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

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

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.

—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.

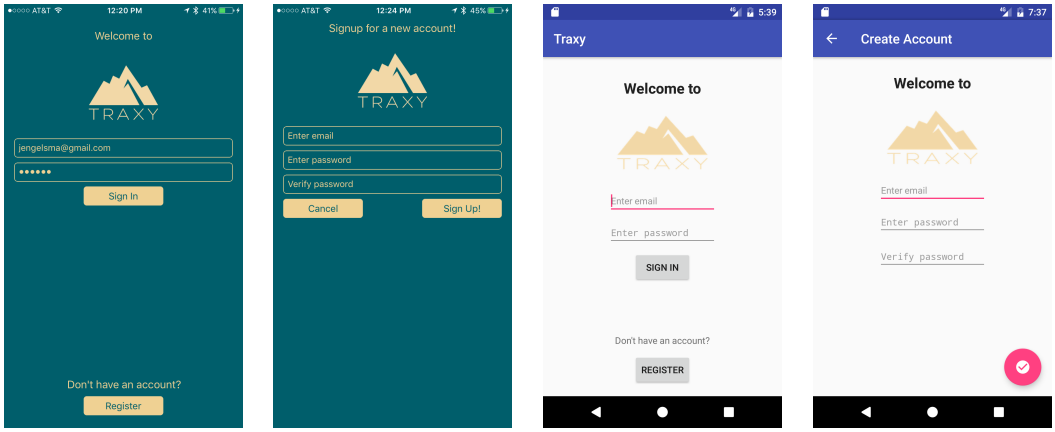
As you read your way through this book, you will incrementally learn the Android and iOS platforms side by side. Our goal is to help the reader understand the commonalities of these two mobile application platforms, as well as some of the areas in which they differ. The main vehicle that will serve to 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.

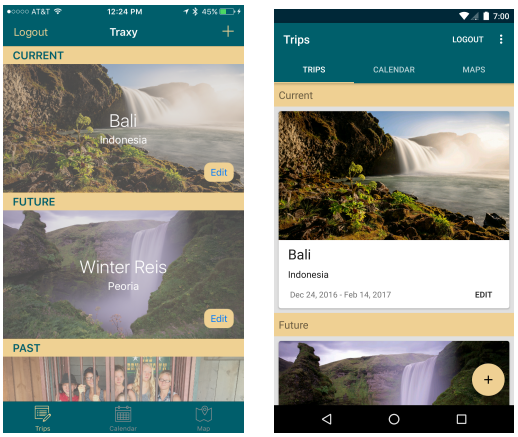
When the user taps on the plus button, the journal editor screen is displayed. The user can describe the vacation destination as well as the anticipated start / end dates. Once the new journal is fully described, the user can save it and return to the journal list screen. Eventually when the user has 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.

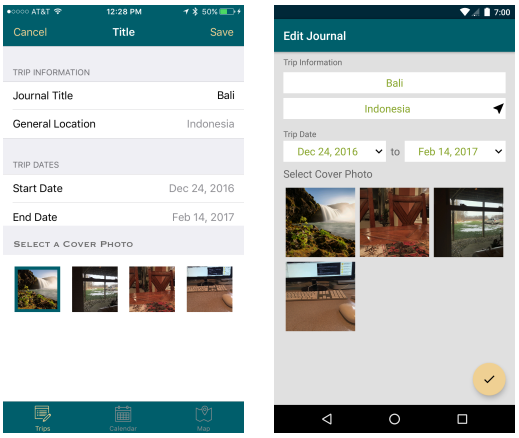
Photo and video entries can be captured within the camera via the device's camera or by selecting them from the device's library of previously captured images, as shown in Figure 2.6.

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



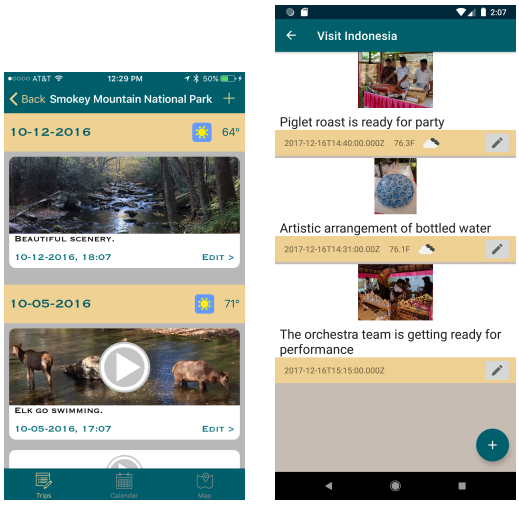
(a) (b)

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



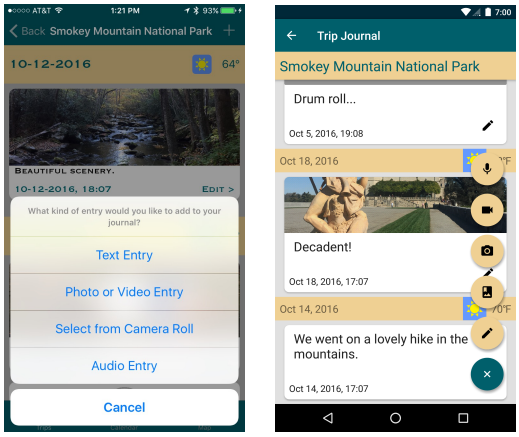
(a) (b)

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



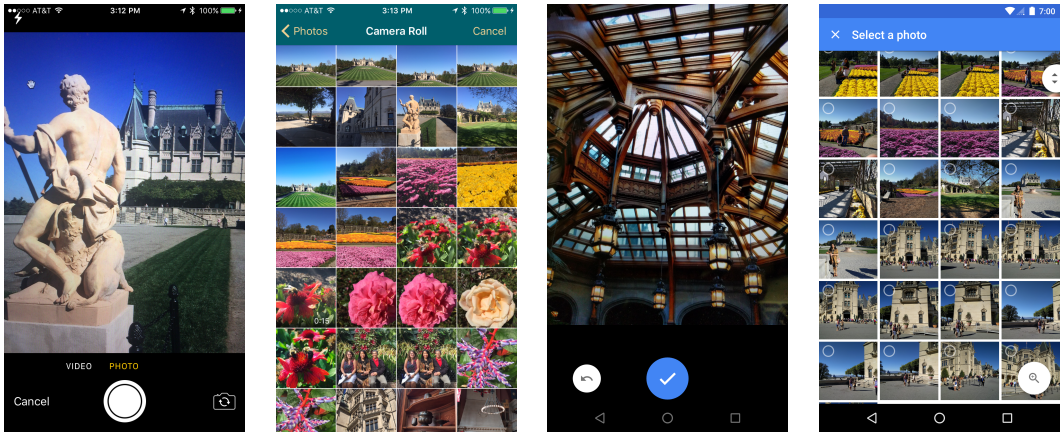
(a) (b)

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



(a) (b)

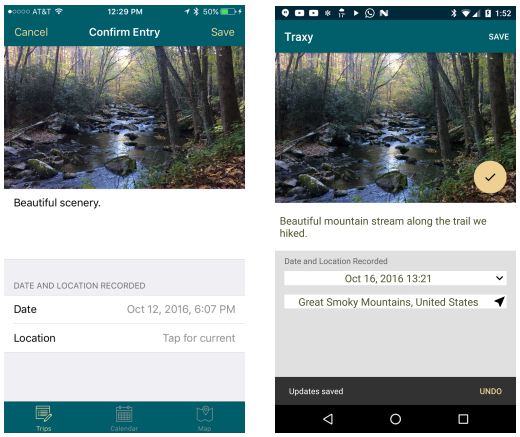
Figure 2.5: a) iOS add journal entry options. b) Android add journal entry options.



(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.

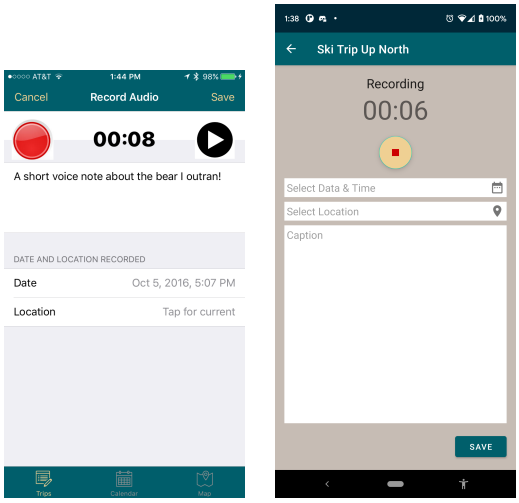


(a) (b)

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.

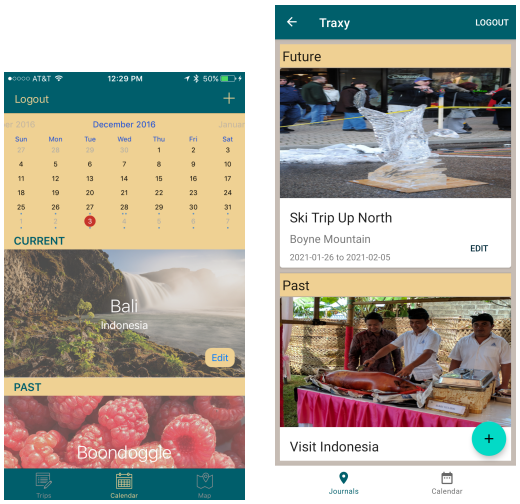
the first tab. The idea of this view is to let a user quickly locate a journal based on its date.

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.

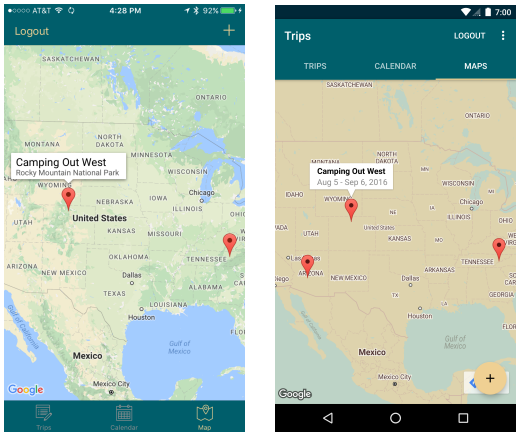
2.2 Writing Our First Mobile App

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.

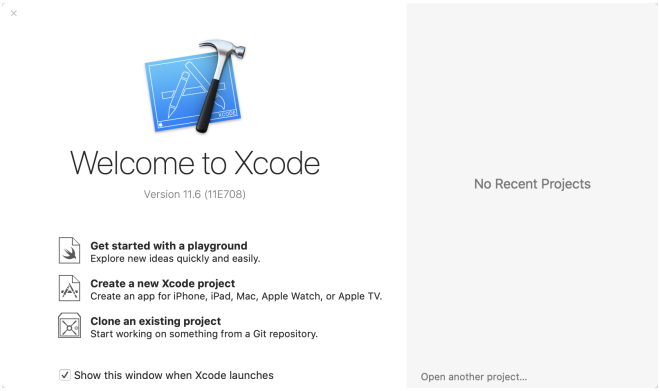


Figure 2.11: The Xcode welcome dialog.

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

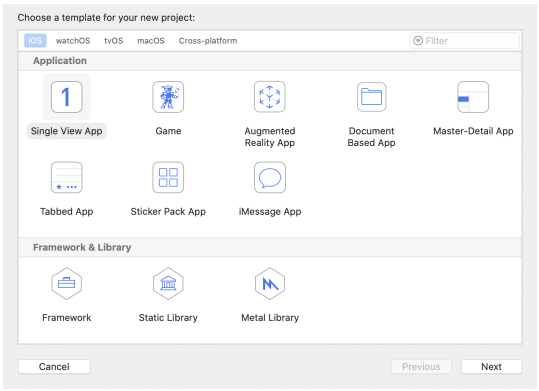


Figure 2.12: The Xcode project template selection dialog provides a variety of application styles to create. Note that we have iOS selected in the upper left of the dialog.

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.

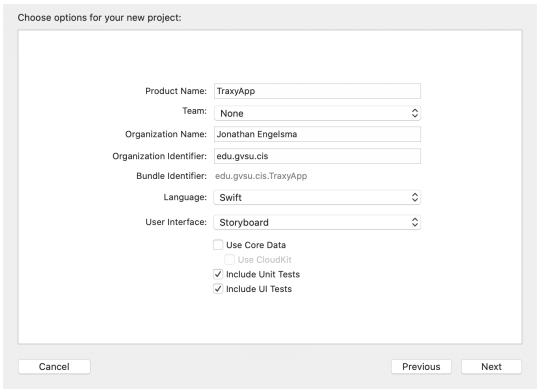


Figure 2.13: The Xcode project template selection dialog provides a variety of application styles to create.

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.

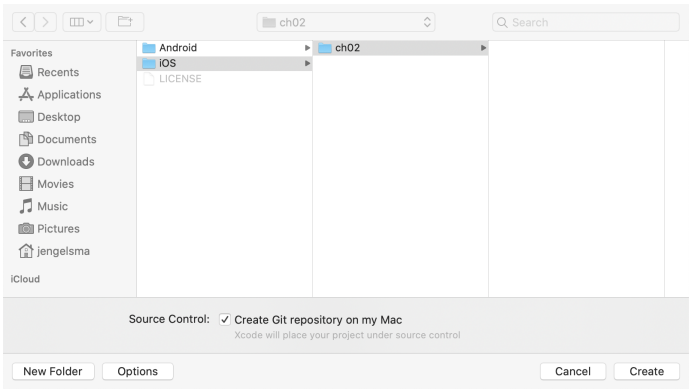


Figure 2.14: The final step of project creation in Xcode is to specify the directory in which the project folder will be created.

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.

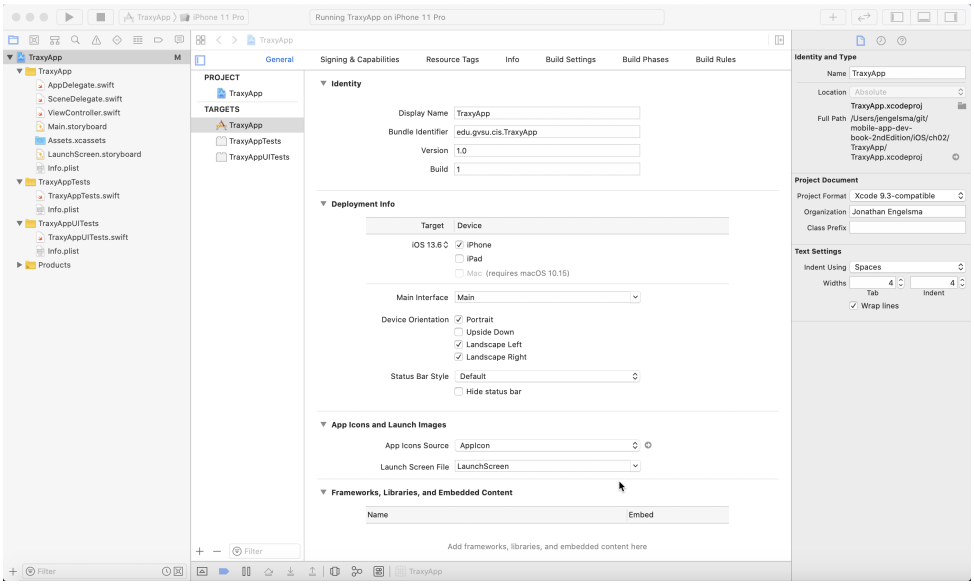


Figure 2.15: The new Traxy app has been created and loaded in Xcode, and we are ready to start customizing it.

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

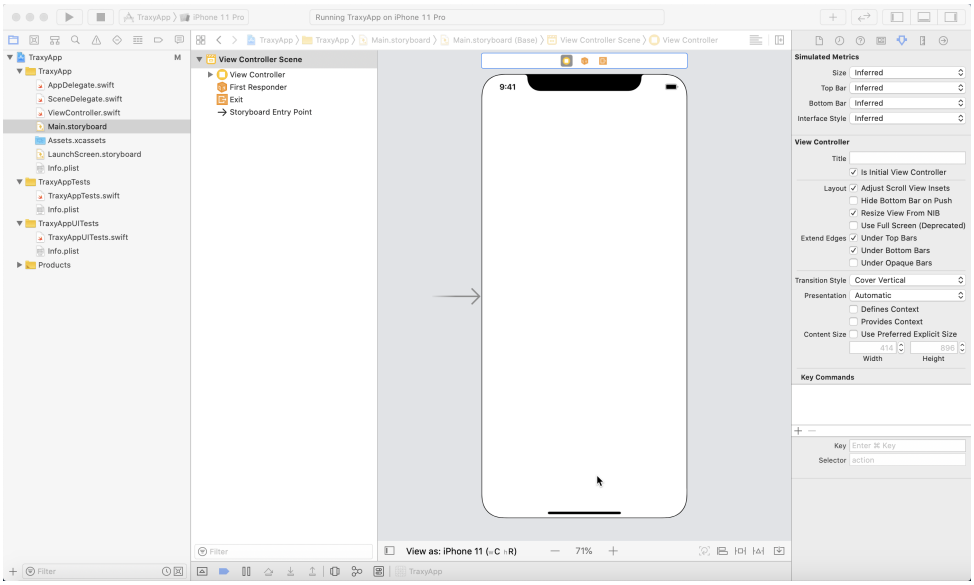


Figure 2.16: Selecting the Main.storyboard file in the Xcode project explorer opens up Interface Builder, allowing us to edit our app's view(s).

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

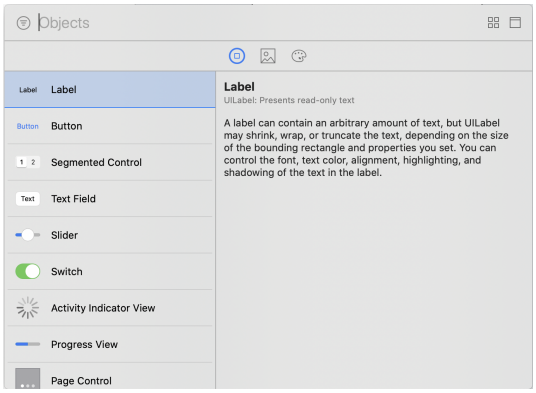


Figure 2.17: You can scroll through the Xcode object library pane to find UI controls, or you can use the search field at the top of the dialog to limit the controls displayed.

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

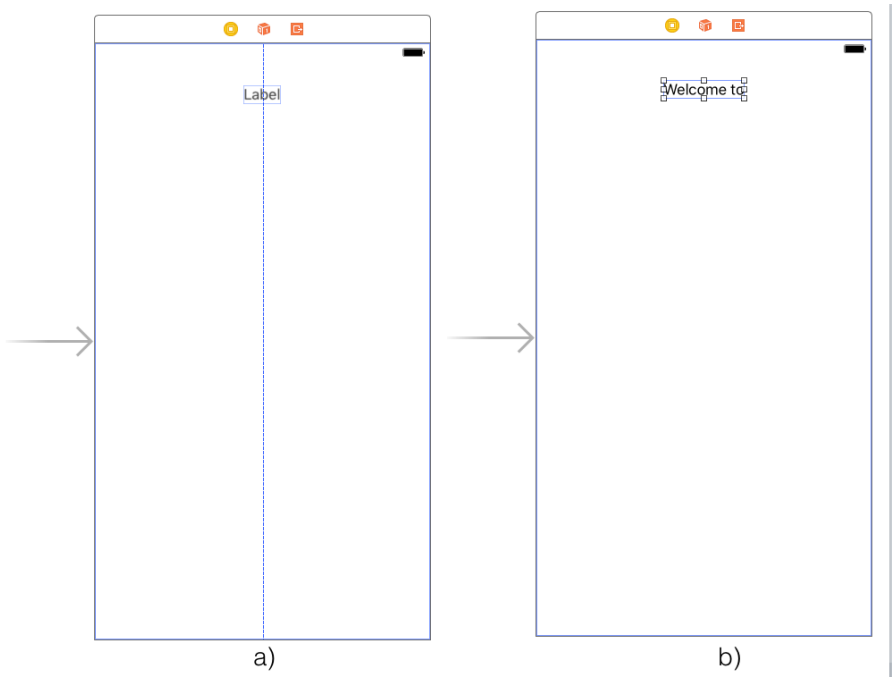


Figure 2.18: a) Drag and drop a label into the scene, centering it towards the top. b) After dropping the label, double click on it to change the text to "Welcome to".

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.

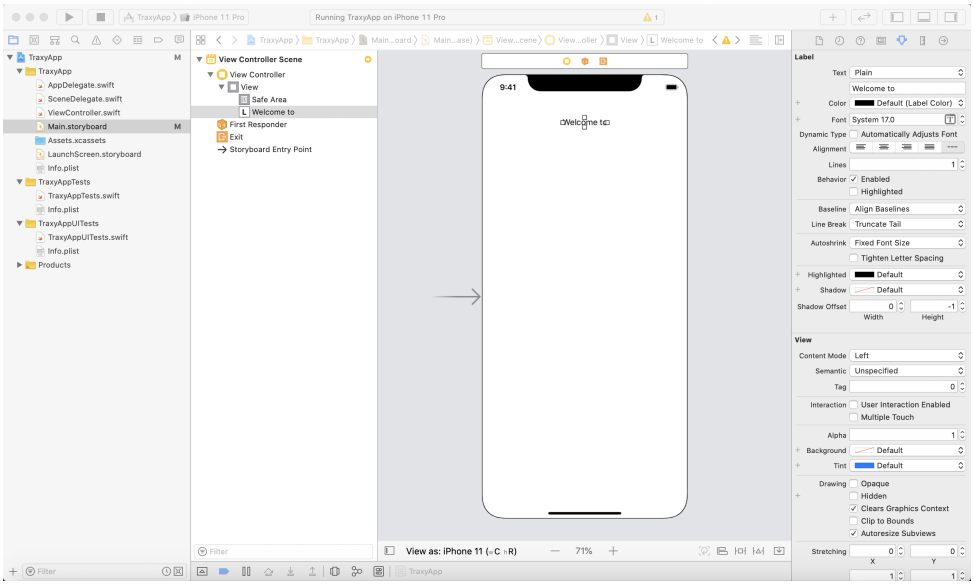


Figure 2.19: Properties of UI controls dropped into a scene can be manipulated by selecting the control, and then editing the properties in the Attribute Inspector pane displayed on the right.

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

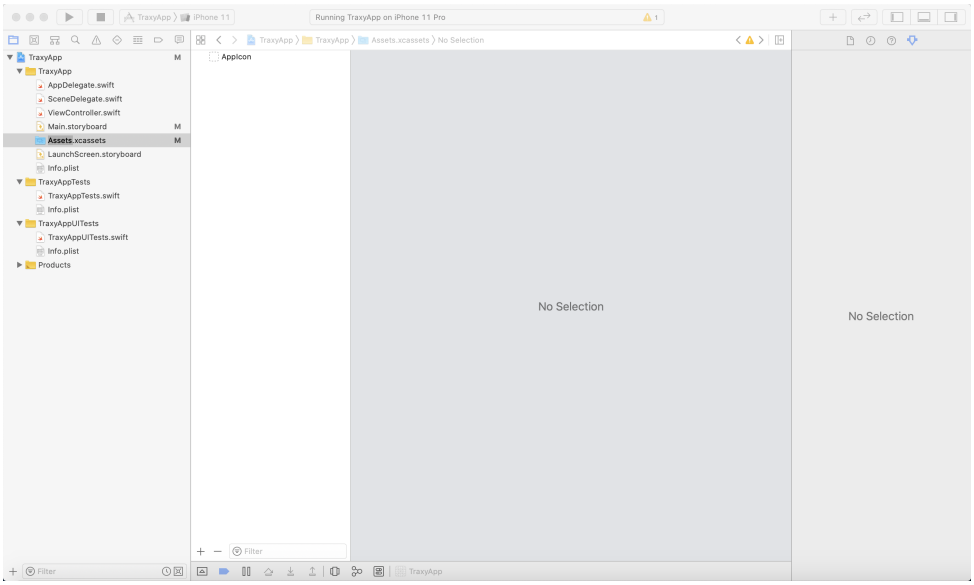


Figure 2.20: Image assets can be added to the project, by selecting the Assets.xcassets file in the Xcode project explorer.

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,

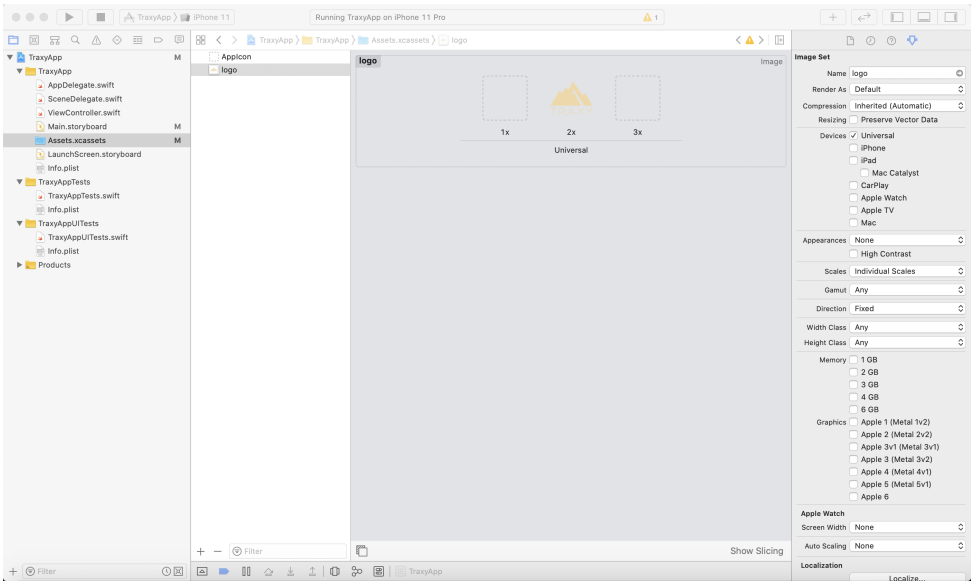


Figure 2.21: The assets catalog after adding our logo image set.

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.

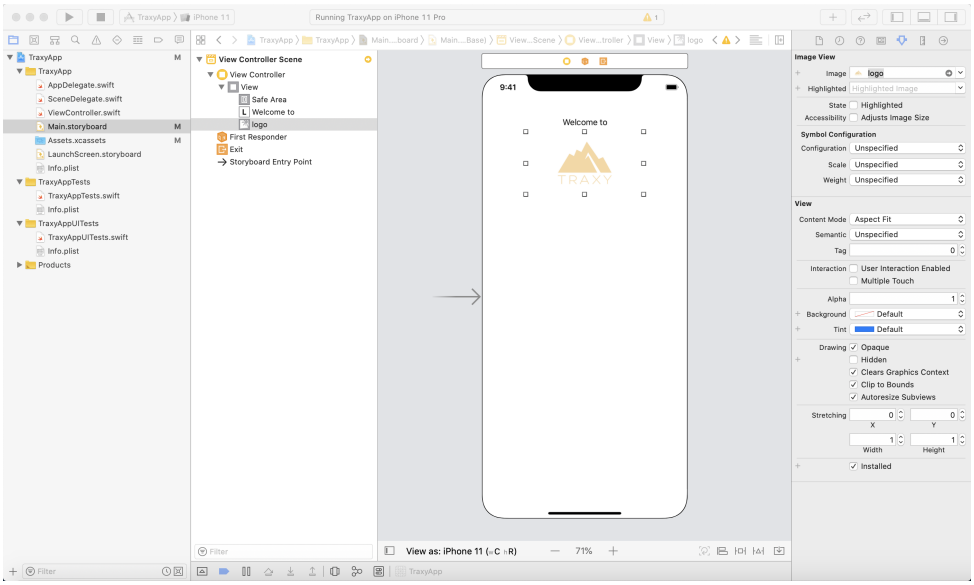


Figure 2.22: You can assign images in your asset catalog to image views in your scenes by selecting the image view and then choosing the image over in the Attribute Inspector on the right.

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`.

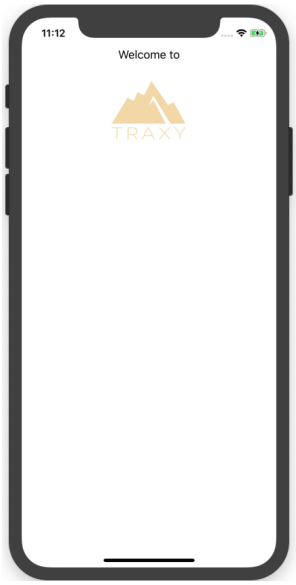


Figure 2.23: Our initial "hello world" app is complete, as shown running in the iPhone 11 simulator.

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

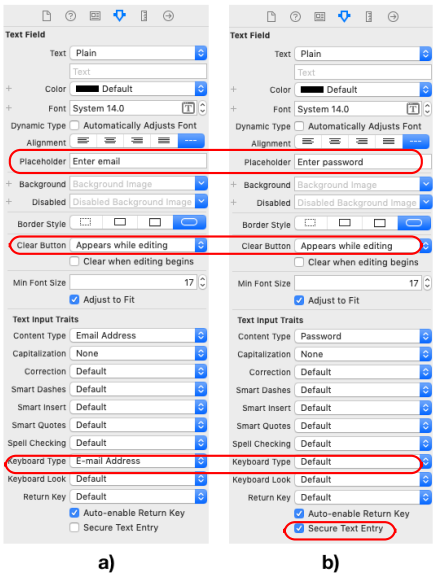


Figure 2.24: The Attribute Inspector settings for a) the email address field and b) the password field.

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

```
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,

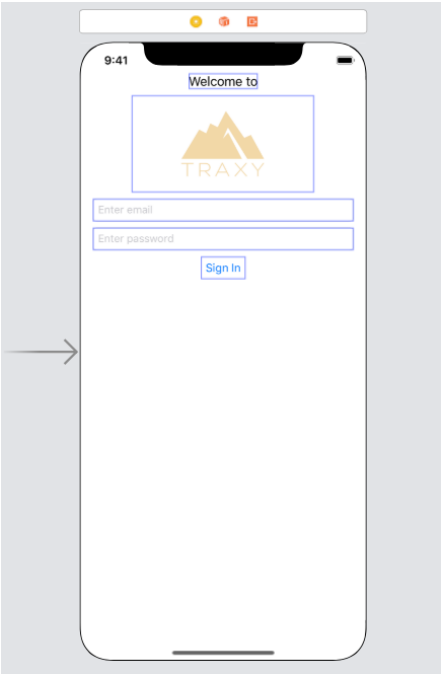


Figure 2.25: A simple login screen for the Traxy app.

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

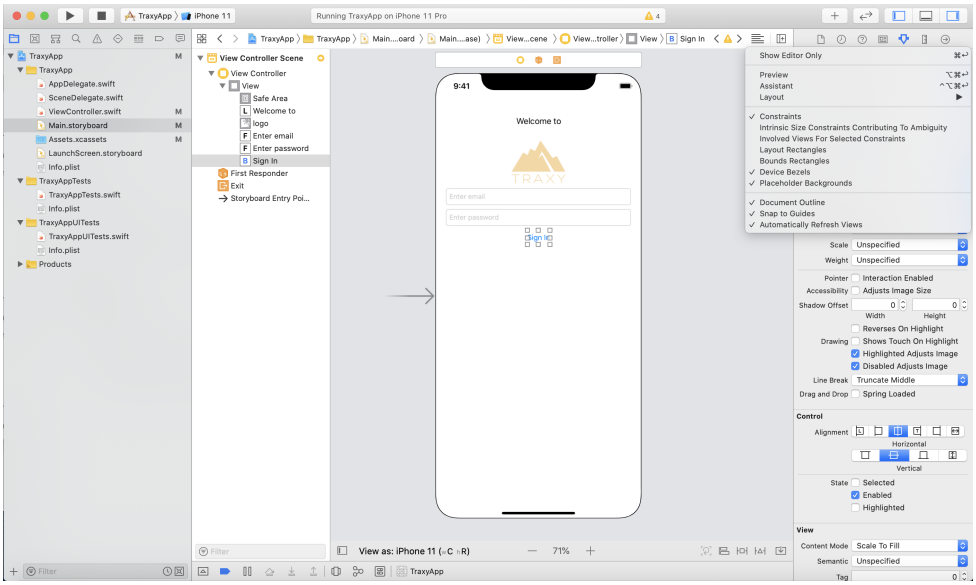


Figure 2.26: Activating the Assistant Editor view when in a storyboard is a little tricky in recent versions of Xcode. To activate, simply click the little hamburger like menu on the right or the Interface Builder tool bar.

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

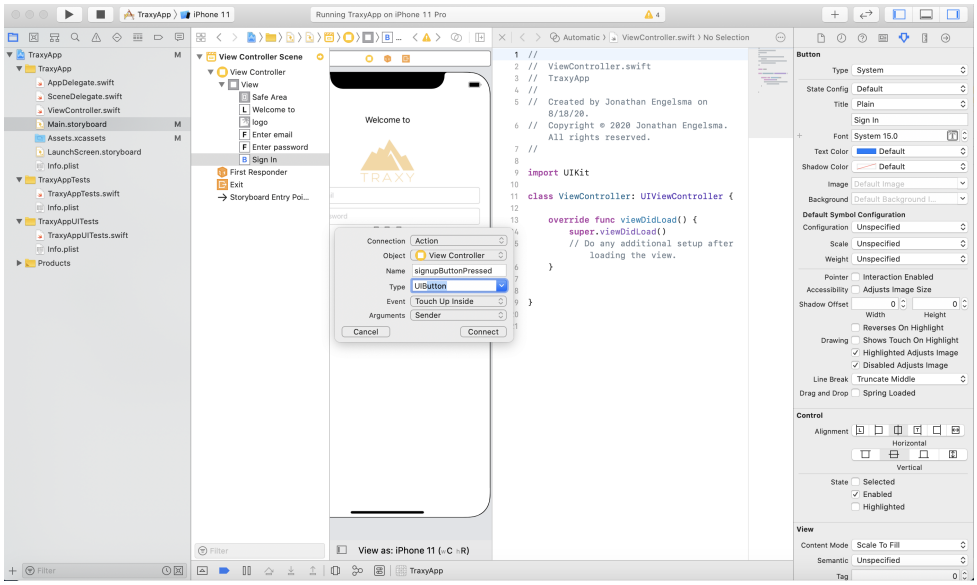


Figure 2.27: Using the assistant editor to add an IBAction to the sign in button.

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.

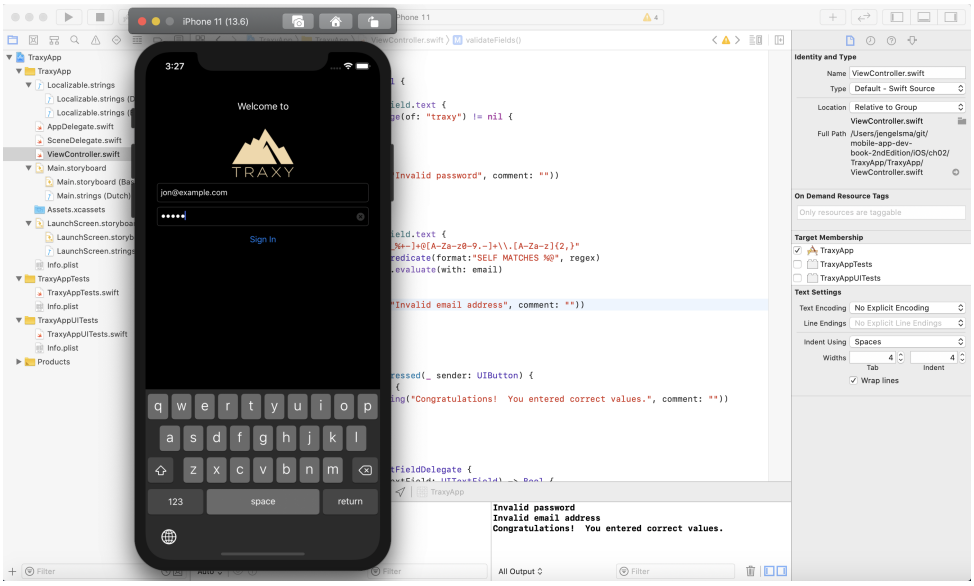


Figure 2.28: When the app is ran in the simulator, the output data generated by the print statement in the field validation logic get printed in the Debug pane on the bottom of the Xcode window.

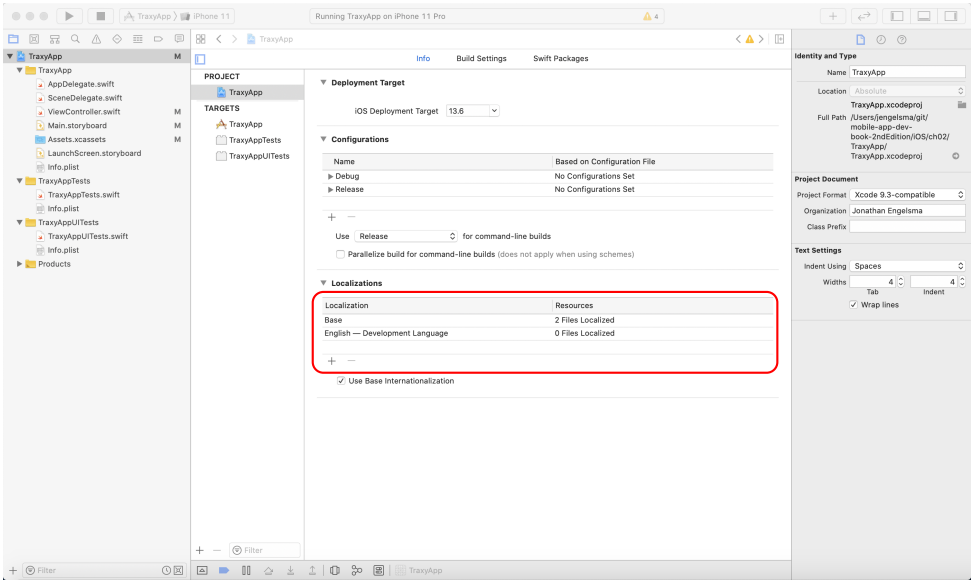


Figure 2.29: To internationalize the app, you need to add additional localizations on the Project Info screen.

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

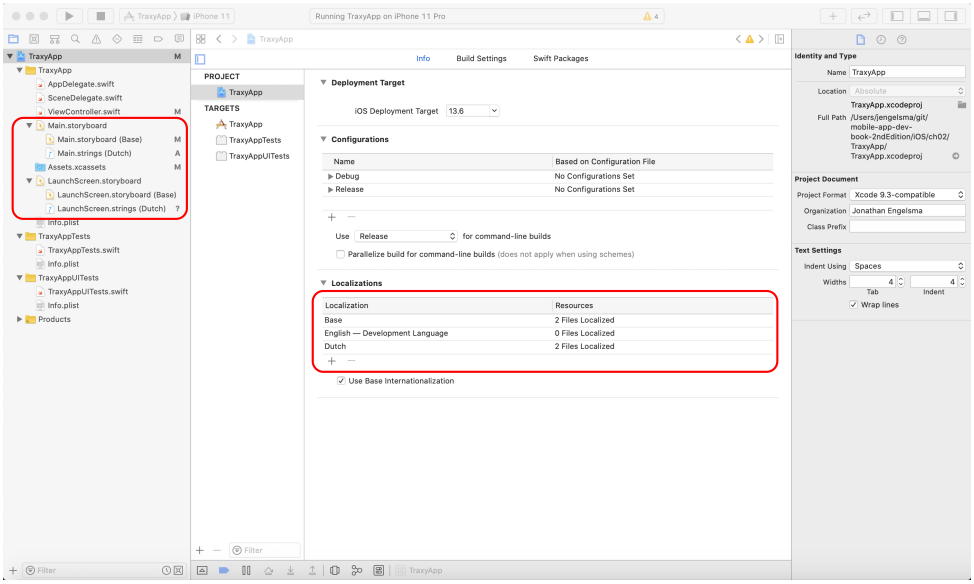


Figure 2.30: Once you've added a localization to your project, you can add translations for any string literals in your storyboard.

```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.

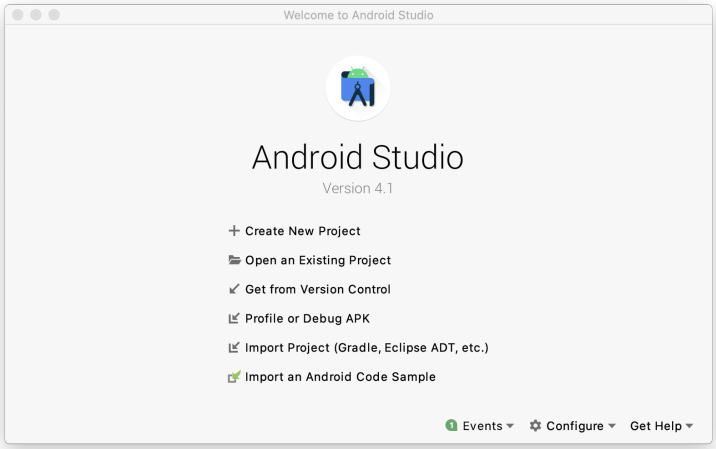


Figure 2.31: Android Studio's welcome screen. Note that if there was a previous history of projects that had been created or opened, the dialog would be split with a scrollable list of previous projects on the left.

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.

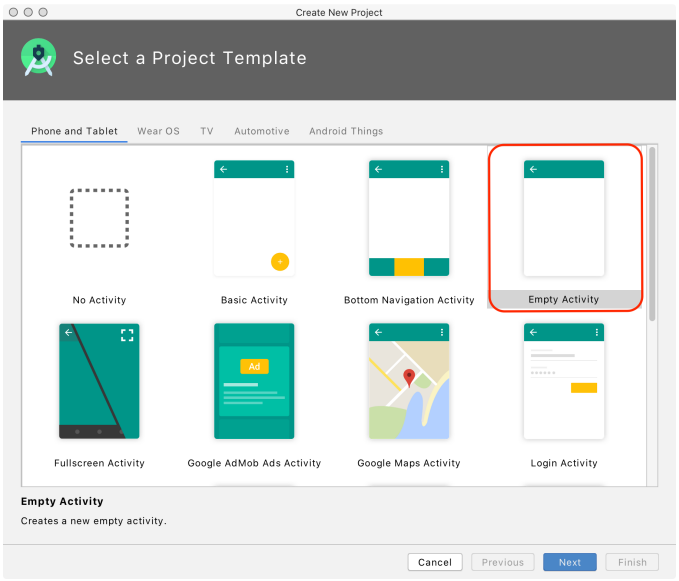


Figure 2.32: Android Studio's Starter Activity dialog.

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

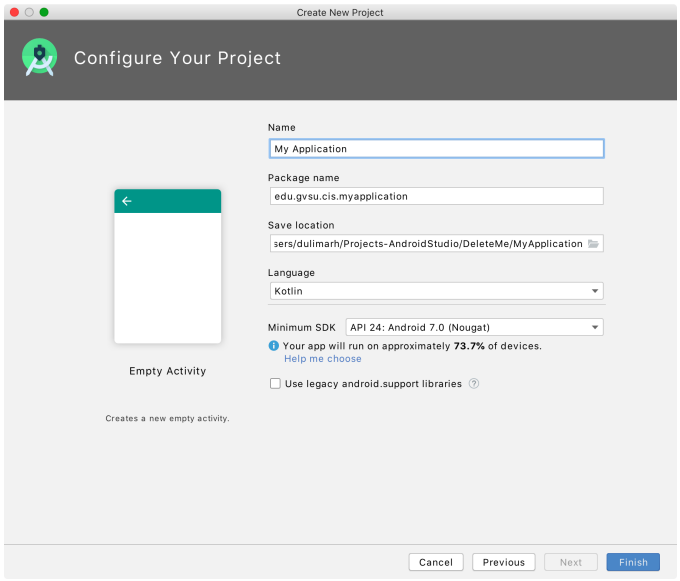


Figure 2.33: Android Studio's new project dialog.

(`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`

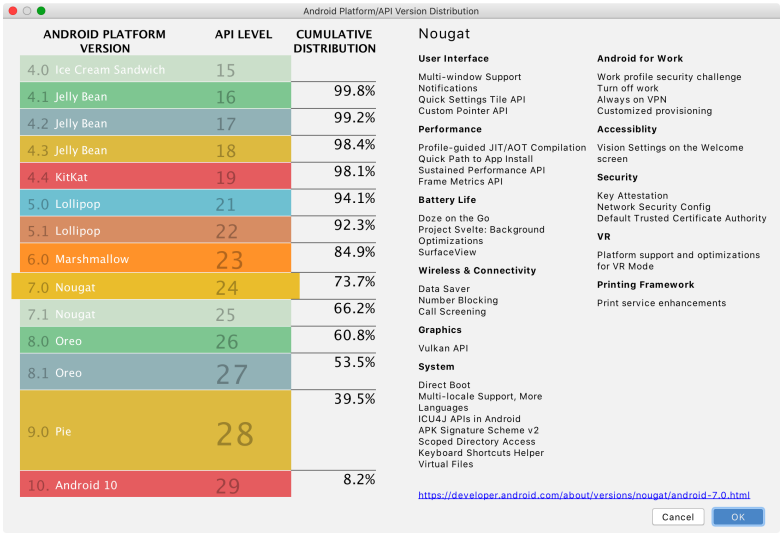


Figure 2.34: API Level distribution percentages

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

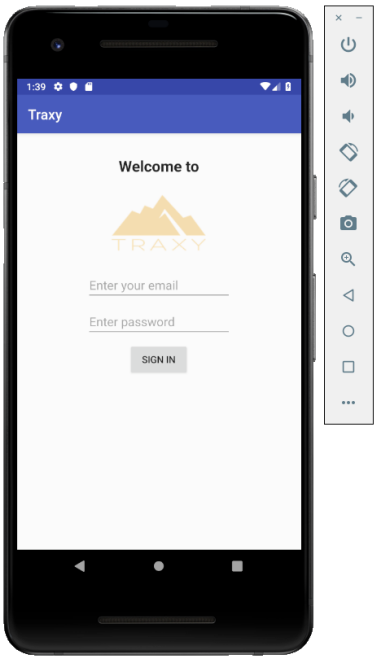


Figure 2.35: Traxy Login Screen

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.

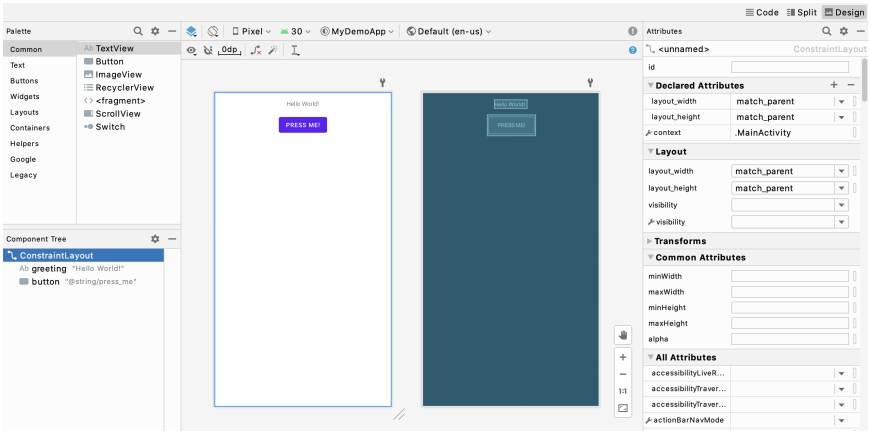


Figure 2.36: Layout Editor

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.

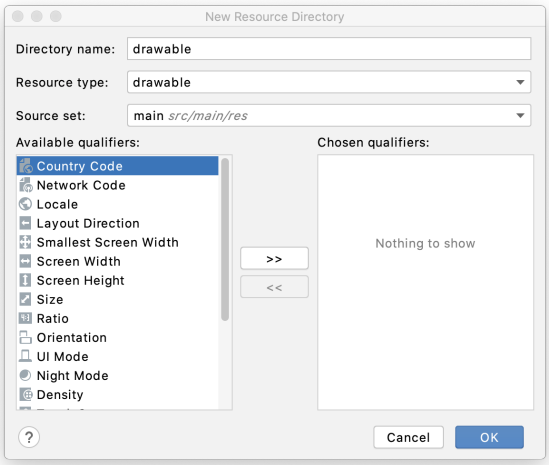


Figure 2.37: Creating a new resource directory

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

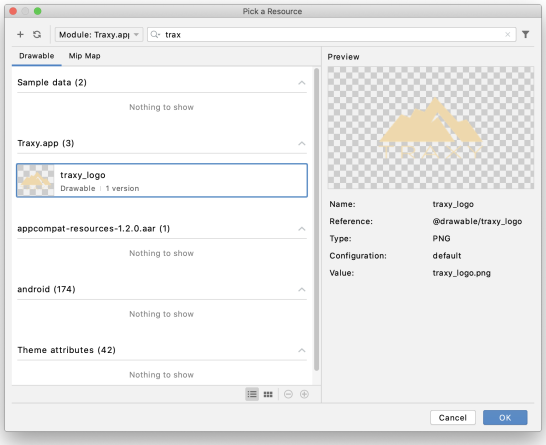


Figure 2.38: Selecting an image source

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.

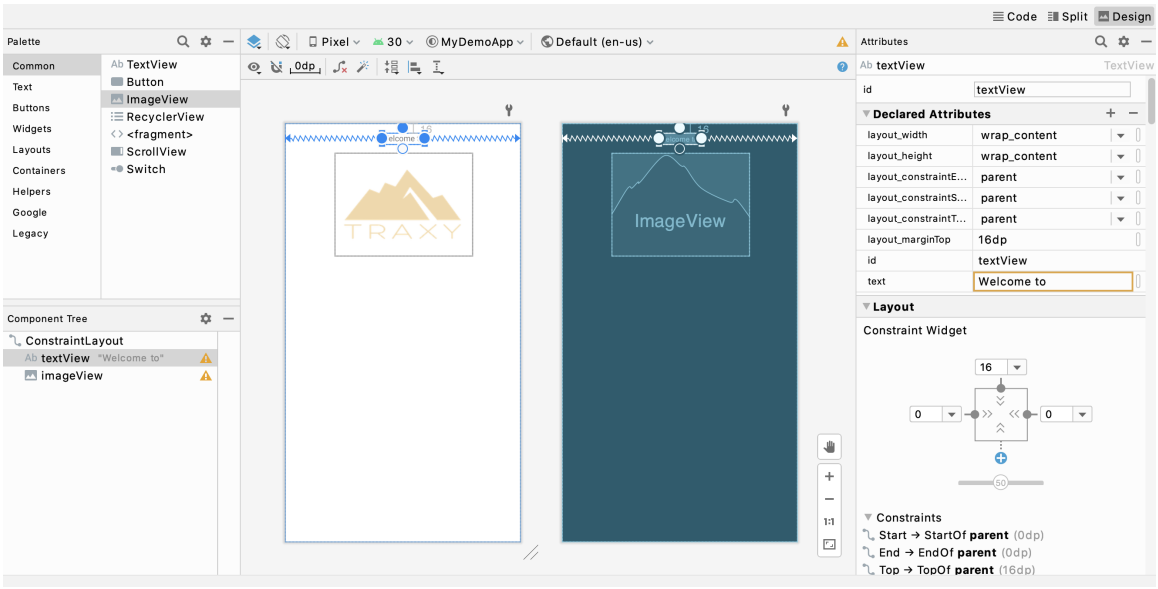


Figure 2.39: Constraints associated with widgets

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).

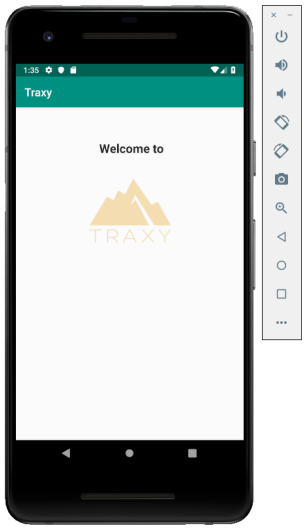


Figure 2.40: Partially Completed Traxy Login Screen

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.

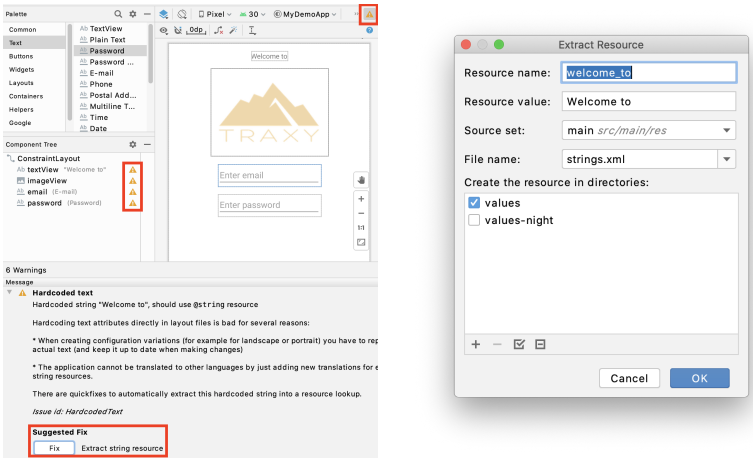


Figure 2.41: Extracting String Resource

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)

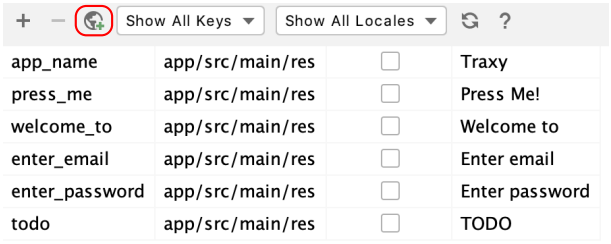


Figure 2.42: Adding a New Locale

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.