing SwiftUI applications. These custom views will comprise groups of other views together with customizations to the appearance and behavior of those views to meet the requirements of the application's user interface.

Figure 18-2, for example, illustrates a scene containing a simple view hierarchy consisting of a Vertical Stack containing a Button and TextView combination:

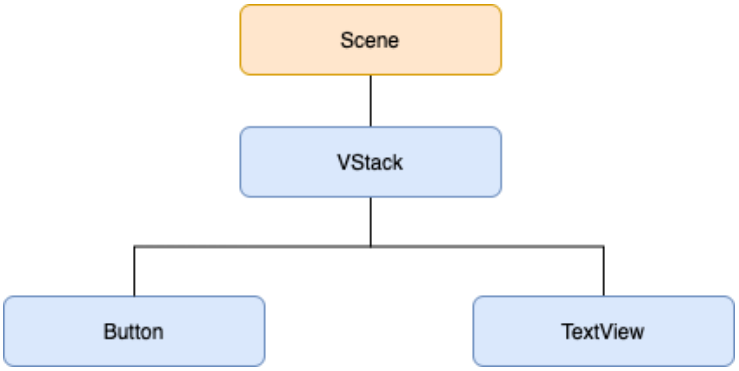


Figure 18-2

18.5 Summary

SwiftUI applications are constructed hierarchically. At the top of the hierarchy is the App instance which is responsible for the launching and lifecycle of the application. One or more child Scene instances contain hierarchies of the View instances that make up the user interface of the application. These scenes can either be derived from one of the SwiftUI primitive Scene types such as WindowGroup, or custom built.

On iOS or watchOS, an application will typically contain a single scene which takes the form of a window occupying the entire display. On a macOS or iPadOS system, however, an application may comprise multiple scene instances, often represented by separate windows which can be displayed simultaneously or grouped together in a tabbed interface.

Chapter 22

22. SwiftUI State Properties, Observation, and Environment Objects

Earlier chapters have described how SwiftUI emphasizes a data-driven approach to app development whereby the views in the user interface are updated in response to changes in the underlying data without the need to write handling code. This approach is achieved by establishing a publisher and subscriber relationship between the data and the views in the user interface.

SwiftUI offers four options for implementing this behavior in the form of *state properties*, *observation*, and *environment objects*, all of which provide the *state* that drives the way the user interface appears and behaves. In SwiftUI, the views that make up a user interface layout are never updated directly within code. Instead, the views are updated automatically based on the state objects to which they have been bound as they change over time.

This chapter will describe these four options and outline when they should be used. Later chapters, *"A SwiftUI Example Tutorial"* and *"SwiftUI Observable and Environment Objects – A Tutorial"*) will provide practical examples demonstrating their use.

22.1 State Properties

The most basic form of state is the state property. State properties are used exclusively to store state that is local to a view layout, such as whether a toggle button is enabled, the text being entered into a text field, or the current selection in a Picker view. State properties are used for storing simple data types such as a String or an Int value and are declared using the *@State* property wrapper, for example:

```
struct ContentView: View {

    @State private var wifiEnabled = true
    @State private var userName = ""

    var body: some View {
.
.
```

Since state values are local to the enclosing view, they should be declared as private properties.

Every change to a state property value signals to SwiftUI that the view hierarchy within which the property is declared needs to be re-rendered. This involves rapidly recreating and displaying all of the views in the hierarchy, which, in turn, ensures that any views that rely on the property in some way are updated to reflect the latest value.

Once declared, bindings can be established between state properties and the views contained in the layout. Changes within views referencing the binding are then automatically reflected in the corresponding state property. A binding could, for example, be established between a Toggle view and the Boolean wifiEnabled

property declared above. SwiftUI automatically updates the state property to match the new toggle setting whenever the user switches the toggle.

A binding to a state property is implemented by prefixing the property name with a '$' sign. In the following example, a TextField view establishes a binding to the userName state property to use as the storage for text entered by the user:

```
struct ContentView: View {

    @State private var wifiEnabled = true
    @State private var userName = ""

    var body: some View {
        VStack {
            TextField("Enter user name", text: $userName)
        }
    }
}
```

With each keystroke performed as the user types into the TextField, the binding will store the current text into the userName property. Each change to the state property will, in turn, cause the view hierarchy to be re-rendered by SwiftUI.

Of course, storing something in a state property is only one side of the process. As previously discussed, a state change usually results in a change to other views in the layout. In this case, a Text view might need to be updated to reflect the user's name as it is typed. This can be achieved by declaring the userName state property value as the content for a Text view:

```
var body: some View {
   VStack {
        TextField("Enter user name", text: $userName)
        Text(userName)
    }
}
```

The Text view will automatically update as the user types to reflect the user's input. The userName property is declared without the '$' prefix in this case. This is because we are now referencing the value assigned to the state property (i.e., the String value being typed by the user) instead of a binding to the property.

Similarly, the hypothetical binding between a Toggle view and the wifiEnabled state property described above could be implemented as follows:

```
var body: some View {

    VStack {
        Toggle(isOn: $wifiEnabled) {
            Text("Enable Wi-Fi")
        }
        TextField("Enter user name", text: $userName)
        Text(userName)
        Image(systemName: wifiEnabled ? "wifi" : "wifi.slash")
    }
```

```
}
```

The above declaration establishes a binding between the Toggle view and the state property. The value assigned to the property is then used to decide which image will be displayed on an Image view.

22.2 State Binding

A state property is local to the view it is declared in and any child views. Situations may occur, however, where a view contains one or more subviews that may also need access to the same state properties. Consider, for example, a situation whereby the WiFi Image view in the above example has been extracted into a subview:

```
.
.
    VStack {
        Toggle(isOn: $wifiEnabled) {
            Text("Enable WiFi")
        }
        TextField("Enter user name", text: $userName)
        WifiImageView()
    }
}

struct WifiImageView: View {

    var body: some View {
        Image(systemName: wifiEnabled ? "wifi" : "wifi.slash")
    }
}
```

Clearly, the WifiImageView subview still needs access to the wifiEnabled state property. As an element of a separate subview, however, the Image view is now out of the scope of the main view. Within the scope of WifiImageView, the wifiEnabled property is an undefined variable.

This problem can be resolved by declaring the property using the *@Binding* property wrapper as follows:

```
struct WifiImageView: View {

    @Binding var wifiEnabled : Bool

    var body: some View {
        Image(systemName: wifiEnabled ? "wifi" : "wifi.slash")
    }
}
```

Now, when the subview is called, it simply needs to be passed a binding to the state property:

```
WifiImageView(wifiEnabled: $wifiEnabled)
```

22.3 Observable Objects

State properties provide a way to store the state of a view locally, are available only to the local view, and, as such, cannot be accessed by other views unless they are subviews and state binding is implemented. State properties are also transient in that when the parent view goes away, the state is also lost. On the other hand, Observable objects represent persistent data that is both external and accessible to multiple views.

An Observable object takes the form of a class that conforms to the ObservableObject protocol. Though the implementation of an observable object will be application-specific depending on the nature and source of the data, it will typically be responsible for gathering and managing one or more data values known to change over time. Observable objects can also handle events such as timers and notifications.

The observable object *publishes* the data values it is responsible for as published properties. Observer objects then *subscribe* to the publisher and receive updates whenever published properties change. As with the state properties outlined above, by binding to these published properties, SwiftUI views will automatically update to reflect changes in the data stored in the observable object.

Before the introduction of iOS 17, observable objects were managed using the Combine framework, which was introduced to make it easier to establish relationships between publishers and subscribers. While this option is still available, a simpler alternative is now available following the introduction of the *Observation framework* (typically referred to as just "Observation").

However, before we look at how to use Observation, we will cover the old Combine framework approach. We are doing this for two reasons. First, learning about the old way will help you to understand how the new Observation works behind the scenes. Second, you will encounter many code examples online that use the Combine framework. Understanding how to migrate to Observation will help you re-purpose those examples for your needs.

22.4 Observation using Combine

The Combine framework provides a platform for building custom publishers for performing various tasks, from merging multiple publishers into a single stream to transforming published data to match subscriber requirements. This allows for complex, enterprise-level data processing chains to be implemented between the original publisher and the resulting subscriber. That being said, one of the built-in publisher types will typically be all that is needed for most requirements. The easiest way to implement a published property within an observable object is to use the *@Published* property wrapper when declaring a property. This wrapper sends updates to all subscribers each time the wrapped property value changes.

The following class shows a simple observable object declaration with two published properties:

```
import Foundation
import Combine

class DemoData : ObservableObject {

    @Published var playerName = ""
    @Published var score = 0

    init() {
        // Code here to initialize data
        updateData()
    }

    func updateData() {
        // Code here to update the data
        score += 1
    }
}
```

A subscriber uses either the *@ObservedObject* or *@StateObject* property wrapper to subscribe to the observable object. Once subscribed, that view and any of its child views access the published properties using the same techniques used with state properties earlier in the chapter. A sample SwiftUI view designed to subscribe to an instance of the above DemoData class might read as follows:

```
import SwiftUI

struct ContentView: View {

    @ObservedObject var demoData : DemoData = DemoData(player: "John")

    var body: some View {
        VStack {
            Text("\(demoData.playerName)'s Score = \(demoData.score)")
            Button(action: {
                demoData.update()
            }, label: {
                Text("Update")
            })
            .padding()
        }
    }
}
```

When the update button is clicked, the published score variable will change, and SwiftUI will automatically re-render the view layout to reflect the new state.

22.5 Combine State Objects

The State Object property wrapper (@StateObject) was introduced in iOS 14 as an alternative to the @ ObservedObject wrapper. The key difference between a state object and an observed object is that an observed object reference is not owned by the view in which it is declared and, as such, is at risk of being destroyed or recreated by the SwiftUI system while still in use (for example as the result of the view being re-rendered).

Using @StateObject instead of @ObservedObject ensures that the reference is owned by the view in which it is declared and, therefore, will not be destroyed by SwiftUI while it is still needed, either by the local view in which it is declared or any child views. For example:

```
import SwiftUI

struct ContentView: View {

    @StateObject var demoData : DemoData = DemoData()

    var body: some View {
.
.
}
```

22.6 Using the Observation Framework

Using Observation instead of the Combine framework will provide us with the same behavior outlined above but with simpler code. To switch the DemoData class to use Observation, we need to make the following changes:

```
import Foundation

@Observable class DemoData ~~: ObservableObject~~ {

    ~~@Published~~ var playerName = ""
    ~~@Published~~ var score = 0

    init(player: String) {
        self.playerName = player
    }

    func update() {
        score += 1
    }
}
```

Instead of declaring the DemoData as a subclass of ObservableObject, we now prefix the declaration with the @Observable macro. We also no longer need to use the @Published property wrappers because the macro handles this automatically.

The code in the ContentView is also simplified by removing the @ObservedObject directive:

```
struct ContentView: View {

    ~~@ObservedObject~~ var demoData : DemoData = DemoData(player: "John")
.
.
```

Where the @StateObject property wrapper was used, this can be replaced with @State as follows:

```
import SwiftUI

struct ContentView: View {

    @State var demoData : DemoData = DemoData()

    var body: some View {
.
.
}
```

22.7 Observation and @Bindable

Earlier in the chapter, we introduced state binding and explained how it is used to pass state properties from one view to another. Suppose that our example layout uses a separate view named ScoreView to display the score as follows:

```
struct ContentView: View {
```

```
    var demoData : DemoData = DemoData(player: "John")

    var body: some View {
        VStack {
            ScoreView(score: $demoData.score) // Syntax error
            Text("\(demoData.playerName)'s Score")
            Button(action: {
                demoData.update()
            }, label: {
                Text("Update")
            })
            .padding()
        }
    }
}

struct ScoreView: View {

    @Binding var score: Int

    var body: some View {
        Text("\(score)")
            .font(.system(size: 150))
    }
}
```

The above code will report an error indicating that $demoData.score cannot be found. To correct this, we need to apply the @Bindable property wrapper to the demoData declaration. This property wrapper is used when we need to create bindings from the properties of observable objects. To resolve the problem with the above example, we need to make the following change:

```
@Bindable var demoData : DemoData = DemoData(player: "John")
```

22.8 Environment Objects

Observed objects are best used when a particular state needs to be used by a few SwiftUI views within an app. When one view navigates to another view that needs access to the same observed or state object, the originating view will need to pass a reference to the observed object to the destination view during the navigation (navigation will be covered in the chapter entitled *"SwiftUI Lists and Navigation"*). Consider, for example, the following code:

```
.
.
var demoData : DemoData = DemoData()
.
.
NavigationLink(destination: SecondView(demoData)) {
    Text("Next Screen")
}
```

In the above declaration, a navigation link is used to navigate to another view named SecondView, passing through a reference to the demoData observed object.

While this technique is acceptable for many situations, it can become complex when many views within an app need access to the same observed object. In this situation, using an environment object may make more sense.

An environment object is declared in the same way as an observable object. The key difference, however, is that the object is stored in the environment of the view in which it is declared and, as such, can be accessed by all child views without needing to be passed from view to view.

Consider the following example observable object declaration:

```
@Observable class SpeedSetting {
    var speed = 0.0
}
```

Views needing to subscribe to an environment object reference the object using the @Environment property wrapper. For example, the following view uses @Environment to access the SpeedSetting data:

```
struct SpeedDisplayView: View {
    @Environment(SpeedSetting.self) var speedsetting: SpeedSetting

    var body: some View {
        Text("Speed = \(speedsetting.speed)")
    }
}
```

Suppose that a second view also needs access to the speed data but needs to create a binding to the *speed* property. In this case, we need to use the @Bindable property wrapper as follows:

```
struct SpeedControlView: View {
    @Environment(SpeedSetting.self) var speedsetting: SpeedSetting

    var body: some View {
        @Bindable var speedsetting = speedsetting
        Slider(value: $speedsetting.speed, in: 0...100)
    }
}
```

At this point, we have an observable object named SpeedSetting and two views that reference an environment object of that type. Still, we have not yet initialized an instance of the observable object. The logical place to perform this task is the parent view of the above sub-views. In the following example, both views are sub-views of the main ContentView:

```
struct ContentView: View {
    let speedsetting = SpeedSetting()

    var body: some View {
        VStack {
            SpeedControlView()
            SpeedDisplayView()
        }
    }
```

```
}
```

If the app were to run at this point, however, it would crash shortly after launching with the following diagnostics:

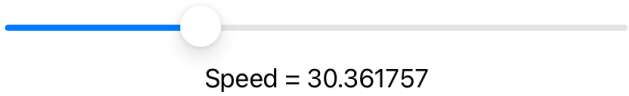
```
Thread 1: Fatal error: No ObservableObject of type SpeedSetting found. A View.
environmentObject(_:) for SpeedSetting may be missing as an ancestor of this
view.
```

The problem is that while we have created an instance of the observable object within ContentView, we still need to insert it into the view hierarchy environment. This is achieved using the *environment()* modifier, passing through the observable object instance as follows:

```
struct ContentView: View {
    let speedsetting = SpeedSetting()

    var body: some View {
        VStack {
            SpeedControlView()
            SpeedDisplayView()
        }
        .environment(speedsetting)
    }
}
```

Once these steps have been taken, the object will behave the same way as an observed object, except that it will be accessible to all child views of the content view without being passed down through the view hierarchy. When the slider in SpeedControlView is moved, the Text view in SpeedDisplayView will update to reflect the current speed setting, thereby demonstrating that both views are accessing the same environment object:



Speed = 30.361757

Figure 22-1

22.9 Summary

SwiftUI provides three ways to bind data to an app's user interface and logic. State properties store the views' state in a user interface layout and are local to the current content view. These transient values are lost when the view goes away.

The Observation framework can be used for data that is external to the user interface and is required only by a subset of the SwiftUI view structures in an app. Using this approach, the @Observable macro must be applied to the class that represents the data. To bind to an observable object property in a view declaration, the property must use the @Bindable property wrapper.

The environment object provides the best solution for data external to the user interface, but for which access is required for many views. Although declared the same way as observable objects, environment object bindings are declared in SwiftUI View files using the @Environment property wrapper. Before becoming accessible to child views, the environment object must also be initialized before being inserted into the view hierarchy using the *environment()* modifier.

Chapter 23

23. A SwiftUI Example Tutorial

Now that some of the fundamentals of SwiftUI development have been covered, this chapter will begin to put this theory into practice by building an example SwiftUI-based project.

This chapter aims to demonstrate using Xcode to design a simple interactive user interface using views, modifiers, state variables, and some basic animation effects. This tutorial will use various techniques to add and modify views. While this may appear inconsistent, the objective is to gain familiarity with the options available.

23.1 Creating the Example Project

Start Xcode and select the option to create a new project. Then, on the template selection screen, make sure Multiplatform is selected and choose the App option as shown in Figure 23-1 before proceeding to the next screen:

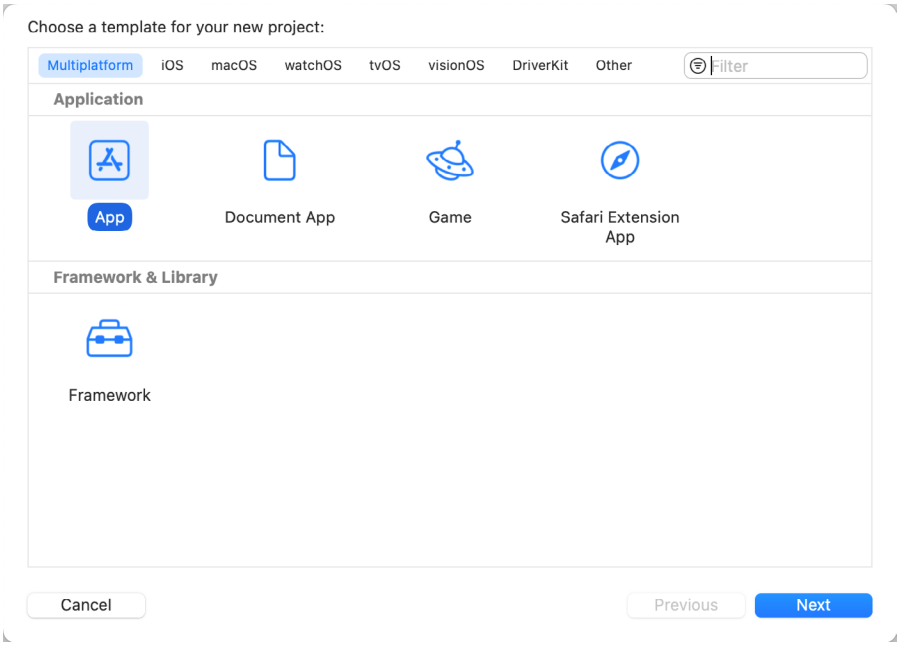


Figure 23-1

On the project options screen, name the project *SwiftUIDemo* before clicking Next to proceed to the final screen. Choose a suitable filesystem location for the project and click on the Create button.

23.2 Reviewing the Project

Once the project has been created, it will contain the *SwiftUIDemoApp.swift* file along with a SwiftUI View file named *ContentView.swift*, which should have loaded into the editor and preview canvas ready for modification (if it has not loaded, select it in the project navigator panel). Next, from the target device menu (Figure 23-2), select an iPhone 15 simulator:

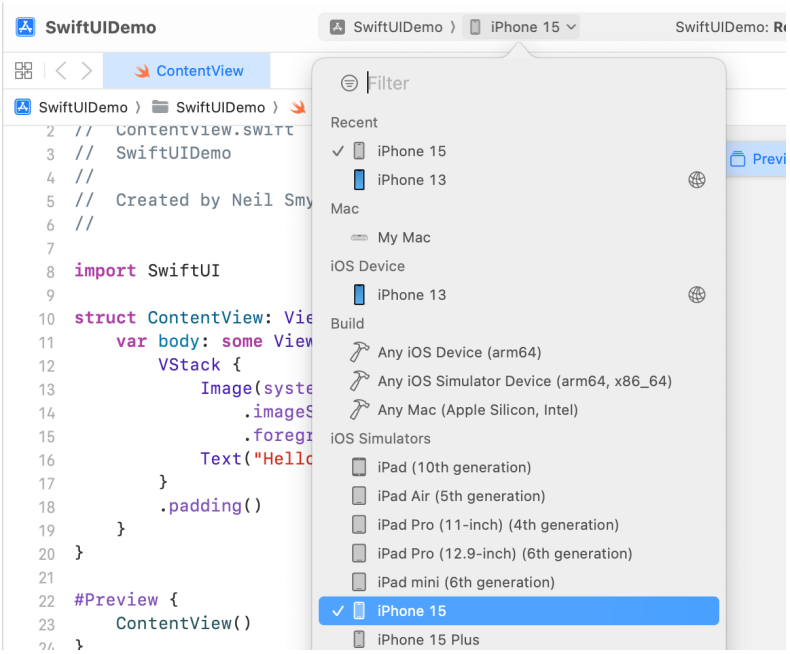


Figure 23-2

If the preview canvas is in the paused state, click on the Resume button to build the project and display the preview:

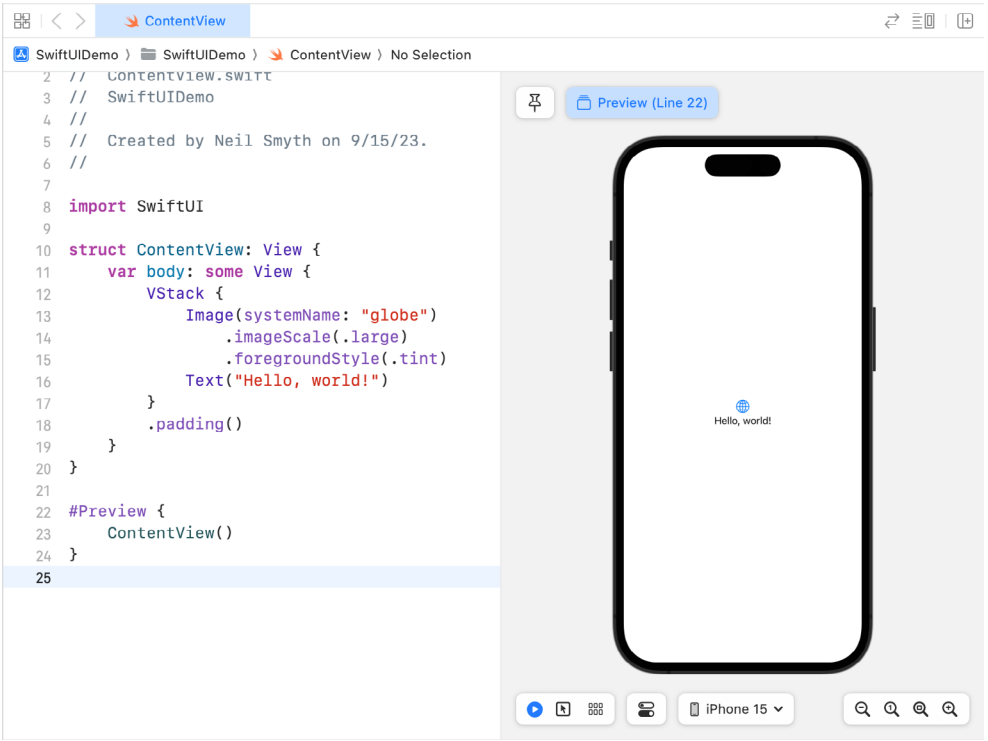


Figure 23-3

23.3 Modifying the Layout

The view body currently consists of a vertical stack layout (VStack) containing an Image and a Text view. Although we could reuse some of the existing layout for our example, we will learn more by deleting the current views and starting over. Within the Code Editor, delete the existing views from the ContentView body:

```
import SwiftUI

struct ContentView: View {
    var body: some View {
        VStack {
            Image(systemName: "globe")
                .imageScale(.large)
                .foregroundColor(.accentColor)
            Text("Hello, world!")
        }
        .padding()
    }
}
```

Next, add a Text view to the layout as follows:

```
struct ContentView: View {
    var body: some View {
        Text("Hello, world!")
    }
}
```

Right-click on the Text view entry within the code editor, and select the Embed in VStack option from the resulting menu:

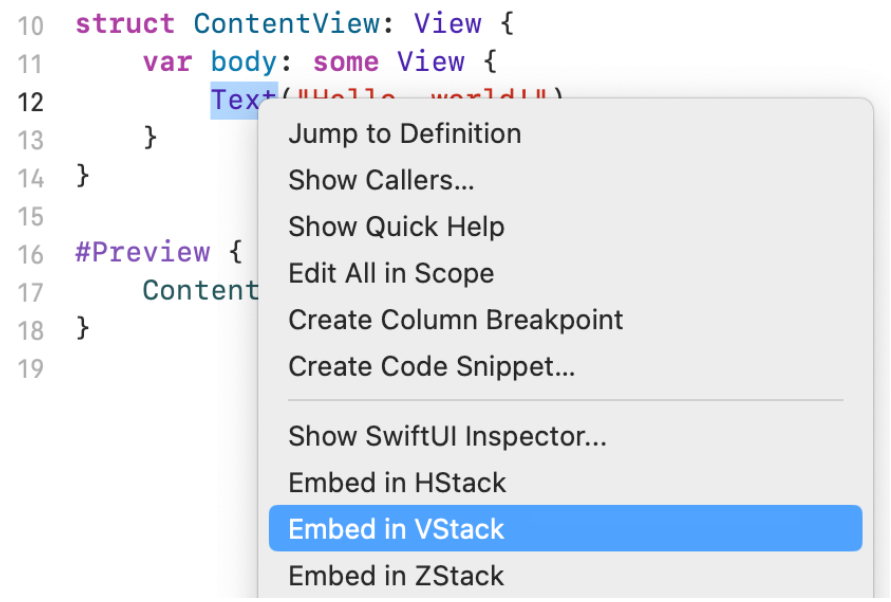


Figure 23-4

Once the Text view has been embedded into the VStack the declaration will read as follows:

```
struct ContentView: View {
    var body: some View {
        VStack {
            Text("Hello, world!")
        }
    }
}
```

23.4 Adding a Slider View to the Stack

The next item to be added to the layout is a Slider view. Display the Library panel by clicking on the '+' button highlighted in Figure 23-5, locating the Slider in the View list, and dragging it into position beneath the Text view in the editor. Ensure that the Slider view will be inserted into the existing stack before dropping the view into place:

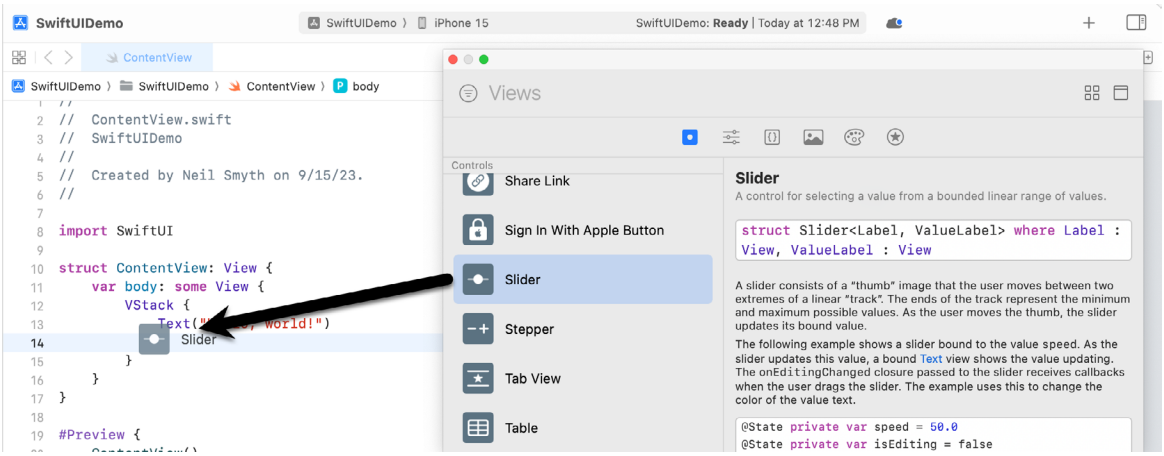


Figure 23-5

Once the slider has been dropped into place, the view implementation should read as follows:

```
struct ContentView: View {
    var body: some View {
        VStack {
            VStack {
                Text("Hello, world!")
                Slider(value: Value)
            }
        }
    }
}
```

23.5 Adding a State Property

The slider will be used to control the degree to which the Text view is to be rotated. As such, a binding must be established between the Slider view and a state property into which the current rotation angle will be stored. Within the code editor, declare this property and configure the Slider to use a range between 0 and 360 in increments of 0.1:

```
struct ContentView: View {

    @State private var rotation: Double = 0

    var body: some View {
        VStack {
            VStack {
                Text("Hello, world!")
                Slider(value: $rotation, in: 0 ... 360, step: 0.1)
            }
        }
    }
}
```

Note that since we are declaring a binding between the Slider view and the rotation state property, it is prefixed by a '$' character.

23.6 Adding Modifiers to the Text View

The next step is to add some modifiers to the Text view to change the font and adopt the rotation value stored by the Slider view. Begin by displaying the Library panel, switch to the modifier list, and drag and drop a font modifier onto the Text view entry in the code editor:

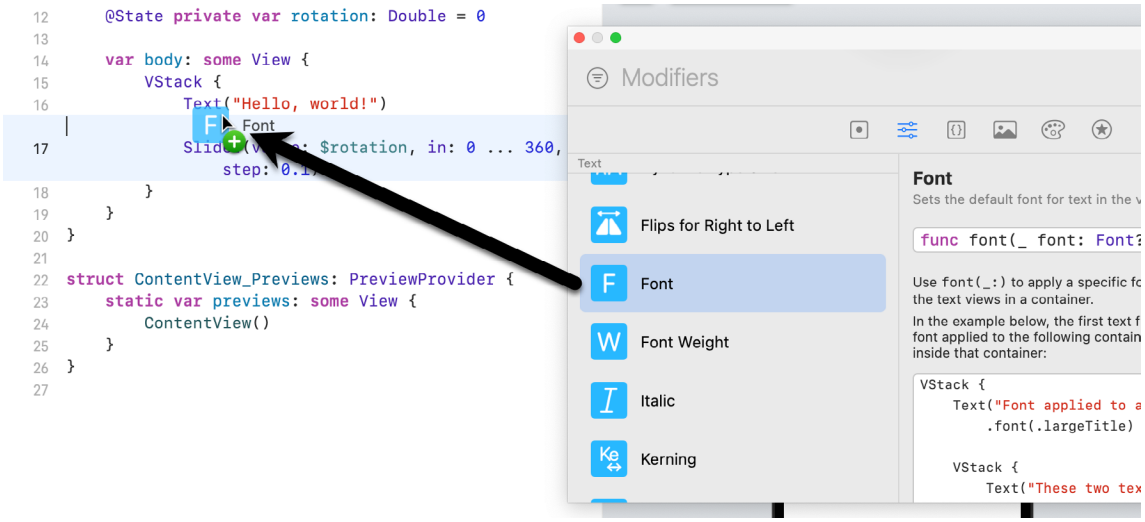


Figure 23-6

Select the modifier line in the editor, refer to the Attributes inspector panel, and change the font property from Title to Large Title, as shown in Figure 23-7:

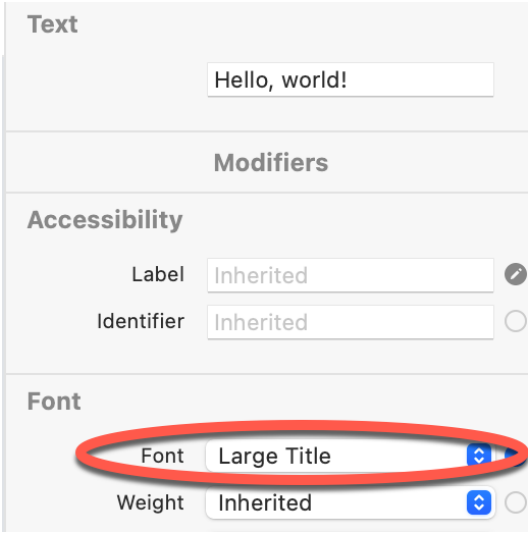


Figure 23-7

Note that the modifier added above does not change the font weight. Since modifiers may also be added to a view from within the Attributes inspector, take this opportunity to change the setting of the Weight menu from Inherited to Heavy.

On completion of these steps, the View body should read as follows:

```
var body: some View {
    VStack {
        VStack {
            Text("Hello, world!")
                .font(.largeTitle)
                .fontWeight(.heavy)
                Slider(value: $rotation, in: 0 ... 360, step: 0.1)
        }
    }
}
```

23.7 Adding Rotation and Animation

The next step is to add the rotation and animation effects to the Text view using the value stored by the Slider (animation is covered in greater detail in the *"SwiftUI Animation and Transitions"* chapter). This can be implemented using a modifier as follows:

```
Text("Hello, world!")
    .font(.largeTitle)
    .fontWeight(.heavy)
    .rotationEffect(.degrees(rotation))
```

Note that since we are simply reading the value assigned to the rotation state property, instead of establishing a binding, the property name is not prefixed with the '$' sign notation.

Click on the Live button (indicated by the arrow in Figure 23-8), wait for the code to compile, then use the slider to rotate the Text view:

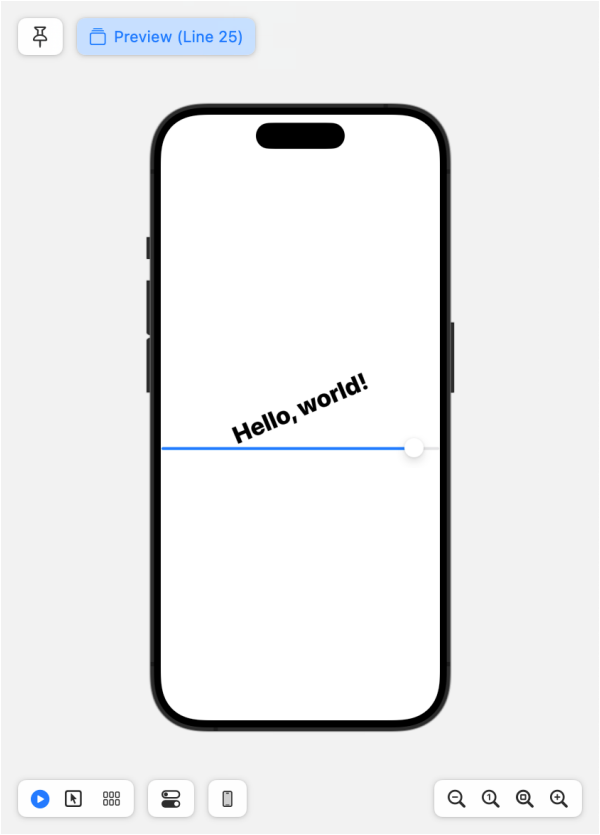


Figure 23-8

Next, add an animation modifier to the Text view to animate the rotation over 5 seconds using the Ease In Out effect:

```
Text("Hello, world!")
    .font(.largeTitle)
    .fontWeight(.heavy)
    .rotationEffect(.degrees(rotation))
    .animation(.easeInOut(duration: 5), value: rotation)
```

Use the slider once again to rotate the text, and note that rotation is now smoothly animated.

23.8 Adding a TextField to the Stack

In addition to supporting text rotation, the app will also allow custom text to be entered and displayed on the Text view. This will require the addition of a TextField view to the project. To achieve this, either directly edit the View structure or use the Library panel to add a TextField so that the structure reads as follows (also note the addition of a state property in which to store the custom text string and the change to the Text view to use this property):

```
struct ContentView: View {

    @State private var rotation: Double = 0
    @State private var text: String = "Welcome to SwiftUI"
```

```
    var body: some View {
        VStack {
            VStack {
                Text(text)
                    .font(.largeTitle)
                    .fontWeight(.heavy)
                    .rotationEffect(.degrees(rotation))
                    .animation(.easeInOut(duration: 5))

                Slider(value: $rotation, in: 0 ... 360, step: 0.1)

                TextField("Enter text here", text: $text)
                    .textFieldStyle(RoundedBorderTextFieldStyle())
            }
        }
    }
}
```

When the user enters text into the TextField view, that text will be stored in the *text* state property and will automatically appear on the Text view via the binding.

Return to the preview canvas and ensure that the changes work as expected.

23.9 Adding a Color Picker

A Picker view is the final view to be added to the stack before we tidy up the layout. The purpose of this view will be to allow the user to choose the foreground color of the Text view from a range of color options. Begin by adding some arrays of color names and Color objects, together with a state property to hold the current array index value as follows:

```
import SwiftUI

struct ContentView: View {

    var colors: [Color] = [.black, .red, .green, .blue]
    var colornames = ["Black", "Red", "Green", "Blue"]

    @State private var colorIndex = 0
    @State private var rotation: Double = 0
    @State private var text: String = "Welcome to SwiftUI"
```

With these variables configured, display the Library panel, locate the Picker in the Views screen, and drag and drop it beneath the TextField view in the code editor to embed it in the existing VStack layout. Once added, the view entry will read as follows:

```
Picker(selection: .constant(1), label: Text("Picker") {
    Text("1").tag(1)
    Text("2").tag(2)
}
```

The Picker view needs to be configured to store the current selection in the *colorIndex* state property and to

display an option for each color name in the *colorNames* array. In addition, to make the Picker more visually appealing, we will change the background color for each Text view to the corresponding color in the *colors* array.

To iterate through the colorNames array, the code will use the SwiftUI *ForEach* structure. At first glance, ForEach looks like just another Swift programming language control flow statement. In fact, *ForEach* is very different from the Swift *forEach()* array method outlined earlier in the book.

ForEach is a SwiftUI view structure designed to generate multiple views by looping through a data set such as an array or range. We may also configure the Picker view to display the color choices in various ways. For this project, we must select the WheelPickerStyle (*.wheel*) style via the *pickerStyle()* modifier. Within the editor, modify the Picker view declaration so that it reads as follows:

```
Picker(selection: $colorIndex, label: Text("Color")) {
    ForEach (0 ..< colornames.count, id:\.self)  { color in
        Text(colornames[color])
            .foregroundColor(colors[color])
    }
}
.pickerStyle(.wheel)
```

In the above implementation, ForEach is used to loop through the elements of the colornames array, generating a Text view for each color and setting the displayed text and background color on each view accordingly.

The ForEach loop in the above example is contained within a closure expression. As outlined in the *"Swift Functions, Methods, and Closures"* chapter, this expression can be simplified using *shorthand argument names*. Using this technique, modify the Picker declaration so that it reads as follows:

```
Picker(selection: $colorIndex, label: Text("Color")) {
    ForEach (0 ..< colornames.count, id:\.self) { color in
        Text(colornames[$0])
            .foregroundColor(colors[$0])
    }
}
.pickerStyle(.wheel)
```

Remaining in the code editor, locate the Text view and add a foreground color modifier to set the foreground color based on the current Picker selection value:

```
Text(text)
    .font(.largeTitle)
    .fontWeight(.heavy)
    .rotationEffect(.degrees(rotation))
    .animation(.easeInOut(duration: 5), value: rotation)
    .foregroundColor(colors[colorIndex])
```

Test the app in the preview canvas and confirm that the Picker view appears with all of the color names using the corresponding foreground color and that color selections are reflected in the Text view.

23.10 Tidying the Layout

Until this point, the focus of this tutorial has been on the appearance and functionality of the individual views. Aside from making sure the views are stacked vertically, however, no attention has been paid to the overall appearance of the layout. At this point, the layout should resemble that shown in Figure 23-9:

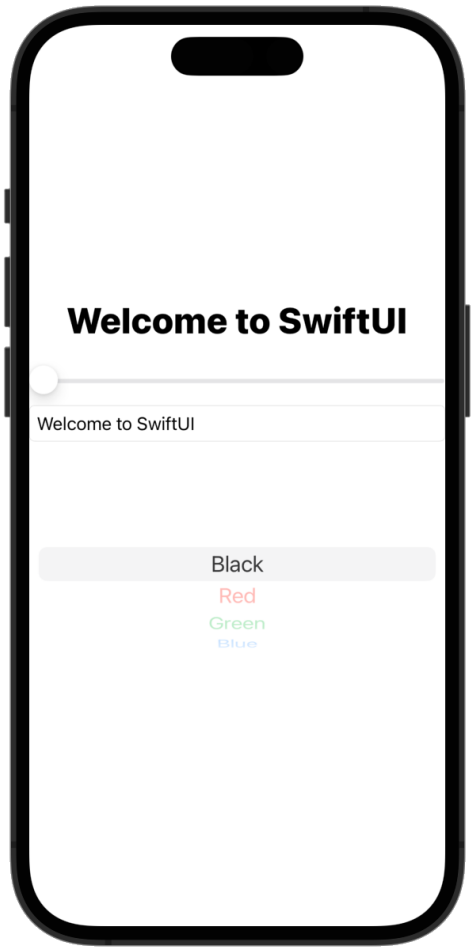


Figure 23-9

The first improvement needed is to add some space around the Slider, TextField, and Picker views so that they are not so close to the edge of the device display. To implement this, we will add some padding modifiers to the views:

```
Slider(value: $rotation, in: 0 ... 360, step: 0.1)
    .padding()

TextField("Enter text here", text: $text)
    .textFieldStyle(RoundedBorderTextFieldStyle())
    .padding()

Picker(selection: $colorIndex, label: Text("Color")) {
    ForEach (0 ..< colornames.count, id:\.self) {
        Text(colornames[$0])
            .foregroundColor(colors[$0])
    }
}
```

```
.pickerStyle(.wheel)
.padding()
```

Next, the layout would look better if the Views were evenly spaced. One way to implement this is to add some Spacer views before and after the Text view:

```
.
.
VStack {
    Spacer()
    Text(text)
        .font(.largeTitle)
        .fontWeight(.heavy)
        .rotationEffect(.degrees(rotation))
        .animation(.easeInOut(duration: 5), value: rotation)
        .foregroundColor(colors[colorIndex])
    Spacer()
    Slider(value: $rotation, in: 0 ... 360, step: 0.1)
            .padding()
.
.
```

The Spacer view provides a flexible space between views that will expand and contract based on the requirements of the layout. If a Spacer is contained in a stack, it will resize along the stack axis. When used outside a stack container, a Spacer view can resize horizontally and vertically.

To make the separation between the Text view and the Slider more obvious, also add a Divider view to the layout:

```
.
.
VStack {
    Spacer()
    Text(text)
        .font(.largeTitle)
        .fontWeight(.heavy)
        .rotationEffect(.degrees(rotation))
        .animation(.easeInOut(duration: 5), value: rotation)
        .foregroundColor(colors[colorIndex])
    Spacer()
    Divider()
.
.
```

The Divider view draws a line to indicate the separation between two views in a stack container.

With these changes made, the layout should now appear in the preview canvas, as shown in Figure 23-10:

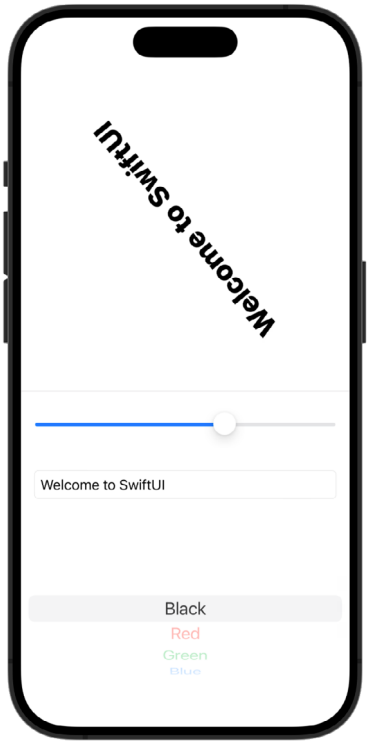


Figure 23-10

23.11 Summary

The goal of this chapter has been to put into practice some of the theory covered in the previous chapters through the creation of an example app project. In particular, the tutorial used various techniques for adding views to a layout and using modifiers and state property bindings. The chapter also introduced the Spacer and Divider views and used the ForEach structure to dynamically generate views from a data array.

Chapter 48

48. An Introduction to Core Data and SwiftUI

A common requirement when developing iOS apps is to store data in some form of structured database. One option is to directly manage data using an embedded database system such as SQLite. While this is a perfectly good approach for working with SQLite in many cases, it does require knowledge of SQL and can lead to some complexity in terms of writing code and maintaining the database structure. This complexity is further compounded by the non-object-oriented nature of the SQLite API functions. In recognition of these shortcomings, Apple introduced the Core Data Framework. Core Data is essentially a framework that places a wrapper around the SQLite database (and other storage environments) enabling the developer to work with data in terms of Swift objects without requiring any knowledge of the underlying database technology.

We will begin this chapter by defining some of the concepts that comprise the Core Data model before providing an overview of the steps involved in working with this framework. Once these topics have been covered, the next chapter will work through a SwiftUI Core Data tutorial.

48.1 The Core Data Stack

Core Data consists of several framework objects that integrate to provide the data storage functionality. This stack can be visually represented as illustrated in Figure 48-1:

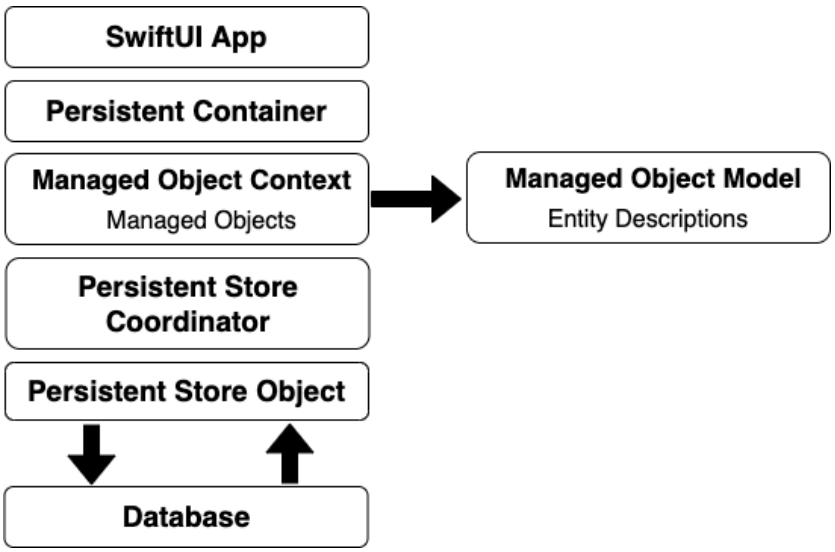


Figure 48-1

As we can see from Figure 48-1, the app sits on top of the stack and interacts with the managed data objects handled by the managed object context. Of particular significance in this diagram is the fact that although the lower levels in the stack perform a considerable amount of the work involved in providing Core Data functionality, the application code does not interact with them directly.

Before moving on to the more practical areas of working with Core Data it is important to spend some time explaining the elements that comprise the Core Data stack in a little more detail.

48.2 Persistent Container

The persistent container handles the creation of the Core Data stack and is designed to be easily subclassed to add additional application-specific methods to the base Core Data functionality. Once initialized, the persistent container instance provides access to the managed object context.

48.3 Managed Objects

Managed objects are the objects that are created by your application code to store data. A managed object may be thought of as a row or a record in a relational database table. For each new record to be added, a new managed object must be created to store the data. Similarly, retrieved data will be returned in the form of managed objects, one for each record matching the defined retrieval criteria. Managed objects are instances of the NSManagedObject class, or a subclass thereof. These objects are contained and maintained by the managed object context.

48.4 Managed Object Context

Core Data-based applications never interact directly with the persistent store. Instead, the application code interacts with the managed objects contained in the managed object context layer of the Core Data stack. The context maintains the status of the objects in relation to the underlying data store and manages the relationships between managed objects defined by the managed object model. All interactions with the underlying database are held temporarily within the context until the context is instructed to save the changes, at which point the changes are passed down through the Core Data stack and written to the persistent store.

48.5 Managed Object Model

So far we have focused on the management of data objects but have not yet looked at how the data models are defined. This is the task of the Managed Object Model which defines a concept referred to as entities.

Much as a class description defines a blueprint for an object instance, entities define the data model for managed objects. In essence, an entity is analogous to the schema that defines a table in a relational database. As such, each entity has a set of attributes associated with it that define the data to be stored in managed objects derived from that entity. For example, a Contacts entity might contain name, address, and phone number attributes.

In addition to attributes, entities can also contain relationships, fetched properties, persistent stores, and fetch requests:

- **Relationships** – In the context of Core Data, relationships are the same as those in other relational database systems in that they refer to how one data object relates to another. Core Data relationships can be one-to-one, one-to-many, or many-to-many.
- **Fetched property** – This provides an alternative to defining relationships. Fetched properties allow properties of one data object to be accessed from another data object as though a relationship had been defined between those entities. Fetched properties lack the flexibility of relationships and are referred to by Apple's Core Data documentation as "weak, one-way relationships" best suited to "loosely coupled relationships".
- **Fetch request** – A predefined query that can be referenced to retrieve data objects based on defined predicates. For example, a fetch request can be configured into an entity to retrieve all contact objects where the name field matches "John Smith".

48.6 Persistent Store Coordinator

The persistent store coordinator is responsible for coordinating access to multiple persistent object stores. As an iOS developer, you will never directly interact with the persistent store coordinator and will very rarely need to develop an application that requires more than one persistent object store. When multiple stores are required, the coordinator presents these stores to the upper layers of the Core Data stack as a single store.

48.7 Persistent Object Store

The term persistent object store refers to the underlying storage environment in which data are stored when using Core Data. Core Data supports three disk-based and one memory-based persistent store. Disk-based options consist of SQLite, XML, and binary. By default, iOS will use SQLite as the persistent store. In practice, the type of store being used is transparent to you as the developer. Regardless of your choice of persistent store, your code will make the same calls to the same Core Data APIs to manage the data objects required by your application.

48.8 Defining an Entity Description

Entity descriptions may be defined from within the Xcode environment. When a new project is created with the option to include Core Data, a template file will be created named *<entityname>.xcdatamodeld*. Xcode also provides a way to manually add entity description files to existing projects. Selecting this file in the Xcode project navigator panel will load the model into the entity editing environment as illustrated in Figure 48-2:

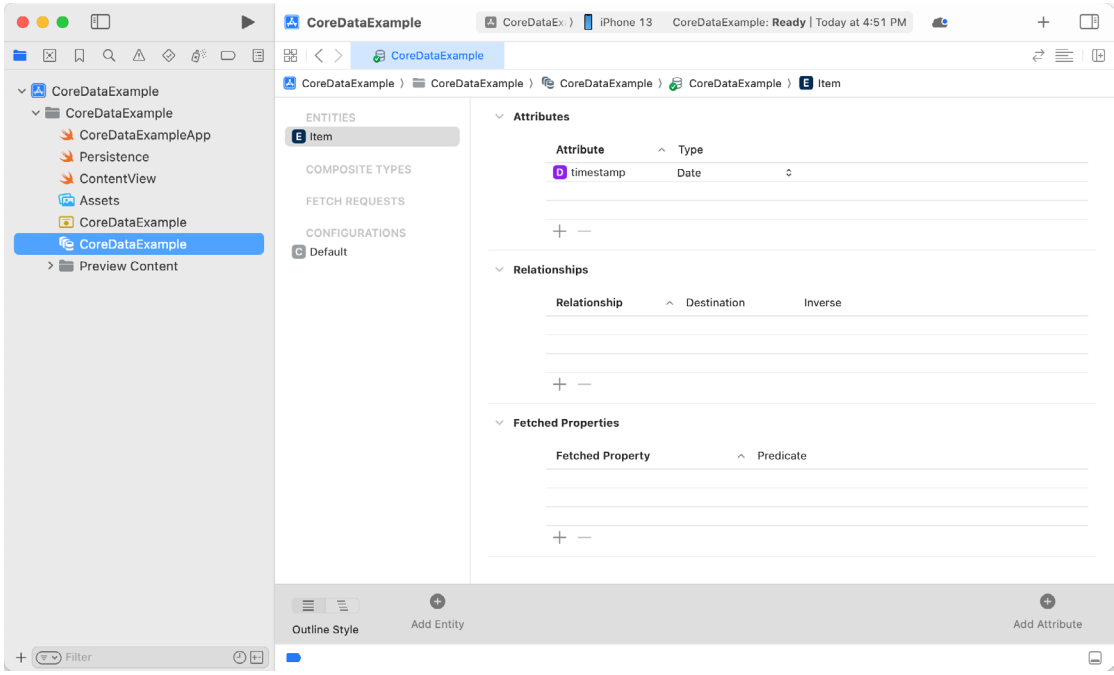


Figure 48-2

Create a new entity by clicking on the Add Entity button located in the bottom panel. The new entity will appear as a text box in the Entities list. By default, this will be named Entity. Double-click on this name to change it.

To add attributes to the entity, click on the Add Attribute button located in the bottom panel, or use the + button located beneath the Attributes section. In the Attributes panel, name the attribute and specify the type and any other options that are required.

Repeat the above steps to add more attributes and additional entities.

The Xcode entity editor also allows relationships to be established between entities. Assume, for example, two entities named Contacts and Sales. To establish a relationship between the two tables select the Contacts entity and click on the + button beneath the Relationships panel. In the detail panel, name the relationship, specify the destination as the Sales entity, and any other options that are required for the relationship:

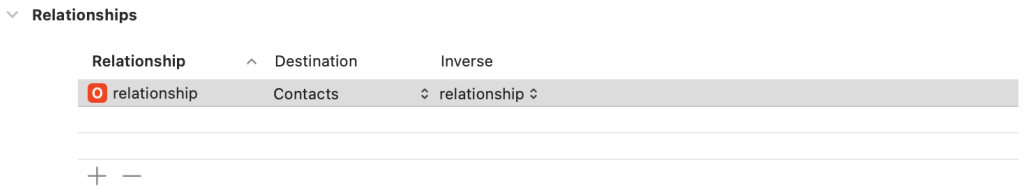


Figure 48-3

48.9 Initializing the Persistent Container

The persistent container is initialized by creating a new NSPersistentContainer instance, passing through the name of the model to be used, and then making a call to the *loadPersistentStores* method of that object as follows:

```
let persistentContainer: NSPersistentContainer

persistentContainer = NSPersistentContainer(name: "DemoData")
persistentContainer.loadPersistentStores { (storeDescription, error) in
    if let error = error as NSError? {
        fatalError("Container load failed: \(error)")
    }
}
```

48.10 Obtaining the Managed Object Context

Since many of the Core Data methods require the managed object context as an argument, the next step after defining entity descriptions often involves obtaining a reference to the context. This can be achieved by accessing the viewContext property of the persistent container instance:

```
let managedObjectContext = persistentContainer.viewContext
```

48.11 Setting the Attributes of a Managed Object

As previously discussed, entities and the managed objects from which they are instantiated contain data in the form of attributes. Once a managed object instance has been created as outlined above, those attribute values can be used to store the data before the object is saved. Assuming a managed object named *contact* with attributes named *name*, *address* and *phone* respectively, the values of these attributes may be set as follows before saving the object to storage:

```
contact.name = "John Smith"
contact.address = "1 Infinite Loop"
contact.phone = "555-564-0980"
```

48.12 Saving a Managed Object

Once a managed object instance has been created and configured with the data to be stored it can be saved to storage using the *save()* method of the managed object context as follows:

```
do {
    try viewContext.save()
```

```
} catch {
    let error = error as NSError
    fatalError("An error occured: \(error)")
}
```

48.13 Fetching Managed Objects

Once managed objects are saved into the persistent object store those objects and the data they contain will likely need to be retrieved. One way to fetch data from Core Data storage is to use the @FetchRequest property wrapper when declaring a variable in which to store the data. The following code, for example, declares a variable named *customers* which will be automatically updated as data is added to or removed from the database:

```
@FetchRequest(entity: Customer.entity(), sortDescriptors: [])
private var customers: FetchedResults<Customer>
```

The @FetchRequest property wrapper may also be configured to sort the fetched results. In the following example, the customer data stored in the *customers* variable will be sorted alphabetically in ascending order based on the *name* entity attribute:

```
@FetchRequest(entity: Customer.entity(),
        sortDescriptors: [NSSortDescriptor(key: "name", ascending: true)])
private var customers: FetchedResults<Customer>
```

48.14 Retrieving Managed Objects based on Criteria

The preceding example retrieved all of the managed objects from the persistent object store. More often than not only managed objects that match specified criteria are required during a retrieval operation. This is performed by defining a predicate that dictates criteria that a managed object must meet to be eligible for retrieval. For example, the following code configures a @FetchRequest property wrapper declaration with a predicate to extract only those managed objects where the name attribute matches "John Smith":

```
@FetchRequest(
  entity: Customer.entity(),
  sortDescriptors: [],
  predicate: NSPredicate(format: "name LIKE %@", "John Smith")
)
private var customers: FetchedResults<Customer>
```

The above example will maintain the *customers* variable so that it always contains the entries that match the specified predicate criteria. It is also possible to perform one-time fetch operations by creating NSFetchRequest instances, configuring them with the entity and predicate settings, and then passing them to the *fetch()* method of the managed object context. For example:

```
@State var matches: [Customer]?
let fetchRequest: NSFetchRequest<Product> = Product.fetchRequest()

fetchRequest.entity = Customer.entity()
fetchRequest.predicate = NSPredicate(
    format: "name LIKE %@", "John Smith"
)

matches = try? viewContext.fetch(fetchRequest)
```

48.15 Summary

The Core Data Framework stack provides a flexible alternative to directly managing data using SQLite or other data storage mechanisms. By providing an object-oriented abstraction layer on top of the data the task of managing data storage is made significantly easier for the SwiftUI application developer. Now that the basics of Core Data have been covered, the next chapter entitled *"A SwiftUI Core Data Tutorial"* will work through the creation of an example application.

Chapter 51

51. A SwiftUI Core Data and CloudKit Tutorial

Using the CoreDataDemo project created in the chapter entitled *"A SwiftUI Core Data Tutorial"*, this chapter will demonstrate how to add CloudKit support to an Xcode project and migrate from Core Data to CloudKit-based storage. This chapter assumes that you have read the chapter entitled *"An Introduction to Core Data and SwiftUI"*.

51.1 Enabling CloudKit Support

Begin by launching Xcode and opening the CoreDataDemo project. Once the project has loaded into Xcode, the first step is to add the iCloud capability to the app. Select the *CoreDataDemo* target located at the top of the Project Navigator panel (marked A in Figure 51-1) so that the main panel displays the project settings. From within this panel, select the *Signing & Capabilities* tab (B) followed by the CoreDataDemo target entry (C):

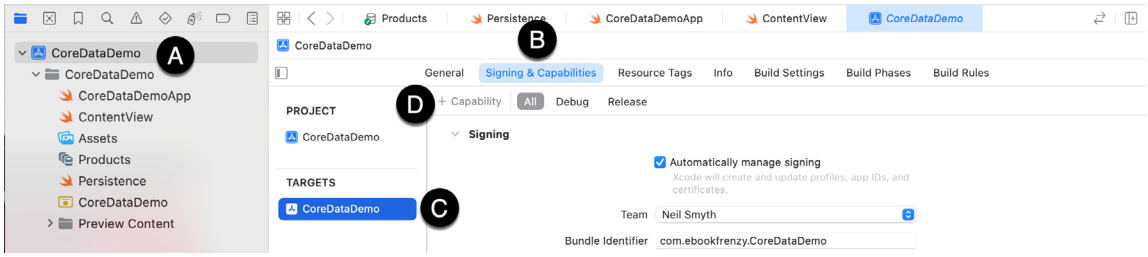


Figure 51-1

Click on the "+ Capability" button (D) to display the dialog shown in Figure 51-2. Enter *iCloud* into the filter bar, select the result and press the keyboard enter key to add the capability to the project:

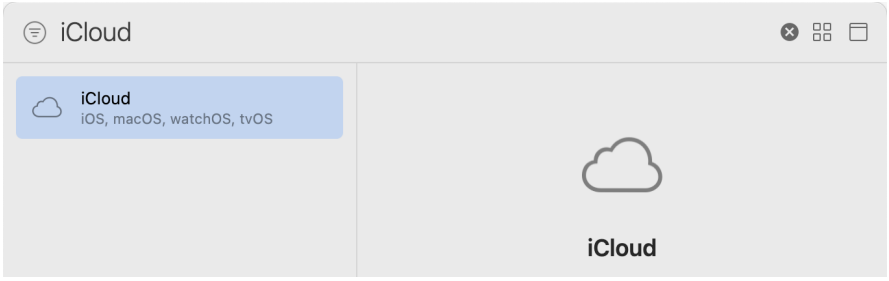


Figure 51-2

If iCloud is not listed as an option, you will need to pay to join the Apple Developer program as outlined in the chapter entitled *"Joining the Apple Developer Program"*. If you are already a member, use the steps outlined in the chapter entitled *"Installing Xcode 15 and the iOS 17 SDK"* to ensure you have created a *Developer ID Application* certificate.

Within the iCloud entitlement settings, make sure that the CloudKit service is enabled before clicking on the "+" button indicated by the arrow in Figure 51-3 below to add an iCloud container for the project:

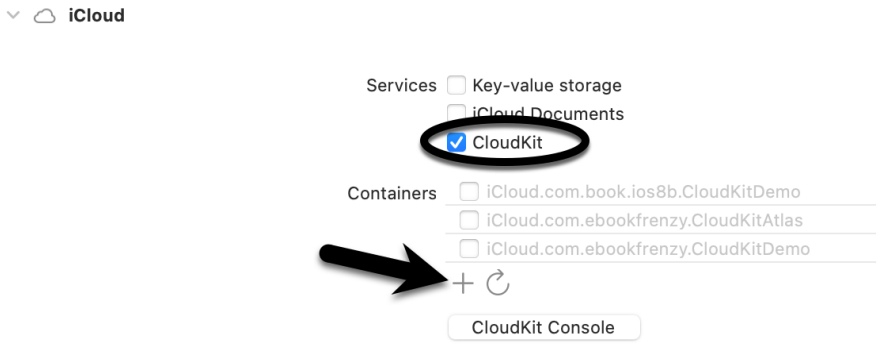


Figure 51-3

After clicking the "+" button, the dialog shown in Figure 51-4 will appear containing a text field into which you will need to enter the container identifier. This entry should uniquely identify the container within the CloudKit ecosystem, generally includes your organization identifier (as defined when the project was created), and should be set to something similar to *iCloud.com.yourcompany.CoreDataDemo*.

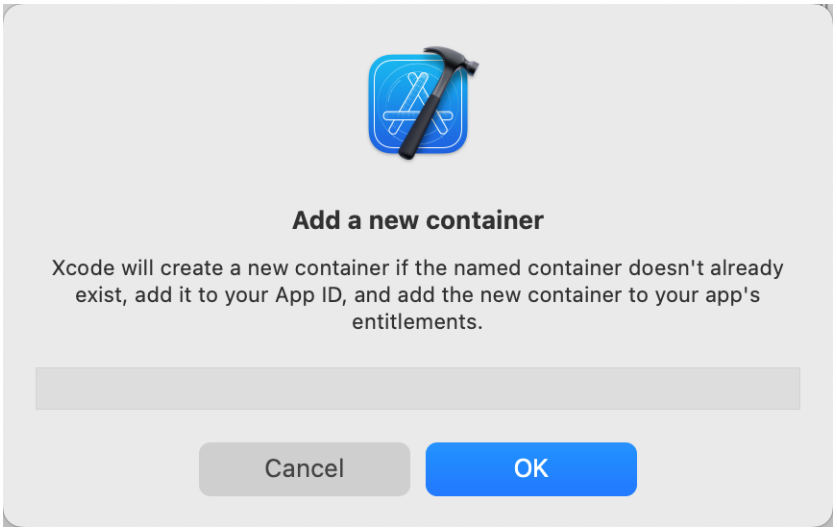


Figure 51-4

Once you have entered the container name, click the OK button to add it to the app entitlements. Returning to the *Signing & Capabilities* screen, make sure that the new container is selected:

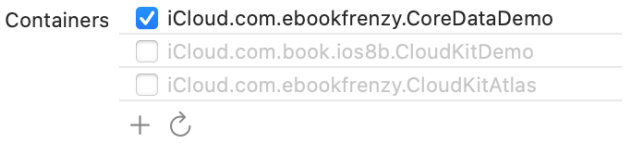


Figure 51-5

51.2 Enabling Background Notifications Support

When the app is running on multiple devices and a data change is made in one instance of the app, CloudKit will use remote notifications to notify other instances of the app to update to the latest data. To enable background

notifications, repeat the above steps, this time adding the *Background Modes* capability. Once the capability has been added, review the settings and make sure that *Remote notifications* mode is enabled as highlighted in Figure 51-6:

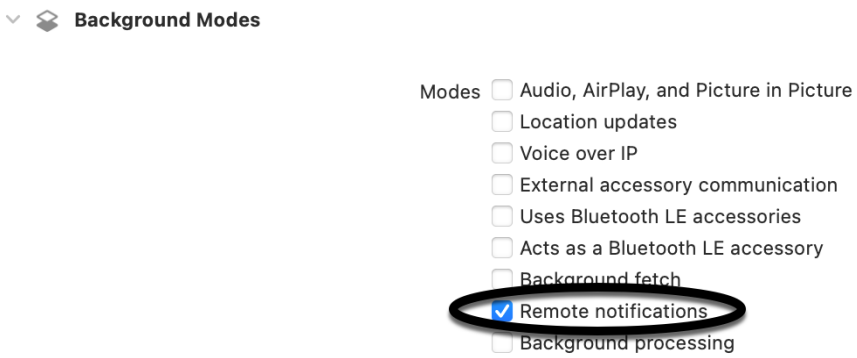


Figure 51-6

Now that the necessary entitlements have been enabled for the app, all that remains is to make some minor code changes to the project.

51.3 Switching to the CloudKit Persistent Container

Locate the *Persistence.swift* file in the project navigator panel and select it so that it loads into the code editor. Within the *init()* function, change the container creation call from NSPersistentContainer to NSPersistentCloudKitContainer as follows:

```
.
.
let container: NSPersistentCloudKitContainer
.
.
init() {
    container = NSPersistentCloudKitContainer(name: "Products")

    container.loadPersistentStores { (storeDescription, error) in
        if let error = error as NSError? {
            fatalError("Container load failed: \(error)")
        }
    }
}
```

Since multiple instances of the app could potentially change the same data at the same time, we also need to define a merge policy to make sure that conflicting changes are handled:

```
init() {
    container = NSPersistentCloudKitContainer(name: "Products")

    container.loadPersistentStores { (storeDescription, error) in
        if let error = error as NSError? {
            fatalError("Container load failed: \(error)")
        }
```

```
    }
    container.viewContext.automaticallyMergesChangesFromParent = true
}
```

51.4 Testing the App

CloudKit storage can be tested on either physical devices, simulators, or a mixture of both. All test devices and simulators must be signed in to iCloud using your Apple developer account and have the iCloud Drive option enabled. Once these requirements have been met, run the CoreDataDemo app and add some product entries. Next, run the app on another device or simulator and check that the newly added products appear. This confirms that the data is being stored and retrieved from iCloud.

With both app instances running, enter a new product in one instance and check that it appears in the other. Note that a bug in the simulator means that you may need to place the app in the background and then restore it before the new data will appear.

51.5 Reviewing the Saved Data in the CloudKit Console

Once some product entries have been added to the database, return to the *Signing & Capabilities* screen for the project (Figure 51-1) and click on the CloudKit Console button. This will launch the default web browser on your system and load the CloudKit Dashboard portal. Enter your Apple developer login and password and, once the dashboard has loaded, the home screen will provide the range of options illustrated in Figure 51-7:

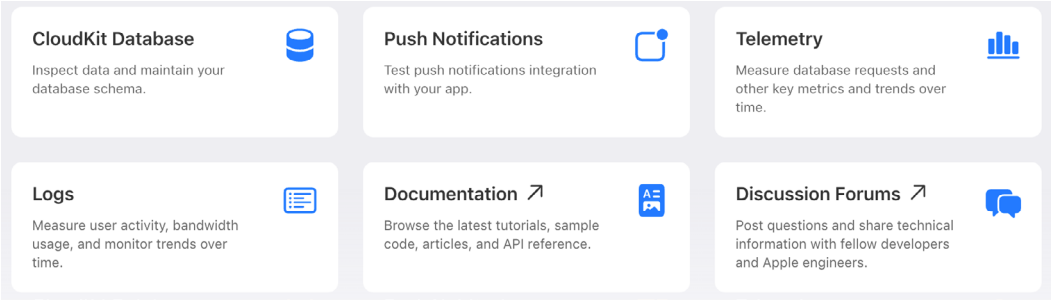


Figure 51-7

Select the CloudKit Database option and, on the resulting web page, select the container for your app from the drop-down menu (marked A in Figure 51-8 below). Since the app is still in development and has not been published to the App Store, make sure that menu B is set to Development and not Production:

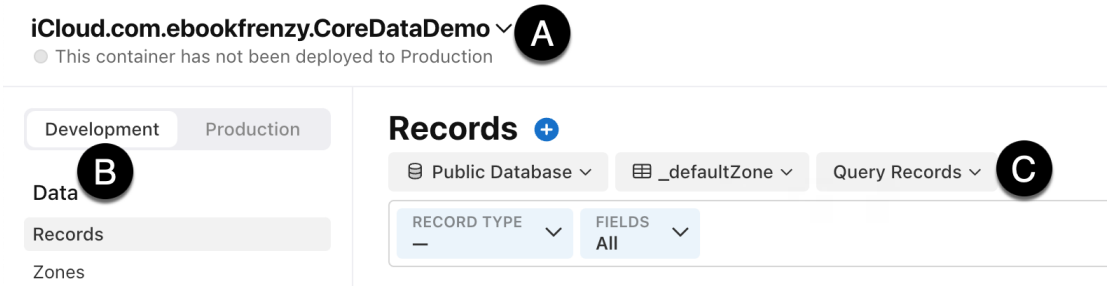


Figure 51-8

Next, we can query the records stored in the app container's private database. Set the row of menus (C) to *Private Database*, *com.apple.coredata.cloudkit.zone*, and *Query Records* respectively. Finally, set the Record Type menu to *CD_Product* and the *Fields* menu to All:

Figure 51-9

Clicking on the Query Records button should display a list of all the product items saved in the database as illustrated in Figure 51-10:

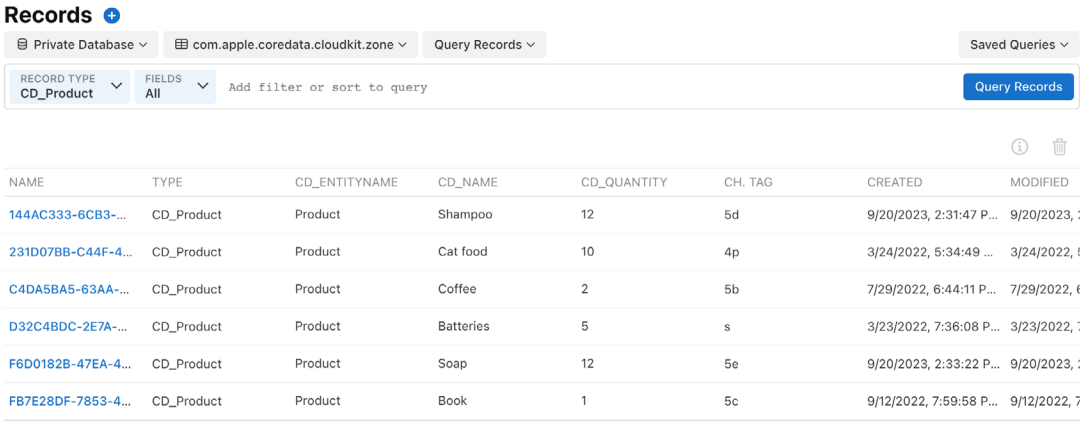


Figure 51-10

51.6 Filtering and Sorting Queries

The queries we have been running so far are returning all of the records in the database. Queries may also be performed based on sorting and filtering criteria by clicking in the "Add filter or sort to query" field. Clicking in this field will display a menu system that will guide you through setting up the criteria. In Figure 51-11, for example, the menu system is being used to set up a filtered query based on the *CD_name* field:

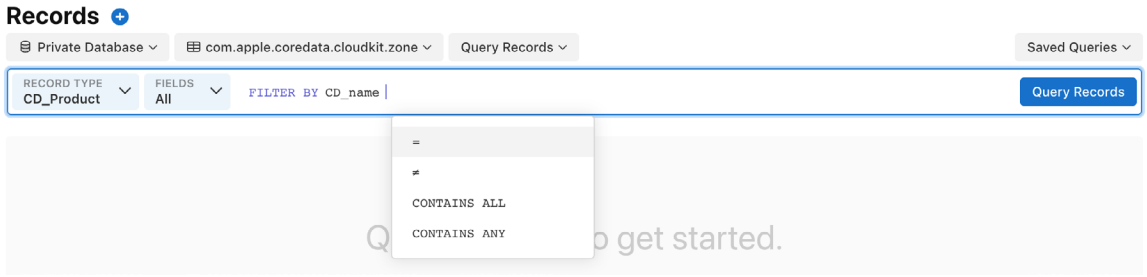


Figure 51-11

Similarly, Figure 51-12 shows the completed filter and query results:

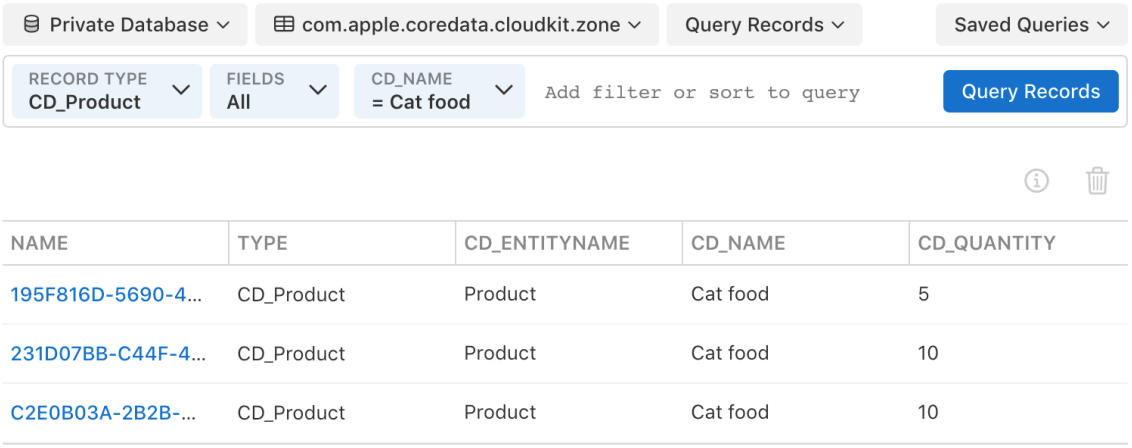


Figure 51-12

The same technique can be used to sort the results in ascending or descending order. You can also combine multiple criteria in a single query. To edit or remove a query criterion, left-click on it and select the appropriate menu option.

51.7 Editing and Deleting Records

In addition to querying the records in the database, the CloudKit Console also allows records to be edited and deleted. To edit or delete a record, locate it in the query list and click on the entry in the name column as highlighted below:

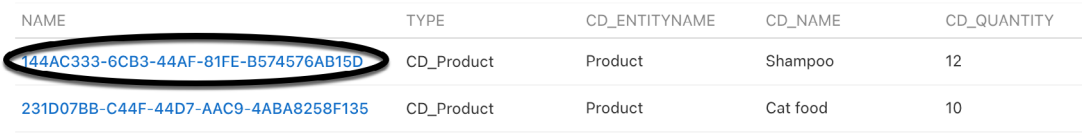


Figure 51-13

Once the record has been selected, the Record Details panel shown in Figure 51-14 will appear. In addition to displaying detailed information about the record, this panel also allows the record to be modified or deleted.

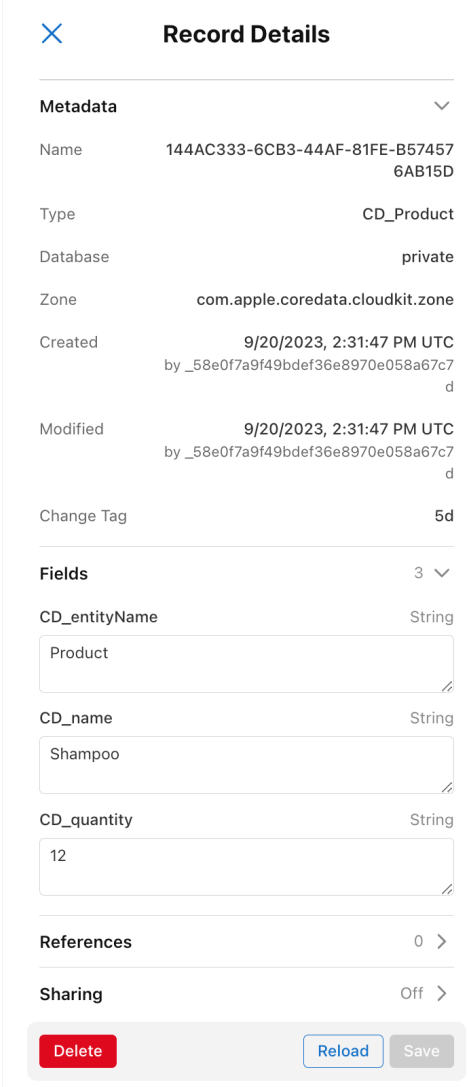


Figure 51-14

51.8 Adding New Records

To add a new record to a database, click on the "+" located at the top of the query results list and select the Create New Record option:

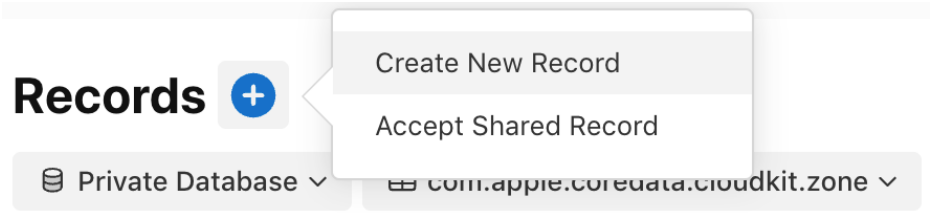


Figure 51-15

When the New Record panel appears (Figure 51-16) enter the new data before clicking the Save button:

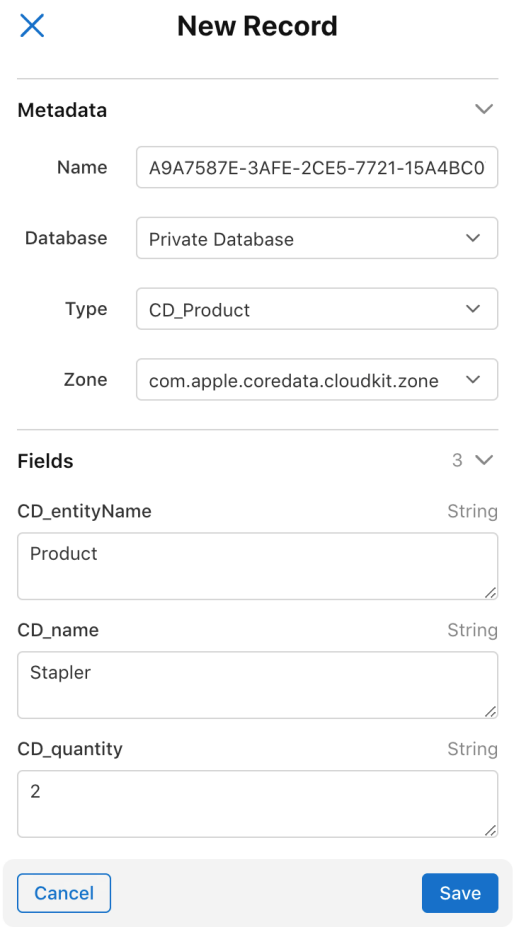


Figure 51-16

51.9 Viewing Telemetry Data

To view telemetry data, return to the console home screen (Figure 51-7) and select the Telemetry option. Within the telemetry screen, select the container, environment, timescale, and database type options:

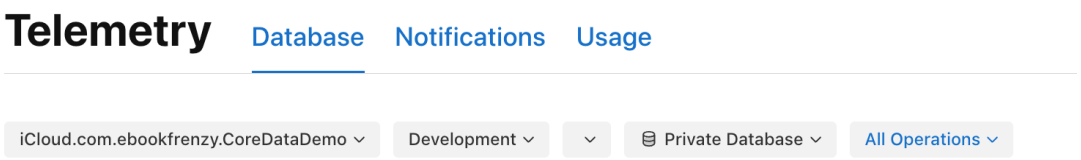


Figure 51-17

Hovering the mouse pointer over a graph will display a key explaining the metric represented by the different line colors:

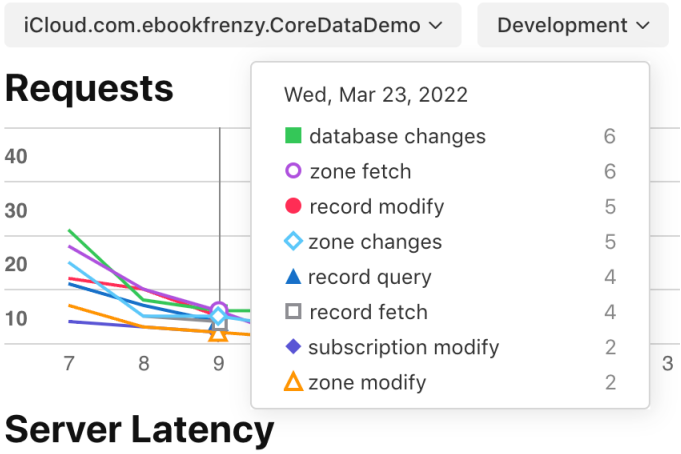


Figure 51-18

By default, telemetry data is displayed for database activity. This can be changed to display data relating to notifications or database usage using tabs highlighted in Figure 51-19:

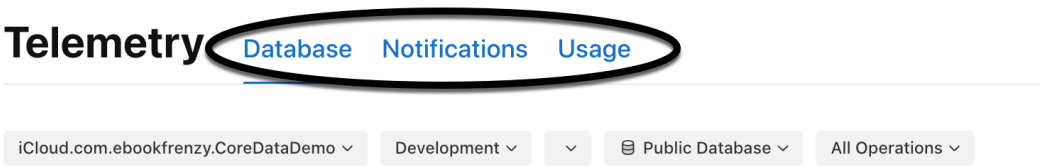


Figure 51-19

51.10 Summary

The first step in adding CloudKit support to an Xcode SwiftUI project is to add the iCloud capability, enabling both the CloudKit service and remote notifications, and configuring a container to store the databases associated with the app. The migration from Core Data to CloudKit is simply a matter of changing the code to use NSPersistentCloudKitContainer instead of NSPersistentContainer and re-building the project.

CloudKit databases can be queried, modified, managed, and monitored from within the CloudKit Console.

Chapter 59

59. An Overview of Live Activities in SwiftUI

The previous chapters introduced WidgetKit and demonstrated how it can be used to display widgets that provide information to the user on the home screen, lock screen, and Today view. Widgets of this type present information based on a timeline you create and pass to WidgetKit. In this chapter, we will introduce ActivityKit and Live Activities and explore how these can be used to present dynamic information to the user via widgets on the lock screen and Dynamic Island.

59.1 Introducing Live Activities

Live Activities are created using the ActivityKit and WidgetKit frameworks and present dynamic information in glanceable form without restricting updates to a predefined timeline.

A single app can have multiple Live Activities, and the information presented can be sourced locally within the app or delivered from a remote server via push notifications. One important caveat is that updates to the Live Activity will not necessarily occur in real-time. Both the local and remote push notification options use background modes of execution, the timing and frequency of which are dictated by the operating system based on various factors, including battery status, the resource-intensive nature of the update task, and user behavior patterns. We will cover this in more detail in the next chapter.

In addition to displaying information, Live Activities may contain Button and Toggle views to add interactive behavior.

59.2 Creating a Live Activity

Once a Widget Extension has been added to an Xcode app project, the process of creating a Live Activity can be separated into the following steps, each of which will be covered in this chapter and put to practical use in the next chapter:

- Declare static and dynamic Activity Attributes.
- Design the Live Activity presentations for the lock screen and Dynamic Island.
- Configure and start the Live Activity.
- Update the Live Activity with the latest information.
- End the Live Activity when updates are no longer required.

59.3 Live Activity Attributes

The purpose of Live Activities is to present information to the user when the corresponding app has been placed in the background. The Live Activity attributes declare the data structure to be presented and are created using ActivityKit's ActivityAttributes class. Two types of attributes can be included. The first type declares the data that will change over the lifecycle of the Live Activity, such as the latest scores of a live sporting event or an estimated flight arrival time. The second attribute type declares values that will remain static while the Live Activity executes, such as the name of the sports teams or the airline and flight number of a tracked flight.

Within the ActivityAttributes declaration, the dynamic attributes are embedded in a ContentState structure using the following syntax:

```
struct DemoWidgetAttributes: ActivityAttributes {
    public struct ContentState: Codable, Hashable {
        // dynamic attributes here
        var arrivalTime: Date
    }

    // static attributes here
    var airlineName: String = "Pending"
    var flightNumber: String = "Pending"
}
```

59.4 Designing the Live Activity Presentations

Live Activities present data to the user via lock screen, Dynamic Island, and banner widgets, each of which must be designed to complete the Live Activity. These presentations are created using SwiftUI views. While the lock screen presentation (also used for the banner widget) consists of a single layout, the Dynamic Island presentations are separated into regions.

The layouts for the Live Activity widgets are defined in a configuration structure subclassed from the WidgetKit framework's Widget class and must conform to the following syntax:

```
struct DemoWidgetLiveActivity: Widget {
    var body: some WidgetConfiguration {
        ActivityConfiguration(for: DemoWidgetAttributes.self) { context in

        } dynamicIsland: { context in

            DynamicIsland {

                DynamicIslandExpandedRegion(.leading) {

                }
                DynamicIslandExpandedRegion(.trailing) {

                }
                DynamicIslandExpandedRegion(.bottom) {

                }
                DynamicIslandExpandedRegion(.center) {

                }
            } compactLeading: {

            } compactTrailing: {

            } minimal: {
```

```
            }
        }
    }
}
```

Each element is passed a context object from which static and current dynamic data values can be accessed for inclusion in the presentation views. For example, the arrival time and flight number from the previous activity attributes declaration could be displayed by the widget as follows:

```
Text("Arrival: \(context.state.arrivalTime)")
Text("Flight: \(context.attributes.flightNumber)")
```

59.4.1 Lock Screen/Banner

Starting at the top of the Widget declaration, the layout for the lock screen and banner presentation consists of an area the size of a typical lock screen notification. The following example will display two Text views in a VStack layout:

```
struct DemoWidgetLiveActivity: Widget {
    var body: some WidgetConfiguration {
        ActivityConfiguration(for: DemoWidgetAttributes.self) { context in
            VStack {
                Text("Arrival: \(context.state.arrivalTime)")
                Text("Flight: \(context.attributes.flightNumber)")

            }
        } dynamicIsland: { context in
.
.
```

59.4.2 Dynamic Island Expanded Regions

The Live Activity will display data using compact layouts on devices with a Dynamic Island. However, a long press performed on the island will display the expanded widget. Unlike the lock screen widget, the expanded Dynamic Island presentation is divided into four regions, as illustrated in Figure 59-1:

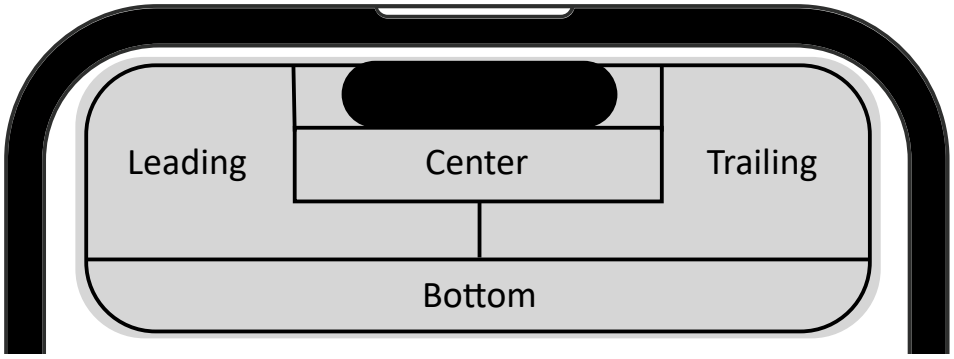


Figure 59-1

The following example highlights the code locations for each Dynamic Island region:

```
.
.
```

```
} dynamicIsland: { context in

    DynamicIsland {

        DynamicIslandExpandedRegion(.leading) {
            Text("Leading")
        }
        DynamicIslandExpandedRegion(.trailing) {
            Text("Trailing")
        }
        DynamicIslandExpandedRegion(.bottom) {
            Text("Bottom")
        }
        DynamicIslandExpandedRegion(.center) {
            Text("Center")
        }
    } compactLeading: {
```

.

.

The default sizing behavior of each region can be changed using priorities. In the following code, for example, the leading and trailing region sizes are set to 25% and 75% of the available presentation width, respectively:

```
DynamicIslandExpandedRegion(.leading, priority: 0.25) {
    Text("Leading")
}
DynamicIslandExpandedRegion(.trailing, priority: 0.75) {
    Text("Trailing")
}
```

59.4.3 Dynamic Island Compact Regions

The compact presentation is divided into regions located on either side of the camera, as illustrated in Figure 59-2:

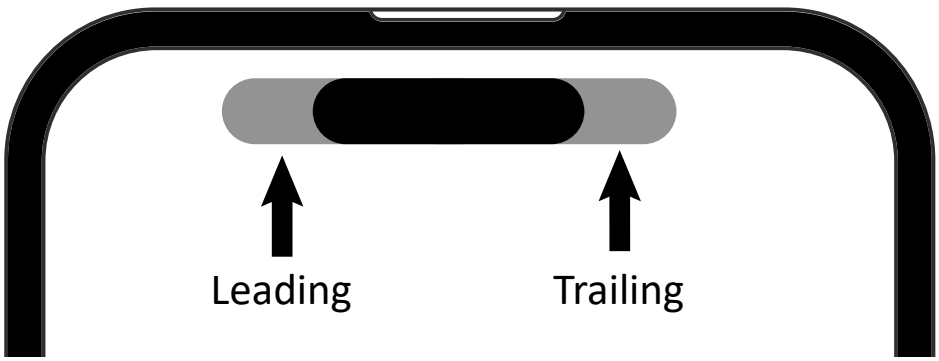


Figure 59-2

An example compact declaration might read as follows:

.

```
.
} compactLeading: {
    Text("L")
} compactTrailing: {
    Text("T")
} minimal: {
.
.
```

59.4.4 Dynamic Island Minimal

The Live Activity uses minimal presentations when multiple Live Activities are running concurrently. In this situation, the minimal presentation for one Live Activity will appear in the compact leading region (referred to as the *attached minimal*), while another appears as a d*etached minimal* positioned to the right of the camera:

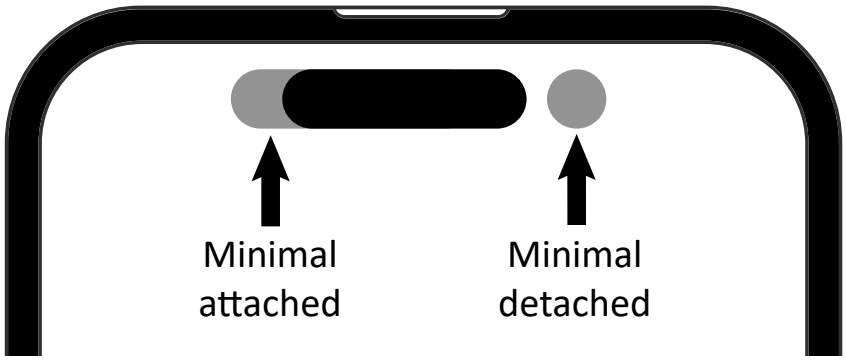


Figure 59-3

For example:

```
.
.
} minimal: {
    Text("M")
}
.
.
```

59.5 Starting a Live Activity

Once the data model has been defined and the presentations designed, the next step is to request and start the Live Activity. This is achieved by a call to the *Activity.request()* method. When the request method is called, an activity attributes instance, an initialized ContentState, and a push type must be provided. The push type should be set to *token* if the data updates will be received via push notifications or *nil* if updates are coming from the app.

An optional *stale date* may also be included. When the stale date is reached, the state of the Live Activity context will update to reflect that the information is out of date, allowing you to notify the user within the widget presentation. To check if the Live Activity is out of date, access the context's *isStale* property. The following code, for example, displays a message in the Dynamic Island expanded presentation when the data needs to be refreshed:

```
DynamicIslandExpandedRegion(.leading) {
```

```
    VStack {
        Text("Arrival: \(context.state.arrivalTime)")
        Text("Flight: \(context.attributes.flightNumber)")

        if (context.isStale) {
            Text("Out of date")
        }
    }
}
```

Set the *staleDate* parameter to *nil* if you do not plan to check the Live Activity status for this property.

Based on the above requirements, the first step is to create an activity attributes object and initialize any static properties, for example:

```
var attributes = DemoWidgetAttributes()
attributes.flightNumber = "Loading..."
```

The second requirement is a ContentState instance configured with initial dynamic values:

```
let contentState = DemoWidgetAttributes.ContentState(arrivalTime: Date.now + 60)
```

With the requirements met, the *Activity.request()* method can be called as follows:

```
private var activity: Activity<DemoWidgetAttributes>?
.
.
   do {
       activity = try Activity.request(
           attributes: attributes,
           content: .init(state: contentState, staleDate: nil),
           pushType: nil
       )
   } catch (let error) {
        print("Error requesting live activity: \(error.localizedDescription).")
   }
}
```

If the request is successful, the Live Activity will launch and be ready to receive updates. In the above example, the push type has been set to nil to indicate the data is generated within the app. This would need to be changed to *token* to support updates using push notifications.

59.6 Updating a Live Activity

To refresh a Live Activity with updated data, a call is made to the *update()* method of the activity instance returned by the earlier call to the *Activity.request()* method. The update call must be passed an ActivityContent instance containing a ContentState initialized with the updated dynamic data values and an optional stale date value. For example:

```
let flightState = DemoWidgetAttributes.ContentState(arrivalTime: newTime)

Task {

    await activity?.update(
```

```
        ActivityContent<DemoWidgetAttributes.ContentState>(
            state: flightState,
            staleDate: Date.now + 120,
            relevanceScore: 0
        ),
        alertConfiguration: nil
    )
}
```

If your app starts multiple concurrent Live Activities, the system will display the one with the highest relevanceScore. When working with push notifications, the content state is updated automatically, and the update call is unnecessary.

59.7 Activity Alert Configurations

Alert configurations are passed to the *update()* method to notify the user of significant events in the Live Activity data. When an alert is triggered, a banner (based on the lock screen presentation layout) appears on the device screen, accompanied by an optional alert sound. The following code example creates an alert configuration when a tracked flight has been significantly delayed:

```
var alertConfig: AlertConfiguration? = nil

if (arrvialTime > Date.now + 84000) {
    alertConfig = AlertConfiguration(
        title: "Flight Delay",
        body: "Flight now arriving tomorrow",
        sound: .default
    )
}
```

Note that the title and body text will only appear on Apple Watch devices.

Once an alert configuration has been created, it can be passed to the *update()* method:

```
await activity?.update(
    ActivityContent<DemoWidgetAttributes.ContentState>(
        state: flightState,
        staleDate: Date.now + 120,
        relevanceScore: 0
    ),
    alertConfiguration: alertConfig
)
```

59.8 Stopping a Live Activity

Live Activities are stopped by calling the *end()* method of the activity instance. The call is passed a ContentState instance initialized with the final data values and a dismissal policy setting. For example:

```
let finalState = DemoWidgetAttributes.ContentState(arrivalTime: Date.now)

await activity?.end(
     .init(state: finalState, staleDate: nil),
     dismissalPolicy: .default
```

```
)
```

When the dismissalPolicy is set to *default*, the Live Activity widget will remain on the lock screen for four hours unless the user removes it. Use *immediate* to instantly remove the Live Activity from the lock screen or *after()* to dismiss the Live Activity at a specific time within the four-hour window.

59.9 Summary

Live Activities provide users with timely updates via widgets on the device lock screen and Dynamic Island. Updated information can be generated locally within the app or sent from a remote server using push notifications. A Live Activity consists of a set of attributes that define the data to be presented and SwiftUI-based layouts for each of the widget presentations. Live Activity instances are started, stopped, and updated using calls to the corresponding Activity object. When working with push notifications, the activity will update automatically on receipt of a notification. Updates may also include an optional alert to attract the user's attention.

Chapter 63

63. Testing Live Activity Push Notifications

The previous chapter explained how to add support for push notifications to a Live Activity. Once enabled for push notifications, the Live Activity is ready for testing.

Test push notifications can be sent from the CloudKit console or the command line using the curl tool. This chapter will demonstrate both options, including generating the authentication key required for sending notifications from the command line and declaring a notification payload.

In the next chapter, *"Troubleshooting Live Activity Push Notifications"*, we will outline techniques for identifying and resolving push notification problems.

63.1 Using the Push Notifications Console

The best way to begin push notification testing is using the CloudKit console. The console provides an easy way to send push notifications and, more importantly, identify why the Live Activity may not have received a notification on the user's device.

To access the CloudKit console, open a browser window, navigate to *https://icloud.developer.apple.com/*, and sign in using your Apple Developer credentials. Once you have signed in, select the Push Notifications option in the dashboard, as highlighted in Figure 63-1:

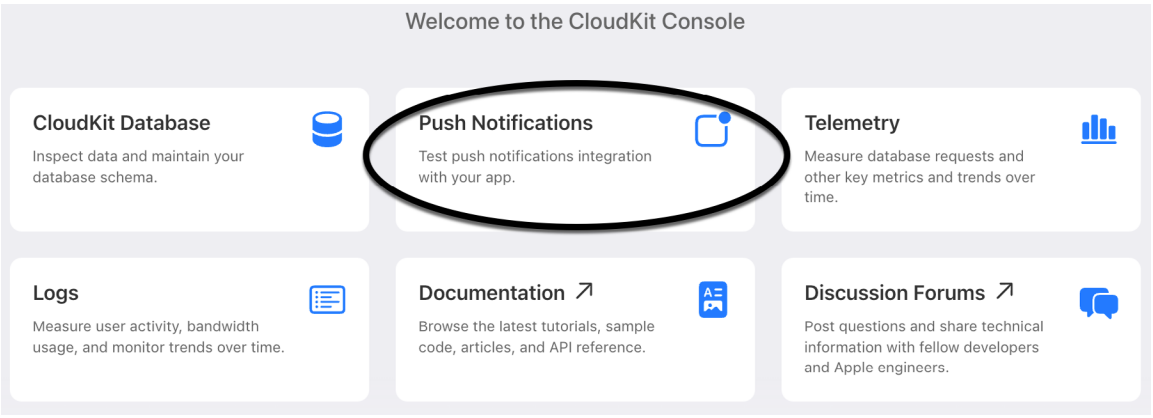


Figure 63-1

Use the drop-down menu (marked A in Figure 63-2) to select the LiveActivityDemo project, then click the *Enable Push Notifications* button (B):

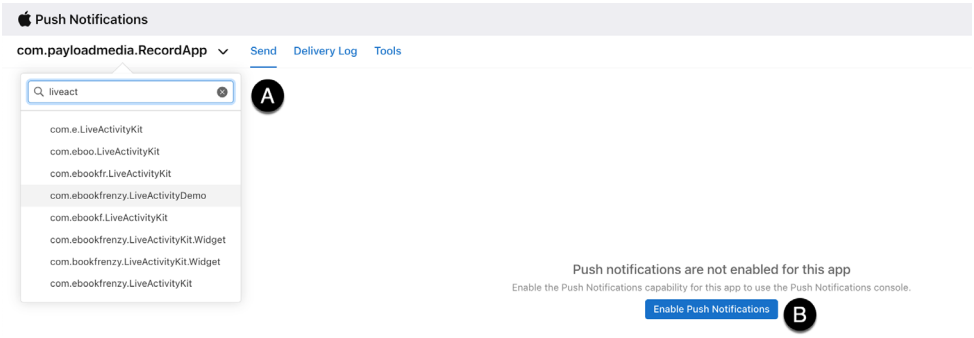


Figure 63-2

Once notifications have been enabled, click on the *Create New Notification* button:

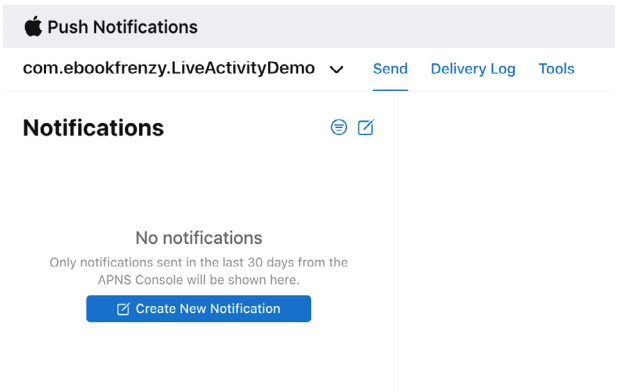


Figure 63-3

The New Notification screen will appear ready for the notification details to be entered.

63.2 Configuring the Notification

In the General section of the New Notification screen, enter a name for the test and set the Environment menu to Development (the Production setting is for when the app has been published on the App Store). Next, return to Xcode, run the app on a device and start tracking, copy the push token from the Xcode console and paste it into the Device Token field in the CloudKit console:

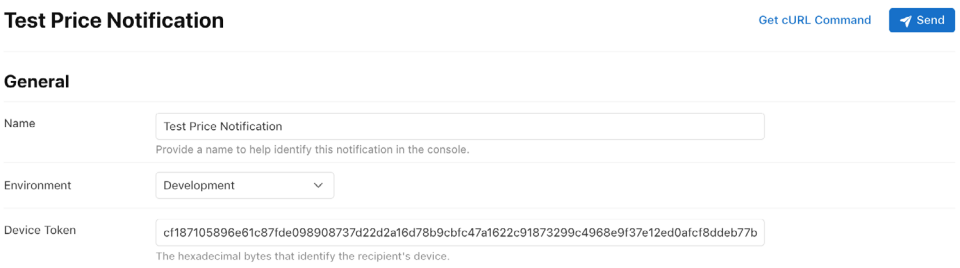


Figure 63-4

In the Request Headers section, the *apns-topic* field is read-only and will already contain your app Bundle Identifier. Select the *apns-push-type* menu and change the selection to *liveactivity*. The *apns-expiration* setting

can specify a date and time when the APNs service should stop trying to send the notification. The default setting will only make one push attempt. The single delivery attempt option is actually more robust than the name suggests, and this setting is adequate for most requirements.

The *apns-priority* value can be set to 1 (low), 5 (medium), or 10 (high). Use low priority for non-time-sensitive updates and high priority for critical alerts. For most uses, however, medium is the recommended priority:

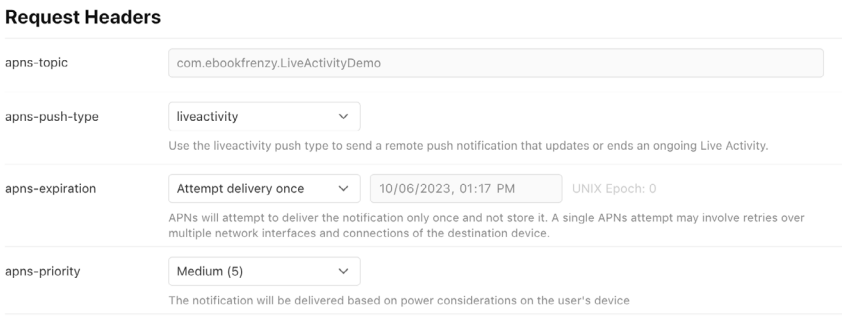


Figure 63-5

The final section will contain the notification payload, which requires additional explanation.

63.3 Defining the Payload

The notification payload is declared using JSON and must contain the following information:

- **timestamp** - The timestamp ensures that the Live Activity is updated only with the most up-to-date push notification. Notifications containing a timestamp identical to or earlier than the previous notification are discarded. For this reason, you must provide a new timestamp each time you send a push notification. The timestamp is calculated as the number of elapsed seconds since January 1, 1970, and can be obtained using the following online Epoch calculator:

https://www.epochconverter.com/

Alternatively, open a Terminal window on your Mac and run the following command:

```
date +%s
```

- **event** - This value specifies the action associated with the push notification and should be set to "update".

- **content-state** - The content state defines the updated data to be displayed by the Live Activity. It must match exactly the dynamic variable names and data types declared in the Live Activity widget attributes structure.

Return to the CloudKit console and enter the following JSON declaration into the Payload section of the New Notification form, where <*recent timestamp*> is replaced with the current value:

```
{
    "aps":{
        "timestamp":<recent time stamp>
        "event":"update"
        "content-state":{
            "currentPrice": 310,
            "changePercent": 37
        }
    }
```

```
}
```

63.4 Sending the Notification

Before sending the notification, ensure the app is still running on your device and check the Xcode console to confirm that the push token is unchanged from when it was pasted into the notification form. Place the app into the background, return to the CloudKit console, and click the Send button. After a short delay, the console will report a problem with the entered information or attempt to send the notification. Check the Live Activity widget on your device to see if the price information has updated to the values contained in the payload. If nothing happens, it is time to troubleshoot the notification using the steps outlined in the next chapter.

63.5 Sending Push Notifications from the Command Line

Another way to test push notifications is from the command line of a Terminal window on your Mac using the *curl* command. This technique has the advantage that it can be used to automate sending multiple notifications without having to create each one in the CloudKit console manually. It also allows us to generate the timestamp dynamically.

Behind the scenes, the CloudKit console automatically generated an authentication token for us that is required to send push notifications. To generate this for the command line, you will need the Key ID and the key file saved in the *"Testing Live Activity Push Notifications"* chapter. You will also need your Apple Developer Team ID, which can be found by selecting the Membership details option in the Apple Developer console. You will also need to specify the Bundle ID of your app (known as the *topic* in this context).

Once this information has been gathered, open a Terminal window, change directory to a suitable location, create a new file named *push.sh,* and edit it as follows:

```
#!/bin/bash
TEAM_ID="<Your Team ID here>"
TOKEN_KEY_FILE_NAME="<Path to your P8 key file here>"
AUTH_KEY_ID="<Your Key ID here>"
TOPIC="<Your app Bundle ID here>"
APNS_HOST_NAME=api.sandbox.push.apple.com

JWT_ISSUE_TIME=$(date +%s)
JWT_HEADER=$(printf '{ "alg": "ES256", "kid": "%s" }' "${AUTH_KEY_ID}" | openssl
base64 -e -A | tr -- '+/' '-_' | tr -d =)
JWT_CLAIMS=$(printf '{ "iss": "%s", "iat": %d }' "${TEAM_ID}" "${JWT_ISSUE_TIME}"
| openssl base64 -e -A | tr -- '+/' '-_' | tr -d =)
JWT_HEADER_CLAIMS="${JWT_HEADER}.${JWT_CLAIMS}"
JWT_SIGNED_HEADER_CLAIMS=$(printf "${JWT_HEADER_CLAIMS}" | openssl dgst -binary
-sha256 -sign "${TOKEN_KEY_FILE_NAME}" | openssl base64 -e -A | tr -- '+/' '-_' |
tr -d =)
AUTHENTICATION_TOKEN="${JWT_HEADER}.${JWT_CLAIMS}.${JWT_SIGNED_HEADER_CLAIMS}"
echo $AUTHENTICATION_TOKEN
```

Save the file and run it using the following command:

```
sh ./push.sh
```

On successful execution, the script should print the authentication token.

Use Xcode to launch the LiveActivityDemo app on a device and copy the latest push token from the console. Edit the *push.sh* script file and add the token as follows:

```
#!/bin/bash
ACTIVITY_PUSH_TOKEN="<Your push token here>"
.
.
```

Finally, the *curl* command can be added to the script. This consists of the authentication and push tokens and the push type, topic, priority, and expiration settings. The command must also include the notification payload with a current timestamp using the same JSON syntax used in the CloudKit console. With these requirements in mind, add the following lines to the end of the *push.sh* file:

```
curl -v \
  --header "authorization: bearer ${AUTHENTICATION_TOKEN}" \
  --header "apns-topic: <your bundle id here>.push-type.liveactivity" \
  --header "apns-push-type: liveactivity" \
  --header "apns-priority: 10" \
  --header "apns-expiration: 0" \
  --data '{"aps":{"timestamp":'$(date +%s)',"event":"update","content-state":{"cu
rrentPrice":500,"changePercent":50}}}' \
  --http2  https://api.development.push.apple.com:443/3/device/$ACTIVITY_PUSH_
TOKEN
```

Note that the topic header consists of your bundle ID followed by *.push-type.liveactivity* and that we are using the *date +%s* command to create the timestamp.

Check the push token is still valid, execute the push script, and check the output for errors. If the APNs accepted the notification, the output will end as follows:

```
> apns-expiration: 0
> Content-Length: 105
> Content-Type: application/x-www-form-urlencoded
>
* We are completely uploaded and fine
< HTTP/2 200
< apns-id: 92F32B4C-9527-0CAD-32FA-AC0B4A9200B1
< apns-unique-id: 4090d3d1-b615-250a-79e5-d39e3801b542
<
* Connection #0 to host api.development.push.apple.com left intact
```

If the notification does not update the Live Activity widgets, record the *apns-unique-id* in the curl output and use it to diagnose the problem using the steps in the *"Troubleshooting Live Activity Push Notifications"* chapter.

63.6 Summary

Test push notifications can be sent to a Live Activity using either the CloudKit console or from the command line using the *curl* command. For both options, the notification must include the push token from the device and a JSON payload containing the updated Live Activity content state. An additional authentication token is required when testing is performed using the command line. The token is generated using the Key ID and file created in the previous chapter.

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z