SECOND EDITION

Covers Xcode 12 / iOS 14 / Swift and Android Studio 4 / Android 11 / Kotlin

MASTERING MOBILE APPLICATION DEVELOPMENT

Learning iOS and Android Side-By-Side



Jonathan Engelsma and Hans Dulimarta

Mastering Mobile Application Development

2nd Edition

Learning iOS and Android Side-By-Side

Jonathan Engelsma and Hans Dulimarta

2021

Mastering Mobile Application Development - 2nd Edition: Learning iOS and Android Side-By-Side
by Jonathan Engelsma and Hans Dulimarta

© 2021 Jonathan Engelsma and Hans Dulimarta. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the authors.

While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

This book is for sale at http://leanpub.com/mobile-app-dev

This version was published on 2021–01–18 (a5889b4).

For Mieke and Liana
who have been patient with us.

Contents

Preface **ix**

1 IDE **1**
1.1 Mobile Development Overview . . . 1
1.2 iOS and Android Contrasted . . . 3
1.3 Development Environment . . . 5
1.4 Xcode . . . 6
1.4.1 Interface Builder . . . 7
1.4.2 Integrated Debugger and Simulators . . . 9
1.4.3 Tools and Frameworks for Testing . . . 10
1.4.4 Xcode Documentation . . . 10
1.4.5 Managing Third Party Components with CocoaPods . . . 11
1.5 Android Studio . . . 12
1.5.1 Layout Editor . . . 15
1.5.2 Gradle . . . 17
1.5.3 Managing SDKs . . . 18
1.5.4 Android Emulators . . . 22
1.5.5 Test Framework and Debugger . . . 25
1.5.6 Managing Libraries . . . 27
1.5.7 Android Support Libraries . . . 27
1.5.8 Upgrading to Android Studio 4.0 . . . 28

2 Writing Your First Mobile App **29**
2.1 Traxy App Overview . . . 29
2.2 Writing Our First Mobile App . . . 34
2.2.1 Platform Similarities . . . 36
2.3 Your Initial iOS App . . . 36
2.3.1 Creating a Project in Xcode . . . 37
2.3.2 Laying out the screen in Interface Builder . . . 40
2.3.3 Adding Outlets and IBActions . . . 45
2.3.4 Internationalization . . . 51
2.4 Your Initial Android App . . . 58
2.4.1 Creating a Project in Android Studio . . . 58
2.4.2 Laying out the screen in Layout Editor . . . 63
2.4.3 Renaming the Activity and Validating the Login Form . . . 68
2.4.4 View Animations . . . 72
2.4.5 String Resource Editor . . . 73

2.4.6 Internationalization . . . 75
2.4.7 Source Code Revision Control . . . 76

3 Mobile App Architectures **77**
3.1 Mobile Software Architecture . . . 77
3.1.1 Process State Transitions . . . 78
3.1.2 Mobile Resource Constraints . . . 79
3.1.3 Application State Transitions . . . 80
3.1.4 Application Process: a Home for UIViewController/Activity . . . 82
3.1.5 Java Virtual Machine . . . 82
3.2 Architectural Choice . . . 83
3.2.1 Model View Presenter . . . 85
3.2.2 Model-View-ViewModel . . . 85
3.3 Lifecycles . . . 86
3.4 Android Manifest . . . 88
3.5 Android Architecture Components . . . 90

4 Scene Transitions **93**
4.1 Overview . . . 93
4.2 Fleshing out Traxy for Login and User Signup . . . 94
4.3 Scene transitions in iOS . . . 94
4.3.1 Adding new view controllers to the storyboard . . . 96
4.3.2 Adding new UIViewController classes . . . 98
4.3.3 Establishing the identity of view controllers in Interface Builder . . . 100
4.3.4 Adding segues in Interface Builder . . . 102
4.3.5 Completing the view controller implementations . . . 107
4.3.6 View Controller Lifecycles . . . 113
4.4 Scene Transitions in Android . . . 116
4.4.1 Adding New `Activity` Classes . . . 117
4.4.2 App Bar Menu Items . . . 123
4.4.3 Using Android `Intents` . . . 124
4.4.4 Android Activity Stack . . . 128
4.4.5 Finalize `SignUpActivity` and `MainActivity` . . . 128
4.4.6 Handling MainActivity Logout Menu . . . 129
4.4.7 Navigation Graphs . . . 130
4.4.8 Adding ViewModel . . . 138
4.4.9 Implicit `Intents` . . . 140
4.4.10 Unresolved `Intents` . . . 141

5 User Interfaces **143**
5.1 Laying out Mobile User Interfaces . . . 144
5.1.1 UI Styles and Themes . . . 145
5.2 Layout and Styling on iOS . . . 146
5.2.1 Auto Layout Overview . . . 147
5.2.2 Introducing Auto Layout Constraints in Interface Builder . . . 150
5.2.3 Adding Auto Layout to Traxy . . . 157
5.2.4 Styling in iOS . . . 160

5.2.5 Adding Styling to the Traxy App . . . 160
5.3 Layout and Styling in Android . . . 164
5.3.1 Which Layout? . . . 167
5.3.2 Login Screen in `LinearLayout` . . . 174
5.3.3 Login Screen in `ConstraintLayout` . . . 175
5.3.4 Laying out the Login Screen . . . 177
5.3.5 Applying Model-View-ViewModel . . . 181
5.3.6 Styling/Theme in Android . . . 182
5.3.7 Introduction to Material Theme . . . 185

6 Collections of Data **189**
6.1 Introduction . . . 189
6.1.1 Challenge 1: List does not fit on the screen! . . . 191
6.1.2 Challenge 2: Where's the data? . . . 191
6.1.3 A Framework for Rendering Lists . . . 192
6.2 Implementing Table Views in iOS . . . 195
6.2.1 Where's the Data? . . . 195
6.2.2 `UITableView` Overview and Some Housekeeping . . . 195
6.2.3 Implementing `UITableViewDataSource` . . . 198
6.2.4 Partitioning a Table View into Sections . . . 203
6.2.5 Making it Pretty via a Custom Cell and `UITableViewDelegate` . . . 204
6.3 Android `RecyclerView` . . . 209
6.3.1 Adapters and View Holders . . . 212
6.3.2 Your First RecyclerView . . . 213
6.3.3 Using ViewModel With RecyclerView . . . 225
6.3.4 Event Handling . . . 226
6.3.5 Custom `CoordinatorLayout.Behavior` . . . 229

7 Libraries **233**
7.1 Use or Produce? . . . 233
7.1.1 Traxy New Feature . . . 234
7.1.2 Using Google Places API . . . 235
7.1.3 Enhancing Traxy with Third Party Libraries . . . 236
7.2 Integrating Third Party Libraries in Xcode . . . 237
7.2.1 Using CocoaPods to Manage External Libraries . . . 237
7.2.2 Enhancing Traxy . . . 239
7.3 Library Management in Android . . . 246
7.3.1 Using Data Generator . . . 249
7.3.2 Google Play Services Libraries . . . 250
7.3.3 Using Google Places API . . . 252
7.3.4 Resolving Version Conflicts . . . 260

8 Cloud Datastore Integration **263**
8.1 Mobile Backend . . . 263
8.1.1 Creating a New Firebase Project . . . 265
8.1.2 Firebase Authentication . . . 266
8.1.3 Firebase Realtime Database . . . 267

8.1.4 Firebase Listeners . . . 268
8.1.5 Firebase Cloud Firestore . . . 269
8.2 Integrating Firebase with iOS . . . 270
8.2.1 Using Firebase Authentication for User Management . . . 271
8.2.2 Using Firebase Cloud Firestore as a Cloud Backend . . . 277
8.3 Firebase in Android . . . 282
8.3.1 Repository . . . 284
8.3.2 Logging Out . . . 286
8.3.3 Storing Data Into Firebase Cloud Firestore . . . 288
8.3.4 Retrieving Data From Cloud Firestore . . . 292
8.3.5 Kotlin Coroutines for Firebase . . . 295

9 Working With Multimedia 297
9.1 Personal Media Players . . . 297
9.1.1 Codecs: Media Data Compressors . . . 298
9.1.2 The Need for Caching . . . 299
9.1.3 Privacy Concerns . . . 299
9.1.4 Enhancements to Traxy . . . 299
9.2 Multimedia in iOS . . . 302
9.2.1 Working with UITableViewController . . . 302
9.2.2 Presenting Action Sheets . . . 304
9.2.3 Working with Multimedia on iOS . . . 304
9.2.4 Capturing Photos and Videos . . . 306
9.2.5 Uploading Media to Firebase . . . 315
9.2.6 Downloading and Displaying Captured Photos and Videos . . . 325
9.3 Multimedia in Android . . . 338
9.3.1 Expandable FABs . . . 341
9.3.2 Taking Thumbnails . . . 344
9.3.3 Taking Full-Size Photos . . . 346
9.3.4 Selecting Photos from Gallery . . . 354
9.3.5 Uploading Media Files to Cloud Storage . . . 354
9.3.6 Recording Videos and Playback . . . 360
9.3.7 Media Streaming with ExoPlayer 2.x . . . 377

10 Working With Audio 379
10.1 Working with Audio in iOS . . . 380
10.1.1 Audio Capture and Playback . . . 380
10.1.2 Saving Audio Journal Entries to Firebase . . . 386
10.1.3 Adding a Segue to AudioViewController . . . 388
10.2 Audio Processing in Android . . . 395
10.2.1 Recording Audio . . . 395
10.2.2 Audio Playback . . . 407
10.2.3 Audio Focus . . . 410

11 Networking **413**
11.1 Accessing Web APIs in Mobile Apps . . . 413
11.1.1 Network Programming Challenges . . . 414
11.1.2 General Guidelines for Networking in Mobile Apps . . . 416
11.1.3 Web API Data Representations . . . 418
11.1.4 Extending Traxy With Weather Data . . . 419
11.2 Networking on iOS . . . 421
11.2.1 Performing HTTP Requests From iOS . . . 421
11.2.2 Processing JSON Representations in iOS . . . 423
11.2.3 Extending Traxy with Weather Data From the Open Weather Web API . . . 424
11.3 Networking on Android . . . 434
11.3.1 HTTP Requests Using OkHttp3 . . . 434
11.3.2 HTTP Requests Using Retrofit . . . 438

12 User Interface **443**
12.1 UI Navigation . . . 443
12.1.1 Tabbed Applications . . . 443
12.1.2 Grid Layouts . . . 445
12.1.3 Extending the Traxy User Interface . . . 445
12.2 Tabs and Collections on iOS . . . 446
12.2.1 Adding Tabs to Traxy . . . 446
12.2.2 Handling Authentication and Refactoring Out Common Code . . . 449
12.2.3 Customizing UITabBarController to Handle Authentication . . . 450
12.2.4 Adding the Calendar Tab . . . 454
12.2.5 Adding Cover Photo Selection . . . 456
12.3 Tabs and Grid Layouts on Android . . . 474
12.3.1 Pager Adapter . . . 475
12.3.2 Update The Navigation Graph . . . 476
12.3.3 Adding Tabs . . . 477
12.3.4 Enhancing `MainFragment` . . . 478
12.3.5 Completing `MonthlyFragment` . . . 481
12.3.6 Adding Calendar . . . 481
12.3.7 Handling the Edit Button . . . 483
12.3.8 Querying and Displaying Photos . . . 488
12.3.9 Selecting Cover Photo . . . 490
12.3.10 Calendar Decorator . . . 493
12.3.11 Bottom Navigation . . . 495

13 Location and Maps **499**
13.1 Location Aware Apps . . . 499
13.1.1 Location Sensing Technologies . . . 501
13.1.2 Privacy Considerations . . . 504
13.1.3 Adding Location and Maps to Traxy . . . 504
13.2 Maps and Location on iOS . . . 504
13.2.1 Displaying and Interacting with Maps . . . 506
13.3 Maps and Location on Android . . . 513

13.3.1 Adding Maps to Traxy . . . 513
13.3.2 Handling Map Interactions . . . 516
13.3.3 Where am I? . . . 517
13.3.4 Mocked Locations in Android Studio . . . 520

A Learning Swift **523**
A.1 Language Overview . . . 523
A.2 Variable Declarations . . . 525
A.3 Strings . . . 526
A.4 Collection Types . . . 527
A.5 Control Flow . . . 528
A.6 Functions . . . 530
A.7 Closures . . . 532
A.8 Tuples . . . 534
A.9 Optionals . . . 535
A.10 Objects . . . 538
A.10.1 Enums . . . 538
A.10.2 Structs . . . 539
A.10.3 Classes . . . 540
A.11 Protocols . . . 541
A.12 Extensions . . . 541
A.13 Further Study . . . 542

B Learning Kotlin **543**
B.1 Inheritance and Overrides . . . 543
B.1.1 Constructors . . . 545
B.1.2 Static Members . . . 547
B.1.3 Data Class . . . 548
B.1.4 Implementing an Interface . . . 548
B.1.5 Child Class Constructor . . . 548
B.2 Variable Declarations . . . 548
B.3 Nullable Types and the Elvis Operator . . . 549
B.3.1 Using `lateinit` . . . 550
B.4 Lambda Expressions . . . 550
B.5 Control Structures . . . 551
B.5.1 Generalized “Switch” . . . 552
B.6 Functions . . . 552
B.7 Java Setters and Getters . . . 553
B.8 Java Setters and Getters . . . 554

Preface

Hundreds of books have been written on iOS and Android since the debut of these platforms some 10 years ago. Many of them are well-written by very capable authors. This leads to the inevitable question, why write another?

Both of us have both taught and developed on these platforms since they appeared, and also have worked on their predecessors (Java MicroEdition, Symbian, etc.) Over the years, we've made a number of observations. First, becoming proficient on either Android or iOS involves a steep learning curve. Apple and Google help address this by providing large and comprehensive sets of how-to guides and reference documentation. Second, while distinct in terms of programming languages, frameworks and basic vocabulary, these platforms actually have more in common than initially appears. Third, while developers tend to specialize on one or the other, it is not unusual for mobile app developers to become proficient on both platforms over time, as many production apps have both iOS and Android implementations.

Often we find ourselves in situations where we will be explaining a concept on one of the platforms to a student or developer in terms of what they already know about the other platform. Similarly, we will have students in one of our courses in which we are teaching iOS tell us they just got assigned to a new project at work involving Android or vice versa. At one point we attempted to teach a semester course in mobile app development teaching both Android and iOS at the same time. We soon realized that other than a few scattered blog posts, there was very little instructional material available that explored the platforms side-by-side in a comparative manner. Though we personally were proficient on both platforms, we had not spent enough time organizing our thoughts and materials in a way that would allow the learner to efficiently bridge concepts that related across the platforms, and simultaneously call out situations where a fundamentally different approach was called for due to intrinsic differences. Instead, our lessons tended to quickly diverge into the details and nomenclature of the specific platform and any conceptual overlap was completely obfuscated and lost to the students. The net result was that students spent an *enormous* amount of effort learning a *little bit* about both platforms.

These experiences and observations led us to the conclusion that we could do much better if we organized our thoughts and intentionally created a grand tour of both platforms that iteratively led the student from concept to concept in a logical manner, calling out the similarities and differences along the way. This vision, followed by a lot of discussion and hard work (and encouragement from students and colleagues!) has resulted in the book that you now hold in your hands.

Our approach is to imagine a fictitious yet non-trivial app that is clearly representative of the many apps consumers currently use daily on their smartphones. Studying how this app is implemented on both iOS and Android in a logical step-by-step iterative fashion, exploring various facets of the platforms as we go. Each chapter therefore focuses on one a single key aspect of modern mobile apps and begins with a discussion of the concept itself, and a discussion of the commonalities and potential differences in iOS and Android, pertaining to that particular topic.

Once the concepts have been fleshed out at a conceptual level, we then introduce concrete examples of how those features are implemented on both iOS and Android.

This comparative approach to learning a software-related topic is not without precedent. For many years accredited university computer science programs have offered courses in programming languages. These courses typically survey a variety of programming language principles, constructs, models, and styles, and illustrate them with concrete examples from a variety of programming languages. A common learning objective is to help the students understand programming language principles that lay beyond the mere syntax, and recognize the concept when it appears again in perhaps an entirely different syntax. Armed with these insights, the student not only learns to write better code in a particular language, but can also rapidly assimilate and learn new languages. For example, a student who has been exposed to lambdas in Ruby and who understand how Ruby implements such constructs, and when they are appropriate to use, will immediately recognize and understand closure expressions when they encounter them for the first time in Swift.

Given the enormous demand for mobile apps and the rapid pace in which iOS and Android are evolving (not mention the platforms that will soon appear on the horizon.) we believe this comparative approach that intentionally calls out concepts in a way that transcends the implementation details will serve the student aspiring to be a professional mobile app developer well. We believe this approach is also germane to the professional developer who is struggling to keep up with the constant avalanche of new mobile implementation frameworks, concepts, and in some cases even entirely new programming languages!

Hence our intended audience consists of both the university level computer science student, as well as the practicing professional developer. We believe this approach will help the mobile app newbie with no prior knowledge of mobile application development become rapidly proficient on both iOS and Android, and well-prepared to rapidly assimilate whatever their mobile app development futures may hold. We also believe this approach is an excellent way for the experienced developer who is already proficient on either iOS or Android and would like to become proficient on the other. By bridging the knowledge they already have to the other platform, they will quickly orient themselves and become productive.

For the newbie, this book can be read in a couple of ways. You could simply read through the entire book, and exploring both platforms as you go. Alternatively, you could also read just the conceptual portion of each chapter, and then focus on the code examples of the platform you are most interested in learning. Once you've completed the book in that manner, you could go back and study the coding examples of the remaining platform.

For readers who are already proficient on one of the platforms, be sure to read through the conceptual discussion at the beginning of each chapter and then dig deeper into the code examples of the platform you are less proficient with.

Prerequisite Knowledge

General

We don't intend for this book to be an introduction to programming. We assume the reader has already attained a reasonable level of programming proficiency. We also assume readers have prior experience using a modern integrated development environment for editing, compiling, running,

and debugging code. It would also be helpful, though not required, for the reader to have some experience with a modern source code version control tool such as git.

Swift

Beginning in 2014, Swift has become Apple's flagship language for all of their product lines, gradually replacing the Objective-C that preceded it. We expect many of our readers will not have prior experience with the Swift programming language. Nevertheless, we are not going to spend a lot of time teaching you Swift. Instead, we're going to point you to the many really excellent learning materials that are already available online. That said, take a look at Appendix A in the back of the book. We've provided a whirlwind tour of the language in which we call out some of the essential Swift concepts that you'll need to master in order to start writing iOS apps. Appendix A also provides a list of helpful online learning resources for coming up to speed in the Swift language.

The good news is that Swift is a modern programming language that can be readily learned by anyone who is already proficient in another language. If you have prior iOS experience using Objective-C, the news is really good in that everything you know about the iOS frameworks is still apropos, just the syntactic sugar is different.

Kotlin

Prior to 2017, Java was the lingua franca of the Android developer. Given its popularity as a teaching language in university computer science programs, as well as its wide adoption in industry, the odds are very high that you've already done your share of bit flipping in this language. At Google I/O 2017, Google announced that Kotlin would be the primary programming language for developing Android applications.

Once again, if your Kotlin experience is limited, we'd suggest you google yourself up some online tutorials and get started. At minimal, you'll need to be proficient in the following areas:

- Using inheritance for writing a (child) class by extending an (abstract) parent class and overriding the parent methods in the child class
- Writing a class that implements a given interface
- Using Kotlin data classes
- Extension methods
- Writing lambda expressions
- Operator overloading
- Coroutines

Take a look at Appendix B for a slightly more detailed look at some of the Kotlin concepts you'll need to be proficient with.

Example Source Code

All of the code examples in this book are available for you on github.com:

https://github.com/gvsucis/mobile-app-dev-book-2ndEdition

The Android sample code is written in Kotlin version 1.4.x., and the iOS sample code is written in Swift version 5.3.x.

The above repository contains two sub-repositories, one for Android and one for iOS. In the sub-respositories, the final snapshot from each chapter is available in buildable project form for each chapter as a branch with the name chXX, where XX can be replaced with the chapter of interest from Chapter 01 to Chapter 13. We would encourage you to work on the code examples on your own, but keep a clone of the official code somewhere on your hard disk for reference when needed.

If you find any errors or typos in this book, please do us a favor and report them by creating an issue in the above GitHub project. If you include a chapter and page number in your entry, that will be very helpful. Thank you in advance.

A Note on Coding Conventions

A somewhat dated practice among Java programmers is to explicitly add the object reference `this` when referring to instance variables or methods. This practice was beneficial primarily in improving the readability of the code, but also triggered auto-completion in the Eclipse IDE that was widely used at the time. Today, modern IDEs such as XCode and Android Studio completely eliminate the need for this with syntax highlighting that clearly differentiates between local and instance variables as well as more proactive auto-completion features. For this reason, most style guides today for modern object-oriented languages specifically recommend against this practice.

In the code examples in this book, we've intentionally adopted this practice simply to improve the readability of the code in printed form. For example, we often show a method of a class or a small fragment of code without the entire class context. Hence, we believe this practice will help the reader better understand the code examples printed in the book. So for Java code examples, we will explicitly prepend the `this` reference to instance variables in our code, and in Swift code example we will do the same with the `self` reference.

Acknowledgments

A huge thank you is due to former graduate student Moe Azuz who dedicated many hours to designing the screens for the Traxy app that is used as an example in this book, and also read through the manuscript and providing us a significant amount of thought provoking feedback. We would also like to thank our GVSU colleague Szymon Machajewski who reviewed our manuscript and also provided many helpful comments along the way.

Thanks also to the many GVSU students enrolled in our mobile apps courses over the past couple of years who also played a huge role in both inspiring and shaping this book. In particular, we want to thank the following students who were especially helpful to us during this project: Alvaro Ardila, Nicolas Arias, Thomas Bailey, Josiah Campbell, Joshua Eldbridge, Roland Heusser, Kent Sinclair, Cindy Vannoy, and David Whitters.

Finally, we're grateful to Amy Zevenbergen of Studio10 Design for the nifty and fitting cover design.

—The expectations of life depend upon diligence; the mechanic that would perfect his work must first sharpen his tools.

Confucius

1 Introduction to Native Mobile Application Development

Chapter Objectives

After reading this chapter you should be able to

- Understand the mobile application development landscape.
- Understand the requirements of and establish a native iOS development environment.
- Understand the requirements of and establish a native Android development environment.

1.1 Mobile Development Overview

Smartphones and tablets have become commonplace in the past decade, and are now the predominant means whereby users produce and consume content over the Internet. By 2012 smartphones and tablets had begun to eclipse desktop PC sales, particularly in emerging economies which represent a lion share of the growth [17]. Along with these ubiquitous, powerful hand-held touch-enabled computing devices, came their respective application ecosystems: Google's Play Store and Apple's iTunes App Store. At their annual World-Wide Developer's Conference in June 2016, Apple announced that the number of apps available in the App Store exceeded 2 million. Google's Play Store boasts slightly more apps than Apple. These two ecosystems remain the most popular means by which end users are introduced to new life enriching experiences on their devices. Both

offer what are properly referred to as *native applications*, or simply *apps* in every day vernacular. Native applications are programs that are implemented by software developers, compiled into binary bundles, and published on the respective application ecosystems. End users can purchase, download and install these applications on their devices.

Native apps are not the only way to get new user experiences onto smartphones and tablets. Figure 1.1 offers a wide spectrum of exemplar techniques and tooling available to software developers that can be used to implement applications for mobile devices. These range from applications that are implemented as web applications on the one side, to downloadable native apps on the other. In between these two extremes there are also a significant number of hybrid approaches - frameworks that allow developers to implement applications in a common tool suite and then generate buildable projects for the various target native platforms.

When deciding how a particular mobile application should be developed, there are a number of trade-offs to consider. For example, mobile web apps are typically easier on the budget, since a single responsive web app can be implemented that renders quite well on a variety of smartphone, tablet, and desktop client devices. Mobile web apps are also much easier to manage from a deployment perspective in that changes can be made on the server side and immediately made available to all client devices using the application. Native mobile apps on the other hand, target a very specific software stack (most often iOS, iPadOS[1] or Android). The programming languages and frameworks used are different, and require a separate implementation for each target platform. To update a native app, the developer must submit the updated binary to the respective app store. While the level of testing and vetting that happens in the approval process varies across (and within!) ecosystems, the simple fact is that the app may not become available to users for a number of hours, or even days. Even then, the update is not made until each individual client device downloads and installs the update. While this can happen automatically on current versions of Android and iOS, many users still do manual updates of their installed apps.

Native apps on the other hand are advantageous for at least a couple of reasons. First and foremost, web apps are implemented via the browsers available on the client devices. From a user experience perspective, this results in a sort of "lowest common denominator" like situation in which a number of the device's underlying capabilities are simply not available to the application via the browser, and browser features vary across implementations. Hence developers will approach the implementation in a way in which the app will function on all devices / browsers, which means certain capabilities will not be used. While HTML5 has begun to change this with its tighter integration with the underlying device features, conventional wisdom is that native mobile apps deliver a far more optimal end user experience than mobile web apps. A second often overlooked advantage that native apps have over web apps is that of being discoverable. Since the advent of the Apple's App Store in 2008, users have very quickly been conditioned to find new app experiences via searching the respective app store on their devices. Popular apps can often be located and installed within a few key strokes. This fact is one of the reasons that most popular consumer facing experiences on devices today are experienced via a native app, even when there is a parallel companion web app available.

Hybrid frameworks attempt to take the best of both web apps and native apps and make them available to developers. Like web apps, hybrid frameworks address the market fragmentation problem that challenges native app developers, by making it possible to develop the app once in the framework's software stack, and via tooling, rendering it into a native app bundles that

[1] iPadOS is the official successor operating system to iOS 12 on iPad. While we will focus on the iPhone in this book, much of what is presented here applies to iPad as well.

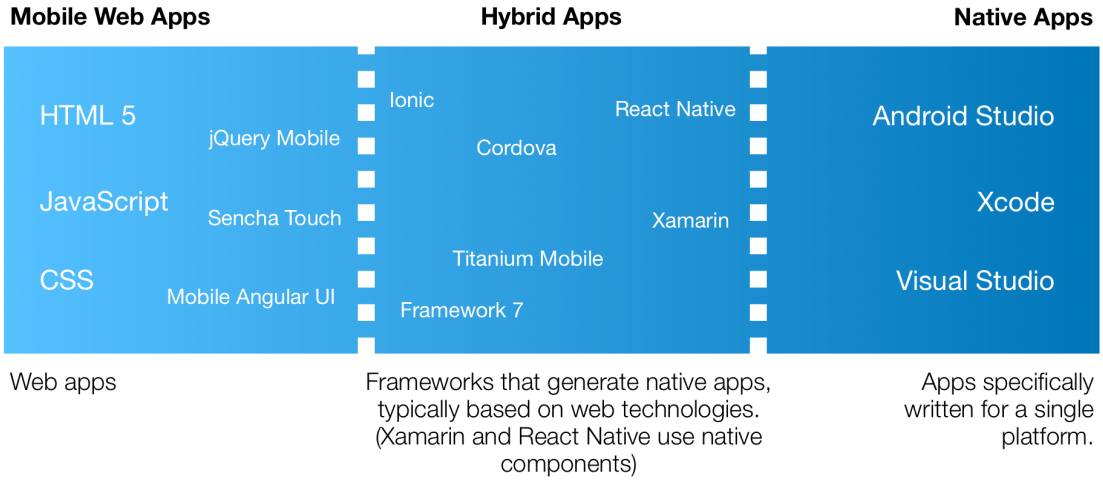


Figure 1.1: New content and experiences can be published on mobile devices via a spectrum of techniques, ranging from web apps to native apps.

can be published in the app stores. While significant progress is being made in this area (such as React Native and Flutter), many of these frameworks are based on HTML5, CSS3, JavaScript and the resulting apps in the app stores are almost immediately recognizable as apps that were not developed using the native SDKs.

In summary, there is no single approach for developing mobile apps that is optimal in every situation. As always, the best approach depends on the unique circumstances (budget, target audience, timeline, etc.) of a given project. A general rule of thumb is that anytime you have a "captive audience" (e.g. a situation in which users have no choice but to use your app), the web app, or hybrid approach is worth looking into. This often occurs when apps are targeting a specific set of users within an organization, such as an enterprise or university community. On the other hand, whenever your audience consist of the general population of consumers (who already have over 2 million apps to choose installing over yours!) building a beautiful, easy to discover, easy to use native app for each of the target platforms will often be a requirement.

In this book, we are going to focus specifically on native app development. In particular, we will be limiting our study to the two most popular mobile app platforms available today: Apple's iOS and Google's Android. In the remainder of this chapter, we will look at the requirements for establishing development environments for Android and iOS, and take a brief tour of the integrated development environments (IDEs) made freely available to developers by Apple and Google. Lets begin by looking at some of the similarities and differences of these two platforms.

1.2 iOS and Android Contrasted

At a surface level, Android and iOS share many similarities. For example:

- Both are highly popular smartphone software platforms.
- Both are controlled solely by a single large tech company, Apple and Google, respectively.

- Both continue to evolve at a fairly rapid rate, which means as a mobile developer, you'll always be on the learning curve.
- Both have large healthy app ecosystems in which developers can publish their apps.
- Both platforms have attracted a large community of active developers.

As might be expected however, there are also a number of significant differences that have ramifications on how developers approach them. For example:

- While Google has exclusive control of the Android software stack, it makes it freely available to a large number of device manufacturers. On the other hand, Apple has exclusive control of both the iOS software stack as well as the hardware platforms it runs on.
- The Android software stack has been open-sourced by Google. The iOS platform is Apple proprietary.
- Native Android development can be done on a variety of desktop operating systems (Windows, macOS, or Linux), but native iOS development can only be done on macOS.
- While both Apple and Google now use humans to vet (e.g. to make sure they meet the store's requirements / guidelines) the apps that get published in their respective ecosystems, the Apple review process takes significantly longer and is usually more rigorous.
- Android development is done in either Java or Kotlin[2] (and C/C++ via Native Development Kit/NDK support). iOS development is done in Swift or Objective-C.

One of the more significant practical ramifications of these differences is that the target Android device population is much more fragmented than that of iOS in terms of both hardware and software. Not only do the physical form factors of the devices differ widely (in terms of screen size / density, keyboards, buttons, etc.) but the platform software also tends to lag behind the most recent release. Given the extreme competition, Android hardware manufacturers would rather invest their resources in designing new hardware products than working hard to make sure legacy products are running the latest greatest version of Android. Hence, the vast majority of the devices in use are typically not running the latest release of Android. Google's Android One program[12], originally launched in 2014, attempts to address this issue by guaranteeing at least two years of OS upgrades and security updates for three years. Nevertheless, a lack of OS updates and the need for app developers to support legacy versions of Android remains a reality within the Android ecosystem.

iOS hardware and software on the other hand is controlled exclusively by Apple. Not only is the target device population more homogeneous, but Apple also makes it easy for a large proportion of the older legacy devices in use to upgrade to the latest version of iOS. It is normal for the majority of iOS devices in use to be running the latest version of iOS within a few weeks of it becoming available for over-the-air update.

The different positions Google and Apple have taken within the tech industry gives rise to trade-off situations that developers should be conscious of. One could argue that Google's model drives more rapid innovation at the hardware level (e.g. manufacturer have to distinguish their

[2] In this edition of the book, we will be using the Kotlin language for Android development.

products via hardware innovations, since they are running more or less the same software as their competitors!) and at the same time offer more economical hardware choices to the consumer. One could also argue that though it often comes with a higher price, the Apple ecosystem offers better software support and hence a more consistent user experience to the consumer.

From a developer perspective, the practical take-away is that Android developers will necessarily spend more time and effort making sure their apps will run and look good on as many target devices as possible. Google has done a good job of anticipating this situation, and as you will soon see, some of the differences in tooling to a certain extent are driven by this reality. New features and techniques released by Google will necessarily be incorporated at a slower rate into production Android apps.

While the iOS developer has an easier time getting production apps up and running and looking good on end user devices, it can also be observed that the amount of change that has occurred in the underlying iOS user interface has been much less than is the case for Android. iOS 7 was the first major change in the iOS user interface since its conception, and even that upgrade was mostly cosmetics and not a major retrofit of how things actually get done on the device. As a consequence to the rather slow pace in which Apple has evolved the iOS UI, many major iOS apps have introduced highly customized user interfaces to better distinguish their app offerings within the App Store. This in turn has lifted the bar a bit for all iOS app developers, and apps that do minimal customizations and rely primarily on Apple's stock UI controls may be viewed as stodgy and undesirable to end users.

1.3 Establishing Your Mobile App Development Environment

If you are a command line interface (CLI) oriented developer who insists on carrying out all software development tasks from the command line, you may figure out a way to make that happen for mobile app development. However, for the rest of us, we're going to do our iOS and Android development using the official integrated development environments Apple and Google provide for us, which are as follows:[3]:

- Xcode 12.3: Apple's IDE for apps running on iOS, watchOS, macOS, and tvOS.
- Android Studio 4.1.1: Google's IDE for apps running on Android on phones/tablets, Wear OS, TV, Android Auto and Android Things.

Besides features commonly found in other modern IDEs like: code completion, code refactor, code analysis, these IDEs support seamless integration with source control management programs (like git). However, one of the most important features is the ability of these IDEs to connect and run and debug your app on either a device simulator or a physical device. To enable this feature, additional "modules" may have to be installed into the respective IDE.

Let's take a brief tour of both of these IDEs in terms of understanding their run-time requirements and the features they provide.

[3]We intentionally exclude Android Development Tools (ADT) plugin for Eclipse from the list; Android Studio is considered to be the *official* tool for Android developers

1.4 Xcode

Xcode is only supported on computers running an up-to-date version of Apple's macOS operating system[4] The software can be freely downloaded via the MacStore app that is preloaded on every macOS installation. Developers can use Xcode to develop and run/debug apps on simulators or physical iOS devices. However, to publish apps on the App Store, developers have to pay an annual $99 developer fee.

Xcode supports iOS development in both Swift and Objective-C. The Swift language (currently version 5) was introduced by Apple in 2014 and has since been open-sourced. Objective-C, a derivative of the stalwart C programming language enhanced with object-oriented syntactical sugar, has long been Apple's flagship programming language. Both languages are currently supported by Xcode, and developers can mix and match modules written in both languages within a single project. While our focus will be entirely on Swift, serious iOS developers should also expect to become somewhat familiar with Objective-C given the tremendous amount of iOS code that is written, and continues to be written in Objective-C. Happily, Apple designed Swift in a way that makes it easy for developers to "port" their knowledge of the underlying iOS frameworks to either language. The reasonable ease in which developers can move between Objective-C and Swift is in fact one of the reasons Apple introduced an entirely new programming language (a rather audacious move, given the vast number of awesome languages available today, and the mindshare around Objective-C) as opposed to adopting an existing language such as Ruby.

The Xcode IDE is a flexible IDE with support for the typical features one would expect in a modern IDE. Figure 1.2 highlights the various functional panes that the IDE window is divided into when editing code. While the exact configuration of the IDE can be customized, and different situations might automatically reconfigure the layout, the general areas highlighted in this figure more or less represent the key areas in which you will spend your time as an iOS developer. Let's briefly describe each.

- **Toolbar (1):** The toolbar displayed at the top of the IDE provides you buttons for starting / stopping the simulator, the status of an ongoing build, and a variety of buttons that allow you to display / hide various panes within the IDE.
- **Nav Area (2):** The navigation area allows us to navigate through the various project artifacts that our app is built from, including source code, image asset catalogs, view layouts, etc. The nav area is actually populated by a tab like view that also provides us information on unit test outcomes, build errors, and the run-time stack and break points during debug sessions.
- **Editor (3):** The editor area takes the bulk of the Xcode window. If you are writing or editing code, the editor pane will will be filled with a code editor that supports syntax highlighting, auto completion, etc. If you are working on the user interface of an app, the editor pane will be filled with an interactive scene editor (named Interface Builder) that allows us to construct a user interface in a visual drag and drop manner.
- **Utility Area (4):** The utility area is a tabbed view of various inspectors. What gets displayed by these inspectors is entirely dependent on what is currently selected within the editor pane. The inspectors let you manipulate attributes of items selected in the editor, such as UI controls within a view. You can also manipulate connections between view artifacts and code, layout constraints, etc.

[4] At the time of writing, Xcode 11.6 is available on macOS 10.15.2 or later.

- **Debug Area (5):** The debug area gives the developer interactive access to the internal state of a running program when running in debug mode. The Xcode debugger provides you with a structured view of your app's current context in the pane on the left, and a command line console like input/out interface on the right.

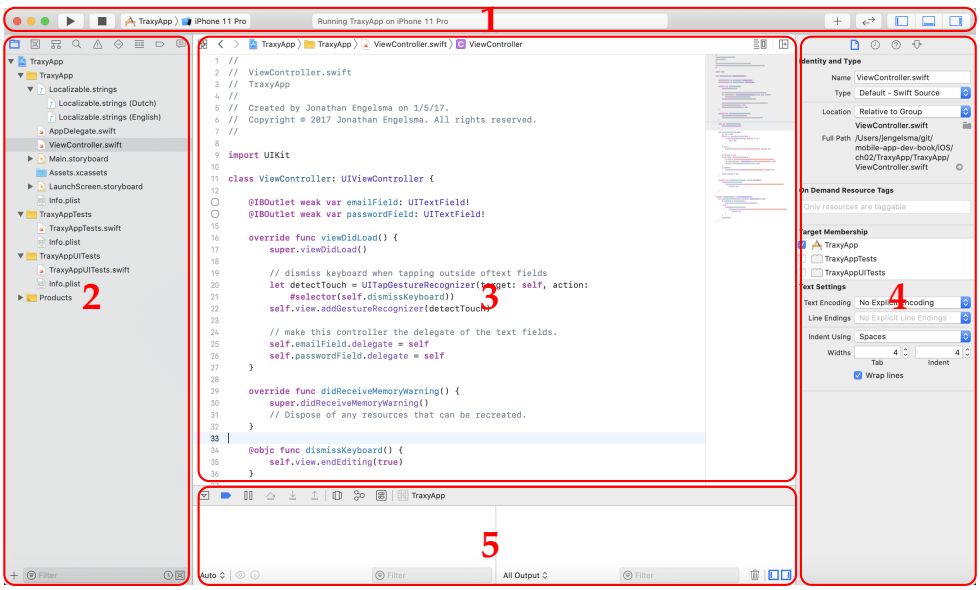


Figure 1.2: The Xcode integrated development environment.

For users familiar with earlier versions of Xcode, one particular confusing change is the location of the library window. Up until Xcode 10.x the library pane always displayed on the lower half of the utility area (4). Beginning in Xcode 10.x, the library pane is implemented as a separate window that can be displayed via the button labeled with a plus on the toolbar (1) or the key sequence ⇧+⌘+L. The content of the library window is determined by what content is open in the editor (3). If you are editing a user interface via Interface Builder, it provides you with a palette of pre-defined user interface controls that you can drag and drop into the Interface Builder editor. If you are editing source code, the library window displays reusable code snippets, file templates, and media components.

1.4.1 Interface Builder

iOS developers have a couple of different options for implementing user interfaces. The most common approach would be to use Xcode's Interface Builder. Interface Builder is a visual scene editor that allows the construction of user interfaces in a drag and drop fashion, as shown in Figure 1.3. Interface Builder stores the view hierarchies constructed in an Apple proprietary XML format. There are two different types of Interface Builder files:

- **.xib files*: Pronounced as "nib" files for historical reasons, these files contain a single scene or view hierarchy. This format predates the storyboard format introduced by Apple in 2011.

- **.storyboard files*: Storyboard files can contain multiple scenes as well as the transitions (also known as *segues*) among them.

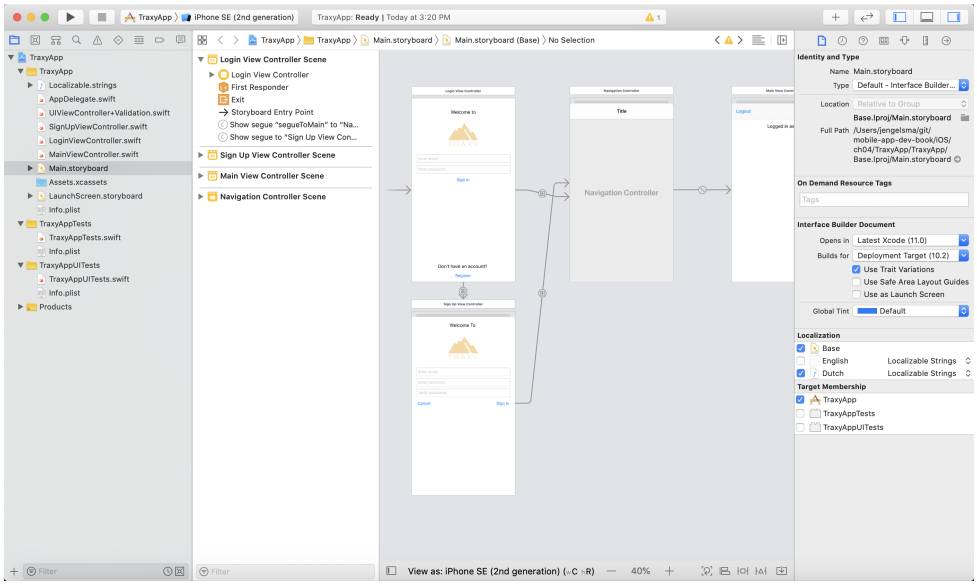


Figure 1.3: The Xcode integrated development environment.

By default, newly created projects in Xcode utilize storyboard files. Using storyboards has some advantages in that it gives the developer a high level view of the application in terms of its various screens and the flow among them. Furthermore, storyboards relieve the developer of some of the tasks previously required in code. For example, the loading and transitioning to a new screen can be accomplished by simply adding a segue in Interface Builder from the source to destination scene. When using xib files, the programmer is responsible for loading and effecting the transition in code. Despite the advantages, some developers continue to use the original xib files even when authoring new applications without a large legacy code base. The reason they do this is that often when working on a project with multiple developers, conflicting simultaneous updates are made independently to the same storyboard. When that occurs, these updates are very difficult to merge, due to the proprietary (and constantly evolving) nature of the XML format used for storing storyboards. If all of the views used within an app are contained in a single or small number of storyboard files, this situation could arise frequently, unless developers come up with some means to better coordinate updates to the storyboard files. For this reason, some teams opt to use xib files where each individual screen layout in the app is self-contained in it's own xib file. While conflicting updates can still occur with this approach, they are less likely as individual developers will often have ownership of a subset of the app in which they focus their development activities.

Still other apps avoid using Interface Builder altogether, as it is possible to construct an application's views entirely in code. This approach will be more common when the app implements an entirely custom user interface and does not utilize the stock UI controls provided by Apple and made available within Interface Builder. This approach would be common in apps that involve

a significant amount of custom graphics such as video games, but is also used by a number of popular apps with highly customized user interfaces.

1.4.2 Integrated Debugger and Simulators

Xcode includes a sophisticated debugger and set of iOS device simulators. Apple's LLDB debugger, a more recent replacement of the gdb used in older versions of Xcode, is tightly integrated within the IDE, and capable of debugging apps via simulated iOS devices or debugging the apps onboard actual physical devices. Figure 1.4 shows Xcode in debug mode along with an attached desktop iOS simulator. Simulators that run on the Mac desktop are available for a variety of Apple products, including various models of the iPhone, iPad, Apple Watch and Apple TV product families. In addition, they are available in multiple versions of each of the respective operating systems, so developers can do a fairly reasonable job of running their code on a simulator of any device an end user might have, without actually having physical possession of the device. That said, there are a number of device features that are not supported by the simulators and require a physical device for complete testing. For example, the iPhone's camera and Bluetooth functionality is not supported by the simulator. Testing an app that utilizes these features would need to be tested on a physical device.

When initially installing Xcode, the complete set of simulated devices are not downloaded by default. Developers can browse and download the available devices via the Xcode application menu (Xcode 〉 Preferences... and on the dialog window that is displayed, click on the Components tab.)

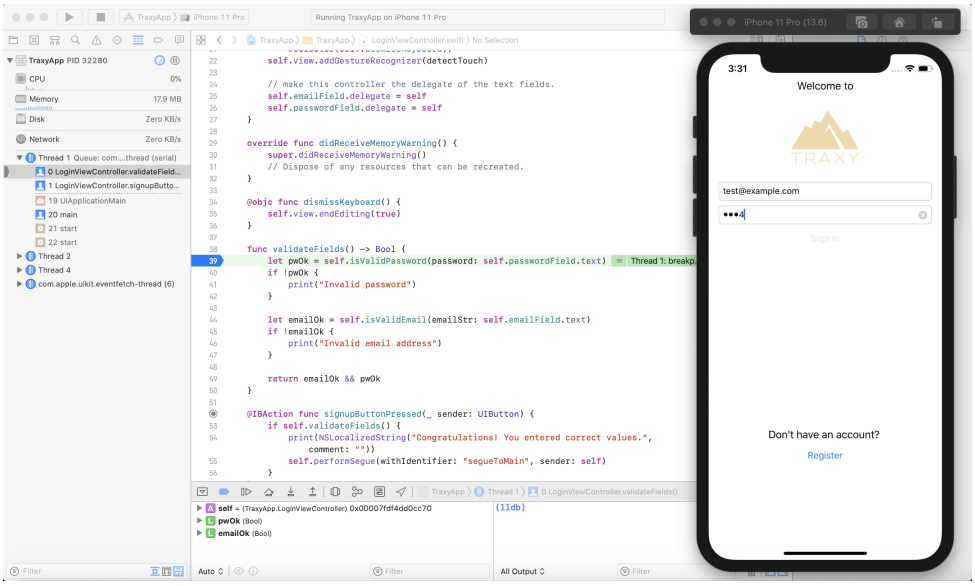


Figure 1.4: Xcode in debug mode attached to a simulator running an app.

To debug an app on a physical device, such as an iPhone, the developer must register as a developer (free account or paid) via Apple's developer site (https://developer.apple.com) and

connect the device to the Mac using the USB cable that came with it. The developer must also follow Apple's well-documented instructions to obtain a signing certificate, provisioning profile, etc. Once registered, a developer can provision his account with Xcode (Xcode ⟩ Preferences...) and then on the dialog window that is displayed click on Accounts) and much of the hassle (provoked by Apple's autocratic tendencies) is automated under the hood by Xcode.

1.4.3 Tools and Frameworks for Testing

The XCTest framework is a unit test framework integrated within Xcode. As of Xcode 7 XCTest also support user interface and performance testing as well. Other test related tooling include Code Coverage which allows you to visually explore which blocks of code in your source have been invoked by unit tests, as well as reports that provide a bird's eye view of your test coverage.

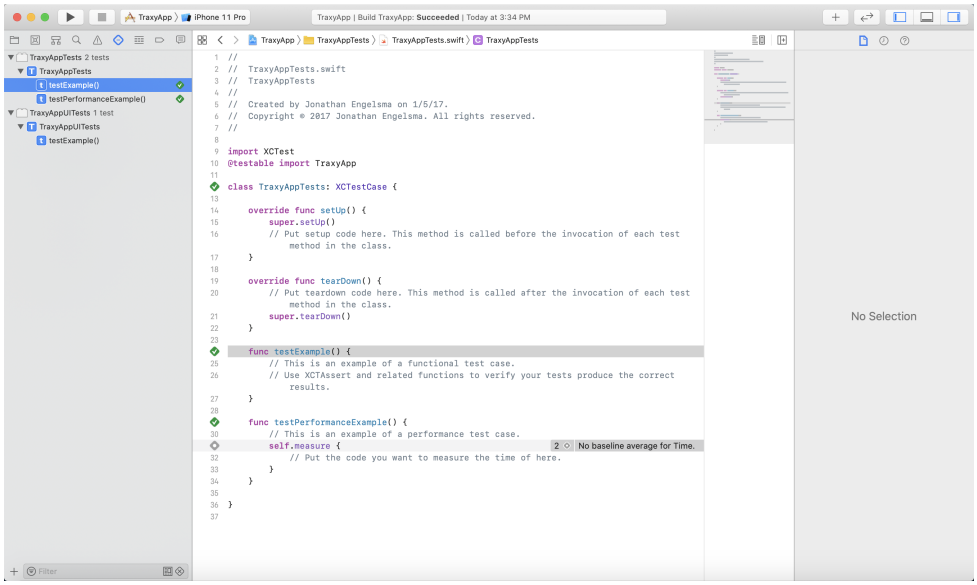


Figure 1.5: The XCTest framework is integrated within Xcode and facilitates automating unit and user interface testing.

1.4.4 Xcode Documentation

Apple provides an extensive set of documentation for its iOS frameworks and tools in the form of developer guides and API reference documentation. The complete set can be accessed within Xcode via the application menu Window ⟩ Documentation and API Reference or alternately with keyboard shortcut ⇧ + ⌘ + 0. Reference documentation can also be retrieved contextually in a "quick help" window when editing code by moving the pointer over any program identifier holding ⌥ and clicking with the mouse. The documentation can be downloaded for off-line reference via the application menu (Xcode ⟩ Preferences ... and select Documentation on the Components

tab.). Apple has done an admirable job of providing a comprehensive and readable documentation set, and you will do well to acquaint and avail yourself of it early on as you ascend the iOS learning curve. At the same time, you'll often find yourself out on stackoverflow.com researching specific issues or questions that might arise.

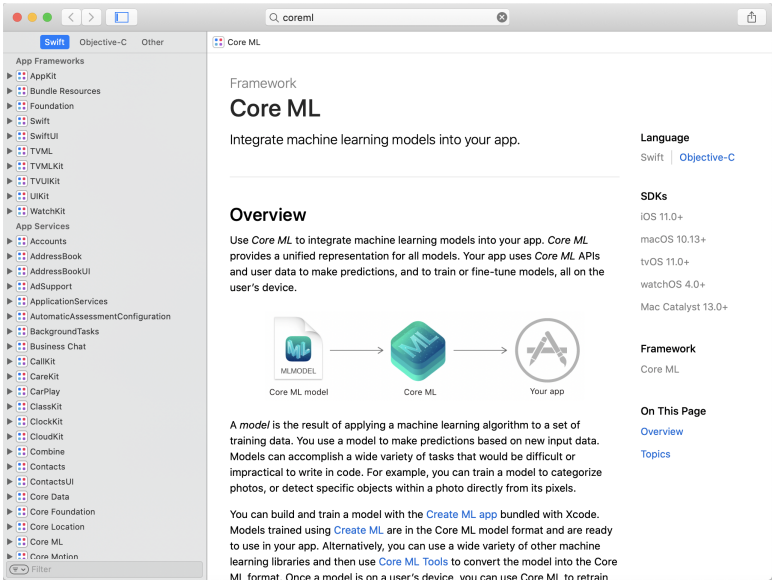


Figure 1.6: The Xcode Document and API Reference screen.

1.4.5 Managing Third Party Components with CocoaPods

A vibrant open-source community has coalesced around the iOS platform, offering everything from components that simplify network programming to beautiful custom user interface controls. Most production apps today, will incorporate a significant number of open-source components.

Originally inspired by Ruby gems, CocoaPods (https://cocoapods.org) has emerged as the de facto industry standard for dependency management of Swift and Objective-C projects. Referred to as *pods*, these components can be specified within a configuration file (*Podfile*) and automatically downloaded and integrated into your app's build process.

Today there are thousands of Objective-C and Swift libraries available in the form of pods, coming from individual open-source developers as well as from large companies such as Google and Facebook.

Unlike gradle, the nearest analog to CocoaPods in the Android world, which manages dependencies as well as the build process, CocoaPods only addresses external dependency management. The actual build process is still managed and implemented by Xcode.

Let's turn to Android Studio now and briefly tour the key features in Google's state-of-the-art IDE.

1.5 Android Studio

Android Studio is a multi-platform IDE that runs on the three major operating systems: Windows, Linux, and macOS. It is available to developers as a free download. The IDE supports the development, execution and debugging of Android applications on simulators and physical Android devices. While the IDE is available for free, in order to publish an app on Google's Play Store, developers have to pay a one time registration fee of $25.

Android Studio is powered by the IntelliJ platform whose architecture clearly separates *program editing* from *program building*[5]. This design decision enables JetBrains (the company that develops IntelliJ) to reuse its editor front-end to create other IDEs (PyCharm, CLion, WebStorm, etc.). For instance, while reusing most of same editor front-end, CLion builds C/C++ programs by incorporating CMake as its build system.

The flexibility and extensibility of the IntelliJ design are among the main criteria that made the Google Android Developer Tool team adopt the IntelliJ core framework for building Android Studio. As a consequence, users with prior experience using IntelliJ will find a very familiar user interface when adopting Android Studio. In addition, IntelliJ users who have favorite plugins (e.g. add-on components to customize the IDE) can carry them over to Android Studio.

The Android Developer Tool team had several options for their choice of program builder modules to use in Android Studio. To complement IntelliJ's core framework extensibility and flexibility, the team was looking for a build system that was also extensible and flexible. They ultimately settled on Gradle as Android Studio build system. The Android Developer Tool team has so much confidence in Gradle that they guarantee the same build results regardless how Gradle was invoked. Whether it runs from the IDE, command line, or development server, in all cases, Gradle delivers the same build results.

Since its first release at Google I/O 2013 (Google's annual developer conference), Android Studio has improved significantly and it is now available as an open-source IDE. Productivity enhancing features found in IntelliJ, such as: code refactoring, auto intention, smart completion, automatically carry over to Android Studio.

Multi-language support is another strong feature inherited from IntelliJ. Developers who need to interface Java code with C or C++ can easily do so in Android Studio. NDK support in Android Studio has improved significantly as well: JNI header editing, compiling, and debugging are seamlessly integrated into Android Studio. More recently, a Kotlin plugin for IntelliJ enables Java and Android developers to write code in Kotlin, a more recent language designed by JetBrains.

The overall layout of the Android Studio window is very similar to other IDEs, where the top menu/tool bar holds commonly-used buttons like open file, save file, copy, cut, paste, run, debug, etc. If you have prior experience with other IDEs, it will not be difficult to guess the function of most toolbar icons. Visual elements that may be unique to IntelliJ/Android Studio are the collapsible panels. Android Studio workspace is organized into several *collapsible panels*. On the left and right, you find a vertical strip that holds vertical tabs of collapsible panels. Similarly, the bottom screen shows a horizontal strip that holds horizontal tabs of collapsible panels. Figure 1.7 shows the workspace with the Editor panel at the center (labeled 1) and three currently expanded (labeled 2, 3, and 4) collapsible panels.

[5] We intentionally use "program building" over "compiling" here, because building a deployable Android application is more than just compiling its source code.

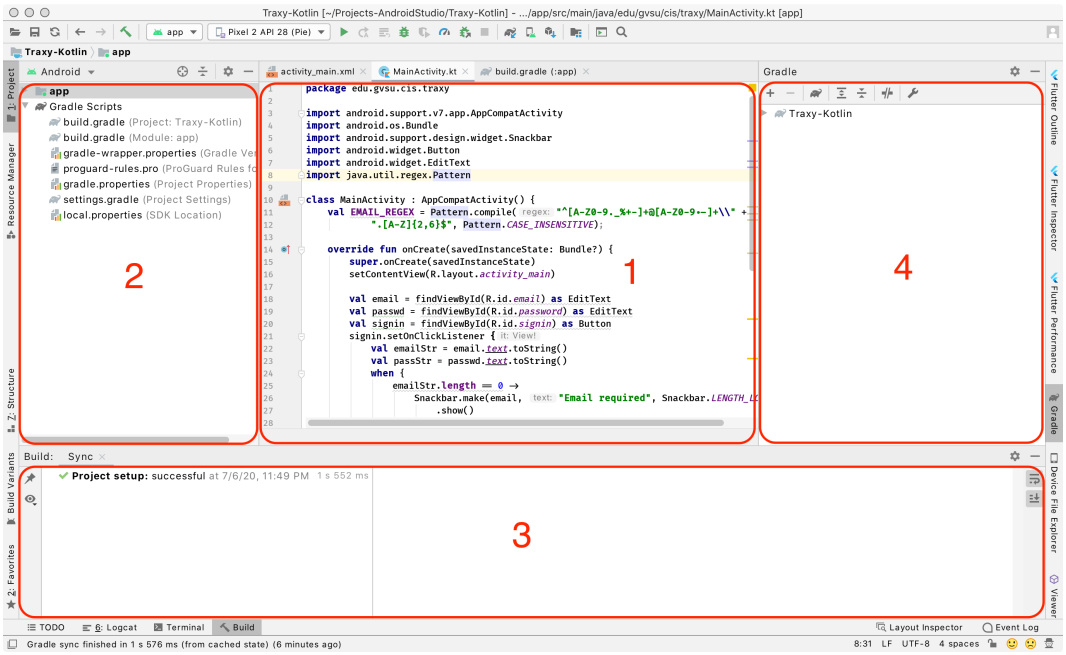


Figure 1.7: The Android Studio IDE.

> **NOTE**
>
> Each tab is labeled with a single-digit number to the left such as: 4:Run, 5:Debug, etc., on the bottom gutter, that tells you the keyboard shortcut to open the tab. For instance, ⌘ + 5 on macOS or ⌥ + 5 on Windows/Linux opens the Debug tab.

Just above the project panel (2) you will find a pull-down menu to select different layouts of the project panel. In Figure 1.7, the current selection is "Android". Other selections of interest include: Project, Packages, and Production.

The upper right corner of each collapsible panel shows an icon with an arrow pointing to the collapse side. For instance, a panel with the icon — indicates it collapses to the left gutter/vertical strip. The tabs on left, bottom, and right gutters of the workspace show the name of panels collapsible to that corresponding side. For instance, the figure shows that the following eight panels are collapsible to the bottom: Debug, TODO, Messages, Version Control, Terminal, Android Monitor, Event Log, and Gradle Console. To show/hide any of these collapsible panels:

- Click the upper right collapse/hide icon to collapse the panel
- Click the tab on the gutter to expand a collapsed panel

While developing Android apps, you will be using the following panels most of the time:

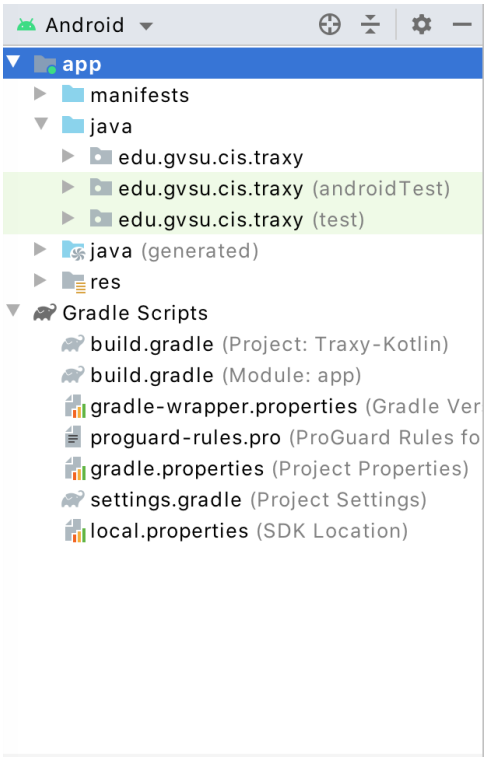


Figure 1.8: Android Studio project structure.

- **Run/Debug**: debugger control (step over, step into, step out, etc.), variable inspector, expression watcher and other debug related features.
- **Version Control**: source code version control using a particular utility (git, CVS, Mercurial, etc.).
- **Android Monitor**: Android LogCat, Monitor CPU/Memory/Network/GPU activity, etc.

After a new Android project is initialized and created, the **Project** (collapsible) panel shows how various components of the project are organized. The tree structure inside this panel is normally shown with all its nodes collapsed. After expanding some of these nodes, you will see more details as shown in Figure 1.8.

- The **manifests** directory contains an XML file (`AndroidManifest.xml`) that describes the various components and permissions of the application.
- The **java** directory holds all the Kotlin (and Java) source files. These files are further organized into subpackages.
- The **res/drawable** directory holds all the images (bitmap, shape drawables, gradient backgrounds, icons, state list drawables, and vector assets)

- The **res/layout** directory holds user interface layout descriptions. These are XML definitions of the view hierarchies of various screens displayed by the app.
- The **res/menu** directory holds the menu structure definitions.
- The **res/values** directory holds key-value pairs of several resource types: colors, strings, and styles.

An Android project may include additional resource subdirectories besides the ones shown in the figure. For instance, animations can be defined declaratively in XML files under **res/anim**. As you dig deeper into Android development, you will encounter other types of resources used by Android apps.

1.5.1 Layout Editor

Android Studio supports a visual editor referred to as the *layout editor* that is somewhat analogous to Xcode's Interface Builder. Using the layout editor, screens can be designed by dragging Android widgets from the layout editor's widget palette and onto the screen. Android supports a variety of layout managers, including the most recent addition, the ConstraintLayout.

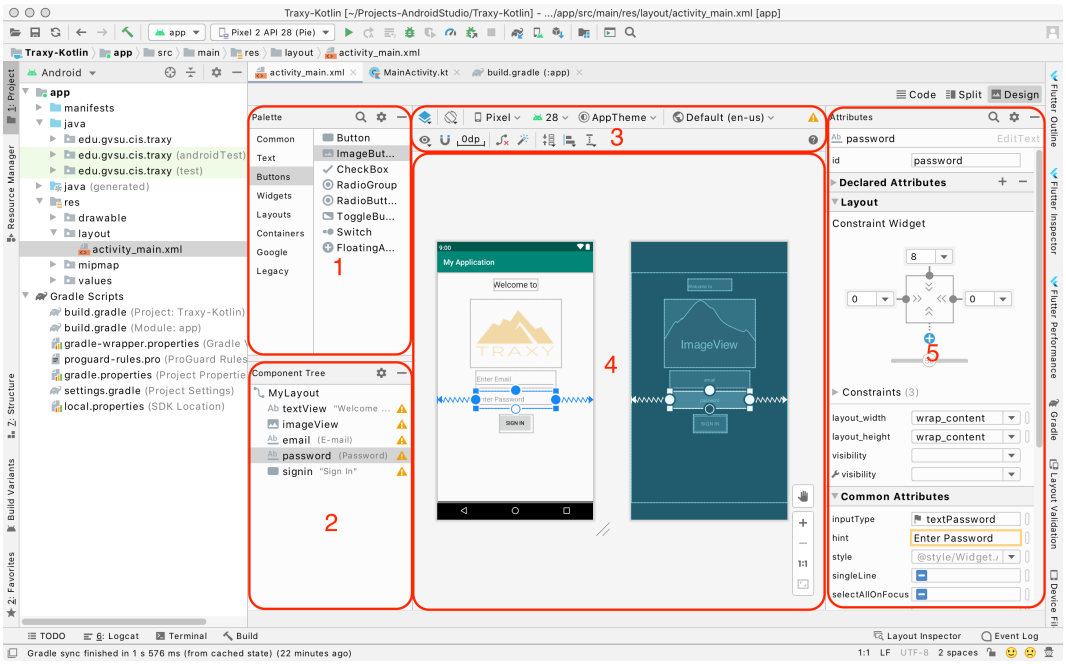


Figure 1.9: Android Studio's Layout Editor.

Figure 2.36 shows the layout editor. With reference to the five annotated areas of the figure, widgets can be dragged from the palette (1) onto the design editor (4). The design editor shows both the design as well as blueprint view. The component view(2) shows the view hierarchy of the layout. The toolbar(3) above the design editor lets your configure the appearance of the layout.

For example, you could modify the selected theme, or envision what the layout will look like on a particular device. The properties panel (5) shows the property editor of the currently selected widget.

NOTE

The Android documentation uses the term *widget* to refer to user interface components. In this book, when discussing Android, we will use the term widget in the same general way. Apple documentation however, does not use the term in this way, but typically uses terms like *controls* (e.g. low level user interface components like buttons and text fields) and *views* (e.g. containers of views/controls). When discussing iOS, we will also follow suit, and use terms like controls/views instead of widgets. We believe this approach will help the reader develop a vocabulary for each platform that is relevant to the respective communities that have formed around them. In case you don't find this confusing enough, also be aware that both Android and iOS also use the term *widget* to refer to chunks or extensions of user interfaces that can be displayed in very specific situations (e.g. for example, on the device's idle screen). Moving forward we will make it abundantly clear when we mean *widget* in this sense of the word!

1.5.2 Gradle

Prior to the release of Android Studio, Android developers would use either Eclipse or IntelliJ and had to depend on the Ant build system to build the app. Using a more powerful plugin mechanism provided by the Gradle build system, Android developers can conveniently use just one IDE to develop Android apps, even if the app incorporates functionalities from non-Android modules.

Gradle manages such heterogeneous apps by organizing a project into modules. Each module typically requires a separate plugin. All the build settings of a gradle module are specified in a `build.gradle` file. For simple projects you are developing here, Android Studio creates your projects with two `gradle.build` files, which can be found under the **Gradle Scripts** node in the Project panel (left collapsible panel). In Figure 1.8 each `build.gradle` file is tagged with its scope: "(Project: Traxy)" and "(Module: app)". Listings 1 and 2 show an example of `build.gradle` files for a typical Android project.

> NOTE
>
> The code sample throughout the book should be compiled using Kotlin 1.4 (or newer).

For Android beginners, keeping up the build environment up-to-date can be a daunting task; they have to keep the following three interrelated components in sync with each other:

- The Gradle Plugin version is set in the `build.gradle` file of the project itself (the parent of the Android module). The sample setup in Listing 1 uses Android Gradle Plugin 4.0.0. If the version is too old compared to the SDK you are using, Android Studio will complain with an error message. The most common fix to this problem is update the build tool to the latest release.
- The Gradle executable plugin is set in another text file `gradle-wrapper.properties`. The sample setup in Listing 3 refers to Gradle Version 6.1.1. When opening an existing project after upgrading to a newer Android Studio release, you may also be prompted to upgrade the Gradle binary to a specific version. This additional step is required because newer Android Plugin for Gradle may require features only available in newer Gradle binary executable.

Gradle is both a program builder and a dependency manager. This dual role of Gradle lessens the number of third-party installs for Android developers. As we have seen previously, the lack of a dependency management feature in Xcode has resulted in the emergence of the CocoaPods dependency management utility.

The `android` clause at lines 7–23 of Listing 2 sets all the Android build parameters for the project. This section includes definitions for the SDK level settings: minimum SDK, compile SDK, and target SDK, etc. The `dependencies` class at lines 25–34 specifies all the third-party library that your app depends on. When any of these libraries is missing from your local computer, Gradle will first download them (and their dependencies) to your computer. The `implementation` instructions specify libraries required for building the "regular" executable, while the `testImplementation` instruction specifies libraries required for building the "test" variant of your app.

```
// Top-level build file where you can add configuration options common to all
// sub-projects/modules.

buildscript {
    ext.kotlin_version = '1.4.10'
    repositories {
        google()
        jcenter()

    }
    dependencies {
        classpath 'com.android.tools.build:gradle:4.0.0'
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
        // NOTE: Do not place your application dependencies here; they belong
        // in the individual module build.gradle files
    }
}

allprojects {
    repositories {
        google()
        jcenter()

    }
}

task clean(type: Delete) {
    delete rootProject.buildDir
}
```

Listing 1: `build.gradle` file (root project)

Running Gradle from Command Line

At the beginning of this chapter, we discouraged you from performing tasks from the command line. However, if you insist to do so, gradle will comply. A typical use case is when you want to git-clone an example Android project and run it. Provided that the sample project uses Gradle, the following five commands will do the job:

```
mkdir sample1
cd sample1
git clone https://path/to/the/cloned/project
cd to/the/subdirectory/of/the/android/app
../gradlew installDebug
```

Typing the above five commands is much easier than running the IDE to edit the source files, compile the code, and install the APK (Android Package).

1.5.3 Managing SDKs

Apple's sole control over the entire iOS ecosystem facilitates speedier deployment of iOS upgrades to user devices. The circumstance on the Android ecosystem is entirely opposite of Apple's. An-

```
apply plugin: 'com.android.application'

apply plugin: 'kotlin-android'

apply plugin: 'kotlin-android-extensions'

android {
    compileSdkVersion 28
    defaultConfig {
        applicationId "edu.gvsu.cis.traxy"
        minSdkVersion 17
        targetSdkVersion 28
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'),
                'proguard-rules.pro'
        }
    }
}

dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version"
    implementation 'com.android.support:appcompat-v7:28.0.0'
    implementation 'com.android.support.constraint:constraint-layout:1.1.3'
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'com.android.support.test:runner:1.0.2'
    androidTestImplementation 'com.android.support.test.espresso:espresso-core:3.0.2'
    implementation 'com.android.support:design:28.0.0'
}
```

Listing 2: `build.gradle` file (Android module)

droid devices are manufactured by numerous manufacturers. To increase market appeal, every manufacture delivers a customized version of Android on their devices. When Google releases an SDK update, these manufactures may not immediately follow suit to upgrade their custom Android. Investing in the development of the next flagship hardware product takes a higher priority than a software upgrade for a legacy product. Thus, compared to iOS adoption rate of the latest iOS release, the adoption rate of new Android releases among Android users always lags behind.

It is important that Android developers consider this fact when deciding the lowest SDK level supported by their app. Essentially, you must weigh the trade off between your app's availability to a larger user base versus incorporating the latest Android features into your app. Lowering the minimum SDK level opens up your app to users who run older Android API levels on their phones, thus widening your user base. On the contrary, lowering the minimum SDK imposes some restrictions in incorporating newer features into your app.

To handle the above issue, an Android app includes the following three SDK related declarations — XML attributes which are ultimately inserted/updated into in the app manifest file by

Gradle

```
# in gradle-wrapper.properties
distributionUrl=https\://services.gradle.org/distributions/gradle-6.1.1-all.zip
```

Listing 3: Gradle Binary Executable

Android Studio. Thanks to the Gradle auto synchronization feature of Android Studio, we do not have to set these attributes manually into the manifest file. Nevertheless, settings these attributes correctly is very important:

- minSdkVersion: the API level specified here is used by Google Play Store to determine whether the app is available (installable) to a particular user device. Hence this attribute applies at *deploy time*.

- compileSdkVersion: the API level used by Android Studio to build your app during development. Using the latest SDK available allows your Android program to include the latest and greatest APIs added in that particular level. Thus, if you try to use a method that was introduces in API 19 but you attempt to compile using SDK level 18 (or lower), you will get a compile error. **It is recommended that we always compile using the latest SDK**. Hence this attribute applies at *compile time*

- targetSdkVersion: this attribute is the least understood by Android developers. This attribute affects the behavior of your app at **run-time**. Let's say that a particular feature has been upgraded from its prior behavior in API 17 to its newer behavior in API 19. Specifying **targetSdkVersion** 17 your app demonstrate the prior behavior, but updating targetSdkVersion to 19 your user will observe the newer behavior. So, this attribute applies at *run-time*.

To learn and read more details about these three attributes, we refer you to a Medium article by Ian Lake [16].

Refer to Listing 2, and observe that the **compileSdkVersion** (line 8) is specified *immediately* under the `android` clause while the other two are inside the `defaultConfig` clause (the default **build variant**). Gradle supports several build variants; each variant builds one APK. A common practical use of multiple build variants is, for instance, you want to deploy both the "demoware" and "premium" variants of your app.

Various versions of iOS SDKs are known to users/developers by their version numbers only. The Android SDKs are identified using three different aliases:

- Numeric version numbers.

- Codenames (that follow alphabetical order).

- API levels.

The following table shows how the three aliases identify various SDK versions:

Newer SDKs can be manually installed from Android Studio from its **Tools** menu: Tools > SDK Manager > SDK Platforms/SDK Tools. Selecting this menu brings up the SDK manager dialog (Figure 1.10).

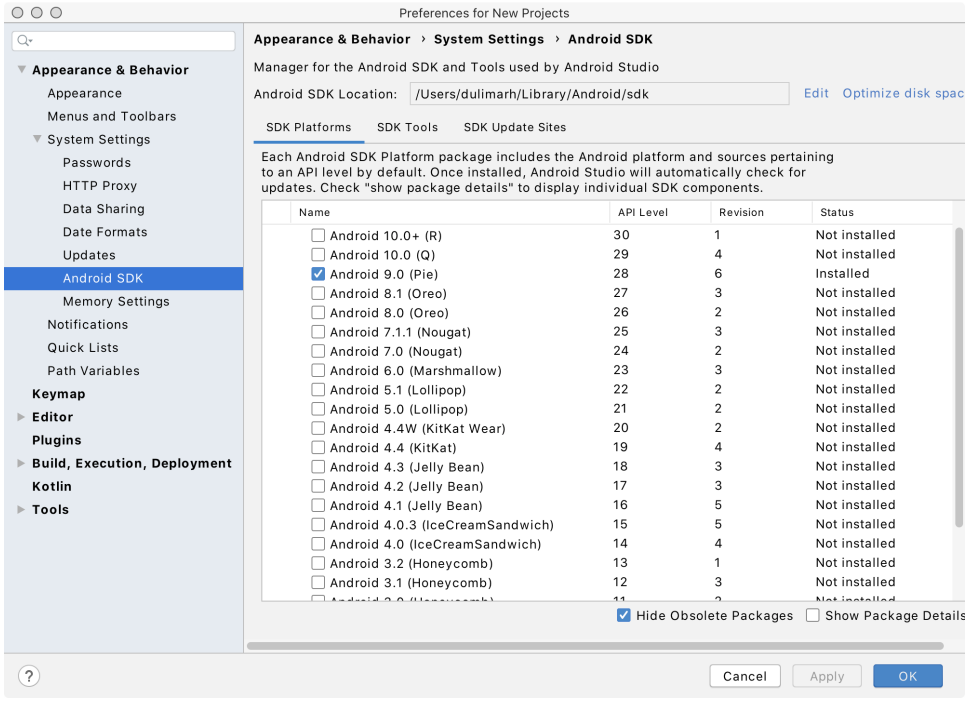


Figure 1.10: Android Studio SDK Manager

Version	Codename	API Level	Release Date	Key Features
1.5	Cupcake	3	Apr 2009	Virtual Keyboard, Clipboard
1.6	Donut	4	Sep 2009	Multiple screen densities
2.0–2.1	Eclair	5–7	Oct 2009	Speech recognition
2.2	Froyo	8	May 2010	Voice actions
2.3	Gingerbread	9–10	Dec 2010	
3.0–3.2	Honeycomb	11–13	Feb 2011	Tablet support
4.0	Ice Cream Sandwich	14–15	Oct 2011	Gestures
4.1–4.3	Jelly Bean	16–18	Jul 2012	Project Butter: Improved animations
4.4	Kit Kat	19–20	Oct 2013	Project Svelte: memory optimization
5.0–5.1	Lollipop	21–22	Nov 2014	Material Design
6.0	Marshmallow	23	Oct 2015	Runtime permission
7.x	Nougat	24–25	Aug 2016	Virtual Reality
8.0	Oreo	26–27	Aug 2017	Project Treble:
9.0	Pie	28	Aug 2018	AI
10	Android 10	29	Sep 2019	Dark Mode and improved security
11	Android 11	30	Sep 2020	Conversations and predictive tools

1.5.4 Android Emulators

What Apple calls "simulators", Google calls "emulators". They provide essentially the same service: a virtual device that allows developers to run the app they are developing without a physical device on hand. Why do we bother using emulators when the average price of Android phones is very affordable, and often significantly less than the those of iOS devices?

The competition among Android device manufacturers is actually beneficial to the consumer. There are many inexpensive phones to choose from. However, most of us would not consider owning more than one phones at a time. Nevertheless, occasionally we may want to see how our app runs on a device different from what we have in our physical possession. Instead of spending money on more hardware, you can create these devices virtually and emulate them with software.

The performance of emulators has improved significantly since Google I/O 2016. At that conference, the Android Developer Tools Team was able to demonstrate speedy app deployment on Android emulators. In fact, it took less time to deploy to the emulator than deploying the app to a physical device.

A new virtual device can be created from the menu Tools » AVD Manager. If no virtual devices exist, the first time you visit the AVD Manager you will see the dialog shown in Figure 1.11. Thereafter, you will be prompted with several dialog screens:

1. Since Android supports several form factors, the AVD configuration dialog (Figure 1.12) prompts you for one of the five categories (TV, Phone, Wear OS, Tablet, Automotive).

2. The system image dialog (Figure 1.13) is the most important step. Android emulators are es-

sentially a virtual machine with a Linux kernel running on top of it. If your current Android Studio installation is missing the required Linux kernel system image, it must be downloaded first.

3. In the final dialog (Figure 1.14) be sure that the AVD name contains no special characters. Otherwise, you will receive an error message on your next attempt to launch the AVD.

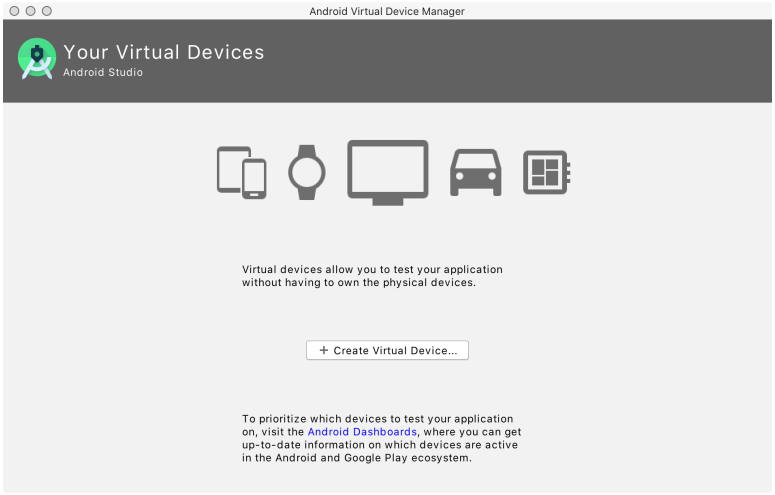


Figure 1.11: Creating a new Android Virtual Device (AVD).

Figure 1.15 shows the emulator screen of a Nexus S virtual device running Lollipop. Pressing the ··· button shows the extended control screen that lets you change the following settings:

- The GPS Location of the virtual device (a single GeoPoint or a series of GeoPoints in GPX/XML format).
- The type of cellular network connection (GSM, EDGE, GPRS, 3D, LTE, etc.).
- The battery charging level and status. This can be useful when you need to emulate a battery low warning.
- Incoming SMS or phone calls
- The navigation pad.
- The finger print sensor.
- The sensors: accelerometer, temperature, pressure, magnetic field, etc.

After discovering these extended configuration controls of Android emulators, we can add two more important reasons why emulators are advantageous: convenience and virtual sensors.

- When testing apps that require physical mobility while gathering geo location (GPS) data, you do not have to leave your desktop and drive your car around to test your app.

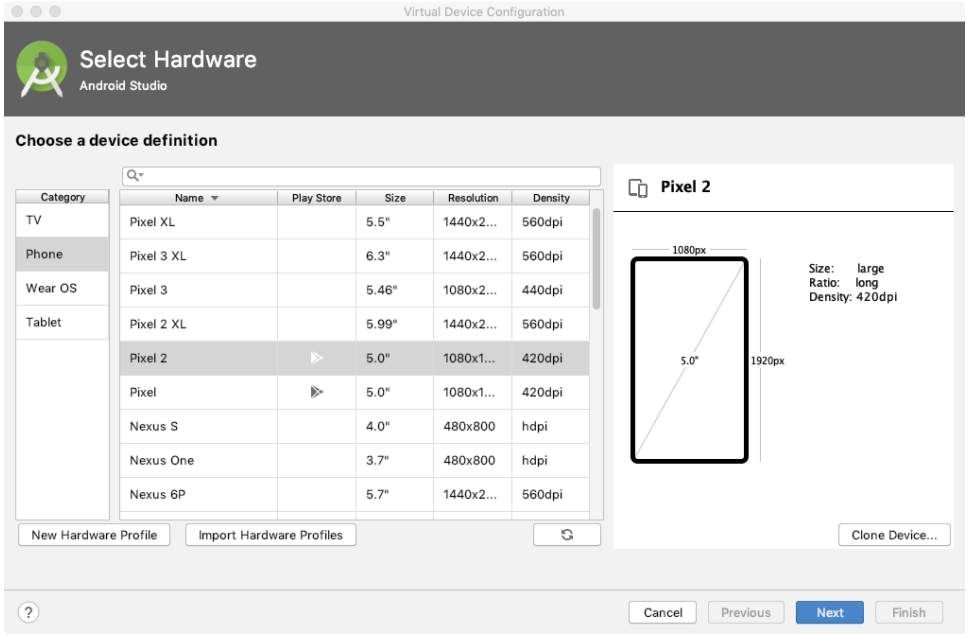


Figure 1.12: Selecting the AVD definition.

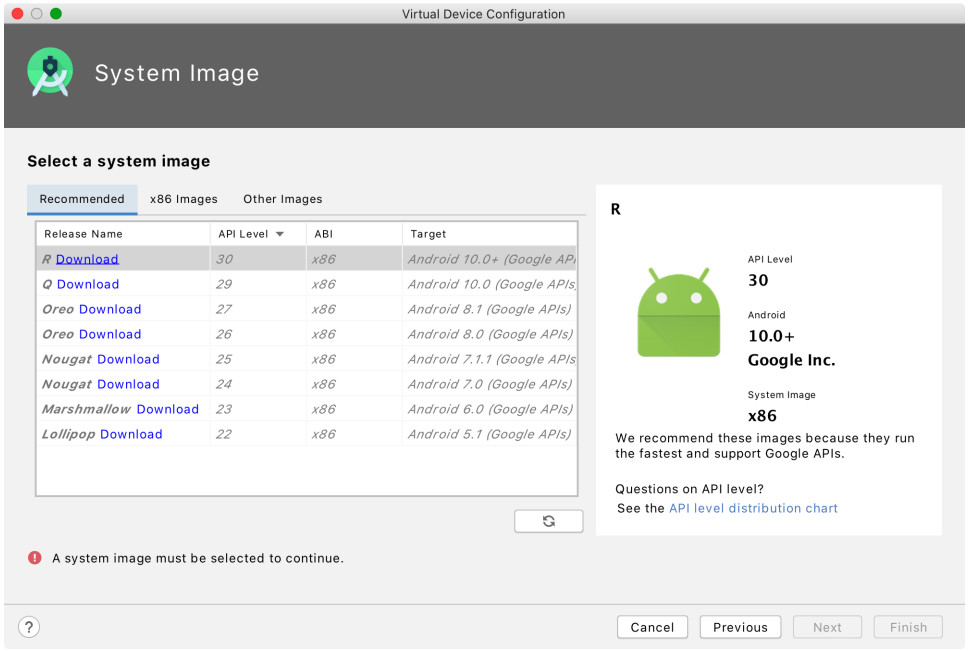


Figure 1.13: Selecting the AVD system image.

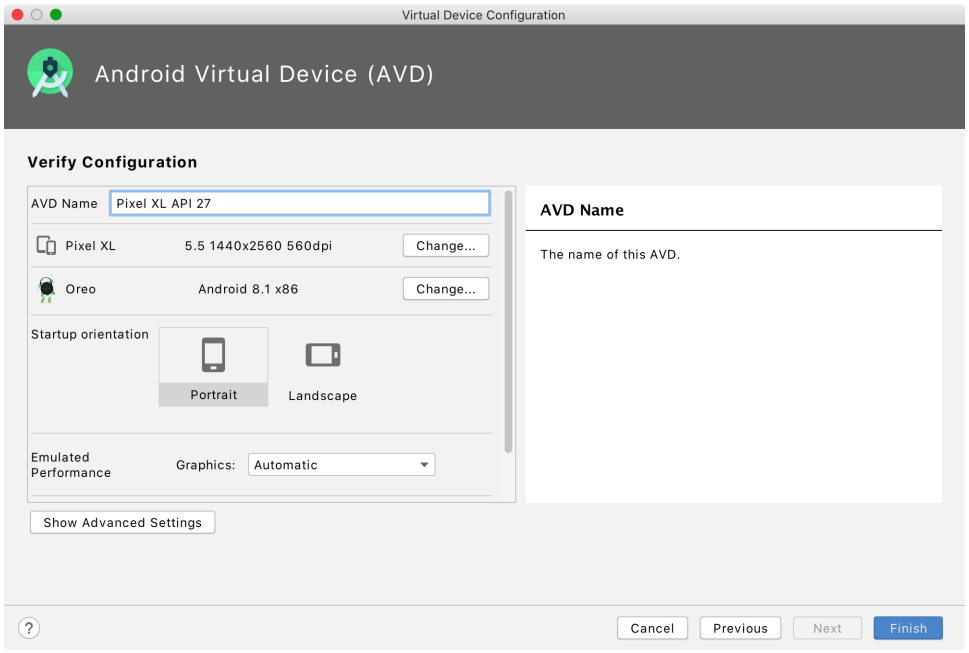


Figure 1.14: Configuring the AVD details.

- The emulator allows you to reconfigure your cellular connection type to a totally different type than what you currently on. For instance, you would like to observe the response of your data download routine on a 3G connection. Without emulators, you must physically be present inside a reception zone of a 3G tower.

- Virtual sensors: Android emulators offers several virtual sensors that do not necessarily exist on the current device you own, for example, barometric pressure sensor, temperature sensor, etc.

1.5.5 Test Framework and Debugger

A project created by Android Studio is automatically configured for test automation. In addition to the "main" package that holds the application logic of your app, Android Studio creates two additional packages designed specifically for testing. Assuming your "main" package is `edu.cis.gvsu.myapp`, the other two packages for the test code are:

- `edu.cis.gvsu.myapp.test`: test code that runs on your local JVM that requires no Android dependency.

- `edu.cis.gvsu.myapp.androidTest`: test code that requires an Android device to run. This is also known as **Instrumentation Test**.

More details on the test framework in Android Studio will be covered in a later chapter.

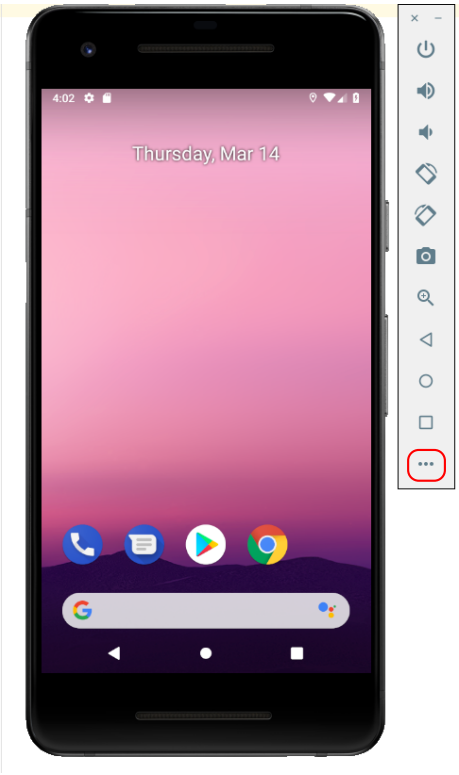


Figure 1.15: The Pixel 2 virtual device emulator.

Seasoned developers know how to use code debuggers by setting up breakpoints, using the debugger common toolbar icons (Step Over, Step Into, Step Out, Step Return), using expression watchers and variable inspections. In addition to these common tasks in a typical debugging sessions, Android Studio also provides features which may not be frequently used by most developers:

- Conditional breakpoints: a breakpoint that becomes active only when its associated boolean expression evaluates to true (at runtime). To attach a condition, right-click on the breakpoint marker on the left gutter of the editor to display the screen in Figure 1.16.
- Logged breakpoints: a breakpoint that also prints debugging output to Android Logcat (see Figure 1.16). This is a convenient way of logging messages without having to recompile code.
- Attach the debugger to an active process: a feature that is useful when we need to reinspect the state of our app at the time (or after) a bug occurs. Activate this feature from Run > Attach to Process.
- Smart Step Into (Run > Smart Step Into or ⇧+F7) is a variant of Step Into with an additional feature for stepping into a **selected method**, making the overall stepping process faster and

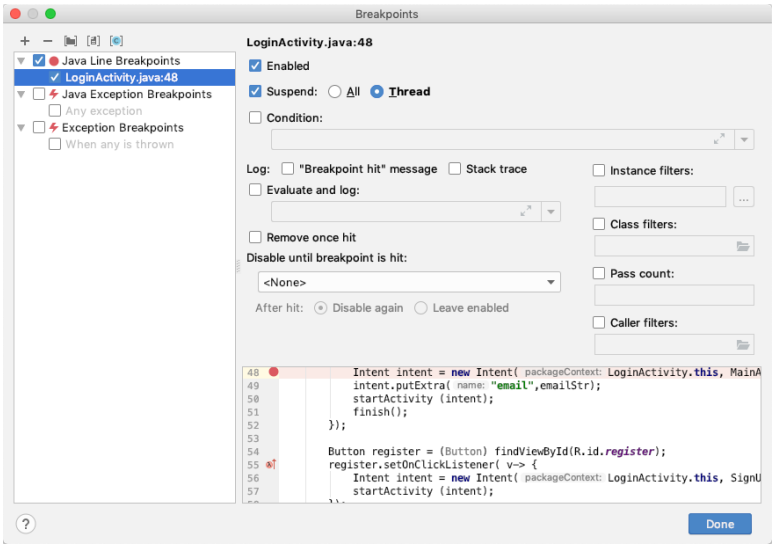


Figure 1.16: Debugger breakpoints

more convenient.

1.5.6 Managing Libraries

Libraries are now inseparable components in Android development. Most of these libraries enhance the features provided by the core Android framework. In the following sections we will look at the two library categories: those maintained by Google Android team and those maintained by third-party independent developers. The naming convention of these libraries follow the guidelines specified by the Apache Maven Project. The name consists of three parts: group id, artifact id, and version number. For instance: the library

```
com.android.support:design:28.0.0
```

its group id is `com.android.support`, artifact id is `design`, and currently at version 28.0.0

1.5.7 Android Support Libraries

The support libraries had been the main venue for the Android team to add new features while at the same time maintaining backwards compatibility with older API levels. The ”vx” infix (”v4”, ”v7”, ”v8”, etc.) in the library name indicates the lowest API level supported by the library.
For instance `com.android.support:cardview-v7:25.0.0` is a library with backwards compatibility to API Level 7.

Prior to API level 28, Android developers had to include one or more Android Support Libraries. These libraries are under the group id `com.android.support`; a partial list of these libraries are listed below:

- `com.android.support:support-compat`

- `com.android.support:support-fragment`
- `com.android.support:appcompat-v7`
- `com.android.support:cardview-v7`
- `com.android.support:gridlayout-v7`
- `com.android.support:palette-v7`
- `com.android.support:recyclerview-v7`
- `com.android.support:design`
- . . . and many more . . .

Since API level 28 the support libraries are now called AndroidX which is part of Jetpack. We will talk more about these libraries as we use them in later chapters.

1.5.8 Upgrading to Android Studio 4.0

Kotlin support began with Android 3.0 which was released in 2017. To help you transition from Java to Kotlin, Appendix B summarizes important Kotlin features you should understand to begin using Kotlin in an Android app.

When you open an existing project created with Android Studio prior to version 4.0, you may be prompted several options:

- Install missing SDK
- Upgrade gradle build tools and plugins
- Upgrade gradle binary executable and wrapper

The latest Android Studio (4.0) seems to be smart enough to perform necessarily update to the project files and your manual intervention as a developer is very minimal if you follow the migration steps suggested by Android Studio when you open a project created by older versions of Android Studio.

—You must take the first step. The first steps will take some effort, maybe pain. But after that, everything that has to be done is real-life movement.

Ben Stein

2 Writing Your First Mobile App

Chapter Objectives

After reading this chapter you should be able to

- Write your first "Hello World" style app.
- Write an app that supports interaction with the user.

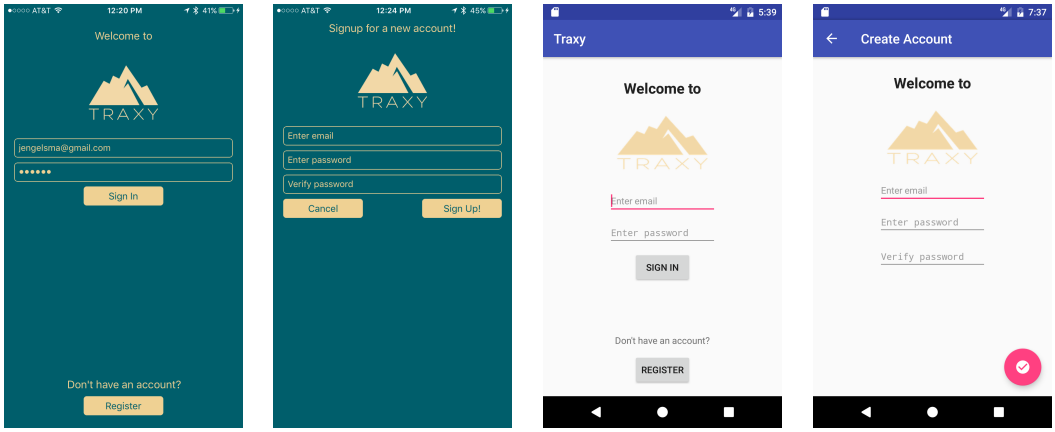
As you read your way through this book, you will incrementally learn the Android and iOS platforms side by side. Our goal is to help the reader understand the commonalities of these two mobile application platforms, as well as some of the areas in which they differ. The main vehicle that will serve to guide our learning of these two software platforms will be a vacation journaling application we have named Traxy. The Traxy app allows users to create a multimedia journal of their vacation experiences. Before we start start looking at implementation details, let's take a few moments and explore the functionality of the proposed Traxy app.

2.1 Traxy App Overview

On its main screen, the Traxy app will allow the user to manage past, present and future trip journals. However, since we eventually want to be able to support features that allow users to share and collaborate on journals, we first need to somehow identify each user. Hence, the initial screen the user will see is a login screen, with an alternate signup screen that will allow us to onboard new users who have downloaded and used the app for the very first time as shown in Figure 2.1.

Notice that both the iOS and Android version of the app display a row of three tabs labeled "Trips", "Calendar", and "Map", respectively. This tabbed organization of the screens in Traxy is

a fairly common approach in both iOS and Android. It gives the end user a high level of view of all the functionality or various use cases supported within the app.



(a) (b) (c) (d)

Figure 2.1: a) iOS login screen. b) iOS signup screen. c) Android login screen. d) Android signup screen.

Once the user is logged into the app, he will be presented with the app's main screen as shown in Figure 2.2. In this case, it is a list of of vacation journals, grouped by current vacations, future vacations and past vacations. On the journal list screen, the user can tap on a vacation journal entry to view / edit it, or they can tap on the plus button in the upper right to create a new vacation journal.

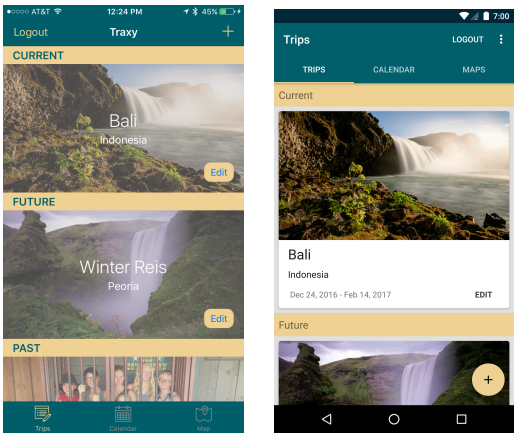
When the user taps on the plus button, the journal editor screen is displayed. The user can describe the vacation destination as well as the anticipated start / end dates. Once the new journal is fully described, the user can save it and return to the journal list screen. Eventually when the user has entered photo entries within a journal, the journal editor will display them and allow the user to select one to be the journal's cover photo. After a journal has been created, the user can edit the journal information by tapping the button labeled "Edit" in the lower left hand corner of the journal's cover image.

When the user taps anywhere on the cover of a journal on the main screen, a list of all the entries the user has made in the journal is displayed in descending date order (e.g newest entries at the top) as shown in Figure 2.4. Notice that in the date header for the entries, the weather conditions for the date and locality the entry was made is displayed. That information will be retrieved from a remote weather service.

From the journal entries screen, if the add button (lower right *floating action bar button* on Android, and upper right *bar button* on iOS) is tapped the user can proceed to add a variety of entry types, including a text entry, photo or video entry via the camera, photo or video entry from the user's previously captured photos and videos, and audio entries. These options, as presented in the iOS and Android versions of the app are shown in Figure 2.5.

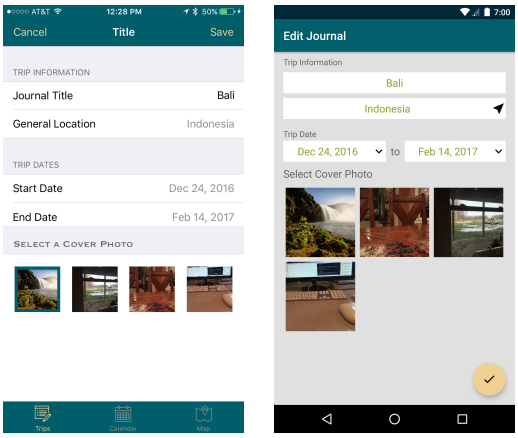
Photo and video entries can be captured within the camera via the device's camera or by selecting them from the device's library of previously captured images, as shown in Figure 2.6.

Once an entry has been made, the user has the opportunity to caption it, set the date, and



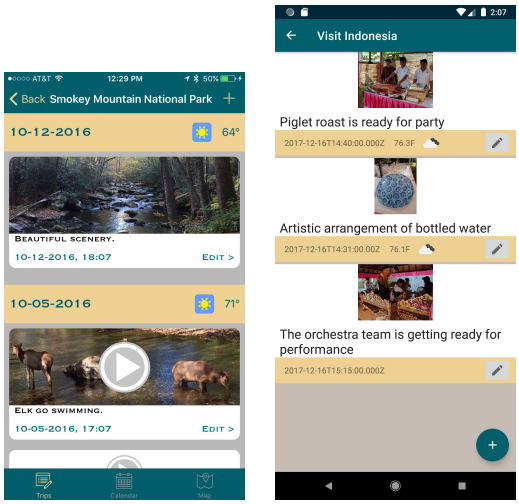
(a) (b)

Figure 2.2: a) iOS main screen listing the set of journals. b) Android main screen listing the set of journals.



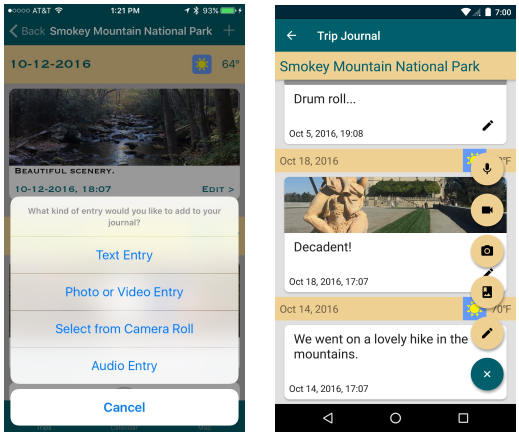
(a) (b)

Figure 2.3: a) iOS journal editor screen. b) Android journal editor screen



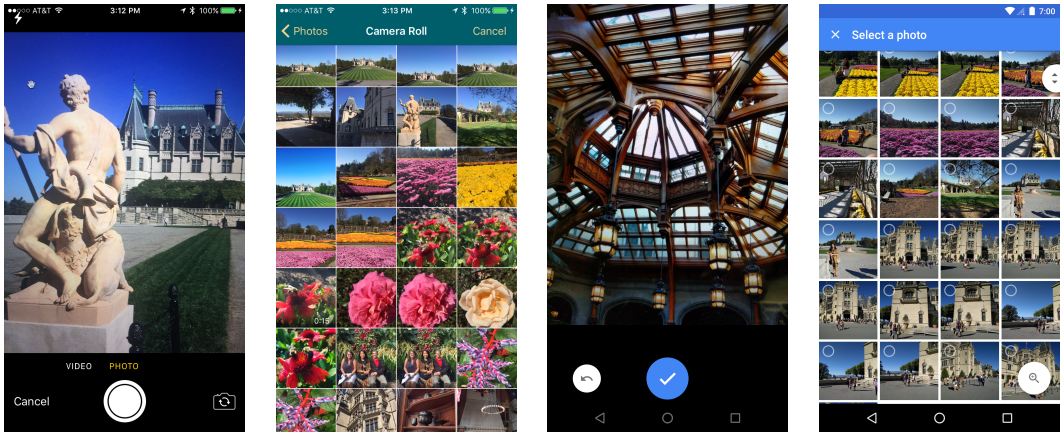
(a) (b)

Figure 2.4: a) iOS journal entries screen. b) Android journal entries screen



(a) (b)

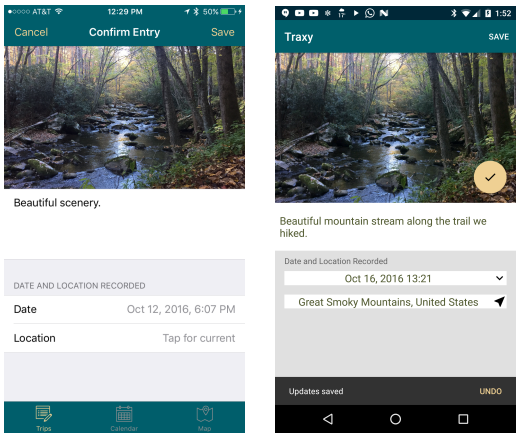
Figure 2.5: a) iOS add journal entry options. b) Android add journal entry options.



(a) (b) (c) (d)

Figure 2.6: a) Capturing a photo or video on iOS. b) Selecting an existing photo or video on iOS. c) Capturing a photo or video on Android. d) Selecting an existing photo or video on Android.

location as shown in Figure 2.7. Note that once the user confirms the entry the data (including the media attachment) will get stored in a cloud database. If the user were to login on a different device, all of their journals will be immediately available.

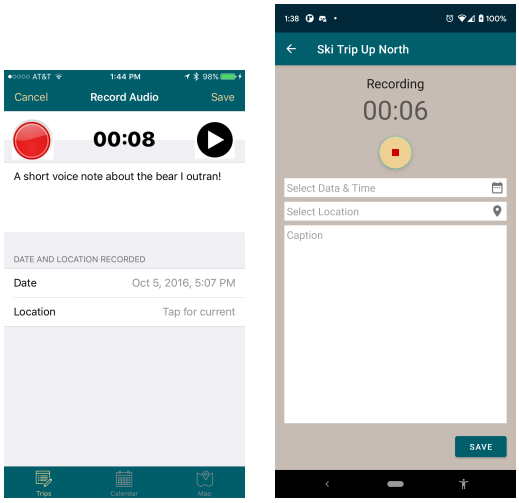


(a) (b)

Figure 2.7: a) Journal entry confirmation on iOS. b) Journal entry confirmation on Android.

Audio entries can be captured by using the device's microphone. Users can playback the captured audio entry before saving it as shown in Figure 2.8.

Selecting the calendar tab in our Traxy app reveals the calendar view shown in Figure 2.9. On the top one third of the screen a calendar is displayed. The user can interactively swipe left and right to change the month that is displayed. Any journal whose range of dates from start to end falls on any day within the displayed month is shown below the calendar in the same format as



(a) (b)

Figure 2.8: a) Creating an audio journal entry on iOS. b) Creating an audio journal entry on Android.

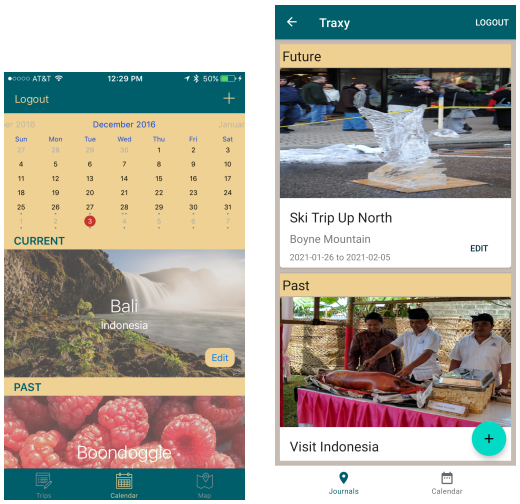
the first tab. The idea of this view is to let a user quickly locate a journal based on its date.

Finally, selecting the map tab in our Traxy app reveals the map view shown in Figure 2.10. A map is displayed and annotated with a pin for every journal that has been recorded. If the user taps on a pin, the title of the journal is displayed, and a tap on that will send us off to the list of entries within that journal. The map, when initially displayed is zoomed into the minimal level that will contain all of the pins. The motivation for this tab is to let a user quickly locate a journal based on where they recorded it. While entries within a journal may be tagged with a multiplicity of locations, when we create a new journal for the first time we will tag it with the general vicinity that the entries will be recorded.

You will implement the Traxy app as you work through the remaining chapters of this book. We will start small and iteratively flesh out the app on both iOS and Android until we have it fully functional on both platforms. Let's roll up our sleeves and write our first app for both iOS and Android.

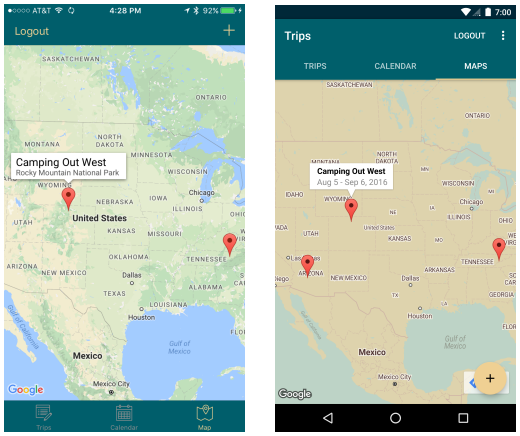
2.2 Writing Our First Mobile App

We will begin our exploration of Android and iOS by implementing a simple approximation of the login screen described in the previous section. True to tradition, we will first focus on a "hello world" form of app in which we will get a welcome message up on the screen consisting of a textual label and our Traxy logo in the form of an image. Once that has been accomplished we will extend our implementation by making it interactive. In particular, we will add text entry fields to collect the user's email address, and password, along with a button that when submitted will execute code that validates what was entered by checking that a properly formatted email address entered, and the correct password. Assuming the data was entered properly, our app will print a



(a) (b)

Figure 2.9: a) The calendar tab on iOS. b) The calendar tab on Android.



(a) (b)

Figure 2.10: a) The map tab on iOS. b) The map tab on Android.

success message to the console. If there were any input errors, we will display a simple dialog that reports the error to the user.

For now, we won't worry too much about getting the screens to look nice like they do in the previous section. The goal here is to basically get the information up on the screen so it is visible to the user, and to be able to allow the interactions described above. In a future chapter will we will dig into the details of how to get the screens to layout properly on the respective platforms.

Upon completion of this chapter, you will have a basic understanding of how to get a simple single screen interactive application running on both Android and iOS.

2.2.1 Platform Similarities

Despite the fact that they are entirely different software stacks, both Android and iOS share some common techniques in organizing the overall structure of a mobile application.

- There is a clear decoupling between code that implements the application logic and the user interface design.
  - Both platforms adopt XML to codify user interface specifications.
  - In Android the user interface of each screen is defined in one XML layout file, while iOS employs both individual layout files (.XIB) and interconnected UI views in a storyboard (.storyboard).
- iOS developers write application logic in classes derived from `UIViewController`. Android developers write application logic in classes derived from `Activity`.
- User interface elements in iOS are derived from `UIView` and Android user interface elements are derived from either `View` or `ViewGroup`.
- Both platforms support mixed-language programming. Android developers can write code in Java or Kotlin and iOS supports both Objective-C and Swift.
- Both XCode and Android Studio require you to enter a unique identifier (bundle identifier in iOS and package name in Android). If you plan to publish your app to either App Store or Google Play Store, the identifier must be unique within the respective store.

2.3 Your Initial iOS App

By default, iOS applications are authored according to the ubiquitous model-view-controller (MVC) architectural pattern. We will discuss MVC in more detail in a later chapter, so at this early stage of our investigation we will simply introduce our application's view, i.e., the visual components that appear on the screen and allow the user to interact, and its controller - the code that gets executed when the user interacts with the application. For example, if a button is displayed by the view, we might associate code with it that will get executed when the button is pressed.

In iOS application views are normally created using Xcode's Interface Builder. Interface Builder is a sort of what-you-see-is-what-you-get interactive view editor that allows the developer to develop non-trivial hierarchies of views from a palette of common customizable user interface controls, defined by Apple. In addition to determining *what* user interface controls will be utilized by

a particular screen, Interface Builder also provides the developer with the ability to constrain *how* the components will be actually laid out on the screen at run-time.

Controllers in iOS are created by introducing a subclass of the `UIViewController` class or one of its derivatives defined in the `UIKit` framework. The code can be written in either Objective-C or the Swift programming languages.

2.3.1 Creating a Project in Xcode

Let's begin by creating a new single view application within Xcode. Once Xcode is started, a welcome dialog is displayed, as shown in Figure 2.11. Initiate the creation of a new project by clicking on the "Create a new Xcode project" option in the lower left of the dialog. Alternately, you can use the File ⟩ New ⟩ Project... selection on Xcode's application menu to initiate the creation of a new project.

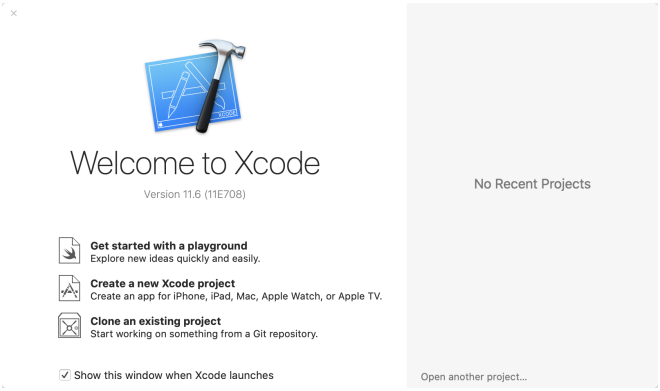


Figure 2.11: The Xcode welcome dialog.

Once project creation is initiated, Xcode will present a variety of template options, as shown in Figure 2.12. Since Xcode serves as the common IDE for all of Apple's product line (iOS, watchOS, macOS, and tvOS) it is important that you select iOS in the upper left of the dialog, prior to selecting a project template style. Finally, select the Single View Application style and then proceed to click the blue Next button.

Once a template has been selected, Xcode presents a number of options in the new project options dialog, as shown in Figure 2.13. Set the Product Name to `TraxyApp`. You can use your free developer account for your team (or use a paid account if you have one). If you haven't registered for an Apple develop account (free or paid) then for the Team field just select None. This will allow you to run your app on a simulator, but you won't be able to run on a physical iOS device until you register for a developer account.

The value you provide in the Organization Name field will be used in the boilerplate code header comments Xcode places at the top of each source file you create in the project. You can always go back and change this later.

The Organization Identifier will be combined with your Product Name to form what is known as the app's Bundle Identifier. The Bundle Identifier is important in that it will uniquely identify your app in the Apple iTunes App Store if you eventually publish it. The convention is to set your

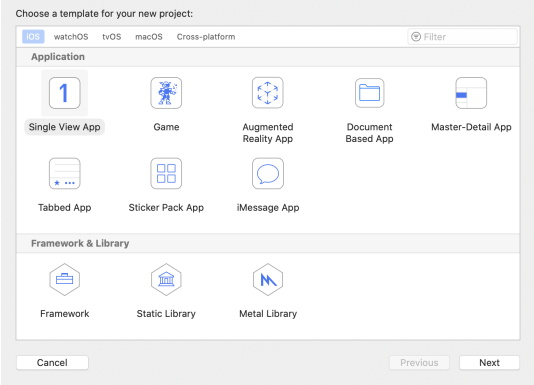


Figure 2.12: The Xcode project template selection dialog provides a variety of application styles to create. Note that we have iOS selected in the upper left of the dialog.

Organization Identifier to your organization's reversed Internet domain. This is a convention very similar to creating unique package names in your Java source code. Since we are faculty in Grand Valley State University's computer science department, we're using our department's reverse Internet domain: `edu.gvsu.cis`.

In this book, all of our iOS coding will be done in the Swift programming language. In particular, we will be using Swift 5.1, Apple's current version of this rapidly evolving language.

We will be implementing the Traxy app's user interface via a conventional storyboard. Another option here that we will not look at is SwiftUI. Apple introduced SwiftUI with Xcode 11 in 2019. It is a new framework that uses a declarative approach to defining an iOS app's user interface. So we'll set the User Interface option to Storyboard.

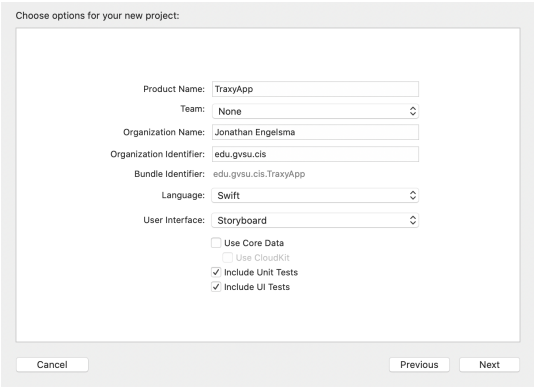


Figure 2.13: The Xcode project template selection dialog provides a variety of application styles to create.

Finally, check the Unit and UI Test options, so that you can eventually introduce some test automation to your project. You can now proceed to the next step by clicking the Next button in the lower left hand corner of the dialog.

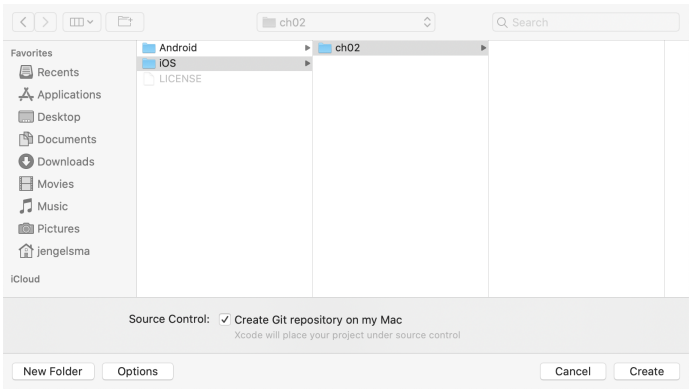


Figure 2.14: The final step of project creation in Xcode is to specify the directory in which the project folder will be created.

The last step in project creation is to specify to Xcode where the project folder is to be located on your local drive. Make sure you select the Source Control checkbox towards the bottom of the dialog. This will cause Xcode to place your project under git control. Even if you are working solo and not part of a larger development team, managing your source code with git has many advantages. If you aren't already proficient with git, we strongly encourage you to add it to your list of technologies to learn. A good starting point is Git online documentation[1] where you can find free books and videos. Once you've navigated to the directory that will contain your project directory, go ahead and press the Create button in the lower left to complete the creation of your new project.

At this point, you've created a runnable single view iOS application and the Xcode IDE has loaded the newly created project as shown in Figure 2.15. You can confirm that the app is runnable, by clicking on the Build and Run button in the upper left hand corner of Xcode (it looks like a play button). Alternately, you can run the app by selecting Product > Run from the Xcode application menu, or the key sequence ⌘ + R. Your app is not very interesting at all, as once it launches in the emulator you will simply see a blank white screen.

If you select the top most item in the project explorer on the left (labeled "TraxyApp"), and the TraxyApp target as shown in Figure 2.15, you'll noticed a field labeled "Devices" under the Deployment Info settings in the middle of the page.

The Devices field allows you to select one or more of three options:

- iPhone - indicates your app will target the various iPhone form factors.
- iPad - indicates your app will target iPadOS.
- Mac - indicates your app will target MacOS. .

The choice you make for the Devices field will have ramifications if you publish in the App Store. For example, if you choose iPhone, then it will only show up in the App Store when users are on an iPhone. Unless iPad users deliberately override the default search settings to allow for iPhone apps to show up, your app will be invisible. If you select iPad, then the app will only be

visible and installable for users browsing the store with an iPad. However, if you select iPhone and iPad, it will be available to iPhone and iPad users, with layouts specifically implemented for these two different form factors. Modern versions of MacOS can also run iOS/iPadOS apps. However, do be aware that more is involved than simply checking the MacOS option here on the settings screen!

Go ahead and select only iPhone for your device setting. Do pay attention when making this decision in general, as it does impact the shape the app will take once published. For example, if you launch a universal app on the App Store, Apple will not allow you to publish a future version of the app that only supports only the iPhone or only the iPad.

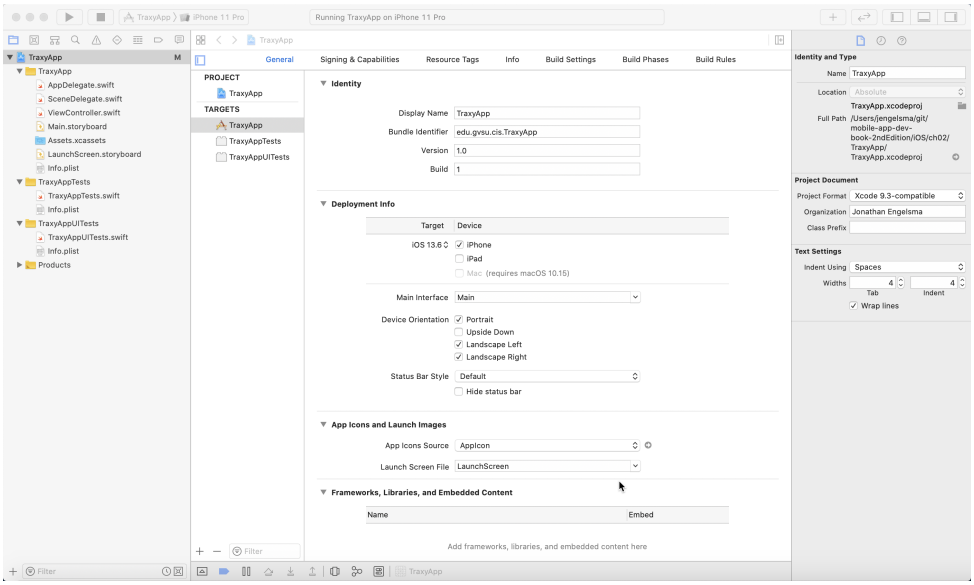


Figure 2.15: The new Traxy app has been created and loaded in Xcode, and we are ready to start customizing it.

2.3.2 Laying out the screen in Interface Builder

Let's go ahead and customize the screen that gets displayed, and turn it into a sort of "hello world" app. You will accomplish this by editing the project's storyboard. Over in Xcode's project explorer, displayed on the left hand side of the window by default, you will see a file named Main.storyboard. If we select that file, Interface Builder will open up in the main body of the IDE as shown in Figure 2.16. At the moment, your storyboard contains only a single scene. This scene will serve as your initial login screen. You want to greet the user with a nice warm welcome, so dress up the scene with some additional welcoming information. In particular, add a `UILabel` with welcome text, and a `UIImageView` displaying the Traxy logo.

Use the key sequence ⇧+⌘+L to display Interface Builder's library of user interface components. To find the label and image view components, scroll through the components displayed in the library dialog. Alternately, you can simply start typing the name of the component you are

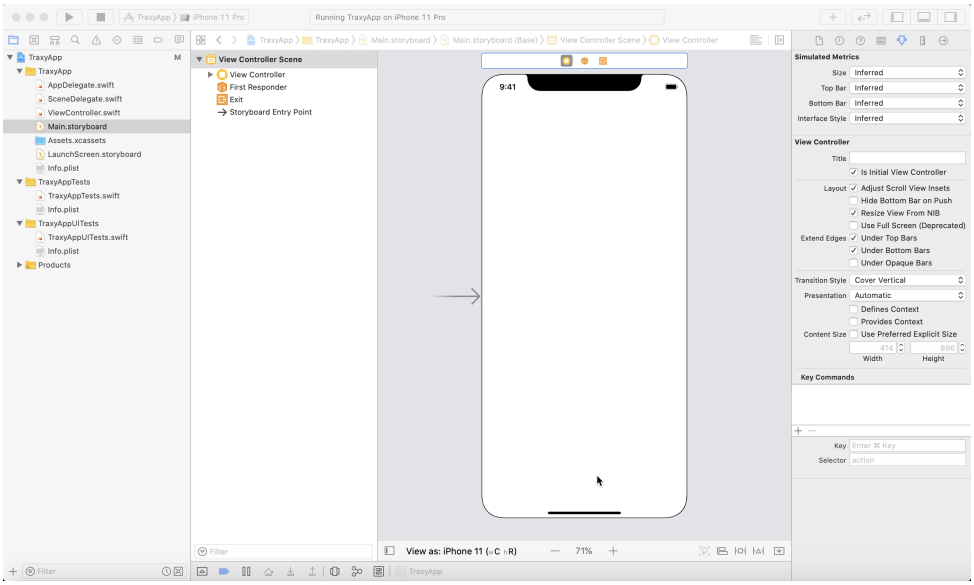


Figure 2.16: Selecting the Main.storyboard file in the Xcode project explorer opens up Interface Builder, allowing us to edit our app's view(s).

looking for in the search field at the top of the dialog as shown in Figure 2.17. Once you find the UI control of interest, simply select and drag it over into the Interface Builder scene.

Don't worry about getting your interface layout to work on a variety of different screen sizes. We will take that up in a later chapter when we talk about AutoLayout, Xcode's constraint-based layout system. Instead, simply assume a particular screen size and make sure you run the same sized simulator when you test your app. Looking back at Figure 2.16 notice right under the scene on the bottom of the Interface Builder editor it reads "`View as: iPhone 11`". As long as you run the app on the iPhone 11 simulator, you should be able to get your layout to work with minimum fuss, which is exactly what you want at this stage.

> **NOTE**
>
> The devices that show up on the **View as** setting in Interface Builder depends on the version of Xcode you have installed. The important thing to note is that when you layout your scene, just make sure you run it on the same simulator (or device) that you are laying out the interface on in Interface Builder. In a later chapter we will study Auto Layout, Apple's constraint-based layout mechanism that helps us define user interfaces that look good on all iOS devices.

With that in mind, drag a label over into the scene and center it towards the top of the scene in Interface Builder. You will notice that Interface Builder provides blue alignment guides informing you when the label is center, as shown in Figure 2.18a. After dropping the label, go ahead and

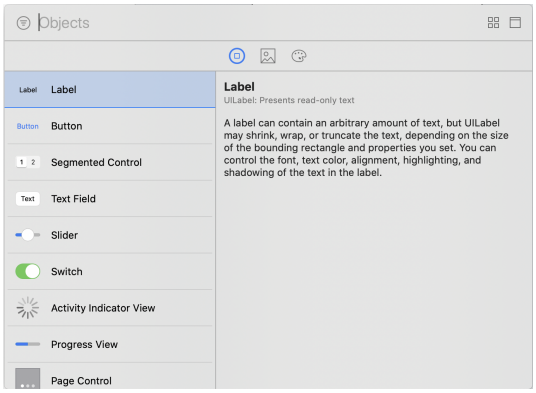


Figure 2.17: You can scroll through the Xcode object library pane to find UI controls, or you can use the search field at the top of the dialog to limit the controls displayed.

double click on it and change the text to read `"Welcome to"` as shown in Figure 2.18b. You may have to re-center the label after changing the text.

In addition to interacting with a UI control directly when it's dropped into a scene, you can also select a control in the Interface Builder editor, and once selected use the Attribute Inspector pane, which by default appears on the right hand side of the IDE (alternate keyboard shortcut is ⌘+⌥+4). For example, you can change the text on the label you just dropped into the scene by selecting it and then changing the appropriate text field in the Attribute Inspector, as shown in Figure 2.19.

Next, we would like to display the official Traxy logo centered under the "welcome" label you just added. You will follow a very similar approach, only this time instead of dragging a label from the object library pane, you will find an Image View object and drag it into the scene, as shown in Figure 2.22. Since your app's logo won't change frequently, go ahead and add an image asset directly to your application. To accomplish this, select the `Assets.xcassets` file in Xcode's project explorer on the left, as shown in Figure 2.20. Having the project's asset catalog opened, you can add an image to the project, by clicking on the small + button at the bottom of the image list which is displayed vertically, just to the right of the project explorer. Initially, you will only see a single entry in this list named `AppIcon`, which is a placeholder for the launch icon that was added when the project was created. On the + button's popup menu, select the `New Image Set` option. A new entry will be added to the asset list, just under the `AppIcon`. Change its name to `logo` by clicking on the default name `Image`. The final step in adding an image asset is to go to Finder on your Mac, browse to your logo image, and drag it over to the 2x slot for the logo image. Upon completion, your asset catalog, should look like Figure 2.21.

Image sets consist of multiple image sizes in order to optimize presentation on the various screens used by different iOS products. For example, the larger iPhone screens (e.g. iPhone 6/7 Plus) will utilize the 3x image, and all other high resolution iPhone devices (iPhone 4, 5/5S, 6/6S, 7) will use the 2x image sizes. The 1x images would be used by standard definition devices (e.g. non-retina display models prior to iPhone 4), but since the current version of iOS does not run on those devices, and very few are still in use,we typically will not supply the 1x version of the images. Eventually, before submission to the App Store, we'd want to create the 3x version of our

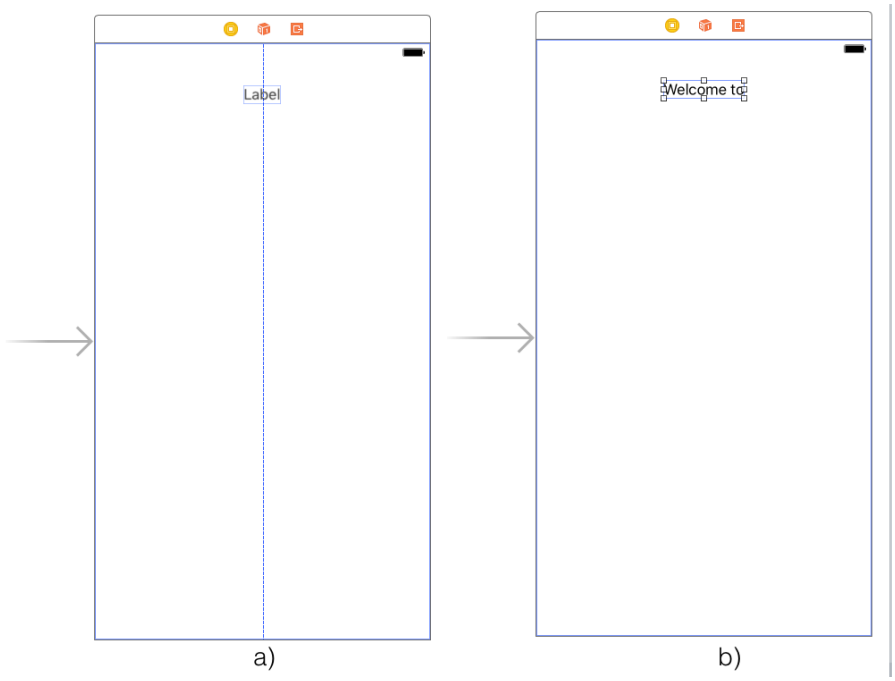


Figure 2.18: a) Drag and drop a label into the scene, centering it towards the top. b) After dropping the label, double click on it to change the text to "Welcome to".

logo by producing a copy of the 2x images scaled to 3 times what the standard resolution image. So for example, if our 2x image was 100 X 100, the 1x version would be 50 X 50 and the 3x version 150 X 150. Given this, you obviously would want to start your image design in the highest resolution needed, or better yet use a vector image format so you can scale up or down as needed.

In order to get the logo to show up in your login scene, you need to return to the storyboard and set it to be the default image on the image view you added previously. After opening the Main.storyboard file, make sure the `UIImageView` is selected. Over to the right in the Attribute Inspector, the image view's attributes are displayed. Clicking on the drop down in the Image field, you can now see that the logo you just added is an option, as shown in Figure 2.22. Go ahead and select the logo, and immediately the scene is updated to display the logo. Depending on the size of your image view, the aspect ratio of the image might seem incorrect as displayed. To correct this you can set the `Content Mode` setting in the Attribute Inspector to `Aspect Fit`.

You now have a fully functional "hello world" app! You are now ready to run the app. Make sure the simulator device you select in the upper left hand corner of the IDE (just to the right of the Run/Stop buttons) matches the device you layed your interface out on in Interface Builder. For example, as you can see from the figures above, we used the iPhone 11 so we would want to test on a simulator that has the same screensize as the iPhone 11. Now go ahead and press ⌘ + R to run the app, and check your handiwork. If you did everything correctly, in a moment the simulator will be running and you will see the Traxy app display its nice warm welcome, as shown in Figure 2.23.

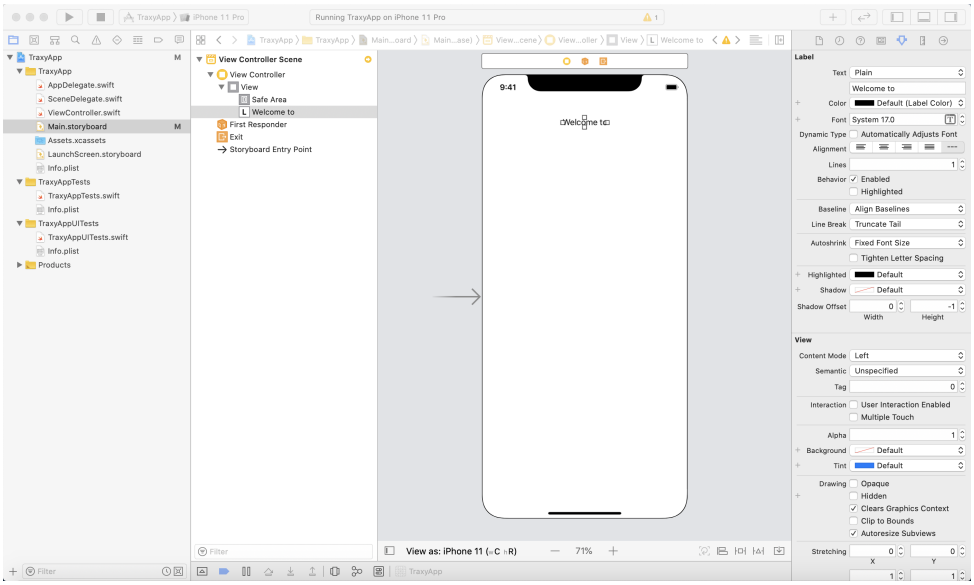


Figure 2.19: Properties of UI controls dropped into a scene can be manipulated by selecting the control, and then editing the properties in the Attribute Inspector pane displayed on the right.

Let's extend the login screen with a user name text field (email address) a password text field, and a button to submit the user's credentials once entered. You can accomplish this the same way you added the welcome label and logo image. First, locate the `UITextField` in the Xcode object library. It is labeled as "Text Field" in the library. Once located, drag two of these controls out onto the scene under the logo. Accept the default height, but do drag out the horizontal edges so that they are standard distances from the leading and trailing edges of the screen. Interface Builder shows the standard distance from the edge of the screen by displaying blue alignment guides on the four sides of the screen when you drag near them. Finally, add a `UIButton` that the user can tap to submit their login credentials. These are labeled as "Button" in the Xcode object library. Drag one onto the scene and center it with standard vertical distance under the second text field you added. Again, the standard distance will show up as a blue alignment guide once you drag the button into the proper position.

The first text field under the logo will serve as the email entry field, and the second the password. Somehow you have to let the user know what each field is for. Unlike your typical desktop computer, phones have very limited screen real estate. A common convention is to label input fields, using the fields themselves. This can be accomplished in iOS by selecting the text fields in Interface Builder and then using the Attribute Inspector on the right to set its Placeholder attribute as desired. For your two text fields, set the placeholders to "Enter email" and "Enter password", respectively. While you are editing the placeholder for the text fields, it would be worthwhile to take a look at some of the additional attributes available for the `UITextField` that allow you to further customize the field's behavior. For both of the text fields, set the "Clear Button" attribute to "Appears while editing". This will give you a convenient way to clear out the field in one tap. Under the Text Input Traits section of the Attribute Inspector set the "Content Type" to "Email

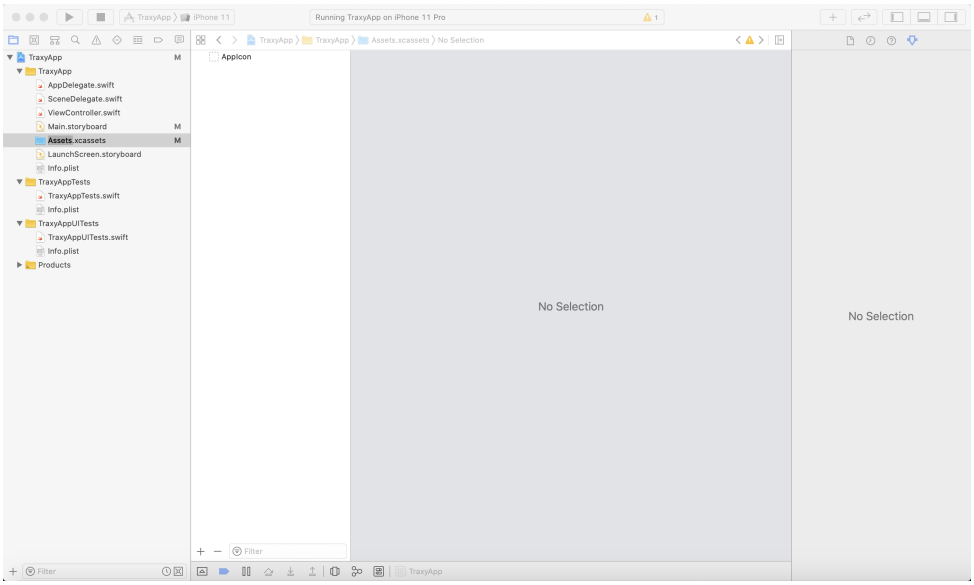


Figure 2.20: Image assets can be added to the project, by selecting the Assets.xcassets file in the Xcode project explorer.

Address" and "Password", respectively. By setting the "Content Type" you are indicating the semantic meaning of the field, which helps iOS exploit auto-fill opportunities on behalf of the user. iOS has a special keyboard tailored for email address entry, so on the email text field, set the "Keyboard Type" attribute to "E-mail Address". Leave the "Keyboard Type" attribute of the password field to "Default", but do check the "Secure Text Entry" box toward the bottom of the text field attributes pane. This will cause the field to not display the password as it is typed. In addition, check the "Auto-enable Return Key" checkbox for both fields. This will make the keyboard's return key inactive until text has been entered. Figure 2.24 shows the final attribute settings for the two text fields.

Your final task in completing the simple login form is to change the button's text label from the default "Button" to the text "Sign In". This can be accomplished either by double clicking on the button in the scene or by simply selecting the button and then using the Attribute Inspector to change the label text. The completed login form is shown in Figure 2.25.

2.3.3 Adding Outlets and IBActions

Thus far, you've used Interface Builder to develop the login view for your app. Now we would like to add some basic validation logic that makes sure a valid email address has been entered, along with the correct password. In the MVC pattern, it is the controller's responsibility to handle user interactions with the view. As was mentioned previously, on iOS that means we need to introduce a new class that extends the `UIViewController` class. When we created our single view Traxy app, Xcode introduced a new class in Swift called `ViewController` which you can find in the Project Explorer. You need to somehow establish the fact that when the "Sign In" button is pressed,

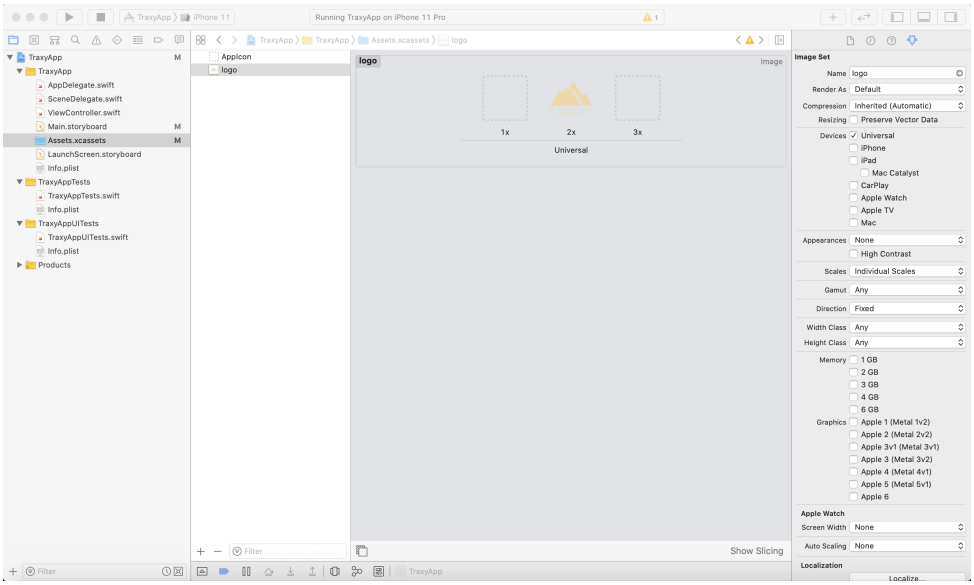


Figure 2.21: The assets catalog after adding our logo image set.

a method implemented in Swift by the view controller will get executed. Since you've defined your view in Interface Builder, you will initiate that process within Interface Builder as well via Xcode's Assistant Editor. Assuming your Main.storyboard is currently open and displayed, enter the Assistant Editor by clicking on the button labeled with horizontal lines on the top right of the Interface Builder toolbar as shown in Figure 2.26. Select Assistant from the popup menu pane. The Assistant Editor view will split the main area of the Xcode IDE into two areas. On the left side will be Interface Builder and on the right side your code editor. By default, whatever you select in Interface Builder will automatically load the associated Swift code of the selected view controller. Every scene in your storyboard represents at least one model-view-controller triad. Since you created the application using Xcode's project creation wizard, it automatically created a scene in the storyboard, generated a controller named `ViewController` and associated it with the view contained within the scene. You can confirm this is the case by clicking on its title bar of the scene of interest within the storyboard, and then displaying the "Identity Inspector" pane on the right hand side of the Xcode window (alternate keyboard shortcut is ⌘+⌥+3). You will notice that the class field is set to `ViewController` which is the Swift class associated with the login scene[1] At this point, if you select the login scene, the Swift code of `ViewController` will be automatically displayed on the right. You can also manually override this behavior by clicking on the "Automatic" label right above the code editor pane and changing it to "Manual" and navigating to the source file you want displayed.

In order to associate a button touch event to code in your controller, while holding the Control key (ctrl) click and drag from the button over to the code pane on the right and release the mouse button when you see a label "Insert Outlet, Action or Outlet Collection" display in the location

[1]The **Class** field may be hidden by default. If this is the case, click on the show button to the right of the **Custom Class** label to display it.

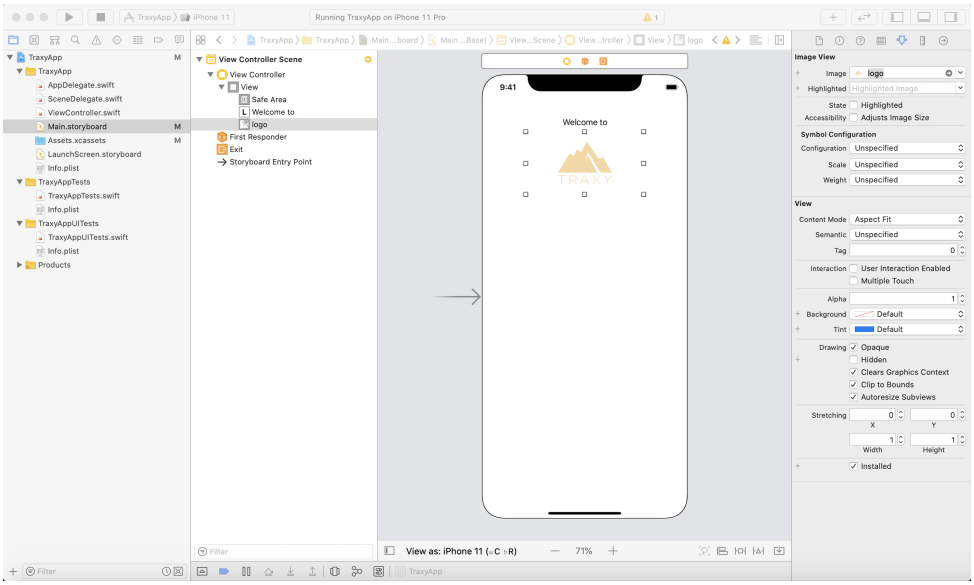


Figure 2.22: You can assign images in your asset catalog to image views in your scenes by selecting the image view and then choosing the image over in the Attribute Inspector on the right.

you'd like to place the code to handle the button touch event. Upon release, a small dialog window is displayed. Set the "Connection" field to "Action", "Name" field to "signupButtonPressed", and "Type" to `UIButton`. Once the dialog looks like the one in Figure 2.27 go ahead and press the connect button. This will automatically stub out a method in `ViewController` that looks like this:

Swift

```
@IBAction func signupButtonPressed(_ sender: UIButton) {
}
```

Methods that can be wired up to events generated by controls specified in Interface Builder are commonly referred to as *actions*. The `@IBAction` is simply a directive that informs the tooling that `signupButtonPressed` is a valid candidate for wiring up UI events to from Interface Builder.

Now that you have a method in which to implement the form validation logic, how do you access the text input entered in the two text fields? Clearly, we need some way to access these controls from our Swift code. This is accomplished using *outlets*. Outlets are simply references in our Swift code to components (UI controls, layout constraints, etc.) that were populated in a scene via Interface Builder. You can create outlets in the controller in the same manner you just created the action for the button. While holding ctrl click on the email address field, and drag over to the code pane on the right. Leave the "Connection" field set to the default "Outlet" and then simply set the "Name" field to `emailField`, leaving all other fields to default. Repeat this process for the password field, only set the "Name" field to `passwordField`. Upon completion, the following outlet definitions will have been added to `ViewController`.

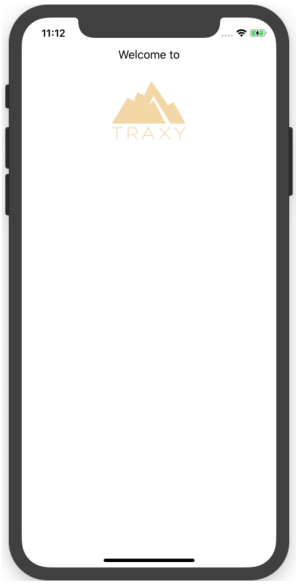


Figure 2.23: Our initial "hello world" app is complete, as shown running in the iPhone 11 simulator.

Swift
```
@IBOutlet weak var emailField: UITextField!
@IBOutlet weak var passwordField: UITextField!
```

Notice that these fields are defined as implicitly unwrapped `UITextField` optionals. This is safe in that these fields will never be set to nil, as they will be automatically populated by UIKit when the view hierarchy is loaded from the storyboard.

> **NOTE**
>
> It is possible to manually remove an outlet connection in Interface Builder, while leaving the declaration intact in the code. This happens frequently to novice iOS developers, and manifests itself by a mysterious crash at runtime! A quick way to confirm that all the outlets in your source file are in fact properly connected to elements in the view is to select the scene in Interface Builder and display the Connection Editor pane on the right side of the IDE by pressing the button labeled with a right arrow in a circle (alternate keyboard shortcut is ⌘+⌥+6).

Now that you have references to the text fields, go ahead and implement the validation logic in `signupButtonPressed`. Checking if the password field is correct is easiest, so implement that as shown in Listing 4. We're basically just checking to see if the pattern "traxy" appears somewhere

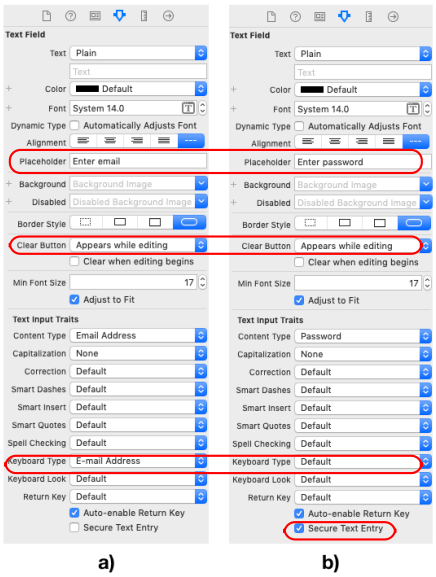


Figure 2.24: The Attribute Inspector settings for a) the email address field and b) the password field.

in the password, regardless of case. Eventually you will of course have to wire this up to a real password authentication service, but this is a reasonable approximation that meets our purposes for now.

Swift

```swift
var pwOk = false
if let pw = self.passwordField.text {
    if pw.lowercased().range(of: "traxy") != nil {
        pwOk = true
    }
  }
if !pwOk {
  print("Password is invalid")
}
```

Listing 4: Validating that the password field contains the string "traxy".

Next, tackle the email address validation. After a bit of investigation in Apple's developer documentation which can be accessed by pressing Window » Documentation and API Reference from the Xcode application menu (alternate keyboard shortcut ⇧+⌘+0), we learn that we can use a regular expression and a class named `NSPredicate` to come up with a reasonable solution shown in Listing 5 .

There are still some a couple of things you can add to the implementation to make the login form behave more properly. For example, when the user taps the Return key at the moment,

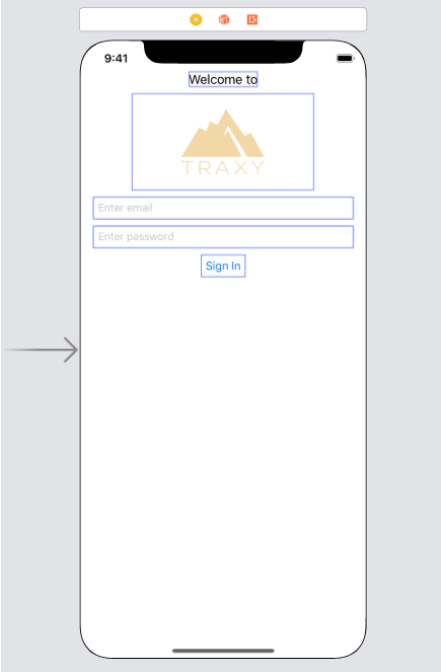


Figure 2.25: A simple login screen for the Traxy app.

nothing happens! It would be desirable to automatically transition to the password field when the return key is tapped in the email field. Furthermore, it would be nice if tapping the return key in the password field, automatically calls the action associated with the SignIn button. Finally, when editing text on an iOS app, it is common to dismiss the keyboard whenever the user taps somewhere outside of a text field.

To handle the Return key behavior, you need to have the scene's view controller behave as the *delegate* of the `UITextField` controls. The delegation pattern is used extensively within the iOS frameworks. What is happening here is that your `UITextField` controls are essentially delegating various events to the controller that is managing them. To establish this, you can implement a Swift extension of the `ViewController` class that implements the `UITextFieldDelegate` protocol. (If you need to brush up on Swift extensions check out Appendix A in the back of the book.) You can dismiss the keyboard when tapping outside of a text field, by adding a `UITapGestureRecognizer` to the scene's parent view. Listing 6 contains the complete code found in the `ViewController.swift` file after implementing these features.

In the `viewDidLoad` method add a tap gesture recognizer to the scene's containing parent view. You must also explicitly set the view controller instance to be the delegate of both text fields. The actual delegated behavior takes the form of an extension to the `ViewController` class that implements the `UITextFieldDelegate` protocol which you can find at the end of the listing. Whenever the Return key is pressed on one of the two fields, the `textFieldShouldReturn` delegate method gets called. If pressed in the email field, we call the `becomeFirstResponder` method on the password field. This essentially gives input focus to the password field. If Return

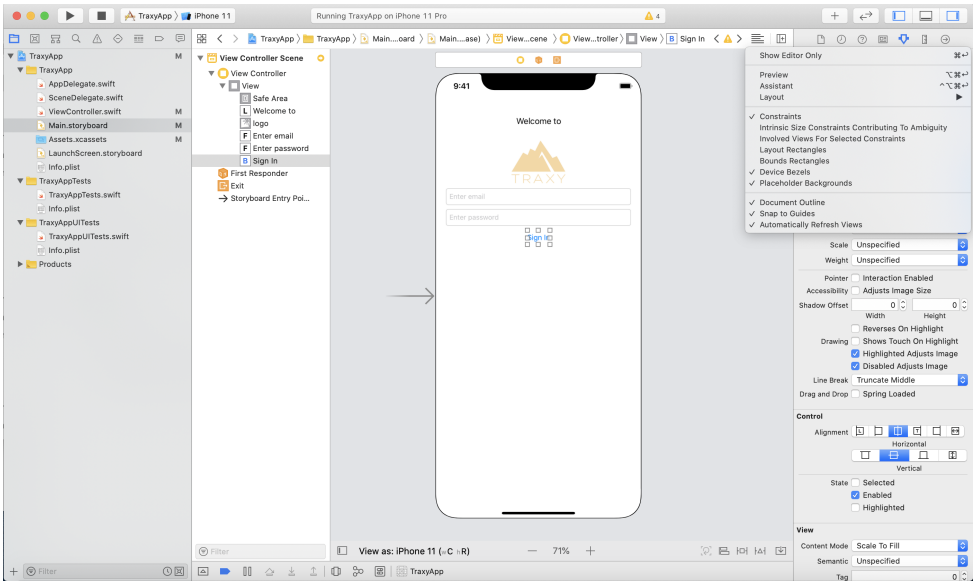


Figure 2.26: Activating the Assistant Editor view when in a storyboard is a little tricky in recent versions of Xcode. To activate, simply click the little hamburger like menu on the right or the Interface Builder tool bar.

is pressed in the password field we simply invoke our field validation logic that we broke out into its own method. This is the same code we invoke if the signup button is tapped.

At this point you can go ahead and build / run the application and confirm that the various features you implemented are working correctly. You can view the output of the print statements in the debug pane displayed towards the bottom of the Xcode window as shown in Figure 2.28.

2.3.4 Internationalization

The vibrant application ecosystems established by Google and Apple means that even independent developer can launch an app and become an international business concern (e.g. sell software in markets beyond the developer's home country). The application ecosystems take care of collecting payments from your end users in their native currency, and converting it your local concurrency and depositing it into your bank account.

However, that does mean that you can no longer assume your end users speak the same language you do. Happily it is fairly easy to internationalize both iOS and Android applications. Let's take a look at how you can internationalize the login screen you have just created.

First, select the top level TraxyApp in the project explorer, and in the editor area, make sure the TraxyApp project is selected as well. At the bottom of the info tab you will see the Localizations section at the bottom, as shown in Figure 2.29. To add a localization click on the + button on the bottom. On the list of languages on the popup menu, select a language that you wish to support, and then click the Finish button on the dialog that is displayed.

If you successfully added a localization to the project, on your project explorer you will see that you can now expand the Main.storyboard, and arranged hierarchically you will see the storyboard

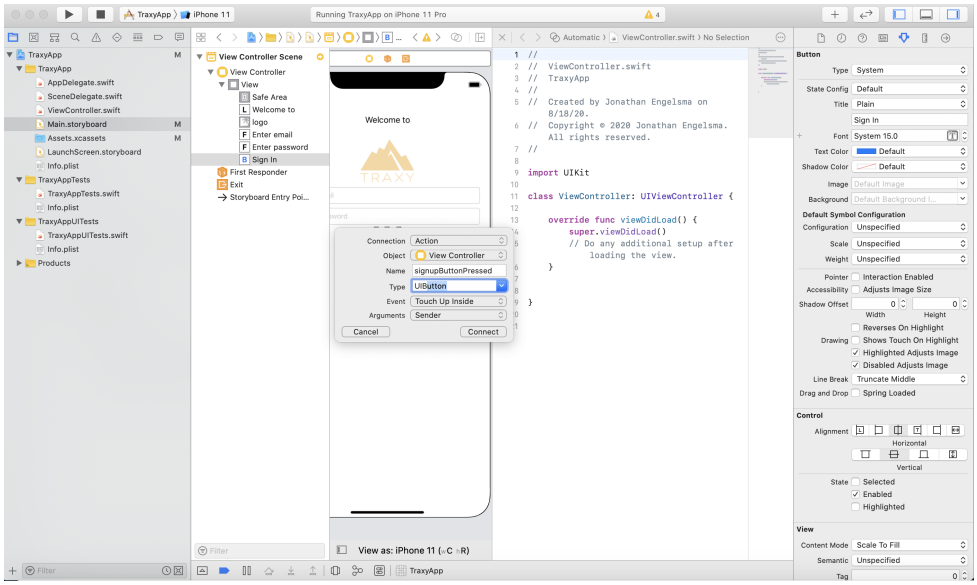


Figure 2.27: Using the assistant editor to add an IBAction to the sign in button.

file itsef (labeled base) as well as a Main.strings file labeled with the language you chose, as shown in Figure 2.30.

In our example, we added a strings file for the Dutch (Nederlands) language, so if we click on the Main.strings file labeled Dutch you will see system defined values for all the English string literals that appear in the storyboard, as is shown in Listing 8. Go ahead and open the equivalent file in your project and change the strings on the right hand of the assignment statements to their equivalent in whatever language you chose. In our example, we chose Dutch, so we're going to modify the file as shown in Listing 9. Be absolutely certain that you do not modify the system generate identifier on the left side of the equal signs.

To see the evidence of our handiwork, go ahead to your iOS simulator and under the Settings app and set the language to the language you chose previously under the General → Language & Region → iPhone Language screen. Next, build and run the app again, and you will see that your app conveniently uses the correct strings. Depending on the length of the translated string, you may need to resize your button!

This approach can be used to internationalize any string data within your storyboard. However, sometimes the strings shown in views are actually created within our code. For example, your app prints out the status of the form validation on the console. We could just as well be displaying that on the app screen, so how would we localize strings within our code?

To do this, you will need to manually create your own strings file to handle any strings created within the code. Go to File » New » File... and select "Strings File" in the Resource section. Click the Next button, name the file *Localizable.strings*, and click the Save button. Open the file, and using the exact same syntax we used for the storyboard strings file, go ahead and define every string literal your code will be dealing with. For the current app, Listing 10 shows the definition of the three string literals we use in the code. As before, a Dutch language version of this file would also

```swift
var emailOk = false
if let email = self.emailField.text {
  let regex = "[A-Z0-9a-z._%+-]+@[A-Za-z0-9.-]+\\.[A-Za-z]{2,}"

  let emailPredicate = NSPredicate(format:"SELF MATCHES %@", regex)
  emailOk = emailPredicate.evaluate(with: email)
}
if !emailOk {
  print("Invalid email address")
}
```

Listing 5: Validating that the email field is in the proper format.

have to be added for completeness. You'll find the complete solution online in the sample code.

Next, in order to get the code to use the strings defined in the string localization files, we need to replace each string literal `"foo"` with a call to the initializer of `NSLocalizedString`. For example, our previous code:

```swift
print("Invalid password")
```

now becomes:

```swift
print(NSLocalizedString("Invalid password", comment: ""))
```

Finally, in order to localize the strings file we just added, we need to select it in the project explorer, and then over on the far right in the File Inspector pane, click on the Localize... button. From the drop down, select English, and continue. The File Inspector pane will now be updated to show a list of all localizations in the project to-date. Make sure the checkbox is selected for each localization you intend to provide a string localization file for. If you do not provide a localization for a particular language, the app will default to the base strings when a device is configured for that particular language.

Having completed our initial iOS implementation of a basic login screen, let's take a look at how you can accomplish the functional equivalence with the Android SDK.

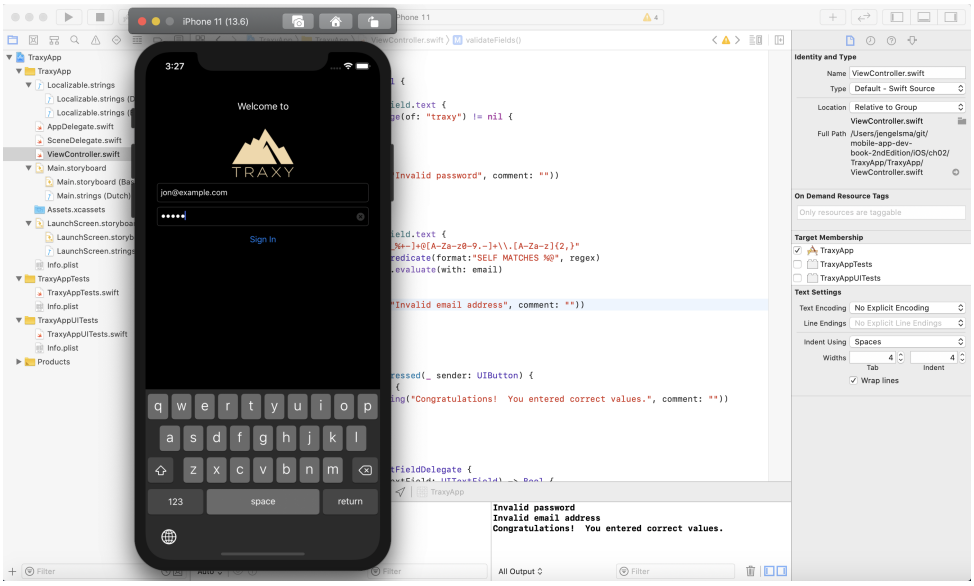


Figure 2.28: When the app is ran in the simulator, the output data generated by the print statement in the field validation logic get printed in the Debug pane on the bottom of the Xcode window.

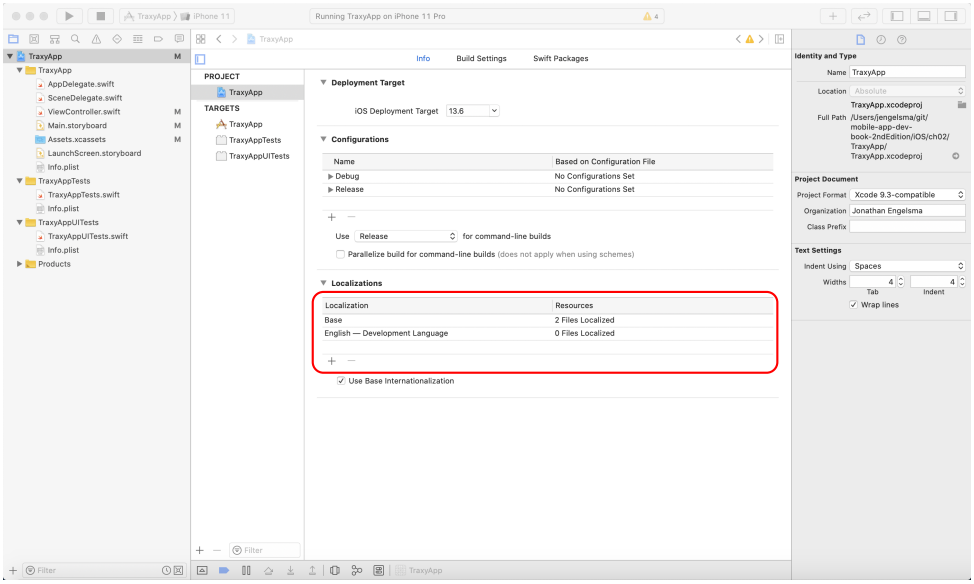


Figure 2.29: To internationalize the app, you need to add additional localizations on the Project Info screen.

iOS

```swift
import UIKit

class ViewController: UIViewController {
  @IBOutlet weak var emailField: UITextField!
  @IBOutlet weak var passwordField: UITextField!

  override func viewDidLoad() {
    super.viewDidLoad()

    // dismiss keyboard when tapping outside of text fields
    let detectTouch = UITapGestureRecognizer(target: self, action:
    #selector(self.dismissKeyboard))
    self.view.addGestureRecognizer(detectTouch)

    // make this controller the delegate of the text fields.
    self.emailField.delegate = self
    self.passwordField.delegate = self
  }

  @objc func dismissKeyboard() {
    self.view.endEditing(true)
  }

  func validateFields() -> Bool {
    var pwOk = false
    if let pw = self.passwordField.text {
        if pw.lowercased().range(of: "traxy") != nil {
            pwOk = true
        }
    }
    if !pwOk {
        print("Invalid password")
    }


    var emailOk = false
    if let email = self.emailField.text {
      let regex = "[A-Z0-9a-z._%+-]+@[A-Za-z0-9.-]+\\.[A-Za-z]{2,}"
      let emailPredicate = NSPredicate(format:"SELF MATCHES %@", regex)
      emailOk = emailPredicate.evaluate(with: email)
    }
    if !emailOk {
      print("Invalid email address")
    }
    return emailOk && pwOk
  }

  @IBAction func signupButtonPressed(_ sender: UIButton) {
    if self.validateFields() {
      print("Congratulations!  You entered correct values.")
    }
  }
```

Listing 6: ViewController.swift: Part 1

iOS

```
54
55 extension ViewController : UITextFieldDelegate {
56   func textFieldShouldReturn(_ textField: UITextField) -> Bool {
57     if textField == self.emailField {
58       self.passwordField.becomeFirstResponder()
59     } else {
60       if self.validateFields() {
61         print("Congratulations!  You entered correct values.")
62       }
63     }
64     return true
65   }
66 }
```

Listing 7: ViewController.swift: Part 2

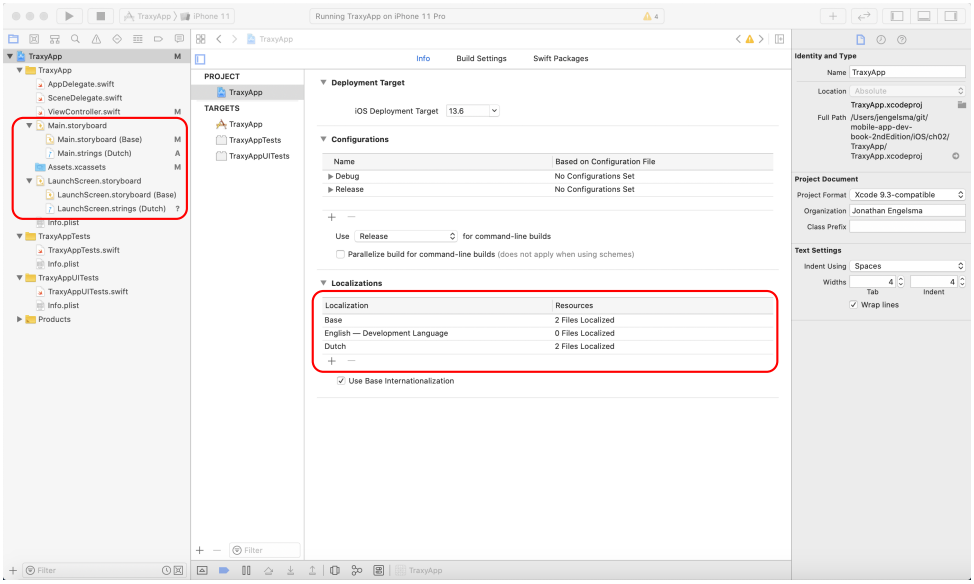


Figure 2.30: Once you've added a localization to your project, you can add translations for any string literals in your storyboard.

```Swift
/* Class = "UITextField"; placeholder = "Enter email"; ObjectID = "bqb-Lx-VuR"; */
"bqb-Lx-VuR.placeholder" = "Enter email";

/* Class = "UILabel"; text = "Welcome to"; ObjectID = "kYz-t7-Aga"; */
"kYz-t7-Aga.text" = "Welcome to";

/* Class = "UITextField"; placeholder = "Enter password"; ObjectID = "v9n-4m-KUv"; */
"v9n-4m-KUv.placeholder" = "Enter password";

/* Class = "UIButton"; normalTitle = "Sign In"; ObjectID = "wKR-bb-zl5"; */
"wKR-bb-zl5.normalTitle" = "Sign In";
```

Listing 8: Original English version of the string literals defined in the storyboard.

```Swift
/* Class = "UITextField"; placeholder = "Enter email"; ObjectID = "bqb-Lx-VuR"; */
"bqb-Lx-VuR.placeholder" = "E-mailadres";

/* Class = "UILabel"; text = "Welcome to"; ObjectID = "kYz-t7-Aga"; */
"kYz-t7-Aga.text" = "Welkom bij";

/* Class = "UITextField"; placeholder = "Enter password"; ObjectID = "v9n-4m-KUv"; */
"v9n-4m-KUv.placeholder" = "Wachtwoord";

/* Class = "UIButton"; normalTitle = "Sign In"; ObjectID = "wKR-bb-zl5"; */
"wKR-bb-zl5.normalTitle" = "Inloggen";
```

Listing 9: Dutch language version of the string literals defined in the storyboard.

```Swift
"Congratulations! You entered correct values." = "Congratulations! You entered correct
values.";
"Invalid password" = "Invalid password";
"Invalid email address" = "Invalid email address";
```

Listing 10: Contents of your Localizable.strings file will define each string literal referenced in your code.

2.4 Your Initial Android App

The basic building blocks of an Android app consist of four different components: **Activities**, **Services**, **Content Providers**, and **Broadcast Receivers**. For the moment, we will focus only on the `Activity` class and its subclasses. As the framework evolves, this class undergoes enhancements via subclassing. One of the important enhancements is the introduction of Material Design at the time Android reached API Level 21 (Lollipop) in 2014. Unfortunately, widgets released before that API level cannot display elements designed for Material Design. To address this issue, the Android team released several libraries, collectively known as the Android Support Library. One of the design goals of the Support Library is to provide backward compatibility for Android devices running lower API levels.

The Material Design guidelines stipulate use of app bar for branding, navigation, and app-specific actions. To enable app bar, you should start using `AppCompatActivity` in place of the traditional `Activity` class. This is the default for new projects created in Android Studio. Another important feature implemented by the AppCompat library is widget tinting to allow consistent theming using material color palette.

Readers with prior iOS programming experience may start to think that Android Activity class is analogous to iOS UIViewController. They both share several common tasks:

1. handle the lifecycle of activity/view controller
2. handle events from UI widgets
3. transfer control to another activity/view controller

However, unlike iOS that provides additional specialized classes (`UITableViewController`, `UINavigationController`, etc.), Android provides no such specialized subclasses.[2]

2.4.1 Creating a Project in Android Studio

Let's start Android Studio to create our first Android app. The welcome screen is shown in Figure 2.31. Select the "Start a new Android Studio project".

The next dialog, shown in Figure 2.32, is a multi-tabbed dialog, one tab per available target device. At the time of this writing, Android Studio has a tab on the dialog for each of the following five options:

- Phone and Tablet
- Wear OS
- TV
- Android Auto
- Android Things

[2]The `Activity` class indeed has a number of subclasses. But within the context of our discussion, we imply Activity to be the newer AppCompatActivity class, which has no specialized subclasses except for the deprecated ActionBarActivity.

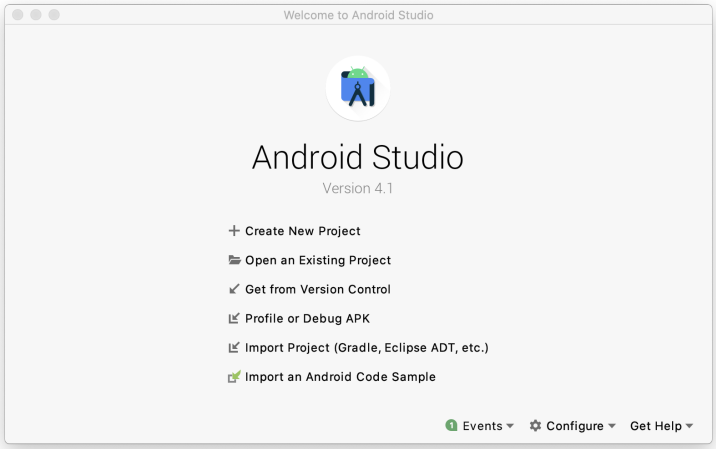


Figure 2.31: Android Studio's welcome screen. Note that if there was a previous history of projects that had been created or opened, the dialog would be split with a scrollable list of previous projects on the left.

We will select the Phone and Tablet option, which should be the default view when the dialog opens. An app needs at least one activity in order to run, so our new project dialog displays a variety of different types of activities that could be created. We will select the "Empty Activity" option[3] and then tap on the "Next" button to continue.

The next dialog screen (Figure 2.33) lets you to configure the project you are creating by entering the application name, package name, project location, programming language and minimal API level. Among these five pieces of information, pay attention to what you enter for the package name. The convention is to use an Internet domain in reverse order, in that the package name is used to uniquely identify the app once it is submitted to the Google Play Store for distribution. It is equivalent to the bundle identifier in an iOS app. The application name is a human readable text, but the package name becomes the **Java package name** in code.

For our first app, we will create a new project with the following information

- Name: **Traxy**
- Package name: **edu.gvsu.cis.traxy**
- Save Location: **a file path of your choice**
- Language: **Kotlin**
- Minimum SDK: **API 24: Android 7.0 (Nougat)**

If you are planning to publish the app to the Google Play Store, the package name must be unique among all the apps in the Google Play Store ecosystem. Once a package name is chosen and your app is published to the Google Play Store, any future updates must be released under the

[3]This activity becomes the main entry point of our application.

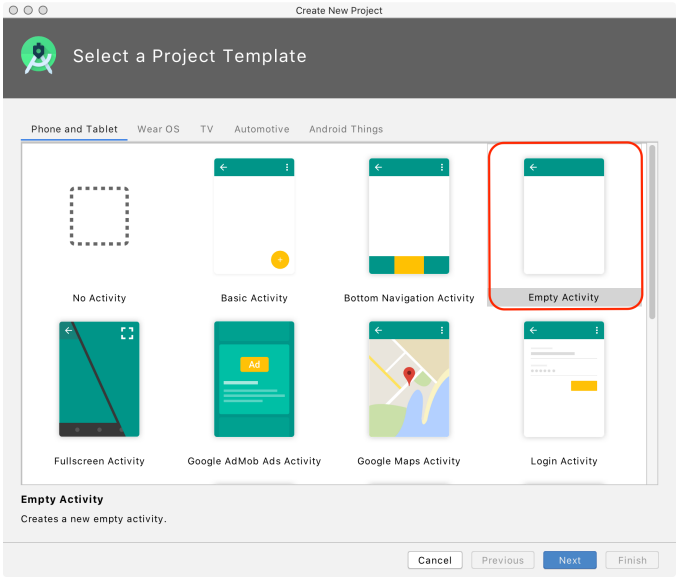


Figure 2.32: Android Studio's Starter Activity dialog.

same package name. The common practice is to use a reverse domain name for package names, so they are guaranteed to be unique.

The APK (Android Package) built for each target device can be customized to use different API level. The dialog screen in Figure 2.33 allows you to select the lowest SDK that will be supported by your app. We will select API 24 (Nougat). Lowering the API level makes your app available to users whose mobile device still runs older versions of Android, but also prevents your app from incorporating features supported by more recent SDKs. For each selected Minimum SDK the dialog shows the percentage of devices that are active on the Google Play Store. Android Studio calculates this number based on the analytics collected by the Google Play Store by actual devices visiting the store. By the time you see the actual number on your own Android Studio installation, expect to see a higher number than what you see in Figure 2.33, as more users upgrade to newer Android versions.

Pressing the "Help Me Choose" link shows the new features included in the selected API level. See Figure 2.34. The list of features shown on this informational dialog should be interpreted as incremental updates to the previous (lower) API levels.

Among many other files, the following two new files are added to your project.

1. A Kotlin class `MainActivity.kt` that extends `AppCompatActivity` (Listing 11). Despite using Kotlin for out implementation, Android Studio organizes the source file(s) under the `java` folder. Notice that the parent activity is a class in the AndroidX library.

2. The layout file (`res/layout/activity_main.xml`) is an XML that describes the layout of the associated UI screen.

The Activity class `MainActivity` contains only one method `onCreate()` which is one of the lifecycle callback functions in Android. The `setContentView()` call loads the layout file

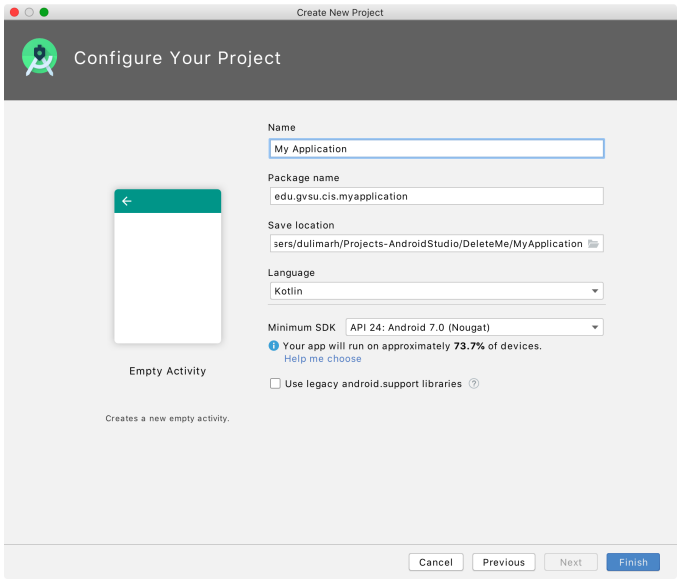


Figure 2.33: Android Studio's new project dialog.

(`res/layout/activity_main.xml`) and renders it on screen.

Kotlin

```kotlin
package edu.gvsu.cis.traxy

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle

class MainActivity : AppCompatActivity {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
      }
  }
```

Listing 11: Starter Activity Class: `MainActivity.kt`

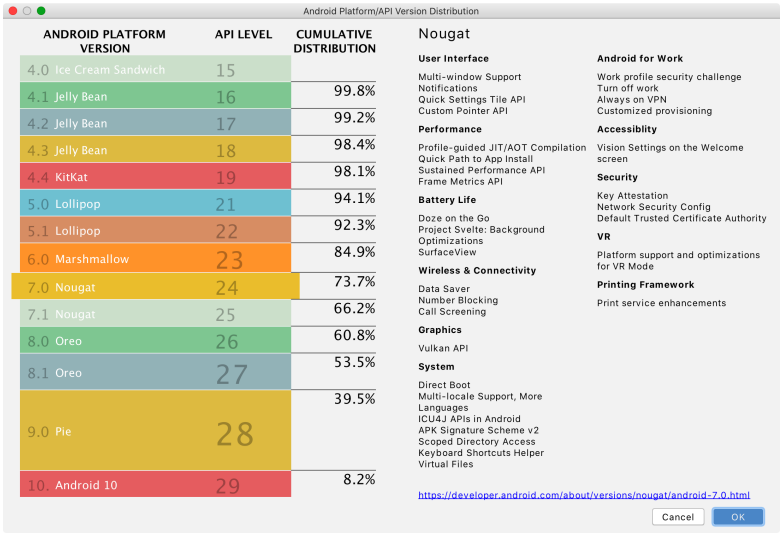


Figure 2.34: API Level distribution percentages

Running the app

Let's run the app and confirm that we do have a runnable app on either a connected device or an emulator. Refer to Section 1.5.4 if you need to create a new emulator.

> **NOTE**
>
> Bu sure to create a virtual device with Play Store enabled.

Use the top menu Run > Run app or press ctrl + R. You will see what the app is currently designed for: an empty screen with white background and an action/navigation bar at the top and a "hello world" greeting. This is a relatively uninteresting app at this point, but its an app nevertheless, that we can build and run without further modification. Let's focus now on constructing the initial login screen for our Traxy app on Android.

2.4.2 Laying out the screen in Layout Editor

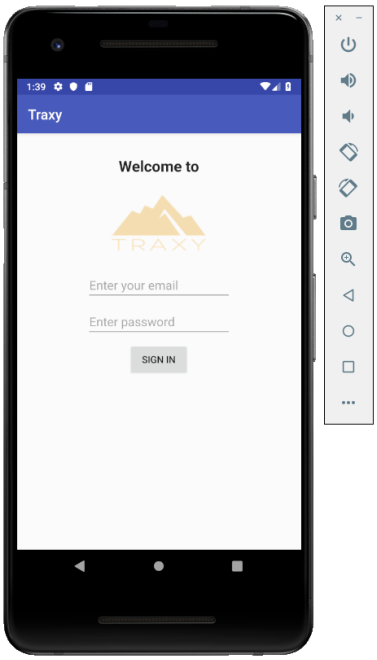


Figure 2.35: Traxy Login Screen

The final presentation of the login screen that we want to achieve at the end of this chapter looks like screenshot in Figure 2.35. It displays the following widgets:

- A `TextView` to show the text "Welcome to Traxy".
- An `ImageView` for the top image, our official Traxy logo.
- Two `EditText` (text fields) widgets for entering the email and password values.
- A "Sign In" `Button`

All these widgets are placed horizontally centered one below another. For our first Android app, we're going to assume that it will run on a Pixel 2 Android device only. We will mock it up specifically for this device in the layout editor and make it look approximately like the screen in Figure 2.35. In later chapters, we'll dig deeper into Android layouts and study how we can get our layouts to look good on any Android device.

Before diving deeper into the details of our app's layout, there are some basic concepts to be aware of. For example, every widget in an Android XML layout file must specify the `layout_width` and `layout_height` attributes which can take one of the following values:

- `wrap_content`: this widget takes as little space as needed, just enough to render its content.
- `match_parent`: this widget takes as much space as its parent

- A (numerical) dimension: this widget is constrained to use exactly the specified dimension.

With that background, go ahead and open the XML layout file `activity_main.xml` and note that Android Studio layout editor shows the canvas in two different modes: **Design** and **Blueprint** as shown in Figure 2.36. The upper left panel is a scrollable list of widgets available for your design. The lower left panel shows the component tree of widgets. Items in the component tree can be rearranged by dragging them to a desired location. To add new widgets to the current design you can drag them either to the main canvas or to the component tree. Dragging a widget directly into the component tree is a preferred technique when the widget being inserted is a child of a specific parent ViewGroup.

> **NOTE**
>
> Android Studio provides incremental search in many of its panels (besides the editor panel). Just start typing the first few letters of the item you are looking for, if the current context is searchable, the keystrokes will navigate you through the searchable items. For instance, place the mouse inside the **Pallette** panel of the layout editor. Start typing these three letters: R, E, C, it should take you to the RecyclerView widget.

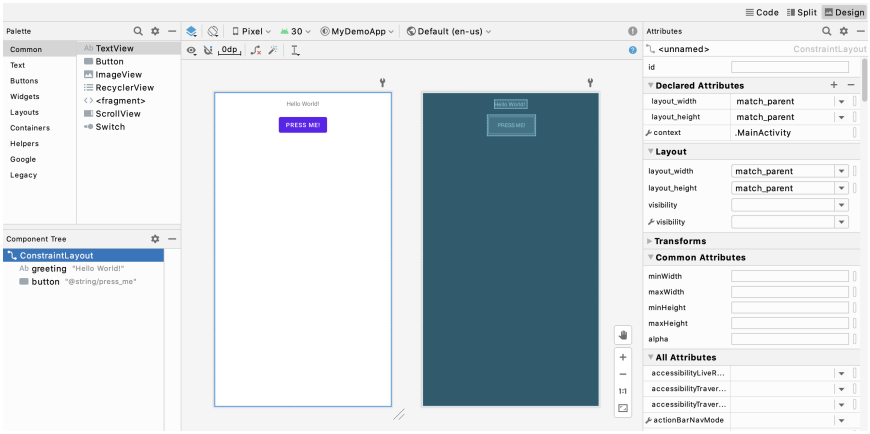


Figure 2.36: Layout Editor

We can now begin designing the login screen. Simply drag the desired widgets from the palette to their approximate position, relative to the containing parent view. On the layout editor's tool bar, be sure to select the Pixel 2 device. Later, if you run the app on a Pixel 2 virtual device, it should look exactly how it does in the layout editor.

Using Figure 2.35 as our visual guide, let's begin by deleting the "Hello World": select the TextView under the component tree panel and press the "Delete" key on your keyboard or select "Delete" from the context menu (use right-click to show the context menu).

Now add a `TextView` widget onto the layout editor canvas. As the widget is dragged towards the center top of the screen you can snap the widget to the vertical guide. In the property editor (to the right of the canvas) set the text to "Welcome to", set its text appearance to `AppCompat.Large`, and its text style to Bold.

Our next step is to insert an `ImageView` to display the Traxy logo, but first we need to add our logo image to the project. Expand the *res* folder under the project panel. (If *res/drawable* already exists in your project, skip this step and continue to the next step for downloading a JPG/PNG image). Right click on the `res` folder and select new » Android Resource Directory. Select the Resource Type to drawable. After completion of this step, you will find a new subdirectory `res/drawable`. Download the image file you want to use as the logo to this directory.

> **NOTE**
>
> If necessary, rename the image file so it contains nothing else but alphabetical and underscore characters.

Next, drag an `ImageView` below the welcome text and center it horizontally as best you can. When prompted for the source image, select the image just downloaded. Use the search box at the top of the dialog window to narrow down the list of available resources as shown in Figure 2.38. Under the properties panel to the right of the campus, change the image view's scaleType to **fitCenter** so the source image is **scaled while retaining its original aspect** and centered inside the provided rectangular area.

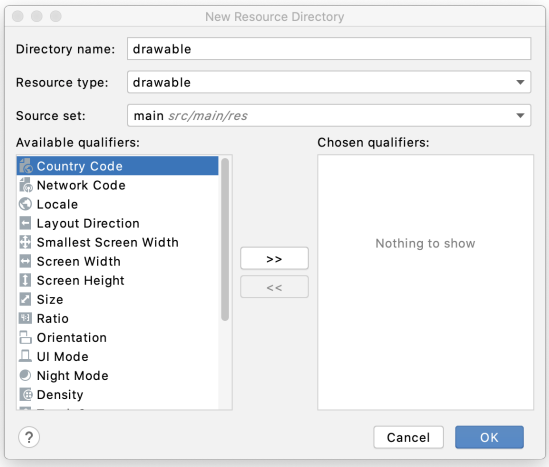


Figure 2.37: Creating a new resource directory

When a widget is selected in the Component Tree panel, the layout editor canvas should show all the constraints attached to the widget as "spring-like" connectors. For instance, Figure 2.39 shows both the left and right sides of the textview constrained to the containing parent. On the

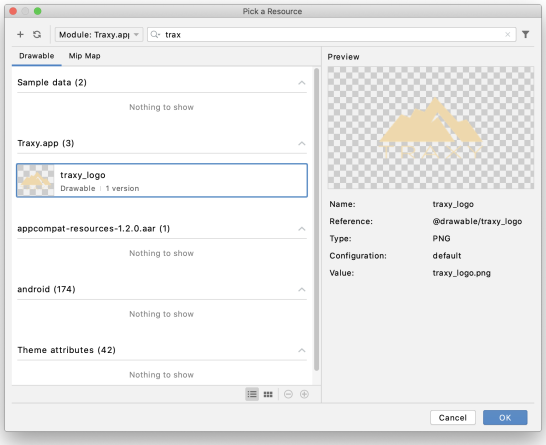


Figure 2.38: Selecting an image source

upper right corner you also see a red exclamation mark that indicates missing constraints. Click the icon to reveal the details of these errors: both the text view and the image view are missing a vertical constraint. This means that they may be rendered at incorrect positions. To add the missing vertical constraints:

- Select the text view and then click the blue plus icon in the "Constraint Widget" to the right (in the Attributes panel). By default this will add a zero-width vertical gap between the text view and the containing parent. You can change the gap distance by selecting a value from the drop-down menu

- Repeat the same step to the image view to add the missing vertical constraint.

To add the missing horizontal constraints:

- Select the text view, and click on the bubble handle on the left edge of the widget. While holding down the mouse button, drag over to the left edge of the parent. Do the same thing with the bubble handle on the right edge of the text view widget, dragging to the right edge of the parent. By default, this will center the text view horizontally.

- Repeat the same operation on the image view to center it horizontally.

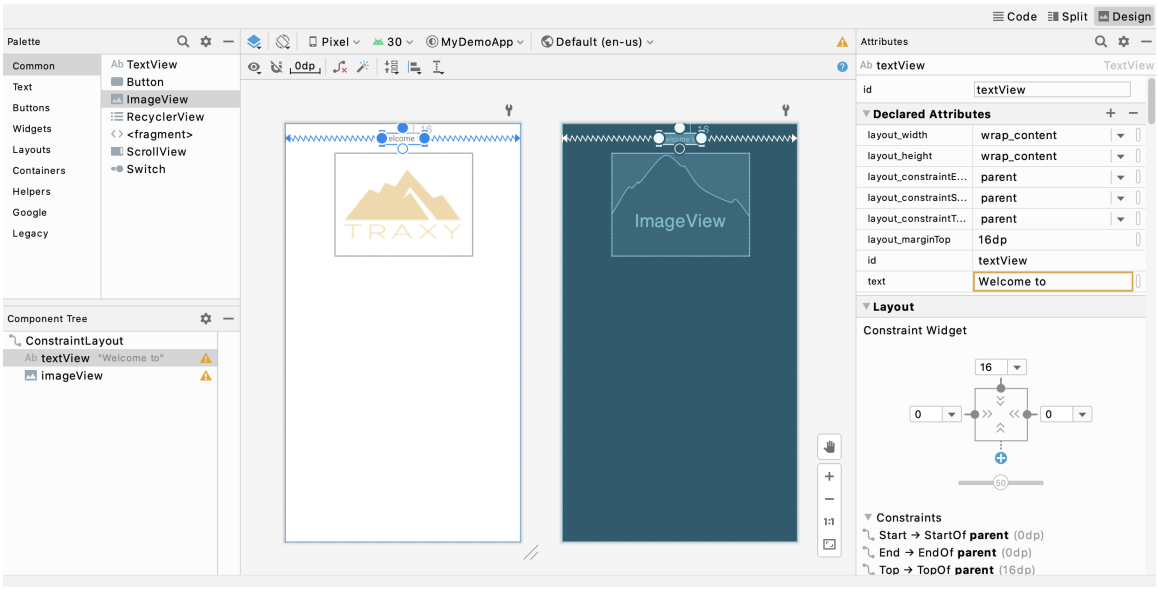


Figure 2.39: Constraints associated with widgets

Run the app to see the current screen design on a Android Pixel 2 emulator. This task can also be invoked using either the top menu Run > Run 'app' or shortcut ctrl + R. Note that if you do not yet have a Pixel 2 virtual device, you can create one with the AVD Manager which is available on Android Studio's Tools menu. If the app built and ran correctly, your screen should look similar to Figure 2.40 (except for the image).

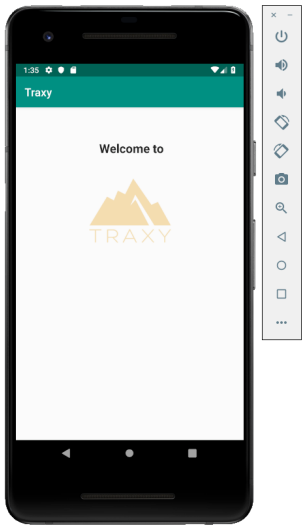


Figure 2.40: Partially Completed Traxy Login Screen

Our next step is to add the two input fields and a button to validate the email and password. Drag an **e-mail** `TextField` into the canvas below the image. Change its id to `email` and its hint property to "Enter email". The layout editor may automatically insert a vertical constraint from the top edge of the email field to the top of the containing parent. This is **not** a preferred constraint. Instead, we want a vertical gap between the logo and the email field. To delete existing constraints, hover your mouse over the anchor point of the constraint until you see the tooltip "Delete xxx constraint" and then click the tooltip. Next, drag a **password** `TextField` below the email field. Change its id to `password` and its hint property to "Enter password".

Finally, drag a button below the password field. Change the button's text to "Sign In" and its id to `signin`. Go ahead an center each of these widgets horizontally within the parent view and spaced horizontally at 16dp, as you did the welcome test view and image previously.

Now if you run the app (ctrl+⌥+R), you should see a screen that looks similar to Figure 2.35 on page 63.

2.4.3 Renaming the Activity and Validating the Login Form

Let's add some code to perform basic verification of the email and password pair when the Signin button is tapped. The completed code is provided in Listing 14. But let's work it out step-by-step.

Our first step will be to rename our Activity and layout file to something more representative. By default when we created the project, it gave us an activity `MainActivity`. Let's rename that to be `LoginActivity`. This is easily accomplished by finding the `MainActivity` class on the left in the project explorer, right click on it (or ctrl-click on Macs) and select Refactor¿¿Rename from the popup menu. This will change both the Kotlin file name as well as the class declaration and any references to it in the code. Use the same approach to change the name of the layout under the res directory from `activity_main` to `activity_login`.

The verification logic must implement the following tasks:

- Verify that both email and password fields are not empty
- The email entered follows common rules for a valid email (it includes an `@` character and the domain name contains at least one dot
- For now, we will accept the password only if it contains "traxy" (case insensitive)

Any feedback to the user will be displayed using Android `SnackBar`. To use this class, we need to import it. At the top of the `LoginActivity.kt` source file add the following important statement to make the SnackBar class visible to your implementation.

Kotlin

```
1 import com.google.android.material.snackbar.Snackbar
```

NOTE

Android Studio "Show Intention" (⌥+Enter) is a context-sensitive shortcut that can help you resolve many different situations. For instance, as you begin to use classes from the Android library, the editor may begin to flag your code with syntax errors due to missing `import`. Use ⌥+Enter to fix the error.

Obtaining References to UI Widgets

The login verification code requires a reference to the two text fields, so their text contents can be verified and checked. The `Activity` class provides the method `findViewById()` to obtain a reference to UI widgets with specific ID. Recall that the ID of the email input field in our layout file is `email`. To obtain a reference to this field from our code, we would use the following code snippet in `onCreate()`

Kotlin

```
1 val email = findViewById<EditText>(R.id.email);
2 val password = findViewById<EditText>(R.id.password);
```

- The `android:id="@+id/email"` attribute you find in the XML layout file becomes `R.id.email` in our code. This technique of resource identification will be used frequently throughout our code and applies to other resources too. For instance, a string resource whose XML id is 'msg' will be referred to in code as `R.string.msg`.

- The invocation of `findViewById()` must be done **after** the layout file is set for the activity via `setContentView()` method call.

Adding a listener to the login button

The validation of the login form described above will occur when the signin button is tapped. The button tap generates a click event that you can listen for in your Kotlin code. To setup an event handling function on the signin button, you call the `setOnClickListener` function. In Kotlin, the function can be passed as a lambda expression.

A lambda expression is essentially a concise syntax for an *anonymous method*. The syntax is so concise that access modifier, parameter types, and return type may be omitted from the syntax as long as the compiler is able to infer the details from the surrounding context. Lambda expressions provide a convenient way for writing event listener classes making our code more concise and less cluttered. Any Java/Kotlin interface that dictates only one method, such as: `ActionListener`, `OnClickListener`, is a good candidate for a lambda expression substitution.

For instance, without lambda expressions the code for setting up the click listener for a button would look like the snippet shown in Listing 12.

Kotlin
```kotlin
val signin = findViewById<Button>(R.id.signin)

signin.setOnClickListener(object : View.OnClickListener{
  override fun onClick(v: View?) {
    /* code here */
  }
})
```

Listing 12: Setting Up Click Listener Without Lambda Expressions

Using lambda expressions, the same code can be shortened as shown in Listing 13, where v is the view that generated the click event, in this case a reference to the signin button.

Kotlin
```kotlin
val signin = findViewById<Button>(R.id.signin)

signin.setOnClickListener { v ->
  /* code here */
}
```

Listing 13: Setting Up Click Listener Without Lambda Expressions

When working with lambda expressions in Kotlin, there are a number of shorthand notations that help simplify the code even more without losing any semantic meaning.

For example, the `setOnClickListener` call whose original extended syntax looks like the following

Kotlin
```kotlin
signin.setOnClickListener(v -> { /* code */ })
```

We can apply the following syntactical shorthand to simply our code:

1. Replace unused parameter `v` with an underscore. In this case, we can use the shorthand because we will not be actually referring to the view reference in our expression. In Kotlin and underscore can serve as a sort of "I don't need that" marker.

2. Place last lambda argument outside the function call parentheses. Since the lambda expression is the only argument we are passing the setOnClickListener method, we don't need the parenthesis. A similar trailing closure shorthand can be used on any method or function call where the lambda expression is the last argument being sent.

Applying these shorthands, the code becomes:

Kotlin
```kotlin
signin.setOnClickListener { _ ->  /* code */ }
```

The `when` control structure is a "generalized" `switch` statement. You can use a when statement to check for the three form validation conditions mentioned previously. Your code using the `when` statement will be patterned as follows:

Kotlin
```
signin.setOnClickListener { _ ->
  when {
    condition1 -> action1
    condition2 -> action2
    condition3 -> action3
    else -> success_action
  }
}
```

In this case, the else case will be executed whenever all three validation conditions are met. For now, you will simply display a message briefly to the user to communicate information regarding the validation of the input provided. Before API Level 21, Android developers would use the `Toast` class to show short messages. `SnackBar` is now the Android preferred way to do the same task. The `make()` method takes three arguments:

- The first argument is a reference to **any** widget on screen. At runtime SnackBar will use the provided widget to search for the root `ViewGroup` of the current screen
- The second argument is obviously the message to display
- The third argument is the duration the message will stay visible before it automatically disappears. The argument takes one of three symbolic values: `LENGTH_SHORT`, `LENGTH_LONG`, or `LENGTH_INDEFINITE`.

With that background information on the `Snackbar` component, let's take a look at how we implement each of the three validation conditions. First, you need to validate that an email address has been entered. You can implement that by simply checking if the string entered in the email field has non-zero length.

Kotlin
```
emailStr.length == 0 ->
  Snackbar.make(email, "You need to enter your email address",
  Snackbar.LENGTH_SHORT).show()
```

The second condition checks to validate that the format of the string entered in the email field is indeed a valid email address. You can use Java's support for regular expressions to accomplish this. After importing `java.util.regex.Pattern` at the top of your class, you can define the following property:

Kotlin
```
val EMAIL_REGEX = Pattern.compile(
  "[A-Z0-9._%+-]+@[A-Z0-9.-]+\\.[A-Z]{2,6}",
  Pattern.CASE_INSENSITIVE)
```

If you aren't familiar with regular expressions, you can go ahead and consult the Java Docs for the `Pattern` class. Having defined the regular expression you can implement the second condition of your `when` statement as follows:

Kotlin
```
!EMAIL_REGEX.matcher(emailStr).find() ->
  Snackbar.make(email,"Enter a valid email address", Snackbar.LENGTH_LONG).show()
```

The third condition is to check that a valid password has been entered. For now, you can just hard code the password to be "traxy". The condition is implemented as follows:

Kotlin

```
!passStr.contains("traxy") ->
  Snackbar.make(password, "Incorrect Password", Snackbar.LENGTH_LONG).show()
```

Finally, if none of the three conditions are true, the form has been successfully validated, and the user can be informed with the `else` clause of the `when` statement as follows:

Kotlin

```
else ->
  Snackbar.make(email,"Login successfull", Snackbar.LENGTH_SHORT).show()
```

Now that you have the form validation functional, let's take a look at a few fundamental techniques to improve the usability of our interface, and the maintainability of our code. In particular, we'll look at how to introduce animations to make the user interface more intuitive, and how we can extract any text the app displays from the code and facilitate the internationalization of our software. That is, if somebody installs our app in a region of the globe that speaks a language other than English, how can make sure the app presents any textual prompts into the user's local language?

2.4.4 View Animations

To make the user interface more appealing, you can add simple animations to your app. Android provides two major techniques for animations: View Animations and Property Animations. You will learn more about the details in a later chapter.

Let's add a view animation to shake the "Sign In" button when the login attempt is rejected. There are at least two different ways to implement this feature: programmatically in code or declaratively in XML. We will use the latter.

Let's create a new XML file `shake.xml` under the directory **res/anim** to specify the animation details. You may have to create the directory first with New 〉 Android Resource Directory. Be sure to select the "Resource Type" to anim on the next dialog. In a similar fashion, create the animation file using New 〉 Animation Resource File. The shake animation is implemented by translating the target view 15% to its left and right in the X direction for 200 milliseconds.

XML

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android"
android:interpolator="@android:anim/cycle_interpolator"
android:repeatMode="reverse" android:duration="200">
<translate android:fromXDelta="-15%" android:toXDelta="+15%" />
</set>
```

To activate the animation in your code you can use the following snippet which should be used *in place* of the `SnackBar` message.

Android

```
val shake = AnimationUtils.loadAnimation(this, R.anim.shake);
signin.startAnimation (shake);
```

2.4.5 String Resource Editor

Android Studio's lint tool can detect a variety or different issues and warn us about them. The layout editor in particular can display lint errors, but you need to enable this feature as it is not turned on by default when you install the software. Open Android Studio's preferences dialog and under Editor 〉 Layout Editor you will see an option labeled *Show lint icons on design surface*. Go ahead and turn this option on, if it is not already.

In the component tree panel, you will notice that some of the widgets are flagged with a warning message (exclamation point in yellow triangle). By clicking on these icons the message pane will show the details of the problem.

Most of these warnings are related to **hardcoded rext** warnings. To support internationalization, Android decouples the textual data of string literals by defining them as a separate resource object. The identity (`android:id`) of the string resource is then uses as a key to a lookup table where the actual text is defined. Using multiple lookup tables (one per language or locale), one XML layout file can be rendered differently at runtime using the selected locale/language setting of the user device.

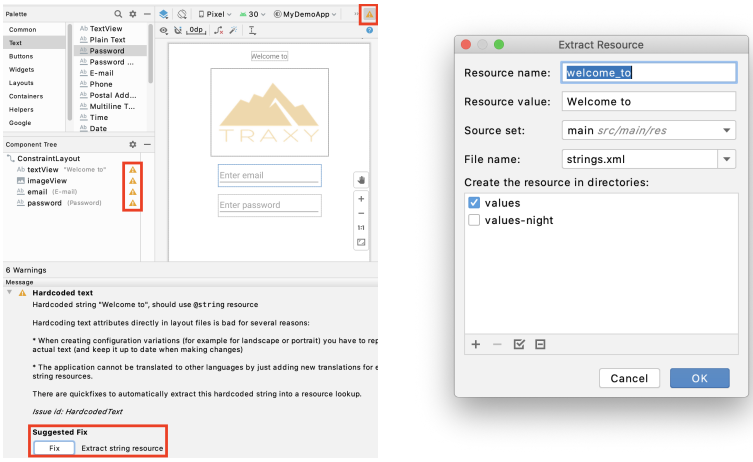


Figure 2.41: Extracting String Resource

To eliminate these warning messages, all string literals in the XML layout file must be extracted into individual string resources:

- Select the warning message to fix and select the "Extract string resource" option. A new dialog will showup (on the right of Figure 2.41). You can customize all the details in this dialog. By default, the string resource will be saved into `res/values/strings.xml`.
- Use the same steps to fix the other warning messages.

String resources can also be applied to string literals in code, especially those strings that make their way to the UI. For instance, the string literal in the following SnackBar message should be extracted to a resource:

Kotlin
```
SnackBar.make (email, "Email required", SnackBar.LENGTH_SHORT).make()
```

NOTE

Android Studio Intent menu (⌥ + Enter) shows a list of relevant actions depending on the current context of the cursor.

- Place the cursor within the string literal of a SnackBar message, then press ⌥ + Enter

- Select the option "Extract String Resource". When a dialog similar to Figure 2.41 shows up, enter the resource name `email_required` and complete the rest of the dialog. Android Studio should automatically update the SnackBar call to the following:

Kotlin
```
SnackBar.make (email, getString(R.string.email_required), SnackBar.LENGTH_SHORT)
.show();
```

If you did everything correctly, your final code should look like that shown in Listing 14. Having extracted out all of the hard-coded strings, let's now take a look at how easy Android Studio makes it to add sets of language translations for our app. Whenever your app is installed on a device, it will select the appropriate set of translations, based on the device's locale setting. Making sure you support the languages your users speak can have a significant impact on how many users decide to install and user your app.

Kotlin

```kotlin
package edu.gvsu.cis.traxy

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.view.animation.AnimationUtils
import android.widget.EditText
import com.google.android.material.snackbar.Snackbar
import kotlinx.android.synthetic.main.activity_main.*
import java.util.regex.Pattern

class LoginActivity : AppCompatActivity() {
    val EMAIL_REGEX = Pattern.compile(
            "[A-Z0-9._%+-]+@[A-Z0-9.-]+\\.[A-Z]{2,6}",
            Pattern.CASE_INSENSITIVE)

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        val email = findViewById<EditText>(R.id.email)
        val password = findViewById<EditText>(R.id.password)
        signin.setOnClickListener { v ->
            val emailStr = email.text.toString()
            val passStr = password.text.toString()
            val shake = AnimationUtils.loadAnimation(this, R.anim.shake)
            when {
                emailStr.length == 0 ->
                    Snackbar
                            .make(email, getString(R.string.email_requirred),
                                  Snackbar.LENGTH_LONG)
                            .show()
                !EMAIL_REGEX.matcher(emailStr).find() ->
                    Snackbar
                            .make(email, getString(R.string.invalid_email),
                                  Snackbar.LENGTH_LONG)
                            .show()
                !passStr.contains("traxy") ->
                    signin.startAnimation(shake)
//                      Snackbar
//                              .make(password, "Incorrrect Password",
//                                    Snackbar.LENGTH_LONG)
//                              .show()

                else ->
                    Snackbar.make(signin, getString(R.string.login_verified),
                        Snackbar.LENGTH_LONG).show()
            }
        }
    }
}
```

Listing 14: The final complete implementation of LoginActivity.

2.4.6 Internationalization

If you plan to deploy the app to non-English speaking users, it may be advisable to include other language translations.

- Right click on `res/values/strings.xml` and select "Open Translation Editors". A dialog shown in Figure 2.42 will show up.
- Press the globe button to see a list of locales and select one of them to add a new locale. Each locale has a unique 2- or 3-letter identification (e,g, no=Norwegian, ja=Japan, egy=Ancient Egyptian)

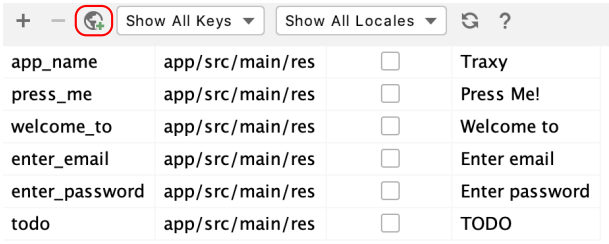


Figure 2.42: Adding a New Locale

For instance if you choose Indonesian(in) as a new translation, the translation editor adds a new column with a column title "Indonesian (in)" and all Indonesian text translations be saved into a separate file `res/values-in/strings.xml` while the default (English) strings are stored in `res/values/strings.xml`. Likewise, a Japanese translation will be stored into `res/values-ja/strings.xml`. It is important to understand that these files must use the same name (`strings.xml`)

> NOTE
>
> Android resource files are organized into sub-directories with predefined suffixes to store different variations of the same resource. At run-time, the Android framework uses these suffixes to resolve a resource name based on the current configuration settings the user selects.

2.4.7 Source Code Revision Control

This is perhaps a good time to start maintaining your source code under a revision control software such as **git**.

- Select the top menu VCS > Enable Version Control Integration and select **Git** from the list
- After the action is confirmed, a new collapsible tab (Version Control) will show up at the bottom. Open this tab to see more details.
- Press the tab labeled Commit on the left border of the screen (right under Project) to commit the staged changes.
- Click on "Unversioned files" check box to add the files to Git staging index.
- Enter a commit message in the next dialog before pressing the **Commit** button.

—The mother art is architecture. Without an architecture of our own we have no soul of our own civilization.

Frank Lloyd Wright

3 Mobile Application Architectures

Chapter Objectives

After reading this chapter you should be able to

- Explain the interaction mechanisms between a mobile app and its host operating system
- Understand the basic lifecycle of an Android `Activity`.
- Understand the basic lifecycle of an iOS `UIViewController`.
- Understand the default software architectures used by iOS and Android applications.

3.1 Software Architecture in a Mobile Environment

When working in a desktop environment, most users might have several applications running at the same time. These applications execute concurrently and are managed by a multitasking operating system (OS). To the host operating system, each application is managed as a process that requires resources to complete its assigned tasks. When these processes are competing for the same set of resources, it is the job of the host OS to manage resource allocations and deallocations to avoid usage conflicts. Let's consider a particular case when two processes produce sound output to the audio system. One of these two processes is currently streaming music, and the other needs to chime a message notification. To allow sharing of the audio system, the operating system must provide an arbitration mechanism to allow the two processes to share the audio system. For this particular example, it makes sense to assign a lower priority to the process streaming music and

a higher priority to the process that needs to send a very short audio notification. The process manager of the host operating system will then be able to resolve how these two processes should share the audio system.

A similar arbitration mechanism can be applied to other components in the system. Let's consider a typical case when the user enters input in a multi-window environment. Although the user sees several windows active on screen, only one of them can interact directly with the user. More specifically, several windows may be actively running and producing output, but to obtain user inputs a window must acquire either the keyboard or mouse focus.

3.1.1 Process State Transitions

A multi-tasking operating system manages its processes using a process model that assumes an "active" process is not always using the CPU throughout its entire lifetime. There are times where a process – although it remains active in the system – is not running on the CPU. It either voluntarily releases control of the CPU (such as when a process needs to wait for user inputs) or is preempted from the CPU due to an external events (such as preempted by a higher priority process). Essentially a process can either be "running" or "not running". However, the operating system scheduler needs a more detailed information whether a "non-running" process is ready to be scheduled to use the CPU or must wait for some external events. For the scheduler, the 2-state model is insufficient and a 3-state model (enclosed in box) is used instead.

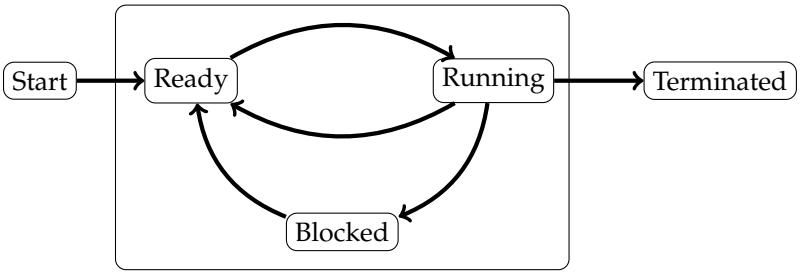


Figure 3.1: Process State Transitions

Throughout its lifetime, each application may undergo these state changes several times in response to either user actions, system-related actions, or specific operations initiated by the application. For instance,

- when the CPU becomes idle, the operating system may select one of the "Ready" applications to use the CPU
- when the user of an application initiates a data download operation, its state changes to "Blocked", it will turn to "Ready" when the download completes
- an application that attempts to write data to a shared file may have to wait (changes state to "Blocked") when the file is currently in use by another application
- when an application requests more memory, it may have to wait until adequate amount of memory becomes available

Figure 3.1 is a *much simplified* model of the state transitions of a real process. In a real system, you will also find other types of state changes which are not captured by the diagram. For instance, in a multi-window environment bringing a window to foreground (background) may increase (lower) its priority. Furthermore, the state transitions of a multi-threaded application cannot be precisely described using a single transition diagram, each thread has its own state transition independent of the others. Nevertheless, the 3-state model above adequately describes the fundamental of process management in both batch and interactive systems.

In an interactive system like a desktop GUI environment, a window manager runs as a layer that separates application windows from the host operating system. In such an environment, it is common to find one process hosting more than one window. User actions on any of these windows affect the state change of the hosting process; another circumstance which cannot be captured by Figure 3.1.

There is a clear separation of concerns between the window manager and the operating system. Resource management operations, low-level I/O operations (disk, socket, etc.), and memory management operations initiated by a GUI application are directly handled by the host operating system, but user inputs (from the keyboard), mouse actions (click, drag, etc.), and individual windows output are mainly handled by the window manager.

An interactive GUI environment also dictates a programming model different from that of console-based applications. The event-driven programming model used in a GUI environment requires a main event-loop that dispatches each incoming event to its corresponding handler. These events fall under several categories:

- Window-related: create, move, minimize, maximize, close
- Mouse-related: click, drag, move, enter, exit
- Keyboard-related: key-up, key-down, key-press,
- Timer-related: alarm, timer expires
- High-level events: item selected, scroll up, scroll down, touch gestures (swipe, fling, pinch, multi-touch, . . .), etc.

In a modern GUI event-driven framework such as: Swing, Cocoa, Gtk, Xaw, etc., the main event-loop mentioned above is subsumed by the framework, application programs are not required to write the event-loop themselves. To handle events of interest, an application program must provide selected *callback functions* (also known as framework *hook methods*) associated with those events. For instance, an application might be interested in getting a notification for events when the user minimizes a window, swipes inside a window, or taps on a menu, etc. To keep the user interface responsive at all time, the event-loop runs on a dedicated thread (usually called the UI thread), while non-UI related tasks are handed over to separate (background) threads. This *inversion of control* is commonplace in modern GUI frameworks, including the respective UI frameworks of Android and iOS. The result is an increased programmer productivity with an upfront cost of a slightly steeper learning curve.

3.1.2 Mobile Resource Constraints

A mobile environment shares much resemblance with a desktop GUI environment, except the former is significantly resource-constrained. Nevertheless, techniques applied to a desktop GUI

environment may also apply to such a resource-constrained mobile environment. The most prominent constraints in a mobile environment are screen space and storage space. Consequently, a mobile environment would allow only one visible application at any time[1]. When an application is brought to foreground and becomes visible, it hides a previously visible application making it invisible. A mobile app is more involved in handling this kind of transitions than handling the state changes of the hosting process as described in Section 3.1.1. For instance, a mobile application is more interested in knowing whether it is currently **visible**, **invisible**, in **foreground**, or **background**. It does not care so much whether the application is currently **running** or **blocked**.

In addition, storage constraint may limit the number of active applications that can reside in memory at a time. In extreme case when the amount of available memory is dangerously low, the operating system may decide to terminate an application and seize its memory to allow system to recover from the memory shortage.

Mobile environments are also be more resource constrained than desktop or machine room computing environments with regard to energy (e.g. battery powered), processor, and available network service.

3.1.3 Application State Transitions

In order to manage the notion of single application in focus and visible at a time, both iOS and Android introduce their respective application lifecycle models. For instance iOS defines the following application states:[4]

- **Active**: the app is running in the foreground and is interacting with the user
- **Not Running**: the app has not been launched or was terminated
- **Inactive**: the app is running in foreground but is not interacting with the user (an app usually stays in this state only briefly as it transitions to a different state)
- **Background**: the app is in the background and executing code. Most apps enter this state briefly on their way to being suspended, or the app itself is launched directly into background (for instance, to play streaming music in background)
- **Suspended**: The app is in the background but is not executing code.

The transitions among these five iOS states are shown in Figure 3.2.

The corresponding definitions for Android applications are slightly different. Each of the following four states of an Android process is based on the state of the currently hosted activity:

- **Foreground**: when its current activity is at the top of the screen, interacting with the user. This state is equivalent to the iOS Active state.
- **Visible**: when its current activity is partially visible, and not interacting with the user. This state is equivalent to the iOS Inactive state.
- **Background**: when its current activity is not visible and has been paused. This state is *almost* similar to the iOS Background state, except an iOS process can directly run in background without being associated to a view controller.

[1] Android 7.x (Nougat) is an exception, it features a split screen mode where two apps can co-exist.

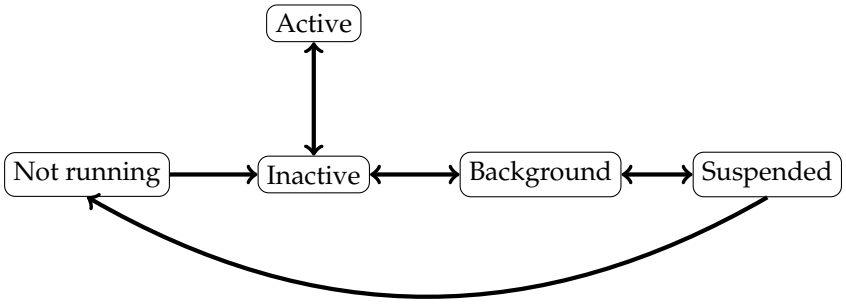


Figure 3.2: iOS Application State Transitions

- **Empty**: when the process is not hosting any activity (the activity has terminated.)

The application states defined by both platforms seem to emphasize the transitions from and into the visible state (or being in foreground/background) of a process. In the following sections we will observe the effect of the screen space constraint to the visibility state of various components of an app.

With the abundant amount of screen space at its disposal, it is not uncommon for a desktop application to partition its window into several subviews, each displaying specific information to the user. The screen space constraint imposed on a mobile app requires the user interface to to be organized differently than a conventional desktop application. The user experience must be broken up into smaller chunks each of which can fit on a single screen. Each of these small screens is managed by a separate component in the app, and the user is required to navigate from one component to the next.

For instance, when reading emails with a desktop application, a user is usually presented with both the list of emails and the detailed contents of a selected email in an adjacent view. Moreover, when the email has a media file attachment, it may also be rendered inline in the same view. On the contrary, reading emails on a mobile device requires the user to navigate from one screen that presents the list of emails, to the next screen that shows the detailed contents of an email when selected, and yet to another screen for rendering a media attachment.

Though conceptually similar, iOS and Android introduce their own abstractions for implementing the user interfaces of screens. On iOS the `UIViewController` class forms the basis for instantiating and managing the user interface of a given screen within an application. On Android we use the `Activity` and `Fragment` classes. These design restrictions require developer extra effort in passing information among the components. A single user task that can be completed on a medium-sized "form" in a desktop application may have to be broken down into several mini screens, with all of them having access to a common data source.

Cascading navigation style such as in the reading email example above, may increase the complexity of data management and error handling. The data needed to complete a single task are entered by user across several screens, but the task can only be completed when the user is at the last screen. Therefore, there has to be an efficient mechanism for carrying over the partially acquired data from one screen to the next. Error handling is also another issue that requires special attention. How do we back out efficiently when amidst the cascading navigation, the user decides to cancel the current task or encounters an error.

iOS

```
#import <UIKit/UIKit.h>
#import "AppDelegate.h"

int main(int argc, char * argv[])
{
    return UIApplicationMain(argc, argv, nil,
        NSStringFromClass([AppDelegate class]));
}
```

Listing 15: Main function of an iOS application.

After you learn the state transitions of iOS/Android applications, in the next section you will see how an iOS application launches its view controller. Likewise, you will also see how an Android application is spawned as a new Linux process until its first Activity takes control.

3.1.4 Application Process: a Home for UIViewController/Activity

Historically, iOS has a technical pedigree that goes back to AT&T Unix and can be summarized by the following lineage: iOS → OS X → NeXTSTEP → BSD Unix / Mach Kernel → (AT&T) Unix. Likewise, the Android framework has a very strong tie to Linux, an open source Unix-like operating system. In both operating systems, an application executes within a process that originates as a main function somewhere in the system.

The main entry point of an iOS application is essentially a `main()` function that instantiates a `UIApplicationMain` object as shown in Listing 15. The main tasks carried out by this object include: loading the user interface from the storyboard file and running custom initialization logic from the `AppDelegate` class. The `UIApplicationMain` object controls the event loop and the `AppDelegate` manages all the View Controllers of the app. In conjunction with the `UIWindow` class, these two classes manage the MVC (Model-View-Controller) triads within the app.

For an Android application, the main entry-point of its associated Linux process also instantiates an instance of the Java Virtual Machine running the Android Run Time (ART), an improved version of Dalvik. But how does a user process get spawned by Android or iOS? In order to answer this question, we should look at the high-level view of Linux/Unix process management and the execution of Java Virtual Machine.

3.1.5 Java Virtual Machine

Developers with knowledge of Java programming will probably recall the two command line utilities used in Java programming: `javac` (the Java compiler) and `java` (the Java VM). The typical sequence to compile and launch a Java program from the command line is:

```
javac SampleAp
java SampleApp            # load SampleApp.class into JVM
```

Compiling and running Kotlin programs follow a similar command sequence:

```
kotlinc SampleApp.kt
kotlin SampleAppKt            # load SampleAppKt.class into JVM
java SampleAppKt              # load SampleAppKt.class into JVM
```

Since the Kotlin compiler generate its output in Java byte code, the resulting `.class` file can be run by JVM. The last two commands are more relevant to our context of discussion here; each command starts a Java Virtual Machine and loads `SampleAppKt.class` into the VM. Now let's back up a step prior to that. Assuming the `java` command is entered from a Linux terminal, a new process is spawned (via `fork()`) by the terminal program to execute the JVM, which in turn executes `SampleAppKt`. When you are actually typing the command from a terminal you may have noticed that loading the JVM may take seconds.

Let's examine the technique used by the Linux kernel in an Android environment to speed up JVM initialization time. At boot time, the Linux kernel creates a special process, `init`, which is in charge of spawning all other processes in the system. This initialization procedure applies to all Linux systems, regardless of its specific environment (Android or non-Android). However in Android, `init` also spawns a special Linux process named *Zygote* that runs the first instance of the Java Virtual Machine. The JVM initialization logic preloads the Android core library classes. Subsequent Android processes in the system are spawned by Zygote, and therefore they inherit both the JVM already running in Zygote and the preloaded Android core libraries. This technique speeds up the creation of subsequent Android processes that benefit from the initial JVM warm start. This also explains why some Android runtime error messages mention Zygote in its detailed error description.

When a new application is launched, the Android runtime system obtains the details about the application from its manifest, which includes information on the application class and the main launcher activity. A new Android processes must first be created to load the application and a request is sent the Zygote process to create it. The new child process will then instantiate a Java object `ActivityThread` which is, by default, bound to `android.app.Application` or the application class mentioned in the manifest. The `ActivityThread` then sends a request to the Activity manager to instantiate the activity and invokes its `onCreate()` method. This sequence essentially shows how an Android Activity receives control from the host operating system.

3.2 Choices of Application Architectures

The modular design of mobile frameworks requires a clear separation between the view (the "user facing" elements) and the model (the "data handling" elements). Several application architectures have been proposed in the spirit of separating the model from the view. Some of the widely-adopted ones are:

- Model-View-Controller,
- Model-View-Presenter, and
- Model-View-ViewModel.

The three architectures seem to agree on which components make the view of an application, but their definition of "model" slightly differ from each other. Obviously, the major difference among

the them lies in the "middle glue" that connects the model to/from the view and level of isolation between these two components.

Model View Controller

The MVC pattern used by Apple is slightly different from the traditional MVC. In the variant implemented by Apple, the View and the Model are disconnected from each other. In this architecture, the Controller is a bridge/mediator between the Model and the View. User actions on the View trigger events that must be handled by the Controller, which in turn translates these events to operations for updating the Model. When this pattern is followed closely by iOS developers, they should use the view controller classes only for converting UIView events to corresponding operations on the (data) model. However, what usually happens is the model consists of classes with only setter and getter methods, lacking any application logic. Even worse, the data is stored directly inside the view controller classes themselves, forcing them to implement the application logic as well. As the result of this mismanagement, many iOS developers jokingly refer to MVC as **Massive View Controller**. The view controller classes are responsible for handling several different tasks:

- Handle events from the UIView classes
- Manage the model object(s)
- Implement the application logic
- Update the view

As a result, the separation among the three components: model, view, and controller becomes blurry and unit testing is difficult to implement. Most of the code that is written is in the view controller classes which tend to get very large and hard to maintain.

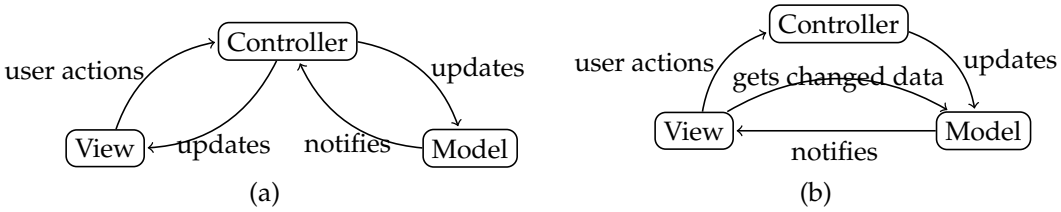


Figure 3.3: Model View Controller: (a) Apple's Variant, (b) Traditional

- The Model handles the application logic and data
- The View talks to both the Model and Controller
- The Controller has a reference to the View and to the Model

In contrast to the iOS approach, Android does not impose any specific software architecture or design pattern on how the developer implements applications. However, both iOS's `UIViewController` and Android's `Activity` share common roles in regards to handling events

and updating the view; they both take the rome of the View and the Controller. Therefore, it is very common for Android developers to be plagued by the same mismanagement pitfall encountered by iOS developers. The Android design team attempts to mitigate this issue by providing several "wrapper" libraries called "Architecture Components" released as part of the JetPack bundle. You will find more details about these libraries in a later section.

To facilitate testing and make code more manageable, several other architectural patterns have been proposed: Model-View-Presenter, Model-View-ViewModel, and VIPER. Let's briefly look at each of these architectures.

3.2.1 Model View Presenter

In both iOS and Android, the view can be very tightly-coupled with its `Activity` / `UIViewController`. A good example is the iOS `UITableView` and Android `RecyclerView` pair for displaying a list of data items. In managing this particular iOS view object, the view controller is required to implement the data source/list manipulation delegate. Likewise, Android's `RecyclerView` requires an adapter class which is essentially equivalent to iOS data source delegate.

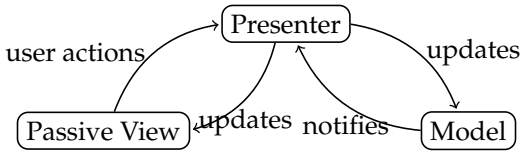


Figure 3.4: Model View Presenter

- The Model handles with the application logic and data
- The View has a reference to the Presenter and notifies user actions to the Presenter
- The Presenter has a reference to the View so it can direct the View what to display. It also has a reference to the Model. Upon receiving a notifications of user actions, the Presenter decides what to do next and direct the Model to update or fetch new data

In this architecture, each View is typically associated with a Presenter and the contract between the is capture using interfaces.

When the View and Controller become inseparable, the MVC architecture is missing the bridging element between the View and the Model. The MVP architecture fills-in this missing mediator with a new role: the Presenter. To accommodate the tight-coupling between views and their associated controllers, MVP coalesces both the View and Controller into a new element: **Passive View**.

3.2.2 Model-View-ViewModel

Model-View-ViewModel is essentially an MVP variant that incorporates data binding between the "View Model" (or "Presenter") and "(Passive) View". In the MVP, the relationship between the Presenter and View can be considered as producer-consumer relationship: the Presenter is producing data consumed by the View. In this case, the Presenter does not really care who will

consume the data; it can just produce a stream of data and whoever interested can "bind" or "subscribe" to the data stream. Therefore it is not necessary for the Presenter to have a reference to the View. When this interpretation is chosen, we actually are in the Model-View-ViewModel world. The main motivation here is to make the View free of application specific data, and let the ViewModel deals with the data. The implementation of data binding may require a third-party library (RxSwift, RxBindings, ButterKnife, . . .)

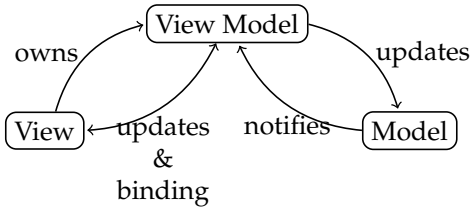


Figure 3.5: Model View ViewModel

The Android Architecture components are well-suited for applications built using the MVVM architecture.

3.3 Lifecycle of `UIViewController` and `Activity`

Both Android and iOS employ callback functions for delivering state change notifications to their applications. This technique is made possible by inheritance and subclassing. Android requires all its applications include one or more subclasses of the `Activity` class, while iOS enforces the callback mechanism via its `UIViewController` class. On their respective platforms, these two classes serve as a base class that provides several hooks for an application to be cooperative in handling system-related events.

The naming style of callback functions chosen by Android and iOS differ, but they are essentially designed to guarantee that our app is cooperating well with the system and other apps. By receiving notifications of system-related events a mobile app is able to coexist with other applications on the system. For instance, an app that is currently controlling a short animation is able to pause it when the app is about to be deactivated. The Android naming style uses the `on` prefix while the iOS naming style uses word phrases that include `will` or `did`.

Table 3.1 is our attempt to compare different stages of iOS lifecycle with their counterpart in Android lifecycle. The table may not accurately capture the correspondence between the two frameworks, but our intention is to show the somewhat similar approach used by both frameworks in communicating system events to their apps.

The state transitions within each respective platform are shown in Figure 3.6 and Figure 3.7. The two states in Android lifecycle: Started and Resumed correspond to an Activity being **visible** on screen, while in the Stopped state an activity is completely invisible. In between the two, when an Activity is Paused, it is partially visible. It is interesting to observe that both platforms share the notion of "visible"/"appeared" and "invisible"/"disappeared".

Figure 3.7 embeds two groups of information: the nodes (in rounded rectangle) correspond to (sub)states and the edges are labeled with callback function that is invoked the each state transition

Android method	iOS method	Invoked when Activity/View Controller
	viewWillLoad	
onCreate	viewDidLoad	was just created
onStart	viewWillAppear	is about to become visible
onResume	viewDidAppear	will start interacting with user
onPause	viewWillDisappear	is about to become invisible
onStop	viewDidDisappear	is no longer visible
	viewWillTransition	
onSaveInstanceState	willRotate	
	willAnimateRotation	
onRestoreInstanceState	didRotate	

Table 3.1: Android and iOS callback methods

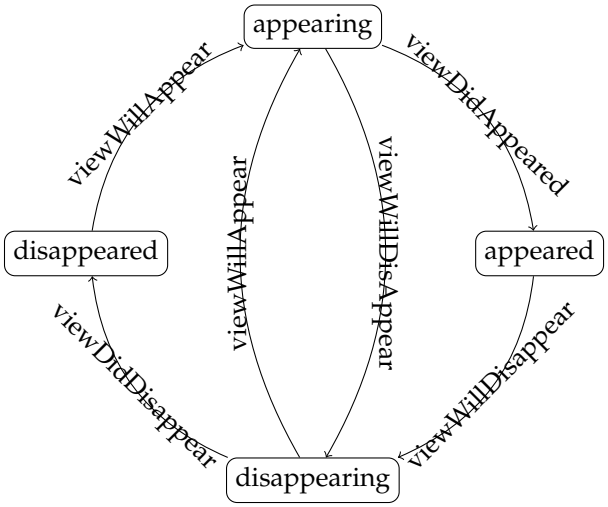


Figure 3.6: Lifecycle of iOS UIViewController

takes place. Out of the seven states (including "None") only four have significant meaning for an Android Activity:

- **Active/Running**: the activity is in the foreground of the screen. This state is reached when the callback method `onResume()` is invoked
- **Paused (iOS Inactive)**: the activity has lost focus but is still visible. This state is reached when the callback method `onPause()` is invoked. This state is possible because Android allows a non-fullsize or transparent activity to obscure another activity.
- **Stopped (iOS Background)**: the activity is completely obscured by another activity. This state is reached when the callback method `onStop()` is invoked.
- **Restarted/Restored**: an activity that was stopped or resumed, then kicked out of the system memory, but got resumed and redisplayed to the user.This state is reached when the callback method `onRestart()` and `onResume()` methods are invoked (in that sequence)

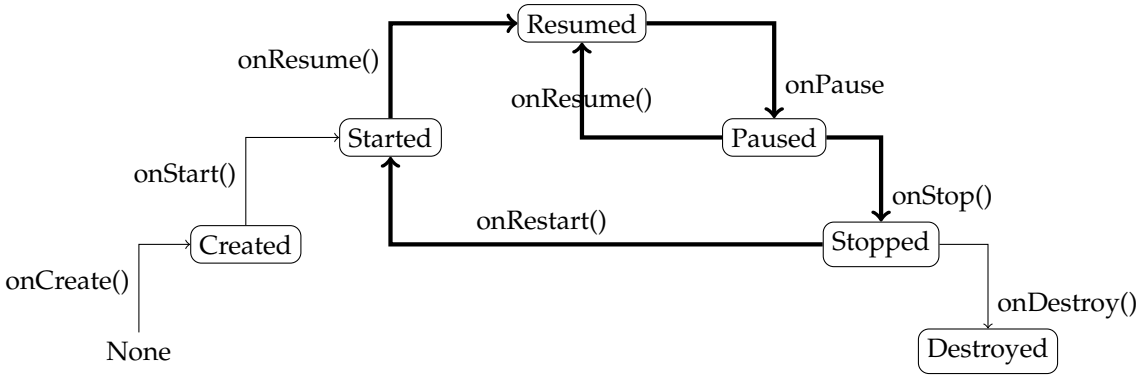


Figure 3.7: Lifecycle of Android Activity (Image credit: https://developer.android.com/training/basics/activity-lifecycle/starting.html)

3.4 Android Manifest

The overall structure of an Android app is described in an XML file `AndroidManifest.xml`. It includes the package name that uniquely identify the app, special permissions required to run the app, the application theme, various components that make the app, application main entry point, etc. A sample of Android manifest file is shown in Listing 16. The package identifier is declared as an attribute of the `manifest` tag. Lines 5–7 list the (runtime) permission needed by this application. Lines 17–26 defines an `Activity` object of the app. The activity object will be instantiated from the `MainActivity` class. The intent filter at lines 21–25 indicates that this activity is the main entry point. The actual manifest that gets built into the Android Package (.apk) contains more data than what you see in the listing. For instance, relevant information about the build found in `build.gradle` are also incorporated into the final manifest.

An Android application typically consists of one or more `Activity` classes and each activity can host one or more Fragments.

XML

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="edu.gvsu.cis.traxy">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity
            android:name=".MainActivity"
            android:label="@string/title_activity_main"
            android:theme="@style/AppTheme.NoActionBar"></activity>
        <activity
            android:name=".SignUpActivity"
            android:label="@string/title_activity_sign_up"
            android:theme="@style/AppTheme.NoActionBar" />
        <activity android:name=".LoginActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

Listing 16: Android Manifest

3.5 Android Architecture Components

Prior to Android 8 (Oreo) in 2017, Android developers had to deal with writing several lines of code in lifecycle methods, typically in `onStart()`, `onResume()`, `onPause()`, and `onStop()`. These lines of code are required when the application or activity requires background services, broadcast receivers, or other general background tasks. Most of the time, the code written in one of these lifecycle methods come in pair. For instance a background service initiated in `onStart()` (or `onResume()`) must be properly terminated in `onStop()` (or `onPause()`). Adding to this complexity is the possibility that a currently running application can be interrupted by many reasons: notifications, incoming phone calls, text messages, battery warning, the user manually switches to a different application, change of device orientation, etc. To resume a previously interrupted task, an application may need to reinitialize it to the state before the interruption took place, and the Android lifecycle provide the methods `onSaveInstanceState)` and `onRestoreInstanceState()` for this purpose.

To tackle this complex handling of background tasks and device rotation, the Architecture Components libraries provide a facility to manage the lifecycle in an easer way. Some classes under these libraries are designed to be lifecycle-aware. It is important to note that thee use of these libraries are not mandatory, however these classes allow you to implement an application using a cleaner design principles. Specifically, it is assumed that the application is designed based on the MVVM architecture with the following separation of tasks:

- The View is responsible for displaying data to the UI and capturing user interactions on the UI
- The ViewModel is responsible for holding all the UI data needed by View
- The Model is responsible for handling the overall application data

The main elements of the Architecture Components include the following libraries:

- Lifecycle library: provides a "wrapper" for easier management of the Android lifecycle. The main classes of this library are:
  - `Lifecycle`: a class that captures the Android lifecycle
  - `LifecycleOwner`: is an abstraction for objects with lifecycle (Activity or Fragment)
  - `LifecycleObserver`: for observing changes in lifecycle
- LiveData library: implements the ViewModel element of the architecture whose main responsibility is to hold the UI data but also it survives configuration changes, such as device orientation changes or switching locale settings. One of the unique features of `LiveData` is its observability; changes to the underlying data model can be observed and the application will receive notifications on data updates
- Room library: handles local data persistence using SQLite. This library provides classes for writing annotations in classes that interface with the SQLite database on the device.

Combining the features of the three libraries above, they all turn a conventional relational database into a "reactive" and persistent object model and the data reactivity is managed in sync

with the Android lifecycle. We speculate that the word "Live" is used in the name of one of the libraries to contrast with traditionally *passive* data.

In addition to the main libraries mentioned above, the Architecture Components libraries also provide the following additional libraries:

- `Paging` library support handling large amount of data where they may have to be fetched lazily one "page" at a time or one a small(er) manageable pieces
- `WorkManager` library to support running concurrent tasks in the background. This library is designed to supersede the older utilities such as `AsyncTask`, `Thread`, `ThreadPool`, etc.
- `Navigation` library for handling UI transition from one view to another by using a navigation graph.
- `Data Binding` library which allows UI contents to be updated automatically by making the UI elements themselves as an observer of a `LiveData`.

4

Understanding Screen Transitions and Navigation Basics

Chapter Objectives

After reading this chapter you should be able to

- Understand the similarities and differences in how Android and iOS handle screen transition.
- Utilize activities and intents to implement an Android app with multiple screens.
- Utilize segues within a storyboard to implement an iOS app with multiple screens.

4.1 Overview

There are a number of useful utility-style apps that consist of a single screen. Flashlight and calculator apps are two common examples come to mind. However, most apps define a number of screens that the user transitions between as they engage in the app. Having developed a better understanding of the details of how Android and iOS applications are architected, it is now time to extend our basic single login screen app with additional screens. In particular we will look at how Android and iOS applications incorporate multiple screens and transition among those screens during the application's lifetime.

4.2 Fleshing out Traxy for Login and User Signup

Our rudimentary Traxy app so far consists of a single login screen. Let's take a look at how we can extend the app with a couple of new screens:

- **Sign Up Screen:** If users install our app and don't already have an account, this screen will allow them to create a new account.
- **Main Screen:** Once users have logged in with an account either via our original login screen, or the new signup screen, they will land on this screen. At the moment this is just a placeholder screen which we will flesh out in more detail later. For now we will simply display the current user's identity and a logout button.

There is a number of transitions among these three screens, as demonstrated in Figure 4.1. A button is added on the bottom of the original login screen that when tapped allows the user to signup for a new account. Once on the signup screen if the user taps cancel, they will be returned to the login screen. On that same screen if they properly fill in the form and press the Sign Up button they will land on the main screen where their email address will be displayed. If they are on the login screen and they fill in the form correctly the Sign In button will land them on the main screen. Finally, if they are on the main screen and they tap the logout button, they will be returned to the login screen with the fields blanked out. The login screen remains the initial screen the user will see when they login.

Both iOS and Android of course allow us to implement these sorts of transitions. However, as we will soon see, how we accomplish this on the two platforms, and the terminology used will differ slightly. Let's go ahead and implement this scenario in both iOS and Android. For now, we won't worry about formally validating a user's credential against some sort of identity management system. For now, on the login screen we will continue with the same validation steps we implemented previously (valid email address entered, and valid password). On the sign up screen we will implement both of those requirements, and will also check that the password verification field matches the previous password field.

4.3 Scene transitions in iOS

In our previous work on the iOS Traxy app, we never had to actually do anything to get our login view controller to appear on the screen. By default, it was the initial scene in our view controller, so it magically appeared when we ran our application. As soon as we introduce additional screens to our iOS app, we have to somehow get the new view controllers to present.

On iOS, there are in general a couple of ways to get a view controller to appear on screen and manage when it is or is not visible. One approach would be to contain the view controller of interest into a *parent* or *containing view controller*. A container view controller typically manages one or more *child view controllers*. The container view controller not only displays the child controller on screen, but can also help manage the transition from one child view controller to another. The UIKit framework contains a number of these specialized container view controllers. For example, `UINavigationViewController` manages a stack of child view controllers. We will look at the the details of this class and other container view controllers in a later chapter.

Another approach that can be used to get a view controller to appear on screen is to have a view controller that is already on screen to present it. The `UIViewController` class has baked



Figure 4.1: Extending the Traxy app to onboard new users. These are the iOS screens without any styling applied. The Android screens will be functionally equivalent, but with some minor cosmetics differences.

within its implementation, the ability to present another view controller instance. In this case the view controller that does the presenting is referred to as the *presenting view controller*, and the view controller that is being presented is referred to as the *presented view controller*.

In terms of implementation, you can initiate the presentation of a view controller using a couple of different techniques. First, you initiate the presentation entirely in code by calling methods provided by the `UIViewController` class. Second, you can use a storyboard *segue* to initiate the presentation of a view controller. Segues are defined within Interface Builder and are essentially a relationship between two view controllers that involve either transitions from the first view control to the second, or a containment relationship. For instance, you can set UI elements that can generate events (e.g., `UIButton`) as the starting point or initiator of a segue, and then set a target view controller as the destination of the segue. When the user interacts with the UI element (e.g. taps the button), the transition to the destination view controller happens automatically. These are known as *action segues*. Alternately, you could add an *embed segue* that maps a child view controller into a portion of a scene being managed by another view controller, or similarly, view controllers are arranged in a parent—child relationships (e.g. a view controller manages a stack of child controllers).

You can also establish what is known as a *manual segue*, which is essentially a segue whose source is a view controller itself, instead of an actionable UI control. The idea is that we can setup such a segue and give it a unique identifier in Interface Builder, and then programmatically initiate the segue in code. A segue known as an *unwind segue* can also be introduced. An unwind segue is helpful when you need to "unwind" back to a previous presenting view controller after a chain of one or more additional view controllers have been presented.

You will utilize segues to implement the scenario described above. You will need to accomplish this by implementing the following tasked described at a high level.

1. Implement the sign up and main screens in Interface Builder.
2. Add two new view controller classes: `SignUpViewController` and `MainViewController`.
3. Associate the new classes with the screens designed in Interface Builder and add any necessary outlets.
4. Add segues in Interface Builder according to the specification above.
5. Implement Swift code to handle form validation and dismissal of presented views where needed.

4.3.1 Adding new view controllers to the storyboard

Your first task is to update the login screen with an additional label and register button, and to add a new view controller scene for the sign up and main screens, as shown in Figure 4.2. Update the login screen by opening up the `Main.storyboard` file in Interface Builder and dragging a label and button into the scene. Set the text attribute on both controls to match the figure.

Use the key sequence ⇧+⌘+L to display Interface Builder's library of user interface components

To add a new view controller scene to the storyboard, use the key sequence ⇧+⌘+L to display Interface Builder's library of user interface components. Scroll down or search for the

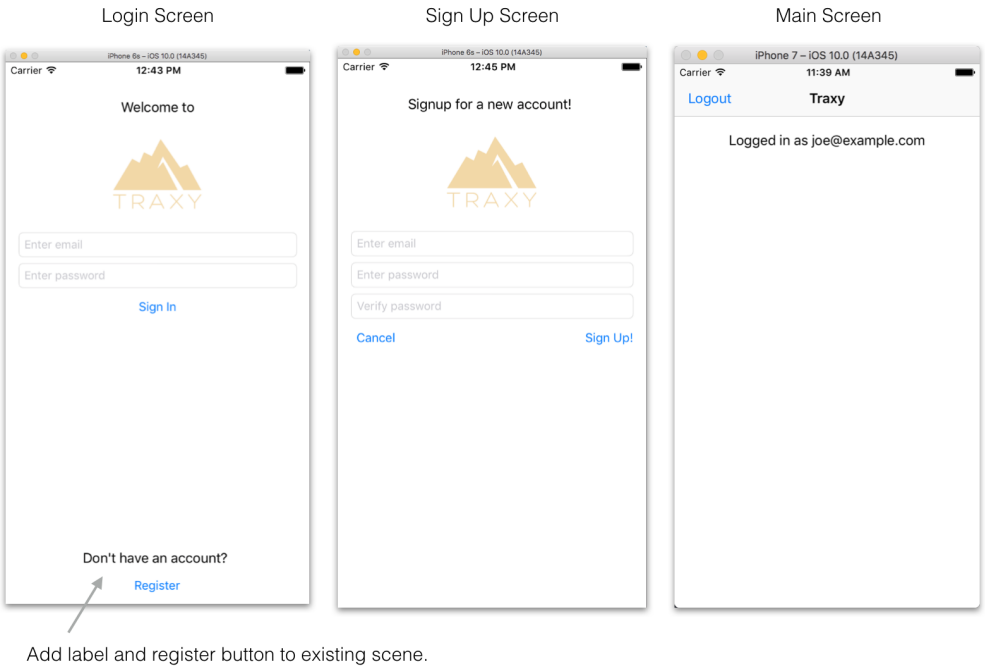


Figure 4.2: The three screens needed by our app: login, signup, and main screens.

object labeled **View Controller** (see Figure 4.3) and drag a couple of them out onto the Interface Builder canvas — one for the sign up screen and one for the main screen.

Drag out the following UI controls into the sign up scene:

- `UILabel` (labeled as **Label** in the Xcode object library) with its text set to "Sign up for a new account".
- `UIImageView` (labeled as **Image View** in the Xcode object library) with its image set to the Traxy logo used previously on the login screen.
- `UITextField` (labeled as **Text Field** in the Xcode object library) with its placeholder attribute set to "Enter email". Set its width to span horizontally from leading margin to trailing margin.
- `UITextField` with its placeholder attribute set to "Enter password". Set its width to span horizontally from leading margin to trailing margin.
- `UITextField` with its placeholder attribute set to "Verify password". Set its width to span horizontally from leading margin to trailing margin.
- `UIButton` with its text set to "Cancel". Set its width large enough to display the entire label and position its left edge on the leading margin.

- `UIButton` with its text set to "Sign Up!" Set its width large enough to display the entire label and position its right edge on the trailing margin.

Go ahead and set the attributes of the text fields on the sign up screen exactly as we did the login screen previously and described above. When you are done, the screen should look like the signup screen in Figure 4.2.

Before you drag any user interface controls out into the main scene's view controller, let's first embed it in a `UINavigationController`. We will be looking at this class in more detail in a future chapter, but it suffices to know at this point that the `UINavigationController` will manage a stack of child view controllers. You can embed the main view controller into a navigation controller by selecting it within Interface Builder, and then selecting Editor » Embed In » Navigation Controller from the Xcode application menu. Immediately, upon doing this you will see a navigation controller inserted into your storyboard and wired up to your main view controller. It also inserts a navigation bar on the top of our main controller scene. The "wiring" here is actually a special kind of segue known as a *relationship segue*. It establishes our main view controller as the navigation controller's initial or *root view controller*.

Next, drag out the following UI controls into the main scene:

- `UILabel` with its text set to "Logged in as . . . ".

- Place a `UIBarButtonItem` (labeled as **Bar Button Item** in the Xcode object library) on the left side of the main controller's navigation bar. After placing it set its text to "Logout" in the attribute inspector.

When complete the main screen should look like that shown in Figure 4.2. Once again, do not worry too much about getting the scenes to look perfect. Just drag out and size the controls approximately as shown. As long as you run the app on a simulator with the same size screen you did the layout on, the user interface should look reasonably good.

4.3.2 Adding new UIViewController classes

Now that we have our views defined in the storyboard, its time to introduce some new Swift controller classes. To introduce new classes, Make sure the `TraxyApp` folder is selected in your project explorer and then select File » New » File. . . from Xcode's application menu, and in the dialog that is displayed, make sure you have iOS selected in the upper left corner of the dialog, and then select the CocoaTouch class and press the Next button in the lower left as shown in 4.4. In the next dialog, set the class field to `SignUpViewController`, and the subclass to `UIViewController`. Make sure the "Also create XIB file" checkbox is unchecked, and then set the language to "Swift". Make sure the form is filled exactly as shown in 4.4 before pressing the Next button. The last dialog is basically asking where you want to the file to be placed. It will default to your project directory, placing the code in the same place the existing code is.

At this point in the Project Explorer, you will find a new file named `SignUpViewController.swift`. If you click on the file you will see the new view controller subclass with a method stub or two. Go ahead and repeat the above process for introducing a new subclass called `MainViewController`. We will revisit the implementation of these classes shortly.

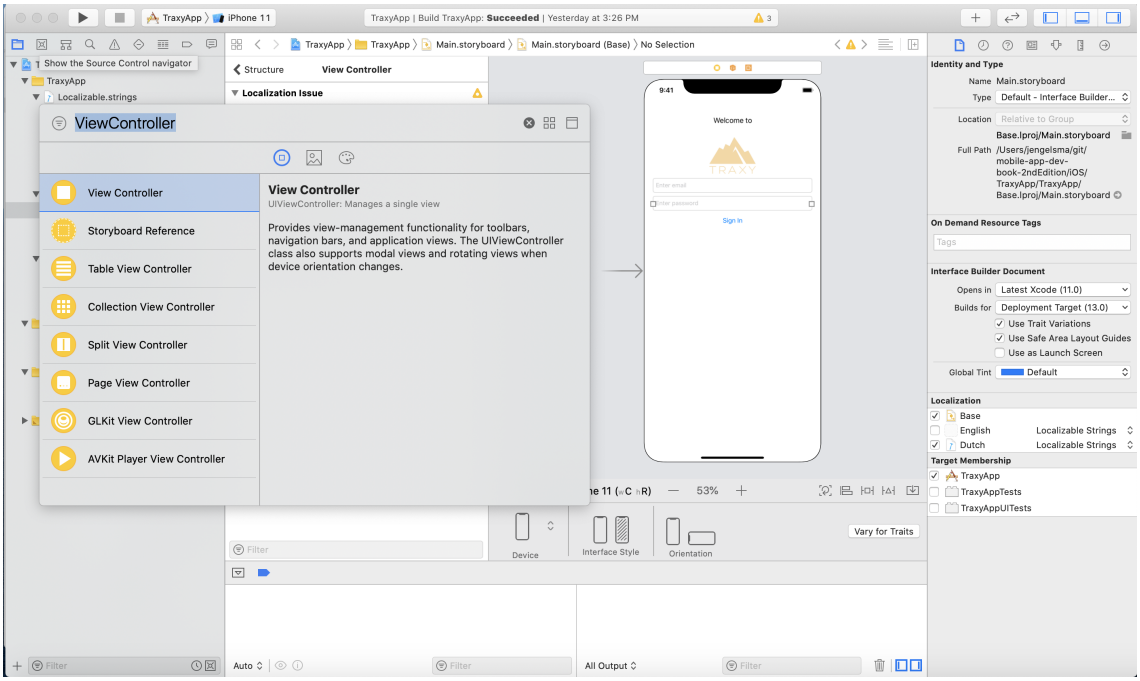


Figure 4.3: Add a new view controller to the storyboard by dragging a **View Controller** from the object library onto the Interface Builder canvas.

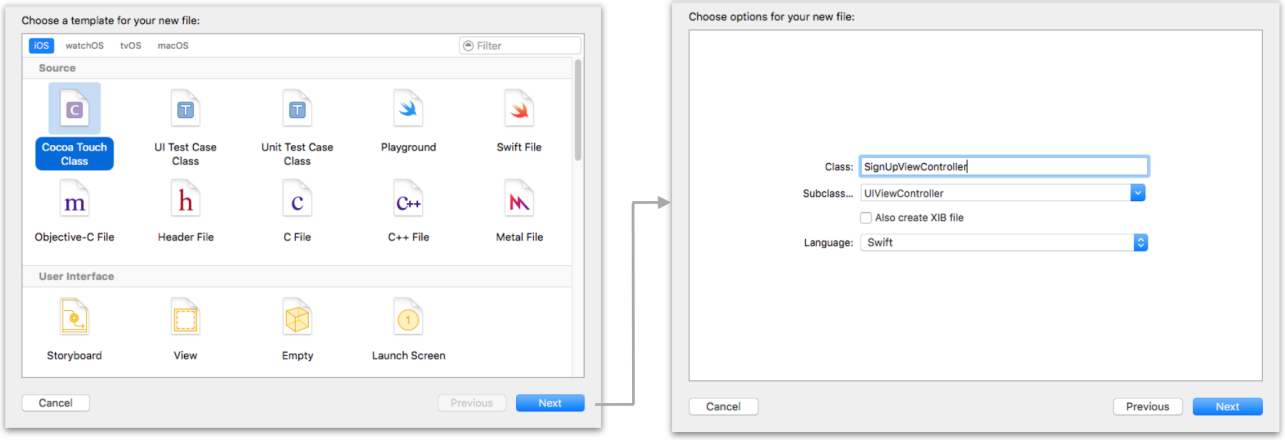


Figure 4.4: Adding new `UIViewController` subclasses for our sign up and main screens.

NOTE

As you begin to add more classes to your project, you can group related classes together in the project explorer in a logical folder (actually called a group). To do this, select 2 or more files in the project explorer pane, and then hold ctrl and click with the mouse. On the popup context menu, select the New Group From Selection option. Instead of having a long linear list of files, you can add some structure to make things easier to locate. Keep in mind that the structure is purely logical within the Xcode project. It is not reflected in the underlying directory containing the project.

4.3.3 Establishing the identity of view controllers in Interface Builder

Now that you have the views implemented, and the custom view controller classes added, you need to associate each custom view controller to its respective view controller scene. This is accomplished via Interface Builder. Open up `Main.storyboard` and select the sign up view controller scene by clicking on its top bar. Next, in the utility area on the right, select the Identity Inspector pane, the third tab or keyboard shortcut ⌘ + ⌥ + 3. Enter the name of the new Swift class `SignUpViewcontroller` in the class field as shown in Figure 4.5. In a similar fashion, go ahead and select the scene for the main screen, and set its class identity to `MainViewController`.

NOTE

A common mistake of inexperienced iOS developers is to introduce new views and controller, and then forget to associate them together via the Identity Inspector. If you forget this step and try to run the app, any customizations you added in your custom class simply won't be available, and things won't work at all as you expect.

Before setting the identity for the original login scene we're going to revisit our login view controller class and do some refactoring. Recall that when we created the Traxy app in Xcode, it automatically introduced a view controller named `ViewController` and associated it with the single scene in the default storyboard file. You will recall that we added form validation to this code previously. You'll want to rename the class to a more descriptive name like `LoginViewController`. Unfortunately, while Xcode has refactor functionality for code written in Objective-C, this functionality is still not available for Swift code! Go ahead and make this change manually in three steps:

1. Open `ViewController.swift` and change the class name to `LoginViewController` in the class declaration at the top of the file. In addition, be sure to update the `UITextFieldDelegate` extension that you added at the bottom of the source file.

2. Rename the `ViewController.swift` file name to `LoginViewController.swift` by long clicking on the file name in the project explorer pain. While this is technically not

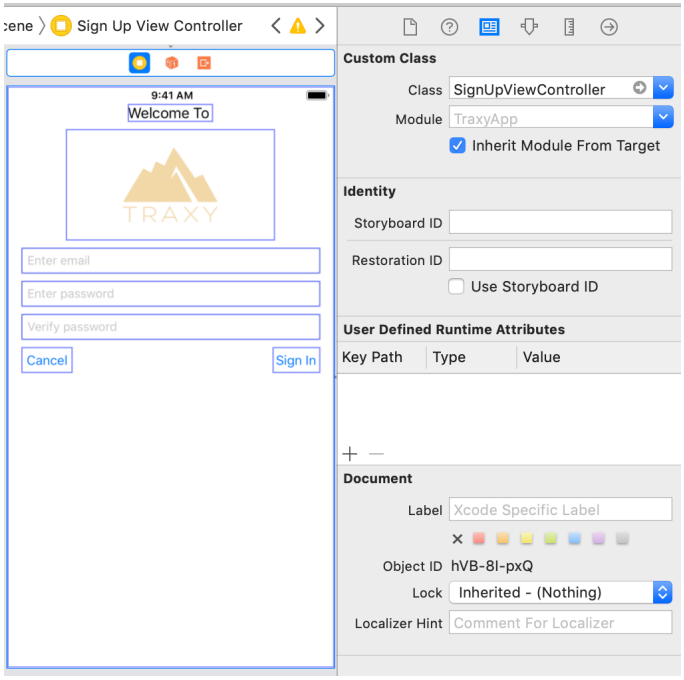


Figure 4.5: Associate the scene in the storyboard with the newly introduced view controller class by filling in the **Class** field via the identity editor in Interface Builder.

necessary, we recommend you adopt a Swift file naming convention where you define a single class per file and name the file `ClassName.swift` where `ClassName` is the name of the class.

3. Open `Main.storyboard`, select the login view controller, and then in the Identity Inspector change the custom class name from `ViewController` to `LoginViewController`.

To check if you did everything correct, go ahead and try to build the project. You can do this by selecting Product > Build from Xcode's application menu or use the keyboard shortcut ⌘+B. If it builds without errors you are in good shape. You can view any build errors by visiting the Issue Navigator in the Nav area on the left. The Issue Navigator is the fourth tab or use the keyboard shortcut ⌘+4. For code related issues, if you click on the error in the Issue Navigator, it will open the corresponding source file and position you at the exact location of the error.

Finally, you need to add outlets to the new view controllers. For the sign up screen we will need to wire up outlets for the email field and the two password fields via the Assistant Editor in Interface Builder as before. Upon completion, we should have the following outlet declarations in the `SignUpViewController` class:

Swift
```
@IBOutlet weak var emailField: UITextField!
@IBOutlet weak var passwordField: UITextField!
@IBOutlet weak var verifyPasswordField: UITextField!
```

Similarly, for the MainViewController class we need to be able to display the email address of the user who has logged in, so we wire up the logged in as label, which results in the following outlet being declared in MainViewController:

Swift
```
@IBOutlet weak var loginLabel: UILabel!
```

If you've forgotten how to wire up the outlets, go back to Chapter 2 and review how you did it previously. Assuming you have a clean build at this point, you are now ready to add segues to the app.

4.3.4 Adding segues in Interface Builder

Recall that segues can be used to model transitions from one scene to another. Examining the original flow between the screens shown in 4.1 you can see that if the user clicks the Sign In button on the login screen the main view controller should be presented, provided of course that the login form has been filled in correctly. Similarly, if the user taps the Register button on the bottom of the login screen, the sign up view controller should be presented. Similarly, if the user presses the Sign Up button on the sign up screen, the main view controller should be presented. It is important at this point to realize there are a variety of ways in which segues can be introduced into

an application. For example, in this app when the Register button on the Login scene is tapped, we want to *automatically* transition to the SignUp scene. However, when the Sign In or Sign Up buttons are tapped, before we transition to the Main scene we need to apply some validation logic to the values the user entered into the scene's text fields. In this case, we want to *programmatically trigger* a segue in our Swift code, only when valid data is entered. Finally, when the Logout bar button is tapped on the Main scene, we want to use an *unwind* segue to return to the original Login scene. Keeping this in mind, let's go ahead and add segues for our app.

Segues get added to the app via Interface Builder, so having opened `Main.storyboard`, while holding Ctrl click and hold on the Register button in the login scene, and while holding the mouse button, drag over to the SignUp view controller scene. When the SignUp view controller scene is highlighted in light blue, go ahead and release the mouse button. On the menu that pops up, select the "Present Modally" option as the kind of segue, and the type of presentation should be set to "Full Screen". This is an example of an *automatic* segue. That is, when the Register button is tapped in the Login scene, the app will segue automatically transition to the SignUp scene automatically. No coding necessary!

In order to trigger a segue programmatically in our Swift code we need to setup the segue a little differently in Interface Builder. Let's do this to establish the segue from the Login scene to the Main scene when the Sign In button is tapped. As was already mentioned, this segue needs to be triggered programmatically, because we only want it to happen after our validation code indicates a valid email and password have been entered in the the text fields. In this case, we will eventually be setting up an IBAction on the Sign Up button that will cause the segue to be performed when appropriate. So at this point, all we need to do in Interface Builder is create the segue and give it an identifier that we can reference later in our Swift code. To do this, instead of using the Ctrl key and our mouse click on the Sign In button, we need to select the entire Login view controller in Interface Builder. There are multiple ways to accomplish this in Interface Builder, but the easiest is to use the left most button on the bar of three buttons that appears directly above the scene in Interface Builder. When you roll over that button in Interface Builder with your mouse pointer, you will see the name of the controller displayed. In this case, you should see "Login View Controller" displayed when you roll over the button. Go ahead and press the Ctrl key while you click and hold on the button and drag over to the Navigation Controller containing the Main scene as shown in Figure 4.6. Release the mouse button once the Navigation Controller is highlighted. On the menu that pops up, select the "Present Modally" option as the kind of segue, and the type of presentation should be set to "Full Screen". Finally, click on the segue once it appears in interface builder and on the attribute inspector pane on the right, give it the identifier "segueToMain" as shown in Figure 4.7.

Repeat this procedure to setup a programmatically triggered segue from the Sign Up scene to the Main scene, but this time assign the identifier "segueToMainFromSignUp" to the segue. These segue identifiers will be referenced in our code when we want to programmatically invoke the segue. At this point, our scene in Interface Builder should look exactly like that in Figure 4.8.

We still have to add the unwind segue from our Logout button on the Main scene back to the Login scene. However, before we do that we'll need to introduce some Swift code. So let's now turn to the code necessary to complete our screen transitions.

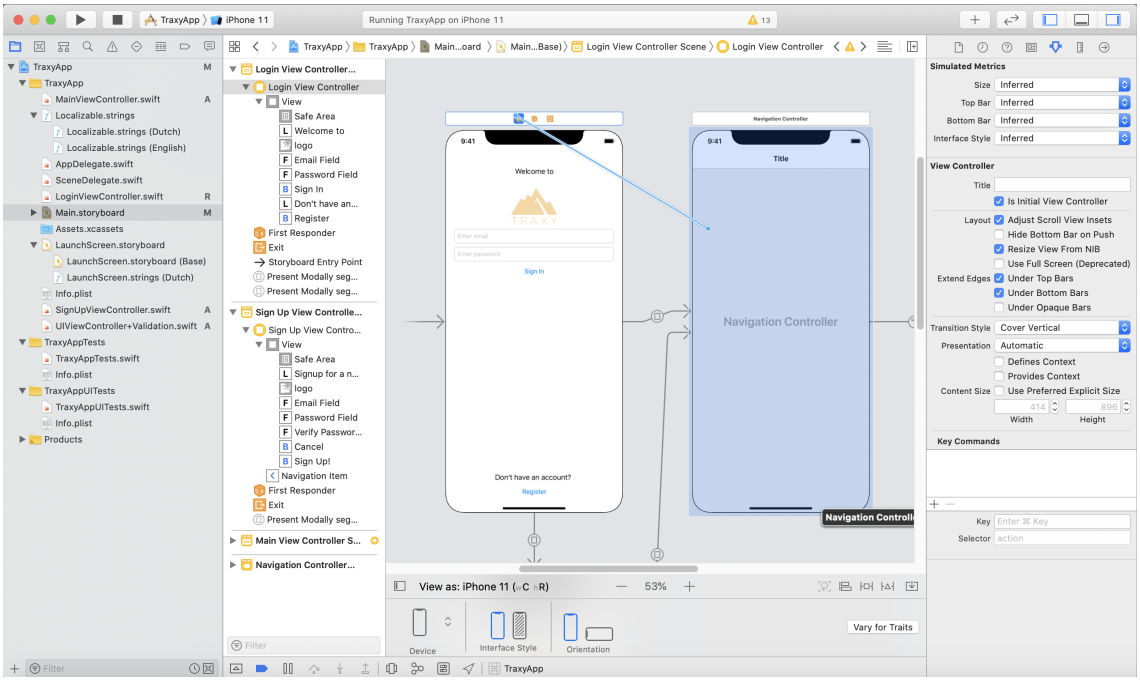


Figure 4.6: Using the button bar above the scene in Interface Builder to setup a segue that will be programmatically triggered in Swift code.

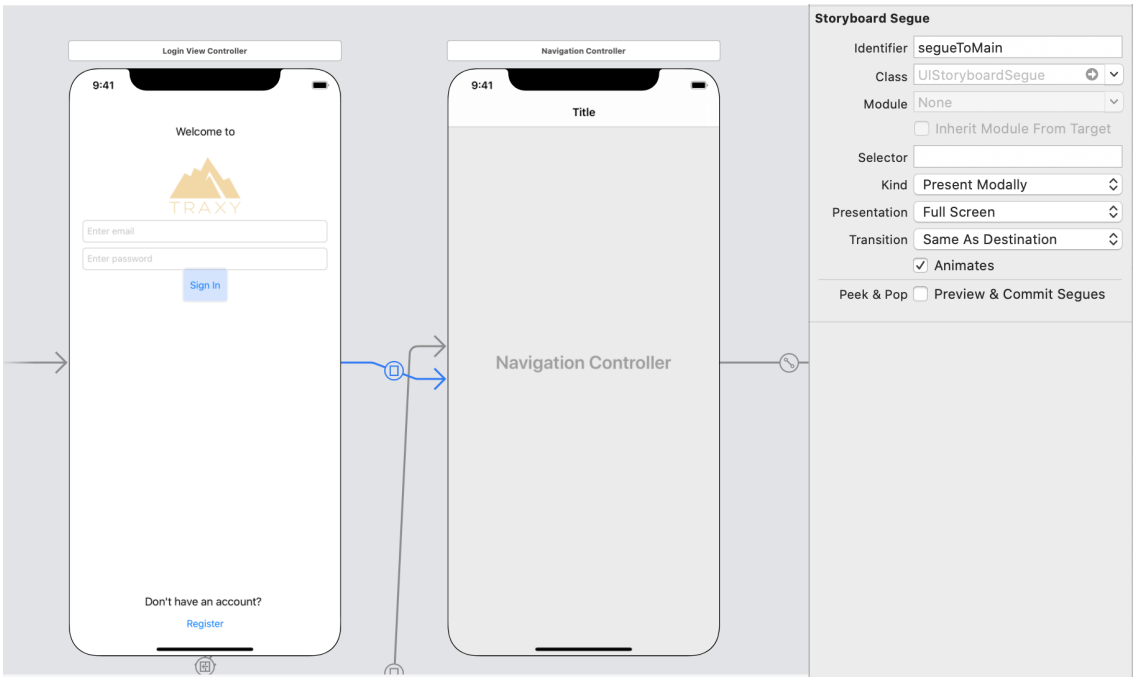


Figure 4.7: Assigning an identity to a segue by selecting the segue and setting its **Identifier** property via the Attribute Inspector.

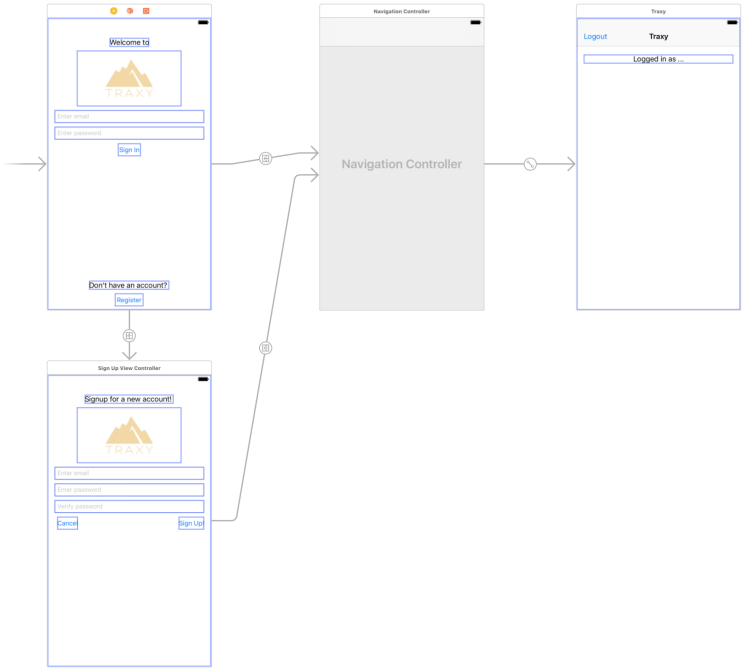


Figure 4.8: The storyboard after segues are added from the signup and register buttons in the login view controller to their respective destinations.

4.3.5 Completing the view controller implementations

Most of what remains is to be implemented in your Swift code. Observe that both the login view controller and the sign up view controller share some common functionality. For example, both need to be able to validate email addresses and verify the password fields are correct. You're going to factor that code out of your `LoginViewController` class and implement it in a way that any view controller in your app can call it. You can accomplish this by introducing a Swift extension. Previously, you saw how you could use an extension to implement the `UITextFieldDelegate` for your view controllers.

Introduce a new Swift file named `UIViewController+Validation.swift` by selecting File > New > File... from Xcode's application menu. Select "Swift File" from the dialog, and name it `UIViewController+Validation.swift`. Open the newly created Swift file and implement the extension as shown in Listing 17. Now, every `UIViewController` object can call these methods!

Swift

```swift
import UIKit

extension UIViewController {

  func isValidPassword(password: String?) -> Bool
  {
    guard let s = password, s.lowercased().range(of: "traxy") != nil else {
      return false
    }
    return true
  }

  func isEmptyOrNil(password: String?) -> Bool
  {
    guard let s = password, s != "" else {
      return false
    }
    return true
  }

  func isValidEmail(emailStr : String? ) -> Bool
  {
    var emailOk = false
    if let email = emailStr {
      let regex = "[A-Z0-9a-z._%+-]+@[A-Za-z0-9.-]+\\.[A-Za-z]{2,}"

      let emailPredicate = NSPredicate(format:"SELF MATCHES %@", regex)
      emailOk = emailPredicate.evaluate(with: email)
    }
    return emailOk
  }
}
```

Listing 17: Refactoring the form validation logic out into an extension of `UIViewController`.

Having factored out the common validation logic you can now finish your controller imple-

mentations. In `LoginViewController` you can rewrite your validation logic as show in Listing 18.

```swift
func validateFields() -> Bool {
  let pwOk = self.isValidPassword(password: self.passwordField.text)
  if !pwOk {
    print(NSLocalizedString("Invalid password",comment: ""))
  }

  let emailOk = self.isValidEmail(emailStr: self.emailField.text)
  if !emailOk {
    print(NSLocalizedString("Invalid email address", comment: ""))
  }

  return emailOk && pwOk
}

@IBAction func signupButtonPressed(_ sender: UIButton) {
  if self.validateFields() {
    print(NSLocalizedString("Congratulations!  You entered correct values.",
      comment: ""))
  }
}
```

Listing 18: Invoking the form validation logic in `LoginViewController`.

The validation logic in the `SignUpViewController` can be written in a similar fashion. After wiring up the sign up button to an `IBAction` called `signUpButtonPressed`, you can go ahead and implement your validation logic as shown in Listing 19.

You'll want to have your `SignUpViewController` implement the `UITextFieldDelegate` to manage the flow on the form, similar to what you did with `LoginViewController` as shown in Listing 20.

Don't forget to wire up the `UITextField` delegates in `viewDidLoad` as shown in Listing 21.

At this point, go ahead and build / run your app. You'll notice that if you click on the register button from the login view, you end up on the sign up view as expected. However, you'll soon discover there is no way to get back to the login view! Furthermore if you click on the sign in button, you'll see the message print on the console that the data was entered correctly (if you indeed did enter the data correctly!) but, no transition happens.

It turns out that since you've added an `IBAction` to the sign in/up buttons, it overrides the automatic segue behavior. You didn't add any `IBActions` to the register button, hence its segue was in fact initiated.

You can correct this latter problem by initiating the segues from your `IBAction` code. This is why previously we went through the pains of assigning identifiers to these two segues. For `LoginViewController` modify the `IBAction` as shown in Listing 22. You'll note the call to `performSegue` with the identifier set to the value you established in the storyboard when you introduced the segue.

After wiring up the sign up button to an `IBAction` called `signUpButtonPressed`, go ahead and implement your validation logic as shown in Listing 23.

Swift

```swift
func validateFields() -> Bool {

  let pwOk = self.isEmptyOrNil(password: self.passwordField.text)
  if !pwOk {
    print(NSLocalizedString("Invalid password", comment: ""))
  }

  let pwMatch = self.passwordField.text == self.verifyPasswordField.text
  if !pwMatch {
    print(NSLocalizedString("Passwords do not match.", comment: ""))
  }

  let emailOk = self.isValidEmail(emailStr: self.emailField.text)
  if !emailOk {
    print(NSLocalizedString("Invalid email address", comment: ""))
  }

  return emailOk && pwOk && pwMatch
}

@IBAction func signupButtonPressed(_ sender: UIButton) {
  if self.validateFields() {
    print(NSLocalizedString("Congratulations!  You entered correct values.",
      comment: ""))
  }
}
```

Listing 19: Invoking form validation logic in `SignUpViewController`.

Swift

```swift
extension SignUpViewController : UITextFieldDelegate {
  func textFieldShouldReturn(_ textField: UITextField) -> Bool {
    if textField == self.emailField {
      self.passwordField.becomeFirstResponder()
    } else if textField == self.passwordField {
      self.verifyPasswordField.becomeFirstResponder()
    } else {
      if self.validateFields() {
        print(NSLocalizedString("Congratulations!  You entered correct values.",
          comment: ""))
      }
    }
    return true
  }
}
```

Listing 20: Extending *SignUpViewController* to act as a *UITextFieldDelegate*.

```swift
override func viewDidLoad() {
  super.viewDidLoad()
  self.emailField.delegate = self
  self.passwordField.delegate = self
  self.verifyPasswordField.delegate = self
}
```

Listing 21: Wiring up SignUpViewController's text field delegates.

```swift
@IBAction func signupButtonPressed(_ sender: UIButton) {
  if self.validateFields() {
    print("Congratulations!  You entered correct values.")
    self.performSegue(withIdentifier: "segueToMain", sender: self)
  }
}
```

Listing 22: Performing the segue from LoginViewController to MainViewController in code.

```swift
@IBAction func signupButtonPressed(_ sender: UIButton) {
  if self.validateFields() {
    print("Congratulations!  You entered correct values.")
    self.performSegue(withIdentifier: "segueToMainFromSignUp", sender: self)
  }
}
```

Listing 23: Performing the segue from SignUpViewController to MainViewController in code.

Now if you build/run the app, you will notice that the validation logic gets ran and the segues happen as expected.

Notice that when you land on the main screen, you are still are not displaying the email address the user logged in with. Somehow you need to tell the main screen what to display. It turns out that the `UIViewController` class defines a hook method that we you implement to prepare for a segue. As you might expect this method gets invoked immediately before the segue is initiated. First you need to add an instance variable to your `MainViewController` class to hold the email address:

Swift

```
var userEmail : String?
```

Then in the `LoginViewController`, add the following method:

Swift

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
  if segue.identifier == "segueToMain" {
    if let destVC = segue.destination.children[0] as? MainViewController {
      destVC.userEmail = self.emailField.text
    }
  }
}
```

Note that the destination view controller of the segue is actually the navigation controller. The main view controller is the initial child view of the navigation controller, hence the reference in the code to the destinations child view controller. And similarly in `SignUpViewController` override the same method as follows:

Swift

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
  if segue.identifier == "segueToMainFromSignUp" {
    if let destVC = segue.destination.children[0] as? MainViewController {
      destVC.userEmail = self.emailField.text
    }
  }
}
```

Then in `MainViewController` you need to bind the value of the `userEmail` instance variable to the text attribute on the label that will be displayed on the screen. Do that in the `viewDidLoad` method:

```swift
override func viewDidLoad() {
  super.viewDidLoad()
  if let email = self.userEmail {
    self.loginLabel.text = email
  }
}
```

Go ahead and build/run at this point, and you'll see that our main view controller faithfully provides the email address the user logged in with.

Now let's go back and figure out how to transition from the sign up screen to the login screen when the cancel button is pressed. Recall that in this scenario the login view controller is the presenting view controller, and the sign up view controller the presented view controller. You can't simply have the sign up view controller present the login view controller when the cancel button is pressed, as the latter is already presented - its just covered up by the signup view controller. What needs to happen when the cancel button is pressed is that you need to dismiss the sign up view controller so that the login view controller can once again be exposed. Do this by adding an `IBAction` named `cancelButtonPressed` to `SignUpViewController`. Implement this method as follows:

```swift
@IBAction func cancelButtonPressed(_ sender: UIButton) {
  self.dismiss(animated: true, completion: nil)
}
```

Your final task is to implement the logout button on the main view controller. Note that if you attempt to implement this in the same way you implemented the cancel button on the sign up screen (e.g. call the `dismiss` method), you will have some strange anomalies. To see this, observe that we can arrive at the main screen directly from the login view controller, or via the sign up view controller. In the former case, merely dismissing the presented view controller will work fine, but in the latter case if we merely dismiss the view controller you will end up on the sign up view controller instead of the login view controller! At that point you'd have to press cancel to get back to the login view controller. It would be nice if you could go from the main view controller to the login view controller on a single logout button tap, no matter how you arrived at the main screen. You can get this sort of behavior this by introducing an *unwind segue*.

To implement an unwind segue, the first step is to implement a target unwind method on the class you want to end up in on an unwind. In this case, you want to end up in the login view controller, so implement the following method on `LoginViewController`:

Swift

```
@IBAction func logout(segue : UIStoryboardSegue) {
  print("Logged out")
  self.passwordField.text = ""
}
```

Note that the name of the method doesn't matter, it is only the signature of the method (e.g. takes a parameter of type `UIStoryboardSegue`) that makes it a valid target for an unwind segue. Having defined the target destination of the unwind, you can now create the segue in Interface Builder. Open `Main.storyboard` and click and drag from the main view controller's logout button on the navigation bar and drag to the top bar of the containing scene, holding it over the little orange box with a right arrow emanating form it, as shown in Figure 4.9. Select the `loginWithSegue` method (this is the one we just defined!). You can now build / run and verify that the unwind works correctly no matter how we arrive at the main screen.



Figure 4.9: To create an unwind segue, click on the logout button and drag to the little orange "exit" box on the views topbar, release and select the appropriate target method on the popup dialog..

.

4.3.6 View Controller Lifecycles

This would be a good time to pause a moment and reflect on the life cycles of the view controllers in your iOS app and how iOS interacts with them from a callback perspective. Figure 4.10 illustrates the calls made to the controllers when segueing directly from `LoginViewController` to `MainViewController`. In our code thus far, we have only overridden the `viewDidLoad()` method. Note that this particular method only gets called once over the lifetime of the view controller. Notice also that the MainViewController instance is destroyed after the unwind segue.

Figure 4.11 demonstrates the life cycle and framework interactions when we land on the main screen via the signup screen. In this case, by the time we arrive on the main screen, both the login and signup view controllers are still around. However, when we do the unwind segue on logout, both the `MainViewController` and the `SignUpViewController` are destroyed. The `LoginViewController` on the other hand will persist for the lifetime of the application. As we now turn to our Android implementation, you will see that the lifecycle of our Android analogues (e.g. activities) will vary a bit from the situation we have here in iOS, even though we have functional equivalence from the end user's perspective.

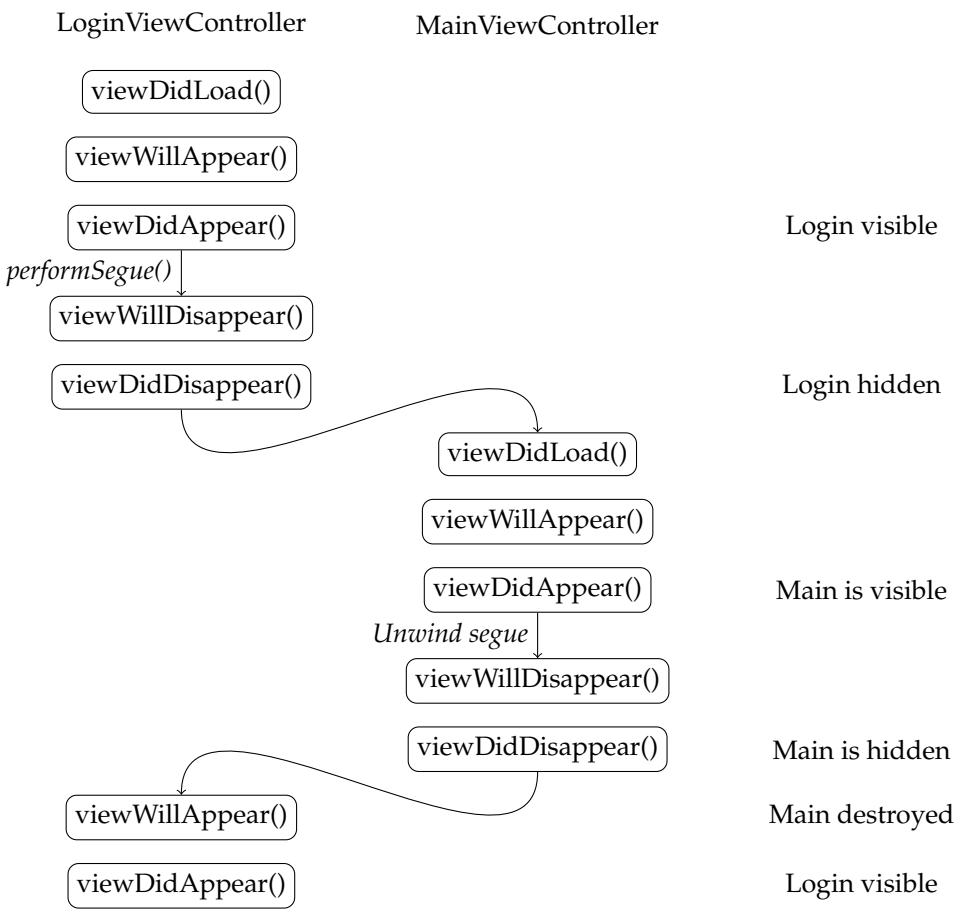


Figure 4.10: Lifecycle interplay between `LoginViewController` and `MainViewController`.

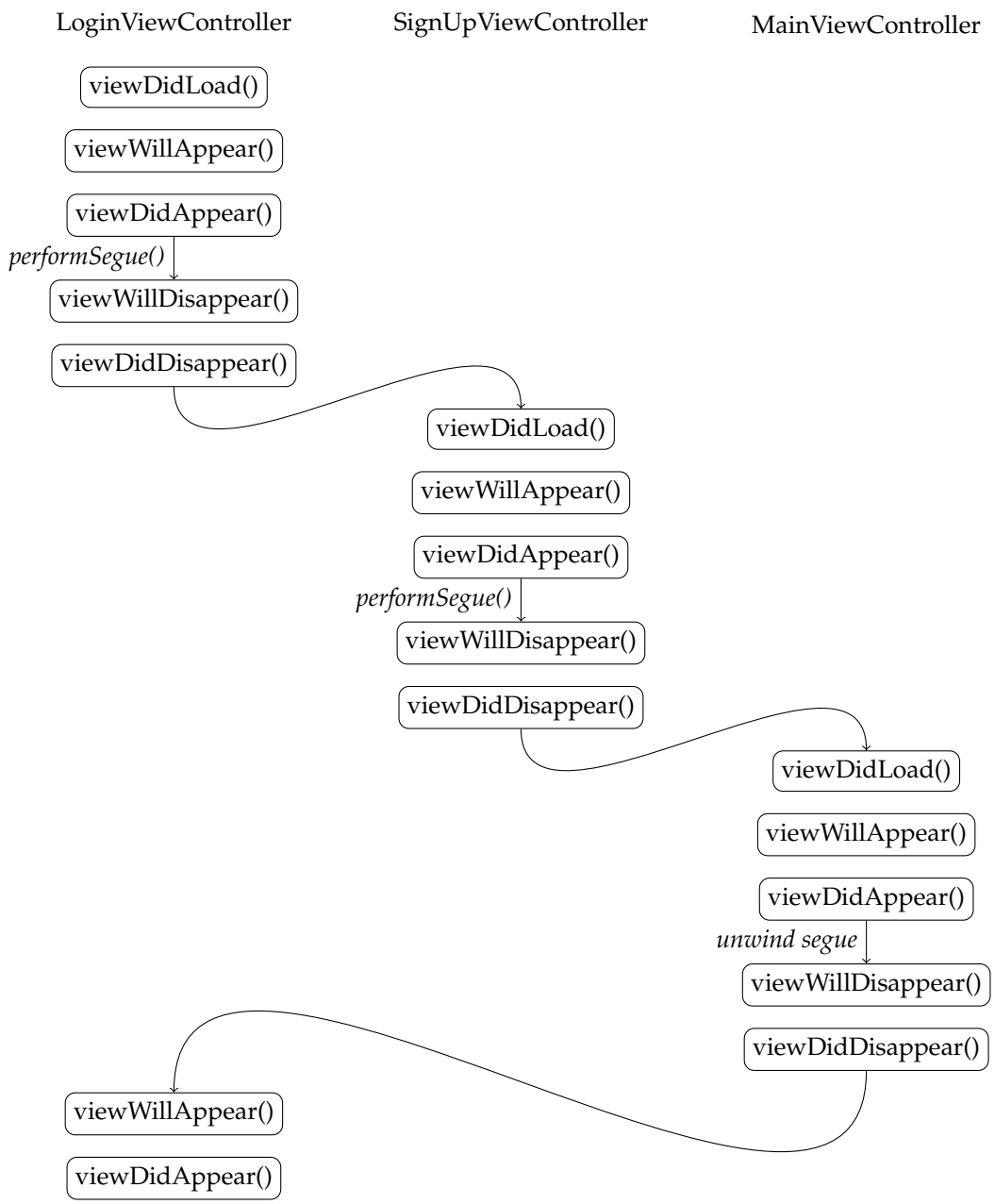


Figure 4.11: Lifecycle interplay between `LoginViewController`, `SignUpViewController`, and `MainViewController`.

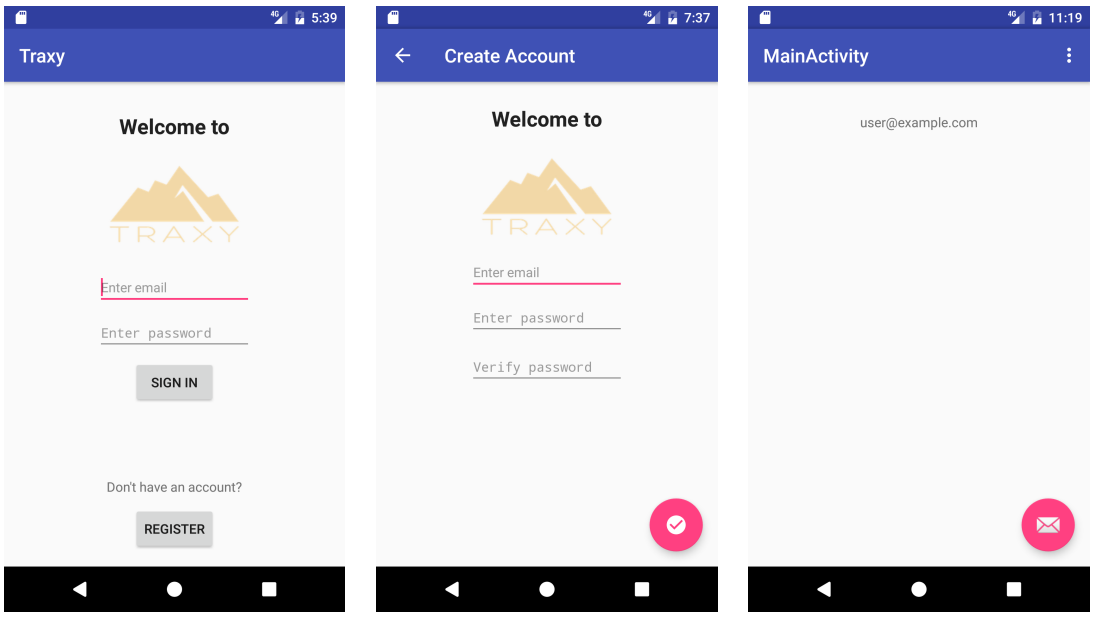
4.4 Scene Transitions in Android

Prior to Android Jetpack, implementing the screen transitions shown previously in Figure 4.1, required the introduction of two new Activity classes. With the Jetpack Navigation library, the recommended practice is to create one activity together with navigation graph and navigation controller. In this chapter we will show both approaches. The completed design of all the three screens (including the login screen completed in Chapter **??**) is shown in Figure 4.12.

First let's begin by using multiple `Activity` classes! The newer approach using navigation graphs will be explained later in this chapter.

Note that the design of the sign up and main screens are slightly different from their iOS counterparts. Minor design adjustments are applied to make the screens in compliance with Android best practices. For instance, in last two screens you will find a Floating Action Button (FAB), in place of a flat button, at the bottom right corner. The Material Design guidelines recommend use of FABs for primary actions that are performed frequently by the user[11].

Take a closer look at the the main activity screen. The iOS design calls for a "Logout" button, which later will later be turned into a tool bar button up on the navigation bar. However, the Android design of this screen (to be completed in a later chapter) will also include a FAB. Placing both the "Logout" button and a FAB violates the Material design recommendations. Therefore, you will move the Logout button somewhere else like a menu item in the action bar[1]. There are other cosmetics variations, but we will explain them later as you design each screen in more details.



(a) Login (b) Sign Up (c) Main (tentative design)

Figure 4.12: Traxy's first three screens

[1]The Google Android team recently introduced a new `Toolbar` widget in API Level 21 that supersedes `ActionBar`. Our manuscript will be using "app bar" when referring to a generic navigation bar at the top of the screen.

Two new Activity classes have to be added to the project. The steps for adding them are very similar to what you have learned in Section 2.4. That is, each activity class will be associated with a user interface layout file in XML. It is important to understand that any new Activity class added in your app must be *registered in the manifest file*. Fortunately, Android Studio automates this step for you when if the proper steps for creating a new Activity class are followed.

Based on what you've learned about screen transitions in iOS, there are a couple of things that are worth noting at this point.

- When dealing with multiple view controllers, iOS distinguishes **presenting** from **presented** view controllers. You will not find such standard naming convention anywhere in the Android technical documentation. However, for ease of reference throughout our manuscript, we will be using **source** and **target**[2] to distinguish the two in Android code.
- Prior to Android Studio 3.4, there was no iOS segue equivalent in Android's layout editor. Transitions between screens cannot be specified declaratively in an XML layout file, but had to be implemented in code. In Android Studio 3.4 and later, you will also find navigation editor to design the transition among multiple screens in your app.

The rest of this chapter focuses on learning various Android techniques for managing Activity interactions. In particular, the following topics will be explored in more detail:

- Adding new Activity classes
- Creating vector drawables
- Adding action buttons
- Using the `Intent` class to transition from a source to a target Activity
- Passing data from a source to a target Activity
- The Android Activity stack
- Manipulating the Activity stack by setting flags on the Intent object.
- Passing result from a target to a source Activity
- Creating and editing navigation graphs

4.4.1 Adding New `Activity` Classes

To implement the required updates, you will need to introduce two new Activity classes:

- `SignUpActivity`: This activity will ultimately coordinate the creation of a new user account when the user clicks the "Sign Up" at the login screen. It is equivalent to the `SignUpViewController` in the iOS implementation.

[2]We are aware of the common practice of using the "source/destination" word pair, but we have opted to use "source/target".

- `MainActivity`: This activity is a placeholder screen for the main screen of the Traxy app, which will be fleshed out further in Chapter 6. It is equivalent to the `MainViewController` in the iOS implementation.

You will create the two classes in the above order, but first update your login screen by adding the "Register" button at the bottom of the screen as shown in Figure 4.12a along with the accompanying label. Use the layout editor to drag to the bottom center a text view with text "Don't have an account?", and a button with text "Register". At this point, these widgets don't need to be perfectly centered, just make it look as much like Figure 4.12a as you can. Be sure to extract any new string literals to Android string resources using a similar technique you learned in Section 2.4.5.

> NOTE
>
> Remember to use Android Studio's ubiquitous incremental search feature so your development time can be more productive. This feature can also be used inside the Widget "Properties" panel. First, highlight any item inside the panel and then start typing part of the phrase you are looking for. For instance, to locate "centerHorizontal" in the list, just type a few letters of the term, e.g. C-E-N or H-O-R

Sign Up Activity

Before you add the second activity as a new class `SignupActivity`, let's revisit the rationale behind the decision to use a slightly different screen design than the iOS counterpart.

- In Android, one of the key functions of the action bar (also called the app bar) is to provide navigational context. For our particular use case, the current context is creating a new account. Therefore, the topmost text "Sign up for a new account" can be subsumed by a short title in the app bar.
- The Material Design guidelines include the following two recommendations: (a) Decide for me but let me have the final say[3] and (b) A floating action button represents the *primary action* in an application.

In the spirit of these two guidelines, the main reason the user arrives at this activity is to *sign up* for a new account. Hence, this action is the *primary* action of the activity and should be implemented using a FAB. The "Cancel" button, is not an action that requires specific application logic. It is just a navigational button that can be subsumed by the "Back" or "Up" navigation.

Once the above design principles are understood, you can now begin adding more code into the project. In Chapter 2.4 you learned how to add a new activity class using the Empty Activity template. However, this template does not provide a FAB. For the Sign-Up screen, you will want to use the *Basic Activity* template which does provide a FAB.

Locate the java folder under Android Studio project panel, right-click and then select New > Activity > Basic Activity. Enter the following information in the next dialog:

[3] https://developer.android.com/design/get-started/principles.html

- Activity Name: `SignUpActivity`
- Layout Name: *accept the default name provided by Android Studio*
- Title: Create Account
- Leave the checkbox Launcher Activity unchecked
- Package name: *accept the default name provided by Android Studio.*
- Source Language: Kotlin

After the entries are entered correctly and the dialog is confirmed, you should see several new files added to your project: one Kotlin file (`SignUpActivity`) and two XML files (`activity_sign_up` and `content_sign_up`), and several other files that we will discuss later. Let's first look at the two layout files in more details.

When you open the layout file `activity_sign_up` in **Text** mode (by selecting one of the tabs at the upper right of the layout editor: "Code", "Split", or "Design"), you may be surprised to see many unfamiliar layout/widget types such as: `CoordinatorLayout`, `AppBarLayout`, etc. We will ignore these names for now, and revisit them in Chapter 5.3.1. Aside from these unfamiliar widgets, you should focus your attention to the `<include>` and `<FloatingActionButton>` tags towards the last third of the file.

XML

```
<!-- other hidden contents here -->
<android.support.design.widget.FloatingActionButton
    android:id="@+id/fab"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="bottom|end"
    android:layout_margin="@dimen/fab_margin"
    app:srcCompat="@android:drawable/ic_dialog_email" />

  <include layout="@layout/content_sign_up" />
<!-- other hidden contents here -->
```

- The `<include>` tag specifies another XML layout file (`content_sign_up.xml`) to be included into this layout
- The FAB is declared with an attribute `layout_gravity="bottom|end"` in order to anchor it to the bottom right corner of our screen. In Chapter 5.3.1 we will uncover the difference between "`bottom|end`" and "`bottom|right`".

Now open `activity_sign_up.xml` in *Design* mode, you will notice that the FAB shows an image of an envelope, which is irrelevant to the current task of creating a new account. We want to replace it with a "done" icon using an Android vector asset.

- Right click on the res directory, then select New > Vector Asset.
- In the next dialog (Figure 4.13 page 120) be sure "Clip Art" radio button is selected and then click the Android icon to create a new icon

- In the Material Icons selection dialog, type "done" in the search box and select the "check mark" icon as shown in Figure 4.14. It is evident that your Android Studio is pre-installed with hundreds of icons from the Material Design icon sets.
- After the OK button is clicked, Android Studio will create a new icon `ic_baseline_done_24.xml`
- In the next dialog, save the icon to its default path

Vector assets/drawables have several advantages over bitmap images. First, they use less storage because each icon is rendered using vector graphics. Second, to avoid pixelation effect when using bitmap images, we have to supply several copies of each bitmap image at various screen densities. Vector drawables do not suffer from this effect; one vector definition can be rendered at different screen densities while maintaining the same image sharpness.

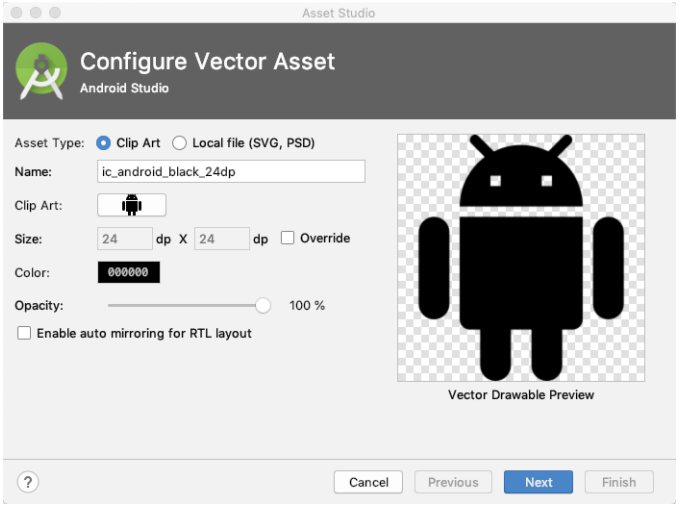


Figure 4.13: Adding Vector Asset

Now you can use the vector asset just created for the FAB icon:

- Open the layout file `activity_sign_up.xml` and highlight the FAB widget
- In the Properties panel click the three dots to the right of **srcCompat**. Use the next dialog to select the vector asset just created above. After completing this step, the FAB should now display a new icon (in black).
- Change its `app:tint` property to `@android:color/white` so it has a better contrast with the current FAB accent color

> **NOTE**
>
> If you have the Material Design Icon Generator plugin installed, another alternative to perform the above step is to select New 〉 Material Design Icon or its keyboard shortcut ⌥+⌘+M.

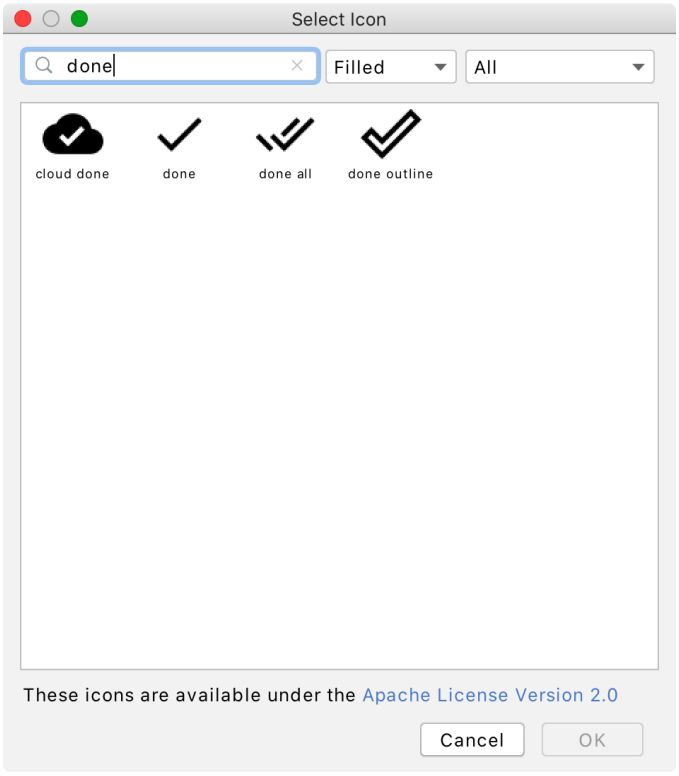


Figure 4.14: Material Icons

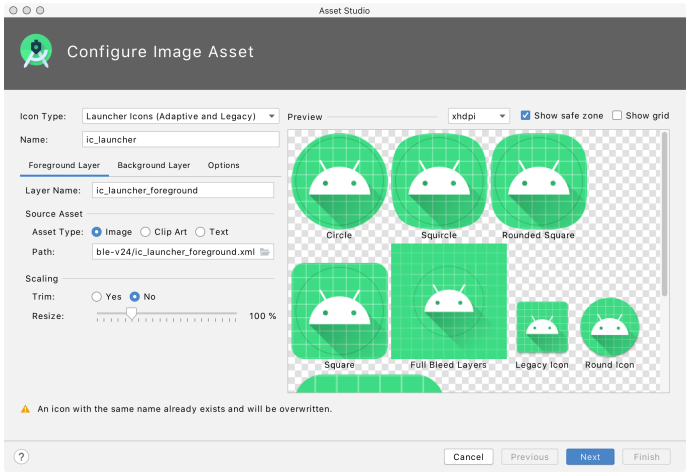


Figure 4.15: Creating a New Image Asset

Vector assets were introduced more recently and not all widgets are able to use them yet. For those widgets, you are required to use *image assets*. Creating an image asset is very similar to creating a vector asset. Invoking New > Image Asset from **res/drawable** will prompt a dialog similar to Figure 4.15. Be sure to set

- "Icon Type" to "Action Bar and Tab Icons"
- "Asset Type" to "Clip Art"

You can select a clip art from the Material Design icon sets by clicking on the "Clip Art" icon (highlighted in Figure 4.15). Recall how you create a vector asset in Section 4.4.1. One copy of *vector asset* is sufficient to render an icon at different screen densities. Unfortunately, an attempt to render a low resolution *image asset* at a high screen density will create pixelation defect. To avoid this visual defect, you must provide one copy of the image per screen density. The most common unit for measuring screen density is "dots per inch" (DPI). The preview panel on the right side of the dialog in Figure 4.15 shows several previews of an image asset with different masked shapes. To preview the icons at different densities select the drop down menu: ultra high (xxhdpi), extra high (xhdpi), high (hdpi), and medium (mdpi).

Now open `content_sign_up.xml` and complete the design of the Sign Up screen based on Figure 4.12b. When completing the following steps, you should reuse the resources (drawables/strings) created earlier for the login screen. Add the following widgets horizontally centered on the screen

- An ImageView using the same logo as the login screen. Reuse the same image created earlier, do not create a new one.
- Three text fields with input hint: "Enter Email", "Enter password", and "Verify password". Reuse the string resources that were created for the LoginActivity screen as much as possible. You probably have to create a new resource for "Verify password" because this string has not been declared yet.

Main Activity

You are now ready to move on to the third activity, `MainActivity`. As explained earlier, in the spirit of Material Design, you should avoid placing two buttons ("Logout" and FAB) on the main activity screen. The solution to this problem is to use the app bar to place "Logout" as a menu item. Therefore, the main activity screen, for now, will hold only the following widgets:

- A `TextView` centered horizontally and vertically on screen to show the user email address entered at the login screen.
- A FAB for performing the primary action. Ignore the details of this action for now, and leave this FAB as a placeholder to be dealt with later
- An action menu item for logging out

Create the `MainActivity` class using the Basic Activity template. In addition to creating a new Kotlin file and the associated layout file(s), Android Studio also adds a new resource string `title_activity_main` which is set to "MainActivity". If you prefer a different title, edit **res/values/strings.xml** and change the title text.

Don't worry about the "Logout" menu for now. The next section will guide you through creating such a menu

4.4.2 App Bar Menu Items

In Android, action buttons are treated like a menu, and as previously explained in Section 1.5, menus are managed as a resource. To place an action button (in the app bar) you must first define a menu structure in XML. Create a new menu resource file (`menu_main.xml`). Right-click on *res/menu*, then select New > Android Resource File then enter the name "`menu_main.xml` as the file name to store the menu structure, and the Resource type as "Menu".

The file will be created in the `res/menu` sub-directory. Open `menu_main.xml` in the menu editor. If the menu was created automatically at the project initialization, you should find one menu item: "Settings". The menu editor has similar features as the layout editor, so you should feel comfortable navigating through different parts of the editor. Now, let's add a new menu item for the Logout action button.

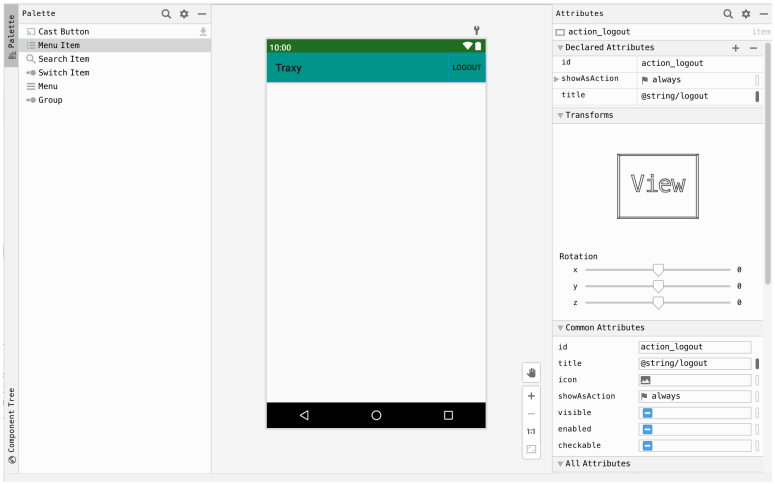


Figure 4.16: Menu Editor

- Drag a Menu Item from the Palette to the Component Tree
- In the "Properties" panel, set its id to `action_logout`, title to "Logout", and *showAsAction* to "always".

Compared to its layout editor counterpart, the "Properties" panel of the menu editor is very desolate. The only interesting attribute here is `showAsAction`, which can take one of the following possible values:

- never: the menu item never shows up on the app bar. It will show up in the drop-down list when the menu icon (vertical dots) is tapped
- ifRoom: the menu item will show up only if there is room in the app bar

- always: opposite of "never"
- withText: this flag is meaningful for a menu item that defines an icon: an accompanying text is also rendered besides the icon
- collapseActionView: the menu item is associated with a collapsible action view.

At this point we have the menu structure defined, but will postpone the detailed steps for including the menu into our Activity class in Section 4.4.6 when we finalize `MainActivity`.

With the three Activity classes in place, you are now ready to assemble them together and build the desired screen flow. The first step is wiring up the "Sign Up" button in the Login Activity.

4.4.3 Using Android `Intents`

In iOS, you learned that screen transitions could be defined visually by associating two view controllers in a storyboard with a segue. Prior to Android API 29, screen transitions can only be done programmatically with the following two lines of code which will transition from `SourceActivity` to `TargetActivity`.

Kotlin

```
val nextPlz = Intent (this@SourceActivity, TargetActivity::class.java);
startActivity (nextPlz);
```

Among the new features in Android 10 (API Level 29) is navigation graph and its accompanying tools and libraries. We will explore this technique later in Section 4.4.7. To some extent, Android navigation graphs are like iOS storyboards. The class name `Intent` chosen by Android is relevant here; it (the Intent object) captures an intention to launch an activity.

Let's modify the event handling logic in `LoginActivity`. You now have two buttons to handle "Sign in" and "Register". The existing code for the "Sign In" button only validates the email and password, without making a new transition to `MainActivity`. We need to add code similar to the sample given above, so when the email and password are verified, your app will transition from `LoginActivity` to `MainActivity`.

Kotlin

```
val mainIntent = Intent(this@LoginActivity, MainActivity::class.java)
startActivity(mainIntent)
```

Using a similar technique, you can implement a click listener for the "Register" button to create a transition into `SignUpActivity`.

Kotlin

```
val register = findViewById<Button>(R.id.signup_button)
register.setOnClickListener{
  val signUpIntent = Intent(this@LoginActivity, SignUpActivity::class.java)
  startActivity(signUpIntent);
}
```

Run the app and confirm that you are able to transition into the two new screens when either the "Sign in" or "Register" button is pressed. It is also equally important to confirm that you are able to go back to `LoginActivity` when you press either the "Back" key.

You are very close to accomplishing the goal of wiring up the two buttons in the login screen, but not quite done yet. Recall that you put the placeholder text view widget in `MainActivity` for

displaying the email address entered at the Login screen (and also the email address at the Sign Up screen, but you can deal with that later!). How do you pass the email address from one Activity class to another? The target Activity was initiated programmatically using an Intent when the source Activity invokes the `startActivity()` method. Your intuition might tell you that some extra data can be transferred either *after* or *at the time of* the `startActivity()` call. Indeed, the `Intent` class does in fact allow use to transfer data to the target activity.

Passing Extra Data Via `Intents`

The `Intent` class provides an overloaded `putExtra()` method which can be used to add extra payload of various types (String, int, double, arrays, ...) to an `Intent`. Java developers who have prior experience in using `java.util.Map` will find commonalities between `Map`'s `put()`/`get()` and `Intent`'s `putExtra()`/`get`*Type*`Extra()` methods.

For instance, if a source Activity of a hypothetical atomic element app needs to pass atomic name (String) and its mass (double) to a target Activity, the associated Intent can be initialized using the code snippet in Listing 24. The Kotlin code equivalent of lines 4–5 in the listing for invoking `putExtra` will look exactly the same.

Kotlin

```
1 val nextPlz = new Intent (sourceActivity.this, targetActivity.class);
2
3 // Titanium atomic mass is 47.867
4 nextPlz.putExtra ("ATOM_NAME", "Titanium")
5 nextPlz.putExtra ("ATOM_MASS", 47.867)
6 startActivity (nextPlz)
```

Listing 24: Using `putExtra()` to add data payload

To retrieve the atomic name-mass pair, the target Activity would use the code snippet in Listing 25, which is typically written inside its `onCreate()` method. The second argument supplied to `getDoubleExtra()` is the default value in case the payload `ATOM_MASS` is undefined. Be aware that the method `putExtra()` is overloaded, but `get__Extra()` is not; you have to invoke different methods depending on the data type you retrieve.

Notice that in Kotlin, the getter method `getIntent()` simply becomes a direct access to the `intent` property.

Kotlin

```
1 val payload = intent
2 if (payload.hasExtra("ATOM_NAME") && payload.hasExtra("ATOM_MASS")) {
3     val element = payload.getStringExtra("ATOM_NAME")
4     val mass = payload.getDoubleExtra ("ATOM_MASS", 0.0)
5     //
6     // code that uses element & mass goes here
7     //
8   }
```

Listing 25: Using `get__Extra()` to retrieve data payload

Let's use these examples to add the necessary code in `LoginActivity` and `MainActivity`

to display the user email on the text view of the main screen. In `LoginActivity` you need to add a single line of code to add the email string as payload to the intent that will bring you to the `MainActivity`.

Kotlin
```
val mainIntent = Intent(this@LoginActivity, MainActivity::class.java)
mainIntent.putExtra("EMAIL", emailStr)
startActivity(mainIntent)
```

The code snippet in Listing 14 shows an example of using the `.text` property to obtain the string content of a `EditText` widget. In Kotlin, using a property in a read context is equivalent to calling a Java "getter" method, and using the property in a write context is equivalent to calling its "setter" method. You can now add the necessary code in `MainActivity` that invokes the `.text` property to update the `TextView` at the center of the screen. In the activity's `onCreate` method, add code that will get a reference to the `TextView` that will display the email address, and then extract the email string from the intent object and bind it to the display.

Kotlin
```
val payload = intent
val userId = findViewById<TextView>(R.id.user_id)
if (payload.hasExtra("EMAIL")) {
  userId.text = payload.getStringExtra("EMAIL")
}
```

Rerun the app ctrl+R, and enter your email and password at the Login screen. After tapping "Sign In" your email should now appear on the main screen. Congratulation! So far, things seem to be working.

At this point, you should have a solid understanding how screen transitions work in Android and be able to write Android apps that call for multiple screens. In order to complete your understanding of the subject, let's relate what you just learned here with lifecycle interplay between the two activities (Login and Main). Refer to Figure 4.17. When `LoginActivity` is created, it begins its lifecycle and the first three callback methods (onCreate, onStart, and onResume) run in that order. When the user press the "Sign In" button, the `startActivity()` call causes the activity to run its `onPause()` and `onStop()` before it becomes hidden. At that point, `MainActivity` begins its lifecycle. Likewise, when the "Back" button is pressed, the `MainActivity` runs its `onPause()`, `onStop()`, and `onDestroy()` callback methods before it disappeared and destroyed. Finally, `LoginActivity()`'s onResume() runs again, it becomes visible (again), and regains control of the screen.

If you've been paying attention at this point you might be wondering about the following legitimate question:

"How is it possible for `LoginActivity` to reappear without being recreated?"

Well, if you have used stacks in your programming endeavors before, you are aware that a push operation on a stack makes the (current) topmost item inaccessible and the new item just pushed at the top is now the only accessible item on the stack. Likewise, a pop operation removes the (current) topmost item, making the item immediately below it accessible (again). The "inaccessible-accessible" effect we observe as a result of push/pop operations fits exactly the "hidden-visible" sequence on our Activity objects.

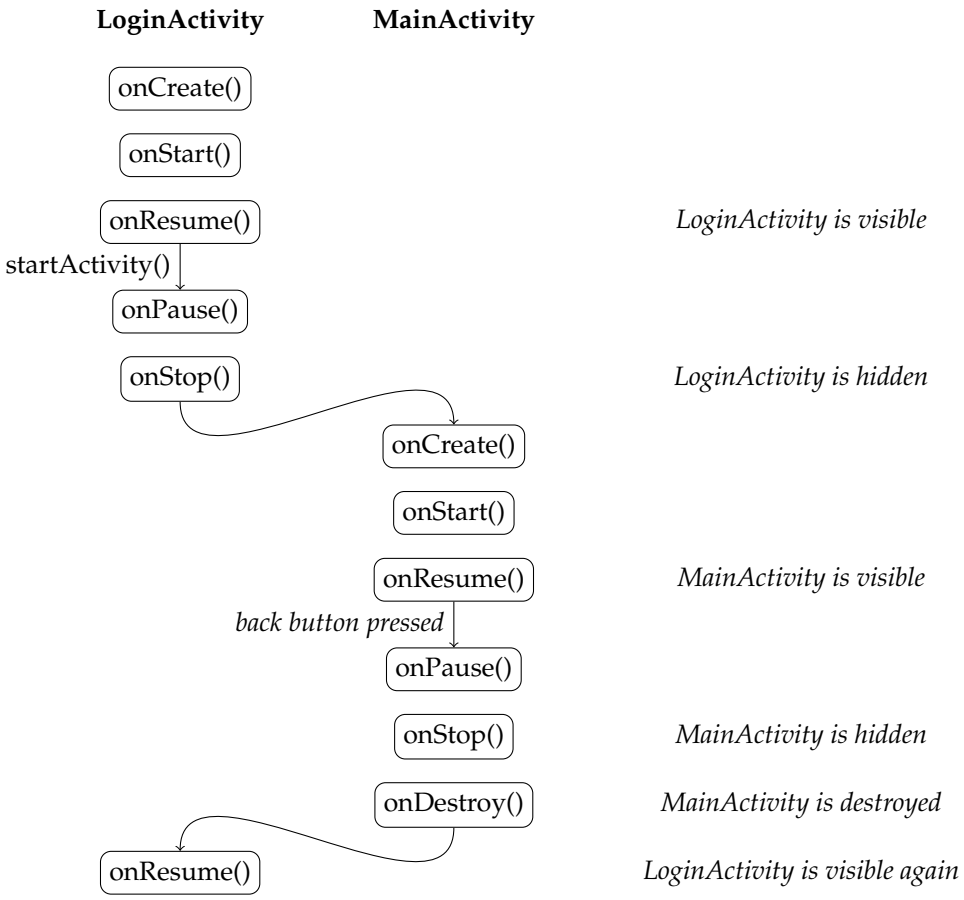


Figure 4.17: Lifecycle interplay between LoginActivity and MainActivity

4.4.4 Android Activity Stack

The Android framework uses a stack data structure for managing activity transitions. The use of a stack for this purpose is actually borrowed from a similar technique for implementing function calls in most programming languages. When a function (`thisFunc()`) calls another function (`nextFunc()`), the compiler uses a stack to store the return address within `thisFunc()` and any parameters required by `nextFunc()`. When `nextFunc()` terminates, the return address into `thisFunc()` is reloaded from the stack allowing `thisFunc()` to regain control.

The runtime stack used by a compiler for function calls and parameters is usually considered sacrosanct; no programmer should alter its contents, unless you are willing to take the risk of crashing your program. On the contrary, Android Activity stack is not untouchable. There are methods in the `Activity` class that let you *indirectly* alter the Activity stack. One such method is `finish()`, which is called by an Activity to terminate itself. More importantly, a side effect of calling `finish()` is, the associated entry in the Activity stack will also be purged. Let's see how we can use this feature to manage the screen transition among `LoginActivity`, `SignUpActivity`, and `MainActivity`.

The desired screen flow is as follows:

1. When the user taps "Sign Up" from `LoginActivity` the app transitions to `SignUpActivity`
2. After the registration process completes, `SignUpActivity` transitions to `MainActivity`
3. But, when the "Back" button is pressed from `MainActivity`, it should go back to `LoginActivity`, and *not* to `SignUpActivity`[4]

Essentially, you have to *skip* over `SignUpActivity` when `MainActivity` terminates. Combining your understanding of Android Activity stack and the (side) effect of `finish()`, you should be able to craft the code for achieving this behavior. *Hint: `SignUpActivity` should purge itself from the Activity stack when it transitions to `MainActivity`.*

As you reflect on how you handled this similar situation in the iOS implementation of these same screens, you may be amazed to see how adding one line of code is the extra effort developers take in implementing the *unwind segue* equivalent in Android! Nevertheless, don't get too excited yet, because you still have a few things to understand and clean up with regard to how these screen transitions really ought to behave.

4.4.5 Finalize `SignUpActivity` and `MainActivity`

After all these building blocks are well-understood, you should now continue adding more code and complete the implementation of both `SignUpActivity` and `MainActivity`.

The default code template generated for `SignUpActivity` includes a section for handling click events on the FAB. Your task is to modify the code to perform the following tasks:

- Verify that the email address is valid
- Verify that both password fields are not empty and match
- Use `SnackBar` to show all warning messages

[4] In the final implementation, the "Logout" action will actually take you back to the login screen

Kotlin

```kotlin
// in MainActivity
override fun onCreateOptionsMenu(menu: Menu?): Boolean {
    menuInflater.inflate(R.menu.menu_main, menu);
    return true; // "true" means the menu should be visible
  }
```

Listing 26: Enabling menu to appear on the app bar.

- Transition to `MainActivity` after the three entries are verified.
  - Remember to pass the email address into your Intent so `MainActivity` can display it
  - Remember to invoke `finish()` to purge its own entry from the Activity stack

You should be familiar with the details of each of these tasks, based on what has been covered so far. If you run stuck, take a look at the code for `LoginActivity` and mimic it to arrive at your solution. After your done, go ahead and consult the sample code for `SignUpActivity` and see how you did.

4.4.6 Handling MainActivity Logout Menu

In Section 4.4.2 you learn how to use the menu editor to create a menu structure for `MainActivity` but we never explain the details of using the menu in the activity. In fact, when you are at the main screen, the upper right corner (of the app bar) shows nothing but the blue background. This is because, the menu must be enabled programmatically from code using the `onCreateOptionsMenu()` method.

> **NOTE**
>
> Android Studio has an intelligent code completion capability. It will prompt you for possible completions of the current text, even when you misspell a few letters! To see this feature in action, just type O-P-T-M and observe the code completion list

Override the `onCreateOptionsMenu()` method, and replace the auto-generated return statement so the entire method looks like snippet code in Listing 26. The menu name `menu_main` should match the name created/used in Section 4.4.2. Recall the naming convention for Android resources: *res/menu/abc.xml* in your project panel becomes `R.menu.abc` in Java code.

Overriding the `onCreateOptionsMenu` will only *show* the menu on the app bar, but handling its selection requires a different method that must also be overridden in your code. Use Android Studio code completion feature to override the `onOptionsItemSelected()` method. When the menu is consumed/handled here, the method should return `true`, otherwise it returns false so the menu handling can propagate upwards. The completed code should look similar to Listing 27. The if statement verifies that Logout was selected. Since you previously made sure the activity stack was cleared regardless of how you arrived at the main screen, you will need to restart the `LoginActivity` in order to get it to appear on the screen. Immediately after you've

Kotlin
```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
  if (item.itemId == R.id.action_logout) {
    finish() // terminate this activity
    return true
  }
  return false
}
```

Listing 27: Implementing the logout menu option.

called `startActivity()` you will have the `MainActivity` call finish to destroy itself. It will get recreated via the login process once the user re-authenticates.

The `Menu` class has a property `title` that returns the string title of the menu. Don't feel tempted to use this method in the if-statement. The menu title may be different at runtime when your app is designed for multi-lingual support. Using `itemId` for comparing the menu with the ID in its XML definition guarantees a consistent result.

In addition to the `startActivity()` method you use in this chapter, there is another important method in Android that you have to use when the source activity expects a result passed back from the target activity. In that particular case, the source activity has to use `startActivityForResult` to launch the target activity and also overrides `onActivityResult` to capture the result. You will learn this technique in more details in a later chapter.

4.4.7 Navigation Graphs

Android Studio 3.4 (and later) provides a new tool that assists you in designing screen transitions with minimal amount of code. These tools and associated libraries are part of the Jetpack Navigation Component. Among the benefits of using Android navigation graphs:

- It automatically handles the activity stack/back stack
- It provides type safe argument passing between screens
- It handles transition animations

Passing data using Intent extras like we did earlier is considered unsafe. For instance, it would not prevent you from placing a numeric data by calling

Kotlin
```
myIntent.putExtra("ATOM_MASS", 47.867)
```

and in the target activity you make a logical error by writing

Kotlin
```
val atomMass = myIntent.getStringExtra("ATOM_MASS")
```

The SafeArgs plugin prevents you from making this logical error and flag it as an error at compile-time.

In navigation components parlance, your screens are a "destination". When the navigation tool was first launched, only Activities or Fragments were supported as a destination. A subsequent update to the tool supports dialogs as another kind of destination.

The navigation library is built from three interconnected elements:

- navigation graph: an XML file that captures the transitions among various screens in your app
- navigation host fragment: a UI fragment to add to your layout whose main job is to swap the destinations in/out of the current view
- navigation controller: a fragment object that works in conjunction with the navigation host fragment to instructs the UI fragment which destination to load/unload

The recommended practice for using the Navigation component is to design your app as a single Activity that controls the transitions among several Fragments. In our case, the three Activity classes you designed so far would turn into fragments and on top of that you have to create one activity that will host the navigation host fragment.

Before we dive deeper into navigation graph, let's introduce a new concept in Android.

Introduction to `Fragments`

The Oxford English dictionary defines "fragment" as:

> **frag·ment** /ˈfraɡmənt/
> *A small part broken off or separated from something*

But, what is a Fragment in Android? It is a class introduced in Android API Level 11 (Gingerbread) which allows us to decompose an Activity into smaller constituents components referred to as fragments. Fragments are typically used when the UI design requires a multiple-pane layout. For instance, an email app running on a large screen (e.g. a tablet) may present both a list of emails and the detailed contents of the selected email. This is a common usage pattern, widely-known as "*master-detail*". The email list can be managed by one fragment, the email details by another. The same application could be ran on a smaller screen (e.g. a smartphone) and the same fragments could be reconfigured to display one at a time. For example, when the email list fragment is shown, when the user taps an email in the list, the fragment would get swapped out with the email detail fragment.

In the context of navigation graph, fragments can be used by a single Activity for swapping between one destination and another.

It is important to understand that fragments must be hosted by an Activity. Hence, in another sense, fragments can be thought of as reusable components that implement a portion of a user interface. Similar to an Activity, a fragment has its own lifecycle. When an activity is composed of more than one fragment, each fragment executes its lifecycle independent from the others. Fragment lifecycle callback methods are similar to those of the Activity class. You will find methods like `onCreate()`, `onStart()`, etc. defined by Fragment. In addition, the Fragment class also defines additional lifecycle callback methods: `onCreateView()`,`onAttach()`,`onDetach()`, `onDestroyView`, etc.

The first step in moving on redesigning your app to use navigation graph is to create the first three fragments while **reusing** the layout files created from the previous steps.

1. Select New » Fragment » Fragment (Blank) to create a new fragment
2. In the next dialog enter `LoginFragment` as the fragment name and keep all the other default settings (layout name and language)

3. Edit `LoginFragment.kt` and remove everything inside the class declaration **except** the `onCreateView()` method. Then replace `R.layout.fragment_login` with `R.layout.activity_login`. Upon completion the file should look similar to Listing 28.

4. Edit `layout/activity_login.xml` and change the `tools.context` attribute from

```xml
tools:context=".LoginActivity"
```

to

```xml
tools:context=".LoginFragment"
```

Repeat the steps for creating `SignUpFragment` and `MainFragment`.

```kotlin
package edu.gvsu.cis.traxy

import android.os.Bundle
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import androidx.fragment.app.Fragment

class LoginFragment:Fragment() {
    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        return inflater.inflate(R.layout.activity_login, container, false);
    }
}
```

Listing 28: LoginFragment.kt

In addition, both `SignUpActivity` and `MainActivity` were created as "Basic" Activities. As such, their respective layouts both include an `AppBarLayout` with a `ToolBar` element embedded as a child. Edit `activity_main.xml` and `activity_sign_up.xml` in the code editor, and remove the `AppBarLayout` in both. Our navigation host component will already have a toolbar, so this will avoid having two of them.

Setting up the Navigation Graph

Under the resource folder you should find an XML file `navigation/nav_graph.xml`. Click to open/edit the navigation graph. You will find several prebuilt "destinations" (rectangular shapes) linked with "actions" (lines connecting one destination to another). You can "Safe Delete" all of them since you are going to replace with your own fragments just created. At the same time you can also remove all the prebuilt `xxxxFragment.kt` and `fragment_xxx.xml` files that Android Studio generated for you.

Your next step is to edit the navigation graph

1. Use the "New Destination" button at the upper left corner of the navigation editor to add all the three fragments just created.
2. In order to make the `LoginFragment` the initial fragment to be displayed, select it and then the "Assign start destination" button (labeled with a "Home" icon.)
3. Hover you mouse over the `LoginFragment` and drag an action arrow from the filled circle at the center of its right edge into the `SignUpFragment`. This arrow represents a transition from one fragment to the next.
4. With the arrow selected, go to its property editor and change its id to `action_login2signup`.
5. Repeat the steps for the other two actions until you accomplish a navigation graph that looks similar to Figure 4.18.

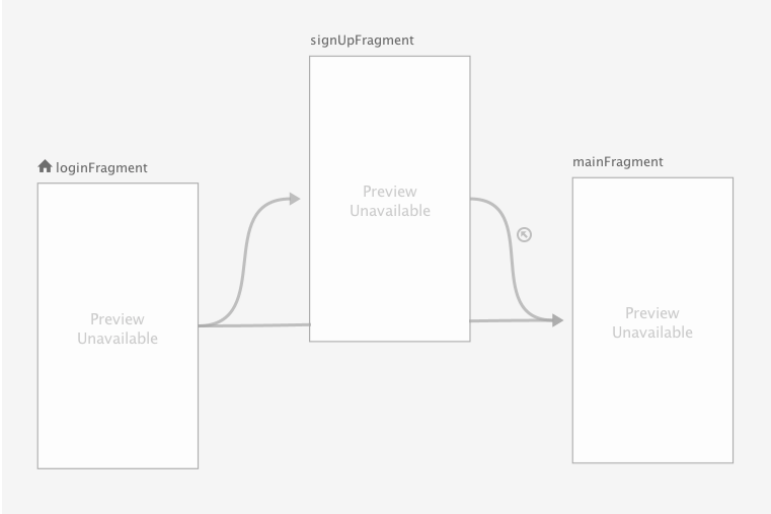


Figure 4.18: Navigation Graph

To tie them all together, you have to create a new Activity in charge of all the fragment transitions. Use New ⟩ Activity ⟩ Empty Activity to create `NavigationActivity` and be sure to check the "Launcher Activity" checkbox. Make the following changes to the your app

1. Open `activity_navigation.xml` and then add the navigation host fragment by inserting the following XML snippet as the only child of the top-level layout (ConstraintLayout):

XML

```
<fragment
  android:id="@+id/my_nav_host_fragment"
  android:name="androidx.navigation.fragment.NavHostFragment"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  app:defaultNavHost="true"
  app:navGraph="@navigation/nav_graph" />
```

The two most important attributes from the above `<fragment>` declaration are the fragment name (`NavHostFragment`) and the link to your navigation graph file (`nav_graph.xml`). The fragment tells `NavigationActivity` to be a navigation controller.

2. Edit `AndroidManifest.xml` and remove the `<intent-filter>` lines from the `LoginActivity` entry so that `NavigationActivity` is the **only** entry configured as the main launcher of your app.

3. To incorporate navigation support to the app bar add the following lines in `NavigationActivity`

Kotlin
```kotlin
// towards the end of onCreate of NavigationActivity.kt
val nc = findNavController(R.id.my_nav_host_fragment)
appBarConfig = AppBarConfiguration(nc.graph)
setupActionBarWithNavController(nc, appBarConfig)
```

The above code snippet requires the following declaration:

Kotlin
```kotlin
private lateinit var appBarConfig: AppBarConfiguration
```

4. To make the "Up" button behave identically to the "Back" button, override the following function

Kotlin
```kotlin
// om NavigationActivity.kt
override fun onSupportNavigateUp(): Boolean {
  val nc = findNavController(R.id.my_nav_host_fragment)
  return nc.navigateUp(appBarConfig) || super.onSupportNavigateUp()
}
```

Next, fill in the code needed in `LoginFragment.kt` by moving you previously wrote in `LoginActivity.kt`:

1. Override the method `onViewCreated` in `LoginFragment`

2. Move the block of code for handling the "Login" button from `LoginActivity.kt` to `onViewCreated()` just added in `LoginFragment`

3. You will see errors on all your calls to `findViewByID` method calls. Previously, this method was called on `LoginActivity`. In this context, you'll need to call it on the `view` reference passed to the method. For example:

Kotlin
```kotlin
val email = findViewById<EditText>(R.id.email)
```

becomes

Kotlin
```kotlin
val email = view.findViewById<EditText>(R.id.email)
```

4. You will also see an error on the line that loads the animation file (`shake.xml`). To fix the error, change

Kotlin
```kotlin
val shake = AnimationUtils.loadAnimation(this, R.anim.shake)
```

becomes

```Kotlin
val shake = AnimationUtils.loadAnimation(this.context, R.anim.shake)
```

5. Remove the intent creation and `startActivity()` calls and replace it with calls to the navigation controller. For example:

```Kotlin
val signUpIntent = Intent(this@LoginActivity, SignUpActivity::class.java)
signUpIntent.putExtra("EMAIL",email.text)
startActivity(signUpIntent);
```

becomes

```Kotlin
findNavController().navigate(R.id.action_login2signup)
```

Recall that `action_login2main` is one of the action arrows in your navigation graph. For now, don't worry about passing the user email from `LoginFragment` to `MainFragment`. You will deal with that in the next few minutes.

When not using navigation graph, to pass data from one activity to another we had to use the following technique:

- Call the `putExtra()` method in the "sender" activity to insert data payload into the Intent
- Call the `getXXXExtra()` method in the "target" to retrieve the payload from the Intent

Using navigation graph, the mechanism for passing data is different. In the navigation graph you have to inform the **receiving** destination the name and type of the incoming data. Let's edit the navigation graph and the affected fragment classes to allow passing data (email address) from the `LoginFragment` to the `MainFragment`,

1. Open the navigation graph and highlight the receiving destination (MainFragment)
2. Under its property editor click the plus button next to "**Arguments**"
3. In the next dialog box enter "`userEmail`" for name and select "String" from the type pull-down menu
4. Open `MainFragment.kt` and make the following changes:
   - Declare an instance variable to receive the incoming data

```Kotlin
lateinit var emailAddr: String
```

The keyword `lateinit` is required because we don't initialize the variable in the class constructor

   - Override `onCreate` and fetch the bundled data from its incoming arguments

```Kotlin
override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState);
  emailAddr = requireArguments().getString("userEmail").toString()
}
```

Here the string `userEmail` must match exactly the argument name defined in the navigation graph

- Override `onViewCreated` to show the email address on its UI

```Kotlin
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
  super.onViewCreated(view, savedInstanceState)
  val userId = view.findViewById<TextView>(R.id.user_id)
  userId.text = emailAddr
}
```

Here, `userId` refers to the widget id (`R.id.user_id`) as defined in the XML layout

5. Open `LoginFragment.kt` and change the `navigate` call from

```Kotlin
findNavController().navigate(R.id.action_login2main)
```

to

```Kotlin
findNavController().navigate(R.id.action_login2main,
  bundleOf("userEmail" to emailStr))
```

Rerun the app to confirm that the email address that you enter at the Login screen is indeed passed to thee Main screen correctly. Using what you have learned so far you should be able to add the missing code.

1. Write necessary code for handling the `register` button in `LoginFragment`

2. Write the necessary for for passing the registered email from the Signup screen to the Main screen

When using an intent, we had to call the `finish()` method when transitioning from `SignUpActivity` to `MainActivity` so pressing the back button from `MainActivity` will not go back to `SignUpActivity` but to the `LoginActivity`. Using navigation graph, this feature requires additional information passed to the navigation controller.

1. Edit the navigation graph

2. Highlight the action arrow from `SignUpFragment` to `MainFragment`

3. In the property editor fill in the "Pop Behavior" section

   - Select `LoginFragment` as the PopUpTo destination
   - Set the `PopUpToInclusive` flag to false

   These settings tell the navigation controller to pop the activity state up to (bot not including) the `LoginFragment`. Compared to iOS this feature is similar to "unwind" segue.

Displaying the Options Menu

To include the "Logout" menu into the newly redesigned fragment:

1. Edit the navigation graph, and add another action arrow `action_main2login` from `MainFragment` to `LoginFragment` and also filling the Pop Up Behavior as follows

   - PopupTo: `LoginFragment`
   - PopUpToInclusive: `true`

   It is important to set the flag to `true` so it would popup the old instance and you will see only a single instance of `LoginFragment` in the stack

2. Edit `MainFragment.kt` override the following two methods:

Kotlin

```
override fun onCreateOptionsMenu(menu: Menu, inflater: MenuInflater) {
  super.onCreateOptionsMenu(menu, inflater)
  inflater.inflate(R.menu.menu_main, menu)
}

override fun onOptionsItemSelected(item: MenuItem): Boolean {
  if (item.itemId == R.id.action_logout) {
    findNavController().navigate(R.id.action_main2login)
    return true
  }
  return false
}
```

In addition to these overridden methods, you also need to signal to the fragment that it has an options menu to display. You can accomplish this by adding a single line of code to `MainFragment`'s `onCreate` method:

Kotlin

```
setHasOptionsMenu(true)
```

Now the scene transitions have been completely redesigned, you should be able to remove unused files:

- Edit `AndroidManifest.xml` and remove the `<activity>` entries for `LoginActivity`, `SignUpActivity`, and `MainActivity`
- Remove the corresponding activity class files: `xxxxActivity.kt`
- Optionally, to reflect the fact that the XML files are associated with a fragment (not an activity) you may want to rename `activity_login.xml` to `fragment_login.xml` and likewise for the other two files.

Again, recompile and rerun confirm that everything works as expected.

4.4.8 Adding ViewModel

All the design architectures that have been proposed for designing an application (mobile, web, or desktop) have a common goal of making clear separation of concerns among various components of an application. Using the Jetpack bundle, the current best practices for organization Android apps is using the Model-View-ViewModel (MVVM) architecture. To encourage Android developers to embrace this architecture, the Android team provides the `ViewModel` class and other related utilities.

In order to practice working with this architecture, we will incorporate the `ViewModel` class into our simple app. Recall that the main responsibility of a ViewModel is to hold all the data for the UI. To allow the UI to store data into the ViewModel, the UI holds a reference to the ViewModel object and this relationship works only one-way; the ViewModel **does not** hold any reference to the UI. To "retrieve" data from the ViewModel, the UI is required to use the "Observer" pattern. In this case, the UI is a passive entity and the ViewModel will notify the UI for any data changes.

In our case the data are: user email and password. But for our starter implementation, we will store only the user email in our ViewModel.

Let's begin by creating the view model class:

Kotlin

```
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.ViewModel

class UserDataViewModel : ViewModel() {
    var _usr = MutableLiveData<String>()

    // Declare a property userId and its getter
    val userId get() = _usr
  }
```

The ViewModel class works together with the `LiveData` generic class. When we instantiate the class, we pass `String` to the generic type because our email address is of type `String`. Specifically for our case we use `MutableLiveData` since we to allow the ViewModel to be altered by the UI.

To demonstrate how the ViewModel works with the navigation graph, we will intentionally refrain from passing the user email when the app navigates into the target destination. Instead each fragment will store/retrieve the data into/from the ViewModel:

- The `LoginFragment` and `SignUpFragment` will store the user email into the ViewModel
- the `MainFragment` will retrieve it using an observer

To access the ViewModel your activity/fragment is required to use the factory class provided by the Jetpack library. Since the ViewModel automatically its data according the the lifecycle of the UI, the factory class must be informed with the lifecycle owner. In our case the three fragments must refer to a **common** shared instance of the ViewModel and the fragments are under the control of `NavigationActivity` the lifecycle owner is actually the activity and not each individual fragment. Therefore, inside `LoginFragment` and `SignUpFragment` you must get the **shared instance** of ViewModel by calling the following statement

Kotlin
```
lateinit var viewModel: UserDataViewModel

override fun onActivityCreated(savedInstanceState: Bundle?) {
  super.onActivityCreated(savedInstanceState)
  viewModel = ViewModelProvider(requireActivity()).get(UserDataViewModel::class.java)
}
```

To make sure that the activity already exists when the view model is initialized, the above statement must be called **after** the `super` call in the `onActivityCreated()` method. The helper class `ViewModelProvider` is defined in the lifecycle extension library and requires the following line added to your `build.gradle`:

Gradle
```
implementation 'androidx.lifecycle:lifecycle-extensions:2.2.0'
```

To store data into the ViewModel and transition, for instance, from `LoginFragment` to `MainFragment` you would replace the following statement:

Kotlin
```
// In LoginFragment.kt
findNavController().navigate(R.id.action_login2main, bundleOf("userEmail" to emailStr))
```

to

Kotlin
```
// In LoginFragment.kt
viewModel.userId.value = emailStr
findNavController().navigate(R.id.action_login2main)
```

To observe changes to the View model data in `MainFragment` you would add the following statement:

Kotlin
```
// in MainFragment.kt
lateinit var viewModel: UserDataViewModel

override fun onActivityCreated(savedInstanceState: Bundle?) {
  super.onActivityCreated(savedInstanceState)
  viewModel = ViewModelProvider(requireActivity()).get(UserDataViewModel::class.java)
  viewModel.userId.observe(this.viewLifecycleOwner, Observer { z ->
    val userId = view?.findViewById<TextView>(R.id.user_id)
    userId?.text = z
  })
}
```

ViewModel Initialization Using Property Delegates

Throughout the sample code we have been using `ViewModelProviders.of` to initialize our ViewModel class. A more Kotlin idiomatic approach of initializing the class is to use property delegate and replace the `lateinit` approach with the following line:

Kotlin
```
val viewModel by activityViewModels<UserDataViewModel>()
```

as well as removing the corresponding line in the `onActivityCreated()` function. Be aware that the Kotlin extension library provides two delegate `viewModels` and `activityViewModels`. In our case, we want to use the latter since the ViewModel should be associated with the Navigation activity and not with individual fragments.

4.4.9 Implicit `Intents`

Throughout the previous sections, you discovered the power of the `Intent` class for launching a target activity from a source activity. In the examples you have seen so far, the target activity resides within your app. In the Android parlance, such an `Intent` is *explicit*; the target activity class is explicitly specified as one of the arguments passed to the `Intent` constructor. In an *implicit* `Intent`, the target component is not explicitly specified, but only its *desired action*. When an implicit `Intent` is issued by an app, the Android runtime searches for an appropriate component that knows how to handle the action, by comparing the desired action in the `Intent` with the `<intent-filter>` in the manifests of the apps installed on the device. If more than one app is able to handle the desired action, the Android runtime will present a dialog to let the user select the app to service the action.

To give you some idea why implicit Intents are of practical use, let's look at some of the common actions invocable via `Intent`:

- `ACTION_CALL` to initiate a phone call
- `ACTION_SET_ALARM` to create an alarm
- `ACTION_SET_TIMER` to create an timer
- `ACTION_IMAGE_CAPTURE` to capture an image
- `ACTION_VIDEO_CAPTURE` to record a video
- `ACTION_PICK` to open the contact list and select a contact
- `ACTION_SENDTO` to send an email/text with no attachment
- `ACTION_SEND` to send an email/text with one attachment
- `ACTION_SEND_MULTIPLE` to send an email/text with multiple attachments
- `ACTION_VIEW`: view a map, a media file, or a web page.

A complete list of common intents can be found at Android API Guides: Intent and Intent Filters.

Recall that you invoke `putExtra()` on an `Intent` to pass data to a target activity. You can do likewise with implicit `Intents`. For instance to set an alarm you would use the following snippet:

Kotlin

```
val alarm = Intent(AlarmClock.ACTION_SET_ALARM)
alarm.putExtra(AlarmClock.EXTRA_HOUR, hour_to_set)
alarm.putExtra(AlarmClock.EXTRA_MINUTES, min_to_set);
startActivity (alarm);
```

When you expect a result from a target activity, you must transition to the target activity using `startActivityForResult()`, and the source activity must override `onActivityResult()` to capture the result. A similar technique also applies to implicit intents. For instance, selecting a contact using `ACTION_PICK` returns a result to the source activity.

Be aware that the same action may be used for more than one task. For instance, `ACTION_VIEW` can either open a map, a media file or a web page. How does the system know which specific task to invoke? In addition to the action name, the runtime resolution algorithm also uses the data (MIME type and URI schemes) and category to resolve implicit `Intent`s. For instance, to open a map you will use `ACTION_VIEW` with URI scheme `geo:`*latitude*,*longitude*. But, to open a web page you will use the URI scheme `http:`*URL-of-the-page*.

4.4.10 Unresolved `Intents`

Since the target component of an implicit `Intent` is resolved at runtime, it is possible that no app is available to handle the desired action. Consider a hypothetical situation where an implicit Intent for viewing maps is issued, but no suitable maps apps are currently installed. An attempt to execute unresolved `Intent`s will crash the issuer. To avoid this situation, you should first invoke `resolveActivity()` on the intent and invoke `startActivity()` only when the former method returns non-null object.

Kotlin

```
1 val impl = Intent(____);
2 if (impl.resolveActivity.packageManager != null) {
3     startActivity (impl);
4   } else {
5     // the intent cannot be resolved
6   }
```

—If we want users to like our software, we should design it to behave like a likeable person.

Alan Cooper

5

Laying Out and Styling Flexible Mobile User Interfaces

Chapter Objectives

After reading this chapter you should be able to

- Understand the unique challenges of creating consistent functional user interfaces for small hand held devices.
- Create constraint-based layouts on iOS and Android user interfaces.
- Apply consistent styling to iOS and Android user interfaces.

Organizing and laying out user interfaces on smartphones and tablets is in many ways similar to laying out interfaces for desktop application. However, there are some distinct differences to be aware of.

- Desktop applications benefit from having ample screen real estate, often making it possible to layout all the components needed to complete one or more tasks on a single screen. On the contrary, the amount of screen real estate available to mobile applications is limited, often forcing the designer to break up even the most simple of tasks into several different screen layouts. Hence the manner in which the user is able to navigate through these screens takes a significant role in the design process of mobile applications. Most mobile applications include some sort of universally available *toolbar* that displays a combination of navigational user interface controls as well as actionable user interface controls associated with the current display context.
- The orientation at which a mobile device is held may change dynamically while the app is

running, so often user interface layouts must be designed to display in both portrait and landscape orientations.

- The underlying physical form factor of mobile devices can vary widely in terms of screen size, resolution, physical keyboard presence/layout, and specialized hardware
- Taps on a touch screen via the user's finger, and possibly a stylus in mobile applications are analogous to mouse pointer / clicks in desktop applications. However, mobile application users frequently use additional gestures to interact, including swipes, zoom, pinch, long hold/press, or even multi-finger gestures.[1]
- Input methods which are unique to most mobile devices are image data via camera, location data via GPS receiver, accelerometer, pressure sensor, and other sensors built into a mobile device. In particular, accelerometers provide a unique input methods on mobile devices such as *shake gesture* and *swinging motions*.

5.1 Laying out Mobile User Interfaces

User interfaces today are often developed via a couple of different approaches. The *programmatic approach* involves writing code that dynamically computes the position and size of every component on the screen. User interface frameworks that utilize this approach (Java Swing, Gnome Gtk, Qt3) will often provide higher level layout mechanisms to make this task easier, but in essence it involves writing code in the SDK's native language to exhaustively compute the exact positions and size of every element on the screen. While this approach gives the programmer very flexible low level control of how exactly the screen will appear, it tends to produce a significant amount of source code that is difficult to understand and maintain.

A more recent alternative approach can be described as the *declarative approach*. In this approach, the developer expresses various geometric relationships among the various components appearing in a a layout, in the form of constraints. The underlying system automatically computes the location and sizes of the components, based on the constraints. Typically, these constraints are expressed via tooling, or in syntactic formats specialized for the task.

Both of these approaches are available to Android and iOS developers. However, it is the declarative approach that is most commonly used for designing flexible layouts for mobile applications.

Android and iOS face many of the same layout challenges in that both must support flexible user interfaces in a situation where target devices increasingly vary widely from each other in terms of screen size, resolution, and physical form factor. Given that Android devices are designed and manufactured by a large number of competing manufacturers, the target devices tend to be even more diverse than iOS. In addition, user interfaces on both must take into account device orientation, gesture input via touch and the accelerometer, and incorporate additional input/output modalities provided by various on-board sensors.

Despite the common challenges and conceptual similarities, the respective tools and techniques used to lay out user interfaces on these two platforms are rather different. This is mostly due to the separate pedigree that each platform has. iOS's Interface Builder descends from an earlier

[1] With touch screens becoming more common on conventional desktop computers, the use of these gestures are becoming more universally available to all applications.

NEXTStep tool with the same name. While Interface Builder's file format for both storyboard and xib files is XML, the format is proprietary, and as such, is not editable outside of Interface Builder.

While the Android layout tools have evolved significantly since its initial release, the same basic process is followed in that layouts are achieved by roughing in the interface using a somewhat primitive (at least compared to Interface Builder) drag and drop editor, and often fine-tuning the interface by editing the resulting XML layout file directly.

5.1.1 UI Styles and Themes

The concept of user interface *styles* and *themes* are familiar to technical and non-technical users alike. While the specific definition of these terms may vary based on the underlying framework, we can conceptually think of a style as a description of how something should appear in a user interface, while a theme is collection of styles that is applied to the entire application. For example, your favorite word processor allows you to vary the styling of text by using various pre-defined styles such as "Heading 1" and "Block Quote". Futhermore, each document template might have its own unique set of such pre-defined styles, which we could of think of as the document's theme. Another common example is web applications where styling is separated from the HTML markup in the form of cascading style sheets (CSS). The beauty of approaching styling in this way is that the appearance of a document or application can be easily managed over time. For example, tweaking a style changes a narrow aspect of the user interface across the entire application. Similarly, changing the theme effectively re-skins an entire application or website.

The concepts of styles and themes are also important in managing the look and feel of mobile applications. For example, you may want all buttons in your application to have a particular background color and rounded corner of certain radius. In the future if you would like to change how buttons are styled, you only need replace the style. You can avoid having to go back through your application and finding and updating the properties of every button on every screen - a process in which you will inevitably miss a few!

Despite Apple's reputation for crafting beautiful easy to use products, the provisions iOS makes for introducing styles and themes are rather primitive compared to the mechanisms available on Android. Apple's UIKit defines a protocol named `UIAppearance` that is integrated into all its user interface controls. Basically, developers can access the appearance property on any user interface control defined by UIKit and receive back a proxy that implements the `UIAppearance` protocol. Style properties set on this proxy are applied to all instances of that class throughout the app. It is up to the developer to build upon this higher level concepts of grouping multiple style settings into themes, handling situations where the same component is styled differently within the same app, and other similar issues.

Android on the other hand, introduces first class mechanisms (essentially XML files) that allow developers to define styles and themes independent of their Java code and XML layouts, and control the scope at which these settings apply within their application.

Perhaps the lack of higher level support for styling and themes on iOS harks back to Apple's traditional minimalist approach to design. Early on, Apple had very specific user interface guidelines for iPhone apps, and developers were strongly encouraged to adopt uniform approaches in their designs, using stock Apple user interface controls, exactly as Apple guidelines recommended. Google on the other hand, always promoted openness and a high degree of customization on Android. Early on many of the device manufacturers took liberty in introducing their own unique UI skins (e.g., MotoBlur, HTC Sense) on their devices, in an attempt to differentiate from

their competitors. Over the years, as both app ecosystems have become more crowded, we've seen both move closer to the same approximate happy medium. For example, most big budget iOS apps in the App Store are highly customized and don't rely as much on Apple's stock UI controls. On the other hand, Google has published user interface guidelines for Android and spent a significant amount of effort in helping developers adopt best practices in implementing their user interfaces. As a consequence, Android apps are beginning to adopt similar approaches to common tasks, and the overall experience has improved significantly in recent years.

In the remainder of this chapter, let's explore the layout and styling features of both Android and iOS. Our goal is to make sure our screens look good, and also layout well on all screen sizes and densities. The color swatch we would like to use to style Traxy app is shown in Figure 5.1.

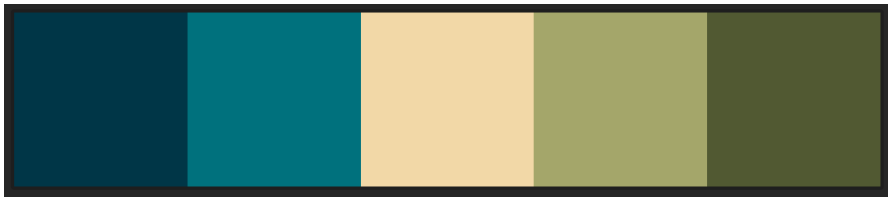


Figure 5.1: The set of colors we are going to use to style our Traxy app.

Color	RGB		Hex
	Percentage	0–255	
	(0.00, 0.21, 0.28)	(0, 54, 72)	`003648`
	(0.00, 0.44, 0.49	(0, 113, 126)	`00717E`
	(0.95, 0.85, 0.64)	(243, 217, 164)	`F3D9A4`
	(0.64, 0.65, 0.40)	(164, 167, 103)	`A4A767`
	(0.32, 0.35, 0.19)	(81, 90, 48)	`515A30`

5.2 Layout and Styling on iOS

In your Traxy application so far, you've deliberately done your layouts in Interface Builder on a specific iPhone model (e.g. an iPhone X or an iPhone XR). As long as you ran the app on the simulator that corresponded to the model you did our layouts on, when running the app, everything appeared exactly as you placed it in Interface Builder. Another important assumption you made was that the device would always be in portrait orientation.

In reality of course, none of your assumptions would be very realistic, as your users will have a variety of different devices, and will likely attempt to use the app in both portrait and landscape orientation. Let's do a bit of experimenting and see what happens if any of your assumptions don't hold. First, try running the app in the simulator you've been targeting so far, and switch the orientation to landscape-right. With the simulator window in focus, go ahead and use the keyboard shortcut ⌘ + → to rotate the device 90 degrees right. As you can see in Figure 5.2, when the screen rotates, the controls are no longer centered. Furthermore, the label and signup button on the bottom of the screen are no longer visible!

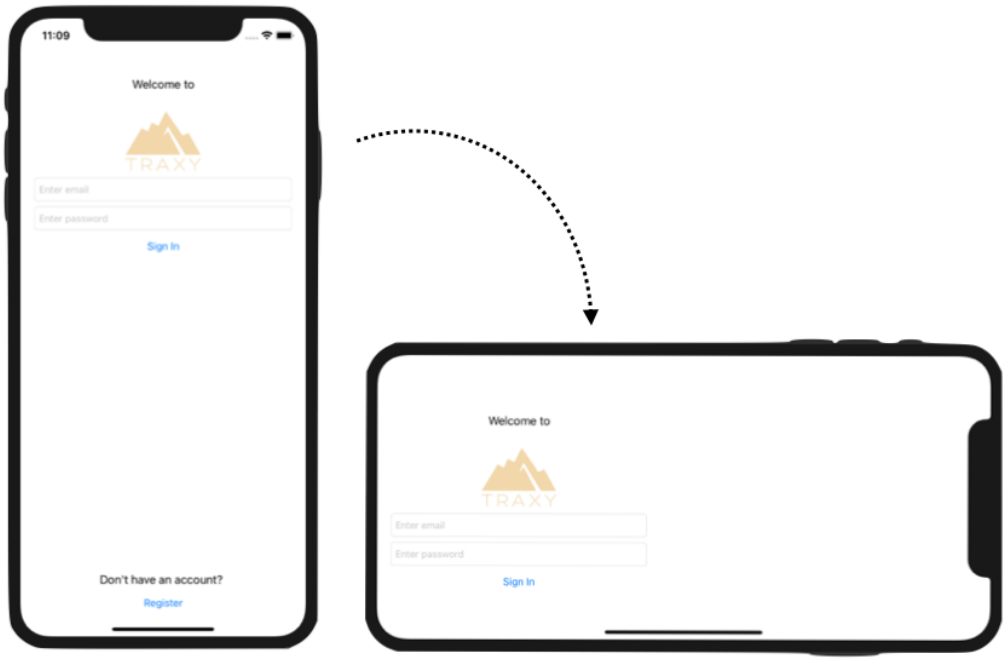


Figure 5.2: When we rotate the login screen to landscape-right orientation, the user interface controls are no longer centered, and the signup button is no longer visible.

Next, go ahead and stop the app, and then change the target device to a larger screen (e.g. the iPhone 11 Pro Max). In this case, we have problems with our layout even in portrait mode. Not only are the fields no longer centered as before, but now the signup label and button are floating a bit above the bottom of the screen, as shown in Figure 5.3. Clearly, these are not outcomes we can accept in a production app. Let's take a look at the provisions made available to get our layout to look good on every device no matter what orientation it is in.

5.2.1 Auto Layout Overview

As mentioned previously, we can programmatically correct the problems identified above. That is, there are APIs that allow us to query the size of our root window that our app is displayed in. We could then use that information and the app-specific information (e.g. our size and location

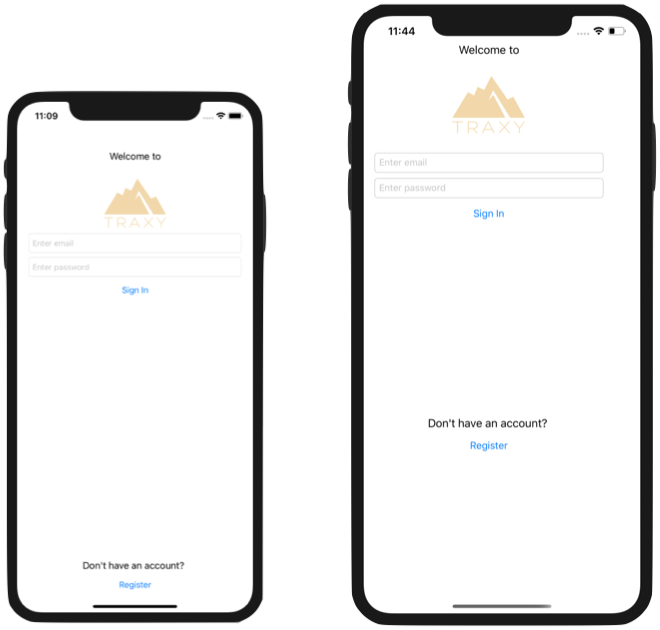


Figure 5.3: When we run our app on a larger screen (e.g. iPhone 11 Pro), the user interface controls are no longer centered, and the signup button floats well above the bottom of the screen.

preferences for each UI component in the view hierarchy) to compute the location and size of each element, and use the appropriate `UIView` attributes and methods to set those values before the view is rendered on screen. This approach will work, but it requires a bit of heavy lifting and will not be resilient to change in the future. It will also be difficult to get an intuitive grasp for how things are being laid out by simply reading the code. Happily, iOS provides us a better way.

Auto Layout was introduced in iOS 6. It is a constraint-based layout engine inspired by the Cassowary constraint solving toolkit developed at the University of Washington [5]. Auto Layout allows the developer to specify a set of constraints, which are essentially equations relating the size and position of UI views to neighboring views or a containing superview. All of the constraints taken together form a system of linear equations that when solved provides the location and size of each view on the screen. Ultimately, for each view we need to be able to compute two horizontal values (e.g. the x position, and width) and two vertical values (e.g. the y position and the height). So for every view we need four constraints.

These constraints can be added to your views directly in your code, via Interface Builder, or using a combination of constraints added (or modified) in your Swift code, and constraints added via Interface Builder. In the remainder of this section you will learn to use Interface Builder for adding constraints to your views.

By default, when you simply dragged views from your object library into the Interface Builder canvas, the views got hardwired right where you placed them. At this point, Auto Layout is simply ignored. However, as soon as you add your first constraint, Auto Layout goes to work and starts trying to solve its set of constraints. At this point there are several possible situations you might find yourself in.

- *Auto Layout has exactly enough information to resolve the position and size of each view.* This is exactly the situation you want to be in!
- *Auto Layout doesn't have enough information to resolve the position and size of each view.* In this case, warning messages will be printed out in yellow in the structured view of your Interface Builder editor on the left. If you click on these warnings, Interface Builder will attempt to generate the missing constraints. Sometimes this works, and sometimes it doesn't. In any case, if you don't finish specifying your constraints completely, things will likely not layout properly.
- *Auto Layout has too much information.* You may accidentally over-constrain a layout. In this case you've given ambiguous or possibly conflicting constraints. Once again Interface Builder will provide error messages in red within the structured view of your Interface Builder editor on the left. If you run the application in this state, Auto Layout will attempt to eliminate the offending constraint or it simply might crash! Either way, information regarding the problematic constraint(s) will be printed out on the console window. This information can be helpful in troubleshooting the problems in your layout.

It turns out that some views have an *intrinsic size*. For example, a UILabel can compute its size based on the value of its text attribute as well as the size and style of its font. Similarly, a UIImageView can compute its size based on the image it is displaying. If you drag one of these views into your scene, you can get away with just specifying constraints to establish its position. At the same time, you could also specify a height and/or width in constraints as well, possibly overriding the implied intrinsic size.

Formally, Auto Layout constraints are equations in the form:

$$\text{Item1.attribute1} = \text{Multiplier} \times \text{Item2.attribute2} + \text{Constant}$$

where Item1 and Item2 denote references to views, and attribute1 and attribute2 refer to either the location or size attributes of the views. Valid location attributes are top, bottom, leading, trailing, horizontal center, or vertical center. Valid size attributes are width or height. You cannot mix location and size attributes within the same equation. The multiplier will often be set to 1.0 (the identity multiplier) but other values can be used as well. The constant simply adds a constant value to the equation. For example, if we wanted to place Button2 standard distance (8) to the right of Button 1, we could introduce the following constraint:

$$\text{Button2.leading} = 1.0 \times \text{Button2.trailing} + 8$$

Note that when dealing with positional attributes, the second item must always be specified. However, when dealing with size attributes, its possible to set the multiplier to zero and Item2 to a special `Not An Attribute` value. This in effect sets that particular size to a constant value.

5.2.2 Introducing Auto Layout Constraints in Interface Builder

Having covered the basic idea behind Auto Layout, let's take a look at how constraints can be created visually in Interface Builder. Interface Builder provides a variety of different mechanisms to establish layout constraints within a scene, including drag and drop, the alignment and pin tools, `UIStackView`, the Xcode Editor menu, or via Interface Builder's Auto Layout constraint resolver. The approach is taken to add constraints to a layout often depends on the specifics of the layout and what exactly is trying to be accomplished.

Adding Auto Layout Contraints via Interface Builder

The *drag and drop approach* to adding constraints is accomplished by simply clicking on an item within a view with the ctrl key down, and then dragging the mouse pointer over to either another child view or to one of the edges of the view containing the selected view, and then releasing the click. Figure 5.4 demonstrates this technique, where we've held ctrl and clicked on the description `UITextField` and then dragged over to the title `UITextField` just above it. In this case, the title field is highlighted in light blue when we hover over it. When we release the mouse button, a contextual popup menu displays a number of different types of constraints. Note that based on the direction we dragged, the most likely type of constraint (in this case **Vertical Spacing**) shows up on the top of the menu. When we select **Vertical Spacing** on the menu, Xcode immediately inserts a constraint which establishes the vertical distance between these two `UITextField` views. The constant in that constraint is set to the actual distance between these two fields as laid out in Interface Builder. If we used the blue guidelines when we dropped or repositioned the views with respect to each other, it will be set to the standard recommended spacing between these elements.

We can also use the drag and drop approach to establish constraints between a view and its containing view. In Figure 5.5 we hold ctrl and click on the title label, and then move left to the leading edge of the containing view. When the entire containing view is highlighted we release the mouse button, and select **Leading Space to Safe Area** from the context menu that is displayed. This places the label the standard distance from the leading edge of the containing view if we used the blue snap guides when we placed it.

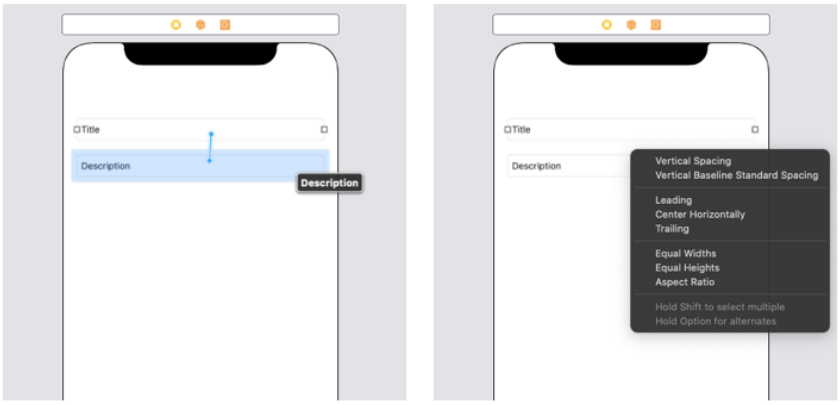


Figure 5.4: The drag and drop method of adding constraints in Interface Builder a) Click and while holding the Control key on a view, and drag to a peer view. b) When the click is released a context menu pops up with various constraint options.

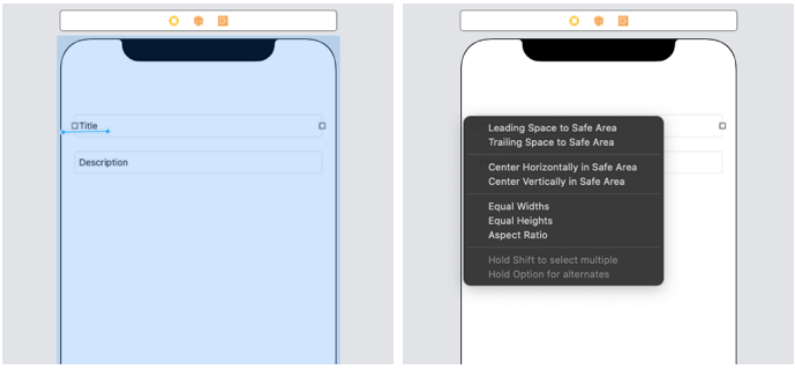


Figure 5.5: Dragging and dropping from a view to the containing view a) Click and while holding the Control key on a view, and drag over to an edge of the containing view b) When the click is released a context menu pops up with various constraint options.

In addition to the drag and drop approach, Xcode also provides us the *alignment and pin tools*. These tools can be activated by selecting a `UIView` that you wish to constrain, and then clicking on the appropriate tool icon in the lower right hand corner of the Interface Builder canvas, as shown in Figure 5.6.

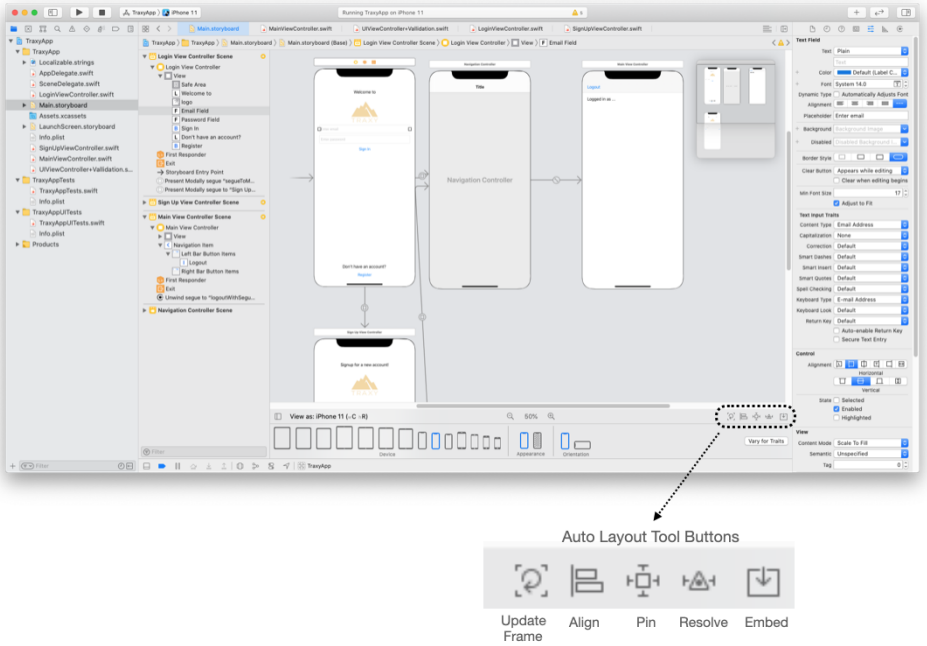


Figure 5.6: The Auto Layout Tool Buttons in the lower right corner of Interface Builder.

The alignment tool button would be typically used by selecting multiple views and then aligning them by edges, or centers. Alternately, we could select a single view and use the alignment tool to center it within its containing view. In Figure 5.7 we've selected the three text fields on the Traxy signup form, and then clicked on the alignment tool button. We can now add constraints to effect the alignment of these two views with respect to each other.

The pin tool button would be typically used by selecting a single view and then pinning down its location with respect to adjacent views or its containing view edges. In addition we could pin its height and width to constants as well. In Figure 5.8 we've selected the welcome label, and then clicked on the pin tool button. Using the schematic at the top, we've designated the offsets of the leading, trailing, and top edges from its nearest neighbor in each of those directions. In this case, the nearest neighbor in all three would be the containing view. The *Constraint to margins* checkbox can be used to indicate the offsets are relative to the containing view's layout margins.

You may have noticed at the bottom of both the alignment and the pin tools there is an option labeled **Update Frames** that defaults to **None**, with additional options *Items of New Constraints* and **All Frames in Container**. This option allows you to have the views impacted by the constraints, or all views in the frame updated. This is useful if you know that the constraints you are adding will change the location or size of one or more views and you would like that update to happen

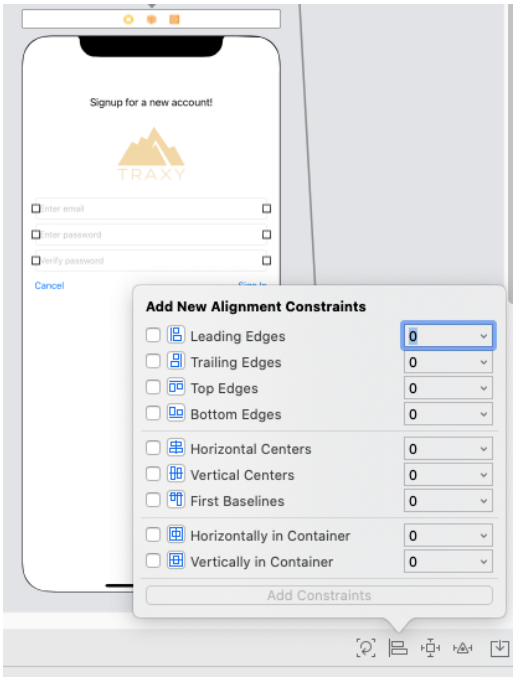


Figure 5.7: The Auto Layout Alignment Tool allows us to align views with other views or within its containing view.

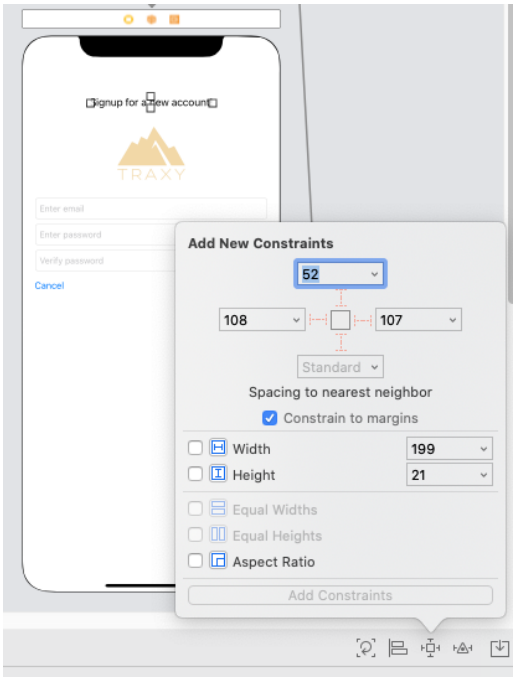


Figure 5.8: The Auto Layout Pin Tool allows us to pin down a view with respect to adjacent views or its containing view. We can also constrain the width and height of a view using this tool.

automatically. If a view in Interface Builder is not positioned or sized according to its Auto Layout constraints, Interface Builder will warn us of this.

UIStackView

Apple introduced the `UIStackView` class in iOS 9. Stack views greatly simplifying the layout of two or more items in either a row (e.g. a stack view in horizontal orientation) or a column (e.g. a stack view in a vertical orientation). A stack view is laid out based on a few simple properties:

- *axis* - Determines whether items are stacked horizontally or vertically.
- *alignment* - Determines how the stack view is laid out perpendicular to its axis.
- *distribution* - Determines the stackview layout along its axis.
- *spacing* - Determines the minimum spacing between views within the stack.

It is also possible to nest stack views within stack views, making this construct a very convenient way to construct a number of more complex layouts, while minimizing the number of Auto Layout constraints you have to introduce. The easiest way to introduce stack views into your layout is to lay out your items in their approximate locations, and then select the set of items you wish to lay out horizontally or vertically, and then click on the embed tool button in the bottom right corner of the Interface Builder window (refer to Figure 5.6). A stack view with the correct orientation will be inserted into the view hierarchy, and the selected views added as its children. You can then use the attributes inspector to set the properties described above, and add any necessary Auto Layout constraints directly to the stack view itself to properly position (and optionally size) it as needed in its containing view. We will demonstrate the use of stack views in our subsequent example.

Viewing and Editing Auto Layout Constraints in Interface Builder

Once constraints have been added to a layout in Interface Builder, there are a number of ways to view and edit these constraints. Interface Builder lists out all constraints within the document outline panel to the left of the layout area of the canvas. The constraints are listed hierarchically in the document outline, by listing them after the closest view that contains both views referenced in the constraint. Alternately, if there is a large layout with many nested views, it may be easier to simply select the view in Interface Builder and view the constraints that reference that view within the size inspector to the left.

Constraints can be edited via the attributes inspector on the right by selecting them in the document outline. Alternately, if viewed within the size inspector, clicking on the Edit button associated with the constraint allows editing within a popup window.

Resolving Auto Layout Issues

As mentioned previously, while adding constraints to a view hierarchy, your layout may be over or under-constrained. This situation will be reported to you within the document outline in the form of error or warning messages. As shown in Figure 5.10, you can click on the red (or orange) dot in the upper right of the document outline pane (when present). Interface Builder will then

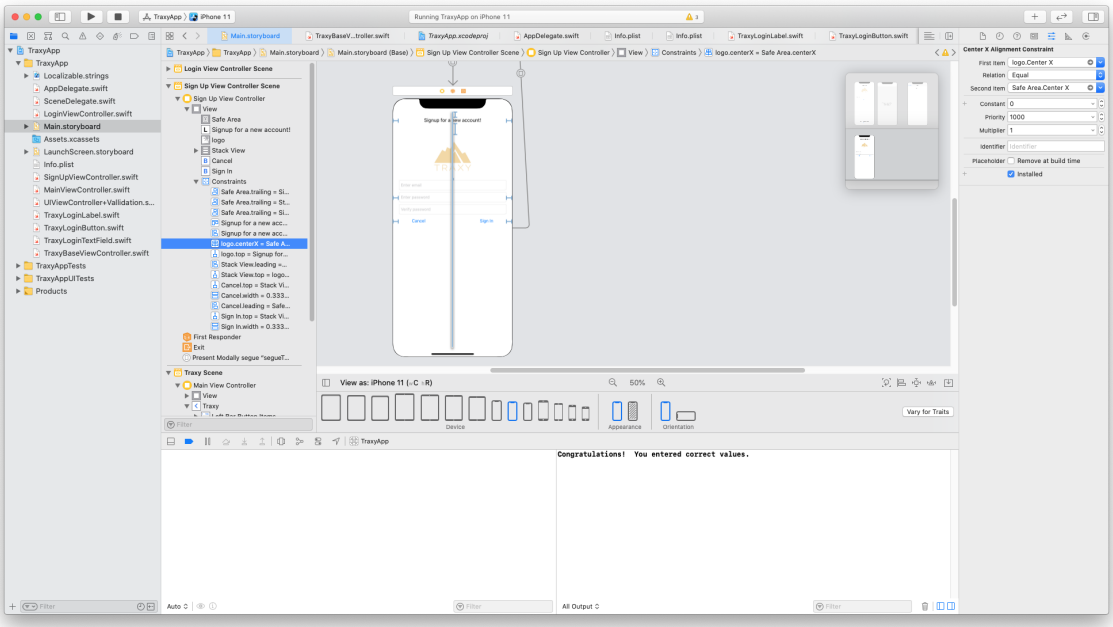


Figure 5.9: Once constraints have been added to a layout, Interface Builder makes it easy to view and edit the constraints.

display a list of all such errors and warnings. If you click again, Auto Layout will offer you possible solutions to the problems. Alternately, you can select a particular view or set of views within a layout and click on the Resolver button in the lower right corner of the Interface Builder canvas and automatically add any missing constraints, or even reset all constraints for the selection with Auto Layout suggested constraints.

These suggested constraints can be a mixed bag. Sometimes they are exactly what you are looking for, but sometimes they aren't. This is mostly dependent on how logically you positioned the views when you drag and dropped them into the canvas, and Xcode's mood at the time!

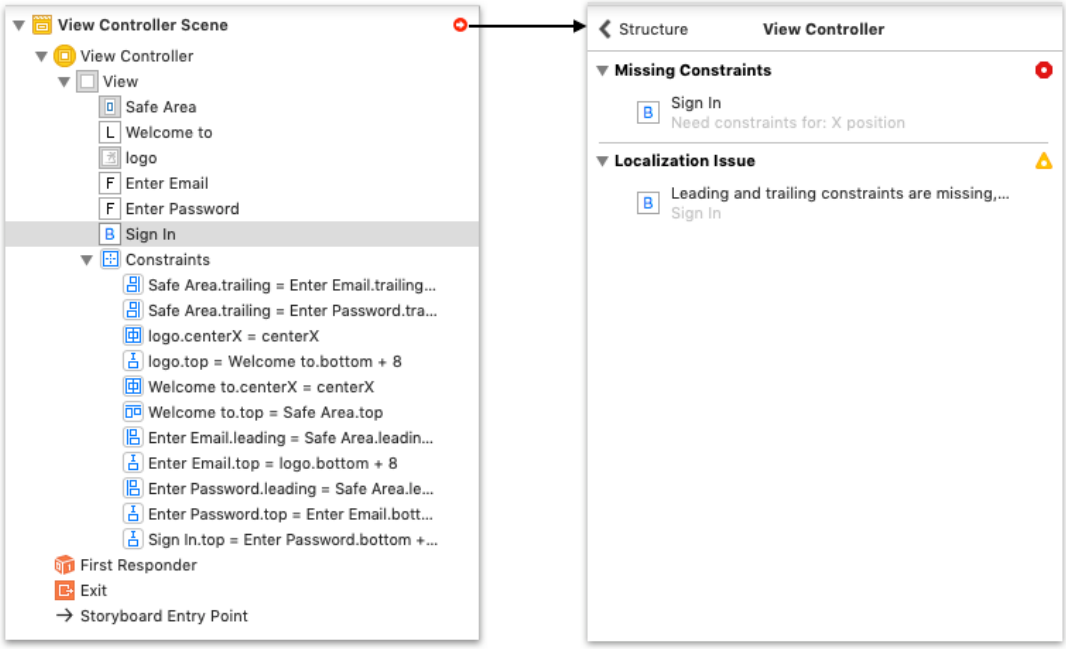


Figure 5.10: The Auto Layout resolver reports problems in the document outline in the form of errors and warning. Collecting on the red (or orange) dot in the upper right of the document outline will give you a verbose list of the issues and provide possible solutions.

As a general rule thumb, we would suggest you not attempt running your app until you've managed to resolve all the Auto Layout errors and warnings listed by Interface Builder. If the issues are not addressed, surprising things may happen in your layout, such as views being positioned entirely off screen at run-time and hence invisible to you.

5.2.3 Adding Auto Layout to Traxy

Going back to our Traxy application, let's go ahead and add Auto Layout constraints to each screen to correct the issues we identified earlier. For each of the screens we will summarize our strategy as to how we will proceed to constrain the layout. We will then provide a detailed description of how the constraints are added. Readers are encouraged to load the accompanying sample code

provided into Xcode and review the details of each set of constraints to best understand how the layouts are implemented.

The Login Screen Layout

Our strategy for layout of this screen is as follows:

1. Pin the welcome label standard distance from the top of and center it in its containing view.
2. Pin the logo image standard distance vertically from the welcome label and center in containing view.
3. Introduce a vertically oriented `UIStackView` containing the email and password. Position the stack view standard distance vertically from the logo, and stretch it out horizontally so it is standard distance from the leading and trailing edges of the containing view.
4. Center the sign in button horizontally, standard distance below the stack view. Set its width to 33 percent of the horizontal space available.
5. Center the register button horizontally, standard distance above the bottom edge. Set its width to 33 percent of the horizontal space available.
6. Pin the "Don't have an account" label standard distance from the register button, and leading and trailing edges of the parent view.

First, we select the welcome label and resize it horizontally so that it fills the screen horizontally. As you resize towards the leading and trailing edges of the screen use the vertical blue dashed guide lines that pop up when you are standard distance. Once you've resize horizontally, then move the label so it is standard distance from the top of the screen. Once again, use the horizontal dashed blue guideline to determine how far to be from the top of the screen. Next, click the Pin tool button in the lower right. In the tool we activate nearest neighbor settings for the leading, trailing, and top edges, setting them to the constants 20 for each. Make sure the **Constrain to margins** box is not checked. Click the button on the bottom labeled **Add 3 Constraints**. As a last step on center the text within the label by selecting it and clicking the text center button in the attributes inspector on the right side of Interface Builder. If the text width changes (e.g. due to localization or other updates) it will still be fully centered on the top of the screen.

Second, ctrl click on the logo image and drag up to the welcome label. Upon release, select **Vertical Standard Spacing** from the contextual menu. Next ctrl click once again on the logo image and drag over to any location in the containing view where there isn't a child view. Once the entire containing view is highlighted, release and select **Center Horizontally in Safe Area** option on the contextual menu. Finally, let's pin the size of the image to the layout size. While the logo is selected, click on the pin tool button, check the width and height checkboxes and use the width/height provided (that is how you sized it after dragging and dropping the `UIImageView` into the scene). Click on the **Add 2 Constraints** button.

Third, select the email and password by holding the ⇧ key and clicking on each in turn. Once both are selected, click the stack view button in the lower right corner of Interface Builder. This introduced a new vertically oriented stack view containing the two selected views. In the attributes inspector, set the alignment and distribution properties both to fill, and set your spacing

to 8. Next, use the pin tool to pin the stack view to its nearest neighbors: leading and trailing edges to the containing view's margins, and top to the logo. This can be accomplished exactly like we positioned the welcome label above with the leading and trailing constants set to 20 and the top to **Standard** (e.g. just clear the field and and the Standard vertical spacing will be used).

Forth, ctrl-click on the sign in button, and drag downward and release anywhere within the containing superview below (e.g. when the superview turns light blue). On the context menu that displays select **Center Horizontally In Safe Area**. Setting the button's width to a third of the horizontal space in the containing superview is a little bit more subtle. Once again ctrl-click on the sign in button, and drag downward and release anywhere within the containing superview. On the context menu that displays, select **Aspect Ratio**. Making sure the button is still selected, you will see a blue bubble immediately under it with the numbers in the format DD:DDD. Clicking on that blue bubble will select the actual aspect ratio contraint that was added. Over in the attribute editor to the right change the second item from Superview.Height to Superview.Width. Set the multiplier to 1:3. Leave the remaining values the same.

You can repeat this procedure on the register button on the bottom of the screen to set its size to a third of the containing superview. Similarly, center it horizontally, and place it standard distance from the bottom edge of the screen.

Finally, pin the "Don't have an account?" label standard distance from the leading and trailing edges, and position its bottom edge standard distance from the register button's top edge. Don't forget to center the text in the label via the attributes inspector.

The Signup Screen Layout

Our strategy for layout of this screen is as follows:

1. Pin the signup label standard distance from the top of and center it in its containing view.

2. Pin the logo image standard distance vertically from the welcome label and center in containing view.

3. Introduce a vertically oriented `UIStackView` containing the email, password, and verify password text fields. Position the stack view standard distance vertically from the logo, and stretch it out horizontally so its standard distance from the leading and trailing edges of the containing view.

4. Pin the cancel button to the leading edge of its containing view, and standard distance under the stack view. Set the width of the button to one third of the available horizontal space in the superview.

5. Pin the signup button to the trailing edge of its containing view, and standard distance under the stack view. Set the width of the button to one third of the available horizontal space in the superview.

Applying what we've learned thus far, we encourage you to go ahead and attempt this layout on your own. You can check your work by examining the constraints in the sample code.

NOTE

At this point, you may have noticed Interface Builder reporting a couple of warnings about trailing and leading constraints missing which may cause overlapping view with other views. The potential problem arises from the fact that we did not set constraints for the trailing edge of the Cancel button or the leading edge of the Sign In button. Since you've sized the widths of these two buttons in proportion to the width of the view, this is simply not going to happen in this case! Nevertheless, the warning is annoying, so you can silence it by putting a horizontal constraint between the two buttons and setting it to be greater than or equal to 8.

The Main Screen Layout

Our strategy for layout of this screen is as follows:

1. Pin the label standard distance from the top, leading and trailing edges of the containing view.

2. Make sure the text is centered within the UILabel via the attributes inspector.

Applying what we've learned thus far, we encourage you to go ahead and attempt this layout on your own. You can check your work by examining the constraints in the sample code.

5.2.4 Styling in iOS

As was mentioned earlier in this chapter, support for styling and themes in iOS is rather primitive. Normally a couple of approaches are used together to provide the equivalent of themes: subclassing UIKit classes and setting display properties via the `UIAppearance` protocol.

The `UIAppearance` protocol gives us rather limited access to a subset of the attributes on UIKit controls. For example, for `UIButton` we can control the foreground color of the text, but we cannot control the width and color of the its border. Another limitation of `UIAppearance` is that once applied to a class, the settings apply to every instance of the class. At the moment, the only way to compensate for these shortcomings is to introduce subclasses of the user interface class of interest and reference the custom class in Interface Builder as needed.

`UiAppearance` does allow us to modify properties of controls based on the type of the container they appear in. This gives us some additional flexibility but we still don't have the flexibility for styling in iOS that we find in Android.

5.2.5 Adding Styling to the Traxy App

Let's revisit Traxy and take a look at how we can use a combination of subclassing and `UIAppearance` proxy settings to style our app. In particular, we would like to have our two login related view controllers adopt a blue background. We would like to have the navigation bars throughout the app to adopt that same blue color. We would like the text in the login view to have a light tan

foreground color, and the buttons to have that same color as background, with the blue as foreground. Furthermore, Since we are using a darker background, we also want to change the text on the status bar to a lighter color so there is adequate contract. Our intended styling is shown in Figure 5.11. Let's look at how we can accomplish this.

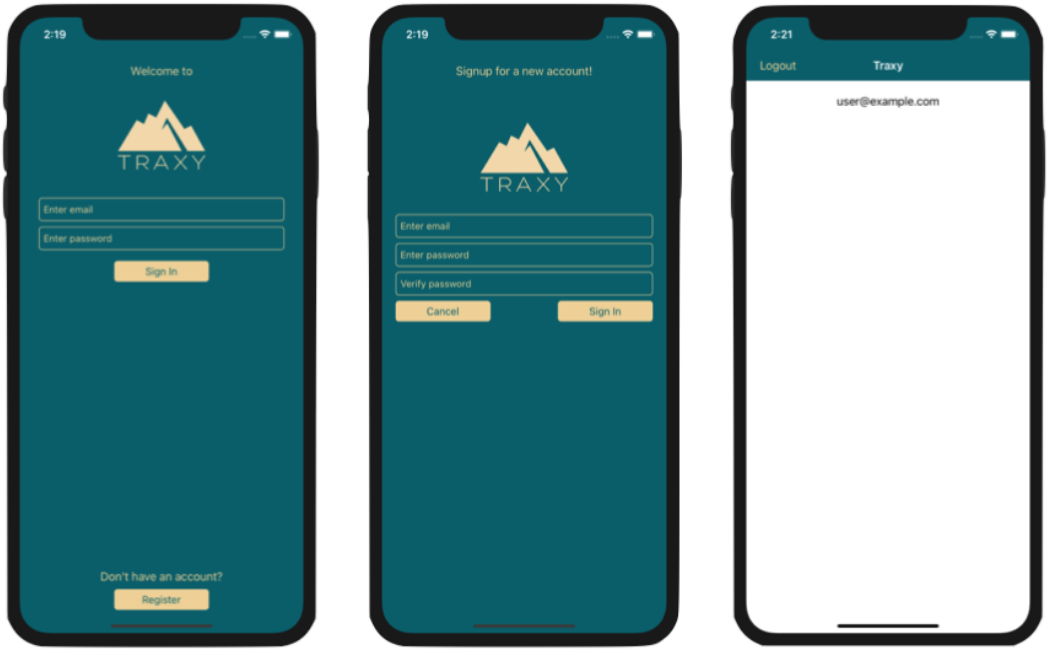


Figure 5.11: How we want our Traxy app to appear after styling.

We could informally refer to the set of colors we intend to use in the app as our "theme". While we don't have a first class mechanism to model this on iOS, we can establish some constants in Swift so that in the future we could change the colors used in our app with a simple update to this class.

In our `AppDelegate.swift` source file, we will capture our "theme" colors with some color constant definitions:

```swift
let THEME_COLOR1 = UIColor.init(red:0.000, green:0.157, blue:0.216,
alpha:1.00)  // DARK BLUE
let THEME_COLOR2 = UIColor.init(red:0.000, green:0.369, blue:0.420,
alpha:1.00) // Light Tan
let THEME_COLOR3 = UIColor.init(red: 0.937, green: 0.820, blue: 0.576,
alpha: 1.0)  // Lighter Blue
let THEME_COLOR4 = UIColor.init(red:0.576, green:0.596, blue:0.329,
alpha:1.00) // Greenish Tan
let THEME_COLOR5 = UIColor.init(red:0.251, green:0.286, blue:0.141,
alpha:1.00) // Army Green
```

First, we can accomplish part of our styling task using the `UIAppearance` protocol. In general, a proxy of this type is available for every UIKit user interface control. When we set properties on this proxy, it impacts every instance of that particular class. The appropriate place to put this code is in our `AppDelegate`'s `application:didFinishLaunchingWithOptions` method, since that will get called only once when our application launches. Here we've used `UIAppearance` to globally set the background and foreground colors of our navigation bars.

```swift
func application(_ application: UIApplication, didFinishLaunchingWithOptions
  launchOptions:  [UIApplicationLaunchOptionsKey: Any]?) -> Bool
{
  let navAppearance = UINavigationBar.appearance()
  navAppearance.barTintColor = THEME_COLOR2
  navAppearance.tintColor = THEME_COLOR3
  navAppearance.isTranslucent = false
  navAppearance.titleTextAttributes =
  [NSAttributedString.Key.foregroundColor : UIColor.white]
  return true
}
```

In order to change the application's status bar to light content, we need to open the Info.plist property file in Xcode and add the property "View controller-based status bar appearance" to "NO". If you click on the + key in the property view and start typing the above name, it will auto populate and let you set it to NO. Alternately, if you edit the Info.plist file directly as text you can insert the following XML key/value entries to accomplish the same:

```xml
<key>UIViewControllerBasedStatusBarAppearance</key>
<false/>
```

In addition to this setting, in order to use the light styled status bar, we also have to make sure any top level view controller (e.g. any view controller not contained by another) overrides the `preferredStatusBar` property of the `UIViewController` class. We will revisit this momentarily.

Unfortunately, at the moment the `UIAppearance` approach does not allow us to obtain the styling we would like for labels, buttons, and text fields on our login screens. There are a couple of issues with using `UIAppearance` for this. First, not all of the properties we need to affect the styling are actually capable of being set via `UIAppearance`. Second, we only want to affect the styling on the login and signup screens. Hence, we will accomplish the remainder of our styling by subclassing `UILabel`, `UIButton`, and `UITextField`:

`UIButton`.

Swift

```
class TraxyLoginLabel: UILabel {

    override func awakeFromNib() {
        self.textColor = THEME_COLOR3
      }
  }

class TraxyLoginButton: UIButton {

    override func awakeFromNib() {
        self.backgroundColor = THEME_COLOR3
        self.tintColor = THEME_COLOR2
        self.layer.borderWidth = 1.0
        self.layer.borderColor = THEME_COLOR3.cgColor
        self.layer.cornerRadius = 5.0
      }
  }

class TraxyLoginTextField: UITextField {

  override func awakeFromNib() {
    self.tintColor = THEME_COLOR3
    self.layer.borderWidth = 1.0
    self.layer.borderColor = THEME_COLOR3.cgColor
    self.layer.cornerRadius = 5.0
    self.textColor = THEME_COLOR3
    self.backgroundColor = UIColor.clear
    self.borderStyle = .roundedRect

    guard let ph = self.placeholder  else {
        return
      }

    self.attributedPlaceholder =
      NSAttributedString(string: ph,
        attributes: [NSAttributedString.Key.foregroundColor :THEME_COLOR3])
  }
}
```

Whenever one of these views is instantiated via the storyboard, iOS calls the `awakeFromNib` method which applies our styling. Once we've introduced the new subclasses, we need to go back to every label, button and text field in our login and signup views and type `TraxyLoginLabel`, `TraxyLoginButton`, and `TraxyLoginTextField`, respectively, via the identity inspector. This

is accomplished in exactly the same manner we set custom classes on our controllers in a previous chapter. Simply select the labels, buttons, and text fields on these two views in terms, and type appropriate class name in via the identity inspector. The keyboard shortcut to display the identity inspector is ⌘+⌥+4.

Finally, our last step is to set the background of our view controllers to the custom color shown above. Once again, this is not something we can accomplish with `UIAppearance`. Instead we will have to subclass the `UIViewController` class and make sure all of our controller classes inherit our custom subclass. We also need to make sure any top level view controllers override their `preferredStatusBar` property to achieve the light status bar styling. In our app so far, the top level view controllers consist of `LoginViewController`, `SignUpViewController` and the `UINavigationController` that contains the `MainViewController`. In the case of `UINavigationController`, because it will be the containing controller of all other view controllers once logged in, we must also have it specify the light status bar style as its preference.

Swift

```
class TraxyBaseViewController: UIViewController {
  override func viewDidLoad() {
    super.viewDidLoad()
    self.view.backgroundColor = THEME_COLOR2
  }

  override var preferredStatusBarStyle: UIStatusBarStyle {
    return .lightContent
  }
}

class TraxyNavigationController : UINavigationController {
  override open var preferredStatusBarStyle : UIStatusBarStyle {
    return .lightContent
  }
}
```

Then in each of the three view controllers we have so far, we need to change the base class from `UIViewController` to `TraxyLoginViewController`. In our storyboard, we need to select the navigation controller containing our main view controller, and in the identity inspector, change its class to be `TraxyNavigationController`.

In summary, iOS provides the developer with enough control knobs to do the equivalent of styles and themes, but as we will soon see, it is not quite as elegant as the provisions Android makes for us in this area.

5.3 Layout and Styling in Android

Now let's explore layout and styling in more depth on the Android platform. Go ahead and begin by examining the effect on switching device orientation between portrait and landscape in the current version of Traxy as you left it off the last chapter.[2]

[2]We will continue our development based on the implementation that uses Jetpack Navigation

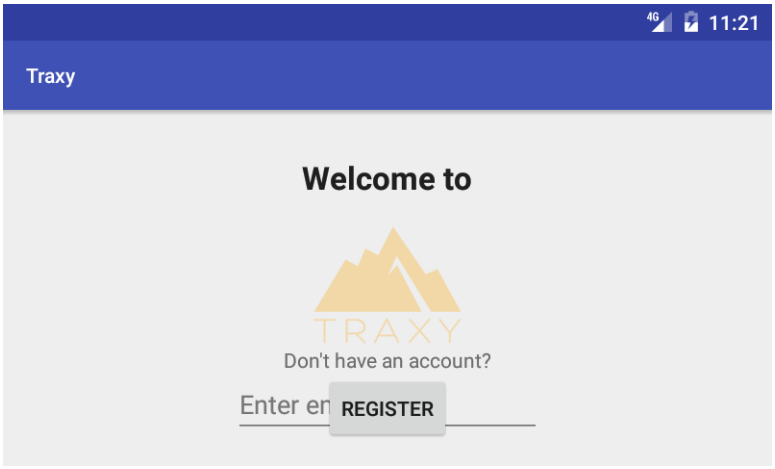


Figure 5.12: The widgets on the login screen overlap when the device is switched to landscape orientation.

Run the app in the emulator and rotate it to landscape orientation using the "device rotate" buttons to change the device orientation. Note that your emulator (or physical) device may have auto rotation disabled by default. If the screen does not rotate when you change to landscape orientation, on the device go to Settings≫Display and enable Auto Rotate. The screen layout in landscape orientation on a Nexus S emulator is shown in Figure 5.12. Recall that in our user interface design, the "Register" button and the "Don't have an account?" label are placed relative to the bottom of the screen, and they are correctly rendered as specified in your XML layout file, but they overlap other widgets. The total screen height in landscape orientation is too short to render all the widgets with no overlaps. For instance, the email input field is occluded by the "Register" button. Evidently, there are other issues.

You have two options to fix these layout issues:

1. Force the activity to allow only portrait orientation, by adding the attribute `android:screenOrientation="portrait"` to the `activity` tag in the manifest file.
2. Provide an alternate layout design when the screen is rendered in landscape orientation.

The easiest solution is, obviously, to implement the first option. However, there will be occasions when the user interface design(er) calls for both orientations. The following section guides you through the steps for providing an alternate layout file. At the same time, you will also learn how layout resources are managed by Android.

The first step is to create a new layout file for the landscape screen orientation. You may be tempted to create a copy of the current layout file and save it into your project. However, a better way to completing this task is to let Android Studio create the landscape layout variation for you.

But before you move further into using the layout editor, let's familiarize yourself with the layout toolbar. In Figure 5.13, the tools from left to right are

- Show Design + Blueprint Configuration
- Layout Variants and Orientation

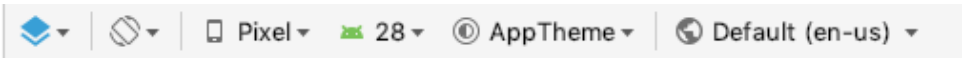


Figure 5.13: The Layout Editor Toolbar.

- Select Device: select a device for previewing the layout
- Select API Level: select an API level for previewing the layout
- Select theme for previewing the layout
- Select Language/Locale for previewing the layout

With `fragment_login.xml` opened in the layout editor, create a new layout variation using the command Orientation Preview » Create Landscape Variation, where **Orientation Preview** is the second button on the layout toolbar, as shown in Figure 5.14. This menu item will automatically disappear if your layout already has a landscape variation. As the result of this operation, Android Studio creates a new layout file under a subdirectory `res/layout-land` and the editor window associates this file with a new tab with a title `land/activity_login.xml`.

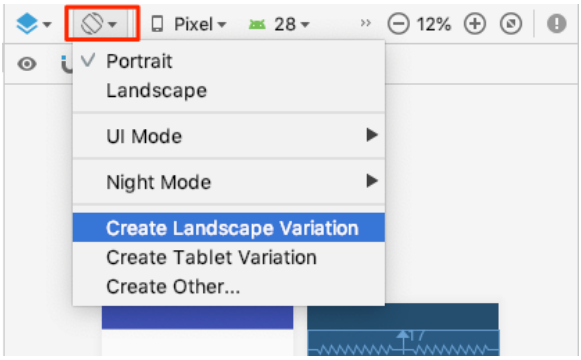


Figure 5.14: Layout Editor: Layout Variations

Let's stop for a moment and review what has just happened:

- The current layout file is `res/layout/fragment_login.xml`
- Its landscape variation is a new file with *the same name* (`fragment_login.xml`), but stored under a *different* directory `res/layout-land`. Recall that this is exactly the same technique used by Android for managing string resource variations for multiple language translations as described in Chapter 2.4.5. In that chapter you learned that the default string resources are stored in `res/values/strings.xml` and the corresponding Indonesian translations are stored in `res/values-in/strings.xml`.

> **NOTE**
>
> You will find a similar technique used in managing other resource files. Variants of the same resource file are stored using the same name but in different directories (and possibly predefined suffixes).

You might wonder why not store the portrait layout file under a subdirectory `res/layout-port`? Actually that is another option to solve the layout issue!

Investigating further, let's see what happens when the following method inside `onCreate()` or `onCreateVie()` is called at *runtime*:

Kotlin

```
// When using activity
setContentView (R.layout.fragment_login)
// When using fragment
inflater.inflate(R.layout.fragment_login, container, false)
```

When the current device orientation is portrait, the Android runtime will first attempt to load the above file from `res/layout-port/fragment_login.xml`, if one exists. Otherwise, it loads the file from `res/layout/fragment_login.xml`. Likewise, in landscape orientation the Android runtime will first attempt to load the layout file from `res/layout-land/fragment_login.xml`, *if one exists*. So, the *non-qualified directory* `res/layout` is the default to use when the specific directory is absent.

A similar resolution strategy is used for other resources. For instance, when the user reconfigures the device to the French Locale, the Android runtime will first attempt to load string resources from `res/values-fr/strings.xml`, if one exists. Otherwise, it will load the default `res/values/strings.xml`.

Now you understand how to organize your resource files if your app is designed to support multiple orientations, languages, or locales. Let's dive deeper into designing your UI using techniques available on Android.

5.3.1 Which Layout?

On Android, the user interface widget classes are a descendant of the `View` class, and all the container classes are a descendant of the `ViewGroup` class. A *container* class is a special kind of `View` that can contain other views, akin to Java Swing Layout Managers. For the rest of this document, both terms "view group" and "layout" are used interchangeably.

When Android Cupcake (API Level 3) was released in 2009, the SDK provided only the layout classes shown in Table 5.1. Out of the five classes shown in the table, we exclude `AbsoluteLayout` from our discussions because it has been deprecated. You will learn how to redesign your layout for landscape orientation using these classes initially, and then you will learn an alternative approach using more recently introduced layout classes.

Layout	Description
AbsoluteLayout	This layout was deprecated in API Level 3, should not be use any longer
FrameLayout	can hold overlapping subordinate widgets.
LinearLayout	arranges subordinate widgets either in a single row (horizontally) or a single column (vertically).
RelativeLayout	arranges subordinated widgets relative to their siblings or parent
TableLayout	arranges subordinate widgets in rows and columns, where widgets in each row are arranged using TableRow

Table 5.1: Layout classes supported by the initial Android SDK release.

NOTE

A common confusion for novice Android developers is understanding the difference between gravity and layout_gravity. These attributes are used through all ViewGroup classes. It is easier to explain using a 3-level example: assume a layout (X) contained inside its parent layout (P) and all child widgets of X.

- gravity attribute of X affects how the child widgets are placed within X.
- layout_gravity attribute of X affects how X itself is placed within its parent P.

FrameLayout

The FrameLayout arranges its subordinate widgets in overlapping layers. That is, assuming the screen is on the XY-plane, a FrameLayout stacks its child widgets along the Z-axis. Widgets are rendered in the order they are declared in the XML layout file, which means the first child declared is the farthest from the viewer and the last child declared is the closest. As a rule of thumb, smaller-sized widgets that are to appear on top of the others shall be declared towards the end of the file. The 2D position of each widget can be fine-tuned using its layout_gravity attribute specified on *individual child* widgets.

LinearLayout

Use LinearLayout when your UI design calls for *non-scrolling* widgets that must be displayed to line up either horizontally or vertically. In some ways, the Android LinearLayout is analogous to the UIStackView you learned in iOS. The layout orientation is determined by the orientation attribute (line 4) in Listing 29. One of the important LinearLayout features that should not be overlooked is its ability to distribute/assign *relative width/height* among its child wid-

gets. The distribution can be fine-tuned using the `android:layout_weight` attribute of each child:

- In a *vertical* `LinearLayout`, `layout_weight` controls the relative height of the child widgets
- In a *horizontal* `LinearLayout`, `layout_weight` controls the relative width of the child widgets

Lines 16 and 23 of Listing 29 set the relative heights of the two widgets to 1:4 and the resulting UI design in Figure 5.15 shows a top widget that fills 20% of the available height and a bottom widget that fills the remaining 80%. Notice that when `layout_weight` is set, we must set `layout_height`[3] to 0dp (to unset the attribute).

In case you didn't already realize it, the `layout_weight` attribute in Android is analogous to the iOS aspect ratio Auto Layout constraints you learned about earlier in this chapter.

XML

```
2  <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3    xmlns:tools="http://schemas.android.com/tools"
4    android:orientation="vertical"
5    android:layout_width="match_parent"
6    android:layout_height="match_parent"
7    android:id="@+id/top"
8    android:gravity="center_horizontal"
9    android:layout_margin="8dp">
10
11   <TextView tools:text="One"
12     android:layout_width="match_parent"
13     android:id="@+id/one"
14     android:background="@color/colorAccent"
15     android:gravity="center"
16     android:layout_weight="1" />
17   <TextView tools:text="Two"
18     android:layout_width="match_parent"
19     android:id="@+id/two"
20     android:background="@color/colorPrimary"
21     android:gravity="center"
22     android:textColor="@android:color/white"
23     android:layout_weight="4" />
```

Listing 29: Using `layout_weight` for setting relative heights.

Now let's revisit the different roles played by `gravity` and `layout_gravity` using a very simple `LinearLayout` shown in Listing 30.

- Figure 5.16(a) is rendered using the XML in Listing 30. Both child widgets (One and Two) are rendered using the default alignment/justification, which is justified left.
- By setting `gravity="center_horizontal"` on the top parent, Figure 5.16(b) renders both "One" and "Two" horizontally-centered.

[3] or `layout_width` if you are using a horizontal LinearLayout

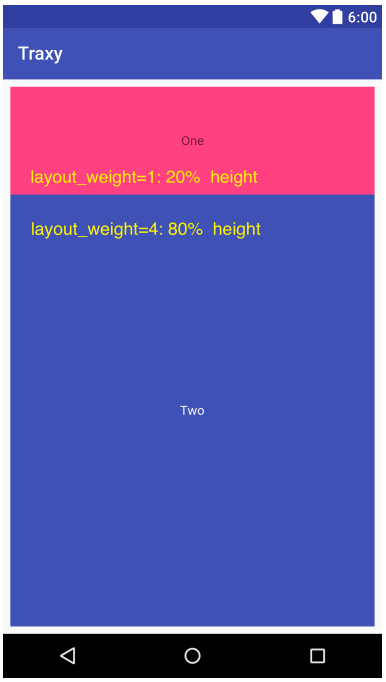


Figure 5.15: Two `TextView` widgets with 20/80 relative heights in a vertical `LinearLayout`.

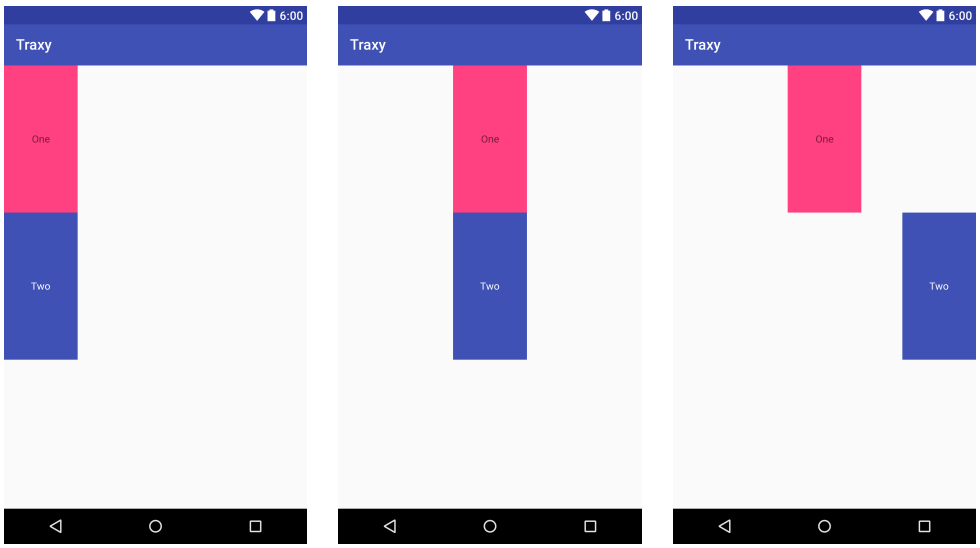
XML

```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
xmlns:tools="http://schemas.android.com/tools"
android:orientation="vertical"
android:layout_width="match_parent"
android:layout_height="match_parent"
android:id="@+id/top">

<TextView
tools:text="One"
android:layout_width="100dp"
android:layout_height="200dp"
android:id="@+id/one"
android:background="@color/colorAccent"/>
<TextView tools:text="Two"
android:layout_width="100dp"
android:layout_height="200dp"
android:id="@+id/two"
android:background="@color/colorPrimary"
android:gravity="center"
android:textColor="@android:color/white"/>
</LinearLayout>
```

Listing 30: The XML layout for the LinearLayout in Figure 5.16.

- Figure 5.16(c) is rendered using the same XML as (b) but also with an additional attribute `layout_gravity="end"` set on "Two". The "centerHorizontal" flag is applied to "One", but "Two" has an additional control of its alignment.



(a) (b) (c)

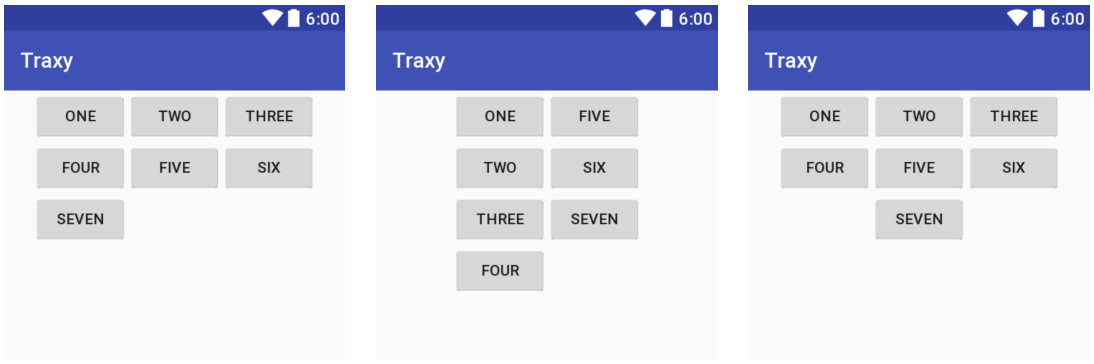
Figure 5.16: The effect of `gravity` and `layout_gravity`.

GridLayout

Before RecyclerView and its GridLayoutManager were available to Android developers, this layout was the preferred choice for arranging visual elements in a 2D grid. Now, this layout is categorized as a legacy layout.

A `LinearLayout` places its children along one axis (vertical/horizontal), and it does so with no regards to space limit. A `GridLayout` also places its children along its axis of orientation, but it shifts along the *orthogonal* axis when the number of children on that axis reaches a specified maximum count. A horizontal `GridLayout` honors the `columnCount` attribute (and *ignores* the `rowCount` attribute). For instance, a horizontal `GridLayout` with column count 3 will begin placing its children on the first row, but it shifts to the next row when 3 children have been placed. Adding 8 children to this layout, the first three go to the first row, the next three to the second row, and the last two in the the third row. This widget placement policy is order-dependent, i.e. the order of a child in the layout determines its location on the screen. Figure 5.17 shows the effect of orientation on the same group of buttons. In Figure 5.17a the grid has a horizontal orientation with column count 3, in (b) it is a vertical grid with row count 4.

If you prefer to avoid the order-dependent placement policy above, you can place a widget by using an explicit zero-based "coordinates". The top-left cell is the grid is at row 0 and column 0. Figure 5.17c places button "Seven" using absolute position (`layout_row` 2, `layout_column` 1)



(a) Horizontal Orientation (b) Vertical Orientation (c) Absolute Position

Figure 5.17: `GridLayout` Orientation & Position examples.

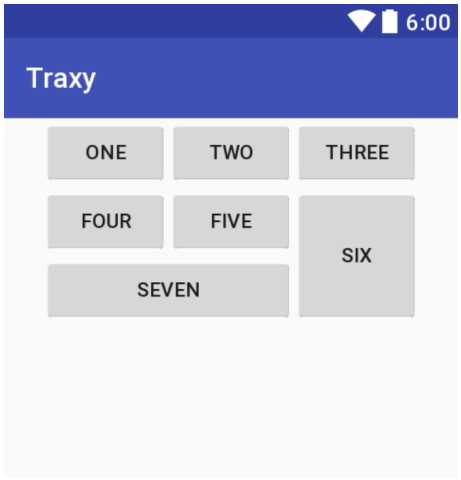


Figure 5.18: An example the `GridLayout`'s span and fill features.

By default, each widget occupies one cell in a `GridLayout`. This default setting can be overridden using the "span" attribute, and most of the time you want to use this attribute in conjunction with the "fill" attribute. For instance, in Figure 5.18:

- Button "Six" is placed with `layout_rowSpan=2`, `layout_gravity=fill_vertical`.
- Button "Seven" is placed with `layout_colSpan=2`, `layout_gravity=fill_horizontal`.

RelativeLayout

We cover this layout for a historical reason. This layout has been deprecated and replaced by a newer and more powerful `ConstraintLayout`. It allows widgets to be placed relative to their siblings or parent using attributes under the following categories:[4]

- Relative positioning to a sibling: `below`, `above`, `toLeftOf`, and `toRightOf`
- Relative alignment to a sibling: `alignLeft`, `alignRight`, `alignTop`, and `alignBottom`,
- Relative position to the containing parent: `alignParent(Left|Right|Top|Bottom)`, `centerVertical`, `centerHorizontal`. Because there is only *one* parent, there is no need to include a reference to the parent container. Hence, these attributes take a boolean value true/false

You don't have to memorize all these attributes, the layout editor picks the right ones as you edit the layout design. The list is provided so you will be able to sensibly browse the XML of a layout describing widgets within a `RelativeLayout`.

Disambiguation of Left/Right in `RelativeLayout`

For most languages, text flows from left to right (LTR). But for some languages like: Arabic, Hebrew, Urdu, text flows from right to left (RTL). For LTR languages, a relative position "after" implies placing a widget to the right of another (higher x-coordinate). However, for RTL languages, the relative position "after" means "left" (lower x-coordinate). In order to use a common expression that applies to both language classes, SDK API level 17 introduces "start" and "end" (in place of "left" and "right") for expressing *horizontal* relationship among widgets. Using these newer terms, for instance, `layout_toLeftOf` and `layout_toRightOf` are replaced with `layout_toStartOf` and `layout_toEndOf` (depending on the current horizontal context).

CoordinatorLayout

The main use of `CoordinatorLayout` is for managing interactions among the child views the layout contains; visual changes triggered by one of the child widgets may trigger its siblings to update their visual appearance. For instance, on a screen that includes a `FloatingActionButton` at the bottom of the screen, a `SnackBar` message may cover the `FloatingActionButton`. To avoid this visual defect, the button should move up, giving enough room for the `SnackBar` to display its message. After the SnackBar disappears, the button should move back to its previous

[4]The `layout_` prefix is intentionally omitted for brevity, when you read `above` in the following explanation, the complete attribute is actually `layout_above`.

position. These interactions are possible when the `FloatingActionButton` and the `SnackBar` are both children of a `CoordinatorLayout`.

The design pattern that best describes the above interactions is the Observer pattern. Using the above example, the button behaves like an observer that will get notified when the "snackbar-appeared" event occurs. In response to the event, the button performs some simple task. In `CoordinatorLayout` parlance, we say that the button demonstrates a *specific behavior*, (in response to the layout changes). If you are familiar with Java Swing event listeners, it should be easy to understand this concept. For instance, Swing `MouseListener` enables your program to intercept mouse events like: mouse pressed, mouse released, mouse entered, and mouse exited. Likewise, `CoordinatorLayout` enables your Android program to intercept view related events such as: view changed, resized, tapped, measured, scrolled, flinged, etc.

- The interaction between a FAB and a SnackBar as described above depends on the "view changed" events. When the `SnackBar` appears, the FAB shifts up by the height of the `SnackBar`.
- Another use of `CoordinatorLayout` is demonstrated by a screen that includes both an AppBar and a scroll list (`RecyclerView`). The bar initially shows a tall header that shrinks as the list scrolls up or grows taller when the list scrolls down. In this particular case, the AppBar listens for scroll events and adjusts its height accordingly.

In Section 6.3.5 you will implement a custom behavior to control the visibility of a FAB when a sibling `RecyclerView` is being scrolled.

5.3.2 Login Screen in `LinearLayout`

Given the basics of `FrameLayout`, `LinearLayout`, and `RelativeLayout`, you will now redesign the landscape variation of the Login screen. The completed design is shown in Figure 5.19. *Warning: the first redesign attempt yields a less efficient solution due to hierarchical nesting of layouts. In the next section you will redesign it using a more efficient layout: `ConstraintLayout`.*

The overall design strategy is as follows. Using horizontal `LinearLayout` we divide the screen vertically into two sub panels of equal width (left and right) and within each half we use vertical `LinearLayout` to manage the widgets in it. The left half holds the "Welcome to" text, the Traxy logo, the "Don't have account" text, and the "Register" button. With the "Register" button anchored to the bottom The email, password, and "Sign In" button are placed horizontally centered within the right half.

Follow these steps to accomplish the desired screen design:

- Switch the layout editor to *Text* mode and replace the top-layout to `LinearLayout`. Be sure to set its `android:orientation` attribute to "horizontal"
- Drag the another LinearLayout into the Component Tree panel. Set its ID to `left_half`, `layout_weight` to 1, `layout_width` to "0dp", and `gravity` to center
- Drag the second LinearLayout into the Component Tree panel. Set its ID to `right_half`, `layout_weight` to 1, `layout_width` to "0dp", and `gravity` to center
- Inside the "Component Tree" panel, move the "Welcome" text label, logo, "Don't have an account" text label and "Sign Up" button into `left_half`. Be sure these widgets appear in that order inside the Component Tree.

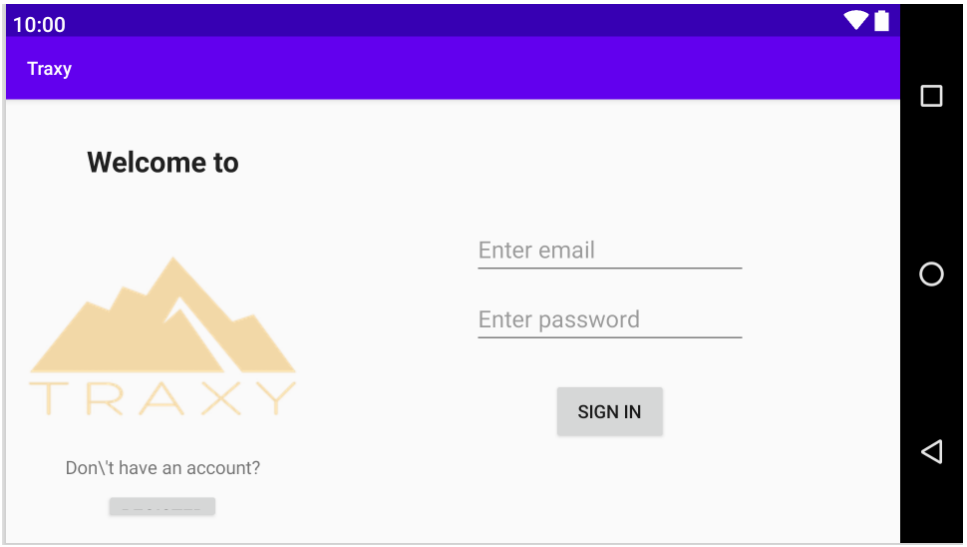


Figure 5.19: The desired landscape screen design.

- Now move the two input fields and "Sign In" button into `right_half`
- In the portrait variant, the email field is placed *relative to* the logo. This is no longer true in the landscape variant and you must remove the contending attribute

Run or debug the app and verify that in landscape orientation the screen shows the new layout and more importantly you are able to login or sign up when the device is in either orientation *without changing your Java program*.

Congratulations! You have mastered the basics of using `LinearLayout` and how to use `layout_weight` to distribute available space to its child widgets. You also learned how to nest layouts inside each other.

5.3.3 Login Screen in `ConstraintLayout`

The approach you took with a LinearLayout and hierarchical nesting of widgets increases the rendering cost exponentially. Let's try to understand this better.

A hierarchy of view groups and widgets are commonly organized into a tree structure. To simplify the explanation, imagine we have a tree of boxes. Each box has its origin, width, and height, and may contain smaller boxes, *ad infinitum*. To render the entire tree of boxes, the rendering algorithm uses a top-down traversal to collect and calculate the correct information to render all the boxes. Beginning from the topmost box at the root of the widget tree, the algorithm has to perform the following three phases in the order given:

1. *Measure phase*: each box sends a query to all its children to determine how much space each child box would take to render itself. The response from each box then propagates bottom-up.

2. *Layout phase*: after aggregating the information from all its children, each box determines how much space can *actually* be assigned to each child box and use that information to position each child box (fix its origin and set its width/height)

3. *Render phase*: each box is then rendered using the position information set in the previous step

The sketch of the above algorithm indicates that each widget is traversed *twice* before it is rendered. Now if the widget itself is a view group, then the view group must also perform the measure phase and layout phase to all its children. Hence the children at this second layer are traversed for a total of *four times*. Likewise, all the child widgets at the third layer are traverses *eight times*. Hence, the exponential cost for rendering nested widgets.

To improve rendering performance, it is best to avoid nested view groups in our user interface design. `ConstraintLayout` is a view group that does indeed deliver better performance. It was introduced at Google I/O 2016 and available since Android Studio 2.2. The layout is designed to be backward compatible up to API Level 9 (Gingerbread). You will soon see that `ConstraintLayout` is conceptually similar to the iOS *Auto Layout* techniques you explored earlier in this chapter.

`ConstraintLayout` improves the capabilities or RelativeLayout further. It places its subordinate widgets using a more expressive language, while organizing them into a *flat structure*. This flatness avoids the exponential rendering cost. In `RelativeLayout` widgets are placed relative to each other using an "axis-aligned" model (right-of, left-of, above, below). `ConstraintLayout` extends this concept into *"axis-free" connections*. A connection can behave either like a "spring" (flexible length) or a "strut" (fixed length). By connecting one widget to another (or to the border of its containing parent), a positional constraint is created between the two.

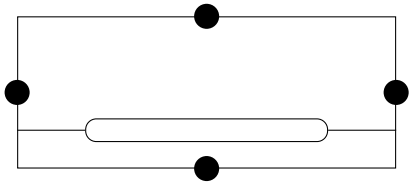


Figure 5.20: Box anchor points and text baseline.

If you are curious enough to inspect the XML file in *Text* mode, you can easily identify `ConstraintLayout` specific attributes by looking for the prefix `layout_constraint`.

To assist developers in editing layout files, Android Studio provides a WYSWIG editor that works well with `ConstraintLayout`. When a widget is selected, its four anchor points will appear. When you right click on a text widget, the popup menu will show an option "Show Baseline" and the baseline marker will also appear as shown in Figure 5.20. These anchors can be used for creating a new connection/constraint by dragging from an anchor in one widget to another.

To create a text baseline constraint you have to drag from the baseline anchor of one text widget to another text widget. In Figure 5.20 the actual text baseline is the *thin line*. The rounded shape on the baseline is an area used by the layout editor to detect mouse drags when a connection is being made. It is important to understand that a connection constraint originates *from an anchor point* and terminates at an *edge* or from one baseline to another. By understanding this principle, the process adding new constraints is very intuitive. It is much easier to understand these concepts by trying them out on the editor canvas rather than reading step-by-step instructions.

In Chapter 2.4 you learned about the layout attributes `layout_width` and `layout_height`. They take three possible values: `wrap_content`, `match_parent`, or a dimension. The symbolic

dimension `match_parent` does not make sense in `ConstraintLayout` and has been replaced with "any size" (0dp) when used within a `ConstraintLayout`.

Horizontal/Vertical Guidelines

When designing the landscape variation of the login screen, you noticed the practical use of `layout_weight` in `LinearLayout` for distributing the remaining width across several widgets. However, this technique induces widget nesting, which we saw may entail additional performance costs.

To provide a similar *size distribution* feature, `ConstraintLayout` introduces guidelines. These are *invisible edges* included in the view for attaching the end point of constraint connections. Guidelines can be placed using either *absolute position* (such as 200dp) or *relative positioning* (such as 20% from left of the screen.) For instance, to have two subareas of equal width like in the first redesign of the login screen, you can add a vertical guideline placed at 50% width. You will explore this topic in more details in Section **??**.

Chains

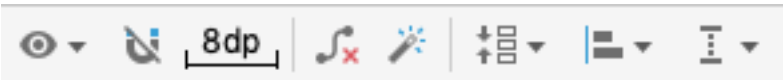
Occasionally you want to have a number of widgets treated as one group and apply placement constraints to the group, instead of individual widgets. For example, if you need to center multiple widgets horizontally, select them and use the popup menu (right click) to select Chains > Create Horizontal Chain followed by Chains > Horizontal Chain Style.

There are several different chain styles supported by Android.

- The *spread* style (the default style) spreads the widgets out evenly in the available space.
- The *spread_inside* style has the head and the tail of the chain affixed to the constraints on the respective ends of the chain, and all remaining widgets spread evenly in the remaining space.
- The *packed* chain style packs the widgets together within the available space.

5.3.4 Laying out the Login Screen

Let's apply this new knowledge you've acquired on `ConstraintLayout` and attempt a second redesign of the login screen. While editing a layout file with `ConstraintLayout` the editor will show the following icons in its toolbar:



These tools from left to right are:

1. Hide/Show constraint (self-explanatory).
2. Auto Connect: when this flag is enabled the editor automatically creates new connections when a widget is placed into the canvas.

3. Default Margin (self-explanatory).

4. Clear Constraints: select a widget and use this tool to remove all connections from the selected widget.

5. Infer Constraints: pressing this button instructs the layout editor to automatically insert constraints and connections. This tool is usually used *after* you place a few widgets on the canvas to their approximate position with Auto Connect disabled, then press this tool to let the layout editor create all the required connections automatically for you.

6. Pack: pack or distribute a group of selected widgets. This command does not create new layout constraints/connections, it affects only the widgets X,Y coordinates on the *design canvas* (not their on-screen appearance at runtime). This button only appears if an item is selected in the layout.

7. Align: change alignment of a group of widgets. Similar to the "Pack" button, this command has no effect on the runtime appearance. This button only appears if an items is selected in the layout.

8. Guidelines: insert a vertical or horizontal guideline.

All the constraints for designing the landscape variant of the login screen is shown in Figure 5.21.

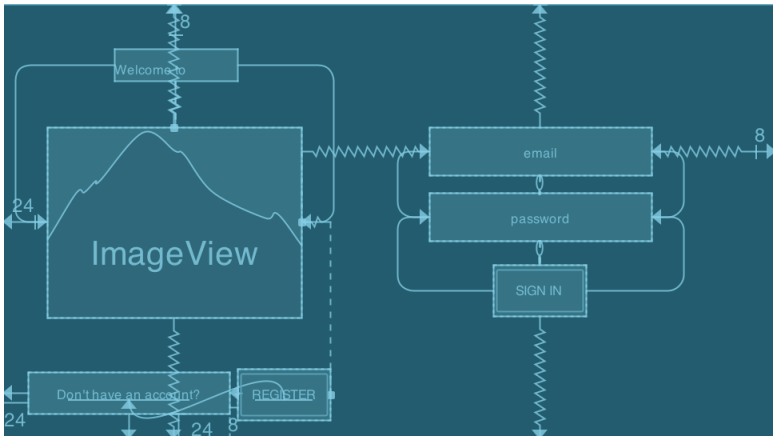


Figure 5.21: Layout Blueprint (landscape)

The design approach of the above screen layout is as follows:

- For aesthetic we place the Traxy logo vertically centered. This requires one constraint from its top anchor to the parent top edge and another constraint from its bottom anchor to the parent bottom edge

- The "Welcome" text is placed vertically in the available space above the logo (two vertical constraints required)

- The baselines of text "Don't have account?' and "Register" are also constrained so they line up nicely. They are both constrained from the bottom edge of the parent, and the text is also constrained from the left edge of the parent (four constraints required)
- To keep a nice balance, the logo is also horizontally centered within the span of the text view and the "Register" button. This requires one constraint from between the left anchors of the logo and the text as well as another constraint between the right anchors of the logo and the button
- The input fields email, password, and the "SignIn" button are chained and packed and centered vertically within the available vertical space. Within the chain we also apply vertical spacing among the three widgets
- The email input field is horizontally centered within the available space between the logo and the right edge of the parent (two constraints required)
- Finally, the password field and sign-in button are horizontally centered among with respect to the email field (four constraints required)

1. Reopen the landscape login screen `land/fragment_login.xml`
2. In case your layout is not yet using ConstraintLayout, select the topmost node in the Component Tree and select Convert View » ConstraintLayout to convert from the current layout to ConstraintLayout.
3. Disable "auto connect" so it won't interfere with your manual rearrangement of the widgets in the next step.
4. Promote all the widgets under the two `LinearLayout` from the previous experiment into the top level by dragging them in the Component Tree. Expect to see all the widgets rendered out of place. If you also encounter "Rendering Problems" error, just close the error dialog and continue editing.
5. If necessary, switch to Text mode and remove all `RelativeLayout` specific attributes, which are technically those that begin with `android:layout_` (except for width and height). Be sure to retain all attributes of the tools namespace (that begin with `tools:`).
6. Remove the two empty `LinearLayouts` as you no longer need them.
7. With AutoConnect remains disabled, move each widget to its approximate final position in the canvas.

Now let's execute our plan of actions. In general, it is much easier to carry out the following operations on the blueprint, because all edges are always visible in the blueprint mode:

1. Start by adding the two vertical constraints for the logo by dragging from the logo top anchor to the top edge of the screen (parent) and also from thee bottom anchor to the bottom edge of the screen. Be sure to set both the top and bottom margin to zero as shown in Figure 5.22.
2. Next add the two vertical constraints from the anchors of the "Welcome To" text and be sure to set the margins to zero

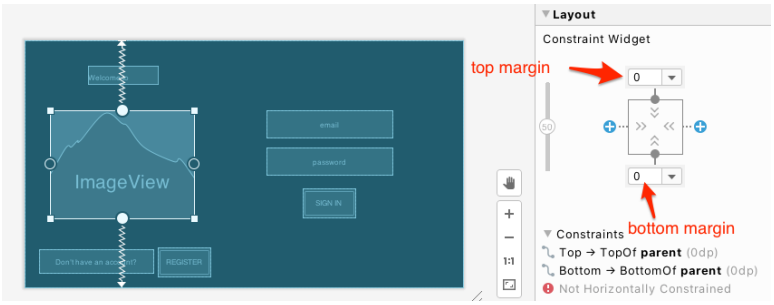


Figure 5.22: Vertical Margins

3. Add one horizontal connection so the the "Don't have account?" text is constrained to the left edge of the parent. Likewise, add one vertical connection to the alignBottom edge of the parent
4. Right click on the Register button to show its baseline, then add a baseline constraint it the "Don't have account?" text. Also add a horizontal constraint from its left anchor to the right edge of the text
5. Add two horizontal constraints to center the logo with respect to the "account" text and "Register" button
6. Add two constraints to center the "Welcome" text with respect to the logo
7. Add two horizontal constraints to place the Email centered in the available space to the right of the logo
8. Add two horizontal constraints to the Password field so it is centered with respect to the Email field.
9. Repeat the same step to the "Sign In" button
10. Select Email, Password, and "Sign In" and then create a vertical chain. Choose a "packed" chain style.
11. Finally, add vertical spacing between the Password to Email, and between the SignIn and Password

If for some reasons, you missed one or more constraints, a red "balloon" icon at the upper-right corner of the layout editor will appear and clicking the icon will display a more detailed explanation of what is missing.

The vertical and horizontal lines inside the square indicates the size constraints. Click on these lines to cycle through the following three symbolic size constraints (Refer to Figure 5.23):

- A straight line: the widget has a fixed width/height
- Chevrons: the widget size is wrap content
- A zig-zag line (spring like): the widget is stretchable to match its constraints

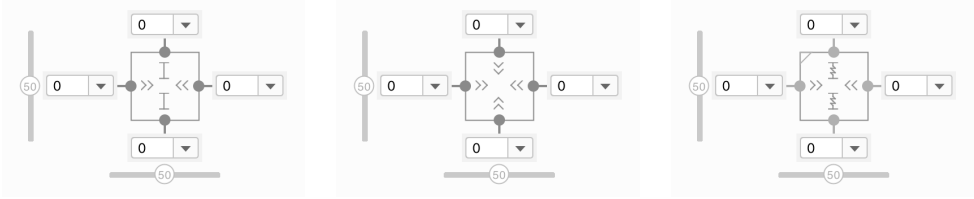


Figure 5.23: Widget size constraints

If, by any chance, you made a mistake along the way. You can always remove an existing connection by clicking one of the four dots in Figure 5.23.

Biases

When a widget is constrained on both opposite anchor points, `ConstraintLayout` allows fine-tuning of the widget alignment using a bias value in the range 0.0–1.0. The vertical and horizontal sliders in Figure 5.23 control the bias value.

- The default bias value 0.5 places the widget at its exact horizontal/vertical center.
- A value higher then 0.5 will pull it to the right, and at the extreme value of 1.0, the text view right edge is aligned with the logo right edge.
- A value lower then 0.5 will pull it to the left

5.3.5 Applying Model-View-ViewModel

Now that you have acquired more knowledge in UI design and learned how to design both the portrait and landscape variations of the login screen, you can apply that new knowledge to other UI screens in your app. In addition to designing the UI, you should also understand the best practices for handling the data to feed the UI, especially using the MVVM architecture as described in Section 3.5.

As a starter, we will keep the user email entered at login as the only data to store in our view model class. Let's create a new class `UserDataViewModel.kt`:

Kotlin

```
class UserDataViewModel : ViewModel() {
  private var _usr = MutableLiveData<String>()

  val userId
      get() = _usr
}
```

For now, the only property defined in our viewmodel class `userId` with a backing store `_usr` declared as a mutable live data. With this property defined, the three fragments now have access to a common data for its UI. For instance when the user successfully logged in, `LoginFragment` can store the user email into the mutable live data and later the other two fragments can observe any changes made to the property. To access the mutable live data in the view model, each of the fragment must first declare a reference to the view model instance:

Kotlin
```
lateinit var viewModel: UserDataViewMode
```

and initialize it in `onActivityCreated`:

Kotlin
```
override fun onActivityCreated(savedInstanceState: Bundle?) {
  super.onActivityCreated(savedInstanceState)
  viewModel = ViewModelProviders.of(requireActivity()).get(UserDataViewModel::class.java)
}
```

To update the view model variable using data entered by the user from the UI, the following snippet can be added, for instance, in a click listener of a button.

Kotlin
```
viewModel.userId.value = "Some text"
```

Specifically, in our case, the click listener of the "Login" button (in LoginFragment) should store the email address into the view model prior to transitioning to MainFragment:

Kotlin
```
signin.setOnClickListener { v ->
  val emailStr = email.text.toString()
  val passStr = password.text.toString()

  // After verifying the email and password
  viewModel.userId.value = emailStr
  findNavController().navigate(R.id.action_login2main)
}
```

To retrieve data from the view model and display it on the UI, a fragment can use the observer pattern. For instance, LoginFragment can use the following snippet to update a TextView (calleed `userEmail`) using the `userId` property in the view model.

Kotlin
```
// In onActivityCreated of LoginFragment
viewModel.userId.observe(this.viewLifecycleOwner, Observer { z ->
  userEmail.text = z
})
```

Go ahead and use these techniques in the three fragments you have so far.

5.3.6 Styling/Theme in Android

Android separates all the visual elements in a user interface from the application logic by defining these visual elements in XML resource files. In the previous sections, you learned various tools for placing widget elements on screen and used an XML layout file for specifying the widget placement details.

> **NOTE**
>
> Ideally, an XML layout file should describe only the *spatial relationships* among the widgets. Visual details, like colors, font size, text color, button background, etc. should *not* be specified in the layout file, but in style files.

A *style* usually has a narrower scope, and is defined as a set of visual properties applied to a particular widget. Therefore, it is technically correct to say: button *style*, checkbox *style*, etc. On the contrary, a *theme* usually has a broader scope and is applied to a larger group of widgets, to an Activity, or even to the entire app.

- To apply a specific style to a widget, set its "style" attribute to the style of your choice. In the layout editor "Properties" panel, you can select from a number of predefined styles

- To apply a specific theme to an activity/app, you have to edit the manifest file and add the `android:theme` attribute on the selected `<activity>` or `<application>`.

XML
```xml
<application android:name="MyActivity"
android:theme="@style/AppTheme"
...
/>
```

Even though styles are conceptually different from themes, they are syntactically equivalent in the Android resource definition system. Both styles and themes are defined in XML using the `<style>` tag.

Various types of visual resources commonly-used in user interface designs include: gradient images, shapes, color definitions, dimensions, animations, etc. The XML syntax allows developers to declare these resources using key/value pairs. Listing 5.3.6 shows an example of three key-value pairs for declaring two color resources and one dimension. The first color resource is associated with key `my_button_background` and value *#FF9800*.

```xml
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <color name="my_button_background">#FF9800</color>
  <color name="primary_dark">#F57C00</color>
  <dimen name="vertical_margin">16dp</dimen>
</resources>
```

Listing 31: Android Resource Declarations in XML

A UI designer can create a style/theme that references these resources. However, defining styles/themes from scratch can be a very tedious task. Instead of defining a *new style/theme* from scratch, the common practice is to *override* an existing (predefined) style/theme using Android style inheritance mechanism. To use this approach, we declare a new style and use the `parent` attribute to specify the parent style such as shown in Listing 32. In the example, `ExampleTheme` overrides the style definitions of `Theme.AppCompat.Light` and `TraxyText` is a variation of Android default `TextView` style.

```
<style name="ExampleTheme" parent="Theme.AppCompat.Light">
  <item name="colorPrimaryDark">@color/primary_dark</item>
  <item name="buttonStyle">@style/MyShinyButton</item>
</style>

<style name="AnotherTheme" parent="ExampleTheme">
  <item name="background">@color/my_button_background</item>
</style>

<style name="TraxyText" parent="@android:style/Widget.TextView">
  <item
    name="android:textAppearance">@android:style/TextAppearance.Medium</item>
  <item name="android:background">@android:color/white</item>
<style>
```

Listing 32: Declaration of `ExampleTheme` using style inheritance

By using `TraxyText` with any `TextView` declaration, all the text views in your app will be rendered in medium text size and white background.

XML
```
<TextView style="@style/TraxyText" .... />
```

Another syntax for inheriting a parent style is to use the dot notation. Lines 6–8 in Listing 32 can be also written using the following notation:

XML
```
<style name="ExampleTheme.AnotherTheme">
<!-- define your properties here -->
</style>
```

Hence the built-in style Theme.AppCompat.Light is a "Light" style inherited from "AppCompat" inherited from "Theme".

When inheriting a parent style, the key name of resources redefined in our own style must match the corresponding keyname in the parent style. Take care in making sure the key name's are not misspelled. If they are, the resource redefinition has no effect and unfortunately Android Studio will silently ignore it.

In Chapter 2.4 you learned the importance of inheriting your Activity classes from `AppCompatActivity`. In conjunction with this class, you also should use themes derived from the AppCompat theme. One of the major benefits of using this theme is its compliance with the Material Design.

The AppCompat theme provides several predefined styles used by various elements:

- Window attributes (background, action bar, frame, . . .)
- Color attributes
- Default widget styles (button, floating action, list, dialog, radio button, action bar), text appearances

In addition to styles, it also provides predefined themes, such as: Theme.NoTitleBar, Theme.Light, Theme.NoTitleBar.Fullscreen, etc. The complete list can be found in the property editor panel of Android Studio.

There are details on using other resources for styling/theme that we will not address at this time. Here, we just present the necessary basics of using styles and themes. The most important question concerns how you apply a specific theme or a customization of it to your app. The next section describes how to apply Material Theme to your entire app.

5.3.7 Introduction to Material Theme

The Material Theme provides a number of options:

- `Theme.Material`
- `Theme.Material.Light`
- `Theme.Material.DarkActionBar`

Unfortunately, these themes were introduced in Android 5.0 API Level 21 (Lollipop). But there is support for developers who target devices running lower API levels; the Material Design *style* can be applied without using the above *themes*. The AppCompat library provides the following themes that implement the Material Design Styles:

- `Theme.AppCompat`
- `Theme.AppCompat.Light`
- `Theme.AppCompat.DarkActionBar`
- `Theme.AppCompat.NoActionBar`
- *and many more*

By default, your project comes with the following style file (`res/values/styles.xml`):

XML

```
1 <resources>
2 <!-- Base application theme. -->
3 <style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">
4 <!-- Customize your theme here. -->
5 <item name="colorPrimary">@color/colorPrimary</item>
6 <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
7 <item name="colorAccent">@color/colorAccent</item>
8 </style>
9 </resources>
```

Line 3 tells us that `AppTheme` is a customization of a parent theme `DarkActionBar` defined as a light theme variation provided by the AppCompat library.

Inside the app manifest file, you will find the following declaration that refers to the theme above.

XML

```
1 <application
2 android:label="@string/app_name"
3 android:theme="@style/AppTheme"
4 ...
5 >
```

In Android Material Design, the background of the app bar is rendered in `colorPrimary`. `colorPrimaryDark` is the background of the status bar, and UI controls are tinted using `colorAccent`. The corresponding color definitions of these symbolic names are found in another resource file `res/values/colors.xml`. To set a different background color on the Login and Sign Up screens, we add a new color definition `windowBackground`.

XML

```xml
<?xml version="1.0" encoding="utf-8"?>
<!-- in colors.xml -->
<resources>
<color name="colorPrimary">#3F51B5</color>
<color name="colorPrimaryDark">#303F9F</color>
<color name="colorAccent">#FF4081</color>
<color name="windowBackground">#c39797</color>
</resources>
```

The color resource for window background can be used in `AppTheme` by declaring a new item `android:windowBackground` as shown in Listing 33.

XML

```xml
<resources>
<!-- Base application theme. -->
<style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">
<!-- Customize your theme here. -->
<item name="colorPrimary">@color/colorPrimary</item>
<item name="colorPrimaryDark">@color/colorPrimaryDark</item>
<item name="colorAccent">@color/colorAccent</item>
<item name="android:windowBackground">@color/windowBackground</item>
</style>
</resources>
```

Listing 33: Adding a new window background setting

Traxy Login and Signup screens are rendered using a darker background, but the rest of the screens are rendered using a gray background. To have custom style only for these two screens, you can define a sub-style of `AppTheme` and change its window background to your preferred color. In the following example we use `@color/colorPrimary`:

XML

```xml
<!-- in styles.xml -->
<style name="FrontPage" parent="AppTheme">
<item name="android:windowBackground">@color/colorPrimary</item>
</style>
```

The above theme can then be applied to `LoginActivity` by specifying the theme attribute of the corresponding `<activity>` in the manifest:

XML

```xml
<activity android:name=".LoginActivity"
android:theme="@style/FrontPage">
</activity>
```

Several screenshots of the LoginActivity using different color settings are shown in Figure 5.2. The user-interface design in the screenshots applies the “Borderless” style to the “Register” button. It is now your turn to customize the entire app using your preferred colors.

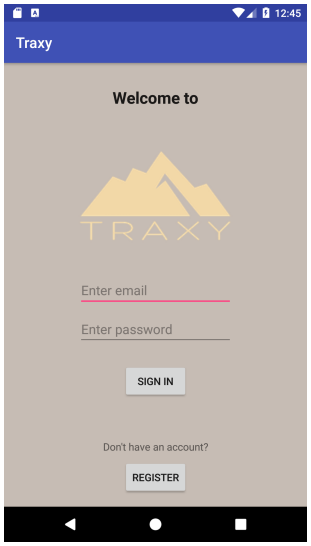
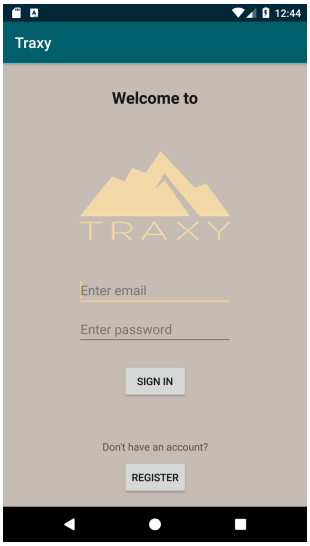
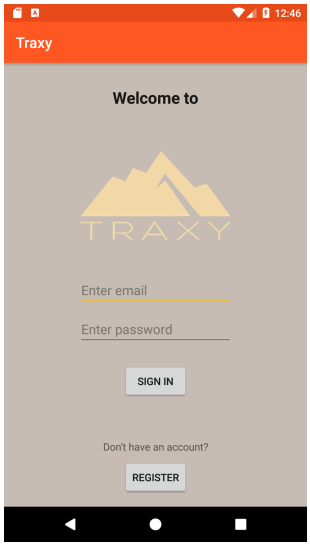
		
primary:#3F51B5 primaryDark:#303F9F accent:#FF4081	primary:#005F6B primaryDark:#002838 accent:#EFD093	primary:#FF5722 primaryDark:#E64A19 accent:#FFC107

Table 5.2: Login Screen in different color themes

—If you want a machine to be able to interact with people, it better not do things that are surprising to people.

Rodney Brooks

6

Displaying and Interacting With Collections of Data

Chapter Objectives

After reading this chapter you should be able to

- Understand the underlying architecture both Andriod and iOS use to efficiently render large scrollable lists of items.
- Create scrollable lists of items in iOS using `UITableView` and its delegates.
- Create scrollable lists of items on Android using `RecyclerView` and its supporting classes.

6.1 Introduction

Presenting a scrollable collection of items is a very common user interface element in most mobile apps. Some examples from popular apps include:

- Facebook: a list of posts where each post includes user name, time of post, number of views, number of comments, and response buttons (Like, Comment, Share).
- YouTube: a list of videos where each video includes a title, name of owner, number of views, channel name, and action menu.
- Instagram: each post includes a photo, action buttons, number of likes, user name, and comments from other users.

- Skype: list of contacts, list of conversations.
- WhatsApp Messenger: a list of contacts (profile photo, phone number), and a list of chats.
- Twitter: a list of tweets.
- Pocket Casts: a list of podcasts, and a list of episodes.

The data displayed by these example applications are representative of lists displayed by most mobile apps. In general, apps display lists of items in a variety of common styles, determined by the use case at hand.

- Items can be presented individually within a single list, or grouped together into one or more sections based on some common properties. For instance, a list of restaurants can be grouped by their cuisine type (Italian, Chinese, Mediterranean, etc.), or a list of electronic products can be grouped by the manufacturer name.
- All the items are presented on views of uniform height (e.g. a list of chats in WhatsApp Messenger) or varying height depending on the content they display (e.g. Facebook posts).
- Items can be displayed in a single-column format (most common) or a multi-column grid-like layout (e.g. list of podcasts in Pocket Casts)

The list of items being displayed by the application is usually much larger than what can be displayed on a single screen, hence these lists are scrollable, allowing a user to scroll up and down through the list. In addition, the user can also interact with each individual items using gestures such as tap, long tap, and swipe. In response to these common gestures, the app may take a number of possible actions:

- Transition to another screen that shows a more detailed information of the selected item. Sometimes, the secondary screen is also another list. For instance, tapping on the "comments" field in a Facebook post will transition to a list of comments associated with the post.
- An item in the list may display a single action button which expands to display more choices. For example, each video item in the YouTube app has a "triple dots" button that expands to a short list of actions when tapped.
- An action such as a tap on an item may be handle in place instead of transitioning to another screen. For instance, marking a podcast episode as "Favorite" will toggle a flag on the podcast item.
- In addition to the list of items, the screen may also display other widgets for updating the list. For instance, at the bottom of its list of comments, Facebook displays a input field for entering new comments.
- Multiple items can be selected and then a single batch operation applied to all selected items.

While the iOS and Android platforms handle displaying and interacting with lists differently in terms of terminology and user interface specifics, the above concepts are more or less functionally realizable on both. Since both iOS and Android target small hand held devices that are often resource constrained, there are also a couple of technical challenges both platforms face in implementing the features described above. Not surprisingly, both platforms handle these challenges in very similar ways. Let's look at both of the challenges and discuss the solution in general terms.

6.1.1 Challenge 1: List does not fit on the screen!

As was already mentioned, often the list of data to be displayed does not fit on the screen at one time. For example, you might implement a screen that will render the posts in your application's activity stream from most recent to oldest. There may be a few thousands posts on the list total, but only five of the items can fit on the screen at any given moment. The obvious user interface solution is of course to add scroll bars and give the user the ability to scroll up and down. This solves part of the problem, but other issues remain. You certainly don't want to layout thousands of items when only five can appear at any given time. You would quickly run out of memory (and possibly battery!) and even if your application managed not to crash, the scrolling would be very sluggish. Instead, as the user scrolls through the list, new data items must be dynamically retrieved and rendered on the screen just in time. To render each item in a desired format on a view, you need a layout for the view. But, as you discovered in a previous chapter, loading the layout description, parsing it, and constructing the view can be costly. This holds for both Android and iOS.

For a snappy fast scrolling app that consumes minimal resources, you can reuse item layouts. For example, as the list scrolls up, the topmost item disappears and a new item shows up at the bottom of the screen. As the list scrolls down, the bottom most item disappears and a new item shows up at the top of the screen. A strategy in which the layouts of disappearing items are reused by appearing items is just whats needed here, and is exactly how both iOS and Android address this challenge. When the list is first initialized, the rendering framework creates enough views to render only the items initially visible on the screen.

To illustrate, assume that the screen can render at most seven items. Using this specific example, the list rendering framework may create *eight* views so the scrolling experience is smooth and uninterrupted. Why eight? With the list currently showing items 1–7, as it scrolls, item 1 will begin to disappear at the top of the list and simultaneously item 8 will begin to appear from the bottom of the list. There is a brief moment where both item 1 and item 8 are partially visible. After item 1 completely disappears, the list now renders items 2–8. As scrolling continues, the layout associated with item 1 can then be *reused* for rendering item 9.

6.1.2 Challenge 2: Where's the data?

A second challenge related to displaying lists of data has to do with where the data is stored. In most cases, the data that is being scrolled through is stored in a cloud datastore and must first be downloaded from a remote server before it can be displayed. Futhermore, media-files such as images are generally stored separately from textual data. For example, an application might use a web API to fetch the last fifty items on an activity stream as JSON text. Each item may refer to the thumbnail image of the user that originally posted it via a separate URL. Hence, to render such a list, the initial JSON representation must be fetched, and then individual fetches for each different thumbnail image must be fetched as well. These fetches can take a non-deterministic amount of time to complete. Often the list can be rendered as soon as the initial JSON representation has been fetched, with place holder images used for the thumbnails. In the meantime, as the thumbnail images are being fetched, the user should still be able to interact with the list. As fetches complete, the views can be updated incrementally. Both Android and iOS adopt a very specific threading model to accomplish this. A user interface thread (commonly referred to as the "main" thread) must handle all user interface related logic, while background threads are dispatched to handle all time consuming operations (e.g. network fetches) asynchronously in the background.

The additional image fetches consume both network bandwith (many users do not have unlimited data plans!) and battery, so each image should only be fetched once. For example, in a twitter feed, a significant proportion of the tweets might have originated from a single prolific tweeter. In a situation like this, we would want to cache the image on device and avoid fetching it each time it might appear in a scrollable list.

We will revisit this particular problem in a future chapter. For now we will focus on how Android and iOS efficiently render lists of items.

6.1.3 A Framework for Rendering Lists

Scrolling through a large list of items on a smartphone screen requires a bit of forethought and planning, if it is to be accomplished in a performant usable manner. Imagine a large list of items (e.g. the entries on your Facebook activity stream) in which you could scroll through seemingly infinite, constantly refreshing data. Even though there may be thousands of entries, perhaps only five or less at a time can actually be visible on the small screen at a time. Since only a small number are visible, it makes sense to only have enough views instantiated to display what is visible or soon to become visible. Furthermore, since each entry may involve a half dozen or more individual view components, rapidly scrolling through a large number of items could involve allocating and deallocating thousands of view components as rows temporarily scroll on and then off screen. The overhead of managing all this object creation / destruction will introduce temporarily performance hits that will make the scrolling less than smooth and snappy. Clearly a better approach is needed.

Based on the fact that many of the view layouts coming on and off screen while scrolling are the same or among a smaller number of layout styles, there is an excellent opportunity to reuse cells that are going off screen rather than to dispose / recreate them. This is exactly what both the underlying iOS and Android platforms do for us. Figure 6.1 depicts the basic architecture of the approach both platforms take. Imagine a Twitter user rapidly scrolling down (e.g. swipe up gesture) a stream of tweets. Initially, no tweet views exist, so assuming 3-5 tweets can be displayed simultaneously, the initial views are created when the list is first displayed. As the user scrolls down, as tweets scroll off the top of the screen (e.g. the Lyft tweet in the figure) the actual view for the row is retained in a pool of reusable views. As new tweets appear on the bottom of the screen (e.g. the tweet about dongles in the figure), the framework will first attempt to dequeue a reusable view from the pool, before creating a new one from scratch. If the user were to to switch directions and scroll up in the list (e.g. swipe down) the process reverses, as is outlined with the dashed arrows in the figure.

Both iOS and Android implement essentially this same framework for reusing views. However, despite the conceptual similarities, as you might expect, the terminology and the framework interfaces for accomplishing this does vary across the platform.

In essence, the main problem that we are trying to solve here is "How do I render a significantly large collection of data onto a palm-sized screen?" We encounter a similar problem in Computer Graphics: "Given a 3D model of objects in the world, how do I view these objects in a (2D) viewport on my screen?" A common solution to this problem is to use a camera model that performs a 3D to 2D transformation, mathematically known as a projection, that also includes perspective and clipping transformations. In the CG world, the camera models bridges the gap between the 3D world and the 2D world; the camera treats the 3D models as an input that must be transformed into a 2D viewport. When the viewport changes to see a different portion of the (3D) world, the

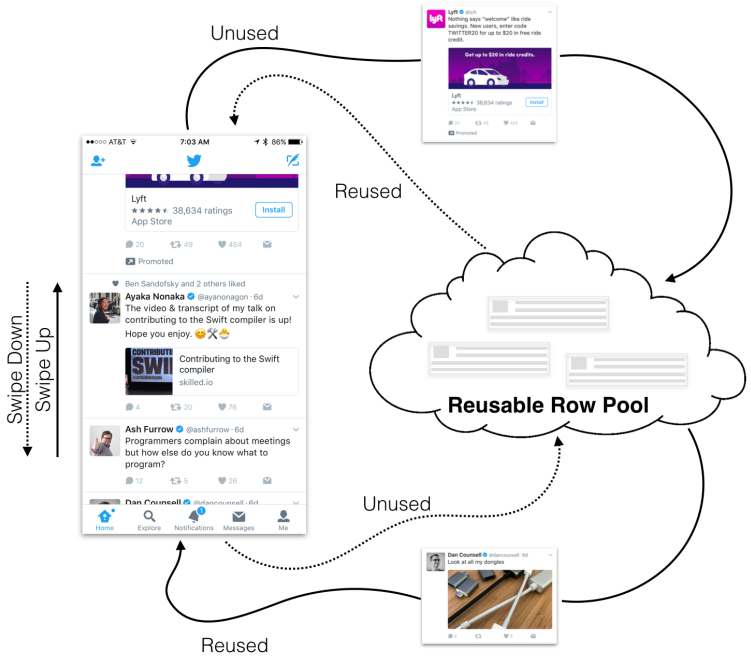


Figure 6.1: The basic architecture behind how Android and iOS efficiently render scrollable lists of items.

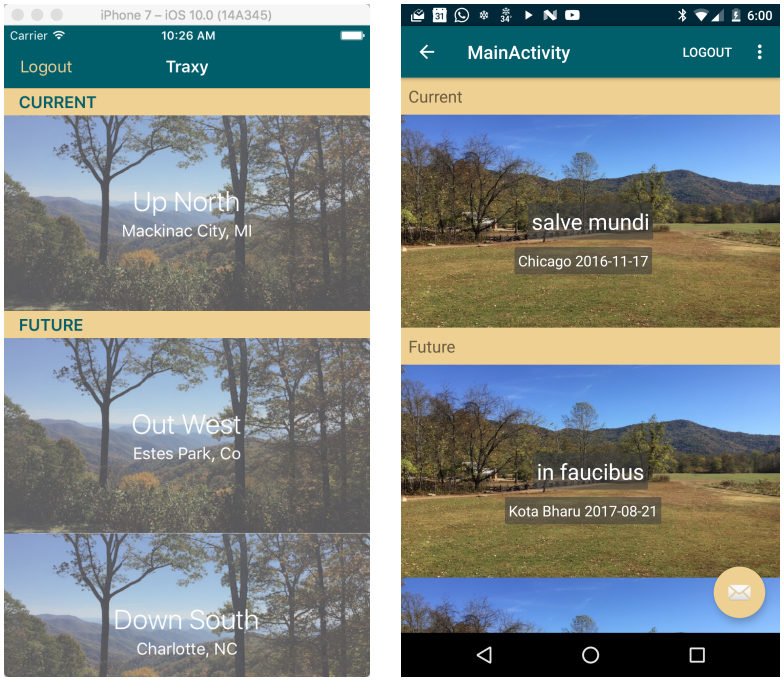
camera has to recalculate the necessary transformations.

The above analogy helps us uncover the three separate components in both the CG world as well as the general data world.

- The model / data source that must be "transformed" to fit our screen
- The viewport on screen in which the user interacts to get a different view of the world / collections of data
- The equivalent of a camera in the CG world as the bridging component, will be called an adapter

When these concepts are implemented in iOS and Android, the "viewport" object is implemented by iOS the `UITableView` or Android `RecyclerView` classes. The behavior of the adapter object is dictated by the iOS `UITableViewDataSource` protocol and Android `RecyclerView.Adapter` interface. Any sequential collection data structure that supports the indexing operation (array, linked-list, Iterable) is a good representation for the actual data source.

In iOS, all events triggered by user interactions with the list are handled by the `UITableViewDelegate` protocol. Unfortunately, Android does not provide specific interface for event handling that is equivalent to iOS `UITableViewDelegate`. On Android you will simply assign listeners to the views to respond to taps and other events.



(a) iOS (b) Android

Figure 6.2: This is what we want our final main table view to look like.

6.2 Implementing Table Views in iOS

In iOS we display scrollable collections of items using `UITableView`. The class receives its ability to scroll from its base class `UIScrollView`. It can display a scrollable stack of zero or more `UITableViewCell` instances and automatically manage the reuse of cells as discussed previously, so that only view hierarchies for those cells visible are in fact allocated. Without a lot of heavy lifting by the application developer, you can easily create nicely styled table views that scroll very efficiently.

In addition, `UITableView` is capable of partitioning the cells into groups, referred to as *sections*. This allows you to logically group collections of cells together to make the user interface more intuitive and easier for the user to scroll to the cells of interest. For example, the example Traxy app is intended to help users create multimedia journals of their travel adventures. Based on the date range of each trip, journal entries can be categorized into three categories: past, present, future. A table view can display the journals in corresponding sections.

Let's update the Traxy main view controller so that it displays a scrollable list of journals. By default, iOS offers a variety of different table view styles, and table view cell formats. In what follows, you will proceed to build your table view incrementally, initially using some of these standard layouts, and then gradually customizing your table view to get achieve the nice multi-section custom styled table view as shown in Figure 6.2.

6.2.1 Where's the Data?

Since we're talking about displaying a list of journals you might be wondering there your journal data is going to come from. Normally the collection of data displayed in a table view, would need to be retrieved from the network via a web API, or from a database on the device. For this chapter however, we will defer those details for later and focus exclusively on `UITableView` and its two delegates. Instead, implement a Swift `struct` called Journal (shown in Listing 34) that models an individual journal.

However, your table view will be displaying a collection of these journals, so define a second class called `JournalModel` (shown in Listing 35) that allows retrieval of a collection of journals. For now just return a hard-coded array of journals. Eventually you can replace this with code that retrieves a list of journals from a web API, but this simple approach will suffice for your exploration of table views in this chapter.

6.2.2 `UITableView` Overview and Some Housekeeping

`UITableView` uses the delegation pattern to decide what to display and how it should look. More specifically, `UITableView` delegates what data will be displayed via the `UITableViewDataSourceDelegate` protocol. We can customize certain aspects of the table view's style via the `UITableViewDelegate`, and even more by coming up with our own custom table view cell designs.

Before you update your Traxy app any further, first do a bit of housekeeping in the Xcode project of your Traxy app. We've discussed the model view controller architecture used by iOS in Chapter 3. Within your Xcode project navigator, create three sub folders: `models`, `views`, `controllers`, and `utilities`. Note that folders are formally referred to as *groups* in Xcode, and you can create them by selecting the top level TraxyApp group at the top of the hierarchy

Swift

```swift
struct Journal {
  var key : String?
  var name : String?
  var location : String?
  var startDate : Date?
  var endDate : Date?
  var lat : Double?
  var lng : Double?
  var placeId : String?

  init(key: String?, name: String?, location: String?, startDate: Date?,
    endDate : Date?, lat: Double?, lng: Double?, placeId : String?)
  {
    self.key = key
    self.name = name
    self.location = location
    self.startDate = startDate
    self.endDate = endDate
    self.lat = lat
    self.lng = lng
    self.placeId = placeId
  }

  init(name: String?, location: String?, startDate: Date?, endDate : Date?,
    lat: Double?, lng: Double?, placeId : String?)
  {
    self.init(key: nil, name: name, location: location, startDate: startDate,
      endDate: endDate, lat: lat, lng: lng, placeId: placeId)
  }

  init() {
    self.init(key: nil, name: nil, location: nil, startDate: nil, endDate: nil,
      lat: nil, lng: nil, placeId: nil)
  }

}
```

Listing 34: The Journal is a model data structure that represents the meta data about a journal.

Swift

```swift
class JournalModel {
  fileprivate var items : [Journal] = [Journal]()

  init() {
    createJournals()
  }

  func getJournals() -> [Journal]
  {
    return self.items
  }

  fileprivate func createJournals()
  {
    items.append(Journal(name: "Out West", location: "Estes Park, Co",
      startDate: Date.distantPaste, endDate: Date.init(timeInterval: 100000,
      since: Date.distantPast), lat: 40.3772059, lng: -105.5216651,
      placeId: "ChIJAxoZu9ZlaYcRDKzKqbeYlts"))
    items.append(Journal(name: "Out West", location: "Estes Park, Co",
      startDate: Date.distantFuture, endDate: Date.init(timeInterval: 100000,
      since: Date.distantFuture), lat: 40.3772059, lng: -105.5216651,
      placeId: "ChIJAxoZu9ZlaYcRDKzKqbeYlts"))
    items.append(Journal(name: "Down South", location: "Charlotte, NC",
      startDate: Date.distantPast, endDate: Date.init(timeInterval: 100000,
      since: Date.distantFuture), lat: 40.3772059, lng: -105.5216651,
      placeId: "ChIJAxoZu9ZlaYcRDKzKqbeYlts"))
    items.append(Journal(name: "Out East", location: "New York, NY",
      startDate: Date.distantFuture, endDate: Date.init(timeInterval: 100000,
      since: Date.distantFuture), lat: 40.3772059, lng: -105.5216651,
      placeId: "ChIJAxoZu9ZlaYcRDKzKqbeYlts"))
  }
}
```

Listing 35: Journal Model class represents (and for now) generates some fake data to be displayed.

in the project navigator and then going to File > New > Group. Any classes involving data representation will be placed in the `models` sub directory. Anything view related, such as storyboard files, custom user interface sub classes, and image assets will go in `views`. All controller code goes in `controllers`, and anything that could be used across those three categories you'll put in the `utilities` folder. Go ahead and move each file in the project to its proper place. Moving forward as you add more files you will clearly delineate between these areas and place each file in its proper place. If you did this correctly, the reorganization of your files in the project navigator will appear as in Figure 6.3. A few of the files you see listed there will be added as we work our way through the rest of this chapter.

> NOTE
>
> In Xcode's project navigator you can create as many sub-folders (groups) as you want to make it easier to organize and navigate your project's files. This will be come particularly helpful when you project grows and consists of many files. However, unlike Android Studio, these folders are merely logical groupings within Xcode. If you go down to the actual project folder via the Finder app and look at the file system, you will not see the analog file folders there. This is not a problem at all, since you will almost always be navigating your project files within Xcode. Nevertheless, for users who are used to Java and the way the folders in Android Studio match the underlying system, Xcode's approach may seem a bit unsettling.

6.2.3 Implementing `UITableViewDataSource`

Your initial focus will be to get the model data - a list of journals, to display in a table view. The high level task description is as follows:

1. Add and wire up a `UITableView` to the main view controller view in the storyboard.
2. Add a prototype `UITableViewCell` to the storyboard.
3. Update the Swift code of `MainViewController` so that it loads the model data and implements `UITableViewDataSource`.

Adding and Wiring Up a `UITableView`

Go ahead and delete the "Logging in as" label in BOTH the storyboard and the outlet that was defined by `MainViewController`, as we will no longer need this. Do make sure you delete it in both places or you will have problems when you run the app! Also make sure you remove the code in your `viewDidLoad` method that references the outlet.

Next, drag a `UITableView` (labeled "Table View") from the object library into the main view controller's scene. We want the table to fill its entire containing view. We can accomplish this easily in Auto Layout, by pinning the table view so that it is zero points from the top, bottom, leading, trailing edges of its container' safe area. This should redraw the tableview in storyboard so that it fills the screen.

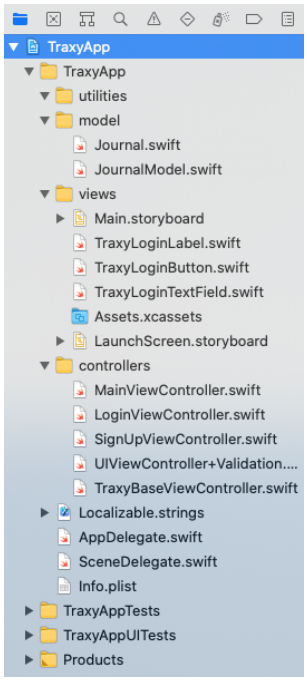


Figure 6.3: Reorganization of our project files into folders.

You also need to add an outlet called `tableView` from the `UITableView` in storyboard to the `MainViewController` class using the Assistant Editor in Interface Builder. Once the outlet is established, we need to set the table view's delegate and data source to refer to `MainViewController` which is the class that will be implementing these protocols on behalf of the table view. This can be accomplished in code by adding these two lines of code to `MainViewController`'s `viewDidLoad` method:

Swift

```
1 self.tableView.delegate = self
2 self.tableView.dataSource = self
```

Alternately, we can also set the `delegate` and `dataSource` visually in Interface Builder, as is illustrated in Figure 6.4. First, holding ctrl select the table view and then drag up to the yellow box on the top of the scene. This icon represents the view controller that will manage the scene, which in this case is an instance of MainViewController. Second, when the mouse is released, select delegate from the popup menu. Repeat this same process to set the dataSource as well.

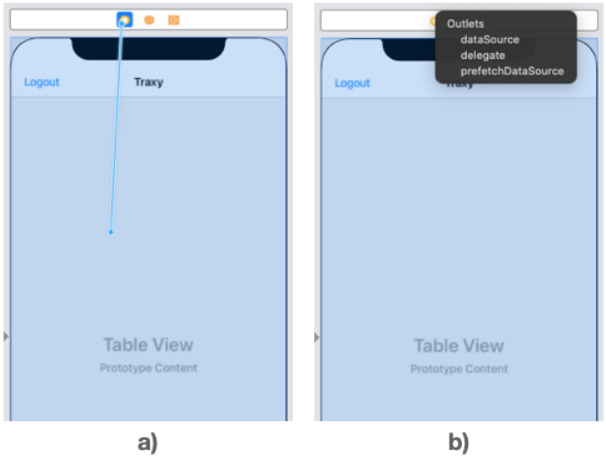


Figure 6.4: Setting the table view's delegate and dataSource properties in Interface Builder. a) Hold the Ctrl key and click on the table view and drag up to the yellow box on top of the scene. b) When the mouse is released select delegate from the popup menu. Repeat both steps for dataSource.

Add a Prototype `UITableViewCell`

It is possible to configure the rows of a `UITableView` *statically* in Interface Builder. You can accomplish this by selecting the table view and going to the attribute inspector and setting the **Content** property to "Static Cells". By default when you make this change, Interface Builder will automatically insert a single section containing three blank table view cells. You could then proceed to add additional sections / cells and configure them in Interface Builder however you would like. Static table views can be useful in certain situations where you know exactly how many cells will appear in the table *a priori*. The classic example of this would be a user preferences or settings screen.

However, in our use case the Traxy app will eventually allow the user to create and delete journals at will, so we are going to look at how to *dynamically* configure the table view. That is, you won't know how many sections or rows will be in the table, or the details of each cell until runtime. This will be accomplished by implementing the `UITableViewDataSource` protocol. Before that though, you need to determine the layout of the cells that are going to be displayed. This is done by adding *prototype cells* to the table view in Interface Builder. These cells are prototypes in the sense that they simply represent the layout with which zero or more cells will be displayed in the table view, once loaded. You can actually add multiple prototypes to the table view, one for every different style cell your table view will display.

For starters, select the table view and make sure its **Content** setting in the attributes inspector is set to "Dynamic Prototypes". Then drag a new `UITableViewCell` from the object library (labeled **Table View Cell**) and into the table view. Once placed, the cell will appear as a blank row in the table view. Select the cell that you just added and go to the attributes inspector. There are five different styles of cells that you can chose from. By default the cell style will be set to "Custom" which means you need to manually drag content from the object library into the cell. The other four options can be thought of as commonly used styles that the framework conveniently

prefabricates and makes available to the developer.

For now go ahead and select the **Subtitle** style. This style provides a standard layout with a title in a larger font, and a subtitle directly below it in a slightly smaller font. Though it is not shown in the prototype, the Subtitle style also supports a small thumbnail image that will be displayed on the left side of the cell if populated with an image. Before you leave Interface Builder and update our controller code, give your cell prototype a unique identifier. Type the string "Cell" into the **Identifier** field in the attribute inspector. You will use this identifier in your code to request a cell instance in the style of a particular prototype.

Implementing `UITableViewDataSource`

Proceed by updating your `MainViewController` class to implement the `UITableViewDataSource` as shown in Listing 36.

First note that in the class definition on the first line we've added `UITableViewDataSource` as well as the `UITableViewDelegate` protocols.

In the `viewDidLoad` method we instantiate an instance of `JournalModel` and then use it to retrieve an array of `Journal` instances, which is saved on an instance variable named `journals`. This is essentially the array of data we will be displaying in our table view. We wired up our table view delegate and data source in Interface Builder, so we do not need to do the assignments here as shown previously.

The remaining code implements three methods of `UITableViewDataSource`:

- `numberOfSections:in` - This method informs the table view how many sections will be displayed. For now, we'll place all the data into a single section, so we return the value 1.

- `tableView:numberOfRowsInSection` - Given a section index, this method will return the number of rows that are in that particular section. Sections, like rows begin at an index value of zero. For the moment, since we are placing all of our journals in a single section, we ignore the section input parameter, and simply return the number of entries in our journals array. Since the journals instance variable is optional, we do need to make sure it is not nil before we reference its count.

- `tableView:cellForRowAt` - Given an `IndexPath`, which is essentially a structure that contains a section and row, this method will attempt to configure a cell of the appropriate style, and bind the corresponding data from our data model. In this case, since we are using a Subtitle styled cell, we know from consulting the iOS documentation, that we can set the two field values as well as an image. For now, we will just hard code our image to the Traxy logo. A more realistic implementation would fetch a separate thumbnail for each journal. We will look at that in more detail in a later chapter.

Note that implementation of `tableView:cellForRowAt` elegantly implements the cell view recycling discussed earlier in this chapter. It does so by invoking the `dequeReusableCell:withIdentifier:for` method. This method, defined by UIKit will attempt to reuse a previously created cell with the given identifier. If no such cell is currently available, it will simply allocate a new one and return that. As displayed cells are scrolled off the screen, they will be marked reusable by the underlying framework, and returned on future dequeue calls. Using this technique only a small number of cell view hierarchies need to be created and can be

Swift

```swift
class MainViewController: TraxyBaseViewController, UITableViewDataSource,
  UITableViewDelegate
{

  @IBOutlet weak var tableView: UITableView!
  var userEmail : String?
  var journals : [Journal]?

  override func viewDidLoad()
  {
    super.viewDidLoad()
    let model = JournalModel()
    self.journals = model.getJournals()
  }

  // MARK: - UITableViewDataSource
  func numberOfSections(in tableView: UITableView) -> Int
  {
    return 1
  }

  func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int
  {
    if let journs = self.journals {
      return journs.count
    } else {
      return 0
    }
  }

  func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) ->
  UITableViewCell
  {
    let cell = self.tableView.dequeueReusableCell(withIdentifier: "Cell",
    for: indexPath)

    if let journal = self.journals?[indexPath.row] {
      cell.textLabel?.text = journal.name
      cell.detailTextLabel?.text = journal.location
      if let defaultImage = UIImage(named: "logo") {
        cell.imageView?.image = defaultImage
      }
    }
    return cell
  }

}
```

Listing 36: Updating `MainViewController` to implement the `UITableViewDataSource` and `UITableViewDelegate` protocols.

used over and over again as the user scrolls through the data. Note that the identifier passed here ("Cell") is the identifier you gave your cell prototype in Interface Builder.

> **NOTE**
>
> A common mistake is to forget to set a cell identifier on a table view cell prototype in Interface Builder, or to pass a non-existing cell identifier when dequeuing.

If you build and run the app, you should see the four journals defined in your model displayed in a single section as shown in Figure 6.5a. Let's take a look at how we can partition the data up in to multiple section as shown in Figure 6.5b.



Figure 6.5: The main view a) configured as a single section. b) partitioned into three sections.

6.2.4 Partitioning a Table View into Sections

You can make it easier for your users to find journals of interest by breaking up the journals into three categories: current, future, and past. The journal's start and end dates provide the basis for the partitioning. Listing 37 demonstrates how to partition the array of journals into past, present and future subsets. The new property tableViewData is an array of tuples that map the section header string to an array containing the journals within that section. Whenever this property is

set, the main UI thread is asked to reload the table view. You need to dispatch this code to the main thread simply because it could be a worker thread that ultimately does the assignment.

We've added a new instance variable called `tableViewData` which is essentially an array of tuples, where the first item in a tuple is a string (e.g. the section header name) and the second item an array of `Journal` (e.g. the journals that appear in a given section). Note that when we define instance variables we can specify code that is to be ran whenever the variable is assigned a new value. In this case, we are going to compute our sections in the form of an array of tuples that once set on our `tableViewData` instance variable will force the table view to redraw itself.

We've also added a method named `sortIntoSections` that takes an array of `Journal` and partitions it into three sub-arrays based on each journals start/end date. Once the partition is complete, we instantiate our array of tuples as described above and set it to our `tableViewData` instance variable. We invoke `sortIntoSections` in `viewDidLoad`.

Our `sortIntoSections` method uses some simple extensions to the String and Date classes. For now, we've simply placed these extensions at the bottom of our `MainViewController`.swift source file. The `Date` extension makes it possible to format a date object into a simple "MM-dd-yyyy" string format, and the `String` extension makes it possible to take a string in this format and turn it back into a `Date` object. Both extensions are shown in Listing 39.

The `UITableViewDataSource` implementation shown in Listing 38 needs to be adjusted to accommodate our new sectioned data structure as well. We can compute the number of sections by determining the number of entries in our tuple array. We can compute the number of rows within a given section by fetching the appropriate tuple and examining the number of journals in the second entry in the tuple. Finally, we can find the appropriate data to bind to a cell by using both the section and row provided to retrieve the appropriate data form our `tableViewData` instance variable.

Finally, a variety of header and footer related methods defined by the `UITableViewDelegate` protocol are also implemented. These methods provide additional information to the table view about styling the headers and footers and should be self explanatory.

If we run our app after these changes, we can see that now our trips are broken up into the three sections as was shown in Figure 6.5b.

6.2.5 Making it Pretty via a Custom Cell and `UITableViewDelegate`

You don't have to use the cell styles that iOS provides. It is fairly easy to customize how your table view looks by introducing a cell prototype with a custom layout. Let's take a look at how you can make your main view controller look even better via a custom cell prototype and implementing additional methods on the `UITableViewDelegate`. You can accomplish this by completing the following tasks:

1. Create a new custom table view cell prototype in Interface Builder.
2. Create a new subclass of `UITableViewCell` and associate it with the cell prototype.
3. Update `MainViewController` to dequeue and bind data to the new custom cell.

Creating a custom cell prototype in Interface Builder

We would like to come up with a custom cell style to match the rows shown in Figure 6.2. These rows are 200 points in height, and are completed filled with a cover photo. The journal name and

Swift

```swift
var tableViewData: [(sectionHeader: String, journals: [Journal])]? {
  didSet {
    DispatchQueue.main.async {
      self.tableView.reloadData()
    }
  }
}

override func viewDidLoad() {
  super.viewDidLoad()
  let model = JournalModel()
  self.journals = model.getJournals()
  self.sortIntoSections(journals: self.journals!)
}

func sortIntoSections(journals: [Journal]) {

  // We assume the model already provides them ascending date order.
  var currentSection  = [Journal]()
  var futureSection = [Journal]()
  var pastSection = [Journal]()

  let today = (Date().short.dateFromShort)!
  for j in journals {
    let endDate = (j.endDate?.short.dateFromShort)!
    let startDate = (j.startDate?.short.dateFromShort)!
    if today <=  endDate && today >= startDate {
      currentSection.append(j)
    } else if today < startDate {
      futureSection.append(j)
    } else {
      pastSection.append(j)
    }
  }

  var tmpData: [(sectionHeader: String, journals: [Journal])] = []
  if currentSection.count > 0 {
    tmpData.append((sectionHeader: "CURRENT", journals: currentSection))
  }
  if futureSection.count > 0 {
    tmpData.append((sectionHeader: "FUTURE", journals: futureSection))
  }
  if pastSection.count > 0 {
    tmpData.append((sectionHeader: "PAST", journals: pastSection))
  }

  self.tableViewData = tmpData
}
```

Listing 37: Breaking the array of journals into sections based on date.

Swift

```swift
// MARK: - UITableViewDataSource
func numberOfSections(in tableView: UITableView) -> Int {
  return self.tableViewData?.count ?? 0
}

func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
  return self.tableViewData?[section].journals.count ?? 0
}

func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath)
  -> UITableViewCell
{
  let cell = self.tableView.dequeueReusableCell(withIdentifier: "Cell", for: indexPath)
  guard let journal = tableViewData?[indexPath.section].journals[indexPath.row]
  else {
    return cell
  }
  cell.textLabel?.text = journal.name
  cell.detailTextLabel?.text = journal.location
  if let defaultImage = UIImage(named: "logo") {
    cell.imageView?.image = defaultImage
  }
  return cell
}

// MARK: - UITableViewDelegate
func tableView(_ tableView: UITableView, titleForHeaderInSection section: Int)
  -> String?
{
  return self.tableViewData?[section].sectionHeader
}

func tableView(_ tableView: UITableView, willDisplayFooterView view: UIView,
  forSection section: Int)
{
  let header = view as! UITableViewHeaderFooterView
  header.textLabel?.textColor = THEME_COLOR2
  header.contentView.backgroundColor = THEME_COLOR3
}

func tableView(_ tableView: UITableView, willDisplayHeaderView view: UIView,
  forSection section: Int)
{
  let header = view as! UITableViewHeaderFooterView
  header.textLabel?.textColor = THEME_COLOR2
  header.contentView.backgroundColor = THEME_COLOR3
}

func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {
  guard let journal = tableViewData?[indexPath.section].journals[indexPath.row] else {
    return
  }
  print("Selected\(String(describing: journal.name))")
}
```

Listing 38: Updated UITableViewDataSource and UITableViewDelegate methods based on the newly partitioned journal data.

Swift

```swift
extension Date {
  struct Formatter {
    static let short: DateFormatter = {
      let formatter = DateFormatter()
      formatter.dateFormat = "MM-dd-yyyy"
      return formatter
    }()
  }

  var short: String {
    return Formatter.short.string(from: self)
  }
}

extension String {
  var dateFromShort: Date? {
    return Date.Formatter.short.date(from: self)
  }
}
```

Listing 39: Simple extensions to the `String` and `Date` classes to process date objects to/from strings in the form MM-dd-yyyy.

additional information is overlaid on the cover photo as `UILabels` with white foreground color. To improve the readability of the text we will put a translucent view on top of the cover photo and dial down its alpha setting.

First, drag a new `UITableViewCell` from the object library into the table view. You can leave the previous subtitle styled cell in place for now, as we will refer to these prototypes in code using unique identifiers. That said, be sure to go to the attribute inspector and set the identifier on your new prototype cell to "FancyCell". While you are at it go to the size inspector of the new cell and give it a custom cell height of 200.

Next, drag out a `UIImageView` into the content view of the cell. Use Auto Layout to pin its leading, trailing, top, and bottom edges to the containing view with zero constants on each. Make sure the **Constrain to Margins** option is unchecked. You can go to Google image search and find a nice pretty royalty free landscape image to add to your image assets (e.g., the file `Assets.xcassets` in your project) or use one of your own vacation photos. For now we will use this image as a placeholder cover image for your custom table cell.

To increase the readability of the text that will be overlayed on the image, a translucent view will be layered over the image. Go ahead and drag out a UIView (labeled **View**) from the object library and add it to the cell's content view. Use Auto Layout to pin its leading, trailing, top, and bottom edges to the containing view with zero constants on each. Make sure the **Constrain to Margins** option is unchecked. With the view selected, goto the attribute inspector tab and change the view's default background color from White to Light Gray, and dial down its Alpha setting from 1 to 0.4. These last two tweaks are what will make this view translucent. That is, it will allow some of the color from the underlying cover photo to show through, but at the tones down the color and sharpness to the extent that it makes the text we will overlay upon the image, more readable.

Next, add two `UILabels` and arrange them within an `UIStackView`. Note that this stack view must be placed within the cell's content view, after the translucent view, and not in the translucent view itself. You can tinker with the font properties of the labels in the attribute inspector to get the right effect. Pin the leading and trailing edges of the stack view to the containing view and add an additional constraint to center it vertically within its containing view. The final assembly is shown in Figure 6.6. Make sure that the structured outline matches that shown in the figure. The order the elements appear and the hierarchical arrangement do impact how the view will be rendered! When dragging and dropping in Interface Builder its easy to inadvertently get the views in the wrong order.



Figure 6.6: The final configuration of the custom prototype cell.

We're assuming at this point you are familiar enough with Auto Layout to be able to create the constraints described above. If not, you can consult the example code made available in the book's github repository.

Subclassing `UITableViewCell`

Once the custom cell has been layed out, you need to introduce a new `UITableViewCell` subclass to simplify your controller code. Go ahead and create a new subclass named `TraxyMainTableViewCell` in the `views` folder. Once the subclass has been defined, go back into Interface Builder, select the prototype cell and update its class in the identity inspector to `TraxyMainTableViewCell`. Then from the assistant editor, add outlets to TraxyMainTableViewCell for the two labels, the translucent view, and the cover image. The code for your subclass

is shown in Listing 40.

Swift

```
class TraxyMainTableViewCell: UITableViewCell {
  @IBOutlet weak var subName: UILabel!
  @IBOutlet weak var name: UILabel!
  @IBOutlet weak var translucentView: UIView!
  @IBOutlet weak var coverImage: UIImageView!

  override func awakeFromNib() {
    super.awakeFromNib()
    self.coverImage.backgroundColor = THEME_COLOR3
  }

  override func setSelected(_ selected: Bool, animated: Bool) {
    super.setSelected(selected, animated: animated)
  }
}
```

Listing 40: A UITableViewCell subclass that will simplify your controller code.

Please note that so far when you select a controller in the assistant editor, Xcode can automatically locate the correct controller source code, making it easy to drag outlets and actions into the source. On other custom classes Xcode isn't so clever. For example, if you select the custom cell prototype, it will not automatically display the custom class in the second pane. To handle this situation, make sure you already have your story board open in an editor window and then tap the **Add Editor on Right** button on the upper right of the editor pane. This will add another editor view on the right. If you click on the custom cell class immediately after opening it will place a copy of your `TraxyMainTableViewCell` to the right of your storyboard editor. You can now proceed to hook up outlets as before.

Updating `MainViewController`

Having introduced a new custom cell, you need to update `MainViewController` to dequeue and bind data to this new cell style. Update your `tableView:cellForRowAt` method in `MainViewController` as shown in Listing 41.

Finally, you need to implement the `tableView:heightForRowAt` method of `UITableViewDelegate` to resize the height of your cells to 200 points as shown in Listing 42. Once you've have made these final changes your main view controller should now look like that shown in Figure 6.2. Go ahead and confirm that is the case.

6.3 Android `RecyclerView`

Prior to Android 5.0 API Level 21 (Lollipop), Android developers had to use `ListView` for displaying a list items. Thereafter, `RecyclerView` has been the better alternative for rendering a list. Both classes require an adapter class for interacting with items in the data source and showing them on the list. `ListView`'s main responsibility is *rendering* a list and *handling* events on the list items. It does not, however, stipulate any rules on how the views are created and populated

```swift
func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath)
  -> UITableViewCell
{
  let cell = self.tableView.dequeueReusableCell(withIdentifier: "FancyCell", for:
  indexPath) as! TraxyMainTableViewCell

  guard let journal = tableViewData?[indexPath.section].journals[indexPath.row] else {
    return cell
  }

  cell.name?.text = journal.name
  cell.subName?.text = journal.location
  cell.coverImage?.image = UIImage(named: "landscape")

  return cell
}
```

Listing 41: Updated UITableViewDataSource method based on the custom cell class.

```swift
func tableView(_ tableView: UITableView, heightForRowAt indexPath: IndexPath) ->
CGFloat
{
  return 200.0
}
```

Listing 42: Updated `UITableViewDelegate` method to specify the custom cell's height.

by the adapter. On the contrary, RecyclerView's main responsibility is *recycling* the views for rendering list items. Event handling under RecyclerView becomes the responsibility of each individual items in the list.

Before a deep dive into RecyclerView, let's take a moment and establish a couple of definitions:

- *Inflating* is converting a layout into a view. This process involves parsing the associated XML file, building the widget tree, performing measure and layout phases, and rendering the view.
- *(Data) Binding* is updating various elements of a view with actual data (e.g. calling setText(), setDrawable(), etc.). Sometimes, we also use the term "*populating* a view with data".

Between the two tasks above, *inflating* is a more expensive operation.

RecyclerView offers improvements over ListView in the following areas:

1. Inflating an XML layout into a "live" view is costly. RecyclerView avoids unnecessary layout inflations by recycling unused views. ListView does not enforce a strong recycling policy. Users of ListView have an opportunity to reuse views, but many miss this opportunity in their code.
2. RecyclerView clearly separates *inflation* from *data binding*. Layout inflation is assigned to one method (onCreateViewHolder) and view binding to another (onBindViewHolder). Recycling of views is handled internally by the framework. The view holder class is also defined internally (RecyclerView.ViewHolder) and serves as in-memory storage for "live" views.
3. ListView's adapter change notification mechanism covers too broad a scope, it can only notify that "something" in the list has changed without providing further details. The notification mechanism of RecyclerView's provides a finer level of control. This allows the host view to respond when, for instance, a range of rows (instead of the entire list) have changed. Contrasting these two approaches, ListView must update the entire list, but RecyclerView's updates are more selective.
4. Visible cells in a ListView can only be rendered in a single-column. RecyclerView offers several classes for arranging its cells into various layouts.
5. Item decoration and animations are seamlessly integrated into RecyclerView.

It should be clear to readers that you are encouraged to use of RecyclerView over ListView. References to ListView are included in this section only for completeness.

This brief introduction should give you a hint that RecyclerView does not work alone, it works in together with other supporting classes, among which are the following:

- Adapter (RecyclerView.Adapter): an object that bridges the data source with the view
- View holder (RecyclerView.ViewHolder): an object for caching inflated views
- Layout manager (RecyclerView.LayoutManager): an object for controlling the layout for presenting all the items in a list

Among these three classes, you will spend most of your time writing a companion adapter and view holder for your RecyclerView.

6.3.1 Adapters and View Holders

The main responsibilities of an adapter are creating views, creating view holders, and binding data from data source to the views. In addition, if you want to make the list items actionable, the adapter has an extra responsibility of setting up event handling for each item. Why do we need a view holder? Well, we know that layout inflation is expensive and we should not discard such expensive objects. A view holders is simply a simple object for caching a view, so it can be retrieved for future use.

Reusing the Views

The general idea of "reusing" the views has been explained in the introduction of this chapter, but we will now revisit the idea in the context of the Android runtime. Imagine we have a list of 100 items stored in our data set and assume that the screen space can only render 7 cells at a time. In order to conserve memory, `RecyclerView` allocates only at most 8 `ViewHolder` objects[1]. As a consequence, the `onCreateViewHolder()` method of your adapter implementation is invoked 8 times. Each call triggers an inflation of the layout associated with each item. Before each cell is rendered, its view holder must be bound to the correct data item from the data set, which is a task delegated to `onBindViewHolder()`.

Now, what happens if the list scrolls up? After cell 1 totally **disappeared** (items 2–8 are now visible), its `ViewHolder` can be **recycled** to render item 9 (assuming the user continues to scroll up). Likewise, the recycled `ViewHolder` has to be (re)bound with the appropriate information from the data set, and `onBindViewHolder()` is invoked one more time on behalf of item 9. It should be obvious from this discussion, that `onCreateViewHolder()` is never invoked again to handle user scroll actions.

Layout Managers

`RecyclerView` provides three layout managers for arranging list items: `LinearLayoutManager` (a one-column layout), `GridLayoutManager` (places items in a 2D multi-column grid layout), and `StaggeredGridLayoutManager` (places items in a staggered 2D grid). In the remainder of this chapter you will be using `LinearLayoutManager`.

You are about to render a list of journals using `RecyclerView` on the main screen. Currently, the main screen shows only the user email address. We will change the screen design to show a list of journal entries. To anticipate for other tasks manipulating the journals, let's create a Kotlin data class for it. As a starter, we will keep only its name and location. Later we will add other details to the class. To anticipate future use of a database for storing our data, we would also add a key field.

Kotlin

```
1 package edu.gvsu.cis.traxy
2
3 data class Journal(val key: String, val name:String, val location: String)
```

Listing 43: Journal Data Class

[1] This is an overly simplified explanation that skips over many technical details of the actual implementation.

6.3.2 Your First RecyclerView

The final presentation of the main screen will look like Figure 6.2, where the journal items are grouped into three sections by date (Current, Future, Past). Each journal item includes a journal name, photo, start date, end date, location (geographic name), and geo location (latitude, longitude). Ultimately, the actual data will be downloaded from a cloud database server, but we will postpone that to a later chapter. For now, all the journal items will be hardcoded in your program, just as we did in the previous iOS example.

The `RecyclerView` class is provided under Jetpack Androidx. Android Studio should automatically insert the following dependency whenever you start using RecyclerView in your app. Otherwise, you may have to add the following line in your `build.gradle` (use newer version when it is available):

```
implementation 'androidx.recyclerview:recyclerview:1.1.0'
```

> **NOTE**
>
> Throughout the following steps of adding new classes / editing code in Android Studio, expect to see some compile errors. The ⌥+⏎ shortcut is your best helper to resolve most of these errors.

Setting Up Layout files

Let's work backward from the UI to the adapter then to the data source. Our first step is to prepare the layout files to render the list and its items.

1. Edit the layout file `content_main.xml`. Replace the TextView with a RecycleView and set its id to `journal_list`.

2. Fix any compile errors as the result of these changes.

3. Create a new layout file `journal_row.xml` for showing each individual item in the list. Be sure to set the layout width of the root component to `match_parent` and its height to `wrap_content` For now, each item will show a photo (`ImageView`), journal name and location (both are of type `TextView`). As a temporary "photo" the image view will show the Traxy logo for now. Set the image view's id to photo. The journal name TextView id should be set to name, and the location `TextView` id should be set to location. The design of the completed layout should look similar to Figure 6.7. In the design, the image looks as if it is a background, but *it is not*. To achieve this effect, the image must be placed as the *first child*, so it won't be covering the other two text view widgets. The journal name is placed horizontally and vertically centered and the location name is horizontally centered below it.

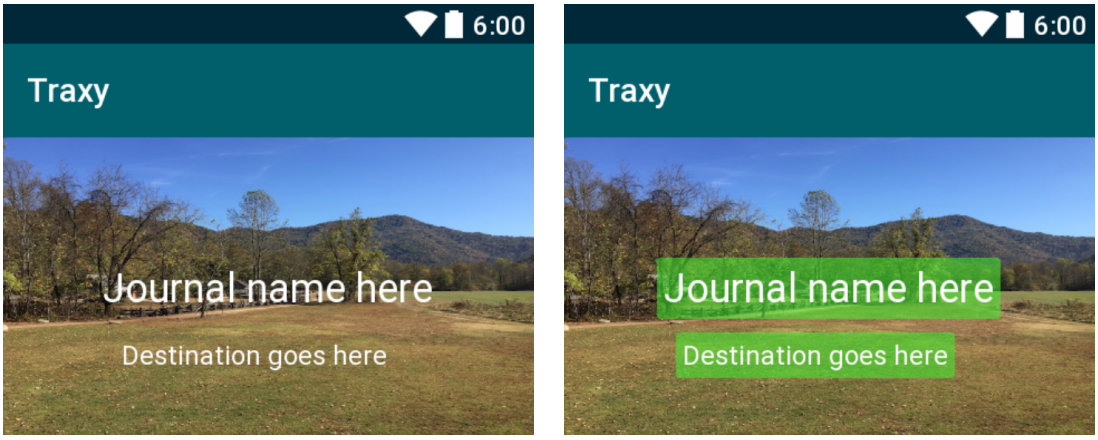


Figure 6.7: Layout of Journal Item (Option 2).

Setting Up Adapter

This section is perhaps the most complicated portion of using RecyclerView; it connects many "loose" pieces together. Most tutorials on this topic would guide you through writing a class that extends `RecyclerView.Adapter`. However, we will take a slightly different approach and write a class that extends the `ListAdapter` class, a subclass of `RecyclerView` that works well with Jetpack `LiveData`. For simplicity of explanation we will connect the adapter to a `LiveData` later.

For efficient rendering of list items, the `ListAdapter` class requires two callback functions to check if two list items are the same. The callback functions are wrapped in another class `DiffUtils.ItemCallback`. Let's call this class `JournalDiffUtil`.

Kotlin

```
package edu.cis.gvsu.traxy
class JournalDiffUtil: DiffUtil.ItemCallback<Journal>() {
  override fun areItemsTheSame(oldItem: Journal, newItem: Journal) =
    oldItem === newItem // Referential equality


  override fun areContentsTheSame(oldItem: Journal, newItem: Journal) =
    oldItem.key == newItem.key &&
    oldItem.name == newItem.name &&
    oldItem.location == newItem.location
}
```

Your next step is to prepare a ViewHolder class to cache the inflated views created by RecyclerView so they can be reused (recycled). The number of "holders" in this class depends on the number of data items displayed on each row of the list. For instance, if each cell of the list renders two `TextView` and one `ImageView` then your own ViewHolder should declare, at the minimum, these three attributes.

Kotlin
```
class JournalViewHolder(v: View) : RecyclerView.ViewHolder(v) {
  val photo: ImageView
  val name: TextView
  val location: TextView

  init {
    // These R.id.xxx must match what you use in journal_row.xml
    photo = v.findViewById(R.id.photo)
    name = v.findViewById(R.id.name)
    location = v.findViewById(R.id.location)
  }


  // Invoked by the adapter
  public fun bindTo(d: Journal) {
    // photo is not yet used
    name.text = d.name;
    location.text = d.location
  }
}
```

Your adapter class must extends the parent class `ListAdapter`. Be sure to refer to import `androidx.recyclerview.widget.ListAdapter` and not the antiquated `android.widget.ListAdapter`! The `ListAdapter` abstract class expects two generic types: the item type (`Journal`) and the view holder type (`JournalViewHolder`)) and its constructor expects a DiffUtil object to be passed as an argument.

Kotlin
```
class JournalAdapter: ListAdapter<Journal, JournalViewHolder>(JournalDiffUtil())
{
  override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
     JournalViewHolder {
    val view = LayoutInflater.from(parent.context)
                             .inflate(R.layout.journal_row, parent, false)
        return JournalViewHolder(view)
      }

  override fun onBindViewHolder(holder: JournalViewHolder, position: Int) {
    holder.bindTo(getItem(position))
  }
}
```

- `onCreateViewHolder()`. This method is invoked as many times as the number of `ViewHolder` needed to fit the screen space (plus 1). In our hypothetical example above, this method is invoked 8 times.
- `onBindViewHolder()`. This method is invoked in response to user scroll gestures, when a new cell must be rendered by binding new data from the next (or previous) journal entry to the view. More frequent scrolls result in higher number invocations of this method.

Using The Adapter

The last step is to attach the adapter to the RecyclerView and inject some sample data.

1. Edit `MainFragment.kt`
2. Add a new instance variable:

Kotlin
```
lateinit var adapter:JournalAdapter
```

3. Change (or add) its `onViewCreated()` to the following

Kotlin
```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
  super.onViewCreated(view, savedInstanceState)
  adapter = JournalAdapter()
  val layoutMgr = LinearLayoutManager(requireContext())
  with(view) {
    journal_list.adapter = adapter

    journal_list.layoutManager = LinearLayoutManager(context)
    // Show a horizontal divider between items
    journal_list.addItemDecoration(
      DividerItemDecoration(context, layoutMgr.orientation))

  }
  // Add 100 sample entries
  adapter.submitList(List(100) {
    Journal("key-$it", "Name $it", "Location $it")
  })
}
```

Listing 44: Attaching the adapter to the RecyclerView

NOTE

The `with` statement helps you write shorter code and avoid unnecessary repetitions. Without it the snippet in Listing 44 would have to be written as follows:

Kotlin
```
view.journal_list.adapter = JournalAdapter()
view.journal_list.layoutManager = LinearLayoutManager(requireContext())
```

Moreover, it is important to understand the the `submitList()` call is required to trigger the adapter to do some work to render the list of items. If for some reasons you forget to include this call, the RecyclerView will show an empty screen.

You are now ready to run the app and see `RecyclerView` in action. Running your app now will take you to the login screen that requires email and password before you see the main screen. For convenience during development and testing, you can manually initialize the email and password fields in code by adding the following lines in `LoginFragment`

Kotlin
```
email.text.insert(0, "user@test.com")
password.text.insert(0, "traxy1234")
```

With these two text fields pre-initialized programmatically, at the login screen you can simply press the "SignIn" button to go directly to the main screen.

Congratulations! You just learned how to use a RecyclerView in a fragment. This is just the beginning of your journey.

NOTE

Among the smart completion features in Android Studio (IntelliJ) is "abbreviated" name of commonly-used expressions. For instance,

- **sout** expands to `System.out.println ()`
- **souf** expands to `System.out.printf ()`
- **logd** expands to `Log.d()` with the current method name included in the log message
- **logt** creates a TAG string constant used in Log messages.
- **soutp**, **soutm**, **logm**: you should try these yourself!

A cousin of this feature is "Postfix Code" completion, where the abbreviated name is typed with a dot after a variable name. For instance, if `shops` has been declared as an array, then typing `shops.fori` expands to a for-loop on each element of the array.

The white font chosen in the row layout design may not be as readable when the image is dominated by light colors. To fix this problem, we should set the text background to a *transparent* shade of gray[2]. You can easily create such a background using a `ShapeDrawable` resource defined in XML. The XML tags accepted by this class allows you to control its shape, corner radius, gradient/solid color, padding, and line stroke.

Before you create the resource, let first understand how Android encodes colors using its hexadecimal notations. A hex color constant begins with a pound character (#), followed by the transparency (optional) and the three color components (red, green, blue). The hex '0' or '00' represents zero percents, and 'F' or 'FF' represents 100 percents. Some examples are provided below:

- `#0F0` or `#00FF00`: this is a color code *without explicit transparency*. The hex constant represents an opaque, fully-saturated green. Using a 1-digit code per channel allows only 16 different shades of that color, using a 2-digit code allows as many as 256 different shades per color channel.
- `#30F0` or `#3200FF00`: the first digit (or the first two) is the transparency. The hex constant represents a 20% transparent, fully-saturated green (in the single-digit encoding the transparent value '3' is $\frac{3}{15}$ away from the maximum, in the 2-digit encoding the transparent value 32 is equivalent to $\frac{50}{255}$),

Let's now create a new `ShapeDrawable`.

[2]The sample screenshot purposefully uses a greenish background for clarity of presentation.

1. Select New > Android Resource File from the `res` folder. In the next dialog: select "Drawable" for Resource type, enter `transparent_box` in File name, and `shape` in Root element.

2. In the editor window, add an attribute `android:shape="rectangle"` to `<shape>`. Then add a child tag `<corner>` and set its radius attribute to 2dp. Android Studio's smart code completion should provide you adequate hint for the detailed syntax of each tag/attribute.

3. Add a child tag `<solid>` and set its color to a transparent gray of your preference. Anywhere your XML includes a color constant, the editor left margin will show a shortcut to the color chooser dialog shown in Figure 6.8.

```
shape
1  <?xml version="1.0" encoding="utf-8"?>
2  <shape xmlns:android="http://schemas.android.com/apk/res/android"
3         android:shape="rectangle">
4      <corners android:radius="2dp" />
5      <solid android:color="#92414141" />
6      <padding android:bottom="4dp" android:left="4dp" android:right="4dp" android:top="4dp" />
7  </shape>
```

4. Add a child tag `<padding>` to set a 4dp padding from its enclosed contents.

5. Edit `content_main.xml` and the the transparent color as a background for the two TextViews

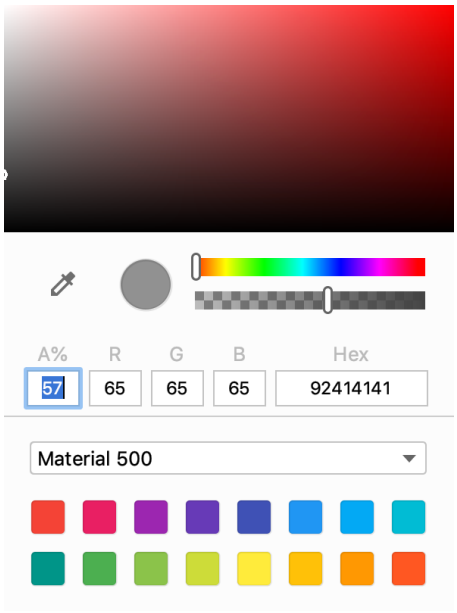


Figure 6.8: Color Chooser Dialog

The next improvement of the app is to add two date fields to the journal class and group the list of journal items into three sections/partitions (current, past, future). The Oracle Java team admits that the `java.util.Date` class exhibits poor API design [9]. Instead of using this class to add date fields, let's use the Joda-Time library by adding the following dependency:

```
implementation 'net.danlew:android.joda:2.10.6.1'
```

With this library in place, it is very easy to do date/time arithmetic operations using readable syntax.

Kotlin

```kotlin
val today = DateTime.now()
val threeDaysAgo = today.minusDays(3)
val nextMonth = today.plusMonth(1)
```

Use the above sample as your reference for generating DateTime, using other JodaTime methods should be fairly intuitive.

The library requires a one-time initialization by calling its `init()` method. The best place for making one-time calls is from `Application.onCreate()`, similar to Listing 45.

Kotlin

```kotlin
class TraxyApp : Application() {
  override fun onCreate() {
    super.onCreate()
    JodaTimeAndroid.init(this)
  }
}
```

Listing 45: Initialize the Joda Time Library

Create a new class `TraxyApp` that extends `Application` and override its `onCreate()` to initialize the Joda Time library Edit the manifest and add the attribute `android:name=".TraxyApp"` to the `<application>` tag.

With the Joda Time library in place, you can now use it to generate journal entries with date.

1. Edit `Journal.kt` add two new fields: `startDate` and `endDate` both are of type `DateTime`
2. Edit `MainFragment.kt` and change the snippet for submitting data to the adapter form

   Kotlin

   ```kotlin
   adapter.submitList(List(100) {
     Journal("key-$it", "Name $it", "Location $it")
   })
   ```

   to

   Kotlin

   ```kotlin
   val today = DateTime.now()
   val rand = Random(0)  // initialize with seed = 0
   adapter.submitList(List(100) {
     val startOn = today.plusDays(rand.nextInt(-100,100))
     val endOn = startOn.plusDays(1 + rand.nextInt(7))
     Journal("key-$it", "Name $it", "Location $it", startOn, endOn)
   })
   ```

   Be sure to use `kotlin.random.Random` and not `java.util.Random`. The code snippet generate journal entries with start date ranges from 100 days before to 100 days after today and the event duration is randomized from 1 to 7 days.

3. Edit `JournalViewHolder` and change the bindTo method to shows the start and end dates after the event location

Kotlin
```
location.text = d.location + " (" +
  d.startDate.toLocalDate().toString() + " to " +
  d.endDate.toLocalDate().toString() + ")"
```

With all these bells and whistles in place, your final list of journal items should look similar to Figure 6.9.

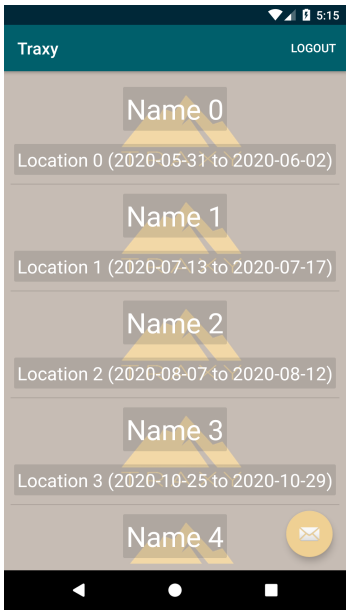


Figure 6.9: Completed List

Partitioning Journal by Date

To display the journal entries into three sections ("Current", "Future", and "Past") we have to make the following changes to the current design

1. In addition to the layout file `journal_row` for showing the journal entry we must also use a second layout file `journal_header_row` for showing the section title. This layout can be implemented using a horizontal `LinearLayout` that holds just a single TextView with id set to `section_title`

2. Consequently we must add a second ViewHolder for caching the view inflated from the new layout file above. Let's call in `TitleViewHolder`

Kotlin
```
class TitleViewHolder(v:View): RecyclerView.ViewHolder(v) {
  val title:TextView
  init {
    title = v.findViewById(R.id.section_title)
  }

  public fun bindTo(h:Header) {
    title.text = h.title
  }
}
```

NOTE

Unlike Java that restricts one `.java` file can only hold one top-level class declaration, Kotlin allows you to declare as many classes in a single `.kt` file. Instead of writing the above class in a new file `TitleViewHolder.kt`, you can just put it as the second class in `JournalViewHolder.kt`

3. Our data source was a `List<Journal>`. To implement this feature, we have at least two options:

   - Use two data source: one `List<Header>` for the section title and the original `List<Journal>`
   - Use only one data source `List<ListItem>` where `ListItem` is the parent class of both `Journal` and `Header`

   We will go for the latter option

Kotlin
```
open class ListItem (){}

data class Journal (val key: String, val name:String,
                    val location: String,
                    val startDate: DateTime,
                    val endDate: DateTime) : ListItem()

data class Header(val title:String):ListItem()
```

4. The `JournalDiffUtil` can't be doubled up either, it must be redesigned to handle both `ListItem` types

Kotlin
```
class JournalDiffUtil : DiffUtil.ItemCallback<ListItem>() {
  override fun areItemsTheSame(oldItem: ListItem, newItem: ListItem):
      Boolean {
    if (oldItem is Journal && newItem is Journal)
      return oldItem.key == newItem.key
    if (oldItem is Header && newItem is Header)
      return oldItem.title == newItem.title
    return false
  }


  override fun areContentsTheSame(oldItem: ListItem, newItem: ListItem):
      Boolean {
    if (oldItem is Journal && newItem is Journal)
      return oldItem.key == newItem.key &&
        oldItem.name == newItem.name &&
        oldItem.location == newItem.location
    if (oldItem is Header && newItem is Header)
      return oldItem.title == newItem.title
    return false
  }
}
```

5. We can't double up the adapter either. One RecyclerView works only with one adapter so the current adapter must be redesigned to work with multiple layout files and multiple ViewHolder

In order to redesign `JournalAdapter` to handle multiple types, we have to make the following changes:

1. Add two new integer constants to identify the type of list item we are dealing with

Kotlin
```
private val JOURNAL_TYPE = 1
private val HEADER_TYPE = 2
```

2. Change the class header from:

Kotlin
```
class JournalAdapter: ListAdapter<Journal, JournalViewHolder>
    (JournalDiffUtil())
```

to

Kotlin
```
class JournalAdapter:
  ListAdapter<ListItem, RecyclerView.ViewHolder>(JournalDiffUtil())
```

3. The revised code for `onCreateViewHolder()` looks like the following:

Kotlin
```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
    RecyclerView.ViewHolder {

  when(viewType) {
    JOURNAL_TYPE -> {
      val view = LayoutInflater.from(parent.context)
                    .inflate(R.layout.journal_row, parent, false)
      return JournalViewHolder(view)
    }
    HEADER_TYPE -> {
      val view = LayoutInflater.from(parent.context)
                    .inflate(R.layout.journal_header_row, parent, false)
      return TitleViewHolder(view)
    }
    else -> throw IllegalArgumentException("Unknown type")
  }
}
```

4. The `onBindViewHolder()` function must also selectively perform different binding logic

Kotlin
```
override fun onBindViewHolder(holder: RecyclerView.ViewHolder,
    position: Int) {
  when(getItemViewType(position)) {
    JOURNAL_TYPE -> {
      val journal = getItem(position) as Journal
      (holder as JournalViewHolder).bindTo(journal)
    }
    HEADER_TYPE -> {
      val header = getItem(position) as Header
      (holder as TitleViewHolder).bindTo(header)
    }
  }
}
```

5. Both functions above depend on another overridden function below

Kotlin
```
override fun getItemViewType(position: Int): Int {
    return when(getItem(position)) {
        is Journal -> JOURNAL_TYPE
        is Header -> HEADER_TYPE
        else -> 0
      }
  }
```

If you rerun the program using the same sample data you had in `MainFragment`, the rendered list output should be the same. We did not change the format of the data source which was a `List<Journal>`. In order add section header, we have to change the data source into a `List<ListItem>`, group the journal objects by their startDate or endDate and insert appropriate header text before each group.

You can replace the old code in `MainFragment.kt`

Kotlin

```kotlin
val today = DateTime.now()
val rand = Random(0)

adapter.submitList(List(100) {
    val startOn = today.plusDays(rand.nextInt(-100,100))
    val endOn = startOn.plusDays(1 + rand.nextInt(7))
    Journal("key-$it", "Name $it", "Location $it", startOn, endOn)
  })
```

with the following Kotlin snippet to preprocess the data

Kotlin

```kotlin
val today = DateTime.now()
val rand = Random(0)

val journalData = List(50) {
    val startOn = today.plusDays(rand.nextInt(-100, 100))
    val endOn = startOn.plusDays(1 + rand.nextInt(7))
    Journal("key-$it", "Name $it", "Location $it", startOn, endOn)
  }
val partitioned = journalData.map {
    when {
        it.endDate.isBeforeNow() -> "Past" to it
        it.startDate.isAfterNow() -> "Future" to it
        else -> "Current" to it
      }
  }
.sortedBy { it.first }
.groupBy({ it.first }, { it.second })
.flatMap {
    listOf(Header(it.key)) + it.value
  }
adapter.submitList(partitioned)
```

Listing 46: Partition Journal Entries by Date

The chain of function calls above works as follows:

- The `map` call turns a `List<Journal>` into `List<Pair<String,Journal>` where the String labels indicate whether event has passed, a future event, or is currently happening
- The `sortedBy` sorts the pairs by its label. Luckily our choice of labels in English yields the desired order of the journal entries (their alphabetical order are "Current", "Future", "Past"). This trick may not work in other languages, in which case you it would be better to use an integer "labels"
- Without the second lambda, the `groupBy` would normally transform the incoming `List<Pair<String,Journal>>` into a `Map<String,List<Journal>>` where the keys of the map are "Current", "Future", and "Past". But the second lambda selects the output to only `List<List<Journal>>` where the outer list hold exactly three sublists

- Finally, the `flatMap` function flattens the nested list into a `List<Header<Journal>>`; a sequence of Header (1 instance) and Journal (N instances). This subsequence repeats 3 times, hence making the list in the proper format ready to be rendered by RecyclerView.

6.3.3 Using ViewModel With RecyclerView

To anticipate a future task of handling data source from a database (either local or remote) we intentionally derive our adapter from the new `ListAdapter` because it works well with Jetpack `LiveData`. We will continue to make further improvements to the current code. In earlier section you learned about storing the user data using `LiveData<String>`. Based on what you learned there, you will now add another variable in the current ViewModel class that stores the journal entries in a `LiveData<List<Journal>>`.

1. Edit `UserDataViewModel` and add the following snippet

Kotlin

```
var _journals = MutableLiveData<MutableList<Journal>>()

val journals get() = _journals

fun addJournals(newData: List<Journal>) {
    // Create a new list when the current one is null
    val current = _journals.value ?: mutableListOf<Journal>()
    current.addAll(newData)
    _journals.value = current
  }
```

2. Move lines 1–8 of to the `onActivityCreated` function and add the data to our ViewModel

Kotlin

```
// After viewModel is initialized in onActivityCreated
val today = DateTime.now()
val rand = Random(0)

val journalData = List(50) {
    val startOn = today.plusDays(rand.nextInt(-100, 100))
    val endOn = startOn.plusDays(1 + rand.nextInt(7))
    Journal("key-$it", "Name $it", "Location $it", startOn, endOn)
  }
viewModel.addJournals(journalData)
```

> NOTE
>
> **Warning:** This is NOT the proper place to insert new data into the ViewModel. We make this shortcut just for testing only. In a later section we will designate a different activity/fragment to add new data to the ViewModel

3. To make the RecyclerView in MainFragment updates the list whenever the LiveData changes, we must move the rest of the code in Listing 2 to the *observe* block

```Kotlin
viewModel.journals.observe(this.viewLifecycleOwner, Observer {
val partitioned = it.map {
    when {
        it.endDate.isBeforeNow -> "Past" to it
        it.startDate.isAfterNow -> "Future" to it
        else -> "Current" to it
      }
  }
  .sortedBy { it.first }
  .groupBy({ it.first }, { it.second })
  .flatMap {
    listOf(Header(it.key)) + it.value
  }
adapter.submitList(partitioned)
})
```

Notice that our strategy above is to save the **unpartitioned** list data into the ViewModel and rebuild the partitioned list data only when the ViewModel changes. This is also why `submitList` is invoked inside the observe block.

6.3.4 Event Handling

In an earlier section we mentioned that the adapter is responsible for handling the click events. Should we handle the event for the **entire** list or only at individual items level? The recommended best practice is to handle the click events for each individual item. However, the adapter itself should not implement any business logic. Instead, it should only call a delegate function passed to it from elsewhere. We employ the same technique for delegating the DiffUtils logic.

To do this in Java, we would have to create a new single method interface and then pass an object that implements the interface to the adapter. In Kotlin, there is a less convoluted way of doing the same thing. We just simply pass a (high-order) function to the adapter. Assuming each journal has a unique key of type String, the said high-order function should take a String parameter.

1. Modify `JournalAdapter` to take an *optional* function argument (`listener`). The function itself takes a single String argument and returns `Unit` Change its class declaration from

   ```Kotlin
   class JournalAdapter : ListAdapter</* same as before*/>
   ```

   to

   ```Kotlin
   class JournalAdapter (val listener: ((String) -> Unit)? = null):
   ListAdapter</* same as before*/>
   ```

   The default `null` initial value makes the adapter callable without supplying the function lambda.

2. Since we want to enable click events only on journal entries (and not on the section title) we must pass the listener when calling `bindTo` for `JOURNAL_TYPE` from `onBindViewHolder()`

Kotlin
```
when(getItemViewType(position)) {
  JOURNAL_TYPE -> (holder as JournalViewHolder).bindTo((getItem(position)
    as Journal), listener)
  HEADER_TYPE -> (holder as TitleViewHolder).bindTo(getItem(position) as Header)
}
```

3. Make the following changes to `JournalViewHolder`

   - Declare a new variable

Kotlin
```
val parentView:View
```

   - Initialize the variable in the `init` block

Kotlin
```
init {
  parentView = v // v is the argument of the primary constructor
  photo = v.findViewById(R.id.photo)
  // same as before
}
```

   - Change the signature of `bindTo` by adding a second parameter of type function and call the function in the onClickListener of `parentView`

Kotlin
```
fun bindTo(d:Journal, listener: ((String) -> Unit)?) {
  if (listener) {
    parentView.setOnClickListener {
      listener(d.key) // Pass the journal key
    }
  }
}
```

4. Edit `MainFragment.kt` so the adapter provides a lambda callback function. Specifically, change the following snippet in `onViewCreated()` from

Kotlin
```
adapter = JournalAdapter()
```

   to

Kotlin
```
// When lambda function is supplied, parentheses can be omitted
adapter = JournalAdapter {
  Log.d("Traxy", "item selected has key $it")
}
```

   so we can confirm that the click listener is wired up correctly.

Place a breakpoint on the `Log.d()` statement above, rerun the app in debug mode and confirm that the click handler works as expected. After the basic connection is setup correctly let's add a new screen to show the detailed information of the selected journal.

1. Let's create a new fragment `JournalDetailsFragment`.

   - Declare a new private variable `viewModel` and initialize it using property delegate:

Kotlin
```
private val viewModel by activityViewModels<UserDataViewModel>()
```

- Declare a new private lateinit variable `journalKey`:

Kotlin
```
private lateinit var journalKey: String
```

2. Edit the layout file to show TextViews (10) for showing the journal details and one button to go back to previous screen.



Figure 6.10: Journal details form with vertical guidelines

The blueprint sketch of the layout is shown in Figure 6.10. To line up the form labels and their associated data we use two vertical guidelines set at 15% and 40%.

3. Edit the navigation graph and add the above fragment as a new destination.

4. Add an action (`journal_details`) from the main fragment to the above fragment. Add an argument `JOURNAL_KEY` to the action

5. Edit the ViewModel by adding the following function

Kotlin
```
fun getJournalByKey(key:String): Journal? =
  _journals.value?.firstOrNull {
      it.key == key
  }
```

6. Override the `onResume()` method of the new fragment

Kotlin
```kotlin
override fun onResume() {
  super.onResume()
  journalKey = arguments?.getString("JOURNAL_KEY","n/a")!!
  val jData = viewModel.getJournalByKey(journalKey)

  // Fill in the UI textviews
  key_text.text = jData?.key
  name_text.text = jData?.name
  location_text.text = jData?.location
  start_date_text.text = jData?.startDate.toString().substring(0,10)
  end_date_text.text = jData?.endDate.toString().substring(0,10)
}
```

It is important to place the above snippet inside `onResume()` so then the UI is updated every time the fragment is resumed.

7. Edit `MainFragment.kt`, add a `navigate()` call to transition into the new fragment while passing additional data (the key of the selected list item)

Kotlin
```kotlin
adapter = JournalAdapter {
  findNavController().navigate(R.id.journal_details,
                  bundleOf("JOURNAL_KEY" to it))
}
```

Rerun the app to confirm that you see the details of the selected list item in the new fragment. The fragment is not complete because when you press the "Hide" button, it is not doing anything *yet*.

8. Override `onViewCreated()` of the new fragment to add a click listener of the "Hide" button

Kotlin
```kotlin
  override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)
    hide_ne.setOnClickListener { findNavController().popBackStack() }
}
```

Popping the backstack will discard the current fragment and the previous fragment on the stack would be resumed.

Congratulations! You have completed the basic and important skill in working with RecyclerView.

6.3.5 Custom `CoordinatorLayout.Behavior`

NOTE

The topic in this section is quite advanced. We recommend Android beginners skip this section and come back after mastering the basics.

You may notice that the FAB at the lower right corner of the main screen may obscure a small portion of the image that may be of interest to your user. To provide unobstructed view of the image, we can use a custom `CoordinatorLayout.Behavior` that observes scrolling events on the `RecyclerView` and temporarily hides the FAB while the list is being scrolled. As its name implies, the `CoordinatorLayout` is designed for orchestrating visual behaviors of its children.

Let's call this class `SimpleHideWhenScrolling` and let's create it under a new sub-package: `behavior`. Highlight the parent package `edu.cis.gvsu.traxy` and use New > Package to create the subpackage.

Before we go into details on writing the class, let's use a hypothetical example to demonstrate how to use the class. For brevity of presentation, the class names in the XML layout in Listing 47 are written in its simplest format (without their fully qualified name). The `RelativeLayout` is intentionally included in the example to demonstrate that `RecyclerView` is a grandchild of `CoordinatorLayout`. Line 7 demonstrates how the custom behavior is attached to the FAB (which is a direct child of `CoordinatorLayout`) using the `app:layout_behavior` attribute.

XML

```
1 <CoordinatorLayout ...>
2   <ConstraintLayout ...>
3     <RecyclerView ..... />
4   </ConstraintLayout>
5
6   <FloatingActionButton ....
7     app:layout_behavior=".behavior.SimpleHideWhenScrolling"
8   />
9 </CoordinatorLayout>
```

Listing 47: Hypothetical `CoordinatorLayout` Example

For beginners, the `CoordinatorLayout.Behavior` class may seem to be overwhelmingly complex, so many methods to choose from because the class is designed to cover a multitude of use cases. Fortunately, for the task you try to implement, only three methods are of your interest: `onStartNestedScroll()`, `onNestedScroll()`, and `onStopNestedScroll()`. These are the methods pertinent to customizing scrolling behavior of a widget.

But what is a *nested* scroll?

Before answering that question, let's first understand that `CoordinatorLayout` can only coordinate its *direct children* (in our example ConstraintLayout and FAB). However, it also accepts *scroll events* that originate from a view deeper in the layout hierarchy, i.e. scroll events from grandchild or grand-grandchild views, and so on. Hence, the name *nested* scrolls.

Out of the three methods above, the first one is a `boolean` method while the other two are `void`. If your custom `Behavior` class is interested in intercepting (nested) scroll events from a widget, the method `onStartNestedScroll` should return true. Refer to Listing 48.

For our use case, the interactions between the list and the FAB are as follows:

1. As the user begins scrolling the list, the `onStartNestedScroll` method is invoked by the Android Runtime. Among the input parameters are three `View` (or its descendant) that hold the following objects:
   - `child`: a reference to view where the behavior is applied (the FAB)
   - `target`: a reference to the view that triggers the nested scroll event (`RecyclerView`)

- directTargetChild: a reference to containing parent of target (ConstraintLayout)

For the purpose of temporarily hiding the obscuring button, your custom behavior will intercept only vertical scrolls, hence the if-statement for checking SCROLL_VERTICAL_AXIS. A true return from onStartNestedScroll causes the other two methods to be invoked.

2. While the user continues scrolling the list but before removing his finger from the screen, the method onNestedScroll is invoked multiple times and the value of dyConsumed and dyUnconsumed provides more details about current scroll action:

dyConsumed	dyUnconsumed	Action
non-zero	zero	the list has scrolled by the non-zero amount
zero	non-zero	the list has reached its extreme point, but the user continues to swipe by the non-zero amount

3. Finally, when the user stops scrolling and removes his finger from the screen, the method onStopNestedScroll() is invoked.

Implement the Behavior class and incorporate it into your layout file fragment_main.xml. Rerun your app to see it in action. As you scroll the list, the behavior class should be responding as expected, but unfortunately the FAB is appearing too abruptly. A more desired effect is to make it reappear gradually *after a delay*. To implement this effect, you can use ViewPropertyAnimator change the alpha (transparency) value gradually. The commented code begins with calling animate() to obtain a reference to the animator object. Following that call you can chain more function calls to customize the desired animation behavior.

Congratulations! You reach your goal for this section. Recall that earlier in this section you modified the login screen to skip entering email and password. It is now a good time to restore the file and enable normal login flow again.

Let's review the important key lessons in this chapter:

- Use RecyclerView in Android layout for rendering a collection of data
- Create an adapter class that inherits from RecyclerView.Adapter for managing the list and list item presentations
- Create view holders for caching inflated layout to improve list rendering performance
- Create a custom XML layout for rendering list items
- Create a shape drawable resource to customize the background of text view widgets
- Use Kotlin high-order functions to inject event handling logic into a list adapter
- Use CoordinatorLayout to orchestrate UI behavior/actions

Kotlin

```kotlin
class SimpleHideWhenScrolling(val context: Context, attrs: AttributeSet) :
    CoordinatorLayout.Behavior<FloatingActionButton>(context, attrs) {
    private var fabTarget: View? = null

    override fun onStartNestedScroll(
        coordinatorLayout: CoordinatorLayout,
        child: FloatingActionButton,
        directTargetChild: View,
        target: View,
        axes: Int
    ): Boolean {
        if (axes == View.SCROLL_AXIS_VERTICAL) {
            fabTarget = child
            return true
        } else return false
    }


    override fun onNestedScroll(
        coordinatorLayout: CoordinatorLayout,
        child: FloatingActionButton,
        target: View,
        dxConsumed: Int,
        dyConsumed: Int,
        dxUnconsumed: Int,
        dyUnconsumed: Int
    ) {
        fabTarget?.alpha = 0f
    }

    override fun onStopNestedScroll(
        coordinatorLayout: CoordinatorLayout,
        child: FloatingActionButton,
        target: View
    ) {
        fabTarget?.alpha = 1f
//        fabTarget!!.animate()
//            .alpha(1.0f)
//            .setDuration(500)
//            .setStartDelay(100)
//            .start()
    }
}
```

Listing 48: Custom `CoordinatorLayout.Behavior`

—Whether it's Google or Apple or free software, we've got some fantastic competitors and it keeps us on our toes.

Bill Gates

7 Using Third-Party Libraries

Chapter Objectives

After reading this chapter you should be able to

- Understand how to use the *CocoaPods* dependency management platform to incorporate third party libraries in iOS applications.
- How to properly use delegation in iOS to have a view controller return data to another view controller.
- Understand how to use the Gradle Build Tool in Android Studio to incorporate third party libraries in Android applications.
- How to have one Android activity return data back to the activity that started it.

7.1 Use or Produce?

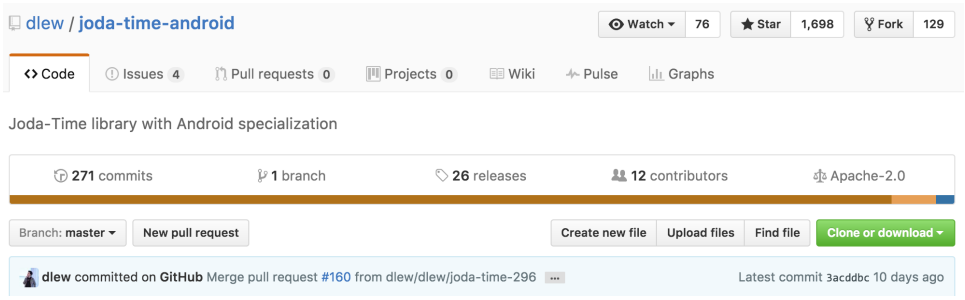
When you are asked to make a ham-n-cheese sandwich, but you don't have the necessary ingredients, it is very unlikely that, in response to this request you would:

- buy flour and wheat to make your own bread.
- buy milk, rennet, cheesecloth, and a strainer to make your own cheese.
- buy pork, curing salt, and a beehive, to cure your own ham.

While you have full control of the quality of your bread, cheese, and ham, producing all the main ingredients from raw materials takes extra effort and time, and risks not being as tasty as the easier alternative — buying a loaf of bread, sliced cheese, and sliced ham from a nearby deli and then making sandwiches. There is always a risk in taking this easier / faster approach. For instance, bacterial contamination in the cheese production line will cause food poisoning to whoever eats your sandwich. However, you can mitigate this risk by buying the ready-made ingredients from a *reputable* source, and paying attention to the expiration dates.

As a developer, you occasionally encounter a similar situation: "Should I *produce* my own code to solve a particular task in my program or *use* a library written by another developer?" If you decide to use third-party libraries, you want to choose one from reputable developers, that have been updated relatively recently. Unlike sandwich makers who must *buy* ingredients from reputable sources, software developers enjoy a plethora of *free, open-source* libraries for various programming tasks.

The majority of open-source developers publish their work on GitHub. This digital hub was established in 2008 with the goal of allowing developers to share their Git repositories and participate in *social coding*. Anyone with an account can create a project and make it available to others. One can also fork an existing project, improve, and contribute to the original project via pull requests. Other features available to GitHub users include: watching project activities, rate a project (star/unstar), and participate in discussion forums.



Unlike the products you buy at the local deli, you can judge the quality of a GitHub project by looking at the number of people following/forking it, and the recency and frequency of updates. The main dashboard of a GitHub project dashboard summary should provide adequate information for you to evaluate if a project is reputable and worth using. The number of forks is also a good indicator for future risk: a higher number of forks implies low risk, as the project has many stakeholders and support for it will likely continue for a while. A project maintained by a single developer and never gets forked is at a very high risk of turning into a discontinued project.

7.1.1 Traxy New Feature

The journal data you use for rendering the list in Chapter 6 are hard coded in the program and the user cannot add new journal entries into the current collections. In this chapter, you will change that. The user will be allowed to add a new journal by initiating the task from the list of journal entries (the main screen). This feature calls for an actionable button to initiate a transition from the main screen to another screen where the user enters the details of a new journal. Notice that you are dealing with two separate screens that may potentially share common data. For ease of

presentation, the second screen will be referred to as the "insert screen" You have several design decisions to make:

- Is there any data that must be passed to the insert screen when the main screen initiates the task?
- Is there any data that must be passed back from the insert screen to the main screen upon completion of the insert task?
- Between the main screen and insert screen, which one is responsible for managing all the operations to the current collection of journals?
- Which validation rules to check for a new journal entry? What is the screen flow for normal data? For erroneous data?

In the current design, each journal entry includes the following properties: name, location, start date, end date, and geo location (latitude & longitude). Among these six attributes, name is the only one that requires the simplest validation: it cannot be blank. Location must be a valid geographical location and therefore the user should not be allowed to enter a new location at will. To avoid this problem, you will be using the Google Places API to "enter" the location name. The library takes the current user input and prompt a list of places (geographical names/locations) that "match" the current user input. This library also automatically resolves the geo location of the selected place. To avoid erroneous date entered by the user, traxy uses a date picker dialog for the user to pick a date from.

After all the data entered by the user is validated, the insert screen will pass back a new journal object to the main screen. In response, the main screen inserts the new journal entry to the current collection and updates the list.

7.1.2 Using Google Places API

One of the libraries that must be incorporated into your project is the Google Places API. The online documentation of the library include a common landing page independent of your choice of implementation language. In addition, you will find a link to both the Android and iOS specific details.

Setting up your program to use this library requires two main steps. The first step is to install specific platform-dependent library into your programming environment. The second step, which is platform-independent, is to create a Google project from the cloud developer console, obtain an API key from the project, and install the API key into your iOS/Android app.

Use the following steps to setup a new project and obtain an API key.

1. Visit this page for an Android project or this page for an iOS Project
2. Create a new project (or select an existing project).
3. Click Continue to enable the Google Places API for iOS.
4. On the Credentials page, get an API key. Note: if you have a key with iOS/Android restrictions, you may use that key.

5. (Optional step) Select "Restrict Key" in the next dialog if you want the API be used only by requests that originate from your app. To enable this restriction you must enter your iOS bundle identifier or Android's app SHA1 fingerprint and package name. Refer to Figure 7.3 for more details.

You will use the keys you created above in the enhancements you will make to Traxy in the the remainder of this chapter.

7.1.3 Enhancing Traxy with Third Party Libraries

Let's enhance our Traxy app so that users can add new trip journals. To accomplish this on Android and iOS, we will go ahead and adopt a number of third party open source libraries. Along the way, we will explore the tools and techniques that developers typically utilize when incorporating third party components into their applications.

Figure 7.1 shows the specific enhancements we will be making in this chapter. We will use a variety of different third party libraries on both Android and iOS. Both versions of our app however, will benefit from Google's Places API. In particular, we will use the libraries Google provides us to determine the general location of the trip.

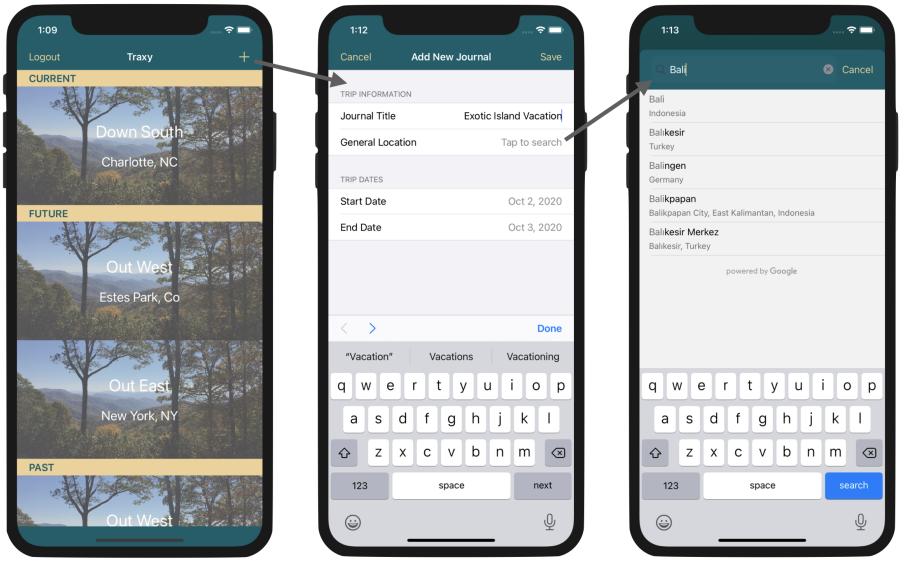


Figure 7.1: Extending the Traxy app to support the creation of new journals.

We will augment the main screen that lists out our journals with the ability to initiate the creation of a new journal. Then we will introduce a new "add journal" screen. While adding a new journal, the user will use a third party place picker that Google provides to help us quickly locate the general location of the trip that we will be journaling about. Later when we add entries to the journal we will use the device's GPS receiver to tag the exact location of the entry. For now, you can think of the Journal that we are adding at this point as simply the container of our

eventual journal entries. The Google Places API will be helpful in that it helps us establish the general vicinity of the trip and eventually we'll also use it to obtain our default cover image in the future.

7.2 Integrating Third Party Libraries in Xcode

Xcode makes it easy to add optional iOS frameworks and libraries to a build via the project properties editor. However, unlike Android Studio, Xcode does not by default provide us with broader dependency management capabilities that facilitate management of external third party libraries. Fortunately, the vibrant open source community has risen to the challenge and provided us an excellent solution.

7.2.1 Using CocoaPods to Manage External Libraries

In recent years CocoaPods has emerged as the de facto standard for managing third party libraries within Xcode. CocoaPods is an open source dependency management tool that makes it easy to incorporate external libraries into our projects. Inspired by the *RubyGems* and *Bundle* projects for the Ruby programming language, users simply enumerate the third party libraries they wish to utilize, which are referred to as *pods*, in a simple textual configuration file, known as a *Podfile*. The tool then fetches those dependencies and makes them available in Xcode via a new workspace that it generates.

If you don't already have CocoaPods installed on your Mac, you can learn more at the project's website: https://cocoapods.org/. If you are in a hurry, a single command entered from a terminal window will do the job:

```
$ sudo gem install cocoapods
```

Please note that OS X comes with the Ruby interpreter pre-installed, so this command should work fine on mosts Macs. However, if you work with Ruby regularly, you may be using RVM, a Ruby version manager that allows you to have multiple versions of Ruby installed in your local user directory, and easily switch between them. Since RVM installs Ruby in a local directory, you don't need to run the above command as superuser, so be sure to omit the `sudo` portion of the command listed above if you are using RVM on your Mac.

To bring our Traxy application under CocoaPods for dependency management, we change directory to the top level directory of our Xcode project and type the following command:

```
pod init
```

This command produces a stubbed out configuration file named Podfile in the directory with content that looks something like this:

```
target 'TraxyApp' do
  use_frameworks!

  # Pods for TraxyApp

  target 'TraxyAppTests' do
    inherit! :search_paths
    # Pods for testing
  end

  target 'TraxyAppUITests' do
    inherit! :search_paths
    # Pods for testing
  end
end
```

We are going to use a total of two different external libraries for our enhancement of the iOS version of the Traxy app:

- Eureka - helps us create elegant input forms in iOS with very little effort.
- GooglePlacePicker - lets us quickly pick a place by entering part or all of its name.

To add these libraries to our project, we edit the Podfile with our favorite text editor and place the following two lines right after the comment `"#Pods for TraxyApp"`:

```
pod 'Eureka'
pod 'GooglePlaces'
```

Note that we can find the exact pod statement by looking up the pod on cocoapods.org and/or visiting the pod's home on github.com and checking out the documentation there. Save your changes and then type the following command on the command line from the same directory containing the `Podfile`:

```
$ pod install
```

This command may take a few seconds as it fetches these two libraries, and their dependencies. Once complete, if you examine the contents of your project directory, you will notice a new file named `TraxyApp.xcworkspace`. Go ahead and type the following command:

```
$ open TraxyApp.xcworkspace
```

Or alternately, browse to your project directory and double click on the `TraxyApp.xcworkspace` file. Either way, Xcode will be launched with a fully configured workspace that makes these external libraries visible to our code.

7.2.2 Enhancing Traxy

Let's use Eureka and GooglePlacePicker to implement our new "add journal" screen in Traxy. When using pods in our project, our first stop is typically the `README.md` file on the project's github.com page. For example, the Eureka project's `README.md` has all of the information we need to get started. When using Google's pods, we can go to their developer website and find extensive guides and sample code.

Since the Google Place Picker uses Google's cloud-based data to find places for us, we need to first generate an API key on their developer console. Google uses the key to implement quotas, and will eventually want to establish a billing relationship with you if you happen to generate a lot of requests. You can generate your own key directly via https://console.developers.google.com if you are familiar with the process, or if this is the first time you've done this, you might want to walk through the process as documented in the Google Places API for iOS Getting Started guide.

Once you've generated the Google Places API key, you need to have your application associate with that key. First, insert the following two lines of code at the top of your AppDelegate.swift source file:

Swift
```
import GooglePlaces
let GOOGLE_PLACES_API_KEY = "YOUR-KEY-VALUE-GOES-HERE"
```

Be sure to replace the string literal with the key you generated on the Google developer website. Next, insert a single line of code in your `AppDelegate`'s `application:didFinishLaunchingWithOptions` method:

Swift
```
GMSPlacesClient.provideAPIKey(GOOGLE_PLACES_API_KEY)
```

You are now ready to take advantage of the Google Places Picker in your code. Before you do that, let's go ahead and add the new segue and screen for adding journals to your storyboard. In Interface Builder drag out a new **View Controller** onto the canvas just to the right of your main view controller. Then drag a **Bar Button Item** onto the right side of the main view controller's navigation bar. In the attribute inspector, set the button's System Item attribute to Add. Next, add a segue from the button over to the newly added view controller and in the attribute inspector give it the identifier `addJournalSegue`. If you did everything correctly, your scene should look approximately like that in Figure 7.2.

To implement the form, introduce a new view controller in Swift named `AddJournalViewController` that extends `FormViewController`. The base class here is defined by the Eureka pod, so you also need to be sure you import it correctly. Since you will eventually be hooking up the Google place picker, you will also add the import statement for that pod as shown in Listing 49.

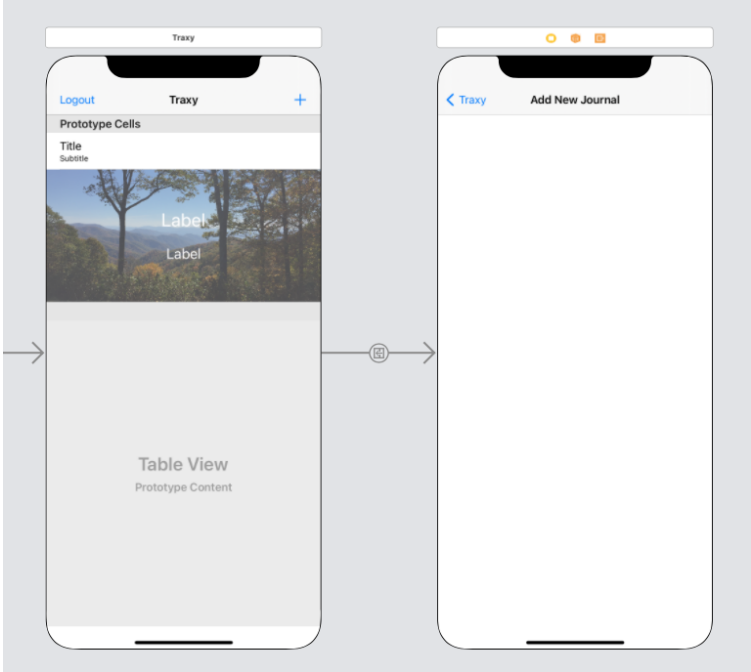


Figure 7.2: Adding a new view controller for the add journal form, and its segue..

Swift

```swift
import UIKit
import Eureka
import GooglePlaces

class AddJournalViewController: FormViewController {
   var journal : Journal?

  override func viewDidLoad() {
    super.viewDidLoad()
    if self.journal == nil {
      self.journal = Journal()
    }
  }
}
```

Listing 49: Defining `AddJournalViewController`, a concrete class of Eureka's `FormViewcontroller`.

By using the Eureka pod you've avoided a couple of frustrating challenges that iOS developers are faced with when implementing forms from scratch. First, in iOS if the text field you are editing in a form is covered up by the software keyboard, the developer has to take steps to reposition the form so that it is above the keyboard. This can be done in a variety of ways, as you can readily

confirm on stackoverflow.com. A second complication in our particular use case is to handle the entry of dates. While iOS provides us with a very nice date entry control called `UIDatePicker` you have to do a little extra work to integrate it into a form with other fields. Interestingly, as you will soon see, both of these issues are addressed by Android without developer involvement. On iOS we will reduce our workload by of simply letting Eureka solve the problem for us.

Defining a Form

If you take a look at the Eureka documentation, you will quickly realize that it is relatively easy to compose non-trivial data input forms in Swift. In our use case, we need to gather four pieces of information: journal title, general location, start date, and an end date. Define your form in with the following code in the `AddJournalViewController.viewDidLoad` method as shown in Listing 50.

The `form` variable here is an instance variable inherited from `FormViewController`. This code declares a form with two sections, "Trip Information" and "Trip Dates", respectively. The "Trip Information" section contains a `TextRow` and a `LabelRow`. A `TextRow` contains a label and a text view to enter text. This particular `TextRow` is used to collect the title text. A `LabelRow` is used to present information, but does not have any particular default input means. However, we can tap on a `LabelRow`. In this particular example, we use the tap on the `LabelRow` to display Google's place picker (implemented by the `GMSAutocompleteViewController`) by overriding the the row's `onCellSelection` method as shown in lines 22-27 in the listing.

The `GMSAutocompleteViewController`'s styling is partially addressed via our earlier use of the`UIAppearance protocol`, but alas, not completely. The color of the text in the controller's search bar defaults to black which is not very visible on our chosen background color. After a bit of research in Google's documentation, we learn that we must also rely on `UIAppearance` protocol to apply further styling to the controller. Hence, to change the search bar foreground text color to white, we add the following code to our `application:didFinishLaunchingWithOptions` method in the source file AppDelegate.swift.

Swift

```
1  let searchAppearance =
2      UITextField.appearance(whenContainedInInstancesOf: [UISearchBar.self])
3  searchAppearance.defaultTextAttributes =
4      [NSAttributedString.Key.foregroundColor : UIColor.white]
```

The location information is returned to the form by having `AddJournalViewController` implement the `GMSAutocompleteViewControllerDelegate` protocol, which you can add to your class via an extension as shown in Listing 51.

The place picker controller will call these methods at the appropriate time. For example, if we cancel out of the place picker the `wasCanceled` method is called. Similarly, if the user selected a place the `viewController:didAutoCompleteWith` method is called. Both of these methods simply attempt to validate the row so that its label attribute can be set to red or black as appropriate.

The "Trip Dates" section of the form contains to `DateRow`s. `DateRow`s automatically display and hide a `UIDatePicker` when selected and unselected. This form defines a row for the start

Swift

```swift
// Describe our beautiful eureka form
form = Section("Trip Information")
  <<< TextRow(){ row in
    row.title = "Journal Title"
    row.placeholder = "Descriptive title"
    row.tag = "TitleTag"
    row.add(rule: RuleRequired())
  }
  <<< LabelRow () { row in
    row.title = "General Location"
    row.value = "Tap to search"
    row.tag = "LocTag"
    var rules = RuleSet<String>()
    rules.add(rule: RuleClosure(closure: { (loc) -> ValidationError? in
      if loc == "Tap to search" {
        return ValidationError(msg: "You must select a location")
      } else {
        return nil
      }
    }))
    row.add(ruleSet:rules)
  }.onCellSelection { cell, row in
    // crank up Google's place picker when row is selected.
    let autocompleteController = GMSAutocompleteViewController()
    autocompleteController.delegate = self
    self.present(autocompleteController, animated: true, completion: nil)
  }
  +++ Section("Trip Dates")
    <<< DateRow(){ row in
      row.title = "Start Date"
      row.value = Date()
      row.tag = "StartDateTag"
      row.add(rule: RuleRequired())
    }
    <<< DateRow(){ row in
      row.title = "End Date"
      row.value = Date(timeIntervalSinceNow: 86401)
      row.tag = "EndDateTag"
      row.validationOptions = .validatesOnChange
      var rules = RuleSet<Date>()
      rules.add(rule: RuleRequired())
      rules.add(rule: RuleClosure(closure: { (date) -> ValidationError? in
        let sdRow: DateRow! = self.form.rowBy(tag: "StartDateTag")
        let sDate = sdRow.value as Date?
        if sDate! > date! {
          return ValidationError(msg:"End date must come after start date")
        } else {
          return nil
        }
      }))
    row.add(ruleSet: rules)
  }
```

Listing 50: Defining the from to add new journals using Eureka forms.

Swift

```swift
extension AddJournalViewController: GMSAutocompleteViewControllerDelegate {

  public func viewController(_ viewController: GMSAutocompleteViewController,
    didFailAutocompleteWithError error: Error)
  {
    print(error.localizedDescription)
  }

  // Handle the user's selection.
  func viewController(_ viewController: GMSAutocompleteViewController,
    didAutocompleteWith place: GMSPlace)
  {
    let row: LabelRow? = form.rowBy(tag: "LocTag")
    row?.value = place.name
    row?.validate()
    self.journal?.placeId = place.placeID
    self.journal?.lat = place.coordinate.latitude
    self.journal?.lng = place.coordinate.longitude
    self.dismiss(animated: true, completion: nil)
  }

  func viewController(viewController: GMSAutocompleteViewController,
    didFailAutocompleteWithError error: NSError)
  {
    // TODO: handle the error.
    print("Error: ", error.description)
  }

  // User canceled the operation.
  func wasCancelled(_ viewController: GMSAutocompleteViewController)
  {
    self.dismiss(animated: true, completion: nil)
    let row: LabelRow? = form.rowBy(tag: "LocTag")
    row?.validate()
  }
}
```

Listing 51: Defining the `GMSAutocompleteViewControllerDelegate` methods as an extension of the `AddJournalViewController` class.

date and the end date.

Form Validation

The Eureka pod also provides built in validation rules which make it relatively easy to do form validation. In the listing above you will see a couple of different types of rules being used. First, the `RuleRequired` rule validates that the field has been filled valid. Second, both the location field and the end date require specialized application specific logic for validation. In particular, we must ensure that the default "Tap to search" label is replaced with a location name, and that the end date falls after the start date. You can implement this specialized validation logic by introducing a `RuleClosure`. The `RuleClosure` allows you to codify your specialized validation logic in a Swift closure. The above code defines the validation rules, but you still need to define *when* validation will occur and *what happens* when a field is not valid. You can accomplish this by setting the `defaultCellUpdate` and `defaultOnRowValidationChanged` properties on each of the classes. As is indicated by the property names, these closures get called every time the value on a row is updated, or whenever the validation status of the row changes. In both cases you simply set the label text on each row to the color red if the current value is not valid, and black otherwise as shown in Listing 52. This code can be inserted in the `viewDidLoad` method so that it is executed once when the controller is created.

Using Delegation to Return Values from a View Controller

You've already seen how you can pass values to a presented view controller that you are segueing to. Recall in Chapter 4 how you passed the email address from the login and signup view controller to the main view controller. To review, here is the general step-by -step technique you used:

1. Add properties (e.g. instance variables) to the destination view controller to hold the data being passed.
2. Implement the `prepareForSegue(for:sender:)` on the presenting view controller. Assuming a segue has been added in the storyboard with an appropriate identifier, implement logic that retrieves a reference to the destination view controller from the passed segue and sets the appropriate properties.
3. In the destination view controllers `viewDidLoad` method, bind the data as needed to views via outlets.

So far you haven't had a scenario in the Traxy app in which a view controller must *return data* to the presenting view controller, but with the new `AddJournalViewController` we now have the opportunity to look at how this gets implemented on iOS. You actually used this technique earlier in this section when you wired up the Google place picker controller, though you may not have realized it at the time. In general, this technique is implemented using the *delegation* pattern. You've already seen delegation in action when you implemented table views earlier, and you'll see it fairly often in the future as well. Here's a general sketch of how you use it to return data form a presented view controller.

1. Have the destination view controller define a delegate protocol and add an optional instance variable that will refer to a delegate that implements the protocol.

Swift

```swift
let textRowValidationUpdate : (TextRow.Cell, TextRow) -> ()  = { cell, row in
  if !row.isValid {
    cell.titleLabel?.textColor = .red
  } else {
    cell.titleLabel?.textColor = .black
  }
}
TextRow.defaultCellUpdate =  textRowValidationUpdate
TextRow.defaultOnRowValidationChanged = textRowValidationUpdate

let dateRowValidationUpdate : (DateRow.Cell, DateRow) -> () = { cell, row in
  if !row.isValid {
    cell.textLabel?.textColor = .red
  } else {
    cell.textLabel?.textColor = .black
  }
}
DateRow.defaultCellUpdate = dateRowValidationUpdate
DateRow.defaultOnRowValidationChanged = dateRowValidationUpdate

let labelRowValidationUpdate : (LabelRow.Cell, LabelRow) -> () = { cell, row in
  if !row.isValid {
    cell.textLabel?.textColor = .red
  } else {
    cell.textLabel?.textColor = .black
  }
}
LabelRow.defaultCellUpdate = labelRowValidationUpdate
LabelRow.defaultOnRowValidationChanged = labelRowValidationUpdate
```

Listing 52: Specifying what happens when the form validation logic passes or fails.

2. Have the presenting view controller implement the delegate protocol, and sets itself to be the presented view controller's delegate in the `prepareForSegue(for:sender:)` method. .

3. The destination view controller calls its delegate at the appropriate time to pass back data to the presenting view controller.

In this particular situation, `MainViewController` is the presenting view controller, and `AddJournalViewController` is the destination view controller. When the user taps the add button on the right side of the main view controller's navigation bar, the add new journal view controller is presented. When the user fills the form and hits the save button, it returns a Journal object instance with all the fields populated by the form data.

Start by defining a delegation protocol in your `AddJournalViewcontroller` class, and adding a `delegate` instance variable as shown in Listing 53.

Swift

```
protocol AddJournalDelegate {
  func save(journal: Journal)
}

class AddJournalViewController: FormViewController {

   var delegate : AddJournalDelegate?

   // ...
 }
```

Listing 53: Specifying what happens when the form validation logic passes or fails.

Next, you need to have the main view controller implement this protocol and set its self to the be the add journal view controller's delegate, right before the segue occurs as shown in Listing 54.

The main controller's `save` method appends the newly created `Journal` instance to its array of journals, and then calls the `sortIntoSections` method which will force the table view to reload.

Finally, you need to pass a newly created Journal instance to the main view controller when the save button is pressed on the add journal view controller. You can add your cancel and save buttons and their corresponding actions in code instead of via Interface Builder as shown in Listing 55.

The Eureka pod has many additional features that we haven't explored yet here, but we have seen how easy it is to avoid extra work by using a third party pod to do the heavy lifting. Feel free to explore the Eureka and Google Places API documentation to learn more about their capabilities. We now turn to the Android implementation.

7.3 Library Management in Android

In a Gradle-based project, third-party libraries can be declared as either a compile-time and runtime dependency or runtime dependency only.

Swift

```swift
class MainViewController: TraxyBaseViewController, UITableViewDataSource,
UITableViewDelegate,
AddJournalDelegate {

  // ...
  // MARK: - Navigation
  override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    if segue.identifier == "addJournalSegue" {
      if let destVC = segue.destination as? AddJournalViewController {
        destVC.delegate = self
      }
    }
  }

  // MARK: - AddJournalDelegate
  func save(journal: Journal) {
    self.journals?.append(journal)
    self.sortIntoSections(journals: self.journals!)
  }

  // ...
}
```

Listing 54: Implementing the `AddJournalDelegate` methods on `MainViewcontroller`.

Dependency Type	Description
implementation	specify the name of a library required to compile the project. This library is also required at runtime
testImplementation	specify the name of a library required to compile the test project. This library is also required at runtime by the test environment

In the build file, library names are written according to Maven standard naming convention. Each name consists of three components: groupId, artifactId, and version and can be written in two different formats: with or without individual qualifiers. For instance, the following two dependency declarations are identical:

Gradle

```gradle
compile group: 'com.google.android', name: 'support-v4', version:'r7'
compile 'com.google.android:support-v4:r7'
```

In the newer Gradle plugin, the dependency type `compile` is renamed to `implementation`. Likewise, `testCompile` becomes `testImplementation`. Using these newer keywords, the above dependencies should be written

Gradle

```gradle
implementation group: 'com.google.android', name: 'support-v4', version:'r7'
implementation 'com.google.android:support-v4:r7'
```

Swift

```swift
class AddJournalViewController: FormViewController {

  override func viewDidLoad() {
    super.viewDidLoad()

    // ...

    let cancelButton : UIBarButtonItem = UIBarButtonItem(title: "Cancel",
      style: .plain, target: self,
      action: #selector(AddJournalViewController.cancelPressed))
    self.navigationItem.leftBarButtonItem = cancelButton

    let saveButton : UIBarButtonItem = UIBarButtonItem(title: "Save",
      style: .plain, target: self,
      action: #selector(AddJournalViewController.savePressed))
    self.navigationItem.rightBarButtonItem = saveButton

  }

  @objc func cancelPressed()
  {
    _ = self.navigationController?.popViewController(animated: true)
  }

  @objc func savePressed()
  {
    let errors = self.form.validate()
    if errors.count > 0 {
      print("fix ur errors!")
    } else {

      // extract the values from the form
      let titleRow: TextRow! = form.rowBy(tag: "TitleTag")
      let locRow: LabelRow! = form.rowBy(tag: "LocTag")
      let startDateRow : DateRow! = form.rowBy(tag: "StartDateTag")
      let endDateRow : DateRow! = form.rowBy(tag: "EndDateTag")

            // return the newly created Journal instance via the delegate
      self.journal?.name = titleRow.value! as String
      self.journal?.location = locRow.value! as String
      self.journal?.startDate = startDateRow.value! as Date
      self.journal?.endDate = endDateRow.value! as Date
      self.delegate?.save(journal: self.journal!)
      _ = self.navigationController?.popViewController(animated: true)
    }
  }

   // remainder of class implementation not shown.

}
```

Listing 55: Implementing the save and cancel buttons on `AddJournalViewController`. Notice how the save action calls the delegate to return the new journal.

7.3.1 Using Data Generator

The textual data (name and location) in the auto generated sample journal data we have so far are too uniform. A more randomized set of sample data will look more interesting and possibly uncover visual presentation issues that may not show up earlier. Searching on the internet you will find several options of data generator library. In this section we will pick one that is based on Kotlin and provides more variety of options to generate "close to real" data. The library that we would explore in this section is documented by this GitHub project:

`https://github.com/serpro69/kotlin-faker#usage`

You can integrate the library with your project by adding this line to the dependencies section of your app module build.gradle file:

```
implementation 'io.github.serpro69:kotlin-faker:1.4.1'
```

The library provides various categories of randomness: address, number, latin words, animal, etc. Let's use this library to generate our sample Journal entries. For temporary testing, we currently generate our sample data inside `MainFragment.kt`. This is not the proper place and should be removed. A more appropriate place would be to directly generate our sample data in `UserDataViewModel`

1. Edit `MainFragment.kt`, remove the `repeat` block that generates the sample Journal data
2. Edit `UserDataViewModel.kt` and add the following `init` block:

Kotlin

```
init {
  val dataGen = Faker()
  val today = DateTime.now()
  val rand = Random(System.currentTimeMillis())
  repeat(100) {
    val startOn = today.plusDays(rand.nextInt(-100, 100))

    // Combine 7 random latin words
    val randomName = generateSequence { dataGen.lorem.words() }
        .take(7).joinToString(" ")

    val d = Journal(
        "Key-$it",
        randomName,
        dataGen.address.cityWithState(),
        startOn, startOn.plusDays(rand.nextInt(10))
    )

    addJournal(d);
  }
}
```

The above init block runs only once when the ViewModel is created. This behavior will closely mimic the actual behavior of our app when later we incorporate a database connection.

Rerun the app to verify that the random data generator works as expected.

7.3.2 Google Play Services Libraries

Your Android app has access to a large collection of libraries for accessing Google APIs. These libraries are collectively known as the Google Play Services. Table 7.1 shows a selected sample of these libraries (under the groupId `com.google.android.gms`). A more complete list can be found at Google APIs for Android: Setup.

`play-services-analytics`	Google Analytics
`play-services-auth`	Account Login
`play-services-base`	Google Actions, Base Client Library
`play-services-drive`	Google Drive
`play-services-identity`	Google Address

Table 7.1: Google Play Services Libraries

Some of these services require an API key that you must obtain from the Google Developer Console. The API key is generated based on the digital signature of your app. For security reasons, any app installed on an Android device must be digitally signed using a certificate containing a public/private key pair. Digital signing is attaching the public key of a certificate to a binary build of an Android Package (.apk). Public/private key pairs are stored in a keystore file, so to prevent unauthorized access to the file, it is password protected.

During development, installing an APK to a device or an emulator also requires a digital signature retrieved from a keystore. Had a real keystore file been used during debugging and development, the user is required to enter its password every time. To avoid this frequent intrusions, a debug keystore is used instead. This keystore is a protected with a password: "`android`".

To view a keystore details, run the `keytool` command:

```
keytool -v -list -keystore debug.keystore
```

from the `.android` subdirectory on your local computer.

> **NOTE**
>
> `keytool` should automatically be available after you installed the Java JDK.

Among the output produced by `keytool`, you will refer to the SHA1 fingerprint more than the others. The fingerprint is used, for instance, to restrict an API key so access is allowed only when the API requests originate from your app.

```
[528]:.android> keytool -v -list -keystore debug.keystore
Enter keystore password:

Keystore type: JKS
Keystore provider: SUN

Your keystore contains 1 entry

Alias name: androiddebugkey
Creation date: Jul 9, 2014
Entry type: PrivateKeyEntry
Certificate chain length: 1
Certificate[1]:
Owner: CN=Android Debug, O=Android, C=US
Issuer: CN=Android Debug, O=Android, C=US
Serial number: 1c38be96
Valid from: Wed Jul 09 23:44:19 EDT 2014 until: Fri Jul 01 23:44:19 EDT 2044
Certificate fingerprints:
         MD5:
         SHA1:
         SHA256:
         Signature algorithm name: SHA256withRSA
         Version: 3
```

Figure 7.3: Using `keytool` to Show a Keystore Details

Using Gradle to Conceal API Keys

Most third party web services require an API key when invoking the API. API keys (and any other secret information) should be kept in developer private possession and not exposed in public repositories. There are several techniques for concealing secret strings using *indirection* principles. The overall idea is to replace the secret string literal in your code with an identifier that can be resolved at compile time, preferably using a technique that does not require writing additional source file. Thanks to Gradle and extra features implemented by the Android plugin, any Android source code can refer to a global variable `BuildConfig` which can be used for declaring variables using the build system.

1. Define the following two variables at the beginning of your app `build.gradle` and add necessary statement for declaring a new field in your Android's `BuildConfig`. Note that the `defaultBuild` configuration element already exists within the `android` element, so you simply need to insert the field `buildConfig` shown below.

Gradle

```
1  // in app's build.gradle
2  def secretFile = rootProject.file("secret.properties")
3  def secretProp = new Properties()
4  secretProp.load(new FileInputStream(secretFile))
5
6  defaultConfig {
7      // more lines not shown here
8      buildConfigField "String", "MY_API_KEY", secretProp["API_KEY"]
9    }
```

2. Create the file `secret.properties` at the root of your project, i.e., the directory where you keep the project's `build.gradle` with the following line

```
API_KEY="your-secret-api-key-goes-here"
```

   Be sure to include the double quotes around your API key string!

Thereafter, your code you can refer to `BuildConfig.MY_API_KEY` (declared at line 8) in place of the actual string literal. Be sure to exclude `secret.properties` from your public repository.

7.3.3 Using Google Places API

Prior to January 2019, the Places APIs are bundled into the Google Play Services library. However, the APIs have been deprecated since then and were completely disabled on July 29, 2019. The Google Places API that you will be using in Traxy below is part of the newer Places SDK for Android. To use this newer library, you must first perform the following steps:

1. Include the Google Places SDK library in your app `build.gradle`

Gradle

```
1  implementation 'com.google.android.libraries.places:places:2.4.0'
```

   Be sure the version number (2.4.0) above matches the version number of the other play services libraries included in your project. Otherwise, you can use the latest version as suggested by the online documentation site.

2. Obtain an API key by visiting the following URL

   `https://developers.google.com/places/android-sdk/get-api-key`

   The API key is somehow tied to your app package name. Be sure to type the package name correctly when obtaining an API key.

3. Update the program to use the Google Places API calls

When your API key is setup correctly, it will show up in your Google cloud console such as shown in Figure 7.4.

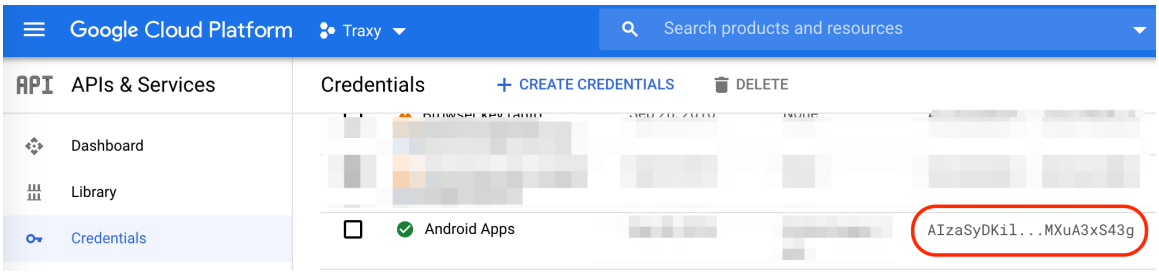


Figure 7.4: Google Cloud Console dashboard

Adding Places API Calls

Upon completion of the above preparatory steps, you can now modify your program to make the actual calls. The steps included below can also be found at Place API Getting Started Guide. You are strongly encouraged to refer to the online documentation, because it is updated more frequently than this manuscript.

Android apps that access the Google Places APIs must first initializes the API by invoking `Places.initialize()` prior to first use. You can place the initialization code in the `onCreate()` of `TraxyApp.kt`.

Kotlin

```
1 Places.initialize(getApplicationContext(), BuildConfig.MY_API_KEY)
```

> **NOTE**
>
> If you see an error because `MY_API_KEY` is undefined, try to clean and rebuild the project (Build > Clean Project and Build > Rebuild Project)

After the API client is initialized, you can now create a new screen for entering new journal information. Create a new fragment `NewJournalFragment` with the UI layout as shown in Figure 7.5. The widgets used are described as follows.

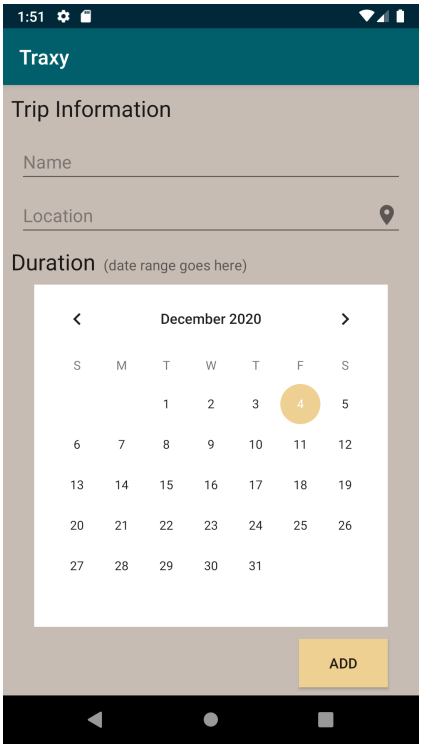


Figure 7.5: New Journal Screen

- The name input (`trip_name`) is just an ordinary `EditText` that your user will type the journal name in.
- The location input (`trip_location`) is also an `EditText` but will be used a bit differently. The user will not type the place name manually but it would be set to the search result from the Places API. We have to set its `focusable` attribute to false so clicking on it will not bring up the Android keyboard. Instead of the keyboard, when we tap this field we'll launch the `AutocompleteActivity` provided by the Places API.
- The location icon displayed by the location field is a vector graphics attached as the `drawableEnd` of the location `EditText`.
- The `TextView` with the text "(date range goes here)" will be used to display the journal's start and end dates as selected with the calendar view below. The id can be set to `trip_duration`.
- A `CalendarView` (`trip_calendar`) to allow the user selects the trip start/end dates.
- A `Button` (`trip_add`) to allow user to confirm adding new journal details. This button will be disabled by default by setting its `enabled` attribute to `false` in the Layout Editor. It will only be enabled when we have valid data in each of the required fields. This will ensure that we never add a new journal without valid data in each of the fields.

Transitioning To `NewJournalFragment` From `MainFragment`

Before you flesh out the details of `NewJournalFragment`, we need to update our navigation graph to transition to this new fragment from the `MainFragment`. First, recall that we left the image on the floating action button (FAB) of the `MainFragment` to be the default email icon. Go ahead and create a new vector asset in the res/drawable directory by searching for "add" in the Clip Art library. Once the vector asset is created, go into your `fragment_main.xml` layout file and change the `srcCompat` attribute of the FAB to be the Add vector asset just added.

Next, open `res/navigation/nav_graph.xml` and tap the "New Destination" icon at the top of the nav graph tool bar then select `NewJournalFragment` to add it to the navigation graph. To add a transition from `mainFragment` to `NewJournalFragment`, drag the mouse from the handle on the right edge of `mainFragment` into `NewJournalFramgment`. In the action attributes panel set its id to `action_new_journal`.

Finally, attach a listener to the `MainFragment`'s FAB to cause navigation to the `NewJournalFragment` whenever the FAB is tapped. This can be accomplished by adding the following code to the end of the `onViewCreated` method of `MainFragment`. Note that you might have to replace `fab` with whatever id you gave the FAB in your layout definition.

Kotlin

```
// in onViewCreated of MainFragment.kt
fab.setOnClickListener {
  findNavController().navigate(R.id.action_new_journal)
}
```

Now let's take a look at the implementation details of the `NewJournalFragment`.

Using the Places API

Override the `onViewCreated` method setup a `onClickListener` location input. There are three different ways of using the Places Autocomplete API in your program.

1. Invoke the service without using any UI widget

2. Embed the Autocomplete widget in a Fragment

3. Invoke the Autocomplete widget using an Intent

In what follows, we will implement the third option. In Chapter 4.4.3 you use `Intent` and `startActivity()` to transition to a new activity. We will use a similar technique with an added feature. When the Place Autocomplete activity (the "callee") terminates, it will *return a result object* to your `NewJournalFragment` (the "caller"). To capture this result, the caller must:

- Call the Places Autocomplete (your target activity) using `startActivityForResult()`. This should be called from the click handler of the location text field.

Kotlin
```
// in onViewCreated of NewJournalFragment
trip_location.setOnClickListener {
  val placeIntent = Autocomplete.IntentBuilder(
    AutocompleteActivityMode.FULLSCREEN,
    listOf<Place.Field>(Place.Field.ID,
                        Place.Field.NAME,
                        Place.Field.ADDRESS,
                        Place.Field.LAT_LNG))
    .setTypeFilter(TypeFilter.ADDRESS)
    .build(requireActivity())
  startActivityForResult(placeIntent, PLACE_REQUEST_CODE)
}
```

The snippet above shows an example of retrieving the place ID, name, and (postal) address from the Places API. If you need to retrieve different field details, adjust the field list accordingly.

In a typical application a fragment/activity may use `startActivityForResult` several times but you can only override `onActivityResult` one time. To disambiguate result from different request, each call must use a unique "request code". Hence the constant `PLACE_REQUEST_CODE` use in code. You can define `PLACE_REQUEST_CODE` as a const in a companion object in your `NewJournalFragment`. The above code will be executed when the location field is tapped, so you'll want to add it to a listener on the field in the `onViewCreated` override.

- The caller (`NewJournalFragment`) must override `onActivityResult` to capture the returned object.

Kotlin
```
override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {
  if (requestCode == PLACE_REQUEST_CODE && resultCode == Activity.RESULT_OK) {
    data?.let {
        val place = Autocomplete.getPlaceFromIntent(it)
        trip_location.text.clear();
        trip_location.text.insert(0, place.name)
    }
  } else
    super.onActivityResult(requestCode, resultCode, data)
}
```

Figure 7.6 shows the Places Autocomplete screen as the user types "disne" (the last 'y' is intentionally excluded). The `Place` class provides a number of methods that may be of interest to you: `getId()`, `getName()`, `getAddress()`, `getLatLng()`.

To understand the interaction between your fragment and the Places Activity Widget:

- Place a debugging breakpoint in the click handler of the location input
- Place another breakpoint inside `onActivityResult`
- Run the app in debug mode and observe the sequence of flow from your fragment to the Places activity and back to your fragment

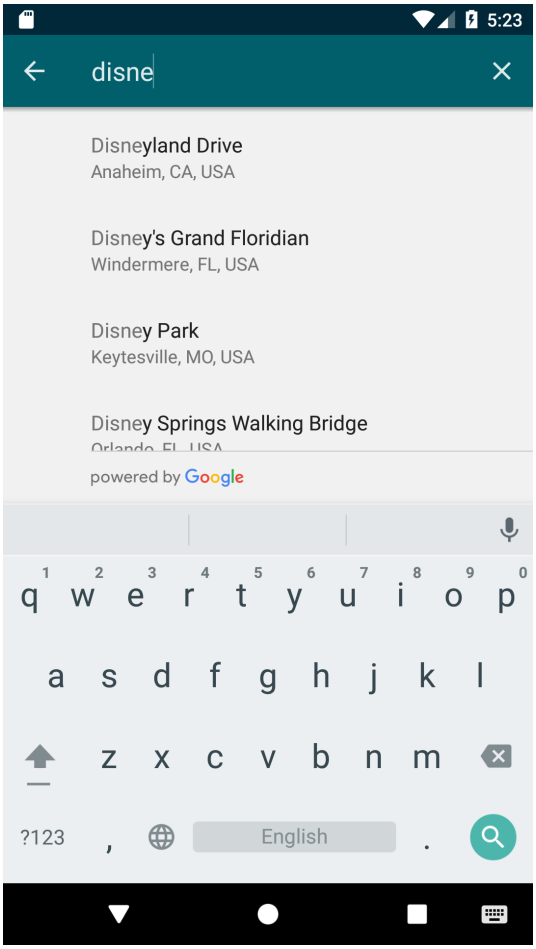


Figure 7.6: Places Autocomplete: after user typed “disne”

To capture the start/end date selections, setup a `onDateChangeListener` for the CalendarView. The listener function takes four arguments and the last three are integers representing the selected date.

- The year is a 4-digit value
- The day is in the range 1-31
- The month is in the range 0-11 (which is inconsistent with other fields. Apparently, this follows the convention used by `java.util.Date`) which differs from JodaTime convention. To convert to a date format understood by JodaTime, we have to add 1 to the month number. Hence the `${mo+1}` in the following format string:

Kotlin
```
// in onViewCreated of NewJournalFragment
trip_calendar.setOnDateChangeListener { view, yr, mo, dy ->
  val d:DateTime = inputFormatter.parseDateTime("$yr/${mo + 1}/$dy")
  // more code goes here (explained below)
}
```

To convert the integers to Joda DateTime, you would use its parse function from the `DateTimeFormatter` class:

Kotlin
```
// defined as properties on the NewJournalFragment class:
private lateinit var inputFormatter: DateTimeFormatter
private lateinit var outputFormatter: DateTimeFormatter

// initialized in onCreate as follows:
inputFormatter = DateTimeFormat.forPattern("yyyy/M/d")
outputFormatter = DateTimeFormat.forPattern("yyyy-MMM-dd")
```

In order not to clutter the UI, we will use the one instance of `CalendarView` for entering both the start and end date. This technique requires extra bookkeeping in code by managing additional state information as properties in our `NewJournalFragment` class. We'll parse the start and end dates sequentially and initialize both date fields to null. Add the following properties to the class:

Kotlin
```
private var startDate:DateTime? = null  // to hold the start data
private var endDate:DateTime? = null    // to hold the end date
private var isChoosingStartDate = true  // are we currently parsing the start date?
```

When `isChoosingStartDate` is initially false, the first date entry will stored to `startDate` and the boolean flag will be set to false. The next date entry will be assigned to `endDate` and the value of `isChoosingStartDate` toggled once again. We can also take the opportunity to do some simple validity checking of the data enter. That is, make sure the end date comes after the start date. Taking this information, we're now ready to put together our listener that listens for date changes on the calendar:

Kotlin
```
// in onViewCreated of NewJournalFragment
trip_calendar.setOnDateChangeListener { calendarView, yr, mo, dy ->
  val d:DateTime = inputFormatter.parseDateTime("$yr/${mo + 1}/$dy")
  trip_calendar.date = d.toDate().time
  if (isChoosingStartDate) {
    startDate = d
    isChoosingStartDate = !isChoosingStartDate
  } else {
    if (d.isAfter(startDate)) {
      endDate = d
      isChoosingStartDate = !isChoosingStartDate
    }
  }
  this.onFocusChange(null, false)
  trip_duration.text = dateRange()
}
```

Recall that we set the add button's `enabled` button to `false` when we created our layout. We can introduce logic to enabled once the valid data is available by overriding the `NewJournalFragment`'s `onFocusChanged` method:

Kotlin
```
 override fun onFocusChange(p0: View?, p1: Boolean) {
  add_button.isEnabled = trip_name.text.length > 0 &&
  trip_location.text.length > 0 &&
  startDate != null && endDate != null
}
```

The helper method `dateRange` simply formats the start and end dates into a format that we can bind to our `trip_curation` `TextView`:

Kotlin
```
private fun dateRange(): String {
  if (startDate == null) return ""
    val s = startDate!!.toString(outputFormatter)
  if (endDate == null) return "(\$s to ??)"
    val e = endDate!!.toString(outputFormatter)
  return "(\$s to \$e)"
}
```

Finally, in the click handler for the "Add" button, we need to implement the necessary code to add the new journal to your `ViewModel` and also pop the back stack to go back to the originating fragment `MainFragment`. We define an initialize our `viewModel` property as we learned in an earlier chapter:

Kotlin
```
// defined as a property on the NewJournalFragment class:
private lateinit var viewModel: UserDataViewModel

// initialized in NewJournalFragment's override of the onActivityCreated method:
viewModel = ViewModelProvider(requireActivity()).get(UserDataViewModel::class.java)
```

Now that we have a reference to our `viewModel` we can finish implementing the add button's listener as follows.

Kotlin

```kotlin
// in onViewCreated of NewJournalFragment
trip_add.setOnClickListener {
  // add new journal to the viewModel
  // Use Kotlin named parameters to avoid bugs of passing incorrect value
  // to the wrong property
  viewModel.addJournal(Journal(key = "key-???",
    name = trip_name.text.toString(),
    address = trip_location.text.toString(),
    startDate = startDate!!,
    endDate = endDate!!))

  // pop back to the MainFragment
  findNavController().popBackStack()
}
```

At this point, the new journal feature has been completely implemented. Go ahead and build/run the app and confirm that on the `MainFragment` screen you see the new journal just added.

7.3.4 Resolving Version Conflicts

A third party library included in your build may depend on one or more external libraries. This may trigger a build error when two such libraries depend on the external libraries which are not compatible with each other. Figure 7.7 shows a screenshot of an actual error message when two libraries depend on Android support libraries of different versions (25.0.1 and 23.4.0) that should not be used together. Unfortunately, the error message does not provide more details about which third-party libraries include the incompatible dependencies.

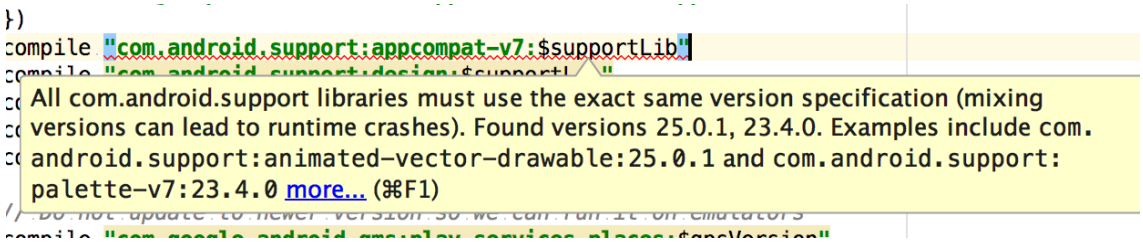


Figure 7.7: Error message due to conflicting external dependencies

The common fix to the above problem is to include only the newer version and exclude the older version (`pallette-v7:23.4.0` in the above example). Gradle comes handy in solving this problem. To show the library dependency graph, you can run a Gradle task as shown in Figure 7.8. The output of the selected task are shown in two separate panels: "Messages Gradle Build" and "Gradle Console", and you should inspect the later as shown in Figure 7.9.

Using the search box at the top of Gradle Console you can immediately locate the Android support library `palette-v7:23.4.0` and the library that depends on it (the output is too long to include as a screenshot here). For our case, the culprit is the `firebase-ui-storage:0.6.0` that includes an older version of the support pallette library. To exclude this contending library, you must change the dependency declaration in the app's `build.gradle` to explicitly exclude it:

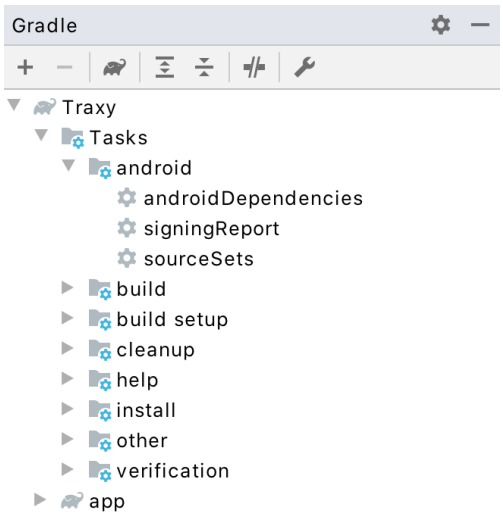


Figure 7.8: A list of tasks in a Gradle Project

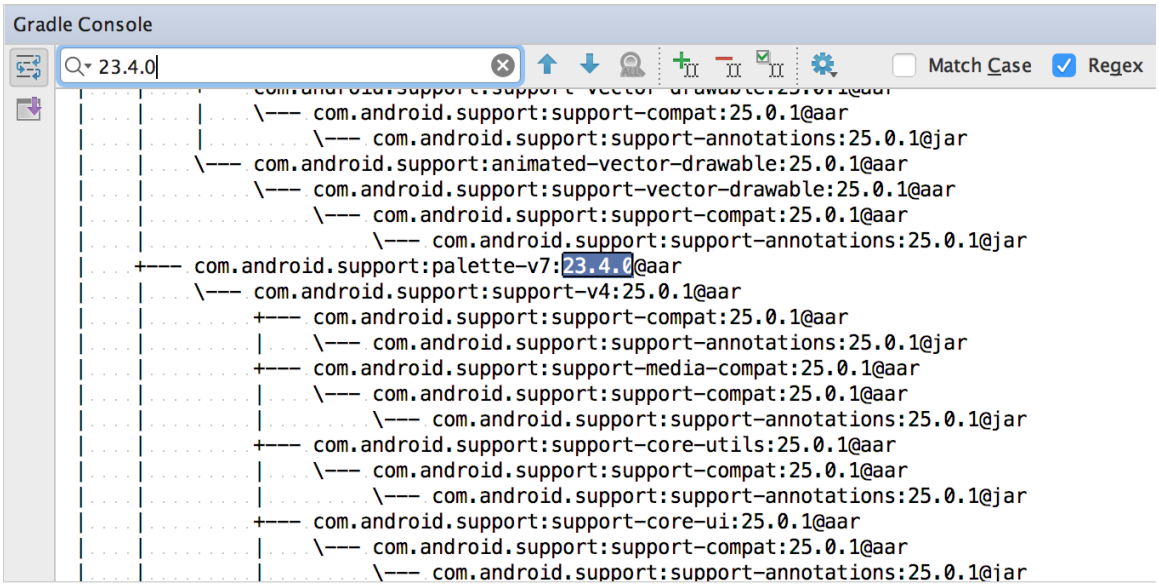


Figure 7.9: Gradle Console showing the output of Gradle Android Dependencies

Gradle

```
implementation ('com.firebaseui:firebase-ui-storage:0.6.0', {
    exclude group: 'com.android.support', module: 'palette-v7'
  })
```

—Data is a precious thing and will last longer than the systems themselves.

Tim Berners-Lee

8 Cloud Datastore Integration

Chapter Objectives

After reading this chapter you should be able to

- Understand the important role a cloud backend plays in mobile apps.
- Know how to integrate Google Firebase Authentication services to manage user identities.
- Know how to integrate Google Firebase Real-time Database services.

8.1 Mobile Backend as a Service (MBaaS)

The emergence of user friendly, touch-enabled smart phones along with the launch of vibrant native application ecosystems such as Apple iTunes App Store and the Google Play Store has created a significant demand for mobile applications. The success of blockbuster apps such as Instagram and Angry Birds has inspired many talented developers to create the app of their dreams. One common trait that almost every successful mobile app to-date shares is that they are all integrated with a highly scalable cloud-based data store. For the first several years (approximately 2008–2011) this requirement was a significant barrier to entry for many aspiring app developers. While there were at the time dozens of different viable cloud platforms that could serve as mobile backends, the developer needed to learn yet another non-trivial software stack in order to build out the custom datastore and accompanying web APIs required by their app. This requirement of course, was in addition to the already non-trivial learning curve involved in learning the iOS and Android SDKs.

Further complicating matters, was the fact that a great deal of the application innovation was taking place among entrepreneurial end users who often had little programming expertise, and even less budget to hire professional software developers to implement the apps they were envisioning. App aspirations far outpaced the development community's ability to deliver. Hence, it became apparent to a number of enterprising technologists, that there was an opportunity lurking here. That is, how could one make it relatively pain free and inexpensive to provide a generic cloud-based data store that could meet the requirements of almost any mobile app?

The multitude of solutions that have emerged since are now known as *mobile backend as a service* platforms, or MBaaS. MBaaS offerings today typically offer a variety of different common backend services for both native mobile and web apps. These services include:

- *Authentication / user management services*: The ability to onboard new users, and provide basic authentication and authorization services to existing users.
- *Push notification services*: Server initiated notifications to one or more users of the app. MBaaS solutions typically provide this in a native app OS agnostic manner, e.g. a single server push could result in notifications being delivered to a mixed population of devices running Android or iOS.
- *Datastore Services*: Typically a shared cloud-based NoSQL datastore that can persist app specific data.
- *Social Media Integration Services*: Services that make it easy to integrate with existing popular consumer-facing web apps, such as Facebook and Twitter. The focus of services in this area is often on providing a federated identity system, whereby users can utilize their credentials on these external applications to identify and manage their content on a third party app.
- *Application Analytics*: Let developers track the usage details of their application over time.

MBaaS offerings typically bundle these services into an easy to use downloadable SDK for each target platform, and offer a set of RESTful web APIs as well. Developers download the appropriate SDK and include it in their build. The SDKs provide a simple set of client-side APIs that make it easy to authenticate users and persist data, completely eliminating the need to do server-side development. The bottom line is that *adopting a serverless MBaaS solution can greatly expedite the deployment of a mobile app and save a significant amount of development cost*.

An early player in this space, was the now defunct company known as Parse. Parse was a startup that launched in 2011. The firm quickly captured mindshare among mobile app developers and became a popular means for providing a cloud backend to a number of apps. Parse was in fact so successful, that the firm garnered the attention of the social media giant Facebook, who eventually acquired the company in 2013 for $85 million dollars. All was well, until early 2016 when Facebook abruptly announced in a blog post that they were discontinuing the service. The service was officially shutdown in January 2017.

Since Parse was based on a freemium monetization model (e.g. developers could use the service for free until certain quotas were reached, at which time a billing relationship must be established), Facebook's decision to shut down the service caused no small amount of consternation and hardship among legions of small independent app developers and some larger companies as well. Fortunately, Facebook had the prudence to open source Parse which softened the blow somewhat. A number of companies, including Google and Microsoft made if relatively straightforward to host a Parse instance on their cloud backends. Nevertheless, an important take-away

from this history is that *developers need to do their homework well, and select an MBaaS platform that is likely to enjoy longevity in the marketplace*. No matter what choice is made, the downside of using an MBaaS platform is that you've created a fairly significant dependency on a third party provider and you'd best have a "plan B" in place if suddenly you discover it is not "business as usual"!

Taking this all into account, we've selected Google's Firebase offering as our example MBaaS solution in this chapter. Like Parse, Firebase began as a startup company in 2011, and was eventually bought by Google in 2014. Google made a number of significant enhancements to the service which were unveiled at their annual developer's conference (Google I/O) in 2016. Firebase is in many ways very similar to Parse. It provides easy to use services in all the areas enumerated above, and is straightforward to integrate into both iOS and Android. In terms of service longevity, given the prominence Google gave Firebase at Google I/O and the recent investment they've made in enhancing the service, all indicators are that it will be around for a while. Furthermore, Firebase is also based on a freemium monetization model that makes it very inexpensive for smaller independent developers to get started.

In our Traxy app, we're going to utilize Firebase's authentication services to onboard new users and authenticate existing users. We will also use Firebase's Cloud Firestore a real-time document database to persist our journal data.

8.1.1 Creating a New Firebase Project

Figure 8.1 shows the main Firebase screen after you login using your Google account. You can either create a new project or import an existing Google project. The latter applies when you had an existing Google project that is being updated to use Firebase. In this chapter, you are creating a new project.

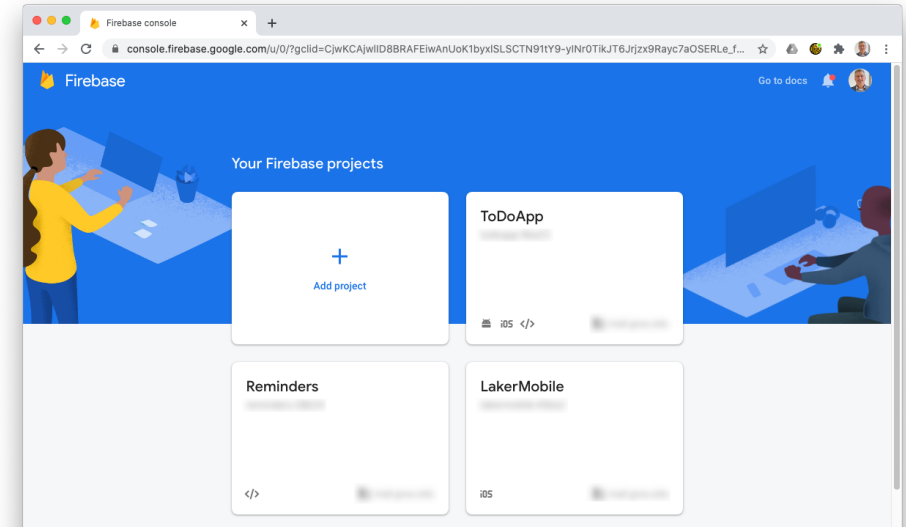


Figure 8.1: Firebase Main Screen

After the details of the new project are entered (project name and geographical region), Firebase

shows a dashboard which is organized into the following task groups:

- Authentication: manage user accounts, configure sign-in methods, and customize email templates.
- Cloud Firestore: manage document-based NoSQL cloud storage, setup access rules, etc.
- Realtime Database: browse the database, setup access rules, and show usage statistics. This style database though available has been replaced by the Firestore database.
- Storage: browse the storage associated with the database. Firebase storage is a repository for storing binary data which cannot be stored directly in the database.
- Hosting: configure and deploy a web app that accesses Firebase database.
- Functions: deploy functions to the cloud.
- Machine Learning: a test farm for your Android apps.

In this chapter, you will be adding Firebase to both your iOS and Android version of the Traxy app, but let's first understand how your Firebase data is secured.

8.1.2 Firebase Authentication

When a new project is created, Firebase automatically sets the access rules for your database to be *production mode* which results in the following JSON expression:

```
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {
    match /{document=**} {
      allow read, write: if false;
    }
  }
}
```

Listing 56: Firebase Database Default Access Rules

which essentially requires authenticated users to read or write the database. During debugging and development, you can eliminate these restrictions by allowing *anyone* to access your database and *temporarily* change the access rules *production mode* which results in the following JSON expression:

```
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {
    match /{document=**} {
      allow read, write: if
      request.time < timestamp.date(2020, 11, 11);
    }
  }
}
```

Firebase provides six different options for enforcing user authentication within your app. These options essentially fall under the following three categories:

- Authenticate using email and password managed directly by Firebase. As an administrator of a project, you can manage these users from the dashboard.
- Authenticate third-party accounts from Google, Facebook, Twitter, and many other options.
- Anonymous users.

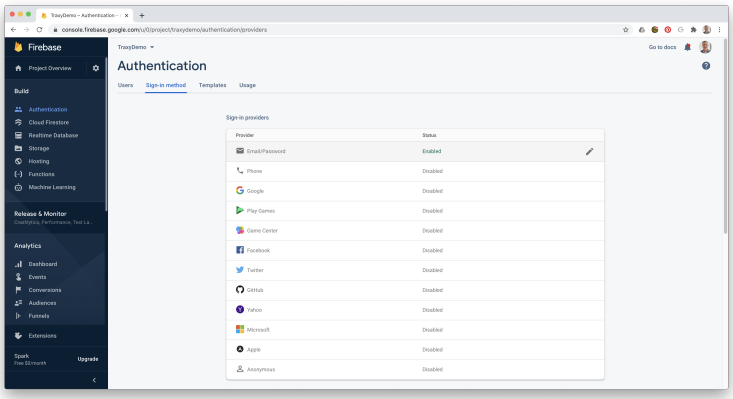


Figure 8.2: Firebase Authentication Options

All these user authentication options are initially disabled. You must manually opt them in by changing its status to Enable in Authentication > Sign-In Method (Figure 8.2). In this chapter you will learn how to authenticate Firebase users using email/password. Be sure to enable the Email/Password option before moving on to the next step below. Authentication using third-party accounts requires additional setup steps on the respective account provider.

8.1.3 Firebase Realtime Database

The logical view used by Firebase Realtime Database (RTDB) in managing its data imposes some limitations and capabilities that affect how developers organize their data. Traditional relational databases store user data into separate records and tables. Relationships between records from two different tables are indicated by common keys in the records. The Firebase RTDB uses a different approach. Instead of storing user data into separate tables, all data are stored in a giant JSON tree. Relationships among data items are inferred from the parent-child relationships among the tree nodes. Having a solid understanding on the above logical view helps you in making decisions on the best data organization for your app. Firebase's RTDB is being replaced by the more recently introduced Cloud Firestore, but for the sake of completeness we briefly discuss it here.

Recall that a JSON expression comprises one or more key-value pairs. Each value itself can be another JSON object. Applying this definition *ad infinitum* allows you to create a JSON tree of any depth. Based on this model, Firebase provides APIs for adding key-value pairs into a JSON tree by either using a key auto generated by Firebase or supplying your own key.

- The Android `push()` or iOS `childByAutoId()` method inserts a new key-value entry as a child of the caller. The method also auto generates the key of the new entry. The auto generated keys are *guaranteed to be unique*. The value of this entry remains uninitialized unless you invoke `setValue()` (described below)
- The `setValue()` method fills in the corresponding value associated with an *existing* key, i.e. at the time `setValue()` is invoked, the key for the object in question must have been set

Consider the following Android snippet:

Kotlin

```
1 val parentRef:DatabaseReference = Firebase.database;
2
3 parentRef.child("author").setValue("Don Knuth");
4 parentRef.push().setValue("Don Knuth");
```

Line 4 creates a child node of `parentRef` with key "author" and value "Don Knuth". Line 5 creates a child node with the same value but its key is auto generated by Firebase.

Once you understand the parent-child relationships and the capabilities of the Firebase APIs, let's consider the design of Firebase JSON tree for Traxy. Using a *pure* JSON expression, the desired structure for storing journal items is to group them into separate arrays, and associate each array with the userid as its key (Table 8.1a). However, using the Firebase APIs, such a *pure* representation using arrays is not currently possible. As an alternative, each object in the JSON array is replaced with a key-value pair (Table 8.1b) where each key is auto generated by Firebase. The top-level keys ("user1", "user2") are the string representation returned by Android `FirebaseUser.getUid()` or iOS `FIRUserInfo.uid`. Lines 3,4,7, and 8 in Table 8.1 are individual journal entries.

```
1  {
2    "user1": [
3    { ... },
4    { ... }
5    ],
6    "user2": [
7    { ... },
8    { ... }
9    ]
10 }
```

(a)

```
1  {
2    "user1": {
3    "key11": { ... },
4    "key12": { ... }
5    },
6    "user2": {
7    "key21": { ... },
8    "key22": { ... }
9    }
10 }
```

(b)

Table 8.1: JSON Representations of Trip Journal

A sample of the actual JSON representation is shown in Figure 8.3. In the figure, the key that begins with `M0a5E` is the uid of the current authenticated user and the two keys that begin with `KXhD` immediately below it are auto generated for each corresponding journal (where "Tropical Island" trip to "Maui, Hawaii" is currently expanded)

8.1.4 Firebase Listeners

Compared to traditional relational databases, modern cloud databases provides a mechanism for the client to get notifications when the data changes due to insertions, updates, or removals. To

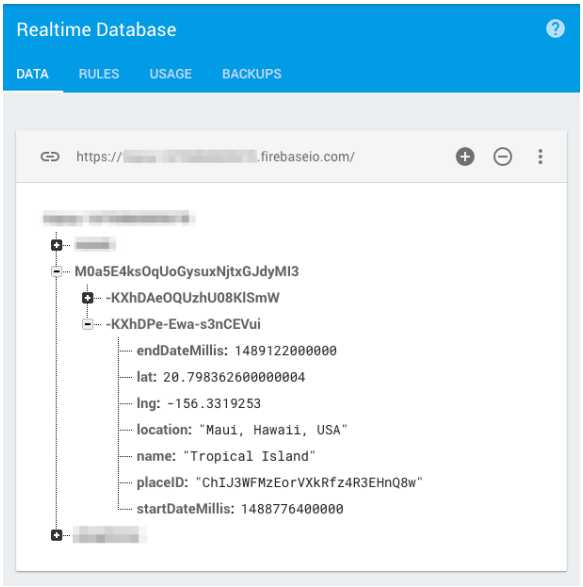


Figure 8.3: Firebase Realtime Database

detect changes in a traditional database, a DB client must run a new query and compare the new result with a previously cached result.

In Firebase, notifications of data changes are delivered to clients as one of the following events:

- Value Event: delivered on initial read of a JSON node or when any part of the JSON subtree changes
- Child Added Event: delivered when a new child is added to a JSON node
- Child Changed Event: delivered when a child of a JSON node is updated
- Child Removed Event: delivered when a child of a JSON node is removed

8.1.5 Firebase Cloud Firestore

In 2017 Firebase launched another alternative, Firestore, for storing your data in the cloud. In Firestore, your data are organized into collections and documents. For readers with strong background in relational databases, you can think a collection as a traditional table and a document as a tradition row in a table. However, this analogy is **not** really accurate for several reasons:

1. A row in a traditional DB has has a fixed structure and can only be modified by redefining the DB schema. Documents in Firebase are unstructured and do not have to follow a specific "template"
2. Similar to traditional databases where a table contains many rows, a collection contains many documents. While in traditional databases, the relationship stops there, Firestore al-

lows us to build a hierarchy of collections and documents ad infinitum constrained by the following rules:

- A collection holds a number of documents
- A document may hold a reference to a (sub)collection
- A collection cannot hod a reference to a (sub)collection
- A document cannot hold a reference to a (sub)document

In the remainder of this chapter we will look at how to integrate our Traxy app with the Firebase Cloud Firestore. Throughout the rest of the book we will use the term "Firestore" to refer to "(Firebase) Cloud Firestore".

8.2 Integrating Firebase with iOS

Now that you've created a Firebase project in the cloud, to add Firebase to your iOS Traxy app, go ahead and click on the "Cloud Firestore" link on the left nav bar, and create a new Firestore database for your project. Once the database is created, click on the "Project Overview" link at the top of the left nav menu, and then tap on the "Add app" option. Tap on the iOS button and proceed to fill out the form (as shown in Figure **??**). The Firebase web app will walk you through a five step process:

1. *Register app*: The only field that needs to be filled in at this point is our iOS app's bundle identifier. For our app we've set that to `edu.gvsu.cis.TraxyApp`. The other two fields are optional and can be left blank.

2. *Download the config file*: An XML file named GoogleService-Info.plist is generated and downloaded. Drag and copy this file into your Xcode project explorer. This file contains configuration information that associates your iOS app with this particular Firebase app instance in the cloud.

3. *Add Firebase SDK*: Since you know you will be using both the authentication and the database services, go ahead and add the `Firebase/Core`, `Firebase/Auth`, `Firebase/Firestore` and `Firebase/Storage` pods to your `Podfile`. You won't need the `Firebase/Storage` pod until the next chapter, but add it now so you have it. Don't forget to run the `pod install` command once you've updated the `Podfile`.

4. *Add initialization code*: In the `AppDelegate.swift` source file add an `import Firebase` statement at the top of the file, and within the `application:didFinishLaunchingWithOption` method add a call to `FIRApp.configure()` method. You will also need to import the Firebase module at the top of the file to make this method available.

5. *Run your app to verify installation*: Once you've successfully completed all of the above steps and build/run your app for the first time, you can validate that your app successfully connected to the Firebase instance you've created.

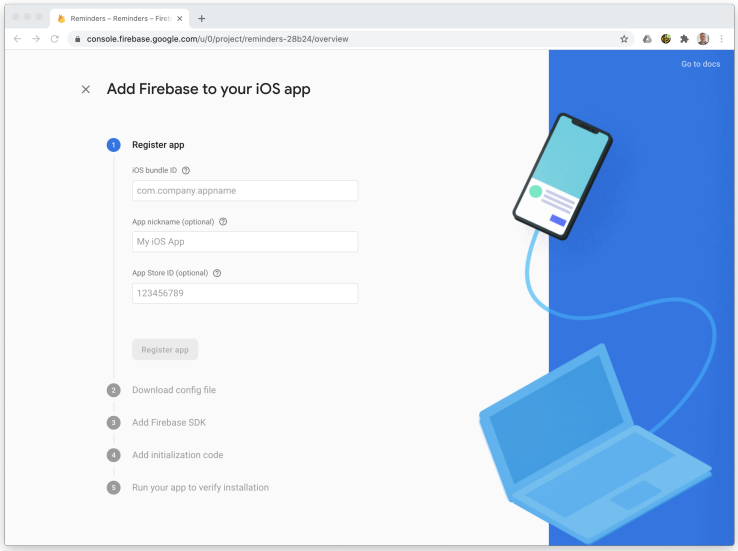


Figure 8.4: Adding an iOS app to a Firebase app instance.

8.2.1 Using Firebase Authentication for User Management

Firebase provides a complete solution to user management that allows to create new user identities native to the Firebase app instance. It also allows you to federate identities from popular websites like Facebook and Twitter. For now, you will simply focus on the former.

Creating the `TraxyRepository` Singleton

We're going to keep all of our Firebase related code in a single singleton class called `TraxyRepository`. This is simply practicing *separation of concerns*. Not only will this make it easier to maintain our code over time, but it will also make it easier to forklift out Firebase and replace with another cloud store if we choose to do so in the future. We've implemented the authentication related methods of the class as shown in Listing 57, with the key methods stubbed out from now.

A single static property named `instance` will hold the reference to our singleton instance of the class, and client code can call the `getInstance()` method to retrieve it. The initializer creates a new `Firestore` object that will be utilized by future Firebase related calls.

Note that many of the calls on this class will involve communication with the Firebase backend. These network calls can take time and hence instead of waiting around for the results, will be passed a closure expression that can be evaluated once the operation completes.

Onboarding New Users With Firebase

On the signup screen, when the signup button is pressed, assuming the fields validate, you need to attempt to create a new user with the provided email address and password. We can do this by invoking the `signUp` method defined by `TraxyRepository`. If the user is successfully created

Swift

```swift
class TraxyRepository {
  fileprivate static let instance =  TraxyRepository()
  fileprivate var db: Firestore!
  fileprivate var ref: DocumentReference?
  fileprivate var listener: ListenerRegistration?

  var userId: String?

  static func getInstance() -> TraxyRepository
  {
    return instance
  }

  init() {
    self.db = Firestore.firestore()
  }

  // MARK: Authentication Operations
  func signIn(email: String, password: String,
    onCompletion: @escaping  (Bool,String?) -> (Void))
  {


  }

  func signUp(email: String, password: String,
    onCompletion: @escaping  (Bool,String?) -> (Void))
  {
  }

  func logout()
  {
  }

  // MARK: Listener Operations
  func listenForAuthenticationChanges(
    onChange: @escaping  (String?) -> (Void))
  {
  }

  func listenForJournalUpdates(onChange: @escaping ([Journal]) -> Void) {
  }

  func stopListeningForUpdates() {
  }

  // MARK: Journal Operations
  func saveJournal(journal: Journal) {
  }
}
```

Listing 57: Stubbed out `TraxyRepository` singleton class.

by Firebase, a segue to the main screen is performed, as you now have a session with the newly created user. If the user was not successfully created the two password fields are cleared out, and focus is set to the first password field. The code for this is shown in Listing 58.

Swift

```
@IBAction func signupButtonPressed(_ sender: UIButton) {
 if self.validateFields() {
   let repo = TraxyRepository.getInstance()
   repo.signUp(email: self.emailField.text!, password: self.passwordField.text!)
   {(success, errorMesg) in
     if success {
       self.performSegue(withIdentifier: "segueToMainFromSignUp", sender: self)
     } else {
       if let msg = errorMesg {
         self.reportError(msg: msg)
       }
       self.passwordField.text = ""
       self.verifyPasswordField.text = ""
       self.passwordField.becomeFirstResponder()
     }
   }
 } else {
   self.passwordField.text = ""
   self.verifyPasswordField.text = ""
   self.passwordField.becomeFirstResponder()
   self.reportError(msg: self.validationErrors)
 }
}
```

Listing 58: Updated IBAction on the `SignUpViewController` to create a new user account using the `TraxyRepository` singleton.

Note that there are a variety of issues that could cause the user creation to fail. For example, if the email address was already used by an existing account on Firebase, or if the password does not muster Firebase's default strong password requirements. The error object returned in these situations provides us a human-friendly textual description of what went wrong that you can display in an error dialog to the user.

Since you will want to display error dialogs in other parts of the app as well, this code has been factored out into a method named `reportError` into the extension `UIViewController` you implemented earlier that defined your validation logic in the `UIViewController+Validation.swift` source file. The implementation of this method is shown in Listing 59. A UIAlertController displays modally on top of the current view (in this case the sign up view). In this case, you will wire up a simple button labeled "OK" which by default dismisses the alert view when pressed.

We can implement the `signUp` method on `TraxyRepository` by forwarding the sign up request to the Firebase SDK's `Auth` instance, and referencing the `onCompletion` closure expression passed, as shown in Listing 60.

You'll also want to cleanup the `SignUpViewController`'s validation logic a bit. Your current implementation simply prints out validation error messages to the Xcode console, but now

```swift
func reportError(msg: String) {
   let alert = UIAlertController(title: "Failed", message: msg, preferredStyle: .alert)
   alert.addAction(UIAlertAction(title: "OK", style: .default, handler: nil))
   self.present(alert, animated: true, completion: nil)
}
```

Listing 59: A method that displays an error dialog, given a message.

```swift
func signUp(email: String, password: String,
    onCompletion: @escaping  (Bool,String?) -> (Void))
{
  Auth.auth().createUser(withEmail: email, password: password ) { (user, error) in
    if let _ = user {
      onCompletion(true,nil)
    } else {
      onCompletion(false,error?.localizedDescription ?? "Unknown error.  Please try
      again later.")
    }
  }
}
```

Listing 60: The implementation of the `signUp` method of `TraxyRepository`.

that you have an error reporting dialog, you need a different approach. Define a string instance variable on the controller:

```swift
var validationErrors = ""
```

Next, re-implement the `validateFields()` method so it stores error messages in this instance variable instead of printing out the error to the console. The code for this is shown in Listing 61.

```swift
func validateFields() -> Bool {

  let pwOk = self.isEmptyOrNil(password: self.passwordField.text)
  if !pwOk {
    self.validationErrors += "Password cannot be blank. "
  }

  let pwMatch = self.passwordField.text == self.verifyPasswordField.text
  if !pwMatch {
    self.validationErrors += "Passwords do not match. "
  }

  let emailOk = self.isValidEmail(emailStr: self.emailField.text)
  if !emailOk {
    self.validationErrors += "Invalid email address."
  }

  return emailOk && pwOk && pwMatch
}
```

Listing 61: Modifying the validation logic so error messages are stored in a string instance variable.

Authenticating Existing Users in Firebase

The approach taken with the login screen is very similar, but instead of calling `signUp` on `TraxyRespository` we invoke the `signIn` method as shown in Listing 62.

We can implement the `signIn` method on `TraxyRepository` by forwarding the sign in request to the Firebase SDK's `Auth` instance, and referencing the `onCompletion` closure expression passed, as shown in Listing 63.

As before, if errors are encountered during the authentication process, a human friendly description is presented as an alert. You will need to make adjustments to your `validateFields()` method on `LoginViewController`, just as you did previously for the `SignUpViewController`.

Logging Off a Firebase User

We'll also provide the ability for a user to logout of the app once they've authenticated. Recall we implemented an unwind segue previously to handle the logout functionality of our Traxy app.

```swift
@IBAction func signupButtonPressed(_ sender: UIButton) {
  if self.validateFields() {
    print("Congratulations!  You entered correct values.")
    let repo = TraxyRepository.getInstance()
    repo.signIn(email: self.emailField.text!, password: self.passwordField.text!)
    {(success, errorMesg) in
      if success {
        self.performSegue(withIdentifier: "segueToMain", sender: self)
      } else {
        if let msg = errorMesg {
          self.reportError(msg: msg)
        }
        self.passwordField.text = ""
        self.passwordField.becomeFirstResponder()
      }
    }
  } else {
    self.reportError(msg: self.validationErrors)
  }
}
```

Listing 62: Modified IBAction in `LoginViewController` to implement user authentication via the `TraxyRepository` singleton.

```swift
func signIn(email: String, password: String,
  onCompletion: @escaping  (Bool,String?) -> (Void))
{
  Auth.auth().signIn(withEmail: email, password: password) { (user, error) in
    if let _ = user {
      onCompletion(true,nil)
    } else {
      onCompletion(false,error?.localizedDescription ?? "Unknown error.  Please try
      again later.")
    }
  }
}
```

Listing 63: The implementation of the `signIn` method of `TraxyRepository`.

We can update the `IBAction` in `LoginViewController` that handled logout with a call to the TraxyRespository singleton as shown in Listing 64.

Swift

```
@IBAction func logout(segue : UIStoryboardSegue) {
  let repo = TraxyRepository.getInstance()
  repo.logout()
  self.passwordField.text = ""
}
```

Listing 64: Updating the `logout()` method on the `LoginViewController` to logout via the `TraxyRepository` singleton.

Once again, the `logout` method on `TraxyRepository` can be implemented via the Firebase `Auth` object as shown in Listing 65 Since this simply clears the status of the session, it happens immediately, and no completion closure is needed. Note that the print statements are simply for debug purposes.

Swift

```
func logout()
{
  do {
    try Auth.auth().signOut()
    print("Logged out")
  } catch let signOutError as NSError {
    print ("Error signing out: %@", signOutError)
  }
}
```

Listing 65: The implementation of the `signIn` method of `TraxyRepository`.

8.2.2 Using Firebase Cloud Firestore as a Cloud Backend

Now that we're able to authenticate a user, we can persists that user's journals in the shared Firestore database in the cloud. We will use the user's identity to manage their own set of journals, but by having a shared cloud datastore, a number of compelling features are now available. A user could access their set of journals from multiple devices. For example, an iPad companion app could be written for our Traxy iPhone app, and the user could user either or both devices to access their journals. We could also eventually integrate features that allow users to share journals, and collaboratively author entries.

For now let's focus on how to store our journals in the Firestore database. Firestore allows us to store our journals in the cloud. In addition, when changes are made to journals, any *listening* clients can immediately receive the journal updates. Our implementation strategy will be as follows:

1. Handle any user level authorization changes. That is, in our main view, determine who the

current logged in user is, and make sure we logout immediately if we don't have a current logged in user, or if the user's authorization changes.

2. Add logic to save new journals to the Firestore database.

3. Add logic to the main view to receive and reload its tableview whenever the journal entries persisted in the Firestore for the current user change.

Handling user level authentication changes

Our `TraxyRepository` singleton provides a method `listenForAuthentication` changes that will listen for any state changes with regard to authenticated user sessions. Any time a new session is detected (e.g. whenever we login or signup for a new account) we want to automatically start listening for journal updates, by calling the `listenForJournalUpdates` method on the `TraxyRespository` singleton. This code can be placed in `MainViewController`'s `viewWillAppear` method, as this controller manages the scene that presents the list of journals to the user. The implementation of this method is shown in Listing 66.

Swift

```
override func viewWillAppear(_ animated: Bool) {
  super.viewWillAppear(animated)
  repo.listenForAuthenticationChanges { userId in
    if let uid = userId {
      self.userId = uid
      if self.userId != nil {
        self.repo.listenForJournalUpdates { (journals) in
          self.journals = journals
          self.sortIntoSections(journals: self.journals!)
        }
      }
    }
  }
}
```

Listing 66: Using the `TraxyRepository` singleton to listen for authentication updates, and listen for journal updates.

Note that since the `viewWillAppear` method retrieves the identity of the currently authenticated user directly from the Firebase SDK, `MainViewController` no longer needs the `emailField` instance variable we defined previously can be deleted. In addition, the `prepareSegue` methods in both the login and signup view controller that set this field can also be commented out.

The `listenForAuthenticationChanges` method of `TraxyRepository` can be implemented by calling the `addStateDidChangeListener` on the Firebase `Auth` object, as shown in Listing 67. When a new authentication takes place (e.g. userId is not nil) or when a user logs out (e.g. the userId is nil), the `onChange` closure expression passed by `MainViewController` is called.

The `listenForJournalUpdates` method of `TraxyRepository` can be implemented by calling the `addSnapshotListener` method on the Firebase collection holding our list of journals as shown in Listing 68. The task at hand is to listen for changes to the data of interest in

```swift
func listenForAuthenticationChanges(onChange: @escaping  (String?) -> (Void)) {
  Auth.auth().addStateDidChangeListener { auth, user in
    if let user = user {
      self.userId = user.uid
      self.ref = self.db.collection("user").document(user.uid)
      onChange(self.userId)
    } else {
      // No user is signed in.
      onChange( nil)
    }
  }
}
```

Listing 67: Implementation of the `TraxyRepository` singleton's `listForAuthenticationChanges` method.

the database, and execute a closure whenever changes are reported. Since Firestore models the data as a collection of journal documents, we can listen specifically to a user's journal collection for updates. In lines 2–4 we call the `addSnapshotListener` method on the user's journal collection.. Recall that in the previous section we stored an instance of type `Dictionary` in this collection, hence in line 4 we retrieve the collection of dictionaries from `snapshot` as an `Array` of `Dictionary`s that map `String`s to `AnyObject`s. In lines 8–21 we iterate through the array, retrieving each journal instance in the form of a `Dictionary` instance. We then append a `Journal` instance created from the individual attributes retrieved from the dictionary, and append it to `self.journals`, our array of `Journals`. Once the complete array of journal instances is processed, we pass it along to the `onChange` closure expression passed in by `MainViewController`.

Now that your app is loading updated data from the Firestore database in the cloud, don't forget to go into the JournalModel class that you created previously (see Listing 35) and comment out the `init()` method. Recall that this method created dummy journals for demonstration purposes.

The capability of listening for updates on our Firestore is a very powerful feature. We could actually have a plurality of devices listening to the same collection in the remote database receive any updates simultaneously. This is a very convenient way to implement a collaborative applications experience that involves an ensemble of physical devices.

Finally, whenever the main view controller scene disappears from the screen (either on logout, or eventually when we drill down into a journal's detail) we need to stop listening for updates to our journals on the cloud Firestore. This is accomplished by invoking the `stopListeningForUpdates` method on the `TraxyRepository` singleton within `MainViewController`'s `viewWillDisappear` method:

```swift
override func viewWillDisappear(_ animated: Bool) {
  super.viewWillDisappear(animated)
  self.repo.stopListeningForUpdates()
}
```

In the implementation of `stopListeningForUpdates` method the listener retained from any previous listen related call is removed as is shown in Listing 69.

```swift
func listenForJournalUpdates(onChange: @escaping ([Journal]) -> Void) {
  self.listener = self.ref?.collection("journals")
  .addSnapshotListener({ (snapshot, error) in
    guard let documents = snapshot?.documents else {
      return
    }
    var journals = [Journal]()
    for doc in documents {
      let key = doc.documentID
      let name : String? = doc["name"] as! String?
      let location : String?  = doc["address"] as! String?
      let startDateStr  = doc["startDate"] as! String?
      let endDateStr = doc["endDate"] as! String?
      let lat = doc["lat"] as! Double?
      let lng = doc["lng"] as! Double?
      let placeId = doc["placeId"] as! String?

      let journal = Journal(key: key, name: name, location: location,
      startDate: startDateStr?.dateFromISO8601, endDate:
      endDateStr?.dateFromISO8601, lat: lat, lng: lng, placeId: placeId)
      journals.append(journal)
    }
    onChange(journals)
  })
}
```

Listing 68: Implementation of the `TraxyRepository` singleton's `listenForJournalUpdates` method.

```swift
 func stopListeningForUpdates() {
  if let l = self.listener {
    l.remove()
  }
}
```

Listing 69: The implementation of the `stopListeningForUpdates` method of the `TraxyRepository` singleton.

Saving new journals to the Firebase database

Recall that in the previous chapter we introduced the `AddJournalViewController` to manage the new journal input form. When the save button was pressed, the handling of this event was delegated back to the `MainViewController` via the `AddJournalDelegate` protocol. The implementation of the `save` method simply passed the new journal along to the `saveJournal` method defined by the `TraxyRepository` singleton:

```swift
func save(journal: Journal) {
  self.repo.saveJournal(journal: journal)
}
```

The `saveJournal` method uses the Firebase SDK to persist the data in the Firestore in the appropriate location. In addition, since we will eventually introduce new features that allow us to edit existing journals, the method also handles updates to existing journals, as shown in Listing 70. Note that the implemention of this method depends on a private helper function `toDictionary` that converts the Journal object into a dictionary, a representation that can be handled by Firebase.

```swift
func saveJournal(journal: Journal) {
  if let r = self.ref {
    if let k = journal.key {
      let child : DocumentReference? = r.collection("journals").document(k)
      child?.setData(self.toDictionary(vals: journal))
    } else {
      r.collection("journals").addDocument(data: self.toDictionary(vals: journal))
    }
  }
}

fileprivate func toDictionary(vals: Journal) -> [String:Any] {
  return [
  "name": vals.name! as NSString,
  "address": vals.location! as NSString,
  "startDate" : NSString(string: (vals.startDate?.iso8601)!) ,
  "endDate": NSString(string: (vals.endDate?.iso8601)!),
  "lat" : NSNumber(value: vals.lat!),
  "lng" : NSNumber(value: vals.lng!),
  "placeId" : vals.placeId! as NSString
  ]
}
```

Listing 70: The `saveJournal` method defined by the `TraxyRespository` singleton utilizes the Firebase SDK to persist a newly added or existing journal. The method implementation depends on a private helper function `toDictionary` that converts a journal object into a dictionary.

Notice that the save method does not update the local in memory model, the `journals` instance variable. Neither does it force the table to reload as before. Instead, our previous call in `MainViewController`'s `viewWillAppear` to the `listenForJournalUpdates` method will asynchronously receive all updates and trigger the user interface to refresh as needed.

Handling Date Strings from Firebase

This last code depends on enhancements to the `Date` and `String` extensions you authored previously. The `dateFromISO8601` method on the `String` extension parses date strings in the format provided by by Firebase (e.g. the ISO8601 standard format) into a `Date` instance. The `iso8601` method on the `Date` extension creates an ISO8601 formatted date string from a `Date` instance. The implementation of these methods are shown in Listing 71.

Swift

```swift
extension Date {
  struct Formatter {
    static let iso8601: DateFormatter = {
      let formatter = DateFormatter()
      formatter.calendar = Calendar(identifier: .iso8601)
      formatter.dateFormat = "yyyy-MM-dd'T'HH:mm:ss.SSSxxx"
      return formatter
    }()

  var iso8601: String {
    return Formatter.iso8601.string(from: self)
  }

  // remaining details omitted ...
}

extension String {
  var dateFromISO8601: Date? {
    return Date.Formatter.iso8601.date(from: self)
  }

  // remaining details omitted ...

}
```

Listing 71: Enhancements to your Date and String extensions to handle ISO8601 format dates, e.g. the format Firebase returns date strings in.

At this point we've achieved our goal of persisting the data of our iOS Traxy app in the cloud. If you go ahead and run the app you can confirm this is the case by adding new journals and then going to the Firebase app in your web browser and browsing the data that is being created.

Let's now turn to our Android version of Traxy and extend it in a similar way.

8.3 Accessing Firebase in Android

In the following section you will be creating a new Firebase Android application on the Firebase web console for accessing your Firebase database. The dialog screen for a new Android application on Firebase web console prompts for the following information:

- Package Name: the name of the Java package of your Android app (`edu.gvsu.cis.traxy`)

- App nickname: a human readable text for easy identification of your app among your other Firebase apps

- Debug SHA1 certificate: the 40-digit hex string printed by the `keytool` command as described in Figure 7.3

Upon completion of the initial setup, Firebase generates a JSON file `google-services.json` containing the necessary details required by your app to access Firebase. The JSON file includes project-related information such as: project id, Firebase URL, API key, etc. When Android Studio compiles your app, its Gradle Google Services plugin loads this file and generates necessary string resources required by the Firebase classes for initializing your app to access the cloud data. The plugin requires this JSON file be saved in the same directory as your app's `build.gradle` (and not the project's `build.gradle`). Curious readers may inspect these generated string resources under the directory `app/build/generated/res/google-services`.

Let's first configure your project to use the Google Services plugin.

1. Add the classpath of the Google Services plugin to your *project's* `build.gradle`.

Gradle
```
// in top-level build.gradle
dependencies {
    classpath 'com.android.tools.build:gradle:4.1.1'
    classpath 'com.google.gms:google-services:4.3.4'
  }
```

   Android Studio will notify you when a never version of these libraries become available.

2. The Google Services plugin itself must be loaded into Gradle by adding the following command in your *app's* `build.gradle`

Gradle
```
// Add to the bottom of your app's build.gradle
apply plugin: 'com.google.gms.google-services'
```

To protect user's privacy in using the shared data cloud backend, our database must be designed to isolate user data. Using Firebase JSON tree structure, this data isolation is relatively easy to implement. By requiring Traxy users to authenticate, you can use the authenticated uid as the topmost node for storing all data owned by a particular user. To enable user authentication, you must include the following library in your app's `build.gradle`:

Gradle
```
implementation 'com.google.firebase:firebase-auth:20.0.1'
```

be sure the version number of the authentication library matches the version number of the other play-services libraries in your project.

The detailed steps for enabling authentication can be found at Firebase online documentation.

8.3.1 Repository

Following the guidelines recommended for using the Android Architecture Components, the UI classes are supposed to communicate only downwards to the ViewModel but not directly to the model classes. Similarly the ViewModel is supposed to communicate only downwards to the model classes. This implies that

- Only the View has a reference to the ViewModel classes and
- Only the ViewModel has a reference to the Model classes

The upward communication lines are implemented using the Observer pattern. The `(Mutable)LiveData` class provides the subscription mechanism needed to implement the observer pattern.

Since the journal data are considered to be part of the model classes, any logic that deals with accesses the Firebase backend should not be written in the UI classes. Moreover, between the ViewModel and the model backend, the MVVM architecture proposed by the Android team recommends an intermediate class, the repository which will handle the details of interfacing with the Firebase backend.

With the repository in between, the ViewModel should not have a reference to the Firebase backend, but only to the repository. Let's redesign our app using this approach for handling the login sequence to Firebase. Our strategy of implementation will be as follows:

- The email and password the user enters in the UI will be passed downwards to the ViewModel. This implies the ViewModel should provide a function that takes the email and password pair and pass them to the repository
- When the login sequence is verified, the UI should be able to observe the login verification result. Among the results returned by Firebase Authentication is the UID (string) associated with the user account. Let's make this UID observable to the UI. This implies the ViewModel should define `MutableLiveData<String>` to store the UID.
- As a "mediator" between the ViewModel and Firebase, the repository will in turn pass the the email and password pair from the ViewModel to Firebase. Moreover, the repository can wrap the *plain* UID string into an *observable* UID string. Specifically it transforms a `String` into `(Mutable)LiveData<String>`

Create a repository class `TraxyRepository.kt` and make the following edits:

1. Define an instance of Firebase Authentication class

Kotlin
```
class TraxyRepository {
  private val auth = FirebaseAuth.getInstance()
}
```

2. Add a function with the following signature

Kotlin
```
fun firebaseSignInWithEmail(email:String, password:String):
    MutableLiveData<String?> {
  // More code to write here
}
```

We do want to make the LiveData nullable to allow us place a `null` value when the email/password pair cannot be verified

Fill in the missing code for `firebaseSignInWithEmail`:

Kotlin
```kotlin
fun firebaseSignInWithEmail(email: String, password:String):
    MutableLiveData<String?> {
  val uidResponse = MutableLiveData<String?>()
  auth.signInWithEmailAndPassword(email,password).addOnCompleteListener {task ->
    if (task.isSuccessful)
      uidResponse.value = auth.currentUser?.uid
    else
      uidResponse.value = null
  }
  return uidResponse
}
```

To make the ViewModel work nicely with the underlying repository we have to make some changes in `UserDataViewModel.kt`:

1. Instantiate a repository object

Kotlin
```kotlin
val repo = TraxyRepository()
```

2. Change the following line

Kotlin
```kotlin
var _usr = MutableLiveData<String>()
```

to

Kotlin
```kotlin
lateinit var userId: MutableLiveData<String?>
```

3. Remove the `userId` getter

4. Add the following function

Kotlin
```kotlin
fun signInWithEmailAndPassword(email:String, password:String) {
  userId = repo.firebaseSignInWithEmail(email, password)
}
```

The next and final step is to change the login button handler logic in `LoginFragment` to call the correct ViewModel function. Change the following snippet

Kotlin
```kotlin
signin.setOnClickListener { v ->
  // code omitted
  when {
    // code omitted
    !passStr.contains("traxy") ->
      signin.startAnimation(shake)
    else -> {
      viewModel.userId.value = emailStr
      findNavController().navigate(R.id.action_login2main)
    }
  }
}
```

to

Kotlin
```
signin.setOnClickListener { v ->
  // code omitted
  when {
    // additional code omitted
    else -> {
      viewModel.signInWithEmailAndPassword(emailStr, passStr)
      viewModel.userId.observe(viewLifecycleOwner) { uid ->
        if (uid != null) {
            findNavController().navigate(R.id.action_login2main)
        } else {
            signin.startAnimation(shake)
        }
      }
    }
  }
}
```

Based on what you learn for signing in, apply the same technique to complete the logic for `SignupFragment`. Make similar changes in your ViewModel and repository class and be sure to use

Kotlin
```
auth.createUserWithEmailAndPassword(email,password)
```

in place of

Kotlin
```
auth.signInWithEmailAndPassword(email,password)
```

in your repository class.

Rerun the app, create/register a new account and verify that you are able to (re)login using the new account. In case you make mistakes in creating the new user, use the Firebase console to delete the unwanted user accounts (under Authentication > Users).

8.3.2 Logging Out

Handling the logout is trivial. Following the same pattern used above for signing in and signing up, you have to make the following changes:

1. Add a new method `firebaseSignOut()` in `TraxyRepository`

Kotlin
```
fun firebaseSignOut() {
  auth.signOut()
}
```

2. Add a new method in `UserDataViewModel`

Kotlin
```
fun signOut() {
  repo.firebaseSignOut()
  userId.value = null
}
```

3. Finally, call the ViewModel function from `MainFragment` in the method where you previously handle the "Logout" menu item:

Kotlin
```
// in MainFragment (when logout is pressed)
viewModel.signOut();
```

Having two separate calls for signing in/out in the app, there will be occasions where the app may terminate without the user pressing "Logout". Consider a case where the user runs Traxy, is interrupted by an incoming call, and never goes back to Traxy. A few minutes later the user decides to terminate all the recent apps (including Traxy). When Traxy is launched again, the user is still logged in. Ideally, the login screen should skip over to the main screen. This additional logic can be implemented by adding an observer to the mutable live data as shown in the code snippet below.

Kotlin
```
// in onViewCreated of LoginFragment
viewModel.userId.observe(viewLifecycleOwner, {
  findNavController().navigate(R.id.action_login2main)
})
```

This observer pattern is the most appropriate strategy to use to handle this case: as the view model data is updated **asynchronously** by the repository, the observer in the view class is able to take proper action. One particular detail that may need extra attention is managing the screen transitions among these three fragments: `LoginFragment`, `SignupFragment`, and `MainFragment`.

- Upon entry to the login screen an unauthenticated user should first fill in the userid and password. However, when the user is already authenticated, the app should skip the login screen altogether and transition to the main screen. This feature is implemented via the observer snippet described above

- On a successful new account sign up, the app should take the user to the main screen. However on a failure to sign up, the app should stay in the signup screen while showing an error message

Instead of explicitly initialing a transition from the signup screen to the main screen, the second bullet point above can be implemented by reusing the observer we added earlier in `LoginFragment`.

Kotlin
```
// In one of the "when" cases of the FAB click listener in SignupFragment.kt
// after the user mail and password match are verified
fab_add.setOnClickListener { v ->
  // additional code omitted
  when {
    // additional code omitted
    else -> {
      viewModel.signUpWithEmailAndPassword(emailStr, passStr)
      viewModel.userId.observe(viewLifecycleOwner) { uid ->
        if (uid != null) {
            findNavController().popBackStack()
        } else {
            // Use SnackBar to show error message
        }
      }
    }
  }
}
```

8.3.3 Storing Data Into Firebase Cloud Firestore

After Traxy users are able to authenticate using email/password, let's now update the app to write new journal entries to Cloud Firestore. The first and foremost step is to include the required library[1]:

Gradle
```
implementation 'com.google.firebase:firebase-firestore-ktx:21.4.0'
```

The `-ktx` suffix in the library name indicates that we are using the Kotlin extension version of the library. Among the benefits of using the Kotlin extension functions is conciseness of your code by using "shortcuts" to perform common tasks. For instance, without Kotlin extensions you have to use the following snippet to initialize your Firestore instance:

Kotlin
```
val dbStore = FirebaseFirestore.getInstance()
```

With Kotlin extension functions the above snippet is much shorter:

Kotlin
```
val dbStore = Firebase.firestore
```

Up to this point, your app is able to get a new Journal object that the user enters in `NewJournalFragment` and saved into a non-persisted list in the ViewModel. As mentioned in the introduction of this chapter, Firestore distinguishes collections from documents and it requires the topmost level be a collection and documents must be an immediate child of a collection. To allow deeper hierarchical structure, a document may contain a reference to a (sub)collection. For our application, the topmost collection would be a collection of users. Specifically each individual user is associated with a Document whose id is the user UID as returned by Firebase Authentication.

Storing a Java object in Cloud Firestore requires the following ingredients:

[1] At the time of this writing, the latest version was 21.6.0. However, using a version higher than 21.4.0 produced build errors due to conflicting version of the jetified library used by Firestore

- Obtain a reference to the collection where your journal entry will be stored.

- The class associated for the journal entry must provide getter methods for all the attributes. Fortunately, Kotlin data class automatically does this for us. However, to make the `Journal` class Firebase compatible we should change the Joda verb—DateTime— type to `String` and store the date/time in ISO8601 format. Conversion between the two types can be easily done using `toString()` and `parse()` as shown below:

Kotlin
```
val today = DateTime.now()
val todayISO:String = today.toString()  // in ISO8601 format
val now:DateTime = DateTime.parse(todayISO)
```

Changing the type of both `startDate` and `endDate` from `DateTime` to `String` would not let you use either `isBeforeNow` or `isAfterNow` for labelling each Journal entry as "past", "present". or "future" for building a partitioned list in `MainFragment`. The following snippet:

Kotlin
```
viewModel.journals.observe(this.viewLifecycleOwner, Observer {
  val partitioned = it.map {
    when {
      it.endDate.isBeforeNow -> "Past" to it
      it.startDate.isAfterNow -> "Future" to it
      else -> "Current" to it
    }
  }
  /* more code not shown */
})
```

should be updated to

Kotlin
```
viewModel.journals.observe(this.viewLifecycleOwner, Observer {
  val partitioned = it.map {
    val now = DateTime.now().toString()
    when {
      it.endDate < now -> "Past" to it
      it.startDate > now -> "Future" to it
      else -> "Current" to it
      }
  }
  /* more code not shown */
})
```

In order to prepare the code for additional features used throughout the book, we will replace the `location` property in `Journal.kt` by expanding it to more fields (address, lat, lng, placeId) that will get initialized by data from the Places API:

Kotlin
```
data class Journal(
  @DocumentId
  val key: String = "",
  val name: String = "",
  val address: String = "",
  val lat: Double = 0.0,
  val lng: Double = 0.0,
  val placeId: String = "",
  val startDate: String = "",
  val endDate: String = ""
) : ListItem()
```

- The annotation `@DocumentId` informs Firestore to automatically fill in the `key` field with the document ID associated with each journal document in Cloud Firestore.

- Firebase requires class used for "reading" data have default constructor. The default value initializers used in the data class above satisfy this requirement.

When we implemented user authentication in the previous section, we delegate the actual task of calling Firebase functions to the repository class; we will follow the same pattern for saving data into Cloud Firestore.

1. Make the following changes to the repository class:

   - Declare two new variables for holding a reference to the Firestore instance and also to the "top" document associated with the current user:

     Kotlin
     ```
     private val dbStore = Firebase.firestore
     private var docRef: DocumentReference? = null
     ```

   - In the completion listeners of `signInWithEmailAndPassword` and `createUserEmailAndPassword`, initialize the `docRef`:

     Kotlin
     ```
     if (task.isSuccessful) {
       auth.currentUser?.let {
         uidResponse.value = it.uid
         docRef = dbStore.collection("user").document(it.uid)
       }
     } else {
       uidResponse.value = null
     }
     ```

   - Add a new function to save a new Journal object to Firebase:

Kotlin
```
fun firebaseAddJournal(d:Journal) {
  val jData = hashMapOf(
    "name" to d.name,
    "address" to d.address,
    "placeId" to d.placeId,
    "lat" to d.lat,
    "lng" to d.lng,
    "startDate" to d.startDate,
    "endDate" to d.endDate
  )
  docRef?.collection("journals")?.let {
    it.add(jData)
  }
}
```

2. Remove the code snippet to add random Journal entries in the `init{}` block of the ViewModel. Since we will be using "real" data, we don't need to generate fake data anymore.

3. Change the `addJournal()` function in the ViewModel class to invoke the above repository function:

Kotlin
```
fun addJournal(z:Journal) {
  repo.firebaseAddJournal(z)
}
```

Before moving on to the next section on retrieving journal data from Firestore, we have a few loose ends to take care of. Currently, `NewJournalFragment` does not fully utilize the power of the view model class. For instance, start date and end date are UI data which are supposed to be stored in the view model class. Let's add more mutable live data to `UserDataViewModel` for holding other data related to the current trip:

Kotlin
```
val tripStart = MutableLiveData<DateTime>()
val tripEnd = MutableLiveData<DateTime>()
val tripPlace = MutableLiveData<Place>()
```

The `tripPlace` property will hold a copy of the data returned by the Google Places autocomplete activity. Insert the following line of code in the `onActivityResult` method of `NewJournalFragment` to store the returned place data in the view model:

Kotlin
```
viewModel.tripPlace.value = place
```

Remove the declaration of `startDate` and `endDate` in `NewJournalFragment` and replace any reference to `startDate` with `viewModel.tripStart.value`. Similarly, replace any reference to `endDate` with `viewModel.tripEnd.value`. For instance, in the code snippet for date change listener of `CalendarView` you should rewrite

Kotlin
```
startDate = d
```

as

Kotlin
```
viewModel.tripStart.value = d
```

With these new live data in place, the code snippet for adding a new journal to Cloud Firestore in the `NewJournalFragment` class can be written as follows:

Kotlin
```
add_button.setOnClickListener {
  val newData = Journal(name = trip_name.text.toString(),
      address = trip_location.text.toString(),
      startDate = viewModel.tripStart.value.toString(),
      endDate = viewModel.tripEnd.value.toString(),
      placeId = viewModel.tripPlace.value?.id ?: "NONE",
      lat = viewModel.tripPlace.value?.latLng?.latitude ?: 0.0,
      lng = viewModel.tripPlace.value?.latLng?.longitude ?: 0.0
  )
  viewModel.addJournal(newData)
  findNavController().popBackStack()
}
```

Finally, be sure to include the proper field names when initializing the intent for the Places API (e.g. add the field to include latitude and longitude in the results):

Kotlin
```
val placeIntent = Autocomplete.IntentBuilder(
  AutocompleteActivityMode.FULLSCREEN,
    listOf<Place.Field>(
      Place.Field.ID,
      Place.Field.NAME,
      Place.Field.ADDRESS,
      Place.Field.LAT_LNG)
        .build(requireActivity())
)
```

That's it. Your next step is to fix any compile errors caused by changes you implemented above to the `Journal` class. You'll notice a few changes required elsewhere that should be straightforward to correct. If not, consult with the book's sample code on GitHub. Once you have a clean build, go ahead and run the app. When you add a new journal entry, the app should insert the entry to Cloud Firestore. At the moment your app is not wired up to retrieve the data from Firestore yet. To confirm that a new data is indeed inserted, go to your Firebase console.

8.3.4 Retrieving Data From Cloud Firestore

Cloud Firestore does not provide a **direct** command (like `add()` above) to read data from the collection. Instead, "reading" data must be implemented through a listener declared using the `EventListener` class.

So how do we bridge the listener in the repository and the `LiveData` in the ViewModel class?

To solve this problem, we will design our own custom live data class. To allow the class be used for different user collection, we would pass a collection reference to its constructor.

Kotlin
```
class JournalLiveData(val topRef: CollectionReference):
    LiveData<List<Journal>>() {

  override fun onActive() {
    // invoked when the data referrer becomes active
    // such as Activity or Fragment onResume())
  }

  override fun onInactive() {
    // invoked when the data referrer becomes inactive
    // such as Activity or Fragment onPause())
  }
}
```

The main task of the live data class is to activate or deactivate a listener for changes to documents inside the target collection. Complete the design of the above class by making the following updates:

1. Define the collection listener

Kotlin
```
val docListener = EventListener<QuerySnapshot>() { snapShot, error ->
  if (error != null)
    return@EventListener
  snapShot?.let {
    val all = ArrayList<Journal>()
    it.documentChanges.forEach {
      val journal = it.document.toObject(Journal::class.java)
      all.add(journal)
    }
    postValue(all)
  }
}
```

2. Declare a field to hold the listener registration

Kotlin
```
lateinit var register: ListenerRegistration
```

3. Attach the document listener in `onActive()` and detach it in `onInactive()`

Kotlin
```
override fun onActive() {
  register = topRef.addSnapshotListener(docListener)
}

override fun onInactive() {
  register.remove()
}
```

To use the custom live data we have to make changes to the view model and repository classes. First, add the following property in the repository class:

```kotlin
val journalLiveData by lazy {
  val userId = auth.currentUser?.uid ?: "NONE"
  val coll = dbStore.collection("user/$userId/journals")
  JournalLiveData(coll)
}
```

In our current implementation we have been using Kotlin `lateinit` to postpone initialization of variables when we can't do it inside the class constructor. Unfortunately, since the login process is running asynchronously in the background there may be a possibility that our custom LiveData variable is not completely initialized when the main fragment is ready to observe it. One technique to handle this situation is to use Kotlin **lazy initialization** instead of `lateinit`.

> NOTE
>
> In Kotlin "lazy initialization" implies initialization on **first use**, which may take place much later than `lateinit`. In the extreme case when a lazy variable is never used, the initialization will never take place.

Then make the following changes to the view model class

1. Remove the `journal` getter and its backing variable `_journals`
2. Declare the live data property

```kotlin
lateinit var journals: LiveData<List<Journal>>
```

3. Initialize the live data inside `signInWithEmailAndPassword`

```kotlin
fun signInWithEmailAndPassword(email:String, password:String) {
  userId = repo.firebaseSignInWithEmail(email, password)
  journals = repo.journalLiveData
}
```

4. Repeat the above step for `signUpWithEmailAndPassword`
5. Replace the reference to `_journals` in the `getJournalByKey` method to `journals`:

```kotlin
fun getJournalByKey(key: String): Journal? = journals.value?.firstOrNull {
  print("Where are we ${it.key}")
  it.key == key
}
```

Rerun the app and verify that the Journal entries recorded in Firebase show up in main fragment. Congratulations! You have acquired the skill for incorporating a cloud datastore into your app.

8.3.5 Kotlin Coroutines for Firebase

As you continue to incorporate more co-routines into the app, this is perhaps the appropriate place to replace (asynchronous) callback functions with co-routines. The main benefits of switching from callbacks to co-routines are mainly code readability, conciseness, and easier thread management. It is easier to understand asynchronous code written in co-routines because they *look* synchronous and reading the code you don't have to remind yourself that some part of your code will later run out of order. The main purpose of having asynchronous callbacks is to allow the current program flow to continue without blocking while providing a block of code for handling results delivered in the future. It is the programmer's job to separate code into the **main** flow and the "side" flow (those that will run in callback blocks. Using co-routines, the programmer does not have to manually separate her code; the compiler does this separation on behalf of the programmer. The programmer's responsibility is now to "mark" which functions are potentially making blocking calls. In Kotlin, the keyword `suspend` marks such functions.

To check the result of Firebase Authentication functions we must provide a completion listener function (or lambda) and check if the task completed successfully or failed with an error. Using the co-routine library designed to work with asynchronous tasks used by Google Play Services, the asynchronous callback functions can be replaced using the `await()` extension function.

To use this library, add the following dependency to your app build.gradle file:

Gradle
```
implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-play-services:1.1.1'
```

Let's now rewrite our Firebase signin function in the repository class. The current code

Kotlin
```
// in TraxyRepository.kt
fun firebaseSignInWithEmail(email: String, password: String):
  MutableLiveData<String?> {
val uidResponse = MutableLiveData<String?>()
auth.signInWithEmailAndPassword(email, password)
  .addOnCompleteListener { task ->
    if (task.isSuccessful) {
      auth.currentUser?.let {
        uidResponse.value = it.uid
        docRef = dbStore.collection("user").document(it.uid)
      }
    } else
      uidResponse.value = null
  }
}
```

can be simplified to

Kotlin
```
// in TraxyRepository.kt
suspend fun firebaseSignInWithEmail(email: String, password: String): String? {
  try {
    val z = auth.signInWithEmailAndPassword(email, password).await()
    return z.user?.uid.also {
      docRef = dbStore.collection("user").document(it ?: "NONE")
    }
  } catch (e: Exception) {
    return null             // Unable to verify login
  }
}
```

In addition to reordering the statements, the two modifications are

1. Adding the `await()` call to the original Firebase Authentication function
2. Mark the function with the `suspend` keyword

At runtime, **after** the call to `signInWithEmailAndPassword` has been issued, the function `firebaseSignInWithEmail` is suspended and will be resumed later when the former function call completed its password verification.

The above modifications to the repository function affect all the callers of the above function. And this will percolate up following the call hierarchy (to the view model and to the view).

Your next step is to update the caller in the ViewModel by changing

Kotlin

```
fun signInWithEmailAndPassword(email: String, password: String) {
  userId = repo.firebaseSignInWithEmail(email, password)
  journals = repo.journalLiveData
}
```

to

Kotlin

```
suspend fun signInWithEmailAndPassword(email: String, password: String):
    String? {
  val u = repo.firebaseSignInWithEmail(email, password)
  userId.postValue(u)   // Must use postValue to update from a non-UI thread
  journals = repo.journalLiveData
  return u
}
```

After switching to using co-routines, the update to userId can be passed as a normal return value, even if that return value originates from an asynchronous function.

At the top of the call hierarchy, `LoginFragment` must call the above function within a co-routine context:

Kotlin

```
CoroutineScope(Dispatchers.IO).launch {
  val uid = viewModel.signInWithEmailAndPassword(emailStr, passStr)
  if (uid != null) {
    findNavController().navigate(R.id.action_login2main)
  } else {
    launch(Dispatchers.Main) {
      signin.startAnimation(shake)
    }
  }
}
```

It is important to call the shake animation from the main (UI) thread. Otherwise you will get a runtime error for an attempt to access the UI from a background thread. Using what you learned here, you should be able to update the signup functions (in the repository and view model) into a co-routine.

—The soul never thinks without a picture.
Aristotle

9 Working With Multimedia

Chapter Objectives

After reading this chapter you should be able to

- Use the Android/iOS media frameworks to capture and playback media files.
- Implement playback of captured audio/video.
- Use Android implicit `Intent`s to access Android media-related features.
- Use the iOS AVKit's `AVPlayerViewController`.
- Work with platform specific permissions schemes to access onboard multimedia and camera functionality.

9.1 Personal Media Players

You likely never carried the first model of the Sony Walkman™ around as your personal music player. By today's standards this cassette-based music player would be considered too bulky to carry around, but nevertheless was a very popular among teenagers in the 1980s. The recording capacity of the cassette was just enough to hold about 20 songs, recorded in analog format on a 3.8mm-wide tape (about $\frac{1}{8}$ inch). The Walkman product line later evolved into Sony Walkman MP3 players with storage capacity of 512MB, large enough to hold up to 500 minutes of music, depending on the bit rate of the recordings. In 2001, approximately 20 years later, Apple released its first generation iPod™. Shortly after its debut, Apple was shipping over a 100 million iPods a year (an unbelievable numbers in those days) and in the process completely redefined the music

industry. With storage capacity up to 120 times larger than that of Sony MP3 player, the iPod could hold thousands of song titles.

Modern mobile devices that users carry around today have far more storage capacity than those early portable players. In addition, mobile phones today include a camera as a standard option, even on inexpensive phones. Higher end models usually include both front and rear-facing cameras. They also have a larger screen, 64-bit processors, may also include a Graphics Processing Unit (GPU) for better graphics performance and a dedicated digital signal processing (DSP) chip. To top off that delightful set of features, the device will also have a high speed network connection that enables the device to *stream* media from remote sites. Modern mobile devices are first class multimedia devices with capabilities users could never of imagined ten to fifteen years ago.

When incorporating media processing capability in their apps, iOS developers don't have to deal with the broad product variances of devices like their Android developer counterparts. Android developers for instance, cannot assume that a front-facing camera is always available on a device.

This chapter covers topics related to multimedia handling. In particular, you will learn how to utilize the onboard camera(s) to capture still photos, and videos, how to persist and retrieve media locally and to Firebase, and how to play back previously captured media.

9.1.1 Codecs: Media Data Compressors

A raw 24-bit bitmap image of size 2560×1920 requires 14.4 megabytes of storage ($2560 \times 1920 \times 3$). The same image encoded in JPEG using 23:1 compression ratio requires only 641K bytes. Audio and video files can obviously benefit from similar data compression techniques, which are commonly known as *codecs* (coders and decoders). Most compression algorithm for media files are lossy compressors. To a compression algorithm, raw data from an image or audio file is treated as a linear (one-dimensional) stream of bytes. Despite the 2D visual appearance of an image, the location of a byte within an image does not affect how the byte is compressed. In addition to the spatial compression of the data bytes within a single video frame, a video compression algorithm also analyzes temporal changes of data bytes between frames.

Some well-known audio and video codecs are listed in Table 9.1.

Audio Codecs	Video Codecs
MPEG-Audio	MPEG-Video
Advanced Audio Codec (AAC)	H.263 (low bit-rate)
Ogg Vorbis	H.264 (Advanced Video Codec/AVC)
Adaptive Multi-Rate (AMR, intended for speech data, not for HiFi audio)	
FLAC (Free Lossless Audio Codec)	

Table 9.1: Audio and Video Codecs

Since your Traxy app is cross-platform and runs on both iOS and Android, care must be taken to select media formats that are easily handled on either device. For example, a user should be able to watch a video capture on iOS from an Android device.

9.1.2 The Need for Caching

Since you will be uploading and downloading media files in this chapter, there are some things you need to pay special attention to. Media files are much larger than the textual data you've downloaded from the Firebase database to-date. You want to avoid downloading excessive amounts of data for a variety of reasons. First, many users have limited data plans and will not take fondly to your application if it needlessly consumes their monthly data allowance. Furthermore, transmitting the data for upload or download requires running the device's radio (WiFi or wide-area) – an operation that consumes battery, which is another precious and very limited commodity on a resource-constrained device. Additionally, depending on the size of a media file, it can take time to download, and your users inevitably have short attention spans. Finally, in the Traxy app there is no provisions for editing media files after they've been submitted, so the files are not going to change once uploaded. For all these reasons, it makes perfect sense to *cache* media files on device once they are downloaded. In passing, this chapter will demonstrate some very simple techniques to accomplish media caching on both iOS and Android.

9.1.3 Privacy Concerns

When applications attempt to access the multimedia capabilities of a device whether it be the camera, microphone or media assets such as photos, videos or audio files persisted on the device, privacy concerns immediately come to the forefront. Users certainly do not want apps to access their private media and push it off to an unauthorized cloud data store somewhere for the world to see. There are other privacy risks as well. For example, users would definitely not appreciate an app that turned on the device's microphone and/or camera and began capturing data without the user even being aware that it is happening!

Due to concerns of this nature, both Android and iOS have default provisions that prevent applications from accessing multimedia features until the user explicitly grants it. In order for a user to be willing to give permission, the app has to offer some sort of value proposition and communicate it the user, so they know exactly how the device level multimedia features might be used. This information gets communicated in a variety of ways, ranging from the application's description and screenshots in the respective application stores, to information provided to the user just in time via prompts within the application. As app developer, you owe it to the user to fully disclose how the multimedia related features will be used, so the user can make an informed judgment call on whether or not the privacy potentially given up by them will be compensated fairly by the value these features in your app offer to them in return.

As you explore the multimedia capabilities of Android and iOS we will necessarily also be showing you how to properly handle the privacy constraints enforced by the platforms with regard to multimedia features.

9.1.4 Enhancements to Traxy

In this chapter you will add a new feature to Traxy to allow the users to associate media files with a journal entry. When a journal entry is selected from the list in the main screen, the user will be presented with the journal details and some action buttons for adding media files to the selected journal.

Previously, in the Traxy app, you added code that gave the user the ability to create a new journal. Journals consisted of a name, general location, and start/end dates, all of which you



Figure 9.1: Creating multimedia journal entries in Traxy.

stored in the Firebase cloud. As described earlier, each journal will consist of a multiplicity of *journal entries*. A journal entry can take on one of four types:

1. text entry: textual information.

2. photo entry: a photo, along with textual caption.

3. video entry: a video, along with a textual caption.

4. audio entry: an audio recording along with a textual caption.

Each journal entry is also tagged with a user specified date. In a future chapter you will also allow users to tag journal entries with a location, so for now you will add a placeholder field to your entries for location.

Journal entries will be stored in Firebase's Cloud Firestore as a document collection under the Journal document that owns it. Figure 9.2 shows a journal document's entries collection. The actual media content (e.g. images, audio, and video files) will be stored under Firebase Storage, yet another storage capability of the Firebase backend. Firebase storage allows apps to store files and larger chunks of data that aren't as easily accommodated by the more structured Firestore.

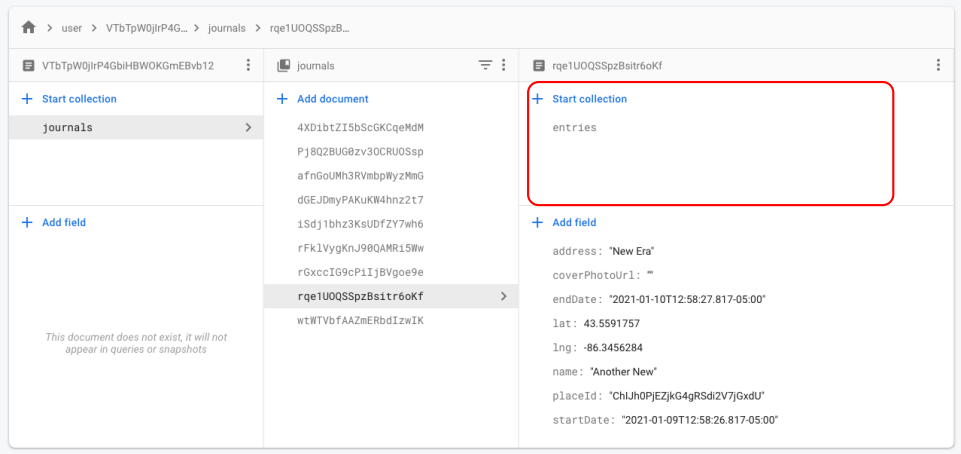


Figure 9.2: The entries collection under a journal document instance in the Firebase Cloud Firestore

Under Firebase Storage, you want to have the media files sandboxed under the user uid and media contents uploaded by one user are further organized into three subdirectories depending on the media type: **audio**, **photos**, and **videos**. Refer to Figure 9.3. At the top of the figure, the text that begins with "M0a5" (highlighted in rounded box), is the user uid. By default the content in Firebase Storage will require an authenticated user to read, write, and delete.

To protect your media files from unauthorized accesses, you should set the Firebase Storage Access Rules to Listing 72.

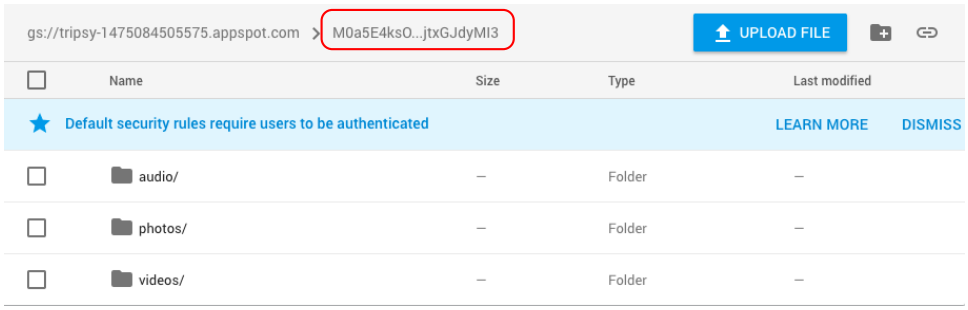


Figure 9.3: Firebase Storage Directory Structure

```
rules_version = '2';
service firebase.storage {
  match /b/{bucket}/o {
    match /{allPaths=**} {
      allow read, write: if request.auth != null;
    }
  }
}
```

Listing 72: Firebase Storage Access Rules: Allow Only Authenticated Users

9.2 Multimedia in iOS

In the previous chapter you implemented the `MainViewController` which displayed the journals that were persisted in the Firebase cloud. Let's flesh out the app further so that when the user selects a journal they can browse and add new journal entries as shown in Figure 9.4

9.2.1 Working with UITableViewController

To get started you will introduce a new controller class that extends the `UITableTableViewController` class. The `UITableViewController` is an extension of the `UIViewController` that manages a `UITableView` and provides some common default behaviors. You've already met the `UITableView` class in Chapter 6.2.2 when you used it to implement your `MainViewController` class. You will use the `UITableViewController` class to manage your table view of journal entries. Here is a step by step summary of what you need to complete:

1. Introduce a new `UITableViewController` class in Swift.
2. Add a corresponding "Table View Controller" view to the storyboard.
3. Hook up a segue from the protoype cell in our `MainControllerView` to this new scene.
4. Add a "Bar Button Item" on the right side of the navigation bar to add new entries. This is the button labeled "+" in the upper right corner of the screens shown in Figure 9.4.

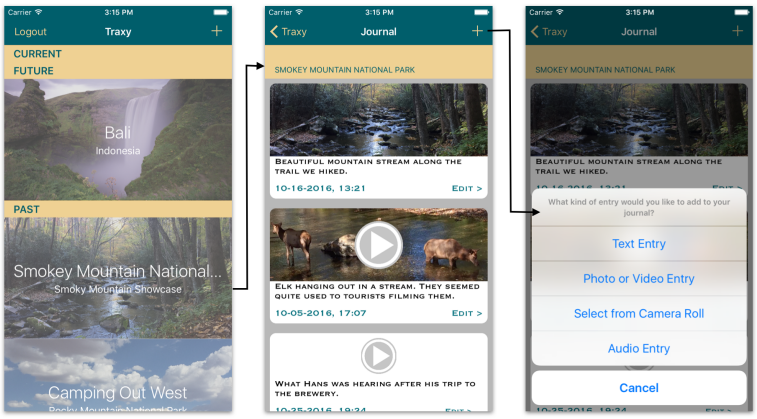


Figure 9.4: Browsing and adding journal entries within a journal.

Select the "Controllers" folder in the Xcode project explorer and introduce a new class named `JournalTableViewController` via File ⟩ New ⟩ File ... and selecting Cocoa Touch Class. Enter the class name and make sure the subclass is set to `UITableViewController`. A new source file containing the stubbed out version of a new class will be added to your project.

You will want the height of the rows in your table to be set dynamically, based on the content contained with in them, make sure selection is cleared whenever the view appears, and set the scene title to the name of the journal. Accomplish this by adding four lines of code to the `viewDidLoad()` method of the controller as shown in Listing 73.

Swift

```
override func viewDidLoad() {
  super.viewDidLoad()
  self.tableView.rowHeight = UITableView.automaticDimension
  self.tableView.estimatedRowHeight = 200
  self.clearsSelectionOnViewWillAppear = true
}
```

Listing 73: Dynamically setting the height of the individual UITableViewCell's based on the content they contain.

Next, in the storyboard go ahead and drag the component named "Table View Controller" from the object library out onto the canvas. In the Identity Inspector set the custom class to `JournalTableViewController`. Select the table view in the scene and in the Attribute Inspector set "Selection" to "No Selection". You will customize the cell layouts and don't want the default selection visuals.

To add the segue, go back to the `MainViewController` in the storyboard and while holding ctrl select the cell that you gave the "FancyCell" identifier to, drag over to the table view scene you just added and release. Select the segue that was just added and set its identifier to "showJournalSegue".

You're now ready to add the bar button item. Even though you already have a navigation bar

displayed in the scene (by merit of adding the show segue from the main scene) you cannot add a button to it unless you first add a "Navigation Item" to the scene. Locate the "Navigation Item" in the object library and drag it over the scene and drop it. Next, locate the "Bar Button Item" in the object library, and drag it over to the right hand side of the navigation bar and release. By default the button will have the label "Item" on it, but you of course want the "+" symbol to denote it is for adding entries. To change this, select the button, and then in the Attribute Inspector change the System Item from "Custom" to "Add" via the drop down control.

9.2.2 Presenting Action Sheets

When the user taps on the add button in the upper right, an array of options is displayed, as shown in Figure 9.4. This can be accomplished using an instance of the `UIAlertController` class with a `preferredStyle` set to `actionSheet`. `UIAlertController` can be used to display modal dialogs in a variety of styles. The action sheet style presents the user with a set of alternatives to proceed with a given task. Additional styles will be used later in this chapter to inform the user about the need to grant permissions.

As can be seen in the figure, you need to present the user with options to enter text, take a photo or video with the camera, select a photo or video from the camera roll, or record audio. This is implemented by adding an `IBAction` to the bar button item on the nav bar and adding the stubbed out code shown in Listing 74.

The alert controller instance is created in line 2, and presented in lines 32–34. For each of the alternatives mentioned a `UIAlertAction` instance is introduced. Each `UIAlertAction` has a Swift closure associated with it that is executed when that alternative is selected. For now just stub these out as shown. You will fill in the details in the sections that follow. If you build and run the application at this point, though no journal entries will yet appear, you will see that the code as implemented is sufficient to get the action sheet to display. Let's turn and look more closely at how you can generate multimedia journal entries.

9.2.3 Working with Multimedia on iOS

iOS and Android provide the programmer with both low level and higher level access to capturing and displaying content with the camera. The simplest way to capture a photo or video in an iOS application is to use the `UIImagePickerController`. This class provides a user interface very similar to the native camera app that Apple preloads on iOS devices. In addition to interacting with the camera, `UIImagePickerController` also can be used to let the user select a previously captured video or photo that is resident in the camera roll on the device. In the iOS version of Traxy you will take advantage of this particular feature of `UIImagePickerController` and allow the user to post journal entries containing photos and videos that were taken previously and stored on the device.

If user interface requirements go beyond the basic camera app, the camera can be controlled directly via the AV Foundation framework. For example, if an app was using the camera to scan a bar code, you might want to display an overlay on the live camera view to help guide the user to position the phone correctly. Stored photos and videos (as well as photos/videos stored in the user's iCloud Photo library) can be accessed (and edited) via the iOS Photos framework.

Audio can be captured on an iOS device using the AV Foundation framework's `AVAudioRecorder` class. Audio playback takes place via the `AVAudioPlayer` class. Both of these classes operate on

Swift

```swift
@IBAction func addEntryButtonPressed(_ sender: UIBarButtonItem) {
    let alertController = UIAlertController(title: nil, message:
       "What kind of entry would you like to add to your journal?",
       preferredStyle: .actionSheet)

    let cancelAction = UIAlertAction(title: "Cancel", style: .cancel) { (action) in
      // TBD ...
    }
    alertController.addAction(cancelAction)

    let addTextAction = UIAlertAction(title: "Text Entry", style: .default) {
    (action) in
      // TBD ...
    }
    alertController.addAction(addTextAction)

    let addCameraAction = UIAlertAction(title: "Photo or Video Entry",
    style: .default) { (action) in
      // TBD ...
    }
    alertController.addAction(addCameraAction)

    let selectCameraRollAction = UIAlertAction(title: "Select from Camera Roll",
    style: .default) { (action) in
      // TBD ...
    }
    alertController.addAction(selectCameraRollAction)

    let addAudioAction = UIAlertAction(title: "Audio Entry", style: .default) {
    (action) in
      // TBD ...
    }
    alertController.addAction(addAudioAction)

    self.present(alertController, animated: true) {
      // TBD ...
    }
}
```

Listing 74: Displaying the action sheet.

local media files only.

The `AVPlayer` class can be used to play both local and remote audio and video files. Furthermore, the AVKit framework provides the `AVPlayerViewController` class which provides a basic media player user interface that interacts with an `AVPlayer` instance. You will take advantage of all of these classes in your implementation.

Before you start coding you'll need to make a few tweaks to your Xcode projects in order to have visibility into the AVFoundation / AVKit frameworks. By default, an iOS Xcode project provides visibility to the Foundation and UIKit frameworks, since essentially every iOS app will utilize the functionality in those frameworks. As soon as you use more specialized frameworks such as AVFoundation and AVKit, you need to explicitly add those frameworks to your project's build. For the code that you will write in the remainder of this section you will need to add three frameworks to your project:

- AVFoundation: provides essential services for working with multimedia on iOS.
- AVKit: provides view-level services for media playback on iOS.
- MobileCoreServices: provides uniform type identifiers to create and manipulate multimedia data.

This can be accomplished via the following steps, as illustrated in Figure 9.5.

1. Select the root of your project explorer in Xcode. (Labeled as "1" in Figure 9.5.)
2. Scroll to the bottom of the General tab content, and click the plus (+) button under the Linked Frameworks and Libraries section. (Labeled as "2" in Figure 9.5.)
3. You can then use the dialog that displays (not shown in the figure) to add the frameworks required by the project — in this case AVFoundation, AVKit, and MobileCoreServices.

9.2.4 Capturing Photos and Videos

Regardless of which iOS interfaces you use to access the camera, or already captured media assets stored on the device, the user must explicitly grant our application permission. Hence, your first task is to determine whether or not adequate permissions have been granted, by calling the `AVCaptureDevice.authorizationStatus` method provided by the AV Foundation framework. Listing 75 shows the prerequisite code needed to check for permission before attempting to display the camera, in the form of a method named `displayCameraIfPermitted` added to the `JournalTableViewController` class. While you are editing the class, be sure to add these imports that will give us visibility to the various SDK's being used in our implementation.

Swift

```
import AVFoundation
import AVKit
import MobileCoreServices
```

```swift
func displayCameraIfPermitted() {
  let cameraMediaType = AVMediaType.video
  let cameraAuthorizationStatus =
  AVCaptureDevice.authorizationStatus(for: cameraMediaType)

  switch cameraAuthorizationStatus {
  case .denied:
    self.displaySettingsAppAlert()
  case .authorized:
    self.displayImagePicker(type: .camera)
  case .restricted: break
  case .notDetermined:
    // Prompting user for the permission to use the camera.
    AVCaptureDevice.requestAccess(for: cameraMediaType) { granted in
      if granted {
        DispatchQueue.main.async {
          self.displayImagePicker(type: .camera)
        }
      } else {
        print("Denied access to \\(cameraMediaType)")
      }
    }
  @unknown default:
    print("Error: unexpected status")
  }
}
```

Listing 75: Determining if the user has granted access to the camera.

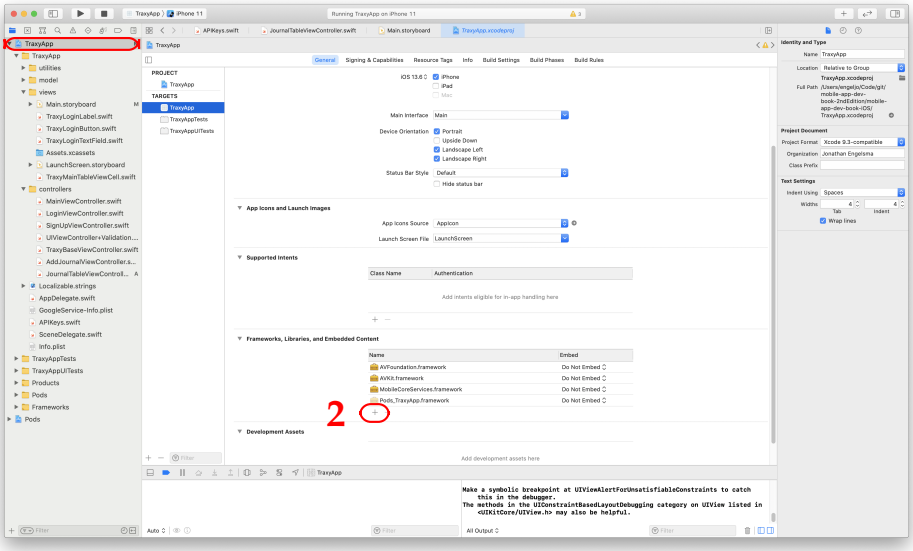


Figure 9.5: Adding additional Apple iOS frameworks to your Xcode project.

The iOS permissions model works by providing the user with a one-time prompt within the application requesting permissions and including a brief reason for why the application is asking for permissions. If the user does not grant permission when this dialog is displayed, it will never be displayed again, unless the app is deleted and reinstalled by the user. However, at any given point in time, the user can access and modify these permissions within the iOS Settings app. A prerequisite for this to work is to add a string setting keyed to `NSCameraUsageDescription` in the application's Info.plist file. This string will be displayed on the dialog when permissions is requested. Similar keys must be added for the camera roll (`NSPhotoLibraryUsageDescription`) and the microphone (`NSMicrophoneUsageDescription`). Representative permissions dialogs for these three functions are shown in Figure 9.6. Note that if you run your app on a simulator, permissions are assumed so you will never see these dialogs. However, when you run on a physical iOS device, the permissions will not be granted unless you add the appropriate prompt keys to your Info.plist file.

With this background information on permissions in mind, let's now return to the code in Listing 75. Lines 2–4 demonstrate how to query the system to find out if access to the camera has been authorized. Note that the valid media type inputs to the `AVCaptureDevice.authorizationStatus` method are `AVMediaTypeVideo` or `AVMediaTypeAudio`. The former is what we use here to denote we are interested in the camera. There are four different status values we can check for.

- `.denied`: The application has previously requested permissions and has been denied. In our app there is not much we can do here, other than inform the user that the app needs camera access and advise them that they can accomplish that via the settings app. This is accomplished via a helper method named displaySettingsAppAlert and show in Listing 76.
- `.authorized`: The application has been granted permission to use the camera. This is what

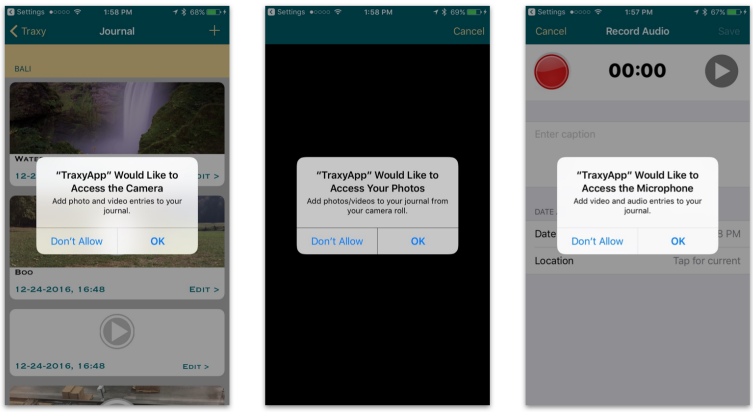


Figure 9.6: One-time permission requests dialogs. The text in the smaller font was provided via entries in the app's Info.plist.

we are hoping for, and we'll precede to display the image picker by calling a helper method `displayImagePicker`.

- `.restricted`: Ending up here will be less common, as the only time this value will be returned is when an iOS device is being managed by some sort of enterprise mobile device management system and the camera has been disabled. Since the user typically can't do anything about this, the code will simply quietly ignore any attempt to add a photo entry. Another option of course would be to kindly inform the user that their camera is not available.

- `.notDetermined`: If the application has yet to ask the user for permission to use the camera, we will get this status value. This is actually a strategic time to make the that one-time request to the user for access by calling the `AVCaptureDevice.requestAccess` method, passing it a completion closure to determine how to handle the outcome of the request. If the application has not yet asked for permission, a dialog requesting permission will be displayed. The provided closure is called with a boolean parameter that is true if permission was granted, or false otherwise. If permission was granted, the camera will be displayed. The permission request dialog will not be displayed more than once. Note that the `DispatchQueue.main.async` is necessary in that the completion closure will be called on a background thread. User interface related code must be executed exclusively on the main thread, hence the dispatch..

Since the Traxy app does not have any special user interface requirements when it comes to the camera, you can use the `UIImagePickerController` which provides a very familiar user interface for capturing photos and videos. In fact, by simply changing a single `sourceType` property when you instantiate the controller, you can cause it to display the camera roll and allow the user to select a previously taken photo or video. Listing 77 demonstrates the helper function `displayImagePicker` that displays the image picker. The type parameter can be set to `.camera` to display the camera or `.photoLibrary` to allow selection from the photo roll. Since your table

```swift
func displaySettingsAppAlert()
{
  let avc = UIAlertController(title: "Camera Permission Required",
    message: "You need to provide this app permissions to use your camera
    for this feature. You can do this by going to your Settings app and going
    to Privacy -> Camera", preferredStyle: .alert)

  let settingsAction = UIAlertAction(title: "Settings", style: .default ) { action in
    UIApplication.shared.open(
      NSURL(string: UIApplication.openSettingsURLString)! as URL,
      options: [:],
      completionHandler: nil)
  }

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

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

Listing 76: Determining if the user has granted access to the camera.

view is displayed within the context of a navigation controller, the image picker will be pushed onto the navigation stack, and a done button added to cancel if the user changes their mind once the camera viewfinder is displayed.

```swift
func displayImagePicker(type: UIImagePickerController.SourceType)
{
     let picker = UIImagePickerController()
     picker.allowsEditing = false
     picker.sourceType = type
     picker.mediaTypes = [kUTTypeImage as String, kUTTypeMovie as String]
     picker.modalPresentationStyle = .fullScreen
     picker.delegate = self
     self.present(picker, animated: true, completion: nil)
}
```

Listing 77: Determining if the user has granted access to the camera.

Having defined these two helper functions, we can now flesh out of some of the details that we left stubbed out in the `addEntryButtonPressed` method. In particular, we can call the `displayCameraIfPermitted` helper in the closure associated with the `addCameraAct` as follows:

```swift
let addCameraAction = UIAlertAction(title: "Photo or Video Entry", style: .default) {
(action) in
  self.displayCameraIfPermitted()
}
```

Similarly, we can call the `displayImagePicker` helper in the `selectCameraRollAction`:

```swift
let selectCameraRollAction = UIAlertAction(
  title: "Select from Camera Roll",
  style: .default) { (action) in
  self.displayImagePicker(type: .photoLibrary)
}
```

Listing 77 of course simply creates and presents the image picker. You still haven't addressed what you want to happen when an photo or video is captured or selected. As you might have guessed, iOS accomplishes this via delegation. In particular, you need to implement the `UIImagePickerControllerDelegate` protocol. Before you implement this however, now that the app is ready to start adding entries to the journal you need to expand the model data structures and add some instance variable to `JournalTableViewController` to hold entries and their data when captured, but not yet persisted to the cloud.

Implement a new structure called `JournalEntry` to model journal entries, as shown in Listing 78. Journal entries can be one of four different types which are defined with an enumerated type called `EntryType`. The `key` is a unique string that Firebase will generate automatically when an entry is persisted in the cloud. The `caption` field will hold a textual caption of the media captured by the entry, with the exception of text entries which only have text. The `url` field will specify the URL to the media in the cloud, either a link to a video, photo, or audio recording. You will see in the next section, that these media files need to be stored separately on Firebase. The `thumbnailUrl` is only used by video entries. You will store a frame from the video as an image and use it in the user interface to display the video.

Next define model data and properties to `JournalTableViewController` to temporarily hold entries as they are being captured and confirmed. These definitions are shown in Listing 79. The property `capturedImage` will temporarily hold a copy of the image captured or selected while being confirmed. For entries of type video, this property will store the thumbnail of the video. The `capturedVideoUrl` property will temporarily hold the local URL to a captured video while it is being confirmed. The `capturedType` property will be set whenever the user attempts to add a new entry after making a selection from the action sheet. The `journal` property is set by the `MainViewController` in its `prepareForSegue` method when a particular journal is selected as shown in Listing 80. It is the journal instance which owns the journal entries being created and edited. The `entries` property holds the array of entries currently being displayed in the table view, and the `entryToEdit` property is set whenever a user attempts to edit a given entry. Now that we know which journal we are adding entries to, we will set the title on the navigation bar to be the name of the journal, by adding this line of code to the `viewDidLoad` method of `JournalTableViewController`:

```swift
self.navigationItem.title = self.journal.name
```

Swift

```swift
enum EntryType : Int {
  case text = 1
  case photo
  case audio
  case video
}

struct JournalEntry {
  var key : String?
  var type: EntryType?
  var caption : String?
  var url : String
  var thumbnailUrl : String
  var date : Date?
  var lat : Double?
  var lng : Double?

  init(key: String?, type: EntryType?, caption: String?, url: String,
    thumbnailUrl: String, date: Date?, lat: Double?, lng: Double?)
  {
    self.key = key
    self.type = type
    self.caption = caption

    self.date = date
    self.lat = lat
    self.lng = lng
    self.url = url
    self.thumbnailUrl = thumbnailUrl
  }
}
```

Listing 78: A struct for modeling journal entries.

Swift

```swift
var capturedImage : UIImage?
var captureVideoUrl : URL?
var captureType : EntryType = .photo

var journal: Journal!
var userId : String!
var entries : [JournalEntry] = []
var entryToEdit : JournalEntry?

let repo = TraxyRepository.getInstance()
```

Listing 79: Properties defined by `JournalTableViewController` for model data and temporary storage of journal entries being created.

```swift
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
  if segue.identifier == "addJournalSegue" {
    if let destVC = segue.destination as? AddJournalViewController {
      destVC.delegate = self
    }
  } else if segue.identifier == "showJournalSegue" {
    if let destVC = segue.destination as? JournalTableViewController {
      let indexPath = self.tableView.indexPathForSelectedRow
      let values = self.tableViewData?[indexPath!.section]
      destVC.journal  = values?.journals[indexPath!.row]
      destVC.userId = self.userId
    }
  }
}
```

Listing 80: Updating the `MainViewController` to setup the journal property of `JournalTableViewController` when seguing to it. .

Having defined these properties on the `JournalTableViewController`, you are now ready to implement the `UIImagePickerControllerDelegate` protocol. Do this in the form of a Swift extension as shown in Listing 81. Note that the `UIImagePickerController` also requires the delegate implement the `UINavigationControllerDelegate` for some of its more advanced navigation features. Though you aren't using those features here, you still need to declare it. The `imagePickerControllerDidCancel` method gets called if the picker is dismissed by tapping the done button before an image is captured or selected. The user of course will do this when they wish to abort the capture, so the proper thing to do is to simply dismiss picker. The `imagePickerController:didFinishPickingMediaWithInfo` method gets called whenever the picker captures or selects a photo or video. By checking the type of video selected it can be determined if the media object is an image or a video. In the former case, you also need to extract a thumbnail image from the video. This is accomplished via a helper function called `thumbnailForVideoAtURL`. Based on the local URL of the captured video, you can use AV Foundation classes to remove the second frame of the video (or the first if there is only one frame). Either way, once an image or video is captured or selected the `captureType` property is set to reflect the media type, and the `capturedImage` property of the controller holds the `UIImage` of the captured photo or a frame from the video. If a video was captured, in addition the `captureVideoUrl` is set to the local url of the captured video. At this point you are ready to give the user the opportunity to provide a caption and tag the media with a date and location. This is accomplished by performing a segue with identifier `confirmSegue`. Incidentally, you will be able to use this same confirmation scene to originate any text journal entries the user wishes to add, so at this point the only user interface left to implement is the audio recording which you will implement in a separate controller in a subsequent section.

Before uploading the media and its related metadata (caption, date, location) to Firebase, the app will give the user the opportunity to provide the metadata via a new confirmation scene. Do this by dragging a new View Controller from the object library out into the storyboard canvas. Add a manual segue from the `JournalTableViewController` to this new controller and giving it the identifier `confirmSegue`. Add a bar button labeled "Cancel" to the left side of the navigation

Swift

```swift
extension JournalTableViewController : UIImagePickerControllerDelegate,
            UINavigationControllerDelegate
{

  func imagePickerControllerDidCancel(_ picker: UIImagePickerController) {
    print("canceled")
    self.dismiss(animated: true, completion: nil)
  }

  func imagePickerController(_ picker: UIImagePickerController,
    didFinishPickingMediaWithInfo info: [UIImagePickerController.InfoKey : Any])
  {
    let info = convertFromUIImagePickerControllerInfoKeyDictionary(info)
    self.dismiss(animated: true, completion: nil)
     if let mediaType =
        info[convertFromUIImagePickerControllerInfoKey(
            UIImagePickerController.InfoKey.mediaType)] as? String
     {
      if mediaType == kUTTypeMovie as String {
        print("got video")
        self.capturedImage = self.thumbnailForVideoAtURL(url:
          info[convertFromUIImagePickerControllerInfoKey(
            UIImagePickerController.InfoKey.mediaURL)]as! URL)
        self.captureVideoUrl =
          info[convertFromUIImagePickerControllerInfoKey(
            UIImagePickerController.InfoKey.mediaURL)] as? URL
        self.captureType = .video
      } else {
        print("got image")
        self.capturedImage =
          info[convertFromUIImagePickerControllerInfoKey(
              UIImagePickerController.InfoKey.originalImage)] as? UIImage
        self.captureType = .photo
      }
    }
    self.performSegue(withIdentifier: "confirmSegue", sender: self)
  }


  func thumbnailForVideoAtURL(url: URL) -> UIImage? {
    let asset = AVURLAsset(url: url)
    let generator = AVAssetImageGenerator(asset: asset)
    generator.appliesPreferredTrackTransform = true
    var time = asset.duration
    time.value = min(time.value, 2)

    do {
      let imageRef = try generator.copyCGImage(at: time, actualTime: nil)
      return UIImage(cgImage: imageRef)
    } catch {
      print("error")
      return nil
    }
  }
}
```

Listing 81: Implementation of the `UIImagePickerControllerDelegate`. This code handles the outcome of display a `UIImagePickerController` instance.

bar and a second bar button item labeled "Save" to the right side of the navigation bar.

Text entry forms are actually a bit more complicated to implement on iOS than on Android. Most of this complexity revolves around managing the keyboard when a text appears under the displayed software keyboard. It is the programmers responsibility to place the fields in a scroll view and position it just above the keyboard in these situations. To keep things simple and avoid these details, we're simply going to use the Eureka pod to implement our confirmation screen. We've worked with the Eureka pod in an earlier chapter, so most of what we need for our confirmation screen will be familiar.

You're going to introduce a new class called `JournalEntryConfirmationViewController` that extends the `FormViewController` defined by the Eureka pod. The implementation of this class is given in Listing 82. After defining this class, you need to go back into the storyboard and set the custom class of the view controller to this class. While you are at it wire up the cancel and save buttons to the `cancelButtonPressed` and `saveButtonPressed` `IBActions`. As usual, `JournalEntryConfirmationViewController` will return the journal entry to be saved to the presenting `JournalTableViewController` via delegation. To this end, the `AddJournalEntryDelegate` class is defined with a single method save. `JournalTableViewController` will implement this protocol and set itself as the `JournalEntryConfirmationViewController`'s delegate when it prepares for segue.

Now that you have all the logic in place to create photo and video journal entries via the `UIImagePickerController` you need to go back and implement the closures in your action sheet code to get call our helper functions. Listing 83 shows the fully implemented actions for text, photos, and video entries.

Finally, in order to correctly setup the properties on our `JournalEntryConfirmationViewController` you need to have the `JournalTableViewController` override the `prepareForSegue` method as is shown in Listing 84.

9.2.5 Uploading Media to Firebase

Recall that all of our Firebase specific code is contained in our `TraxyRepository` class. In the previous chapter, we've stubbed out the additional methods that we need to persist individual journal entries. Before we finish implementing these methods, let's first wire up `JournalTableViewController` to listen and stop listening to journal entry updates when it appears and disappears respectively. We can accomplish this in the controller's `viewWillAppear` and `viewWillDisappear` methods as shown in Listing 85. You'll notice the code here is very similar to what we saw in the previous chapter when handling journal updates in `MainViewController`.

The `listenFoJournalEntryUpdates` method implemented by the `TraxyRepository` singleton in many ways resembles the code we wrote in the previous section for listening to journal updates, as you can see in Listing 86. We'll assume this code is easily understood, but if not, go back and review the more detailed discussion of the analogous code in Chapter 8.

Recall that new entries will be passed back to `JournalTableViewController` via the `AddJournalEntryDelegate` protocol. That is, `JournalTableViewController` must implement this protocol and set itself as the delegate of `JournalEntryConfirmationViewController`. Listing 87 shows the implementation of this protocol at a high level. The details are implemented a series of calls to the `TraxyRespository`

iOS

```
import UIKit
import Eureka

protocol AddJournalEntryDelegate : class {
    func save(entry: JournalEntry)
}

var previewImage : UIImage?

class JournalEntryConfirmationViewController: FormViewController {

    var imageToConfirm : UIImage?
    weak var delegate : AddJournalEntryDelegate?
    var type : EntryType?
    var entry : JournalEntry?
    var journal : Journal!


    override func viewDidLoad() {
        super.viewDidLoad()
        previewImage = imageToConfirm

        let textRowValidationUpdate : (TextRow.Cell, TextRow) -> ()  = { cell, row in
            if !row.isValid {
                cell.titleLabel?.textColor = .red
            } else {
                cell.titleLabel?.textColor = .black
            }
        }
        TextRow.defaultCellUpdate =  textRowValidationUpdate
        TextRow.defaultOnRowValidationChanged = textRowValidationUpdate

        let dateRowValidationUpdate : (DateRow.Cell, DateRow) -> () = { cell, row in
            if !row.isValid {
                cell.textLabel?.textColor = .red
            } else {
                cell.textLabel?.textColor = .black
            }
        }
        DateRow.defaultCellUpdate = dateRowValidationUpdate
        DateRow.defaultOnRowValidationChanged = dateRowValidationUpdate

        let labelRowValidationUpdate : (LabelRow.Cell, LabelRow) -> () = { cell, row in
            if !row.isValid {
                cell.textLabel?.textColor = .red
            } else {
                cell.textLabel?.textColor = .black
            }
        }
        LabelRow.defaultCellUpdate = labelRowValidationUpdate
        LabelRow.defaultOnRowValidationChanged = labelRowValidationUpdate

        var textEntryLabel = "Enter caption"
        if self.type == .text {
            textEntryLabel = "Enter text entry"
            self.navigationItem.title = "Text Entry"
        }
```

iOS

```
66
67      var caption : String = ""
68      var date : Date = self.journal.startDate!
69      if let e = self.entry {
70          caption = e.caption!
71          date = e.date!
72      } else {
73          self.entry = JournalEntry(key: nil, type: self.type, caption: caption,
74              url: "", thumbnailUrl: "", date: date, lat: 0.0, lng: 0.0)
75      }
76
77      form = Section() {
78          $0.tag = "FirstSection"
79          if self.type != .text  && self.type != .audio {
80              $0.header = HeaderFooterView<MediaPreviewView>(.class)
81          }
82      }
83          <<< TextAreaRow(textEntryLabel){ row in
84              row.placeholder = textEntryLabel
85              row.value = caption
86              row.tag = "CaptionTag"
87              row.add(rule: RuleRequired())
88          }
89
90          +++ Section("Date and Location Recorded")
91          <<< DateTimeInlineRow(){ row in
92              row.title = "Date"
93              row.value = date
94              row.tag = "DateTag"
95              row.maximumDate = self.journal.endDate
96              row.minimumDate = self.journal.startDate
97              row.dateFormatter?.dateStyle = .medium
98              row.dateFormatter?.timeStyle = .short
99              row.add(rule: RuleRequired())
100         }
101
102         <<< LabelRow () { row in
103             row.title = "Location"
104             row.value = "Tap for current"
105             row.tag = "LocTag"
106             var rules = RuleSet<String>()
107             rules.add(rule: RuleClosure(closure: { (loc) -> ValidationError? in
108                 if loc == "Tap to search" {
109                     return ValidationError(msg: "You must select a location")
110                 } else {
111                     return nil
112                 }
113             }))
114             row.add(ruleSet:rules)
115
116             }.onCellSelection { cell, row in
117                 print("TODO: will finish this next chapter!")
118     }
119 }
```

iOS

```
120
121 override func didReceiveMemoryWarning() {
122     super.didReceiveMemoryWarning()
123     // Dispose of any resources that can be recreated.
124 }
125
126 @IBAction func cancelButtonPressed(_ sender: UIBarButtonItem) {
127     _ = self.navigationController?.popViewController(animated: true)
128 }
129
130 @IBAction func saveButtonPressed(_ sender: UIBarButtonItem) {
131     if let del = self.delegate {
132
133         let (caption,date,_) = self.extractFormValues()
134
135         if var e = self.entry {
136             e.caption = caption
137             e.date = date
138             del.save(entry: e)
139         }
140     }
141     _ = self.navigationController?.popViewController(animated: true)
142 }
143
144 func extractFormValues() -> (String, Date, String)
145 {
146     let captionRow: TextAreaRow! = form.rowBy(tag: "CaptionTag")
147     //let locRow: LabelRow! = form.rowBy(tag: "LocTag")
148     let dateRow : DateTimeInlineRow! = form.rowBy(tag: "DateTag")
149     let locationRow : LabelRow! = form.rowBy(tag: "LocTag")
150
151     let caption = captionRow.value! as String
152     //let location = locRow.value! as String
153     let date = dateRow.value! as Date
154     let loc = locationRow.value! as String
155
156     return (caption,date,loc)
157 }
158
159 class MediaPreviewView: UIView {
160
161     override init(frame: CGRect) {
162         super.init(frame: frame)
163
164         // size image view to a third of available vertical space.
165         let screenSize: CGRect = UIScreen.main.bounds
166         let width = screenSize.width
167         let height = screenSize.height / 3.0
168
169         var image : UIImage
170         if let img = previewImage {
171             image = img
172         } else {
173             image = UIImage()
174         }
175         let imageView = UIImageView(image: image)
```

iOS

```
            imageView.contentMode = .scaleAspectFill
            imageView.frame = CGRect(x: 0, y: 0, width: width, height: height)
            self.frame = CGRect(x: 0, y: 0, width: width, height: height)
            self.clipsToBounds = true
            self.addSubview(imageView)
        }

        required init?(coder aDecoder: NSCoder) {
            fatalError("init(coder:) has not been implemented")
        }
    }
}
```

Listing 82: The JournalEntryConfirmationViewController is used to confirm new entries of type text, photo, and video.

singleton. Let's systematically examine how the singleton implements each of the journal types.

Text Journal Entries to Firebase

Let's begin by looking at how text entries are handled. First, recall from the previous chapter that Firebase expects us to encapsulate the values we are going to set on a given key in the form of dictionary instance. We introduce a helper function named toDictionary that does just that for all journal entry types. For a text entry we will assume the caption field contains the text, so all we need to do is update the appropriate child in our current journal's entry array, or place a new child with a generated key into the array if a new entry is being added. We accomplish this with the `saveEntryToFirebase` helper method. The method and the dependent helpers are shown in Listing 88.

Saving Photo Journal Entries to Firebase

As was presented earlier in this chapter, the media files for audio, video and images cannot be stored in Firebase's real-time database, but must be uploaded to Firebase Storage. Once a file is uploaded to Firebase, a unique URL is generated and can be stored in the Firebase Database as needed. Hence each time we create a new journal entry we will have to do a series of updates to Firebase. First, we will create a child node for the new journal entry without the URL(s). After the media has been uploaded, then we need to circle around and add the URL(s) to the journal entry's child node. For video entries, we actually have two media files to upload — the video, as well as the thumbnail image. For audio and photos, we only have a single file upload.

Before we look at the details of persisting and retrieving the various media types in Firebase, we do need to add a bit of setup code to `TraxyRepository` to be able to have it read/write files to the Storage of our Firebase instance. Just as we previously defined a `DocumentReference` property to refer to the top level document containing the currently authenticated user's collection of journals, we need to add a `StorageReference` property to the `TraxyRepository` class to refer to the top of the Storage under which we will deposit our media files:

Swift

```swift
@IBAction func addEntryButtonPressed(_ sender: UIBarButtonItem) {
      let alertController = UIAlertController(title: nil, message:
      "What kind of entry would you like to add to your journal?",
      preferredStyle: .actionSheet)

      let cancelAction = UIAlertAction(title: "Cancel", style: .cancel) { (action) in
        // ...
      }
      alertController.addAction(cancelAction)

      let addTextAction = UIAlertAction(title: "Text Entry", style: .default) {
      (action) in
        self.captureType = .text
        self.capturedImage = nil
        self.performSegue(withIdentifier: "confirmSegue", sender: self)
      }
      alertController.addAction(addTextAction)

      let addCameraAction = UIAlertAction(title: "Photo or Video Entry", style: .default) {
      (action) in
        self.displayCameraIfPermitted()

      }
      alertController.addAction(addCameraAction)

      let selectCameraRollAction = UIAlertAction(title: "Select from Camera Roll",
      style: .default) { (action) in
        self.displayImagePicker(type: .photoLibrary)
      }
      alertController.addAction(selectCameraRollAction)


      let addAudioAction = UIAlertAction(title: "Audio Entry", style: .default) {
      (action) in
        // TBD
      }
      alertController.addAction(addAudioAction)

      self.present(alertController, animated: true) {
        self.entryToEdit = nil // creating a new entry when this alert is displayed.
      }

}
```

Listing 83: Implementing the closures on the action sheet actions to display the image picker.

```swift
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
  if segue.identifier == "confirmSegue" {
    if let destCtrl = segue.destination as? JournalEntryConfirmationViewController {
      destCtrl.imageToConfirm = self.capturedImage
      destCtrl.delegate = self
      destCtrl.type = self.captureType
      destCtrl.entry = self.entryToEdit  // will be nil on new item
      destCtrl.journal = self.journal
    }
  }
}
```

Listing 84: Overriding the `prepareSegue` method of the `JournalTableViewController` class to setup the confirmation view.

```swift
override func viewWillAppear(_ animated: Bool) {
  super.viewWillAppear(animated)
  if let key = self.journal.key {
    repo.listenForJournalEntryUpdates(journalKey: key) { entries in
      self.entries = entries
      self.entries.sort {$0.date! > $1.date! }
      self.tableView.reloadData()
    }
  }
}

override func viewWillDisappear(_ animated: Bool) {
  super.viewWillDisappear(animated)
  repo.stopListeningForUpdates()
}
```

Listing 85: Framework hook methods implemented in the `JournalTableViewController` class to listen and stop listening to journal entry updates when the scene appears / disappears.

Swift

```swift
func listenForJournalEntryUpdates(journalKey: String,
  onChange: @escaping ([JournalEntry]) -> Void)
{
  self.listener = self.ref?.collection("journals").document(journalKey)
  .collection("entries").addSnapshotListener({(snapshot, error) in
    guard let documents = snapshot?.documents else {
      return
    }
    var entries = [JournalEntry]()
    for entry in documents {
      let key = entry.documentID
      let caption : String? = entry["caption"] as! String?
      var url : String? = entry["url"] as? String
      if url == nil {
        url = ""
      }
      var thumbnailUrl : String? = entry["thumbnailUrl"] as? String
      if thumbnailUrl == nil {
        thumbnailUrl = ""
      }
      let dateStr  = entry["date"] as! String?
      let lat = entry["lat"] as! Double?
      let lng = entry["lng"] as! Double?
      let typeRaw = entry["type"] as! Int?
      let type = EntryType(rawValue: typeRaw!)

      entries.append(JournalEntry(key: key, type: type, caption: caption, url:
      url!, thumbnailUrl: thumbnailUrl!, date: dateStr?.dateFromISO8601, lat: lat,
      lng: lng))
    }
    onChange(entries)
  })
}
```

Listing 86: Implementation of the `listenForJournalEntryUpdates` of the `TraxyRepository` singleton.

```swift
extension JournalTableViewController : AddJournalEntryDelegate {
  func save(entry: JournalEntry) {
    switch(entry.type!) {
      case .photo:
        repo.savePhotoEntry(journalKey: journal.key!,
          image: self.capturedImage, entry: entry)
      case .video:
        repo.saveVideoEntry(journalKey: journal.key!,
          captureVideoUrl: self.captureVideoUrl,
          videoThumbNail: self.capturedImage,
          entry: entry)
      case .text:
        repo.saveTextEntry(journalKey: journal.key!, entry: entry)
      case .audio:
        repo.saveAudioEntry(journalKey: journal.key!, entry: entry)
    }
  }
}
```

Listing 87: Implementing the `AddJournalEntryDelegate` protocol to persist journal entries and their associated media via the `TraxyRepository` singleton.

```swift
fileprivate var storageRef : StorageReference?
```

We then initialize this reference by adding these two lines of code to the `init` method:

```swift
let storageUrl = FirebaseApp.app()?.options.storageBucket
self.storageRef = Storage.storage().reference(forURL: "gs://" + storageUrl!)
```

That is the only setup we need to start using Firebase Storage to persist our media files! Listing 89 shows the implementation of the `savePhoto` helper method. The `setCurrentJournal` method sets the journal context that we are operating on, so subsequent calls to helpers don't need to be explicitly passed the context. The remaining logic here is straight-forward and involves potentially two tasks. First we save the entry information to the Firebase database (lines 4–5) and then if the key is nil, that indicates this is a new photo entry. In that case, the image data needs to be persisted in Firebase storage (line 6), and the URL of the uploaded image stored in the Firebase database once the image is uploaded (lines 9–12)

We've broken out the code that actually uses the Firebase SDK to upload the image file into a separate helper, because eventually when we upload video files, we also need to upload a thumbnail image of the video. This code is shown in Listing 90. The helper function `saveImageToFirebase` takes the `UIImage` to be saved and a closure to call upon completing the upload. The closure, which expects a single `String` argument, provides us with the ability to have the download URL of the image updated once the upload completes. A unique file name is generated using the current time, and path components as described earlier in this chapter.

Swift

```swift
func saveTextEntry(journalKey: String, entry: JournalEntry) {
  setCurrentJournal(journalKey: journalKey)
  var vals = self.toDictionary(vals: entry)
  vals["url"]  = ""
  _ = self.saveEntryToFireStore(key: entry.key, vals: vals)
}

fileprivate func setCurrentJournal(journalKey: String) {
  self.currentDocRef = self.db.collection("user").document(self.userId!)
    .collection("journals").document(journalKey)
}

fileprivate func saveEntryToFireStore(key: String?,
  vals: [String:Any]) -> DocumentReference?
{
  var child : DocumentReference?
  if let k = key {
    child = currentDocRef?.collection("entries").document(k)
    child?.setData(vals)
  } else {
    child = currentDocRef?.collection("entries").addDocument(data: vals)
  }
  return child
}

fileprivate func toDictionary(vals: JournalEntry) -> [String:Any] {
  var retval =  [
  "caption": vals.caption! as NSString,
  "lat": vals.lat! as NSNumber,
  "lng": vals.lng! as NSNumber,
  "date" : NSString(string: (vals.date?.iso8601)!) ,
  "type" : NSNumber(value: vals.type!.rawValue),
  "url" : vals.url as NSString,
  "thumbnailUrl" : vals.thumbnailUrl as NSString
  ]
  return retval
}
```

Listing 88: The `saveTextEntry` method defined by `TraxyRepository` and associated private helper functions.

```swift
func savePhotoEntry(journalKey: String, image: UIImage?, entry: JournalEntry) {
  setCurrentJournal(journalKey: journalKey)
  let vals = self.toDictionary(vals: entry)
  let entryRef = self.saveEntryToFireStore(key: entry.key, vals: vals)
  if entry.key == nil {
    self.saveImageToFirebase(imageToSave: image, saveRefClosure: {
      (downloadUrl) in
      // store the image URL
      let vals = [
      "url" : downloadUrl as NSString
      ]
      entryRef?.setData(vals, merge: true)
    })
  }
}
```

Listing 89: Implementation of the `savePhotoEntry` method of the `TraxyRepository` singleton.

Saving Video Journal Entries to Firebase

Video journal entries are perhaps the most involved in that we need to save two media files per entry - the video itself as well as a thumbnail image of the video. Our strategy is to initially save the journal entry (line 8 in Listing 91) independent of whether or not we are saving a new journal entry, or updating a second. Then, if the key of the entry is nil, we know this is a new video journal entry and hence we need to save the video as well as the thumbnail. (Lines 12–41) For the latter, we use the same `saveImageToFirebase` helper that we wrote previously for saving photos (lines 32–41). Because the logic for saving an audio file is identical to saving a video file (except for the location and the mime type) we broke this logic out into a separate helper function `saveMediaFileToFirebase` which is called in lines 18–28.

The `saveMediaFileToFirebase` helper, shown in Listing 92 is very similar to the logic for saving an image file, only for video and audio capture we will load the data being uploaded from a local file URL.

9.2.6 Downloading and Displaying Captured Photos and Videos

We need to download and display our list of journal entries in the table view managed by `JournalTableViewController`, as shown in Figure 9.7. For each entry, if we tap on the edit button we will return to the scene managed by the `JournalEntryConfirmationViewController`, where we can change the caption, date, or location, but not the media attachment (if any) itself. If the user taps on the media itself in any of the non-text entries, then the media is rendered. For viewing photo entries we will save ourselves some effort by using an existing pod named Lightbox. For audio and video entries we will render the media with components provided to us by iOS. We will get the entries to display as shown in Figure 9.7 by defining three different custom cell prototypes in the storyboard, but before we look at that let's first first take a look at the code needed to download our journal entries and associated media files from Firebase.

```swift
fileprivate func saveImageToFirebase(imageToSave : UIImage?,
  saveRefClosure: @escaping (String) -> ())
{
  if let image = imageToSave {
    let imageData = image.jpegData(compressionQuality: 0.8)
    let imagePath =
      "\(self.userId!)/photos/\(Int(Date.timeIntervalSinceReferenceDate * 1000)).jpg"
    let metadata = StorageMetadata()
    metadata.contentType = "image/jpeg"
    if let sr = self.storageRef {
      sr.child(imagePath)
      .putData(imageData!, metadata: metadata) { (metadata, error) in
        if let error = error {
          print("Error uploading: \(error)")
          return
        }
        let imageRef = sr.child(imagePath)
        imageRef.downloadURL(completion: { (url, error) in
          if let error = error {
            // Handle any errors
            print("Error getting url to uploaded image: \(error)")
          } else {
            if let str = url?.absoluteString {
              saveRefClosure(str)
            }
          }
        })
      }
    }
  }
}
```

Listing 90: Implementation of the `saveImageToFIrebase` helper of the `TraxyRepository` singleton. Save an image to Firebase Storage.

Swift

```swift
func saveVideoEntry(journalKey: String,
  captureVideoUrl: URL?,
  videoThumbNail: UIImage?,
  entry: JournalEntry)
{
  setCurrentJournal(journalKey: journalKey)
  let vals = self.toDictionary(vals: entry)
  let entryRef = self.saveEntryToFireStore(key: entry.key,  vals: vals)

  // if we have a nil key, then this is a new entry and we have an video file to save.
  // as well as a thmbnai image.
  if entry.key == nil {

    // save captured video
    if let url = captureVideoUrl {
      var newEntry = entry
      newEntry.url = url.absoluteString
      self.saveMediaFileToFirebase(entry: newEntry, saveRefClosure: { (downloadUrl)
        in

        // record the URL of the video
        let vals = [
        "url" : downloadUrl as NSString
        ]
        print("Updating URL video: \(downloadUrl)")
        entryRef?.setData(vals, merge: true)

      })
    }

    // save video's thumbnail image
    self.saveImageToFirebase(imageToSave: videoThumbNail, saveRefClosure: {
      (downloadUrl) in

      // record the URL of the thumbnail
      let vals = [
      "thumbnailUrl" : downloadUrl as NSString
      ]
      print("Updating thumbnail URL : \(downloadUrl)")
      entryRef?.setData(vals, merge: true)
    })
  }
}
```

Listing 91: Implementation of the `saveVideoEntry` method of the `TraxyRepository` singleton.

```Swift
func saveMediaFileToFirebase(entry: JournalEntry, saveRefClosure: @escaping (String)
  -> () ) {

  let type : String = entry.type! == .audio ? "audio" : "video"
  let ext : String = entry.type! == .audio ? "m4a" : "mp4"
  let mime : String = entry.type! == .audio ? "audio/mp4" : "video/mp4"

  do {

    if  entry.url != ""  {
      let url = URL(string: entry.url)
      let media = try Data(contentsOf: url!)
      print("got data")
      let mediaPath =
       "\(self.userId!)/\(type)/\(Int(Date.timeIntervalSinceReferenceDate * 1000)).\(ext)"
      let metadata = StorageMetadata()
      metadata.contentType = mime
      if let sr = self.storageRef {
        sr.child(mediaPath)
        .putData(media, metadata: metadata) {(metadata, error) in
          if let error = error {
            print("Error uploading: \(error)")
            return
          }

          let videoRef = sr.child(mediaPath)
          videoRef.downloadURL(completion: { (url, error) in
            if let error = error {
              // Handle any errors
              print("Error getting url to uploaded video: \(error)")
            } else {
              if let str = url?.absoluteString {
                saveRefClosure(str)
              }
            }
          })
        }
      }
    }
  } catch {
    print("oops that wasn't good now")
  }
}
```

Listing 92: Private helper method of `TraxyRepository` that saves a video (or audio file) to Firebase storage.

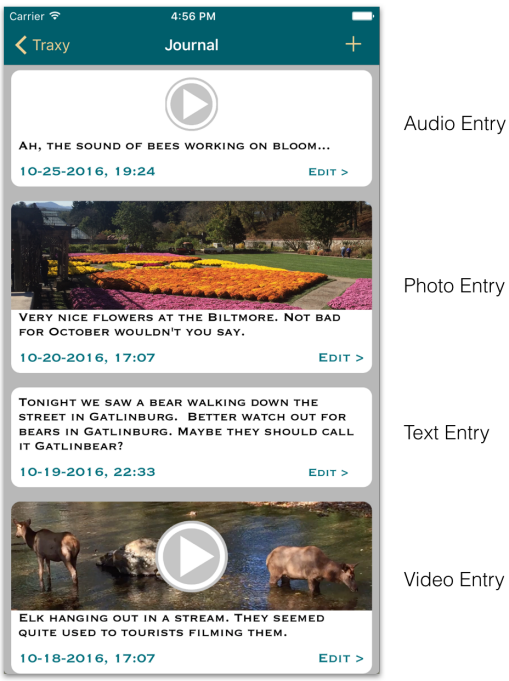


Figure 9.7: A trip journal rendered by JournalTableViewController showing each of the cell styles.

At this point, we've only written the code that gets us the journal entries from the database. We have not yet *downloaded* and displayed any media from Firebase storage. We've already mentioned the need to cache media data. We haven't yet looked at how to integrate directly with web APIs in iOS (that is the topic of our next chapter) so in the spirit of "lazy programmers are good programmers" we are going to simply add the Kingfisher pod to our workspace by adding this entry to our `Podfile` and doing a pod install command. While you're at it, you can add the Lightbox pod as well, since we will be using that eventually to display photos when their tapped on. The lines you need to add to your `Podfile` are:

```
pod 'Kingfisher'
pod 'Lightbox'
```

Don't forget after you've updated your `Podfile` you'll need to run the following pod command to pick them up in your workspace:

```
pod install
```

Kingfisher is a really simple Swift library that makes downloading and caching images a relatively simple task. Make sure you also import the Kingfisher library at the top of the `JournalTableViewController` class. We will look at the details shortly when we discuss the implementation of the `UITableViewDataSource` protocol. But let's first talk about our user interface requirements for `JournalTableViewController`.

As was mentioned previously, we need to layout a series of custom `UITableViewCell` prototypes in our storyboard. Figure 9.8 shows the three custom cell prototypes we've defined in the storyboard. By now, you should be familiar enough with Auto Layout to be able to figure out the constraints that are needed, but let's look briefly at how these prototypes are composed.

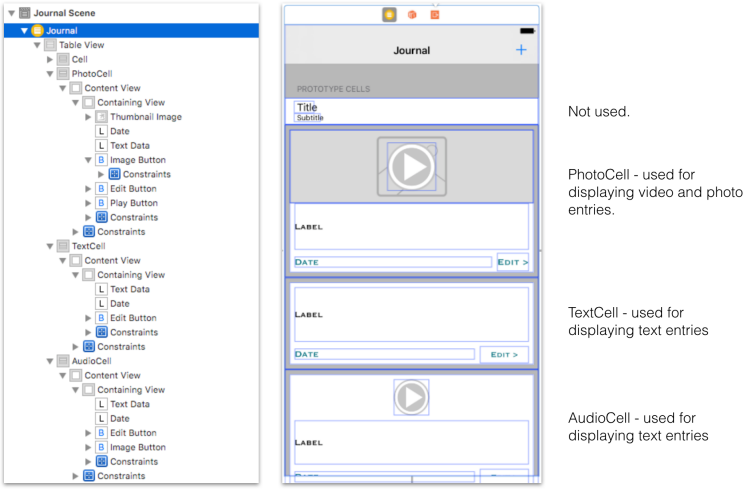


Figure 9.8: Layout of JournalTableViewController's custom cells in the storyboard.

The cell with identifier `PhotoCell` is actually used to display both photo and video journal entries. The only difference between the two is that we overlay a "play" button on top of the

thumbnail image in the case of video. If we are rendering a photo entry, we simply hide the button. The thumbnail image is actually rendered in the form of a UIButton, which of course is what will cause the corresponding media to to render when tapped. As can be seen in the structured view on the left of the image, each of the three cells as a label for the textual data, date, and in addition an edit button.

An IBAction is wired up to the edit buttons and image buttons. These actions are implemented by JournalTableViewController and perform the corresponding tasks. The interesting problem of course is that we will have a multiplicity of these cells in a given table view, so once tapped, how do we know which one has been tapped? A common approach in iOS is to simply tag the buttons with their row index. These actions and the helper functions they depend on are shown in Listing 93. Note how an `IndexPath` instance is created, keyed by the tag of the button that sent the event.

Since audio, video, and photo type cells all have image buttons, the `imageButtonPressed` action can be invoked by any of the three. As was already mentioned, photos are displayed via an off-the-shelf controller named `LightboxController` that is provided to us by the `Lightbox` pod. This controller will display an interactive full-screen view of the photo (double tab to zoom in/out, pinch to zoom functionality, tap to display caption, and swipe up to dismiss).

Video and audio entries are going to be rendered via an `AVPlayerViewController` instance. This component is provided to us via the iOS AVKit framework. The beauty of `AVPlayerViewController` is that it gives the standard and familiar iOS media player user interface, eliminating the need to roll our own. We hand the `AVPlayerViewController` and instance of `AVPlayer`. `AVPlayer` comes to us via the AVFoundation framework. It manges the playback and timing of media assets, and the real beauty of it for our purposes here is that it is capable of playing media via *local and remote URLs*. Recall in the previous section we were able to obtain Firebase URLs for our various media files once uploaded. Hence, to get an audio or video file that is resident out in the Firebase cloud, all we need to do is give the `AVPlayer` instance the URL and we're off to the races. This is what the `showContentOfUrlWithAVPlayer` helper method does in Listing 93.

To accomodate binding data values to our custom cells, a custom cell class `JournalEntryTableViewCell` has been introduced and is shown in Listing 94. We've defined outlets for every possible configuration, even though some of these will be unused for audio and textual entries. In addition to providing a helper function `setValues` to do data binding to the fields all entries have in common, we also do some cosmetic fine tuning in `awakeFromNib`. Basically, we're giving our cells those nice rounded corners, and making sure the aspect ratio on the images doesn't get messed up, as the images will be coming in an all different sizes. We'll look into this in more detail shortly when we implement a custom `CardView` class.

An extension is introduced to implement the methods defined by the `UITableViewDelegate` and the `UITableViewDataSource` protocols, and shown in Listing 95. Note that the extension declaration does not explicitly list these protocols in its definition. That is because the `JournalTableViewController` subclasses `UITableViewController` which already declares that it implements these protocols.

We only have a single section in the table, so `numberOfSections` returns a hard coded 1, while `tableview:numberOfRowsInSection` returns the number of elements in `self.entries`. The more interesting code of course is in `tableView:cellForRowAt`. We use the entry's type designator to determine the cell style to dequeue in lines 19–21. The edit button (and the image button if present) are tagged with the row index in lines 23–26. We provide the cell with that data

Swift

```swift
@IBAction func imageButtonPressed(_ sender: UIButton) {
  let row = Int(sender.tag)
  let indexPath = IndexPath(row: row, section: 0)
  let cell = self.tableView.cellForRow(at: indexPath) as! JournalEntryTableViewCell
  if let tnImg = cell.thumbnailImage {
    self.capturedImage = tnImg.image
  }
  self.entryToEdit = cell.entry
  if let entry = cell.entry {
    switch(entry.type!) {
      case .photo:
        self.showPhoto(image: self.capturedImage, caption: entry.caption)
      case.video:
        self.showContentOfUrlWithAVPlayer(url: entry.url)
      case .audio:
        self.showContentOfUrlWithAVPlayer(url: entry.url)
      \default: break
    }
  }
}

func showPhoto(image: UIImage?, caption: String?) {
  guard let img = image, let cap = caption else {
    return
  }
  let images = [
    LightboxImage(
      image: img,
      text: cap
    )
  ]
  let photoCtrl = LightboxController(images: images)
  photoCtrl.dynamicBackground = true
  present(photoCtrl, animated: true, completion: nil)
}

func showContentOfUrlWithAVPlayer(url : String) {
     if url == "" { return}
     let mediaUrl = URL(string: url)
     let player = AVPlayer(url: mediaUrl!)
     let playerViewController = AVPlayerViewController()
     playerViewController.player = player
     self.present(playerViewController, animated: true) {
       playerViewController.player!.play()
     }
}

@IBAction func editButtonPressed(_ sender: UIButton) {
     let row = Int(sender.tag)
     let indexPath = IndexPath(row: row, section: 0)
     let cell = self.tableView.cellForRow(at: indexPath) as! JournalEntryTableViewCell
     if let tnImg = cell.thumbnailImage {
       self.capturedImage = tnImg.image
     }
     self.entryToEdit = cell.entry
     self.captureType = (self.entryToEdit?.type)!
     self.performSegue(withIdentifier: "confirmSegue", sender: self)
}
```

Listing 93: IBActions implemented by JournalTableViewController and wired up to the image and edit buttons on the custom entry cells.

```swift
class JournalEntryTableViewCell: UITableViewCell {

  @IBOutlet weak var containingView: UIView!
  @IBOutlet weak var playButton: UIButton!
  @IBOutlet weak var date: UILabel!
  @IBOutlet weak var textData : UILabel!
  @IBOutlet weak var imageButton: UIButton!
  @IBOutlet weak var thumbnailImage: UIImageView!
  @IBOutlet weak var editButton: UIButton!
  var entry : JournalEntry?

  override func awakeFromNib() {
    super.awakeFromNib()

    if let button = self.imageButton {
      button.imageView?.contentMode = .scaleAspectFill
    }

    // round top corners of image, as the clip to bounds is set to false on containing
    // CardView, causing the corner to leak over the bounds without this workaround
    if let img = thumbnailImage {
      img.layer.cornerRadius = 3
      img.layer.maskedCorners = [.layerMinXMinYCorner, .layerMaxXMinYCorner]
    }
  }

  override func setSelected(_ selected: Bool, animated: Bool) {
    super.setSelected(selected, animated: animated)

    // Configure the view for the selected state
  }

  func setValues(entry : JournalEntry) {
    self.entry = entry
    self.textData.text = entry.caption
    self.date.text = entry.date?.shortWithTime
  }

}
```

Listing 94: A custom table view cell class for JournalTableViewController.

iOS

```
// MARK: - UITableViewDelegate and UITableViewDataSource
extension JournalTableViewController  {

    override func numberOfSections(in tableView: UITableView) -> Int {
        return 1
    }

    override func tableView(_ tableView: UITableView, numberOfRowsInSection
      section: Int) -> Int {
        return self.entries.count
    }

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

        let entry = self.entries[indexPath.row]
        let cellIds = ["NA", "TextCell", "PhotoCell", "AudioCell", "PhotoCell"]

        let cell = tableView.dequeueReusableCell(withIdentifier:
          cellIds[entry.type!.rawValue],
          for: indexPath) as! JournalEntryTableViewCell

        cell.editButton?.tag = indexPath.row
        if let imgButton = cell.imageButton {
            imgButton.tag = cell.editButton!.tag
        }

        cell.setValues(entry: entry)
        if let iv = cell.thumbnailImage {
            iv.image = UIImage(named: "landscape")
        }
        switch(entry.type!) {
        case .photo:
            let url = URL(string: entry.url)
            cell.thumbnailImage.kf.indicatorType = .activity
            cell.thumbnailImage.kf.setImage(with: url)
            cell.playButton.isHidden = true
        case .video:
            let url = URL(string: entry.thumbnailUrl)
            cell.thumbnailImage.kf.indicatorType = .activity
            cell.thumbnailImage.kf.setImage(with: url)
            cell.playButton.isHidden = false
            cell.playButton.tag = indexPath.row
        default: break
        }

        return cell
    }
}
```

Listing 95: An extension that provides the table view datasource/delegate protocol methods for JournalTableViewController.

for binding (line 28) and if an thumbnail image view is present on the cell we set it to our filler image (lines 29–31). Then, based on the type of entry we may need to fetch some images. Recall earlier that we added the Kingfisher pod to our workspace. This pod extends `UIImageView` that makes it possible for us to load an image by simply handing the image view an URL! As a bonus, we can even set an `indicatorType` which makes a nice little progress spinner show up while the image is downloading. After all the dust settles, and the images have arrived, Kingfisher will actually cache them for us. You'll notice this right away if you back out of the journal view back to the main view and then back into the journal view. The images load instantly!

Let's conclude this section with a nice cosmetic improvement to our journal entries. While we have customized the layout of the various entry cells in our storyboard, let's take a look at how we can add a nice card-like view for each entry, similar to what we get on the Android platform for free. When we say card-like view, we're basically talking about a thin shadowed border around each entry that visually clearly separates each of the entries. A really simple way to do this is to introduce a new subclass of `UIView` as is shown in Listing 96.

You'll notice we've used some annotations in our Swift code that we have not yet used. Basically, when defining a new `UIView` the `@IBDesignable` can be used to instruct Interface Builder to render the custom view on the canvas. This particular custom view will set some of our layer properties to make sure we have a clearly defined border and drop shadow displayed. The setters/getters annotated with `IBInspectable` inform Interface Builder that it should display the properties on the attribute inspector when used in an interface.

To utilize this custom class all we need to do is go back to our `JournalTableViewController` scene in our storyboard, and in each of our cell prototypes, change the class of the `containingView` in each to be `CardView` instead of the default `UIView`. Once that is accomplished, you can go the attribute inspector and set the corner radius, shadow opacity, and shadow offset. You can fiddle with these settings to get things to suit your fancy. Figure 9.9 provides a screenshot of the attribute inspector displaying these properties of the custom `CardView`. Remember however, in order to get these properties to display, the

Swift

```swift
@IBDesignable
class CardView: UIView {

  @IBInspectable var cornerRadius: CGFloat {
    get {
      return layer.cornerRadius
    }
    set {
      layer.cornerRadius = newValue
      layer.shadowRadius = newValue
      layer.masksToBounds = false
    }
  }

  @IBInspectable var shadowOpacity: Float {
    get {
      return layer.shadowOpacity
    }
    set {
      layer.shadowOpacity = newValue
      layer.shadowColor = UIColor.lightGray.cgColor
    }
  }

  @IBInspectable var shadowOffset: CGSize {
    get {
      return layer.shadowOffset
    }
    set {
      layer.shadowOffset = newValue
      layer.shadowColor = UIColor.darkGray.cgColor
      layer.masksToBounds = false
    }
  }

}
```

Listing 96: A useful subclass of UIView that provides a ”card-like” rendering of our journal entries. Set the custom class of any UIView in storyboard and use the the attributes inspector to set the defined properties.

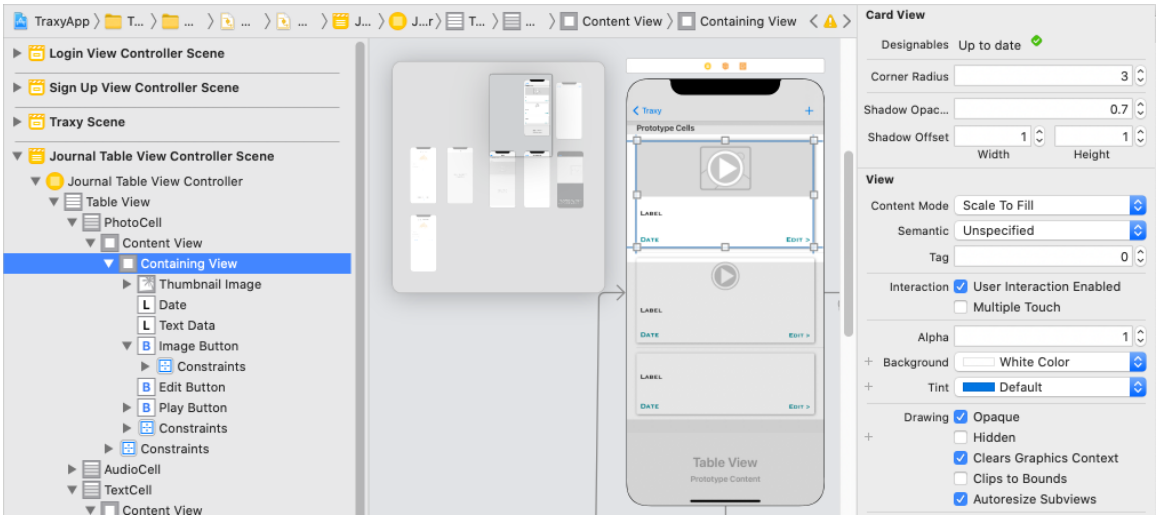


Figure 9.9: Manipulating the properties of the custom `CardView` class in Interface Builder's Attribute Inspector.

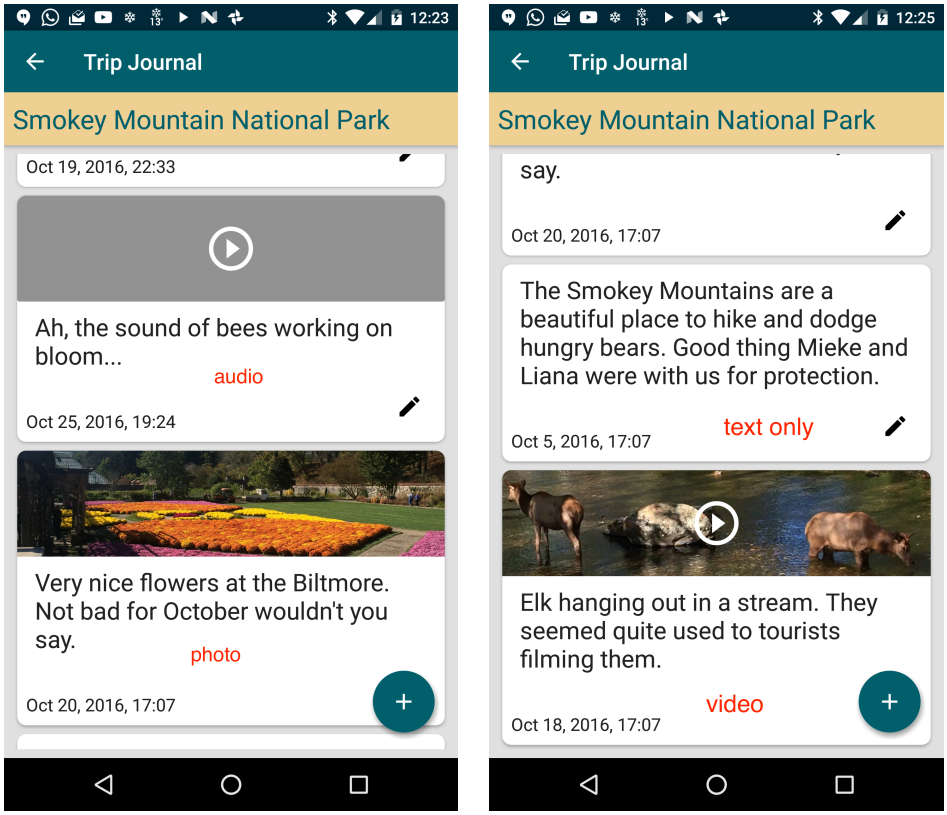


Figure 9.10: Trip Journal with Four Different Media Types

9.3 Multimedia in Android

Recall that in Section 6.3.4 you created `JournalDetailsFragment` for showing the details of a selected journal entry. We will replace this fragment with `JournalViewFragment` that will allow users to add photos, audio, videos, and textual entries to a journal. Remember to add this new fragment as a new destination in the navigation graph and add an action arrow from `MainFragment` to this one and call it `action_to_mediaList`.

The completed UI should look like Figure 9.10. The text view at the top, which is inserted as part of the `AppBarLayout`, shows the name of the selected trip ("Smokey Mountain National Park" in the screenshot). The list of journal media occupies the rest of the screen space below the app bar. The top part of each entry shows the media associated with that entry (if one exists). To render this list you will create another `RecyclerView` and an adapter, but that will be discussed towards the end of the chapter after your app has the necessary code for uploading the three media types to Firebase Cloud Firestore.

The transition from the main fragment to the new fragment is actually handled by `JournalAdapter`, however the adapter is currently designed to work with a listener function that takes only **one** String argument. To pass more information about the selected journal to the

rest of the UI, we will store the selected journal as a live data in the view model class. In addition, we have to change the listener function signature from

Kotlin
```
class JournalAdapter(val listener: ((String) -> Unit)? = null)
```

to

Kotlin
```
class JournalAdapter(val listener: ((Journal) -> Unit)? = null)
```

This update will trigger compile errors in other classes. For instance, the `bindTo` function in `JournalViewHolder` must be updated from

Kotlin
```
public fun bindTo(d: Journal, listener: ((String) -> Unit)?) {
  if (listener != null)
      parentView.setOnClickListener {
          listener(d.key)
      }
  // more code here
}
```

to

Kotlin
```
public fun bindTo(d: Journal, listener: ((Journal) -> Unit)?) {
  if (listener != null)
    parentView.setOnClickListener {
      listener(d)
    }
  // more code here
}
```

Using Navigation Safe Args

Before moving on further, this is perhaps the most appropriate time to introduce another feature of navigation graph. When a particular destination in the navigation graph requires input data to initialize its work, the data must be passed using the Android `Bundle` which essentially is a map (key-value pairs) whose keys are of type string. Unfortunately, misspelling the key name will result in failure to pass the data. And this type of misspelling bug would **not** be detected at compile time.

The SafeArgs library is intended to avoid this kind of bug and detect the mistake at compile time. The library requires an extra plugin and dependency:

1. Add the plugin in your app `build.gradle`:

Gradle
```
apply plugin: 'androidx.navigation.safeargs.kotlin'
```

2. Add the following library in your project `build.gradle`

Gradle
```
classpath "androidx.navigation:navigation-safe-args-gradle-plugin:2.3.0"
```

For the sake of explanation below a transition (action arrow) in the navigation graph involves the **originating** fragment/activity and a **destination** fragment/activity. The Safe Args plugin will automatically generate new Kotlin files based on the information parsed from your navigation graph.

- For each originating fragment/activity `MyOrigin` the plugin generates a Kotlin class `MyOriginDirections`. Further more, for each action arrow from that fragment/activity, the plugin generate an inner class of `MyOriginDirections`. Any action id in "kebab case" is converted to "snake case". For instance, an action arrow `my_action` from `MyOrigin` translate to an inner class `MyOriginDirections.myAction`. If the destination of `my_action` is a fragment (or activity) that define N arguments, then `MyOriginDirections.myAction` constructor takes N arguments as well. This class is mainly used by the source fragment/activity to navigate into a new destination (most likely inside a click listener) The same rule applies for action arrow that originates from an activity.
- For each destination/receiving fragment (or activity) `FragDest` that takes additional arguments the library generates a Kotlin class `FragDestArgs`. This class is mainly used by the receiving fragment to extract the argument (most likely inside its `onResume()`)

Let's begin using the plugin into practice.

In `LoginFragment.kt` you will find two `navigate()` calls. The first call that transitions into MainFragment

Kotlin
```
viewModel.userId?.let {
  findNavController().navigate(R.id.action_login2main)
}
```

can be replaced with

Kotlin
```
viewModel.userId?.let {
  val action = LocationFragmentDirections.actionLogin2main(it)
  findNavController().navigate(action)
}
```

To access the incoming argument passed from `LoginFragment`, `MainFragment.kt` can declare the following property delegation:

Kotlin
```
val args by navArgs<MainFragmentArgs>()
```

and the incoming user email can be accessed anywhere in `MainFragment` with the expression `args.userEmail` such as in the following Log call:

Kotlin
```
Log.d("Traxy", args.userEmail)
```

To manage media files from this activity, we will use a separate ViewModel class. Let's call this class `MediaViewModel`. Based on our discussion so far, the selected journal should be a good candidate of data item to keep in this class:

Kotlin
```
class MediaViewModel: ViewModel() {
  val selectedJournal = MutableLiveData<Journal>()
}
```

The above mutable live data should should be initialized when a particular journal is selected by the user.

Kotlin
```
adapter = JournalAdapter {
  mediaModel.selectedJournal.value = it
  val action = MainFragmentDirections.actionToMediaList()
  findNavController().navigate(action)
}
```

Let's now look at the UI design of `JournalViewFragment`. This landing screen will show a list of journal entries with media files, and this screen would be the center hub for all media related tasks. All other screens will transition from here.

With only one FAB at the lower right corner, it is impossible to distinguish which media type the user wants to add. You could add a dialog or another activity/fragment for the user to enter her choice, but that requires a significant amount of work for setting up a dialog or a new activity class. To avoid this extra screen interaction, you can take advantage of the following Material Design Guideline for "Speed Dial":

> The floating action button can fling out related actions upon press. The button should remain on screen after the menu is invoked.

Our design will be slightly different from the iOS counterpart where it employs Action Sheets to present a list of actions.

Before writing the actual code for adding media, you will first update the user interface to make the FAB fling out into more actionable FABs. Refer to Figure 9.11.

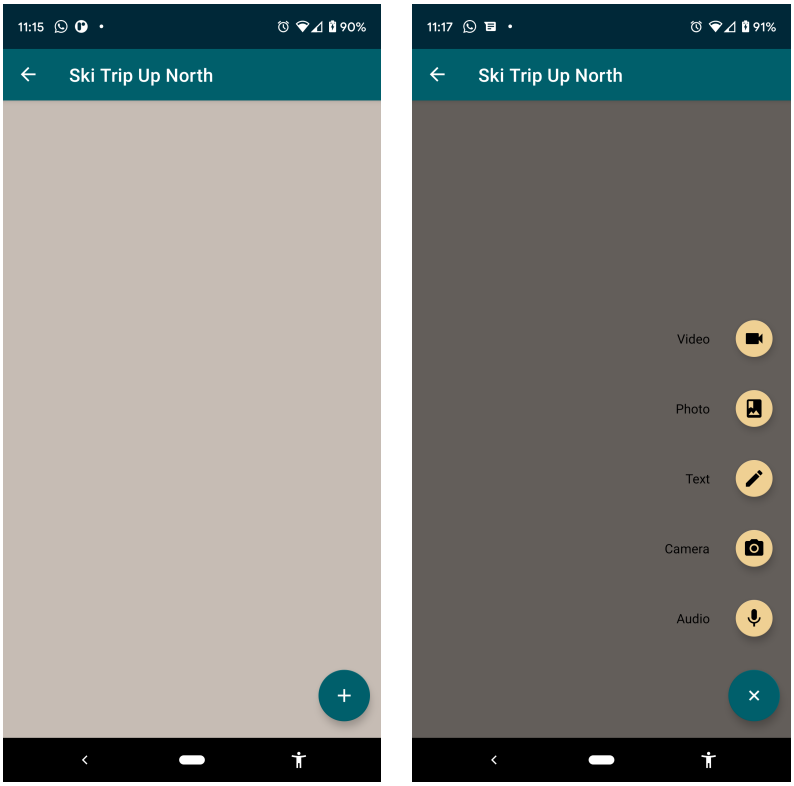
9.3.1 Expandable FABs

Implementation of this UI pattern is trivial when you use the right third-party library. There several options available, but for the moment we will be using The Material FAB Speed Dial https://github.com/kobakei/MaterialFabSpeedDial is among those that renders this fling animation. To use the library, add the following dependency to your app `build.gradle`:

Gradle
```
implementation 'com.github.kobakei:MaterialFabSpeedDial:2.0.0'
```

and add the following to your project `build.gradle`:

Gradle
```
allprojects {
repositories {
    // other repositories here
    maven { url "https://jitpack.io" }
  }
}
```



(a) Action Buttons Collapsed (b) Action Buttons Expanded

Figure 9.11: Expandable FloatingActionButton

The library creates an expandable FAB using two different resources: a FAB and a menu (for the expanded contents).

To use this button in your `JournalViewFragment` edit its layout file and include the following snippet:

XML
```
<io.github.kobakei.materialfabspeeddial.FabSpeedDial
  android:id="@+id/media_fab"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  app:fab_fabDrawable="@drawable/ic_baseline_add_24"
  app:fab_fabBackgroundColor="@color/colorPrimary"
  app:fab_miniFabBackgroundColor="@color/colorAccent"
  app:fab_fabDrawableTint="@android:color/white"
  app:fab_menu="@menu/menu_media" />
```

Some of the XML attributes above may need some explanation:

- All the attributes with prefix `fab_` are specific to the expandable FAB from the library above.
- `fabBackgroundColor` controls the background of the "parent" FAB (the main one that initially shows up at the lower right corner)
- `miniFabBackgroundColor` controls the background of the "children" (the expanded FABS displayed above the "parent" FAB)
- The menu resource file `menu_media.xml` to hold the menu structure shows below

XML
```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item android:id="@+id/add_audio"
    android:icon="@drawable/ic_baseline_mic_24"
    android:title="@string/audio_icon" />
  <item android:id="@+id/add_video"
    android:icon="@drawable/ic_baseline_videocam_24"
    android:title="@string/video_icon" />
  <item android:id="@+id/add_photo_from_camera"
    android:icon="@drawable/ic_baseline_camera_alt_24"
    android:title="@string/camera_icon" />
  <item android:id="@+id/add_photo_from_album"
    android:icon="@drawable/ic_baseline_photo_album_24"
    android:title="@string/photo_icon" />
  <item android:id="@+id/add_text"
    android:icon="@drawable/ic_baseline_edit_24"
    android:title="@string/edit_icon" />
</menu>
```

The five custom icons used in the menu are created from the following Material Icon Clip Art Sets: `av/mic`, `av/videocam`, `image/camera alt`, `image/photo album`, and `editor/mode edit`. The FAB click listener in `JournalViewFragment` should be defined as follows:

```Kotlin
media_fab.addOnMenuItemClickListener { miniFab, label, itemId ->
  when(itemId) {
    R.id.add_audio -> println ("Add audio")
    R.id.add_photo_from_camera -> println ("Add photo from camera")
    R.id.add_photo_from_album -> println("Add photo from album")
    R.id.add_video -> println ("Add video")
    R.id.add_text -> println("Add text only")
    else -> {

    }
  }
}
```

You are now ready to tackle the main task intended for this chapter: capturing photos and videos. We will cover audio recording in the next chapter.

9.3.2 Taking Thumbnails

Looking back to Section 4.4.9, you will notice that Android provides implicit intents for capturing photos and videos, so incorporating these tasks into your app should be straight forward. Capturing a thumbnail is just a temporary step to setup your code to work correctly with the Camera app. The final code will capture a full-size image which will be uploaded to Firebase Cloud Storage.

The main ingredient for taking photos is to invoke `startActivityForResult()` with an `Intent` initialized with action `MediaStore.ACTION_IMAGE_CAPTURE`. However, this implicit intent gives you two options, capture a thumbnail or a full-size image. The former is an easier task so we will address that one first.

Let's make the following changes to `JournalViewFragment`:

1. Define an integer constant `CAPTURE_PHOTO_REQUEST` initialize to a 16-bit value of your preference. This constant will uniquely identify the return result when the intent completed and your activity regained control inside its `onActivityResult` function

2. Add the following snippet in the `when` clause for `add_photo`:

```Kotlin
val capture = Intent(MediaStore.ACTION_IMAGE_CAPTURE)
capture.resolveActivity(requireActivity().packageManager)?.let {
  startActivityForResult(capture, CAPTURE_PHOTO_REQUEST)
}
```

3. To display the thumbnail image, you must provide an `ImageView` in your layout file. Notice that in the layout file `fragment_journal_view.xml` Android Studio may use an `<include>` element to include another layout file `content_journal_view`. In that case, edit the latter and add an `ImageView` as a child of the topmost parent, call it `photo_view`.

4. Override the `onActivityResult()` for capturing the thumbnail photo. Refer to Listing 97. On entry to this function, the parameter `requestCode` would be the value passed to the second argument of `startActivityForResult`. The snippet assumes that `photo_view` is the id of ImageView just added above.

Kotlin

```
override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?)
{
  when (requestCode) {
    CAPTURE_PHOTO_REQUEST -> {
      if (resultCode == Activity.RESULT_OK) {
        val thumbnail = data?.extras?.get("data") as Bitmap
        photo_view.setImageBitmap(thumbnail)
      }
    }
  else -> super.onActivityResult(requestCode, resultCode, data)
  }
}
```

Listing 97: Retrieving a Thumbnail Image from Image Capture Intent

Run the app and capture an image, you should see a preview of the thumbnail on your screen. If you are running the app on an emulator, be sure the cameras are not set to None. To confirm the settings, invoke the AVD Manager Tools > Android > AVD Manager. Under the "Actions" column you will fine the "Edit" button (Figure **??**). Invoke the editor on the emulator of your choice and select Show Advanced Settings and change the camera settings, if necessary.

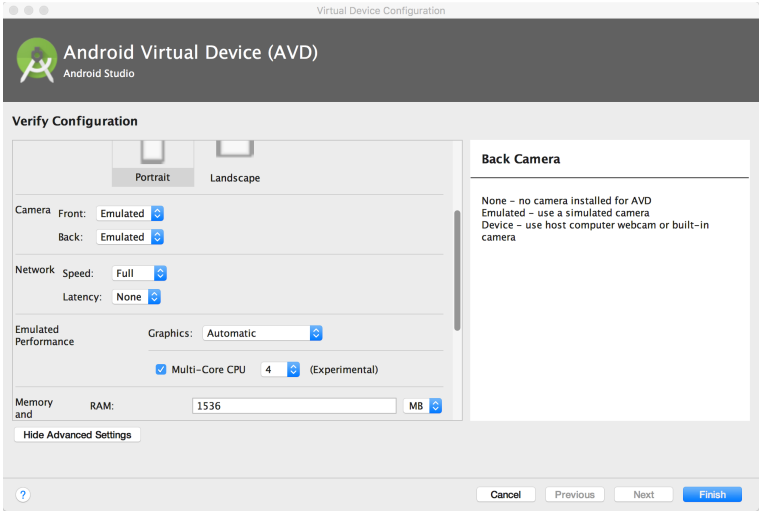


Figure 9.12: Advanced Emulator Settings

Now you know how to capture low resolution thumbnails, let's also learn how to capture full-size images, because in many situations, you will use both a thumbnail and a full-size photo. Thumbnails are suitable for displaying an iconic representation of the actual image, for instance inside a list item. In the next section you will explore a minor variation to the technique above, in order to capture a full-size photo.

9.3.3 Taking Full-Size Photos

The technique for capturing a full-size photo is pretty much similar to capturing a thumbnail photo. The only extra step is passing additional runtime argument to the `MediaStore Intent`. When taking a full-size image, most likely you want the image stored permanently on your device. Fortunately, the Android system camera automatically saves the image for you when you tell it the location on the device to save the image. However, you must also consider the following:

- your app may have to obtain permission to write to the storage on your device
- file name must be carefully chosen to prevent newly captured photos overwriting existing ones

We will address both of these issues in the following sections.

Understanding Android Permissions

You may be the only user of your Android device, but to the underlying Linux operating system, you are just one of the users. There are several other "users" sharing the device with you. Each installed application is considered another user and assigned a unique Linux uid and gid. Each app uid is permanently assigned at install time and is guaranteed to be unique within one device. Therefore, when five apps are currently running on your device, the underlying Linux OS thinks there are five individual users, and each user is isolated from the others.

By default, no Android applications (i.e. no Linux users) have permissions to perform operations that may adversely impact the privacy of the device owner (your privacy), the device performance, the system, or other applications.

Anytime you add a new contact (name, address, emails, phone numbers), the contact is saved in a file owned by your userid. Other apps that want to access your contact are treated as a different user by the Linux OS that must be given permissions to access your contact, because reading your contact information affects your privacy.

One of the many reasons you are using a mobile device is because you want to stay connected to your cyberspace. Nevertheless, you may be surprised to learn that connecting to the Internet requires permission. Why? Because it affects the device performance, i.e. battery drain.

To simplify the management of app permissions, Android defines two protection levels:

- *Normal* permissions: associated with operations that have very little risk to the user's privacy or other apps. Accessing the Internet falls under this category.
- *Dangerous* permissions: associate with operations that have higher risk to the user's privacy or other apps. Reading contacts falls under this category

When an app requires a **normal** permission, the system automatically grants it without user approval. However, an app requesting a **dangerous** permission must be explicitly approved by the user. Prior to Android 6.0 (Marshmallow), the approval to *all permission requests* must be granted at install time and the user must respond with "*All or None*". This may pose a dilemma for a user who is about to install an app because she wants to use only a subset of its features, but does not care about the other fancier features. For instance, what if Traxy is designed to allow friends to coauthor a shared journal. Implementation of this feature requires a "Read Contacts" permission.

For users who plan to use Traxy only for personal journaling, this feature does not apply and hence the user should not be prompted to approve this dangerous permission at install time.

On all versions of Android, the manifest must declare both normal and dangerous permissions. However, an app built using target SDK 23 (or higher) that is running on Android 6.0 must also request each dangerous permission at runtime. By strategically placing the permission request in your code, your app will prompt the user only when she is about to perform operations that require specific permissions.

File Organization in Android

Each app can store the files it creates at runtime under one of the following directories:

- *App specific* directories. Files under these directories are automatically deleted when the app itself is uninstalled. Directories under this category are further organized into two groups:
  - Private internal directories: files are accessible only to your app.
  - Public external directories: files are readable (Linux: world-readable) by other apps which have "Read External Storage" permission

  Because the above directories are specific to your app, storing files under any of them does not require any specific permission, except when you are running on API level 18 (Jelly Bean) or earlier.

- *User external* directories. Files under these directories are not directly associated with any app and will continue to linger even when the app that created them is uninstalled. No other apps can access the files under these directories, *unless* they are given permissions to do so.

  To understand this access policy, recall that under the Android security model, each app is assigned a unique Linux userid, and so is your Google account. An attempt by an app to access your files under the "User external" directories are treated by the underlying Linux as an attempt by uid X to access files owned by uid Y.

The actual path of these directories are provided by several different methods as shown in Table 9.2, and Table 9.3 summarizes the permission required by an app to access various storage types.

On Lollipop any files saved under `getExternalMediaDir()` are scanned by MediaStore API and made available to other apps.

After being introduced to a brief background on file organization and various possible locations for storing your files, you should be able to decide where to store your photos. Let's use the app's public external directories to store your photos, so other apps can potentially access them. Next, you have to use file naming scheme to prevent overwriting existing photos. One simple trick to generate unique file names is to include the current timestamp in the filename. The private method in Listing 98 generates a file whose name includes the current timestamp. Line 3 initializes a date formatter that, for instance, formats time and date 2:34:56p Aug 19, 2016 to "`20160819-143456`". Line 5 creates a new file under the app's external cache directory.

[1] only on Android 5.0 Lollipop
[2] Deprecated in API Level 29

Storage Type	Path Getter Method
App internal	`getFilesDir()` `getCacheDir()`

App external	`getExternalFilesDir()`
	`getExternalFilesDirs()`
	`getExternalCacheDir()`
	`getExternalCacheDirs()`
	`getExternalMediaDir()`[1]

User external	`getExternalStorageDirectory()`[2]

Table 9.2: Directories Accessible to Apps

Storage Type	API Level	
	Pre Kitkat ($\leq$ 18)	Post Jellybean ($\geq$ 19)
App's Internal	No	No
App's External	Yes	No
User External	Yes	Yes

Table 9.3: Storage Permission Required by an App

Kotlin

```kotlin
private fun createFileName(prefix: String, ext: String): File {
  val now = DateTime.now()
  val fmt = DateTimeFormat.forPattern("yyyyMMdd-HHmmss")
  val storageDir = requireContext().getExternalFilesDir(Environment.DIRECTORY_PICTURES)
  val media = File.createTempFile("${prefix}-" + fmt.print(now),
      ext, storageDir)
  return media
}
```

Listing 98: Using Timestamp for Generating a Unique File Name

To capture a full-size photo, file location for saving the photo must be passed as an URI (Uniform Resource Identifier). The diagram in Figure **??** shows that after a full-size photo is captured, the app transitions to a new screen to allow the user enter detailed information about the picture. This design calls for another fragment (`MediaDetailsFragment`) to be included as another destination in the navigation graph. In addition, this new fragment should be able to access the same URI used by `JournalViewFragment`. Since the URI is needed to update the UI, we should make it part of the view model. In addition, we will store the URI together with its associated file. Let's change `MediaViewModel.kt` to include a new mutable LiveData:

Kotlin

```
// In MediaViewModel.kt
val photoUri = MutableLiveData<Uri>()
val mediaFile = MutableLiveData<File>()
```

Since both `JournalViewFragment` and `MediaDetailsFragment` will require access to the view model, these fragments should define the following variable:

Kotlin

```
val mediaModel by activityViewModels<MediaViewModel>()
```

Prior to launching the camera app, `JournalViewFragment` should store the URI in the view model:

Kotlin

```
val capture = Intent(MediaStore.ACTION_IMAGE_CAPTURE)
capture.resolveActivity(packageManager)?.let {
  val photoFile = createFileName("traxypic", ".jpg")
  val photoUri = Uri.fromFile(photoFile)
  capture.putExtra(MediaStore.EXTRA_OUTPUT, photoUri)
  mediaModel.photoUri.value = photoUri
  mediaModel.mediaFile.value = photoFile
  startActivityForResult(capture, CAPTURE_PHOTO_REQUEST)
}
```

If your app is configured for target SDK 24 (Android 7) or higher, running your app at this point will give you a runtime error "File URI Exposed". To prevent this runtime error you must setup URI-based permission as described below.

URI-Based Permissions

The permission-by-directory approach seems to work adequately well, but it offers only all-or-none policy which may not be adequate for apps that require a more granular control. In an "all-or-none" environment, an app that is granted access to a specific directory will be allowed to access *all the files* under that directory. This policy is too broad and has a potential for misuse or abuse. A better approach is to allow access to individual files regardless of its location. To implement a more granular policy, your app should use the , `FileProvider` class, a content provider which can be configured to handle access control via the `Intent` mechanism.

To incorporate a system component like `FileProvider` you must declare it in your manifest as a `<provider>` tag. The attributes associated with the tag configure the details how the component fits your app's requirements.

Add a `<provider>` section in your manifest as a sibling node of your activities. Line 3 of Listing 99 shows an attribute `authorities` which is initialized to `${applicationId}.provider`.

The dollar sign notation refers to `applicationId` setting in your `build.gradle`, which is set to the app *package name*. The attribute `grantUriPermissions` indicates that the file provider will grant access permissions to file paths specified as `<meta-data>`. The actual paths are specified in another XML file `file_paths.xml` (Listing 100) that you must create manually. Use New > Android Resource Directory from the **res** folder to create a new XML directory. Then use New > XML Resource File from **res/xml** to create the path specification file.

XML

```
<provider android:name="android.support.v4.content.FileProvider"
  android:authorities="${applicationId}.provider"
  android:exported="false"
  android:grantUriPermissions="true">
  <meta-data android:name="android.support.FILE_PROVIDER_PATHS"
      android:resource="@xml/file_paths"/>
</provider>
```

Listing 99: Adding `FileProvider` to Manifest

The actual paths are specified in another XML file `file_paths.xml` (Listing 100) that you must create manually. Use New > Android Resource Directory from the **res** section of your project explorer and select "xml" as the resource type to create a new XML directory. Then use New > XML Resource File from **res/xml** to create `file_paths.xml`. Remove any other extra tags that may have been generated by Android Studio into the XML file. The root XML tag in this file is `<paths>`.

XML

```
<!--- res/xml/file_paths.xml -->
<?xml version="1.0" encoding="utf-8"?>
<paths>
  <!-- for getExternalFilesDir() -->
  <external-files-path name="traxy_photos" path="." />
</paths>
```

Listing 100: `FileProvider` Path Configuration

Refer to Listing 100. The root node of this XML file is `<paths>`. Under this root node, you can declare several children as shown in Table 9.4.

Tag	Associated Method
`<files-path>`	`getFileDir()`
`<cache-path>`	`getCacheDir()`
`<external-path>`	`getExternalStorageDirectory()`
`<external-files-path>`	`getExternalFilesDir()`
`<external-cache-path>`	`getExternalCacheDir()`
`<external-media-path>`	`getExternalMediaDirs()`

Table 9.4: XML Tags Associated with FileProvider Paths

For our case, the photo file paths are generated by the function `getExternalFilesDir()`, hence the child tag `<external-files-path>`. The value of the `path` attribute of these tags determines which **subdirectory** (under the parent directory associated with the tag) you are granting access to. For clarity of explanation, let's assume that the path returned by `getExternalFilesDir()` is `/storage/data/sample`. Using this hypothetical example, the external path specification at line 5 grants access to *all files* under `/storage/data/sample/.` directory (*notice the dot at the end of the path*). Had the specification changed to

XML
```
<external-files-path name="traxy_photos" path="captures" />
```

then it grants access to all the files under `/storage/data/sample/captures`.

Let's now modify the existing code to use a `FileProvider`. Instead of using `Uri.fromFile()` you must now invoke a new library method `FileProvider.getUriForFile()`. Be sure that the authority name passed as the second argument of the function `getUriForFile` matches the "authorities" attribute specified in the Android manifest.

Kotlin
```
val capture = Intent(MediaStore.ACTION_IMAGE_CAPTURE)
capture.resolveActivity(packageManager)?.let {
  val photoFile = createFileName("traxypic", ".jpg")
  val photoUri = FileProvider.getUriForFile(this,
      packageName + ".provider", photoFile)
  capture.putExtra(MediaStore.EXTRA_OUTPUT, photoUri)
  mediaModel.photoUri.value = photoUri
  mediaModel.mediaFile.value = photoFile
  startActivityForResult(capture, CAPTURE_PHOTO_REQUEST)
  }
```

Listing 101: Taking a Full-Size Picture (target SDK 24 or later)

To render the full-size image, we have to make minor changes to the code for rendering the thumbnail image as shown in a previous section. In the illustration shown at the beginning of this chapter, after an image is captured our app transitions to a new screen to allow the user enter detailed information about the image. We will create a new fragment and update the navigation graph.

1. Create a new fragment `MediaDetailsFragment` and add an image view in its layout call it `photo_view`

2. Edit the navigation graph `media_graph.xml`, add this new fragment as a new transition destination from `JournalViewFragment`. Be sure to also add the transition arrow and give it an id `action_to_mediaDetails`

3. In `onActivityResult` of `JournalViewFragment` replace the code for showing the thumbnail

```Kotlin
val thumbnail = data?.extras?.get("data") as Bitmap?
thumbnail?.let {
  photo_view.setImageBitmap(it)
}
```

with the following snippet to navigate into `MediaDetailsFragment`

```Kotlin
findNavController().navigate(R.id.action_to_mediaDetails)
```

4. Override `onResume` of `MediaDetailsFragment` and add the necessary code to render the full-size image captured from the camera

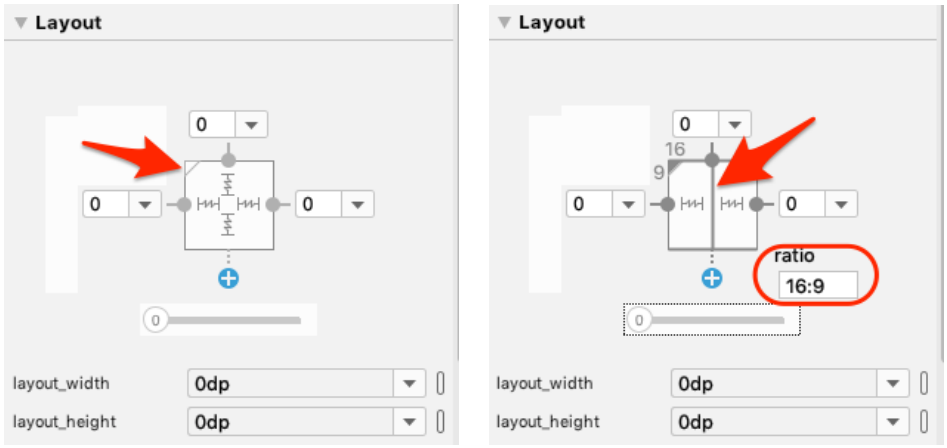
```Kotlin
mediaModel.photoUri.observe(viewLifecycleOwner) {
  val istream = requireContext().contentResolver.openInputStream(it)
  val bmp = BitmapFactory.decodeStream(istream)
  photo_view.setImageBitmap(bmp)
  istream?.close()

}
```

Run the app again to confirm that you are able to capture a full-size photo using this alternative technique. Sometimes, it is preferable to display a full-size image with certain aspect ratio. In the following section, we will enhance the UI layout to include this additional feature. The final design should look like Figure **??**c. The image at the top of the screen is constrained with aspect ratio set to 16:9.

1. Edit the XML layout of `MediaDetailsFragment`
2. Select the `ImageView` widget in the layout file. To set aspect ratio, one of its dimensions must be "fixed". For our particular case, let the image fills the available width and then apply the 16:9 ratio to constraint its height:
   - To make the image fills the screen width, we have to add two horizontal connections (one from its left edge to the screen left edge and from its right edge to the screen right edge)
   - To position the image close to the top of the screen, we have to add one vertical connection from its top edge to the screen top edge.
   - To use aspect ratio we have too set both its `layout_width` and `layout_height` to 0dp (match constraints).
   - Under the "Layout" tab, click the upper left corner of the constraint details view (See Figure 9.13a) to toggle the aspect ratio input box.

     Click the toggle (a) several times until you see a vertical solid line as shown in the right image (b) and enter the desired aspect ratio (16:9) in the input box (See Figure 9.13b).
3. To make sure the image fits the entire rectangle of the image view, you should set its `layout_gravity` to "center" and `scaleType` to "centerCrop"



(a) (b)

Figure 9.13: Setting Aspect Ratio of a Widget: (a) constraint toggle switch, (b) aspect ratio entry

NOTE

The aspect ratio control switch is actually a 3-state switch (not a boolean "toggle") that cycles through these three states: "None", "Width Constraint", or "Height Constraint". When the current state is either "Width" or "Height" a horizontal (or vertical) solid line is drawn in the widget constraint editor box. A solid vertical line indicates that the widget height is determined from its width using the ratio. Likewise, a solid horizontal line indicates that the widget width is calculated from its height.

Run the app, capture a photo using the system camera and the picture should appear on the screen. If you follow the instructions so far, `MediaDetailsFragment` shows only one image view, it is missing other fields needed to populate our database later. The UI views used on this screen are very similar to what you built earlier. Refer to Figure 7.5 on page 254. In addition, you will also be using date picker dialog and Google Places API calls to assist the user in entering journal entry date and location.

> **NOTE**
>
> In the sample code repository provided as the supplementary material of this book, we use a library to show an input dialog for both date and time. Therefore, we can merge the two input fields into one view, and change it from `EditText` to a `TextView` to prevent the user from editing the date/time text manually. The other input fields on the screen are for place selection (via the Google Places API) and caption text for the media file. Similarly, we also change the place selection from `EditText` to a `TextView`. The UI design and blueprint used by the sample code are shown in Figure 9.14. To place the FAB vertically center across the bottom edge of the image, you must add two new connections from its north and south anchor points to the bottom edge of the image.

9.3.4 Selecting Photos from Gallery

Selecting pictures from photo gallery accessible on your Android device your app will use another system intent (`ACTION_PICK`). Similar to lauching intent for capturing a full-size photo or video, you can use the following snippet in one of the code blocks for handling the expandable FAB menu:

Kotlin

```
val photoGallery = Intent(Intent.ACTION_PICK)
photoGallery.type = "image/*"
startActivityForResult(Intent.createChooser(photoGallery, "Select Photo"),
    GALLERY_PHOTO_REQUEST)
```

In response to your user action of selecting picture(s) from the gallery, you must update the function `onActivityResult` to handle `GALLERY_PHOTO_REQUEST`:

Kotlin

```
when(requestCode) {
  CAPTURE_PHOTO_REQUEST, CAPTURE_VIDEEO_REQUEST -> {
    /* code you added earlier */
  }
  GALLERY_PHOTO_REQUEST -> {
    if (resultCode == RESULT_OK) {
      data?.clipData?.let {
        val photoUri = it.getItemAt(0).uri
        mediaModel.photoUri.value = photoUri
        mediaModel.mediaFile.value = File(photoUri.path)
        findNavController().navigate(R.id.action_to_mediaDetails)
      }
    }
  }
}
```

9.3.5 Uploading Media Files to Cloud Storage

Now let's write necessary code to upload the photo to your Firebase instance. We will continue to use your Cloud Firestore using a similar technique you learned in Section **??**. However, we can't

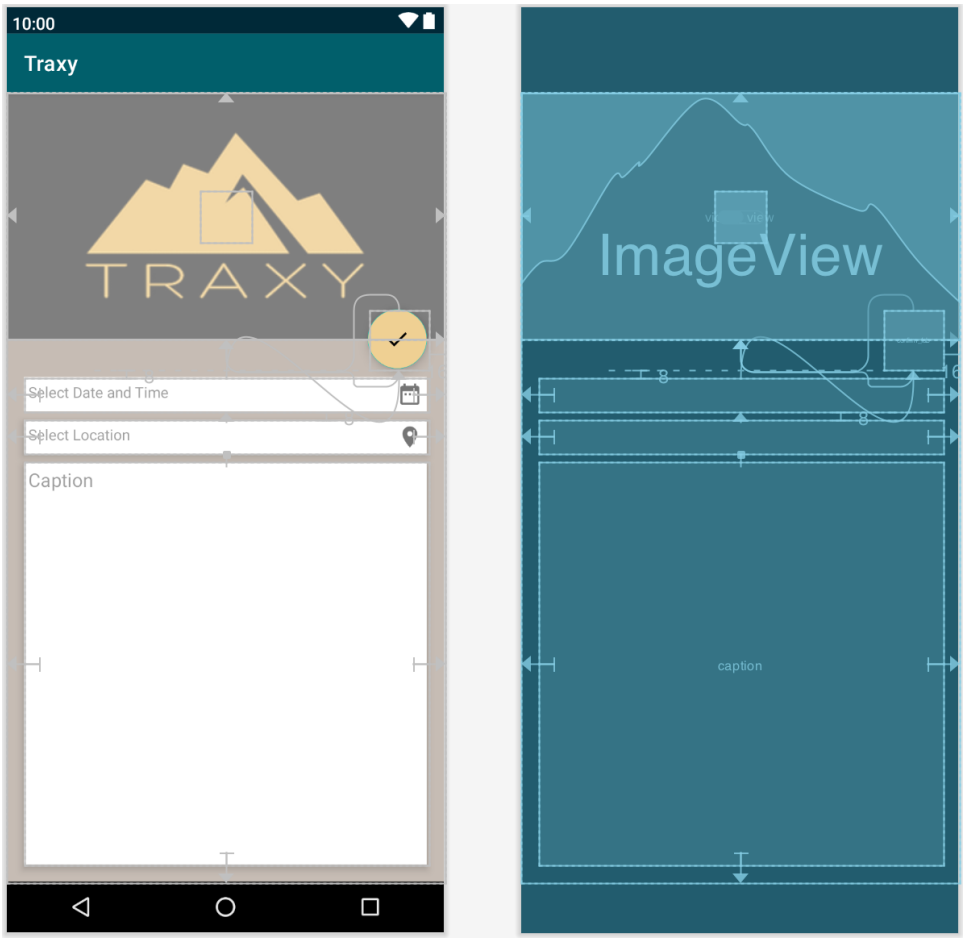


Figure 9.14: UI Design and Blueprint of Media Details

store bitmap images directly in Cloud Firestore. The list of supported data type does not include images. To store media files you must use Firebase *Storage*. The APIs for using Firebase Storage are similar to that of Firebase Database/Cloud Firestore. You should feel at home writing code for uploading or downloading data from Firebase Storage. You can access the digital contents of Firebase Storage using the APIs or download them using a networking library via the URL associated with each storage item. To access the storage from your Android app, you must include its dependency in your app's `build.gradle`, be sure to match the version number *x.y.z* with the other Firebase libraries in your project.

Gradle

```
// in app build.gradle
compile 'com.google.firebase:firebase-storage-ktx:19.2.0'
```

The actual code for uploading files to Cloud Firestore will be delegated to the repository class. However we now have to ViewModel classes referring to one instance of the repository. Hence we

have to change the repository class into a singleton by changing its declaration from a class into an object:

Kotlin
```
object TraxyRepository {
  /* same content as before */
}
```

But this update requires minor changes to the first view model class we use earlier. For instance, the following snippet

Kotlin
```
val repo = TraxyRepository()
repo.firebaseSignInWithEmail(email, password)
```

must be updated to

Kotlin
```
TraxyRepository.firebaseSignInWithEmail(email, password)
```

Go ahead and make necessary changes to the other functions calls.

To upload journal entries to Cloud Firestore you must first declare the class associated with the object in Listing 102. The `url` field stores the Firebase Storage URL of the media file associated with a journal entry. The `thumbnailUrl` field stores the URL to the thumbnail image of a video entry. For text only entries (`type`:1), the value of the `url` field is ignored. *The Kotlin data class must provide initializer value for each instance variable*, this is because the Firebase methods for converting data to local objects require a constructor with no arguments.

Ideally, the class should be defined as follows:

Kotlin
```
enum class MediaType { NONE, TEXT, PHOTO, AUDIO, VIDEO }

data class JournalMedia(
  var _key: String = "",
  var caption: String = "",
  var date: String = "",
  var lat: Double = 0.0,
  var lng: Double = 0.0,
  var type: MediaType = MediaType.NONE, // enum type
  var url: String = "",
  var thumbnailUrl: String = ""
)
```

However, using pure `enum` type will add more work in storing/retrieving data from Cloud Firestore. So as a trade-off, we will replace `MediaType` with —Int— and calls the `ordinal` property to set the media type:

Kotlin
```
val newItem = JournalMedia()
newItem.type = MediaType.TEXT.ordinal
```

The final design that we will use for storing to Cloud Firestore is shown in Listing 102.

NOTE

In this chapter, the two geo location properties (`lat` and `lng`) will not be used for storing actual geo coordinates. They will be used in future chapters for fetching weather data and for rendering the journal location on a map.

Kotlin

```kotlin
enum class MediaType { NONE, TEXT, PHOTO, AUDIO, VIDEO }

data class JournalMedia(
  var _key: String = "",
  var caption: String = "",
  var date: String = "",
  var lat: Double = 0.0,
  var lng: Double = 0.0,
  var type: Int = 0,  // use Int in place of enum
  var url: String = "",
  var thumbnailUrl: String = ""
)
```

Listing 102: `JournalMedia` Class for Cloud Firestore

Storing the media to Firebase requires the following two-step process:

1. Persist the media contents to Firebase *Storage*, a step that may last up to several seconds, depending on the size of the media file. Upon completion of the upload, Firebase provides the URL to the media

2. Persist the journal media object to Cloud Firestore, a much faster step that requires, on average, only milliseconds to complete. The URL obtained from the previous step is then stored inside the journal entry

Our general strategy is to implement the each of the steps above as a function in `TraxyRepository`

1. `uploadMediaFile()`. Upon successful file upload this function returns the URL to the file in Cloud Firestore. This function will be implemented as a co-routine

2. `addMediaEntry()` add a new journal media entry to Cloud Firestore. In order to associate the media with this entry, the URL returned by the above function is recorded in the journal media entry

Compared to writing data to Firebase Database or Cloud Firestore, uploading digital contents to Firebase Storage requires a slightly longer code. The very first step is obtaining a reference to the Firebase storage instance. In our Firebase database design, user data are partitioned based on the user's UID. We will follow the same approach for partitioning the associated Firebase storage. Therefore, the Firebase storage reference must be initialized using the code snippet in Listing **??** after the user successfully logged in. Implementation of this feature requires minor changes to our repository class.

1. Declare the following two new variables:

```Kotlin
// in TraxyRepository.kt
private val storageRef = Firebase.storage
private var userMediaStore: StorageReference? = null
```

2. Initialize `userMediaStore` after the user successfully logged in by updating the `firebaseSignInWithEmail` function

```Kotlin
val z = auth.signInWithEmailAndPassword(email, password).await()
if (z.user != null) {
  docRef = dbStore.document("user/${z.user!!.uid}")
  userMediaStore = storageRef.getReference(z.user!!.uid)
}
```

3. Add the following two functions:

```Kotlin
suspend fun uploadMediaFile(mediaUri: Uri, mediaFile:File): String? {
  // details will be added later
}

fun addMediaEntry(key: String, m: JournalMedia) {
  // details will be added later
}
```

The last two functions will be invoked from `MediaViewModel` by the following function:

```Kotlin
  suspend fun addMediaEntry(media:JournalMedia, mediaUri: Uri) {
    val url = TraxyRepository.uploadMediaFile(mediaUri, mediaFile.value!!)
    if (url != null)
        media.url = url
    TraxyRepository.addMediaEntry(journalKey.value?: "NO-KEY", media)
}
```

> **NOTE**
>
> It is important to implement `uploadMediaFile` as a Kotlin co-routine so we don't have to deal with writing callback function for handling the upload completion/error.

Firebase Storage provides three main methods for uploading binary contents: `putStream()`, `putBytes()`, or `putFile()`. For instance, to upload a file on your device `img1234.jpg` and store it on Firebase Storage as `photos/personal/profile.jpg`, you would use the following snippet:

```Kotlin
val myFile = File("img1234.jpg")
val fileUri = Uri.fromFile (myFile)
// storageRef is a reference to Firebase storage
storageRef.child("photos/personal/profile.jpg").putFile(fileUri)
```

The complete code for uploading media file to Firebase Storage is shown in Listing 103. The `await()` function is a Kotlin extension function that transforms asynchronous callback into Kotlin suspend. Hence making our code look like synchronous calls.

Kotlin

```kotlin
suspend fun uploadMediaFile(mediaUri: Uri, mediaFile:File): String? {
  userMediaStore?.let {
    val fileName = mediaUri.lastPathSegment
    val mediaRef = it.child("photos/").child(fileName!!)
    val mediaTask = mediaRef.putFile(mediaUri)
    mediaTask.await()
    mediaFile.delete()
    return mediaRef.downloadUrl.await().toString()
  }
  return null
}
```

Listing 103: Upload Media File to Firebase Storage

The complete code for adding media entry to Cloud Firestore is shown in Listing 104.

Kotlin

```kotlin
fun addMediaEntry(key: String, m: JournalMedia) {
  docRef?.let {
    var mediaData = hashMapOf(
      "type" to m.type.ordinal,
      "caption" to m.caption,
      "date" to m.date,
      "url" to m.url,
      "lat" to m.lat,
      "lng" to m.lng
    )
    if (m.type == MediaType.VIDEO.ordinal)
      mediaData["thumbnailUrl"] = m.thumbnailUrl
    it.collection("journals")
      .document(key)
      .collection("media")
      .add(mediaData)
  }
}
```

Listing 104: Adding Journal Media Entry to Cloud Firestore

The function `addMediaEntry` in `MediaViewModel` will be called from `MediaDetailsFragment` in response to pressing the "Save" button as shown in Listing 105. However this function requires an object of type `JournalMedia` which is initialized from three user input fields: event date, event location (set by Google Places API), and event caption. These three input fields should be associated with LiveData in `MediaViewModel`:

Kotlin

```kotlin
val mediaDate = MutableLiveData<DateTime>()
val mediaCaption = MutableLiveData<String>()
```

Kotlin
```kotlin
// in MediaDetailsFragment (in response to "Save")
var mType = MediaType.TEXT
mediaUri?.lastPathSegment?.let {
  when {
    it.endsWith(".mp4") -> mType = MediaType.VIDEO
    else -> mType = MediaType.PHOTO
  }
}
CoroutineScope(Dispatchers.IO).launch {
  val mediaObj = JournalMedia(
      caption = caption.text.toString(),
      date = date_time.text.toString(),
      type = mType.ordinal,
      lat = mediaModel.selectedJournal.value?.lat ?: 0.0,
      lng = mediaModel.selectedJournal.value?.Lng ?: 0.0)
  mediaModel.addMediaEntry(mediaObj, mediaUri!!)
}
findNavController().popBackStack()
```

Listing 105: Uploading Media Files to Firebase Storage

Rerun the application and verify that your files are uploaded to Cloud Firestore and Firebase Storage by inspecting your Firebase console. Navigate through the directory structure until you find the JPG file of interest.

Using the new knowledge you acquired for capturing photos, you will now replicate most of the steps above for handling videos. Most of the steps are very similar, you can probably replicate most of the functionalities by copy and paste, then replace a few lines here and there.

9.3.6 Recording Videos and Playback

Recording a video is also achieved using an implicit `Intent` and the necessary code to perform the task is shown in Listing 106 which is a minor variation of Listing 101. Go ahead and use the code for the menu selection handler of your "Media FAB". Since we are going to use the Uri variable in our `MediaViewModel` to hold metadata for photos and videos, it seems appropriate to rename the variable `photoUri` to `mediaUri`

Kotlin
```kotlin
val capture = Intent(MediaStore.ACTION_VIDEO_CAPTURE)
capture.resolveActivity(requireActivity().packageManager)?.let {
    val videoFile = createFileName("traxypic", ".mp4")
    mediaModel.mediaFile.value = videoFile
    val videoUri = FileProvider.getUriForFile(requireContext(),
        "${requireActivity().packageName}.provider", videoFile)
    capture.putExtra(MediaStore.EXTRA_OUTPUT, videoUri)
    mediaModel.mediaUri.value = videoUri
    startActivityForResult(capture, CAPTURE_VIDEO_REQUEST)
}
```

Listing 106: Recording a Video (targetSdk 24 or later)

The above code snippet requires you declare a new constant `CAPTURE_VIDEO_REQUEST`.

The `onActivityResult()` method of `JournalViewFragment` should now be redesigned to handle the result of capturing photos and videos. Moreover we can reuse `MediaDetailsFragment` for the user to enter the event details associated with the video. To make this works, you must add `CAPTURE_VIDEO_REQUEST` as another branch condition of the `when` statement. Simply change

Kotlin

```
when(requestCode) {
  CAPTURE_PHOTO_REQUEST -> {

  }
}
```

to

Kotlin

```
when(requestCode) {
  CAPTURE_PHOTO_REQUEST, CAPTURE_VIDEO_REQUEST -> {

  }
}
```

To render the video in `MediaDetailsFragment`, you must add a `VideoView` widget to its layout file. But currently the layout already includes an `ImageView`. How should you arrange the two widgets, `ImageView` and `VideoView`, when you need only *one of them at a time*?

Recall that, as discussed in Section 5.3.1, `FrameLayout` stacks up its children along the Z-axis potentially overlapping each other. Apparently, this is a perfect layout to place both widgets to occupy the same space. By applying the aspect ratio constraint 16:9 to the `FrameLayout` itself and setting the width and height of both children to `match_parent`, then both `ImageView` and `VideoView` are automatically constrained to the parent aspect ratio. The overall idea is shown in the snippet in Listing 107. Be aware that for brevity of presentation, all the other non-relevant attributes are intentionally omitted. In our new layout file the `FrameLayout` is an immediate child of `ConstraintLayout`, and there are many other attributes required for its proper placement not shown in the snippet. The `layout_gravity` attribute on the two child widgets guarantee that when the image/video does not fill the available space, it will stay centered.

Make necessary changes to `fragment_media_details_entry.xml` to incorporate this technique. It is recommended that you modify the XML file while editing it in "Text" mode (as opposed to "Design" mode).

XML

```xml
<!-- this FrameLayout is a direct child of its ConstraintLayout parent -->
<FrameLayout
  android:layout_width="0dp"
  android:layout_height="0dp"
  app:layout_constraintDimensionRatio="h,16:9"
  ...other-attributes-not-shown...
  >
  <ImageView android:id="@+id/photoView"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_gravity="center"/>
  <VideoView android:id="@+id/videoView"
    android:layout_height="match_parent"
    android:layout_width="match_parent"
    android:layout_gravity="center" />
</FrameLayout>
```

Listing 107: Using `FrameLayout` as a container for both `ImageView` and `VideoView`

Since both the `ImageView` and `VideoView` widgets occupy the same space, one of them must be made invisible when you are using the other to render a media. For instance, you can use the following code snippet when rendering a photo:

Kotlin

```kotlin
photoView.visibility = View.VISIBLE
videoView.visibility = View.INVISIBLE
```

The code block for observing `mediaUri` in the `onResume()` of (`MediaDetailsFragment`), must be updated to handle both photos and videos as shown in Listing 108

Kotlin

```kotlin
mediaModel.mediaUri.observe(viewLifecycleOwner) {
  mediaUri = it
  confirm_fab.isEnabled = true
  if (it.lastPathSegment!!.endsWith(".mp4")) {
      video_view.setVideoURI(it)
      video_view.start()
      val mc = MediaController(requireContext())
      video_view.setMediaController(mc)
      video_view.visibility = View.VISIBLE
      photo_view.visibility = View.INVISIBLE
  } else {
      val istream = requireContext().contentResolver.openInputStream(it)
      val bmp = BitmapFactory.decodeStream(istream)
      photo_view.setImageBitmap(bmp)
      photo_view.visibility = View.VISIBLE
      video_view.visibility = View.INVISIBLE
      istream?.close()
  }
}
```

Listing 108: Rendering Photos or Videos

To allow minimal playback control of the video, we can attach a media controller to the video

view widget as shown in the above snippet. When the user taps on the video, a basic playback control will appear below the video.

The necessary code for uploading your video to Firebase storage will look similar to Listing 105, except videos should be stored under the `"videos"` subdirectory. The updated code is shown below:

Kotlin

```
suspend fun uploadMediaFile(mediaUri: Uri, mediaFile:File): String? {
  userMediaStore?.let {
    val fileName = mediaUri.lastPathSegment!!
    val dirName = if (fileName.endsWith(".mp4")) "videos/" else "photos/"
    val mediaRef = it.child(dirName).child(fileName)
    val mediaTask = mediaRef.putFile(mediaUri)
    mediaTask.await()
    mediaFile.delete()
    return mediaRef.downloadUrl.await().toString()
  }
  return null
}
```

In addition to storing the entire video, you should also store a thumbnail image of the video to Firebase Storage. The thumbnail image will be used by a `RecyclerView` adapter (later in the chapter) for displaying a preview of the video.

To implement this additional requirement, we have to make minor changes to both `MediaViewModel` and `TraxyRepository`.

1. The function `addMediaEntry` in `MediaViewModel` should include additional logic to extract a thumbnail image from the video frames when the incoming media is a video file. The updated function is shown in Listing 109

2. A new function `uploadMediaContent` must be added to `TraxyRepository`. This (suspend) function will be called from the view model function above

Kotlin
```
suspend fun addMediaEntry(media: JournalMedia, mediaUri: Uri) {
  mediaFile.value?.let {
    // When uploading videos we use the first frame as its thumbnail
    // We must upload the thumbnail before the video file is
    // DELETED by uploadMediaFile (below)
    if (it.path.endsWith(".mp4")) {
      // Get the first frame
      val metadataRetriever = MediaMetadataRetriever()
      metadataRetriever.setDataSource(it.absolutePath)
      val thumbImg:Bitmap = metadataRetriever.getFrameAtTime(0)
      metadataRetriever.release()

      // Specify the path to store it in the cloud
      val thumbImgPath = "photos/" + it.name
          .replace(".mp4", "-thumb.jpg")
      // Convert to byte array and upload
      val byteOutputStream = ByteArrayOutputStream()
      thumbImg.compress(Bitmap.CompressFormat.JPEG,
          100, byteOutputStream)
      TraxyRepository.uploadMediaContent(byteOutputStream.toByteArray(),
          thumbImgPath)?.also {
        media.thumbnailUrl = it
      }
    }
    TraxyRepository.uploadMediaFile(mediaUri, it)?.also {
      media.url = it
    }
  }
  TraxyRepository.addMediaEntry(journalKey.value ?: "NO-KEY", media)
}
```

Listing 109: Additional code to extract thumbnail from video

Since the updated function also depends on another function `uploadMediaContent`, we have to add that function to the repository class:

Kotlin
```
suspend fun uploadMediaContent(media:ByteArray, pathName:String): String? {
  userMediaStore?.let {
    val mediaRef =it.child(pathName)
    val mediaTask = mediaRef.putBytes(media)
    mediaTask.await()
    return mediaRef.downloadUrl.await().toString()
  }
  return null
}
```

Run the app, capture a video and confirm that the corresponding MP4 file and its thumbnail are uploaded to Firebase Storage. Also verify that the `url` and `thumbnailUrl` fields of the corresponding entry in Firebase Cloud Firestore are set correctly.

Journal Media List

After you are able to upload some media contents to Firebase Storage, it is now the right time to add more logic into `JournalViewFragment` to render the list of journal media entries. You can

use the same approach using `RecyclerView` as discussed in Section 6.3. However, instead of writing your own adapter, we will introduce a new library for completing this task.

The FirebaseUI-Android GitHub project provides the following three modules for Android apps to interact with Firebase:

- `firebase-ui-database`
- `firebase-ui-firestore`
- `firebase-ui-auth`
- `firebase-ui-storage`

The name of each module is self-explanatory, and you will be incorporating the first module into your app. As with using the other libraries, the very first step is to add the following dependency to your app's `build.gradle`:

Gradle
```
implementation 'com.firebaseui:firebase-ui-firestore:6.3.0'
implementation 'com.firebaseui:firebase-ui-storage:6.3.0'
```

The Firebase-UI library can work seamlessly with the `RecyclerView` in your UI. Each journal media entry will be rendered using the layout in Figure 9.15. The top portion holds two image views in a `FrameLayout` so they can be stacked together. The containing `FrameLayout` has a fixed height of 96dp and a width of `match_parent`. The larger gray rectangle is an image view (`topImage`) with its width and height set to `match_parent`. The smaller square (with a circled play icon) is rendered with a transparent box (used earlier for rendering journal name in the journal list). Essentially we are repeating a similar technique used earlier in this chapter. Go ahead and create a new XML layout (`media_item`) based on Figure 9.15 and the description above.

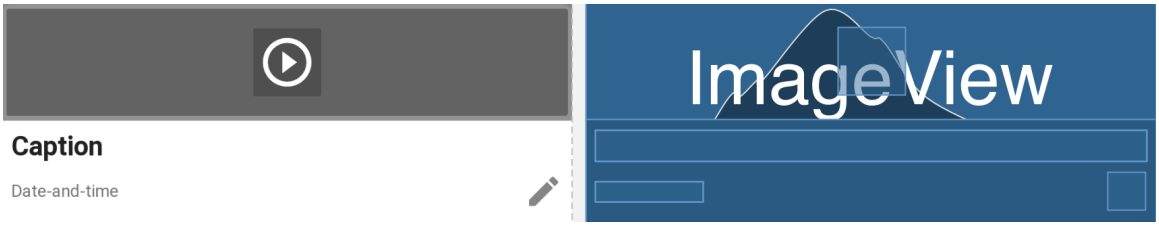


Figure 9.15: XML Layout of Journal Media

Next we have to a `RecyclerView` to your `JournalViewFragment` layout. Make the following changes to incorporate the new layout file into your app:

1. Edit `fragment_journal_view.xml` and add a `RecyclerView` and call it `media_list`.
2. To see a preview of the item layout on the recycler view, add the following attribute to the recycler view:

XML
```
tools:listitem="@layout/media_item"
```

Any recycler view requires a view holder class for binding the model data to the UI. We will create a new class `JournalMediaViewHolder` for this purpose. For now we will focus only the textual elements.

Kotlin

```
class JournalMediaViewHolder(v: View): RecyclerView.ViewHolder(v) {
  val caption:TextView
  val dateTime:TextView

  init {
    caption = v.findViewById(R.id.media_caption)
    dateTime = v.findViewById(R.id.media_date_time)
  }

  public fun bindTo(m: JournalMedia) {
    caption.text = m.caption
    dateTime.text = m.date
  }
}
```

Recall that a RecyclerView needs an adapter to pull data from the Cloud Firestore and Firebase Storage. We will call this class `JournalMediaAdapter`. Fortunately, the Firebase-UI package provides an abstract adapter class that you can inherit from.

Kotlin

```
class JournalMediaAdapter(opt:FirestoreRecyclerOptions<JournalMedia>):
  FirestoreRecyclerAdapter<JournalMedia,JournalMediaViewHolder>(opt) {

  override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
   JournalMediaViewHolder {
      val view = LayoutInflater.from(parent.context)
          .inflate(R.layout.media_item, parent, false)
      return JournalMediaViewHolder(view)
  }

  override fun onBindViewHolder(
      holder: JournalMediaViewHolder,
      position: Int,
      model: JournalMedia,
  ) {
      holder.bindTo(model)
  }
}
```

The constructor of the adapter class takes an option object that can be used to direct the class where to pull the data source. One of its configuration options is the Firebase `Query` object that encodes the path to the Cloud Firestore document/collection. In the previous chapters we depend on the view model and repository classes to handle the interactions with the data store and we will continue this strategy here. In our application, the path to the media collection in Cloud Firestore is determined by the userid and the journal entry that owns the media files and we have the following situation to deal with:

- The adapter must be initialized by the view layer
- The journal key is managed by the view model layer

- The userid is managed by the repository

Based on the current settings, we will use the following strategy:

1. Add a function in the repository class that takes a journal key (String) and returns a Firebase query

Kotlin
```
fun getMediaQuery(key:String): Query {
  val userId = auth.currentUser?.uid ?: "NONE"
  return dbStore.collection("user/$userId/journals/$key/media")
}
```

2. Add a function in the view model class that calls the above function

Kotlin
```
fun mediaQuery() : Query {
  return TraxyRepository.getMediaQuery(journalKey.value!!)
}
```

3. In `JournalViewFragment` declare the recycler adapter `adapter` as a `lateinit` variable:

Kotlin
```
lateinit var adapter: FirestoreRecyclerAdapter<JournalMedia,JournalMediaViewHolder>
```

4. Use the new view model function above to initialize the necessary recycler adapter inside `onViewCreated` of `JournalViewFragment`

Kotlin
```
mediaModel.journalKey.observe(this.viewLifecycleOwner, Observer {
  val option = FirestoreRecyclerOptions.Builder<JournalMedia>()
     .setLifecycleOwner(viewLifecycleOwner)
     .setQuery(mediaModel.mediaQuery(), JournalMedia::class.java)
     .build()
  adapter = JournalMediaAdapter(option)
  media_list.adapter = adapter
  media_list.layoutManager = LinearLayoutManager(requireContext())
})
```

Recompile and run you app and select a journal entry in which you add photos or videos. You should see a list of journal media entries showing the caption and date. In the next section, you will make improvements to the layout and contents of the journal media entries so they will show more than just textual information.

Fetching Media Files

To render the photo or video thumbnail on the card view they must first be fetched from Firebase Storage. Technically, it requires you send an HTTP GET request to Firebase, handle the result in background and render the image on the UI. Fortunately, you can use the Glide library that handles the image fetching and rendering using only a few lines of code. The following Kotlin annotation processes must be included in your app's `build.gradle` to use the Glide library.

Gradle
```
kapt 'com.github.bumptech.glide:compiler:4.11.0'
```

But let's update the `JournalMediaViewHolder` by adding more variables:

Kotlin
```
val caption:TextView
val dateTime:TextView
val image:ImageView          // new one added
val playButton: ImageView    // new one added
val editButton: ImageButton  // new one added
val view:View                // new one addeed
```

and update the `init{ }` block accordingly:

Kotlin
```
init {
  view = v
  caption = v.findViewById(R.id.media_caption)
  dateTime = v.findViewById(R.id.media_date_time)
  image = v.findViewById(R.id.media_image)
  playButton = v.findViewById(R.id.play_media)
  editButton = v.findViewById(R.id.edit_button)
}
```

To render the photo (or first frame of video) add the following snippet to the `bindTo` function of `JournalMediaViewHolder`:

Kotlin
```
when (m.type) {
  MediaType.PHOTO.ordinal -> Glide.with(view).load(m.url).into(image)
  MediaType.VIDEO.ordinal -> Glide.with(view).load(m.thumbnailUrl).into(image)
}
```

The user may interact with each journal media entry in two different ways: tapping on the media display area to render/view the media in full-size or on the edit button to edit the journal media details. Since the next few fragments described below will depend on the details of the selected media entry, it is a good idea to keep this object in the `MediaViewModel` by adding the following mutable live data:

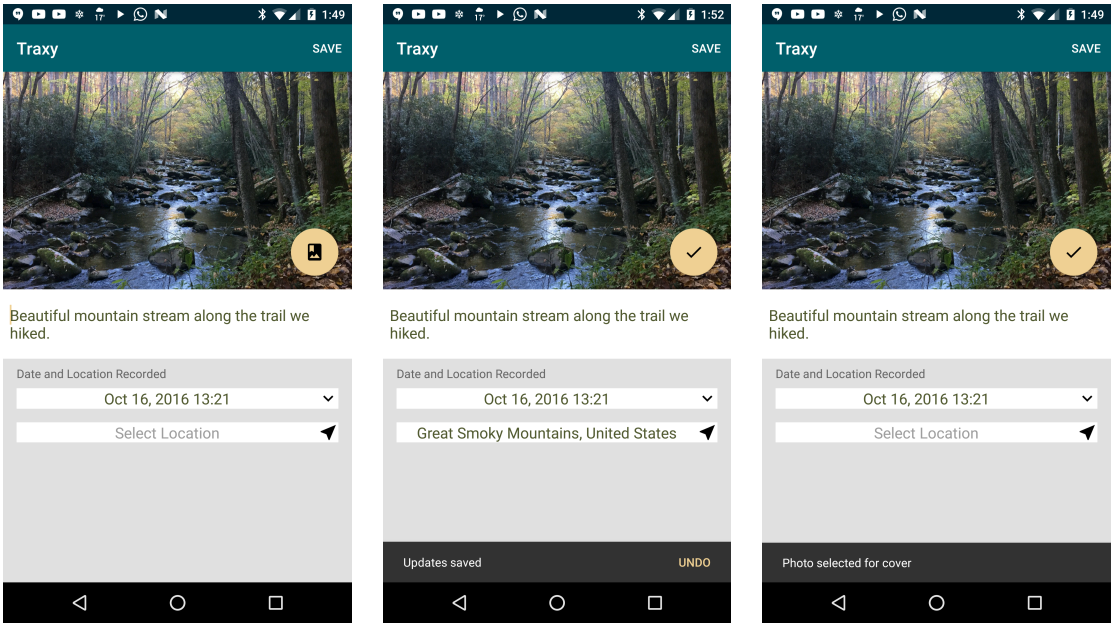
Kotlin
```
// In MediaViewModel.kt
val selectedMedia = MutableLiveData<JournalMedia>()
```

MediaEditFragment

We will now create a new fragment for editing journal media entry. The completed screen design of this fragment should look like Figure 9.16.

The FAB at the lower right corner of the photo allows the user to select the current photo as the cover photo of the journal. It is rendered using a *state drawable*: it renders one icon when the button is in normal state, and renders another when the button is selected. In the XML layout, the FAB is declared like any other button:

XML
```
<android.support.design.widget.FloatingActionButton
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:src="@drawable/cover_photo_toggle"
  other-attributes-not-shown   />
```



(a) (b) (c)

Figure 9.16: Journal Media Entry Editor

Under the **res/drawable** directory the resource `cover_photo_toggle.xml` is declared using a `<selector>` tag:

XML

```xml
<?xml version="1.0" encoding="utf-8"?>
  <selector xmlns:android="http://schemas.android.com/apk/res/android">
  <!-- the order of these items matters. Place the default icon LAST -->
  <item android:drawable="@drawable/ic_done_black_24dp"
    android:state_selected="true" />
  <item android:drawable="@drawable/ic_photo_album_black_24dp" />
</selector>
```

The attribute `android:state_selected="true"` informs the button rendering routine that `ic_done_black_24dp` is the icon to render when the button is selected. Otherwise, in its normal state the button shall be rendered using `ic_photo_album_black_24dp`. Using state drawables makes your code much simpler to write, when the FAB is tapped, you just need to toggle its selected state to true or false.

Kotlin

```kotlin
// In response to button click
fab_cover_photo.isSelected = !fab_cover_photo.isSelected
```

Go ahead and add the fragment and incorporate all the details mentioned above. Include the "Save" action as indicated in the screenshot, using the same approach for creating the "Logout" action in `MainFragment`. Don't worry about writing the actual code for saving the entry to Firebase. You will find more details in the next section.

Clicking on the date input field should bring up a date picker and a time picker dialogs. We can use the Material Design picker to this purpose and include the following third-party library:

Gradle
```
implementation 'com.google.android.material:material:1.3.0-alpha03'
```

This library requires that your app theme is derived from Theme.MaterialComponents and you are required to update `values/styles.xml` and change the AppTheme from

XML
```
<!-- Base application theme. -->
<style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">
```

to

XML
```
  <!-- Base application theme. -->
<style name="AppTheme" parent="Theme.MaterialComponents.Light.DarkActionBar">
```

To use the data and time picker make the following changes in `MediaEditFragment`:

1. Define two picker dialogs

Kotlin
```
val datePicker = MaterialDatePicker.Builder.datePicker().build()
val timePicker = MaterialTimePicker.Builder().build()
```

We will show the two dialogs one after the other (date first)

2. Setup three "click" listeners in `onViewCreated`

(a) First, the click listener for the date input field to show the date picker dialog

Kotlin
```
media_date_time.setOnClickListener {
  datePicker.show(parentFragmentManager, "MediaDate")
}
```

(b) In response to the user confirming the date selection, your app should then save the selected date and show the time dialog:

Kotlin
```
datePicker.addOnPositiveButtonClickListener {
  mediaModel.mediaDate.value = DateTime(it, DateTimeZone.UTC)
  timePicker.show(parentFragmentManager, "MediaTime")
}
```

By default, the selected date (in UTC) is constructed with time at the beginning of the day (00:00)

(c) Finally, when the time selection is confirmed, it's hour and minute should be added to the selected date.

Kotlin
```
timePicker.addOnPositiveButtonClickListener {
  with(mediaModel.mediaDate) {
    value = value?.run {
      plusHours(timePicker.hour).plusMinutes(timePicker.minute)
    }
  }
}
```

Handling the click events on the location input field is exactly similar to what you learned in a previous chapter. So the code snippet will not be repeated here. After completing the UI design, you can add the fragment as another destination from `JournalViewFragment`. Let's edit the navigation graph as follows:

1. Add `MediaEditFragment` as a new destination
2. Add a new action arrow from `JournalViewFragment` to this new fragment call the action `action_to_mediaEdit`

The transition from `JournalViewFragment` to `MediaEditFragment` will be triggered by `JournalMediaAdapter`. Using a similar strategy used by `JournalAdapter` we will add a listener function as another parameter of `JournalMediaAdapter` constructor by

Kotlin

```kotlin
class JournalMediaAdapter(
    opt: FirestoreRecyclerOptions<JournalMedia>,
    /* the second arg is either "EDIT" or "VIEW" */
    val listener: (JournalMedia, String) -> Unit,
) :
    FirestoreRecyclerAdapter<JournalMedia, JournalMediaViewHolder>(opt) {
    //
}
```

The second argument to the listener is a string that will be set to either "EDIT" or "VIEW" depending on how the user interacts with journal media item so your app knows which destination to navigate to. The listener would then be passed into the `bindTo` function in `onBindViewHolder`:

Kotlin

```kotlin
override fun onBindViewHolder(
  holder: JournalMediaViewHolder,
  position: Int,
  model: JournalMedia,
) {
    holder.bindTo(model, listener)
}
```

This update will propagate to the `bindTo` function of `JournalMediaViewHolder` where it will install a click listener for the appropriate widgets:

Kotlin

```kotlin
if (listener != null) {
  editButton.setOnClickListener {
    listener(m, "EDIT")
  }
  playButton.setOnClickListener {
    listener(m, "VIEW")
  }
  image.setOnClickListener {
    listener(m, "VIEW")
  }
}
```

Consequently, the adapter initialization in `JournalViewFragment` must be updated from:

Kotlin
```
adapter = JournalMediaAdapter(option)
```

to include a lambda function:

Kotlin
```
adapter = JournalMediaAdapter(option) { media:JournalMedia, action:String ->
  mediaModel.selectedMedia.value = media
  if (action == "EDIT") {
    val navAction = JournalViewFragmentDirections.actionToMediaEdit()
    findNavController().navigate(navAction)
  } else if (action == "VIEW") {
    // To be added later
  }
}
```

> **NOTE**
>
> The class `JournalViewFragmentDirections` is automatically generated by the Navigation Safe Args plugin. Unfortunately, the plugin generates the class only after a manual (re)build.

The destination fragment can pull the details of the journal media from the media view model class. For instance, in `onResume` of `MediaEditFragment` you can render the photo using Glide as follows:

Kotlin
```
override fun onResume() {
  super.onResume()
  mediaModel.selectedMedia.observe(viewLifecycleOwner) {
    when(it.type) {
        MediaType.PHOTO.ordinal -> {
          Glide.with(this@MediaEditFragment)
              .load(it.url)
              .into(media_image)
          media_image.visibility = View.VISIBLE
        }
        MediaType.VIDEO.ordinal -> {
          Glide.with(this@MediaEditFragment)
              .load(it.thumbnailUrl)
              .into(media_image)
          media_image.visibility = View.VISIBLE
        }
        else -> media_image.visibility = View.GONE
    }
    // Add more code as needed
  }
}
```

Saving Updated Entries

Handling the "Save" menu will trigger an update to Cloud Firestore. The Firestore API for updating a document requires that we supply the document key and provide the new key value pairs

in a map. We will continue to delegate all the backend update tasks to the repository class. Let's first add a new function `updateMediaEntry`

Kotlin
```
fun updateMediaEntry(journalKey: String, m: JournalMedia) {
  docRef?.let {
    var mediaData = hashMapOf(
        "type" to m.type,
        "caption" to m.caption,
        "date" to m.date,
        "url" to m.url,
        "lat" to m.lat,
        "lng" to m.lng
    )
    if (m.type == MediaType.VIDEO.ordinal)
      mediaData["thumbnailUrl"] = m.thumbnailUrl
    it
      .collection("journals/$journalKey/media")
      .document(m._key)
      .set(mediaData)
}
```

The above function will be called from the the media view model class from the following function:

Kotlin
```
fun updateJournalMedia(media: JournalMedia) {
  selectedJournal.value?.let {
    TraxyRepository.updateMediaEntry(it.key, media)
  }
}
```

And finally, in response to clicking the "Save" menu, `MediaEditFragment` calls the above function:

Kotlin
```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
  if (item.itemId == R.id.action_save_media) {
    mediaModel.mediaCaption.value = media_caption.text.toString()
    mediaModel.selectedMedia.value?.apply {
        caption = media_caption.text.toString()
        date = mediaModel.mediaDate.value.toString()
        lat = mediaModel.selectedJournal.value?.lat ?: 0.0
        lng = mediaModel.selectedJournal.value?.Lng ?: 0.0
    }?.let {
        mediaModel.updateJournalMedia(it)
    }
    return true
  }
  return false
}
```

Actionable SnackBars We understand that updating the database is a *destructive* operation. The general approach for avoiding unintentional updates is to ask the user to confirm a yes/no dialog. However, the Android Design guidelines takes a slightly different approach. The guidelines include a section on Confirming and Acknowledging which shows a flowchart how to handle user

actions that require confirmation or acknowledgment. Instead of prompting for a confirmation dialog when the user selects the "Save" menu, your app should perform the update while offering an "Undo" option when notifying the completion of the update using a Snackbar as shown in Figure 9.16b.

To support this "undo" feature the media view model class should keep a backup copy of the selected media provide functions to save and restore the between these two copies:

Kotlin

```
// In MediaViewModel.kt
var backupMedia: JournalMedia? = null

fun saveMediaCopy() {
  selectedMedia.value?.let {
    // Clone from the live data
    backupMedia = it.copy()
  }
}

fun restoreMediaCopy() {
  backupMedia?.let {
    // Restore into the live data
    selectedMedia.value = it.copy()
  }
}
```

To make the SnackBar message shows up after the update is complete, we can modify the `updateMediaEntry` function in the repository class into a coroutine so we don't have to provide a completion listener callback. Surprisingly the modification required is very minimal:

1. Change the first two lines of the function from

Kotlin

```
fun updateMediaEntry(journalKey: String, m: JournalMedia) {
  docRef?.let {
    // your code here
  }
```

to

Kotlin

```
suspend fun updateMediaEntry(journalKey: String, m: JournalMedia): Void? {
  return docRef?.let {
    // your code here
  }

}
```

2. Add the asynchronous `await()` call at the end of function calls for updating the journal media in Cloud Firestore:

Kotlin

```
it
  .collection("journals/$journalKey/media")
  .document(m._key)
  .set(mediaData)
  .await()        // Append this call to make it suspendable
```

The above updates require its caller (in the media view class) to be declared as a co-routine as well (you just need to add the keyword `suspend`)

Kotlin
```
suspend fun updateJournalMedia(media: JournalMedia) {
  selectedJournal.value?.let {
      TraxyRepository.updateMediaEntry(it.key, media)
  }
}
```

These changes cascade up to the view class `MediaEditFragment`—. The one line function call in the click listener of the "Save" button

Kotlin
```
mediaModel.updateJournalMedia(it)
```

must change into a longer block:

Kotlin
```
CoroutineScope(Dispatchers.IO).launch {
  mediaModel.updateJournalMedia(it)
  withContext(Dispatchers.Main) {
    Snackbar.make(media_caption, "Updates saved", Snackbar.LENGTH_LONG)
        .setAction("Undo") { undoSavedMedia() }
        .show();
  }
}
```

and `undoSavedMedia` is just a private function:

Kotlin
```
private fun undoSavedMedia() = with(mediaModel) {
  restoreMediaCopy()
  selectedMedia.value?.let {
    CoroutineScope(Dispatchers.IO).launch {
      updateJournalMedia(it)
    }
  }
}
```

Go ahead and implement the update logic as described above. We give you the puzzle pieces in various snippets, it is your turn to put those snippets together. As always, remember to use Alt + Enter whenever there is a compiler error due to missing declaration, missing import, etc. Android Studio is smart enough to give you suggestions based on the current context of the errors.

MediaPreviewFragment

The second action in response to screen taps is viewing the media in full-screen mode. Let's create a new fragment `MediaPreviewFragment`. The UI design of this fragment is very simple, because its main responsibility is to show either an image using an `ImageView` or a video using a `VideoView`. Practically, you can replicate the XML layout for Figure **??** and keep only the top part. Except you want to set the dimension of the containing `FrameLayout` to `match_parent`.

Also, viewing photos and videos are better done in landscape mode, so you should specify its screen orientation programmatically:

Kotlin
```
private var incomingOrientation = 0

override fun onResume() {
  super.onResume()
  incomingOrientation = requireActivity().requestedOrientation
  requireActivity().requestedOrientation = ActivityInfo.SCREEN_ORIENTATION_LANDSCAPE
}

override fun onPause() {
  super.onPause()
  requireActivity().requestedOrientation = incomingOrientation
}
```

You can reuse the Glide library to render the image and the fragment requires the following changes:

1. Declare the view model

Kotlin
```
private val mediaModel by activityViewModels<MediaViewModel>()
```

2. Use Glide to render the image in `onResume()`. The following snippet assumes that the ImageView in your layout file is named `photoView`.

Kotlin
```
  mediaModel.selectedMedia.observe(viewLifecycleOwner) {
    when (it.type) {
        MediaType.PHOTO.ordinal ->
            Glide.with(this)
                .load(it.url)
                .into(photoView)
        MediaType.VIDEO.ordinal ->
            /* code for rendering video file */
    }
}
```

Instead of using Android default an `ImageView` we will be using a third party library that allows pan and zoom control. Add the following dependency to your app `build.gradle`

Gradle
```
compile 'com.github.chrisbanes:PhotoView:2.0.0'
```

and use `PhotoView` in place of the Android native `ImageView` in your layout file:

XML
```
<com.github.chrisbanes.photoview.PhotoView
  android:id="@+id/photoView"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  app:srcCompat="@drawable/traxy_logo" />
```

That's it. You can try your app to view and zoom your photos.

9.3.7 Media Streaming with ExoPlayer 2.x

When rendering an audio or video, usually we want to employ media streaming. You can use ExoPlayer to handle this task. Follow its developer guide for details of adding the library into your project. The Gradle dependency to use this library is

Gradle
```
implementation 'com.google.android.exoplayer:exoplayer:2.5.4'
```

At the time of this writing, version 2.5.4 is the latest.

ExoPlayer offers many performance improvements over the traditional `MediaPlayer`. In addition, various components of `ExoPlayer` are extensible and customizable. A typical streaming media player comprises the following building blocks:

- Media Loader: a component responsible for fetching media data from its source (either a local file or remote data in the cloud). This component is also responsible for buffer management.
- Media Decoder: data bytes fetched from the source are then decompressed for further processing
- Media Playback: renders the uncompressed data stream as a video and/or audio signal.

ExoPlayer depends on its own widget, `SimpleExoPlayerView` to render video. You must add this widget to your layout. Insert it as a sibling to `PhotoView` added earlier. So both `PhotoView` and `SimpleExoPlayerView` are immediate children of the root `FrameLayout`. At runtime you will have to change their visibility so only one of them is visible, a trick you did earlier in this chapter. During video playback, this widget will render the video, but during audio playback this widget will stay blank. The rest of this chapter assumes that the ExoPlayer widget is called `videoView`.

The initialization step to use ExoPlayer is shown requires only the following two lines:

Kotlin
```
// Inside onViewCreated
exoPlayer = SimpleExoPlayer.Builder(requireContext()).build()
videoView.player = exoPlayer
```

To load and playback a video file/source, you just need to add four additional lines in the observer block of `selectedMedia`:

Kotlin
```
// In onResume()
mediaModel.selectedMedia.observe(viewLifecycleOwner) {
  when (it.type) {
      MediaType.PHOTO.ordinal ->
          // Use Glide to render photo
      MediaType.VIDEO.ordinal -> {
          val mediaContent = MediaItem.fromUri(it.url)
          exoPlayer.setMediaItem(mediaContent)
          exoPlayer.prepare()
          exoPlayer.playWhenReady = true
      }
  }
}
```

Add the ExoPlayer code into your activity. Rerun your app and select a journal entry with video type. The video playback control will automatically appear/disappear when you touch the player area. During video playback, this widget will render the video, but during audio playback this widget will stay blank.

Before you exit out from this fragment be sure to stop the media player and release its resources by adding the following code in `onStop()`

Android

```
// in onStop
videoView.player = null
player.release();
```

The class also provides `ExoPlayer.EventListener` for monitoring any changes in the player state. For instance, the method `onLoadingChanged` can be used to hide the progress indicator when the loading completes.

Kotlin

```
object exoHandler : EventListener {
    override fun onLoadingChanged(isLoading: Boolean) {
      if (!isLoading) {
        // Assuming you have a progress indicator in the UI
        loadProgress.setVisibility(View.GONE);
      }
    }
}
```

The above event listener must be attached to the media player object

Kotlin

```
// Inside onViewCreated
exoPlayer.addListener(exoHandler)
```

References:

- Forget the Storage Permission: Alternatives for sharing and collaborating (Big Android BBQ 2015)
- Runtime Permissions in Android 6.0 Marshmallow (Android Development Patterns Episode 3)
- Android Marshmallow 6.0: Asking For Permission
- Media Playback with Media SessionCompat
- Media playback the right way (BABBQ 2015)
- Best practices in media playback (Google IO 2016)
- Streaming media with ExoPlayer (Google IO 2016)
- ExoPlayer: Flexible Media Playback for Android (Google IO 2017)

—Imagine if Beethoven had a tape recorder. Then you'd know exactly what he meant. Maybe he meant 'Da da da da' instead of 'Boom boom boom boom!' Who knows?

Eddie Van Halen

10 Working With Audio

Chapter Objectives

After reading this chapter you should be able to

- Implement audio playback and capture features in iOS apps.
- Implement audio playback and capture features in Android apps.

Mobile phones had audio input and output capabilities long before iPhones and Android phones appeared on the scene. In fact, full duplex voice communication was the original killer app for mobile phones. It wasn't until the mid 1990s where short message services (now known as "texting") started to appear, and it was the late 1990s early 2000s before widespread adoption was realized. Interestingly today, many mobile phone users (especially younger ones) still use their phone as a communication device, but rarely in the form of voice communication. Instead communication takes place via texting, Snapchat, Facebook, and other social media applications.

Given this usage trend, perhaps you are wondering why you should bother studying how to record and playback audio in your applications. There are actually a number of reasons why:

1. Make it easy to capture information without having to type. Many record-keeping apps will allow users to create voice notes instead of text. The audio can be stored for future playback, or in some cases might be automatically transcribed into text via cloud-based speech recognition.

2. Convey information to users when they are situationally impaired. For example, navigation apps assume the user is driving and produce audio driving instructions in addition to what is displayed on the screen.

3. Integrate audio from the device's media library. For example, if you were developing a fitness app, you might want to allow the user to play MP3 files in their music libraries while working out.
4. The recent advances in speech recognition / synthesis and widespread adoption in general of conversational technologies in digital experiences is making people realize that audio input/output can be a very useful interface modality.
5. Audio as sensory input. Given advances in machine learning, the mic on a smart phone can be used to capture information that can promote commerce (Shazam), manage health (Novid), diagnose mechanical issues (Skoda's Sound Analyzer app) and assist in non-trivial configuration tasks (GuitarTuna).

As soon as you decide to use audio in your application, you are suddenly faced with a number of other interesting questions. For example, what should happen if an application is playing audio and the user sends it to the background? What should happen if a voice call arrives on the device while an app is recording or playing audio? The phone only has one microphone and one speaker capability, so all apps on a given device must share these resources. If an app, implemented on both iOS and Android, posts audio to an application server, what audio format should the audio be stored in so it is interchangeable on both platforms? Both the Android and iOS platforms offer audio recording and playback capabilities that provide you ways to adequately address these questions and more in the form of policies.

Let's take a close look at how you can incorporate audio in your mobile apps.

10.1 Working with Audio in iOS

In this section we will explore audio capture and playback on the iOS platform. In addition, we will discuss audio formats in sufficient detail in order to ensure cross platform playback in our implementations. We will do so by adding an audio journal entry feature to the Traxy app.

10.1.1 Audio Capture and Playback

There are a multitude of APIs and frameworks provided by iOS that let you capture and play audio. For capturing audio, we are going to introduce a new controller named `AudioViewController` to our Traxy application. We will be using the AVFoundation classes for recording and playing back the audio we capture using this controller. iOS also provides a companion framework to AVFoundation called AVKit which provides common audio/visual media-related user interface components, such as the `AVPlayerViewController`. For our `AudioViewController` however, we will roll our own user interface and simply use the classes provided us by the AVFoundation framework. There are essentially three classes in particular that we will focus on: `AVAudioSession`, `AVAudioRecorder`, and `AVAudioPlayer`.

Working with audio is a bit more involved than might be initially anticipated. Keep in mind that there are many applications sharing the underlying resources of the iOS device your application is installed on. Naturally, some of these applications also utilize the same audio capabilities the Traxy app does. Things get more complicated when we realize that unlike the user interface facets of our application that typically only get rendered when our application has focus, it is possible for applications to continue playing audio from the background when they are not on screen.

Consider for example the Music app that Apple pre-installs on the iPhone. It can play music from your iTunes library while your email app is in the foreground. You could also be interacting on a voice phone call, while looking up a phone number for a contact in your Contacts application. Furthermore, it is not always easy to predict when other apps might require the device's audio capabilities. For example, while our app is playing audio, the Phone app might receive a phone call and need to play a ring alert to the user.

Given these various considerations, our application somehow needs to provide some information to iOS about our intentions or policies with regard to how we will utilize the audio capabilities of the device, and how it will interact with other apps that are doing the same. For example, if our application decides to play audio, and there is already audio being played by the Pandora app, we may want to have the Pandora app fade and our audio begin playing. We define our application's audio behavior via the `AVAudioSession` class. This class let's us define what kind of audio formats our app will utilize, how our app will integrate audio from other apps, or what will happen when our app is interrupted by the audio needs of other apps. In short, `AVAudioSession` is used to set the overall audio context for our application.

Having established our audio session we can than proceed to capture and play audio. We accomplish the former via the `AVAudioRecorder` class. This class is capable of capturing audio continuously until we tell it to stop. It can also be configured to capture audio for a specific duration, as well as the ability to pause and resume an ongoing recording. While recording audio, we can simultaneously obtain information back from the audio recorder that can be used in our user interface to provide audio-level metering or progress. As you might expect by now, the `AVAudioRecorder` communicates back to our code via the delegation pattern. In particular, we will need to implement the `AVAudioRecorderDelegate`.

We will allow the user to confirm captured audio by being able to play it back via the `AVAudioPlayer` class. This class enables the playback of local audio from files or memory buffers. It can also provide looping (e.g. audio assets get played repetitively — think about the background music in Mario Run!), play and synchronize the playback of multiple sounds simultaneously, and provide fast forward, and rewind capabilities. It is also possible to obtain data during playback that can be used for audio playback level metering. `AVAudioPlayer` interacts with our code via the `AVAudioPlayerDelegate`.

Establishing the Audio Record User Interface

Our approach to implementing `AudioViewController` is going to take advantage of iOS's container views. In the storyboard, a "Container View" can be dragged from the object library into a view controller's scene. This container view (actually just an ordinary `UIView`) can contain or embed an arbitrary view controller of our choice. Figure 10.1a shows the actual rendering of the view controller in the app. Notice that the confirmation fields below the audio controls in the user interface looks identical to what we used in the `JournalEntryConfirmationViewController`. That's because it is! Using a container view, we've been able to embed this controller into our `AudioViewController`, and hence reuse it entirely. Figure 10.1b shows an `embed segue` emanating from our `AudioViewController` scene. We need to set the class identifier in the identity inspector for the `JournalEntryConfirmationViewController` scene represented in Figure 10.1c. This approach makes it possible for us to both record and confirm our audio entries on a single screen, and reuse a significant chunk of code we've already written. However, unlike the original instantiation of the `JournalEntryConfirmationViewController`, the naviga-

tion bar is associated with our `AudioViewController` scene here, not the embedded controller, so we're going to go ahead and add save/cancel buttons to the AudioViewController scene's navigation bar as shown. Later we will have to implement IBActions for these.

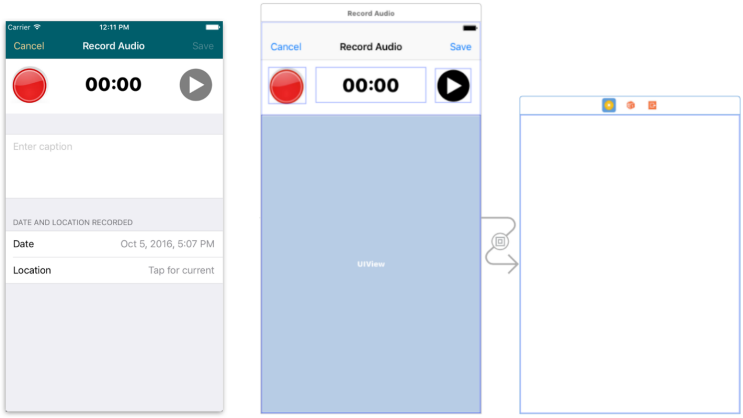


Figure 10.1: (a) The actual layout of the AudioViewController scene in traxy. (b) The layout in the storyboard. (c) The confirmation form (implemented previously by JournalEntryConfirmationViewController) embedded in AudioViewController.

The red button on the left serves as the record button. Once recording begins, the button will morph into the stop recording button. Similarly, the greyed out inactive button on the left is the play button. It will remain in an inactive state until we have audio recorded. Once playback begins, it morphs into the playback stop button. The text label between the buttons will be used during both recording and playback to indicate how much audio has been captured, or the current position we are in during playback.

The user interface controls are wired up to outlets in our controller class, and defined a few images we will use to morph our buttons in mid stream. The various properties defined by the AudioViewController class are shown in Listing 110.

We configure the embedded `JournalEntryConfirmationViewController` by overriding the `prepare:forSegue` method. When the `AudioViewController` loads, it will automatically call this method for any *embedded segue*. We show the code for this overridden method in Listing 111. You'll notice immediately that it resembles the code we wrote in `JournalTableViewController` to handle the push segues we performed to the confirmation screens for the other entry types. Be sure to select the embedded segue marker in your storyboard (e.g. the arrow emanating from the `AudioViewController` to the embedded controller) and set its identifier to the string "setupAudioCaptionForm" so it works properly with the prepare for segue method.

We capture a reference to the embedded `JournalEntryConfirmationViewController` when preparing for the embedded segue, and store it in a property defined by `AudioViewController`. This will be needed later when the save button is pressed, to extract the caption, date, and location for the audio journal entry.

```swift
@IBOutlet weak var saveButton: UIBarButtonItem!
@IBOutlet weak var timeLabel: UILabel!
@IBOutlet weak var recordButton: UIButton!
@IBOutlet weak var playButton: UIButton!
var recording = false
var playing = false
var entry : JournalEntry?
var journal : Journal!
var captionEntryCtrl : JournalEntryConfirmationViewController?
var recorder: AVAudioRecorder!
var player: AVAudioPlayer!
var audioTimer:Timer!
var audioFileUrl:URL!
weak var delegate : AddJournalEntryDelegate?

let playImage =  UIImage(named: "play")
let stopImage = UIImage(named: "stop")
let recordImage = UIImage(named: "record")
```

Listing 110: AudioViewController properties..

```swift
// MARK: - Navigation
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
  if segue.identifier == "setupAudioCaptionForm" {
    if let destCtrl = segue.destination as? JournalEntryConfirmationViewController {
      destCtrl.type = .audio
      destCtrl.entry = self.entry
      destCtrl.journal = self.journal
      self.captionEntryCtrl = destCtrl
    }
  }
}
```

Listing 111: Overridden method in AudioViewController to setup the embedded JournalEntryConfirmationViewController.

Setting up the AVAudioSession

We've implemented a couple of helper functions to establish our audio session properties which are shown in Listing 112. The `setSessionPlayback` method is defines our session for playback and `setSessionPlayAndRecord` method for recording. Setting the category to `AVAudioSessionCategoryPlayback` indicates that our audio playback will suppress any other audio that happens to be playing when our app plays audio. It also means that our audio will play regardless of whether the device's Silent switch is on or off. It will not continue playing if we send our application to the background, unless we take further steps (e.g. take a look at the `UIBackgroundModes` key in the Apple documentation). For recording we set the category to `AVAudioSessionCategoryPlayAndRecord`, which has similar properties but also expressed our intent to record.

Swift

```
func setSessionPlayback() {
  let session:AVAudioSession = AVAudioSession.sharedInstance()
  do {
    try session.setCategory(
      AVAudioSession.Category(rawValue:
        AVAudioSession.Category.playback.rawValue))
  } catch let error as NSError {
    print(error.localizedDescription)
  }
  do {
    try session.setActive(true)
  } catch let error as NSError {
    print(error.localizedDescription)
  }
}

func setSessionPlayAndRecord() {
  let session = AVAudioSession.sharedInstance()
  do {
    try session.setCategory(
      AVAudioSession.Category(rawValue:
        AVAudioSession.Category.playAndRecord.rawValue))
  } catch let error as NSError {
    print(error.localizedDescription)
  }
  do {
    try session.setActive(true)
  } catch let error as NSError {
    print(error.localizedDescription)
  }
}
```

Listing 112: Establishing the audio session context.

Once we have these helper functions defined, we can establish the initial state of the controller by implementing its `viewDidLoad` method, as shown in Listing 113. By default the `playButton` is disabled, since there is no audio to play initially. Similarly, the `saveButton` is also disabled, since there is nothing to save. Once audio is recorded, both of these buttons will be enabled. In

addition, if the entry property is nil a new `JournalEntry` object with default values is created to hold the data entered for the audio entry.

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

  self.playButton.isEnabled = false
  self.setSessionPlayback()
  if self.entry == nil  {
    self.entry = JournalEntry(key: nil, type: .audio, caption: "", url: "",
      thumbnailUrl: "", date: Date(), lat: 0.0, lng: 0.0)
  }

  self.saveButton.isEnabled = false
}
```

Listing 113: Implementing AudioViewController's viewDidLoad method.

Recording Audio

Like capturing photos and videos, using the device's microphone to capture audio data is something that our application has to be granted permission for. Part of this involves defining a string prompt to the key `NSMicrophoneUsageDescription` in our workspace's `Info.plist` file.

The record button is wired to the IBAction shown in Listing 114. Note how the image associated with the button varies based on whether or not recording is taking place when it is pressed.

```swift
@IBAction func recordButtonPressed(_ sender: UIButton) {
  if self.recording {
    self.recordButton.setImage(self.recordImage, for: .normal)
    self.playButton.isEnabled = true
    self.recording = false
    self.recorder.stop()
  } else {
    // Note we adjust the UI in recordIfPermitted, as we don't know
    // at this point if permissions have been granted.
    if recorder == nil {
      self.recordIfPermitted(setup: true)
    } else {
      self.recordIfPermitted(setup: false)
    }
  }
}
```

Listing 114: IBAction that handles the record button press.

The helper function `recordIfPermitted` and related helper functions shown in Listing 115

requests permission to record, and if permitted initiates recording. A Timer object is started to update our time elapsed label every tenth of a second. The play button is disabled anytime recording starts to eliminate a situation where the user might attempt to play while recording. If the user has not granted the app permissions to record audio, the user is informed via an alert and given the option to launch the settings app on the device to effect that change. The code for this can be seen in Listing 116.

The first time we call this permission, we won't yet have a record instance allocated. This is accomplished by the setupRecorder helper method shown in Listing 117.

Setting up the record involves specifying the file to hold the audio data, configuring the audio format, sampling rates, etc, and establishing other properties on the audio recorder, including its delegate. Once that is all established, and the recorder instance is created, we call its `prepareToRecord` method. This method essentially creates the specified audio file, or overwrites an existing file. If we were to call the recorder's `record` method before calling prepareToRecord it will be called automatically. Calling `prepareToRecord` prior to `record` simply makes record startup faster.

`AVAudioRecorder` communicates back to our controller via delegation, so we must implement the AVAudioRecorderDelegate protocol. We've implemented this in the form of the extension shown in Listing 118. As can be readily seen, our main task here is to simply update our boolean state variables and button images when recording completes.

Playing Back Audio

Once the user has recorded some audio and the recording action is stopped, the play button is activated and the captured audio can be previewed prior to captioning and saving to the cloud. The code for playing back the captured audio is much less involved than the code we looked at previously for recording. Listing 119 shows the action `playButtonPressed` that is executed when the play button is tapped. Note once again that the behavior depends on whether or not audio is already playing when the button is tapped. The `play` helper function instantiates a player, sets the controller as the delegate and begins to play the captured audio.

Like the `AVAudioRecorder`, `AVAudioPlayer` communicates back to our controller via delegation. Listing 120 shows the implementation of the `AVAudioPlayerDelegate` protocol by `AudioViewController`. As was the case for recording, the primary task is to update our state and button images when playback stops. This could occur when we reach the end of the recording, or when the user taps the "play" button and we call stop on the player.

10.1.2 Saving Audio Journal Entries to Firebase

Let's look at the code for persisting audio to Firebase. As previously mentioned, `AudioViewController` places save / cancel buttons on its navigation bar. The IBActions wired to these buttons in the storyboard are shown in Listing 121. The form values are extracted from the embedded controller, and the audio URL from the recorder. This information is tucked into the journal entry instance and delegated back to the JournalTableViewController where the data is actually persisted via the `TrayRepository` singleton. In a previous chapter, we simply stubbed out the `saveAudioEntry` method. Let's now take a look at how to implement that.

Uploading audio files to Firebase is similar to saving video. In fact you can reuse the `saveMediaFileToFirebase` helper method that you introduced in the previous chapter. How-

```swift
func recordIfPermitted(setup:Bool)  {
  let session:AVAudioSession = AVAudioSession.sharedInstance()
  if (session.responds(to:
  #selector(AVAudioSession.requestRecordPermission(_:))))
  {
    AVAudioSession.sharedInstance().requestRecordPermission({(granted: Bool)->
      Void in
      DispatchQueue.main.async {
        if granted {
          self.setSessionPlayAndRecord()
          if setup {
            self.setupRecorder()
          }
          self.recorder.record()
          self.audioTimer = Timer.scheduledTimer(timeInterval: 0.1, target:self,
            selector:#selector(AudioViewController.updateTimeLabel(timer:)),
            userInfo:nil,repeats:true)

          // we're rolling!  update the UI
          self.recordButton.setImage(self.stopImage, for: .normal)
          self.playButton.isEnabled = false
          self.recording = true
        } else {
          self.displaySettingsAppAlert()
        }
      }
    })
  } else {
    // technically, this happens when the device doesn't have a mic.
    self.displaySettingsAppAlert()
  }
}

@objc func updateTimeLabel(timer:Timer) {
  if self.recorder.isRecording {
    let min = Int(recorder.currentTime / 60)
    let sec = Int(recorder.currentTime.truncatingRemainder(dividingBy: 60))
    let s = String(format: "%02d:%02d", min, sec)
    self.timeLabel.text = s
    recorder.updateMeters()
  }
}
```

Listing 115: Recording related helper functions.

```swift
func displaySettingsAppAlert()
{
  let avc = UIAlertController(title: "Mic Permission Required", message: "You need to
    provide this app permissions to use your microphone for this feature. You can do
    this by going to your Settings app and going to Privacy -> Microphone",
    preferredStyle: .alert)

  let settingsAction = UIAlertAction(title: "Settings", style: .default ) {action in
    UIApplication.shared.open(NSURL(string: UIApplication.openSettingsURLString)! as
    URL,
    options: [:], completionHandler: nil)
  }

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

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

Listing 116: Launching the settings app via an UIAlertController if the user hasn't yet granted audio recording permissions.

ever, unlike video files, we do not have a thumbnail image to extract for audio files. Go ahead and implement the `saveAudioEntry` helper method on the `TraxyRepository` class as shown in Listing 122. Audio journal entries only involve an audio recording and the associated metadata. There are no images involved, hence we can save audio by simply writing out the journal entry node (lines 3-4). If the key is nil (line 6) that means its a brand new audio recording, that needs to be saved calling the `saveMediaFileToFirebase` helper method to upload the audio file (line 7) and updating the entry's audio download URL once the upload completes (lines 9-12).

10.1.3 Adding a Segue to AudioViewController

Our last task is to add a segue from the existing `JournalTableViewController` to the new `AudioViewController`. In the storyboard, add a manual segue from the former to the latter and give the new segue the id "recordAudio". Next, in the `JournalTableViewController` the prepare:for segue method needs to be updated to perform the setup for this segue, as shown in Listing 123.

Finally, recall from the previous chapter, that new media is added to the journal when the user taps the add entry button on the `JournalTableViewController`'s navigation bar, which in turn displays a `UIAlertViewController`. In the previous chapter we stubbed out the code that created a `UIAlertAction` associated with the "Audio Entry" option displayed on the alert. A line of code needs to be added to this UIAlertAction that performs the segue that was just added. The updated `UIAlertAction` code is shown in Listing 124).

That's it! You should now be able to build and run Traxy on an iPhone, and confirm that you can store audio journal entries.

```swift
func setupRecorder() {
  let format = DateFormatter()
  format.dateFormat="yyyy-MM-dd-HH-mm-ss"
  let currentFileName = "audio-\\(format.string(from: Date())).m4a"
  let documentsDirectory = FileManager.default.urls(for: .documentDirectory,
    in: .userDomainMask)[0]
  self.audioFileUrl =
    documentsDirectory.appendingPathComponent(currentFileName)
  if FileManager.default.fileExists(atPath: audioFileUrl.absoluteString) {
    print("File \\(audioFileUrl.absoluteString) already exists!")
  }
  let recordSettings:[String : AnyObject] = [
    AVFormatIDKey:             NSNumber(value: kAudioFormatMPEG4AAC),
    AVEncoderAudioQualityKey : NSNumber(value:AVAudioQuality.max.rawValue),
    AVEncoderBitRateKey :      NSNumber(value:320000),
    AVNumberOfChannelsKey:     NSNumber(value:2),
    AVSampleRateKey :          NSNumber(value:44100.0)
  ]
  do {
    recorder = try AVAudioRecorder(url: audioFileUrl, settings: recordSettings)
    recorder.delegate = self
    recorder.isMeteringEnabled = true
    recorder.prepareToRecord()
  } catch let error as NSError {
    recorder = nil
    print(error.localizedDescription)
  }
}
```

Listing 117: Setting up the AVAudioRecorder instance.

```swift
extension AudioViewController : AVAudioRecorderDelegate {

  func audioRecorderDidFinishRecording(_ recorder: AVAudioRecorder,
    successfully flag: Bool)
  {
    self.saveButton.isEnabled = true
    self.recordButton.isEnabled = true
    self.playButton.isEnabled = true
    self.playButton.setImage(self.playImage, for: .normal)
    self.recordButton.setImage(self.recordImage, for: .normal)
  }

  func audioRecorderEncodeErrorDidOccur(_ recorder: AVAudioRecorder,
    error: Error?)
  {
    if let e = error {
      print("\(e.localizedDescription)")
    }
  }
}
```

Listing 118: Implementing the AVAudioRecorderDelegate protocol.

Swift

```swift
@IBAction func playButtonPressed(_ sender: UIButton) {
  if self.playing {
    self.playButton.setImage(self.playImage, for: .normal)
    self.recordButton.isEnabled = true
    self.playing = false
    self.player.stop()
  } else {
    self.playButton.setImage(self.stopImage, for: .normal)
    self.recordButton.isEnabled = false
    self.playing = true
    self.setSessionPlayback()
    self.play()
  }
}

func play() {
  var url:URL?
  if self.recorder != nil {
    url = self.recorder.url
  } else {
    url = self.audioFileUrl!
  }
  do {
    self.player = try AVAudioPlayer(contentsOf: url!)
    player.delegate = self
    player.prepareToPlay()
    player.volume = 1.0
    player.play()
  } catch let error as NSError {
    self.player = nil
    print(error.localizedDescription)
  }
}
```

Listing 119: Audio playback action and helper method.

```swift
extension AudioViewController : AVAudioPlayerDelegate {
  func audioPlayerDidFinishPlaying(_ player: AVAudioPlayer, successfully flag: Bool) {
    self.recordButton.isEnabled = true
    self.playButton.isEnabled = true
    self.playButton.setImage(self.playImage, for: .normal)
    self.recordButton.setImage(self.recordImage, for: .normal)
  }

  func audioPlayerDecodeErrorDidOccur(_ player: AVAudioPlayer, error: Error?) {
    if let e = error {
      print("\(e.localizedDescription)")
    }
  }
}
```

Listing 120: Implementing the AVAudioPlayerDelegate protocol.

```swift
@IBAction func saveButtonPressed(_ sender: UIBarButtonItem) {
  if let del = self.delegate {
    if let (caption,date,_) = (self.captionEntryCtrl?.extractFormValues()) {
      if var e = self.entry {
        e.url = self.recorder.url.absoluteString
        e.caption = caption
        e.date = date
        del.save(entry: e)
      }
    }
  }
  _ = self.navigationController?.popViewController(animated: true)
}

@IBAction func cancelButtonPressed(_ sender: UIBarButtonItem) {
  _ = self.navigationController?.popViewController(animated: true)
}
```

Listing 121: IBActions that implement the AudioViewController's save/cancel buttons..

```swift
func saveAudioEntry(journalKey: String, entry: JournalEntry) {
  setCurrentJournal(journalKey: journalKey)
  let vals = self.toDictionary(vals: entry)
  let entryRef = self.saveEntryToFireStore(key: entry.key, vals: vals)
  // if we have a nil key, then this is a new entry and we have an audio file to save.
  if entry.key == nil {
    self.saveMediaFileToFirebase(entry: entry, saveRefClosure: { (downloadUrl) in
      // having uploaded the audio data, now store its URL.
      let vals = [
      "url" : downloadUrl as NSString
      ]
      entryRef?.setData(vals, merge: true)
    })
  }
}
```

Listing 122: Helper function that saves a video or audio file to Firebase storage.

```swift
// MARK: - Navigation
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
  if segue.identifier == "confirmSegue" {
    if let destCtrl = segue.destination as? JournalEntryConfirmationViewController {
      destCtrl.imageToConfirm = self.capturedImage
      destCtrl.delegate = self
      destCtrl.type = self.captureType
      destCtrl.entry = self.entryToEdit  // will be nil on new item
      destCtrl.journal = self.journal
    }
  } else if segue.identifier == "recordAudio" {
    if let destCtrl = segue.destination as? AudioViewController {
      destCtrl.entry = self.entryToEdit // will be nil if new item.
      destCtrl.delegate = self
      destCtrl.journal = self.journal
    }
  }
}
```

Listing 123: Updating JournalTableViewController's prepare for segue method to handle the segue to AudioViewController.

```swift
let addAudioAction = UIAlertAction(title: "Audio Entry", style: .default) {
  (action) in
  self.performSegue(withIdentifier: "recordAudio", sender: self)
}
alertController.addAction(addAudioAction)
```

Listing 124: Updating JournalTableViewController's addEntryButtonPressed IBAction to initiate a segue to the AudioViewController.

10.2 Audio Processing in Android

If you feel tempted to search for implicit common intents for audio recording and playback, you will be disappointed to discover none. Implementation of these tasks generally require writing your own code. The Android Audio framework provides several audio-related classes and you can choose one of the two options for completing the task, use either lower-level or higher-level classes. Let's first look the the lower-level classes.

- `AudioTrack`: an abstraction of audio output ports on your device (built-in speaker or speaker on a wired or Bluetooth headset)
- `AudioRecord`: an abstraction for the audio input ports on your device (built-in microphone, or microphone on a wired or Bluetooth headset),

Using these lower-level classes requires a significant amount of manual settings. An easier approach to perform audio processing tasks is to use the higher-level classes: `MediaRecorder`, `MediaPlayer`, and `ExoPlayer`. The class names are self-explanatory and they don't need further explanations. Moreover, these classes are properly named using the "Media" prefix because they are capable of handling other media types, besides audio. `ExoPlayer` is a highly customized player used in YouTube Android app, which has been released as an open source by Google. You actually use `ExoPlayer` in the previous chapter to playback video files.

Regardless of the approach you choose for audio processing, your code will likely be using methods from the following two classes:

- `AudioManager`: a class that handles volume and other audio controls
- `AudioFocus`: a class for managing arbitration among apps that compete to use audio *output*, hence this topic is relevant only in the context of audio playback.

The discussions in this section are organized into two major sections: *recording* and *playback*.

10.2.1 Recording Audio

To record audio using the `AudioRecord` class, you essentially follow these steps:

1. Create and *configure* the `AudioRecord` object
2. Invoke the `AudioRecord.startRecording()` method
3. Use a loop to read audio data from the system audio buffer to your own buffer. The loop continues to run until the user stops the recording.
4. Invoke the `AudioRecord.stop()` method

These steps look very trivial, and anyone who knows how to use a loop and arrays should be able to write the code. However, there are details you need to know for proper audio recording setup:

- How big a buffer do you have to allocate? Allocating too small a buffer causes a common overrun error and your program may miss some of the audio data. Allocating to much buffer space is unnecessarily wasting your device memory.

- Should you allocate a `byte` (8-bit), `short` (16-bit), of `float` (32-bit) buffer?

Unfortunately, these parameters are hardware and device-driver dependent. Moreover, they may vary among devices and API levels. For instance, Table 10.1 shows the variations of these values among several Nexus devices.

	Sample Rate	Buffer Size	
		(frames)	(ms)
Nexus S	44100	880	19.95
Galaxy Nexus	44100	144 (Jellybean) 968 (Ice Cream Sandwich)	3.26 21.95
Nexus 4	48000	240	5
Nexus 7	44100	512	11.6
Nexus 10	44100	256	5.8

Table 10.1: Audio Sampling Parameters Across Nexus Devices (source: High Performance Audio, Google I/O 2013)

The audio recorded using the above approach is a "raw" audio. Neither compression nor specific codec[1] algorithm is applied to the incoming audio data. Raw audio files are *not* automatically recognizable by standard media players.

For general applications, you want to record and save audio files in one of the well-known formats (MP3, 3GP, WAV, etc.). For that very reason, the `AudioRecord` class is usually not the best fit for most applications. Hence, we want to look for another alternative: `MediaRecorder`.

Using the `MediaRecorder` class to record audio is very straight forward. There is no need to allocate your own data buffer and write a loop, and more importantly you don't have to worry about the actual sampling frequency used by the audio driver on your mobile device. All these details are handled internally by `MediaRecorder`. Audio recording with using this class requires only a small number of code:

Kotlin

```
// In response to "Record" button
// Prepare MediaRecorder
audioRec = new MediaRecorder();
audioRec.setAudioSource(MediaRecorder.AudioSource.MIC);
audioRec.setOutputFormat(MediaRecorder.OutputFormat.MPEG_4);
audioRec.setOutputFile(path_to_your_audio_file);
audioRec.setAudioEncoder(MediaRecorder.AudioEncoder.AMR_NB);
audioRec.setAudioSamplingRate(256);
audioRec.prepare();

// start recording
audioRec.start();
```

Prior to calling `start()` to begin recording, the class must be configured for its input, encoding algorithm, and audio format. The `MediaRecorder` class supports many different media

[1] "coding" and "decoding"

formats, but we are looking for a format that plays on both Android and iOS. The MP4 format seems toi work well on both platforms.

In the snippet we use:

- Microphone as audio source
- Adaptive Multi-Rate Narrow-Band encoding[2]
- MPEG-4 file output format
- 256 Hz sampling rate

You can save the recorded audio in different formats by changing the actual arguments passed to `setOutputFormat()` and `setAudioEncoder()`. Refer to Android Supported Media Format for various possible combinations of these arguments.

To stop recording, we need even fewer lines of code:

Kotlin

```
// Stop recording
audioRec.stop()
audioRec.release()
audioRec = null
```

> **NOTE**
>
> At the time of this writing we are using Android emulator version 30.1.5 and the recorded audio was distorted. If you have the same issue, you should run the app on a physical Android device.

Let's put all these concepts together into code by writing a new fragment class, `AudioFragment` for recording audio (and also playback).

To maintain uniform user interactions in handling all media types (photos, videos, and audio), we will follow the screen transition sequence in Figure **??**. The completed user interface should look like Figure 10.2. The overall user interface design includes the following widgets:

- The top text view shows the current state of the audio activity, such as "playing", "recording", "paused", etc.
- The bigger text view immediately below the current state shows the current time marker of the recording/playback in "minute:second". Use the builtin style **AppCompat.Display2**, to make the text bigger.
- The three action buttons are (from left to right)
  1. "Reset" (also "Rewind" or "Retake")
  2. "Record" (also "Stop" or "Confirm"), and
  3. "Play" (also "Pause")

[2] frequency range 200–3000Hz which is equivalent to analog conventional phone line quality

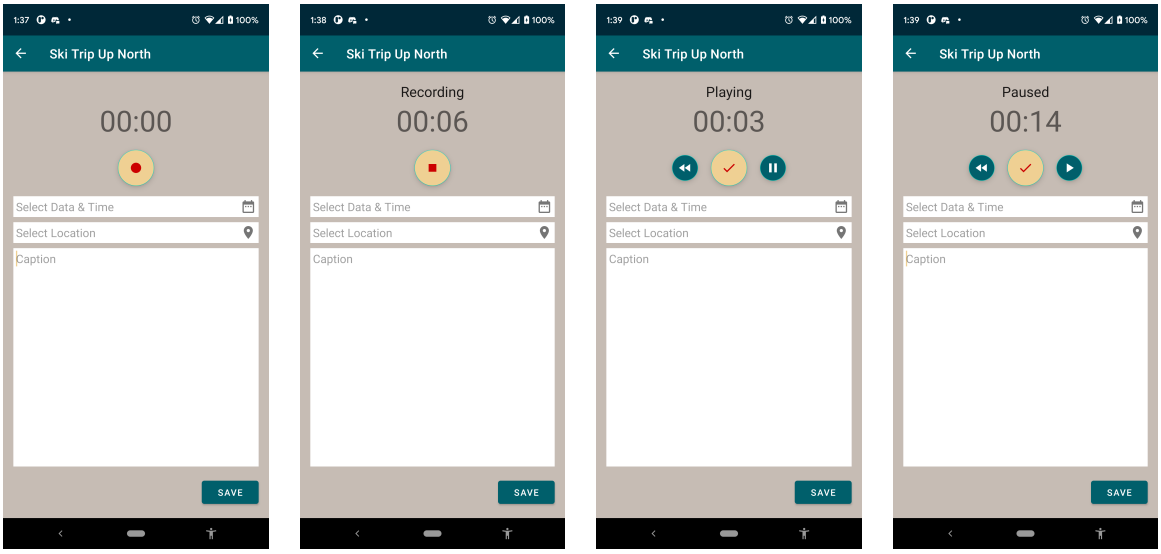


Figure 10.2: Various states of audio control

The off-center FABs are set to "mini" size and its backgroundTint to "colorPrimary". Recommended Material Design icons to use for your action buttons are shown in Table 10.2.

The visibility of the three action buttons are controlled programmatically depending on how your user interacts with the activity.

- At the start of a recording session, only one button ("Record") is visible.
- During a recording session, this button turns into a "Stop" button.
- When the recording stops, the center button turns into a "Confirm" button and two smaller buttons appear: "Retake" (to discard the current recording and take a new one) and"Playback" (to play the audio just recorded). The "Play" button also toggles into "Pause".

Function	Icon
Record	`av/fiber_manual_record`
Play	`av/play_arrow`
Rewind	`av/fast_rewind`
Stop	`av/stop`
Pause	`av/pause`

Table 10.2: Audio Icons

Changing the buttons visibility and icon can be done in a few lines of code, similar to what you have done earlier. For instance, during recording, the left and right buttons are hidden by setting its visibility to GONE and the middle image button icon is updated to a the "Stop" icon:

Kotlin

```
this.resetBtn.setVisibility (View.GONE);
this.playBtn.setVisibility (View.GONE);
this.recordBtn.setImageResource (R.drawable.ic_stop_black);
```

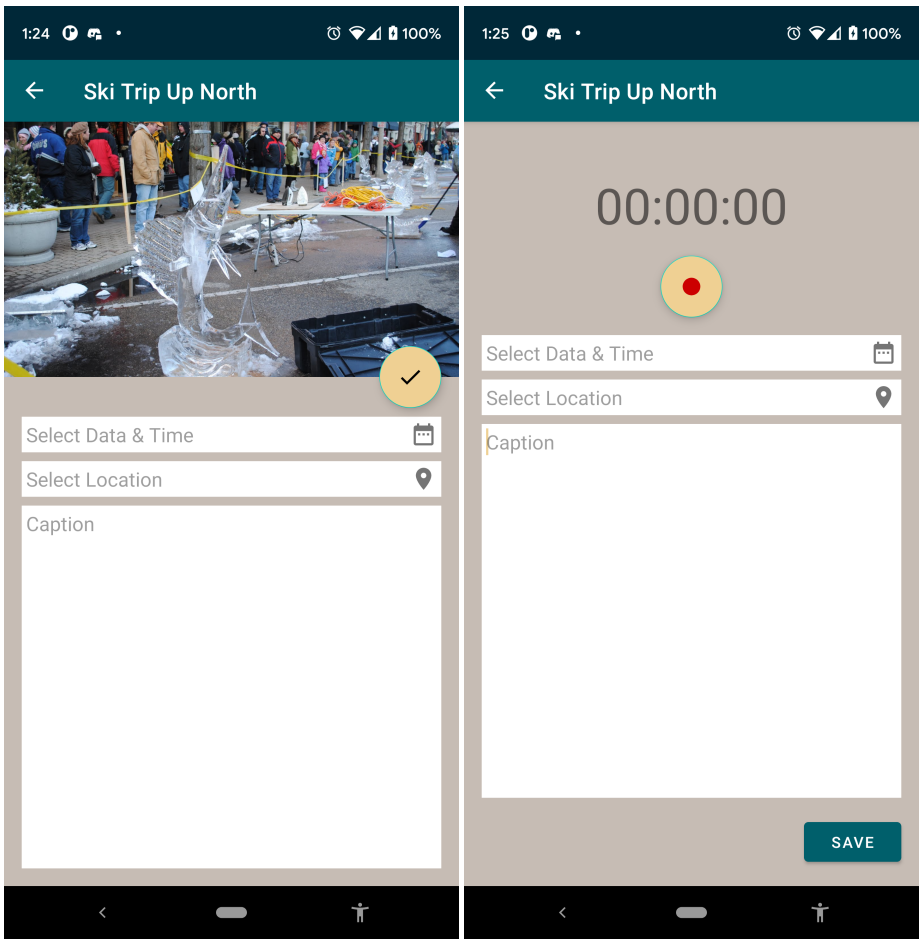


Figure 10.3: The screen sharing a common input form

At this point you may notice that our UI shows a common and repetitive design: every time the user interacts with a form to enter detailed information of a media file, the UI shows the same three fields (date, location, and caption). In our app, this form is repeated in three different places: `MediaDetailsFragment`, `MediaEditFragment`, and `AudioFragment`. Such an example is shown in Figure 10.3.



Figure 10.4: Common input fields for media meta data

For easy maintenance of the code, it would be wise to factor out this section of the UI into its own fragment. Let's call this fragment: `MediaMetaDataFragment`. Its UI design is shown in Figure 10.4 and can be designed using **vertical** `LinearLayout`. The three main widgets are:

1. `date_time` a TextView to allow the user select a date and time from a date picker dialog
2. `location` a TextView to allow the user select a geographic location using the Google Places API
3. `caption` a multiline EditText for entering the caption of the media. To make this widget fill the remaining vertical free space, set its `layout_weight` to 1 (but do not set the `layout_weight` of the other two widgets above).

Handling user input should be done by this fragment and stored into the view model class. Therefore, most of the code you added in Section (displaying the date picker and time picker) should be moved to `MediaMetaDataFragment`. To observe any changes to the multiline text field you can use its text changed listener:

Kotlin

```
caption.addTextChangedListener {
  mediaModel.mediaCaption.value = caption.text.toString()
}
```

The overall sketch of `MediaDetailsFragment` is shown in Listing **??**. Refer to the details from the code you added in an earlier chapter.

Kotlin

```kotlin
class MediaMetaDataFragment : Fragment() {
  val mediaModel by activityViewModels<MediaViewModel>()
  val datePickerDialog = MaterialDatePicker.Builder.datePicker().build()
  val timePickerDialog = MaterialTimePicker.Builder().build()

  override fun onCreateView(
      inflater: LayoutInflater, container: ViewGroup?,
      savedInstanceState: Bundle?,
  ): View? {
      // Inflate the layout for this fragment
      return inflater.inflate(R.layout.fragment_media_meta_data, container, false)
  }

  override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
      super.onViewCreated(view, savedInstanceState)
      datePickerDialog.addOnPositiveButtonClickListener {
          mediaModel.mediaDate.value = DateTime(it, DateTimeZone.UTC)
          timePickerDialog.show(parentFragmentManager, "timePick")
      }
      timePickerDialog.addOnPositiveButtonClickListener {
          val updated = mediaModel.mediaDate.value?.run {
              plusHours(timePickerDialog.hour).plusMinutes(timePickerDialog.minute)
          }
          mediaModel.mediaDate.value = updated
      }
      date_time.setOnClickListener {
          datePickerDialog.show(parentFragmentManager, "datePick")
      }
      location.setOnClickListener {
        // Launch the Place API UI here
      }
      caption.addTextChangedListener {
          mediaModel.mediaCaption.value = caption.text.toString()
      }
  }

  override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {
    // store the place details into mediaModel
  }
}
```

Once the `MediaMetaDataFragment` is fully implemented, you can reuse it in `MediaDetailsFragment`, `MediaEditFragment`, and `AudioFragment`. Simply remove the three fields from the layout of one of these fragments, and replace it with the following:

XML
```
<fragment
  android:id="@+id/media_meta"
  android:name="edu.gvsu.cis.traxy.MediaMetaDataFragment"
  ... other attributes here
  tools:layout="@layout/fragment_media_meta_data"
  />
```

The last attribute (`tools:layout`) is needed to inform the layout editor to show the preview of the fragment. Specifically, the above snippet assumes that the layout file name used by `MediaDetailsFragment` is `fragment_media_meta_data.xml`.

Go ahead and implement the user interface as described above!

Because the UI provides only three buttons for performing six different functions (*record*, *stop*, *play*, *pause*, *save*, and *reset*), you need a mini state machine to handle the user interactions with only three buttons. One possible implementation is to use enum with the following six states: `START`, `RECORDING`, `RECORD_STOP`, `PLAYING`, and `PLAY_PAUSE`. Refer to Figure 10.5. In the figure, red lines show the transitions for the center ("Record") button, blue lines show the transitions for the right ("Play") button, and the broken lines for the left ("Reset") button.

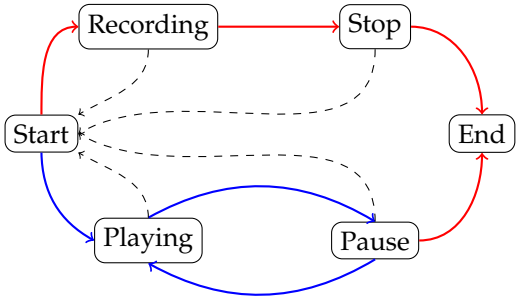


Figure 10.5: `AudioFragment` State Machine. The fragment terminates at the "End" state

Requesting Permissions at Runtime

Recording audio requires the following permission in your manifest:

XML
```
<uses-permission android:name="android.permission.RECORD_AUDIO" />
```

However, this permission is considered dangerous and starting with Android 6.0 (API level 23) apps that use this permission are required to request user approval at runtime. This process calls for the following three steps added to your fragment:

1. Invoke `requestPermission()` from `onViewCreated()` to prompt the user for approval if the desired permission has not been granted. This method takes the following two arguments:

(a) The first argument is an array of permissions requested

(b) The second argument (the `requestCode`) is an integer value that uniquely identifies this permission request so proper action can be taken in the function `onRequestPermissionsResult()`. Your code must define and initialize this value.

Kotlin

```
requestPermissions(arrayOf(Manifest.permission.RECORD_AUDIO),
  REQUEST_RECORD_AUDIO_PERM)
```

2. Override the `onRequestPermissionsResult` to determine your user's response. This method takes the following three arguments:

(a) The first argument is the request code passed in the previous step

(b) The second argument is an array of permissions

(c) The third argument holds the user response (allow or deny)

Kotlin

```
override fun onRequestPermissionsResult(
  requestCode: Int,
  permissions: Array<out String>,
  grantResults: IntArray,
) {
  super.onRequestPermissionsResult(requestCode, permissions, grantResults)
  if (requestCode == RECORD_AUDIO_PERM_REQUEST) {
      recordPermissionGranted = grantResults[0] == PackageManager.PERMISSION_GRANTED
  }
  if (!recordPermissionGranted)
      findNavController().popBackStack()
}
```

Let's make the following changes to your `AudioActivity` to add the necessary code for requesting user permission approval at runtime:

On Android 6.0 devices, the runtime permission dialog looks like Figure 10.6. When your callback method is invoked, the `grantResults` array will hold either `PERMISSION_GRANTED` or `PERMISSION_DENIED` depending on the user's response. Moreover, to determine if the user checked "Don't ask again" when responding "Deny", the callback method should invoke `shouldShowPermissionRequestRationale()`, a method that returns a boolean result (`true` when the user **did not** check the checkbox).

When this method returns `false`, subsequent launches of `AudioFragment` **will not** show the permission dialog but the above callback method will continue to get `PERMISSION_DENIED` and executes the `popBackStack()` method. When your user relaunches this activity, let's say, a few weeks later, she probably forgot that she permanently denied the audio recording permission, but she now thinks that the `AudioFragment` failed to launch. To avoid this confusing situation, you should probably disable Traxy audio recording feature when the user decides to deny the permission "forever". We leave the innermost if-statement for you to complete the necessary bookkeeping logic.

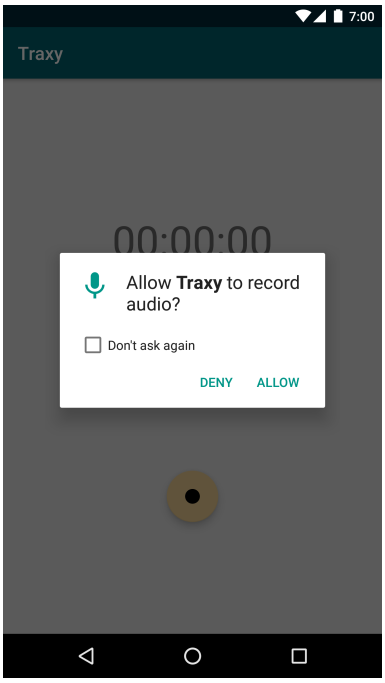


Figure 10.6: Runtime Permission Dialog

Kotlin

```
when (currentState) {
  Status.START -> {
    startRecording()
    currentState = Status.RECORDING
    audio_state.setText("Recording")
  }
  Status.RECORDING -> {
    stopRecording()
    currentState = Status.RECORD_STOP
    audio_state.setText("Recorded")
  }
  Status.RECORD_STOP, Status.PLAY_PAUSE ->
    findNavController().popBackStack()
}
```

Listing 125: State Machine Implementation of the "Record" button

The finite state machine implementation of the center/record button (the red line in Figure 10.5) is shown in Listing 125, and for the right/play button (the blue line in the figure) is shown in Listing 126. For brevity of presentation, other details not related to audio device manipulation (such as updating icons, hiding unused buttons, updating text views, resetting counters, error handling, etc.) are *omitted from the listings*.

Timer Update in Background

To let your user know how long she has recorded the audio, the UI shows a timer marker at the top of the screen. Updating the time marker requires a special technique. You can't simply write a loop that includes a one second delay for updating the time marker text view:

Android

```
duration = 0;
while (inRecordMode) {
    timeMarker.setText (toMinSec(duration));
    duration ++;
    Thread.sleep(1000); // delay 1000 millisecs
  }
```

The above code **would not work**! The loop deprives the UI thread from the rest of your app, making it unresponsive to any subsequent UI events. The call to `setText()` for updating the time marker must, however, take place within the UI thread. Your challenge now is to perform a *periodic* UI update without using a loop. It looks like an "alarm-like" signaling mechanism is a great help for solving this problem. If we can find a simple periodic signaling mechanism which is also attached to the UI thread, that will be the most preferred solution.

There are several options for implementing periodic timers. In the following example we will use the Android `Handler` together with Java `Runnable` class. The two `Handler` methods of our interest are: `post` and `postDelay`. Both of these methods take a `Runnable` object as an argument, and we can place the code needed to update the time marker in a Runnable.

The very first step is to declare a `Handler` object:

Kotlin

```
private val myHandler = Handler()
```

and then define the `Runnable`:

Kotlin

```
private val myRunner:Runnable = Runnable {
  elapse_time++
  time_marker.setText(elapse_time.toMinuteSecond())
  myHandler.postDelayed(myRunner, 1000) // post this Runnable 1 second later
}
```

However, the above code does not work in Kotlin; the compiler will complain that `myRunner` is not initialized (yet). To fix the issue we should use *lazy initialization* below:

Kotlin

```
private val myRunner:Runnable by lazy {
  Runnable {
    elapse_time++
    time_marker.setText(elapse_time.toMinuteSecond())
    myHandler.postDelayed(myRunner, 1000)
  }
}
```

The function `toMinuteSecond` in a user-defined extension on `Int` type defined as follows:

Kotlin
```kotlin
fun Int.toMinuteSecond() : String {
  var seconds = this
  val h = seconds / 3600    /* unused */
  seconds %= 3600
  val m = seconds / 60
  seconds %= 60
  return String.format("%02d:%02d", m, seconds)
}
```

The function can be defined in any Kotlin file, and in the sample code you'll find this function defined inside `AudioFragment.kt`, but outside the class `AudioFragment` itself.

With the Handler and Runnable defined, you can initiate periodic events by calling:

Kotlin
```kotlin
myHandler.post(myRunner)
```

and stop the periodic events by calling

Kotlin
```kotlin
myHandler.removeCallbacks(myRunner)
```

Incorporating these two function calls into the audio recording logic, the listener of the center button should now look like:

Kotlin
```kotlin
when (currentState) {
  Status.START -> {
      startRecording()
      currentState = Status.RECORDING
      elapse_time = 0
      myHandler.post(myRunner)  // fire periodic events
  }
  Status.RECORDING -> {
      stopRecording()
      currentState = Status.RECORD_STOP
      myHandler.removeCallbacks(myRunner) // stop periodic events
      elapse_time = 0
  }
  Status.RECORD_STOP, Status.PLAY_PAUSE ->
      findNavController().popBackStack()
}
```

Go ahead and make necessary changes to your current state machine, incorporate the snippet in Listing **??**.

Run your app and be sure the UI behaves as expected, i.e.:

- Buttons should appear and disappear correctly
- The icons on the center and right buttons should update correctly
- The timer marker is updating correctly when you are in "Record" or "Play" mode
- The text view status above the time maker reflects the current state of your recording/playback

10.2.2 Audio Playback

Implementation of audio playback parallels the recording counterpart. The Android class to use for playback is `MediaPlayer`.[3] As you may notice that this class is also used for video playback, but in this section you will use it only for audio. To simplify the code further, you will ignore handling audio focus for now. You will revisit that topic towards the end of this chapter.

The state machine for the play button (the rightmost button) simply toggles between two states so its implementation is pretty straight forward as shown in Listing 126 with the details of the respective private methods in Listings 127 and 128.

Unlike audio recording where the user initiates a stop to end the recording, playing back an audio stream requires the app to automatically detect the end of the stream. Line 3 of Listing 127 attaches a completion listener to the media player object and the corresponding listener is shown in Listing **??**.

Kotlin

```
1  rightBtn.setOnClickListener {
2    when (currentState) {
3      Status.PLAYING -> {
4        pausePlayback()
5        currentState = Status.PLAY_PAUSE
6        myHandler.removeCallbacks(myRunner)
7      }
8      else -> {
9        startPlayback()
10       currentState = Status.PLAYING
11       myHandler.post(myRunner)
12     }
13   }
14 }
```

Listing 126: State Machine Implementation of the "Play" button

Unlike audio recording where the user initiates a stop to end it, when playing back an audio stream, you want the app automatically detects the end of the stream. Fortunately the `MediaPlayer` class provides a completion listener for this purpose.

Run you app on a physical device and be sure that you are able to record and playback audio. The latest version of Android emulator used at the time of this writing cannot handle audio very well yet. The resulting audio recording is distorted.

The audio programming technique implemented so far works great, but there is one lingering issue that may make your user feels unhappy. When the user exits from your app or moves back to the originating fragment, the audio may continue to record or playback.

To automatically stop/pause audio recording/playback when you app becomes hidden, you have to write the necessary code in `onPause()`

[3] You can, of course, use the ExoPlayer class to play audio files as well.

Kotlin

```
private fun initAudioPlayer() {
  audioPlay = MediaPlayer().apply {
    val audioAttr = AudioAttributes.Builder()
      .setContentType(AudioAttributes.CONTENT_TYPE_MUSIC)
      .setUsage(AudioAttributes.USAGE_MEDIA)
      .build()
    setAudioAttributes(audioAttr)
    setOnCompletionListener {
      currentState = Status.PLAY_PAUSE
      rightBtn.setImageResource(R.drawable.ic_baseline_play_arrow_24)
    }
    mediaModel.mediaUri.value?.let {
      setDataSource(requireContext(), it)
      prepare()
    }
  }
}

private fun startPlayback() {
  try {
    if (audioPlay == null)
      initAudioPlayer()
    audioPlay?.start()
    rightBtn.setImageResource(R.drawable.ic_baseline_pause_24)
  } catch (ioe: IOException) {
      // error handling
  }
  audio_state.setText("Playing")
}
```

Listing 127: Start Playback

Kotlin

```
override fun onPause() {
  super.onPause()
  audioRec?.let {
    stopRecording();
  }
  audioPlay?.let {
    pausePlayback();
  }
}
```

The final step is handling the "Reset" button, which essentially performs a number of re-initializing actions:

- Reset the current state to `START`
- Restore the icon and visibility of all the action buttons
- Reset the time marker
- Remove any pending runnable from the `Handler`

Kotlin
```
private fun pausePlayback() {
  rightBtn.setImageResource(R.drawable.ic_baseline_play_arrow_24)
  audioPlay?.pause()
  audio_state.setText("Paused")
}
```

Listing 128: Pause Playback

- Stop current recording or audio playback

We leave the details of this method as an exercise for the readers.

Our next step is to upload the audio file to Firebase storage when the user confirms to save the audio. Using the code snippet in Listing 105 as your reference, the code for uploading the audio file to Firebase is:

Kotlin
```
// Inside the click listener of the "Save" button
with(mediaModel) {
  val mediaObj = JournalMedia(
      caption = mediaCaption.value ?: "None",
      date = mediaDate.value.toString(),
      type = MediaType.AUDIO.ordinal,
      lat = selectedJournal.value?.lat ?: 0.0,
      lng = selectedJournal.value?.Lng ?: 0.0
  )
  CoroutineScope(Dispatchers.IO).launch {
      mediaModel.addMediaEntry(mediaObj, mediaUri.value!!)
  }
  findNavController().popBackStack()
}
```

- Set its content type to `audio/m4a`
- Store all audio files under the `audio` sub-directory

After verifying that audio files are uploaded correctly to your Firebase Storage, you should make additional changes to other classes so your app handles audio files correctly. Essentially any classes that manipulate photos or videos should be enhanced to properly handle audio files as well. This includes:

1. `JournalMediaViewHolder` to show/hide the proper icons suitable for audio media items
2. `MediaDetailsFragment` to adjust the visibility of some of the icons in the input form and to set the correct media type before the media file is uploaded
3. `MediaEditFragment` to adjust the visibility of some of the icons in the input form
4. `MediaViewFragment` to adjust the visibility of some of the icons in the input form and enable ExoPlayer for playing the audio file

10.2.3 Audio Focus

What is "audio focus" and why should you bother understand it? Well, let's consider a common task that you perform frequently: typing text. Prior to using your keyboard to enter data into a form with multiple input fields, you must first click on a selected input field. This action of clicking an input field assigns the *keyboard focus* to the selected field. Why is keyboard focus important? Because the keyboard cursor is shared among the fields, and the keys you press should be delivered only to one of them.

A similar situation applies to using audio output devices. Your app is not the only one using audio output, many other apps compete with yours for using audio output devices. There has to be a mechanism for controlling which app is allowed access. For instance, when an incoming phone call interrupts your app while it is playing a sound, you should stop the playback, let the system phone use the speaker, and resume playback when your app regains control. To be considered a courteous citizen among the Android app ecosystem, your app should comply with the Android standard protocol for using audio devices.

To enable such a frictionless cooperation between your app and the others, Android provides the `AudioFocus` class. Before using audio output, your should request for audio focus. Once granted, the focus may be taken away anytime. Hence, you must setup a focus change listener to get notifications for audio focus gain or loss events. The `AudioManager` class provides methods for setting up a `OnAudioFocusChangeListener`.

There are at least four different events to handle in your audio focus listener, three possible causes of focus loss and one case of focus gain:

1. `AUDIOFOCUS_LOSS`: upon receiving this event, you app should stop using the audio output. This is a permanent loss of focus, your app will never gain focus again unless it asks for it.

2. `AUDIOFOCUS_LOSS_TRANSIENT`: upon receiving this event, you app should pause audio output temporarily

3. `AUDIOFOCUS_LOSS_TRANSIENT_CAN_DUCK`: this event implies that your app may continue using the audio output, but using a lower volume. To adjust output volume, your app should call `AudioManager.adjustStreamVolume()`.

4. `AUDIOFOCUS_GAIN` or `AUDIOFOCUS_GAIN_TRANSIENT`: your app regains full control of the audio output device. The former is for an extended amount of time and the latter is only for a short amount of time.

> NOTE
>
> The code snippet below is adapted from the Android Developers documentation page on https://developer.android.com/guide/topics/media-apps/audio-focus with a number of simplification.

The audio focus listener should be installed via Android `AudioManager` service which can be initialized by calling `getSystemService`:

Kotlin
```
override fun onActivityCreated(savedInstanceState: Bundle?) {
  super.onActivityCreated(savedInstanceState)
  audioMgr = requireActivity().getSystemService(Context.AUDIO_SERVICE) as AudioManager
}
```

The Focus change listener can then be designed to respond to various focus change events. In the following snippet, we simply use the listener to resume or pause playback.

Kotlin
```
val afChangeListener = AudioManager.OnAudioFocusChangeListener {focusChange ->
  when(focusChange) {
      AudioManager.AUDIOFOCUS_GAIN -> startPlayback()
      AudioManager.AUDIOFOCUS_LOSS -> stopPlayback()
      // more when clauses as needed
      else -> { /* default action */}
  }
}
```

Thereafter, request for audio focus can be performed inside your `initAudioPlayer()`. Depending on the API level of the target device, initializing audio focus requires different function calls and arguments. Prior to Android 8.0 (Oreo), it requires only a few lines of code:

Kotlin
```
// Prior to Android 8.0
val result = audioMgr.requestAudioFocus(afChangeListener,
  AudioManager.STREAM_MUSIC,
  AudioManager.AUDIOFOCUS_GAIN) // Request permanent focus of output dev
playbackAuthorized = result == AudioManager.AUDIOFOCUS_REQUEST_GRANTED
```

On the contrary, on devices running Android 8 or newer, the initialization requires a longer amount of code.

Kotlin
```
// Android 8.0 or later
audioFocusRequest = AudioFocusRequest.Builder(AudioManager.AUDIOFOCUS_GAIN).run {
    setAudioAttributes(AudioAttributes.Builder().run {
        setUsage(AudioAttributes.USAGE_MEDIA)
        setContentType(AudioAttributes.CONTENT_TYPE_SPEECH)
        build()
    })
    setAcceptsDelayedFocusGain(false) // Request immediate focus
    setOnAudioFocusChangeListener (afChangeListener)
    build()
}
val result = audioMgr.requestAudioFocus(audioFocusRequest)
playbackAuthorized = result == AudioManager.AUDIOFOCUS_REQUEST_GRANTED
```

If you plan to support devices from both pre and post Android 8.0, you can combine the two snippets above in the following if-else statement:

Kotlin
```
if (android.os.Build.VERSION.SDK_INT >= android.os.Build.VERSION_CODES.O) {
  // initialize auto focus for Android 8.0 or later
} else {
  // initialize auto focus for older API levels
}
```

The boolean flag `playbackAuthorized` can be use to conditionally start the audio playback:

Kotlin

```
private fun startPlayback() {
  try {
      if (audioPlay == null)
          initAudioPlayer()
      if (playbackAuthorized) {   // start playing only after gaining focus
          audioPlay?.start()
          rightBtn.setImageResource(R.drawable.ic_baseline_pause_24)
      }
  } catch (ioe: IOException) {
      // error handling
  }
  audio_state.setText("Playing")
}
```

—The most compelling reason for most people to buy a computer for the home will be to link it to a nationwide communications network. We're just in the beginning stages of what will be a truly remarkable breakthrough for most people – as remarkable as the telephone.

Steve Jobs

11 Networking

Chapter Objectives

After reading this chapter you should be able to

- Understand how to make requests on a remote web API.
- Understand how to parse JSON representations.
- Understand how to deal with the asynchronous behavior of network integration on Android and iOS.

11.1 Accessing Web APIs in Mobile Apps

Before the mobile phone was a general purpose application platform, its only application was dedicated to communicating in the form of full-duplex voice communication. The conventional smart phone continues its original role in facilitating communication, however today, the context of this communication is often the apps installed on the phone. In fact, it is very difficult to find a popular mobile application today that does not make extensive use of the mobile phone's ubiquitous network connectivity to interact with various web application programming interfaces (APIs). Before flashlight apps became bundled with our phones from the factory, even the multitude of flashlight apps in the app stores used network connectivity, as they were often monetized by advertising!

Mobile applications typically use network services to make HTTP requests to web APIs provided by applications in the cloud. For example, a mobile app might transmit its local geocoordinates to a remote web API to locate nearby Chinese restaurants, or it might use a Twitter API to tweet a photo it just captured. Our Traxy app has already been using network services in

the form of the Firebase SDK pods we incorporated previously. These components made it very easy for us to use Firebase cloud services for user management, database services, and persisting / retrieving multimedia. However, there are many interesting web APIs applications can access that do not provide an easy to use custom SDK. Instead, we have to familiarize ourself with how to interact with the API in terms of HTTP requests and responses.

Happily, both Android and iOS provide frameworks for making HTTP requests and processing the responses. In addition, there are many third party libraries for both platforms that provide us with additional functionality for handling authentication, caching and other network-related features.

In this chapter we are going to extend our Traxy app to access an external web API directly. However, before we look at how that's done on iOS and Android, let's consider some of the challenges we face and some general guidelines that we can apply to both platforms.

11.1.1 Network Programming Challenges

Unlike network programming on a high performance server mounted in a rack in a machine room, network programming on a handheld device has some challenges that require special consideration. Many, but not all of these challenges deal with the fact that the device is *resource constrained*. Other challenges relate to security and privacy, and also the inherent complexities of network programming.

Resource Challenges

Smart phones and tablets all depend on a very limited portable energy source — their onboard batteries. Accessing the network is accomplished using the device's radio (or possibly radios!) Transmitting and receiving data takes a non-trivial amount of energy and an application that makes injudicious use of this resource will quickly find itself deleted from the device, perhaps along with a negative review in the app store.

In addition, the nature of wireless networking today poses challenges of its own in terms of latency constraints, bandwidth constraints, variability / reliability, and service restrictions. In networking, *latency* refers to the delay involved in relaying a signal from point A to point B, while *bandwidth* refers to the amount of data that can be transmitted from point A to point B in a given quantum of time. The latency in wireless networks is typically an order of magnitude higher than wired network transmissions. Latency can vary also depending on the wireless technology being utilized. For example, though the more modern LTE networks have latency somewhat lower or at least similar to WiFi, many of the older wide area data networks still widely used have even higher latency. Even in the face of plenteous bandwidth, a network context characterized by high latency can have a very negative impact on the usability of the app. For example, imagine the frustration of playing a fast twitch, multiplayer video game in the face of high latencies, or the frustration of experiencing significant lag in a full duplex video chat with a friend.

Modern LTE networks represent a 10X improvement over older 3G technologies, and in many markets (around 80% penetration in the USA) offer download speeds that rival or surpass what most consumers in the USA get from their wired home broadband services. However, given the dismal state of affairs when it comes to broadband services in the USA (when compared to other developed countries) this might make the situation sound better than it actually is. In any case, many service providers such as Verizon now offer LTE-based home broadband services that are

comparable with the broadband services offered by cable and landline phone companies. However, despite the bump in bandwidth we've been enjoying on our wide area wireless networks in recent years, users (and hence their apps!) are seriously constrained by the restrictions the network operators place on their services. For example, many smart phone users are limited to a quota of 1-2Gb of date usage per month, and face significant performance degradation or additional fees when they go over the established limit. Many apps take this into account and warn (or prevent) users from transmitting or receiving large amounts of data over the wide area network services.

Finally, even if a user has an unlimited data plan, another constraint is the lack of consistency in terms of service availability, and the variability of performance. While we like to think that network access is fairly universal nowadays, there is plenty of anecdotal evidence that often our applications are running in an environment where they lack service altogether, or have very limited service in terms of latency / bandwidth (e.g. take a look at some of the reports from the companies who provide mobile analytic services for starters). Furthermore, the latency can be highly variable on wide area networks which can be problematic for streaming media and similar sorts of use cases. Anybody who lives in a rural community in the USA (around 25% of the population in our home state of Michigan) is all to familiar with these issues when using data services on their phone.

Security and Privacy Challenges

In a certain sense, security and privacy are top trending concerns for computing in general. There has been no shortage of coverage of major security breaches in recent years that compromise our personal identities in harmful ways, and risk compromising elections, national defense systems, and much more. Nevertheless, from an end user perspective, these issues are even more preponderant when it comes to our mobile phone. Our smart phones are actually our first truly *personal computers*. The devices traditionally known as personal computers are "personal" in the sense that they are typically used by one person at a time, unlike the massive computers they replaced that were locked into machine rooms and used by hundreds of people simultaneously. However, they are not "personal" in the true sense of the word, as they sit out in public lobbies on college campuses for all to use, are shared at the local library, or even at home where we refer to them as the "family computer" since all members of the household use them at times.

The smart phone however, is truly personal. It contains messages, photos, access to our financial accounts, and many other things that we as individuals hold dearly. Most smart phones are used by one and only one person and closely guarded. For example, if you were approached by a stranger in the supermarket parking lot who explained their car broke down and they need to call for help, you likely would not simply hand them your $600 smartphone over to them, but rather offer to make a call on their behalf.

Furthermore, the smart phone, unlike the personal computer of the past is sort of the de facto ubiquitous computing device - always with us, always connected. Although assertion has been challenged in the literature (e.g. our phones are not on and connected as much as we think they are [18, 8].) the smart phone, more any widely adopted device that has come before it, enables the sort of computing that Mark Weiser and other thought leaders in ubiquitous computing envisioned decades ago [20]. For example, along with its network connectivity, the smart phone is a context aware computing platform, with the ability to intelligently sense its environment (location sensing via GPS, light/proximity sensing, audio sensing, motion sensing etc.) and inference in ways that introduce new forms of computing. Along with all the potential comes the risks.

Sometimes the more rigorous vetting that takes place before an app is published in the Apple iTunes App Store gives iOS users a sense of false security, after all its Android that is the wild west of mobile ecosystems and has all the unruly and malicious apps. However, there is no shortage of vulnerabilities that have been reported on iOS, including the recent SSL vulnerability and the Early Random PRNG kernel exploit, not to mention Apple's very own infamous "LocationGate" debacle. The simple fact is that users of all smart phones face security and privacy issues, and we as app developers need to understand and mitigate these risks as much as possible for users who use the apps we write.

Programming Challenges

The challenges concerning network programming arise from two simple observations: (1) network requests take time and (2) wireless networks are inherently unreliable. The former observation means that the programmer must make special provisions to ensure the application remains responsive while data is being fetched from the network. In other words, network requests are typically not going to be handled *synchronously*, but rather *asynchronously*. That is, the application might initiate a fetch to the network when the user hits a search button, but is not going to wait around for the search results. Instead, we'll hand it a chunk of code that will eventually get executed by another background thread in the future, when the fetch completes (or runs into problems.) On both Android and iOS this means, that if the results impact our user interface (almost always) we need to somehow dispatch that update back over to the main thread which has the exclusive responsibility of maintaining the user interface. Occasionally, your app may have to launch *several* concurrent background threads to fetch data from the network. For instance, to render a list of thumbnail images, each image file may have to be downloaded using a separate fetch. A potential problem that may arise from having concurrent background threads is a *race condition* when these threads attempt to update a shared data structure.

In addition to the complexities around synchronicity, there is myriad other details we often need to be concerned with when fetching data from the network. For example, we might want to have some sort of network activity indicator on screen to inform the user that the network is being accessed. Hence, we need to make sure the user interface is updated at the right time to alert the user of ongoing network activity. Another detail is whether or not the result of a network access is going to be cached. If so does the library we are using handle that for us or do we need to roll our own cache? What if there are dependent follow on fetches? For example, sometimes a representation return from one fetch has links to additional resources that also need to be fetched. How to we organize this code? How do we know when all of the fetching has been completed?

The inherent unreliability of wireless networks means we often have to retry requests and make reasonable decisions about when to back off and let the user know we're having problems and that they should try later. By default the APIs we get in iOS and Android do not handle these sorts of issues, which helps explain in part why there are so many third party libraries in usage today.

11.1.2 General Guidelines for Networking in Mobile Apps

It turns out that a bit of common sense and paying attention to best practices can mitigate most of the challenges above to the extent that your app (even though it consumes some battery and quota from the user's monthly data plan) can still offer the end user an attractive value proposition once installed. Let's look at some general guidelines that apply to both iOS and Android, as well as any

other mobile platform you might be working with. (Don't get any ideas, there are probably more devices in the wild running Ubuntu Touch than there are running Windows Phone!)

Some general guidelines to become familiar with and practice include:

1. *Make sure the user interface remains responsive when network access is occurring*. The threading model on both iOS and Android almost force this to happen, but do make sure your app doesn't "lock up" while waiting for a network fetch to complete.

2. *Monitor conditions and help the user understand when things are going off the tracks*. For example, if a user is currently connected to a spotty 3G network and trying to upload a funny video of their cat in 1080p, maybe suggest they try the upload later when they have a WiFi connection. At minimal, when the app cannot utilize the network, be sure to inform the user what is going on. We've failed Apple's App Store review in the past for not adequately handling the error legs on our network fetches!

3. *Eliminate all unnecessary network accesses via strategically caching responses*. For example, the app is fetching the user's activity stream, and their favorite Aunt Eleanor posts a funny cat video at five minute intervals. By all means, cache Aunt Eleanor's thumbnail! Don't fetch it from the network every time it needs to be displayed.

4. *Minimize the payload size*. As a rule of thumb, try to keep the JSON representations you fetch to less than 5k. This means if the API supports pagination you'll want to take advantage of it when necessary. If you don't own or control the web API you are integrating you are of course at the mercy of those who do. Most of the APIs provided by large technology concerns such as Google and Facebook are well-designed and provide the features you need. That said however, there is no shortage of APIs out there that are not very amenable to mobile applications so do make your API choices wisely.

5. *Use API versioning and abstractions when possible*. Web APIs like your app are organic entities and subject to change. The fact is you can't always control when your users install your latest update on their devices. At the same time, you may want to take advantage of the latest rev in a new release of your app without impacting users who choose not to upgrade. If you take advantage of versioning features offered by the API, all of this is possible. At the same time, you may want to replace a third party API altogether in the future with a better one with minimal impact on your code. Fire-walling any third party APIs behind abstract interfaces is always a good way to plan for these types of inevitabilities.

6. *Make sure your app bundle itself is as lean as possible*. Often the image assets in the app come from a professional designer. Developers will frequently create placeholder images during development and incorporate the final artwork at a later phase, before submission to the store. We've worked on long lived apps in which the application bundle was severely bloated by generations of legacy image assets and other dependencies that were no longer being used. Happily, both Apple and Google provide us tooling that eliminates all excuses for posting bloated app bundles to the app stores. For example, Android Studio has a lint program which can scan our projects for potential problems, including orphaned assets. Apple recently introduced *on demand resources* and *app thinning*. The latter happens for us automatically, but the programmer needs to get involved to support on demand resources in their app.

Some guidelines that more specifically address safety, security and privacy concerns, consider the following practices when writing mobile apps:

1. *Publish a written privacy policy within your app, or via a link to content online*. Your privacy policy should make clear to your users what data you do collect, why, and what you will or won't do with it.

2. *Always present your user's with meaningful choices*. Some of our research has shown that users who authenticate with your app are consistently more loyal and active than anonymous users, which seems intuitive enough. However, that doesn't mean we should force users to authenticate. If possible, offer some level of functionality to users who do not wish to authenticate.

3. *Minimize data collection and limit retention.* Don't collect personal data from your users simply because you think you might want it for some feature that is languishing on our backlog. Only collect data that you absolutely need and then dispose of it promptly when it is no longer needed.

4. *Educate your users*. Inform them of any possible security/safety risks that your app might entail. For example, if your app has a "family plan" feature, explain the importance of each family member having their own login credentials and not sharing one account. If your app offers users a barter board to sell used items, encourage them to exchange items in a public place.

5. *Practice security by design*. Certain standard's bodies require every proposed draft standard section entitle "Security Concerns" whether or not the technology being proposed has any security implications. The motivation of course is to force system and application builders to bake security best practices into their designs from the get go. When we simply duct tape in some security provisions right before deployment we are most likely simply erecting a facade and deceiving ourselves, not to mention our users.

6. *Don't be creepy.* Need we say more? Don't create your own version of a "LocationGate" debacle. The golden rule works very well here: "Do unto others as you would have them do unto you."

11.1.3 Web API Data Representations

When interacting with web APIs, there are a tremendous number of different formats that can be used to represent data in the payloads of our HTTP messages. Early on, the W3C Web Services specifications relied heavily (literally!) on XML formats, but as web applications became increasingly more interactive via what was known at the time as asynchronous JavaScript and XML, JavaScript Object Notation or JSON has become the de facto standard for data representations in web APIs.

JSON is particularly amendable for APIs that will be consumed by native mobile applications, in that the JSON syntax is typed, and much less verbose than XML. This translates into smaller message payloads that can be efficiently processed on a resource constrained device, saving our users money, and battery. It should be no surprise therefore to learn that both iOS and Android provide us JSON parsing capabilities out of the box.

11.1.4 Extending Traxy With Weather Data

We're going to add another feature to Traxy that provides a good excuse to tinker with integrating an external web API. In particular, we are going to fetch weather data from the Open Weather API and display it within our journal entries. OpenWeather offers several pricing options. As of 2020, the free tier sets a quota of 60 calls per minute or one million calls per month, which should be sufficient for our experimental purposes.

Since a journal typically will span multiple days, and could entail a number of entries within each day, we're going to modify the journal entry listing so that each row displays an icon illustrating the conditions, and a temperature, for noon on the day the entry was created. We will use the top level journal latitude / longitude coordinates as an approximate location of where the entry was captured. Take a look at the screen shot in Figure 11.1 to see how this is going to look.

The OpenWeather APIs are self-explanatory and relatively easy to use. For instance, the following request

```
https://api.openweathermap.org/data/2.5/weather?lat=35&lon=139&appid=API_KEY_HERE
```

returns the JSON response[1] partially shown in Listing 129.

```
{"coord":
  { "lon": 139,"lat": 35},
    "weather": [
      {
        "id": 800,
        "main": "Clear",
        "description": "clear sky",
        "icon": "01n"
      }
    ],
    "main": {
      "temp": 281.52,
      "feels_like": 278.99,
      "temp_min": 280.15,
      "temp_max": 283.71,
      "pressure": 1016,
      "humidity": 93
    },
  }
```

Listing 129: Sample JSON Response from OpenWeather APIs

The OpenWeather API documentation provides a page that explain how to render icons associated with a particular weather condition. The `icon` field in the JSON response refers to the name of a PNG image. These icons can be downloaded directly from OpenWeather page.

[1] The default unit of temperature used by OpenWeatherMap is Kelvin

NOTE

You can use your favorite browser apps/extensions (like Postman or Advanced REST client) to explore how a given web API functions as well as debugging APIs while implementing your app. These extensions provide fine-grained control in sending HTTP requests to any API endpoint, and display the body of the responses received.

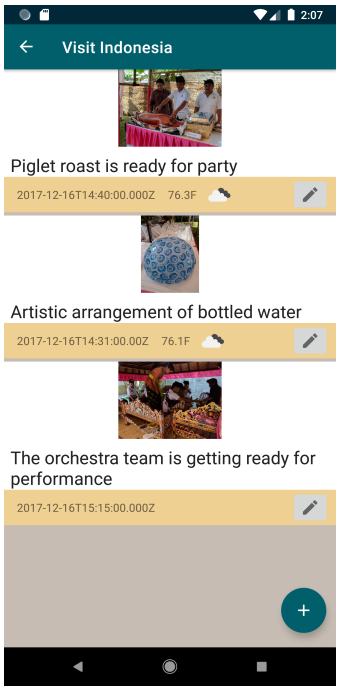


Figure 11.1: Journal entries showing weather conditions and temperature at the time the entry was made.

Due to potentially large variations of weather conditions across a relatively narrow geographic region, most Weather APIs require we supply geo coordinates when fetching weather data. The Traxy app uses geo coordinates in at least two different places:

- When a new journal is recorded, the geo coordinates represent the geographic vicinity where the travel adventures take place which may span over a longer period of time (days or weeks)
- When a journal entry is recorded, the geo coordinates represent more a specific location, point of interest visited at a particular time.

Ideally, the geo coordinates of each journal entry should be inferred from the current device location at the time the journal entry is recorded. In this chapter, instead of using ihe device location, we will "borrow" the geo coordinates of the general vicinity for fetching weather data.

11.2 Networking on iOS

Developers have several choices when it comes to network programming on iOS. Understanding the needs of your application and selecting the correct API is of course important.

1. Foundation Layer: A high level framework that provides interfaces to issue HTTP requests to web APIs and process the responses. The Foundation framework's primary focus is on accessing and transferring data via HTTP.

2. Core Foundation Layer: A lower level framework that has network protocols as its primary focus. Lower level access means as developers we can obtain more configurability and better performance, at the cost of dealing with more complexity directly in our code.

3. POSIX Layer: Essentially the same low level socket programming library that is found on any UNIX or Linux-based operating system.

Out of these options, the vast majority of the network use cases you will encounter can be implemented using the Foundation framework, and perhaps occaisonal use of the Core Foundation framework when more configurability is required than supported in the Foundation framework. Use of the POSIX layer on iOS is actually rare, and in fact discouraged by Apple in their technical documentation. One reason for this is that the POSIX layer APIs do not automatically activate the cellular radio or on-demand VPN. Nevertheless, it is available and may make sense in unusual use cases (e.g. researchy proof-of-concept type apps).

For the remainder of this chapter we will focus our attention exclusively on the Foundation framework.

11.2.1 Performing HTTP Requests From iOS

Anytime we need to request a resource with an HTTP URL in iOS, the primary class we will use is the `URLSession` class defined by the Foundation framework. `URLSession` can be used with and without the use of a delegate. When using `URLSession` without a delegate, we can pass a completion handler closure that will get executed when a response is received, or the request fails.

Use `URLSession` with a delegate whenever your application needs more control over the requests your app is issuing. For example, some web APIs may have authentication mechanisms that require more direct involvement by the programmer. Or, if you are downloading a larger resource and you'd like to provide progress feedback to the user while the transfer is taking place, use of a delegate would also be required. `URLSession` can be used to store the contents of the fetched resource to disk or to memory, depending on what method is invoked to initiate the fetch.

In essence an `URLSession` instance provides a context for a set of HTTP requests all of which share a common configuration. Examples of the types of features a session can define in its configuration could include setting the maximum number of simultaneous requests, or constrain whether or not the cellular data service can be used, establish timeout values or even add additional HTTP headers. There are essentially four types of sessions we can configure.

1. `shared`: a singleton session manged by iOS that is a good choice when you have a very simple HTTP fetch and no special configuration considerations.

2. `defaultSessionConfiguration`: a default session that uses a disk-based global cache. Default sessions behave like the shared session but if we implement delegation it can inform us of transfer status.

3. `ephemeralSessionConfiguration`: Similar to default sessions, but cache in memory instead of disk.

4. `backgroundSessionConfiguration`: Sessions that can upload or download in the background, and can continue on even if the app is suspended or terminated.

Once we have a session object, we can use it to create one or more tasks. There are three flavors of tasks that can be created within a session:

1. `URLSessionDataTask`: capable of issuing a HTTP GET request and storing the response in memory. This type of tasks is commonly used for requests with small response payloads when implementing user interfaces.

2. `URLSessionUploadTask`: capable of issuing a HTTP PUT or POST request. This type of task is used when we want to upload a file.

3. `URLSessionDownloadTask`: used to download a file to a temporary location on disk.

Creating a task without a delegate means we must pass a completion handler in the form of a Swift closure. The closure will be executed when the task complete, or terminated due to error. Since HTTP fetches can take time, it is important that the main thread (which often issues HTTP requests) is not blocked. Hence, the request can be issued immediately from the main thread which returns immediately. Later when the fetch completes, another thread can execute the closure we provide.

Swift

```swift
var urlSession = URLSession.shared
let url = URL(string: "http://example.com/some/api/call")
let task = urlSession.dataTask(with: url!) {
     (data, response, error) in
     if let e = error {
       // handle error here
     } else if let r = response {
       // handle response here.
  }
}
```

The closure receives an object of type `Data` holding the payload, an `URLResponse` instance that encapsulates response metadata, and an `Error` instance with more information in the face on an error. The error instance will be `nil` if the request succeeded.

Optionally, if our app is issuing an `URLSessionDownloadTask`, we could provide progress to the user by implementing the `URLSessionDownloadDelegate` instead. This protocol defines three methods — one that get's called when a paused download is resumed, one that is called periodically to report download progress, and a third method that is called when the download task completes.

If our app is issuing a `URLSessionDataTask` or an `URLSessionUploadTask` we can optionally implement the `URLSessionDataDelegate` protocol. In addition to telling us when the

task complete, the `URLSessionDataDelegate` can be used to get involved in authentication, redirect handling, as well as periodic progress reports of data sent. This latter feature is of course important if the user interface is providing progress reports on an upload.

We've only scratched the surface here in terms of the features of Apple's URL Loading System. Apple has done a particularly good job of documenting URLSession and the related Foundation classes. We'd highly encourage you to read your way through Apple's *URL Session Programming Guide* to take advantage of features that are required by your app's use case.

11.2.2 Processing JSON Representations in iOS

After issuing an HTTP request via an URLSession task, the hope is that you'll eventually get a response message with an interesting payload. The representation used for the data in the payload will necessarily be JSON. The Foundation framework provides the `JSONSerialization` class which lets us take a JSON formatted string and parse it into a data structure consisting of `Dictionary`, `Array`, `Number`, `Strings`, and `Bools`.

Consider the following JSON content:

```
[
  {
    "_id": "585c1b1f56d9490c5abeb774",
    "image": "http://placehold.it/32x32",
    "age": 21,
    "eyeColor": "green",
    "name": "Carter Blankenship",
    "gender": "male",
    "company": "GEOSTELE",
    "latitude": -86.246973,
    "longitude": -8.062469
  },
  {
    "_id": "585c1b1f5c1af5b41ad1cc1e",
    "image": "http://placehold.it/32x32",
    "age": 28,
    "eyeColor": "brown",
    "name": "Suarez Goff",
    "gender": "male",
    "company": "CORMORAN",
    "latitude": -40.083622,
    "longitude": 95.430979
  },
  {
    "_id": "585c1b1f8fcf230fc6f9ac5a",
    "image": "http://placehold.it/32x32",
    "age": 27,
    "eyeColor": "green",
    "name": "Sharlene Bradford",
    "gender": "female",
    "company": "PHORMULA",
    "latitude": -25.057562,
    "longitude": 12.469884
  }
]
```

Assuming we have a variable of type `String` named `jsonStr` that contains this JSON, we can proceed to parse it using `JSONSerialization` without much difficulty as shown in Listing 130.

Swift

```
var jsonData : Data!  = jsonStr.data(using: .ascii)

var customers: Array<Any>!
do {
  customers = try JSONSerialization.jsonObject(with: jsonData, options: .allowFragments)
  as? Array<Any>
  for c in customers {
    guard let customer = c as? Dictionary<String,Any>,
    let image = customer["image"] as? String,
    let name = customer["name"] as? String,
    let lat = customer["latitude"] as? Double,
    let age = customer["age"] as? Int else {
      continue
    }
    print("\\(name), \\(lat), \\(age), \\(image)")
  }
} catch {
  print(error)
}
```

Listing 130: Example of how to process a simple JSON string using the JSONSerialization class.

11.2.3 Extending Traxy with Weather Data From the Open Weather Web API

Having looked at some of the essential concepts of networking in iOS, let's go ahead and integrate the Open Weather Web API into Traxy. If you consult the Open Weather Web API documentation online, you'll notice a variety of different URL patterns that can be used to fetch weather conditions. We will simply capture the current weather each time a new journal entry is created, based on the latitude and longitude. For now, we will assume that the default location of each journal entry will be set to the geo-coordinates retrieved from the Google Place Picker in Chapter 7 and stored on our containing `Journal` instance. Later in Chapter 13 we will revisit this and allow the user to edit the location of each journal entry. We will use the default geo coordinates to fetch current weather conditions at that location, and will store the retrieved information in Firebase, and display it with the journal entry.

The Open Weather Web API endpoint we will use is in the following pattern:

```
https://api.openweathermap.org/data/2.5/weather?units=imperial&lat=xxx&lon=yyy&appid=zzz
```

Note that the `units=imperial` gives us temperature in Fahrenheit. Furthermore, `xxx`, `yyy`, and `zzz` need to be replaced with the latitude, longitude, and API key, respectively. The JSON response for this call will look as shown in 131. From this we will extract the weather icon name and temperature as marked in the listing with the "extract this" comments.

```
{
  "coord":{
    "lon":-79.5439,
    "lat":43.8676
  },
  "weather":[
  {
    "id":802,
    "main":"Clouds",
    "description":"scattered clouds",
    "icon":"03d"          // <-- extract this
  }
  ],
  "base":"stations",
  "main":{
    "temp":33.01,         // <-- extract this
    "feels_like":24.17,
    "temp_min":32,
    "temp_max":34,
    "pressure":1021,
    "humidity":76
  },
  // more data irrelevant to our needs omitted
}
```

Listing 131: Typical JSON representation returned by the Open Weather Web API.

Integrating the Open Weather Web API

The Open Weather Web API is relatively simple, so your implementation can get the job done by simply using the shared `URLSession` singleton. Begin by defining a protocol named `WeatherService` to firewall the details of the specific web service we are integrating with. Recall that this is a good practice to make it easier to switch over to a different weather API in the future with minimal impact on our code. The WeatherService and its companion data structure Weather is shown in Listing 132.

Applying what you've learned earlier, go ahead and implement this protocol in a separate class named `OpenWeatherService` as shown in Listing 133.

Hopefully you don't have much difficulty in reading the code and seeing how `URLSession` is used to create the data task, and then `JSONSerialization` is used to process the resulting JSON. If you haven't already, you'll want to go over to the Open Weather website, signup for a free account and generate an API key. You will need to assign the value of your key to the constant `OPEN_WEATHER_API` to get the code to actually run!

Our next step is to wire up the code to fetch to the API and persist the results in our journal entries so it can be displayed in the future. Recall that previously we introduced a scene with controller `JournalEntryConfirmationViewController` that we transition to each time we add a new journal entry. At the very end of this controller's `viewDidLoad` method we can conditional logic to fetch the weather conditions whenever a new entry is added, as shown in Listing 134.

On the first time a journal entry is added, `self.entry` will be `nil` hence the else branch of the conditional will be executed. After creating a new entry instance to hold the new entry, we check to see if a latitude/longitude has been associated with the containing joural and if so, we

Swift

```swift
struct Weather {
  var iconName : String
  var temperature : Double

  init(iconName: String, temperature: Double) {
    self.iconName = iconName
    self.temperature = temperature
  }
}

protocol WeatherService {
  func getWeather(date: Date, forLocation location: (Double, Double),
    completion: @escaping (Weather?) -> Void)
}
```

Listing 132: An abstract weather service interface, to hide the specifics of the Open Weather Web API from our code.

will fetch the weather conditions for that location using the code discuss previously.

Note that our `getWeatherForLocation` method receives a closure expression. The closure contains code we want to execute when the fetch completes. In this case, we want to store the temperature and weather icon name in our journal entry. There are a couple of important things to be aware of with regard to this closure. First of all, we have a potential race condition that needs to be resolved. When the HTTP fetch is initiated, it could take multiple seconds to resolve, which means if a user hits the save button on this screen before the fetch returns, the entry will get saved without the weather information. We resolve this by temporarily disabling the save button right before initiating the fetch, and then re-enabling the button once the HTTP fetch completes.

A second potential issue to be aware of here is that while the `viewDidLoad` method is called by the iOS main user interface thread, we cannot assume the closure expression will be executed by that same thread. This is an important observation because on the iOS platform, our user interface data structures can only be modified by the main user interface thread. Since we are modifying the state of the bar button on the right side (e.g. the save button) we have to make sure that this code in our closure is executed by the main thread. That is the reason we have the call to `DisplayQueue.main.async` passing it the code we want to have executed on the main thread.

Now that we've fetched the weather data, we need to make sure we can persist and retrieve this information to/from Firebase. First, we update our `JournalEntry` struct by adding members `temperature` and `weatherIcon`, and making them optional parameters on the initializer as shown in Listing 135.

This change impacts code in the `TraxyRepository` in a couple of places: when writing new journal entries to the Firebase Cloudstore and when reading entries from the Cloudstore. The former can be facilitated by adding conditional logic to the `toDictionary` helper method as shown in Listing 136.

Similarly, any weather data that is written to the Cloudstore will also be come available when reading data, hence the `registerForFireBaseUpdates` method also needs to be updated. We basically need to insert a couple of lines of code to extract the temperature and weather icon name from the document snapshot and pass it to the journal entry initializer, as shown in Listing 137,

```swift
let sharedOpenWeatherInstance = OpenWeatherService()

class OpenWeatherService: WeatherService {

  let API_BASE = "https://api.openweathermap.org/data/2.5/weather"
  var urlSession = URLSession.shared

  class func getInstance() -> OpenWeatherService {
    return sharedOpenWeatherInstance
  }

  func getWeather(forLocation location: (Double, Double),
  completion: @escaping (Weather?) -> Void)
  {
    let urlStr = API_BASE +
     "?units=imperial&lat=\(location.0)&lon=\(location.1)&appid=\(OPEN_WEATHER_API_KEY)"

    let url = URL(string: urlStr)

    let task = self.urlSession.dataTask(with: url!) {
      (data, response, error) in
      if let error = error {
        print(error.localizedDescription)
      } else if let _ = response {
        let parsedObj : Dictionary<String,AnyObject>!
        do {
          parsedObj = try JSONSerialization.jsonObject(with: data!, options:
          .allowFragments) as? Dictionary<String,AnyObject>

          guard let forecast = parsedObj["weather"],
          let main = parsedObj["main"],
          let details = forecast[0] as? Dictionary<String, AnyObject>,
          let iconName = details["icon"] as? String,
          let temperature = main["temp"] as? Double
          else {
            completion(nil)
            return
          }

          let weather = Weather(iconName: iconName, temperature: temperature)
          completion(weather)

        }  catch {
          completion(nil)
        }
      }
    }

    task.resume()
  }
}
```

Listing 133: Implementation of a Open Weather Weather Service singleton.

Swift

```
if let e = self.entry {
  caption = e.caption!
  date = e.date!
} else {
  self.entry = JournalEntry(key: nil, type: self.type, caption: caption,
  url: "", thumbnailUrl: "", date: date, lat: self.journal.lat, lng: self.journal.lng)
  // if there is a default location for entry, fetch the weather.
  if let lat = self.journal.lat, let lng = self.journal.lng {
    self.navigationItem.rightBarButtonItem?.isEnabled = false
    self.entry?.lat = lat
    self.entry?.lng = lng
    let weatherService = OpenWeatherService.getInstance()
    weatherService.getWeather(forLocation: (lat, lng)) { (weather) in
      if let w = weather {
        self.entry?.temperature = w.temperature
        self.entry?.weatherIcon = w.iconName
      }
      DispatchQueue.main.async() {
        self.navigationItem.rightBarButtonItem?.isEnabled = true
      }
    }
  }
}
```

Listing 134: Updating the `viewDidLoad` method of `JournalEntryConfirmationViewController` to conditionally fetch weather conditions.

Swift

```swift
struct JournalEntry {
  var key : String?
  var type: EntryType?
  var caption : String?
  var url : String
  var thumbnailUrl : String
  var date : Date?
  var lat : Double?
  var lng : Double?
  var temperature: Double?
  var weatherIcon: String?

  init(key: String?, type: EntryType?, caption: String?, url: String,
    thumbnailUrl: String, date: Date?, lat: Double?, lng: Double?,
    temperature: Double? = nil, weatherIcon: String? = nil)
  {
    self.key = key
    self.type = type
    self.caption = caption

    self.date = date
    self.lat = lat
    self.lng = lng
    self.url = url
    self.thumbnailUrl = thumbnailUrl
    self.temperature = temperature
    self.weatherIcon = weatherIcon
  }
}
```

Listing 135: The `JournalEntry` struct is updated to include the weather information.

```swift
fileprivate func toDictionary(vals: JournalEntry) -> [String:Any] {
  var retval =  [
    "caption": vals.caption! as NSString,
    "lat": vals.lat! as NSNumber,
    "lng": vals.lng! as NSNumber,
    "date" : NSString(string: (vals.date?.iso8601)!) ,
    "type" : NSNumber(value: vals.type!.rawValue),
    "url" : vals.url as NSString,
    "thumbnailUrl" : vals.thumbnailUrl as NSString
  ]
  if let icon = vals.weatherIcon, let temp = vals.temperature {
    retval["weatherIcon"] = icon as NSString
    retval["temperature"] = temp as NSNumber
  }
  return retval
}
```

Listing 136: The `toDictionary` helper method of `TraxyRepository` updated with conditional logic to handle any weather data associated with the entry.

lines 28–33. Note that if the information is not available in Cloudstore, the values will just be nil which is fine since the new members are defined as optionals.

Now that we're able to fetch the weather for each journal and persist / retrieve it from the Firebase Cloudstore, our only remaining task is to update the view of the `JournalTableViewController` to display the information.

Displaying the weather information

We will display the temperature and weather icon associated with each journal entry immediately right of the date string, as shown in Figure 11.2. By now we will assume you are familiar enough with Auto Layout to go into the storyboard and add a `UILabel` for the temperature, and a `UIImage` to display the weather icon, as shown in the figure. In additional to the storyboard updates, you'll also need to define new outlets named `weatherImage` and `temperature` in the class `JournalEntryTableViewCell`. In addition, we need to update the `setValues` helper method of the class to bind the weather data (if available) to our new user interface components. You can view these updates in Listing 138.

You'll notice that we've used the Kingfisher pod that we installed in an earlier chapter to fetch the weather icons. This is accomplished by simply passing the appropriate url as computed by the icon name, and passing it to the `kf.setImage` method provided to us by the Kingfisher pod. The Kingfisher pod will handle caching for us under the hood which greatly simplifies our job. For the weather data fetched from the API, iOS's URL Loading System will cache per protocol policy. So in the case of HTTP it is the headers returned by the Open Weather web server that will dictate how the results will be cached. If you access the web API directly (take a look at the Postman plugin for Chrome) you can readily determine from the HTTP headers on the response message that the results will be cached for one hour.

Swift

```
func listenForJournalEntryUpdates(journalKey: String,
  onChange: @escaping ([JournalEntry]) -> Void)
{
  self.listener =
  self.ref?.collection("journals").document(journalKey)
  .collection("entries").addSnapshotListener({(snapshot,
   error) in
    guard let documents = snapshot?.documents else {
      return
    }
    var entries = [JournalEntry]()
    for entry in documents {
      let key = entry.documentID
      let caption : String? = entry["caption"] as! String?
      var url : String? = entry["url"] as? String
      if url == nil {
        url = ""
      }
      var thumbnailUrl : String? = entry["thumbnailUrl"] as? String
      if thumbnailUrl == nil {
        thumbnailUrl = ""
      }
      let dateStr  = entry["date"] as! String?
      let lat = entry["lat"] as! Double?
      let lng = entry["lng"] as! Double?
      let typeRaw = entry["type"] as! Int?
      let type = EntryType(rawValue: typeRaw!)

      let weatherIcon : String? = entry["weatherIcon"] as? String
      let temperature : Double? = entry["temperature"] as? Double

      entries.append(JournalEntry(key: key, type: type, caption: caption,
        url:url!, thumbnailUrl: thumbnailUrl!, date: dateStr?.dateFromISO8601,
        lat: lat, lng: lng, temperature: temperature, weatherIcon: weatherIcon))
    }
    onChange(entries)
  })
}
```

Listing 137: The `listenForJournalEntryUpdates` method of `TraxyRepository` updated with conditional logic to handle any weather data associated with the entry.

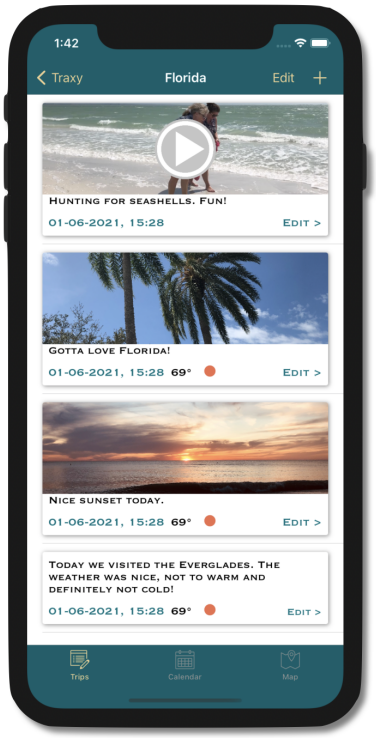


Figure 11.2: Displaying weather data for each journal entry.

Swift

```swift
import UIKit
import Kingfisher

class JournalEntryTableViewCell: UITableViewCell {

  @IBOutlet weak var containingView: UIView!
  @IBOutlet weak var playButton: UIButton!
  @IBOutlet weak var date: UILabel!
  @IBOutlet weak var textData : UILabel!
  @IBOutlet weak var imageButton: UIButton!
  @IBOutlet weak var thumbnailImage: UIImageView!
  @IBOutlet weak var temperature: UILabel!
  @IBOutlet weak var weatherImage: UIImageView!
  @IBOutlet weak var editButton: UIButton!
  var entry : JournalEntry?

  override func awakeFromNib() {
    super.awakeFromNib()

    if let button = self.imageButton {
      button.imageView?.contentMode = .scaleAspectFill
    }

    // round top corners of image, as the clip to bounds is set to false on containing
    // CardView, causing the corner to leak over the bounds without this
    workaround
    if let img = thumbnailImage {
      img.layer.cornerRadius = 3
      img.layer.maskedCorners = [.layerMinXMinYCorner, .layerMaxXMinYCorner]
    }
  }

  override func setSelected(_ selected: Bool, animated: Bool) {
    super.setSelected(selected, animated: animated)
    // Configure the view for the selected state
  }

  func setValues(entry : JournalEntry) {
    self.entry = entry
    self.textData.text = entry.caption
    self.date.text = entry.date?.shortWithTime
    if let temp = entry.temperature, let w = entry.weatherIcon {
      let tstr = String(format: "%.0f°", temp)
      self.temperature.text = tstr
      let iconUrl = "https://openweathermap.org/img/wn/\(w)@2x.png"
      self.weatherImage.kf.setImage(with: URL(string: iconUrl))
      self.temperature.isHidden = false
      self.weatherImage.isHidden = false
    } else {
      self.temperature.isHidden = true
      self.weatherImage.isHidden = true
    }
  }
}
```

Listing 138: Adding a couple of outlets for weather information to the `JournalEntryTableViewCell` class. Note the helper function `setValues` is also updated with some conditional binding logic.

11.3 Networking on Android

A resourceful developer knows how to tackle a problem using a variety of tools and picks one that delivers the best performance. To prepare you to be such a developer, this section presents several techniques for developing Android apps that communicate with web services. You will tackle this task using a Kotlin-based third party library OkHttp. The following dependency is required to use this library.

Gradle
```
implementation "com.squareup.okhttp3:okhttp:4.9.0"
```

The last digit "3" in the package name `OkHttp3` and the version number 4.x may raise some confusion for the readers, some may think that this manuscript has a typographic error. The package documentation, which can bn found at its GitHub repo explains that version 4.x is actually a Kotlin rewrite of version 3.12 which was originally written in Java. Moreover, the latest version maintains binary and source compatibility for fot Java and Kotlin source files.

11.3.1 HTTP Requests Using OkHttp3

The necessary code required for sending HTTP requests is relatively short. It can be expressed in only a few lines:

Kotlin
```
val httpClient = OkHttpClient()   // ONE TIME initialization

// The following lines will be called on every HTTP request
fun getDataFromURL(url: String): String? {
  val request = Request.Builder()
    .url(url).build()
  httpClient.newCall(request).execute().run {
    if (!isSuccessful)
      return null
    else
      return body()!!.string()
  }
}
```

However, sending an HTTP request and obtaining its result is a *blocking operation* that prevents your program from doing any other tasks until the connection to the remote server is established and result is returned.

In Section 8.3.5 you learned how to use Kotlin coroutines to delegate tasks to a background thread. You can reapply the same techniques for running the above HTTP requests and transform the above function into a Kotlin `suspend` function dispatched on the I/O thread:

Kotlin
```
private suspend fun getDataFromURL(url: String): String? = withContext(Dispatchers.IO) {
  val request = Request.Builder()
    .url(url).build()
  httpClient.newCall(request).execute().run {
    if (!isSuccessful)
      return@withContext null
    return@withContext body()!!.string()
  }
  return@withContext null
}
```

Following the code organization adopted in earlier chapters, the above function should also be written in `TraxyRepository.kt`. Specifically, it works in conjunction with another suspend function below:

Kotlin

```
suspend fun getWeatherData(lat: Double, lng: Double):Pair<Double,String>? {
  val tmp = getDataFromURL("https://api.openweathermap.org/data/2.5/weather?" +
      "lat=$lat&lon=$lng&appid=${BuildConfig.OWM_API_KEY}&units=imperial")
  tmp?.let {
    with(JSONObject(it)) {
      val icon = getJSONArray("weather")
        .getJSONObject(0)
        .getString("icon");

      val temp = getJSONObject("main").getDouble("temp")
      return Pair(temp, icon)
    }
  }
  return null
}
```

By default, OpenWeatherMap returns the temperature in Kelvin. To get the temperature in Fahrenheit, we add `units=imperial` to the query parameters.

The function `getWeatherData` will be called when a new journal media entry is about to be saved to the Cloudstore in `MediaDetailsFragment`. The FAB click listener in that fragment should include the following snippet for initiating the function call to store a new journal media:

Kotlin

```
val mediaObj = JournalMedia(
  caption = mediaModel.mediaCaption.value ?: "None",
  date = mediaModel.mediaDate.value?.toString() ?: "None",
  type = mType.ordinal,
  lat = mediaModel.selectedJournal.value?.lat ?: 0.0,
  lng = mediaModel.selectedJournal.value?.lng ?: 0.0)
mediaModel.addMediaEntry(mediaObj, mediaUri!!)
```

Weather data should be fetched prior to calling `addMediaEntry`, so the updated code should look like the following:

Kotlin

```
val mediaObj = JournalMedia(
  caption = mediaModel.mediaCaption.value ?: "None",
  date = mediaModel.mediaDate.value?.toString() ?: "None",
  type = mType.ordinal,
  lat = mediaModel.selectedJournal.value?.lat ?: 0.0,
  lng = mediaModel.selectedJournal.value?.lng ?: 0.0)

/* New code snippet below */
if (mediaObj.lat != 0.0 && mediaObj.lng != 0.0) {
  val weatherData = TraxyRepository.getWeatherData(mediaObj.lat, mediaObj.lng)
  mediaObj.temperature = weatherData?.first
  mediaObj.weatherIcon = weatherData?.second
}

mediaModel.addMediaEntry(mediaObj, mediaUri!!)
```

To store weather related data in `JournalMedia`, the class must be updated by adding two new properties:

Kotlin
```kotlin
data class JournalMedia(
  @DocumentId
  val _key: String = "",

  var caption: String = "",
  var date: String = "",
  var lat: Double = 0.0,
  var lng: Double = 0.0,
  var type: Int = 0,
  var url: String = "",
  var thumbnailUrl: String = "",
  var temperature: Double? = null,  // new property
  var weatherIcon: String? = null   // new property
)
```

Since the current sample data in your Cloud Firestore may not include weather data yet, we should make the last two properties optional. These changes affect the two functions in `TraxyRepository` that deal with saving and updating journal media entries.

Kotlin
```kotlin
fun addMediaEntry(journalKey: String, m: JournalMedia) {
  docRef?.let {
    var mediaData = hashMapOf(
      "type" to m.type,
      "caption" to m.caption,
      "date" to m.date,
      "url" to m.url,
      "lat" to m.lat,
      "lng" to m.lng,
      "temperature" to m.temperature,   // add this line
      "weatherIcon" to m.weatherIcon    // add this line
    )
    // More code not shown
  }
}
```

A similar update also applies to `updateMediaEntry`

Kotlin

```
suspend fun updateMediaEntry(journalKey: String, m: JournalMedia): Void? {
  return docRef?.let {
    var mediaData = hashMapOf(
      "type" to m.type,
      "caption" to m.caption,
      "date" to m.date,
      "url" to m.url,
      "lat" to m.lat,
      "lng" to m.lng
    )
    if (m.temperature != null)              // add this line
      mediaData.put("temperature", m.temperature!!)   // add this line
    if (m.weatherIcon != null)              // add this line
      mediaData.put("weatherIcon", m.weatherIcon!!) // add this line
    // More code not shown
  }
}
```

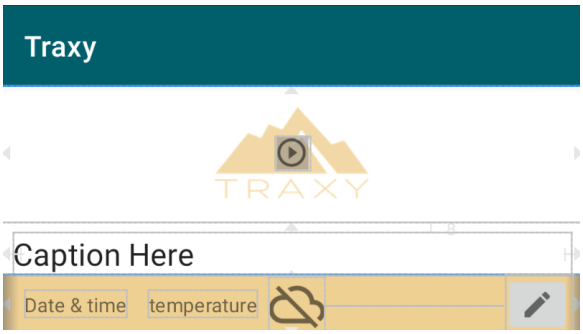


Figure 11.3: Journal Media Updated UI Layout `media_item.xml`

The updated UI design to show the weather icon and temperature is shown in Figure 11.3. The new design calls for the following minor changes across several files:

- `media_item.xml`: add a TextView (`temperature`) to show the temperature and an ImageView (`weather_icon`) next to the media timestamp

- `JournalMediaViewHolder`: parallel to the updates applied to the XML file above, the view holder must also define the same two fields above.

Kotlin
```
class JournalMediaViewHolder(v: View) : RecyclerView.ViewHolder(v) {
  // other fields definition go here
  val weather_temp: TextView
  val weather_icon: ImageView
  val decimalFormat = DecimalFormat("#.##F")

  init {
    // other fields initialization go here
    weather_temp = v.findViewById(R.id.temperature)
    weather_icon = v.findViewById(R.id.weather_icon)
  }
}
```

In addition, its `bindTo` method should also bound the two new fields to the UI components

Kotlin
```
public fun bindTo(m: JournalMedia, listener: ((JournalMedia, String) -> Unit)?) {
  // code for binding other fields go here
  if (m.temperature != null) {
    weather_temp.visibility = View.VISIBLE
    weather_temp.text = decimalFormat.format(m.temperature)
  } else
    weather_temp.visibility = View.GONE
  if (m.weatherIcon != null) {
    // Require HTTPS
    val iconUrl = "https://openweathermap.org/img/wn/${m.weatherIcon}@2x.png"
    Glide.with(view).load(iconUrl).into(weather_icon)
    weather_icon.visibility = View.VISIBLE
  } else
    weather_icon.visibility = View.GONE
}
```

11.3.2 HTTP Requests Using Retrofit

After mastering the networking basics using the OkHttp library, let's implement the same task using a 'high-level" library, Retrofit. This library takes a significantly different approach, it allows user apps to use *own methods* for sending HTTP requests. Using Java annotations, Retrofit magically wires up your functions to the underlying network infrastructure on your Android device. With Retrofit assistance, sending HTTP requests to remote servers feels like making local method calls.

Retrofit is an excellent open-source library released by the team at Square Inc., the company that makes mini card readers for mobile payment. The main strength of this library is its capability to abstract HTTP requests into your own Kotlin functions. The library handles many low-level details of network input/output, buffering, caching, secure connection, etc. In fact, when OkHttp is installed, Retrofit will use it automatically.

Be sure to include Retrofit dependencies in your `build.gradle`.

Gradle
```
// in build.gradle
implementation 'com.squareup.retrofit2:retrofit:2.6.0'
implementation 'com.squareup.retrofit2:converter-gson:2.5.0'
```

The recipe for using Retrofit in your app calls for the following steps:

1. Declares an *annotated* interface that describes methods for invoking the web services.
2. Define the converter class(es) required by the methods defined in the above interface
3. Build the service object and attached data converters
4. Invoke the methods from the service object, the library manages the background threads on your behalf

Let's first define an interface for sending HTTP requests to OpenWeatherMap web services, assuming the following target URL as a starting example:

```
1 https://api.openweathermap.org/data/2.5/weather?lat=xxx&lon=yyy&appid=zzz
```

In general, a typical URL can be broken down into the following components:

- **Scheme**: `http:`
- **Host**: `api.openweathermap.org`
- **Path**: `/data/2.5/weather`
- **Query Parameters**: `lat=xxx&lon=yyy&appid=zzz` (essentially the rest of the URL after the question mark '?')

HTTP requests sent from Traxy will vary in the last two values (latitude and longitude) of the **query parameters** and your app should be allowed to supply these values at runtime. Therefore an appropriate interface definition for Traxy to connect to OpenWeatherMap would look include a method that takes two (or three) parameters.

> **NOTE**
>
> The `appid` is technically not a "parameter" of our function, but it must be included as an HTTP **query parameter**. We will begin our first attempt by including the OWM API key in the function calls. Later, we will incorporate an "interceptor" to inject the API key into the HTTP calls on the fly.

The interface calls for a new data class definition (`OWMWeather`) whose fields are property names in the JSON weather data. In the JSON output shown in Listing 140, we are interested only in the following fields:

- `description` and `icon` under `weather` array
- `temp` under `main` object

Please understand that your Kotlin data classes *are not required* to provide a matching attribute for every JSON key in the response, but only for those fields you are interested in. It is ok to ignore the other fields not of our concerns. Also, since `weather` is technically a JSON array, the corresponding data class declaration uses the `List` interface.

Once the interface is defined, you can then build a client object and start using it for fetching weather data.

Kotlin
```kotlin
// in webservice/OpenWeatherMapServices.kt
import retrofit2.http.GET
import retrofit2.http.Query

public interface OpenWeatherMapServices {
    @GET("data/2.5/weather/")
    suspend fun getWeatherAt (
      @Query("lat") latitude: Double,
      @Query("lon") longitude: Double,
      @Query("appid") apiKey: String) : OWMWeather
  }
```

Listing 139: Retrofit Interface for OpenWeatherMap APIs

1. Use Retrofit builder to create a client object from the interface. You need to do this step *only once*.

Kotlin
```kotlin
// Also in webservice/OpenWeatherMapServices.kt
object OpenWeatherMap {
  private const val BASE_URL = "https://api.openweathermap.org/"

  private fun getRetrofit(): Retrofit =
      Retrofit.Builder()
          .baseUrl(BASE_URL)
          .addConverterFactory(GsonConverterFactory.create())
          .build()

  val apiService = getRetrofit().create(OpenWeatherMapServices::class.java)
}
```

   Please note that the URL in the `baseUrl` call ends with a slash (/) while the path in the `@GET` annotation in Listing 139 specifies a *relative path* with no leading (/). This a recommended practice, and if `@GET()` refers to an *absolute path*, it will always *override* any additional path specified in base URL. The converter factory handles JSON to Kotlin object conversion using the classes provided in Listing 140.

2. Invoke the service from `TraxyRepository` replacing the current code of using the OkHttp library.

Kotlin
```kotlin
suspend fun getWeatherData(lat: Double, lng: Double):Pair<Double,String>? {
  return try {
    val w = OpenWeatherMap.apiService.getWeatherAt(lat,lng, BuildConfig.OWM_API_KEY)
    Pair(w.main.temp, w.weather.get(0).icon)
  }
  catch(e:Throwable) {
    null
  }
}
```

Run your app to verify that your weather data fetches are working as expected.

```
{
  "coord": { "lon": 139,"lat": 35},
  "weather": [
    {
      "id": 800,
      "main": "Clear",
      "description": "clear sky",
      "icon": "01n"
    }
  ],
  "base": "stations",
  "main": {
    "temp": 281.52,
    "feels_like": 278.99,
    "temp_min": 280.15,
    "temp_max": 283.71,
    "pressure": 1016,
    "humidity": 93
  },
  // other properties
  // irrelevant to our need
}
```

(a)

```
// in webservice/OWMWeather.kt
data class OWMWeather(
  val weather: List<WeatherDescription>,
  val main: WeatherData,
)

data class WeatherDescription(
  val description: String,
  val icon: String,
)

data class WeatherData(
    val temp: Double,
)
```

(b)

Listing 140: Mapping OpenWeatherMap JSON Response to Kotlin Data Classes

Incorporating API Key Into Query Parameters

The next improvement to our weather service is to inject the API key automatically on every HTTP request so we can remove the third parameter of `getWeatherAt` function:

Kotlin

```
  interface OpenWeatherMapServices {
    @GET("data/2.5/weather")
    suspend fun getWeatherAt(
        @Query("lat") lat: Double,
        @Query("lon") lon: Double,
//        @Query("appid") apiKey: String,
    ): OWMWeather
}
```

Be sure to edit the invocation of `getWeatherAt` in `TraxyRepository` to it passes only two arguments: latitude and longitude.

Retrofit2 handles HTTP requests using OkHttp and the latter provides a mechanism to intercept HTTP requests. In the context of the OpenWeatherMap API, `appid` is a query parameter for passing the API key.

To enable this feature, we have to use a custom HTTP client (instead of the default one used by Retrofit). The improved `OpenWeatherMap` class is shown below:

Kotlin

```kotlin
object OpenWeatherMap {
  private const val BASE_URL = "https://api.openweathermap.org/"

  val apiKeyInjector = Interceptor {
    it.request().run {
      val newUrl = url.newBuilder()
          .addQueryParameter("appid", BuildConfig.OWM_API_KEY)
          .addQueryParameter("units", "imperial")
          .build()
      val newRequest = newBuilder().url(newUrl).build()
      it.proceed(newRequest)
    }
  }

  val httpClient = OkHttpClient().newBuilder().addInterceptor(apiKeyInjector).build()

  private fun getRetrofit(): Retrofit =
    Retrofit.Builder()
        .baseUrl(BASE_URL)
        .client(httpClient) // use a custom HTTP client
        .addConverterFactory(GsonConverterFactory.create())
        .build()

  val apiService = getRetrofit().create(OpenWeatherMapServices::class.java)
}
```

Congratulations! You now have more tools in your belt and you are now more capable in handling network-related tasks and able pick the most appropriate tool for the tasks.

— In many cases, the user interface to a program is the most important part for a commercial company: whether the programs works correctly or not seems to be secondary.

Linus Torvalds

12 User Interface Revisited

Chapter Objectives

After reading this chapter you should be able to

- Know when and how to create applications with tabbed navigation.
- Understand how to layout views in a scrollable two dimensional grid-like manner
- Use Android Fragment Manager to add fragments programatically

12.1 Navigating Your App's User Interface

So far our user interface exploration has involved stacks of scenes (navigation stacks on iOS and activity stacks on Android) in conjunction with scrollable list of items. There are other more powerful navigation schemes that can be combined with these elements on both the Android and iOS platform. One such scheme is that of *tabbed applications*. In addition, both Android and iOS allow us to display collections of elements in a two dimensional grid as well, in a way that is very similar to how they display a vertical scrollable list of elements.

12.1.1 Tabbed Applications

Thus far, we've learned how to manage a stack of view controllers via the `UINavigationController` in iOS and a stack of activities via Android's activity stack. This concept works very well, particularly in a situation where the user is presented with a multiplicity of items and has the option to drill down into more detail on any individual item by tapping on it.

Human factors experts have studied human computer interfaces and there is no shortage of research studies that demonstrate that the more interactions required to complete a particular task, the less likely it is users will complete it. If this is true on traditional desktop computer interfaces, it is even more so on mobile devices, where users are often using the device while on the go, situationally constrained and often multitasking. In the stack based navigation approach, a screen is a single tap away from its successor or predecessor, but may be a number of taps (pushes or pops) away from any arbitrary scene that is neither a successor or a predecessor. Its not unusual for a single application to support a number of different tasks or use cases. For example, the music app on your iPhone lets you browse and play music from your personal library, view your favorites, browse new music available for purchase, and play curated music streams Clearly, organizing these different tasks into a single stack of controllers is not going to work very well.

A common approach on both iOS and Android is to arrange each top level task or use case into a single tab, and then reserve a small portion of the screen to display a row of buttons — one per tab. Both the Apple Music and Books apps on iOS use this approach as can be scene in Figures 12.1a and 12.1b. At any given time, the user can see all (or at least the important) supported use cases at any time. This is particularly helpful in that the user can immediately get an idea of what the app's possibilities might be without having to go looking for them.

There are other ways to organize multi-task / use case mobile apps as well. Arising in part out of popular responsive Javascript web frontend frameworks are the *hamburger menus* (also known as *side menus* or *side navigation*) which many native apps use. The Twitter app on iOS, as shown in Figure 12.1c is a good example of this approach. The rationale for hamburger menus is that permanently reserving a chunk of screen real estate for a row of tab buttons is a waste of space and doesn't scale well. Instead, a small button is displayed (the icon is typically a stack of three horizontal lines, hence the name "hamburger" menu), often in the left or right corners of the screen. When tapped, a menu of options slides out from the side, partially covering the current view. The menu is dismissed as soon as an option is chosen.

When highly popular native apps such as Facebook and Twitter adopted the hamburger approach several years back, many app developers began to use this technique. Android has native support (navigation drawers), and there are no shortage of third party solutions for iOS. However, in recent years conventional wisdom suggests that hamburger menus should be avoided. The data suggests that due to the consequential *out of sight out of mind* factor of this approach can reduce user engagement in an app [19].

As to the merit of it being more scalable than tabs, the recent trend has been towards unbundling larger apps into a suite of smaller apps that do at most 2–3 tasks / use cases really well. Take for example, Facebook's mobile apps. Originally it was one large monolithic app in which you did everything. Today they have a suite of simpler more focused apps (e.g. Messenger for messaging, Business Suite for managing Facebook pages, Instagram for photo sharing, etc.).

For these reasons, more and more apps that have experimented with hamburger menus in the past have moved back to the tabbed approach that actually predates the hamburger menus. If the app has more than 4–5 high level use cases, some of its functionality will be split off into smaller focused companion apps. At the time of writing, the Twitter app is actually one notable hold out, as is Google with many of its productivity apps (Docs, Sheets, etc.) utilizing hamburger menus.

In reality, even when our top level navigation paradigm is tabbed, within the tabs we may very well be using other navigational paradigms. For example, it is common for each tab in a tabbed app manage a stack of different scenes.

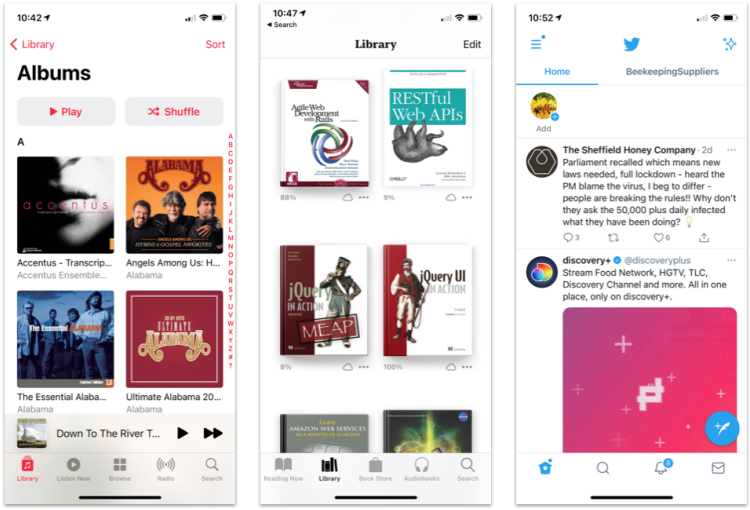


Figure 12.1: (a) The Apple Music app on iOS uses tabs to separate the key top level use cases. (b) The Apples Books app displaying book cover thumbnails in a scrollable grid-like and also uses tabs. (c) The Twitter iOS app uses tabs for its main navigation paradigm, but also employs a "hamburger" menu in the upper left which slides out an additional menu to let users tweak their profile, bookmarks, lists, and more.

12.1.2 Grid Layouts

In Chapter 6 we looked at how to display scrollable collections of items. On both platforms the implementation consisted of a vertical stack of rows, where each row consisted of a single cell or view — in essence an $N \times 1$ matrix. Often we may want to display a collection of items in a scrollable (either vertically or horizontally) $M \times N$ grid-like layout. For example, the "Library" tab of Apple's Music app on iOS as well as the "My Books" tab of the iBooks app on iOS present item thumbnails in a grid-like fashion that is scrollable on the vertical axis, as shown in Figures 12.1a and 12.1b. Given the frequency at which this occurs, both the iOS and Android SDKs provide support for these types of layouts. Give the resource constraints, as you might expect these are implemented in a manner very similar to what we've already learned. At the heart of the solution is a way to efficiently recycle the predictably homogeneous cells when they go off screen.

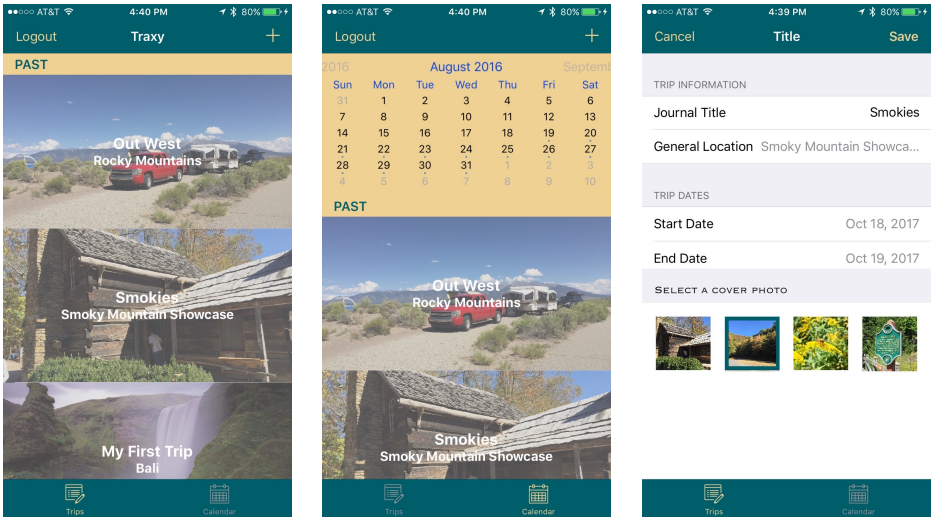
12.1.3 Extending the Traxy User Interface

We're going to extend our Traxy app. First, we'll move to a tabbed approach as our top level navigation paradigm. The first tab will contain our current main screen showing a list of all the journals that have been created to-date. We'll add a second tab named "Calendar" that will display a scrollable calendar and filter and display underneath it only the trips that have entries for the month the calendar is currently displaying. As the user adds more journals overtime, the second tab will make it more convenient to browse the set of journals by date.

Secondly, now that we're able to enter photos within our journals, we should be able to desig-

nate one of the photos as our cover photo, and hence eliminate the ugly place holder image we've been using there up until now. We're going to do this by displaying the thumbnails of all photos captured within a journal under the form on the journal editor.

While we're inevitably going to have to introduce some new code to accomplish this, we're also going to try to do as much code reuse as possible. For example, the layout and functionality of our list of filtered journals on the calendar screen is essentially the same as the list on the main screen, its just displaying a subset of the journals. We should be able to reuse this code as much as possible. The iOS version of our updated Traxy app is shown in Figure 12.2. The Android version of Traxy will be enhanced in a functionally equivalent manner. Let's first update the iOS version.



(a) (b) (c)

Figure 12.2: (a) Main tab showing all journals. (b) Calendar tab showing only those journals in the displayed month. (c) Enhancement to the journal edit view to select a cover photo.

12.2 Tabs and Collections on iOS

In this section, we will reorganize Traxy's top level navigation into tabs. The first tab will be our existing complete list of journals sectioned in to past, present, future. The second tab will be a new scene which displays a monthly calendar and a list of journals for that month on the bottom half of the screen. We will take advantage of a third party calendar pod and our existing code to implement this new tab. Finally, we will also explore how to use the `UICollectionView` to allow user's to select a default cover image for each of their journals.

12.2.1 Adding Tabs to Traxy

Moving your existing iOS implementation into a tabbed style app using the `UITabBarController` at this point is going to be a little bit more involved than starting from the outset of the app. Complicating things here is that per Apple's documentation, the `UITabBarController` MUST be the

root controller of your application. You should never make a `UITabBarController` the child of another view controller. You'll recall that the root view controller (also known as the root view controller) has been our `LoginViewController`. We need to introduce a tab view controller as the root, and have it check for a valid user session, and modally display the login view if a valid user session has not been established. This will require us to do a bit of rearrangement in our storyboard. So pay close attention and we'll walk you through what's needed.

Step 1: Introduce a new `UITabBarController` and embed the navigation stack containing the `MainViewController` within it. As we saw previously, the quick and easy way to accomplish this is to select the `UINavigationController` that contains our top level `MainViewController` and then from the application menu select Editor > Embed In > Tab Bar Controller. If you do this correctly you will notice that that our navigation controller now is contained by a tab bar controller as shown in Figure 12.3.

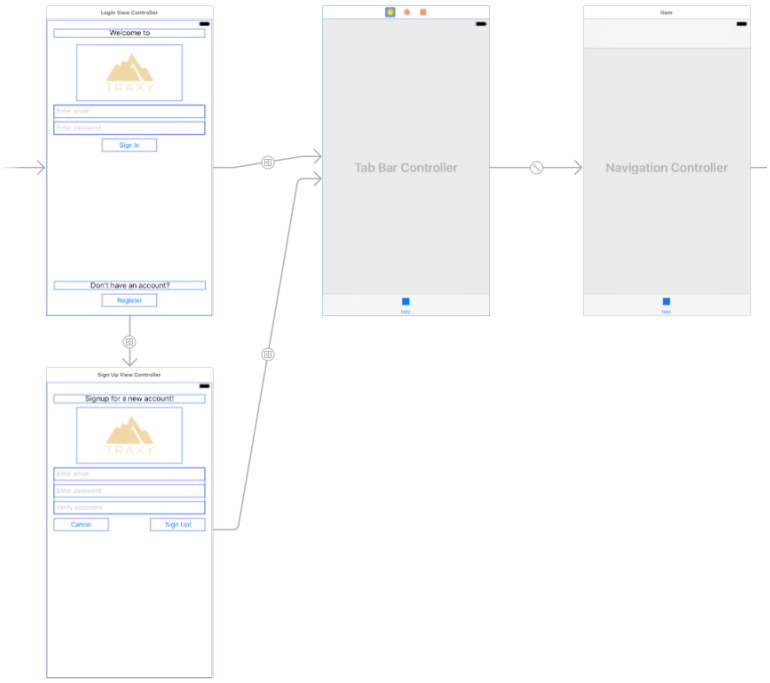


Figure 12.3: After placing the navigation controller into a tab bar controller.

Step 2: Make the `UITabBarController` the initial view controller, and add a modal segue to the login scene. Drag the initial view controller arrow pointing at the login view over to the tab bar controller. Next, delete the `segueToMain` and `segueToMainFromSignUp` segues emanating from the login and signup scenes, respectively. Since the tab bar controller will be presenting these scenes modally (e.g. on top of itself) you won't need these segues back to the tab bar controller. Instead, the login view controller will simply dismiss itself when it wants to return to the tab bar controller. The signup view controller has a bit more work to do in that it has to dismiss itself as well as the login view it covers. That can be accomplished with an unwind segue. Having deleted the two segues, select the tab bar controller scene and control click/drag over to the login scene to

add a new manual segue. Give this segue the identifier `presentLogin` and make sure it's kind option is set to Present Modally and the presentation option to Full Screen. You might want to drag the login and signup controllers around on the canvas to visually clarify that they are now subordinate to the tab bar controller. A helpful practice is to place the login / signup related scenes directly above the root controller. Upon completing this step the impacted controllers should be as shown in Figure 12.4.

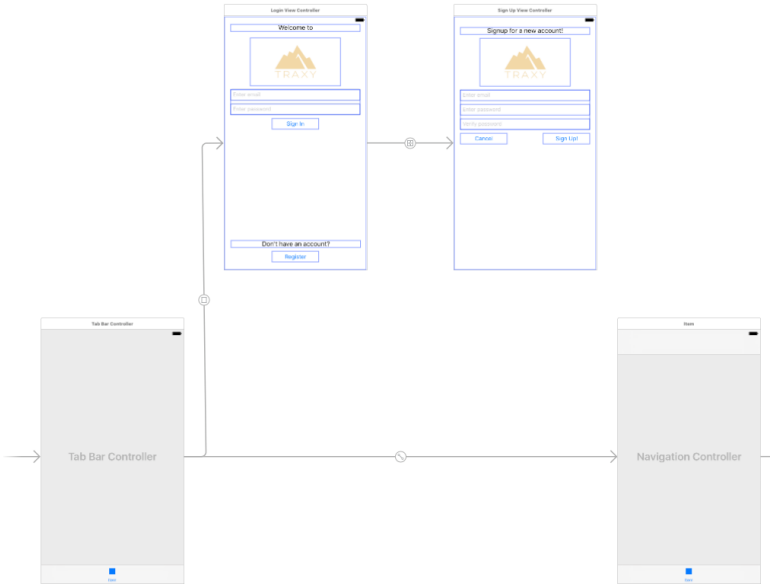


Figure 12.4: After making the tab bar controller the root, removing the unneeded segues, and adding the manual segue to the login scene from the tab bar scene.

Step 3: Next we need to add a new scene for the calendar tab and embed it in its own navigation controller. Drag a new view controller out into your storyboard, placing it under the scene of your main view controller. Embed the new view controller in a navigation controller by selecting it and then from the Xcode application menu selecting (Editor » Embed In » Navigation Controller). Next add a relationship segue from the tab bar controller to the navigation controller. This will set the new controller and its containing navigation controller as the second tab in your app. At this point go ahead and add a couple of image sets for your tabs to your project's Assets file . You can grab our images from the book's github repository, or you can create or steal your own. Once you've added the images to your project, you can set them along with the text for the tab bar buttons by clicking on the buttons displayed in the two navigation controller scenes and editing the text label (set to "Trips" and "Calendar" respectively) and setting the appropriate image asset. Upon completing this step the impacted controllers should be as shown in Figure 12.5.

You still have more changes to make to your storyboard, but this will suffice for now so let's look at the code needed to accommodate the changes made so far.

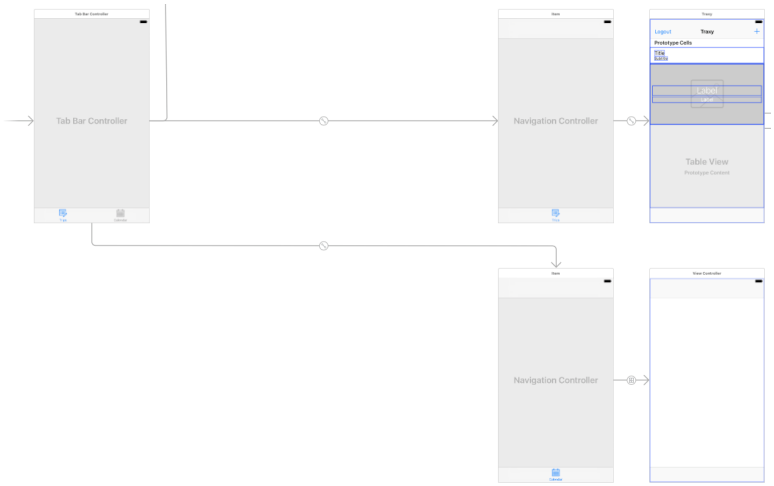


Figure 12.5: The storyboard after adding the calendar tab, adding button images and updating the labels.

12.2.2 Handling Authentication and Refactoring Out Common Code

One thing to note is that both tabs will display lists of journals. The first tab will display all journals and the second tab a subset depending on which month the calendar is displaying. Because of this, the code that your `MainViewController` previously implemented to retrieve journals from the Firebase database in the cloud can be shared by all of the tabs. Hence, you can factor this code out of your `MainViewController` and into a new class called `TraxyTopLevelViewController` so it can be shared across the tabs. In addition, all of your view controllers will support a log out button on the navigation bar so you can also move the logout `IBAction` out of `MainViewController` into this new class. Your `TraxyTopLevelViewController` should look like the code in Listing 141. Next, be sure to update your `MainViewController` class to extend `TraxyTopLevelViewController` instead of `TraxyBaseViewController`. Also in `MainViewController`, delete all the properties and functions that have been refactored into `TraxyTopLevelViewController` and override the hook method `journalsDidLoad` as follows:

Swift

```
override func journalsDidLoad() {
  if let j = self.journals {
    self.sortIntoSections(journals: j)
  } else {
    self.tableViewData?.removeAll()
  }
}
```

The code to be removed in `MainViewController` would include all of the code in `viewDidLoad`, `viewWillAppear` and `viewWillDisappear` except for the chained call to the corresponding super class method. Each of the top level view controllers in your tabs will need to know which user has authenticated, and the set of journals they own. The `TraxyTopLevelViewController`

class implements properties for this information. However, since each of them might possibly operate on a different subset of journals depending on their local context, a stubbed out hook method named `journalsDidLoad()` is defined and called whenever the `journals` property is set. You will override this method in the derived classes to customize what set of journals each will display.

Shortly, when you implement your custom tab bar controller class, it will listen for any user authentication changes via the `TraxyRepository` singleton. Anytime it observes a new user session it will inform all of its contained top level view controllers by setting the userId property. Hence the `TraxyTopLevelViewController` needs to do some housekeeping whenever this happens. Consider for example a scenario in which you are in the first tab, and have just entered a new journal entry and are on the confirmation screen. Then you switch to the second tab and hit the logout button. In this case, the tab bar controller will observe the authentication change and modally present the login scene. The interesting question at this point is what happens when a new user is authenticated? The first tab by default still contains its stack of view controllers even though the user and data has been cleared! Given this, whenever the userId gets set to a valid value we need to make sure we reset each tab to its pristine never been used yet configuration, which means we must pop any pushed views until we reach the top most controller. This is implemented in the `didSet` handler of the `userId` property.

Another interesting item to note in this code is the `shouldLoad` boolean. Since you will be reusing the code of the `MainViewController` (e.g. using it to populate the subset of journals on the calendar view) , you want to be able to hand it already loaded journals instead of having it try to load them directly from Firebase. Setting this property to false will cause it not to observe journal updates on Firebase.

12.2.3 Customizing UITabBarController to Handle Authentication

Introduce a new class that extends the `UITabBarController` class and name it `TraxyTabBarController` as shown in Listing 142. When the tab bar controller loads initially, it will adjust its appearance per theme constants established earlier. More importantly it observes user authentication changes for the duration of the application's lifetime, informing any of it's child view controllers of updates. The method named unwindFromSignup has been added for the sake of giving the `SignUpViewController` a target to unwind to. We will look at that shortly.

Don't forget to go into your storyboard and set the tab bar controller scene's class to `TraxyTabBarController` via the Identity Inspector or none of this goodness will be in effect! You also need to make some minor modifications to the login and signup view controllers in order to get them to dismiss themselves properly. Previously, you implemented these transitions as segues to the main view's navigation controller, but now since they are presented modally by the tab bar controller they will dismiss themselves.

Since `LoginViewController` is always presented directly by `TraxyTabBarController`. All you need to do is replace the original `performSegue` call with a `dismiss` call in the IBAction associated with the login button as shown in Listing 143. You can also remove the `IBAction` method named `logout` that we did unwind segues to previously, since you logout logic will now be handle by `TraxyTopLevelViewController`.

Since `SignUpViewController` is always presented by `LoginViewController`, if you simply call dismiss upon successful signup, it will dismiss itself, but we will still have the login screen displayed. To handle this, we will use an unwind segue to accomplish this and target the `unwindFromSignup` method defined by the `TraxyTabBarController`. Recall you imple-

iOS

```
class TraxyTopLevelViewController: TraxyBaseViewController {
    var shouldLoad = true
    var userEmail : String?
    var journals : [Journal]? {
        didSet {
            self.journalsDidLoad()
        }
    }

    let repo = TraxyRepository.getInstance()

    var userId : String? = "" {
        didSet {
            if userId != nil && userId != "" {
                // pop off any controllers beyond this one.
                if var count = self.navigationController?.children.count
                {
                    if count > 1 {
                        count -= 1
                        for _ in 1...count {
                            _ = self.navigationController?.popViewController(
                                animated: true)
                        }
                    }
                }
                if self.shouldLoad {
                    repo.listenForJournalUpdates { (journals) in
                        self.journals = journals
                    }
                }
            }
        }
    }

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

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

    override func viewWillAppear(_ animated: Bool) {
        super.viewWillAppear(animated)
        self.userId = repo.userId
    }

    override func viewWillDisappear(_ animated: Bool) {
        super.viewWillDisappear(animated)
        repo.stopListeningForUpdates()
    }

    @IBAction func logout() {
        // Note we need not explicitly do a segue as the auth listener on our
        // top level tab bar controller will detect and put up the login.
        repo.logout()
        self.journals?.removeAll()
```

iOS

```
58          self.journals = nil
59          self.userEmail = nil
60      }
61
62      // Hook that gets called after journals are loaded.
63      func journalsDidLoad()
64      {
65      }
66  }
```

Listing 141: Refactored authentication, journal loading, and logout code into a common base class named `TraxyTopLevelViewController`.

mented an unwind segue earlier. We can add an unwind segue in the storyboard by holding ctrl and clicking on the yellowish box on the top of the signup scene, and dragging over to the orange "exit" box to the right, as shown in Figure 12.6. When you release the mouse button a popup menu will display with all elgible target unwind methods, which at this time would be only the one defined in your `TraxyTabBarController`. Select the newly added unwind segue in the hierachical view to the left of your storyboard, and set it to `segueToMainFromSignUp` in the identity inspector. Since the previous code did exactly this, you need not make any additional changes to the `SignUpViewController` code. The last thing you need to do is to wire up the logout button on the top of the scene for the `MainViewController` to the logout action defined by `TraxyTopLevelViewController`. If your assistant editor is loading in automatic mode it will automatically load the source code for `MainViewController` when the controller is selected, so you may need to override the source selection to "manual" and navigate into the `TraxyTopLevelViewController` to wire up the action. Go ahead and build your project and run it, confirming that the login/signup and first tab are now fully functional.



Figure 12.6: Adding an unwind segue to the signup scene to get it (and the login scene under it) to dismiss upon successful signup.

Swift

```swift
class TraxyTabBarController: UITabBarController {

  var userId : String? = ""

  override func viewDidLoad() {
    super.viewDidLoad()
    UITabBar.appearance().isTranslucent = false
    UITabBar.appearance().barTintColor = THEME_COLOR2
    UITabBar.appearance().tintColor = THEME_COLOR3
    let repo = TraxyRepository.getInstance()
    repo.listenForAuthenticationChanges{ userId in
      if let uid = userId {
        self.userId = uid
        for child in self.children {
          if let nc = child as? UINavigationController {
            if let c = nc.children[0]
            as? TraxyTopLevelViewController {
              c.userId = self.userId
            }
          }
        }
      } else {
        // No user is signed in.
        self.performSegue(withIdentifier: "presentLogin", sender: self)
      }
    }
  }

  override func viewWillAppear(_ animated: Bool) {
    super.viewWillAppear(animated)
  }

  override func viewWillDisappear(_ animated: Bool) {
    super.viewWillDisappear(animated)
  }

  @IBAction func unwindFromSignup(segue: UIStoryboardSegue) {
    // we end up here when the user signs up for a new account.
  }

  override func didReceiveMemoryWarning() {
    super.didReceiveMemoryWarning()
  }
}
```

Listing 142: Customization of UITabBarController to handle changes in user authentication.

```swift
    @IBAction func signupButtonPressed(_ sender: UIButton) {
  if self.validateFields() {
    let repo = TraxyRepository.getInstance()
    repo.signIn(email: self.emailField.text!, password: self.passwordField.text!)
    {(success, errorMesg) in
      if success {
        self.dismiss(animated: true, completion: nil)
      } else {
        if let msg = errorMesg {
          self.reportError(msg: msg)
        }
        self.passwordField.text = ""
        self.passwordField.becomeFirstResponder()
      }
    }
  } else {
    self.reportError(msg: self.validationErrors)
  }
}
```

Listing 143: Updated IBAction of `LoginViewController` to handle successful login. Note the performSegue replaced with a dismiss call.

12.2.4 Adding the Calendar Tab

The second tab of Traxy will display a calendar on the top third of the screen, and below it a list of journals displayed whose date range falls into the month the calendar is displaying. In addition, you will annotate the calendar to show which days of the month are covered by journals. There are a number of useful third party pods available for displaying interactive calendar views in iOS. You will use the FSCalendar pod in Traxy. Go ahead and add this line to your project's Podfile:

```
pod 'FSCalendar'
```

Don't forget to install the pod after saving your Podfile by entering this command from a shell prompt:

```
pod install
```

Next, introduce a new view controller class in your project named `CalendarViewController`, then in your storyboard make sure to set the class of the view controller scene in the second tab to this class, using the identity editor. Have `CalendarViewController` extend `TraxyTopLevelViewController` instead of `UIViewController`. You are going to modify the storyboard scene in several ways. First you will add bar buttons to the navigation bar for logging out and for adding a new journal. Second, you will add a calendar to the top portion of the scene and third, display the journals under the calendar by embedding our original `MainViewController` within the `CalendarView` Controller.

Adding Logout and Add Bar Buttons

Drag a bar button labeled "Logout" to the upper left of the `CalendarViewController`'s navigation bar. Go ahead and wire it up to the `logout` IBAction defined by `TraxyTopLevelViewController` as before. Next, on the right side of the navigation bar add a bar button item of type Add. Drag a Show segue from the Add button to the `AddJournalViewController` scene and give it the identifier `addJournalSegue`.

Adding the Calendar View

Add a generic `UIView` (labeled simply "View" in the Xcode object library) to the top one third of the screen using Auto Layout constraints. Using the identity editor, set the view's class to `FSCalendar`. This type of course is defined by the FSCalendar pod, and will display an interactive calendar for us. Create an outlet from this view in `CalendarViewController` named `calendar` of type `FSCalendar!`. Don't forget to import the `FSCalendar` pod at the top of your `CalendarViewController` source file. Drop a container view (labeled **Container View** in the Xcode object library) into the space underneath the calendar and set Auto Layout constraints to have it fill that region. Next, make the `CalendarViewController` both the delegate and the data source of the calendar object. Click on the calendar object while holding the ctrl key, and drag to the round yellow icon on the top of the scene in the storyboard. When you release you will have options for both delegate and datasource. Select one, and then repeat the click / drag to add the second.

Embedding MainViewController

Our next step is rather nifty, so do pay attention. Recall we previously observed that the list of journals displayed under the calendar is essentially the same as the list of journals displayed on the main list under the first tab. The only different is that it will display only a subset of the journals based on what month the calendar is displaying. By simply embedding the `MainViewController` scene into the `CalendarViewController` scene and making sure we initialize it with the right data we can reuse all of our existing code. In essence, the calendar tab is just a simple filter in front of the code we've already implemented!

Delete the default view controller embedded in the lower region of the calendar scene, and instead add a segue between the embedded view and the MainViewController's scene. Be sure when the segue popup menu is displayed that you select *Embed* and not *Show*. Once the segue is created, select it in storyboard and set its identifier to `embeddedJournalViewSegue`.

We've been able to reuse an existing controller in its entirety using the Container View in Interface Builder within our application. This is a great way to be more productive, and is a good example of the DRY (don't repeat yourself) in action. Of course, there are some differences in how you will initialize this particular instantiation of MainViewController, and that's what we'll look at next.

Implementing CalendarViewController

Now let's implement the rest of the CalendarViewController class. The class is shown in its entirety in Listing 144. We store a reference to the embedded main view controller when we handle

the embed segue in the prepare:for:segue method. In addition, we make sure to set the embedded main view controller's shouldLoad property to false. It does not need to listen to Firebase for journal updates, because the CalendarViewController already is.

Note that the journalsDidLoad() hook method gets called whenever the base class's journals property is set. In this case, that is where you are going to place your code that filters the current month's journals which you will can see in the `filterJournalsByDisplayDate` helper method. Once you've filtered the journals, you will set them on the embedded main view controller's journals property which will cause it to render them in its tableview.

In order to compare the dates by month we had to add another computed property `monthYear` to our Date extension that formats the date into the format "yyyy-MM". You can implement that as homework, or go look at how we did it in the book's git repo.

Finally, we set the CalendarViewController class to be the delegate and data source for the calendar, so you need to implement those methods. Listing 145 shows how to implement the `FSCalendarDelegate` and `FSCalendarDataSource` as extensions. For the former, all you need to be concerned with is running the filter again, which can be accomplished by calling the `journalsDidLoad` method again. For the data source, we make the individual days unselectable and provide counts for the number of journals on a given day.

You should be able to build and run your project at this point and both tabs should be fully functional.

12.2.5 Adding Cover Photo Selection

The final task is to give the user the opportunity to select a cover photo for their journal as was shown in Figure 12.2c. Previously, you implemented logic that would automatically select the first journal entry as a photo to serve as the cover photo. To give the user more control, we will use the iOS `UICollectionView` to display the thumbnails underneath the form we use to enter and edit journals, giving the user the ability to select one of them to serve as the journal's cover photo. To accomplish you are going to once again do a bit of code refactoring. In particular, a new class called `JournalEditorViewController` will be implemented that embeds your existing `AddJournalViewController`, and a new controller named `CoverPhotoCollectionViewController` that will display the photo thumbnails. These changes will also result in a number of updates to existing classes, so pay attention as you work through the details below.

Stub Out the New Classes for the Cover Photo Feature

Before you update the storyboard, go ahead and stub out the following new Swift classes that will be implemented in this section.

- `JournalEditorViewController`: extends the `UIViewController` class. This class will be used as your top level controller for adding and editing journals.
- `CoverPhotoCollectionViewController`: extends the `UIViewController` class. This class will display photo thumbnails and allow the user to select one via `UICollectionView`.
- `CoverPhotoCollectionViewCell`: extends the `UICollectionViewCell` class. This class will be used for your customized cells to display/select photos.

Swift

```swift
class CalendarViewController: TraxyTopLevelViewController {

  var journalView : MainViewController?
  @IBOutlet weak var calendar: FSCalendar!

  override func viewDidLoad() {
    super.viewDidLoad()
    self.calendar.backgroundColor = THEME_COLOR3
  }

  override func didReceiveMemoryWarning() {
    super.didReceiveMemoryWarning()
    // Dispose of any resources that can be recreated.
  }

  override func journalsDidLoad() {
    if let _ = self.journals, let jv = self.journalView{
      let fj = filterJournalsByDisplayDate()
      jv.sortIntoSections(journals: fj)
    } else if let jv = self.journalView {
      jv.tableViewData?.removeAll()
    }
  }

  func filterJournalsByDisplayDate() -> [Journal]
  {
    var filteredJournals : [Journal] = []
    if let journals = self.journals {
      let date = self.calendar.currentPage
      for j in journals {
        if date.monthYear >= j.startDate!.monthYear
        && date.monthYear <= j.endDate!.monthYear
        {
          filteredJournals.append(j)
        }
      }
    }
    return filteredJournals
  }

  // MARK: - Navigation
  override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    if segue.identifier == "embeddedJournalViewSegue" {
      self.journalView = segue.destination as? MainViewController
      self.journalView?.shouldLoad = false
    } else  if segue.identifier == "addJournalSegue" {
      if let destVC = segue.destination as? JournalEditorViewController {
        destVC.delegate = self.journalView
      }
    }
  }
}
```

Listing 144: The implementation of CalendarViewController..

```swift
extension CalendarViewController : FSCalendarDelegate {
  func calendar(_ calendar: FSCalendar, boundingRectWillChange bounds: CGRect,
  animated: Bool) {
  }

  func calendar(_ calendar: FSCalendar, didSelect date: Date) {
  }

  func calendarCurrentPageDidChange(_ calendar: FSCalendar) {
    self.journalsDidLoad()
  }
}

extension CalendarViewController : FSCalendarDataSource {
  func numberOfJournalsOnDate(date: Date) -> Int {
    var cnt = 0
    if let items = self.journals {
      for j in items {
        guard let startDate = j.startDate, let endDate = j.endDate else {
          continue
        }
        if date >= startDate && date <= endDate {
          cnt+=1
        }
      }
    }
    return cnt
  }

  func calendar(_ calendar: FSCalendar, numberOfEventsFor date: Date) -> Int {
    return self.numberOfJournalsOnDate(date: date)
  }


  @objc(calendar:shouldSelectDate:) func calendar(_ calendar: FSCalendar,
    shouldSelect date: Date) -> Bool
  {
    return false
  }
}
```

Listing 145: The implementation of the FSCalendar..

Updating the Storyboard for the Cover Photo Feature

The final goal for the scenes impacted is shown in Figure 12.7. Let's walk through this in detail to make sure you get everything setup as needed before we implement the new classes described above.

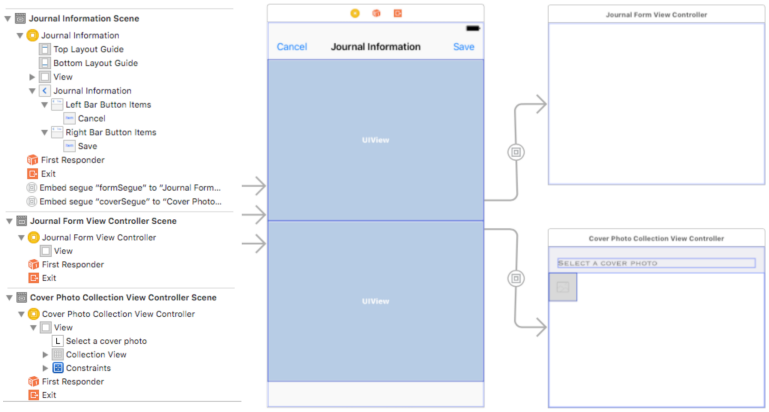


Figure 12.7: The storyboard after adding making the modifications to display cover photos in the journal editor.

Open up your storyboard, and locate the scene associated with the `AddJournalViewController`. In the identity inspector change its class to the `JournalEditorViewController` class. Next, add a container view (labeled **Container View** in the object library) to the scene. Using Auto Layout constraints pin this container view to the top, leading, and trailing edges of the parent view, and set its height to 280. Using the identity inspector, set the class of the embedded view controller emanating from the container to be `AddJournalViewController`. Set the identifier on the embed segue to `formSegue`. This embedded scene will contain the journal form fields. As before, you need not add any additional controls to it as the Eureka form defined in the `AddJournalViewController` will populate its view hierarchy.

Fill the space in the `JournalEditorViewController` scene beneath the embedded form with another container view. Using Auto Layout contraints pin the top of the view to the bottom of the embedded form view, and its leading, trailing, and bottom edges to the parent view. Using the identity inspector, set the class of the embedded view controller emanating from the view controller to be `CoverPhotoCollectionViewController`. Set the embed segue identifier to `coverSegue`.

Set the background of the `CoverPhotoCollectionViewController`'s top level view to "System Grouped Background Color" in the Attribute Inspector. Drag out a `UILabel` with the text "Select cover photo" to the top left of the scene. To match the font used in the form section header above, set the label's font to "Copperplate", style "Light" and size "16".

Drag out a `UICollectionView` (labeled "Collection View" in the library) underneath the label, and pin its top standard distance from the bottom of the label. Pin its leading, trailing, and bottom edges to the containing parent view. Go into the Size Inspector and set the collection view's cell size to 50, and set Estimate Size to None. Collection views work very similar to table views

in iOS, so we'll want to have the `CoverPhotoCollectionViewController` implement the `UICollectionViewDelegate` and `UICollectionViewDataSource` protocols. Control click on the collection view in the scene and drag up to the yellow circle representing the view controller, release and select delegate. Repeat the process to set the data source as well. Also, you need to create an outlet named `collectionView` on the `CoverPhotoCollectionViewController` class.

You need to customize the prototype `UICollectionViewCell` that was populated by default when you dragged the collection view into the scene. Select the prototype cell and on the Identity Inspector change the class to `CoverPhotoCollectionViewCell`, and its identifier to be `PhotoCell`. Drag a `UIImage` into the cell, pin it to the containing view's leading, trailing, top, bottom edges, and in the Attributes Inspector set the Content mode to "Aspect Fill". You can implement the `CoverPhotoCollectionViewCell` as shown in Listing 146. Be sure to wire up the image in the scene to the outlet named `photo`.

Swift

```swift
class CoverPhotoCollectionViewCell: UICollectionViewCell {

  @IBOutlet weak var photo: UIImageView!

  override var isSelected: Bool {
    didSet {
      self.photo.layer.borderWidth = isSelected ? 5 : 0
    }
  }

  override func awakeFromNib() {
    super.awakeFromNib()
    self.photo.layer.borderColor = THEME_COLOR2.cgColor
    isSelected = false
  }
}
```

Listing 146: The custom UICollectionViewCell implementation.

The custom cell overrides the awakeFromNib and isSelected methods to effectively implement the selection visualization. The selected cell will have a 5 pixel colored border drawn around it to set it off from the others.

Add a cancel button to the left side of the navigation bar and a save button to the left side as shown in 12.7. You will implement and wire up their respective actions in a subsequent section. You need not worry about the scene's title in Interface Builder as we will be setting this in code based on whether a new journal is being added or an existing journal edited.

Finally, the new `JournalEditorViewController` scene will be displayed from a couple of different contexts. First, as before, it will be displayed whenever a user taps the '+' button on the right side of `MainViewController`'s scene to create a new journal. This segue (`addJournalSegue`) was added previously, so nothing more needs to be done. Second, the `JournalEditorViewController` will be displayed by the `JournalTableViewController` to allow the user to edit the journal's metadata, and select a new cover photo for the journal. To facilitate this, go ahead and add a new "Edit" button to the `JournalTableViewController`'s

navigation bar, directly left of the '+' button that is already there. Next, add a segue from this new button to the `JournalEditorViewController` and give it the identifier `editJournalSegue`. This segue will allow the user to edit the journal title, location, dates and cover photo for the journal currently being displayed by the `JournalTableViewController`.

subsubsectionUpdating the Journal struct and its Dependencies

The `CoverPhotoCollectionViewController` will allow the user to select a cover photo for a journal. We need to add a new property called `coverPhotoUrl` to the `JournalStruct` to store the url of the selected cover photo as shown in Listing 147.

This update of course is going to break any code that initializes Journal instances. In particular, the `TraxyTopLevelViewController` class has a helper method named `registerForFireBaseUpdates` that needs to be updated to use the initializer as is shown in Listing 148.

Implementing the CoverPhotoCollectionViewController Class

The `UICollectionView` class defined by iOS is in many ways analogous to the `UITableView` class that you're already familiar with. In order to efficiently manage a multiplicity of of cells that can be scrolling off and on the screen, some sort of recycling needs to take place. This is accomplished by implementing the `UICollectionViewDataSource` protocol. One of the things we need to display is copies of each of the photos found within the journal so the user can select one of them to serve as the cover photo. We could easily pass along the copy we already have in the `JournalTableViewController`'s model, but to demonstrate how to do a query on our Firestore (e.g. not listen perpetually for updates, but just execute a single query) let's add a new method to our `TraxyRepository` class called `queryEntries` as shown in Listing 149. We'll implement the query in a way that allows us to retrieve all entries, or entries of a specific type.

Now Implement the `CoverPhotoCollectionViewController` class as shown in Listing 150. Most of the meat will be in your implementation of the delegate and the data source protocols. The main task here is to issue a query to the `TraxyRepository` whenever the journal object property is set. When the view loads, an empty journal is created (e.g. when the user adds a new journal), and the collection view also has to disable the selection of multiple cells in this particular use case.

You can control the size and the layout flow of your `UICollectionView` by implementing the `UICollectionViewDelegateFlowLayout` as shown in Listing 151. In this particular situation the collection view has defaulted to vertical scrolling, and the number of cells per row has been set to 4.

Next, go ahead and implement the `UICollectionViewDataSource` protocol as an extension as shown in Listing 152. Notice how the implementation is very similar to what you've already seen with table views. You're basically dequeuing photo cells and populating them with the image data associated with the entries of type photo that you filtered out previously. Note that if the URL of any entry that you are binding to a cell is equal to the URL that is already set on the journal, that particular cell must be programmatically "selected" in order for the selection visuals implemented in the custom `UICollectionViewCell` will be displayed.

Finally, complete your implementation of the class by implementing the UICollectionViewDelegate protocol as shown in Listing 153. This code will be executed whenever the user taps a cover, and it simply updates the journal's cover photo URL to be the URL of the tapped photo.

Swift

```swift
struct Journal {
  var key : String?
  var name : String?
  var location : String?
  var startDate : Date?
  var endDate : Date?
  var lat : Double?
  var lng : Double?
  var placeId : String?
  var coverPhotoUrl : String?

  init(key: String?, name: String?, location: String?, startDate: Date?,
    endDate : Date?, lat: Double?, lng: Double?, placeId : String?,
    coverPhotoUrl: String?)
  {
    self.key = key
    self.name = name
    self.location = location
    self.startDate = startDate
    self.endDate = endDate
    self.lat = lat
    self.lng = lng
    self.placeId = placeId
    if let cover = coverPhotoUrl {
      self.coverPhotoUrl = cover
    } else {
      self.coverPhotoUrl = ""
    }
  }

  init(name: String?, location: String?, startDate: Date?, endDate : Date?,
    lat: Double?, lng: Double?, placeId : String?)
  {
    self.init(key: nil, name: name, location: location, startDate: startDate,
    endDate: endDate, lat: lat, lng: lng, placeId: placeId, coverPhotoUrl: "")
  }

  init() {
    self.init(key: nil, name: nil, location: nil, startDate: nil, endDate: nil,
    lat: nil, lng: nil, placeId: nil, coverPhotoUrl: "")
  }
}
```

Listing 147: The updated Journal struct with its new coverPhotoUrl property.

```swift
func listenForJournalUpdates(onChange: @escaping ([Journal]) -> Void) {
  self.listener = self.ref?
  .collection("journals").addSnapshotListener({ (snapshot, error) in
    guard let documents = snapshot?.documents else {
      return
    }
    var journals = [Journal]()
    for doc in documents {
      let key = doc.documentID
      let name : String? = doc["name"] as! String?
      let location : String?  = doc["address"] as! String?
      let startDateStr  = doc["startDate"] as! String?
      let endDateStr = doc["endDate"] as! String?
      let lat = doc["lat"] as! Double?
      let lng = doc["lng"] as! Double?
      let placeId = doc["placeId"] as! String?
      let coverPhotoUrl = doc["coverPhotoUrl"] as! String?

      let journal = Journal(key: key, name: name, location: location,
        startDate: startDateStr?.dateFromISO8601, endDate:
        endDateStr?.dateFromISO8601, lat: lat, lng: lng,
        placeId: placeId, coverPhotoUrl: coverPhotoUrl)
      journals.append(journal)
    }
    onChange(journals)
  })
}
```

Listing 148: The updated `listenForJournalUpdates` method defined by `TrayRepository`.

Swift

```swift
func queryEntries(key: String, type: EntryType?, onCompletion:
  @escaping ([[String : Any]]?) -> Void)
{
  if let uid = userId {
    let entries = Firestore.firestore()
      .collection("user/\(uid)/journals/\(key)/entries")
    let query: Query
    if let t = type {
      query = entries.whereField("type", isEqualTo: t.rawValue)
      query.getDocuments() {(snapshot, err) in
        if err != nil {
          print("whoops query error")
          onCompletion(nil)
        } else {
          var vals = [[String:Any]]()
          for doc in snapshot!.documents {
            vals.append(doc.data())
          }
          onCompletion(vals)
        }
      }
    } else {
      entries.getDocuments() {(snapshot,err) in
        if err != nil {
          print("whoops query error")
          onCompletion(nil)
        } else {
          var vals = [[String:Any]]()
          for doc in snapshot!.documents {
            vals.append(doc.data())
          }
          onCompletion(vals)
        }
      }
    }
  } else {
    onCompletion(nil)
  }
}
```

Listing 149: The implementation of `queryEntries` method on `TraxyRepository`.

Swift

```swift
class CoverPhotoCollectionViewController: UIViewController {

  fileprivate let insets = UIEdgeInsets(
    top: 18.0, left: 18.0,
    bottom: 18.0, right: 18.0
  )
  fileprivate let ROW_SIZE : CGFloat  = 4.0
  fileprivate let cellId = "PhotoCell"

  @IBOutlet weak var collectionView: UICollectionView!

  var entries : [[String:Any]]?
  let repo = TraxyRepository.getInstance()

  var journal : Journal? {
    didSet {
      if let key = journal?.key {
        repo.queryEntries(key: key, type: .photo) { (entries) in
          self.entries = entries
          self.collectionView.reloadData()
        }
      }
    }
  }

  override func viewDidLoad() {
    super.viewDidLoad()
    self.collectionView.allowsMultipleSelection = false
    if self.journal == nil {
      self.journal = Journal()
    }
  }

  override func didReceiveMemoryWarning() {
    super.didReceiveMemoryWarning()
  }
}
```

Listing 150: The implementation of the CoverPhotoCollectionViewController class.

```swift
extension CoverPhotoCollectionViewController : UICollectionViewDelegateFlowLayout {
  func collectionView(_ collectionView: UICollectionView,
    layout collectionViewLayout: UICollectionViewLayout,
    sizeForItemAt indexPath: IndexPath) -> CGSize
  {

    let paddingSpace = self.insets.left * (ROW_SIZE + 1.0)
    let availableWidth = view.frame.width - paddingSpace
    let widthPerItem = availableWidth / ROW_SIZE

    return CGSize(width: widthPerItem, height: widthPerItem)
  }

  func collectionView(_ collectionView: UICollectionView,
    layout collectionViewLayout: UICollectionViewLayout,
    insetForSectionAt section: Int) -> UIEdgeInsets
  {
    return self.insets
  }

  func collectionView(_ collectionView: UICollectionView,
    layout collectionViewLayout: UICollectionViewLayout,
    minimumLineSpacingForSectionAt section: Int) -> CGFloat
  {
    return self.insets.left
  }
}
```

Listing 151: Implementing the UICollectionViewDelegateFlowLayout to impact the flow, size and positioning of the cells.

```swift
extension CoverPhotoCollectionViewController : UICollectionViewDataSource {
  func numberOfSections(in collectionView: UICollectionView) -> Int {
    return 1
  }

  func collectionView(_ collectionView: UICollectionView,
  numberOfItemsInSection section: Int) -> Int
  {
    if let entries = self.entries {
      return entries.count
    }
    return 0
  }

  func collectionView(_ collectionView: UICollectionView,
  cellForItemAt indexPath: IndexPath) -> UICollectionViewCell
  {
    let cell = collectionView.dequeueReusableCell(withReuseIdentifier: cellId,
    for: indexPath) as! CoverPhotoCollectionViewCell
    cell.photo.image = UIImage(named: "landscape")
    if let entry = self.entries?[indexPath.row] {
      if let url = entry["url"] as? String {
        cell.photo?.kf.indicatorType = .activity
        cell.photo?.kf.setImage(with: URL(string: url))
        if self.journal?.coverPhotoUrl == url {
          cell.isSelected = true
          self.collectionView.selectItem(at: indexPath, animated: true,
          scrollPosition: .top)
        }
      }
    }
    return cell
  }
}
```

Listing 152: Implementing the UICollectionViewDataSource protocol.

```swift
extension CoverPhotoCollectionViewController : UICollectionViewDelegate {
  func collectionView(_ collectionView: UICollectionView,
    didSelectItemAt indexPath: IndexPath)
  {
    if let entry = self.entries?[indexPath.row] {
      self.journal?.coverPhotoUrl = entry["url"] as? String
    }
  }
}
```

Listing 153: Implementing the UICollectionViewDelegate protocol.

Implementing the JournalEditorViewController Class

Implement the JournalEditorViewController as shown in Listing 154. In lines 67 - 75 references are picked up for the two embedded controllers. The save action is essentially the same as originally implemented by the `AddJournalViewController` class, only now the form values are accessed via the embedded form view controller. Since the `JournalEditorViewController` is now handling the save and cancel buttons instead of the embedded `AddJournalViewController`, it is also responsible for delegating the persisting of these changes back to the `MainViewController`. Therefore, the `AddJournalDelegate` has been renamed `JournalEditorDelegate`. You'll have to make the appropriate changes to `MainViewController` to accomodate this change, which will be discussed in the section that follows.

Before that however, now that you have the actions for the save and cancel buttons implemented in the `JournalEditorViewController`, be sure to return to your storyboard and hook these actions up to the buttons in the controller's scene.

Updating `AddJournalViewController`

A number of our existing classes need updating in order to complete the implementation of the cover photo feature. Parts of `AddJournalViewController` have been refactored out into the `JournalEditorViewController` class can be removed. Since the form's view controller is now embedded, the navigation bar is actually associated with the `JournalEditorViewController` scene, so the `IBActions` written previously for the cancel and save button can be removed from `AddJournalViewController`, as well as the original code in the `viewDidLoad` method that instantiated these buttons. You also need to make some minor modifications to the code that creates each row in the Eureka form instantiated in `AddJournalViewController`'s `viewDidLoad` method. Note that when segued to from the `JournalTableViewController` the form will be used to *edit* an existing journal, so the form should be pre-populated with the values in the `journal` property.

For the title row the value is conditionally assigned the journal name:

Swift

```
1 row.value = self.journal?.name
```

For the location row the default value is conditionally assigned the journal location:

Swift

```
1 if let loc = journal?.location {
2   row.value = loc
3 } else {
4   row.value = "Tap to search"
5 }
```

For start date row, the default value is conditionally assigned the journal start date:

iOS

```
import UIKit
import Eureka

protocol JournalEditorDelegate : class {
    func save(journal: Journal)
}

class JournalEditorViewController: UIViewController {

    var journalForm : AddJournalViewController!
    var coverSelect : CoverPhotoCollectionViewController!

    weak var delegate : JournalEditorDelegate?
    var journal : Journal?

    override func viewDidLoad() {
        super.viewDidLoad()
        if self.journal == nil {
            self.navigationItem.title = "Add Journal"
            self.journal = Journal()
        } else {
            self.navigationItem.title = "Edit Journal"
        }
    }

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

    @IBAction func cancelPressed(_ sender: UIBarButtonItem)
    {
        _ = self.navigationController?.popViewController(animated: true)
    }

    @IBAction func savePressed(_ sender: UIBarButtonItem)
    {
        let form = self.journalForm.form
        let errors = form.validate()
        if errors.count > 0 {
            print("fix ur errors!")
        } else {
            let titleRow: TextRow! = form.rowBy(tag: "TitleTag")
            let locRow: LabelRow! = form.rowBy(tag: "LocTag")
            let startDateRow : DateRow! = form.rowBy(tag: "StartDateTag")
            let endDateRow : DateRow! = form.rowBy(tag: "EndDateTag")

            let title = titleRow.value! as String
            let location = locRow.value! as String
            let startDate = startDateRow.value! as Date
            let endDate = endDateRow.value! as Date

            self.journal?.name = title
            self.journal?.location = location
            self.journal?.startDate = startDate
            self.journal?.endDate = endDate
            self.journal?.coverPhotoUrl = self.coverSelect.journal?.coverPhotoUrl
            self.journal?.lat = self.journalForm.journal?.lat
```

iOS
```
            self.journal?.lng = self.journalForm.journal?.lng
            self.journal?.placeId = self.journalForm.journal?.placeId

            self.delegate?.save(journal: self.journal!)
            _ = self.navigationController?.popViewController(animated: true)
        }
    }

    // MARK: - Navigation
    override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
        if segue.identifier == "formSegue" {
            self.journalForm = segue.destination as? AddJournalViewController
            self.journalForm?.journal = self.journal
        } else if segue.identifier == "coverSegue" {
            self.coverSelect = segue.destination as? CoverPhotoCollectionViewController
            self.coverSelect?.journal = journal
        }
    }
}
```

Listing 154: The complete implementation of the new `JournalEntryEditorViewController` class.

Swift
```
if let date = journal?.startDate {
  row.value = date
} else {
  row.value = Date()
}
```

For end date row, the default value is conditionally assigned the journal end date:

Swift
```
if let date = journal?.endDate {
  row.value = date
} else {
  row.value = Date(timeIntervalSinceNow: 86401)
}
```

Updating JournalTableViewController

Recall that when you edited the storyboard earlier, you added a new edit button to `JournalTableViewController`'s scene, and a segue (with identifier `editJournalSegue`) to the `JournalEditorViewController`. The purpose of this was to allow the user to edit information associated with an existing journal and to select a cover photo for the journal, from the

photos associated with that journal.

Hence, you need to make a couple of updates to this controller. First, note that `JournalEditorViewController` is going to delegate its save button behavior back to the `MainViewController` via the new `JournalEditorDelegate`. The `MainViewController` already has the logic to persist the updated journal data to Firebase with its implementation of the original `AddJournalDelegate`. Rather than replicating that, we'll add an instance variable to `JournalTableViewController` to reference the `JournalEditorDelegate`. This reference can be passed along to the `JournalEditorViewController` whenever `JournalTableViewController` segues to it.

Swift

```
var journalEditorDelegate : JournalEditorDelegate!
```

Second, you need to update the prepareForSegue method to handle the new segue. In particular, the current journal needs to be passed to the JournalEditorViewController, along with the reference to the `JournalEditorDelegate` so the changes can be properly persisted. This code is shown in Listing 155.

Swift

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
  if segue.identifier == "confirmSegue" {
    if let destCtrl = segue.destination as? JournalEntryConfirmationViewController {
      destCtrl.imageToConfirm = self.capturedImage
      destCtrl.delegate = self
      destCtrl.type = self.captureType
      destCtrl.entry = self.entryToEdit  // will be nil on new item
      destCtrl.journal = self.journal
    }
  } else if segue.identifier == "recordAudio" {
    if let destCtrl = segue.destination as? AudioViewController {
      destCtrl.entry = self.entryToEdit // will be nil if new item.
      destCtrl.delegate = self
      destCtrl.journal = self.journal
    }
  } else if segue.identifier == "editJournalSegue" {
    if let destVC = segue.destination as? JournalEditorViewController {
      destVC.delegate = self.journalEditorDelegate
      destVC.journal = self.journal
    }
  }
}
```

Listing 155: The updated prepareForSegue method implemented by `JournalTableViewController`. Note the `else if` added for the segue identifier `editJournalSegue`.

Updating MainViewController

You're almost done with this feature! A few minor adjustments need to be made to `MainViewController` to accommodate our previous changes. The `AddJournalDelegate` was renamed to `JournalEditorDelegate`, so be sure to make this update in `MainViewController`'s class definition.

Next, in the previous section you had `JournalTableViewController` pass a reference to the `MainViewController` (acting as the `JournalEditorDelegate`) on to `JournalEditorViewController` when the edit button was tapped. Hence, you need to make sure that `MainViewController` passes a reference to itself along to the `JournalEditorViewController` when segueing to it. This can be accomplished by adding a single assignment statement to the `prepareForSegue` method as shown in Listing 156.

Swift

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
  if segue.identifier == "addJournalSegue" {
    if let destVC = segue.destination as? JournalEditorViewController {
      destVC.delegate = self
    }
  } else if segue.identifier == "showJournalSegue" {
    if let destVC = segue.destination as? JournalTableViewController {
      let indexPath = self.tableView.indexPathForSelectedRow
      let values = self.tableViewData?[indexPath!.section]
      destVC.journal  = values?.journals[indexPath!.row]
      destVC.userId = self.userId
      destVC.journalEditorDelegate = self
    }
  }
}
```

Listing 156: The updated prepareForSegue method implemented by `MainViewController`. Note the additional assignment statement added to the `else if` clause for `showJournalSegue`.

Another little but important details is that, we need to update the `TraxyRepository toDictionary` helper function to include the new `coverPhotoUrl` property when we persist to Firebase. The code for this update is shown in Listing 157.

Finally, now that you are selecting a cover image for each journal, we need to update the code in our `MainViewController`'s `tableview:cellForRowAt` method so that it loads the correct cover image if we in fact have a cover image URL stored for the journal. This is easily accomplished as shown in Listing 158.

At this point you should be able to build your project and confirm that both the features we've added are fully functional. Let's now turn to Android and see how you can extend the Android version of Traxy in a functionally equivalent manner.

```Swift
fileprivate func toDictionary(vals: Journal) -> [String:Any] {
  return [
    "name": vals.name! as NSString,
    "address": vals.location! as NSString,
    "startDate" : NSString(string: (vals.startDate?.iso8601)!) ,
    "endDate": NSString(string: (vals.endDate?.iso8601)!),
    "lat" : NSNumber(value: vals.lat!),
    "lng" : NSNumber(value: vals.lng!),
    "placeId" : vals.placeId! as NSString,
    "coverPhotoUrl" : vals.coverPhotoUrl! as NSString
  ]
}
```

Listing 157: The updated implementation of `toDictionary` helper on `TraxyRepository`.

```Swift
if let coverUrl = journal.coverPhotoUrl {
  if coverUrl != "" {
    let url = URL(string: coverUrl)
    cell.coverImage?.kf.indicatorType = .activity
    cell.coverImage?.kf.setImage(with: url)
  } else {
    cell.coverImage?.image = UIImage(named: "landscape")
  }
} else {
  cell.coverImage?.image = UIImage(named: "landscape")
}
```

Listing 158: Updating the `tableview:cellForRowAt` method on `MainViewController` so it loads the cover image if we have one.

12.3 Tabs and Grid Layouts on Android

Material Design Guidelines recommend use of tabs for easy switching between different views. This is a perfect use case for the next feature you are about to add to Traxy: allowing your users to view a list of either *all* the journals or only *selected* journals in a particular month.

Android refers to this kind of view switching as *lateral navigation* and the user can either use a (horizontal) swipe gesture or select a tab of interest. Using a similar paradigm for handling (vertical) scrolls through a list of items, implementation of lateral navigation used to rely on a `ViewPager` and a `PagerAdapter`. As of April 2020, these classes were deprecated and replaced with the newer component `ViewPager2` which no longer requires the accompanying pager adapter. Instead, `ViewPager2` was designed to work with `RecyclerView.Adapter`, the same adapter used for handling vertical scroll list in an earlier chapter.

This class requires the following dependencies (or newer version) in your app `build.gradle`:

Gradle

```
implementation 'androidx.viewpager2:viewpager2:1.0.0'
implementation 'com.google.android.material:material:1.3.x'
```

At the time of this writing, the Android material library is being upgraded from the `1.3.0-alpha`*xx* to `1.3.0-beta`*yy*. In this section you will upgrade the main screen to present two views: a list of journals (currently implemented by `MainFragment`) and a "monthly view" of journals. Switching between the two views will be implemented using two different techniques: (a) swipe gesture only, and (b) swipe gesture and tab selection. You will first implement the feature using only swipe gestures.

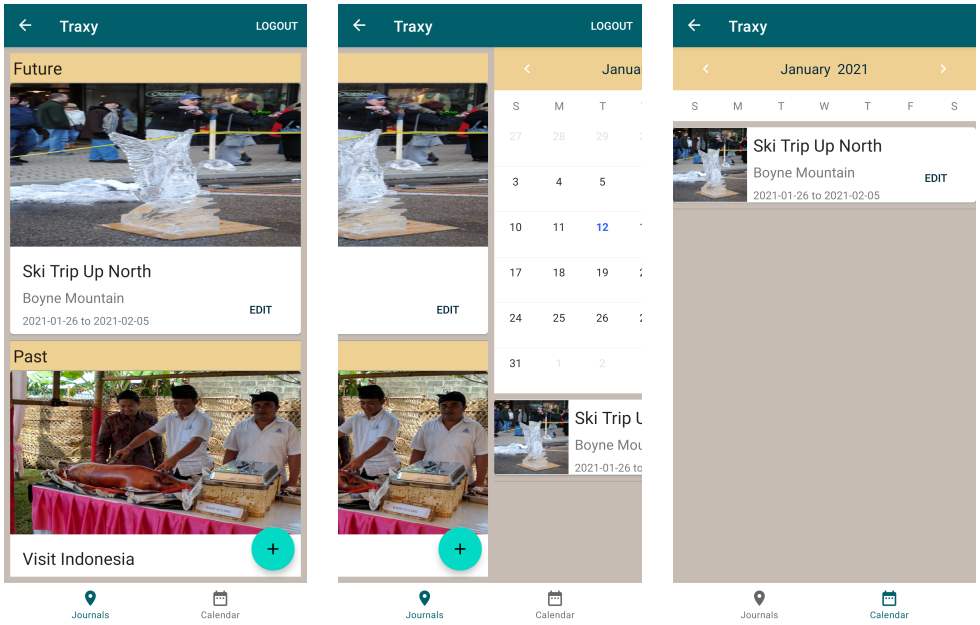
The completed screen looks like Figure 12.8. Let's make some changes to your app:

1. Create a new blank fragment `JournalPagerFragment`. This fragment will serve as the "parent container" of the two "child" fragments: `MainFragment` (already existed) and `MonthLyFragment` to be created below

2. Edit the XML layout file associated with this fragment

   - In the component tree of the layout editor, right-click on the top level container (most likely a FrameLayout) and select the "Convert View" menu option to change it to a `LinearLayout`. Then use its attribute panel (on the right side) to set its orientation to "vertical"
   - Using the same approach, convert the only child element (most likely a TextView) to a `ViewPager2`. Since this view class is not shown in the default selection, you must type its name in the search box at the top of the pop-up dialog.
   - Set the id of the `ViewPager2` component to `pager` (this name will be used later below)

3. Create another blank fragment `MonthlyFragment` and for debugging purposes, change its TextView so it shows "Monthly" (or some other text for easy identification). This fragment will show journals filtered by month. Don't worry about completing its detail; your first order of business is to make sure that you are able to switch between the two views.



(a) MainFragment *swipe gesture* (b) MonthlyFragment

Figure 12.8: Lateral Navigation Using Horizontal Swipes

12.3.1 Pager Adapter

Following the same principle of creating a scrollable list discussed earlier, the view pager class coordinates its work using an adapter. However, in this case the class must inherit from `FragmentStateAdapter`. Go ahead and create a new class `JournalPagerAdapter`:

Kotlin

```
class JournalPagerAdapter(fragment: Fragment) : FragmentStateAdapter(fragment) {
  // We have only two child fragments
  override fun getItemCount(): Int = 2

  override fun createFragment(position: Int): Fragment {
      when (position) {
          0 -> return MainFragment()
          1 -> return MonthlyFragment()
          else -> TODO()  // Should not be called
      }
  }
}
```

> **NOTE**
>
> The `TODO()` line above is required to avoid compile error because the function is required to return a fragment. It is essentially a Kotlin shortcut for throwing an exception. The way the code is setup with item count is set to 2, we know that when the `createFragment` is called the value of position either 0 or 1. However, the compiler does not know this fact and it incorrectly determines that the function is missing a necessary `return` statement.

To use the above class with the ViewPagers above, add the following statement in `onViewCreated` of `JournalPagerFragment`:

Kotlin

```
// in onViewCreated of JournalPagerFragment.kotlin
pager.adapter = JournalPagerAdapter(this)
```

12.3.2 Update The Navigation Graph

Our navigation graph currently designate `MainFragment` as the primary "landing" screen after the user is authenticated with userid and password. With the introduction of the view pager, this fragment becomes a child of `JournalPagerFragment`. The latter should now be designated as the primary "landing" screen in the navigation graph and the following changes to the navigation graph are necessary:

1. Add `JournalPagerFragment` as a new destination in the navigation graph
2. The destination of the following actions must be updated from MainFragment to JournalPagerFragment. This update can be accomplished by changing the destination in the attribute panel (to the right)
   - `action_login2main`: from LoginFragment to MainFragment
   - `action_signup2main`: from SignUpFragment to MainFragment
3. Likewise, the source of the following action must be changed from MainFragment to JournalPagerFragment.
   - `action_logout`: from MainFragment to LoginFragment
   - `action_new_journal`: from MainFragment to NewJournalFragment
   - `action_to_medialist`: from MainFragment to JournalViewFragment

   Unfortunately, the navigation graph editor currently does not support a drag-and-drop feature to redirect action arrows to change its source. You have to either manually edit the XML file (by moving the `<action>` tag to a new "parent" or recreate the action arrow altogether.
4. You may see compile errors in all the affected fragments after the above changes. For instance in `LoginFragment` and `SignUpFragment`, the new actions are no longer have userid payload. The passing of userid is not needed anymore and can be removed. In other fragments,

the errors were caused by new names for the `xxxxxDirections` object. Locate the errors around the `navigate()` calls and remove the unused parameter/incorrect name when the action objects are created.

Rerun your app and make sure that it can handle swipe gestures to switch between the two fragments. The newly created fragment shows nothing interesting other than one-line text. With a few more lines of code you can add tabs to the action bar.

12.3.3 Adding Tabs

Without additional visual cues, most users may not realize that they can use swipe gesture to switch between the two views. Adding more visual cues, like tabs, would help these users to navigate between the two views. One of the options is to add the navigation tabs above the view pager. This can be done by adding a few lines of code in XML layout and parent fragment.

1. Edit the XML layout file of `JournalPagerFragment` and add a `TabLayout` as a first child of the LinearLayout

XML

```xml
<LinearLayout ...>
  <com.google.android.material.tabs.TabLayout
      android:id="@+id/tabs"
      android:layout_width="match_parent"
      android:layout_height="wrap_content" />

  <androidx.viewpager2.widget.ViewPager2
      android:id="@+id/pager"
      android:layout_width="match_parent"
      android:layout_height="match_parent"
      android:layout_weight="1" />

</LinearLayout>
```

   To make sure the view pager fills the remaining available space below the tabs, we set its `layout_weight` to 1

2. Link the view pager to the `TabLayout` just inserted so the tabs are visible and updated in sync with the swipe gestures

Kotlin

```kotlin
// in onViewCreated() of JournalPagerFragment

TabLayoutMediator (tabs, pager) { tab, pos ->
  when (pos) {
    0 -> tab.text = "Journals"
    1 -> tab.text = "Calendar"
  }
}.attach()
```

Rerun the app and make sure you can switch between the two views by using either horizontal swipes or tapping the tabs. Once this feature is implemented correctly, you can begin to revise both the user interface and the overall design of `MainFragment`.

The ultimate goal is use both views to show the **same contents** but presented in **two different formats**. We will use this principle in making the enhancements described below.

12.3.4 Enhancing `MainFragment`

The list of journals shown on the main screen currently provides only limited operation: the user can only select an item from the list to view its media contents. We will enhance the list in two aspects:

- Cosmetics enhancement: redesign the UI layout to use `CardView` and make it in compliance with the Material Design guidelines
- Functional enhancement: provide an "Edit" action to allow the user to edit the journal details

The overall structure of the new layout is shown in Listing 159 with the blueprints shown in Figure 12.9. In the blueprint the three text views, from top to bottom, are: name, location, and date. The "tall" design will be used for showing journal details in MainFragment and the "mini" layout in `MonthlyFragment`. Experiment with builtin text style to render these text fields in different font sizes. Notice that we use the "Borderless" style for the Edit button. We recommend that you keep the original layout (`journal_row`) and create two new ones.

The new design also requires an update to `JournalViewHolder`. The view holder currently binds the journal address (location) and date range to the same text view. Since the new design splits the location and data, the view holder needs a new TextView added to it. In addition, preparing for user edit action, we should add a Button as well.

Kotlin

```
// in JournalViewHolder.kt
/* more lines from existing code here */
val date: TextView
val editBtn: Button

init {
  /* more lines from existing code here */
  date = v.findViewByIt(R.id.date)
  editBtn = v.findViewByIt(R.id.edit_button)
}
```

Inside its `bindTo` function, the date can be initialized by simple string concatenation:

Kotlin

```
// in bindTo of JournalViewHolder
date.text = d.startDate + " to " + d.endDate
```

However, Android linter recommends that hard coded string concatenation be replaced with string resources. This linter warning actually facilitates our initial effort in making Traxy multilingual. For this specific task, the string resource should contain two placeholders (one for each date string).

1. Open `res/values/strings.xml`
2. Add the following string resource:

   XML

   ```
   <string name="date_range">%1$s to %2$s</string>
   ```

   Within the string definition text, `%1$` and `%2$` are the two placeholders. The suffix "`s`" indicates that the argument should be treated as a string. The detail syntax of the format string is adopted from `java.util.Formatter` and they are not Android specific.

With the string resource defined, initializing the date range above can be written as:

Kotlin

```
// in bindTo of JournalViewHolder
date.text = parentView.context.getString(R.string.date_range,
  d.startDate.substring(0, 10),
  d.endDate.substring(0, 10))
```

There are at least two different packages that provide card-based layout in Android: `androidx.cardview.widget` and `com.google.android.material`. We will be using the former by adding the following dependency:

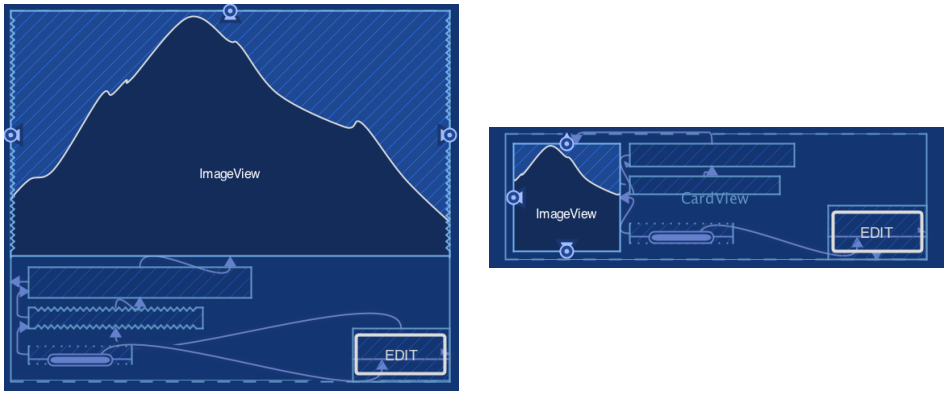
Gradle

```
implementation "androidx.cardview:cardview:1.0.0"
```

XML

```
<!-- journal_card_tall.xml or journal_card_mini.xml -->
<androidx.cardview.widget.CardView>
  <androidx.constraintlayout.widget.ConstraintLayout>
    <ImageView android:id="@+id/photo" ... />
    <TextView android:id="@+id/name" ... />
    <TextView android:id="@+id/location" ... />
    <TextView android:id="@+id/date" ... />
    <Button android:id="@+id/edit_button"
      style="@style/Widget.AppCompat.Button.Borderless" .../>
  </androidx.constraintlayout.widget.ConstraintLayout>
</androidx.cardview.widget.CardView>
```

Listing 159: Updated XML Layout of Journal List Item



(a) Tall (b) Mini

Figure 12.9: Blueprint of Journal Item Layout

Both `MainFragment` and `MonthlyFragment` show the list from the same data source using one *common* adapter, but *two different* layout files: `journal_card_tall` (for the standalone list) and `journal_card_mini` (for the list under the calendar). In the "mini" layout, we set the image

to 96×96dp. To share *only one adapter* for rendering both layouts, the names of all the widgets must be kept the same across both layout files (`photo`, `name`, etc.). Essentially, these two layout files differ only in the layout constraints for arranging the widgets.

Recall that `MainFragment` depends on `JournalAdapter` to render the list. Ideally, we want `MonthlyFragment` to reuse the same adapter, but with a different layout. To allow switching layout files at runtime, the desired layout can be provided as another argument to the adapter.

Let's edit `JournalAdapter` and make the following changes

1. Add an integer parameter to the primary constructor and annotate it as a layout resource. Change the class header from

```
Kotlin
class JournalAdapter(val listener: ((Journal) -> Unit)? = null) : ListAdapter<...>
```

   to

```
Kotlin
class JournalAdapter(@LayoutRes val layoutId: Int,
                     val listener: ((Journal) -> Unit)? = null) : ListAdapter<...>
```

   The `@LayoutRes` annotation prevents you from supplying arbitrary integers not representing a valid layout file.

2. Modify its `onCreateViewHolder()` to inflate the resource layout passed to the constructor in place of the hardcoded layout name `R.layout.journal_row`):

```
Kotlin
when (viewType) {
  JOURNAL_TYPE -> {
    val view = LayoutInflater.from(parent.context)
        .inflate(layoutId, parent, false)    // Updated line
    return JournalViewHolder(view)
  }
  /* more code hidden */
}
```

This update creates a cascading effect to the underlying adapter called in `onViewCreated` of `MainFragment` and you should replace

```
Kotlin
// in onViewCreated of MainFragment.kt
adapter = JournalAdapter {
  /* listener lambda expression here */
}
```

to

```
Kotlin
  // in onViewCreated of MainFragment.kt
  adapter = JournalAdapter(R.layout.journal_card_tall) {
  /* listener lambda expression here */
}
```

Rerun your app and verify that everything in MainFragment works as expected.

12.3.5 Completing `MonthlyFragment`

For now, most of the details in `MonthLyFragment` will be 80% similar to `MainFragment`. Edit the layout file `fragment_monthly.xml` to resemble the appearance of `fragment_main` Most importantly it should define a RecyclerView whose id is `journal_list`. Our (list) adapter depends on the fact the recycler view in both fragments share a common name.

To complete the Kotlin class, practically you can copy the following two functions from `MainFragment` to `MonthLyFragment`:

- `onViewCreated`: just remember to replace `R.layout.journal_card_full` to `R.layout.journal_card_mini` when instantiating the `JournalAdapter`
- `onActivityCreated`

These updates will trigger compile errors due to missing variable declarations and imports, but you should be able to fix them by referring to `MainFragment`.

Rerun your app and verify that swiping across the two views you see a different layout to present the list of journals ("tall" layout in MainFragment and "mini" layout in MonthLyFragment).

12.3.6 Adding Calendar

The other main component missing from `MonthlyFragment` is a calendar view. We potentially can reuse the same `CalendarView` as we did in an earlier chapter. However, we need a calendar viewer that allows us capture the "month change" events. By listening to these events, the app will be update to update the monthly journal list for travel activities taking place only in the selected month. And the builtin CalendarView does not support this feature.

An alternative to the builtin widget we can use `MaterialCalendarView` provided by the following library written in Kotlin:

Gradle
```
implementation "com.applandeo:material-calendar-view:1.7.0"
```

Your first step is to update the layout of `MonthLyFragment` to include an instance of the material calendar view:

XML
```
<LinearLayout android:orientation="vertical" ...>
  <com.applandeo.materialcalendarview.CalendarView
      android:id="@+id/calendar"
      android:layout_width="match_parent"
      android:layout_height="wrap_content" />

  <androidx.recyclerview.widget.RecyclerView
      android:id="@+id/journal_list"
      android:layout_weight="1"
      ... />
</LinearLayout>
```

Journal entries may possibly cross month boundaries (such as thus those begin within the last week of a month and continue into the first week of the following month). To filter such data correctly, it would be easier to use the calculate the intersection between two date intervals:

- The first interval spans the first day until the last day of the selected month
- The second interval spans the duration of a journal.

When the intersection between the two is not empty, the journal takes place within the selected month.

We will be using the JodaTime to implement the journal filtering as described above use its `Interval` class. By default, Joda intervals are half-opened: its beginning date is inclusive but its ending date is exclusive. You can add necessary code so `MonthlyFragment` displays only journals in the given month. This code will be written in the following "month changed listener" of the Material CalendarView

- `OnForwardPageChangeListener` when the calender view proceeds to the next month
- `OnPreviousPageChangeListener` when the calender view proceeds to the previous month

These listeners can be setup in in `MonthlyFragment` as follows:

Kotlin

```kotlin
// in MonthlyFragment.kt
val monthChangeListener = OnCalendarPageChangeListener  {
  viewModel.remoteJournals.value?.let {
    adapter.submitList(filteredByCurrentMonth(it))
  }
}

override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
  // other irrelevant code not shown
  calendar.setOnForwardPageChangeListener (monthChangeListener)
  calendar.setOnPreviousPageChangeListener (monthChangeListener)
}

override fun onActivityCreated(savedInstanceState: Bundle?) {
  super.onActivityCreated(savedInstanceState)
  viewModel.remoteJournals.observe(this.viewLifecycleOwner, Observer {
      adapter.submitList(filteredByCurrentMonth(it))
  })
}
```

The function `filteredByCurrentMonth` is responsible for feeding the adapter with the filtered journal data based on the selected month:

Kotlin

```kotlin
// in MonthLyFragment.kt
private fun filteredByCurrentMonth(input: List<Journal>): List<Journal> {
  // Step 1: get the current month
  val thisMonth = DateTime(calendar.currentPageDate)
  // Step 2: create the date interval representing the current month
  val d1 = thisMonth.dayOfMonth().dateTime
  val d2 = thisMonth.plusMonths(1).dayOfMonth().dateTime
  val range = Interval(d1, d2)
  // Step 3: filter the input based of its date range
  return input.filter {
    val jStart = DateTime(it.startDate)
    val jEnd = DateTime(it.endDate)
    jStart.isBefore(jEnd) && Interval(jStart, jEnd).overlaps(range)
  }
```

12.3.7 Handling the Edit Button

In either view of the main screen ("all journals" or "monthly journal") the user can perform two actions: edit an existing journal or create a new one. To distinguish the two user action we can use a similar trick used earlier in designing `JournalMediaAdapter`, which is adding a second (string) argument to the listener function. Using a similar approach, our next step is to change the signature of `JournalAdapter` from:

Kotlin
```
class JournalAdapter(@LayoutRes val layoutId: Int,
                     val listener: ((Journal) -> Unit)? = null) : ListAdapter<...>
```

to

Kotlin
```
class JournalAdapter(@LayoutRes val layoutId: Int,
                     val listener: ((Journal, String) -> Unit)? = null) : ListAdapter<...>
```

The above update will cascade the associated view holder class and you should change the signature of `bindTo` from

Kotlin
```
// in JournalViewHolder.kt
public fun bindTo(d: Journal, listener: ((Journal) -> Unit)?) {
```

to

Kotlin
```
public fun bindTo(d: Journal, listener: ((Journal, String) -> Unit)?) {
```

The "Edit" action is handled by adding another click listener in the `bindTo` function above:

Kotlin
```
// in bindTo() of JournalViewAdapter.kt
if (listener != null) {
  parentView.setOnClickListener { //  existing click listener
      listener(d, "VIEW")
  }
  editBtn.setOnClickListener {    // new click listener
      listener(d, "EDIT")
  }
}
```

Ideally, the new "EDIT" action requires a new destination in our navigation graph. However, since adding a new journal and editing an existing journal prompt the user to enter (almost) the same information we should be able to reuse the existing `NewJournalFragment` as the same destination for both tasks. Recall the this fragment was launched when the user taps on the "Add" fab from the main screen. With this strategy chosen, it seems appropriate to rename this fragment to `JournalEditFragment`. Let's make the required changes in the navigation graph to add this feature to your app

1. Use Android Refactor
   Rightarrow Rename menu to rename `NewJournalFragment` to `JournalEditFragment`

2. Rename the action arrow originating from `JournalPagerFragment` to `JournalEditFragment` from `action_new_journal` to `action_edit_journal` from `PagerFragment` to `JournalEditFragment`. *You may have to edit the XML code of the navigation graph directly. Using the attribute panel does not seem to work.*

3. Rebuild your app so the `xxxxDirections` classed are regenerated by the Gradle navigation plugin

Some of you may raise the question "How do we distinguish the user intention upon entry to `JournalEditFragment`. We can answer this question in at least two different ways:

- We can pass an extra navigation argument (using the SafeArgs plugin) to inform the fragment whether we are editing or inserting a new journal
- We can check the current `selectedJournal` property in `MediaViewModel` class. When this property is null then we are adding a new journal, otherwise we are editing an existing one.

Out of the two approaches above, we would prefer the second approach and use the extra navigation argument to set the title to show on the app bar. Passing a title text requires the following update in the navigation graph to add the extra argument `fragment_title` below. At the time of this writing, Android Studio has no support to edit existing action arrows so you have to the edit the XML code directly.

XML

```xml
<fragment
  android:id="@+id/journalEditFragment"
  android:name="edu.gvsu.cis.traxy.JournalEditFragment"
  android:label="{fragment_title}"
  tools:layout="@layout/fragment_edit_journal">
  <argument
      android:name="fragment_title"
      app:argType="string" />
</fragment>
```

However to make this works, we should reset the said property prior to navigating into JournalEditFragment to insert a new journal. The proper place to do this would be in the click listener of the FAB in `MainFragment`:

Kotlin

```kotlin
// in onViewCreated of MainFragment.kt
fab_add.setOnClickListener {
  mediaModel.selectedJournal.value = null     // new line added
  val action = JournalPagerFragmentDirections.actionEditJournal("New Journal")
  findNavController().navigate(action)
}
```

With the improved functionality of `JournalAdapter` the listener lambda expression passed from `MainFragment` and `MonthlyFragment` require minor updates as well:

1. Add two explicitly name arguments to the lambda expression (in place of the implicit Kotlin `it`)

Kotlin
```
adapter = JournalAdapter(R.layout.journal_card_tall) { jData, action ->
  // more code not shown
}
```

2. Create the proper action depending on the chosen action by replacing this snippet

Kotlin
```
val action = JournalPagerFragmentDirections.actionToMediaList(jData.name)
findNavController().navigate(action)
```

to

Kotlin
```
val destination = if (action == "VIEW")
    JournalPagerFragmentDirections.actionToMediaList(jData.name)
  else
    JournalPagerFragmentDirections.actionEditJournal(jData.name)
findNavController().navigate(destination)
```

Either action will take the user to a new screen as shown in Figure 12.10. The two screens look very similar except for the list of photos from media entries associated with the selected journal. The intention is to allow the user selects one of them as a cover photo. The list will look more pleasing when the photos are presented in multi-column format.

After learning more about using a better calendar view, let's now revise the UI design of `JournalEditFragment` to use the same component:

1. Edit the XML layout of `JournalEditFragment` and replace the builtin `CalendarView` with the material variant. Be sure to keep the same ID (`trip_calendar`)

2. Specify the type of the new calendar view as a date range picker:

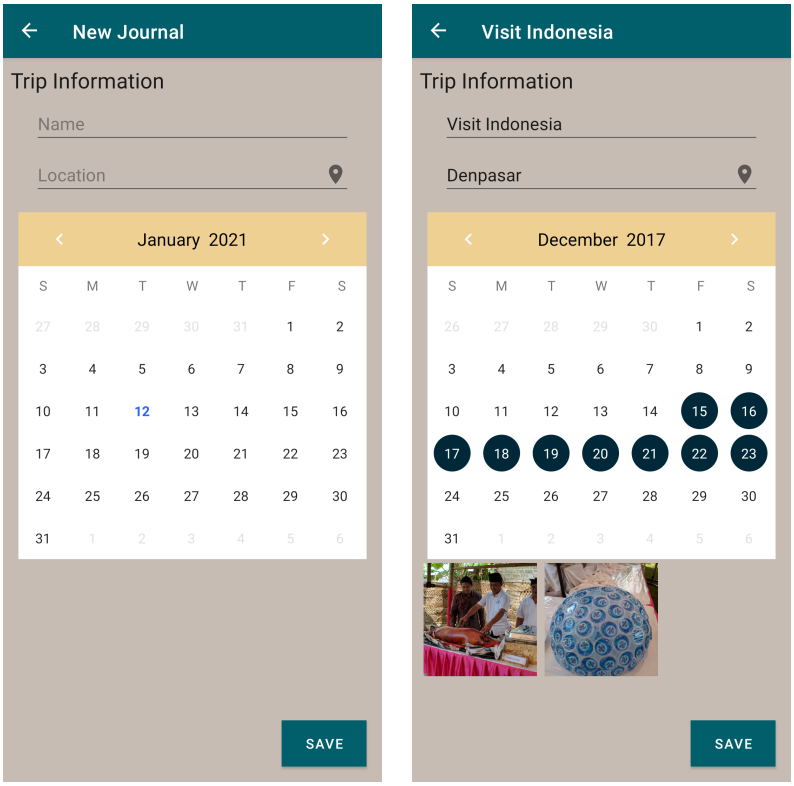
XML
```
<com.applandeo.materialcalendarview.CalendarView
  android:id="@+id/trip_calendar"
  app:type="range_picker" ... />
```

3. If you prefer, you can also customize its color to match our current theme

XML
```
<com.applandeo.materialcalendarview.CalendarView
  android:id="@+id/trip_calendar"
  app:headerColor="@color/colorAccent"
  app:headerLabelColor="#000000"
  app:selectionColor="@color/colorPrimaryDark"
  ... />
```

4. Delete the TextView for showing the date range. Since the new material calendar view automatically highlight the selected date range, the user sees adequate visual feedback directly from the calendar.

Updating the calender view component will also force you to update the associated source code pertinent to the calendar view. For instance, the date selection handler must be replaced format



(a) Create (b) Edit

Figure 12.10: Create New or Edit Existing Journal

Kotlin

```kotlin
// onViewCreated of JournalEditFragment.kt
trip_calendar.setOnDateChangeListener { , yr, mon, dy ->
  // more code not shown
}
```

to

Kotlin

```
trip_calendar.setOnDayClickListener {
  val current = DateTime(it.calendar)
  val range = trip_calendar.selectedDates
  if (range.size > 0) {  // do we have previous date selected?
    val last = DateTime(range[0])
    if (current.isBefore(last)) {
      startDate = current
      endDate = last
    } else {
      startDate = last
      endDate = current
    }
  }
}
```

Configured as a date range picker, Material CalendarView holds the multiple date selections in two properties:

- `calendar` holds the most recent date selection
- `range` a Java List that holds the "historical" date selections

Upon entry to `JournalEditFragment` the fragment should inspect the `selectedJournal` property and then populate the UI accordingly:

Kotlin

```
// in onResume of JournalEditFragment.kt
if (mediaModel.selectedJournal.value == null) {
  isEditing = false
}
else {
  isEditing = true
  mediaModel.selectedJournal.value?.let {
    trip_name.setText(it.name)
    trip_location.setText(it.address)
    trip_calendar.setDate(start)

    // Build a date range to highlight the calendar cells
    val start = DateTime(it.startDate).minusDays(1).toGregorianCalendar()
    val end = DateTime(it.endDate).plusDays(1).toGregorianCalendar()
    val dateRange = CalendarUtils.getDatesRange(start, end)
    trip_calendar.setSelectedDates(dateRange)
  }
}
```

The above code snippet definitely requires you declare `mediaModel` as you did in the other classes. When the user is adding a new journal, there will be no media files to show yet in which case we should hide the photo list from the UI. On the contrary, when editing an existing journal the user should be presented with the details of the selected journal.

The date range produced by `CalendarUtils` seems to be exclusive, i.e. the two end dates are excluded from the range. To compensate for the excluded dates we extend the date range by calling `minusDays(1)` and `plusDays(1)`.

12.3.8 Querying and Displaying Photos

Our intention is to show all the photos recorded under the current journal in a multi-column format below the calendar view. However, the available screen height is too narrow. To allow the user to comfortable browse the photos, we should allow the calendar to "disappear" when the user brings the list of photos into view. Android UI widgets provide several containers for implementation this feature: `ScrollView`, `HorizontalScrollView`, and `NestedScrollView`. Since one of the child widgets in this container is a RecyclerView which **also** need to scroll, the former two containers will not work and that forces us to use `NestedScrollView`.

Let's make the following UI changes to the layout of `JournalEditFragment`:

1. Drag a `NestedScrollView` into the component tree. By default, Android Studio creates a `LinearLayout` as a child of NestedScrollView
2. Move the calendar so it becomes the child of the LinearLayout
3. Drag a RecyclerView to be the second child of the LinearLayout, name is `photo_list`

The hierarchical structure of the updated XML layout should look like the following:

XML

```
<androidx.constraintlayout.widget.ConstraintLayout ...>
  <TextView android:id="@+id/info_label"... />
  <EditText android:id="@+id/trip_name" ... />
  <EditText android:id="@+id/trip_location" ... />

  <androidx.core.widget.NestedScrollView ...>
    <LinearLayout android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:orientation="vertical">

      <com.applandeo.materialcalendarview.CalendarView ...
        android:id="@+id/trip_calendar" ... />
      <androidx.recyclerview.widget.RecyclerView ...
        android:id="@+id/photo_list"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_weight="1" />
    </LinearLayout>
  </androidx.core.widget.NestedScrollView>

  <Button android:id="@+id/add_button" android:text="Save"... />
</androidx.constraintlayout.widget.ConstraintLayout>
```

The RecyclerView that we use here requires its own adapter, view holder, and XML layout. More over it is essentially similar to what we use in `JournalViewFragment` for showing journal media files, except this one should show photos only. So you can use those files as a reference to create new ones below.

- The layout of each individual photo will be controlled by `photo_item.xml`. In addition to an ImageView for showing the actual photo from Firebase Storage, the layout will also include an "check mark" icon as a visual cue when the user selects this photo as a cover for the journal.

XML
```
<androidx.constraintlayout.widget.ConstraintLayout ...>
  <ImageView
    android:id="@+id/photo"
    android:layout_width="0dp"
    android:layout_height="0dp"
    android:scaleType="centerCrop"
    app:layout_constraintDimensionRatio="W,1:1" ... />

  <ImageView
    android:id="@+id/selected"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:visibility="invisible"
    app:srcCompat="@drawable/ic_baseline_done_24" ... />
</androidx.constraintlayout.widget.ConstraintLayout>
```

The first ImageView (`photo`) is constrained with an aspect ratio 1:1 and its height should be scaled to match its fixed width. The second ImageView (`selected`) is a small icon constrained to the bottom right of the parent view.

- The view holder `PhotoViewHolder` mimics the functionality of `JournalMediaViewHolder`, except the former is much simpler In its `bindTo` function, use the Glide library to fetch the image from Firebase Storage into `photo`
- `PhotoAdapter` is almost a line-by-line copy of `JournalMediaAdapter`

In `JournalViewFragment` you have the following code snippet for showing all the media files:

Kotlin
```
// in onViewCreated of JournalViewFragment.kt
mediaModel.selectedJournal.observe(this.viewLifecycleOwner, Observer {
  val option = FirestoreRecyclerOptions.Builder<JournalMedia>()
    .setLifecycleOwner(viewLifecycleOwner)
    .setQuery(mediaModel.mediaQuery(), JournalMedia::class.java)
    .build()
  adapter = JournalMediaAdapter(option) { media: JournalMedia, action: String ->
    // details not shown
  }
  media_list.adapter = adapter
  media_list.layoutManager = LinearLayoutManager(requireContext())
})
```

The function `mediaQuery()` returns a list of all the media files associated with currently selected journal. To show the list of photos in `JournalEditFragment` you can copy the above code snippet almost line-by-line with a few minor modifications as described below:

- The Firestore query returned by `mediaQuery()` requires an additional condition

Kotlin
```
whereEqualTo("type", MediaType.PHOTO.ordinal)
```

Essentially, this is akin to the SQL condition "`WHERE type == PHOTO`"

- The layout manager is a `GridLayoutManager` with a 3-column format (as opposed to Linear)

Kotlin
```
// in onViewCreated of JournalEditFragment.kt
mediaModel.selectedJournal.observe(viewLifecycleOwner, Observer {
  val photoQuery = mediaModel.mediaQuery()
    .whereEqualTo("type", MediaType.PHOTO.ordinal)           // new condition added
  val option = FirestoreRecyclerOptions.Builder<JournalMedia>()
    .setLifecycleOwner(viewLifecycleOwner)
    .setQuery(photoQuery, JournalMedia::class.java)
    .build()
  photo_list.adapter = PhotoAdapter(option)
  // Use 3-column grid format
  photo_list.layoutManager = GridLayoutManager(requireContext(), 3)
})
```

12.3.9 Selecting Cover Photo

To allow the user select a photo as the journal cover, we must supply a listener closure to PhotoAdapter, similar to a technique we used earlier for the other adapters, especially JournalMediaAdapter. Let's first modify `PhotoAdapter`

1. Change the class header from

Kotlin
```
class PhotoAdapter(
  opt: FirestoreRecyclerOptions<JournalMedia>) : FirestoreRecyclerAdapter<...>
```

   to

Kotlin
```
class PhotoAdapter(
  opt: FirestoreRecyclerOptions<JournalMedia>,
  val listener: (JournalMedia) -> Unit,    // function expression
) : FirestoreRecyclerAdapter<...>
```

2. Pass the listener as a second argument to `bindTo`

Kotlin
```
override fun onBindViewHolder(holder: PhotoViewHolder, position: Int,
    model: JournalMedia) {
  holder.bindTo(model, listener);
}
```

   Next, we have to modify `bindTo` in `PhotoViewHolder` from:

Kotlin
```
public fun bindTo(m: JournalMedia) {
  Glide.with(view).load(m.url).into(photo)
}
```

to:

Kotlin
```
public fun bindTo(m: JournalMedia, listener: ((JournalMedia) -> Unit)?) {
  Glide.with(view).load(m.url).into(photo)
  if (listener != null) {
    photo.setOnClickListener {
      listener(m)
    }
  }
}
```

The above modifications cause a compile error in `JournalEditFragment` because now you are required to supply a function lambda expression when creating `PhotoAdapter`. The current code looks like the following:

Kotlin
```
// in onViewCreated of JournalEditFragment.kt
mediaModel.selectedJournal.observe(....) {
  // more code not shown
  photo_list.adapter = PhotoAdapter()
}
```

In the updated code you must append a function lambda as the "argument" to the invocation of `PhotoAdapter`:

Kotlin
```
// in onViewCreated of JournalEditFragment.kt
mediaModel.selectedJournal.observe(....) {
  // more code not shown
  photo_list.adapter = PhotoAdapter() {
    // function lambda expression that will get called when a photo is selected
  }
}
```

To provide visual feedback when the user selects a cover photo, we must show (and hide) the check mark icon associated with each photo in the list. There are several solutions to accomplish this task. The first is to write additional logic in the "photo click listener" inside `JournalEditFragment`, This seems to be the most natural choice for many developers. However, the code to shod/hide the check mark icon requires access to data hidden in the adapter and view holder classes. So, this choice is not actually the best.

The second alternative is to write the logic directly within the adapter class. This technique is a better option because the adapter has direct access to **individual** view holders which store the actual instance of icons that we are going to show/hide. If we can keep track which view holder instance was selected by the user for cover photo, we can "inject" additional statement when the click listener is invoked. So the issue is where do we write the injected code? Fortunately, the syntax of Kotlin lambda expressions allows us to accomplish this task relatively easily. By replacing the `bindTo` call from

Kotlin
```
// in PhotoAdapter.kt
override fun onBindViewHolder(holder: PhotoViewHolder, position: Int,
  model: JournalMedia)
{
  holder.bindTo(model, listener)
}
```

to

Kotlin
```
override fun onBindViewHolder(holder: PhotoViewHolder, position: Int,
  model: JournalMedia)
{
  holder.bindTo(model) {
    // injected code goes here (shown below)
    listener(it)
  }
}
```

The injected code must perform the following tasks:

1. Show the check mark associated with the selected photo
2. Hide the previous check mark (in case the user selects another one earlier)

Hiding the previous check mark requires memorization of the corresponding holder which can be done by declaring a nullable instance variable below:

Kotlin
```
// in PhotoAdapter.kt
var selectedForCover: PhotoViewHolder? = null
```

Finally, the complete code should look like:

Kotlin
```
 override fun onBindViewHolder(holder: PhotoViewHolder, position: Int,
  model: JournalMedia)
{
   holder.bindTo(model) {
     selectedForCover?.run { selected.visibility = View.INVISIBLE }  // Hide the old icon
     selectedForCover = holder
     holder.selected.visibility = View.visible       // un-hide the new icon
     listener(it)
   }
 }
```

To record the user selection of cover photo, we have to add a new field in our data class `Journal`. Go ahead modify `Journal.kt` and add a nullable string `coverPhotoUrl`

Kotlin
```
data class Journal(
  @DocumentId
  var key: String = "",
  var name: String = "",
  var startDate:String = "", var endDate: String = "",
  var address: String = "",
  var placeId: String = "",
  var lat:Double = 0.0, val lng:Double = 0.0,
  var coverPhotoUrl: String? = null         // new field
) : ListItem() {}
```

With the new field in place, we can store the URL of the selected photo when the user clicks on a particular photo. Locate the following code snippet in `onViewCreated`

Kotlin
```
// in onViewCreated of JournalEditFragment.kt
mediaModel.selectedJournal.observe(....) {
  // more code not shown
  photo_list.adapter = PhotoAdapter() {
    // function lambda expression that will get called when a photo is selected
  }
}
```

and complete the function lambda expression as follows:

Kotlin
```
// in onViewCreated of JournalEditFragment.kt
mediaModel.selectedJournal.observe(....) {
  // more code not shown
  photo_list.adapter = PhotoAdapter() {
    // function lambda expression that will get called when a photo is selected
    mediaModel.selectedJournal.value?.apply {
      coverPhotoUrl = it.url
    }
  }
}
```

The code for handling the "Save" button should now distinguish the two different operations based on the value of the `isEditing` flag:

Kotlin
```
add_button.setOnClickListener {
  if (isEditing) {
    mediaModel.selectedJournal.value?.let {
      viewModel.updateJournal(it)
    }
  }
  else {
    viewModel.addJournal(Journal(key = "key-???",
      name = trip_name.text.toString(),
      address = trip_location.text.toString(),
      startDate = startDate?.toString()!!,
      endDate = endDate?.toString()!!))
  }
  findNavController().popBackStack()
}
```

For the sake of not repeating the programming techniques that have been covered in the other chapters, we intentionally include only the details which are relevant to the main topics of this chapter, i.e. using tabs and grid layout. Other details such as saving the edited data into Firestore is left as an exercise for you. We would recommend that you first try to implement the missing details yourself before looking up for our solution in the code repository.

12.3.10 Calendar Decorator

If you notice the details of Figure **??**b, the calendar view shows dots on days of the trips within a particular month. This extra embellishments are rendered as `EventDat` and can be incorporated in the calendar page change listener which initially shows only a minimal amount of code:

Kotlin

```kotlin
val monthChangeListener = OnCalendarPageChangeListener {
  viewModel.remoteJournals?.let {
    adapter.submitList(filteredByCurrentMonth(it))
  }
}
```

Tp render visual markers on the days of all the trips within a given month we have to extract the start and end dates of each trip, and create a marker of each day of the trip. This overall process can be viewed as an N-to-M mapping where $M \geq N$. For instance, 3 trips in a month may span across a total of 8 days ($N = 3, M = 8$). In our case (a list of) N journals should produce (a list of) M event days. The Kotlin standard library provides several operations for manipulating *collections*: filter, zip, map, partition, flatmap, join, etc. Of the map operations available to us, we do want to use "flatmap" because for each input item (Journal) we must produces multiple output items (EventDay). In the function body of `flatMap` the while loop enumerates the each day of the trip, create an event, and store it to a mutable set. Finally we call `setEvents` to render all the markers on the calendar view.

Kotlin

```kotlin
val monthChangeListener = OnCalendarPageChangeListener {
  viewModel.remoteJournals.value?.let {
    val trips = filteredByCurrentMonth(it)
    adapter.submitList(trips)
    val markers =  trips.flatMap {
      var curr = DateTime(it.startDate).withTimeAtStartOfDay()
      val end = DateTime(it.endDate).withTimeAtStartOfDay()
      val events: MutableSet<EventDay> = mutableSetOf()

      while (!curr.isAfter(end)) {
        val cal = curr.toCalendar(Locale.getDefault())
        val evt = EventDay(cal, R.drawable.event_icon)
        events.add(evt)
        curr = curr.plusDays(1)
      }
      events
    }
    calendar.setEvents(markers)
  }
}
```

The icon used for markers is just an 8-pt dot drawn from a drawable resource file defined below:

XML

```xml
<!-- in res/drawable/event_icon.xml -->
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android"
    android:shape="oval">
    <solid android:color="@color/colorPrimaryDark" />
    <size
        android:width="8dp"
        android:height="8dp" />
</shape>
```

12.3.11 Bottom Navigation

Unlike iOS `UITabController`, Android places the tabs at the top of the screen. However, Android Nougat 7.1 (API Level 25) introduces the `BottomNavigationView` that allows Android developers create similar UI designs as used by iOS.

> **NOTE**
>
> The Material Design guidelines[a] recommend use of bottom navigation in mobile apps for the following use cases:
>
> - Top-level destinations that need to be accessible from anywhere in the app
> - Three to five destinations
>
> The example we use below does not meet the criteria as "top-level" destinations, but we cover the technique for the sake of completeness of coverage around app navigation in Android.
>
> [a] `https://material.io/components/bottom-navigation#usage`

Android bottom navigation is designed to work seamlessly with both ViewPager2 and Jetpack Navigation. Since we already included ViewPager2 in the layout, we will keep it and add necessary code for BottomNavigationView to work with it.

The very first step is to edit the layout file of `PagerFragment`:

1. Remove `TabLayout`
2. Insert `BottomNavigationView` below the ViewPager2 call it `bottom_nav`

Despite the widget name that starts with "Bottom", it bears no semantic significance. It will not automatically place itself at the bottom of your UI, you have to manually arrange its position. Each "tab" in the bottom navigation view is represented by a menu item. The overall structure of structure of the updated layout is shown in Listing 160.

In addition, for UI cosmetics you also want to set the following color attributes on the bottom navigation: `android:background`, `app:itemBackground`, `app:itemTextColor`, and `app:itemIconTint`. Be sure to set the two background attributes to the same value to create the illusion that the navigation bar fills the entire screen width.

You also notice that `BottomNavigationView` takes a menu attribute. Go ahead create the menu `menu_bottom_nav`. It follows the same structure as the other menu you used for the app bar. For each menu item, be sure to set the following attributes

- `android:id` for menu/navigation handling logic in code
- `android:title` for displaying the tab name
- `android:icon` for the accompanying icon above the title text (optional)

XML

```xml
<!-- in fragment_pager.xml  -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical"
  tools:context=".JournalPagerFragment">

  <androidx.viewpager2.widget.ViewPager2
      android:id="@+id/pager"
      android:layout_width="match_parent"
      android:layout_height="match_parent"
      android:layout_weight="1" />

  <com.google.android.material.bottomnavigation.BottomNavigationView
    android:id="@+id/bottom_nav"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      app:menu="@menu/menu_bottom_nav" />
</LinearLayout>
```

Listing 160: Using `BottomNavigationView` with `ViewPager`

The code snippet bellow assumes that the two tabs have the following id and title:

android:id	android:title
all_journals	Journals
monthly_journals	Calendar

Synchronizing BottomNavigation and ViewPager

`TabLayout` and `ViewPager2` can be synchronized using only one simple method call. On the contrary, synchronizing bottom navigation and view pager requires more lines of code across the following two listener interfaces:

1. `ViewPager2.OnPageChangeCallback` to handle user swipes on the view pager

2. `BottomNavigationView.onNavigationItemSelectedListener` to handle bottom menu or tab selection

Basically, the listener of one widget must update the state of the other, and vice versa. Let's begin with the bottom navigation listener. Since the interface defines only one method, you can write it as a Java lambda expression. The input parameter is the selected menu on the bottom navigation. The action in the switch statement updates the view pager state to select its current page.

Kotlin
```
// in onViewCreated of JournalPagerFragment.kt
bottom_nav.setOnNavigationItemSelectedListener {
  when(it.itemId) {
      R.id.all_journals -> pager.setCurrentItem(0, true)
      R.id.monthly_journals -> pager.setCurrentItem(1, true)
  }
  true
}
```

NOTE

Notice that the "when labels" above must match exactly the `android:id` used in the menu of the bottom navigation.

The other listener defines three methods, but you need to write your code only in one of the three. The `onPageSelected` method simply select/unselect the menu items on the bottom navigation. Let's first defined the object implementing the OnPageChangeCallback interface:

Kotlin
```
val pageChangeCallback  =  object:ViewPager2.OnPageChangeCallback(){
  override fun onPageSelected(position: Int) {
      val tab = bottom_nav.menu.getItem(position)
      tab.setChecked(true)
  }
}
```

NOTE

If you were to rewrite the above snippet in Java, you must first define a class that implements the interface `OnPageChangeCallback` before you can instantiate an object from the class. Kotlin `object` syntax makes the process look very concise.

After the callback object is defined, you can then register it for the view pager:

Kotlin
```
// in onViewCreated of JournalPagerFragment.kt
pager.registerOnPageChangeCallback (pageChangeCallback)
```

As final note of this topic, it was mentioned earlier in this section that BottomNavigationView also works with Jetpack navigation framework. It requires even fewer lines of Kotlin code to integrate the two. The major improvement over ViewPager2 is that 2-way event synchronization between BottomNavigationView and Jetpack navigation is implemented automatically by a single function call and you don't have to provide the two listeners as described above.

Congratulations! It's time to pat yourself in your back. With your new skills of using tabs, collections, and list with grid layout, you now have the practical knowledge of writing apps whose design calls for multiple forms of screen navigation.

—If all the ways I have been along were marked on a map and joined up with a line, it might represent a minotaur.

Pablo Picasso

13 Working With Location and Maps

Chapter Objectives

After reading this chapter you should be able to

- Understand how to write location-based apps.
- Understand how to display and interact with maps.

13.1 What's There To Do Around Here?

When you are on an excursion to a place of interest, you also want to know other interesting places nearby. Your main destination may be the Eyjfjallajökull volcano in Iceland or Platja d'en Bossa beach in Balearic Islands, Spain, but you don't stay on the beach all day, you also want to immerse in the hustle and bustle of the community, talk to people, learn about the culture, taste their authentic food, etc.

All information relevant to your place of interest can be useful when you are to decide where (not) to visit. A number of factors will affect your decision making process. Is the place safe? Is it easy to get there? What do others say about their food or services? Is it expensive? This list may continue. Most travelers will visit popular travel web sites and look for specific sections like "Things To Do", "Places to Stay", or "Reviews" to learn more about their travel destination.

As a mobile app developer, you may be looking to incorporate these sources of information into an app. There are many options to incorporate such sources of information into an app. For example, web services like Qype, Trippin'in, Gogobot, Rezgo, Contiki make their APIs accessible to programmers. Using the skill you acquire in Chapter 11 (Networking) you should be able to incorporate these services into your app without much difficulty. However, in this chapter we

will be incorporating a *small portion* of the Google Places APIs to enhance our Traxy app. Within Google Places APIs, a "place" holds semantically more interesting information than just its physical location (latitude, longitude, and altitude) such as:

- phone number
- URL to the place's web site (for instance if the place is a business)
- Type of place (church, cafe, school, train station, etc.)
- Price level
- Rating of user reviews

We emphasize the qualitative size of what is covered in this chapter to let the readers aware that the Google Places APIs actually comprise several more specialized services:

- Current Place Detection: discovers the place where the device is currently located.
- Place Picker: presents a map view dialog to select a place of interest
- Place Autocomplete: returns place predictions in response to user queries for place names
- Place Details: returns detailed information about a specific place
- Place Photos: returns photos of a specific place
- Place Add: adds a place to Google's Places database
- Place Report: creates a place report to confirm that your device is currently at a particular place.

At Google I/O 2016, Google announced a new set of APIs, collectively known as the (Context) Awareness APIs that subsume the Google Places APIs under a bigger umbrella. The Awareness APIs concern not only the current *location* of a device but also its (current) *context*

- Information pertinent to current location is provided by several APIs: Fused Location Provider, Geofencing, and Places APIs
- Information pertinent to current user context is provided by Activity Recognition API, Google Fit Platform, Sensors API
- Information pertinent to current device context is provided by Nearby APIs

You certainly can incorporate any of the above APIs of interest into your app, provide the necessary callback, and combine the results from each API *yourself*. However, there are cases where an API should only be enabled only on certain conditions. As a simple example, it does not make sense to activate Geofencing when the device is known to be sitting on your desk. The main goal of the Awareness APIs is to combine various sources of signals while preserving battery life.

The current version of Awareness APIs provide information about the following context:

- Activity (walking, running, biking, driving)

- Beacons (nearby devices)
- Headphones (plugged or unplugged)
- Location (latitude, longitude)
- Local Time
- Place ("Coffee Shop", "Grand Central Terminal")
- Weather (current temperature and conditions)

Instead of pulling individual context, your app will be using either the Fence API or Snapshot API. While Geofencing is purely location-oriented, Fence API provides semantically richer coverage. Geofencing allows you to setup an Intent that will be delivered to your app when the device is within a certain radius of a particular geolocation. On the contrary, Fence API lets you setup an Intent that gets delivered when, for instance, "you are near an ice cream shop when the temperature is at least 75°".

You can read more details about these APIs at the Awareness and Location APIs home page or watch the some of the informative Google I/O presentations on the topic [2] . There is also some excellent videos on Google's YouTube Developer channel on styling maps [7].

13.1.1 Location Sensing Technologies

In response to questions like "How *far* is the gas station?", you would sometimes reply "It's only *five minutes* from here", instead of responding with the actual distance. For average city traffic, that gas station is about 2–3 miles away. Astronomers measure distances between celestial objects in same fashion, they use "light years". The moon is 1.3 light seconds from the earth with actual distance of approximately 240,000 miles. Likewise, the sun is 500 light seconds from the earth with actual distance of 93 million miles.

Measuring distances using time of travel has several practical applications.

- You may have learned that using the difference between speed of light and sound, you can estimate the distance of a lighting from your current location by measuring the number of seconds elapsed until you hear its thunder.
- By listening to your own echo you may be able to estimate how far you are standing from a cliff across a canyon
- To measure the size of a room, home builders now use laser measuring devices for faster results
- The speed detectors used by law enforcement officers do not *measure* your car speed. These detectors actually listen to the echo of its own laser signal, they measure several distances to your car and then *calculate* its speed from these distance measurements.

Let's see how we can use time to pinpoint (*not measure*) position. Imagine you are traveling on Highway 46, between Gackle and Enderlin, North Dakota.[1] Refer to the maps in Figure 13.1.

The geo coordinates of these two cities are listed below.

[1] According to Google search result, this is the *longest straight* east-west road (120+ miles) in the US.

City	Lat	Lon
Gackle	46.624 080 3	−99.150 752 8
Enderlin	46.619 857 6	−97.609 13

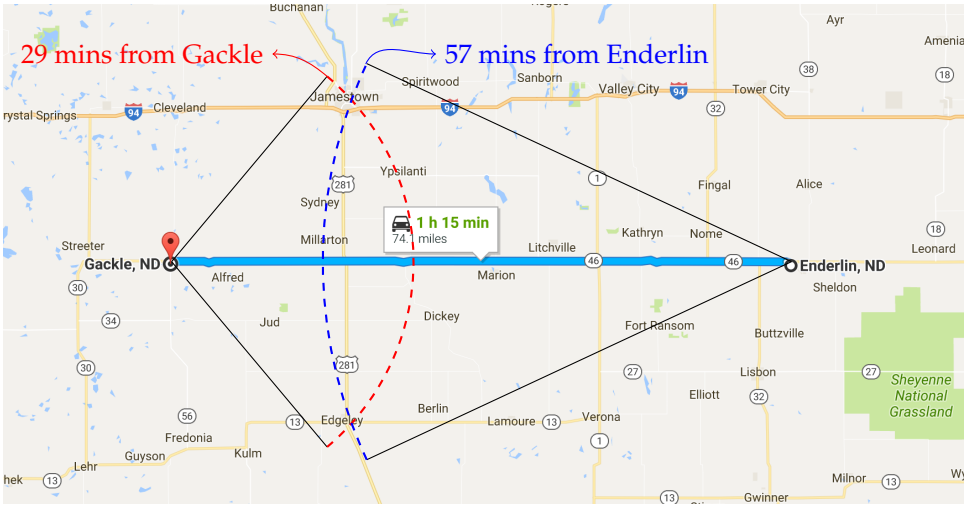


Figure 13.1: Time of Travel From Two Locations

Since the road between the two cities stretches along an east-west section of Highway 46, it would be very easy to figure out that the *latitude* of any location along this highway section would be a number around 46.62. But how do you determine its longitude? Well, for this very special case it is sufficient to know *only* the elapse time from one of the cities (Gackle or Enderline) and the longitude of that place can be calculated using interpolation. In Figure 13.1, for instance, 29 minutes from Gackle gives you a unique point along the blue route. However, if the location of interest is *outside* Highway 46, its geo coordinates cannot be determined using the elapsed time only from *two known locations*, let alone from one known location. The dashed red arc shows, *theoretically*, all potential locations that can be reached within 29 minutes from Grackle. Adding secondary information from another location will narrow down our choices of location to two places. For instance, 29 minutes from Ederlin and 57 minutes from Gackle point to *two places* on the map: Jamestown or Edgeley. Geometrically, now you are looking at intersection points of two circles of different radius. In order to pinpoint a unique city we need a tertiary source of information, the elapse time from a *third* location. For instance, if we also know that the place of interest is 14 minutes from Buchanan, then we are able to pinpoint to a unique city: Jamestown. See Figure 13.2.

The above example shows that it is possible to resolve a unique location in 2D using only elapse time from at least *three known locations*. Likewise, to resolve a unique location in 3D (latitude, longitude, and altitude), we need elapse time from at least *four known locations*, a geometric equivalent to solving intersection of four *spheres*.

The Global Positioning System (GPS) works using a similar principle. Instead of using fixed ground landmarks (like the three cities in the above example), the system uses several orbiting

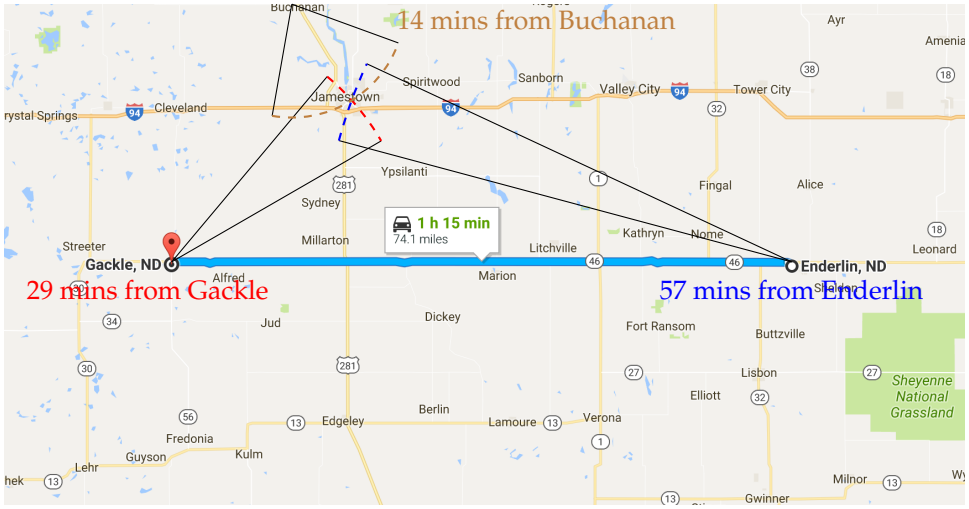


Figure 13.2: Time of Travel From Three Known Locations

satellites that transmit *synchronized* current time with nanosecond accuracy. Using the time information in the signal transmitted from the GPS satellites, a GPS receiver can determine how long it takes for each signal to reach the receiver. Using the receiver's current time (*time of arrival*) and temporal data embedded in each GPS signal (*time of transmission*), the receiver calculates how long it takes for each signal to reach it (*time of flight*). Since the location of each satellite is known, the receiver can then interpolate its own location using the same principle depicted in Figure 13.2.

When we talk about GPS receivers most readers will probably think of car-mounted units that cost a significant amount of money. You may be surprised to learn that GPS receivers are also mass produced on a 5mm $\times$ 5mm chip that costs only about $2.50 a piece. Most mobile devices have plenty of space to add a GPS receiver chip on its electronic board. Consequently, mobile apps can easily obtain its current geo location *almost* anytime and anywhere, except in areas where the mobile device is blocked from the GPS satellites.

To overcome the blind spots problem mentioned above, a mobile device usually incorporates data from other sources. For instance, data from an accelerometer or earth magnetometer can be applied to a time integrator to estimate the velocity (speed and direction) of travel. Applying another time integrator to the velocity yields the position estimates. Recall that, mathematically, position can be determined from acceleration by using a second-order *differential* equation. A time integrator is the mathematical equivalent of calculating an integral over the time domain.

In addition to using data from internal sensors as described above, location fixes can also be calculated from the signal transmitted by cell towers or Wifi hubs your device is connected to.

As a developer, you don't have to worry about the details how locations are determined on your mobile device. The system software on both iOS and Android handles this level of detail for you. Android developers should use the fused location provider (FLP) for obtaining the current device location. This location service, introduced at Google I/O 2013 [10], incorporates more data sources than those mentioned in the above paragraph: accelerometer, gyroscope, compass/magnetometer, and barometer.

13.1.2 Privacy Considerations

With knowledge of the current user location, an app may provide additional benefits to the user. For instance a hypothetical Android Auto app may automatically direct the driver to the nearest gas station when it detects low fuel, a calendar app may popup a notification to remind the user to leave earlier for a meeting appointment due to busier traffic along the route to the meeting. However, some users feel uncomfortable with the apps on their device knowing so much about their whereabouts. If you are a Science Friday podcast listener, you may have heard Ira Flatow say:

> *"...because every story has a flip side"*

The users' concern on disclosing their whereabouts is perhaps the flip side to responsible use of location data. To help mitigate this concern and put the user in control of their privacy, both iOS and Android prompt a permission request dialog whenever an app is about to obtain the user's current location. The user can either deny or grant the request

As a general rule of thumb, apps should always respect the user's privacy, and put them in control of what information will be collected and how that information will be used. In addition to the platform-level features that help put the user in control of their information (e.g. permission dialogs) your app should also have a privacy policy that clearly describes what data is collected, how long it might be stored, and how it will be used (or not used).

13.1.3 Adding Location and Maps to Traxy

In the remainder of this chapter we will enhance the Traxy app with location information and interactive maps. The two existing tabs allow Traxy users to browse the time line of their travels:

- the main tab partitions the journals into "current", "future", and "past"
- the calendar tab shows all the travels made in a particular month

An excellent additional feature to the two views is to let Traxy users browse their travel details on a map. The three tabs combined allow Traxy users to browse their travel details both in time and in space.

In this chapter you will add this extra feature as the third tab on the main screen. In addition, we will also incorporate the Place Picker API to let the user fine-tune the place when recording a new journal entry.

To include Google Maps into your app, you would normally login to Google Cloud Developer Console (using your Gmail account) to setup an API key. Fortunately, you already did this step prior to using Google Places Autocomplete in Section 7.3.3.

13.2 Maps and Location on iOS

Apple's Map application on iOS has some advantages from an end user perspective. Not only has the content within the app improved significantly since its 2012 debut, but as of iOS 10, third party map extensions were introduced, which allows apps like Uber and Yelp to integrate their app's content more directly into the Apple Maps app.

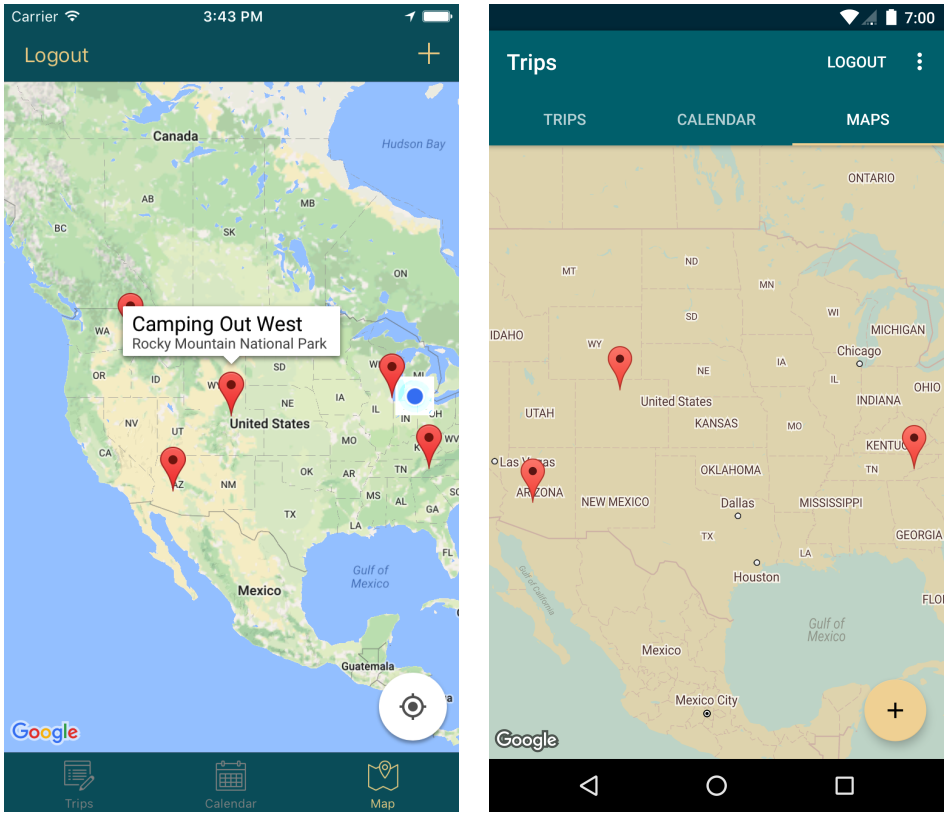


Figure 13.3: Rendering trip (and current) locations with Google Map markers (left) iOS (right) Android.

For developers, Apple provides the MapKit and CoreLocation frameworks. The former allows apps to integrate interactive maps directly, while the latter provides APIs that allows apps to access the device's geographic location, altitude and other positional information relative to deployed iBeacons. Both frameworks are straightforward for developers to use and provide adequate functionality for most map / location-aware features typically found in apps.

Nevertheless, there are some significant advantages to the Maps SDK for iOS provided by Google. First and fundamentally, the map tiles provided by Google have significantly more annotations and detail than Apple maps. Secondly, based on personal usage, we tend to prefer the accuracy, routing and real-time traffic updates (helped in part by Google's acquisition of Waze). Google Maps also provides more features with StreetView, public transport info, etc. Finally, Google has made most of its iOS SDKs available via Cocoapods, so integrating the Google Maps SDK into your Xcode project is almost as easy as adding Apple Maps! In what follows, you will use the Google Maps SDK for iOS to implement Traxy's map feature.

13.2.1 Displaying and Interacting with Maps

Add Google Maps to the Podfile in your top level project directory by adding this pod:

```
pod 'GoogleMaps'
```

Don't forget to do run a `pod install` after updating the Podfile. Next, enable the Google Maps SDK for iOS on your Google Developer Console just as you did previously for the Google Places client. Once you have enabled the API, you can add the following import to the top of `AppDelegate`:

Swift

```
1 import GoogleMaps
```

Then, in the `AppDelegate`'s `application:didFinishLaunchingWithOptions` method, add the following line of code:

Swift

```
1 GMSServices.provideAPIKey(YOUR-KEY-GOES-HERE)
```

where `YOUR-KEY-GOES-HERE` is the same key value used earlier for Google Places.

Implementing the `MapViewController`

Next, introduce a new class named `MapViewController` and flesh out its implementation as shown in Listing 161. In the `viewDidLoad` method (lines 9–20) manually instantiate a Google maps view and sets it to the view controller's view. The current location button is enabled (lines 16–17) so the user can quickly pan to the device's current location. The current controller is established as the delegate for the map view, and the implementation of this delegate is shown as an extension in Listing 162. Note how the delegate performs the manual segue you added previously and prior to that stores the journal tapped in an instance variable.

In the `viewWillAppear` method (lines 22–42) a marker is added to the map view for each journal in the model. Note how the journal is stored on the `userData` property of each marker (line 34). This data is accessed by the delegate whenever a marker is tapped.

Two helper methods are invoked by the `viewWillAppear()` method: `getCurrentLocation()` and `focusMapToShowMarkers()`. The former uses the iOS CoreLocation framework to determine the device's current location and the latter attempts to set the zoom level of the map view to the max zoom level that includes all the markers.

The implementation of `focusMapToShowMarkers()` is shown in Listing 163. The method calls the map method on the array of markers to create a `GMSCoordinateBounds` that includes all of the coordinates. If the current location is known, that coordinate is included in the mapping. Once the mapping has completed, the view is animated to the zoom level that includes those markers.

Swift

```swift
import UIKit
import GoogleMaps

class MapViewController: TraxyTopLevelViewController {

  var tappedJournal : Journal? = nil
  var locationManager:CLLocationManager!
  var currentLocation:CLLocation?

  var markers:[GMSMarker] = []

  override func viewDidLoad() {
    super.viewDidLoad()

    // zoom map to center of USA by default
    let camera = GMSCameraPosition.camera(withLatitude: 39.844546,  longitude:
    -101.304128, zoom: 3.0)
    let mapView = GMSMapView.map(withFrame: CGRect.zero, camera: camera)
    mapView.isMyLocationEnabled = true
    mapView.settings.myLocationButton = true
    mapView.delegate = self
    view = mapView
  }

  override func viewWillAppear(_ animated: Bool) {
    super.viewWillAppear(animated)
    self.getCurrentLocation()
    if let mapView = self.view as? GMSMapView {
      mapView.clear()
      self.markers.removeAll()
      if let journs = self.journals {
        for j in journs {
          let marker = GMSMarker()
          marker.position = CLLocationCoordinate2D(latitude: j.lat!, longitude: j.lng!)
          marker.title = j.name
          marker.snippet =  j.location
          marker.userData = j
          marker.map = mapView
          markers.append(marker)
        }
      }

      self.focusMapToShowMarkers(markers: markers)
    }
  }
}
```

Listing 161: Implementation of `MapViewController` (referenced helper functions are defined below).

```swift
extension MapViewController : GMSMapViewDelegate {
  func mapView(_ mapView: GMSMapView, didTapInfoWindowOf marker: GMSMarker) {
    if let data = marker.userData as? Journal  {
      self.tappedJournal = data
      self.performSegue(withIdentifier: "showJournalSegue", sender: self)
    } else {
      self.tappedJournal = nil
    }
  }
}
```

Listing 162: Implementation of GMSMapViewDelegate by `MapViewController`. The code is called when any journal marker on the map is tapped.

```swift
func focusMapToShowMarkers(markers: [GMSMarker]) {

  if let mapView = self.view as? GMSMapView {

    var bounds : GMSCoordinateBounds!
    if let current = self.currentLocation {
      bounds = GMSCoordinateBounds(coordinate: current.coordinate,
      coordinate: current.coordinate)
    } else {
      bounds = GMSCoordinateBounds()
    }


    _ = markers.map {
      bounds = bounds.includingCoordinate($0.position)
    }

    mapView.animate(with: GMSCameraUpdate.fit(bounds, withPadding: 15.0))

  }
}
```

Listing 163: Helper function to zoom the map view into the max zoom level that includes all markers.

The implementation of `getCurrentLocation()` and an associated helper function are shown in Listing 164. A `CLLocationManager` instance can be used to startup location services. Since location services have privacy implications, you need to first determine if location services have been enabled by the user. This is accomplished by calling the static `locationServicesEnabled()` method on `CLLocationManager`. If location services are enabled, then the next order of business is to determine what level of authorization the user has granted to the app by calling the `authorizationStatus()` method. There are a number of different possibilities to handle here:

- .notDetermined: authorization has not yet been specified by the user. In this case, you'll want to request for permission, which is acocmplished by calling `requestWhenInUseAuthorization()` on the location manager instance. The system will prompt the user once (and only once) when this method is called. This will typically occur the first time this scene is invoked by the user.
- `.restricted` or `.denied`: the .restricted will be returned rather infrequently. When a device (typically owned by a corporation or large organization) is managed by a enterprise mobile device management system location information may be restricted. If the user has previously denied access to the app, the `.denied` value will be returned. In both cases, a dialog will be presented that informs the user that location information has not been granted, and provides instructions for enabling.
- `.authorizedAlways`, `.authorizedWhenInUse`: In this app you are only asking for access when the app is in the foreground (e.g. see line 10) so if the user has granted authorization, the .authorizationWhenInUse value will be returned. Applications can also ask for access to location information, even if there app is not in use.

> **NOTE**
>
> The CoreLocation framework must be manually added as a linked framework to your Xcode project, as it will not be linked by default when you create a new project.

The `askForLocationPermits()` helper function shown in Listing 164 "remembers" each time it is called and will only put up a dialog to the user once a day. This is to avoid popping up an annoying dialog every time the user lands on this tab! To determine the number of days since we asked for permission last, we need to add another function to our `Date` extension (in the source file `Date+ISO8601.swift`) which is shown in Listing 165.

A `UIAlertController` is used to inform the user that location services are disabled, and that more functionality is available to the user if they enable them (see Listing 166). The dialog will also offer to launch the iOS Settings app in order to facilitate this.

Notice that the CoreLocation code examined thus far simply gets the process started. The actual location information arrives via delegation. In particular, you need to have the `MapViewController` implement the `CLLocationManagerDelegate` in order to receive the information. Go ahead and implement this as an extension as shown in Listing 167. Notice that the delegate simply stores the current coordinate as an instance variable, stops location updates, and then refreshes the map view's zoom level.

```swift
func getCurrentLocation() {
  self.locationManager = CLLocationManager()
  self.locationManager.delegate = self
  self.locationManager.desiredAccuracy = kCLLocationAccuracyBest

  if CLLocationManager.locationServicesEnabled() {
    switch(CLLocationManager.authorizationStatus()) {
    case .notDetermined:
      print("Not Determined")
      self.locationManager.requestWhenInUseAuthorization()
    case .restricted, .denied:
      print("No access")
      self.askForLocationPermits()
    case .authorizedAlways, .authorizedWhenInUse:
      print("Access")
      self.locationManager.startUpdatingLocation()
    default:
      self.askForLocationPermits()
    }
  } else {
    self.askForLocationPermits()
  }
}

func askForLocationPermits()
{
  // only display once a day!
  if let lastUpdate = UserDefaults.standard.object(forKey:
  "LastAskedForLocationPermits") as? Date {

    if lastUpdate.days(from: Date()) >= 1  {
      displayNoLocationAccessAlert()
    }
  } else {
    displayNoLocationAccessAlert()
  }
}
```

Listing 164: Helper function obtain the device's current location using the CoreLocation framework.

```swift
func days(from date: Date) -> Int {
  return Calendar.current.dateComponents([.day], from: date, to: self).day!
}
```

Listing 165: An additional function added to the Date extension to compute the number of days between two Date objects.

```Swift
func displayNoLocationAccessAlert()
{
    // remember when we last asked
    UserDefaults.standard.set(Date(), forKey: "LastAskedForLocationPermits")
    let alertController = UIAlertController(
    title: "Location Services disabled",
    message: "The app would like your current location but Location Services on your
    device is currently disabled for this app.",
    preferredStyle: .alert)

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

    let openAction = UIAlertAction(title: "Location Settings", style: .default)
    { (action) in
        if let url = URL(string:UIApplication.openSettingsURLString) {
            UIApplication.shared.open(url, options:
            convertToUIApplicationOpenExternalURLOptionsKeyDictionary([:]),
            completionHandler: nil)
          }
      }
    alertController.addAction(openAction)
    self.present(alertController, animated: true, completion: nil)
  }

fileprivate func convertToUIApplicationOpenExternalURLOptionsKeyDictionary(
_ input: [String: Any]) ->  [UIApplication.OpenExternalURLOptionsKey: Any]
{
return Dictionary(uniqueKeysWithValues: input.map { key, value in
    (UIApplication.OpenExternalURLOptionsKey(rawValue: key), value)})
}
```

Listing 166: Helper function obtain the device's current location using the CoreLocation framework.

```Swift
extension MapViewController : CLLocationManagerDelegate {
func locationManager(_ manager: CLLocationManager, didUpdateLocations locations:
[CLLocation]) {
    self.currentLocation = locations[0] as CLLocation
    manager.stopUpdatingLocation()
    self.focusMapToShowMarkers(markers: self.markers)
  }
}
```

Listing 167: An extension to `MapViewController` that handles incoming location updates by implementing the `CLLocationManagerDelegate`.

You saw previously how the `GMSMapViewDelegate` extension stored the journal associated with the tapped marker on an instance variable and then performed the segue of to `JournalTableViewController` so the user could see the contents of the journal. You still need to pass this journal along on the segue by overriding the `prepareForSegue` method as shown in Listing 168.

Swift

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
  if segue.identifier == "showJournalSegue" {
    if let destVC = segue.destination as? JournalTableViewController {
      destVC.journal  = self.tappedJournal
      destVC.userId = self.userId
    }
  }
}
```

Listing 168: Overriding the `prepareForSegue` method in order to pass the tapped journal along to the `JournalTableViewController`.

Finally, whenever your app asks for permission to location info when in use, you must define the key `NSLocationWhenInUseUsageDescription` in your project's Info.plist to specify the prompt that will be displayed when the system requests the user for location information. The purpose of this prompt is to give the user insight into why your app would like the information, and let them determine based on that information if they actually will grant it. Hence, in your app you can define a key/value pair in your Info.plist as follows:

```
<key>NSLocationWhenInUseUsageDescription</key>
<string>Allow this app access to your current location.</string>
```

Adding the map tab to the storyboard

In the storyboard we will add a third tab to hold the Map tab scene. Introduce a new view controller scene to the storyboard and set its custom class to `MapViewController`. Embed this scene into a navigation controller. Add in a manual segue from the `MapViewController` scene to the `JournalTableViewController` scene in the storyboard. This segue will be performed whenever a user taps on a displayed journal marker on the map. Set the segue's identifier to "showJournalSegue". Also, add a left bar button item with label set to "Logout" on the scene, and be sure to connect it to the unwind segue as you did previously for the calendar tab.

That's it! You should now be ready to build and run the app and try it out. If you do not see the map tiles load, you may need to double check that you properly enabled the Maps SDK for iOS on your Google Developer Console, and that you've properly initialized your client code with a valid API key as discussed above. If you are running the app on an iOS simulator, Xcode makes provisions for you to be able to "mock" your current location. Simply tap on the tool button with the "navigation" icon in the debug window toolbar to explore the various options. In addition the simulator's Features > Location menu provides a variety of options to simulate custom locations, and dynamic locations changes simulating a user running, or moving within an auto or on a bicycle.

This concludes the map tab of your iOS implementation of Traxy. Now let's take a look at how the equivalent functionality can be added to the Android version of the app.

13.3 Maps and Location on Android

The current design of the two fragments associated with the two tabs seems to be working well so far. Your next step is to add a third fragment for showing maps of places in the journals.

> **NOTE**
>
> For easier navigation during development, the code sample for this chapter has been reorganized into various sub-packages (subdirectories)
>
> - `model` holds the model classes (`Journal` and `JournalMedia`)
> - `view` holds all the UI-related files (fragment and activity classes)
> - `viewmodel` holds all the view model classes
> - `adapter` holds the recycler view adapters and their corresponding view holder classes
>
> To follow the instructions in this chapter, it is not necessary for you to re-organize your project structure. If you decide to do so, be sure to Android Studio drag-n-drop refactoring feature to move your files into respective subdirectories. If you encounter compile error regarding "undefine reference to `R`" simply add the following import in the affected file:
>
> Kotlin
> ```
> 1 import edu.gvsu.cis.traxy.R
> ```
>
> Other compile errors should be self-explanatory to fix.

13.3.1 Adding Maps to Traxy

The GoogleMaps library provides two options for incorporating maps into your app: (a) as a fragment or (b) as a view widget (MapView). The overall steps for implementing both options look very similar, if you know how to implement one, you can implement the other relatively easily. In this section, you are going to implement the first option: showing maps in a fragment. Since Google Maps fragment has all lifecycle functions pre-populated with necessary logic, you don't have to write them yourself, hence it is much simpler to use Google Maps Fragment than MapView.

1. Add the library required to use Google Maps

   Gradle
   ```
   1 compile 'com.google.android.gms:play-services-maps:17.0.0'
   ```

   As always, be sure to match the version number *x.y.z* with the other play services libraries

2. Create `TripMapsFragment` from the Google Maps Fragment template. Android Studio will automatically create several files:

   - Kotlin source file `TripMapsFragment.kt`
   - Layout file `fragment_trip_maps.xml`
   - XML resource files `google_maps_api.xml` for defining string resource `google_maps_key`. You will find two copies of this file (under `app/src/debug` and `app/src/release`)

   To conceal the Google Maps API Key, you can remove these two files and define the string resource in your app `build.gradle`, using a similar technique described in Section 7.3.2. Since we already have an API key for the Places API, the same key can be used for Maps API:

   Gradle
   ```
   buildConfigField "String", "GOOGLE_API_KEY", secretProp['API_KEY']
   resValue "string", "google_maps_key", secretProp['API_KEY']
   ```

   be sure that "string" for `resValue` is typed in lowercase.

To include the map fragment as the third view in our view pager, you must modify the following files:

- `JournalPagerAdapter` should now handle three fragments
- `JournalPagerFragment` the `TabLayoutMediator` should also handle three tabs.

Rerun your app to make sure the maps shows up correctly the map marker shows up in the correct position.

Let's move on to adding code to show markers at travel destinations in our journals. Rendering the markers must be performed only after the map and the view model data are completely initialized. Be aware that these two events are **asynchronous** and we don't know for sure which one will finish before the order. Each of these events can be captured by their respective observer:

- Upon completion of the map initialization, the closer given by `OnMapReadyCallback` will run

  Kotlin
  ```
  private val callback = OnMapReadyCallback {
    // code that will run when map is completely initialized
  }
  ```

- Changes to media model live data is captured by the LiveData observer block:

  Kotlin
  ```
  // in onViewCreated() of TripMapsFragment.kt
  viewmodel.remoteJournals.observe(viewLifecycleOwner, Observer{
    // code that will run the the journals are fetched from Cloud Firestore
  })
  ```

Each of these blocks should call the same function to show the markers on the maps:

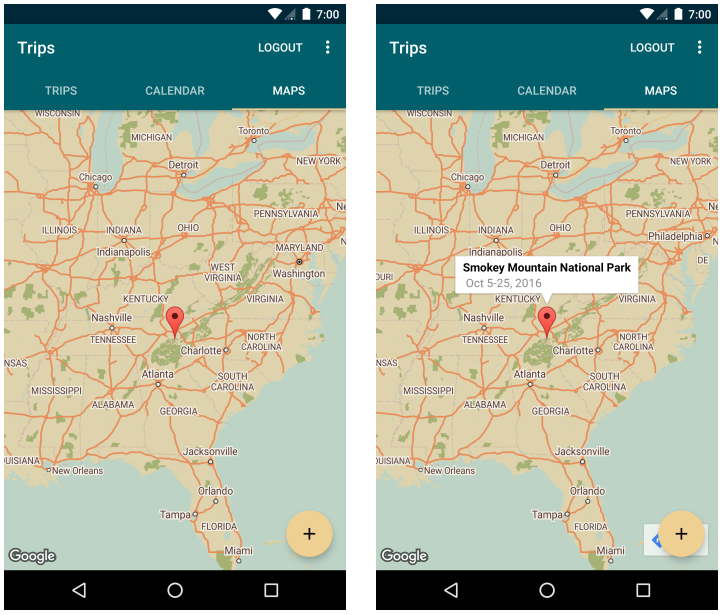


Figure 13.4: Map marker (left) with its info window opened (right)

Kotlin

```kotlin
private fun refreshMarkers(theMap: GoogleMap, theJournal: List<MarkerOptions>) {
  theMap.clear()
  theJournal.forEach {
      mapBound.include(it.position)
      theMap.addMarker(it)
  }

  val screenWidth = requireActivity().resources.displayMetrics.widthPixels
  val screenHeight = requireActivity().resources.displayMetrics.heightPixels
  val cameraUpdate = CameraUpdateFactory.newLatLngBounds(mapBound.build(),
      screenWidth, screenHeight, 128)
  theMap.animateCamera(cameraUpdate)
}
```

Google Maps markers are build from the `MarkerOptions` class and it can be initialized and stored into to a list (`allMarker`) in the view model observer block:

Kotlin

```kotlin
viewmodel.remoteJournals.observe(viewLifecycleOwner, Observer {
  it.forEach {
      val pos = LatLng(it.lat, it.lng)
      allMarkers.add(MarkerOptions().position(pos).title(it.name))
  }
  if (gMap != null)
      refreshMarkers(gMap!!, allMarkers)
})
```

Likewise, the map ready callback can reuse the private function above:

Kotlin
```kotlin
private val callback = OnMapReadyCallback {
  gMap = it
  if (allMarkers.size > 0)
    refreshMarkers(gMap!!, allMarkers)
}
```

13.3.2 Handling Map Interactions

Traxy users can interact with the map like they can with the Google Maps app on their phone.

Ideally, all standard gestures: pan, zoom, pinch, rotate, one-finger double tap to zoom-in, two-finger tap to zoom-out are handled by the map fragment. However, these gestures are also consumed and handled by the parent ViewPager2. To allow the map fragment to handle them without being intercepted we have to wrap the map element with a custom view that intercept the touch event and tell the parent view to not intercept them as well.

Kotlin
```kotlin
class TraxyNestedScrollHost : FrameLayout {
  constructor(context: Context) : super(context)
  constructor(context: Context, attrs: AttributeSet) : super(context, attrs)

  override fun onInterceptTouchEvent(ev: MotionEvent?): Boolean {
    parent.requestDisallowInterceptTouchEvent(true)
    return super.onInterceptTouchEvent(ev)
  }
}
```

To use the above custom view, update your XML layout from:

XML
```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android" ...>
  <fragment android:id="@+id/map"
      android:name="com.google.android.gms.maps.SupportMapFragment"  ... />
</LinearLayout>
```

to

XML
```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android" ...>
  <edu.gvsu.cis.traxy.TraxyNestedScrollHost ...>

    <fragment android:id="@+id/map"
        android:name="com.google.android.gms.maps.SupportMapFragment"  ... />
  </edu.gvsu.cis.traxy.TraxyNestedScrollHost>
</LinearLayout>
```

The only additional feature you have to implement is handling taps on the Info Window which should take the user to edit the selected journal. Since the transition has been registered in the navigation graph, this transition can be implemented with only four lines of code below:

Kotlin
```
private callback = OnMapReadyCallback {
  // other code not shown

  gMap!!.setOnInfoWindowClickListener {
    val action = JournalPagerFragmentDirections.actionEditJournal(it.title)
    val ctrl = Navigation.findNavController(requireView())
    ctrl.navigate(action)
  }
}
```

13.3.3 Where am I?

When viewing points of interest on a map, you usually want to see their relative position and bearing with respect to your current position (shown as a blue dot on the maps). Traxy is currently missing this information and you will add a few more lines of code to implement this feature.

The Android framework provides two ways of obtaining the current device location: **on-demand** or **periodic** update. For our objectives, we don't need to have continuous location updates, on-demand location updates would be sufficient. Moreover, the latter drains more your battery power.

Prior to Google Maps for Android v2, the current user location is provided via the interface `OnMyLocationChangeListener`. This interface has been deprecated and its functionality is now provided by the Fused Location Provider API. This service requires one of the two permissions shown.

XML
```
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
<!-- or -->
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
```

The coarser location option provides a location estimate with an error as large as 2,000 meters, while the finer option provides a location estimate with an accuracy of less than 50 meters.

These two permissions are also considered dangerous and starting with Android 6.0 (API Level 23) they require runtime user approval. The steps for prompting the runtime permission dialog is similar to what you did in Section 10.2.1 (requesting runtime permission for audio recording). The main difference here is that the map renders in a fragment, so naturally the access location permission request should be initiated from the fragment.

Using what you learned in Section 10.2.1, modify `TripMapsFragment` to prompt for the access location permission.

1. Modify the class header to implement `onRequestPermissionsResult`:

Kotlin
```
class TripMapsFragment : Fragment(),
    ActivityCompat.OnRequestPermissionsResultCallback {
  /* class body not shown */
}
```

2. Add the following definition of constants and variable

Kotlin
```kotlin
// inside OnMapReadyCallback of TripMapsFragment.kt
private val LOCATION_PERM_REQUEST = 148
private val PERM<_ARRAY = arrayOf (Manifest.permission.ACCESS_FINE_LOCATION)
```

3. Add the following code to the beginning of `onMapReadyCallback` to request user approval on using the device location.

Kotlin
```kotlin
if (ActivityCompat.checkSelfPermission(requireContext(),
  Manifest.permission.ACCESS_FINE_LOCATION) == PackageManager.PERMISSION_GRANTED)
    gMap!!.isMyLocationEnabled = true
  else
    requestPermissions(PERM_ARRAY, LOCATION_PERM_REQUEST)
```

The initialization of `isMyLocationEnabled` will be flagged as a compiler error, unless the following annotation is added to the beginning of the callback declaration.

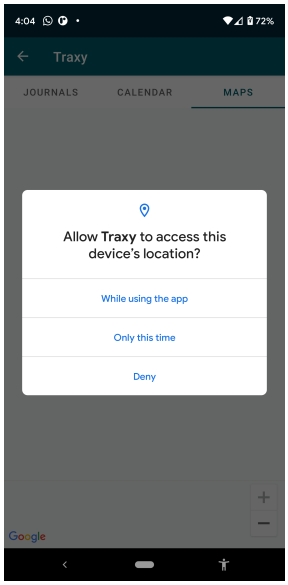
Kotlin
```kotlin
@SuppressLint("MissingPermission")
```

Use Android Studio "autofix" (press ⌥-Enter) to automatically insert the annotation

4. To obtain the user response tn the permission dialog, override the callback method `onRequestPermissionsResult`:

Kotlin
```kotlin
@SuppressLint("MissingPermission")
override fun onRequestPermissionsResult(
  requestCode: Int, permissions: Array<out String>, grantResults: IntArray) {
    super.onRequestPermissionsResult(requestCode, permissions, grantResults)
    if (requestCode == LOCATION_PERM_REQUEST)
      gMap?.isMyLocationEnabled = grantResults[0] ==
        PackageManager.PERMISSION_GRANTED
}
```

If the user granted the permission dialog, you will see the blue dot (indicating your current location) displayed on the maps.



By placing the prompt for access location permission request in the `TripMapFragment` class, the user will be prompted for permission only if she selects the "Maps" tab. Furthermore, by asking the permission in `onMapReady` method, your app can immediately enable the current user location to the appropriate `GoogleMap` object.

After the permission is setup properly,

1. Declare the variable needed for fused location provider and initialize it inside `onCreate`:

Kotlin
```kotlin
private lateinit var fusedLocation: FusedLocationProviderClient

override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)
  fusedLocation = LocationServices.getFusedLocationProviderClient(requireContext())
}
```

2. To handle the user request for retrieving the device current location, implement the location button click listener:

Kotlin
```kotlin
class TripMapsFragment : Fragment(), GoogleMap.OnMyLocationButtonClickListener
```

this will force your class to override the following function:

Kotlin
```
override fun onMyLocationButtonClick(): Boolean {
  if (!::fusedLocation.isInitialized) return false

  fusedLocation.lastLocation.addOnSuccessListener {
      gMap?.run {
          val camUpdate = CameraUpdateFactory.newLatLng(LatLng(it.latitude,
            it.longitude))
          animateCamera(camUpdate)
      }
  }
  return true
}
```

3. Register the location button click listener

Kotlin
```
// inside OnMapReadyCallback
gMap!!.setOnMyLocationButtonClickListener(this)
```

Rerun Traxy and allow it to access your device location. A blue dot should appear on the map to indicate your current location. If you are running the app on an emulator, you may have to enter a specific geo location. The next section describes how to do that.

13.3.4 Mocked Locations in Android Studio

At times, you do need to test how your app responds to geo location updates. To achieve this effect on a real device, you have to physically move the device to different places either by walking or driving a few miles. The Android emulator provides a convenient way to inject new geo locations into your emulated devices from the comfort of your desk. The three horizontal dots at the bottom of Android emulator control invokes the extended controls window as shown in Figure 13.5.

The "Location" tab on the left of Figure 13.5 selects the GPS related settings that allow you to search for places (similar to Google Places API)

The current user location is displayed on the Android device (physical or emulator) as the blue dot. Figure 13.5 currently shows Niagara Falls, New York as the emulated current location. Entering a new location sets the emulator current GPS location to the new location. To recenter the map to the new location, you must press the "My Location" icon on the maps.

> NOTE
>
> FusedLocationProvider will return the **last known location**. When you run your app on a physical device other apps on your phone may already request for a GPS fix and the function —lastLocation— will return that coordinates. However, if you test your app on an emulator, your app may be the first one requesting for GPS fix and the function may return `null`. As easy trick to overcome this issue is to run another app (such as Google Maps) on your emulator prior to running yours.

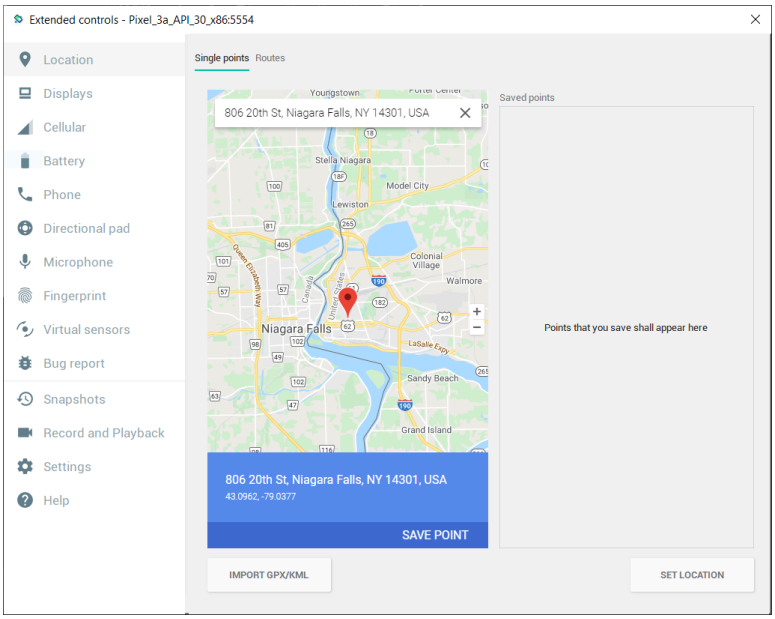


Figure 13.5: Android Emulator Extended Controls

A Learning Swift

A.1 Language Overview

Apple's announcement of a brand new language named Swift, at their 2014 World-Wide Developer Conference came as quite a shock to many programmers who had spent years building products on Apple's software stack. Though it had been years in the making, there was not much of a hint of its arrival to those outside of Apple.

As the shock wore off, and developers started to dig into the details of the new language, there was a sort of deja vu experience, as the Swift language and the underlying platform frameworks and some of Objective-C's language qualities are readily recognizable in the new Swift world. For example, Objective-C's named parameters (one of the things that new Objective-C programmers were always confused by, but in the long run improve code readability) found their way into Swift, as did Objective-C's object model.

It also introduces modern language features that are more often found in scripting languages and less often found in compiled system languages. Apple claims that Swift is the first industrial-quality systems programming language that is as expressive and enjoyable as a scripting language. If you are an old school command line C systems programmer, you might disagree with that rather subjective claim, but for most of us, as we find our way up the Swift learning curve, we'll tend to agree with Apple. Swift is modern, fun, and not at all difficult to learn.

That said, one might still question the audacity of Apple to clean-room implement an entirely new language and force it upon its development community. For example, if Objective-C was beginning to show its age, why not adopt an existing language like Ruby or Python?

Part of the answer to that question is that Apple wanted to make sure that all of its application frameworks (which happened to be written in Objective-C) were available and essentially the same in Swift. They also wanted to make it possible for developers to incrementally adopt the new language. For example, if you had a successful production app deployed that was written in Objective-C, you should be able to add new features to the app in Swift. Alternately, if you were

writing a new green field app in Swift, and wanted to reuse parts of an app your wrote previously in Objective-C, you should be able to include those components without rewriting them. This ability to access the established frameworks and mix and match languages was one of the reasons Apple chose to implement an entirely new language.

Those reasons aside, Objective-C, a derivative of C with some object-oriented features, was a difficult language to learn for developers who cut their teeth on Java and other "memory managed" scripting languages. Apparently, Apple, the UX rockstar, also wanted us programmers to have a nice happy experience, and Objective-C didn't really provide that unless you're over 40 years old and into retro.

In what follows, we don't intend to provide you with a comprehensive reference manual on the Swift programming language. Instead, we just want to take you on a quick tour of the language, pointing out some of the key features that you will be using in your iOS development. A list of more extensive learning resources are listed at the end.

Swift is a compiled language, not interpreted. It has strong static typing, which means types are clearly identifiable and cannot change once they are established. This isn't of course designed to make your life less flexible as a program, but rather puts the compiler into a position to generate more optimized code.

Unlike Java and many of the more modern scripting languages you may have worked with, Swift does not perform garbage collection (e.g. periodically sweep memory and remove any unreachable orphaned allocations). At the same time, memory management is not totally manual either, like in C and C++ where you the programmer have to manually release every chunk of memory you allocated. Instead, Swift implements *automatic reference counting* (referred to as ARC in the Apple developer community). That is, counters are associated with every reference created for allocated memory. When a reference is no longer needed, the counter is decremented. Any time the counter reaches zero, the underlying system software will automatically release the allocation.

Swift also supports name spaces, which makes it very easy for you code to co-exist with other people's code. This is particularly important today as open source software is now mainstream, even in mission critical systems.

Before we get started, let's take a quick look at the obligatory "hello world" program in Swift:

Swift

```
1 print("Hello World")
```

Congratulations, you've just written your first complete Swift program. That's all there is to it - no main program, no import statements, and of course no semicolons! Just a simple line of code written at global scope.

One final caveat before we start looking at the language in more detail. Apple is aggressively evolving Swift as we speak. Hence, every year since its 2014 debut, Apple has either released a minor or major release, which means the language is still changing! The following tour has assumed Swift 5, but you'll still find a lot of legacy 3.x and earlier syntax in projects on github and unfortunately in answers on stackoverflow.com. The good news is that Xcode recognizes older versions of Swift and will automatically updated it to the most recent version if you'd like. This works pretty well, most of the time. Nevertheless, expect continuous change and evolution to the language in the future. Swift 5.3.2 was released in December 2020, and you can expect additional updates to the language in the future.

Apple open-sourced the Swift language at the end of 2015 and since, has enjoyed a fair amount of adoption beyond the Apple family of platforms. A number of other companies, such as IBM and others have interesting projects involving Swift. There have even been rumors online that Google was going to adopt it for Android, though that doesn't seem likely in our lifetime. The bottom line is, Swift is growing mind share rapidly (#14 on the TIOBE Index[6]) and it is not much of a stretch to assume the effort you put into learning the language today will eventually pay dividends on projects you engage in beyond the Apple family of platforms.

A.2 Variable Declarations

There are two types of variable declarations in Swift. Stuff that can't change once defined (e.g. constants) and stuff that can (e.g. variables). Constants are defined with the `let` statement:

Swift
```
let implicitConst = 25
```

In this case, the constant's name is `implicitConst`, and it is assigned the integer value of 25. Notice that the type here is implicit, not defined. That is, the fact that we assigned the constant to a integer literal 25 means that the constant being declared here is implied to be an integer, or in Swift types, `Int`.

We can also explicitly type constants:

Swift
```
let explicitDouble :Double
explicitDouble = 3.14
```

Here, the constant `explicitDouble` is explicitly typed to `Double`, Swift's type for representing double precision floating point numbers. Note also, that we don't have to include an initializer when we declare the constant. We can assign it later. However, once we assign it, any subsequent attempts to assign the constant a value of course would be flagged as a compiler error.

Non-constants or variables can be defined with the `var` keyword:

Swift
```
var aNiceMessage = "bye"
var myCounter: Int = 1
```

Here two variables are defined, `aNiceMessage` is implicitly typed to `String`, and assigned the string literal `"bye"`. The variable `myCounter` is explicitly typed to Int and assigned the value `1`. Unlike constants, these variables can be mutated over time.

Xcode gets pretty excited about constants. For example, if you define something as a `var`, but then never mutate it in the subsequent code, Xcode will jump up and down in front of you and suggest with brilliance that you really meant to define a constant via `let`. Don't let it get to you (no pun intended). Xcode is just trying to get you to be more intentional as a programmer, which will help you write better code in the long run.

A.3 Strings

Swift supports a built in string type `String`, which is essentially an ordered collection of `Character`. Consider the following example:

```swift
var aString = "favorites: "
let g,r,b: String
g="green"; r="red"; b="blue"
```

Note that `aString` is implicitly typed, while `g`, `r`, and `b` are explicitly typed to `String`. Also note that multiple variables or constants can be introduced on a single line by using commas. Similarly, multiple Swift statements can be on the same line as long as they are delimited by semicolons. That is in fact the only time you use a semicolon in Swift. If you are a long time Java or C programmer it probably will take a while for you to get used to not typing semicolons at the end of lines! The good news is that it doesn't matter if you do. They will simply get ignored by the compiler.

As you might expect, we can do string concatenation with the + and += operators:

```swift
aString += g +   ", " + r + ", " + b
```

Based on the above definitions, this value `aString` holds after executing this statement would be `"favorites:  green, red, blue"`.

Swift strings are automatically bridged to Objective-C's string type, `NSString`. What that means is that any iOS framework method that receives a string (e.g. `NSString`) parameters, can be called from Swift and passed a value of type `String`. Similarly, if we have legacy code of our own written in Objective-C, any method that takes an `NSString` can receive a `String` when called from Swift.

However, unlike `NSString`, the Swift `String` type is what we call a *value type*. What that means is that if we pass a String parameter to a function, the Swift runtime will make a copy of the original string and pass a copy (e.g. the value) of the original parameter instead of a reference to the original string. In Objective-C, `NSString` is a *reference type*. That is, when it is passed to a function, a reference (e.g address) to the original parameter is sent.

Swift, like many of the more popular scripting languages also supports string interpolation, or the ability to construct new strings from a mix of values and expressions. For example, consider the following code:

```swift
var age = 101
var name = "Bob"
var sayit = "\(name) is \(age) years old!"
```

The variable `sayit` will have the string value "Bob is 101 years old!", hence the interpolation operator `\()` converts the variable within the parenthesis into a value of type `String`, and then concatenates it with what comes before and after.

Strings can be compared to each other using the == operator:

```swift
if name == "Bob" {
  print(sayit)
}
```

The Swift `String` type also defines a number of useful methods such as the `hasPrefix` and `hasSuffix` methods:

```swift
if name.hasPrefix("Bo") {
  print("Bo Bo Bo")
} else if name.hasSuffix("sma") {
  print("Frisian alert")
}
```

A.4 Collection Types

Swift has built in types for the common types of collections you might want to model in your data structure. Arrays are used to store ordered lists of values. In Swift, all the values in a given array are of the same type. However, note that due to object oriented inheritance, you may encounter situations in which it seems an array instance is populated with heterogenous types. A closer observation will bring to light that what is actually happening is that all of the values share a common ancestor class.

Values of arrays can be accessed via methods, properties, and subscripting:

```swift
var shoppingList: [String] = ["milk", "eggs", "bread"]
```

Here, we've defined an array variable named `shoppingList`. It contains a collection of values of type `String`. Hence the type specifier for an array of strings is `[String]`. Elements within the array can be accessed via subscripting:

```swift
print("The last item is " + shoppingList[2])
```

We can determine how many elements are in an array by accessing its `count` property:

```swift
print("There are \(shoppingList.count) items in the shopping list.")
```

Arrays can be iterated via a simple for loop:

```swift
for item in shoppingList {
  print(item)
}
```

Note that the loop control variable `item` is simply populated with the ith element in the array. If we want the actual index to the items, we need to take a slightly different approach:

```swift
for (item, value) in shoppingList.enumerated() {
  print("Item \(item+1): \(value)")
}
```

We can also append items to an array by using the append method or the += operator:

```swift
shoppingList.append("chips")
shoppingList += ["dogfood", "catfood"]
```

Dictionaries in Swift allow you to map values to specific key values. Like arrays, dictionaries in Swift store multiple values of the same type. Each value in the dictionary is associated with a unique key. There is not predefined ordering on the elements. The type of the key must be *hashable*, which in Swift simply means that the type must conform to the `Hashable` protocol. We'll talk about protocols in a moment, so for now you can just stash this last factoid.

Dictionaries are defined in Swift with a syntax of their own:

```swift
var languageLikes: [String:Int] = [
  "Java":10, "Objective-C":2, "Swift":12, "VB":0]
```

The dictionary languageLikes maps integers to string keys. We can retrieve values stored in a dictionary or set them using the key values in subscripts:

```swift
var javaLikes = languageLikes["Java"]
languageLikes["Ruby"] = 15
```

Like arrays, we can iterate through a dictionary using a for loop:

```swift
for (language, likes) in languageLikes {
  print("\(language): \(likes)")
}
```

A.5 Control Flow

Swift has all of the control constructs you might anticipate in a modern language. We've already seen the for loop in action. There are a lot of other interesting things we can get done with for loops as well. For example, we can iterate over a range:

```swift
for index in 1...5 {
  print("\(index) times 5 is \(index * 5)")
}
```

One important think to note here is that when specifying a range, there are three dots between the start/stop numbers, not two!

If we didn't actually need to access the loop control variable, we don't need to give it a name:

```swift
let base = 3, power = 10
var answer = 1
for _ in 1...power {
  answer *= base
}
print("\(base) to the power of \(power) is \(answer)")
```

This notion of using an underscore instead of a identifier can be used in other constructs as well. In fact, if you define a value in an expression like this (or in an if statement) and do not access it in the associated code, Xcode will get in your face and ask you to change the identifier to an underscore!

We can also write while loops in Swift:

```swift
var n = 5, cnt = 1
while (cnt <= 5) {
  print("\(cnt) times 5 is \(cnt * 5)")
  cnt+=1;
}
```

And as you might expect, we have the repeat / while loop as well:

```swift
var n = 5, cnt = 1
repeat {
  print("\(cnt) times 5 is \(cnt * 5)")
  cnt+=1;
} while (cnt <= 5)
```

The if and if-else statements can be used for conditionals:

```swift
var temp: Int = 50
if temp <= 32 {
  print("It is freezing out here")
} else if temp > 32 && temp <= 80 {
  print("How temperate it is here!")
} else if temp > 80 && temp <= 100 {
  print("It is a sweat shop here!")
} else {
  print("The thermometer is broken!")
}
```

The switch statement in Swift takes switching to a new level! First off, the *implicit fall through* that created so many confounding bugs in your C and Java programs has been relegated to the bit bucket. Hence, no break statements are needed in your switch statements! However, Swift will require you to have at least one executable statement in each case. The case conditions themselves can get quite interesting. Not only can they consist of scalar values (like Java and C) but you can also specify a range or even a tuple (we'll talk about tuples shortly). Furthermore, the case conditions can actually bind data as well.

```swift
var t: Int = 32
switch t {
  case 0...32:
    print("It is freezing out here")
  case 33...80:
    print("How temperate it is here!")
  case 81...100:
    print("It is a sweat shop here!")
  default:
    print("the thermometer is broken!")
}
```

A.6 Functions

In Swift, we can write a sequence of instructions that takes input parameters and returns out values, and give it a name. Such a construct is known as a *function*. Functions in Swift are similar to functions in other languages, and analogous to what other languages refer to as routines, procedures, or methods. Functions are typed by both their return type as well as the types of its parameters. Swift supports the notion of *first-class functions*, that is the notation that functions themselves can be treated as data and passed as arguments and serve as return values. Functions may be nested, and may also have *multiple return values*.

```swift
// define a function
func stateYoMission(actionVerb: String) -> String {
  let mission = "I will " + actionVerb + " dog!"
  return mission
}
```

The function name is `stateYoMission`. It takes a single parameter named `actionVerb` of type `String` and also returns a `String`. We can invoke the method using its name. Notice that the parameter is named as well.

```swift
// invoke a function
print(stateYoMission(actionVerb: "walk"))
```

Functions can have multiple arguments. This is where the named parameters start to shine. If we didn't have the function implementation in front of us for reference (a common situation when coding) we can look at the code that invokes it and figure out fairly quickly what the parameters are.

```swift
func stateYoMission(actionVerb: String, noun:String) -> String {
  let mission = "I will " + actionVerb + " the " + noun + "!"
  return mission
}
print(stateYoMission(actionVerb: "walk", noun: "cat"))
```

Functions return multiple values by way of *tuples*. We have yet to formally present the concept of tuples in Swift, but this example function that has multiple return values shouldn't be too hard to figure out:

Swift
```
func findTheCenter(x1:Int, y1:Int, x2:Int, y2:Int) -> (x:Int, y:Int) {
  return ((x2-x1)/2, (y2-y1)/2)
}
print(findTheCenter(x1: 0,y1: 0,x2: 10,y2: 10))
```

So far, all the functions we've seen have defined what are known as *local parameter names*. Sometimes it is useful to have external parameter names - names that are used when the method is invoked. For example:

Swift
```
func join(string s1:String, toString s2:String, withJoiner joiner:String) -> String {
  return s1 + joiner + s2
}
```

Note that the first parameter has external name `string` and internal name `s1`. Similarly the second parameter is `toString` externally and `s2` internally, and the third `withJoiner` externally and `joiner` internally. When we invoke such a function, the parameter names we use are the external names:

Swift
```
join(string:"hello", toString:"world", withJoiner:", ")
```

In early versions of Swift, the first parameter of a function had no external parameter name by default, but as of Swift 3 all of the parameters have external names. If we don't explicitly provide the external parameter names, then by default both the internal and external names are the same:

Swift
```
// external and internal parameter names are the same
func join(s1:String, s2:String, joiner:String) -> String {
  return s1 + joiner + s2
}
join(s1:"hello", s2:"world", joiner:", ")
```

We can define a variable that is assigned a function, and then invoke that function via the variable:

Swift
```
func addTwoInts(a: Int, b: Int) -> Int {
  return a + b
}
var mathFunction: (Int, Int) -> Int = addTwoInts
print("Results: \(mathFunction(4,6))")
```

You can also pass a function as an argument to another function. In this example, we have a function whose first parameter is named `mathFunction`. The type of `mathFunction` is a function that takes two `Int` values and returns an `Int` value.

```Swift
func printMathResults(mathFunction: (Int, Int) -> Int, a: Int, b: Int) {
  print("Results: \(mathFunction(a,b))")
}
printMathResults(mathFunction: addTwoInts,  a: 3, b: 5)
```

You can write functions that have a function as a return value:

```Swift
func stepBackward(input: Int) -> Int {
  return input - 1
}

func stepForward(input: Int) -> Int {
  return input + 1
}

func chooseStepFunction(backwards: Bool) -> (Int) -> Int {
  return backwards ? stepBackward: stepForward
}

var curVal = 3
let moveNearToZero = chooseStepFunction(backwards: true)
curVal = moveNearToZero(curVal)
```

The `chooseStepFunction` function returns a function that can in turn be invoked.

Finally, we can also nest functions in Swift. For example, we could rewrite the previous example like this:

```Swift
func chooseStepFunctionV2(backwards: Bool) -> (Int) -> Int {
  func stepBackward(input: Int) -> Int {
    return input - 1
  }
  func stepForward(input: Int) -> Int {
    return input + 1
  }

  return backwards ? stepBackward: stepForward
}

var curVal = 3
let moveNearToZero = chooseStepFunctionV2(backwards: true)
curVal = moveNearToZero(curVal)
```

Note the functions `stepBackward` and `stepForward` are defined within the scope of `chooseStepFunctionV2`, yet when return from that function that can still be invoked outside of the function's scope by way of the return value.

A.7 Closures

Closures are self-contained blocks of code that can be passed around like data and invoked by your code. The name *closure* derives from the fact that closures automatically capture and store

references to constants and variables visible in the scope in which they are defined. This term for this automatic capture of context is known as *closing*, hence the name closure.

This concept is found in many programming languages today. For example, Objective-C was extended to include a construct known as a *block*. An Objective-C block is analogous to a Swift closure. In Java and Ruby we were to similar constructs as *lambdas*.

In Swift functions themselves are simply closures that have names. They may or may not capture any values, depending on the scope at which they are defined. More commonly however, we think of closures as *closure expressions*, which are essentially unnamed or anonymous closures that capture values that are visible within their scope.

When we author closure expressions in Swift, we can take advantage of some nice syntactical optimizations that makes our closures elegant and compact.

In general, the syntax for a closure expression follows this form:

Swift
```
{ (params) -> returnType in
  statements
}
```

The parameter list follows the exact syntax as parameter lists in functions, which we've seen previously. One or more Swift statements can be placed between the keyword `in` and the closing brace.

For example, arrays in Swift have a map function that receives a closure as a parameter. The closure expects an input parameter typed to the same type that the array stores, and returns the same type. The idea here, is that we can map an array of values into a new array of values by simply providing a chunk of code that does that mapping.

Swift
```
let students = ["joe", "camila", "ryan", "roland"]
students.map({ (student: String) -> String in
  student + " is getting swifter!"
})
```

Swift allows a number of syntactical optimizations when writing closure expressions. For example, Swift can do *type inferencing*. For example, Swift knows we are storing type String in the previous example, so there is no need to specify the types in our closure expression.

Swift
```
let students = ["joe", "camila", "ryan", "roland"]
students.map({ student in
  student + " is getting swifter!"
})
```

Furthermore, we can leave out the explicit parameter list and simply use short hand expressions to refer to arguments.

Swift
```
let students = ["joe", "camila", "ryan", "roland"]
students.map({
  $0 + "  is getting swifter!"
})
```

In this case, $0 is shorthand for the first parameter. $1 would be shorthand for the second parameter, and so forth. If a closure is passed as the last argument of a function, it can appear outside of the parenthesis.

Swift
```swift
let students = ["joe", "camila", "ryan", "roland"]
students.map() {
  $0 + "  is getting swifter!"
}
```

If there are no parameters other than the closure itself, than we can get rid of the parenthesis altogether.

Swift
```swift
let students = ["joe", "camila", "ryan", "roland"]
students.map { $0 + "  is getting swifter!" }
```

A typical use case for closures in iOS development is anytime you call a function in which the functionality of the function is asynchronous. That is, the function kicks off some sort of operations (e.g. a fetch from the network) that will complete at some arbitrary time in the future. Since we don't know when it will finish, we simply provide the function with a closure expression that defines what we want to have happen when the operation completes.

This notion of when a closure actually gets executed brings to the foreground another interesting Swift language feature. If you pass a closure expression to a function, and the expression is to be invoked after the function returns, it must be marked as an *escaping closure*.

Swift
```swift
func doSomethingNonImmediate(onCompletion: @escaping () -> String) {
  // code here presumably saves onCompletion so it can be invoked later.
}
```

In this example, we can save that the closure expression `onCompletion`, is allowed to *escape* the function and be called after the function returns.

Finally, it is important to realize that closures are implemented in Swift as *reference types*. If you were to define variables that refer to closures, and assign them to the same closure, you don't have two closures. Instead your have two references to the same closure.

A.8 Tuples

A tuple is a collection of ordered values. Tuples can be customized as needed in that each value can have its own unique name and type. For example, a program might want to model a location with a tuple consisting of a string and two floating point values.

Swift
```swift
var location: (String, Float, Float)
location = ("GVSU", 42.965175, -85.889405)
```

Tuples have a variety of handy use cases. We've already seen previously how they can be used to return multiple values from a function. For example, we can use tuples to assign multiple values simultaneously.

Swift
```
var someInt: Int
var someString: String
(someInt, someString) = (10, "hello")
```

We can also use tuple syntax to swap values in two variables.

Swift
```
var alpha = "first"
var omega = "last"
// the first shall become last, and the last first!
(omega, alpha) = (alpha,omega)
```

We can access individual elements within a tuple, using a variety of approaches. First we can use the underscore notation. In this example, we extract the description of the location by way of the assignment statement. The values we don't care about are replaced with underscores on the left hand side of the assignment.

Swift
```
var location = ("Grand Rapids", 42.960253, -85.657857)
var (description, _, _) = location
```

We can also use positional index literals to access individual elements within a tuple. Note that the positions are zero-based, e.g. the first item is position 0.

Swift
```
var latitude = location.1
```

Finally, its also possible to name the individual values within a tuple.

Swift
```
var favLocation = (name: "Newaygo", lat:43.419143, lng:-85.800993)
print(favLocation.name)
```

A.9 Optionals

You can think of an *optional* in Swift as a cardboard box that may or may not contain something. To relate the analogy more closely to the Swift programming language, and optional type represents an entity that may or may not wrap another object. That's where the name optional comes from.

We declare a variable as optional, by simply placing a question mark behind its type specifier:

Swift
```
var mightContainAStr: String? = "Boo!"
```

In this case, the variable `mightContainAStr` is NOT of type `String`. Its type is actually `Optional`. However, in this particular example, `mightContainAStr` *wraps* or *contains* the string `"Boo!"`. However, an optional variable might not always wrap an object. Anytime a variable of type Optional does not wrap another object, it simply has the value `nil`.

```swift
var mightContainAStr: String? = "Boo!"
// at this point, the variable wraps the string "Boo!"
mightContainAStr = nil
// at this point the variable doesn't wrap anything.
```

Note that when we assign a value to an optional variable and it is the type that the optional variable was declared to wrap, than the value being assigned gets automatically wrapped.

```swift
// the string "like magic" gets automatically wrapped and stored in mightContainAStr
mightContainAStr = "like magic"
```

Optionals can be passed to functions, as you might expect. If we pass the wrapped type, Swift will automatically wrap the value before it gets passed. For example, in the following code, when we call the function with a string literal, the string gets wrapped as an optional, before the function is called:

```swift
func gimmeAnOptional(s:String?) {
  // some code goes here
}
let poof : String? = "kapoof!"
gimmeAnOptional(s:poof) // no surprises here...

//  now watch this... string gets wrapped automatically!
gimmeAnOptional(s:"cheese")
```

On the other hand, you cannot pass an optional type where the wrapped type is actually expected. For example, the following code will get flagged as an error by the Swift compiler.

```swift
var problemo : String? = "Make some trouble dudes"
func gimmeAStringPls(s:String) {
  // some code goes here
}
// This line of code will generate a compiler error!
gimmeAStringPls(s:problemo)
```

To correct this problem, we need to explicitly unwrap an optional before using it as its wrapped type. For the previous example, we could call the function if we explicitly unwrapped the string variable with the unwrap operator (e.g. a postfix !).

```swift
gimmeAStringPls(s:problemo!)
```

Similarly, we cannot send a message to an optional before unwrapping it.

```swift
var mightBeValid : String? = "howdy"
mightBeValid.uppercased() // this will generate a compiler error!
```

But if we unwrap the object we can send messages just fine.

Swift
```
// this works great
mightBeValid!.uppercased()
```

Swift also gives us the ability to define implicitly unwrapped optionals. These are essentially the same as the optional as defined to-date, however when we use them where their wrapped type is expected, Swift just quietly unwraps them for us without complaining. We can define these kinds of optionals, by simply replacing the postfix ? with a postfix ! in a variables type specifier.

Swift
```
func gimmeANiceStr(s: String) {
  // some code goes here
}
var mightBeNice : String! = "yeehah"
// works like a charm!
gimmeANiceStr(s: mightBeNice)
```

You need to be a little careful with implicitly unwrapped optionals, because if the optional doesn't happen to wrap an object (e.g. it is nil) than the code above would literally produce a runtime crash.

In fact, anytime you try to unwrap an optional type that is nil, either explicitly with a postfix ! or implicitly, you will get a runtime exception. That means you better either know your optionals have objects, or check them at runtime. A common Swift idiom is to unwrap your optionals by way of an if statement.

Swift
```
var str : String?
//some code goes here that may or may not assign to str

if let s = str {
  // if str is not nil, s will have the unwrapped value
  print(s)
}
```

Sometimes you may need to determine if a number of optionals are wrapping values and do something only if they all are. If we do this using the if statement approach, we could end up with a lot of nested if statements.

Swift
```
var str : String?
var cnt : Int?
var age: Int?
//some code goes here that may or may not assign values to above options

if let s = str {
  if let c = cnt {
    if let a = age {
      // all three have values if we get here!
    }
  }
}
```

A nice convenient alternative is to use the guard statement.,

```swift
var str : String?
var cnt : Int?
var age: Int?
//some code goes here that may or may not assign values to above options

guard let s = str, let c = cnt, let a = age else {
  return
}

// all three values have objects (now in s,c and a respectively) if we get  here!
```

Swift also provides us with a conditional unwrap operator in the form of a postfix ?.

```swift
var mightNotBeValid : String? = nil

// This will generate a runtime exception!
mightNotBeValid!.uppercased()

// A conditional unwrap is just what we need:
mightNotBeValid?.uppercased()
```

In this case, the `uppercased()` message only gets sent if mightNotBeValid wraps a string object.

A.10 Objects

There are essentially three kinds of objects in Swift: enum, class, and struct. They can be declared anywhere in a source file, nested within other objects, or within functions. The scope of the entity is determined by where it was declared.

Not matter what kind of object we are defining in swift, our declarations can have several different items: *initializers*, *properties*, and *methods*. Initializers are commonly referred to as constructors in other languages. They are basically specialized methods that help initialize (or construct) an object instance. Properties are variables that are declared at the top level of an object. Properties can be associated with object instances, or the class itself (e.g. shared by all the objects of the class). Methods are functions declared at the top level of an object declaration. Like properties, methods can be associated with the object instance or the class itself.

A.10.1 Enums

Enums are an object type that represent a named set of distinct values. In Swift we declare them with the enum construct.

Swift
```swift
enum BeeType {
  case Queen
  case Worker
  case Drone
}
```

Once defined, we can define variables and constants that are set to a particular value of the enum. In this example, `myBee` is typed as `BeeType` and has the value `BeeType.Queen`.

Swift
```swift
let myBee = BeeType.Queen
```

Enum values default to `Int`, but can actually be typed to any valid Swift type. For example, here's an enum declaration where the values are defined as fixed strings. We can use the `rawValue` property to access the actual value associated with the case.

Swift
```swift
enum FancyBeeType : String {
  case Queen = "Queen"
  case Worker = "Worker"
  case Drone = "Drone"
}

let fancyBee : FancyBeeType = .Queen
print(fancyBee.rawValue)
```

Finally, enums are *value types* in Swift. If you pass one to a function a copy will be made and sent along, not a reference.

A.10.2 Structs

An object defined by a struct in Swift can be thought of as a knocked down version of a class. In terms of capabilities it is somewhere in between enum and class. Just like classes and enums, it can define initializers, properties, and methods.

Swift
```swift
struct Bee {
  var type: FancyBeeType = .Worker
  var age: Int = 0
  init(type: FancyBeeType, age: Int) {
    self.type = type
    self.age = age
  }

  func description() -> String {
    return "This bee is a \(self.type.rawValue) of age \(self.age) days old."
  }
}
```

Once defined, we can instantiate objects from our struct declaration. Methods are invoked on the object, using the variable name followed by a dot, and then the method name. You can call methods associated with the class by using the class name.

```swift
var busyBee: Bee = Bee(type: .Worker, age: 14)
print(busyBee.description())
```

Nearly all of Swift's built in types, such as `String`, `Int`, `Array`, etc. are all defined used structs. Furthermore, structs are *value types*. That of course has important ramifications. For example, if I pass a parameter of type `String` to a function, and have the function mutate it, once I return from the function, the original string value is not mutated, since the function was operating on a copy.

Unlike classes, there is no object-oriented inheritance associated with structs.

A.10.3 Classes

Classes give us full-blown object nirvana! We can do everything we could do with structs and in addition have object-oriented inheritance. There is also one more really significant different between classes on the one hand, and enums and structs on the other, in that classes, like closures are *reference types* in Swift.

```swift
class Employee {
  var firstName: String
  var lastName: String
  var bio: String

  // no bio provided
  convenience init(firstName: String, lastName: String) {
    self.init(firstName: firstName, lastName: lastName, bio: "I love Swift!")
  }

  // designated initializer.
  init(firstName: String, lastName: String, bio: String) {
    self.firstName = firstName
    self.lastName = lastName
    self.bio = bio
  }

  func computePay() -> Double {
    return 1000.0 * Double(bio.count)
  }
}
```

Once defined, we can instantiate objects from class declarations using the initializers. Each object can in fact define multiple initializers. The initializer that completely initializes the object is known as the designated initializer. Convenience initializers can be written in terms of the designated initializer and provide alternate ways to initialize an instance.

```swift
var steve = Employee(firstName: "Steve", lastName: "Jobs")
var tim = Employee(firstName: "Tim", lastName: "Cook", bio: "I love Apple Watch!!!")
```

New classes can be introduced that inherit the data and behavior of their base class. Behavior of the base class can be selectively overridden as needed.

Swift
```swift
class SalariedEmployee : Employee {
  func computeSalary() -> Double {
    return self.computePay()
  }

  override func computePay() -> Double {
    return 10000000.0 * Double(bio.count)
  }
}
```

A.11 Protocols

Protocols in Swift provide us the ability to make disparate objects adhere to the exact same interface. For example, airplanes crash and so to programs! We could create an abstract interface of what it means and give it the name `BrokenThing`.

Swift
```swift
protocol BrokenThing {
  func crash() -> String
}

class AirPlane : BrokenThing {
  func crash() -> String {
    return "Oops, this airplane just crashed"
  }
}

class Program : BrokenThing {
  func crash() -> String {
    return "I think this program has a bug."
  }
}
```

Protocols in swift are equivalent to interfaces in Java and abstract classes in C++. Other languages have similar constructs, though they may refer to them with slighly different vocabulary.

To get a glimpse of the glories of this construct, we highly recommend you watch Dave Abraham's talk at the 2015 WWDC conference[3].

A.12 Extensions

Extensions in Swift allow us to add behavior to an existing class. For example, if I wanted the built in Date object to be capable of rendering a string in ISO8601 format, I could extend the date class as follows:

```Swift
extension Date {
  struct Formatter {
    static let iso8601: DateFormatter = {
      let formatter = DateFormatter()
      formatter.calendar = Calendar(identifier: .iso8601)
      formatter.dateFormat = "yyyy-MM-dd'T'HH:mm:ss.SSSxxx"
      return formatter
    }()
  }

  var iso8601: String {
    return Formatter.iso8601.string(from: self)
  }
}
```

Having done that, you can now invoke the method defined by the extension as if it was part of the type.

```Swift
let today = Date()
print(today.iso8601)
```

Having done that, you could extend the build in String class to turn itself into a Date object, if in fact it was in the right format.

```Swift
extension String {
  var dateFromISO8601: Date? {
    return Date.Formatter.iso8601.date(from: self)
  }
}
```

A.13 Further Study

We've only touched the surface of the Swift language in this brief tour. There is much more to be learned, but this should get you a decent lay of the land. For further study, we would refer you to three free e-books that Apple has published (via their iBooks app or via the web on their developer site).

- Develop in Swift Fundamentals. [13]
- Swift Documentation. [14]
- The Swift Programming Language (5.3). [15]

In addition to these sources, you will find many excellent online tutorials for learning Swift. We also recommend you pay attention to the Swift-related talks given at recent Apple WWDC conferences: https://developer.apple.com/wwdc/. These in particular will help you understand how the language is evolving and why.

B
Learning Kotlin

The announcement of Kotlin in 2011 as a new programming language predates Swift. It took five years for the JetBrains team to officially release Kotlin 1.0 in 2016. Its interoperability with Java makes Kotlin an attractive choice as an alternative language of implementation for Android apps. Kotlin programs can invoke methods from a library written in Java, and vice versa.

Among the major Kotlin features well-liked by Java developers is Kotlin's (more) explicit handling of null pointers. Similar to Swift's *optional*, Kotlin expands Java's data type by adding a *nullable* type.

This appendix is not intended as a complete reference to Kotlin. Our goal is to provide a "quick start" tutorial to use Kotlin in Android. The order of Kotlin topics presented below follows the Java features that you encounter when writing your first activity. Immediately after creating a new project, you are presented with an `Activity` class that inherits from `AppCompatActivity`. The `onCreate` method of that class overrides the associated parent's method. Therefore, we decided to begin this appendix by comparing the syntax of inheritance between Java and Kotlin.

B.1 Inheritance and Overrides

Let's begin our excursion into Kotlin by comparing how to inherit a child class from its parent class. Specifically, how to declare an Android Activity from its parent `AppCompatActivity`. See Listing 169.

Comparing the two code snippets below, you should immediately notice the following facts:

- your are not required to terminate each Kotlin statement with a semicolon.
- `package` and `import` in both languages are similar
- The Java keyword `extends` is just a colon (`:`) in Kotlin

Android

```java
package edu.gvsu.cis.trip;

import androidx.appcompat.app.AppCompatActivity;
import android.os.Bundle;

public class LoginActivity extends AppCompatActivity {
    @Override
    protected void onCreate (Bundle saveInstanceState) {
        super.onCreate(saveInstanceState);
        // more code not shown
      }
  }
```

Kotlin

```kotlin
package edu.gvsu.cis.trip

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle

class LoginActivity: AppCompatActivity() {
    override fun onCreate(saveInstanceState: Bundle?) {
        super.onCreate(saveInstanceState)
      }
  }
```

Listing 169: Inheritance in Java and Kotlin

- While the `@Override` annotation in Java is optional, the keyword `override` in Kotlin is mandatory when a method overrides its corresponding method in the parent. Otherwise, you will get a compiler error

  ```
  'onCreate' hides member of supertype 'AppCompatActivity' ...
  ```

- Declaring a function in Kotlin is akin to having `fun`.
- Identifier declaration in Kotlin begins with its name and its type is written after a colon. For instance, compare the declaration of the `Bundle` parameter of the `onCreate` method.

  ```
  // Java declaration
  ... (Bundle savedInstanceState)

  // Kotlin declaration
  ... (savedInstanceState: Bundle?)
  ```

  The question mark suffix tells the compiler that the parameter is nullable. This topic will be discussed in the next section.

Now, why is the parent class name (`AppCompatActivity`) ends with parentheses? Recall that when using inheritance in Java, the constructor of a child class is responsible for invoking its parent constructor via `super()` (or `super(`*arg1*, *arg2*, ...`)` if the parent constructor requires some input

parameters). So the parentheses at line 6 of the Kotlin snippet invokes the `AppCompatActivity` default constructor.

```
6  class Parent {
7    public Parent (float v) {
8      // more code here
9    }
10 }
11
12 class Child extends Parent {
13   public Child(Boolean x, float y) {
14     super(y);
15     // more code here
16   }
17 }
```

The equivalent classes in Kotlin require only two lines of code:

```
18 // Without "open", Parent is a final class
19 open class Parent (val v : Float)
20
21 class Child (val x : Boolean, val y : Float) : Parent(y)
```

Here are a few other "shortcuts" in Kotlin:

- Java `instanceof` = Kotlin `is`
- Java `x.getClass()` = Kotlin `x.javaClass`
- Java `MyClass.class` = Kotlin `MyClass::class.java`
- Java `MyClass.this` = Kotlin `this@MyClass`

B.1.1 Constructors

Java programmers are familiar with default constructors and user-defined constructors. Kotlin introduces three additional concepts:

- *Primary* constructors: those defined at the class header
- *Secondary* constructors: those defined inside the class body exactly like Java constructors, except the class name for the constructor is replaced with the Kotlin keyword `constructor`.
- Initializer blocks which works in conjunction with the primary constructor.

A Kotlin class can have at most one primary constructor and one initializer block, but it can as many secondary constructors as you wish.

A constructor that initializes the instance variables of the class from the input parameters can be written using Kotlin primary constructor without the associated initialization block.

```
// Java
public class Rational {
  private int num, denum;

  public class Rational (int n, int d) {
     num = n;
     denum = d;
  }
}
```

The following one-line class declaration performs all the work of the above Java constructor. The expression between the parentheses is the primary constructor. The following listing show several variations of writing primary constructors for a rational number with two attributes: `num` and `denum`.

```
// Kotlin: primary constructor
class Rational constructor (n: Int, d: Int) {
  val num = n
  val denum = d
}

class Rational (n: Int, d: Int) {  // "constructor" keyword omitted
  val num = n
  val denum = d
}

class Rational (val num: Int, val denum: Int)
```

When additional logic is needed besides field-by-field assignments, Kotlin provides the initializer block that works *in conjunction* with the primary constructor:

```
class Rational (val num: Int, val denum: Int) {
  val scaleFactor : Int
  val num : Int
  val denum : Int

  init { /* initializer block */
     this.num = num
     this.denum = denum
     if (denum != 0)
        scaleFactor = gcd (num, denum)
     else
        throw RunTimeException ("Denominator should not be zero")
  }

  private fun gcd (a: Int, b: Int) : Int {
    // more code here
  }
}
```

Alternatively, you can use a secondary constructor to perform the same task above

```
class Rational  {
  val scaleFactor : Int
  val num : Int
  val denum : Int

  // Secondary constructor
  constructor (num: Int, denum: Int) {
    this.num = num
    this.denum = denum
    if (denum != 0)
      scaleFactor = gcd (num, denum)
    else
      throw RunTimeException ("Denominator should not be zero")
    }
  }

  private fun gcd (a: Int, b: Int) : Int {
     // more code here
  }
}
```

B.1.2 Static Members

Unfortunately, Kotlin does not recognize the `static` keyword as Java does. Declaring static fields or function in Kotlin requires a different approach: *companion objects*. In Kotlin, the keyword `object` itself performs two simultaneous actions: defines a class and also creates an instance of the class.

Recall that `static` a field/method in a Java class causes the compiler to create a single copy of the field/method which can be accessed without first instantiating any object of the class. Essentially, the lifetime of a static field/method is similar to that of a global variable/function in a C program, but it is globally visible only through the containing class. To mimic this special behavior in Kotlin we can use the `object` keyword within the scope of a class. By declaring an `object` construct within a class, the object is created irrespective of object creation from the class itself. However, the object created using the above technique lacks permissions to access non-public members of the containing class. To solve this problem, Kotlin provides the `companion` keyword. Readers with C++ background may associate `companion` with C++ `friend`.

One of the common use cases in Android is provided a static method for creating fragments. Hence the sample code below:

```
public class SampleFragment {   // Java

protected SampleFragment() { /* some code here */ }

  public static SampleFragment newInstance() {
    SampleFragment frag = new SampleFragment();
    // more code here
    return frag;
  }
}
```

```
class SampleFragment {   // Kotlin
  protected constructor() { /* some code here */ }

  companion object {
    fun newInstance() : SampleFragment {
      val frag = SampleFragment()
      // more code here
      return frag
    }
  }
}
```

B.1.3 Data Class

In many occasions you have to create POJO in Java mainly for storing your data. Kotlin provides a convenient syntax for declaring such classes

```
// Declare the class (only one line required)
data class AtomicElement (var name: String, val weight: Int)

// Use the class
val oxy = AtomicElement("Oxygen", 16)
val oxz = AtomicElement("Oxygen", 16)
val nitrog = AtomicElement("Nitrogen", 14)
println (oxy == oxz)     // true: compare content
println (oxy === oxz)    // false: compare address
nitro.name = "Nitro"     // OK
nitro.weight = 2 * 7     // ERROR: no setter
```

The above one line declaration does so much behind the scene. In addition to providing getter (and/or setter) functions, the compiler also generates all the `Object` related methods for you: `toString()`, `equals()`, `hashCode()`, etc. Also notice that data classes must be declared using a primary constructor.

B.1.4 Implementing an Interface

B.1.5 Child Class Constructor

Kotlin

```
constructor { }

init { }
```

B.2 Variable Declarations

Java provides both basic primitive data types (`int`, `float`, `double`, ...), their corresponding wrapper classes (`Integer`, `Float`, `Double`, ...) and automatic *boxing* and *unboxing* between them. Kotlin does not want to deal with this dual representations of these basic types, choose to support only the class types.

Kotlin compilers have the capability to infer a type of a variable based on its context:

```
var lineCount = 0                // inferred type is Int (mutable)
var population = 2_000_000_000L  // inferred type is Long (mutable)
val gravityAccel = 9.8f          // inferred type is Float (immutable)
val gravityAxxel = 9.8           // inferred type is Double (immutable)

var avgVisitor : Float
var numVisitor : Long = 0        // explicit type is Long (mutable)
```

When the type cannot be inferred from the current context, you must provide its data type as in declaring `aveVisitor` above.

Declaring an identifier with `val` makes it *immutable*, you can't change its value after its *first initialization*.

Unlike Java, uninitialized variables in Kotlin are not automatically set to default value (0, 0.0, null, false). The first attempt to access the content of an uninitialized variable is flagged as a compile error. Referring to the variables `numVisitor` and `avgVisitor` above:

```
print ("Yesterday we had " + numVisitor + " visitors"); // OK
// The following line uses string interpolation
print ("Yesterday we had ${numVisitor} visitors");      // OK
print ("The average daily visitor is " + avgVisitor);   // compile-error
```

B.3 Nullable Types and the Elvis Operator

Java `NullPointerException` (NPE) is one of major sources of headache for Android and Java programmers. They spend a significant amount of time on debugging the cause of NPEs. They discovered the issue after the fact, at run-time. Kotlin saves those precious programmers' hours by detecting the issue much earlier at compile time. When used properly, a `null` pointer gives us so many benefits. Can you imagine implementing a tree data structure without having `null`?

Identifiers declared with a data type that ends with a question mark are *nullable*, they can hold a null value at run-time. Otherwise, they are non-nullable types. For instance, the `Bundle` parameter of the `onCreate` method should be declared as a nullable type, because at runtime it can take a null value. An attempt to assign `null` to a non-nullable type results in a compile error

```
var interClickTime = Double?    // nullable
var interClockTime = Double     // non-nullable

interClickTime = null // OK
interClockTime = null // compile error
```

The real benefit of nullable variables is demonstrated using the following examples:

```
var noMsg : String? = null
var funMsg : String = "Kotlin is fun"
println (noMsg)                              // output: null
println (noMsg?.toUpperCase())               // safe call oper output: null
println (noMsg ?: "R U Sure?")               // Elvis oper     output: R U Sure?
println (noMsg?.toUpperCase() ?: "Nothing")  // output: Nothing

println (funMsg.toUpperCase())               // output: KOTLIN IS FUN
```

Had lines 40–41 above written in Java, they would have thrown a `NullPointerException`. The two special operators introduced above are:

- The "safe call" operator (`?.`) which essentially combines nullity check and function call in one expression. If the pointer is null, the rest of the expression evaluates to null. Otherwise it performs the operation using the non-null value as usual

- The "Elvis" operator[1] (`?:`) which can be viewed as a "degenerate" ternary operator in C or Java

Android

```
1 System.out.println (x % 2 == 0 ? "Even" : "Odd")
2 System.out.println (aRef == null ? "Text when null" : "Other text")
3
4 // when the "true" action evaluates to nothing
5 System.out.println (aRef == null ? {/*noop*/} : "Other text")
```

but with the "true" action **omitted**.

B.3.1 Using `lateinit`

Watch out for memory leak when using lateinit for View classes!

B.4 Lambda Expressions

Lambda expressions are supported by both Java 8 and Kotlin. However, their primary use in Java seems to be targeted for replacing anonymous inner classes having only one method with less bulkier code. Refer to Appendix **??** for such an example.

The syntax used by both languages shows only subtle differences. Java uses curly braces to enclose only the function body, while Kotlin encloses the entire expression. In addition, parameters in Kotlin are not enclosed inside parentheses. For instance, a lambda expressions that takes two parameters of type `Char` and integer are shown in the following snippet:

```
6  // Java
7  (Char  code,  int value) -> { /* body */ }
8
9  // Kotlin
10 { code : Char, value : Int -> /* body */ }
```

Let's say we have a TreeMap for storing the numerical value of each character used in roman numerals (I, V, X, ...) and we want to print all its entries:

[1] If you look at the operator with your head turned sideways, you can visualize the `?` resembling Elvis Prestley pompadour hair style and the `:` are his eyes.

```
var roman = TreeMap<Char,Int>()
roman['I'] = 1
roman['V'] = 5
roman['X'] = 10
roman['L'] = 50
roman['C'] = 100
roman['D'] = 500
roman['M'] = 1000

roman.forEach({ ch : Char, v : Int ->
  println ("Value of ${ch} is ${v}")
})
```

The entire expression can be further simplified using several Kotlin rules:

- A lambda expression can be moved out of the parentheses when it is the last argument:

```
roman.forEach() { ch : Char, v : Int -> println ("Value of ${ch} is ${v}") }
```

- When lambda expression is the only expression to a function call, the empty parentheses can also be removed:

```
roman.forEach { ch : Char, v : Int -> println ("Value of ${ch} is ${v}") }
```

- The lambda parameter types can be inferred from the surrounding context:

```
roman.forEach { ch, v -> println ("Value of ${ch} is ${v}") }
```

B.5 Control Structures

The syntax for `if`, `while`, and `do-while` is exactly the same. In addition to the common `for`-loop syntax, Kotlin provides two additional forms of for-loop

- For ranges with inclusive beginning and end values (closed interval)

```
for (k in 3 .. 11) {
  /* this loop repeats 9 times */
}
```

- For ranges that iterates with a step

```
for (k in 3 .. 11 step 4) {
  /* this loop repeats 3 times (k = 3, 7, 11) */
}
```

- For ranges that iterates with a step

```
 for (k in 17 downTo 0 step 4) {
  /* this loop repeats 5 times (k = 17, 13, 9, 5, 1) */
}
```

Android

```
char ch;
String out;

switch (ch) {
    case '+':
    out = "positive";
    break;
    case '-':
    out = "negative";
    break;
    default:
    out = "other"
  }
```

Kotlin

```
var ch : Char
var out : String
/* other operation that alter ch */

when (ch) {
    '+' -> out = "positive";
    '-' -> out = "negative";
    else -> out = "other"
  }
```

- For ranges with inclusive beginning value but exclusive end value (half-open interval)

```
for (k in 3 until 11) {
  /* this loop repeats 8 times (k=3, 4, ..., 10) */
}
```

The other control flow not found in Java is Kotlin's `when`, which may look like a Java `switch` statement but with more features. The simplest form of `when` is the pure counterpart of Java/C `switch`

Notice that no `break` is required. In Kotlin `when` checks all the branches. If a match is found, *only that branch executes*.

B.5.1 Generalized "Switch"

The other common form of `when` uses *no argument*. Instead of searching for a match, this form of `when` requires each branch condition be a Boolean expression.

```
when { // <== no arg
  x in 11..19 -> println ("Teen")
  x % 2 == 0 && x >= 0 -> println ("Even")
  else -> println ("Odd")
}
```

Again, `when` will take the *first* matching condition. So, for instance when `x` is 13 the "Teen" branch will run, not the "Odd" branch.

B.6 Functions

In Java methods must be declared inside a class, but Kotlin allows functions either inside or outside a class. Similar to the format of identifier declarations where we separate the name and its type by a colon, declaring a function follows the same format.

```java
// in Java
boolean isMultiple (int value, int factor) {
  if (factor == 0)
     return false;
  else if (factor == 1)
     return true;
  else
     return value % factor == 0;
}
```

```kotlin
// in Kotlin
fun isMultiple (value: Int, factor: Int): Boolean {
  if (factor == 0)
    return true
  else if (factor == 1)
    return false
  else
    return value % factor == 0
}
```

Named arguments
Default value for parameters

B.7 Java Setters and Getters

Java objects usually provide setters and getters to access/manipulate the objects properties. However, these boilerplate methods add verbosity of your code. Using Kotlin properties, setters and getters in Java can be accessed by directly using the property name:

```java
public class Counter {
  private int data;

  public int getValue() {
    return data;
  }

  public void setValue (int v) {
    data = v;
  }

  public void inc() {
    data++;
  }
}
```

```kotlin
var z = Counter()
z.value = 100        // invoke setValue()
z.inc()
println (z.value)    // invoke getValue()
```

Named arguments
Default value for parameters

B.8 Java Setters and Getters

Java objects usually provide setters and getters to access/manipulate the objects properties. However, these boilerplate methods add verbosity of your code. Using Kotlin properties, setters and getters in Java can be accessed by directly using the property name:

```
52 public class Counter {
53   private int data;
54
55   public int getValue() {
56     return data;
57   }
58
59   public void setValue (int v) {
60     data = v;
61   }
62
63   public void inc() {
64     data++;
65   }
66 }
```

```
67 var z = Counter()
68 z.value = 100        // invoke setValue()
69 z.inc()
70 println (z.value)    // invoke getValue()
```

Bibliography

[1] Git documentation. https://git-scm.com/doc. Online; accessed 04-May-2017. 2.3.1

[2] Google I/O 2016. Introducing the awareness api, an easy way to make your apps context aware. https://youtu.be/37ia7S4Lsv4. Online; accessed 04-May-2017. 13.1

[3] Dave Abrahams. Protocol-oriented programming in swift. https://developer.apple.com/videos/play/wwdc2015/408/, 2015. WWDC 2015, Online; accessed 12-January-20017. A.11

[4] Apple. App programming guide for ios. https://developer.apple.com/library/content/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/RevisionHistory.html#//apple_ref/doc/uid/TP40007072-CH10-SW1. Online; accessed 04-May-2017. 3.1.3

[5] Greg J. Badros, Alan Borning, and Peter J. Stuckey. The cassowary linear arithmetic constraint solving algorithm. *ACM Trans. Comput.-Hum. Interact.*, 8(4):267–306, December 2001. 5.2.1

[6] TIOBE Software BV. Tiobe index for january 2017. http://www.tiobe.com/tiobe-index/, 2017. Online; accessed 10-January-2017. A.1

[7] Google Developers YouTube Channel. Styling your maps - geocasts. https://youtu.be/sLZxAB1el6w. Online; accessed 04-May-2017. 13.1

[8] Anind K. Dey, Katarzyna Wac, Denzil Ferreira, Kevin Tassini, Jin-Hyuk Hong, and Julian Ramos. Getting closer: An empirical investigation of the proximity of user to their smart phones. In *Proceedings of the 13th International Conference on Ubiquitous Computing*, UbiComp '11, pages 163–172, New York, NY, USA, 2011. ACM. 11.1.1

[9] Ben Evans and Richard Warburton. Java se 8 date and time. http://www.oracle.com/technetwork/articles/java/jf14-date-time-2125367.html, 2014. Online; accessed 04-May-2017. 6.3.2

[10] Jaikumar Ganesh and Waleed Kadous. Beyond the blue dot: New features in android location. https://www.youtube.com/watch?v=Bte_GHuxUGc. Online; accessed 04-May-2017. 13.1.1

[11] Google. Material design guidelines. https://material.io/guidelines/, 2017. Online; accessed 2-February-2017. 4.4

[12] Google. Android one. https://www.android.com/one/, 2020. Online; accessed 4-August-2020. 1.2

[13] Apple Inc. Develop in swift - fundamentals. https://books.apple.com/us/book/develop-in-swift-fundamentals/id1511184145, 2020. Online; accessed 18-January-2021. A.13

[14] Apple Inc. Swift documentation. https://developer.apple.com/documentation/swift, 2020. Online; accessed 18-January-2021. A.13

[15] Apple Inc. The swift programming language. https://developer.apple.com/library/prerelease/content/documentation/Swift/Conceptual/Swift_Programming_Language/index.html, 2020. Online; accessed 18-January-2021. A.13

[16] Ian Lake. Picking your compilesdkversion, minsdkversion, and targetsdkversion. https://medium.com/google-developers/picking-your-compilesdkversion-minsdkversion-targetsdkversion-a098a0341ebd#.99xzrf98w. Online; accessed 04-May-2017. 1.5.3

[17] Joe Myers. 4 charts that explain the decline of the pc. https://www.weforum.org/agenda/2016/04/4-charts-that-explain-the-decline-of-the-pc/, Apr 20, 2016. Online; accessed 19-December-2016. 1.1

[18] Shwetak N. Patel, Julie A. Kientz, Gillian R. Hayes, Sooraj Bhat, and Gregory D. Abowd. *Farther Than You May Think: An Empirical Investigation of the Proximity of Users to Their Mobile Phones*, pages 123–140. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. 11.1.1

[19] Kara Pernice and Raluca Budiu. Hamburger menus and hidden navigation hurt ux metrics. https://www.nngroup.com/articles/hamburger-menus/, 2016. Online; accessed 16-October-20017. 12.1.1

[20] Mark Weiser. The computer for the 21st century. *SIGMOBILE Mob. Comput. Commun. Rev.*, 3(3):3–11, July 1999. 11.1.1