

# **iOS 17 App Development Essentials**

---

iOS 17 App Development Essentials

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

© 2023 Neil Smyth / Payload Media, Inc. All Rights Reserved.

This book is provided for personal use only. Unauthorized use, reproduction and/or distribution strictly prohibited. All rights reserved.

The content of this book is provided for informational purposes only. Neither the publisher nor the author offers any warranties or representation, express or implied, with regard to the accuracy of information contained in this book, nor do they accept any liability for any loss or damage arising from any errors or omissions.

This book contains trademarked terms that are used solely for editorial purposes and to the benefit of the respective trademark owner. The terms used within this book are not intended as infringement of any trademarks.

Rev: 1.0

## Table of Contents

<b>1. Start Here.....</b>	<b>1</b>
1.1 For Swift Programmers.....	1
1.2 For Non-Swift Programmers .....	1
1.3 Source Code Download.....	2
1.4 Feedback.....	2
1.5 Errata.....	2
<b>2. Joining the Apple Developer Program.....</b>	<b>3</b>
2.1 Downloading Xcode 15 and the iOS 17 SDK .....	3
2.2 Apple Developer Program.....	3
2.3 When to Enroll in the Apple Developer Program?.....	3
2.4 Enrolling in the Apple Developer Program .....	4
2.5 Summary .....	5
<b>3. Installing Xcode 15 and the iOS 17 SDK .....</b>	<b>7</b>
3.1 Identifying Your macOS Version .....	7
3.2 Installing Xcode 15 and the iOS 17 SDK.....	7
3.3 Starting Xcode .....	8
3.4 Adding Your Apple ID to the Xcode Preferences.....	8
3.5 Developer and Distribution Signing Identities .....	9
3.6 Summary .....	9
<b>4. An Introduction to Xcode 15 Playgrounds.....</b>	<b>11</b>
4.1 What is a Playground? .....	11
4.2 Creating a New Playground .....	11
4.3 A Swift Playground Example .....	12
4.4 Viewing Results .....	14
4.5 Adding Rich Text Comments .....	16
4.6 Working with Playground Pages .....	17
4.7 Working with SwiftUI and Live View in Playgrounds .....	17
4.8 Summary .....	20
<b>5. Swift Data Types, Constants, and Variables .....</b>	<b>21</b>
5.1 Using a Swift Playground .....	21
5.2 Swift Data Types .....	21
5.2.1 Integer Data Types .....	22
5.2.2 Floating Point Data Types .....	22
5.2.3 Bool Data Type .....	23
5.2.4 Character Data Type.....	23
5.2.5 String Data Type.....	23
5.2.6 Special Characters/Escape Sequences .....	24
5.3 Swift Variables.....	25
5.4 Swift Constants .....	25
5.5 Declaring Constants and Variables .....	25
5.6 Type Annotations and Type Inference .....	25

## Table of Contents

5.7 The Swift Tuple .....	26
5.8 The Swift Optional Type.....	27
5.9 Type Casting and Type Checking.....	30
5.10 Summary .....	32
<b>6. Swift Operators and Expressions .....</b>	<b>33</b>
6.1 Expression Syntax in Swift .....	33
6.2 The Basic Assignment Operator.....	33
6.3 Swift Arithmetic Operators.....	33
6.4 Compound Assignment Operators.....	34
6.5 Comparison Operators.....	34
6.6 Boolean Logical Operators.....	35
6.7 Range Operators.....	35
6.8 The Ternary Operator .....	36
6.9 Nil Coalescing Operator.....	36
6.10 Bitwise Operators .....	37
6.10.1 Bitwise NOT .....	37
6.10.2 Bitwise AND .....	37
6.10.3 Bitwise OR.....	38
6.10.4 Bitwise XOR.....	38
6.10.5 Bitwise Left Shift.....	38
6.10.6 Bitwise Right Shift.....	39
6.11 Compound Bitwise Operators.....	39
6.12 Summary .....	40
<b>7. Swift Control Flow.....</b>	<b>41</b>
7.1 Looping Control Flow .....	41
7.2 The Swift for-in Statement.....	41
7.2.1 The while Loop .....	42
7.3 The repeat ... while loop .....	42
7.4 Breaking from Loops .....	43
7.5 The continue Statement .....	43
7.6 Conditional Control Flow .....	44
7.7 Using the if Statement .....	44
7.8 Using if ... else ... Statements .....	44
7.9 Using if ... else if ... Statements .....	45
7.10 The guard Statement .....	45
7.11 Summary .....	46
<b>8. The Swift Switch Statement .....</b>	<b>47</b>
8.1 Why Use a switch Statement? .....	47
8.2 Using the switch Statement Syntax .....	47
8.3 A Swift switch Statement Example .....	47
8.4 Combining case Statements .....	48
8.5 Range Matching in a switch Statement.....	49
8.6 Using the where statement.....	49
8.7 Fallthrough.....	50
8.8 Summary .....	50
<b>9. Swift Functions, Methods, and Closures.....</b>	<b>51</b>

9.1 What is a Function? .....	51
9.2 What is a Method? .....	51
9.3 How to Declare a Swift Function .....	51
9.4 Implicit Returns from Single Expressions.....	52
9.5 Calling a Swift Function .....	52
9.6 Handling Return Values .....	52
9.7 Local and External Parameter Names .....	53
9.8 Declaring Default Function Parameters.....	53
9.9 Returning Multiple Results from a Function.....	54
9.10 Variable Numbers of Function Parameters .....	54
9.11 Parameters as Variables .....	55
9.12 Working with In-Out Parameters .....	55
9.13 Functions as Parameters.....	56
9.14 Closure Expressions.....	58
9.15 Shorthand Argument Names.....	59
9.16 Closures in Swift.....	59
9.17 Summary .....	60
<b>10. The Basics of Swift Object-Oriented Programming .....</b>	<b>61</b>
10.1 What is an Instance? .....	61
10.2 What is a Class? .....	61
10.3 Declaring a Swift Class .....	61
10.4 Adding Instance Properties to a Class.....	62
10.5 Defining Methods .....	62
10.6 Declaring and Initializing a Class Instance.....	63
10.7 Initializing and De-initializing a Class Instance .....	63
10.8 Calling Methods and Accessing Properties .....	64
10.9 Stored and Computed Properties.....	65
10.10 Lazy Stored Properties.....	66
10.11 Using self in Swift.....	67
10.12 Understanding Swift Protocols.....	68
10.13 Opaque Return Types.....	69
10.14 Summary .....	70
<b>11. An Introduction to Swift Subclassing and Extensions .....</b>	<b>71</b>
11.1 Inheritance, Classes, and Subclasses.....	71
11.2 A Swift Inheritance Example .....	71
11.3 Extending the Functionality of a Subclass .....	72
11.4 Overriding Inherited Methods.....	72
11.5 Initializing the Subclass.....	73
11.6 Using the SavingsAccount Class .....	74
11.7 Swift Class Extensions .....	74
11.8 Summary .....	75
<b>12. An Introduction to Swift Structures and Enumerations .....</b>	<b>77</b>
12.1 An Overview of Swift Structures.....	77
12.2 Value Types vs. Reference Types .....	78
12.3 When to Use Structures or Classes .....	80
12.4 An Overview of Enumerations.....	80

## Table of Contents

12.5 Summary .....	81
<b>13. An Introduction to Swift Property Wrappers.....</b>	<b>83</b>
13.1 Understanding Property Wrappers.....	83
13.2 A Simple Property Wrapper Example .....	83
13.3 Supporting Multiple Variables and Types .....	85
13.4 Summary .....	87
<b>14. Working with Array and Dictionary Collections in Swift.....</b>	<b>89</b>
14.1 Mutable and Immutable Collections .....	89
14.2 Swift Array Initialization.....	89
14.3 Working with Arrays in Swift .....	90
14.3.1 Array Item Count .....	90
14.3.2 Accessing Array Items .....	90
14.3.3 Random Items and Shuffling .....	90
14.3.4 Appending Items to an Array .....	91
14.3.5 Inserting and Deleting Array Items .....	91
14.3.6 Array Iteration .....	91
14.4 Creating Mixed Type Arrays.....	92
14.5 Swift Dictionary Collections.....	92
14.6 Swift Dictionary Initialization .....	92
14.7 Sequence-based Dictionary Initialization.....	93
14.8 Dictionary Item Count.....	94
14.9 Accessing and Updating Dictionary Items .....	94
14.10 Adding and Removing Dictionary Entries .....	94
14.11 Dictionary Iteration .....	94
14.12 Summary.....	95
<b>15. Understanding Error Handling in Swift 5 .....</b>	<b>97</b>
15.1 Understanding Error Handling .....	97
15.2 Declaring Error Types .....	97
15.3 Throwing an Error .....	98
15.4 Calling Throwing Methods and Functions.....	98
15.5 Accessing the Error Object .....	100
15.6 Disabling Error Catching .....	100
15.7 Using the defer Statement .....	100
15.8 Summary .....	101
<b>16. An Overview of SwiftUI .....</b>	<b>103</b>
16.1 UIKit and Interface Builder .....	103
16.2 SwiftUI Declarative Syntax .....	103
16.3 SwiftUI is Data Driven .....	104
16.4 SwiftUI vs. UIKit .....	104
16.5 Summary .....	105
<b>17. Using Xcode in SwiftUI Mode .....</b>	<b>107</b>
17.1 Starting Xcode 15 .....	107
17.2 Creating a SwiftUI Project .....	107
17.3 Xcode in SwiftUI Mode.....	109
17.4 The Preview Canvas.....	111

17.5 Preview Pinning .....	112
17.6 The Preview Toolbar .....	113
17.7 Modifying the Design .....	114
17.8 Editor Context Menu .....	116
17.9 Running the App on a Simulator .....	117
17.10 Running the App on a Physical iOS Device .....	117
17.11 Managing Devices and Simulators .....	118
17.12 Enabling Network Testing .....	119
17.13 Dealing with Build Errors .....	119
17.14 Monitoring Application Performance .....	119
17.15 Exploring the User Interface Layout Hierarchy .....	120
17.16 Summary .....	122
<b>18. SwiftUI Architecture .....</b>	<b>123</b>
18.1 SwiftUI App Hierarchy .....	123
18.2 App .....	123
18.3 Scenes .....	123
18.4 Views .....	124
18.5 Summary .....	124
<b>19. The Anatomy of a Basic SwiftUI Project .....</b>	<b>125</b>
19.1 Creating an Example Project .....	125
19.2 The DemoProjectApp.swift File .....	125
19.3 The ContentView.swift File .....	126
19.4 Assets.xcassets .....	126
19.5 DemoProject.entitlements .....	126
19.6 Preview Content .....	127
19.7 Summary .....	127
<b>20. Creating Custom Views with SwiftUI .....</b>	<b>129</b>
20.1 SwiftUI Views .....	129
20.2 Creating a Basic View .....	129
20.3 Adding Views .....	130
20.4 SwiftUI Hierarchies .....	131
20.5 Concatenating Text Views .....	132
20.6 Working with Subviews .....	132
20.7 Views as Properties .....	133
20.8 Modifying Views .....	133
20.9 Working with Text Styles .....	134
20.10 Modifier Ordering .....	135
20.11 Custom Modifiers .....	136
20.12 Basic Event Handling .....	137
20.13 Building Custom Container Views .....	137
20.14 Working with the Label View .....	139
20.15 Summary .....	140
<b>21. SwiftUI Stacks and Frames .....</b>	<b>141</b>
21.1 SwiftUI Stacks .....	141
21.2 Spacers, Alignment and Padding .....	143
21.3 Grouping Views .....	145

## Table of Contents

21.4 Dynamic HStack and VStack Conversion .....	145
21.5 Text Line Limits and Layout Priority.....	146
21.6 Traditional vs. Lazy Stacks .....	148
21.7 SwiftUI Frames .....	148
21.8 Frames and the Geometry Reader .....	150
21.9 Summary .....	150
<b>22. SwiftUI State Properties, Observation, and Environment Objects.....</b>	<b>153</b>
22.1 State Properties.....	153
22.2 State Binding.....	155
22.3 Observable Objects .....	155
22.4 Observation using Combine.....	156
22.5 Combine State Objects .....	157
22.6 Using the Observation Framework.....	158
22.7 Observation and @Bindable .....	158
22.8 Environment Objects.....	159
22.9 Summary .....	161
<b>23. A SwiftUI Example Tutorial .....</b>	<b>163</b>
23.1 Creating the Example Project.....	163
23.2 Reviewing the Project .....	163
23.3 Modifying the Layout .....	165
23.4 Adding a Slider View to the Stack.....	166
23.5 Adding a State Property .....	166
23.6 Adding Modifiers to the Text View.....	167
23.7 Adding Rotation and Animation .....	168
23.8 Adding a TextField to the Stack.....	169
23.9 Adding a Color Picker .....	170
23.10 Tidying the Layout.....	171
23.11 Summary.....	174
<b>24. An Overview of Swift Structured Concurrency.....</b>	<b>175</b>
24.1 An Overview of Threads .....	175
24.2 The Application Main Thread.....	175
24.3 Completion Handlers .....	175
24.4 Structured Concurrency.....	176
24.5 Preparing the Project.....	176
24.6 Non-Concurrent Code .....	176
24.7 Introducing async/await Concurrency.....	177
24.8 Asynchronous Calls from Synchronous Functions .....	178
24.9 The await Keyword.....	178
24.10 Using async-let Bindings.....	179
24.11 Handling Errors.....	180
24.12 Understanding Tasks .....	181
24.13 Unstructured Concurrency .....	181
24.14 Detached Tasks.....	182
24.15 Task Management .....	183
24.16 Working with Task Groups .....	183
24.17 Avoiding Data Races.....	184



24.18 The for-await Loop.....	185
24.19 Asynchronous Properties.....	186
24.20 Summary .....	187
<b>25. An Introduction to Swift Actors.....</b>	<b>189</b>
25.1 An Overview of Actors.....	189
25.2 Declaring an Actor.....	189
25.3 Understanding Data Isolation .....	190
25.4 A Swift Actor Example .....	191
25.5 Introducing the MainActor.....	192
25.6 Summary .....	193
<b>26. SwiftUI Concurrency and Lifecycle Event Modifiers .....</b>	<b>195</b>
26.1 Creating the LifecycleDemo Project.....	195
26.2 Designing the App .....	195
26.3 The onAppear and onDisappear Modifiers .....	196
26.4 The onChange Modifier .....	197
26.5 ScenePhase and the onChange Modifier.....	197
26.6 Launching Concurrent Tasks.....	199
26.7 Summary .....	200
<b>27. SwiftUI Observable and Environment Objects – A Tutorial.....</b>	<b>201</b>
27.1 About the ObservableDemo Project.....	201
27.2 Creating the Project .....	201
27.3 Adding the Observable Object .....	201
27.4 Designing the ContentView Layout.....	202
27.5 Adding the Second View .....	203
27.6 Adding Navigation .....	204
27.7 Using an Environment Object.....	205
27.8 Summary .....	206
<b>28. SwiftUI Data Persistence using AppStorage and SceneStorage.....</b>	<b>207</b>
28.1 The @SceneStorage Property Wrapper.....	207
28.2 The @AppStorage Property Wrapper .....	207
28.3 Creating and Preparing the StorageDemo Project .....	208
28.4 Using Scene Storage .....	209
28.5 Using App Storage.....	210
28.6 Storing Custom Types.....	211
28.7 Summary .....	213
<b>29. SwiftUI Stack Alignment and Alignment Guides.....</b>	<b>215</b>
29.1 Container Alignment.....	215
29.2 Alignment Guides .....	217
29.3 Custom Alignment Types .....	220
29.4 Cross Stack Alignment .....	223
29.5 ZStack Custom Alignment.....	225
29.6 Summary .....	229
<b>30. SwiftUI Lists and Navigation .....</b>	<b>231</b>
30.1 SwiftUI Lists.....	231

## Table of Contents

30.2 Modifying List Separators and Rows .....	232
30.3 SwiftUI Dynamic Lists.....	233
30.4 Creating a Refreshable List .....	235
30.5 SwiftUI NavigationStack and NavigationLink .....	236
30.6 Navigation by Value Type.....	238
30.7 Working with Navigation Paths .....	239
30.8 Navigation Bar Customization .....	239
30.9 Making the List Editable .....	240
30.10 Hierarchical Lists .....	242
30.11 Multicolumn Navigation.....	243
30.12 Summary.....	243
<b>31. A SwiftUI List and NavigationStack Tutorial .....</b>	<b>245</b>
31.1 About the ListNavDemo Project.....	245
31.2 Creating the ListNavDemo Project.....	245
31.3 Preparing the Project.....	245
31.4 Adding the Car Structure.....	246
31.5 Adding the Data Store .....	246
31.6 Designing the Content View.....	247
31.7 Designing the Detail View .....	248
31.8 Adding Navigation to the List .....	250
31.9 Designing the Add Car View.....	251
31.10 Implementing Add and Edit Buttons .....	253
31.11 Adding a Navigation Path.....	255
31.12 Adding the Edit Button Methods.....	256
31.13 Summary.....	257
<b>32. An Overview of Split View Navigation .....</b>	<b>259</b>
32.1 Introducing NavigationSplitView .....	259
32.2 Using NavigationSplitView .....	259
32.3 Handling List Selection .....	260
32.4 NavigationSplitView Configuration .....	260
32.5 Controlling Column Visibility.....	261
32.6 Summary .....	262
<b>33. A NavigationSplitView Tutorial.....</b>	<b>263</b>
33.1 About the Project .....	263
33.2 Creating the NavSplitDemo Project .....	263
33.3 Adding the Project Data.....	263
33.4 Creating the Navigation View .....	264
33.5 Building the Sidebar Column .....	264
33.6 Adding the Content Column List .....	265
33.7 Adding the Detail Column .....	266
33.8 Configuring the Split Navigation Experience.....	267
33.9 Summary .....	268
<b>34. An Overview of List, OutlineGroup and DisclosureGroup .....</b>	<b>269</b>
34.1 Hierarchical Data and Disclosures.....	269
34.2 Hierarchies and Disclosure in SwiftUI Lists.....	270
34.3 Using OutlineGroup .....	272

34.4 Using DisclosureGroup .....	273
34.5 Summary .....	275
<b>35. A SwiftUI List, OutlineGroup, and DisclosureGroup Tutorial.....</b>	<b>277</b>
35.1 About the Example Project .....	277
35.2 Creating the OutlineGroupDemo Project .....	277
35.3 Adding the Data Structure .....	277
35.4 Adding the List View .....	279
35.5 Testing the Project.....	280
35.6 Using the Sidebar List Style.....	280
35.7 Using OutlineGroup .....	281
35.8 Working with DisclosureGroups .....	282
35.9 Summary .....	286
<b>36. Building SwiftUI Grids with LazyVGrid and LazyHGrid .....</b>	<b>287</b>
36.1 SwiftUI Grids .....	287
36.2 GridItems .....	287
36.3 Creating the GridDemo Project .....	288
36.4 Working with Flexible GridItems .....	289
36.5 Adding Scrolling Support to a Grid.....	290
36.6 Working with Adaptive GridItems .....	292
36.7 Working with Fixed GridItems .....	293
36.8 Using the LazyHGrid View.....	295
36.9 Summary .....	297
<b>37. Building SwiftUI Grids with Grid and GridRow .....</b>	<b>299</b>
37.1 Grid and GridRow Views.....	299
37.2 Creating the GridRowDemo Project .....	299
37.3 A Simple Grid Layout .....	300
37.4 Non-GridRow Children .....	301
37.5 Automatic Empty Grid Cells .....	302
37.6 Adding Empty Cells.....	303
37.7 Column Spanning .....	304
37.8 Grid Alignment and Spacing.....	304
37.9 Summary .....	309
<b>38. Building Tabbed and Paged Views in SwiftUI .....</b>	<b>311</b>
38.1 An Overview of SwiftUI TabView.....	311
38.2 Creating the TabViewDemo App.....	312
38.3 Adding the TabView Container .....	312
38.4 Adding the Content Views.....	312
38.5 Adding View Paging .....	312
38.6 Adding the Tab Items.....	313
38.7 Adding Tab Item Tags.....	313
38.8 Summary .....	314
<b>39. Building Context Menus in SwiftUI.....</b>	<b>315</b>
39.1 Creating the ContextMenuDemo Project.....	315
39.2 Preparing the Content View .....	315
39.3 Adding the Context Menu .....	315

## Table of Contents

39.4 Testing the Context Menu.....	317
39.5 Summary .....	317
<b>40. Basic SwiftUI Graphics Drawing .....</b>	<b>319</b>
40.1 Creating the DrawDemo Project.....	319
40.2 SwiftUI Shapes.....	319
40.3 Using Overlays.....	321
40.4 Drawing Custom Paths and Shapes .....	322
40.5 Color Gradients and Shadows .....	324
40.6 Drawing Gradients.....	325
40.7 Summary .....	327
<b>41. SwiftUI Animation and Transitions.....</b>	<b>329</b>
41.1 Creating the AnimationDemo Example Project.....	329
41.2 Implicit Animation .....	329
41.3 Repeating an Animation .....	331
41.4 Explicit Animation.....	332
41.5 Animation and State Bindings.....	333
41.6 Automatically Starting an Animation .....	334
41.7 SwiftUI Transitions .....	336
41.8 Combining Transitions.....	337
41.9 Asymmetrical Transitions.....	338
41.10 Summary.....	338
<b>42. Working with Gesture Recognizers in SwiftUI.....</b>	<b>339</b>
42.1 Creating the GestureDemo Example Project .....	339
42.2 Basic Gestures .....	339
42.3 The onChange Action Callback.....	340
42.4 The updating Callback Action.....	342
42.5 Composing Gestures.....	343
42.6 Summary .....	345
<b>43. Creating a Customized SwiftUI ProgressView .....</b>	<b>347</b>
43.1 ProgressView Styles .....	347
43.2 Creating the ProgressViewDemo Project .....	348
43.3 Adding a ProgressView .....	348
43.4 Using the Circular ProgressView Style.....	348
43.5 Declaring an Indeterminate ProgressView .....	349
43.6 ProgressView Customization.....	349
43.7 Summary .....	352
<b>44. Presenting Data with SwiftUI Charts .....</b>	<b>353</b>
44.1 Introducing SwiftUI Charts.....	353
44.2 Passing Data to the Chart.....	354
44.3 Combining Mark Types.....	355
44.4 Filtering Data into Multiple Graphs .....	356
44.5 Changing the Chart Background .....	357
44.6 Changing the Interpolation Method .....	357
44.7 Summary .....	358
<b>45. A SwiftUI Charts Tutorial.....</b>	<b>359</b>

45.1 Creating the ChartDemo Project.....	359
45.2 Adding the Project Data.....	359
45.3 Adding the Chart View.....	360
45.4 Creating Multiple Graphs.....	361
45.5 Summary .....	362
<b>46. An Overview of SwiftUI DocumentGroup Scenes .....</b>	<b>363</b>
46.1 Documents in Apps .....	363
46.2 Creating the DocDemo App.....	363
46.3 The DocumentGroup Scene .....	364
46.4 Declaring File Type Support.....	365
46.4.1 Document Content Type Identifier .....	365
46.4.2 Handler Rank.....	365
46.4.3 Type Identifiers.....	365
46.4.4 Filename Extensions .....	365
46.4.5 Custom Type Document Content Identifiers.....	365
46.4.6 Exported vs. Imported Type Identifiers .....	366
46.5 Configuring File Type Support in Xcode.....	366
46.6 The Document Structure.....	367
46.7 The Content View.....	369
46.8 Adding Navigation .....	369
46.9 Running the Example App.....	370
46.10 Summary .....	371
<b>47. A SwiftUI DocumentGroup Tutorial .....</b>	<b>373</b>
47.1 Creating the ImageDocDemo Project.....	373
47.2 Modifying the Info.plist File.....	373
47.3 Adding an Image Asset.....	374
47.4 Modifying the ImageDocDemoDocument.swift File .....	374
47.5 Designing the Content View.....	375
47.6 Filtering the Image.....	377
47.7 Testing the App.....	378
47.8 Summary .....	378
<b>48. An Introduction to Core Data and SwiftUI.....</b>	<b>379</b>
48.1 The Core Data Stack.....	379
48.2 Persistent Container.....	380
48.3 Managed Objects.....	380
48.4 Managed Object Context .....	380
48.5 Managed Object Model.....	380
48.6 Persistent Store Coordinator.....	381
48.7 Persistent Object Store.....	381
48.8 Defining an Entity Description .....	381
48.9 Initializing the Persistent Container.....	382
48.10 Obtaining the Managed Object Context.....	382
48.11 Setting the Attributes of a Managed Object.....	382
48.12 Saving a Managed Object.....	382
48.13 Fetching Managed Objects.....	383
48.14 Retrieving Managed Objects based on Criteria .....	383

## Table of Contents

48.15 Summary .....	384
<b>49. A SwiftUI Core Data Tutorial.....</b>	<b>385</b>
49.1 Creating the CoreDataDemo Project .....	385
49.2 Defining the Entity Description .....	385
49.3 Creating the Persistence Controller.....	387
49.4 Setting up the View Context .....	387
49.5 Preparing the ContentView for Core Data .....	388
49.6 Designing the User Interface .....	388
49.7 Saving Products .....	390
49.8 Testing the addProduct() Function .....	392
49.9 Deleting Products.....	392
49.10 Adding the Search Function .....	393
49.11 Testing the Completed App .....	396
49.12 Summary .....	396
<b>50. An Overview of SwiftUI Core Data and CloudKit Storage .....</b>	<b>397</b>
50.1 An Overview of CloudKit .....	397
50.2 CloudKit Containers.....	397
50.3 CloudKit Public Database.....	397
50.4 CloudKit Private Databases .....	398
50.5 Data Storage Quotas .....	398
50.6 CloudKit Records.....	398
50.7 CloudKit Record IDs .....	399
50.8 CloudKit References .....	399
50.9 Record Zones .....	399
50.10 CloudKit Console.....	399
50.11 CloudKit Sharing .....	400
50.12 CloudKit Subscriptions .....	400
50.13 Summary .....	400
<b>51. A SwiftUI Core Data and CloudKit Tutorial .....</b>	<b>401</b>
51.1 Enabling CloudKit Support .....	401
51.2 Enabling Background Notifications Support.....	402
51.3 Switching to the CloudKit Persistent Container .....	403
51.4 Testing the App.....	404
51.5 Reviewing the Saved Data in the CloudKit Console .....	404
51.6 Filtering and Sorting Queries .....	405
51.7 Editing and Deleting Records.....	406
51.8 Adding New Records.....	407
51.9 Viewing Telemetry Data.....	408
51.10 Summary .....	409
<b>52. An Introduction to SwiftData .....</b>	<b>411</b>
52.1 Introducing SwiftData .....	411
52.2 Model Classes .....	411
52.3 Model Container .....	412
52.4 Model Configuration .....	412
52.5 Model Context .....	412
52.6 Predicates and FetchDescriptors.....	413

52.7 The @Query Macro .....	413
52.8 Model Relationships .....	413
52.9 Model Attributes .....	414
52.10 Summary .....	415
<b>53. A SwiftData Tutorial.....</b>	<b>417</b>
53.1 About the SwiftData Project .....	417
53.2 Creating the SwiftDataDemo Project .....	417
53.3 Adding the Data Models .....	417
53.4 Setting up the Model Container .....	418
53.5 Accessing the Model Context .....	418
53.6 Designing the Visitor List View.....	418
53.7 Establishing the Relationship .....	419
53.8 Creating the Visitor Detail View .....	420
53.9 Modifying the Content View .....	421
53.10 Testing the SwiftData Demo App.....	422
53.11 Adding the Search Predicate.....	422
53.12 Summary .....	425
<b>54. Building Widgets with SwiftUI and WidgetKit .....</b>	<b>427</b>
54.1 An Overview of Widgets .....	427
54.2 The Widget Extension.....	427
54.3 Widget Configuration Types .....	428
54.4 Widget Entry View.....	429
54.5 Widget Timeline Entries .....	429
54.6 Widget Timeline .....	429
54.7 Widget Provider .....	430
54.8 Reload Policy .....	430
54.9 Relevance.....	431
54.10 Forcing a Timeline Reload.....	431
54.11 Widget Sizes.....	432
54.12 Widget Placeholder.....	432
54.13 Summary .....	433
<b>55. A SwiftUI WidgetKit Tutorial .....</b>	<b>435</b>
55.1 About the WidgetDemo Project.....	435
55.2 Creating the WidgetDemo Project .....	435
55.3 Building the App .....	435
55.4 Adding the Widget Extension .....	438
55.5 Adding the Widget Data .....	439
55.6 Creating Sample Timelines .....	440
55.7 Adding Image and Color Assets.....	441
55.8 Designing the Widget View .....	443
55.9 Modifying the Widget Provider .....	445
55.10 Configuring the Placeholder View.....	446
55.11 Previewing the Widget .....	446
55.12 Summary .....	448
<b>56. Supporting WidgetKit Size Families .....</b>	<b>449</b>
56.1 Supporting Multiple Size Families .....	449

## Table of Contents

56.2 Adding Size Support to the Widget View .....	450
56.3 Summary .....	453
<b>57. A SwiftUI WidgetKit Deep Link Tutorial .....</b>	<b>455</b>
57.1 Adding Deep Link Support to the Widget.....	455
57.2 Adding Deep Link Support to the App .....	458
57.3 Testing the Widget .....	459
57.4 Summary .....	459
<b>58. Adding Configuration Options to a WidgetKit Widget.....</b>	<b>461</b>
58.1 Reviewing the Project Code.....	461
58.2 Adding an App Entity .....	462
58.3 Adding Entity Query .....	463
58.4 Modifying the App Intent .....	463
58.5 Modifying the Timeline Code .....	464
58.6 Testing Widget Configuration .....	465
58.7 Customizing the Configuration Intent UI .....	466
58.8 Summary .....	467
<b>59. An Overview of Live Activities in SwiftUI .....</b>	<b>469</b>
59.1 Introducing Live Activities .....	469
59.2 Creating a Live Activity .....	469
59.3 Live Activity Attributes.....	469
59.4 Designing the Live Activity Presentations .....	470
59.4.1 Lock Screen/Banner.....	471
59.4.2 Dynamic Island Expanded Regions.....	471
59.4.3 Dynamic Island Compact Regions .....	472
59.4.4 Dynamic Island Minimal .....	473
59.5 Starting a Live Activity .....	473
59.6 Updating a Live Activity.....	474
59.7 Activity Alert Configurations .....	475
59.8 Stopping a Live Activity.....	475
59.9 Summary .....	476
<b>60. A SwiftUI Live Activity Tutorial.....</b>	<b>477</b>
60.1 About the LiveActivityDemo Project .....	477
60.2 Creating the Project .....	477
60.3 Building the View Model .....	477
60.4 Designing the Content View.....	478
60.5 Adding the Live Activity Extension.....	480
60.6 Enabling Live Activities Support.....	482
60.7 Enabling the Background Fetch Capability .....	482
60.8 Defining the Activity Widget Attributes .....	483
60.9 Adding the Percentage and Lock Screen Views .....	484
60.10 Designing the Widget Layouts .....	486
60.11 Launching the Live Activity.....	488
60.12 Updating the Live Activity.....	489
60.13 Stopping the Live Activity.....	489
60.14 Testing the App.....	490
60.15 Adding an Alert Notification.....	491



60.16 Understanding Background Updates .....	492
60.17 Summary .....	493
<b>61. Adding a Refresh Button to a Live Activity.....</b>	<b>495</b>
61.1 Adding Interactivity to Live Activities .....	495
61.2 Adding the App Intent.....	495
61.3 Setting a Stale Date.....	496
61.4 Detecting Stale Data.....	497
61.5 Testing the Live Activity Intent .....	498
61.6 Summary .....	498
<b>62. A Live Activity Push Notifications Tutorial.....</b>	<b>499</b>
62.1 An Overview of Push Notifications .....	499
62.2 Registering an APNs Key .....	500
62.3 Enabling Push Notifications for the App .....	501
62.4 Enabling Frequent Updates.....	502
62.5 Requesting User Permission .....	502
62.6 Changing the Push Type .....	504
62.7 Obtaining a Push Token .....	505
62.8 Removing the Refresh Button .....	506
62.9 Summary .....	506
<b>63. Testing Live Activity Push Notifications.....</b>	<b>507</b>
63.1 Using the Push Notifications Console.....	507
63.2 Configuring the Notification .....	508
63.3 Defining the Payload .....	509
63.4 Sending the Notification .....	510
63.5 Sending Push Notifications from the Command Line.....	510
63.6 Summary .....	511
<b>64. Troubleshooting Live Activity Push Notifications .....</b>	<b>513</b>
64.1 Push Notification Problems .....	513
64.2 Push Notification Delivery .....	513
64.3 Check the Payload Structure .....	514
64.4 Validating the Push and Authentication Tokens.....	514
64.5 Checking the Device Log .....	515
64.6 Summary .....	515
<b>65. Integrating UIViews with SwiftUI .....</b>	<b>517</b>
65.1 SwiftUI and UIKit Integration.....	517
65.2 Integrating UIViews into SwiftUI .....	517
65.3 Adding a Coordinator .....	519
65.4 Handling UIKit Delegation and Data Sources .....	520
65.5 An Example Project .....	521
65.6 Wrapping the UIScrollView .....	521
65.7 Implementing the Coordinator .....	522
65.8 Using MyScrollView .....	523
65.9 Summary .....	523
<b>66. Integrating UIViewControllers with SwiftUI.....</b>	<b>525</b>

## Table of Contents

66.1 UIViewControllers and SwiftUI.....	525
66.2 Creating the ViewControllerDemo project.....	525
66.3 Wrapping the UIImagePickerController.....	525
66.4 Designing the Content View.....	526
66.5 Completing MyImagePicker.....	528
66.6 Completing the Content View.....	530
66.7 Testing the App.....	530
66.8 Summary.....	531
<b>67. Integrating SwiftUI with UIKit.....</b>	<b>533</b>
67.1 An Overview of the Hosting Controller.....	533
67.2 A UIHostingController Example Project.....	533
67.3 Adding the SwiftUI Content View.....	534
67.4 Preparing the Storyboard.....	535
67.5 Adding a Hosting Controller.....	536
67.6 Configuring the Segue Action.....	537
67.7 Embedding a Container View.....	540
67.8 Embedding SwiftUI in Code.....	542
67.9 Summary.....	543
<b>68. Preparing and Submitting an iOS 17 Application to the App Store.....</b>	<b>545</b>
68.1 Verifying the iOS Distribution Certificate.....	545
68.2 Adding App Icons.....	547
68.3 Assign the Project to a Team.....	548
68.4 Archiving the Application for Distribution.....	548
68.5 Configuring the Application in App Store Connect.....	549
68.6 Validating and Submitting the Application.....	550
68.7 Configuring and Submitting the App for Review.....	551
<b>Index.....</b>	<b>553</b>

## 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.

Start Here

## 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](mailto: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](mailto:feedback@ebookfrenzy.com)*.

## 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:

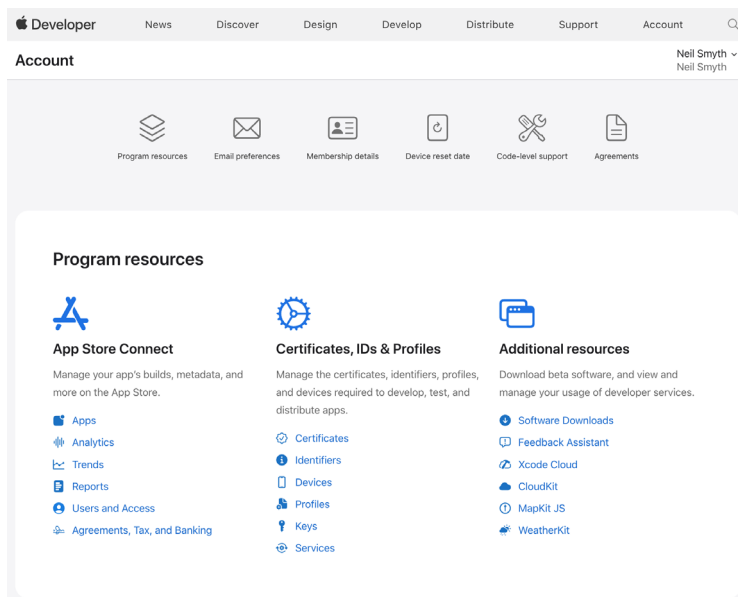


Figure 2-1

## 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.





## 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:

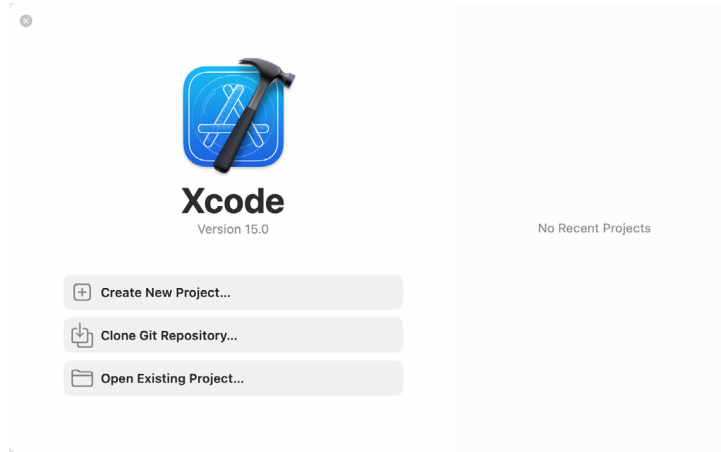


Figure 3-2

### 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.

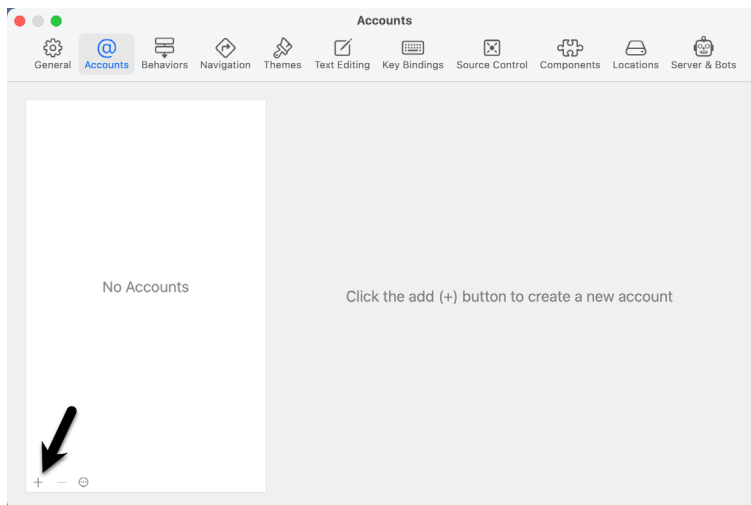


Figure 3-3

### 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:

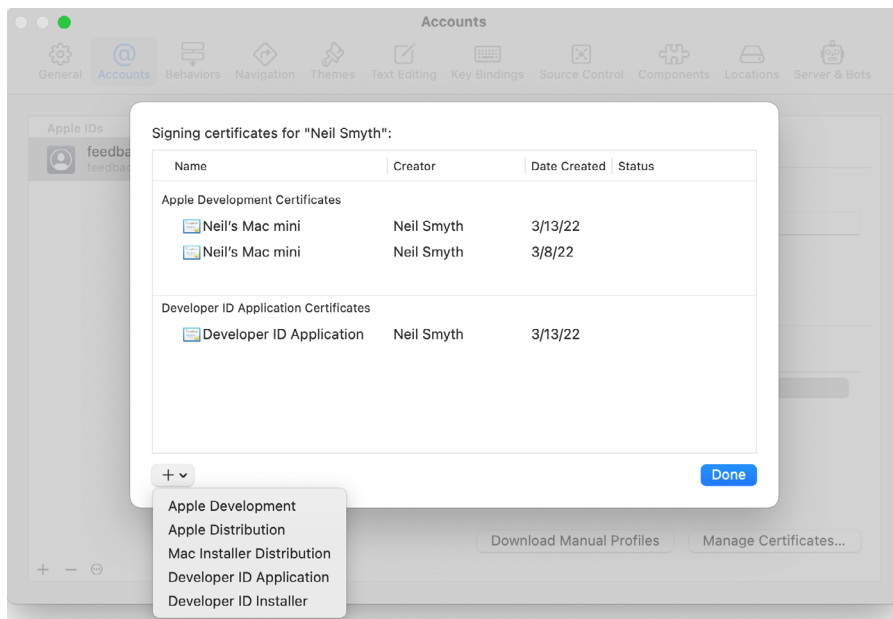


Figure 3-4

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.



## 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

## Working with Array and Dictionary Collections in Swift

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.



## 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:

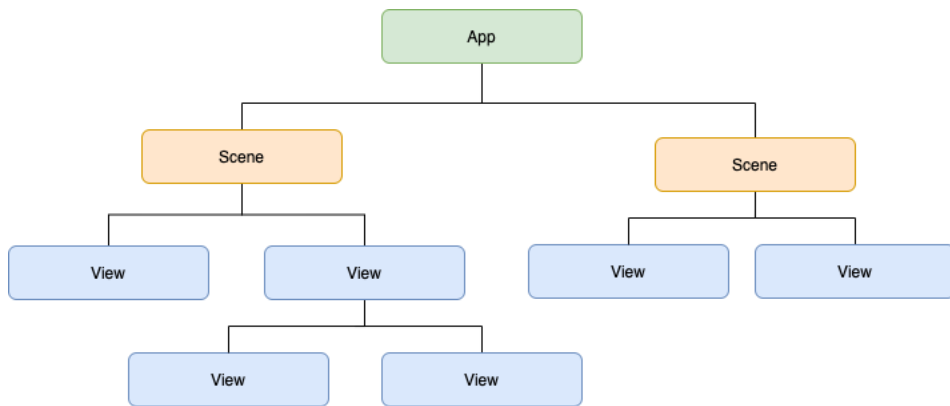


Figure 18-1

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:

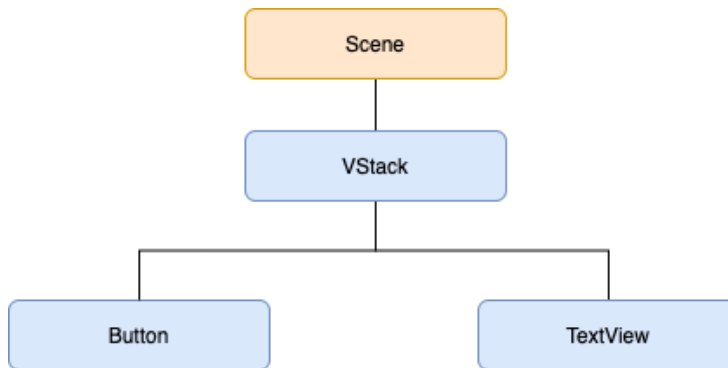


Figure 18-2

## 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.

## 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:

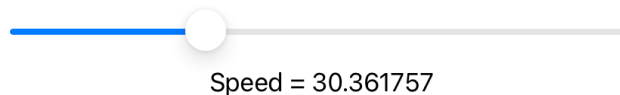


Figure 22-1

## 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.





## 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:

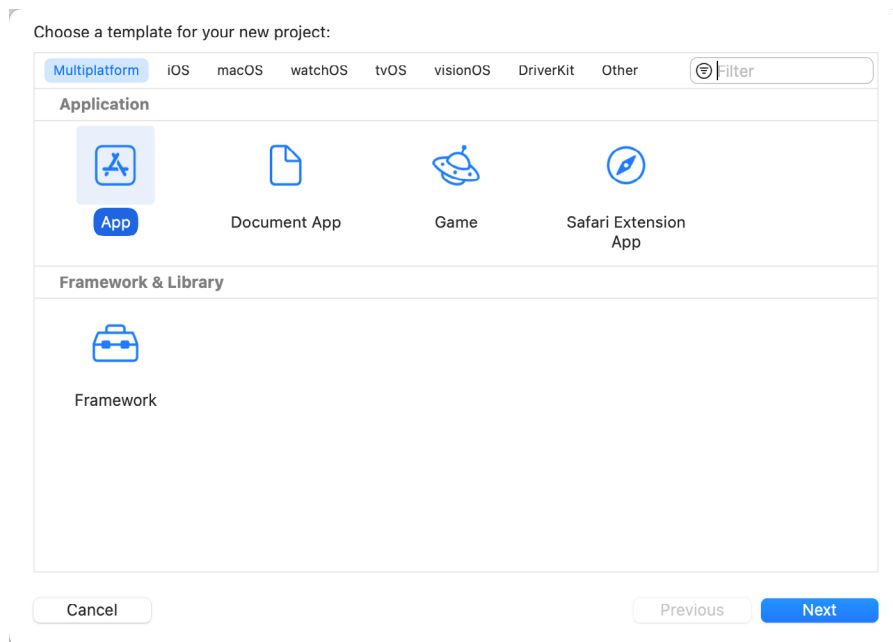


Figure 23-1

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:

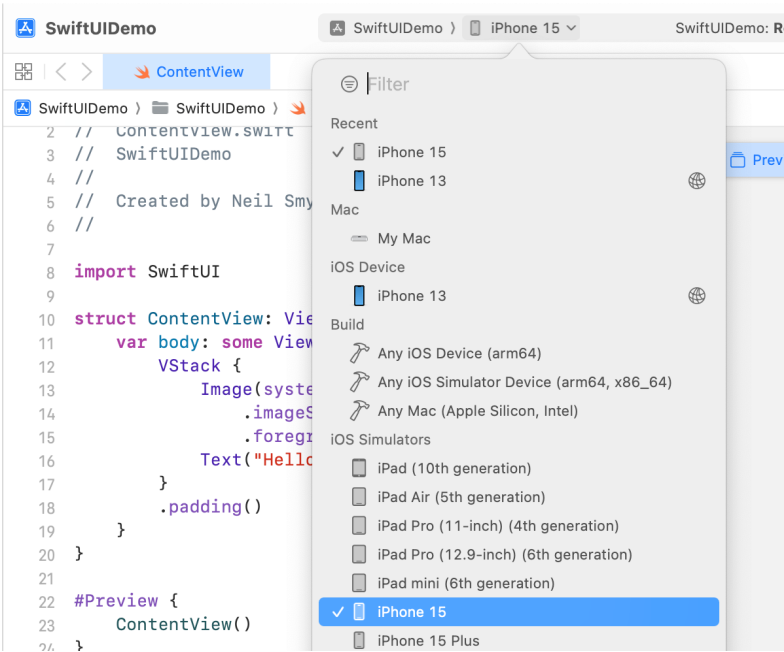


Figure 23-2

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

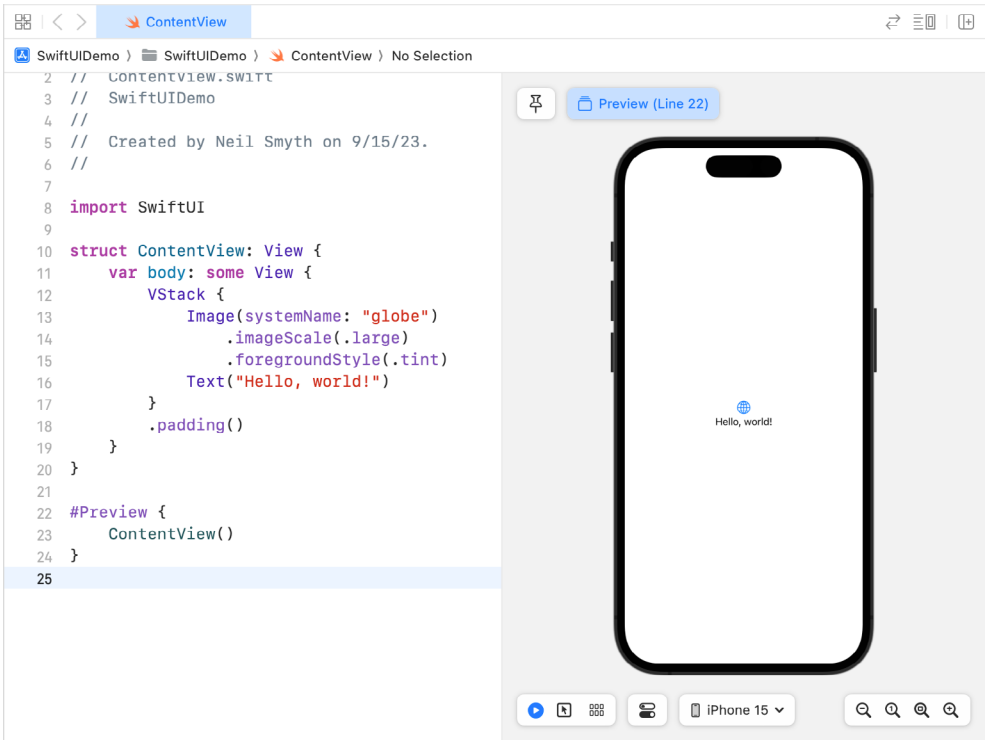


Figure 23-3

## 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:

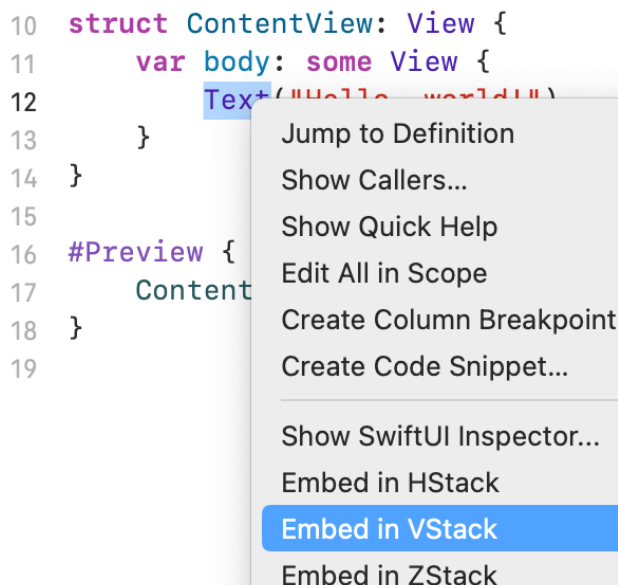


Figure 23-4

## A SwiftUI Example Tutorial

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:

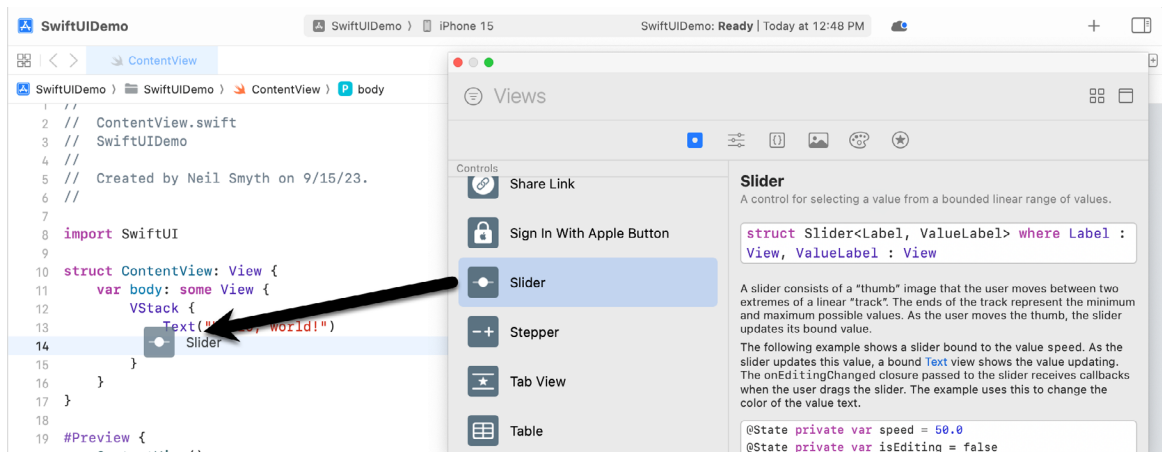


Figure 23-5

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:

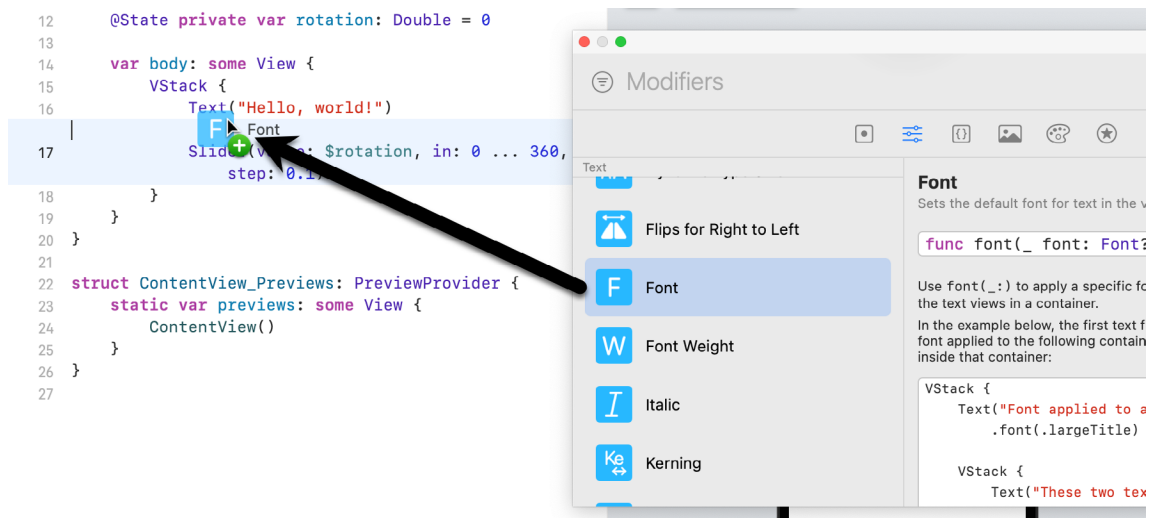


Figure 23-6

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:

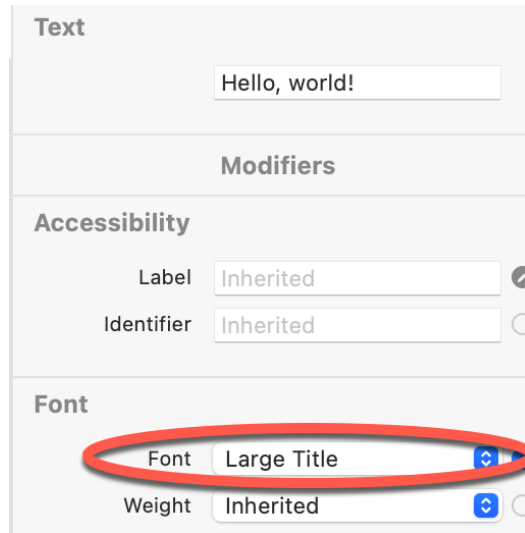


Figure 23-7

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:

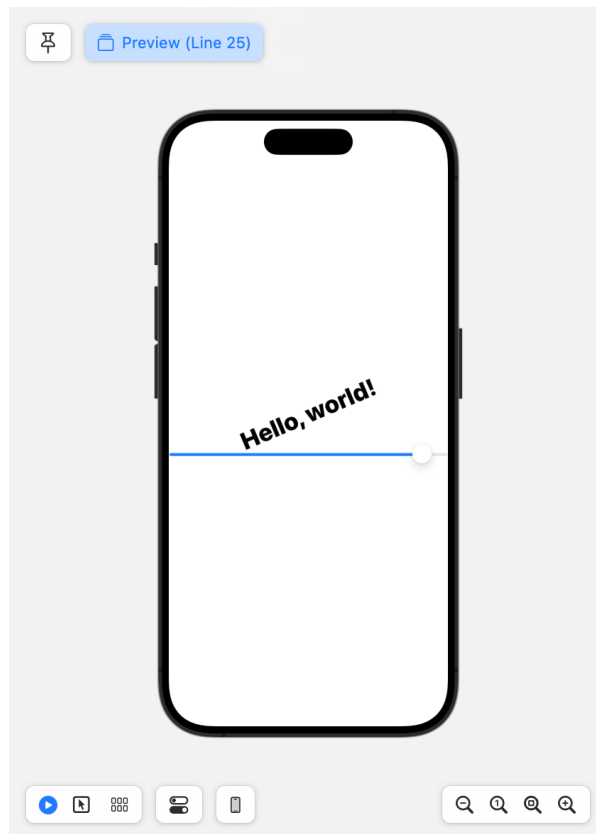


Figure 23-8

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"
```

## A SwiftUI Example Tutorial

```
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:

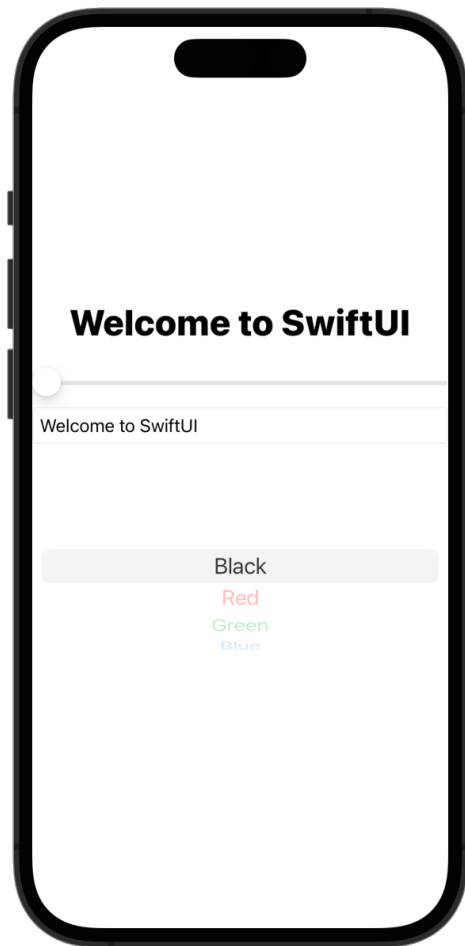


Figure 23-9

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:

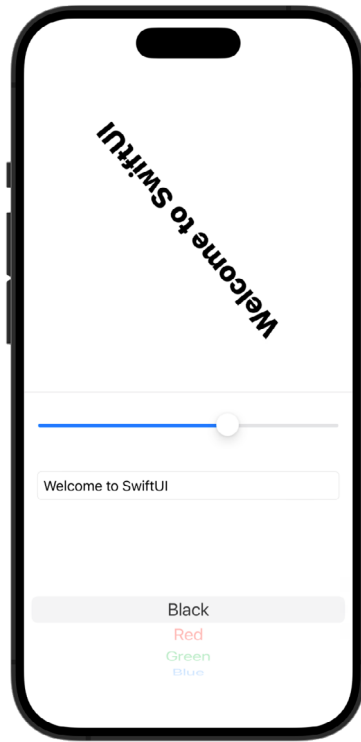


Figure 23-10

## 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.

## 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:

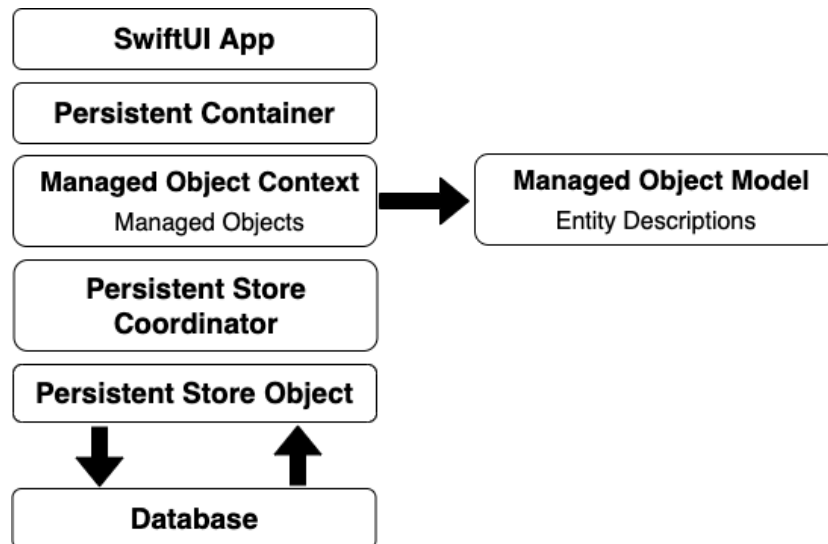


Figure 48-1

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:

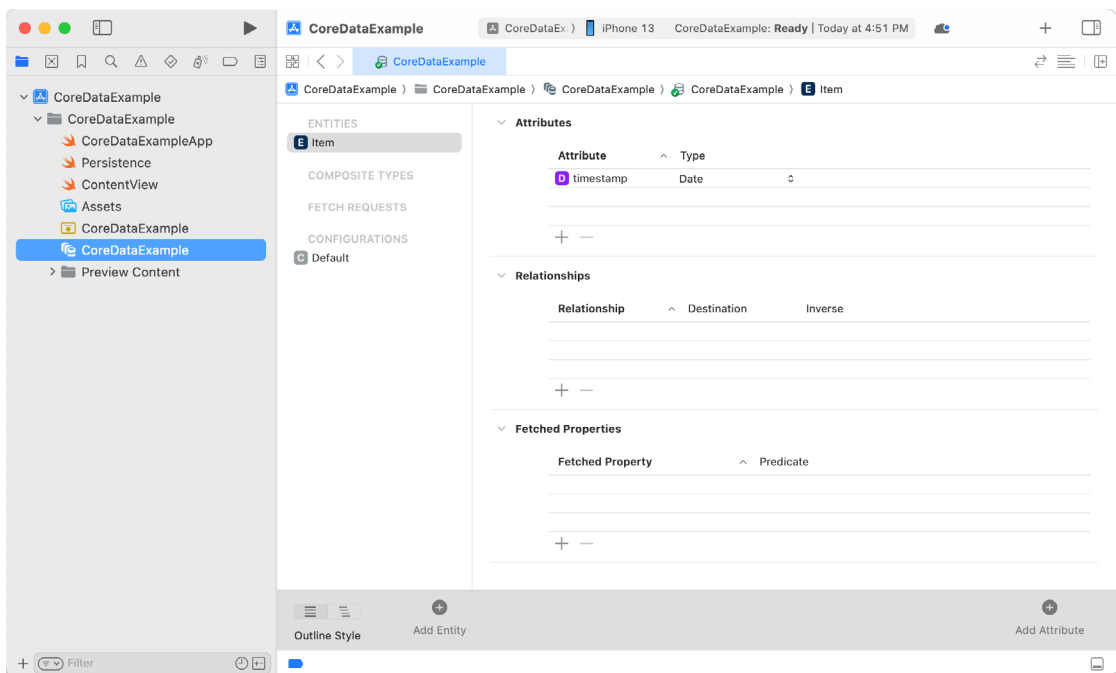


Figure 48-2

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:

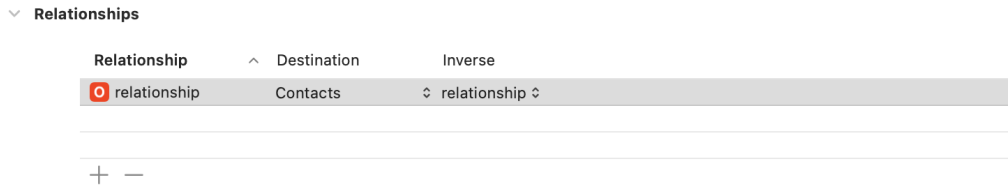


Figure 48-3

## 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 occurred: \(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.

## 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):

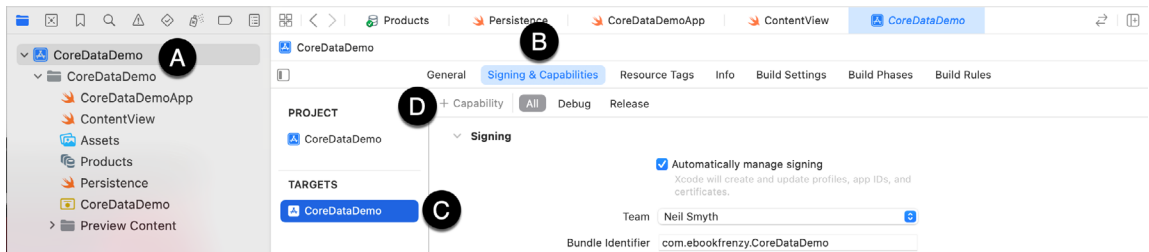


Figure 51-1

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:

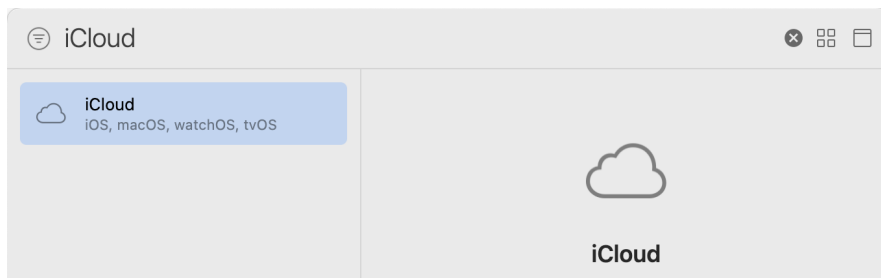


Figure 51-2

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:

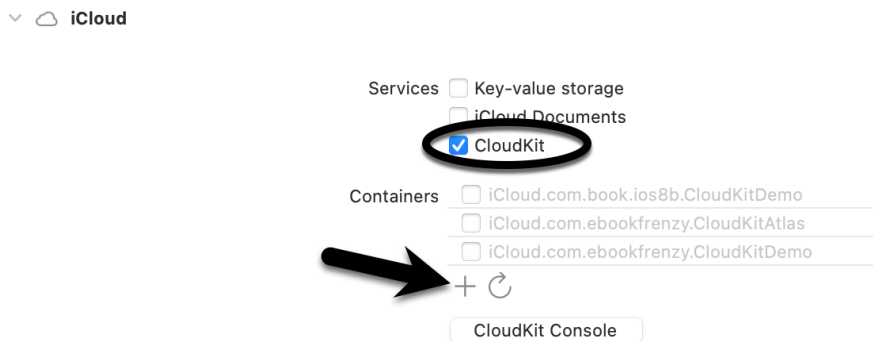


Figure 51-3

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*.

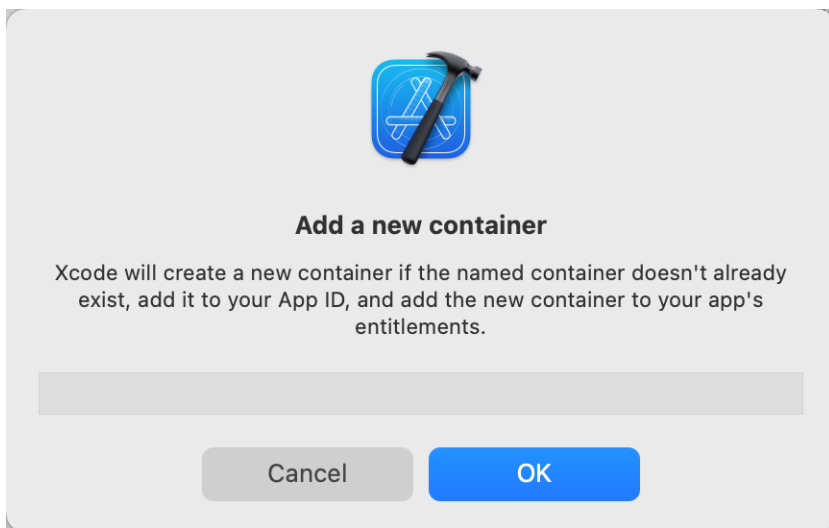


Figure 51-4

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:

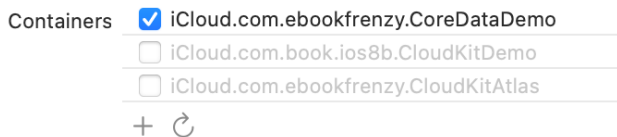


Figure 51-5

## 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:

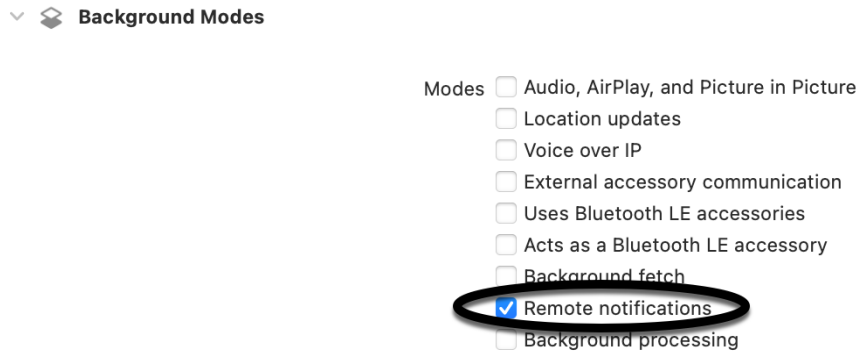


Figure 51-6

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:

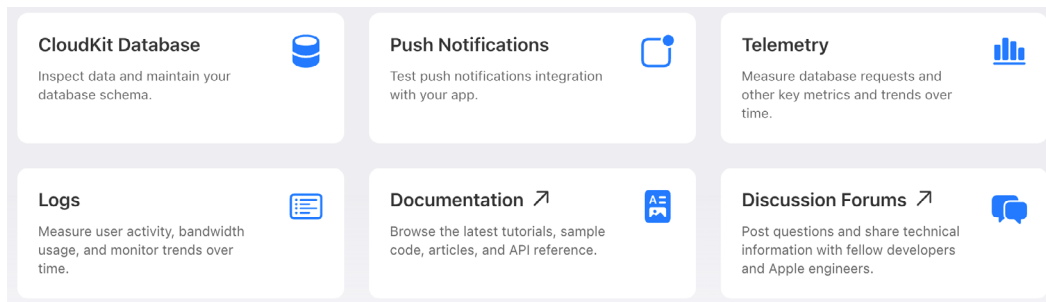


Figure 51-7

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:

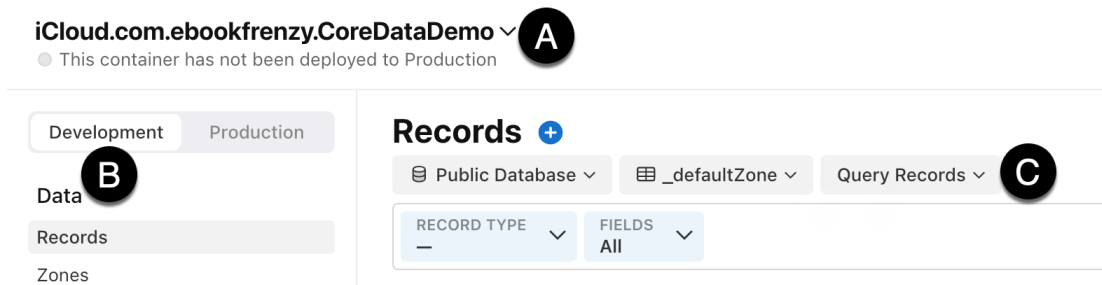


Figure 51-8

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:

 The image shows the 'Records' view after clicking 'Query Records'. It displays a table of product items. The table has columns: NAME, TYPE, CD\_ENTITYNAME, CD\_NAME, CD\_QUANTITY, CH. TAG, CREATED, and MODIFIED. There are six rows of data, each representing a different product like Shampoo, Cat food, Coffee, Batteries, Soap, and Book.
 

NAME	TYPE	CD_ENTITYNAME	CD_NAME	CD_QUANTITY	CH. TAG	CREATED	MODIFIED
144AC333-6CB3-...	CD_Product	Product	Shampoo	12	5d	9/20/2023, 2:31:47 P...	9/20/2023, ;
231D07BB-C44F-4...	CD_Product	Product	Cat food	10	4p	3/24/2022, 5:34:49 ...	3/24/2022, t
C4DA5BA5-63AA-...	CD_Product	Product	Coffee	2	5b	7/29/2022, 6:44:11 P...	7/29/2022, t
D32C4BDC-2E7A-...	CD_Product	Product	Batteries	5	s	3/23/2022, 7:36:08 P...	3/23/2022, ;
F6D0182B-47EA-4...	CD_Product	Product	Soap	12	5e	9/20/2023, 2:33:22 P...	9/20/2023, ;
FB7E28DF-7853-4...	CD_Product	Product	Book	1	5c	9/12/2022, 7:59:58 P...	9/12/2022, 7

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:

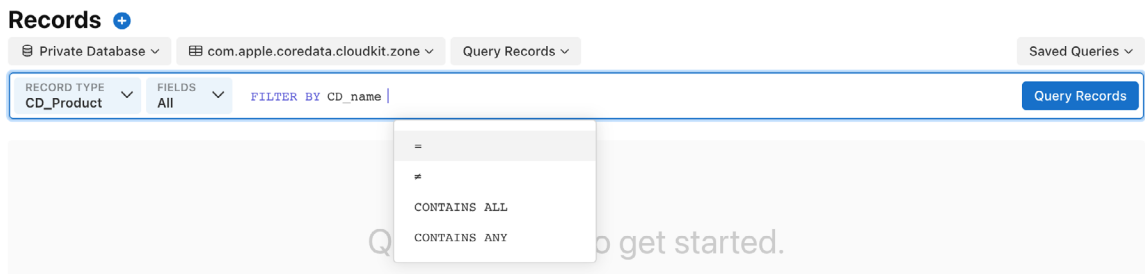


Figure 51-11

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

Private Database ▾

com.apple.coredata.cloudkit.zone ▾

Query Records ▾

Saved Queries ▾

RECORD TYPE  
CD\_Product ▾

FIELDS  
All ▾

CD\_NAME  
= Cat food ▾

Add filter or sort to query

Query Records

NAME	TYPE	CD_ENTITYNAME	CD_NAME	CD_QUANTITY
195F816D-5690-4...	CD_Product	Product	Cat food	5
231D07BB-C44F-4...	CD_Product	Product	Cat food	10
C2E0B03A-2B2B-...	CD_Product	Product	Cat food	10

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:

NAME	TYPE	CD_ENTITYNAME	CD_NAME	CD_QUANTITY
144AC333-6CB3-44AF-81FE-B574576AB15D	CD_Product	Product	Shampoo	12
231D07BB-C44F-44D7-AAC9-4ABA8258F135	CD_Product	Product	Cat food	10

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.



**Record Details**

**Metadata**

Name 144AC333-6CB3-44AF-81FE-B574576AB15D

Type CD\_Product

Database private

Zone com.apple.coredata.cloudkit.zone

Created 9/20/2023, 2:31:47 PM UTC  
by \_58e0f7a9f49bdef36e8970e058a67c7d

Modified 9/20/2023, 2:31:47 PM UTC  
by \_58e0f7a9f49bdef36e8970e058a67c7d

Change Tag 5d

**Fields** 3

CD\_entityName String  
Product

CD\_name String  
Shampoo

CD\_quantity String  
12

**References** 0

**Sharing** Off

**Delete** **Reload** **Save**

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:

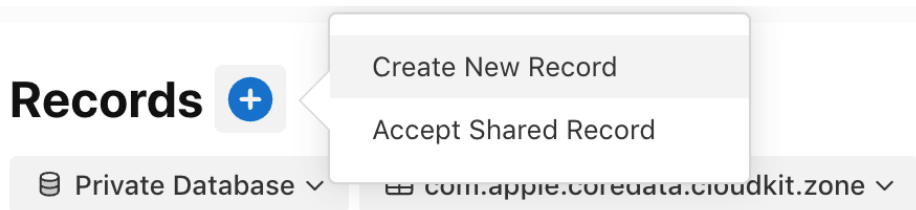


Figure 51-15

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

×

New Record

Metadata

Name

A9A7587E-3AFE-2CE5-7721-15A4BC0

Database

Private Database

Type

CD\_Product

Zone

com.apple.coredata.cloudkit.zone

Fields

3

CD\_entityName

String

Product

CD\_name

String

Stapler

CD\_quantity

String

2

Cancel

Save

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:

Telemetry

Database

Notifications

Usage

iCloud.com.ebookfrenzy.CoreDataDemo

Development

Private Database

All Operations

Figure 51-17

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

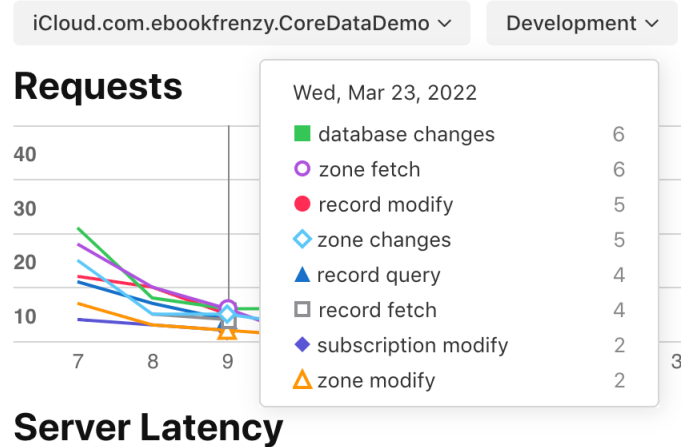


Figure 51-18

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:

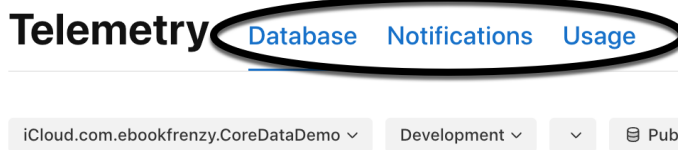


Figure 51-19

## 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.



## 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:

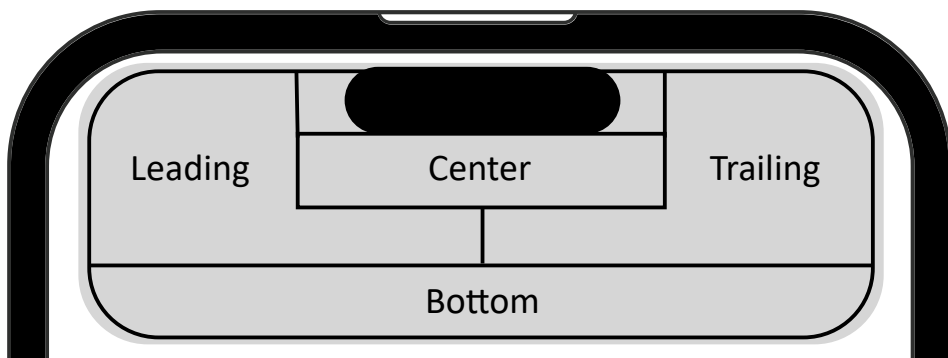


Figure 59-1

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

```

.
.

```

## An Overview of Live Activities in SwiftUI

```
} 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:

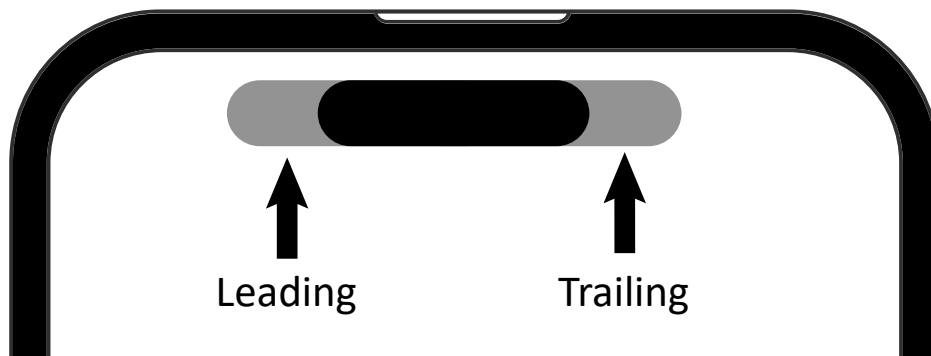


Figure 59-2

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 *detached minimal* positioned to the right of the camera:

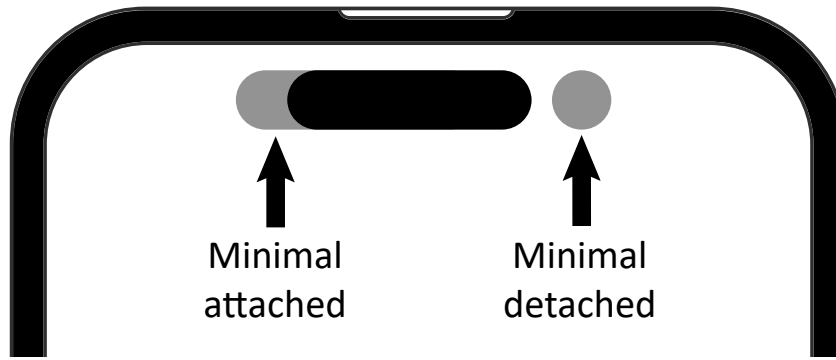


Figure 59-3

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 (arrivalTime > 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.

## 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:

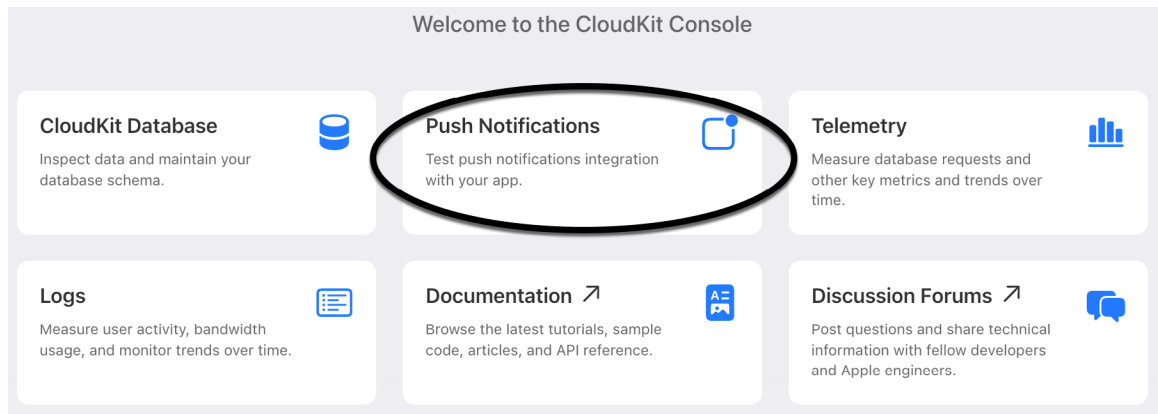


Figure 63-1

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

Testing Live Activity Push Notifications

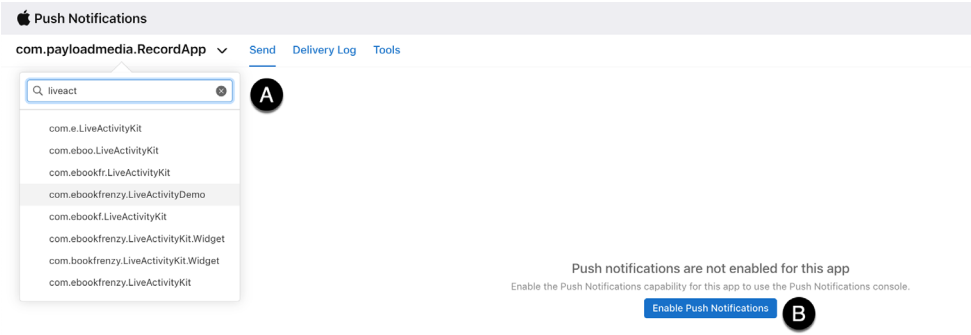


Figure 63-2

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

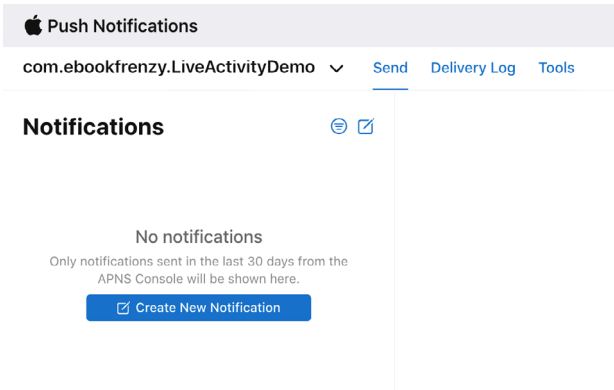


Figure 63-3

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:

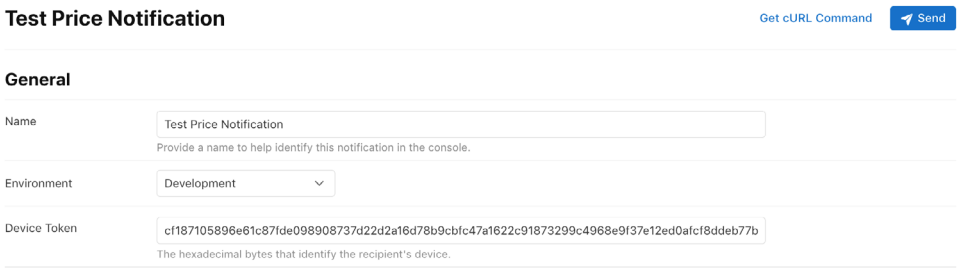


Figure 63-4

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:

**Request Headers**

apns-topic	com.ebookfrenzy.LiveActivityDemo		
apns-push-type	liveactivity	Use the liveactivity push type to send a remote push notification that updates or ends an ongoing Live Activity.	
apns-expiration	Attempt delivery once	10/06/2023, 01:17 PM	UNIX Epoch: 0
APNs will attempt to deliver the notification only once and not store it. A single APNs attempt may involve retries over multiple network interfaces and connections of the destination device.			
apns-priority	Medium (5)	The notification will be delivered based on power considerations on the user's device	

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":{"currentPrice":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

- & 37
- ^ 38
- ^= 39
- << 39
- <= 39
- &= 39
- >> 39
- >= 39
- | 38
- |= 39
- ~ 37
- \$ 154
- \$0 59
- @AppStorage 207, 208, 211, 213
- @Attributes 414
- @Bindable 158
- @Binding 155, 251
- @Environment 160
- @FetchRequest 388, 392
- @GestureState 342
- @main 126
- @MainActor 193
- @Model 411, 417
- @ObservedObject 157
- ?? operator 36
- @Parameter 463
- @Published 156
- @Query 413, 419, 423
- @Relationship 413, 419
- @SceneStorage 207, 209, 211, 213
- @State 153
- @StateObject 157
- Text Styles
  - body 135

- callout 135
- caption 135
- footnote 135
- headline 135
- subheadline 135
- @Transient 415

### A

- Actors 189
  - data isolation 190
  - Declaring 189
  - example 191
  - @MainActor 192
  - MainActor 192
  - nonisolated keyword 190
- adaptable padding 143
- addArc() 322
- addCurve() 322
- addLine() 322
- addLines() 322
- addQuadCurve() 322
- addTask() function 183
- Alignment 143
  - Cross Stack 223
- alignmentGuide() modifier 219
- Alignment Guides 215, 217
- AlignmentID protocol 221
- Alignment Types
  - custom 220
- AND (&&) operator 35
- AND operator 37
- Animation 168, 329
  - automatically starting 334
  - autoreverse 331
  - easeIn 330
  - easeInOut 330
  - easeOut 330
  - explicit 332
  - implicit 329

## Index

- linear 330
- repeating 331
- animation() modifier 329
- AnyObject 92
- AnyTransition 337
- APNs 499
- apns-expiration 508
- APNs Key
  - registering 500
- apns-priority 509
- apns-topic 508
- apns-unique-id 511
- App 123
- app delegate 502
- append() method 239
- AppEntity protocol 462
- App Hierarchy 123
- App Icons 547
- App Intent Configuration 428
- AppIntentConfiguration 462
- App Intents framework 495
- AppIntentTimelineProvider 461
- AppIntentTimelineProvider protocol 430
- Apple Developer Program 3
- Apple Push Notification service 499
- Application Performance 119
- AppStorage 207
- App Store
  - creating archive 548
  - submission 545
- App Store Connect 549
- Architecture
  - overview 123
- AreaMark 353
- Array
  - forEach() 91
  - mixed type 92
- Array Initialization 89
- Array Item Count 90
- Array Items
  - accessing 90
  - appending 91

- inserting and deleting 91
- Array Iteration 91
- Arrays
  - immutable 89
  - mutable 89
- as! keyword 31
- Assets.xcassets 126
- Assistant Editor 537
- async
  - suspend points 177
- async/await 177
- asynchronous functions 176
- Asynchronous Properties 186
- async keyword 177
- async-let bindings 179
- AsyncSequence protocol 185
- Attributes inspector 116
- await keyword 177, 178

## B

- Background Notifications
  - enabling 402
- BarMark 353
- Bézier curves 322
- binary operators 33
- bit operators 37
- Bitwise AND 37
- Bitwise Left Shift 38
- bitwise OR 38
- bitwise right shift 39
- bitwise XOR 38
- body 135
- Boolean Logical Operators 35
- break statement 43
- Build Errors 119

## C

- callout 135
- cancelAll() function 184
- Capsule() 320
- caption 135
- cardinal 358

- case Statement 48
- catch statement 99
  - multiple matches 99
- catmullRom 358
- CGRect 322, 323
- Character data type 23
- chartPlotStyle() 357
- Charts 353, 359
  - chartPlotStyle() 357
  - foregroundColor() modifier 356, 357, 361
  - interpolationMethod() modifier 357
  - interpolation options 357
  - mark type combining 355
  - mark types 353
  - multiple graphs 356
  - passing data to 354
  - PlottableValue 353
  - PointMark 362
  - symbol() modifier 357
- checkCancellation() method 183
- Child Limit 145
- CircularProgressViewStyle 348
- Class Extensions 74
- closed range operator 35
- closeSubPath() 322
- Closure Expressions 58
  - shorthand argument names 59
- closures 51
- Closures 59
- CloudKit 397
  - add container 402
  - Console 399, 404
  - Containers 397
  - Data Storage Quotas 398
  - enabling in Xcode 401
  - filtering and sorting 405
  - NSPersistentCloudKitContainer 403
  - Persistence Container 403
  - Record IDs 399
  - Records 398
  - Record Zones 399
  - References 399
  - Sharing 400
  - Subscriptions 400
  - Telemetry Data 408
- CloudKit console 507
  - push notifications 507
- CloudKit Console 399, 404
- CloudKit Sharing 400
- CloudKit Subscriptions 400
- code editor 109
  - context menu 116
- Color
  - drop() modifier 324
  - .gradient 324
  - inner() modifier 324
  - shadow() modifier 324
- Combine framework 156
- Comparable protocol 86
- Comparison Operators 34
- Completion Handlers 175
- Compound Bitwise Operators 39
- computed properties 65
- concrete type 69
- Concurrent Tasks
  - launching 199
- Conditional Control Flow 44
- Configuration Intent UI 466
- constants 25
- Container Alignment 215
- Container Child Limit 145
- Containers 397
- Container Views 137
- ContentView.swift file 109, 126
- Context Menus 315
- continue Statement 43
- Coordinator 519
- Core Data 379, 385
  - enabling in Xcode 385
  - Entity Description 381
  - Fetch property 380
  - Fetch request 380
  - @FetchRequest 388, 392
  - loadPersistentStores() method 382

## Index

- Managed Object 382
- Managed Object Context 380, 382
- Managed Object Model 380
- Managed Objects 380
- NSFetchRequest 383, 395
- NSPersistentContainer 382
- NSSortDescriptor 392
- Persistence Controller 387
- Persistent Container 380
- Persistent Object Store 381
- Persistent Store Coordinator 381
- Private Databases 398
- Public Database 397
- Relationships 380
- tutorial 385
- View Context 387
- viewContext property 382
- Core Data Stack 379
- CPU cores 175
- curl tool 507
- Custom Alignment Types 220
- Custom Container Views 137
- custom fonts 135
- Custom Paths 322
- Custom Shapes 322

## D

- data encapsulation 62
- Data Isolation 190
- data race 189
- Data Races 184
- Data Storage Quotas 398
- Debug Navigator 119
- debug panel 109
- Debug View Hierarchy 120
- Declarative Syntax 103
- Deep Links 455, 458
- Default Function Parameters 53
- defer statement 100
- Detached Tasks 182
- Developer Mode setting 118
- Developer Program 3

- Devices
  - managing 118
- Dictionary Collections 92
- Dictionary Entries
  - adding and removing 94
- Dictionary Initialization 92
- Dictionary Item Count 94
- Dictionary Items
  - accessing and updating 94
- Dictionary Iteration 94
- didFinishLaunchingWithOptions 503
- DisclosureGroup 243, 269, 282
  - syntax 274
  - tutorial 277
  - using 273
- Disclosures 269
- dismantleUIView() 518
- Divider view 173
- do-catch statement 99
  - multiple matches 99
- Document App
  - creating 363
- Document Content Type Identifier 365
- DocumentGroup 124, 363, 364
  - Content Type Identifier 365
  - Document Structure 367
  - Filename Extensions 365
  - File Type Support 365
  - Handler Rank 365
  - Info.plist 373
  - navigation 369
  - overview 363
  - tutorial 373, 385
  - Type Identifiers 365
- DocumentGroups
  - Exported Type Identifiers 366
  - Imported Type Identifiers 366
- Double 22
- downcasting 30
- DragGesture.Value 342
- drop() modifier 324
- Dynamic Lists 233

**E**

- easeIn 330
- easeInOut 330
- easeOut 330
- EditButton view 256
- Entity Description 381, 385
  - defining 381, 385
- EntityQuery 463
- enum 80, 97
  - associated values 81
- Enumeration 80
- environment() 161
- environmentObject() 161
- Environment Object 159
  - example 201, 205
- Errata 2
- Error
  - throwing 98
- Error Catching
  - disabling 100
- Error Object
  - accessing 100
- ErrorType protocol 97
- Event handling 137
- exclusive OR 38
- Explicit Animation 332
- Expression Syntax 33
- external parameter names 53

**F**

- fallthrough statement 50
- FetchDescriptors 413
- Fetches property 380
- fetch() method 383
- Fetch request 380
- FileDocument class 368
- FileWrapper 368
- fill() modifier 319
- Flexible frames. *See* Frames
- Float 22
- flow control 41
- font
  - create custom 135

- footnote 135
- for-await 185
- forced unwrapping 27
- forEach() 91
- ForEach 171, 233, 247
- foregroundColor() modifier 320
- foregroundStyle() modifier 356, 357, 361
- for loop 41
- Form container 248
- Frames 141, 148
  - Geometry Reader 150
  - infinity 150
- function 461
  - arguments 51
  - parameters 51
- Function Parameters
  - variable number of 54
- functions 51
  - as parameters 56
  - default function parameters 53
  - external parameter names 53
  - In-Out Parameters 55
  - parameters as variables 55
  - return multiple results 54

**G**

- GeometryReader 150
- gesture() modifier 339
- gesture recognizer
  - removal of 340
- Gesture Recognizers 339
  - exclusive 343
  - onChanged 340
  - sequenced 343
  - simultaneous 343
  - updating 342
- Gestures
  - composing 343
- getSnapshot() 430
- getTimeline() 430
- gradient 324

## Index

### Gradients

- drawing 325
- LinearGradient 326
- RadialGradient 326

### Graphics

- drawing 319
- overlays 321

### Graphics Drawing 319

### Grid 299

- adaptive 288
- alignment 304
- column spanning 304
- empty cells 302
- fixed 288
- flexible 288
- spacing 304

### gridCellAnchor() modifier 309

### gridCellColumns() modifier 304

### gridCellUnsizeAxes() modifier 303

### GridItems 287

- adaptive 292
- fixed 293

### Grid Layouts 287

### GridRow 299

- alignment 307
- empty cells 303
- .gridCellAnchor() modifier 309
- .gridCellColumns() modifier 304
- .gridCellUnsizeAxes() modifier 303

### guard statement 45

## H

### half-closed range operator 36

### Handler Rank 365

### headline 135

### Hierarchical data

- displaying 270

### HorizontalAlignment 220, 221

### Hosting Controller 533

- adding 536

### HStack 132, 141

- conversion to VStack 145

## I

### if ... else if ... Statements 45

### if ... else ... Statements 44

### if-let 28

### if Statement 44

### Image view 141

### implicit alignment 215

### Implicit Animation 329

### implicitly unwrapped 30

### Inheritance, Classes and Subclasses 71

### init method 63

### in keyword 58

### inner() modifier 324

### inout keyword 56

### In-Out Parameters 55

### Instance Properties 62

### IntentTimelineProvider 456

### Interface Builder 103

### Interpolation

#### cardinal 358

#### catmullRom 358

#### monotone 358

#### stepCenter 358

#### stepEnd 358

#### stepStart 358

### interpolationMethod() modifier 357

### iOS Distribution Certificate 545

### iOS SDK

#### installation 7

#### system requirements 7

### isCancelled property 183

### isEmpty property 184

### is keyword 32

## L

### Label view 139

### Layout Hierarchy 120

### Layout Priority 146

### lazy

#### keyword 67

### LazyHGrid 287, 295

### LazyHStack 148



- Lazy properties 66
- Lazy Stacks 148
  - vs. traditional 148
- LazyVGrid 287
- LazyVStack 148
- Left Shift Operator 38
- Library panel 114
- Lifecycle Events 195
- linear 330, 358
- LinearGradient 326
- lineLimit() modifier 147
- LineMark 353
- listRowSeparator() modifier 232
- listRowSeparatorTint() modifier 232
- Lists 231
  - dynamic 233
  - hierarchical 242
  - listRowSeparator() modifier 232
  - listRowSeparatorTint() modifier 232
  - making editable 240
  - refreshable 235
  - separators 232
- listStyle() modifier 280
- List view
  - adding navigation 250
  - .listStyle() modifier 280
  - SidebarListStyle 280
- List view
  - tutorial 245
- Live Activity
  - adding interactivity 495
  - App Intent 495
  - frequent updates 502
  - isStale 497
  - payload 509
  - push notifications 507
  - Push Notifications 499
  - Push Token 505
  - pushTokenUpdates() 505
  - stale date 496
- LiveActivityIntent protocol 495
- Live View 17

- loadPersistentStores() method 382
- localizedStandardContains() 424
- local parameter names 53
- Loops
  - breaking from 43

## M

- MainActor 192
- Main.storyboard file 535
- Main Thread , 175
- makeBody() 350
- makeCoordinator() 523, 524
- makeUIView() 518
- Managed Object
  - fetch() method 383
  - saving a 382
  - setting attributes 382
- Managed Object Context 380, 382
- Managed Object Model 380
- Managed Objects 380
  - retrieving 383
- mathematical expressions 33
- Methods
  - declaring 62
- minimap 110
- Mixed Type Arrays 92
- Model Attributes 414
- Model Classes 411
- Model Configuration 412
- Model Container 412, 418
- modelContainer(for:) 418
- modelContainer(for:) modifier 412
- Model Context 412, 418
  - delete() 413
  - fetch() 413
  - insert() 413
  - save() 413
- Model Relationships 413
- modifier() method 136
- Modifiers 136
- monotone 358

## Index

## N

Navigation 231

- implementing 204
- tutorial 245

`navigationDestination(for:)` modifier 237  
`navigationDestination()` modifier 250, 255  
`NavigationLink` 231, 235, 236, 250, 254  
navigation path 239  
`NavigationPath` 239, 255

- `append()` method 239
- `removeLast()` method 239

`NavigationSplitView` 243  
`NavigationStack` 231, 236, 250

- `navigationDestination(for:)` modifier 237
- `NavigationPath` 239
- path 239

`navigationTitle()` modifier 239, 253  
Network Testing 119  
new line 24  
nil coalescing operator 36  
nonisolated keyword 190  
NOT (!) operator 35  
`NSFetchRequest` 383, 395  
`NSPersistentCloudKitContainer` 403  
`NSPersistentContainer` 382  
`NSSortDescriptor` 392

## O

Objective-C 21  
Observable Object

- example 201

`ObservableObject` 153  
`ObservableObject` protocol 156  
Observation 156

- `@Bindable` 158

Observation Framework 158  
`onAppear()` 335  
`onAppear` modifier 196  
`onChanged()` 340  
`onChange` modifier 197  
`onDelete()` 240, 256  
`onDisappear` modifier 196

`onMove()` 241, 256  
`onOpenUrl()` 458  
Opaque Return Types 69  
operands 33  
optional

- implicitly unwrapped 30

optional binding 28  
Optional Type 27  
OR (||) operator 35  
OR operator 38  
`OutlineGroup` 243, 269, 281

- tutorial 277
- using 272

Overlays 321

## P

Padding 143  
`padding()` modifier 143  
`PageTabViewStyle()` 312  
Parameter Names 53

- external 53
- local 53

parent class 61  
Path object 322  
Paths 322  
Performance

- monitoring 119

Persistence Container

- switching to 403

Persistence Controller

- creating 387

Persistent Container 380, 382

- initialization 382

Persistent Object Store 381  
Persistent Store Coordinator 381  
Physical iOS Device 117

- running app on 117

Picker view 153  
`placeholder()` 430, 456  
Playground 11

- creating a 11
- Live View 17

- pages 17
- rich text comments 16
- Rich Text Comments 16
- Playground editor 12
- PlaygroundPage 18
- PlaygroundSupport module 17
- Playground Timelines 14
- PlottableValue 353, 356
- PlottableValue.value 354
- PointMark 353, 362
- Predicates 413
- preferred text size 134
- Preview Canvas 111
- Preview on Device 113
- Preview Pinning 112
- Private Databases 398
- Profile in Instruments 120
- ProgressView 347
  - circular 347, 348
  - CircularProgressViewStyle 348
  - customization 349
  - indeterminate 347, 349
  - linear 347, 348
  - makeBody() 349, 350
  - progressViewStyle() 349
  - ProgressViewStyle 349
  - styles 347
- progressViewStyle() 349
- ProgressViewStyle 349
- Property Wrappers 83
  - example 83
  - Multiple Variables and Types 85
- Protocols 68
- Public Database 397
- push notifications 507
  - curl tool 507
  - troubleshooting 513
- Push Notifications 499
  - enabling 501
- Push Token 505
- pushTokenUpdates() 505

## R

- Range Operators 35
- Record IDs 399
- Record Zones 399
- Rectangle() 319
- RectangleMark 353
- Reference Types 78
- Refreshable lists 235
- refreshable() modifier 235
- removeLast() method 239
- repeatCount() modifier 331
- repeatForever() modifier 331
- repeat ... while loop 42
- Resume button 112
- Right Shift Operator 39
- Rotation 168
- RuleMark 353
- running an app 117

## S

- scale 337
- Scene 123
- ScenePhase 197
- SceneStorage 207
- ScrollView 148, 291
- searchable() modifier 422
- Segue Action 537
- self 67
- SF Symbols 139
  - macOS app 139
- shadow() modifier 324
- Shapes 322
  - drawing 319
- shorthand argument names 59, 91, 171
- SidebarListSyle 280
- sign bit 39
- Signing Identities 9
- Simulator
  - running app 117
- Simulators
  - managing 118
- sleep() method 176

## Index

- slide 337
- Slider view 166
- snapshot() 430, 445, 456
- some
  - keyword 69
- SortDescriptor 413
- source code
  - download 2
- Spacers 143
- Spacer view 173
- Spacer View 143
- spring() modifier 331
- SQLite 379
- Stacks 141
  - alignment 215
  - alignment guides 215
  - child limit 145
  - cross stack alignment 223
  - implicit alignment 215
  - Layout Priority 146
- State Binding 155
- State Objects 157
- State properties 153
  - binding 154
  - example 166
- stepCenter 358
- stepStart 358
- Stored and Computed Properties 65
- stored properties 65
- String
  - data type 23
- stroke() modifier 320
- StrokeStyle 320
- struct keyword 77
- Structured Concurrency 175, 176, 185
  - addTask() function 183
  - async/await 177
  - Asynchronous Properties 186
  - async keyword 177
  - async-let bindings 179
  - await keyword 177, 178
  - cancelAll() function 184
  - cancel() method 183
  - Data Races 184
  - detached tasks 182
  - error handling 180
  - for-await 185
  - isCancelled property 183
  - isEmpty property 184
  - priority 182
  - suspend point 179
  - suspend points 177
  - synchronous code 176
  - Task Groups 183
  - task hierarchy 181
  - Task object 178
  - Tasks 181
  - throw/do/try/catch 180
  - withTaskGroup() 183
  - withThrowingTaskGroup() 183
  - yield() method 183
- Structures 77
- subheadline 135
- subtraction operator 33
- Subviews 132
- suspend points 177, 179
- Swift
  - Actors 189
  - Arithmetic Operators 33
  - array iteration 91
  - arrays 89
  - Assignment Operator 33
  - async/await 177
  - async keyword 177
  - async-let bindings 179
  - await keyword 177, 178
  - base class 71
  - Binary Operators 34
  - Bitwise AND 37
  - Bitwise Left Shift 38
  - Bitwise NOT 37
  - Bitwise Operators 37
  - Bitwise OR 38
  - Bitwise Right Shift 39

- Bitwise XOR 38
- Bool 23
- Boolean Logical Operators 35
- break statement 43
- calling a function 52
- case statement 47
- character data type 23
- child class 71
- class declaration 61
- class deinitialization 63
- class extensions 74
- class hierarchy 71
- class initialization 63
- Class Methods 62
- class properties 61
- closed range operator 35
- Closure Expressions 58
- Closures 59
- Comparison Operators 34
- Compound Bitwise Operators 39
- constant declaration 25
- constants 25
- continue statement 43
- control flow 41
- data types 21
- Dictionaries 92
- do ... while loop 42
- error handling 97
- Escape Sequences 24
- exclusive OR 38
- expressions 33
- floating point 22
- for Statement 41
- function declaration 51
- functions 51
- guard statement 45
- half-closed range operator 36
- if ... else ... Statements 44
- if Statement 44
- implicit returns 21, 52
- Inheritance, Classes and Subclasses 71
- Instance Properties 62
- instance variables 62
- integers 22
- methods 61
- opaque return types 69
- operators 33
- optional binding 28
- optional type 27
- Overriding 72
- parent class 71
- Property Wrappers 83
- protocols 68
- Range Operators 35
- Reference Types 78
- root class 71
- single expression functions 52
- single expression returns 52
- single inheritance 71
- Special Characters 24
- Stored and Computed Properties 65
- String data type 23
- structured concurrency 175
- structures 77
- subclass 71
- suspend points 177
- switch fallthrough 50
- switch statement 47
  - syntax 47
- Ternary Operator 36
- tuples 26
- type annotations 25
- type casting 30
- type checking 30
- type inference 25
- Value Types 78
- variable declaration 25
- variables 25
- while loop 42
- Swift Actors 189
- SwiftData 411, 417
  - @Attributes 414
  - FetchDescriptors 413
  - @Model 411

## Index

- Model Attributes 414
- Model Classes 411
- Model Container 412, 418
- modelContainer(for:) 418
- modelContainer(for:) modifier 412
- Model Context 412, 418
- Model Relationships 413
- Predicates 413
- @Query 419, 423
- @Relationship 413, 419
- SortDescriptor 413
- @Transient 415
- Swift Playground 11
- Swift Structures 77
- SwiftUI
  - create project 107
  - custom views 129
  - data driven 104
  - Declarative Syntax 103
  - example project 163
  - overview 103
  - Subviews 132
  - Views 129
- SwiftUI Project
  - anatomy of 125
  - creating 107
- SwiftUI Views 129
- SwiftUI View template 203
- SwiftUI vs. UIKit 104
- switch statement 47
  - example 47
- switch Statement 47
  - example 47
  - range matching 49
- symbol() modifier 357
- synchronous code 176

## T

- Tabbed Views 311
- tabItem() 313
- Tab Items 313
- Tab Item Tags 313

- TabView 311
  - PageTabViewStyle() 312
  - page view style 312
  - tab items 313
- tag() 313
- Task.detached() method 182
- Task Groups 183
  - addTask() function 183
  - cancelAll() function 184
  - isEmpty property 184
  - withTaskGroup() 183
  - withThrowingTaskGroup() 183
- Task Hierarchy 181
- task modifier 199
- Task object 178
- Tasks 182
  - cancel() 183
  - detached tasks 182
  - isCancelled property 183
  - overview 181
  - priority 182
- Telemetry Data 408
- ternary operator 36
- TextField view 169
- Text Styles 134
- Text view
  - adding modifiers 167
  - line limits 146
- Threads
  - overview , 175
- throw statement 98
- timeline() 430, 461, 464
- timeline entries 428
- TimelineEntryRelevance 431
- timeline() method 445
- ToggleButton view 154
- ToolBarItem 240
- toolbar() modifier 240, 253
- transition() modifier 337
- Transitions 329, 336
  - asymmetrical 338
  - combining 337

- .move(edge)
  - edge) 337
- .opacity 337
- .scale 337
- .slide 337
- try statement 98
- try! statement 100
- Tuple 26
- TupleView 130
- Tutorial
  - Charts 359
- Type Annotations 25
- type casting 30
- Type Checking 30
- Type Identifiers 365
- Type Inference 25
- type safe programming 25

## U

- UINavigationController 533
- UIImagePickerController 525
- UIKit 103
- UIKit integration
  - data sources 520
  - delegates 520
- UIKit Integration 517
  - Coordinator 519
- UINavigationController 503
- UInt8 22
- UInt16 22
- UInt32 22
- UInt64 22
- UIRefreshControl 519
- UIScrollView 520
- UIView 517
  - SwiftUI integration 517
- UIViewController 525
  - SwiftUI integration 525
- UIViewControllerRepresentable protocol 525
- UIViewRepresentable protocol 519
  - makeCoordinator() 519
- unary negative operator 33

- Unicode scalar 25
- Uniform Type Identifier 365
- Unstructured Concurrency 181
  - cancel() method 183
  - detached tasks 182
  - isCancelled property 183
  - priority 182
  - yield() method 183
- upcasting 30
- updateView() 518
- UserDefaults 208
- UTI 365
- UTType 368
- UUID() method 233

## V

- Value Types 78
- variables 25
- variadic parameters 54
- VerticalAlignment 220, 221
- View 124
- ViewBuilder 138
- View Context 387
- viewController property 382
- ViewDimensions 221
- ViewDimensions object 219
- View Hierarchy
  - exploring the 120
- ViewModifier protocol 136
- Views
  - adding 203
  - as properties 133
  - modifying 133
- VStack 141
  - conversion to HStack 145

## W

- where clause 29
- where statement 49
- while Loop 42
- WidgetCenter 431
- Widget Configuration 427

## Index

WidgetConfiguration 427  
WidgetConfigurationIntent 428, 461  
WidgetConfiguration protocol 427  
Widget Configuration Types 428  
Widget Entry View 427, 429  
Widget Extension 427  
widgetFamily 453, 454  
Widget kind 427  
WidgetKit 449, 455  
    Configuration Intent UI 466  
    Deep Links 455, 458  
    Intent Configuration 427, 428  
    introduction 427  
    Reload Policy 430  
    ReloadPolicy  
        .after(Date) 430  
        .atEnd 430  
        .never 430  
    size families 449  
    snapshot() 430  
    Static Configuration 427, 428  
    timeline() 430  
    timeline entries 428  
    TimelineEntryRelevance 431  
    timeline example 440  
    timeline() method 445  
    Timeline Reload 431  
    tutorial 435  
    Widget Configuration 427  
    WidgetConfiguration protocol 427  
    widget entry view 443  
    Widget Entry View 427, 429  
    Widget Extension 427, 438  
    widgetFamily 453, 454  
    widget gallery 452  
    Widget kind 427  
    Widget Provider 430, 445  
    Widget Sizes 432  
    widget timeline 428  
Widget Provider 430  
Widget Sizes 432  
widget timeline 428

widgetUrl() 457  
WindowGroup 124, 126  
withAnimation() closure 332  
withTaskGroup() 183  
withThrowingTaskGroup() 183

## X

Xcode  
    account configuration 8  
    Attributes inspector 116  
    code editor 109  
    create project 107  
    debug panel 109  
    device log 515  
    enabling CloudKit 401  
    entity editor 381  
    installation 7  
    Library panel 114  
    preferences 8  
    preview canvas 111  
    Preview Resume button 112  
    project navigation panel 109  
    SwiftUI mode 107  
XOR operator 38

## Y

yield() method 183

## Z

ZStack 141, 215  
    alignment 225  
ZStack Custom Alignment 225