



Program the Internet of Things with Swift for iOS

Learn How to Program Apps for the
Internet of Things

—
Second Edition

—
Ahmed Bakir

Apress®

Program the Internet of Things with Swift for iOS

Learn How to Program Apps for the Internet of Things

Second Edition

Ahmed Bakir

Apress®

Program the Internet of Things with Swift for iOS: Learn How to Program Apps for the Internet of Things

Ahmed Bakir
devAtelier, Tokyo, Japan

ISBN-13 (pbk): 978-1-4842-3512-6
<https://doi.org/10.1007/978-1-4842-3513-3>

ISBN-13 (electronic): 978-1-4842-3513-3

Library of Congress Control Number: 2018964570

Copyright © 2018 by Ahmed Bakir

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image, we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the author nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Aaron Black
Development Editor: James Markham
Coordinating Editor: Jessica Vakili

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-3512-6. For more detailed information, please visit www.apress.com/source-code.

Printed on acid-free paper

Dedicated to my mother, Layla Bakir, who shared with me her love of teaching and enduring optimism

Table of Contents

About the Author	xi
About the Technical Reviewer	xiii
Acknowledgments	xv
Introduction	xvii
Part 1: Building Health Apps for the Internet of Things.....	1
Chapter 1: Laying the Foundation for Your First IoT App.....	3
Learning Objectives	4
Setting Up the Project.....	5
Linking Your Apple Developer Account to Xcode	13
Building an Adaptive User Interface.....	17
Renaming Classes from the Base Template	19
Laying Out the User Interface.....	22
Applying Auto Layout Constraints.....	24
Customizing the Appearance of Items.....	27
Resolving Auto Layout Issues.....	37
Connecting the Storyboard to Your Code	41
Defining Interface Builder-Compatible Properties and Methods (Actions)	42
Using the Connection Inspector to Make the Final Storyboard Connections.....	45
Summary.....	50

TABLE OF CONTENTS

- Chapter 2: Using Core Location to Build a Workout Tracking App..... 51**
 - Learning Objectives 52
 - Configuring Your Project for Background Location Activity..... 52
 - Requesting Location Permission..... 58
 - Checking for Hardware Availability..... 62
 - Responding to Changes in Location Permission Status 64
 - Requesting Location Updates 72
 - Responding to Location Updates 74
 - Displaying Location Data on a Map..... 81
 - Using the Codable Protocol for File-Based Data Storage 82
 - Displaying Saved Locations on a Map 93
 - Summary..... 96

- Chapter 3: Using Core Motion to Add Physical Activity Data to Your Apps 97**
 - Learning Objectives 98
 - Requesting Motion Permission from the User 99
 - Receiving Real-Time Step Count Updates from the iPhone's Pedometer 104
 - Updating the User Interface..... 108
 - Stopping and Pausing Pedometer Updates 111
 - Getting Activity Type..... 113
 - Handling Altimeter Updates 118
 - Summary..... 121

- Chapter 4: Using HealthKit to Securely Retrieve and Store Health Data..... 123**
 - Learning Objectives 124
 - Requesting HealthKit Permission..... 125
 - Writing Data to HealthKit..... 132
 - Understanding How HealthKit Represents Data 132
 - Creating and Saving HealthKit Samples 133
 - Reading Workout Data from HealthKit 143
 - Using a Table View Controller to Display Data 146
 - Summary..... 160

Part 2: Building Your Own Internet Things	161
Chapter 5: Building Arduino-Based Peripherals	163
Learning Objectives	164
Building the Wireless Door-Sensor Hardware	166
Part List	166
Assembling the Hardware	168
Writing an Arduino Solution (Program).....	178
Setting Up the Arduino Programming Environment.....	179
Using GPIO to Monitor Input Pins and Control Output Pins.....	184
Calculating Battery Status	187
Running and Monitoring the Arduino Solution.....	188
Summary.....	191
Chapter 6: Building a Bluetooth LE Hardware Companion App	193
Learning Objectives	193
A Quick Introduction to Bluetooth LE	195
Adding Bluetooth Functionality to an Arduino Solution.....	197
Installing the ESP32_BLE_Arduino Library for Bluetooth Communication	198
Setting Up the Arduino As a Bluetooth Peripheral	201
Sending Data Updates via Bluetooth LE	208
Using Core Bluetooth to Communicate with Bluetooth LE Devices.....	210
Setting Up the IOTHome Project	211
Setting Up the App As a Central Manager.....	216
Connecting to a Bluetooth LE Peripheral	222
Monitoring Characteristic Updates	226
Monitoring Updates While the App Is in the Background	229
Summary.....	233

TABLE OF CONTENTS

- Chapter 7: Setting Up a Raspberry Pi and Using It As a HomeKit Bridge 235**
 - Learning Objectives 236
 - Setting Up the Raspberry Pi HomeKit Bridge 237
 - Putting Together the Hardware 237
 - Bootstrapping the Raspberry Pi 241
 - Installing HomeBridge 248
 - Configuring HomeBridge to Read Data from a Temperature Sensor 253
 - Configuring HomeBridge to Connect to a Bluetooth LE Accessory 255
 - Connecting to Your New HomeKit Bridge 261
 - Summary 266

- Chapter 8: Building a Web Server on a Raspberry Pi 267**
 - Learning Objectives 268
 - Creating a Web Server to Share Data over HTTPS 269
 - Using Express to Expose Web Services 269
 - Reading Values from the DHT Temperature Sensor 273
 - Reading Information from Bluetooth Devices 276
 - Using HTTPS to Provide Secure HTTP Connections 283
 - Configuring the Server to Start Up with the Raspberry Pi 290
 - Connecting to Your Server from an iOS App 292
 - Setting Up the User Interface 292
 - Making and Responding to HTTPS Requests 297
 - Summary 309

- Part 3: Building Apps Using Apple’s Advanced Internet of Things Technologies 311**
 - Chapter 9: Using tvOS to Build an Apple TV Dashboard App 313**
 - Learning Objectives 314
 - Setting Up the tvOS Target 315
 - Creating the User Interface 319
 - Programmatically Styling Elements to Match the tvOS Design Language 324
 - Using Font Awesome for Font-Based Graphics 328

Adding Data Sources to the tvOS App	332
Requesting User Location.....	335
Connecting to the OpenWeatherMap API.....	339
Handling Touch Input from the Siri Remote	351
Debugging the App on an Apple TV	354
Summary.....	358
Chapter 10: Using watchOS to Build an Apple Watch App.....	359
Learning Objectives	360
Setting Up the Project	361
Building a watchOS User Interface	368
Setting Up a Table View Using the WKInterfaceTable Class.....	380
Adding Force Touch Support.....	383
Creating a New Workout Using Core Location and Core Motion	388
Using HealthKit to Populate the Workout History Table.....	401
Summary.....	405
Chapter 11: Using Face ID, Touch ID, and Keychain Services to Secure Your Apps	407
Learning Objectives	408
Setting Up the Project.....	409
Creating a Lock Screen User Interface	410
Querying for Sensor Availability	421
Using Face ID or Touch ID to Restrict Access to Features.....	424
Using Keychain Services to Secure Data	430
Using Biometrics or an App Password to Lock Keychain Items.....	437
Detecting When an App Returns to the Foreground.....	441
Summary.....	443
Index.....	445

About the Author

Ahmed Bakir is an iOS author, teacher, and entrepreneur. After starting his career as a firmware engineer, he made the mistake of telling someone at a party that he was developing iPhone apps for fun and has been inundated with work ever since. He has worked on more than 30 mobile projects, ranging from advising startups to architecting apps for Fortune 500 companies. In 2014, he published his first book, *Beginning iOS Media App Development*, followed, in 2016, by the first edition of *Program the Internet of Things with Swift for iOS*. In 2015, he was invited to develop and teach iOS development at UCSD-Extension. He is currently building cool stuff in Tokyo, Japan! You can find him online at www.devatelier.com.

About the Technical Reviewer

Massimo Nardone has more than 22 years' experience in security, web/mobile development, and cloud and IT architecture. His true IT passions are security and Android. He has been programming and teaching how to program with Android, Perl, PHP, Java, VB, Python, C/C++, and MySQL for more than 20 years.

Massimo holds a master of science degree in computer science from the University of Salerno, Italy. He has worked as a project manager, software engineer, research engineer, chief security architect, information security manager, PCI/SCADA auditor, and senior lead IT security/cloud/SCADA architect for many years. Among his technical skills are security, Android, cloud, Java, MySQL, Drupal, Cobol, Perl, web and mobile development, MongoDB, D3, Joomla, Couchbase, C/C++, WebGL, Python, Pro Rails, Django CMS, Jekyll, and Scratch.

Massimo currently works as chief information security officer at Cargotec Oyj. He is a visiting lecturer and supervisor for exercises at the Networking Laboratory of the Helsinki University of Technology (Aalto University), Finland. He holds four international patents (related to PKI, SIP, SAML, and Proxy).

Acknowledgments

Every author writes a book to tell a story, whether to entertain, educate, or provide an experience for the reader that lies somewhere in between. What I have learned from writing these past few years is that the process itself also has a story. The process of writing this book had a more elaborate story than my first two, but on this page, I'd like to thank as many people as possible who had a part in it.

First and foremost, I would like to thank my editors, Jessica Vakili and Aaron Black, whose incredible patience and support allowed me to complete this work. I truly apologize for any headaches I may have given you, but I am really proud of what we produced together.

Next, I would like to thank my original coauthors on the first edition of this book, Gheorghe Chesler and Manny de la Torriente. Beyond writing together, our experiences while collaborating greatly shaped me as a developer and provided many fond memories that I still look back on fondly today. The first edition of this book is something I will always be proud of, especially because we produced it at a time when so much of Swift and the Internet of Things was still up in the air!

Finally, I must thank my incredible family and friends, who have always supported me, even on my crazy adventure of moving to Japan. They helped me realize that I was seeking a new challenge, and they constantly encouraged me to persevere through every part of it. Thank you, all!

To everyone I may have missed, don't forget the legend of Hana no Asuka-gumi, and I'll be back again soon for our next misadventures at the Bonaventure!

Introduction

Welcome back to the Internet of Things (IoT). When my original coauthors, Gheorghe Chesler and Manny de la Torriente, and I wrote the first edition of this book in 2016, we combined what we learned from our careers, consulting projects, and dreams to put together a narrative we hoped would help future developers write their own IoT apps for iOS, using Swift. Looking at the Apress web site today, it appears that more than 30,000 people have read the book through their site alone, so I am glad we were able to reach out to at least a few readers!

However, time marches on, and since the publication of the first edition, Swift and the Internet of Things have evolved considerably. Some of the technologies we originally wanted to write about have finally become real, and others have fallen by the wayside. Most important, Swift as a language has finally begun to stabilize, and the developer community has been establishing a clearer set of design patterns and coding standards that are appropriate for the language. In addition to the Swift version from the original edition (2.0) being incompatible with modern versions of the language, the time was right to overhaul the original book.

This edition strikes a balance between revisiting some of the most loved content from the first edition and presenting new concepts that were not available when the previous edition was published. I have eliminated or streamlined concepts that fizzled out and greatly expanded on those that have proven to be more important than before. In particular, this book offers more coverage on building hardware projects, as the availability of high-quality, affordable parts has expanded rapidly, as has their ease of use. Beyond this, I noticed that, in general, more people want to build their own hardware, and apps to go along with it.

Additionally, I have tried to make this edition accessible to more readers. While every book I have written has been a narrative, aimed at developers with a basic understanding of iOS development, in this volume, I have modified the format to reflect the feedback I received while teaching and speaking to readers. Namely, this edition

INTRODUCTION

allows you to skip around more between topics and provides more detailed step-by-step instructions. Rather than simply giving you code to copy, I try to explain what you need to put together to accomplish each task in a project. My hope is that this will allow you to debug problems easier and use this book beyond the initial Swift version it was written for (Swift 4.2).

While it would be impossible to describe the IoT fully in one paragraph, in this book, I focus on how you can build, interact with, and network hardware-based sensors to make your users' lives more informed and convenient. In my home, I am able to use IoT devices to turn off my lights with my voice, determine if I am using my appliances at times when power is too expensive, and to figure out if I am brewing my tea at the perfect temperature (I am, but I need to remember to get up when the timer goes off). In this book, some of the projects you will implement will help you to keep better track of your workouts, determine whether you have locked your door, and show you a dashboard for your home on your Apple TV. Pretty cool for do-it-yourself projects!

Finally, remember, this book exists for you. The ultimate goal of writing it was to help you and the developer community gain a stronger understanding of IoT development as it relates to iOS and provide a wealth of inspiration for your own future projects. If you would like to contribute to making sure that the code stays up to date, please submit pull requests to the GitHub repository for the book (<https://github.com/Apress/program-internet-of-things-w-swift-for-ios>). If you have any feedback, I would be excited to hear it via Apress or my web site (www.devatelier.com). Happy hacking!

PART 1

Building Health Apps for the Internet of Things

CHAPTER 1

Laying the Foundation for Your First IoT App

Before taking you on a deep dive into Apple’s Internet of Things (IoT) technologies, I thought it would be useful to begin with a brief introduction to some of the tools and workflows you will use throughout the book to build your projects. In this book, your primary tools for building IoT apps will be the Xcode Integrated Development Environment from Apple and its Interface Builder tool for building user interfaces. When teaching iOS development, I always notice that both novice and experience developers find these tools to encompass some of the most challenging concepts to master, even more than Swift or the eccentricities of any framework.

To practice using Xcode, in this chapter, you will begin developing your first IoT app for iOS: IOTFit, a workout app that uses the location services (GPS) on a user’s phone to help them keep track of how long they exercised and where. If you are an avid user of such workout apps as Runkeeper or Nike Plus, you will recognize time tracking as a minimum requirement and location tracking as a feature that engages users and keeps them coming back for more.

Throughout this book, you will continue to expand the IOTFit app to integrate more of Apple’s IoT frameworks. I hope this approach will help you to notice when some solutions are more appropriate than others. Additionally, I feel this approach is very reminiscent of many Agile working environments, in which you constantly refine or expand a product, based on one strongly defined feature set at a time.

If you are already familiar with iOS development, you can safely skim through this chapter, but I highly recommend glancing through the screenshots to see if anything has changed since you last worked with Xcode. This chapter has been designed and tested for Xcode 9 and 10.

Learning Objectives

In this chapter, you will learn the following critical skills for IoT development on iOS by starting development on the IOTFit application.

- Setting up a project in Xcode
- Modifying a project from its default settings
- Linking Xcode to your Apple Developer Program account
- Using Interface Builder to lay out an adaptive user interface
- Connecting your visual layout-based user interface to your code

One of the most aggravating points of Apple platform development is that you have to master certain workflows and aspects of using Xcode in order to achieve critical milestones as an iOS developer, such as releasing your first app. To help you with this process, in this chapter, I will try to include as many detailed screenshots and explanations as possible for the project and user interface setup steps.

Unfortunately, Apple frequently replaces its old tools and workflows. This allows Apple to build tools that are more relevant to current trends, but it also places a burden on developers to keep up to date with the latest changes and caveats. In my personal experience, I have noticed that while a workflow may change only every few years, the project settings change with almost every major release. In the past two years, iOS 11 added mandatory settings for enabling permission-locked features like location sharing and iOS 12 went even further by replacing the default build system.

In this chapter, I emphasize adaptive (Apple's term) user interface development, because the current range of supported iOS devices is so vast. While devices with the form factor of the iPhone 8 and 8 Plus are the most widely used at the time of writing, the tiny iPhone SE still makes up a huge segment of active devices, and Apple's bezel-less devices (iPhone X, XR, XS, XS Max) are being positioned strongly as the future of the platform. If you take the iPad lineup into consideration, you will notice that the iPad mini is nearly the same size as the iPhone 8 Plus, and the iPad Pro 13-inch model is larger than many of today's laptop computers. As a developer, it is amazing that the same code can run on all of these devices; however, it comes at the cost of having to do some careful

preparation work and debugging, to make sure the experience is consistent across all devices. In this chapter, I will share the workflow I use to build adaptive user interfaces and tips I think will help make it easier for you.

Note For the sake of clarity, the diagrams in this book illustrate the iPhone implementation of these projects. Most of the projects in this book will run fine on iPad as well, except for the ones that require device-specific hardware (for example, Core Motion, Face ID). I will indicate what these projects are at the beginning of the relevant chapter.

This chapter is intended to guide you through the process of setting up a project, but if you run into any trouble along the way and would like to look at the completed project for reference, it is available from the GitHub repository for this book, under the Chapter 1 folder (<https://github.com/Apress/program-internet-of-things-w-swift-for-ios>).

Setting Up the Project

Before I start developing a project, I always want to know what I am building. To help you gain a better understanding of what the first version of the IOTFit app will encompass, please look at the wireframe diagrams in Figure 1-1. In design terms, a wireframe is usually an initial sketch (hand-drawn or computer-generated) that lays out the most critical components of a user interface.

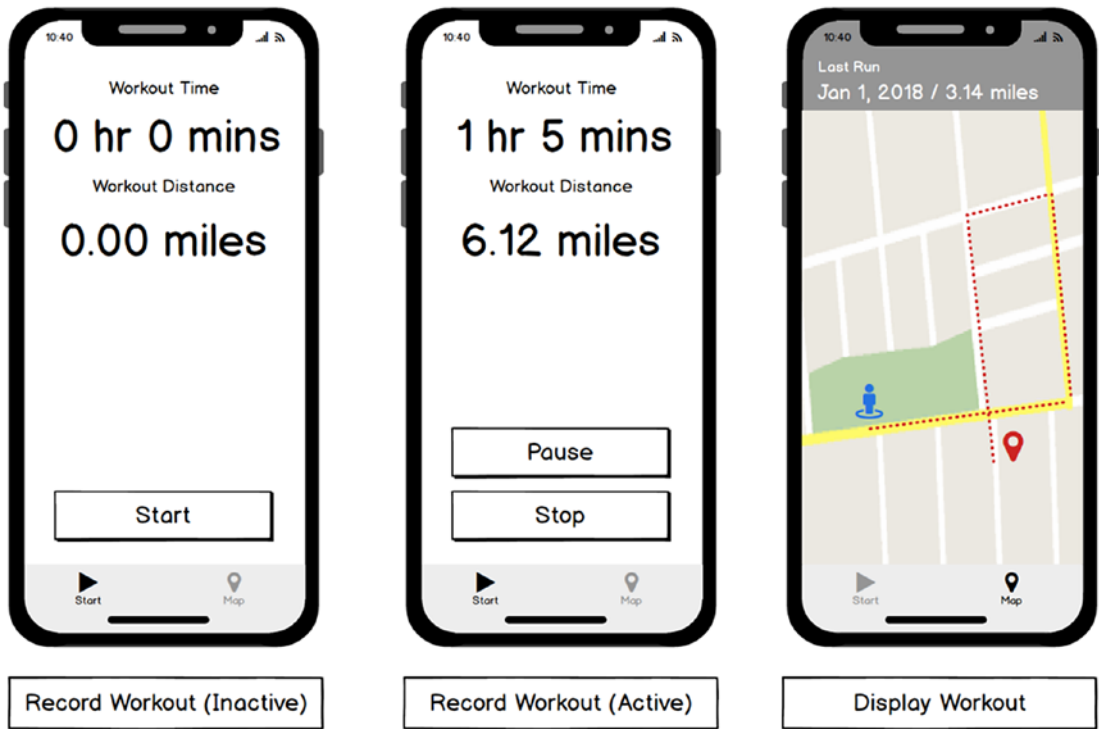


Figure 1-1. Wireframe diagrams for the IOTFit app

In reviewing the wireframes, you will notice three main states for the application: recording a workout (inactive), recording a workout (active), and displaying a workout on a map. When users want to start a workout, they press the Start button on the record screen, and the text on the buttons and labels change to indicate that the workout is being recorded. If users want to view their progress on a map, they press the Map icon in the bottom tab bar, and the app will show an annotated map in place of the record screen. If you use the App Store or Facebook apps for iOS, you will already be familiar with the tab bar as a convenient way of switching screens in an app, while still preserving the state of each tab. Although it is not listed on the wireframe, the background state preservation you will implement for the app allows users to keep tracking a workout, even when the app is in the background.

WIREFRAMES VS. MOCKUPS

I like to use wireframes at the beginning of a project to put the stakeholders, developers, and design team on the same page about what an app has to do, before committing to the time-consuming work of generating mockups—the Photoshop- or Sketch-generated design resources that specify the fine details of implementation, including exact colors, font sizes, and shadows. It is much easier to throw away or redo a wireframe than it is a mockup!

To begin the development process, you will have to set up a new Xcode project for the IOTFit app and configure it for the iOS frameworks you will need to use. Apple provides a very rich toolbox for you to work with, but it always requires some careful preparation. Before getting started, take a second to think about what frameworks you would like to use to implement the requirements of the app, then refer to Table 1-1 for the final list of what application programming interfaces (APIs) you will end up using in this project.

Table 1-1. *IOTFit Features and Their Corresponding iOS APIs*

Requirement	Application Programming Interface	Parent Framework
Switching between screens easily	Tab View Controller (UITabViewController)	UIKit
Displaying a map	Map View (MKMapViewController)	MapKit
Accessing GPS hardware	Location Manager (CLLocationManager)	Core Location
Requesting location permission	Location Manager (CLLocationManager)	Core Location

The UITabViewController and MKMapViewController classes will drive the most complicated parts of the user interface. The Core Location framework will do the heavy lifting for requesting and tracking the user's location.

Now that you have a better idea of the technical and design aspects of the project, you can begin implementation. First, create a new Xcode Project by opening Xcode on your Mac and clicking Create a new Xcode project from the Welcome to Xcode screen shown in Figure 1-2. Alternatively, if Xcode is already open, you can click the File menu and then select New ► Project.

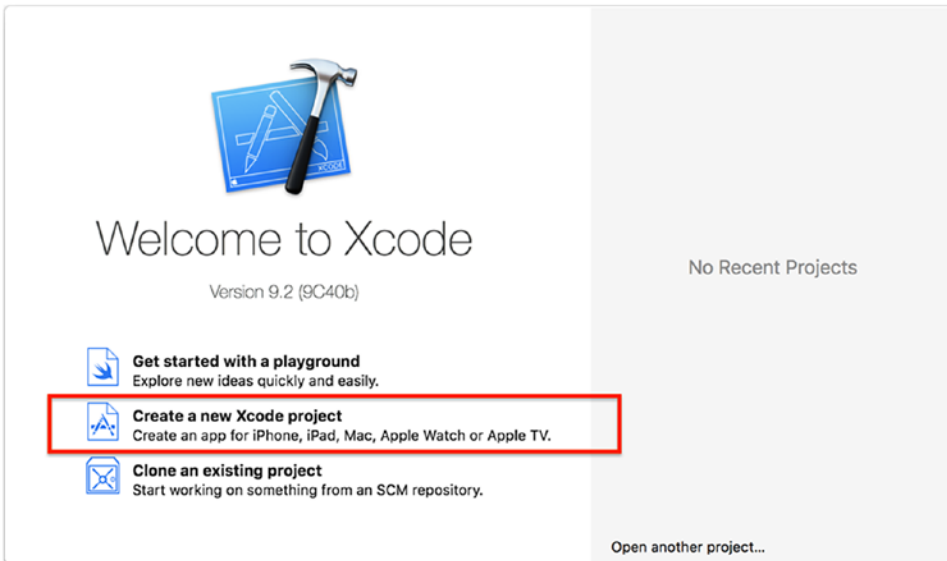


Figure 1-2. *Creating a new project from the Xcode welcome screen*

Next, Xcode will present a pop-up window asking you to select the template type for your project. As shown in Figure 1-3, select Tabbed App to create a project based on a template that includes a Tab Bar Controller and two empty View Controllers. This template is close to the general user interface you must use for the IOTFit app and saves a lot of time over manually setting everything up yourself.

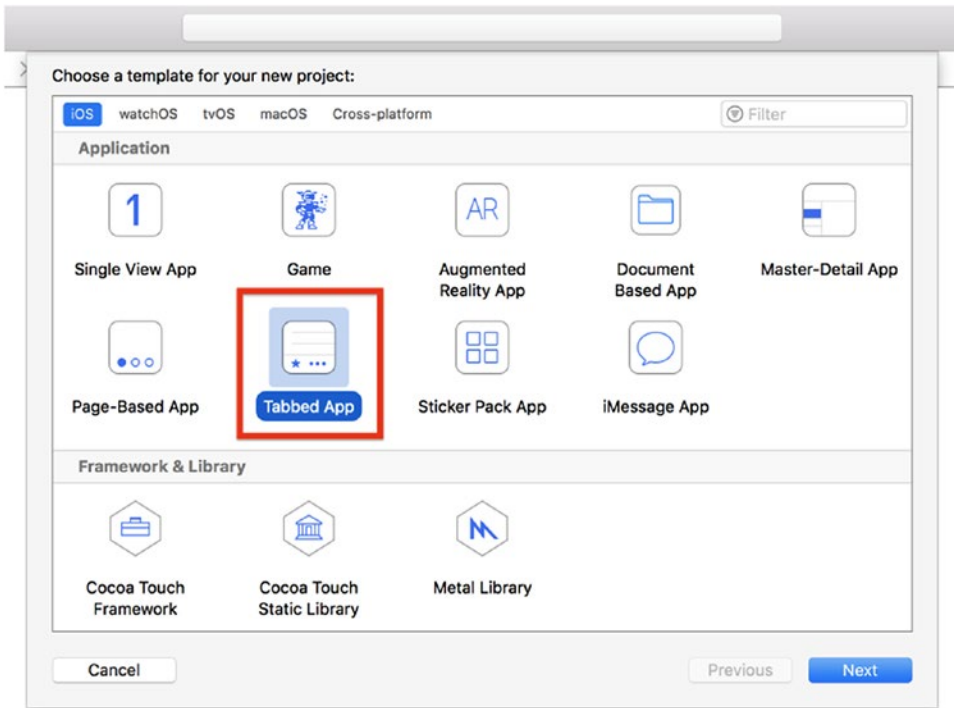


Figure 1-3. *Selecting the Tabbed App template*

After selecting the correct template, click the Next button, then when asked to set the options for the project, enter “IOTFit” as the project name, as shown in Figure 1-4. This will be used as a general identifier throughout the project and as the default display name of the app on the iOS home screen. If you have an organization name or organization identifier you would like to use, you can enter those at this time too. You do not have to set a team for the project at this time, as you will do that later, after verifying that the project has been created successfully.

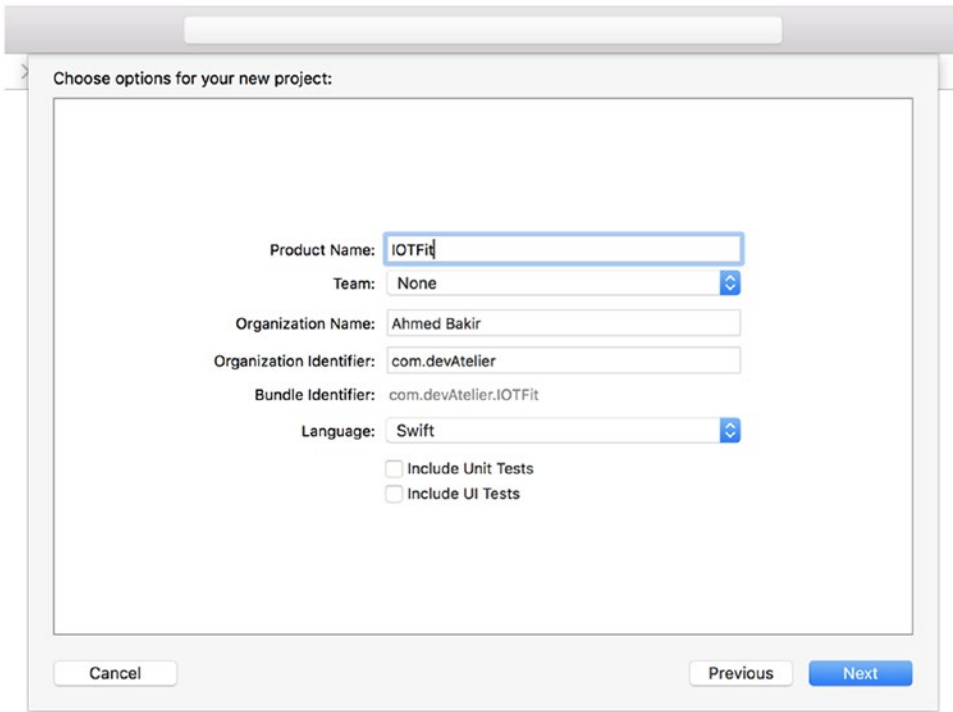


Figure 1-4. Initial options for IOTFit project

After confirming these options, click the Next button, then select a location to save the project. I like to place my projects in an easy-to-find folder in my home directory. As shown in Figure 1-5, click the Create button to generate the project.

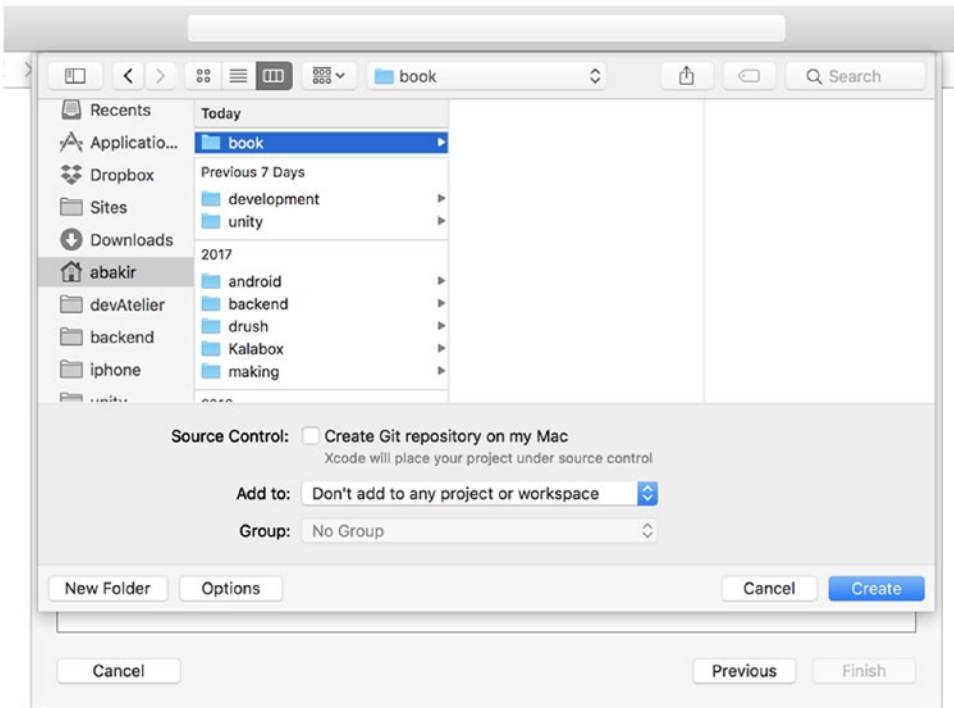


Figure 1-5. *Selecting a destination for the IOTFit project*

After the project has been generated successfully, Xcode will greet you with the Xcode editor window for your new project, as shown in Figure 1-6.

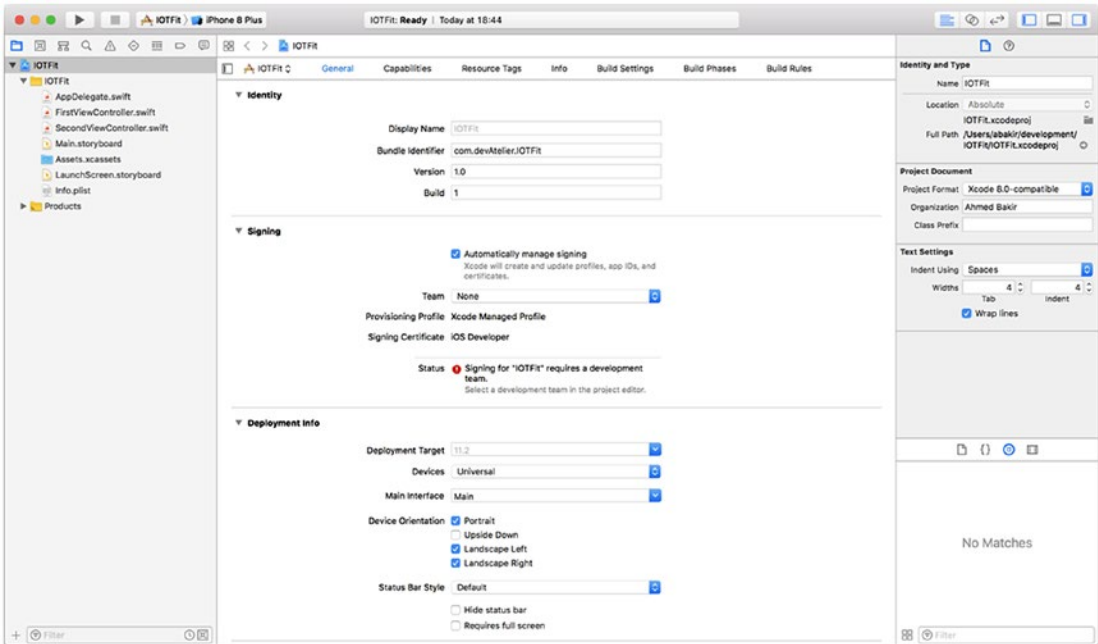


Figure 1-6. Default Xcode editor window for the IOTFit project

If you have been developing apps for a while, this editor window should be very familiar to you. For newer users, the major areas of this screen are described following.

- *Navigator pane (left):* The atlas for your project. This allows you to manage your project hierarchy, search for text in your project, and quickly navigate to debugging issues.
- *Editor pane (center):* Your main editing workspace. This lets you modify source code, build settings, and view diffs of source-control-managed files.
- *Utilities pane (right):* Your source code concierge. With this, you can manage additional settings for individual files and view quick help tips on classes you are working with (by simply highlighting them).

Getting back to the project at hand, verify that the settings for the generated project are similar to those in the zoomed-in screenshot in Figure 1-7. In particular, verify that the project has source files in the navigator pane and that the app’s Display Name and Bundle Identifier match what you entered into the previous screens.

Tip If the project setting did not automatically appear with the editor window, manually select them by clicking the project name in the Project Navigator (the topmost item in the navigation pane).

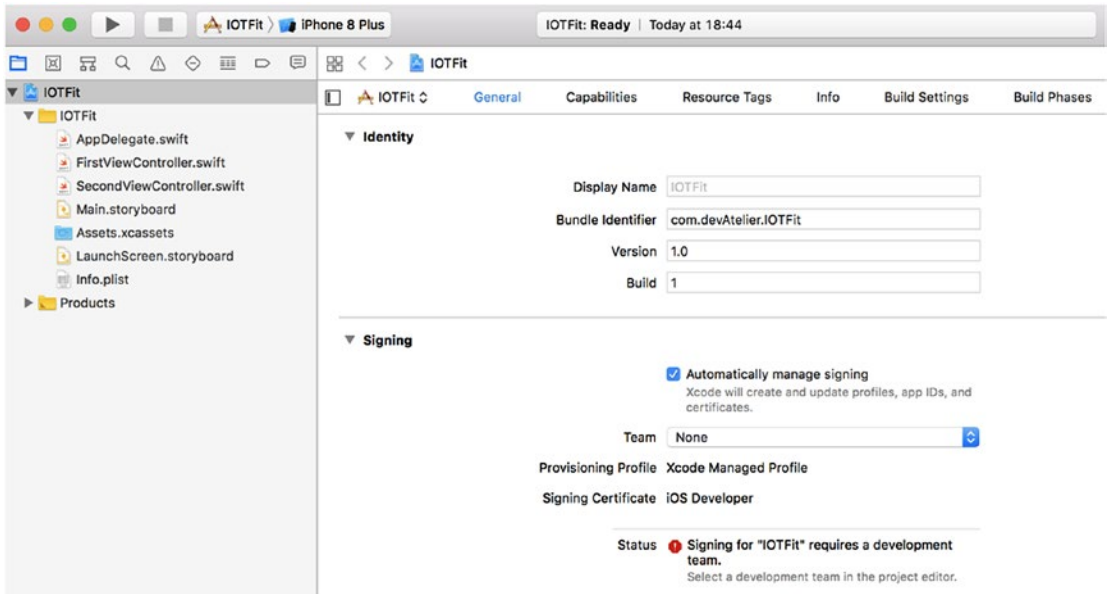


Figure 1-7. Default Xcode project settings for IOTFit

Linking Your Apple Developer Account to Xcode

The first time you open Xcode or a project that you did not create yourself, such as one that you cloned from GitHub, on your computer, the Signing section of your project settings will have one of the error messages shown in Figure 1-8 or Figure 1-9, indicating that it cannot find the signing credentials required to build the project.

▼ Signing

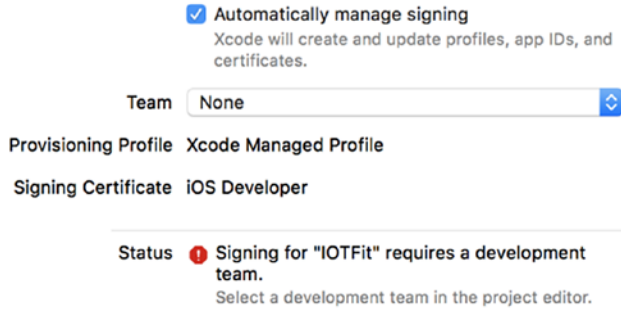


Figure 1-8. Signing error for fresh Xcode installation

▼ Signing

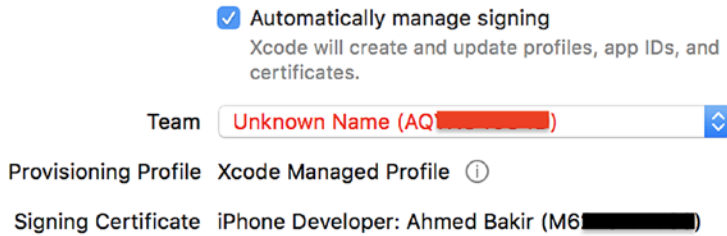


Figure 1-9. Signing error for missing Apple Developer account

To resolve these issues, click the drop-down menu next to Team, as shown in Figure 1-10. Click Add an Account.

▼ Signing

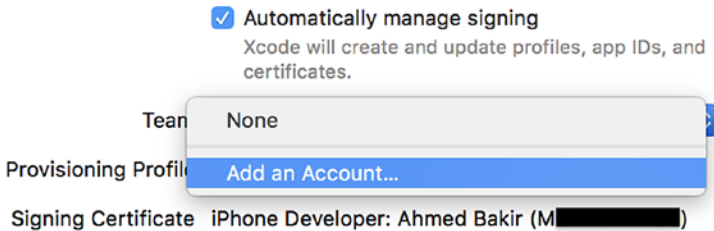


Figure 1-10. Team selection drop-down menu

After making your selection, Xcode will display the sign in prompt shown in Figure 1-11. If you have a paid Apple Developer Program membership, sign in with that. Otherwise, enter a valid Apple ID that you use with iTunes or on the App Store.

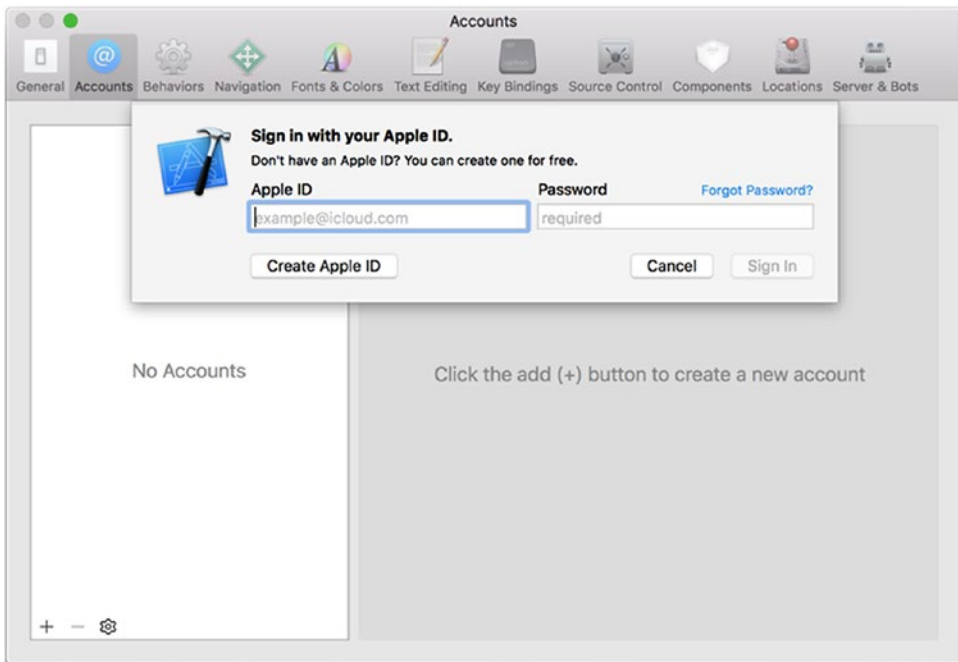


Figure 1-11. Xcode's Apple ID sign-in prompt

Note Apple allows you to create a limited Developer account without a paid membership that lets you test up to three devices per Apple ID. For this book, this type of account is sufficient. However, when you reach the stage where you must release your app on the App Store, or wish to share it via TestFlight or Enterprise distribution, you will have to upgrade your account to a paid tier.

After signing in, the Accounts window will show a list of all the Apple Developer Program teams your Apple ID is associated with, as shown in Figure 1-12. To sign onto another Apple ID account or add a Source Control Management account (for example, GitHub, Bitbucket) to Xcode, you can click the plus (+) button at the bottom-left (circled in Figure 1-12).

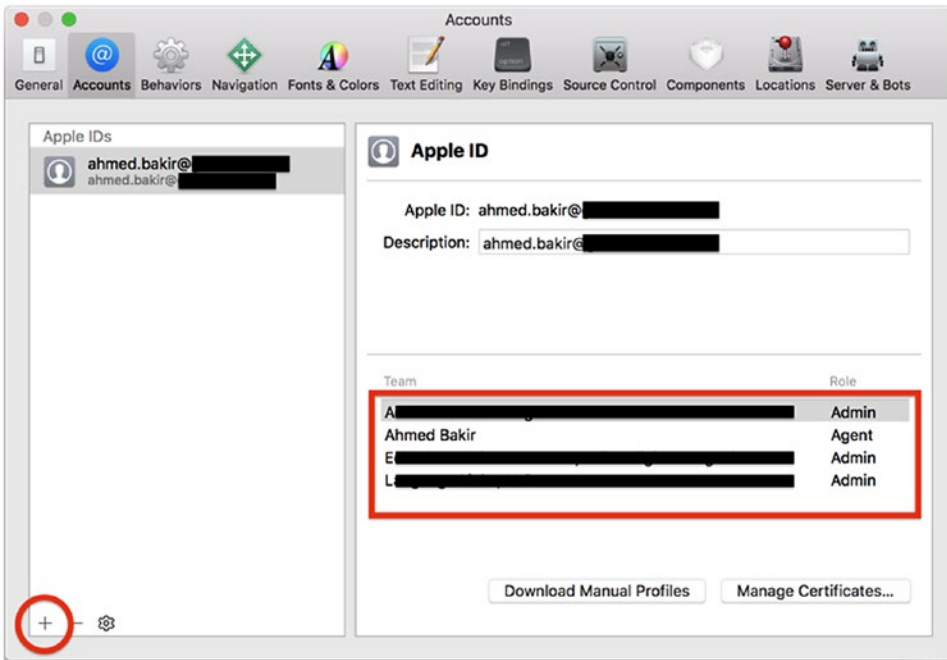


Figure 1-12. Accounts window for successful sign-in

After successfully linking your account, it is safe to close the Accounts window. When you return to the editor window, your project settings should resemble those in Figure 1-13, but with your team set as the team for the project.

▼ **Signing**

Automatically manage signing
Xcode will create and update profiles, app IDs, and certificates.

Team

Provisioning Profile ⓘ

Signing Certificate

Figure 1-13. Project settings after successfully signing in

If your selection still does not appear after closing the Accounts window, click the Team drop-down menu again, to select one of your linked teams manually. You can use this same method to change the team a project is associated with.

These steps complete the project setup phase for this chapter. In later chapters, you will return to the project settings to add new features to the app. Feel free to use this section as a quick refresher at those times.

Building an Adaptive User Interface

As an app developer, you always have to keep three thoughts in the back of your head: “How do I build it?,” “How do I convince people to download it?,” and “How do I keep people coming back for more?” Although marketing strategies are critical to building and maintaining a customer base, as a developer, you can make an app more attractive to your users by making sure it has a solid feature set and provides a compelling, consistent user experience across all devices. In this section, I will introduce you to using the Auto Layout features of iOS, to build a user interface that can *adapt* to all of the different devices iOS runs on.

Until the release of the iPhone 5, iOS developers only had to worry about two devices: the iPhone and the iPad. If a developer wanted to be on the cutting edge, he or she could also configure all of the user interface elements to handle rotating with the device. For this workflow, many of us could get by programming the entire layout ourselves and maintaining separate Interface Builder (.xib) files for the iPhone and iPad and for portrait and landscape modes. However, things started to change rapidly after the iPhone 4 was released, and today’s device lineup for iOS has expanded greatly since those early days. As an example, in Figure 1-14, I have provided a screenshot of the device preview options in Interface Builder as of late 2018.

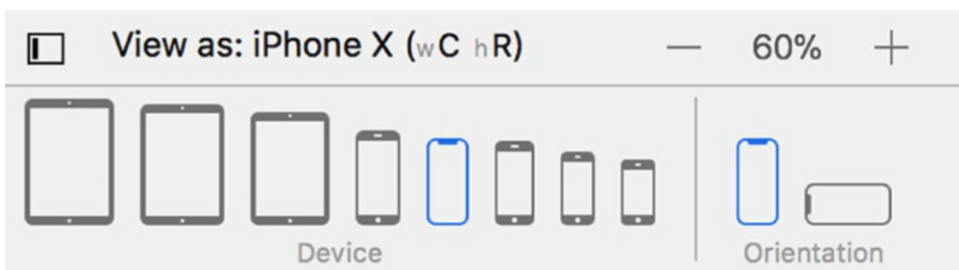


Figure 1-14. Xcode 9’s Interface Builder device preview selector

In order of decreasing screen size, you are able to view previews for the following devices:

- iPad Pro 13"
- iPad Pro 10.5"
- iPad Pro 9.7"
- iPhone 8 Plus
- iPhone X
- iPhone 8
- iPhone SE
- iPhone 4S

Going back to Figure 1-14 for a second, if you look carefully, you will notice that every iPhone in the list has a different aspect ratio. When you combine these variables with portrait and landscape mode, it starts to become obvious why iOS developers moved away from hard-coding pixel positions for user interface elements a long time ago. It sends you down an endless if statement rabbit hole!

By using Apple's trait API's (`horizontalSizeClass`, `verticalSizeClass`, `displayScale`, `userInterfaceIdiom`), you can determine at runtime the display characteristics of the device you are running on, without having to worry about the exact device model or pixel size. You can then use these to define rules (*constraints*) for how Auto Layout should adapt items for different screen configurations (for example, iPad Pro in landscape mode, iPhone SE in portrait mode).

Each size class has two possible values: `regular` or `compact`. Auto Layout defines iPads as purely regular devices. Both its horizontal and vertical size classes return regular values, regardless of what the screen orientation is. iPhones, however, introduce compact values, based on their device orientation. An iPhone in portrait orientation returns `regular` for its vertical size class and `compact` for its horizontal size class. An iPhone in landscape orientation returns `compact` for its vertical size class and `compact` for its horizontal size class. If you are wondering why, Figure 1-15 should help. When you put an iPhone X in portrait orientation next to a ten-inch iPad Pro in landscape orientation, you will notice their screen heights are about the same, but the iPhone X is much smaller in width. When you rotate the iPhone X to landscape orientation, both dimensions will be much smaller than the iPad Pro.

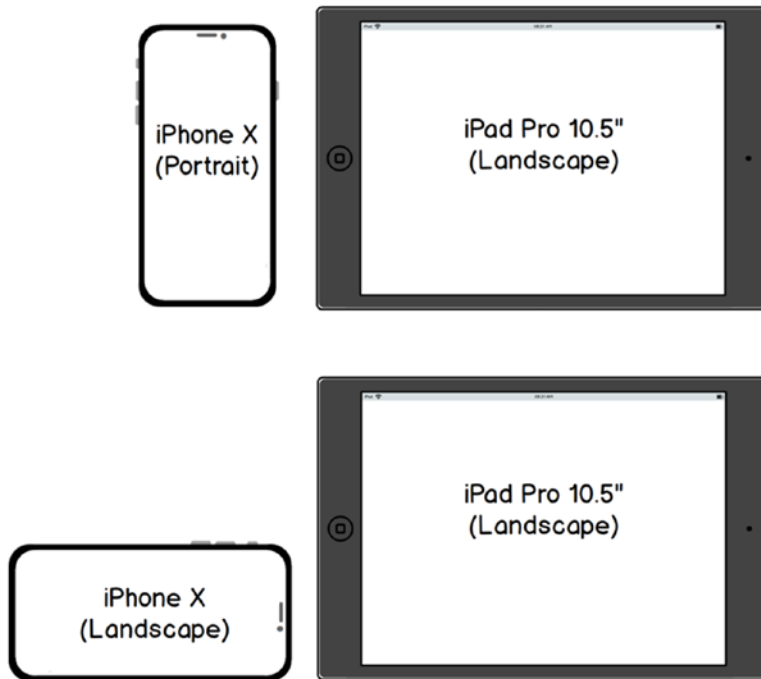


Figure 1-15. Comparing the physical dimensions of an iPhone X and iPad Pro

The way you will use Auto Layout in this project is by placing elements on the main storyboard for the app via Interface Builder and then using the IDE to set the Auto Layout constraints for each. Interface Builder’s preview tools let you toggle between different device configurations, so you can see if your rules are sufficient to be a good fit for the devices you are targeting. What has worked best for me in the past has been to work out most of the details of Auto Layout in Interface Builder first and then fine-tune settings in my code later.

Renaming Classes from the Base Template

As mentioned earlier, a huge advantage of using the Tabbed App template from Apple is that it pre-populates your project with a storyboard and empty classes for an application that uses a Tab interface as its primary method for navigation between screens. Earlier, you verified that the project itself was generated successfully. To verify that your storyboard was generated successfully, click the `Main.storyboard` file in the Project

Navigator. The center pane of the editor window should switch to Interface Builder and display the contents of the default storyboard, as shown in Figure 1-16. The storyboard should contain a Tab View Controller linked to two blank View Controllers.

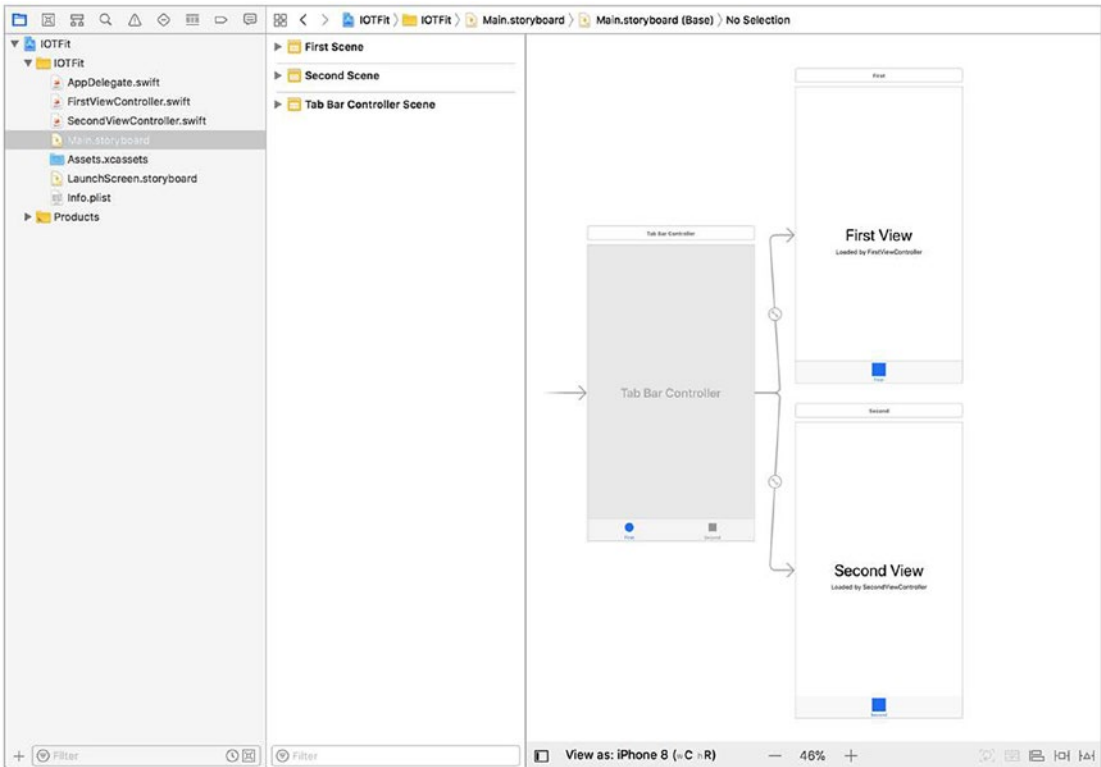


Figure 1-16. Default storyboard for a Tabbed App project

The child View Controllers are named FirstViewController and SecondViewController; however, these class names will make maintainability difficult as the project starts to grow. As with a natural ecosystem, items in an Xcode project are linked deeper than what you see on the surface. In the case of classes that are used by storyboards, changing the class name also means changing references in other classes and the storyboard file itself. To manage this complicated process, you can use the refactoring tools in Xcode. To rename the FirstViewController class, secondary-click (right-click or long-press) the symbol name, scroll down to Refactor in the contextual menu, and then select Rename, as shown in Figure 1-17.

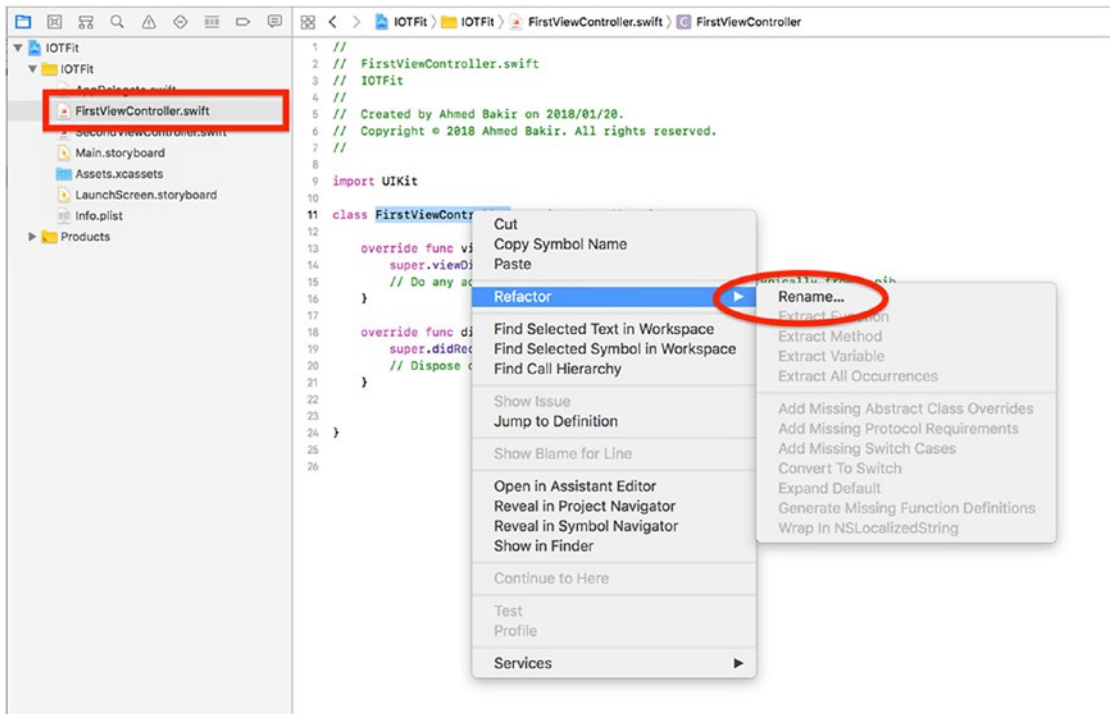


Figure 1-17. Presenting the Refactor menu for a symbol

After clicking Rename, the center pane will display an accordion-style table. In the case of renaming a class, the first row shows a preview of the file name, the second provides an editing area where you can make changes to the class name, and the third row and later show previews of changes that are made in files that use your class as a dependency (such as storyboards or other classes). Enter “CreateWorkoutViewController” as the new name for the View Controller and verify that the output is similar to that in Figure 1-18. Click the Rename button at the top-right, to apply your changes.

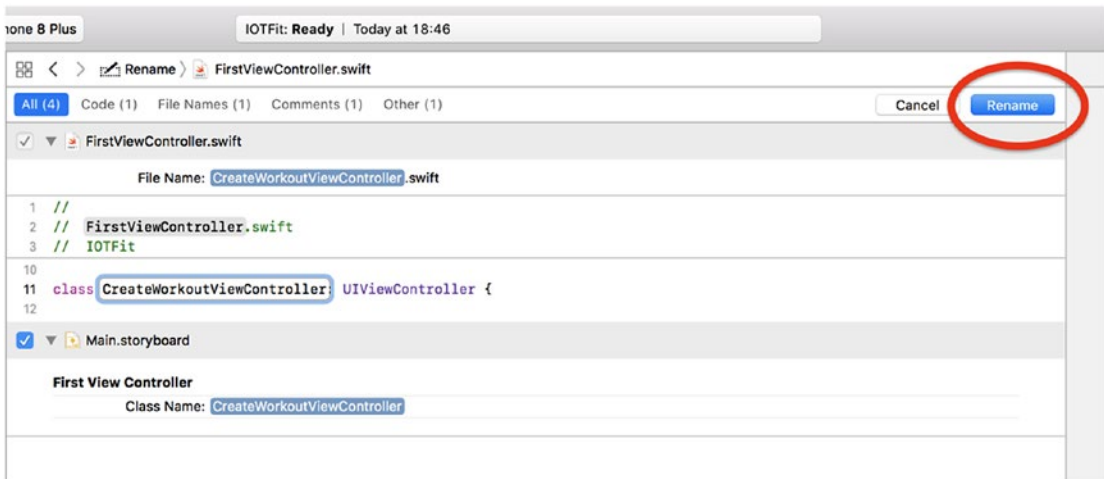


Figure 1-18. Xcode editor preview for refactoring

Follow the same process to rename `SecondViewController` to `WorkoutMapViewController`. You can use the Rename tool in your own code later, to rename variable names and data structures with the same visual editor.

Note If the files did not get renamed automatically, you can fix this with two click actions. First, click the file name once, to select the item, then click again, to edit the file name. In Swift, the file name does not affect compilation.

Laying Out the User Interface

Now that the storyboard's dependencies have been resolved, you can start placing user interface elements. If you have been developing iOS apps for a while, this should be a fairly routine exercise, but for newer users, it may clear up some confusion from the past.

Your first objective will be to layout the Create Workout screen from the wireframe in Figure 1-1. It has four labels (two for information and two for values) to display your workout progress and two buttons the user can use to start/stop the workout or pause/resume a workout.

First, start by clicking `Main.storyboard` from the Project Navigator. From there, make sure the utilities pane (right pane) is open. The bottom right has a split screen with an icon that looks like the old iPhone Home button. This is called the *object library* and contains user interface elements that you can drag onto View Controllers in the

storyboard. Scroll down or search for “Label,” to find a UILabel. As shown in Figure 1-19, drag-and-drop the Label object from the object library to the Create Workout View Controller (it should still have the “First View Controller” label on it from the template).

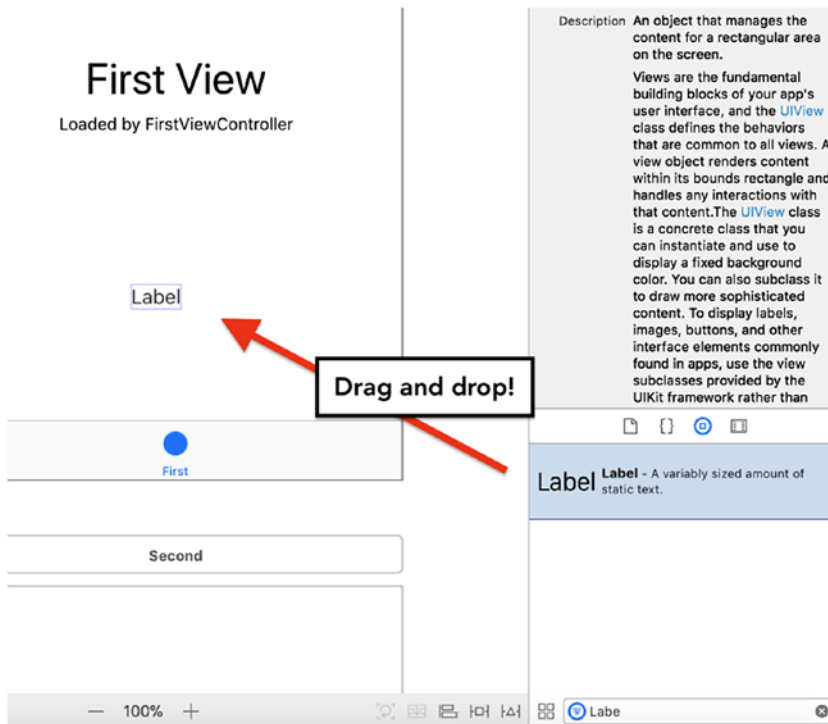


Figure 1-19. Adding a label to a View Controller

Tip To speed up navigation, you can open files within tabs in Xcode, by pressing Command+T (⌘+T).

Continue this process, adding the other three labels and two buttons. You can delete the old labels with the “First View” text as well. At this point, don’t worry about the exact placement of the items, as you are about to learn how to use constraints to set the size and Auto Layout rules.

Applying Auto Layout Constraints

When building a user interface with Auto Layout, instead of thinking about the (x, y) coordinate position of each item, you should think about where they should sit relative to the boundaries of the screen and other elements. In iOS, these rules are managed by *constraints*. At runtime, Auto Layout will use these constraints to resize or reposition elements on the screen. The most common types of constraints are *pinned* (fixed position) and *relative* (their position is relative to a boundary on the screen or other items). Constraints can be greater than, equal to, or less than a value.

For the IOTFit app, I felt the key feature of the Create Workout screen was that users should be able to press the Start and Pause buttons easily, no matter what size iPhone they are using. By pinning the buttons to the bottom of the screen, I would not have to worry about the button placement becoming too difficult to use with changes in screen size. Users are already accustomed to pressing buttons on the bottom of the screen, because the placement of the Home button on the bottom of every iOS device has taught them to interact with the bottom of the screen for control gestures. For the status labels, I wanted to accomplish two goals: keeping the screen balanced and not interfering with the placement of the buttons, so I pinned them to the top of the screen. As shown in Figure 1-20, with this layout in mind, when the app runs on an iPhone SE, iPhone X, and iPhone 8 Plus, the only things that will change are the scale of the elements and the vertical space in between the status labels and action buttons.

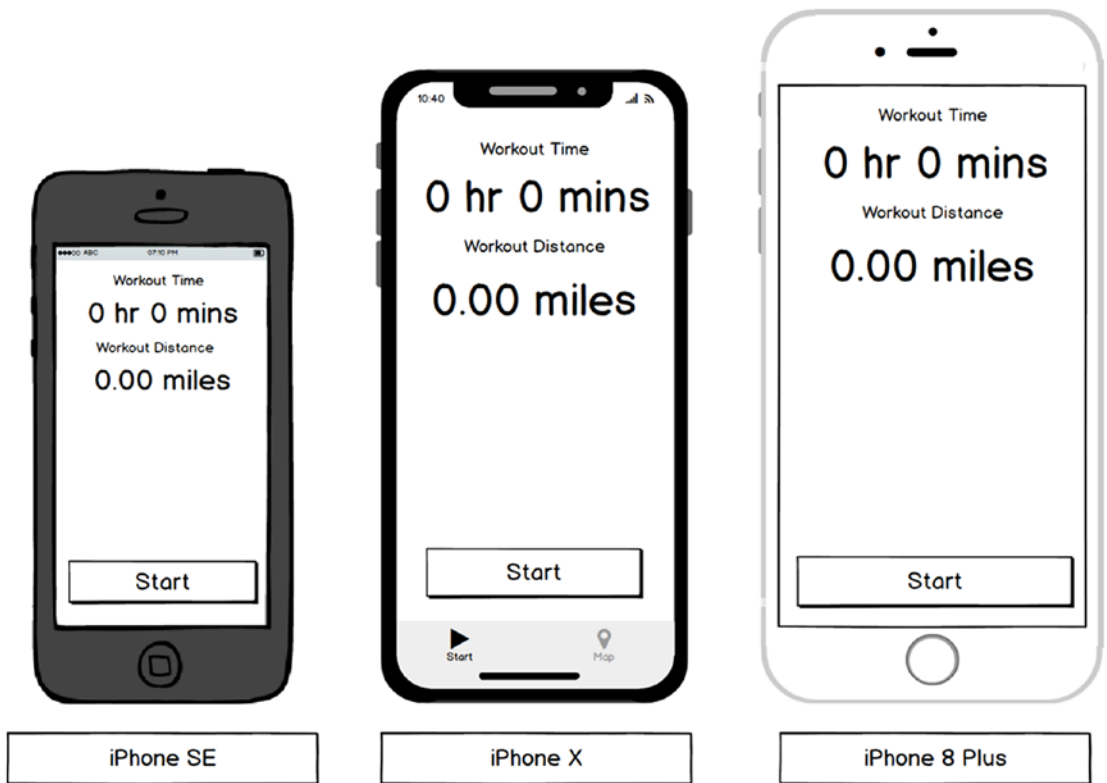


Figure 1-20. How the IOTFit user interface adapts to different iPhone sizes

Tip Another good Auto Layout strategy to use is to center an item in the middle of the screen and place items below or above that.

In Interface Builder, you can use the Add New Constraints tool to set constraints, using a graphical user interface. As shown in Figure 1-21, select the topmost label, then in the bottom toolbar for the editor window, find the Add New Constraints button (the icon looks like a box plot graph), and click it. The pop-up window that appears will display the label's distance to its closest neighbors.

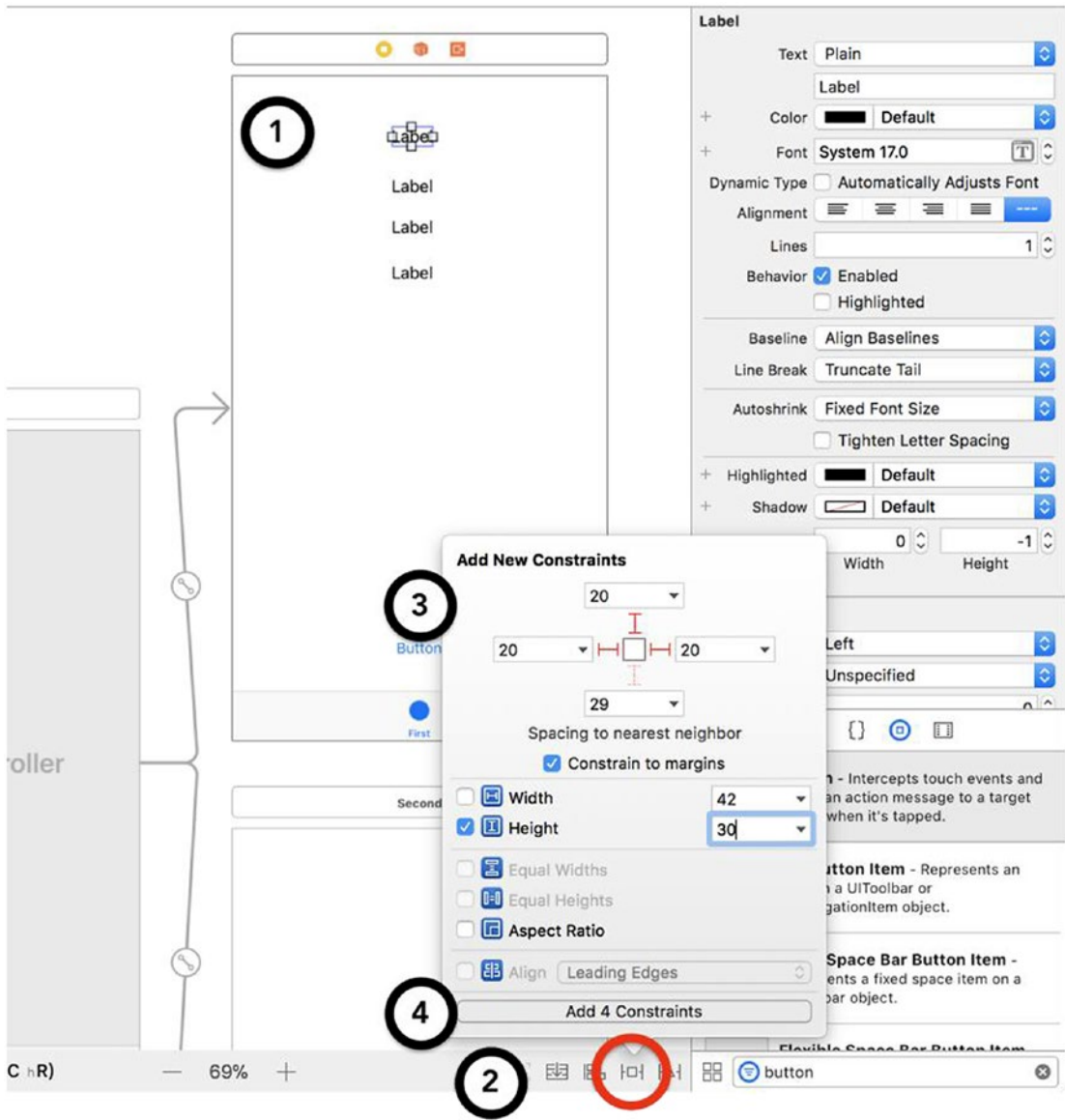


Figure 1-21. Steps for setting constraints via Interface Builder’s Add New Constraints tool

To set a fixed constant, click the text field containing the value you want to fix and type in the new value. The line next to the text field will change from a dotted red line to a solid red line, indicating that your value was read successfully. For the topmost label, fix the label to 20 pixels from the top, left, and right of the screen. Set the height of the label

to 30 pixels. Check the check box next to height, to make sure your change is recorded. Finally, click the Add 4 Constraints button at the bottom of the pop-up, to save your changes. The modified storyboard should be similar to the screenshot in Figure 1-22.



Figure 1-22. Create Workout View Controller, after setting constraints for the top label

Customizing the Appearance of Items

Although you have set the constraints for the top label in the Create Workout View Controller, the appearance of the label does not match the initial design in the wireframe. To modify the text, apply the correct text justification (centered), and set the text size. Follow the steps indicated in Figure 1-23. First, click the label in the storyboard, then, in the utilities pane (right), click the Attributes Inspector (center icon). To change the text, insert the new text you want in the Text row. Click the T icon next to Font to change the font. A pop-up will appear. Instead of typing in a font size, pick one of Apple’s pre-defined styles, Title 3. This will help the text app scale better for users with vision impairments. Additionally, you will use the style in this family later, enabling a consistent user interface. Click the “center” graphic next to Alignment, to center the text alignment. Finally, to prevent text from clipping on small screens, set the Autoshrink property to Minimum Font Scale. This will shrink the text as low as 50% of the original size before clipping the text.

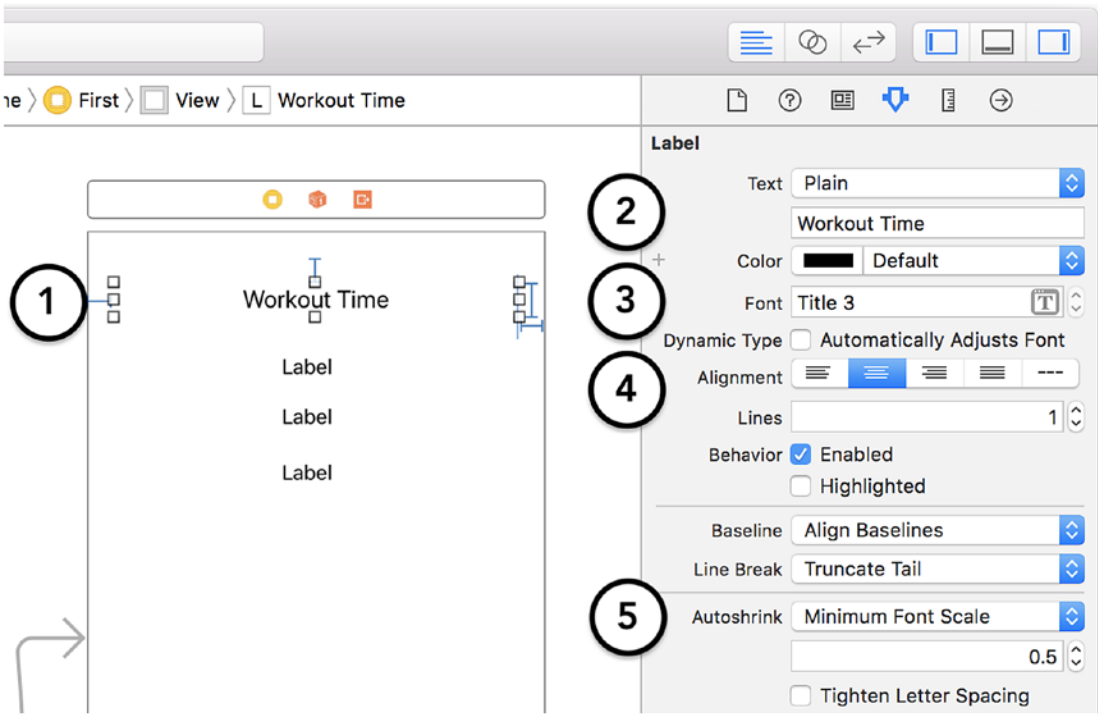


Figure 1-23. Steps for setting the correct display properties for the Workout Time label

Follow these same steps of setting constraints and text properties for all the labels and buttons, according to the parameters specified in Table 1-2.

Table 1-2. Constraints and Font Settings for Create Workout View Controller User Interface Elements

Display Text	Top	Left	Right	Bottom	Height	Text Class
Workout Time	20	20	20	--	30	Title 3
0 hrs 00 mins	10	20	20	--	50	Title 1
Workout Distance	10	20	20	--	30	Title 3
0.00 meters	10	20	20	≥20	50	Title 1
Pause	--	20	20	20	60	Title 2
Stop	20	20	20	20	60	Title 2

After applying the constraints and font settings, your storyboard should resemble the screenshot in Figure 1-24.

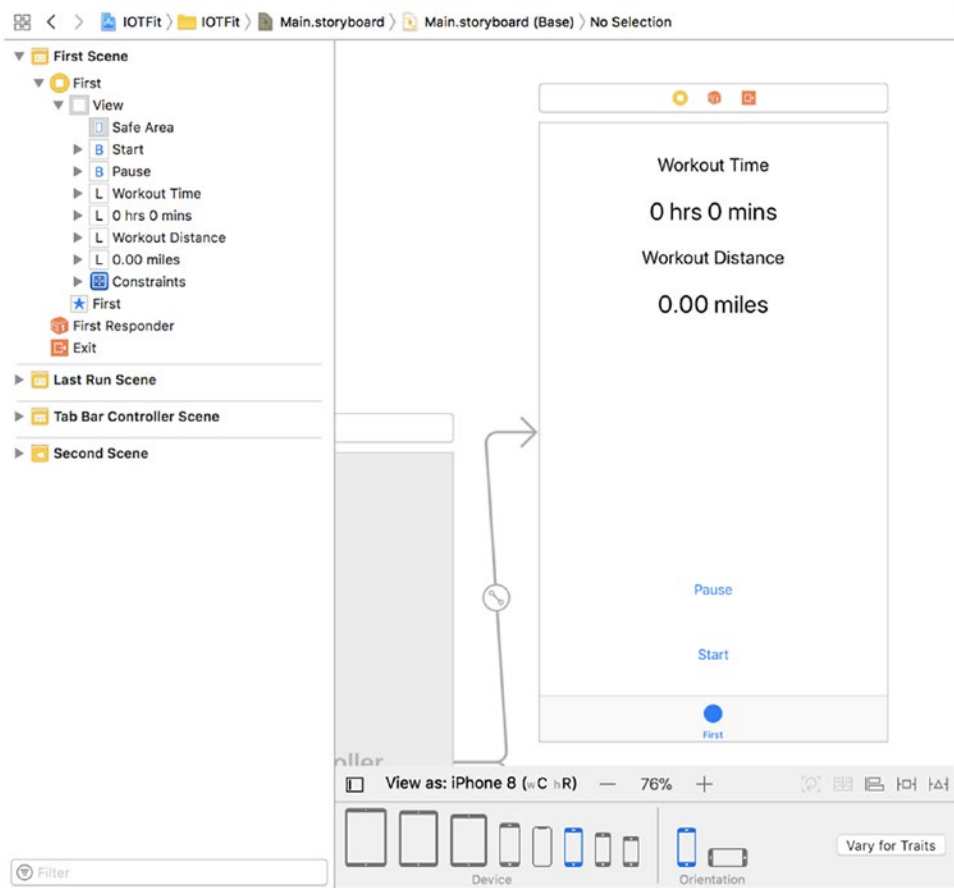


Figure 1-24. Create Workout View Controller after setting all constraints

By default, UIButton objects are styled with a transparent background and text style that match the default tint color of your app (as of this writing, the default tint color is a shade of blue). To make the app easier to use, in my wireframes, I suggested a button with a large colored touchable area and white text. Using the Attributes Inspector, once again, you can modify the text color and background color of a UIButton element.

Click the Pause button to select it, then click the Attributes Inspector (tab icon) in the utilities pane, if you have not already. As shown in Figure 1-25, click the Text Color drop-down menu, to bring up a secondary menu with frequently used colors. Select White Color to make the button text white. The button will appear invisible, but it will reappear after setting the background color.

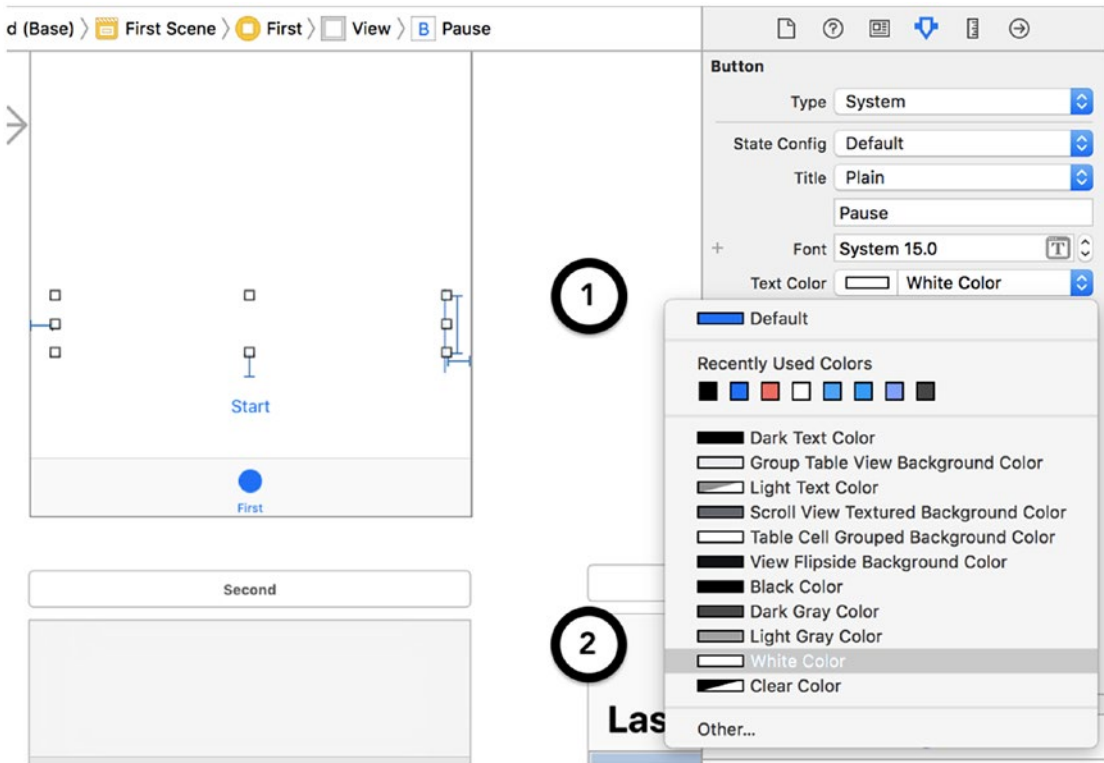


Figure 1-25. *Setting the text color for a button*

To set the background color, scroll down in the Attributes Inspector. Under the View section, select the Background drop-down menu. This time, when the secondary menu appears, select Other. As shown in Figure 1-26, this will bring up a color wheel. Select a shade of red, as this is a color that many users recognize in the context of stop or pause actions (e.g., a red stop sign).

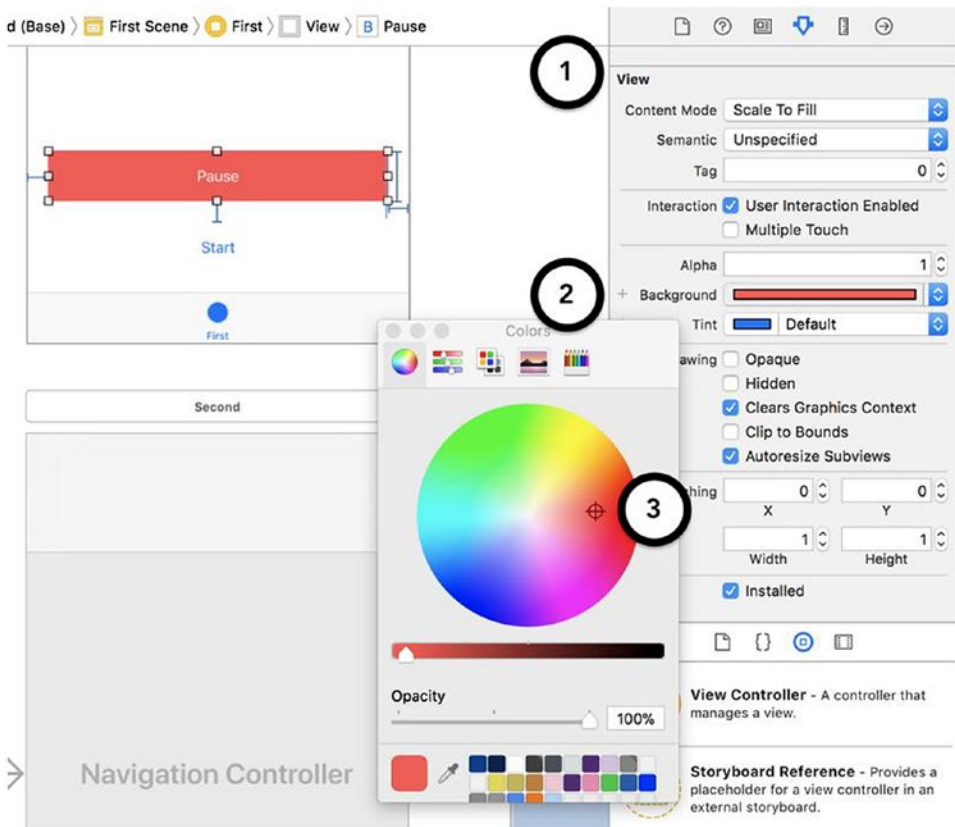


Figure 1-26. Color wheel for selecting background color

Follow these same steps to select a blue background color for the Start button. Your final layout for the Create Workout View Controller should be similar to the screenshot in Figure 1-27.

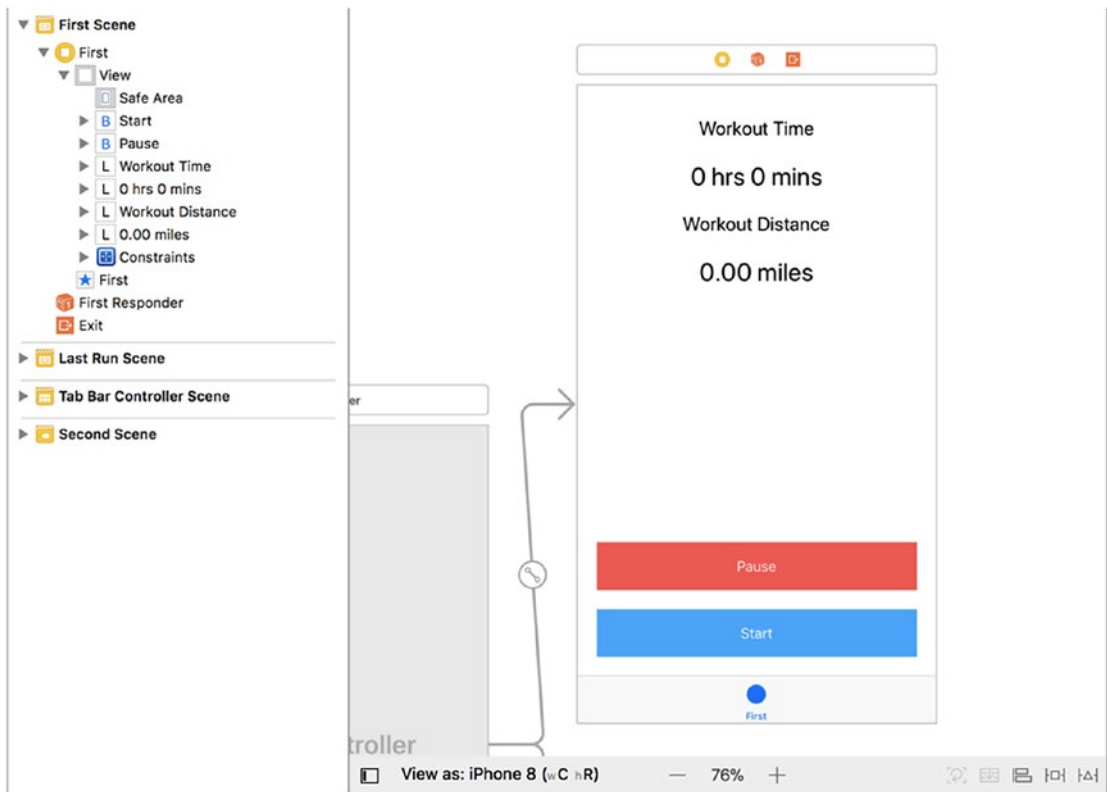


Figure 1-27. Final layout for Create Workout View Controller

Tip You can use the hierarchy panel (the left pane of the editor) to quickly verify that items have been added to a View Controller. Similarly, by rearranging items in the hierarchy, you can quickly move items in front of or behind other items.

Now that the Create Workout View Controller has been successfully set up, you can move on to the Workout Map View Controller. If you remember Figure 1-1, this screen consists of a navigation bar at the top, with some information about the last recorded workout and a map that fills most of the screen.

The easiest component to start with is adding the navigation bar. Navigation bars are one of iOS’s core navigation features, which emulate the behavior of a browser window, by providing a common information bar at the top. A navigation bar is loaded with a *Root View Controller* (similar to a home page). When buttons (or links) inside the

Root View Controller are clicked, the navigation bar pushes their content into the main window, while also providing a back button at the top right, allowing the user to navigate back to the last screen.

To easily add a navigation bar to the blank Workout Map View Controller, click to select the View Controller in Interface Builder (it should still have the “Second View” label on it) and, as pictured in Figure 1-28, go to the Editor menu and select Embed In ► Navigation Controller.

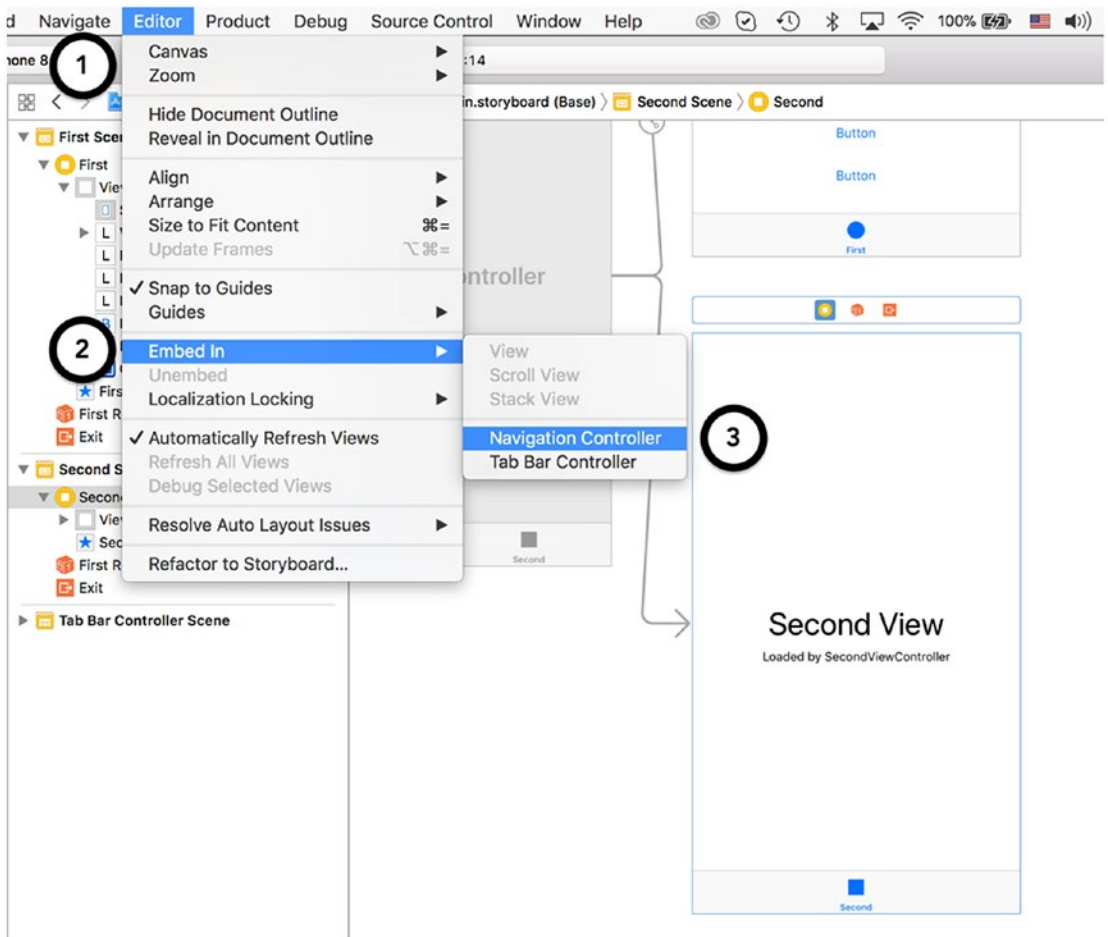


Figure 1-28. Adding a navigation controller via the Editor menu

As shown in Figure 1-29, this will add a Navigation Controller in between the Tab Bar Controller and Workout Map View Controller. By holding down on the Workout Map View Controller, you can change its position on the storyboard, to make it more visually appealing. As you will notice, there are directed arrows between the Tab Bar Controller, Navigation Controller, and Workout Map View Controller. These are called *segues* and specify how View Controllers are linked to each other on Storyboards, either through relationships (for example, parent-child) or actions (for example, pressing a button). The Embed In function is very convenient, because it prevents you from having to set up all these segues yourself, which, although possible, is very time-consuming and error-prone.

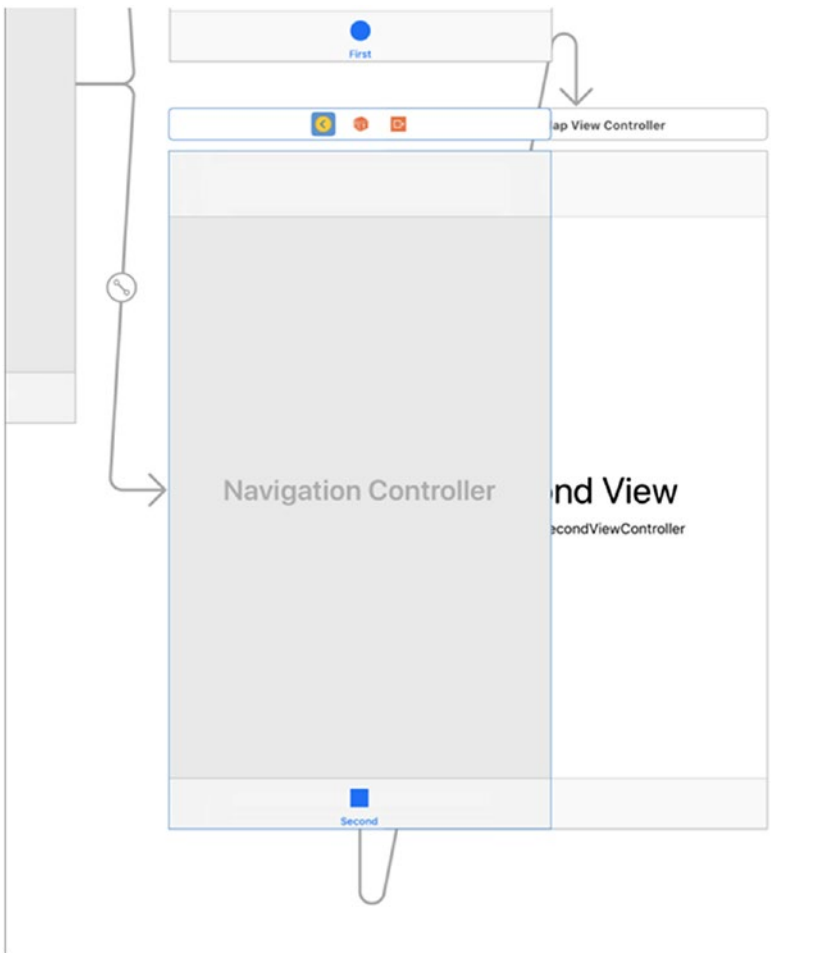


Figure 1-29. Storyboard after embedding Navigation Controller

To customize the title of the Workout Map View Controller, click in the middle of the Navigation Item (bar) on the Workout Map View Controller and begin typing, as shown in Figure 1-30. You may also use the Attributes Inspector to enter your text.

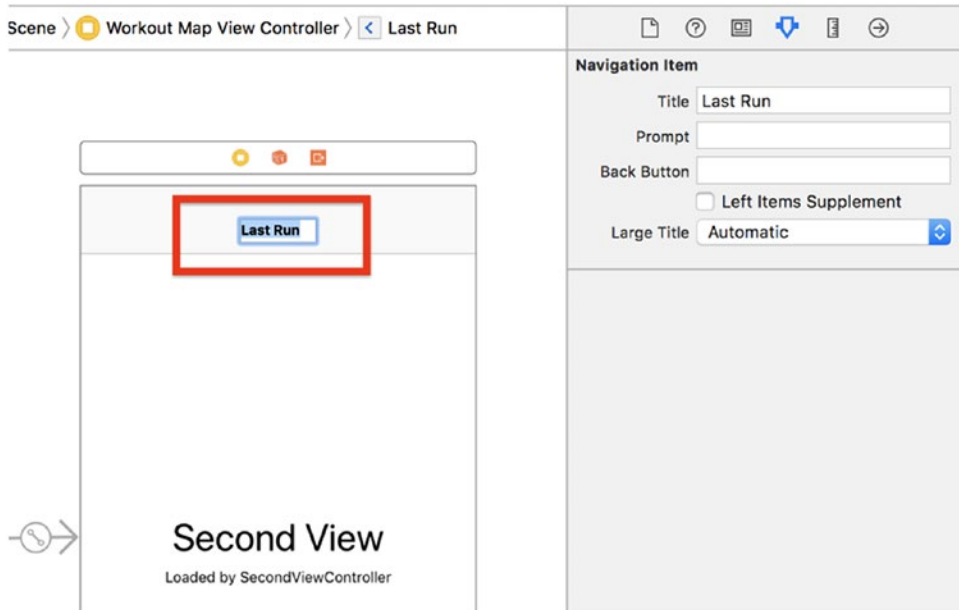


Figure 1-30. *Editing the title for the Workout Map View Controller*

To make the title text large, the new preferred style for iOS 11, click the Navigation Controller, then in the Attributes Inspector, select *Prefers Large Titles*, as shown in Figure 1-31.

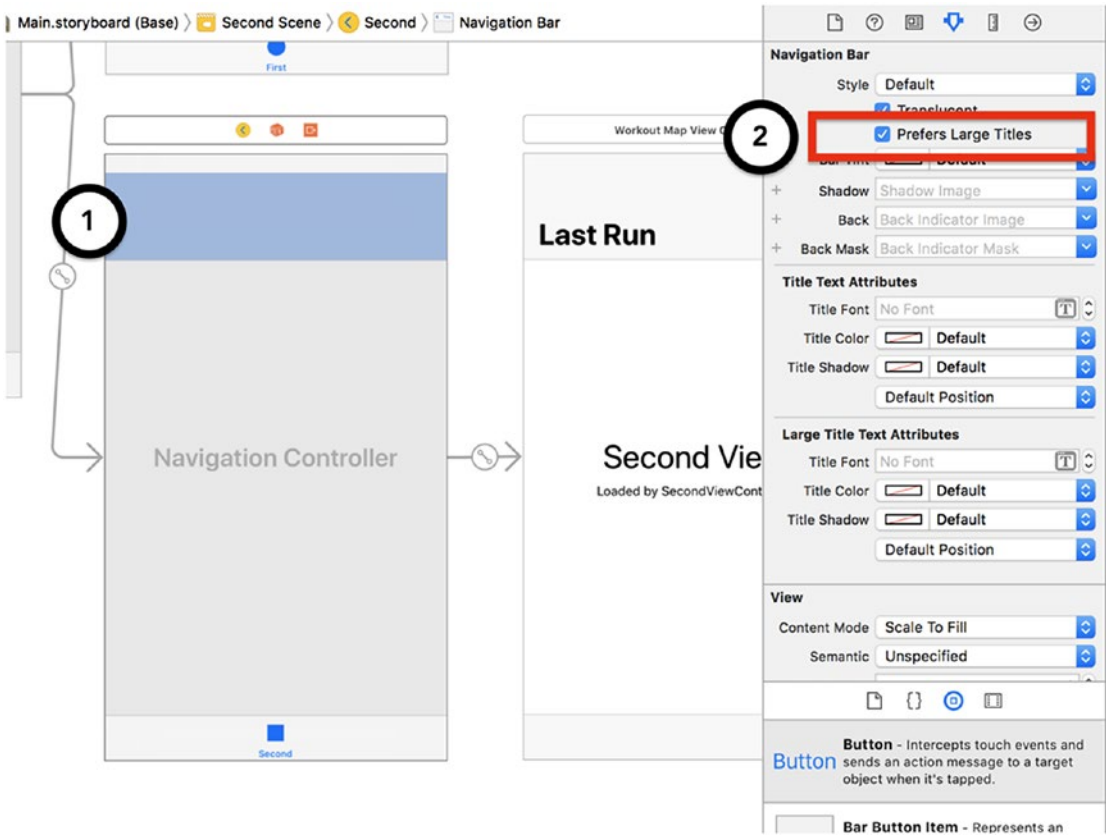


Figure 1-31. Enabling iOS 11-style Navigation Item titles

For the final step in preparing the Workout Map View Controller, you must add a Map View. First, delete the old labels from the template-generated View Controller, drag a Map View from the Object Library and set its Top, Bottom, Right, and Left constraints to 0. Then click the Add 4 Constraints button to apply the constraints. If you must jog your memory on how to do this, go back to the “Applying Auto Layout Constraints” section earlier in this chapter. Your output should look similar to that in Figure 1-32.

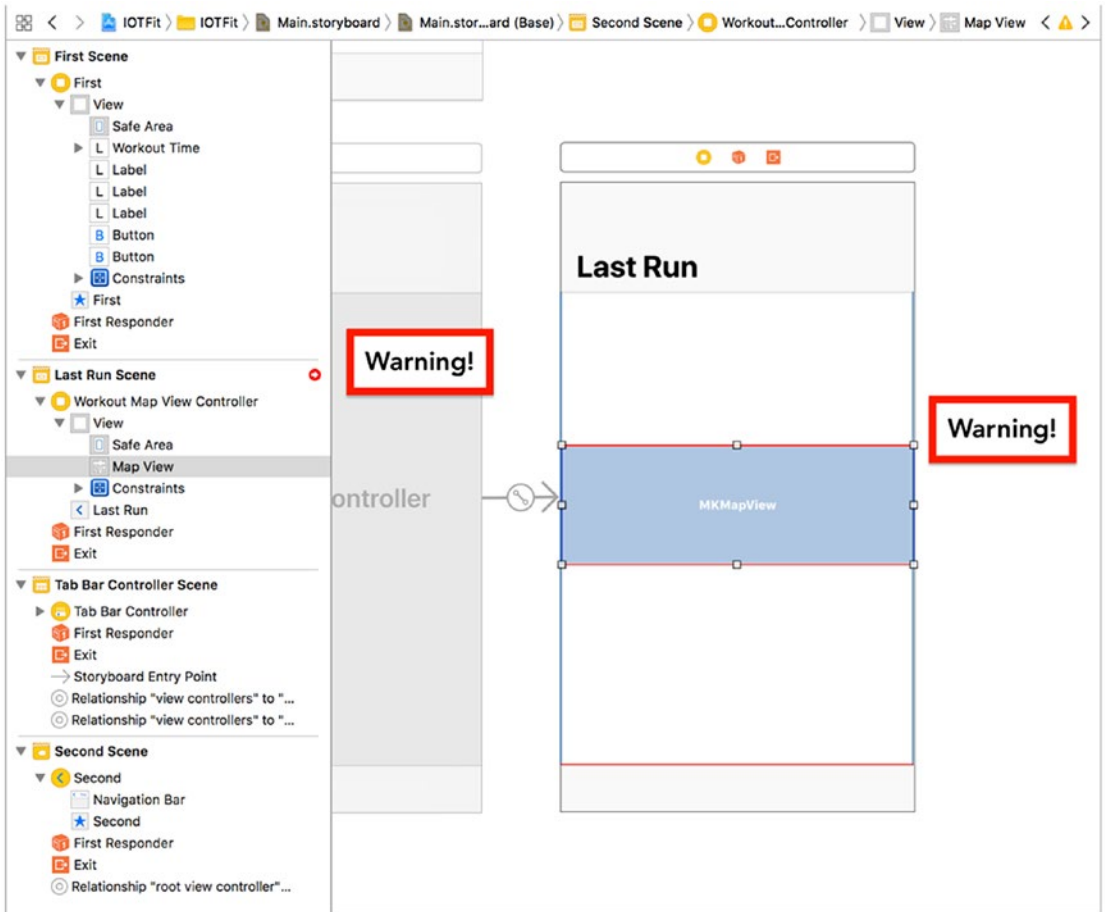


Figure 1-32. Workout Map View Controller after adding Map View

Resolving Auto Layout Issues

The Map was added successfully to the View Controller, as indicated in Figure 1-32, but the frame was not as you might have expected (a small centered rectangle instead of a full-screen one). Don't worry! You did nothing wrong.

One of the issues raised by the introduction of the iPhone X (Apple's first bezel-less phone) is developers' need to detect the new edges of the phone. To make things more complicated, the top edge is interrupted by a sensor bar, and the bottom edge shares space with the new iOS app switcher. To address this issue, Apple introduced the concept of *safe areas* to iOS 11. Safe areas in storyboards act as new boundaries for

Auto Layout. When you pin an item to a safe area, the system will take care of the logic required to work around the “unsafe” areas on the iPhone X. Safe areas can be used on storyboards for all iOS devices.

Unfortunately, sometimes Apple is not ready with all of its APIs, or there is a conflict in how these rules should be applied, such as with the Map View in this example. To resolve this, I will introduce techniques I like to use to obviate Auto Layout conflicts in Interface Builder.

The first place I look when I run into Auto Layout conflicts is the Document Outline (the left pane in Interface Builder). View Controllers with faulty Auto Layout constraints will have a red stop sign icon next to them, like the one that appears to the left in Figure 1-33. If you click the stop sign icon, the Document Outline will drill down into that View Controller and provide the list of faulty constraints (often, it will be more than one). As shown in Figure 1-33, clicking the stop sign one more time will provide you with a pop-up dialog suggesting a solution.

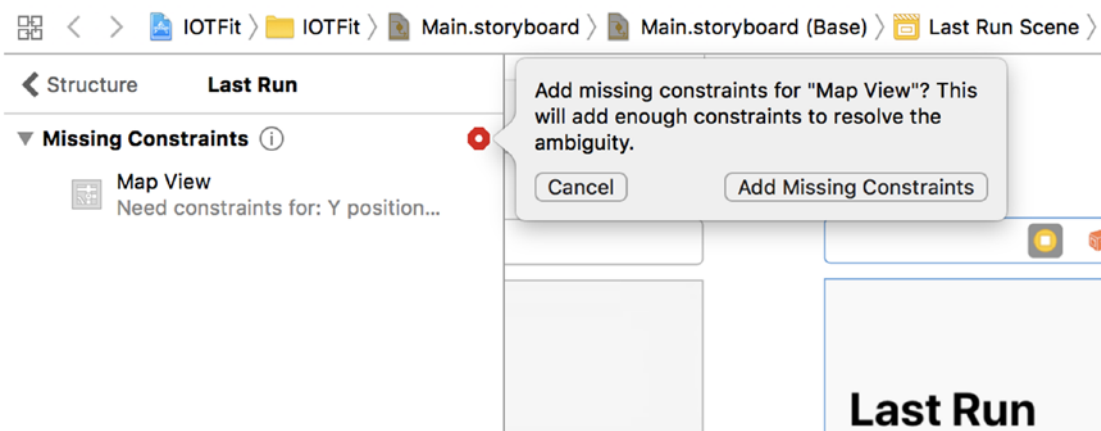


Figure 1-33. Document Online pop-up: Change okay? for solving constraint issues

For this issue, the pop-up was not sufficient; another problem-solving technique is required. My second go-to fix would be to use the Resolve Auto Layout Issues tool at the bottom right of the Interface Builder Editor. As shown in Figure 1-34, clicking this will provide you with a context menu that asks you to fill in missing constraints, delete all constraints for the selected view, or delete all constraints for the View Controller. In my experience, the best use of this tool is to reset the constraints for the troublesome

view and then try to apply the constraints again. Many times, when you rearrange items in a View Controller, you unexpectedly break other constraints that depend on the old position. This tool provides a good way to create a clean slate for one view.

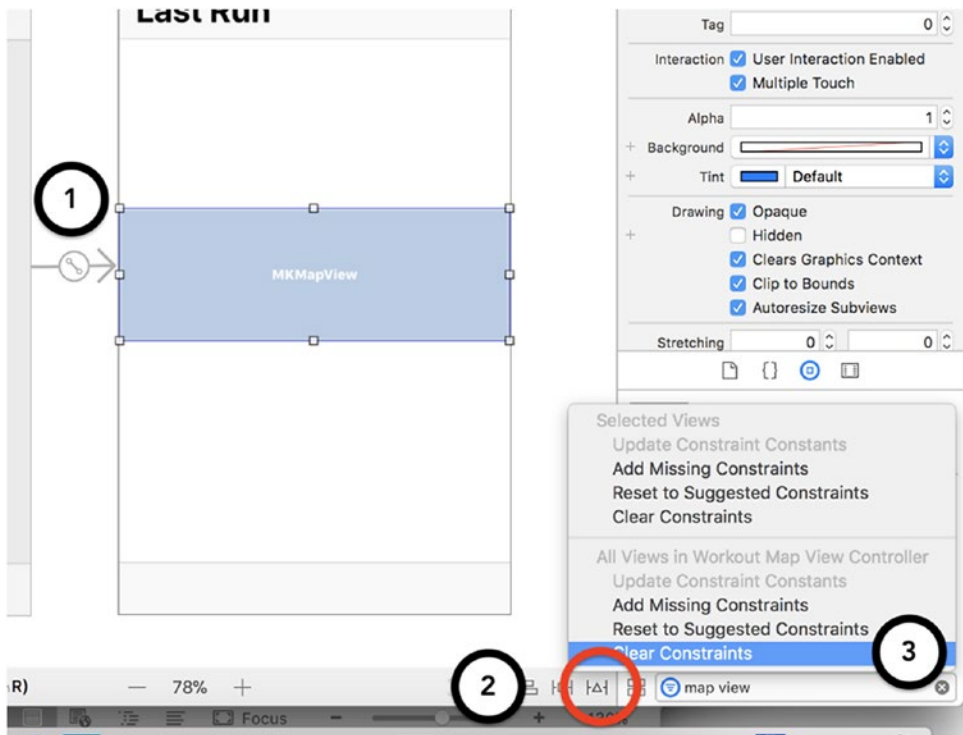


Figure 1-34. Presenting the Resolve Auto Layout Issues context menu

Unfortunately, this did not solve the issue either. When nothing else goes right, I like to open up the Size Inspector tool for the troublesome view (indicated by the Ruler icon in the right-hand side utilities pane), and I like to start manually modifying values until the warnings disappear. As shown in Figure 1-35, you can edit values by typing new ones into the text fields or by clicking Edit, next to a constraint, to bring up a pop-up with its properties.

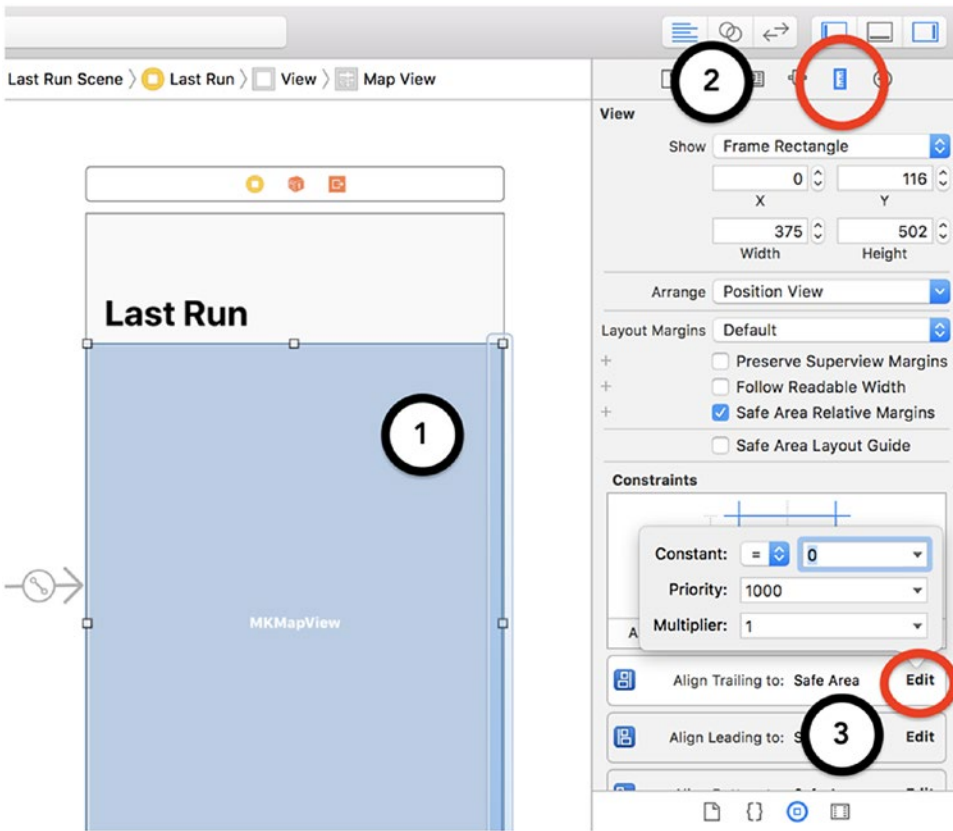


Figure 1-35. Using the Size Inspector to modify constraints

For the Map View, the issue was that the constraints did not pin the top and bottom positions of the view, so I set the y position to 0 (the top of the screen) and make the height match the height of the screen. I verified it by making sure the Document Outline warnings disappeared.

After resolving the Auto Layout issues, the final storyboard for the project should look similar to Figure 1-36.

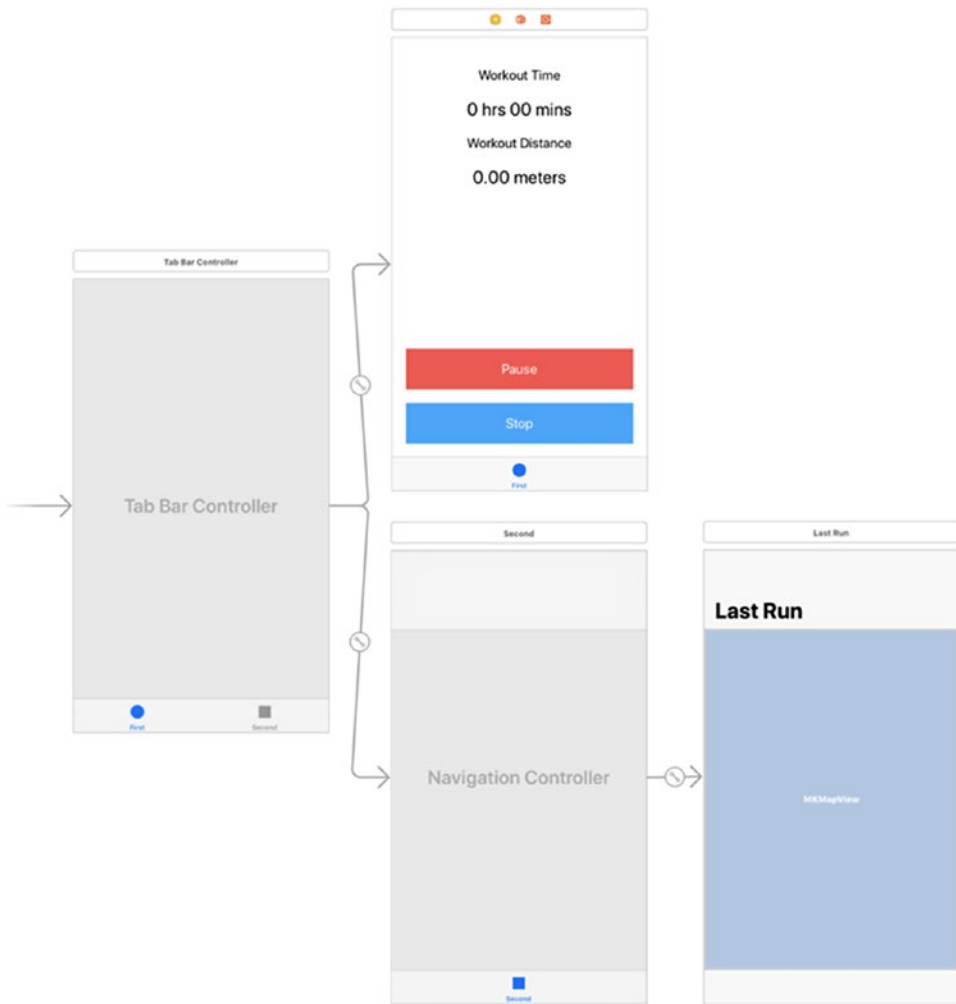


Figure 1-36. Completed storyboard for IOTFit project

Connecting the Storyboard to Your Code

Although the completed storyboard is beautiful, right now it will not do anything if the user presses any of the buttons, because the new user interface objects are not attached to any code. In Xcode, the way you connect your code to your storyboards is by defining properties and methods that must interact with Interface Builder, using special keywords (IBOutlet, IBAction). Then, you find these items in Interface Builder and drag-and-drop connections between the storyboard and your code.

Defining Interface Builder-Compatible Properties and Methods (Actions)

When you created the IOTFit project from Xcode's Tabbed App template, it generated a blank storyboard and blank classes for the first (Create Workout) and second (Map Workout) View Controllers. If you click the `CreateWorkoutViewController.swift` file in the Project Navigator, the editor will show you its contents, shown in Listing 1-1. The `WorkoutMapViewViewController.swift` file will have similar contents, at this point.

Listing 1-1. Initial Contents of the `CreateWorkoutViewController` Class

```
import UIKit

class CreateWorkoutViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
    }
}
```

This is not much to write home about, as most of the critical code required to initialize the View Controller itself is provided by its parent class, `UIViewController`. You may also notice that there is no code for the labels. Generally speaking, you do not have to declare *properties* (class members) for user interface elements whose appearance will not change significantly at runtime. However, items that will change should be represented as properties. For the Create Workout View Controller, you can ignore the Workout Distance and Workout Time labels, but you will have to create properties to represent the labels that hold the values.

Interface builder-compatible properties are initialized much like normal properties, with two strengths: you must pay extra attention to their type and you must add a keyword that allows Interface Builder to expose them to its drag-and-drop interface (`@IBOutlet`).

Unlike a normal variable, user interface properties that are tied to a storyboard will be initialized by that storyboard. Therefore, you need to pay extra attention to the reference strength and type of the variable. For most properties, this means you will have to define the property as being an optional (a variable that can be initialized with a value or remain in a detectable, uninitialized state) and having a weak reference (its contents will not stay in memory indefinitely).

You can find the updated class definition for the Create Workout View Controller, including the new, properly declared properties, in Listing 1-2.

Listing 1-2. CreateWorkoutViewController Class Definition, Including User Interface Properties

```
import UIKit

class CreateWorkoutViewController: UIViewController {

    @IBOutlet weak var workoutTimeLabel: UILabel?
    @IBOutlet weak var workoutDistanceLabel: UILabel?
    @IBOutlet weak var toggleWorkoutButton: UIButton?
    @IBOutlet weak var pauseWorkoutButton: UIButton?

    override func viewDidLoad() {
        super.viewDidLoad()
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
    }
}
```

The process for declaring a method that can be discovered by Interface is similar. Add the `@IBAction` keyword in front of the `func` keyword. For the Create Workout View Controller, there are only two user-initiated actions to worry about at this point: starting/stopping (toggling) a workout and pausing/resuming a workout. The modified definition for the `CreateWorkoutViewController` class, including the new methods, is provided in Listing 1-3.

Listing 1-3. CreateWorkoutViewController Class Definition, Including Interface Builder-Compatible Method Definitions

```
import UIKit

class CreateWorkoutViewController: UIViewController {

    @IBOutlet weak var workoutTimeLabel: UILabel?
    @IBOutlet weak var workoutDistanceLabel: UILabel?
    @IBOutlet weak var toggleWorkoutButton: UIButton?
    @IBOutlet weak var pauseWorkoutButton: UIButton?

    override func viewDidLoad() {
        super.viewDidLoad()
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
    }

    @IBAction func toggleWorkout() {
        NSLog("Toggle workout button pressed")
    }

    @IBAction func pauseWorkout() {
        NSLog("Pause workout button pressed")
    }
}
```

I added `NSLog()` statements to this example, to allow you to view output messages in the Xcode debugging pane when you click the buttons.

Moving on to the Workout Map View Controller, the setup is straightforward. You simply have to add a Map View to the class. All of the user interface operations on this View Controller are handled by the Map View, so there are no additional methods you need to define. One last caveat, though. In order to declare a `MKMapView` (Map View) object, you must import the `MapKit` framework into your class. The modified class definition for the `WorkoutMapViewController` class, including these changes, is included in Listing 1-4.

Listing 1-4. WorkoutMapViewController Class Definition, Including MapKit Framework and Map View Property

```
import UIKit
import MapKit

class WorkoutMapViewController: UIViewController {
    @IBOutlet weak var mapView: MKMapView?

    override func viewDidLoad() {
        super.viewDidLoad()
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
    }
}
```

Using the Connection Inspector to Make the Final Storyboard Connections

Now that the classes for the project have been fully defined, you can use Interface Builder to make the connections. As shown in Figure 1-37, click the `Main.storyboard` file, then select the Create Workout View Controller and click the Connection Inspector (the circled arrow icon in the utilities pane).

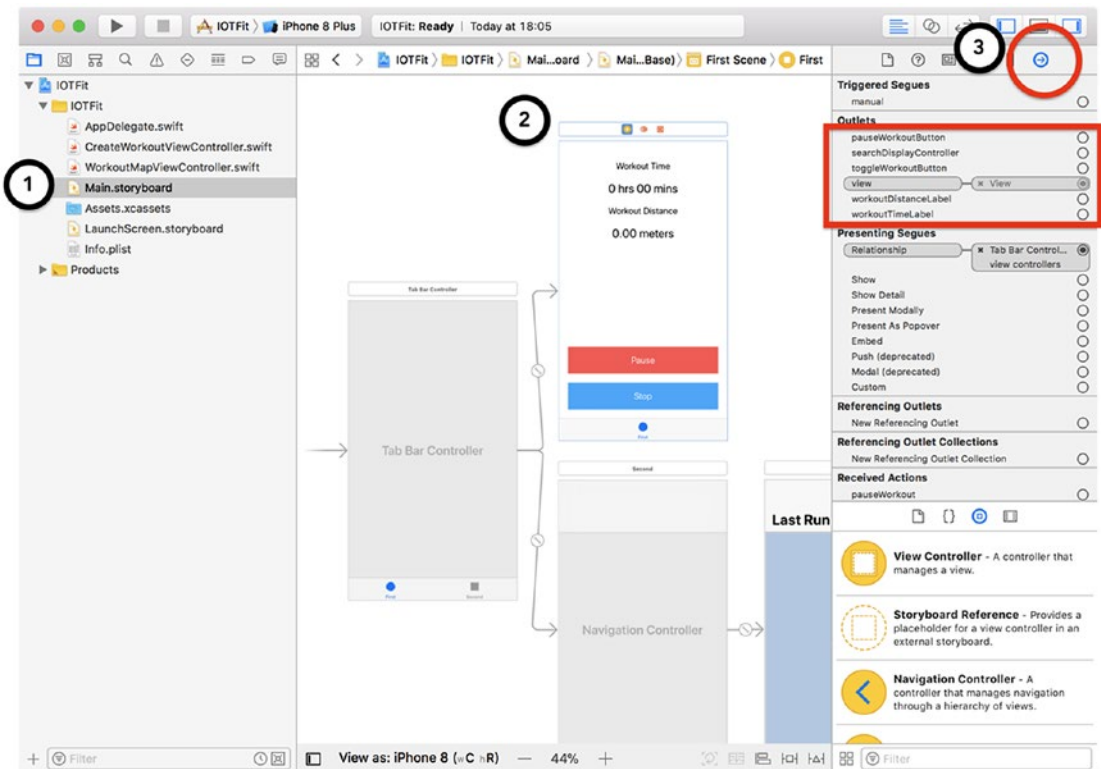


Figure 1-37. Presenting the Connection Inspector for the `CreateWorkoutViewController` class

The Connection Inspector contains a list of all available outlets for your class. Among those will be the properties you just defined (`pauseWorkoutButton`, `toggleWorkoutButton`, `workoutDistanceLabel`, and `workoutTimeLabel`). To connect an item, hold down the radio button next to its name and then release it over the item you want to connect, as shown in Figure 1-38.

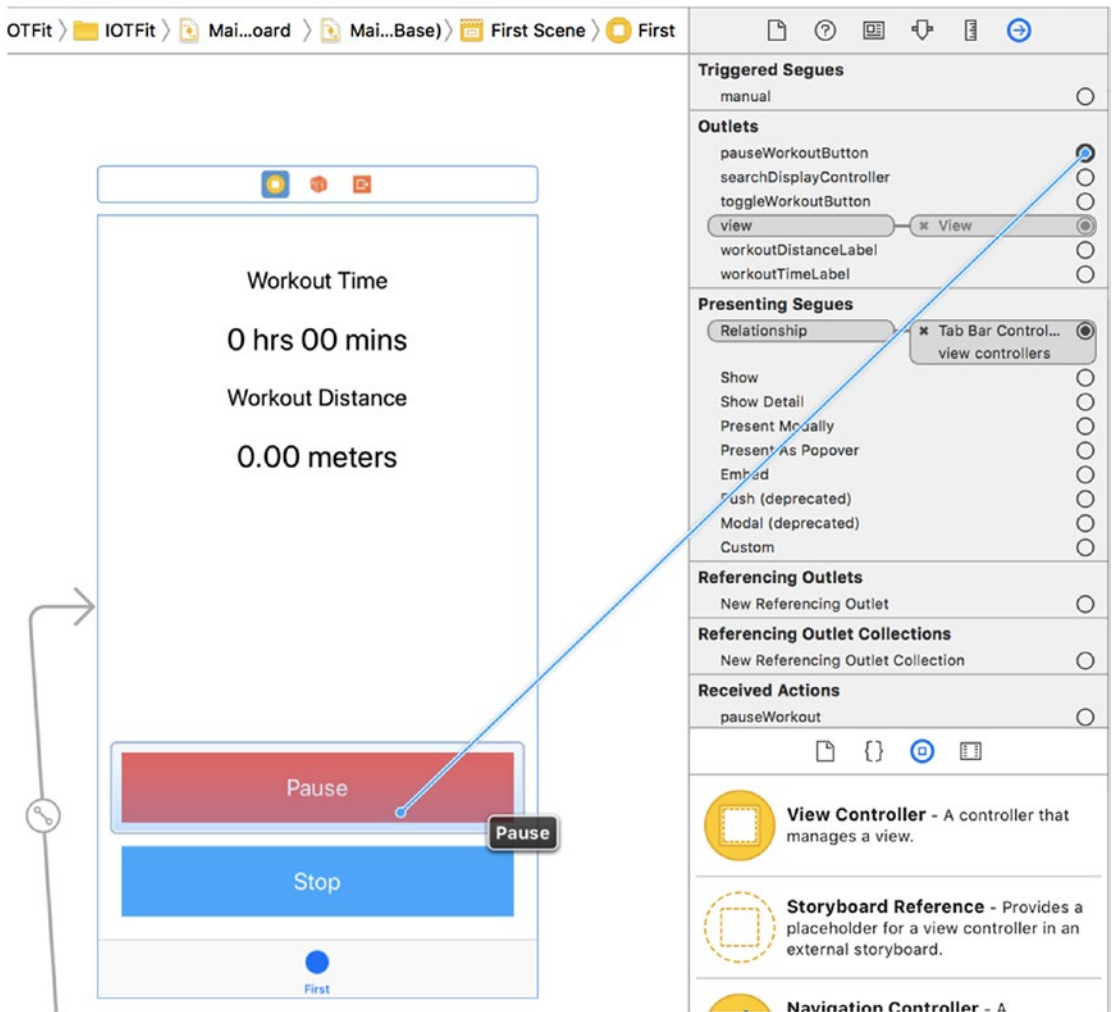


Figure 1-38. Connecting the Pause Workout button to its property

When you have successfully made the connection, the radio button will be replaced with a text bubble containing the property name, as shown in Figure 1-39.

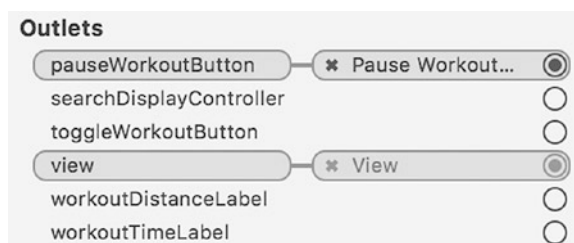


Figure 1-39. Connection Inspector after successfully linking a property

Repeat this process for the rest of the properties in the `CreateWorkoutViewController` and `WorkoutMapViewViewController` classes.

To link a method to a user interface event (such as pressing a button or tapping an area on a view), you will follow mostly the same process, with a few differences. As shown in Figure 1-40, start by selecting the Pause Workout button. The Connection Inspector will now display a list of all the possible user interface events that are applicable to a button, including Touch Up Inside, Touch Down, Value Changed, and Touch Cancel.

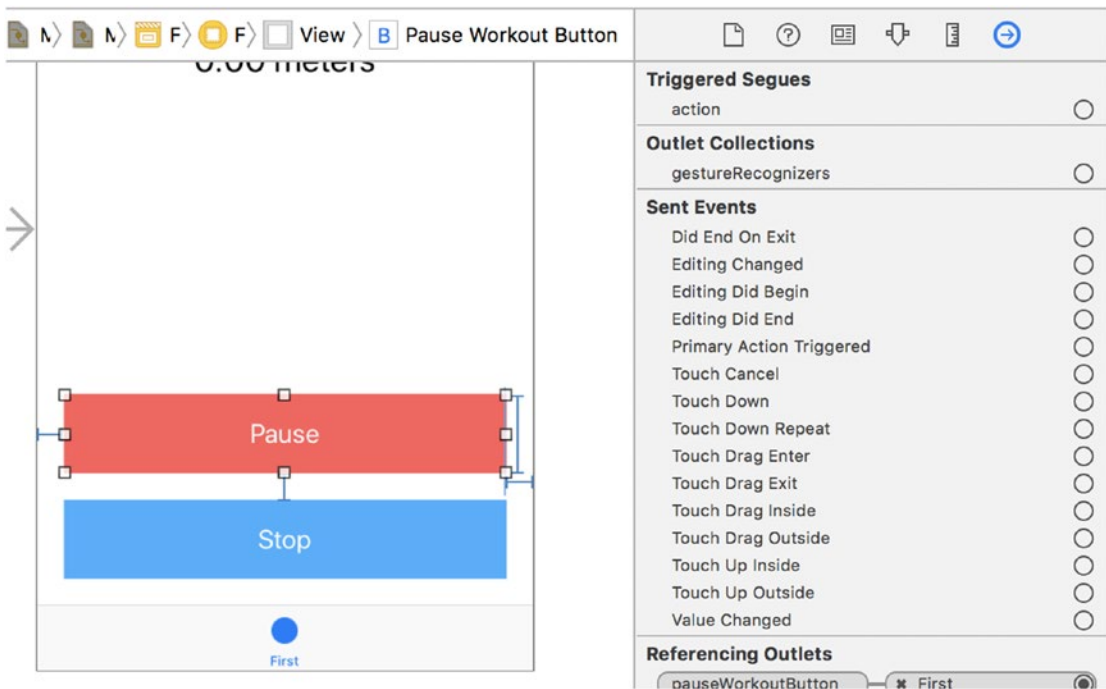


Figure 1-40. Connection Inspector events for the Pause Workout button

The event you will use most often for detecting when the user presses a button is Touch Up Inside. This event fires once after the user holds down and releases a button. To connect the action to its handler method, `toggleWorkout()`, select the radio button in the Touch Up Inside row and then release it over the `CreateWorkoutViewController` class. As shown in Figure 1-41, when you release the mouse, a contextual menu will appear, listing available Interface Builder-compatible methods in the class. Select `toggleWorkout`.

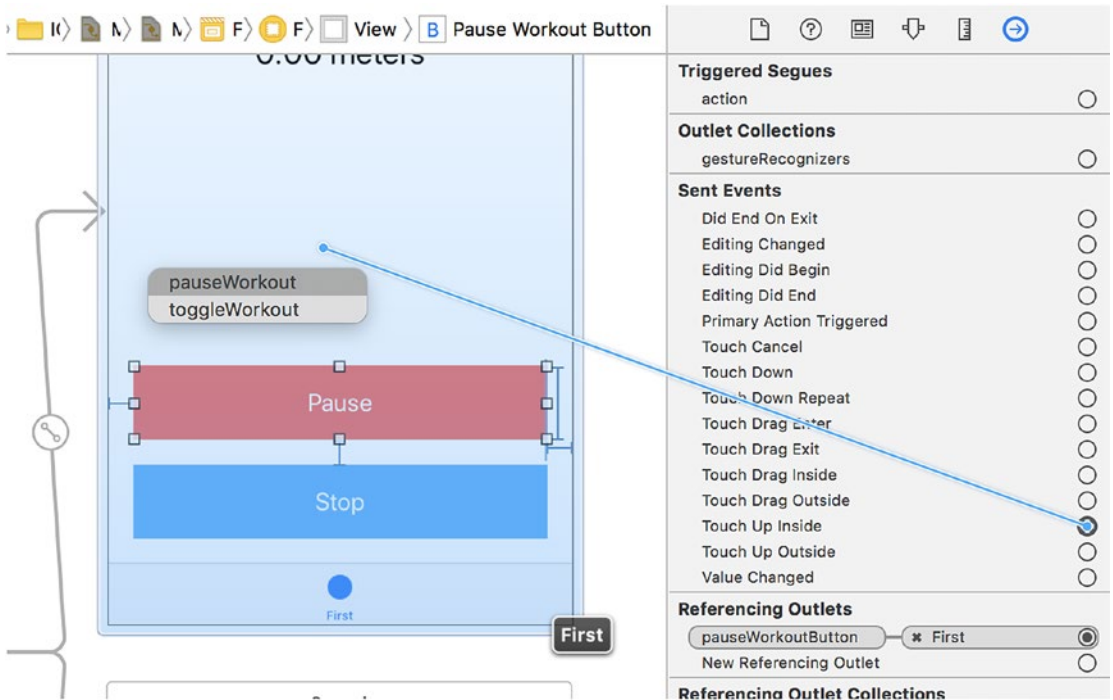


Figure 1-41. Connecting a user interface event to a method in your class

After making your selection, the `pauseWorkout()` method's name should appear in a text bubble next to the Touch Up Inside event, as shown in Figure 1-42.

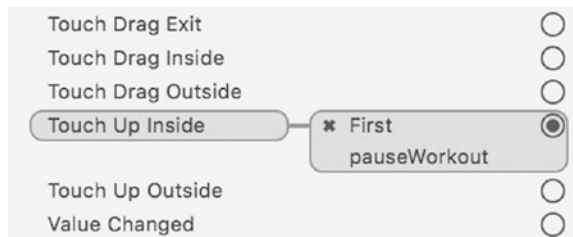


Figure 1-42. Verifying the successful event connection for the Pause Workout button

Repeat this process for the Toggle Workout button and you will be all done! To verify that everything is working as expected, click the Run button at the top right of the Xcode editor window. This will compile and run your application. By default, Xcode will run your project on its default setting (currently, the iPhone 8 Plus simulator), but you can change it to another simulator or a connected device from the drop-down menu

to the right of the Run button. As shown in Figure 1-43, after pressing the Run button, the simulator will start up with your app in the foreground, and when you click the Toggle Workout or Pause Workout buttons, the Debugging Console (bottom right) will display the log messages you specified earlier.

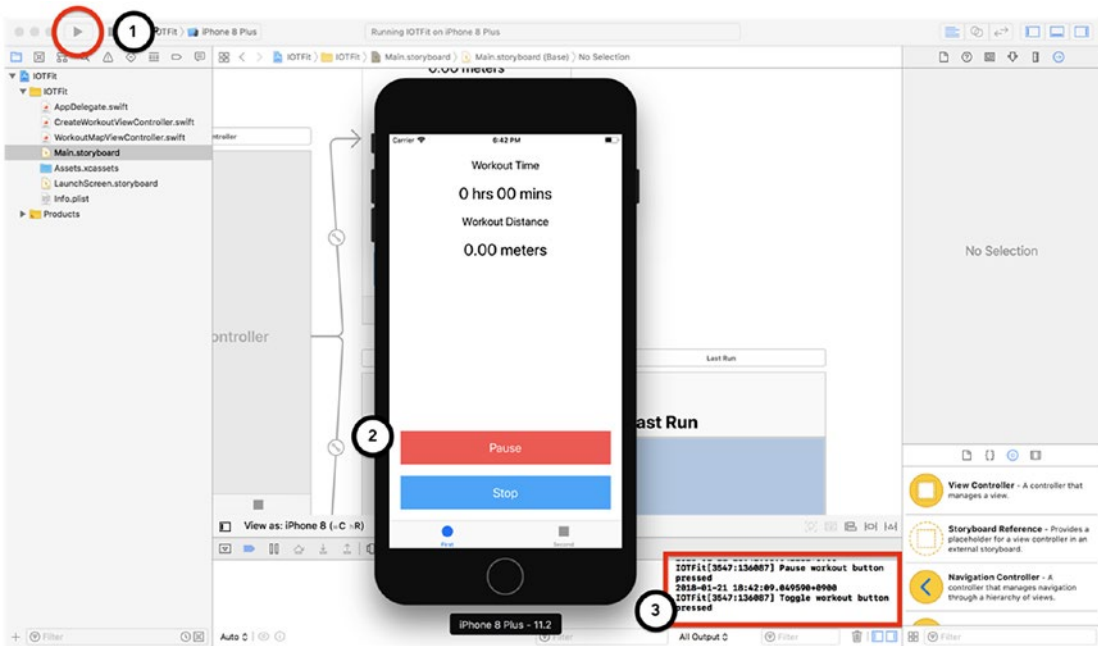


Figure 1-43. Testing the IOTFit app using the Xcode iPhone 8 Plus simulator

Summary

In this chapter, you learned how to perform many critical project life cycle tasks in Xcode, which you will use throughout this book and in your own projects. These tasks included creating a new project, connecting your Apple Developer Program account to Xcode, using Xcode’s refactoring tools, building a user interface, and using Interface Builder to connect storyboards to your code. The chapter was heavy on images and step-by-step instructions, reflecting the visual nature of working with Xcode. In my experience, so much of the joy and frustration of iOS development comes from learning workflows in Xcode. I hope that this chapter made the learning process a little less painful!

CHAPTER 2

Using Core Location to Build a Workout Tracking App

In the first chapter, you learned how to use Xcode and Interface Builder to set up the project for the IOTFit workout app and create its user interface. However, due to the complicated setup process, you did not have an opportunity to make it a true Internet of Things (IoT) app by accessing the GPS hardware on the phone. In this chapter, you will learn how to take advantage of Apple's Core Location framework to request location permission from the user, receive periodic updates on the user's location, and plot those locations on a map.

This chapter will begin addressing a misconception that you will hear throughout your time as an iOS developer: the platform is too hard and inconvenient to use. Although there are elements of truth to this, as evidenced by Chapter 1's complicated user interface setup process, you will learn that as you become more familiar with Apple's development patterns, the learning curve for picking up new frameworks will drop significantly. Additionally, you will notice that the amount of work to learn a new framework is significantly less than the work required to implement all of the functionality yourself. In the case of Core Location, the framework does the hard work of making the location requests, processing the data from the GPS hardware, and delivering it via asynchronous events that your app can handle. Imagine having to write all of this by yourself, on a deadline!

In keeping with the iterative process of this book, this chapter will focus on how to implement the features of the IOTFit app that are responsible for recording a workout's duration, tracking the path the user took during the workout, and displaying this location information on a map.

Learning Objectives

In this chapter, you will learn the following critical skills for IoT development on iOS by building the location-based functions of the IOTFit application:

- Configuring an application for background activity
- Checking for availability of hardware resources
- Asking the user for permission to access sensitive permissions
- Requesting and responding to location updates
- Displaying saved locations on a map

One of the most critical lessons you will learn in this chapter is the workflow of checking if a hardware feature is available and then asking the user for permission to access that resource before doing any work with it. Not only is this a good strategy to help you design your app's flow (for example, thinking about what to do when the user does not want to allow access), but it also helps you prevent some of the most common sources of runtime crashes. Apple's SDKs will cause your applications to crash, if you try to access unavailable or forbidden resources.

As with the first chapter, the code for this project is available on the GitHub repository for this book, under the Chapter 2 folder (<https://github.com/Apress/program-internet-of-things-w-swift-for-ios>).

Configuring Your Project for Background Location Activity

During a workout, users will often put their phone in their pocket, in a case on their arm, or on top of their workout equipment. To enable the IOTFit app to be useful to users in these cases, you should configure the IOTFit app to continue workout tracking while the app is backgrounded. Users will have to enable this functionality the first time the app is launched, via an alert that iOS presents in your app. To allow these features to work, you will learn how to declare the app as one that would like to use background location updates and how to configure the message for the permission alert. It is important to perform these setup steps early in the development process, as both affect how your app is compiled and can only be configured through the Project Settings editor.

Apple imposes strict limitations on the background actions individual apps can take in iOS, to save power and prevent memory leaks. Unless your app is correctly configured to ask for permission to a background-enabled feature (mode), your app will not be able to perform any actions while it is in the background. The special features Apple allows you to configure your app to access in the background are described in Table 2-1. For the IOTFit application, you will use background location updates. In later chapters in the book, you will enable the Bluetooth LE features.

Table 2-1. *Configurable Background Modes for iOS Apps*

Background Mode Name	Purpose
Audio, AirPlay, and Picture-in-Picture	Allows multimedia apps to continue playback uninterrupted when the user backgrounds the app
Location updates	Allows developers to perform small tasks, based on location change events that happen while an app is backgrounded
Newsstand downloads	Allows Newsstand apps (for example, magazines, newspapers) to fetch new content while the app is inactive
External accessory communication	Allows apps to maintain active data channels with physically connected, Made for iPhone (MFI) hardware
Uses Bluetooth LE accessories	Allows an app to act as a Bluetooth LE central manager and communicate with external devices that are configured as Bluetooth LE Peripherals
Acts as a Bluetooth LE accessory	Allows an app to serve as a Bluetooth LE Peripheral and accept messages from Bluetooth LE central manager devices
Background fetch	Allows an app to periodically fetch data from HTTPS end points in the background (frequency determined by iOS task manager)
Remote notifications	Allows apps to respond to Apple Push Notifications

To start developing this chapter's iteration of the IOTFit app, begin by copying your completed project from Chapter 1 or by downloading the code from the Chapter 1 folder of this book's GitHub repository (<https://github.com/Apress/program-internet-of-things-w-swift-for-ios>).

Open your new project and click the project name (IOTFit) in the Project Navigator (within Xcode's left pane). To configure the IOTFit project to use the Location updates background mode, click on the Capabilities tab of the Project Settings editor view, indicated in Figure 2-1.

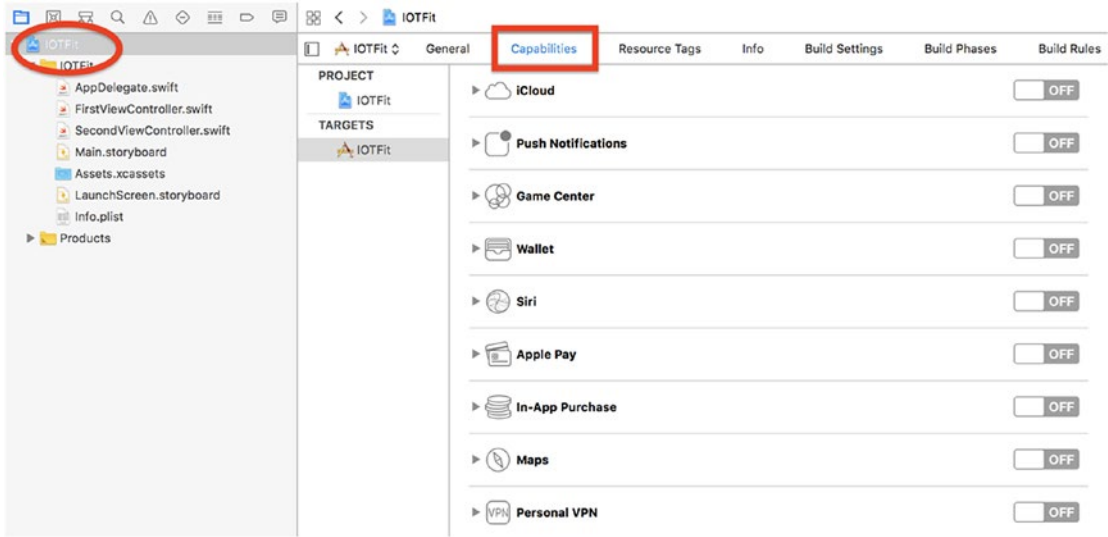


Figure 2-1. Capabilities tab within Project Settings

Scroll down to Background Modes and click the switch to turn it on. Click the check box labeled Location updates, to enable background location updates. After the changes, your capabilities screen should match Figure 2-2.

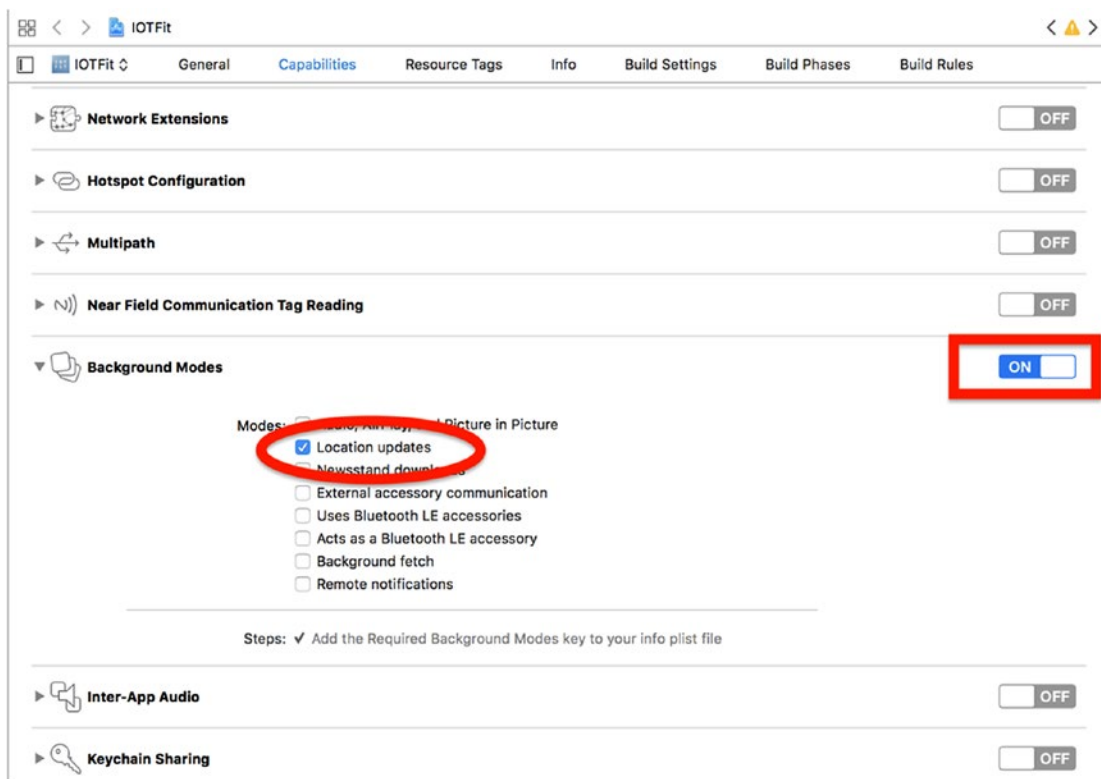


Figure 2-2. Capabilities tab with location updates configured correctly

Note If you did not finish linking your Apple ID account to Xcode, you will not be able to successfully set any background modes.

Although it would be convenient for everything to be managed on the same screen, you will have to switch to the Info tab, to edit the permission messages for the app. Click the Info tab in Project Settings. You will be presented with a property list editor populated with the default settings for your project, as shown in Figure 2-3.

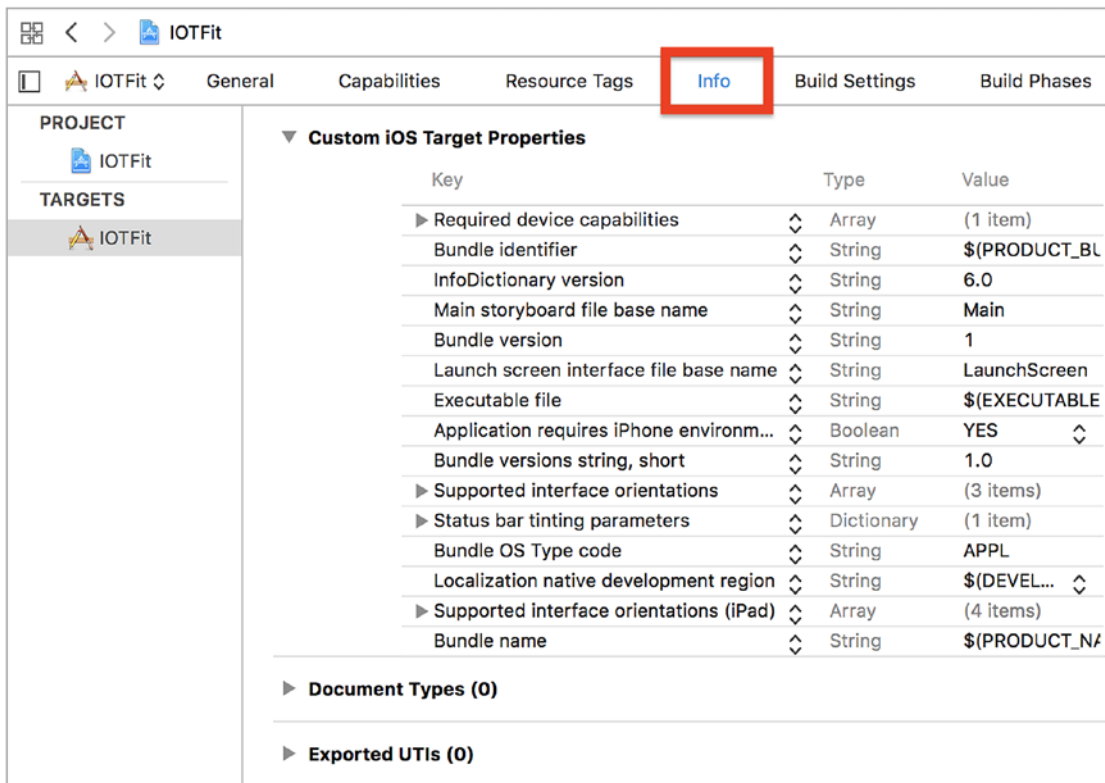


Figure 2-3. Default property list within Info tab

Property lists (.plist files) are XML-based text files. Configuration settings within a property list are stored as key-value pairs (dictionaries). Their ease of readability and ease of use make them Apple’s preferred method for managing optional project settings. This file is usually referred to by its file name, Info.plist.

Although you can edit property list files in a text editor, Xcode provides the visual editor you saw in Figure 2-3 to help make property list management even easier. This editor will show up for both auto-generated files (such as Info.plist) and files you create manually.

To add a key-value pair to a property list, click any row and then click the plus (+) button. For project settings, a drop-down menu will appear with suggested build settings keys. Scroll down the menu and select Privacy - Location Always and When In Use Usage Description as shown in Figure 2-4.

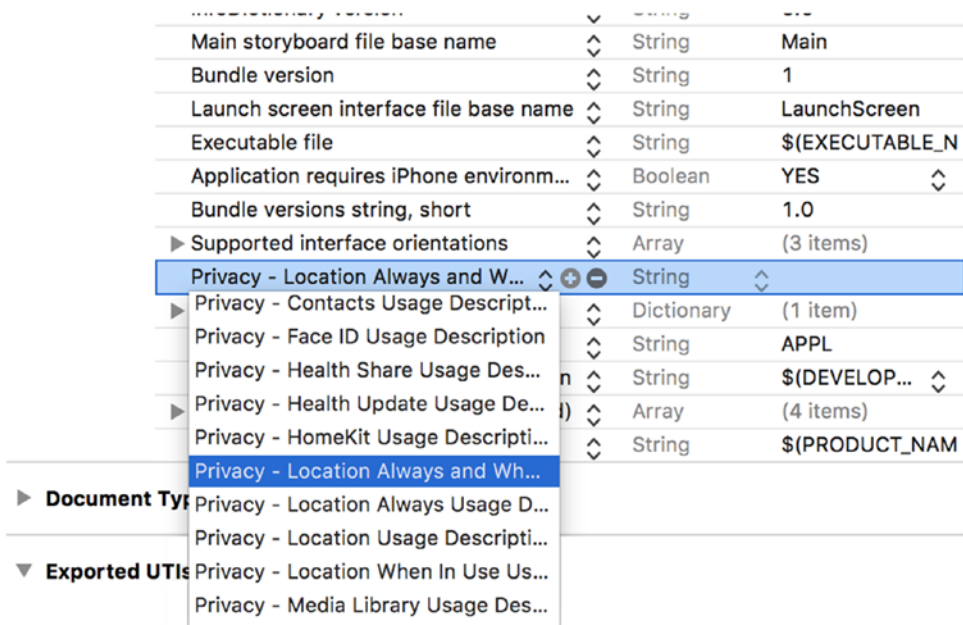


Figure 2-4. Adding a key-value pair to the Info property list

Double-click the blank space in the Value column for the new row and enter the text you would like to display in the permission alert window. One of Apple’s App Store submission guidelines is that your permission prompts describe your intended use of a protected feature. For this application, I used the following string as my permission prompt description:

IOTFit would like to use location permission to plot your location during workouts. This information will not be shared outside of the app.

There are three variations for location permissions that users can select in iOS: Always, When In Use, and Always and When in Use. Create additional rows for the Privacy - Location When In Use Usage Description and Privacy - Location Usage Description key-value pairs and specify the permission prompts for those as well. When your work is complete, your output should be similar to that in Figure 2-5.

▼ Custom iOS Target Properties

Key	Type	Value
Bundle name	String	\$(PRODUCT_NAME)
Launch screen interface file base name	String	LaunchScreen
Localization native development region	String	\$(DEVELOPMENT_LANGUAGE)
Bundle version	String	1
▶ Required background modes	Array	(1 item)
▶ Status bar tinting parameters	Dictionary	(1 item)
Bundle OS Type code	String	APPL
Main storyboard file base name	String	Main
Bundle versions string, short	String	1.0
InfoDictionary version	String	6.0
Executable file	String	\$(EXECUTABLE_NAME)
▶ Required device capabilities	Array	(1 item)
Privacy - Location Usage Description	String	IOTFit would like to use location permission to plot y
▶ Supported interface orientations (iPad)	Array	(4 items)
Bundle identifier	String	\$(PRODUCT_BUNDLE_IDENTIFIER)
Application requires iPhone environm...	Boolean	YES
▶ Supported interface orientations	Array	(3 items)
Privacy - Location Always Usage...	String	IOTFit would like to use location permission to plot y
Privacy - Location Always and When I...	String	IOTFit would like to use location permission to plot y
Privacy - Location When In Use Usag...	String	IOTFit would like to use location permission to plot y

▶ Document Types (0)

▼ Exported UTIs (0)

Figure 2-5. Completed Info property list, including all location permission strings

Note The position where your new key-value pairs appear in the list has no impact on compilation.

Requesting Location Permission

Now that the project dependencies have been taken care of, you can start implementing the logic to request location permission from the user. One of the most critical questions you need to ask yourself at this point is: “Where should I place the location permission pop-up?”

To begin answering this question, first take a look at Figure 2-6, to review the user interface you built in Chapter 1. The user can start or stop a workout by pressing the Start button and can pause or resume a workout by pressing the Pause button. If users want to view their map, they press the Second tab to switch to the Map View.

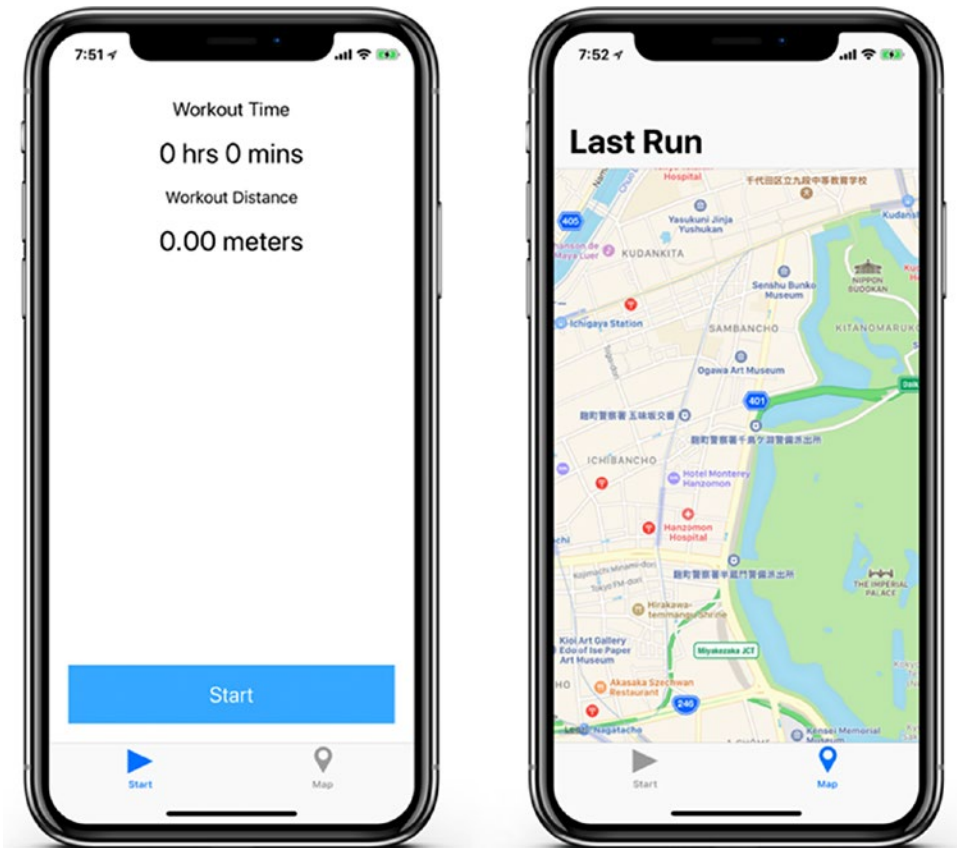


Figure 2-6. Completed IOTFit user interface from Chapter 1

The recent trend in mobile applications is to ask for a permission the first time a user attempts to perform an action that requires the resource you want to use. In the case of the IOTFit app, this would be the first time the user presses the Start button. In addition to asking for location permission, at that time, you would also update the user interface to indicate that the workout has started. Once you have confirmed that everything is ready, you can start recording the data for the user's workout.

Thinking about the other activities on the Create Workout View Controller (pausing and stopping the workout), these would require additional user interface updates and some kind of mechanism to pause the location and time tracking.

In Figure 2-7, I have created a flowchart that records all of these decisions. You will use this throughout the chapter as a guide to implementing the behavior of the Create Workout View Controller.

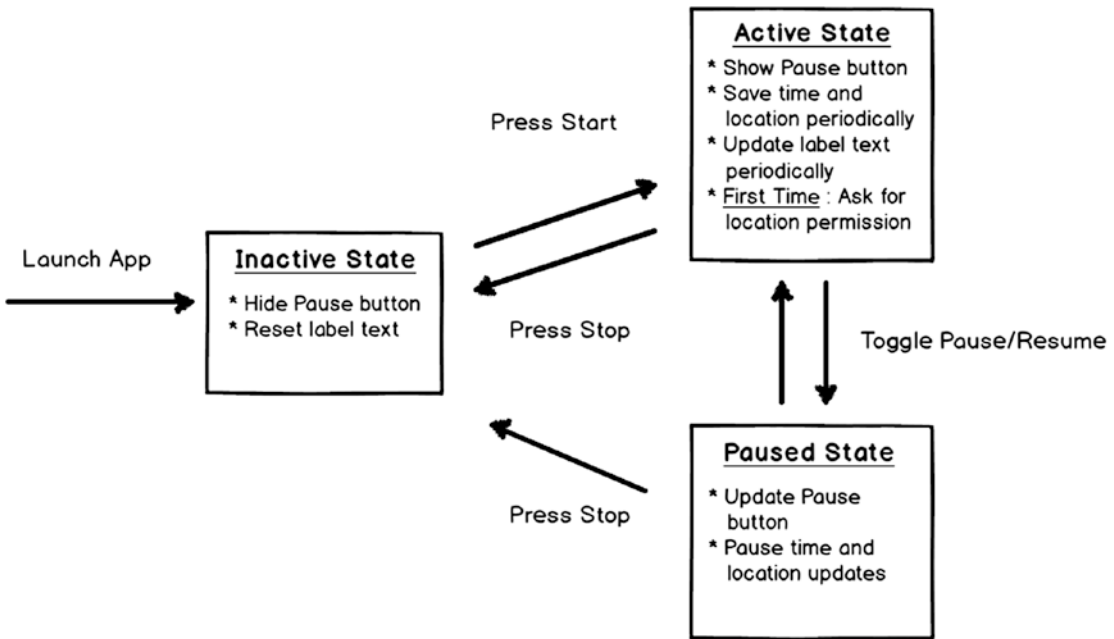


Figure 2-7. Flowchart for the Create Workout View Controller

The transition between the Inactive and Active states is handled by the `toggleWorkout()` method, which is tied to the Touch Up Inside event for the Toggle Workout button. The code for the location request should be initiated there. To help jog your memory, in Listing 2-1, I have provided the code from the `CreateWorkoutViewController` class that was implemented in Chapter 1.

Listing 2-1. `CreateWorkoutViewController` Class from Chapter 1

```

import UIKit

class CreateWorkoutViewController: UIViewController {

    @IBOutlet weak var workoutTimeLabel :UILabel?
    @IBOutlet weak var workoutDistanceLabel :UILabel?

    @IBOutlet weak var toggleWorkoutButton :UIButton?
    @IBOutlet weak var pauseWorkoutButton :UIButton?

    override func viewDidLoad() {
        super.viewDidLoad()
    }
}
  
```

```

override func didReceiveMemoryWarning() {
    super.didReceiveMemoryWarning()
}

@IBAction func toggleWorkout() {
    NSLog("Toggle workout button pressed")
}

@IBAction func pauseWorkout() {
    NSLog("Pause workout button pressed")
}
}

```

To represent the different states of the app, you can create an enum that corresponds to the states in the flowchart. Inside the class, create a property that uses the enum as its type. When the user presses the Start button, update the state property and make the call to request location permission. To implement this logic, update the `CreateWorkoutViewController` class, as indicated in Listing 2-2.

Listing 2-2. Adding State Tracking to the `CreateWorkoutViewController` Class

```

import UIKit

enum WorkoutState {
    case inactive
    case active
    case paused
}

class CreateWorkoutViewController: UIViewController {

    @IBOutlet weak var workoutTimeLabel: UILabel?
    @IBOutlet weak var workoutDistanceLabel: UILabel?
    var currentWorkoutState = WorkoutState.inactive
    ...
}

```

```

@IBAction func toggleWorkout() {
    switch currentWorkoutState {
        case .inactive:
            currentWorkoutState = .active
            requestLocationPermission()
        case .active:
            currentWorkoutState = .inactive
        default:
            NSLog("toggleWorkout() called out of context!")
    }
    NSLog("Toggle workout button pressed")
}

func requestLocationPermission() {
    NSLog("Location permission requested")
}

...
}

```

Checking for Hardware Availability

As I mentioned at the beginning of the chapter, Apple’s recommended design pattern is to first check if a hardware device is available and ask for permission to use it. Apple has always differentiated its devices by hardware features, favoring the newest device with the most advanced features. As a developer, you naturally want to use these features, but making decisions based on device model number becomes unwieldy quickly. For many hardware features, including the iPhone’s GPS and camera, there is an API that allows you to determine the availability of the feature.

In the case of the GPS, the framework that manages its operation is Core Location. It provides a class method, `CLLocationManager.locationServicesEnabled()`, that you can use to query the availability of location on the device. To use this method, you must import the Core Location framework into the `CreateWorkoutViewController` class. Update the class, as indicated in Listing 2-3.

Listing 2-3. Adding Location Service Querying to the CreateWorkoutViewController Class

```

import UIKit
import CoreLocation

class CreateWorkoutViewController: UIViewController {
    ...

    func requestLocationPermission() {
        if CLLocationManager.locationServicesEnabled() {
            NSLog("Location services are available")
        } else {
            presentEnableLocationAlert()
        }
    }

    func presentEnableLocationAlert() {
        let alert = UIAlertController(title: "Permission
            Error", message: "Please enable location services on your
            device", preferredStyle: UIAlertControllerStyle.alert)
        let okAction = UIAlertAction(title: "OK", style:
            UIAlertActionStyle.default, handler: nil)
        alert.addAction(okAction)
        self.present(alert, animated: true, completion: nil)
    }

    ...
}

```

As you will notice, I added a method to present an Alert View, if location services are not enabled globally on the device (owing to user settings or hardware unavailability). This is a good way to inform your users that location permissions are important for the best experience with your app.

Responding to Changes in Location Permission Status

Having verified that the GPS hardware is available on the user’s device, you are now ready to request location permission from iOS. Once again, the Core Location framework manages this operation. However, in order to respond to the changes, you must instantiate an object that takes care of the interface with the Core Location framework, and you have to declare the `CreateWorkoutViewController` class as *delegate* of the `CLLocationManagerDelegate` protocol.

A *protocol* is a programming concept that allows you to define a light interface between two classes. These are used frequently in Apple’s hardware frameworks, in which you just need to be able to call one or two methods on the target hardware, without knowing all the details of its implementation. An object that implements the functions specified by a protocol is called a delegate. It plays the role of the object that initiates the call to the hardware and *receives* the output.

To manage the interface to the Core Location framework, add a `CLLocationManager` property to the `CreateWorkoutViewController` class. As shown in Listing 2-4, initialize the object when you declare it and then inside the `toggleWorkout()` method, set the delegate property of the manger object to `self` (a reference to the `CreateWorkoutViewController` class). This will allow the class to respond to messages from the Core Location framework.

Listing 2-4. Adding a `CLLocationManager` Property to the `CreateWorkoutViewController` Class

```
class CreateWorkoutViewController: UIViewController {
    ...
    let locationManager = CLLocationManager()
    @IBOutlet weak var workoutTimeLabel: UILabel?
    ...
    func requestLocationPermission() {
        if CLLocationManager.locationServicesEnabled(){
            locationManager.delegate = self
        }
    }
}
```

```

        NSLog("Location services are available")
    } else {
        ...
    }
}
...
}

```

The Swift compiler will flag your code with an error saying that you are attempting to assign the property to an incompatible class. To resolve this issue, add the definition for the `CLLocationManagerDelegate` protocol.

The way I like to declare classes as implementing protocols in Swift is by adding an *extension* under the class definition. In the long term, this helps with code clarity. An extension is a code block that allows you to add additional functionality to a class without modifying the original class itself. A common example is when you want to *extend* the `UIColor` class by adding a method for a color you have open in your app that is not defined by Apple.

The extension for the `CreateWorkoutViewController` class is provided in Listing 2-5. The protocol method you must implement to handle location permission status updates is `func locationManager:didChangeAuthorizationStatus:`. If you already requested permission before, the delegate method will immediately return with the authorized value. If you have not requested permission before, it will return when the user has made his or her decision.

Listing 2-5. Adding an Extension to the `CLLocationManagerDelegate` Protocol

```

func requestLocationPermission() {
    if CLLocationManager.locationServicesEnabled(){
        ...
    } else {
        ...
    }
}
}
} //end of class

```

extension CreateWorkoutViewController:

```

CLLocationManagerDelegate {
    func locationManager(_ manager: CLLocationManager,
                        didChangeAuthorization status: CLAuthorizationStatus) {
        NSLog("Received permission change update!")
    }
}

```

Similar to how the workout's state is managed by the `WorkoutState` enum you defined, the permission state for your app is managed by the `CLAuthorizationStatus` enum provided by the Core Location framework. The possible authorization status values and their implications are listed in Table 2-2.

Table 2-2. Core Location Authorization States

Value	User Action	Impact on App
<code>notDetermined</code>	The user has not seen the permission pop-up for your app yet.	Your app cannot use any location-based features until you present the location permission pop-up and the user approves it.
<code>restricted</code>	Due to Parental Controls or Mobile Device Management settings, location services are disabled on their device.	Your app cannot use any location-based features until the management policy changes.
<code>denied</code>	The user has denied location services for your app.	Your app cannot use any location-based features until the user allows permission status in the iOS Settings app.
<code>authorizedWhenInUse</code>	The user has allowed your app to use location services while the app is in the foreground.	Your app can access location services only when the app is in the foreground.
<code>authorizedAlways</code>	The user has allowed your app to use location services in the foreground and background.	Your app can use location services while it is active and in the background.

The `didChangeAuthorizationStatus()` method will fire for all states. Similarly, when you check for location permission, the call you make should be different, based on the user's existing authorization state. You can find out the current authorization state by calling the `CLLocationManager.authorizationStatus()` method. In Listing 2-6, I have updated the `requestLocationPermission()` and `didChangeAuthorizationStatus()` methods to include the new authorization state-based logic.

Listing 2-6. Using Authorization State to Present Location Permission and Respond to Changes in Authorization Status

```
class CreateWorkoutViewController: UIViewController {
    ...
    func requestLocationPermission() {
        if CLLocationManager.locationServicesEnabled(){
            locationManager.delegate = self

            switch(CLLocationManager.authorizationStatus()) {
                case .notDetermined:
                    locationManager.requestWhenInUseAuthorization()
                case .authorizedWhenInUse :
                    requestAlwaysPermission()
                case .authorizedAlways:
                    startWorkout()
                default:
                    presentPermissionErrorAlert()
            }
        } else {
            presentEnableLocationAlert()
        }
    }

    func requestAlwaysPermission() {
        if let isConfigured = UserDefaults.standard.value(forKey:
        "isConfigured") as? Bool, isConfigured == true {
            startWorkout()
        }
    }
}
```

```

    } else {
        locationManager.requestAlwaysAuthorization()
    }
}

func startWorkout() {
    currentState = .active
    UserDefaults.standard.setValue(true, forKey:
        "isConfigured")
    UserDefaults.standard.synchronize()
}

func presentPermissionErrorAlert() {
    let alert = UIAlertController(title: "Permission
        Error", message: "Please enable location services
        for this app", preferredStyle:
        UIAlertControllerStyle.alert)
    let okAction = UIAlertAction(title: "OK", style:
        UIAlertActionStyle.default, handler: nil)
    alert.addAction(okAction)
    self.present(alert, animated: true, completion: nil)
}
...
}

extension CreateWorkoutViewController :
    CLLocationManagerDelegate {
    func locationManager(_ manager:
        CLLocationManager, didChangeAuthorization
        status: CLAuthorizationStatus) {
        switch status {
        case .authorizedWhenInUse:
            requestAlwaysPermission()
        case .authorizedAlways:
            startWorkout()
        case .denied:
            presentPermissionErrorAlert()
        }
    }
}

```

```

default:
    NSLog("Unhandled Location Manager Status:
        \(status)")
    }
}

```

In the `requestLocationPermission()` method, I added a `switch` statement to direct the next action, based on the existing authorization status. In the case of `notDetermined`, I mapped this to the first time the user opened the app and called the `locationManager.requestWhenInUseAuthorization()` method to ask the user for in-use permission. Apple's recommended design pattern is to first request `authorizedWhenInUse` permission and then `authorizedAlways` permission.

Continuing through the `switch` statement, if the app has already been authorized for in-use permission, this is when you can request always permission. To handle Apple's workflow, I wrapped the always permission request in a method. If the user is coming directly from the `authorizedWhenInUse` permission request, you should show the always permission request; otherwise, start the workout. I needed to go outside of Core Location and use the `UserDefaults` class to track whether the app has asked for permission before, as checking for `authorizedWhenInUse` or `denied` authorization here would cause too many false positives. The `isConfigured` value is set to `true` when users start their first workout.

Finally, in the case of `authorizedAlways` permission, you can start the workout normally. For `denied`, I show an error alert. For all other statements, I record the unknown statement to the console log for assistance in later debugging.

The `didChangeAuthorizationStatus()` method uses the same logic as the `requestLocationPermission()` method to handle states. If you think of the permission request like a conversation, the delegate method is the listener's reply to the speaker's question.

To make sure everything is working, compile and run the app. When you press the Start button for the first time, you should see the alerts pictured in Figure 2-8.

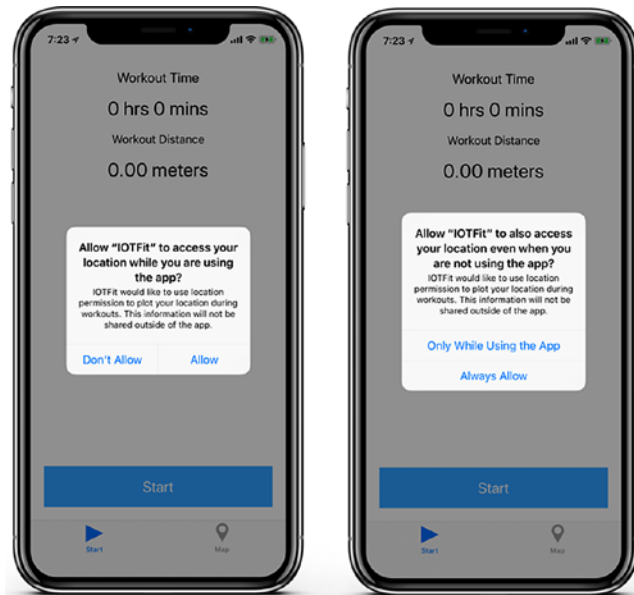


Figure 2-8. Location permission alerts for IOTFit

Asking the User to Change App Settings

The only way to recover from the denied permission is when the user allows it in the page for your app within iOS's Settings app. A common way to help encourage this activity is by creating an alert that sends the user directly to this page. Although iOS has much more limited settings for sharing data between apps when compared to a desktop computer or Android phone, *URL Schemes* (specially formatted URLs) have been an officially supported way of passing tiny bits of data between apps that register for their own scheme (for example, `mySocialApp://`). Data is passed back and forth as URL parameters (for example, `mySocialApp://?request=shareLink`).

The iOS Settings app registers the `prefs://` scheme. Luckily, Apple makes finding the settings-specific URL for you. All you have to do is call the `UIApplicationOpenSettingsURLSetting` macro. In Listing 2-7, I have modified the `presentPermissionErrorAlert()` method to launch the Settings app from this URL.

Listing 2-7. Sending the User to the Settings Page for Your App

```
func presentPermissionErrorAlert() {
    let alert = UIAlertController(title: "Permission Error",
    message: "Please enable location services for this app",
    preferredStyle: UIAlertControllerStyle.alert)
```

```

let okAction = UIAlertAction(title: "OK", style:
        UIAlertActionStyle.default, handler: {
                (action: UIAlertAction) in
                        if let settingsUrl = URL(string:
                                UIApplicationOpenSettingsURLString),
                                UIApplication.shared.canOpenURL(settingsUrl) {
                                        UIApplication.shared.open(settingsUrl,
                                            options: [:], completionHandler: nil)
                                }
                        }
            })
let cancelAction = UIAlertAction(title: "Cancel", style:
        UIAlertActionStyle.cancel, handler: nil)
alert.addAction(okAction)
alert.addAction(cancelAction)
self.present(alert, animated: true, completion: nil)
}

```

In the same way you had to check if location services were available before requesting permission, you must ask if the app can open the Settings URL before attempting to open it. One of the most unfortunate side effects of how Apple implements its protected resources is that if you do not ask for permission or availability, your app may crash at runtime. Runtime crashes are the number-one reason for App Store rejection.

If users reject location permission for the app, the next time they press the Start button, they will see the flow in Figure 2-9. Pressing the OK button will take them to the Settings page for IOTFit. When they return to the app, they can use it without a problem. You can test this by disabling location permission in the Settings app and then pressing the Start button within the IOTFit app.

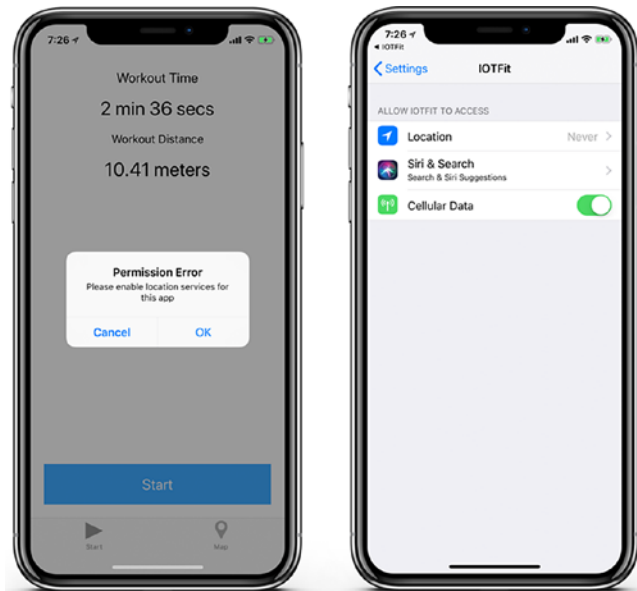


Figure 2-9. Settings request flow for IOTFit

Requesting Location Updates

Now that the app is allowed to interact with the location features of iOS, you can begin polling for location updates. As indicated in the flowchart for the app from Figure 2-7, every few seconds, you must save the most recent location, calculate the new workout time, and calculate the new workout distance. When the user pauses the workout, you must pause the updates. Resuming continues the time and distance updates.

The flow for handling location updates in iOS is very similar to the flow for requesting location permission. First, let your location manager instance know you want location updates, then respond to these updates through the `locationManager:didUpdateLocations:` delegate method. To reduce the number of updates, you will have to set a limit on the accuracy of the location manager as well.

In Listing 2-8, I have updated the `requestLocationPermission()` and `startWorkout()` methods and `CLLocationManagerDelegate` extension to include the location update request and delegate handler method.

Listing 2-8. Requesting Location Permission When the Workout Begins

```

class CreateWorkoutViewController: UIViewController {
    ...
    func requestLocationPermission() {
        if CLLocationManager.locationServicesEnabled(){

            locationManager.desiredAccuracy =
                kCLLocationAccuracyHundredMeters
            locationManager.distanceFilter = 10.0 //meters
            locationManager.delegate = self
                ...
        }
    } else {
        presentEnableLocationAlert()
    }
}

func startWorkout() {
    currentWorkoutState = .active
    ...
    locationManager.startUpdatingLocation()
}
}

extension CreateWorkoutViewController:
    CLLocationManagerDelegate {
    ...

    func locationManager(_ manager: CLLocationManager,
        didUpdateLocations locations: [CLLocation]) {

        guard let mostRecentLocation = locations.last else {
            NSLog("Unable to read most recent location")
            return
        }
    }
}

```

```

        NSLog("Most recent location: \(String(describing:
            mostRecentLocation))")
    }

```

The `locationManager:didUpdateLocations:` delegate method returns multiple locations, but for the sake of calculating distance, the last one is sufficient. In my code, I added a guard-let to validate the value before accessing it. With optional values that come back from hardware, the risk that they were not initialized is high (such as when the hardware is not ready, or a connection is down). It is not wise to force-unwrap in these situations.

Responding to Location Updates

The `locationManager:didUpdateLocations:` delegate method will fire anytime it detects a location change greater than the `distanceFilter` value you specified. However, you will have to update the user interface at these times, to show the new value. Additionally, when the user changes the state of the workout, you should also change the labels in the buttons (and hide the Pause button, if the workout is stopped altogether). Finally, you will also require some way of keeping track of the time that has elapsed.

The easiest of these tasks is updating the buttons. When the workout is started, you will show the Pause button and change the label of the Start button to “Stop.” When the Pause button is pressed, change the label of this button to “Resume.” Finally, when the Stop button is pressed, reset the user interface (Show Start button, hide Pause button). Because this logic is the same for all state changes in the app, you can wrap it in a method and use that method throughout your code. In Listing 2-9, I have created a method called `updateUserInterface()` to handle these updates and added calls to it in the `viewDidLoad()`, `toggleWorkout()`, and `pauseWorkout()` methods. These are the points at which the view is created, and the user interacts with the screen.

Listing 2-9. Updating the Buttons When the App Changes State

```

class CreateWorkoutViewController: UIViewController {
    ...
    override func viewDidLoad() {
        super.viewDidLoad()
        updateUserInterface()
    }
}

```

```

@IBAction func toggleWorkout() {
    switch currentWorkoutState {
    case .inactive:
        requestLocationPermission()
    case .active:
        currentWorkoutState = .inactive
    default:
        NSLog("toggleWorkout() called out of
        context!")
    }
}

updateUserInterface()
}

@IBAction func pauseWorkout() {
    updateUserInterface()
}

func updateUserInterface() {
    switch(currentWorkoutState) {
    case .active:
        toggleWorkoutButton?.setTitle("Stop", for:
            UIControlState.normal)
        pauseWorkoutButton?.setTitle("Pause", for:
            UIControlState.normal)
        pauseWorkoutButton?.isHidden = false
    case .paused:
        pauseWorkoutButton?.setTitle("Resume", for:
            UIControlState.normal)
        pauseWorkoutButton?.isHidden = false
    default:
        toggleWorkoutButton?.setTitle("Start", for:
            UIControlState.normal)
    }
}

```

```

        pauseWorkoutButton?.setTitle("Pause", for:
            UIControlState.normal)
        pauseWorkoutButton?.isHidden = true
    }
}
...
}

```

The next easiest task is to update the time display. Because the location updates fire at unpredictable times (whenever the threshold has changed), you will have to come up with an independent time-tracking mechanism. Luckily, there's an API for that! You can use the `Timer` class to create a timer object, which can call a method at a later time or repeatedly at an interval you specify. For the IOTFit app, every time the timer fires, you must calculate the new total time of the workout and update the workout time label. In Listing 2-10, I have updated the `CreateWorkoutViewController` class with properties and helper methods to assist with managing the timer and inserted the appropriate calls for timer state management.

Listing 2-10. Using a Timer to Track Workout Time Updates

```

import UIKit
import CoreLocation
...
let timerInterval: TimeInterval = 1.0

class CreateWorkoutViewController: UIViewController {
    ...
    var lastSavedTime: Date?
    var workoutDuration: TimeInterval = 0.0
    var workoutTimer: Timer?
    ...

    func startWorkout() {
        currentWorkoutState = .active
        UserDefaults.standard.setValue(true, forKey:
            "isConfigured")
    }
}

```

```

UserDefaults.standard.synchronize()
    workoutDuration = 0.0
    workoutTimer = Timer.scheduledTimer(timeInterval:
        timerInterval, target: self, selector:
        #selector(updateWorkoutData), userInfo: nil,
        repeats: true)
    locationManager.startUpdatingLocation()
}

@objc func updateWorkoutData() {
    let now = Date()
    if let lastTime = lastSavedTime {
        self.workoutDuration +=
            now.timeIntervalSince(lastTime)
    }
    self.workoutDuration += timerInterval
    workoutTimeLabel?.text = stringFromTime(timeInterval:
        self.workoutDuration)
}

func stringFromTime(timeInterval: TimeInterval) -> String{
    let integerDuration = Int(timeInterval)
    let seconds = integerDuration % 60
    let minutes = (integerDuration / 60) % 60
    let hours = (integerDuration / 3600)

    if hours > 0 {
        return String("\(hours) hrs \(minutes) mins
            \(seconds) secs")
    } else {
        return String("\(minutes) min \(seconds) secs")
    }
}

@IBAction func toggleWorkout() {
    switch currentWorkoutState {
    case .inactive:
        requestLocationPermission()
    }
}

```

```

    case .active:
        currentWorkoutState = .inactive
        stopWorkoutTimer()
    default:
        NSLog("toggleWorkout() called out of
            context!")
    }

    updateUserInterface()
}

@IBAction func pauseWorkout() {
    switch currentWorkoutState {
        case .paused:
            startWorkout()
        case .active:
            currentWorkoutState = .paused
            stopWorkoutTimer()
        default:
            NSLog("pauseWorkout() called out of
                context!")
    }
    updateUserInterface()
}
...
func stopWorkoutTimer() {
    workoutTimer?.invalidate()
    lastSavedTime = nil
}
}

```

To track the time elapsed, I have created a `TimeInterval` object. Timer objects are initialized with `TimeInterval` values, so using this same class for tracking time makes the code much more consistent. Next, you initialize the `Timer` and reset the `workoutDuration` inside the `startWorkout()` method. The `Timer` is initialized to repeat every second. When it fires, it calls the `updateWorkoutData()` method. Inside this

method, I use the `stringFromTime()` helper method to convert the `workoutDuration` property to a string and update the `workoutTime` label. I calculated the difference in time by comparing two `Date` objects (`lastSavedTime` and `now`). This allowed me to keep an accurate track of time, even when the app is in the background.

For the final timer-related tasks, you may notice that I inserted the appropriate calls to `stopWorkoutTimer()` inside the `toggleWorkout()` and `pauseWorkout()` methods and fleshed out the `pauseWorkout()` method with the logic it requires to implement the states in the flowchart from Figure 2-7.

With all of this user interface update groundwork now in place, it is straightforward to update the total workout distance. In the same way you created a property to track total time, you can create properties to track workout distance and the last location saved. When the `locationManager:didUpdateLocations:` delegate method fires, you can add the distance between the most recent location and last saved location using the `distance` property of the `CLLocation` object (the class that represents locations as coordinate points). In Listing 2-11, I have updated the `CreateWorkoutViewController` class, `locationManager:didUpdateLocations:` delegate method, and `updateWorkoutData()` method, to implement this logic.

Listing 2-11. Calculating Workout Distance

```
class CreateWorkoutViewController: UIViewController {
    ...
    var workoutDistance: Double = 0.0
    var lastSavedLocation: CLLocation?
    ...
    @objc func updateWorkoutData() {
        self.workoutDuration += timerInterval
        workoutTimeLabel?.text =
            stringFromTime(timeInterval:
                self.workoutDuration)

        workoutDistanceLabel?.text = String(format: "%.2f
            meters", arguments: [workoutDistance])
    }
    ...
}
```



```

extension CreateWorkoutViewController: CLLocationManagerDelegate {
    ...
    func locationManager(_ manager:
        CLLocationManager, didUpdateLocations
        locations: [CLLocation]) {

        guard let mostRecentLocation = locations.last else {
            NSLog("Unable to read most recent location")
            return
        }

        if let savedLocation = lastSavedLocation {
            let distanceDelta = savedLocation.distance(from:
                mostRecentLocation)
            workoutDistance += distanceDelta
        }

        lastSavedLocation = mostRecentLocation
    }
}

```

Programmatically Enabling Background Updates

As mentioned earlier, the `locationManager:didUpdateLocations:` delegate method will fire anytime the location change detected is greater than the minimum threshold you set, including while the app is in the background. For the safest implementation, you must modify the setup code for the Create Workout View Controller's location manager, to declare that the app will be using location in the background and that the updates can be paused. In addition to code clarity, this will help your app become a responsible power citizen. GPS is one of the most power-hungry features on mobile devices! In Listing 2-12, I have modified the `requestLocationPermission()` method with these changes.

Listing 2-12. Enabling Background Updates for the Location Manager

```

func requestLocationPermission() {
    if CLLocationManager.locationServicesEnabled() {
        locationManager.distanceFilter = 10.0
    }
}

```

```

locationManager.pausesLocationUpdatesAutomatically
                                = true
locationManager.allowsBackgroundLocationUpdates
                                = true

locationManager.desiredAccuracy =
    kCLLocationAccuracyHundredMeters

locationManager.delegate = self

...

} else {
    presentEnableLocationAlert()
}
}

```

With these last few changes, you can now use the Create Workout screen to track workouts! Try walking around your home or office for a few meters, you will see the time and distance increase, until you press the Pause or Stop buttons!

Displaying Location Data on a Map

Now that the IOTFit app is able to use location services on the iPhone to help users figure out how far they ran, it would be good to make this data useful to the users. When building an IoT application, you should think of how to use the data to help the user improve his/her life, as much as what it takes to talk to the hardware to collect that data. By allowing your users to visualize their workouts on a map, you can help them think about the effect of a path on their performance, research a path they liked, or give them a fun image to share with their friends.

In the first half of this chapter, you learned how to use the `CLLocation` class to save location coordinate data and calculate the distance between two points. In this section, you will learn how to take that further, by converting this data into map pinpoints and a path. As you will learn, it is actually quite simple. Unfortunately, these operations require a set of data to work on, and Apple does not give you a free store for locations out of the box, so you will have to build one yourself. Luckily, however, this has also become a lot easier with iOS 11, with the help of the Codable protocol.

Using the Codable Protocol for File-Based Data Storage

One of the key features of any workout app is a history feature, which allows users to see their past workouts. In this first iteration of IOTFit, the history feature is a map that shows a path between the user's starting and end points (represented as pinpoints). To enable this feature, you must add a class to save and retrieve workout data and modify the existing classes, to use it as their data source. For the IOTFit app, you will make a new class called `WorkoutDataManager`, to manage these duties and access it through a singleton object.

A singleton is an object that can only be instantiated once within an application. It is a controversial design pattern, as many developers think singletons serve the same purpose as global variables (having side effects and being shared everywhere in an application), but I want to introduce them, as Apple uses them frequently in its hardware APIs, and I feel they are appropriate in situations in which you must manage access to a shared resource within an application.

To let users access data after closing the app, the `WorkoutDataManager` will use a property-list (`.plist`) file as its persistent data store. Property-lists, like the one you used earlier to configure location permissions for the app, are frequently used to store settings and simple data sets in iOS, both as bundled files and documents. In the old days, you would have to write the entire data input/output (I/O) stack yourself, but now, thanks to the Codable protocol, all you have to do is abstract your information into basic data types (for example, `Int`, `String`) that implement the protocol, write adapters to help serialize more complex types (like `CLLocation`) to your basic ones, and then specify the format you want to store your data as (for example, JSON, Property-List).

Caution The code in this section will only work in iOS 11 and later versions. The Codable protocol was introduced in iOS 11, along with Swift 4.

To get started, you must create the `WorkoutDataManager` class. As shown in Figure 2-10, go to the File menu and select `New` ➤ `File`.

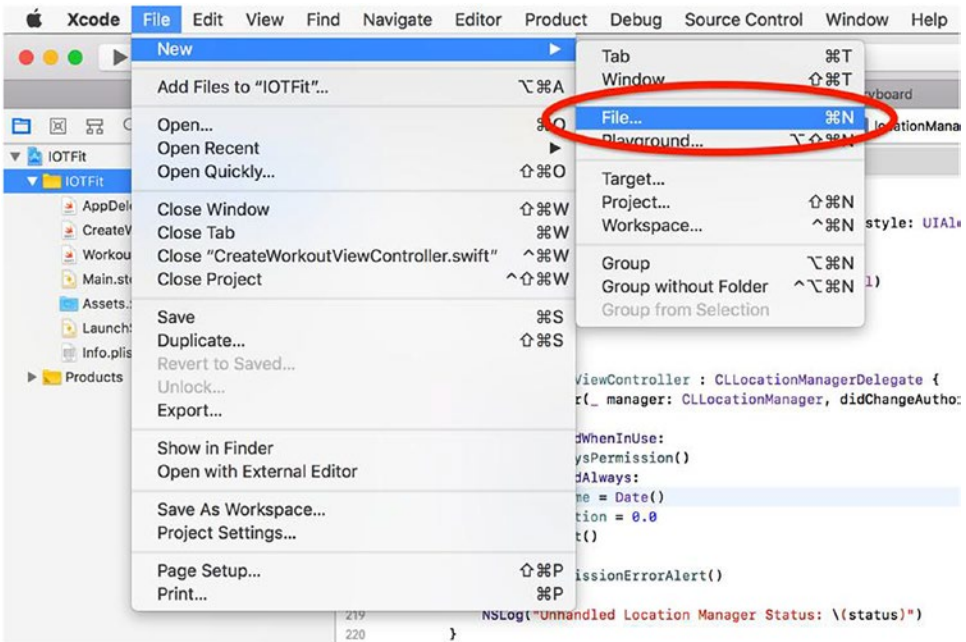


Figure 2-10. Creating a new file in Xcode

When asked to choose a type, choose `Swift File`, as shown in Figure 2-11. Name the new file `WorkoutDataManager.swift`.

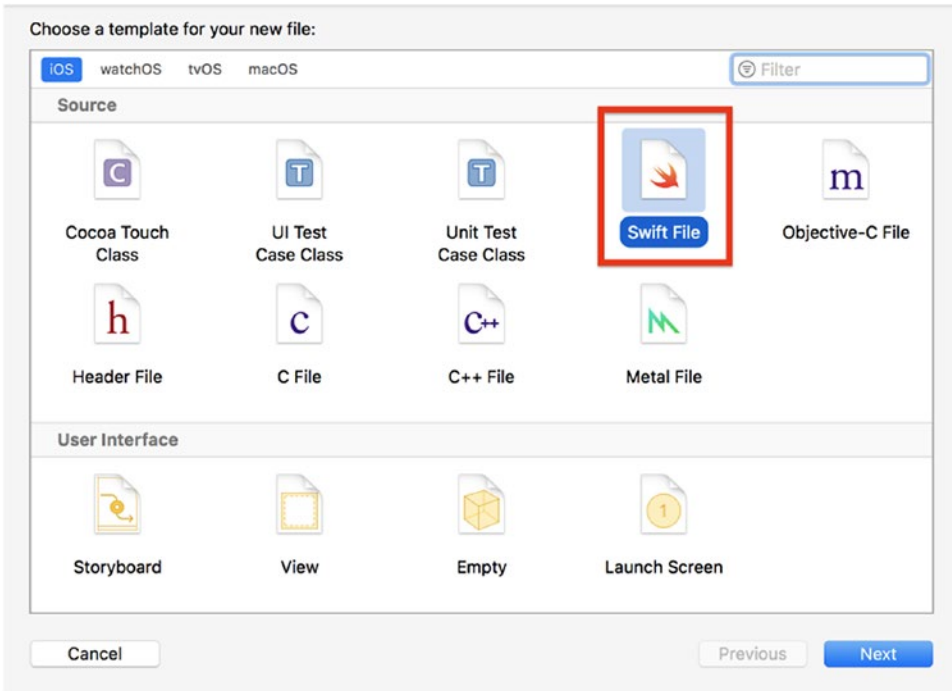


Figure 2-11. Xcode new file templates

Your new class should resemble Listing 2-13.

Listing 2-13. Blank WorkoutDataManager Class

```
import Foundation

class WorkoutDataManager {
}
```

Singletons in Swift are implemented by creating a static instance of the class and a custom initializer to *lazy-load* the object (initialize it the first time it is called). To implement these, modify the blank class, as shown in Listing 2-14.

Listing 2-14. Implementing the WorkoutDataManager Class As a Singleton

```
class WorkoutDataManager {
    static let sharedManager = WorkoutDataManager()

    private init() {
        print("Singleton initialized")
    }
}
```

As mentioned earlier, to take advantage of the data serialization features of the Codable protocol, you must define data types that implement it. In keeping with the name of the class, the core data type for the IOTFit app will be a *workout*. If you think about the Create Workout View Controller, the data you displayed there was the workout duration and distance. To help draw the map, you should also keep a list of locations that were detected during the workout. In Listing 2-15, I added the data types you will use to build the Workout Data Manager.

Listing 2-15. Adding Codable-Compatible Data Types to the WorkoutDataManager Class

```
import Foundation

struct Coordinate: Codable {
    var latitude: Double
    var longitude: Double
}

struct Workout: Codable {
    var endTime: Date
    var duration: TimeInterval
    var locations: [Coordinate]
}

typealias Workouts = [Workout]

class WorkoutDataManager {
    static let sharedManager = WorkoutDataManager()
```

private var workouts: Workouts?

```
private init() {
    print("Singleton initialized")
}
}
```

As mentioned earlier, to use the Codable protocol, you must abstract everything into basic data types. To assist with this, I created the `Coordinate` data type, which you will use later to help convert data from `CLLocation` objects. Because users will avail themselves of the app more than once, I created a typealias called `Workouts`, to represent an array of workout items. One of the great things about the Codable protocol is that when you want to store an entire array of items, all you have to do is *encode* (write) or *decode* (read) that array from your data store.

Implementing File I/O

iOS is famous for keeping apps in sandboxes, meaning that apps can only access and manage resources (such as files) within a space for that app alone, with a few exceptions allowed by Apple. One of the most common ways developers work with this sandbox is by creating files in the Documents folder of the app bundle at runtime. For the IOTFit app, you will do the same. Workouts will be stored in a property-list file named `Workouts.plist`.

One of the challenges of working with the Documents folder of your app is that you must find its file path at runtime. Every time your app is installed, it will be created in a unique folder generated by iOS. To help manage this, I have added logic in Listing 2-16 that looks up the path for IOTFit's Documents folder at runtime and appends the `Workouts.plist` file name to the end.

Listing 2-16. Finding a File Path in an App's Documents Directory at Runtime

```
class WorkoutDataManager {
    ...
    private var workoutsFileUrl: URL? {
        guard let documentsUrl = documentsDirectoryUrl() else {
            return nil
        }
    }
}
```

```

    return documentsUrl.appendingPathComponent(
        "Workouts.plist")
}

func documentsDirectoryUrl() -> URL? {
    let fileManager = FileManager.default
    return fileManager.urls(for: .documentDirectory, in:
        .userDomainMask).first
}
}

```

The `FileManager` class is primarily used to do manual file I/O; however, it comes with convenient helper methods to help manage your program's sandbox. You may have also noticed that I used the guard-let pattern here. For times when you must stop execution of a block after a condition fails, guard-let is a perfect choice. It also helps increase readability, as subsequent logic is nested in fewer chained if-let statements.

To load the data, you must create a Decoder. This is an object that uses one of Apple's pre-built file interfaces (or one you write yourself) and your Codable-compatible data type to map the data in the file to something you can use in your app. In Listing 2-17, I have added the `loadFromPlist()` method to handle this operation, and a call to the method in the custom initializer. To keep data in sync, the first operation you should do after initializing the `WorkoutDataManager` class is load past data.

Listing 2-17. Loading Data from a Property-List Using `PropertyListDecoder`

```

class WorkoutDataManager {
    ...

    private init() {
        print("Singleton initialized")
        loadFromPlist()
    }

    ...
}

```



```

func loadFromPlist() {
    workouts = [Workout]()

    guard let fileUrl = workoutsFileUrl else {
        return
    }

    do {
        let workoutData = try Data(contentsOf: fileUrl)
        let decoder = PropertyListDecoder()
        workouts = try decoder.decode(Workouts.self, from:
            workoutData)
    } catch {
        NSLog("Error reading plist")
    }
}
}

```

Writing to a file is a very similar process, except rather than using a decoder, you use an encoder to read the contents of a file. To handle saving to files, add the `saveToList()` method to the `WorkoutDataManager` class, as shown in Listing 2-18.

Listing 2-18. Writing to a Property-List Using `PropertyListEncoder`

```

class WorkoutDataManager {
    ...

    func saveToPlist() {
        guard let fileUrl = workoutsFileUrl else {
            return
        }

        let encoder = PropertyListEncoder()
        encoder.outputFormat = .xml
        do {
            let workoutData = try encoder.encode(workouts)
            try workoutData.write(to: fileUrl)
        }
    }
}

```

```

    } catch {
        NSLog("Error writing plist")
    }
}
}

```

Using the map() Method to Serialize Data

The final tasks you must perform to complete the Workout Data Manager are the interfaces to Create Workout View Controller and Workout Map View Controller. These include the operations required to create a workout, close a workout, append locations to a workout, and retrieve the most recent workout. These View Controllers receive their data from the Core Location framework as `CLLocation` objects, so you will also have to add some logic to convert from `CLLocation` to the `Coordinate` data type you created in this section. Luckily, this is where the `map()` higher-order function (method) that was introduced in Swift 3.0 comes in

As with prior examples, before thinking about the details of implementation, think about how the user interface will need to connect to the workout life-cycle logic. When you start a new workout, you will have to create a new workout. Every time Core Location detects a new location, you will have to append it to the current workout. When you finish a workout, you should close the current workout and save it. Finally, when the map appears, you should load the last saved workout. In Listing 2-19, I have modified the `CreateWorkoutViewController` class to include these calls, based on the information each method has when it is called.

Listing 2-19. Interacting with the `WorkoutDataManager` Class from the `Create Workout View Controller`

```

class CreateWorkoutViewController: UIViewController {
    ...
    func startWorkout() {
        ...
        locationManager.startUpdatingLocation()
        lastSavedTime = Date()
        WorkoutDataManager.sharedManager.createNewWorkout()
    }
    ...
}

```

```

@IBAction func toggleWorkout() {
    switch currentWorkoutState {
    case .inactive:
        requestLocationPermission()
    case .active:
        currentWorkoutState = .inactive
        stopWorkoutTimer()
        WorkoutDataManager.sharedManager.saveWorkout(
            duration: workoutDuration)
    default:
        NSLog("toggleWorkout() called out of
            context!")
    }
    updateUserInterface()
}
...
}

extension CreateWorkoutViewController: CLLocationManagerDelegate {
    ...

    func locationManager(_ manager:CLLocationManager,
        didUpdateLocations locations: [CLLocation]) {

        guard let mostRecentLocation = locations.last
        else{
            return
        }

        ...

        WorkoutDataManager.sharedManager.addLocation(coordinate:
            mostRecentLocation.coordinate)
    }
}

```

To manage the coordinate data for an active workout, add a property that holds an array of `CLLocationCoordinate2D` objects to the `WorkoutDataManager` class. When a workout is reset, clear the array. When a location update is posted, append the latest

location to the array. In Listing 2-20, I have updated the `WorkoutDataManager` class to include this logic. The `CLLocationCoordinate2D` class is a good way to get coordinate information from a `CLLocation` object without requiring too much extra code.

Listing 2-20. Adding Active Workout Location Management to the `WorkoutDataManager` Class

```
import Foundation
import CoreLocation
...
class WorkoutDataManager {
    static let sharedManager = WorkoutDataManager()
    private var workouts: Workouts?
    private var activeLocations: [CLLocationCoordinate2D]?
    ...
    func createNewWorkout() {
        activeLocations = [CLLocationCoordinate2D]()
    }
    func addLocation(coordinate: CLLocationCoordinate2D) {
        activeLocations?.append(coordinate)
    }
    ...
}
```

Creating and updating workout locations turned out to be straightforward; however, for the save and retrieve operations, you will have to convert these locations to/from `Coordinate` objects. Rather than creating a `for`-loop to iterate through all these items, starting with Swift 3.0, you can use the `map()` higher-order function to define an operation to apply or *map* onto each item in a collection (for example, add 3 to every item in an array). For the Workout Data Manager, you can use `map()` to convert each item to/from the `Coordinate` data type. In Listing 2-21, I have added the `saveWorkout()` and `getLastWorkout()` methods to the `WorkoutDataManager` class, which implement this logic.

Listing 2-21. Using the `map()` Method to Serialize Data in the `saveWorkout()` and `getLastWorkout()` Methods

```
class WorkoutDataManager {
    ...
    func saveWorkout(duration: TimeInterval) {
        guard let activeLocations = activeLocations else {
            return
        }

        let mappedCoordinates = activeLocations.map{(value:
            CLLocationCoordinate2D) in
            return Coordinate(latitude: value.latitude,
                longitude: value.longitude)
        }

        let currentWorkout = Workout(endTime: Date(), duration:
            duration, locations: mappedCoordinates)

        workouts?.append(currentWorkout)

        saveToPlist()
    }

    func getLastWorkout() -> [CLLocationCoordinate2D]? {
        guard let workouts = workouts, let lastWorkout =
            workouts.last else {
            return nil
        }

        let locations = lastWorkout.locations.map{(value:
            Coordinate) in
            return CLLocationCoordinate2D(latitude:
                value.latitude, longitude: value.longitude)
        }

        return locations
    }
}
```

Just as with a callback handler or block, the logic in `map()` that must occur on each item is defined as an anonymous function. To help make it easier to read, I gave a label to the iterated value. This completes the Workout Data Manager; now you can use it to display saved locations on a map!

Displaying Saved Locations on a Map

Of all the capabilities you will learn in this chapter, mapping the data is the easiest. To map the data, you must retrieve the data from the Workout Location Manager, add pins showing the starting and ending points of the workout, and draw a path between them. For the data retrieval operation, you will use the `getLastWorkout:` method from the previous exercise. To draw the pinpoints and path, you can use MapKit's built-in annotation (`MKPointAnnotation`) and polyline (`MKPolylineRenderer`) APIs.

Creating a pinpoint (*annotation* in Apple's terminology) is straightforward. All you have to do is initialize an `MKPointAnnotation` object with a `CLLocationCoordinate2D` object and assign a title to the annotation. Displaying the annotations is even easier. Just call the `showAnnotations:` method on the Map View. It will automatically zoom in to fit these annotations on your map. In Listing 2-22, I have added the logic to generate these annotations in the `viewWillAppear:` method of the `WorkoutMapViewController` class. This is called whenever the Map tab is displayed.

Listing 2-22. Generating and Displaying Annotations on a Map View

```
import MapKit

class WorkoutMapViewController: UIViewController {
    @IBOutlet weak var mapView: MKMapView?

    ...
    override func viewWillAppear(_ animated: Bool) {
        super.viewWillAppear(animated)

        guard var locations =
            WorkoutDataManager.sharedManager.getLastWorkout(),
            let first = locations.first,
```

```

    let last = locations.last else {
        return
    }
    let startPin = workoutAnnotation(title: "Start",
        coordinate: first)
    let finishPin = workoutAnnotation(title: "Finish",
        coordinate: last)

    if let oldAnnotations = mapView?.annotations {
        mapView?.removeAnnotations(oldAnnotations)
    }

    mapView?.showAnnotations([startPin, finishPin],
        animated: true)
}

func workoutAnnotation(title: String, coordinate:
    CLLocationCoordinate2D) -> MKPointAnnotation {
    let annotation = MKPointAnnotation()
    annotation.coordinate = coordinate
    annotation.title = title
    return annotation
}
}

```

An important point worth mentioning about Listing 2-22 is that you must call the `removeAnnotations:` method to clear old annotations before displaying new ones. This is a shortcoming in MapKit's implementation, but it's not too hard to overcome.

Drawing a path is also straightforward but requires you to declare the `WorkoutMapViewController` as implementing the `MKMapViewDelegate` protocol. To draw the path, you simply create an `MKPolyline` object, based on the complete set of saved locations, and apply it as an overlay on the map. To define the shape of the path, you have to implement the `mapView:rendererFor:overlay:` method from the `MKMapViewDelegate` protocol. However, the details of implementation are quite simple. This method simply specifies the color, size, and other display properties of the line. In Listing 2-23, I have updated the `WorkoutMapViewController` class to include the protocol implementation and call for drawing the polyline. As with the `CreateWorkoutViewController` class, I implemented the protocol through an extension, to improve readability.

Listing 2-23. Generating and Displaying a Path on a Map View

```

class WorkoutMapViewController: UIViewController {
    @IBOutlet weak var mapView: MKMapView?

    override func viewDidLoad() {
        super.viewDidLoad()
        mapView?.delegate = self
    }

    override func viewWillAppear(_ animated: Bool) {
        super.viewWillAppear(animated)

        guard var locations =
            WorkoutDataManager.sharedManager.getLastWorkout(), let first =
            locations.first, let last = locations.last else {
            return
        }
        ...
        let workoutRoute = MKPolyline(coordinates:
            &locations, count: locations.count)
            mapView?.addOverlays([workoutRoute])
    }
    ...
}

extension WorkoutMapViewController: MKMapViewDelegate {
    func mapView(_ mapView: MKMapView, rendererFor overlay:
        MKOverlay) -> MKOverlayRenderer {

        let pathRenderer = MKPolylineRenderer(overlay: overlay)
            pathRenderer.strokeColor = UIColor.red
            pathRenderer.lineWidth = 3
            return pathRenderer
    }
}

```


This completes the map portion of the IOTFit app! After saving your first workout, when you press the Map tab, you will see two pinpoints indicating the start and end points of your workout, as well as a red line between them, similar to the screenshot in Figure 2-12.

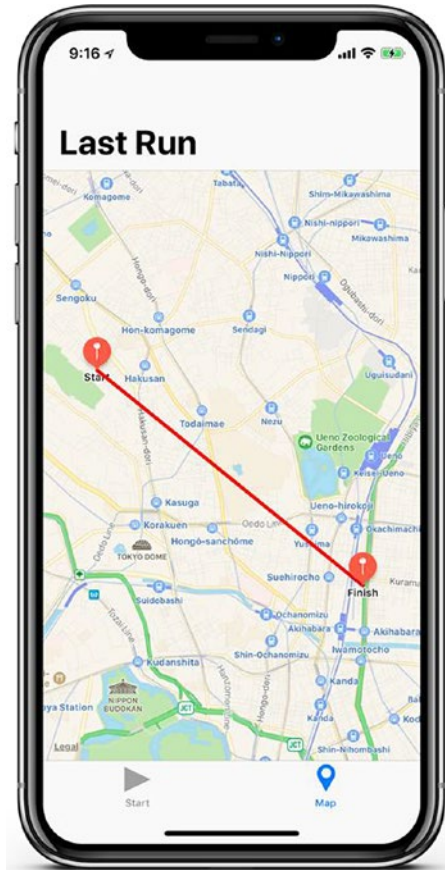


Figure 2-12. IOTFit app with completed workout in Workout Map screen

Summary

In this chapter, you took the user interface skeleton of the IOTFit app from the first chapter and fleshed it out by adding location permissions, workout distance calculation, and the ability to display the locations on a map. Along the way, you learned that when accessing sensitive permissions on a user's device, Apple makes you jump through a lot of hoops in project setup and make the calls to display the request to the user. You also learned how protocols can help you implement callbacks from Apple's hardware frameworks and even define how to draw paths on a map. These concepts will keep coming back as you progress through this book and IoT app development.

CHAPTER 3

Using Core Motion to Add Physical Activity Data to Your Apps

In the last two chapters, you started with a drawing of a workout app and fleshed it out with a functioning user interface that could be used to record workout distance based on GPS readings and display the path on a map. Unfortunately, as you may have noticed by playing with the app, the locations only updated when the user moved and the app was susceptible to lost GPS signals, occasionally producing inaccurate distance data. To make a more accurate and appropriate workout app, in this chapter, you will learn how to use Core Motion, Apple's interface to the M-series motion coprocessor, the chip that provides the pedometer, accelerometer, and gyroscope features on the iPhone and Apple Watch.

With the features provided by the Core Motion framework, you can not only provide users with data they expect from other workout trackers (such as step count) but reduce your app's battery impact, as GPS is one of the most power-hungry features of the iPhone. As iOS devices continued to evolve, many workout apps followed this same transformation by shifting GPS from being the primary sensing method to an opt-in feature for power users.

One last bonus for you as a developer is that you will be able to leverage the knowledge from this chapter again later in the book when you build a watchOS version of the IOTFit app. When Apple was trying to strengthen watchOS during its 2.0 release, it added slimmed-down versions of iOS frameworks to the watch itself. Core Motion and many of the other workout-related frameworks retain the most APIs from iOS, so the porting process should be painless.

Caution Core Motion relies on a discrete microprocessor chip to perform its motion-tracking functions. While a variation of this chip is available on iPhones and Apple Watch devices, as of this writing, it is not available on iPad. You can still develop this project with an iPad, but you will not be able to test it successfully.

Learning Objectives

In this chapter, you will learn how to use Core Motion to extend the IOTFit app to display live-updating reports on step count and activity type, perform step-based distance calculations, and measure changes in elevation. By performing these tasks, you will learn the following key skills for Internet of Things (IoT) app development:

- Setting up an app for motion (activity) permission
- Requesting step data from the iPhone's pedometer
- Measuring difference in altitude during the iPhone's altimeter
- Responding to fast-changing events (step count, activity type, altitude)
- Performing calculations based on motion data

Just as with Core Location, Core Motion provides your app access to hardware on the iPhone that has the potential to collect sensitive data about the user. You will apply the lessons you learned from Chapter 2 (location features) to check for the availability of motion features on the user's device and ask for his/her permission. This chapter acts like the third iteration, or third sprint, of IOTFit. It builds on the foundations in user interface development and permission-based resource development you learned in Chapters 1 and 2 to introduce a new concept. If you are still unsure of either topic, please review those chapters before continuing with this one.

As with the previous chapters, the project in this chapter is meant to be built as you progress through the narrative. If you get stuck on anything or need some reference, the completed code for this project is available on the GitHub repository for this book, under the Chapter 3 folder (<https://github.com/Apress/program-internet-of-things-w-swift-for-ios>).

Requesting Motion Permission from the User

When you built the location-based features of the IOTFit app in Chapters 1 and 2, the changes consisted of three major components:

- Modifying the project file to declare it would use a sensitive permission (and why)
- Checking for the availability of a hardware resource before attempting to use it
- Displaying a pop-up asking the user to allow the app to use the sensitive permission

These concepts all apply to Core Motion as well; however, unlike Core Location, you have less control over how to present the permissions request. With Core Motion, the request is presented the first time an attempt to access a protected resource is made. While this may seem like it is great, because it means less code, the implication is that you will have to move your resource availability check from the beginning of execution to before each operation begins.

The easiest place to start is by declaring your project as requiring motion permission. You enable this operation by adding the `NSMotionUsageDescription` key-value pair to your project's `Info.plist` file. First, make a copy of the IOTFit project from Chapter 2 (either from your own code or the GitHub repository) and open the project. As shown in Figure 3-1, click the project name in the project hierarchy to open the Project Setting editor, then click the Info tab. Scroll down to the vicinity of the location permission key-value pairs (for example, Privacy When In Use Description) and click the plus (+) button to add a new key-value pair. The key for motion permission is Privacy - Motion Usage Description.

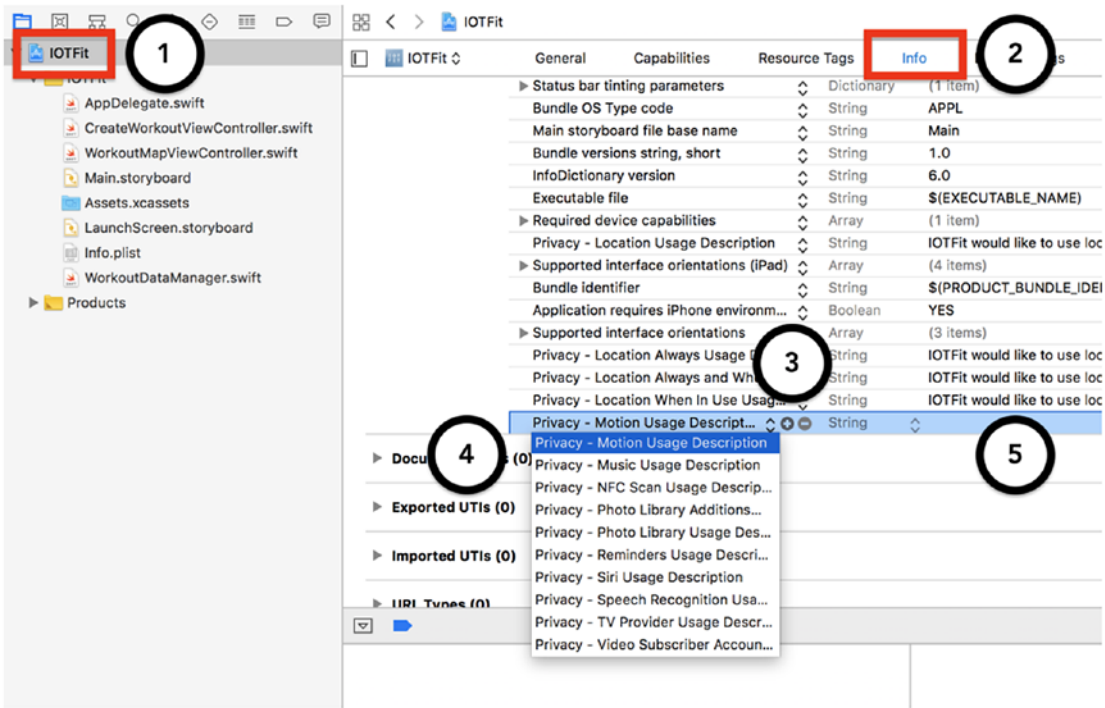


Figure 3-1. Adding the motion permission description to the IOTFit project

Click inside the text field to enter in the description that will appear when the user is prompted for motion permission. For the IOTFit project, I used the following text:

IOTFit would like to use motion permission to help you measure the step count and altitude of your workouts. This information will not be shared outside of the app.

To help your users (and to prevent getting rejected by the App Store submission team), you should always describe why you need to use a restricted resource as accurately as possible. During App Store submissions (initial requests and updates), Apple will run your app and attempt to verify that you are actually using the permissions you requested.

Now that the project is set up for motion permission, you must check if the M-series motion coprocessor is available for use on the user’s device. In a similar manner to Core Location, you can query the `CMMotionManager` class to check for hardware availability. You can perform this operation by querying the `isDeviceMotionAvailable` calculated property. There is more variation in the motion sensing hardware in iOS devices, as compared to GPS hardware, so you will also have to add checks for the features you are trying to use (step counting, altitude). These have similar class methods that you can call

to check availability. In Listing 3-1, I have updated the `CreateWorkoutViewController` class to include the Core Motion framework and check for hardware availability inside the `startWorkout()` method.

Listing 3-1. Adding Motion Permission Checking to the `CreateWorkoutViewController` Class

```
import UIKit
import CoreLocation
import CoreMotion
...
class CreateWorkoutViewController: UIViewController {
    let locationManager = CLLocationManager()
    ...
    var isMotionAvailable: Bool = false
    ...
    func startWorkout() {
        currentWorkoutState = .active
        ...
        if (CMMotionManager().isDeviceMotionAvailable &&
                CMPedometer.isStepCountingAvailable() &&
                CMAltimeter.isRelativeAltitudeAvailable()) {
            //Start motion updates
            isMotionAvailable = true
        } else {
            NSLog("Motion acitivity not available on device.")
            isMotionAvailable = false
        }
    }
}
```

When adopting a permission-based feature, try to design your app to provide value to the user, even when the desired feature is disabled. In the case of motion permission, I did not prevent the user from starting a workout, but I did set the `isMotionAvailable` flag in the `CreateWorkoutViewController` class to `false`, so that I could disable hooks for motion-based features later in the class.

Finally, to make the permission prompt appear, you must attempt to collect data through a CoreMotion API. Because one of the goals of this third iteration of IOTFit is to add step counting, you can trigger the motion permission by initiating a request for pedometer updates. As you may have figured out from Listing 3-1, the class that manages the pedometer on iOS is `CMPPedometer`. To initiate a request for pedometer updates, initialize a `CMPPedometer` object and call the `startUpdates(from:withHandler:)` method on that object, specifying a start date and completion handler. In Listing 3-2, I have updated the `CreateWorkoutViewController` class to include this logic. The response to the request is initiated from the `startWorkout()` method.

Listing 3-2. Adding a Step Count Request to the `CreateWorkoutViewController` Class

```
class CreateWorkoutViewController: UIViewController {
    ...
    var lastSavedTime: Date?
    var workoutStartTime: Date?
    var pedometer: CMPPedometer?
    ...
    func startWorkout() {
        ...
        lastSavedTime = Date()
        workoutStartTime = Date()
        WorkoutDataManager.sharedManager.
            createNewWorkout()

        if(CMMotionManager().isDeviceMotionAvailable
            && CMPPedometer.isStepCountingAvailable()
            && CMAltimeter.isRelativeAltitudeAvailable()){

            isMotionAvailable = true
            startPedometerUpdates()
        }
    }
}
```

```

    } else {
        NSLog("Motion activity not available on
              device.")
        isMotionAvailable = false
    }
}

func startPedometerUpdates() {
    guard let workoutStartTime = workoutStartTime else {
        return
    }
    pedometer = CMPedometer()
    pedometer?.startUpdates(from: workoutStartTime,
        withHandler: { (pedometerData:
            CMPPedometerData?, error: Error?) in
                NSLog("Received pedometer update!")
        })
    }
    ...
}

```

The completion handler is intentionally empty, as the purpose of this exercise is to learn how to present the permission alert. In the next section, you will learn how to work with the data to display step count in your apps.

Now, run the modified application on an iPhone and press the Start button. The first time you press the Start button, you will see a permission alert on your phone, similar to that in the screenshot in [Figure 3-2](#).

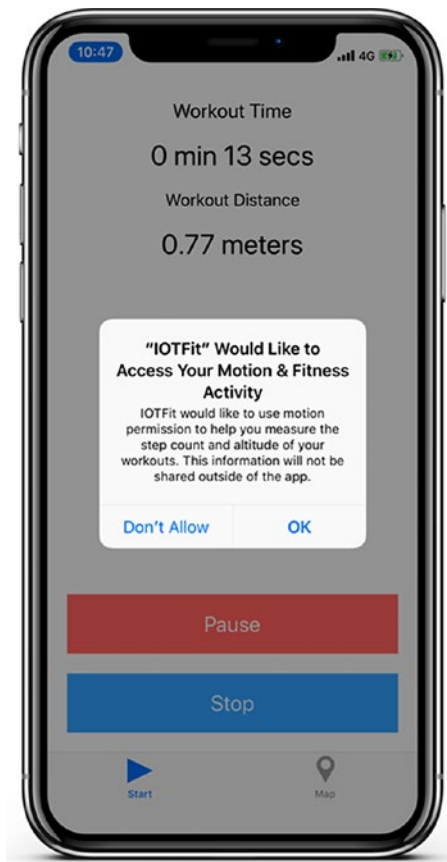


Figure 3-2. Motion permission alert for the IOTFit app

Receiving Real-Time Step Count Updates from the iPhone’s Pedometer

One of the technologies that kicked off the Quantified Self movement in the early 2000s was the introduction of inexpensive standalone pedometers that displayed step counts on an LCD screen. As technology evolved, they began to integrate other statistics, such as floors climbed and smartphone integration (the most famous being the Fitbit). Luckily, iPhones now include very advanced, accurate pedometer-like features built into the M-series motion coprocessor chips, without the need for additional external hardware.

In Listing 3-2, the method I used to initiate the permission request was `startUpdates` (from:withHandler:). This method tells the pedometer to start collecting data and send it back to your app via the completion handler *closure* you specify.

A closure in Swift operates the same way as a block in Objective-C or an anonymous function in other high-level programming languages. Closures are often defined and passed as parameters for other methods. They are an appropriate choice for long-running methods or methods whose response time is unclear (such as updates from hardware), or in place of *protocols*, whose implementations often end up requiring a lot of supporting code.

The closure for pedometer updates returns a `CMPedometerData` object, if the operation was a success (or an uninitialized optional value, in case of failure). By inspecting this value, you can determine the number of steps traveled, floors ascended or descended, distance, and pace of the user between the starting time and the time of the update. In Listing 3-3, I have updated the `CreateWorkoutViewController` class to include new properties for tracking pace and floors climbed. I also updated the `startPedometerUpdates()` method to use the pedometer-based values instead of the GPS values and refactored the logic to reset the workout tracking values into a new `resetWorkoutData()` method.

Listing 3-3. Adding Real-Time Pedometer Updating Handling to the `CreateWorkoutViewController` Class

```
class CreateWorkoutViewController: UIViewController {
    ...
    var workoutDistance: Double = 0.0
    var averagePace: Double = 0.0
    var workoutSteps: Int = 0
    var floorsAscended: Int = 0
    ...
    func resetWorkoutData() {
        lastSavedTime = Date()
        workoutDuration = 0.0
        workoutDistance = 0.0
        workoutSteps = 0
        floorsAscended = 0
        averagePace = 0.0
    }
    ...
}
```

```

func startPedometerUpdates() {
    guard let workoutStartTime = workoutStartTime
        else { return }
    pedometer = CMPedometer()
    pedometer?.startUpdates(from: workoutStartTime,
        withHandler: { [weak self] (pedometerData:
            CMPedometerData?, error: Error?) in
            if let error = error {
                NSLog("Error reading pedometer data:
                    \(error.localizedDescription)")
                return
            }

            guard let pedometerData = pedometerData,
                let distance = pedometerData.distance
                    as? Double,
                let averagePace = pedometerData.averageActivePace
                    as? Double,
                let steps = pedometerData.numberOfSteps
                    as? Int,

                let floorsAscended = pedometerData.floorsAscended
                    as? Int else { return }
                self?.workoutDistance = distance
                self?.workoutSteps = steps
                self?.floorsAscended = floorsAscended
                self?.averagePace = averagePace
        })
    }

func requestLocationPermission() {
    if CLLocationManager.locationServicesEnabled(){
        ...
        switch(CLLocationManager.authorizationStatus()){
            ...
            case .authorizedAlways:
                resetWorkoutData()
        }
    }
}

```

```

        startWorkout()
    default:
        presentPermissionErrorAlert()
    }
} else {
    presentEnableLocationAlert()
}
}
...
}

extension CreateWorkoutViewController: CLLocationManagerDelegate {
    func locationManager(_ manager:
        CLLocationManager, didChangeAuthorization status:
        CLAuthorizationStatus) {
        switch status {
            ...
            case .authorizedAlways:
                resetWorkoutData()
                startWorkout()
            ...
        }
    }
}

func locationManager(_ manager:
    CLLocationManager, didUpdateLocations
    locations: [CLLocation]) {

    guard let mostRecentLocation = locations.last
        else { return }
    //Disable the old location calculation code
    //if let savedLocation = lastSavedLocation {
    //    let distanceDelta = savedLocation.distance(from:
    mostRecentLocation)
    //    workoutDistance += distanceDelta
    //}
}

```

```
    lastSavedLocation = mostRecentLocation
    WorkoutDataManager.sharedManager.addLocation(
        coordinate: mostRecentLocation.coordinate)
}
}
```

One of the challenges of working with optional values in Swift is that you always have to check if the value has been initialized. When working with optional values inside of another optional value, you must perform both of these checks. To get around this issue, many developers have been chaining unwrapping checks in single `if-let` or `guard-let` statements, as I did in my example. I recommend against force-unwrapping using `!` syntax, as it can crash your app at runtime.

Updating the User Interface

Now that the IOTFit app can detect more information about a user's workout, you must modify the user interface to display this information. The way I chose to deal with this was by tightening the spacing in between the labels on the Create Workout View Controller, repurposing the distance label, and adding an extra label to show the average pace, as shown in Figure 3-3.

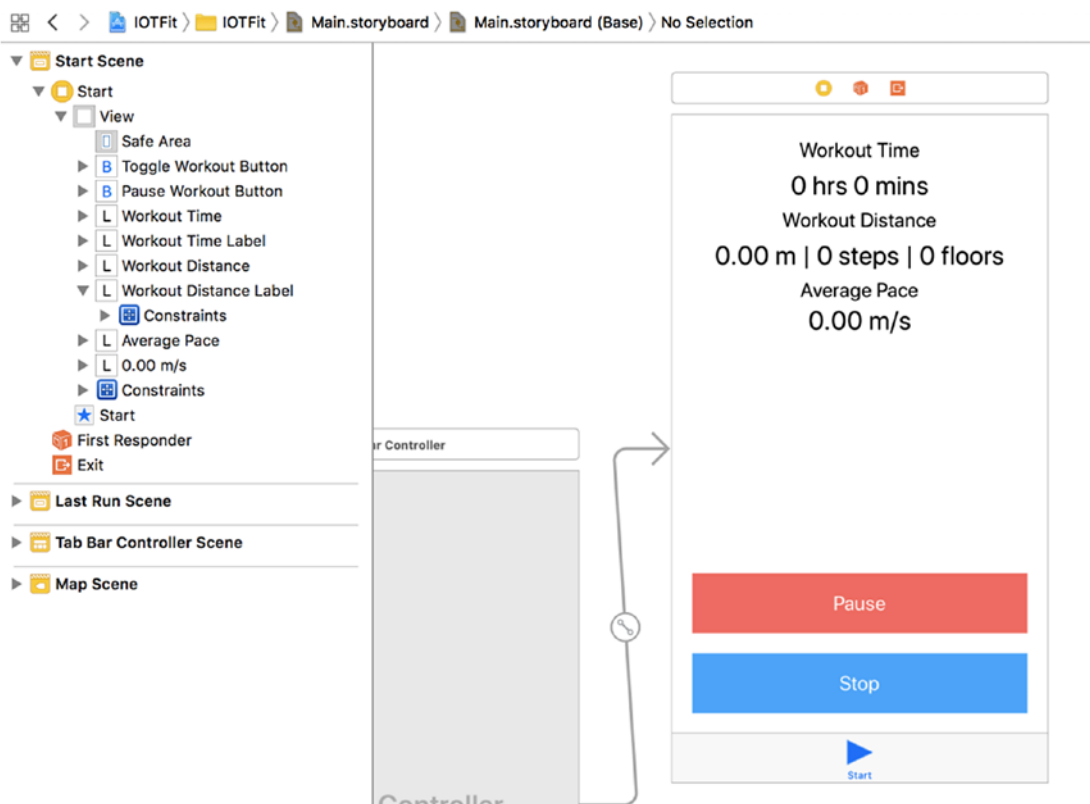


Figure 3-3. Modified user interface for Create Workout View Controller

Update the layout for the Create Workout View Controller in your `Main.storyboard` file by adding two `UILabel` objects and adjusting the Auto Layout constraints to match the values in Table 3-1. If you need a refresher on how to do this, refer back to Chapter 1.

Table 3-1. New Auto Layout Constraints for the Create Workout View Controller

Display Text	Top	Left	Right	Bottom	Height	Text Class
Workout Time	10	20	20	--	30	Title 3
0 hrs 00 mins	0	20	20	--	50	Title 1
Workout Distance	0	20	20	--	30	Title 3
0.00 m 0 steps 0 floors	0	20	20	--	50	Title 1
Workout Pace	0	20	20	--	30	Title 3
0.00 m / s	0	20	20	≥ 20	50	Title 1

In the same manner as Chapter 1, in order to update the Workout Pace label from the code, you have to add a UILabel property to the WorkoutViewController class, as shown in Listing 3-4.

Listing 3-4. Adding the Workout Pace Label to the Create Workout View Controller

```
class CreateWorkoutViewController: UIViewController {
    ...
    @IBOutlet weak var workoutDistanceLabel: UILabel?
    @IBOutlet weak var workoutPaceLabel: UILabel?
    ...
}
```

Next, connect the label to the class, by switching back to the Main.storyboard file, clicking the Connection Inspector for the Create View Controller, holding down the radio button next to workoutPaceLabel and releasing it over the Pace label, as shown in Figure 3-4. Once again, if you run into trouble during this operation, refer to Chapter 1, to quickly review the process.

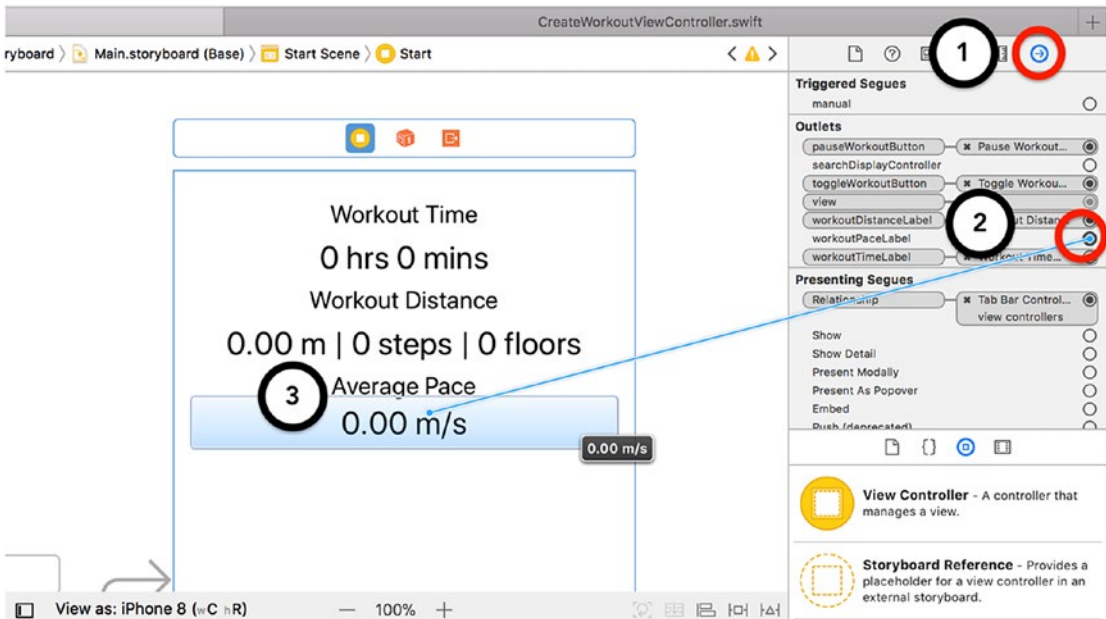


Figure 3-4. Connecting the Workout Pace label to the Create Workout View Controller

Finally, to update the values in the labels, modify the `updateWorkoutData()` method, as shown in Listing 3-5. Read the values from the `averagePace`, `workoutDistance` and `workoutSteps` properties and include them in the text for the labels. You do not have to worry about adding extra events for the data updates. The timer event for updating the time (once per second) should provide sufficient accuracy for most users.

Listing 3-5. Adding Workout Data Updates to the `updateWorkoutData()` Method

```
@objc func updateWorkoutData() {
    let now = Date()
    if let lastTime = lastSavedTime {
        self.workoutDuration +=
            now.timeIntervalSince(lastTime)
    }
    workoutTimeLabel?.text =
        stringFromTime(timeInterval:
            self.workoutDuration)

    workoutDistanceLabel?.text = String(format: "%.2fm | %d steps
        | %d floors", arguments: [workoutDistance, workoutSteps,
        floorsAscended])

    workoutPaceLabel?.text = String(format: "%.2f m /s",
        arguments: [averagePace])

    lastSavedTime = now
}
```

Stopping and Pausing Pedometer Updates

For the final pedometer-related tasks, you must be able to stop and pause pedometer updates. To stop pedometer updates, simply call the `stopUpdates()` method on your `CMPedometer` object as part of the `toggleWorkout()` method, as shown in Listing 3-6.

Listing 3-6. Stopping Pedometer Updates

```

@IBAction func toggleWorkout() {
    switch currentWorkoutState {
    case .inactive:
        requestLocationPermission()
    case .active:
        currentWorkoutState = .inactive
        stopWorkoutTimer()
        pedometer?.stopUpdates()
        WorkoutDataManager.sharedManager.
            saveWorkout(duration: workoutDuration)
    default:
        NSLog("toggleWorkout() called out of
            context!")
    }
    updateUserInterface()
}

```

There is no pause method available for the pedometer; however, you can accomplish the same task through careful management of which variable you use to manage the time updates. The original implementation of the `startPedometerUpdates()` method used the `workoutStartTime` property as the baseline for all pedometer updates. All updates would generate data based on the period between the original start time and the last update. You would not be able to implement accurate pausing with this behavior. However, if you were to use `lastSavedTime`, the property you created to manage the time updates for the workout duration label, you could get a value that queried pedometer updates for a starting date that would be updated when the user pressed the Pause and Resume buttons. In Listing 3-7, I have updated the `startPedometerUpdates()` method to use the `lastSavedTime` property and incremental data updates in its calculation of workout distance.

Listing 3-7. Incorporating Pausing into Pedometer Distance Calculations

```

func startPedometerUpdates() {
    guard let lastSavedTime = lastSavedTime
    else { return }
}

```

```

pedometer = CMPedometer()
pedometer?.startUpdates(from: lastSavedTime, withHandler: {
    [weak self] (pedometerData: CMPedometerData?, error: Error?) in
        NSLog("Received pedometer update!")
        if let error = error {
            NSLog("Error reading data:
                \(error.localizedDescription)")
            return
        }
        guard let pedometerData = pedometerData,
            let distance = pedometerData.distance
                as? Double,
            let averagePace =
                pedometerData.averageActivePace
                as? Double,
            let steps = pedometerData.numberOfSteps
                as? Int,
            let floorsAscended =
                pedometerData.floorsAscended as? Int else {
            return
        }
        self?.workoutDistance += distance
        self?.floorsAscended += floorsAscended
        self?.workoutSteps += steps
        self?.averagePace = averagePace
    })
}

```

Getting Activity Type

As you learned previously, Core Motion provides you with a very accurate, easy-to-use pedometer. However, it can provide more than just raw data alone. When Apple announced the Apple Watch, one of the slides it was proud of was the public reveal of their fitness lab, where they employ a large staff of engineers and researchers in the hope of finding better ways of measuring and utilizing the sensor data generated by

the M-series chips and Apple Watch. One of the fruits of this labor was the ability to determine the type of activity the user is participating in (for example, running, walking, or bicycling). In a process similar to receiving pedometer updates, you can use an instance of `CMMotionActivityManager` in your class to listen to asynchronous updates for activity type.

The method you use to listen to motion updates from a `CMMotionActivityManager` object is `startActivityUpdates(to:withHandler:)`. It takes two parameters: an `Operation Queue` and a completion handler that returns information on the activity type. In iOS programming, the concept of an `Operation Queue` is similar to a *thread* in other higher-level programming languages. It is a way of making a set of instructions (tasks or *Operations*) run in its own path of execution, with the hope of preventing others from being able to execute. These are frequently adopted for long-running tasks, such as heavy Core Data database operations. A developer will make a separate `OperationQueue` object for Core Data tasks and use a protocol or completion handler to tell another part of the program that the task has completed running.

The `startActivityUpdates(to:withHandler:)` method requires you to specify to which `OperationQueue` the motion updates should be delivered. For the IOTFit app, you will primarily use motion data to update the user interface, so you should deliver the updates to the main `OperationQueue` for the application. In Listing 3-8, I have updated the `CreateWorkoutViewController` class to include a `CMMotionActivityManager` property and added a `startActivityUpdates()` method to monitor for activity changes when the workout begins.

Listing 3-8. Initiating Activity-Type Updates

```
class CreateWorkoutViewController: UIViewController {
    ...
    var pedometer: CMPedometer?
    var motionManager: CMMotionActivityManager?
    ...
    func startWorkout() {
        ...
        if (CMMotionManager().isDeviceMotionAvailable
            && CMPedometer.isStepCountingAvailable()
            && CMAltimeter.isRelativeAltitudeAvailable()){
            isMotionAvailable = true
        }
    }
}
```

```

        startPedometerUpdates()
        startActivityUpdates()
    } else {
        isMotionAvailable = false
    }
}
...
func startActivityUpdates() {
    motionManager = CMMotionActivityManager()
    motionManager?.startActivityUpdates(to:
        OperationQueue.main, withHandler: { (activity:
            CMMotionActivity?) in
            //received motion update
        })
    }
}

```

Note If you are having issues with your iOS applications not displaying items to the user interface correctly, verify that your calls are occurring on the main thread (main `OperationQueue`). iOS only executes user interface updates from the main thread.

The `CMMotionActivity` response object contains a series of `Bool` properties, indicating Core Motion's estimation of the current activity type and a confidence property indicating the accuracy of the estimation (low, medium, high). For the IOTFit app, all you must do is display the activity type on the Create Workout View Controller, so you only have to evaluate the responses in the response handler and save the best value. In Listing 3-9, I have updated the `CreateWorkoutViewController` class to include a `currentActivity` property and save this value as part of the motion completion handler in the `startActivityUpdates()` method.

Listing 3-9. Saving the Updated Activity Type

```

import UIKit
...
struct WorkoutType {
    static let automotive = "Driving"
    static let running = "Running"
    static let bicycling = "Bicycling"
    static let stationary = "Stationary"
    static let walking = "Walking"
    static let unknown = "Unknown"
}

class CreateWorkoutViewController: UIViewController {
    ...
    var currentWorkoutState = WorkoutState.inactive
    var currentWorkoutType = WorkoutType.unknown
    ...
    func startActivityUpdates() {
        motionManager = CMMotionActivityManager()
        motionManager?.startActivityUpdates(to:
            OperationQueue.main, withHandler: { [weak self]
                (activity: CMMotionActivity?) in
                guard let activity = activity else { return }
                if activity.walking {
                    self?.currentWorkoutType = WorkoutType.walking
                } else if activity.running {
                    self?.currentWorkoutType = WorkoutType.running
                } else if activity.cycling {
                    self?.currentWorkoutType =
                        WorkoutType.bicycling
                } else if activity.stationary {
                    self?.currentWorkoutType =
                        WorkoutType.stationary
                }
            }
        )
    }
}

```

```

        } else {
            self?.currentWorkoutType = WorkoutType.unknown
        }
    })
}
...
}

```

One of the unfortunate aspects of the design of the `CMMotionActivity` class is that it requires you to define what you want to be the priority of the activities, rather than using an enumerated value to represent the data. I prefaced the completion handler with `[weak self]`. When working with closures, accessing `self` with a strong reference has the potential of retaining the object, causing memory leaks. Using `weak` prevents this.

To display the value on the user interface, simply update the `updateWorkoutData()` method to include inspecting the `currentActivity` property, as shown in Listing 3-10.

Listing 3-10. Displaying Activity Type

```

@objc func updateWorkoutData() {
    let now = Date()

    var workoutPaceText = String(format: "%.2f m/s", arguments:
        [averagePace])

    if let lastTime = lastSavedTime {
        self.workoutDuration +=
            now.timeIntervalSince(lastTime)
    }

    if currentWorkoutType != WorkoutType.unknown {
        workoutPaceText.append(" | \(currentWorkoutType)")
    }
    ...
    workoutPaceLabel?.text = workoutPaceText
    ...
}

```

In this example, I modified the pace label to include the activity type. If the activity type is unknown, I do not display the value. It would serve to confuse users more than it would help them.

To stop motion (activity type) updates, simply update the `toggleWorkout()` method to tell the `CMMotionActivityManager` to stop listening for motion updates, in the same manner you used to stop pedometer updates, as shown in Listing 3-11.

Listing 3-11. Stopping Motion Updates

```
@IBAction func toggleWorkout() {
    switch currentWorkoutState {
    ...
    case .active:
        currentWorkoutState = .inactive
        stopWorkoutTimer()
        pedometer?.stopUpdates()
        motionManager?.stopActivityUpdates()
        ...
    default:
        ...
    }
    updateUserInterface()
}
```

Handling Altimeter Updates

For your final experiment with Core Motion, you can also retrieve the user’s altitude. The best part is that it follows the design of the other Core Motion sensor APIs, namely to

- Instantiate a manager object
- Define a completion block to save the most recent data
- Update the user interface
- Stop updates when complete

As you should now be familiar with this design pattern, I have included the code for adding altimeter data to the Create Workout View Controller in Listing 3-12.

Listing 3-12. Adding Altitude Tracking to the Create Workout View Controller

```

class CreateWorkoutViewController: UIViewController {
    ...
    var workoutAltitude: Double = 0.0
    var workoutDistance: Double = 0.0
    ...
    var motionManager: CMMotionActivityManager?
    var altimeter: CMAltimeter?
    ...
    @IBAction func toggleWorkout() {
        ...
        switch currentWorkoutState {
            ...
            case .active:
                ...
                motionManager?.stopActivityUpdates()
                altimeter?.stopRelativeAltitudeUpdates()
            default:
                ...
        }
        updateUserInterface()
    }
    ...
    @objc func updateWorkoutData() {
        let now = Date()

        var workoutPaceText = String(format: "%.2f m/s |
            %0.2fm ", arguments: [averagePace, workoutAltitude])
        ...
    }
    ...
    func startWorkout() {
        currentWorkoutState = .active
        ...
        if (CMMotionManager().isDeviceMotionAvailable
            && CMPedometer.isStepCountingAvailable()) &&

```



```

    CMAltimeter.isRelativeAltitudeAvailable()) {
        ...
        startActivityUpdates()
        startAltimeterUpdates()
    } else {
        isMotionAvailable = false
    }
}
...
func startAltimeterUpdates() {
    altimeter = CMAltimeter()
    altimeter?.startRelativeAltitudeUpdates(to:
        OperationQueue.main, withHandler: { [weak self]
            (altitudeData: CMAltitudeData?, error: Error?) in
            if let error = error {
                NSLog("Error reading altimeter data:
                    \(error.localizedDescription)")
                return
            }
            guard let altitudeData = altitudeData,
                let relativeAltitude =
                    altitudeData.relativeAltitude as? Double
                    else { return }
            self?.workoutAltitude += relativeAltitude
        }))
}
...
func resetWorkoutData() {
    ...
    workoutDistance = 0.0
    workoutAltitude = 0.0
    currentWorkoutType = WorkoutType.unknown
}
}

```

Now, when you run the IOTFit app on your iPhone and walk around a little (or shake your phone), you should see faster, more accurate activity updates, similar to those in the screenshot in Figure 3-5.



Figure 3-5. Screenshot of IOTFit app with additional workout data

Summary

In this chapter, you took the GPS-powered IOTFit app and turned it into a more accurate, feature-rich, and battery-friendly version by using the Core Motion framework to access the M-series motion co-processor on iPhone and Apple Watch. You learned how to set up manager objects for several of the sensors and how to respond to their asynchronous updates using completion handlers. After a few examples,

the process of setting up a manager, responding to updates, and stopping requests became very obvious as a widely-used pattern, making altimeter integration extremely straightforward.

For the sake of readability, and to keep up with current programming trends, Apple is moving toward more and more completion handler-based workflows in its frameworks. Completion handlers are extremely convenient in situations in which you must execute a quick action after another long-running method completes, and I am sure you will reuse this knowledge many times again in this book and your own projects.

CHAPTER 4

Using HealthKit to Securely Retrieve and Store Health Data

So far, you have learned how to use the motion and GPS sensors on iOS devices to build a workout app called IOTFit, with the capabilities to track activity data (steps, altitude, step count) and display the user's workout path on a map. Additionally, you've learned how to store this data to a property-list (.plist) file inside the app's Documents folder.

While it is useful to store data within one app, you can help the user even more by saving the data in a way that is accessible to other workout apps. For this purpose, you can use the HealthKit framework. HealthKit allows you to abstract workouts into HKWorkout objects and store them in the iPhone's HealthKit store. The HealthKit store is an encrypted area in memory in which the iOS Health app and HealthKit-authorized third-party apps can share information about a user. For example, by enabling HealthKit, users can see workouts they created in IOTFit as discrete items on the iOS Health app. They can also use HealthKit to read display workouts from other apps within IOTFit.

Aside from security, one of HealthKit's most powerful features is the breadth of data it can store via its HKSample class. For the sake of clarity, this chapter will focus on how to apply it to workout-related data, but you can make apps to manage everything from heart rate to UV exposure to vitamin C consumption.

Learning Objectives

In this chapter, by integrating HealthKit into IOTFit, you will learn the following key concepts of Internet of Things (IoT) app development:

- Requesting permission to the iOS *HealthKit Store*
- How to save data to HealthKit
- How to load data from HealthKit
- How to display data on a Table View Controller

These changes will be expressed by expanding the previously developed *WorkoutDataManager* class and adding a History tab to the app, which will contain a table view documenting the user’s workout history, as shown in Figure 4-1.

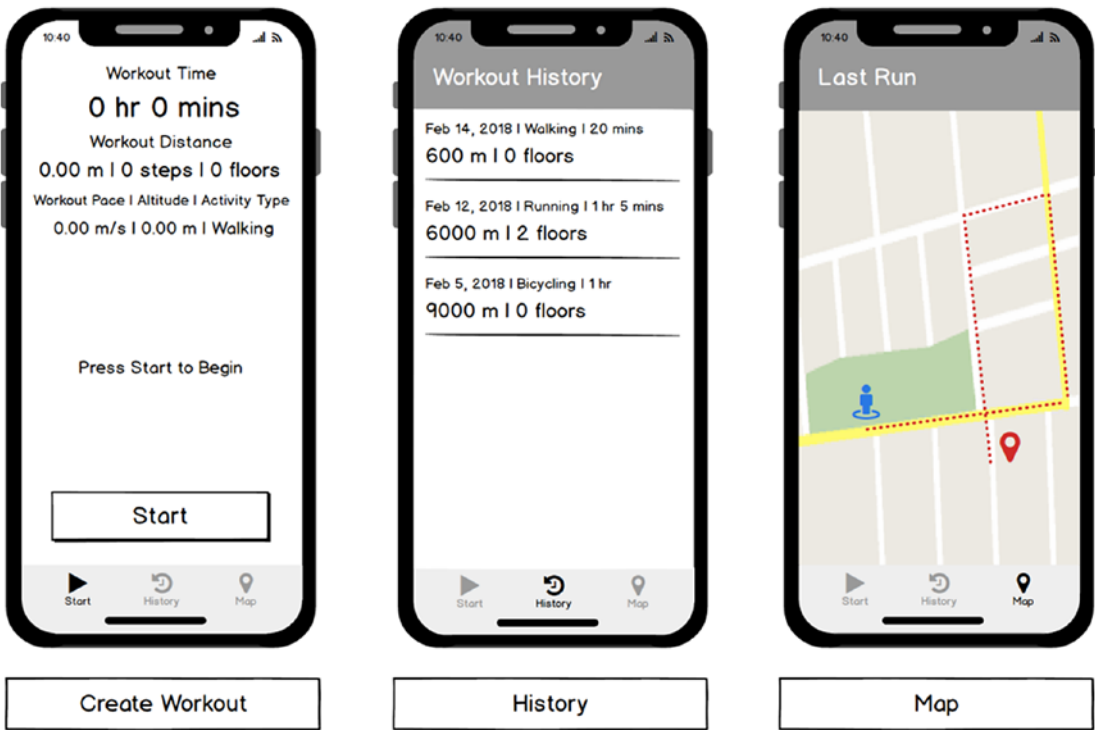


Figure 4-1. Modified wireframe for the IOTFit app, including the new History tab

As in the previous chapters, this project builds on the progress you have made through Chapter 3. If you get stuck or have to reference the completed code for this project, it is available on the GitHub repository for this book, under the Chapter 4 folder (<https://github.com/Apress/program-internet-of-things-w-swift-for-ios>).

Requesting HealthKit Permission

Given the sensitive nature of health data, it should come as no surprise that using HealthKit requires you to modify your app to declare that it wants to use health-related features and presents a permission alert when users want to access these features. Similar to Core Motion, HealthKit is only available on newer iPhones, iPod touches, and the Apple Watch, requiring you also to query for the availability of these features. Luckily, you can use the same workflow you learned for requesting permissions in Core Motion and Core Location to add HealthKit to your app.

Note Similar to the Core Motion features discussed in Chapter 3, this chapter is designed to run on an iPhone or iPod touch. As of this writing, Apple does not expose the Health app or HealthKit on iPad.

To begin, make a copy of the IOTFit app you developed in Chapter 3 or download a copy from the GitHub repository for this book. Next, select the project settings for the app, by clicking IOTFit in the project hierarchy navigator. As shown in Figure 4-2, to declare that the app wants to advantage HealthKit, add the HealthKit capability to the app by clicking the Capabilities tab. Scrolling down to HealthKit, and clicking the switch once to flip it to ON.

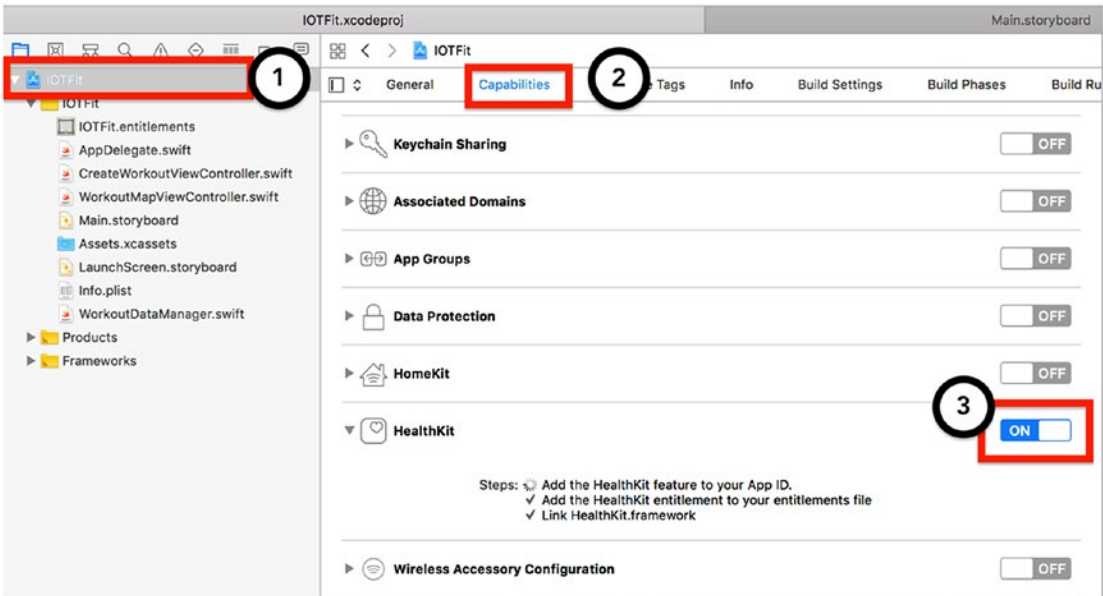


Figure 4-2. Enabling the HealthKit capability in the IOTFit project

If the app is not linked to a valid Apple Developer account (paid or free are both acceptable), enabling the HealthKit capability will fail. This will be expressed by the switch reverting to the OFF position. If you are having trouble remembering how to connect your project to an Apple Developer account, refer back to Chapter 1.

Next, as with all permission-based features, you must add keys to the project’s information property list (Info.plist), to define the messages that will appear in the system-generated permission alerts. As shown in Figure 4-3, click the Info tab, then click the (+) button that appears when you hover over the Privacy – Location When in Use Description row, to add a new key.

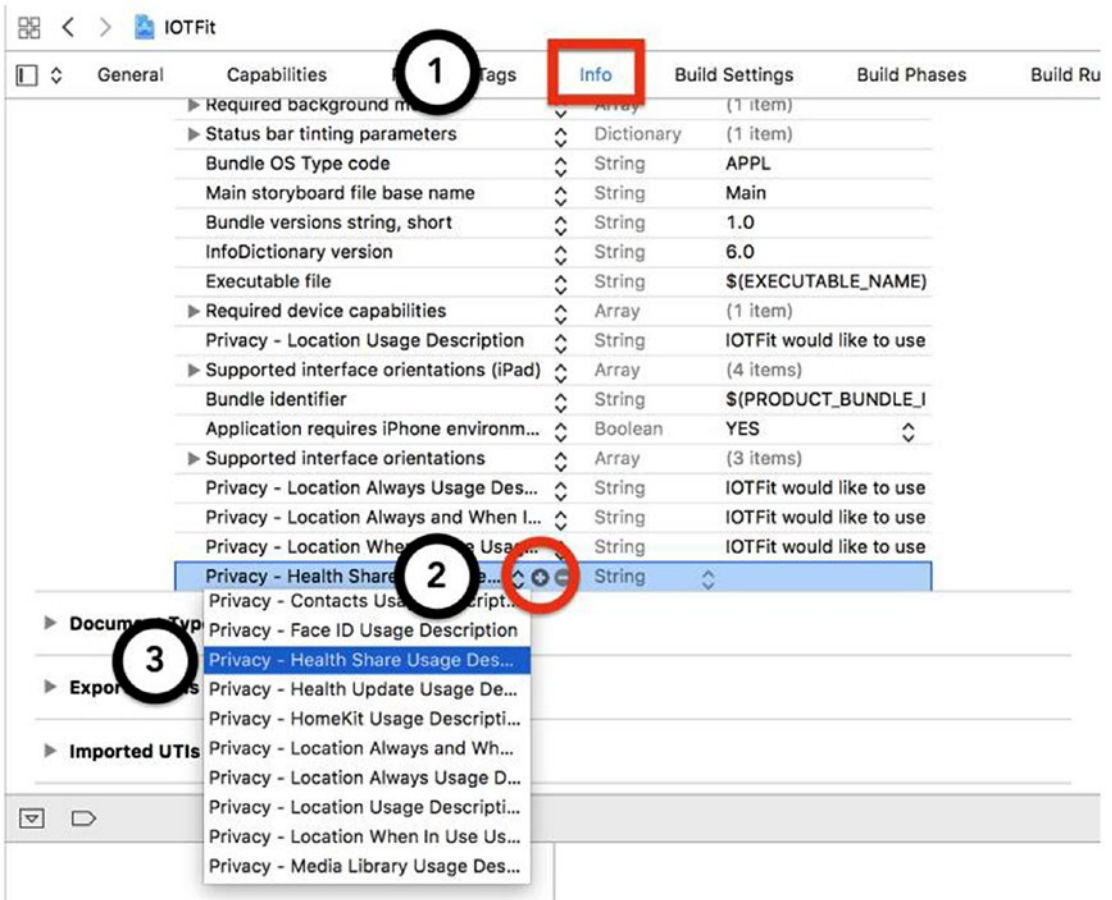


Figure 4-3. Adding new keys to IOTFit's information property list

To enable reading data from HealthKit, add the Privacy - Health Share Usage Description key-value pair. For the description string, the text I used was “IOTFit would like to use HealthKit permission to import workout data from the Health app into the History feature of the app. This information will not be shared online or with third parties.” To enable saving data to HealthKit, add the Privacy - Health Share Usage Description key-value pair. The text I used for the description string was “IOTFit would like to use HealthKit permission to export workout data to the Health app. This information will not be shared online or with third parties.”

In the same manner as Core Location, you should query for HealthKit availability and display its permission prompt before your first operation that attempts to use the framework. The primary class for accessing HealthKit from your applications is `HKHealthStore`. Similar to the manager objects from Core Motion, you instantiate an

instance of the class and make calls to HealthKit through that. Unlike Core Motion, there is a hard requirement from Apple that you only create one instance of the class in your application. For this purpose, the most appropriate place to put the HKHealthStore object would be in the WorkoutDataManager singleton. In Listing 4-1, I have modified the WorkoutDataManager class to add a HKHealthStore object as a property and instantiate it if the feature is available on the device.

Listing 4-1. Adding an HKHealthStore Property to the WorkoutDataManager Class

```
import Foundation
import CoreLocation
import HealthKit
...
class WorkoutDataManager {
    static let sharedManager = WorkoutDataManager()
    ...
    private var healthStore: HKHealthStore?

    private init() {
        print("Singleton initialized")
        loadFromPlist()

        if HKHealthStore.isHealthDataAvailable() {
            healthStore = HKHealthStore.init()
        }
    }
    ...
}
```

For the IOTFit app, the two points at which you will be interacting with HealthKit are when you must save a workout and when you have to read the list of workouts to display them on the Workout History Table View Controller. To make the code easier to read, I will perform these operations through `loadWorkoutsFromHealthKit()` and `saveWorkoutToHealthKit()` methods in the WorkoutDataManager class. As you do not know which operation the user is going to perform first, you should make the permission request in both methods. If the user has already accepted or declined the permission request, no alert will be shown, and the next instruction will execute immediately.

The method HealthKit uses for requesting health permission is `requestAuthorization(toShare:read:completion:)`, which is part of the `HKHealthStore` class. As its parameters, this method takes in a set of the types of health data you want to read and write and a completion handler that will execute when the user has made his/her decision (or immediately upon a second call). To work with HealthKit, you must specify every type of data you wish to read or write. My secondary reason for suggesting making the permission call before every HealthKit operation is because if your type list changes between app versions, iOS will show the alert pop-up again, asking to authorize your app for the new permissions. In Listing 4-2, I have added the new `loadWorkoutsFromHealthKit` and `saveWorkoutToHealthKit()` methods to the `WorkoutDataManager` class and added a call to `saveWorkoutToHealthKit()` in the `saveWorkout(duration:)` method, to trigger the health permission pop-up. Thanks to the `healthStore` property being declared as an optional value, if the availability request in the `init()` method fails, the permission request and its subsequent HealthKit operations will not execute.

Listing 4-2. Requesting Health Permissions Before Reading or Writing Health Data

```
class WorkoutDataManager {
    ...
    private var hkDataTypes: Set<HKSampleType> {
        var hkTypesSet = Set<HKSampleType>()
        if let stepCountType =
            HKQuantityType.quantityType(forIdentifier:
            HKQuantityTypeIdentifier.stepCount) {
            hkTypesSet.insert(stepCountType)
        }
        if let flightsClimbedType =
            HKQuantityType.quantityType(forIdentifier:
            HKQuantityTypeIdentifier.flightsClimbed) {
            hkTypesSet.insert(flightsClimbedType)
        }
        if let cyclingDistanceType =
            HKQuantityType.quantityType(forIdentifier:
            HKQuantityTypeIdentifier.distanceCycling) {
```

```

        hkTypesSet.insert(cyclingDistanceType)
    }
    if let walkingDistanceType =
        HKQuantityType.quantityType(forIdentifier:
            HKQuantityTypeIdentifier.distanceWalkingRunning) {
        hkTypesSet.insert(walkingDistanceType)
    }

    hkTypesSet.insert(HKObjectType.workoutType())
    return hkTypesSet
}
...
func saveWorkout(duration: TimeInterval) {
    ...
    saveToPlist()
    saveWorkoutToHealthKit()
}

func loadWorkoutsFromHealthKit() {
    healthStore?.requestAuthorization(toShare: hkDataTypes,
        read: hkDataTypes, completion: { (isAuthorized:
            Bool, error: Error?) in
        //Request completed, it is now safe to use HealthKit
    })
}

func saveWorkoutToHealthKit() {
    healthStore?.requestAuthorization(toShare: hkDataTypes,
        read: hkDataTypes, completion: { (isAuthorized:
            Bool, error: Error?) in
        //Request completed, it is now safe to use HealthKit
    })
}
}
}

```

To reduce duplicated code, I made the `hkDataTypes` computed property to represent the list of data types the app required. The types you must use to request permissions are implementations of the `HKSampleType` abstract class. As the name suggests, they represent data that is saved and measured in samples. You will learn more about how HealthKit represents data in the next section. Samples are just one of many types you can use. The most important point to remember now is that, as with like health permissions, not all sample types are available on all iOS devices, and you must query for the types you want to use before attempting to use them.

Now, if you run the IOTFit app and try to save a workout, you will be presented with the health permission alert, as shown in Figure 4-4. Users can selectively choose permissions they want to allow from this screen or turn everything on. When they press the Allow button, the settings will be saved, and the completion handler in your app will execute.

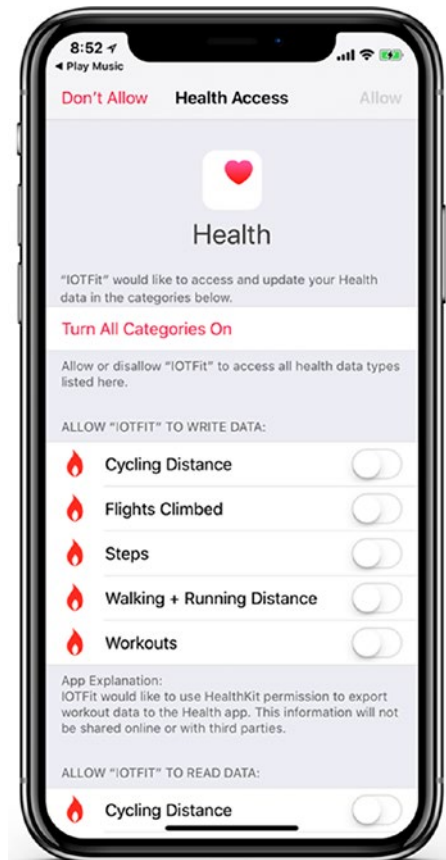


Figure 4-4. Health permission alert for the IOTFit app

Writing Data to HealthKit

The easiest way to start working with HealthKit is to use it to save workouts to the HealthKit store. In the preceding chapters, you generated enough data to get a good snapshot of the user's activity, but it needs to be serialized (massaged) into a format that can be consumed by HealthKit. Rather than storing data as simple data types, such as `String` or `Float`, HealthKit employs a hierarchical system that allows you to group related statistics together. It also has its own system of units that you must learn to work with, in order to make your data conform to a format the HealthKit store can recognize. In this section, you will learn the conceptual foundations for this system and how to use it to convert raw numbers into health data.

Understanding How HealthKit Represents Data

The HealthKit store in iOS classifies data into two primary categories: *characteristic data* and *sample data*. Characteristic data refers to qualitative information about a user, which is described in characteristics, rather than units, and does not change frequently. For instance, a user's blood type or skin color. In HealthKit, these are represented by instantiations of the `HKCharacteristicType` class. Sample data refers to information arising from a user action, which can be measured and described in quantitative units (for example, meters) or quantitative descriptors of the data (for example, steps) or the aggregate action (for example, workouts). In HealthKit, these are represented by subclasses of the `HKSample` class. In the IOTFit app, you will be working primarily with sample data.

In HealthKit terminology, quantitative sample data (such as step count or distance) are called *quantity samples* and represented by objects of the `HKQuantitySample` class. Sample data that describes the characteristics of an activity (such as if a workout should be described as bicycling or running) are called *category samples* and represented by `HKCategorySample` objects. For describing aggregate activities, HealthKit defines two final sample types: *workouts* and *correlations*. Workouts are represented by `HKWorkout` objects and fit the purpose their name suggests: they are meant to represent a set of data about a workout the user has performed. Correlations are represented by `HKCollection` objects and although their name is a bit hard to derive meaning from, their purpose is similar to that of workouts: they are specifically intended for grouping data about food that was consumed or a blood pressure reading.

Units in HealthKit are represented by the `HKUnit` objects. When storing quantitative samples, you must specify which `HKUnit` class to use to represent the data. You can do this by either using a convenience method provided by Apple for commonly used units (for example, `HKUnit.meter()`) or defining your own by passing a `String` to the `init(from:)` convenience initializer method for the `HKUnit` class.

Creating and Saving HealthKit Samples

For the IOTFit app and your own projects, you must follow these steps to save data to HealthKit:

- Verify that the device is capable of storing the samples you want to work with (workouts, correlations, and quantity samples)
- Create an aggregate sample object (workout or correlation)
- Create quantity samples
- Save the samples to a workout

In Listing 4-2, you performed the verify step by checking the quantity types you wanted to work with (step count, flights climbed, workout type, walking distance, and cycling distance). Now that you have a better idea of how HealthKit represents data, you can begin to convert the simple numerical data from the Create Workout View Controller into something HealthKit can consume.

The first step you must perform to save data to HealthKit is to create a `HKWorkout` object to represent the workout. There are several convenience constructor methods for the `HKWorkout` class, but the most appropriate one for storing workouts from IOTFit is `init(activityType:start:end:duration:totalEnergyBurned:totalDistance:distanceQuantity:device:metadata)`, which will allow you to specify a workout type (`HKWorkoutType`), start and end dates (`Date` objects), and workout distance. To enable this operation, as well as eventually saving such other statistics as flights climbed, you have to modify the `Workout` struct to add the new parameters and refactor the calling functions to use a `Workout` as a parameter, instead of a long list of parameters. In Listing 4-3, I have modified the `CreateWorkoutViewController` and `WorkoutDataManager` classes to include these changes.

Listing 4-3. Modifying the CreateWorkoutViewController and WorkoutDataManager Classes to Include All Workout Data

```

class CreateWorkoutViewController: UIViewController {
    ...
    @IBAction func toggleWorkout() {
        switch currentWorkoutState {
            ...
            case .active:
                currentWorkoutState = .inactive
            ...
            if let workoutStartTime = workoutStartTime {
                let workout = Workout(startTime: workoutStartTime,
                    endTime: Date(), duration: workoutDuration,
                    locations: [], workoutType:
                    self.currentWorkoutType, totalSteps:
                    workoutSteps, flightsClimbed: floorsAscended,
                    distance: workoutDistance)
                WorkoutDataManager.sharedManager.saveWorkout(workout)
            }
            default:
                NSLog("Error")
        }
        updateUserInterface()
    }
}

```

// The following lines are part of the
// WorkoutDataManager.swift file

```

struct Workout: Codable {
    var startTime: Date
    var endTime: Date
    var duration: TimeInterval
    var locations: [Coordinate]
    var workoutType: String
    var totalSteps: Double
}

```

```

var flightsClimbed: Double
var distance: Double
}

class WorkoutDataManager {
    ...
    func saveWorkout(_ workout: Workout) {
        var activeWorkout = workout
        ...
        saveToPlist()
        workouts?.append(activeWorkout)
        saveWorkoutToHealthKit(activeWorkout)
    }

    func saveWorkoutToHealthKit(_ workout: Workout) {
        healthStore?.requestAuthorization(toShare:
            hkDataTypes, read: hkDataTypes, completion: {
            [weak self] (isAuthorized: Bool, error:
                Error?) in
                ...
            })
    }
}

```

Now that you have access to all the data you require, you can try to create the HKWorkout object. In Listing 4-4, I have modified the `saveWorkoutToHealthKit(...)` method to include the new call to create the HKWorkout object. I created the `createHKWorkout(workoutType:startDate:endDate)` method to help manage converting the custom WorkoutType struct to HealthKit's HKWorkoutActivityType category sample type. Use a guard-let or if-let block to verify that the object was created successfully, before attempting to use it.

Listing 4-4. Creating an HKWorkout Object

```

class WorkoutDataManager {
    func saveWorkoutToHealthKit(stepCount: Double,
        flightsClimbed: Double, distance: Double,
        workoutType: String, startDate: Date, endDate:
        Date) {
        healthStore?.requestAuthorization(toShare:
        hkDataTypes, read: hkDataTypes, completion: {
            [weak self] (isAuthorized: Bool, error:
            Error?) in

                if let error = error {
                    NSLog("Error accessing HealthKit")
                } else {
                    guard let workoutObject =
                        self?.createHKWorkout(workout)
                        else { return }
                }
            })
    }

func createHKWorkout(_ workout: Workout) -> HKWorkout? {
    let distanceQuantity = HKQuantity(unit: HKUnit.meter(),
        doubleValue: workout.distance)
    var activityType = HKWorkoutActivityType.walking

    switch(workout.workoutType) {
    case WorkoutType.running:
        activityType = HKWorkoutActivityType.running
    case WorkoutType.bicycling:
        activityType = HKWorkoutActivityType.cycling
    default:
        activityType = HKWorkoutActivityType.walking
    }
    return HKWorkout(activityType: activityType, start:
        workout.startTime, end: workout.endTime, duration:

```

```

workout.duration, totalEnergyBurned: nil,
totalDistance: distanceQuantity , device: nil,
metadata: nil)
    }
}

```

One of the downsides of working with HealthKit is that the setup steps required to use it are very rigid. After creating a workout, the only way to attach samples to it is by first saving it to the HealthKit store. If the operation is successful, you can begin to add samples to the workout. In Listing 4-5, I have modified the `saveWorkoutToHealthKit(...)` method to save the workout and then make a call to a function that will be used to add the samples.

Listing 4-5. Preparing to Add Samples to a Workout

```

class WorkoutDataManager {
    ...
    func saveWorkoutToHealthKit(_ workout: Workout) {
        healthStore?.requestAuthorization(toShare:
            hkDataTypes, read: hkDataTypes, completion: {
            [weak self] (isAuthorized: Bool,error:Error?)
            in
                if let error = error {
                    NSLog("Error accessing HealthKit")
                } else {
                    guard let workoutObject =
                        self?.createHKWorkout(workout)
                    else { return }

                    self?.healthStore?.save(workoutObject,
                        withCompletion: { (completed: Bool,
                        error: Error?) in
                            if let error = error {
                                NSLog("Error creating workout")
                            }
                        }
                    )
                }
            }
        )
    }
}

```

```

        } else {
            self?.addSamples(hkWorkout:
                workoutObject, workoutData:
                    workout)
        }
    })
}
})
}
...
func addSamples(hkWorkout: HKWorkout, workoutData: Workout){
    var samples = [HKSample]()
    addStepCountSample(workoutData, objectArray: &samples)
    addFlightsClimbedSample(workoutData, objectArray:
        &samples)
    addDistanceSample(workoutData, activityType:
        hkWorkout.workoutActivityType, objectArray: &samples)
    self.healthStore?.add(samples, to: hkWorkout, completion: {
        (saveCompleted: Bool, saveError: Error?) in
            if let saveError = saveError {
                NSLog("Error adding workout samples")
            } else {
                NSLog("Workout samples added successfully!")
            }
        })
}
}
}

```

As shown in the `addSamples(...)` method, to add samples to a workout, you build a series of `HKSample` objects and call the `add(to:completion:)` method on the iOS health store, using the workout object you created earlier. The setup code for an `HKSample` object can get lengthy, so I created methods to generate each sample and append it to the array.

Harkening back to the discussion on sample types in HealthKit, to store quantitative data about the workout, you will want to use the `HKQuantitySample` class. In comparing the convenience initializers, the best one to use would be `init(type: quantity: start: end:)`. At this point, the scaffolding code starts to get lengthy. To use the quantity parameter, you must generate a `HKQuantity` object and specify an `HKQuantityType` object to represent the quantity type. Similarly, you will also have to specify the unit type. In Listing 4-6, I have implemented the `addStepCountSample(...)` and `addFlightsClimbedSample(...)` methods, which implement all of these steps.

Listing 4-6. Creating “Step Count” and “Flights Climbed” Sample Objects

```
class WorkoutDataManager {
    ...
    func addStepCountSample(_ workoutData: Workout,
        objectArray: inout [HKSample]) {
        guard let stepQuantityType =
            HKQuantityType.quantityType(forIdentifier:
                HKQuantityTypeIdentifier.stepCount)
        else { return }
        let stepUnit = HKUnit.count()
        let stepQuantity = HKQuantity(unit: stepUnit,
            doubleValue: workoutData.totalSteps)
        let stepSample = HKQuantitySample(type:
            stepQuantityType, quantity: stepQuantity, start:
                workoutData.startTime, end: workoutData.endTime)
        objectArray.append(stepSample)
    }

    func addFlightsClimbedSample(_ workoutData: Workout,
        objectArray: inout [HKSample]) {
        guard let flightQuantityType =
            HKQuantityType.quantityType(forIdentifier:
                HKQuantityTypeIdentifier.flightsClimbed)
        else { return }
        let flightUnit = HKUnit.count()
        let flightQuantity = HKQuantity(unit: flightUnit,
            doubleValue: workoutData.flightsClimbed)
    }
}
```

```

    let flightSample = HKQuantitySample(type:
        flightQuantityType, quantity: flightQuantity,
        start: workoutData.startTime, end:
        workoutData.endTime)
    objectArray.append(flightSample)
}
}

```

In these methods, the two lines that stand out are the method signature and lookup of the quantity type. The `inout` keyword specifies that a parameter should be *passed by reference*. For C/C++ programmers, this term should be very familiar. Passing by reference is a way to modify the contents of a parameter from within a method, rather than operating on a copy of that data. When you make the method call, you add the `&` symbol in front of the name of the variable that will be modified to pass it *by reference*. Apple uses this pattern frequently in their methods that pass back an `Error` object.

In the line where you look up the quantity type, once again you will have to check that the result is valid before using it. As mentioned earlier, some sample types are not available on all iOS devices or iOS versions. Better to be safe than sorry!

The final quantity type you must add is workout distance. One of the particularities of this quantity type is that Apple treats walking/running distance and cycling distance as separate quantity types. To remove this limitation, in Listing 4-7, I have implemented the `addWorkoutDistance(...)` method, using the `workout` type to specify the quantity type.

Listing 4-7. Adding the “Workout Distance” Sample Object

```

class WorkoutDataManager {
    ...
    func addDistanceSample(_ workoutData: Workout,
        activityType: HKWorkoutActivityType, objectArray: inout
        [HKSample]) {
        guard let cyclingDistanceType =
            HKQuantityType.quantityType(forIdentifier:
            HKQuantityTypeIdentifier.distanceCycling),
            let walkingDistanceType =
            HKQuantityType.quantityType(forIdentifier:
            HKQuantityTypeIdentifier.distanceWalkingRunning)

```

```

else { return }
let distanceUnit = HKUnit.meter()
let distanceQuantity = HKQuantity(unit: distanceUnit,
    doubleValue: workoutData.distance)

let distanceQuantityType = activityType ==
    HKWorkoutActivityType.cycling ? cyclingDistanceType:
    walkingDistanceType

let distanceSample = HKQuantitySample(type:
    distanceQuantityType, quantity: distanceQuantity,
    start: workoutData.startTime, end:
    workoutData.endTime)
objectArray.append(distanceSample)
}
}

```

Believe it or not, this completes all the steps required to save data to HealthKit! Now, if you complete a workout on the Create Workout View Controller, it will create a new workout that you can view in the iOS Health app, as shown in Figure 4-5. To view the workout, in the Health app, click the Sources tab, then IOTFit, then Data, and finally Workouts.

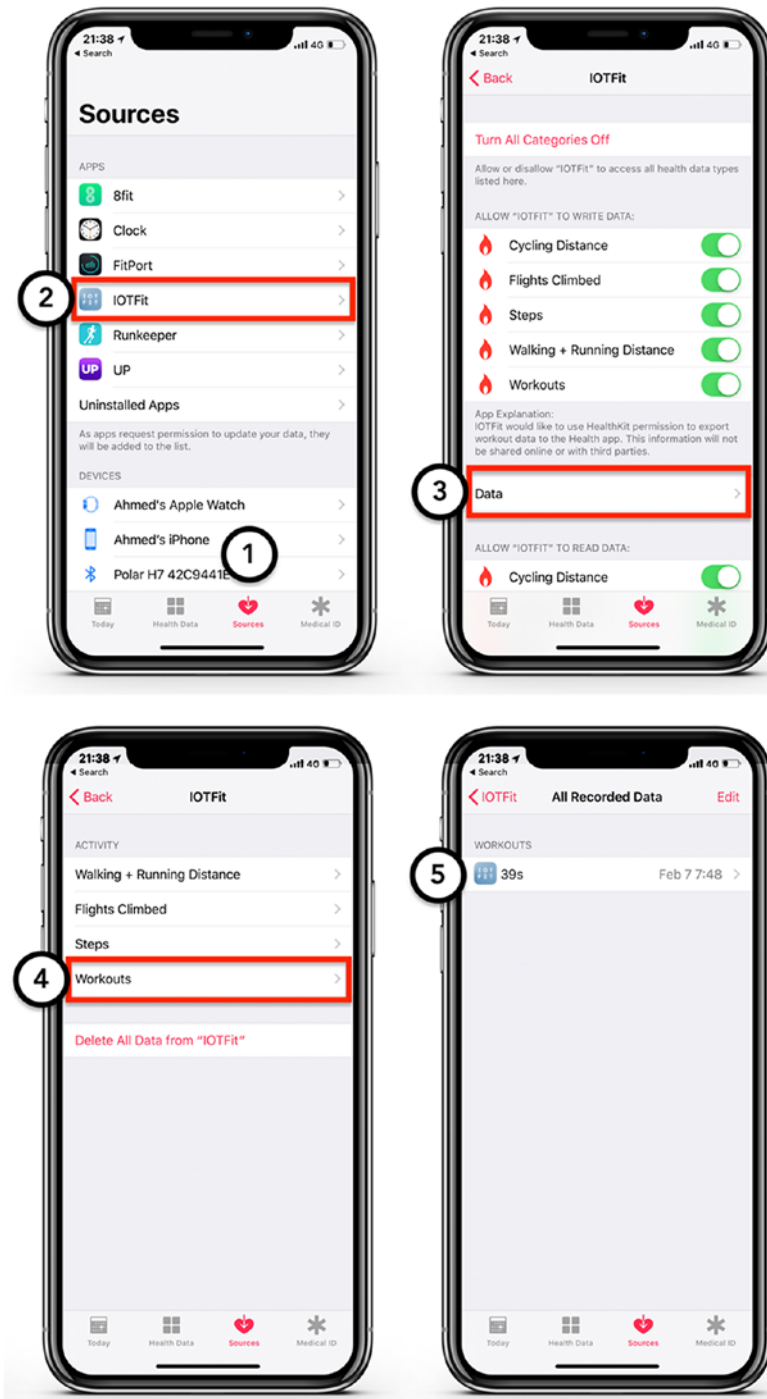


Figure 4-5. Viewing a saved workout in the iOS Health app

Reading Workout Data from HealthKit

Now that you understand how HealthKit represents data, as well as the steps required to convert simple data types into HealthKit sample data, you can use that knowledge to read data from the iOS HealthKit store. In the IOTFit app, you will use this to fetch the user's workout history. Much like writing data, the primary data type you will use for reading data from the HealthKit store is `HKSample`.

To fetch data, you must perform a *query* on the HealthKit store, using the `HKSampleQuery` class. As with creating an `HKSample` object, there are several convenience initializers you can use, but for the IOTFit app, `HKSampleQuery(sampleType:predicate:limit:, sortDescriptors:resultsHandler)` is an appropriate choice. There is a lot to unpack in this, but the important concepts are that you must specify a sample type to fetch, a *predicate* to filter the search results, and a completion handler to process the results. In Listing 4-8, I have updated the `loadWorkoutsFromHealthKit()` method to include a query for workouts, in decreasing date order (newest first), limited to the newest ten from the preceding week.

Listing 4-8. Setting Up a HealthKit Sample Query

```
func loadWorkoutsFromHealthKit(completion: @escaping
(([Workout]?) -> Void)) {
    healthStore?.requestAuthorization(toShare:
        hkDataTypes, read: hkDataTypes, completion: {
        [weak self] (isAuthorized: Bool, error :
Error?) in
            if let error = error {
                NSLog("Error accessing HealthKit")
            } else {
                let workoutType = HKCategoryType.workoutType()
                let weekAgo = Date(timeIntervalSinceNow:
                    -3600 * 24 * 7)
                let predicate = HKQuery.predicateForSamples(withStart:
                    weekAgo, end: Date(), options: [])
                let sortDescriptor = NSSortDescriptor(key: "startDate",
                    ascending: false)
            }
        }
    )
}
```



```

let query = HKSampleQuery(sampleType: workoutType,
    predicate: predicate, limit: 10, sortDescriptors:
    [sortDescriptor], resultsHandler: { (query:
    HKSampleQuery, samples: [HKSample]?, error:
    Error?) in
        if let error = error {
            NSLog("Error fetching items from HealthKit ")
            completion(nil)
        } else {
            let workouts = [Workout]()
            completion(workouts)
        }
    })
    self?.healthStore?.execute(query)
}
})
}

```

Unless a user disables location and step counting, iOS is always collecting health data on a user. To avoid the risk of your query taking too long, I recommend using a date-based predicate or (page) limit to restrict the query results. You can always use your user interface to allow the user to see more search results. Another important point to remember is that after you declare the query, it will not start until you call the `execute()` method on your `HKHealthStore` object.

Next, to use the results of the sample query in the app, you must convert the `HKSample` array to custom `Workout` structs you have been using throughout the project. In Listing 4-9, I have further expanded the `loadWorkoutsFromHealthKit(completion:)` method to perform the conversion logic.

Listing 4-9. Converting HealthKit Sample Objects to Simple Data Types

```

func loadWorkoutsFromHealthKit(completion: @escaping ([[Workout]?) -> Void) {
    healthStore?.requestAuthorization(toShare:
        hkDataTypes, read: hkDataTypes, completion: {
        [weak self] (isAuthorized: Bool, error:
        Error?) in
            if let error = error {

```

```

    NSLog("Error accessing HealthKit")
} else {
    ...
let query = HKSampleQuery(sampleType:
    workoutType, predicate: predicate, limit:
    10, sortDescriptors: [sortDescriptor],
    resultsHandler: { (query: HKSampleQuery,
    samples: [HKSample]?, error: Error?) in
    if let error = error {
        NSLog("Error fetching items")
        completion(nil)
    } else {
        guard let hkWorkouts = samples as?
            [HKWorkout] else {
            completion(nil)
            return
        }
        let workouts = hkWorkouts.map({ (hkWorkout:
            HKWorkout) -> Workout in
            let totalDistance =
                hkWorkout.totalDistance?.doubleValue(
                    for: HKUnit.meter()) ?? 0
            let flightsClimbed =
                hkWorkout.totalFlightsClimbed?.
                doubleValue(for: HKUnit.count()) ?? 0
            var workoutType = WorkoutType.walking
            switch(hkWorkout.workoutActivityType) {
            case .running:
                workoutType = WorkoutType.running
            case .cycling:
                workoutType = WorkoutType.bicycling
            default:
                workoutType = WorkoutType.walking
            }
        }
    }
}

```

```

        return Workout(startTime:
            hkWorkout.startDate, endTime:
            hkWorkout.endDate, duration:
            hkWorkout.duration, locations: [],
            workoutType: workoutType, totalSteps: 0,
            flightsClimbed: flightsClimbed,
            distance: totalDistance)
    })
    completion(workouts)
}
})
self?.healthStore?.execute(query)
}
})
}

```

In this function, the first challenge I had to tackle was making sure the sample was a `HKWorkout` object. In order to make a generic completion handler, Apple must use a type general enough to handle all sample data. However, for the purposes of `IOTFit`, the type I had to work with was `HKWorkout`. Next, I had to extract simple data types from `HKSample` objects. To cast the data down, I used the `doubleValue(for:)` method and specified the unit to convert from. If the operation fails, I return 0 as a default value. The final challenge was to convert the `HKWorkoutType` property to a `String` with which to build a `Workout` object. This, however, was easily taken care of with a `switch()` statement.

Using a Table View Controller to Display Data

The final step in this chapter is to display the workout history. For many data-based apps, a `Table View Controller` is a great choice for letting users interact with the data. Out of the box, it provides you with a scrolling user interface, an easy way to display results in a uniform manner (cells), and components that you can easily customize through `Interface Builder`. The difficulty, however, is that the setup code for the `UITableViewController` class is very *protocol*-heavy and can be intimidating for newer iOS developers. In learning how to display the workout results for `IOTFit` on a table, you will learn three fundamental skills for working with the `UITableViewController` class that you can apply to other apps: adding a `Table View Controller` from `Interface Builder`,

setting up the `UITableViewDataSource` delegate methods for populating the table, and setting up the `UITableViewDelegate` delegate methods for displaying the workout content in each cell.

To begin, use Xcode's template feature to create a new `UITableViewControllersubclass`. As in the case of the examples in Chapters 1 and 2, go to the File menu, then select New ► File and, from the template picker, choose Cocoa Touch Class, as shown in Figure 4-6.

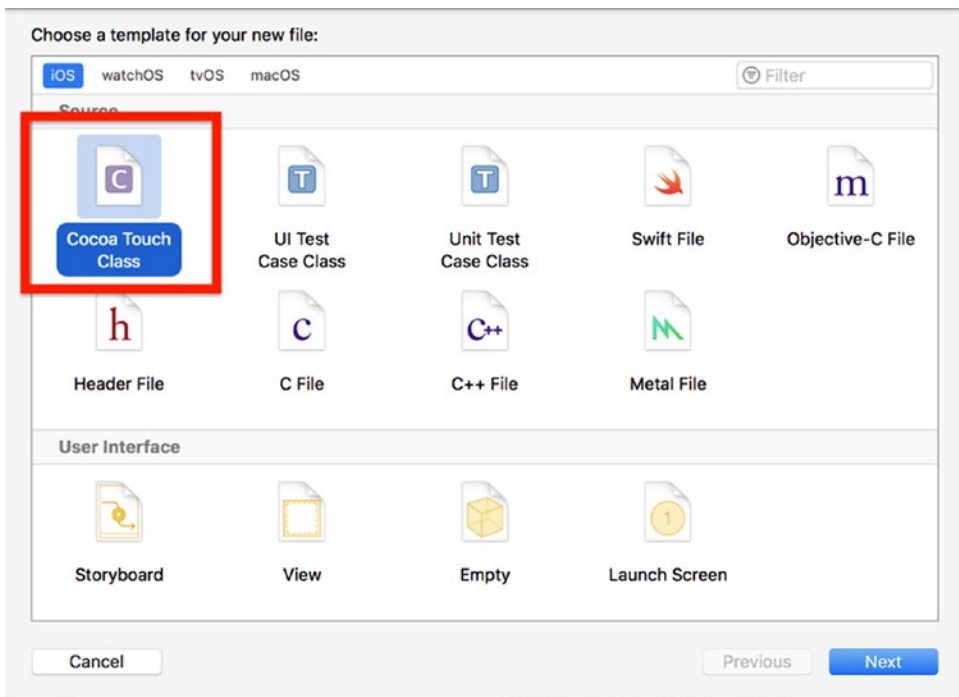


Figure 4-6. *Creating a new Cocoa Touch class*

Next you will be asked to name the new class and select a base class. As shown in Figure 4-7, name the new file `WorkoutTableViewCellController` and choose `UITableViewCell` as the base class.

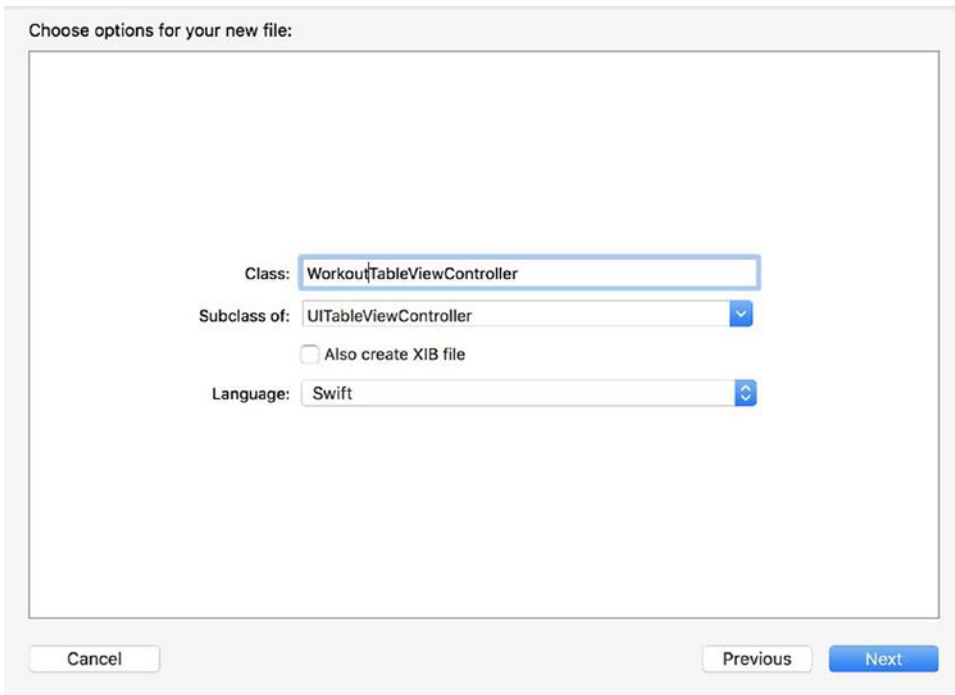


Figure 4-7. Creating a new `UITableViewController` class based on a template

Your new `WorkoutTableViewCell` class should resemble Listing 4-10, in which Apple’s template provides you with stubs for the required methods you require to implement the `UITableViewDelegate` and `UITableViewDataSource` protocols.

Listing 4-10. Empty `WorkoutTableViewCell` Class

```
import UIKit

class WorkoutTableViewCell: UITableViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
    }
    // MARK: - Table view data source

    override func numberOfSections(in tableView:
        UITableView) -> Int {
        return 0
    }
}
```

```
override func tableView(_ tableView: UITableView,
    numberOfRowsInSection section: Int) -> Int {
    return 0
}

override func tableView(_ tableView: UITableView,
    cellForRowAt indexPath: IndexPath) ->
    UITableViewCell {
    let cell = tableView.dequeueReusableCell(
        withIdentifier: "Identifier", for:
        indexPath)
    return cell
}
}
```

Setting Up the User Interface

First, select the `Main.storyboard` file from the Project Navigator to open Interface Builder in the Xcode's editor (center) pane. As shown in Figure 4-8, drag and drop a Table View Controller object from the Object Library to the storyboard.

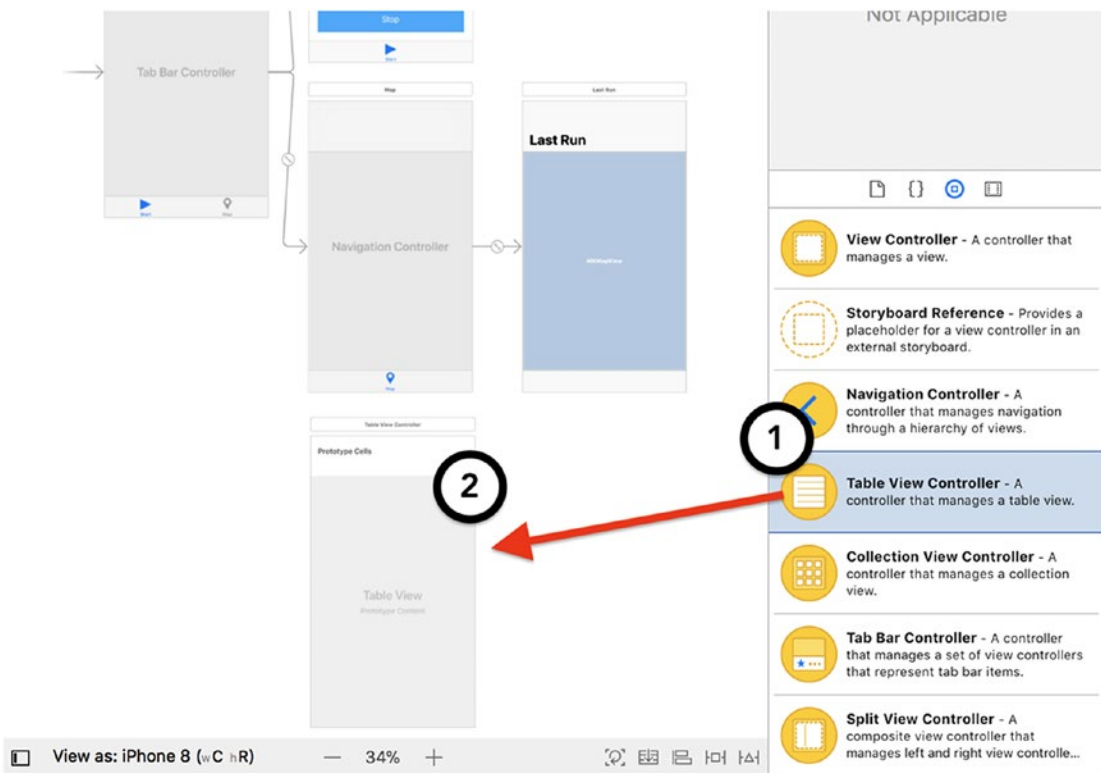


Figure 4-8. Adding a Table View Controller to the storyboard

To attach the Table View Controller to the Tab View Controller, hold down the Control button on your keyboard and drag a line to the Tab View Controller. From the context menu, select View Controllers as the relationship type, as shown in Figure 4-9.

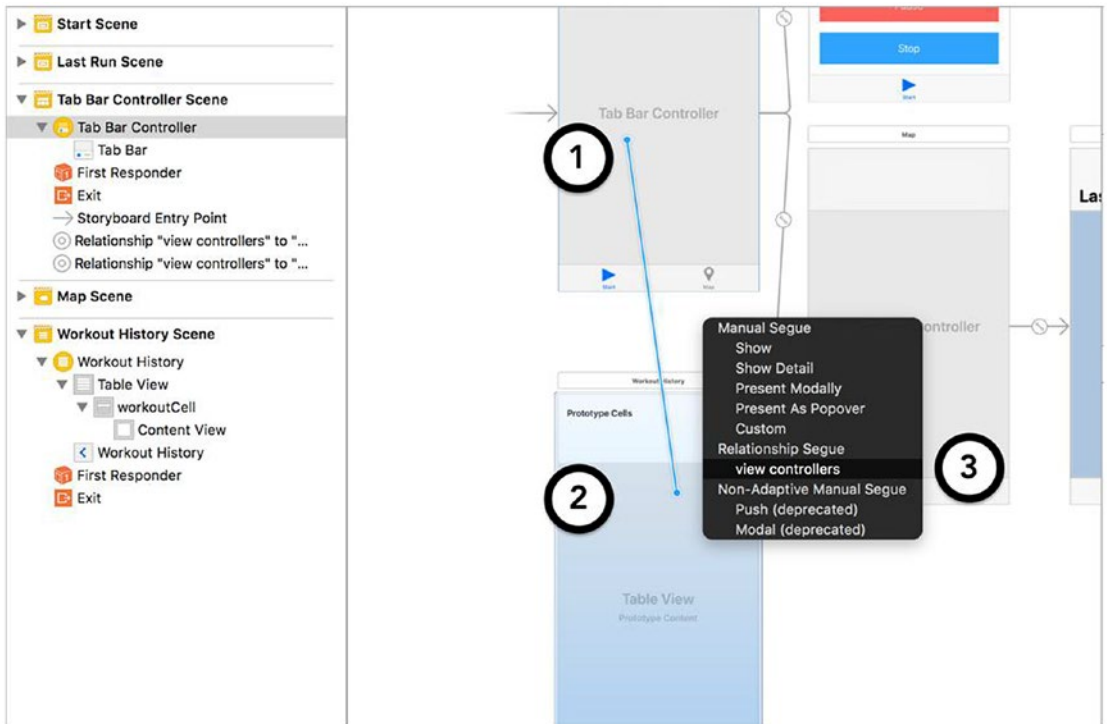


Figure 4-9. Adding new items to a Tab View Controller

To change the icon on the Tab bar, click the Tab Bar under the Table View Controller, and in the Attributes Inspector, select History from the System Item row, as shown in Figure 4-10.

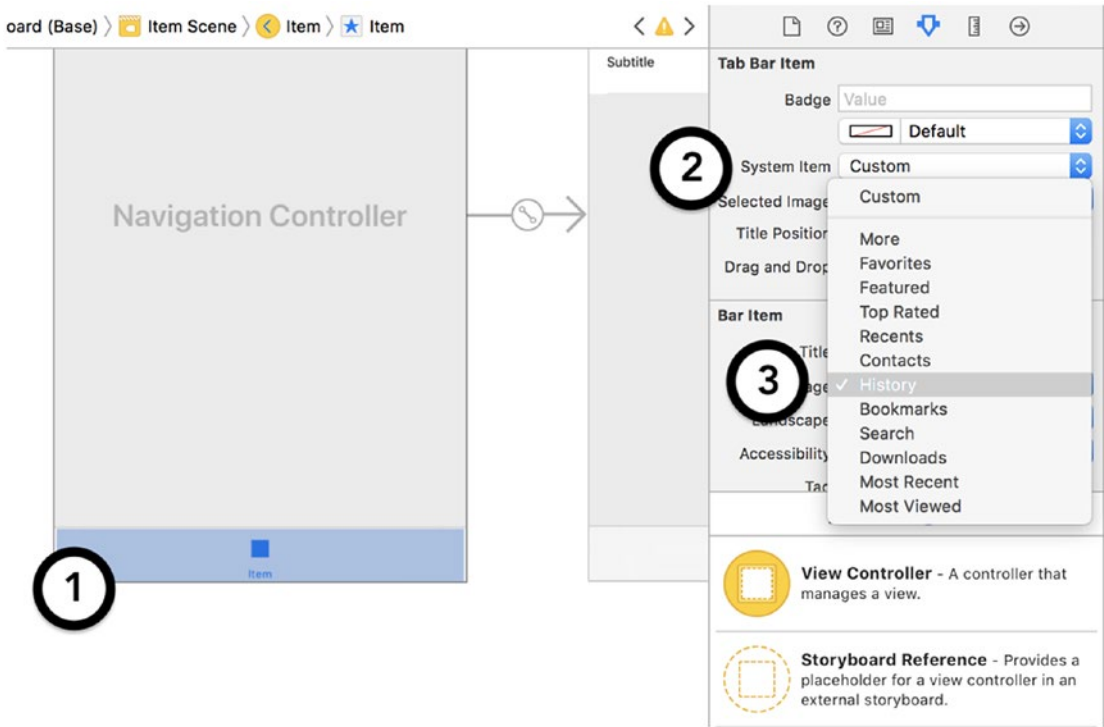


Figure 4-10. Changing the icon for a Tab bar item

To make the cell match the wireframe in Figure 4-1, where there are two lines of text, click the cell, and in the Attributes Inspector, set the style to Subtitle, as shown in Figure 4-11.

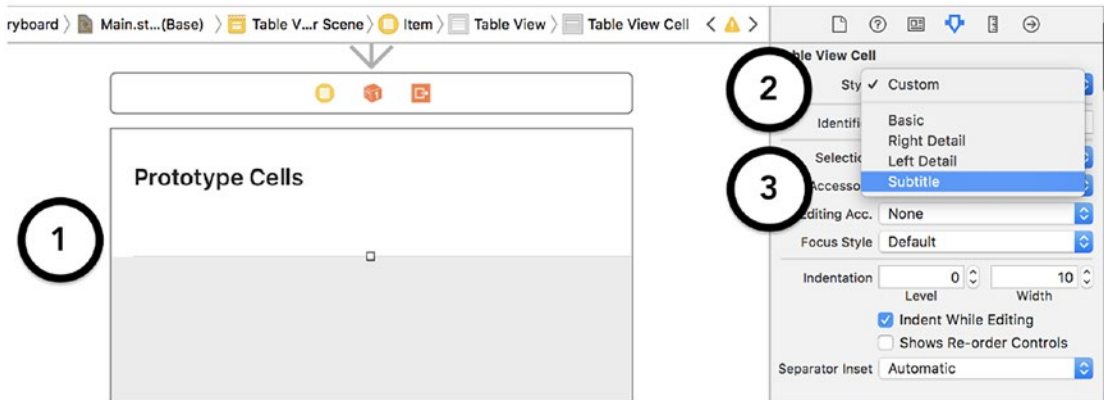


Figure 4-11. Changing the style of a Table View Cell

For the final layout task, you must embed the Table View Controller in a navigation controller. This will help the app maintain consistency between the workout history and map screens. To perform this operation, click to select the view controller and then, from the Editor menu, select Embed In ► Navigation Controller. To edit the title of the navigation item for the Workout Table View Controller, double-click the navigation bar above the screen and begin typing in the text field that appears. To make the text large, enable the Prefers Large Titles option for the navigation controller. When all of these operations are completed, your Workout Table View Controller should be similar to the screenshot in Figure 4-12.

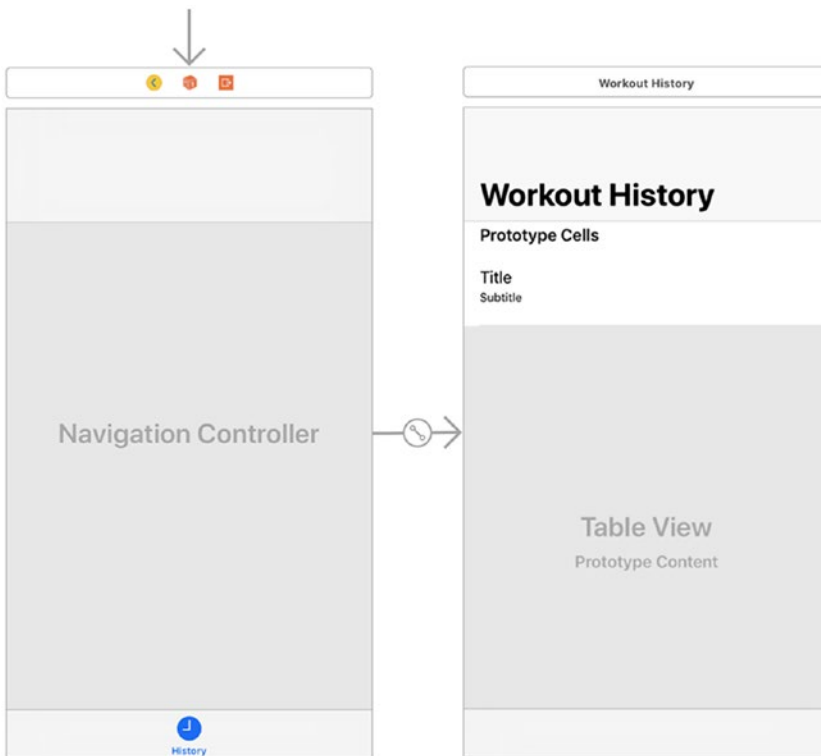


Figure 4-12. Completed storyboard for Workout Table View Controller

Now that the layout is complete, you must modify the ownership and outlets of the Workout Table View Controller so that it can interact with the `WorkoutTableViewController.swift` file. Click the Table View Controller, then navigate to the Identity Inspector, and in the Class text field, type “`WorkoutTableViewController`,” to set the ownership of the file, as shown in Figure 4-13.

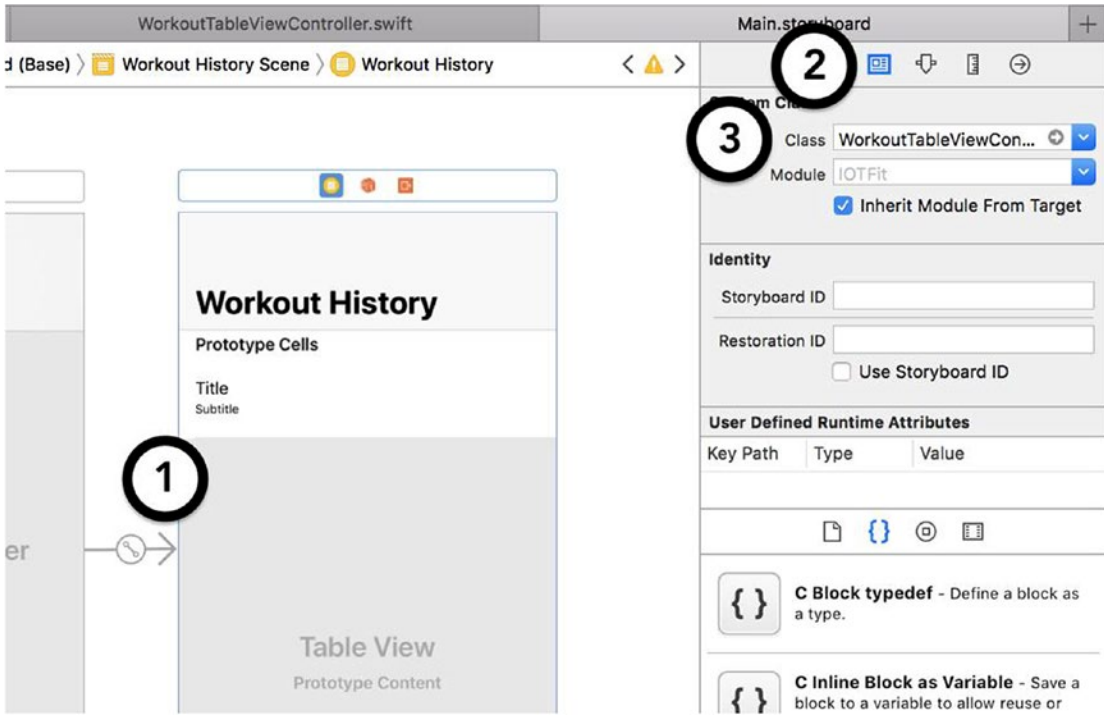


Figure 4-13. Setting ownership for the Table View Controller

Note Setting ownership only works when a class or property is defined in your project using a subclass based on the object library template you are trying to use (for example, Table View Controller, Button).

In Chapters 1 and 2, you used outlets to connect button handlers to methods in an object’s owning class. For a Table View Controller, you must perform a similar operation to identify the `UITableViewDataSource` and `UITableViewDelegate` delegate objects. As shown in Figure 4-14, click Connections Inspector, then drag lines from the `dataSource` and `delegate` outlets to the Workout Table View Controller.

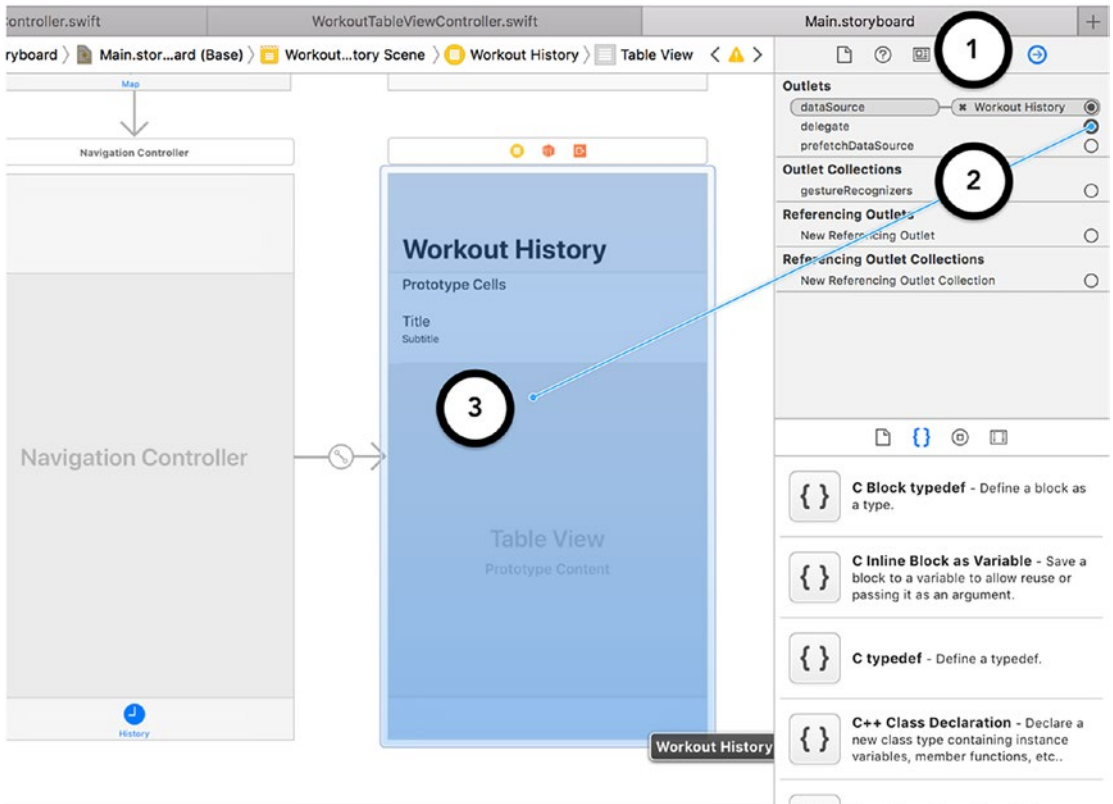


Figure 4-14. Setting delegate outlets for the Table View Controller

For the final connection step, you must set an identifier for the cell template in the Workout Table View Controller. Based on this identifier, you can look up the cell and modify its contents at runtime. As shown in Figure 4-15, select the cell, then navigate to the Attributes Inspector and enter a title in the Identifier text field, to set the identifier for the cell.

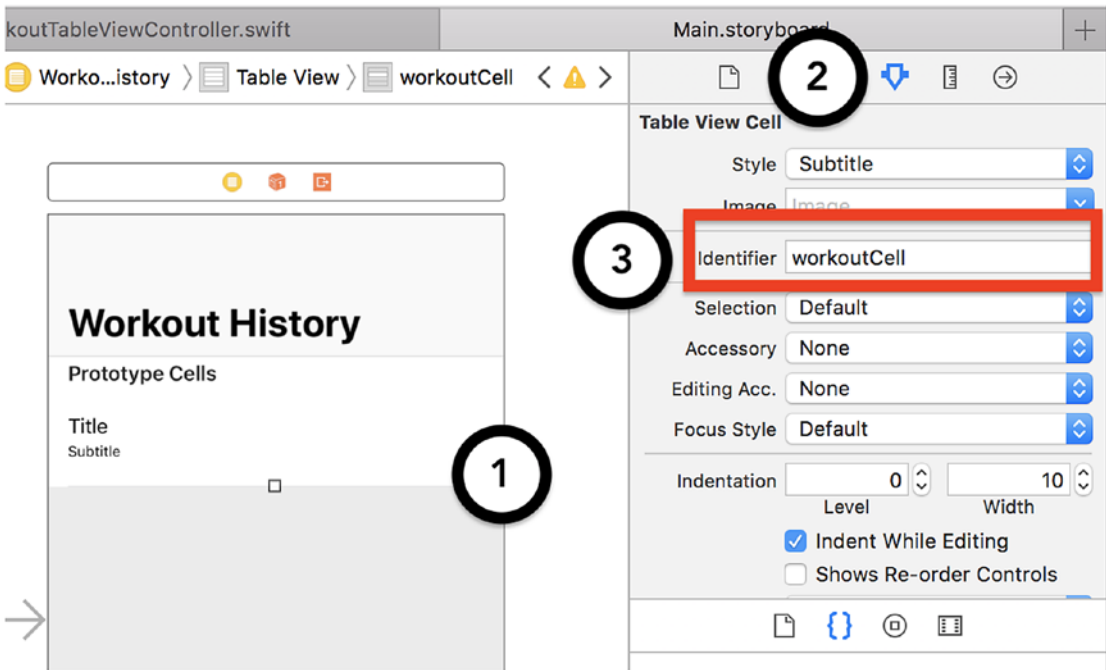


Figure 4-15. Setting an identifier for a Table View Cell

Using the UITableViewDataSource Protocol to Populate the Table View

Now that the Workout Table View Controller is completely laid out, you must implement the UITableViewDataSource protocol methods to populate the table view. These let you specify the number of rows, sections, titles, and editing features of the table view, such as if you want the user to be able to rearrange cells in the table. For the IOTFit app, the methods you must implement are focused on the number of sections and rows in the table view. The pattern most developers use to implement this behavior is to define a single or multidimensional array to represent the data and extract the hierarchy of the data from there. In Listing 4-11, I added the workouts property to the WorkoutTableViewController class to hold the workout data and used that to determine the values for the numberOfSections() and tableView(numberOfRowsInSection:) delegate methods. Apple’s Table View Controller template provided empty implementations (stubs) for these methods.

Listing 4-11. Using an Array to Determine the Rows and Sections in a Table View Controller

```
class WorkoutTableViewController: UITableViewController {

    var workouts: [Workout]?

    override func numberOfSections(in tableView:
        UITableView) -> Int {
        return 1
    }

    override func tableView(_ tableView: UITableView,
        numberOfRowsInSection section: Int) -> Int {
        return self.workouts?.count ?? 0
    }

    ...
}
```

Although you specified the number of rows and sections for the Table View Controller in Listing 4-11, the `workouts` property will be empty until you populate it. For this purpose, you can bring together everything you learned so far in this chapter and use the `loadWorkoutsFromHealthKit()` method from the Workout Data Manager to populate the `workouts` array. In Listing 4-12, I have modified the `WorkoutTableViewController` class to override the `viewWillAppear()` method and make the call to load the data there. The `viewWillAppear()` method is called every time the table view is about to be presented, such as switching from another tab in the app, and is an appropriate place to check for updates.

Listing 4-12. Triggering Updates for the Table View Controller

```
class WorkoutTableViewController: UITableViewController {

    ...

    override func viewWillAppear(_ animated: Bool) {
        super.viewWillAppear(animated)
        WorkoutDataManager.sharedManager.loadWorkoutsFromHealthKit
        { [weak self] (fetchedWorkouts: [Workout]?) in
            if let fetchedWorkouts = fetchedWorkouts {
                self?.workouts = fetchedWorkouts
            }
        }
    }
}
```

```

        DispatchQueue.main.async {
            self?.tableView?.reloadData()
        }
    }
}
...
}

```

Inside the completion handler, I called the `reloadData()` method on the `tableView` property of the class to force it to reload the data. Although the array is populated, the table view will not reload the data until you ask it to.

Using the UITableViewDelegate Protocol to Map Data to Cells

To display the data from the workouts array on the Workout Table View Controller, you must implement the `UITableViewDelegate` protocol. This protocol is responsible for the display and general user interaction properties of the table view, such as the height of each cell, assigning methods to populate cell contents, and what should happen when a user selects a cell. As with the `UITableViewDataSource` protocol, for the IOTFit app, you do not have to implement all of the methods provided by this protocol. To populate the cells in the IOTFit app, you will implement the `tableView(cellForRow:)` method. This allows use of a section and row number to determine the display of a cell at runtime. Additionally, every time you reload the table view, this method is called for each of the cells.

The advantage of using an array to manage the data for a Table View Controller is that it makes mapping data extremely simple. You can simply correlate a row number to the position in the array. In Listing 4-13, I use this logic to implement the `tableView(cellForRow:)` method for the `WorkoutTableViewController` class. I use the `workoutCell` identifier to look up the cell template, then create formatted strings, based on the values in each `Workout` item.

Listing 4-13. Populating Table View Cells with Data

```

class WorkoutTableViewController: UITableViewController {
    ...
    let dateFormatter = DateFormatter()
    ...
}

```

```

override func tableView(_ tableView: UITableView,
cellForRowAt indexPath: IndexPath) ->
UITableViewCell {
    let cell = tableView.dequeueReusableCell(withIdentifier:
        "workoutCell", for: indexPath)

    guard let workouts = workouts else {
        return cell
    }

    let selectedWorkout = workouts[indexPath.row]
    let dateString = dateFormatter.string(from:
        selectedWorkout.startTime)
    let durationString =
        WorkoutDataManager.stringFromTime(timeInterval:
        selectedWorkout.duration)

    let titleText = "\(dateString) |
        \(selectedWorkout.workoutType) | \(durationString)"
    let detailText = String(format: "%.0f m | %.0f floors",
        arguments: [selectedWorkout.distance,
        selectedWorkout.flightsClimbed])

    cell.textLabel?.text = titleText
    cell.detailTextLabel?.text = detailText

    return cell
}
}

```

Now, if you re-compile the app and save a few more workouts, when you go to the History tab, the table view will display the start time, distance, total steps, and floor counts for the last ten workouts in the user's HealthKit store, similar to the screenshot in Figure 4-16.

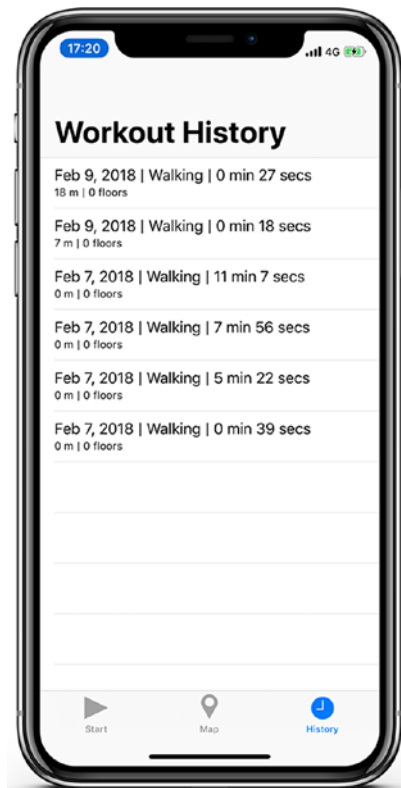


Figure 4-16. The completed History tab for the IOTFit app

Summary

In this chapter, by expanding the Workout Data Manager and adding a Table View Controller for displaying workout history, you learned how you to harness the power of HealthKit to securely import and export health data in the IOTFit app. In the first iterations of the app, you had to rely on a property-list file for data management and had no way of getting it out of the app. After applying the familiar lessons of accessing permission-based hardware resources and learning how HealthKit represents data, you were able to help users manage all of their workouts in one place and see your workout data from other apps they may also use.

Although HealthKit is unique in its representations of data, the protocol and completion handler-based communication methods you practiced in this chapter are applied often through Apple's IoT-related frameworks. Furthermore, with small modifications, most of the code in this chapter can be reused in the Apple Watch version of this app later in the book, just as with your Core Motion code from the previous chapter.

PART 2

Building Your Own Internet Things

CHAPTER 5

Building Arduino-Based Peripherals

One of the greatest forces in empowering the Internet of Things (IoT) as you know it today has been the expansion of access to lower-cost, higher-quality electronics components and communications standards that were once only available to large companies. Previously, if you wanted to learn how to program a microprocessor, you would have to buy a high-performance computer, chip programmer, blank chips, and an expensive software license from the chip manufacturer. Today, you can program all the hardware projects you want with a single \$30 Arduino, Raspberry Pi, or BeagleBone microcontroller board, and you can find a wide variety of sensors with two- or three-pin connections built in for less than \$10.

In a similar vein, Bluetooth used to be a technology that was only available by using limited-purpose hardware chips and programming the interface for the protocol yourself. Today, Bluetooth chips are built in to most connected devices and are available through many open source implementations of the protocol. As part of iOS, Apple provides a framework, Core Bluetooth, that allows you to develop apps that communicate with the Bluetooth accessory or make the iPhone act as a Bluetooth accessory.

Just as the Apple II opened up personal computers and programming to people who couldn't put together their own computer, these low-cost programming environments are opening up hardware development to new waves of students, hobbyists, and startups. In this chapter, you will join their ranks and learn how to start making your own IoT!

Learning Objectives

This chapter starts the second section of the book, centered on building apps that interact with external hardware. Leaving Apple’s universe briefly, you will learn how you can take advantage of popular general-purpose hardware, such as Arduino and Raspberry Pi, to create your own sensors and apply industry best practices to your custom hardware. To make apps for this hardware, you will learn about iOS frameworks that allow you to communicate over common protocols, such as HTTPS and Bluetooth.

In the same manner as the first section of the book, in this section, you will build a smart home management app, IOTHome, and continue to iterate on it with each successive chapter. In this chapter, you will learn how to put together and write the software for an Arduino-based wireless door sensor. In the next chapter, you will learn how to make it communicate with an iOS app via Bluetooth LE.

Unlike Raspberry Pi or BeagleBone, which are designed and produced by single entities, the Arduino project takes a more hands-off approach to its hardware designs by open sourcing their base designs and programming environment (IDE), and encouraging others to iterate on them. The Adafruit HUZZAH32, pictured in Figure 5-1, is one of those iterations, based on the Espressif ESP32 system on a chip (SoC), which adds more interface pins, Bluetooth, Wi-Fi, and a built-in battery plug/charger to the original Arduino design, while still retaining compatibility with the common programming environment.



Figure 5-1. *Adafruit HUZZAH32 microcontroller*

In the process of building the door-sensing hardware and its control software, you will learn the following key skills for IoT app development:

- Building a simple circuit for detecting input from a switch
- Setting up the Arduino programming environment
- Writing a C++-based Arduino “solution” (program)
- Monitoring and controlling devices using general purpose input/output (GPIO) and analog-to-digital conversion (ADC)

As this book is targeted at software engineers and enthusiasts, the electronics sections of this chapter are presented in a very straightforward, practical manner. My intention is to give hardware newbies enough of a taste of the work to explore it further, while not making it too boring for readers with a bit more experience. Furthermore, the project does not require any soldering, and all of the parts required can easily be ordered from Amazon.com, Adafruit, or Mouser Electronics.

The C++ section of this chapter will piggyback off of concepts you are familiar with from Swift. Thanks to the hard work of the developers of the Arduino project, you will

notice that the C++ syntax used by the Arduino IDE is very modern and should, once again, look very similar to what you are familiar with from Swift.

Building the Wireless Door-Sensor Hardware

Now that you have a better understanding of Bluetooth, you can start building the peripheral for IOTHome, the door sensor. For this task, you will have to assemble the hardware, set up the Arduino programming environment, write the code for the Arduino program, and download it to the HUZZAH32 microcontroller. Today, more than ever, these tasks have become more streamlined and reliable, but they still require some careful attention. In this section, I will guide you through the steps I took to create the door sensor and the design considerations that helped inform my decisions.

Part List

To build the door sensor for this chapter, you will have to acquire the parts listed in Table 5-1. The major components are the Adafruit HUZZAH32 microcontroller, which will run the control software; a breadboard, to assemble the circuit on; a magnetic switch, to detect if the door is closed or opened; two light emitting diodes (LEDs), to visually indicate the status of the system; and some resistors, to regulate voltage.

Table 5-1. Part List for Door Sensor

Part Name	Quantity	Mouser Part #
Adafruit HUZZAH32 microcontroller (pre-soldered)	1	485-3591
Solderless breadboard	1	589-TW-E40-1020
Breadboard jumper wire kit	1	424-WIRE-KIT
3.7V 190mAh lithium polymer battery	1	932-MIKROE-2759
Magnetic proximity aensor	1	934-59140-1-S-03-F
10K Ω resistor	1	603-FMP200JR-52-10K
200 Ω resistor	2	603-CFR-12JB-52-200R
Blue LED (through hole, 3.4V, 5mm package)	1	941-C5SMFBJECR0U045
Red LED (through hole, 2.1V, 5mm package)	1	941-C5SMFRJFCT0W0BB2

Adafruit offers three varieties of the HUZZAH32: an unassembled version, which requires you to solder a header or parts directly to its pins, in order to use it; an assembled version with a header already soldered to the pins, allowing you to plug the microcontroller into a breadboard; and an assembled version with stacking headers attached, allowing you to plug the microcontroller into a motherboard and use jumper wires to attach the pins. For this project, I recommend the latter two versions, as they will save you considerable amounts of time and frustration, in exchange for a \$1 or \$2 price difference.

For the remaining parts, you will notice that I did not mention a brand name or place strict restrictions on the parts. My goal in designing this project was to create something I felt you could create with spare parts from other projects or the contents of an inexpensive Arduino Starter Kit you may have seen on Amazon.com or at a local electronics parts store.

All of the parts in this project were specifically selected to be easy to purchase on Amazon.com, Adafruit (www.adafruit.com), or Mouser Electronics (www.mouser.com). Comparing the three retailers, Adafruit is the most accessible to hobbyists, providing a curated collection of reliable parts and a wealth of tutorials and sample projects to accompany them. Mouser is the best choice for professionals and those looking to order specialty parts or large quantities of items. Finally, Amazon is the retailer of all things. Searching for parts on Amazon is significantly harder than the other two, and it is harder to guarantee the quality of what you are getting (many small manufacturers are able to sell directly through Amazon), but it can be a good resource to get popular parts and generic, lower-priced parts as part of the same order.

Your completed set of parts for the project should resemble that in the photograph in Figure 5-2.

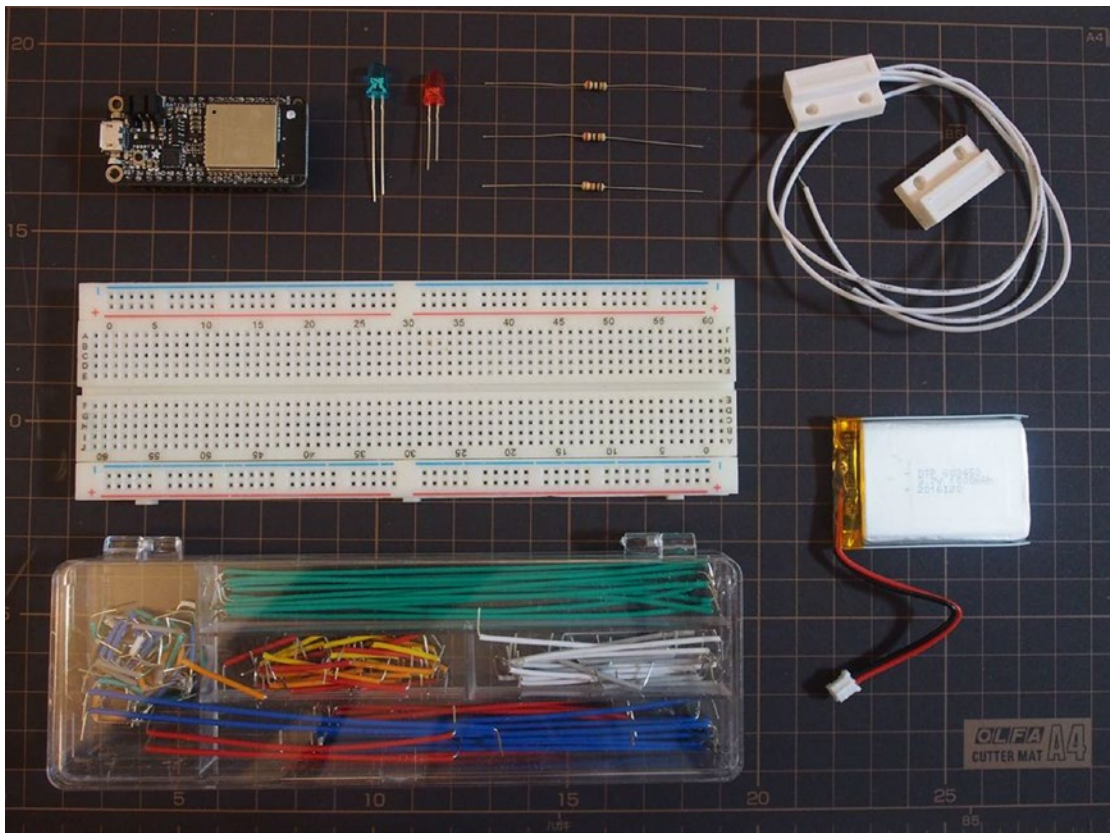


Figure 5-2. Photograph of collected parts for the IOTHome project

Assembling the Hardware

My first boss told me that one of the greatest things about working at NASA was that all of the engineers had to take NASA’s soldering training classes. That being said, soldering is an exercise that requires practice and precision. When working with a sensitive part, such as a microcontroller, you must be careful to not apply too much solder (the easy-to-melt metal that forms the bond between parts), or you will risk creating a short circuit (an unintentional physical connection between two parts that should not touch). You must also be careful not to apply the soldering iron (the tool that heats up the solder) for too long or at too hot a temperature, or you will risk the chance of physically burning out the part.

To alleviate both of these risks, many engineers use breadboards (like the one shown in Figure 5-3), to create connections between parts using jumper wires. This is particularly useful during the prototyping stage, when you are constantly experimenting with different parts and changing the circuit. To more accurately reflect the real world, you, too, will use a breadboard to assemble the hardware for this project. If you feel comfortable soldering and want to transfer the design to a more compact or stable medium, you are welcome to do so.

Although breadboards come in various shapes and sizes, there are a few common characteristics shared by all of them.

- The case is frequently made of solid plastic, with several metal peg holes.
- Frequently, there are rows at the top and bottom, with markings for shared positive and negative connections.
- Most of the physical space on the breadboard will be occupied by closely spaced columns of pins, running across the length of the device.
- There is usually a gap in the center row of the breadboard.

As with all things, there are reasons for all of these characteristics. First, plastic is an insulator, meaning it does not conduct electricity (form a path for electricity to travel). You can handle the plastic area freely, without concern of it electrocuting you. The rows on the top and bottom are shared connections. Every part or wire you plug into a pin on these rows will share an electrical connection with every other pin on these rows. Most electrical parts run on DC voltage, which requires a source (positive) connection and ground (negative) connection, like the connections coming from a battery, so it makes sense that these pins are the ones that will be shared the most in a circuit. For smaller shared connections, you will use the columns in the center of the breadboard. All pins in a column are shared. Because most integrated circuits (ICs) come in a case that follows the dual in-line package (DIP) standard, the gap in the center of the breadboard allows you enough space to plug an IC and use a single column for each pin. In this manner, you can connect other parts to the pins without soldering. To help make this easier to visualize, I have included an annotated photograph of a breadboard in Figure 5-3, indicating the shared connections.

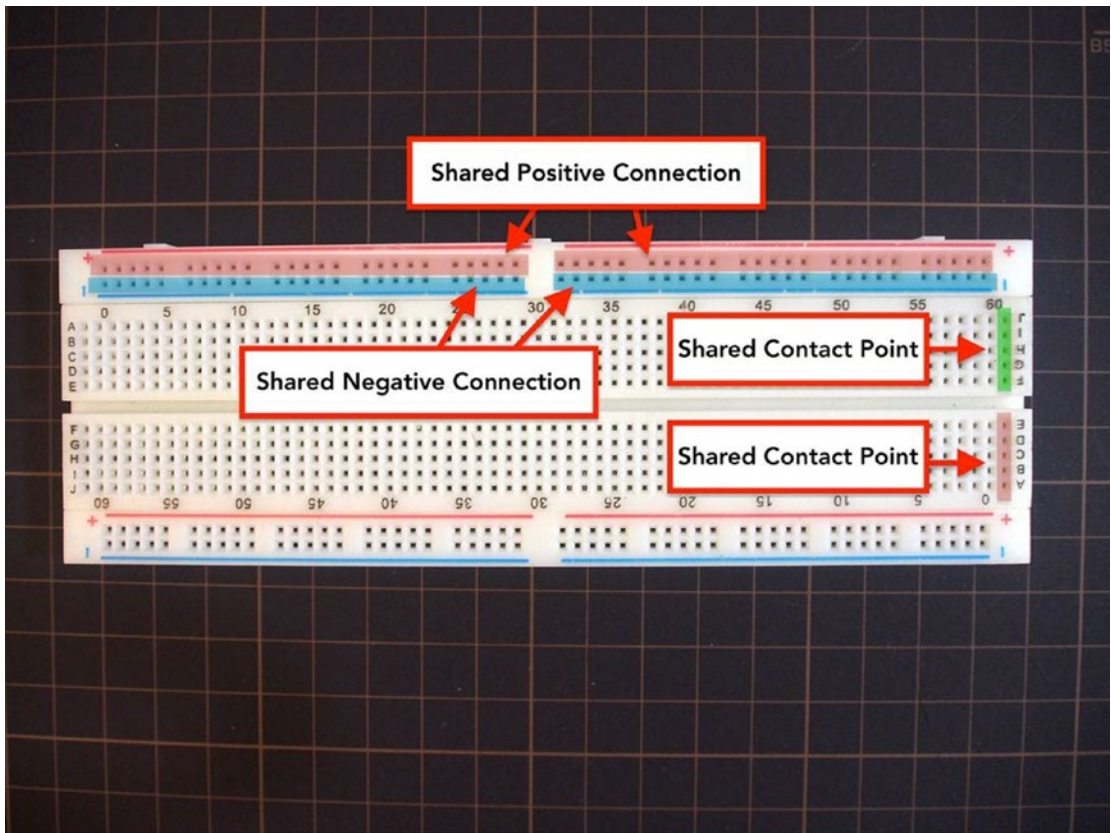


Figure 5-3. Physical connections on a breadboard

To begin assembling the project, take your HUZAZH32 and place it on the center-left side of the breadboard. As shown in Figure 5-4, make sure you place the HUZAZH32 on the breadboard with the USB port sticking out. This will make it easier to insert a USB cable for programming and give you more space on the breadboard to insert parts.

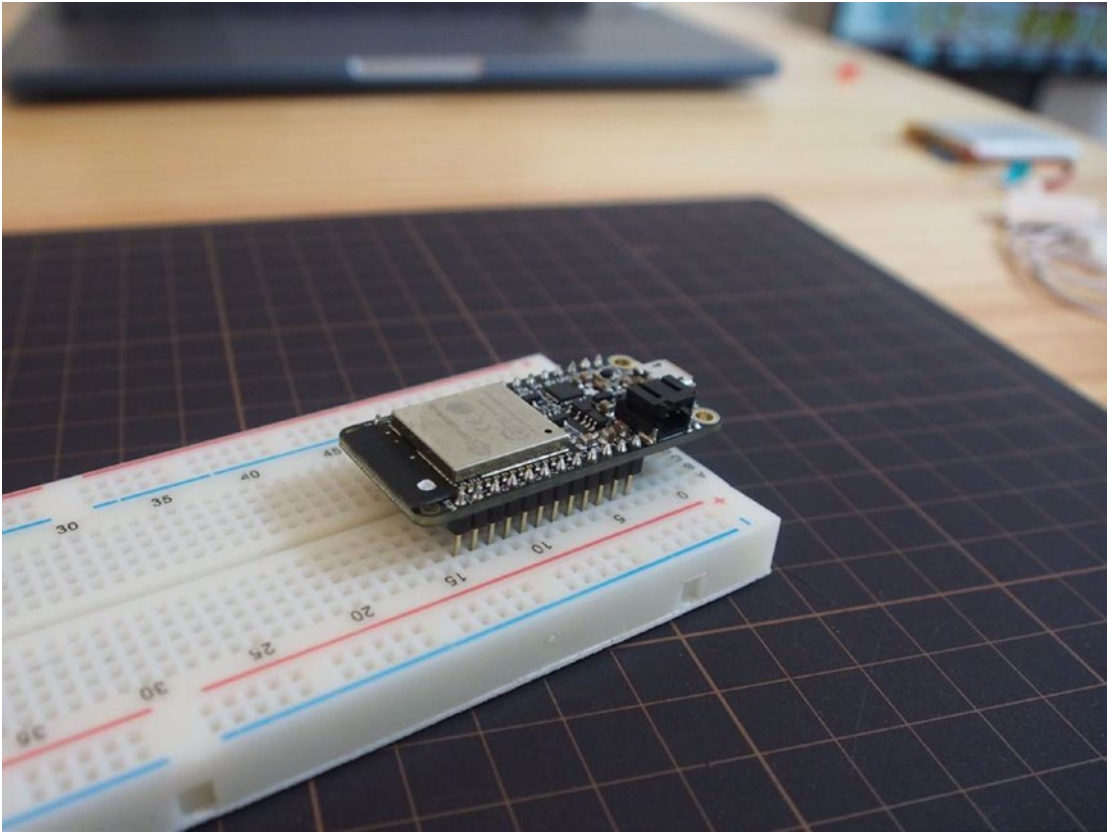


Figure 5-4. Breadboard with the HUZAZH32 chip placed on top

To prevent bending or breaking the header pins, gently push down on the right side of the chip until the pins start to feel a little secure in the breadboard, as shown in Figure 5-5.

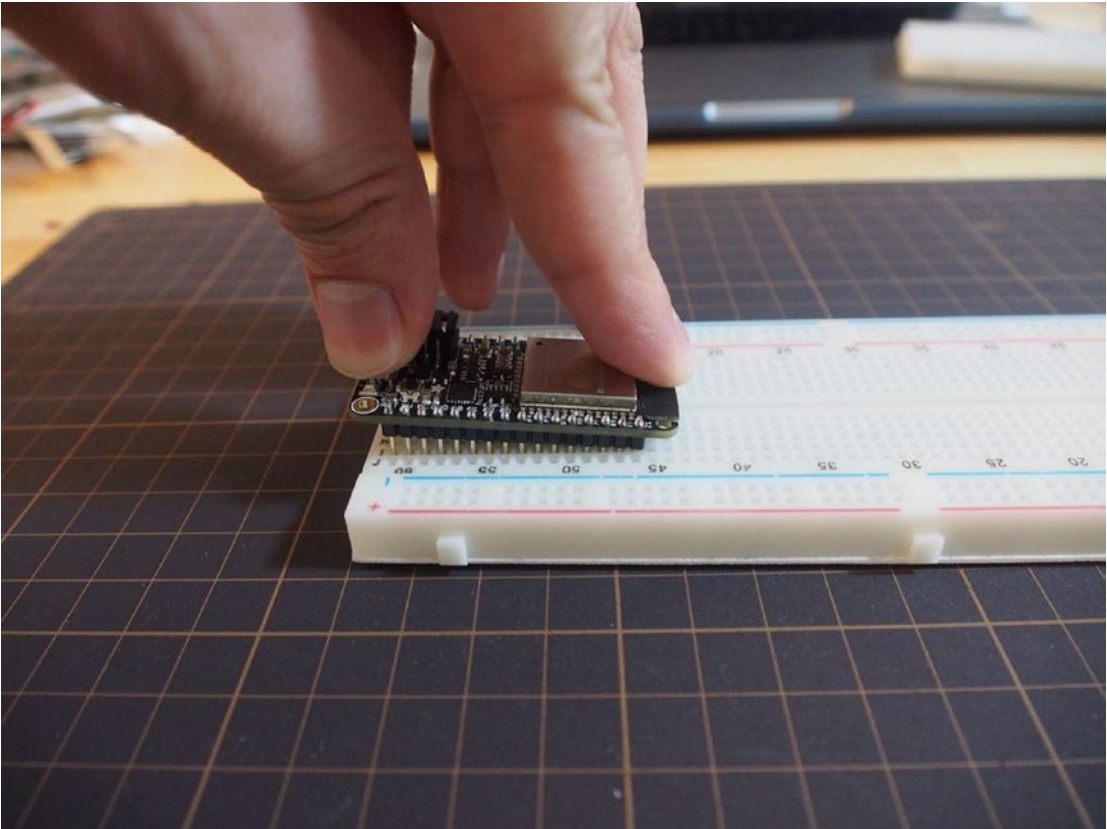


Figure 5-5. *Securing the HUZAZH32 chip onto the breadboard*

Continue this process by pushing down on the left side of the chip, then alternate pushes between the left and right sides, until the bottom of the header is flush with the breadboard, as shown in Figure 5-6.

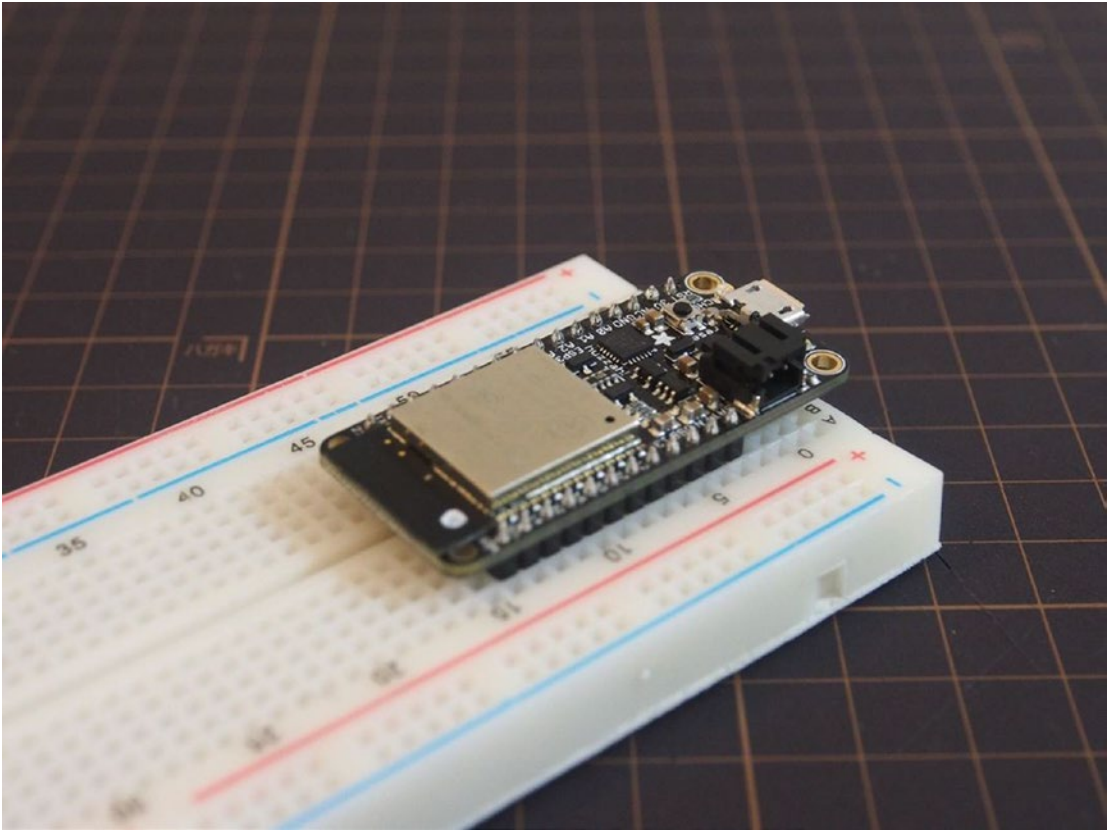


Figure 5-6. Fully secured HUZAH32 chip

You will use this same process of plugging in parts and connecting them with header wires to complete the circuit for this project. Before going further, take a second to review the schematic in Figure 5-7. A schematic is a design diagram that specifies which parts are used in a circuit and how the connections are made between these parts. All schematics are drawn using standardized symbols for the parts and connections, so any engineer can quickly look at the schematic and understand how to build or analyze the circuit.

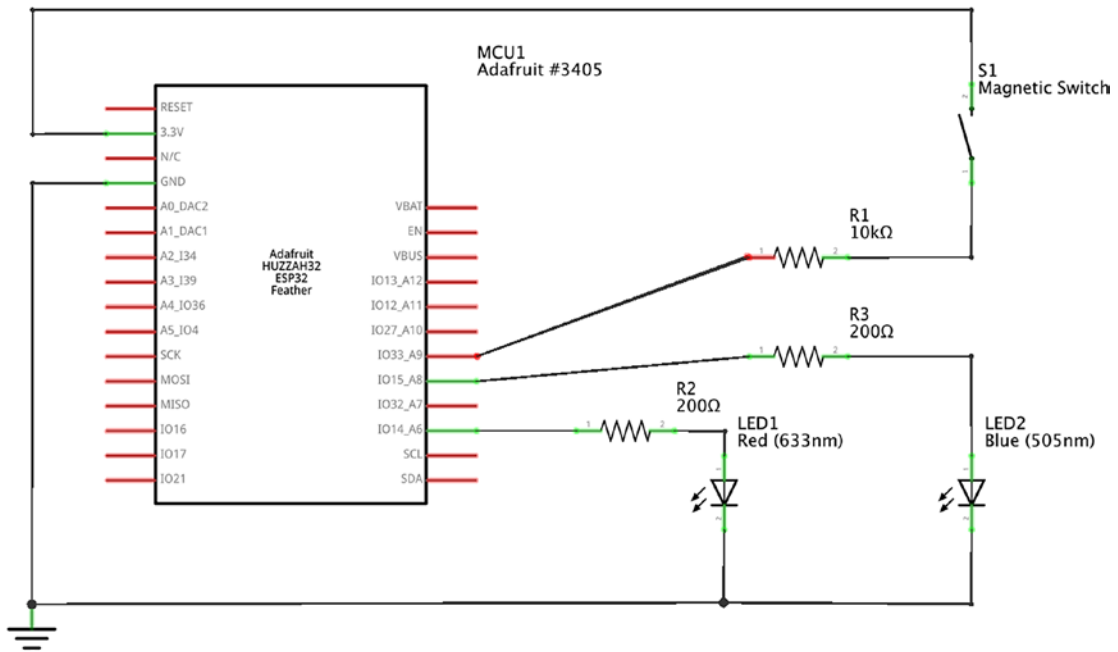


Figure 5-7. Schematic diagram for the door sensor

For the door sensor, the heart of the circuit is the HUZAZH32 chip. Its 3V and GND pins are used to provide the power and ground connections for the rest of the circuit. Connected to the GPIO14 and GPIO15 pins are LEDs, which will be used to indicate if the door is locked and if the Bluetooth connection has been established. Many parts, especially LEDs, have specific requirements on the voltage and amperage that are required to drive (operate) them. The resistors placed in between the pins and LEDs are used to reduce the voltage to a level that allows the LEDs to operate without burning out from voltage levels higher than what they were designed for. The final connection in the circuit is for the switch. One side of the switch is connected to the 3V pin and power connection, and the other side is attached to a resistor placed between the other side of the switch and the ground connection. This is called a *pull-down resistor* and is used to establish a stable ground connection when the switch is opened. Without the pull-down resistor, the microcontroller is unable to reliably read if the pin is *high* (connected to power) or *low* (connected to ground), and you cannot tell if the switch is open or closed.

First, start by establishing the power and ground connections for the circuit. Take out red and green header wires. Find the pin marked 3V on the HUZAZH32 chip. Insert one side of the red wire (red, to represent power) into the row for this pin, and the other side into the column marked “positive” on the breadboard. Repeat this process for the GND

pin, except use a green pin (to represent ground) and terminate the connection at the “negative” column. The breadboard with the HUZAZH32 chip and power/ground wires attached is pictured in Figure 5-8. When the HUZAZH32 is powered through the USB cable or a battery, these two connections will provide power to the rest of the circuit.

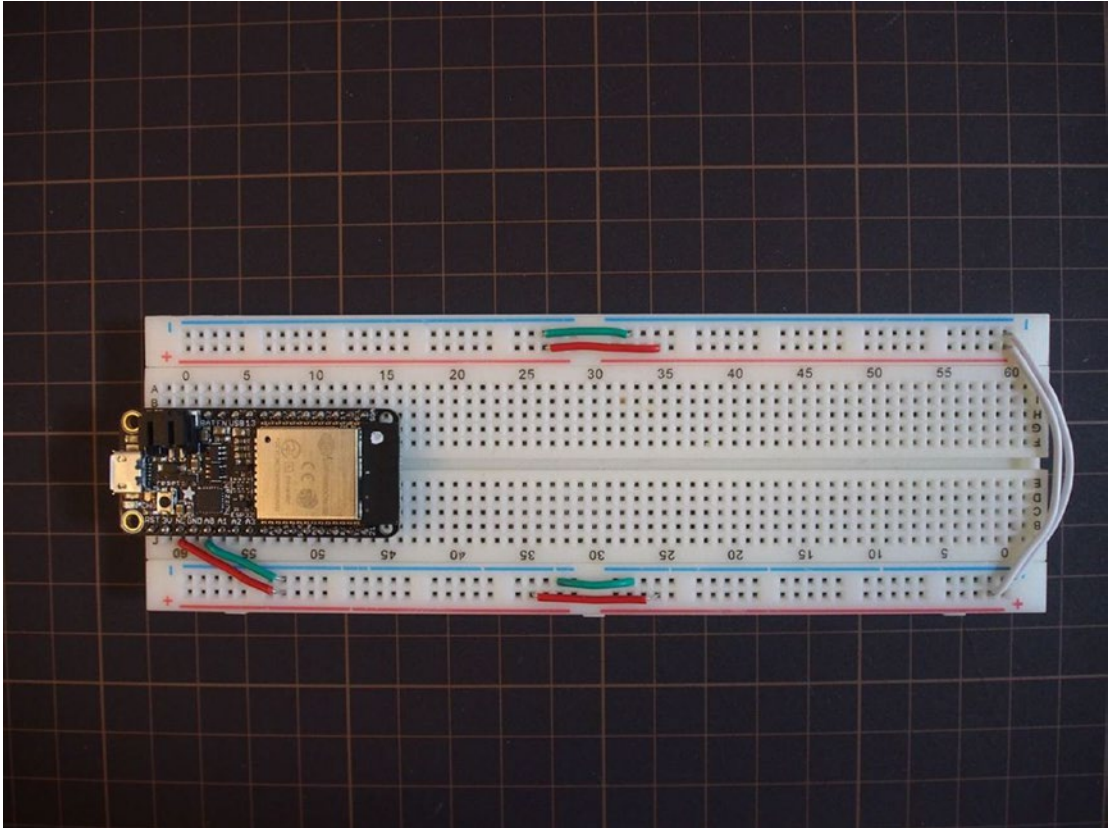


Figure 5-8. Breadboard with HUZAZH32 chip and power/ground connections

Now you are ready to start assembling the real guts of the circuit. First, begin by snipping off the extra ends of the resistors with a pair of wire cutters. Leave approximately 10mm (0.4 inches) remaining, so the resistors are easy to insert and remove from the breadboard. You will also need to snip the LEDs leads down to size. As indicated by their schematic symbol, LEDs are *polarized*, meaning they only allow current to flow in one direction—from the *anode* (positive) to *cathode* (negative). To indicate which pin is the anode, manufacturers ship LEDs with the anode pin intentionally cut

longer than the cathode. As shown in Figure 5-9, use wire cutters to make a diagonal cut across the pins of the resistor, paying attention to leave the anode as the longer pin.

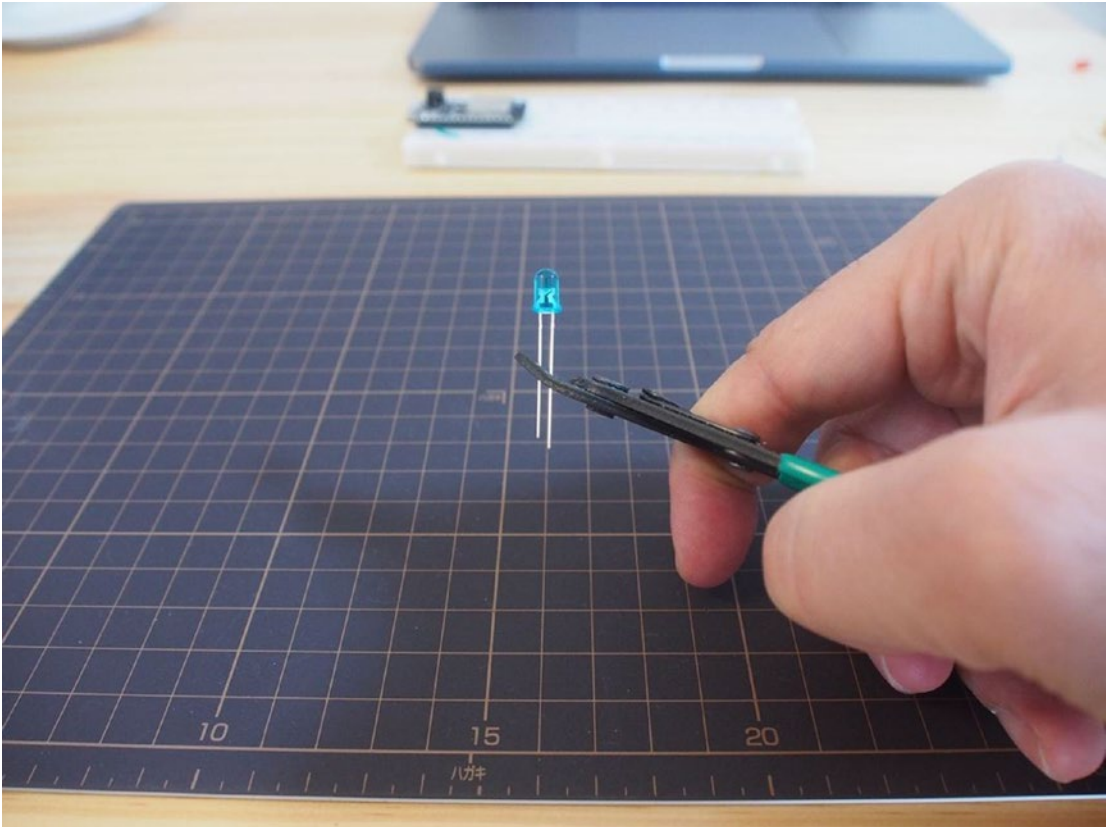


Figure 5-9. *Cutting the pins on an LED*

Now that the parts are ready, you can begin inserting them into the breadboard. First, establish the circuit for the red LED. Start by taking a gray, blue, or white header wire (any color that is not red or green will do) and run it from GPIO14 to an empty area of the breadboard. Then, plug a 200 Ω resistor into the breadboard, vertically across the empty gap in the center, connecting two columns. Resistors are not polar, so the direction does not matter. Then, insert the red LED into the breadboard horizontally, with the anode (long, positive terminal) pin inserted in the same column as the end of the resistor (bottom column), and the cathode (short, negative terminal) inserted into an empty column. To finish things off, take a green header wire and connect it between the cathode's column and the ground row. This process establishes the connection from the

GPIO pin to the resistor, into the LED, and eventually terminating at the ground. At this point, your breadboard should resemble the photograph in Figure 5-10.

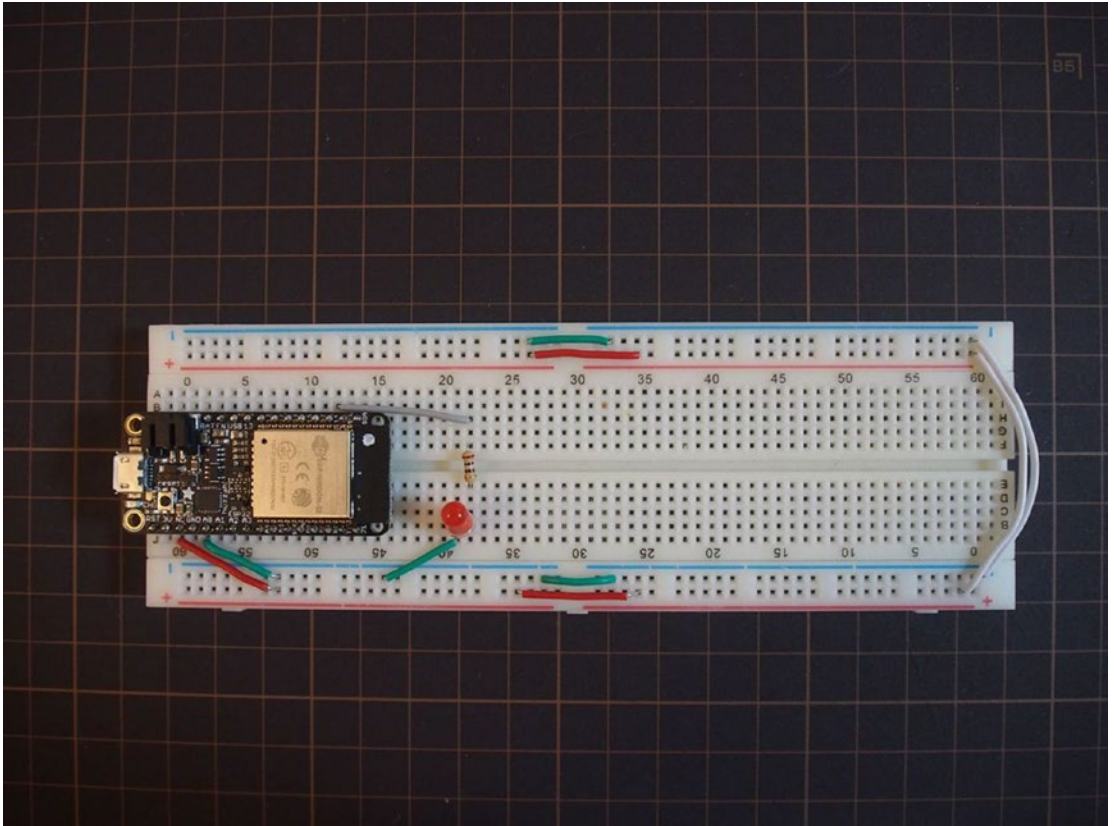


Figure 5-10. Breadboard with red LED and supporting parts attached

Follow these same steps to attach the blue LED, except start at the pin marked GPIO15, and make sure all the connections are using empty columns.

Note Engineers will usually color-code connections on a breadboard to help them identify the purpose of the header wire. This is especially important when dealing with the power pins. If you have to reuse colors, try to place them in different physical areas of the breadboard, to make debugging easier.

The final part to connect at this point is the magnetic switch. Once again, start by taking a header wire and extending it from the pin marked GPIO33 to an empty column. Next, attach the 10KΩ resistor across the column gap. Finally, attach one end of the switch to the terminating end of the resistor, and the other to the shared positive connection row at the bottom of the breadboard. Similar to resistors, switches are not polar, so the direction does not matter. Your completed circuit should resemble the photograph in Figure 5-11.

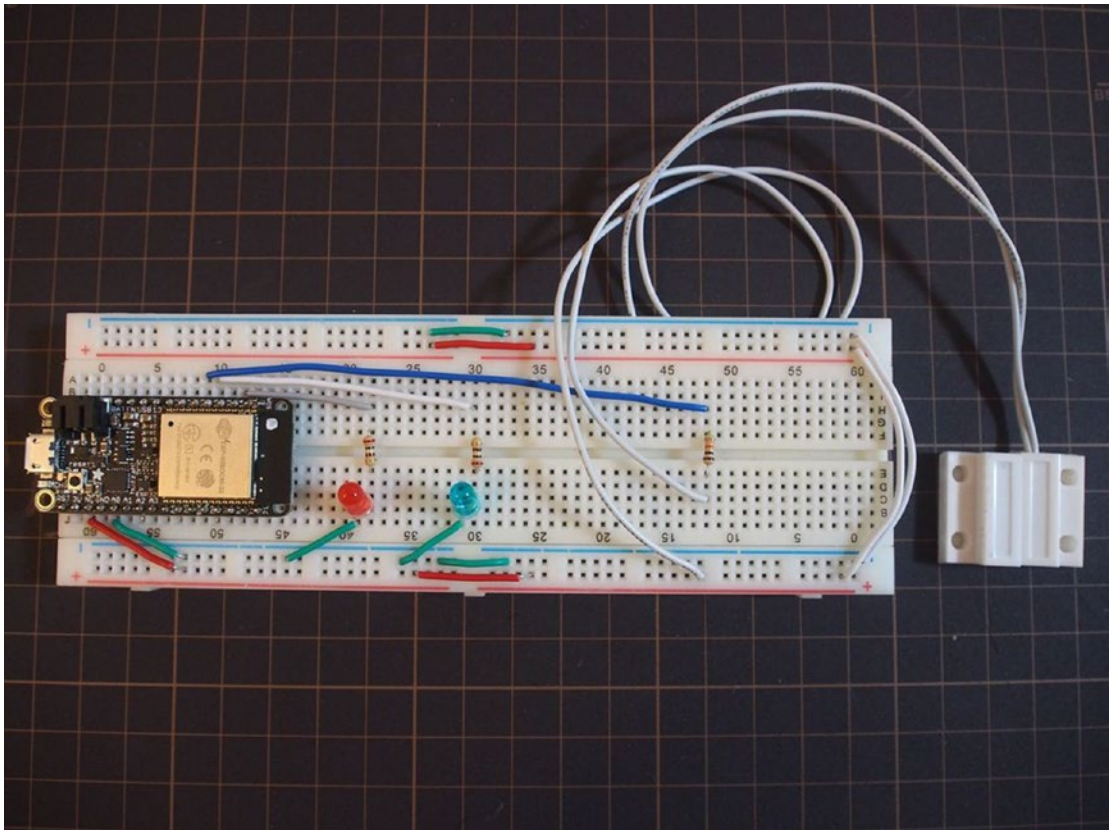


Figure 5-11. Completed circuit for the door sensor

Writing an Arduino Solution (Program)

Now that the hardware work is completed, you can start to move back into the realm of software. Before the advent of Arduino, primitive reprogrammable chips were the only option you had for developing your own embedded systems. The parts you would use in your circuits would have a fixed purpose, and you would have to pour over the instruction

sheet for the part to figure out what each pin did and if there were any other conditions required to use them (such as current limitations or dependence on another part).

With the Arduino, there is still the requirement that some pins can only be used for one purpose, but the GPIO and ADC pins are your playground. You can write programs that use them as simple inputs or outputs (GPIO) or more fine-grained, precise ones (ADC). Just as with third-party libraries in iOS, you can also download third-party libraries that control more complex parts over these pins. In the next section, you will learn how to write your own Arduino solution (program) to use these programmable pins to read the status of the magnetic proximity sensor (switch) and battery level and share the results via the LEDs and Bluetooth.

Setting Up the Arduino Programming Environment

To get started on your Arduino programming experience, you must download the free, official Arduino Integrated Development Environment software. As shown in Figure 5-12, navigate to www.arduino.cc and click the Software tab at the top. On the Software page, scroll down to Download the Arduino IDE, and click the link to download the OS X version of the Arduino IDE.

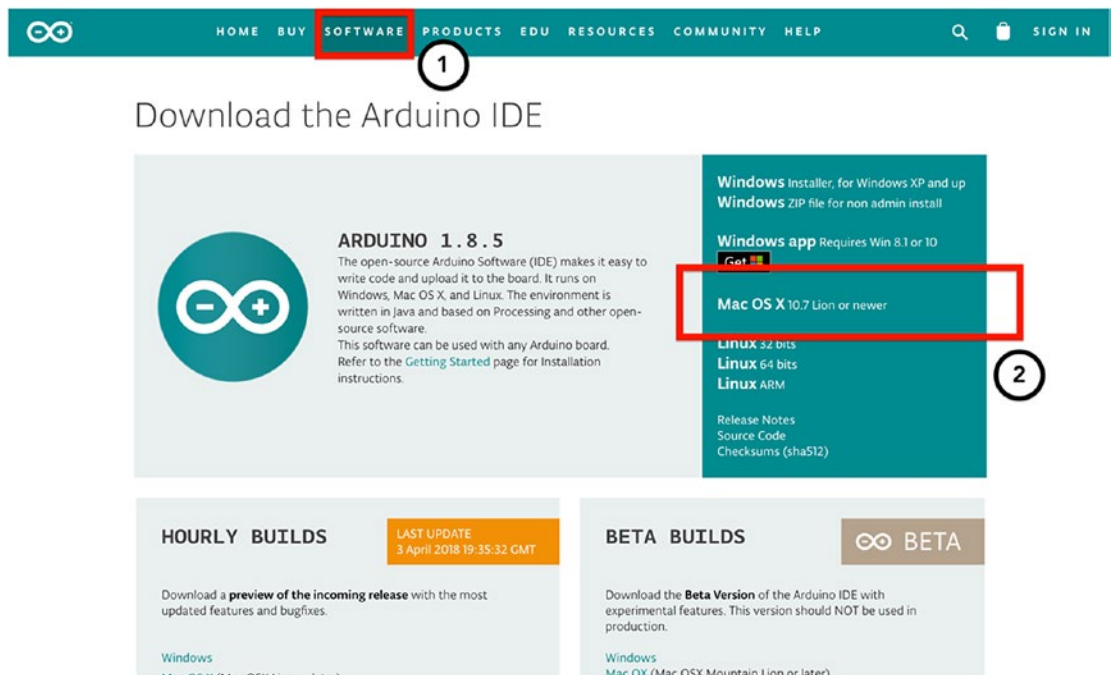


Figure 5-12. Downloading the Arduino IDE

Caution The Arduino development team does an excellent job of ensuring near-identical usability on the OS X, Windows, and Linux versions of the IDE; however, the instructions in this book require command-line instructions that were tested on the OS X and Linux versions only.

When the download is complete, double-click the zip file to extract it. Drag and drop the Arduino binary file to your Applications folder, as shown in Figure 5-13.

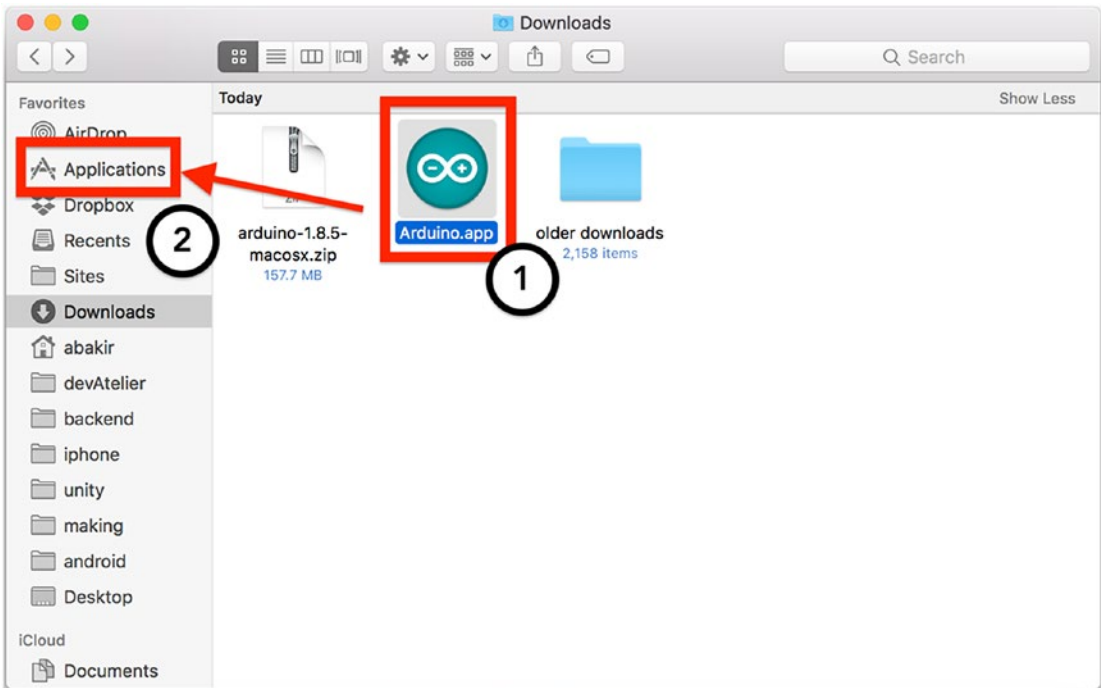


Figure 5-13. *Installing the Arduino IDE*

Next, click the Arduino icon from the Applications folder to open the IDE. You will be greeted with a text editor containing the stub functions for a new program, as shown in Figure 5-14. As with any IDE, there are compile and run buttons in the top-left corner of the screen. Compile is represented by the button with the check mark, and Run is represented by the right-pointing arrow.

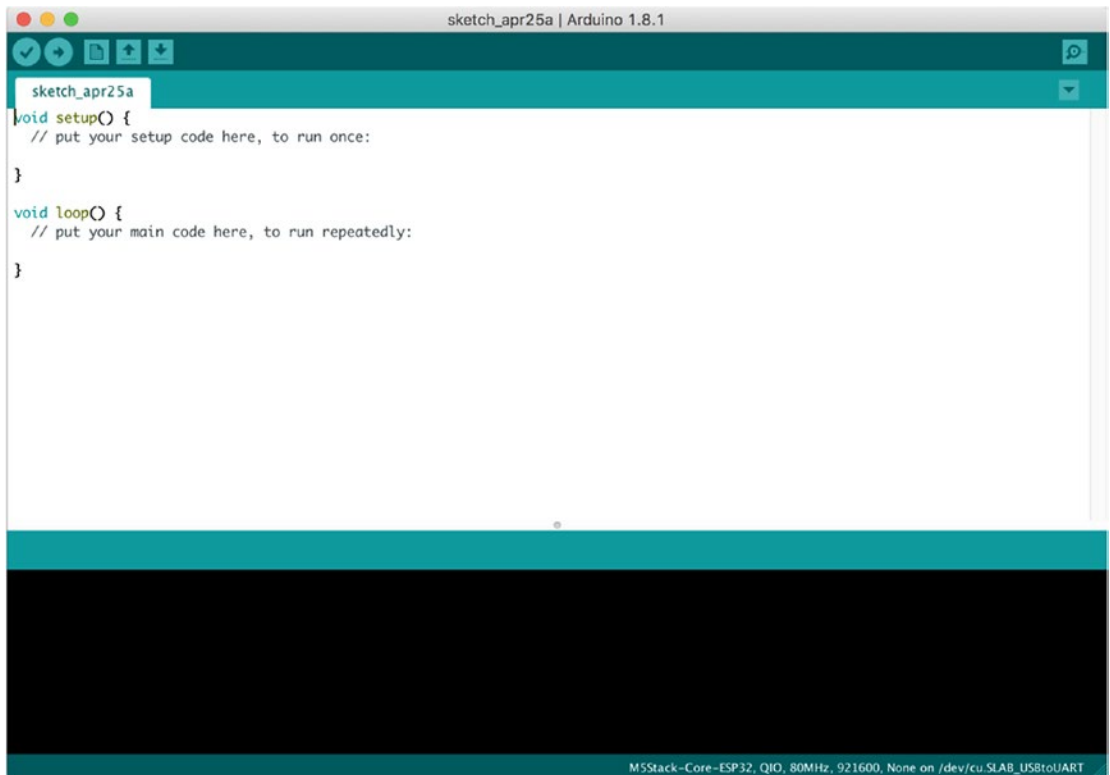


Figure 5-14. Default display for the Arduino IDE

Before you can use the HUZAZH32 with the Arduino IDE, you must run a configuration script provided by the manufacturer of the SoC it is based on, the ESP32, manufactured by Espressif. The SoC provides the core functions of Arduino compatibility and pinmapping. Manufacturers such as Adafruit can then choose what features they want to use and how to package them. To install the build scripts, you must check them out via `git`. If you have installed the Xcode command-line tools in the past, the `git` version control software will already be installed on your computer. If have not, you can install the tools by opening up the terminal and then typing in the following command:

```
xcode-select --install
```

To run the ESP32 installation script, first close the Arduino IDE, then open a terminal window. Inside the terminal window, enter the command in Listing 5-1. It will create a new directory, check out the code from `git`, and run the installation script.

Listing 5-1. Command-Line Instructions to Add ESP32 Support to the Arduino IDE

```
mkdir -p ~/arduino && \
cd ~/arduino && \
git clone https://github.com/espressif/arduino-esp32.git esp32 && \
cd esp32 && \
git submodule update --init --recursive && \
cd tools && \
python get.py
```

If the script ran successfully, the last line of the output should contain “Done!”

Now that you have added ESP32 support to the Arduino, you must select the HUZAZH32 as the target device. First, reopen the Arduino IDE, then navigate to the Tools menu. As shown in Figure 5-15, select the Board: ... menu item and scroll down until you find ESP32 Feather.

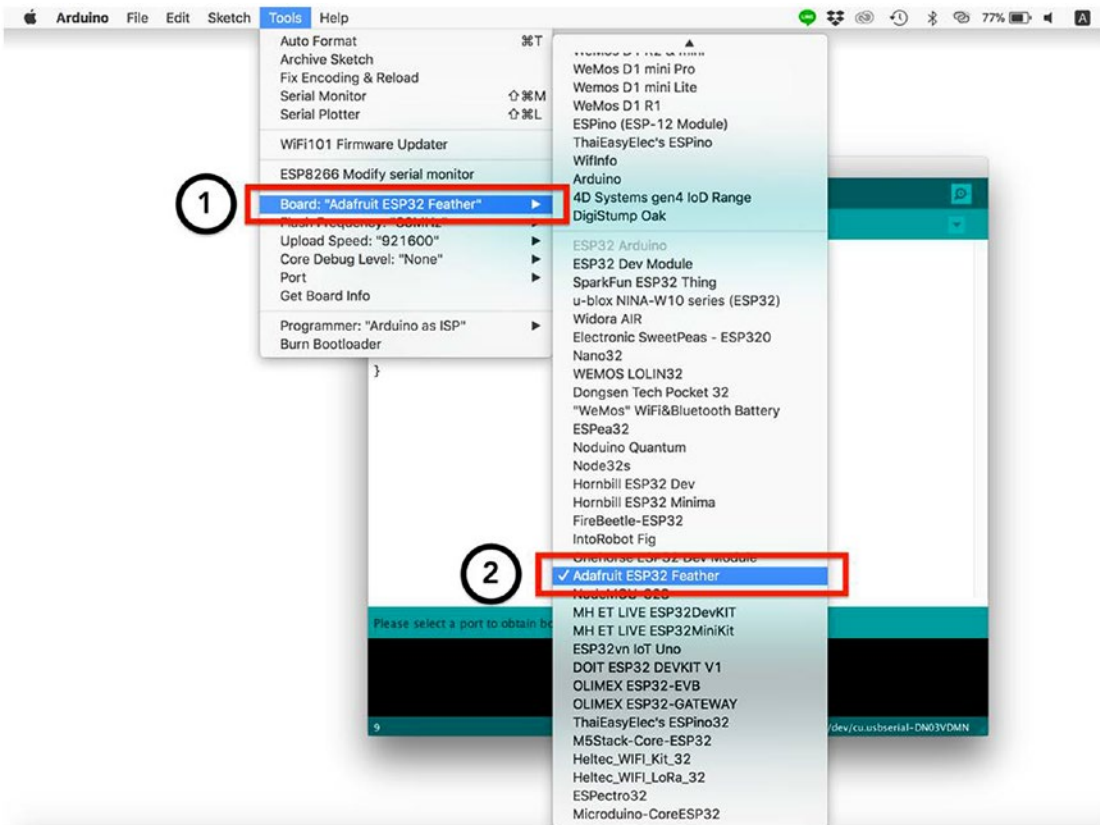


Figure 5-15. Selecting target hardware within the Arduino IDE

Next, you have to select your computer's USB port as the interface for debugging. Navigate to the Tools menu once more and then select `/dev/cu.SLAB_USBtoUART` as the port, as shown in Figure 5-16. For consistency's sake, also confirm that the Flash Frequency is set to 80Hz and that the Upload Speed is set to 921600, while you are in the Tools menu.

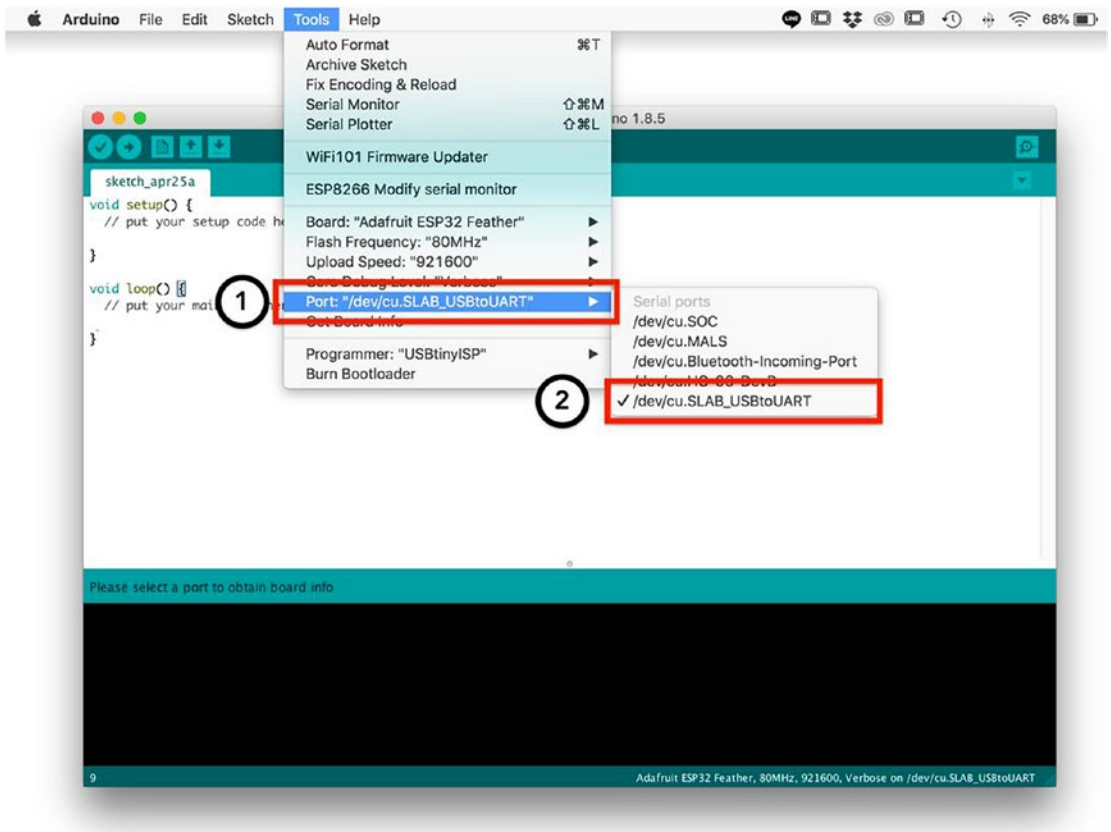


Figure 5-16. Selecting the USB port for debugging

Finally, to verify that the Arduino IDE is able to communicate with the HUZAZH32, go to the Tools menu and select Get Board Info. If the connection was established successfully, you will be presented with a dialog similar to the one in Figure 5-17, listing unique identifier values for your microcontroller.



Figure 5-17. Verifying the connection to the HUZZAH32

Using GPIO to Monitor Input Pins and Control Output Pins

Now that the Arduino IDE is set up for programming, you can finally start writing some Arduino code! The Arduino solution for the door sensor can be split into two components: monitoring the hardware and transmitting the results over Bluetooth. In this section, you will focus on the hardware-monitoring code, as you can debug that by monitoring the Arduino IDE’s debugging console and visually inspecting the LEDs in the circuit.

To begin writing a new Arduino solution, go to the File menu and select New. You will be presented with an empty solution file, containing two stub methods: `setup()` and `loop()`. As the names suggest, Arduino solutions have a simple architecture: the logic in the `setup()` method gets executed as soon the device powers on, and the logic in the `loop()` method is repeated until the device is powered off. The initial Arduino solution should resemble Listing 5-2.

Listing 5-2. Initial, Empty Arduino Solution

```
void setup() {
    // put your setup code here, to run once:
}

void loop() {
    // put your main code here, to run repeatedly:
}
```

Working with GPIO in Arduino is a straightforward operation. In the `setup()` method, specify the pin number and whether it will be used for input or output. The API for specifying this is `pinMode()`. In Listing 5-3, I have updated the solution to make these calls. To help manage the code, I have defined the pin numbers as compiler macros using the `#define` keyword. If you are familiar with Objective-C, you will recognize this keyword, as it is used there for the same purpose. The pin numbers match the ones you used to put together the circuit in the “Assembling the Hardware” section.

Listing 5-3. Setting Up GPIO Pin Modes in an Arduino Solution

```
#define RED_LED_PIN 14
#define BLUE_LED_PIN 15
#define SWITCH_PIN 32

void setup() {
    pinMode(RED_LED_PIN, OUTPUT);
    pinMode(BLUE_LED_PIN, OUTPUT);
    pinMode(SWITCH_PIN, INPUT);
}

void loop() {
    // put your main code here, to run repeatedly:
}
```

In Listing 5-3, I set the `BLUE_LED` and `RED_LED` pins as outputs, and the `DOOR_SENSOR` pin as an input. Now, it is time to take advantage of them. To interact with GPIO pins on the Arduino, you use the `DigitalRead()` and `DigitalWrite()` APIs. As the names suggest, they are used to read or write a value to a digital pin. For this introductory part

of the program, you will turn the RED_LED on or off, based on whether the magnetic sensors are touching each other (door closed) or not (door open). Because I designed the circuit with a pull-down resistor between GPIO pin and ground, the switch will read OFF while the door is open and ON when the door is closed. In Listing 5-4, I have updated the loop() method to include this logic. To read the value of the magnetic switch more accurately, I have added a 1 second (1000 millisecond) delay to the end of the loop() method, using the delay() API.

Listing 5-4. Turning an LED On or Off, Based on GPIO Status

```
#define RED_LED_PIN 14
#define BLUE_LED_PIN 15
#define SWITCH_PIN 32

void setup() {
  pinMode(RED_LED_PIN, OUTPUT);
  pinMode(BLUE_LED_PIN, OUTPUT);
  pinMode(SWITCH_PIN, INPUT);
  Serial.begin(9600);
  Serial.println(" Program start");
}

void loop() {
  checkSwitch();
  delay(1000);
}

void checkSwitch() {
  int currentState = digitalRead(SWITCH_PIN);
  if (currentState != switchState) {
    updateLockBLE(currentState);
  }
  switchState = currentState;
  digitalWrite(LED_PIN, switchState);
  Serial.print("Switch state: ");
  Serial.println(switchState);
}
```

In the next chapter, you will use the Blue LED to indicate whether the Bluetooth connection has been established.

Calculating Battery Status

One of the most convenient aspects of the HUZAZH32 is that its built-in battery port allows you to power it from an inexpensive lithium polymer (LiPo) battery and recharge it when the HUZAZH32 is powered via USB. However, to help the user get the most out of the device, it would be wise to transmit the battery's charge level through the companion app. Luckily, we can use the power of ADC to read the battery level and transmit it over Bluetooth.

Just like software developers, hardware developers often finding themselves adapting their designs to fit the limitations they are given. In the case of the HUZAZH32, its designers were able to fit more functionality in the device, by allowing some pins to have multiple functions. To read battery level, you can take advantage of their engineering ingenuity on Pin 35 (ADC #13). When nothing is plugged into this pin, you can use it to read exactly half of the voltage level of the connected battery. ADC pins on the HUZAZH32 work just like RGB values in iOS code, in that you can read or write 12-bit levels from them. This allows you to fine-tune the speed of a motor or read a non-binary value from a pin, as we have to in this example. To calculate the battery level accurately, divide the ADC output value by 4095 ($2^{12} - 1$), to find what percent the battery level is at (out of the maximum), double this by 2, then multiply it by 1.1 (the reference voltage of the ADC pin), and 3.3 (the reference voltage of the HUZAZH32's output). If you want to report the battery level as a percentage, divide this number by 4.2, the maximum possible voltage of a LiPo battery that can be connected to the chip. In Listing 5-5, I have modified the Arduino solution to include these calculations. For now, I display the output value on the Arduino IDE console. In the next chapter, you will learn how to transmit it across Bluetooth LE.

Listing 5-5. Reading the Battery Level from ADC Pin 13

```
#define RED_LED_PIN 14
#define BLUE_LED_PIN 15
#define SWITCH_PIN 32
#define BATTERY_PIN 35
```

```

void setup() {
  pinMode(RED_LED_PIN, OUTPUT);
  pinMode(BLUE_LED_PIN, OUTPUT);
  pinMode(SWITCH_PIN, INPUT);
  pinMode(BATTERY_PIN, INPUT);

  Serial.begin(9600);
  Serial.println(" Program start");
}

void loop() {
  checkSwitch();
  checkBattery();
  delay(1000);
}

...

void checkBattery() {
  float currentLevel = analogRead(BATTERY_PIN);
  currentLevel = ((currentLevel / 4095) * 2 * 3.3 * 1.1)
    * 100 / 4.3;
  batteryLevel = currentLevel;
  Serial.print("Battery Level: ");
  Serial.print(t);
  Serial.println("%");
}

```

Running and Monitoring the Arduino Solution

Now that the Arduino hardware and its control software are complete, the only step left is to try to run the program and see if the sensor is performing its functions correctly: turning on a switch when the magnetic sensor is open and regularly reporting the voltage level of the connected battery.

When you were writing your Arduino solution, you may have found yourself constantly saving the program. If you were adventurous, you may have tried to compile it a few times, to see if the code was valid. To fully take advantage of the Arduino IDE, refer to Figure 5-18, for a more detailed description of its user interface.

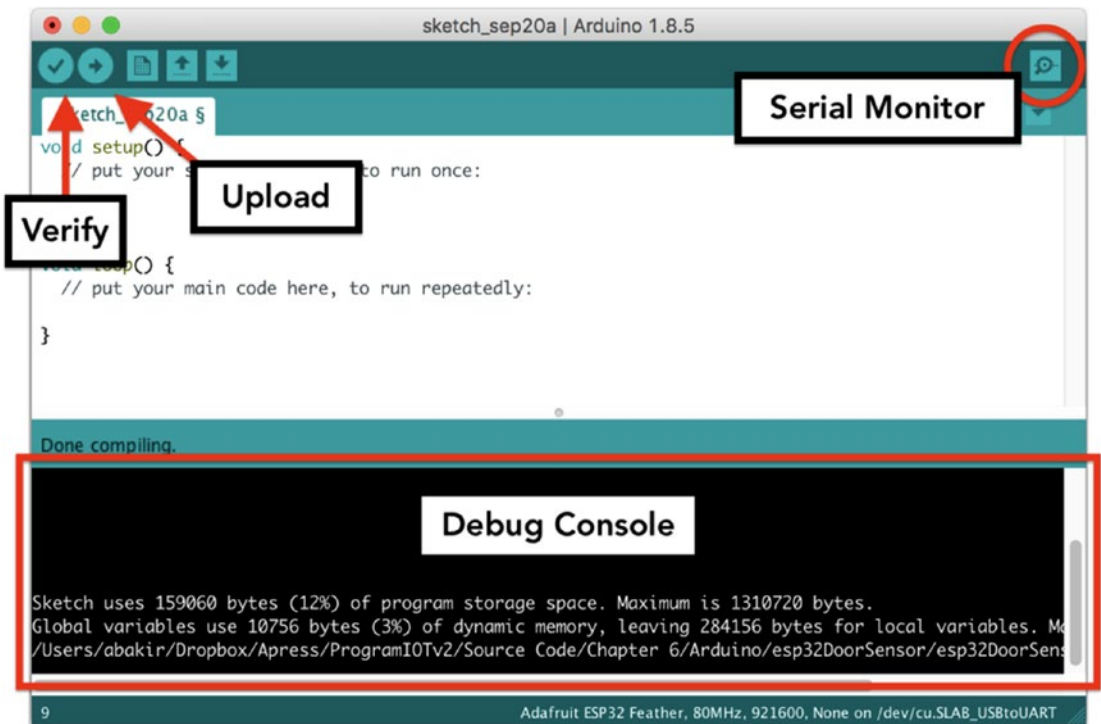


Figure 5-18. Identifying the Run button and Serial Monitor in the Arduino IDE

Unlike Xcode, you will notice that the Arduino has two Run buttons—Verify and Upload. The Verify button performs compilation only, operating like the Build command in Xcode, allowing you to check if your code compiles and fulfills basic runtime requirements, such as fitting within the available storage on the target device. The Upload button compiles the solution and uploads it to the target device’s hardware. Because the HUZAZH32 is an Arduino-compatible device, the scripts you installed earlier are used in this step to complete the upload to the device. At the far right of the top bar, you will find the Serial Monitor button. This adds output from the UART port to the Debug Console pane at the bottom of the IDE, if it is not already present. When new messages are ready from the compiler or device, they will appear in this pane.

To begin debugging the application, click the Upload button. In the serial console, you should see a series of messages indicating the upload progress. After the upload has completed successfully, you should see the switch and battery status being printed out once per second (the duration of the delay in the loop() method). You should also be able to turn on the red LED by opening the magnetic switch. The Arduino solutions run continuously after they are uploaded, so you do not have to re-upload the solution every

time you want to run it. Simply power the device, and the `start()` and `loop()` methods will start executing immediately.

TROUBLESHOOTING COMPILATION

Before I was an iOS developer, I worked in embedded systems. The most frustrating and time-consuming part of the process was always trying to get the program to run for the first time. Following, I have included some of most common problems I encountered when I was trying to run solutions on the Arduino devices.

The circuit is working correctly, but the serial console is showing junk characters.

Serial communication is only able to work when the sender and receiver are communicating at the same frequency (literally!). To resolve this issue, verify that the serial monitor is set to the same value you used to configure the serial Port in the Arduino solution. For the door monitor, you can find it in the serial Port initialization code:

```
Serial.begin(9600);
```

The Arduino IDE is uploading the solution successfully, but I am unable to see serial messages or get the switch to work.

This problem is frustrating because it prevents you from verifying the behavior of the program. In my experience, there are two common causes for this error:

1. The device needs to be reset. This is a simple operation to perform. Simply press the Reset button on the HUZAZH32 chip while it is still connected to power. When the device restarts, you should see the correct output in the serial monitor and on the circuit.
2. You must hold down the Reset button while uploading the solution to the HUZAZH32. On older ESP32-based boards, Espressif requires the Reset signal to be held down to a negative value during programming. If the hardware is stuck in a state in which you cannot reprogram the device, or the program is not running, try holding down the Reset button of the device during the entire duration of the Upload operation. This should re-enable programming the chip.

The Arduino IDE is unable to find the serial port for the HUZZAH32.

This is the most frustrating issue to track down. The HUZZAH32 may be working during one programming session and suddenly become unresponsive. In my experience, this issue is often caused by the OS X USB-to-Serial driver for the HUZZAH32, rather than your hardware itself. As the vast majority of modern computers no longer ship with Serial (RS-232) ports, device manufacturers use an integrated circuit to transmit serial signals over USB. Because the community of users who need this functionality is small compared to general users, the drivers are often unstable. To fix this problem, I recommend rebooting your Mac and then trying to plug the HUZZAH32 into your computer again, after the reboot completes.

Much like debugging connected devices on Xcode, once you are able to identify the symptoms of a connection issue, it becomes a trivial matter to resolve it.

Summary

In this chapter, you learned how to make your first sensor using an Arduino. On the hardware side, you learned how a breadboard can help you connect components without soldering. With the exception of pull-down resistors, which are used to stabilize the signal from the switch, most of putting the circuit together is connecting dots from the schematic to the breadboard. On the software side, you learned that the Arduino IDE and the Arduino solutions operate much like simplified Swift applications. Instead of relying on UIKit or other Apple frameworks, most of the code you wrote in the Arduino solution took advantage of the GPIO and ADC pins on HUZZAH32 chip. In the `setup()` method, you configured the roles and modes for each pin you would use, then in the `loop()` method, you read the values and used them to turn pins on or off, to control the hardware. In the next chapter, you will extend this simple program to transmit this data to the companion iOS app over Bluetooth LE.

CHAPTER 6

Building a Bluetooth LE Hardware Companion App

In Chapter 5, you dipped your toes into the world of embedded systems by building an Arduino-based wireless door sensor. In this chapter, you will continue your hardware engineering journey by developing the Bluetooth LE-based companion app for the door sensor. Bluetooth LE is an extremely popular radio-based communication protocol that allows devices to communicate at ranges up to 77 meters (~250 feet), with a very low-energy requirement, compared to other wireless communication methods (this is the LE part of Bluetooth LE).

One of the wonderful things about iOS development today is that Apple's Core Bluetooth framework includes everything you need to make your app communicate with (or as) a Bluetooth LE device, using the built-in Bluetooth hardware on your iOS devices. Beyond simply establishing the connection with the device, this chapter will also introduce you to some best practices for improving your users' Bluetooth experience, such as saving the paired device and displaying messages from the hardware as push notifications on your users' iOS devices. As with so many things in iOS, Apple provides theoretical guidance on implementing the Bluetooth LE stack in their documentation, but leaves the burden of clarifying the implementation details up to the developers. To help fill this knowledge gap, in this chapter I will share what has worked best for me in the past.

Learning Objectives

In this chapter, you will take the door sensor you built in Chapter 5 and learn how to make it into a full Bluetooth LE solution by adding Bluetooth pairing and data transmission capabilities to your existing Arduino program, and then by building an iOS companion app to monitor the sensor. This chapter will start the new app that you will iterate throughout the section, IOTHome. IOTHome is an Internet of Things (IoT)

home-management system, which will eventually be used to track multiple IoT devices and provide a secure, Siri-compatible interface to the hardware via HomeKit. For the sake of introduction, the design wireframes for the first iteration of the IOTHome app are provided in Figure 6-1.

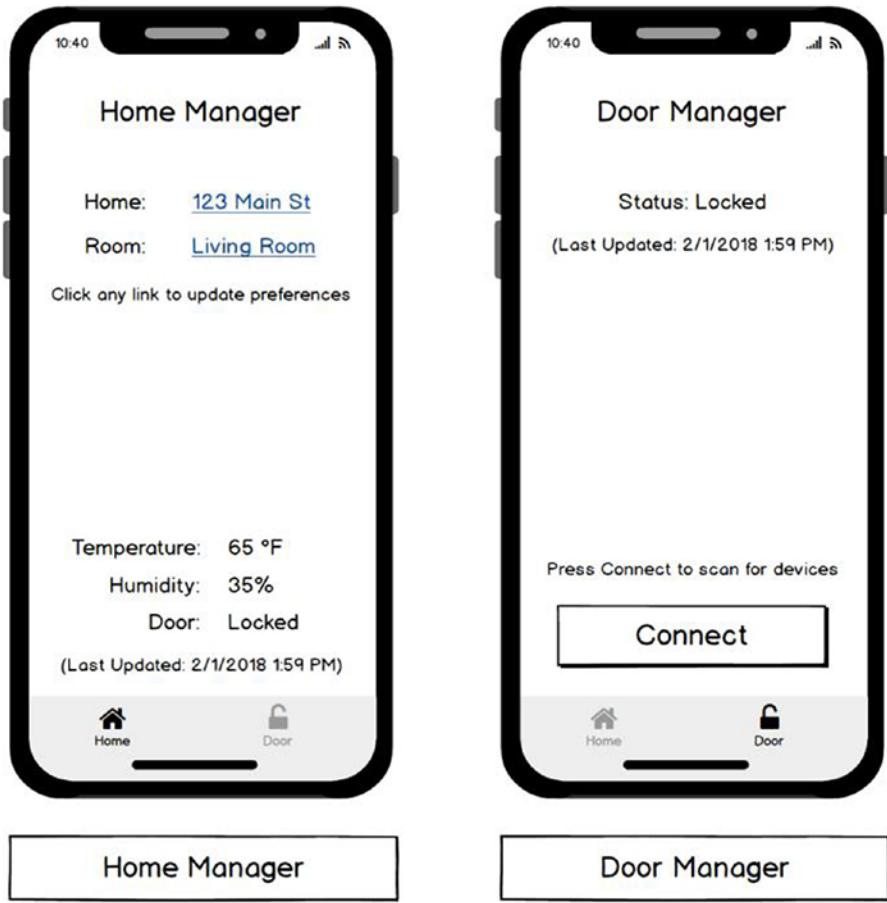


Figure 6-1. Design wireframes for the IOTHome app

To make the iteration process easy, you will once again use a Tab View Controller-based design to switch between screens in the application. In this chapter, you will focus on building the “Door” tab, which allows users to connect to the door-sensing hardware and monitor the status of their door and the battery level of

the device. In the process of building the companion iPhone app, you will learn the following key skills for IoT app development:

- Bluetooth LE core concepts
- Advertising an Arduino as a Bluetooth peripheral
- Sending data updates from an Arduino over Bluetooth
- Using Core Bluetooth on iOS to discover Bluetooth devices
- Using Core Bluetooth on IOS to listen for device updates
- Responding to Bluetooth updates in the background

This chapter will ask you to jump between the Arduino IDE and Xcode. If you feel you need to refresh your knowledge of Arduino, I recommend reviewing Chapter 5 and checking out some of Adafruit’s online tutorials at <https://learn.adafruit.com>.

A Quick Introduction to Bluetooth LE

To make the rest of the chapter easier for you to understand, I would first like to introduce some key concepts and terminology relating to Bluetooth LE. In this chapter, you will implement different aspects of the Bluetooth LE stack in the Arduino and iOS programs, and having some knowledge of the big picture at the beginning will help make the components easier to put together.

Bluetooth was originally developed as a short-distance, low-power wireless communications protocol by the Bluetooth Special Interest Group (SIG), a standards organization run by members from various computer hardware and telecommunications companies. It was always intended to serve a different purpose from LTE or Wi-Fi, aiming for efficient, short-message communication between computers or computers and peripherals (accessories). Bluetooth LE (Low Energy) was developed as part of the 4.0 specification of the Bluetooth standard and was designed to provide a lower-power, lower-cost implementation of the standard, without sacrificing range or requiring drastically different hardware to implement either version of the standard (Classic Bluetooth or Bluetooth LE). In practice, Apple and many other smartphone manufacturers use the lower-power, slightly slower Bluetooth LE standard to communicate with accessories that require little data (such as IoT health sensors or location beacons) and the higher-power, higher-throughput Bluetooth standard for devices that require less latency (such as phone headsets or speakers).

When it comes to implementing Bluetooth, there are two firmly defined *roles* (the actual name of the term) that a device can play. A *peripheral* is a device whose role is to provide supporting features or data for another device. You can think about a Bluetooth peripheral in the same way that you would a peripheral for a personal computer. You connect to a peripheral, and it provides *services* (features) that were previously not available on the main processing unit. When you think of a Bluetooth keyboard or headset, these are devices that act as *peripherals*. The other major role a device can play is that of a *central* device, a device that connects to and manages peripherals. In most cases, when you think of a computer or smartphone, when it is sending its audio to a Bluetooth headset or using a Bluetooth keyboard for input, it is serving the role of a central device.

Bluetooth devices are not limited by their hardware to serve only one role. For example, within the same application, you can create a mode in which the app is acting as a central device and another in which the app is acting as a peripheral (for example, if you were making an offline photo-sharing app), but a device will only serve one role during a communication session. In this chapter, the Arduino will act as the peripheral, and the IOTHome app will act as the central device.

If you have connected to a Bluetooth accessory from your iPhone, you may remember that the process follows this order:

1. Scan for nearby devices.
2. Click a device in the list to try to connect to it.
3. Confirm the connection using a PIN code.
4. Begin Bluetooth communication.

The way a central device finds peripherals is by *scanning* (searching) for their *advertising packets* that are being broadcast by peripherals. Similar to an Ethernet packet, these are special, short messages that advertise the device's name, *services* (features), and *characteristics* (data types) it provides. In a similar manner to pairing from iOS, in most applications, you will use this advertising data to let the user select which device he or she wants to connect to. Once the device is connected, two-way communication can proceed. To save power on both sides, scanning and advertising are operations that are initiated by a user and stop as soon as a connection is established. In this chapter, the Arduino will start advertising its services as soon as it is powered on, and the Connect button in the IOTHome app will be used to scan for and connect to the Arduino.

A final detail to mention about Bluetooth is the idea of a *profile*. Services and characteristics are identified by 128-bit universally unique identifiers (UUIDs). However, for reoccurring hardware, such as heart rate monitors and headsets, the Bluetooth SIG asks you to use the commonly agreed upon set of services and characteristics of UUIDs for those devices. These are what are referred to as *profiles* and can help you provide a better experience for your users, by taking advantage of any optimizations the central device has for known profiles. For example, iOS will show battery levels and a headphone icon when you connect to a Bluetooth headset. The door sensor for the IOTHome app does not fit into any of the predefined profiles, so you will use randomly generated UUIDs to identify it.

Adding Bluetooth Functionality to an Arduino Solution

Now that you have a better understanding of Bluetooth, you are ready to start implementing it in the Arduino solution (program) for the door sensor. To configure the door sensor as a Bluetooth peripheral, you must do the following to the Arduino solution:

- Establish the device as a Bluetooth server (peripheral), so it can accept incoming connections
- Advertise the services and characteristics the peripheral provides
- Send data updates over Bluetooth LE when the magnetic switch state or battery level changes

Although Bluetooth LE itself is an efficient protocol, you can make your implementation even more efficient by pushing updates only when something changes (rather than every second). Keeping the connection established is a low-power operation, but pushing data over it has a cost. Reducing communication will help battery life on the Arduino and iOS app significantly.

As does iOS, Arduino has a vast collection of libraries provided by the Arduino foundation and open source libraries developed by the user community. In this chapter, you will use the `ESP32_BLE_Arduino` library, developed by Neil Kolban, to establish the Bluetooth LE server, handle incoming connections, and push updates over Bluetooth LE. His hard work means you do not have to study timing diagrams or implement the low-level protocol yourself. You can focus on how to use Bluetooth in your app, rather than on how to build it. Additionally, his library provides customization, specifically for ESP32-based devices such as the Adafruit Huzzah32 you are using for this project.

Although the form factor and available features for ESP32-based devices vary widely, they all use the same system-on-a-chip (SoC) from Espressif. The device manufacturers (OEMs) simply decide what they want to use and how to expose it.

Installing the ESP32_BLE_Arduino Library for Bluetooth Communication

To take advantage of the ESP32_BLE_Arduino library, you first must import it into the Arduino IDE. After you have imported it, you can use it again in other projects, without having to repeat the import step. To begin, navigate to the GitHub page for the library at https://github.com/nkolban/ESP32_BLE_Arduino. To download the latest version of the library, click the Clone or download button at the top right, as shown in Figure 6-2. You will have options to Open in Desktop (normally this will open GitHub or Sourcetree, if they are installed on your computer) or Download ZIP (download the files as a zipped archive). Because you will have to import only a subset of the files into the Arduino IDE, I recommend selecting Download ZIP.

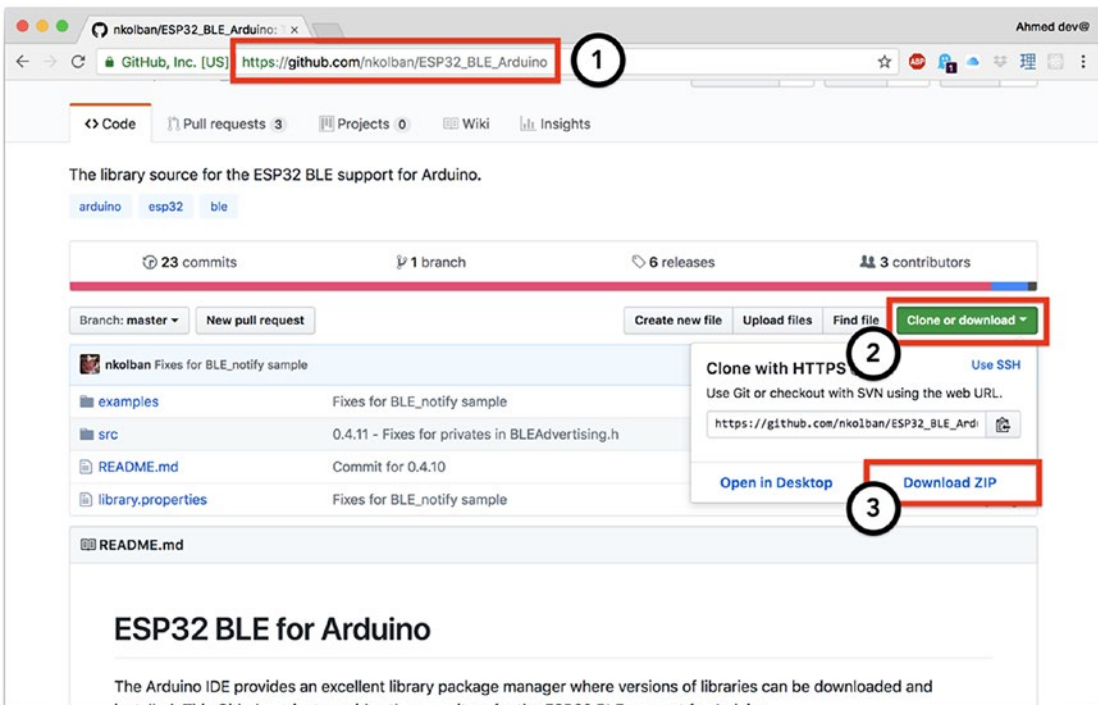


Figure 6-2. Downloading a repository from GitHub

Once the repository is on your computer, return to the Arduino IDE and go to the Sketch menu. As shown in Figure 6-3, select Include Library and then Add .ZIP Library. When the file browser window appears, select the zip file you just downloaded.

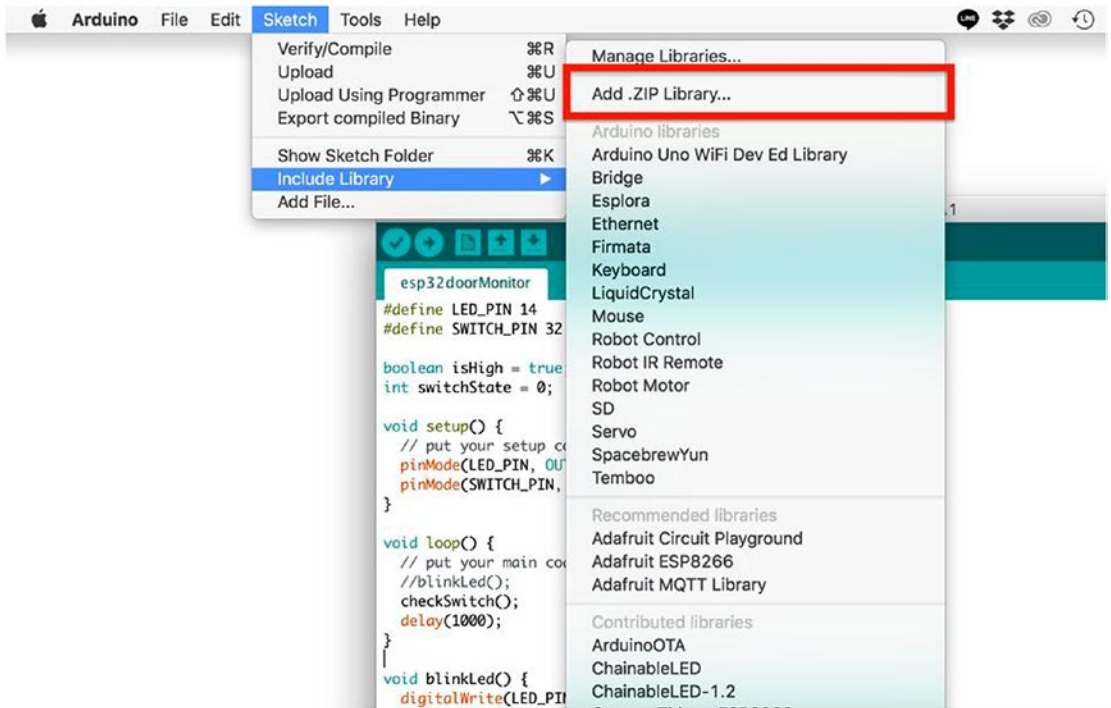


Figure 6-3. *Importing a zip archive*

After importing the file, return to the Sketch menu. As shown in Figure 6-4, select Include Library one more time. This time, the library you just imported, ESP32 BLE Arduino, should appear in the context menu.

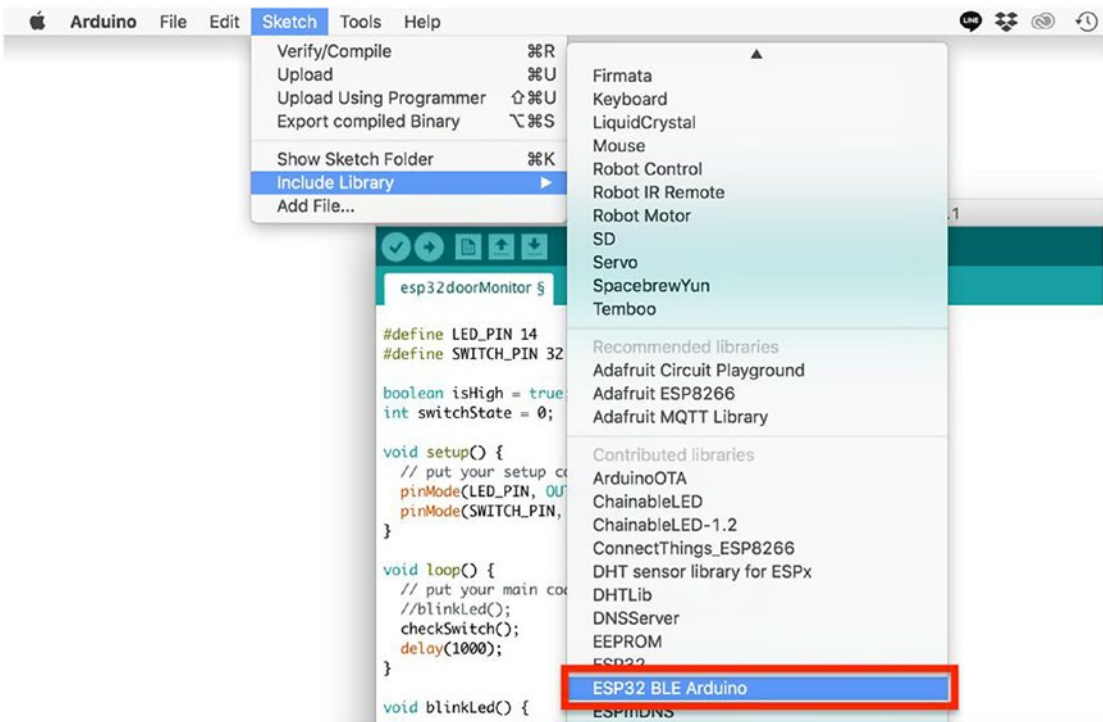


Figure 6-4. Including the ESP32 BLE Arduino library in your project

After selecting the library, you will notice that the source code for your solution has been modified to include some of the files from the library, as shown in Listing 6-1.

Listing 6-1. Arduino Solution After Including ESP32 BLE Library

```

#include <BLE.h>
#include <BLEUtils.h>
#include <BLEScan.h>
#include <BLEAdvertisedDevice.h>

#define RED_LED_PIN 14
...

void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600);

```

```

Serial.println(" Program start");
...
}
...

```

For this project, you must use the Bluetooth server and data transmission features of the library, which are not provided by the default set of included files. Modify your solution, as shown in Listing 6-2, to include the correct files.

Listing 6-2. Arduino Solution with Files for Bluetooth Server Features

```

#include <BLEDevice.h>
#include <BLEUtils.h>
#include <BLEServer.h>
#include <BLE2902.h>

#define RED_LED_PIN 14
...

void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600);
  Serial.println(" Program start");

  pinMode(RED_LED_PIN, OUTPUT);
  ...
}
...

```

To confirm that the files are compatible with the project, click the Verify button in the Arduino IDE. The project should compile successfully. You can repeat these steps to include other libraries in your future projects.

Setting Up the Arduino As a Bluetooth Peripheral

Now that the Bluetooth library has been imported, you can start implementing the Bluetooth server, which will allow the device to advertise itself as a peripheral and accept incoming connections. To revisit the earlier overview of the Bluetooth specification, this is accomplished by advertising the services (features) and data types (characteristics)

the device provides. Both of these values are represented as 128-bit UUIDs. A UUID is a mostly random pattern of hexadecimal characters; however, in order to register the door sensor as a Bluetooth LE-compatible device, you will have to replace some of the values with known identifiers from the Bluetooth LE specification.

To get started, you will have to generate four UUIDs—two to represent the services being advertised (information about the door, information about the battery) and two to represent the characteristics (battery level, door lock status). On your Mac, you can generate UUIDs from the command line, using the `uuidgen` tool. I have shared my result in Listing 6-3.

Listing 6-3. Generating a UUID from the OS X Command Line

```
ahmeds-macbook:arduino abakir$ uuidgen
8CED1808-7984-47CE-BE9C-E2DD56317575
```

As you can see, no additional parameters are required to run the `uuidgen` tool. The result is printed immediately as a string on the command line. Run this command four times to generate your four UUIDs and save the results. I have appended my results to the Arduino solution, as shown in Listing 6-4.

Listing 6-4. Appending UUIDs to the Arduino Solution

```
#include <BLEDevice.h>
#include <BLEUtils.h>
#include <BLEServer.h>
#include <BLE2902.h>

#define RED_LED_PIN 14
...
#define LOCK_SERVICE_UUID "83b46845-6e9c-4b25-89cf-871cc74cc68e"
#define BATT_SERVICE_UUID "7d6925f3-6e19-48c6-a503-05585abe761e"
#define LOCK_CHARACTERISTIC_UUID "4b61d6b9-2e29-4fdf-a74a-7b8bf70ecd9a"
#define BATT_CHARACTERISTIC_UUID "8e628af6-0275-4f80-bb64-58f2b2771cba"

void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600);
  Serial.println(" Program start");
}
```

```
pinMode(RED_LED_PIN, OUTPUT);
...
}
...
```

Next, you must replace with known values from the Bluetooth specification the second two bytes (the four characters starting at the fifth character in the string). This master record of values is called the Bluetooth GATT (Generic Attributes). You can find the values for services at www.bluetooth.com/specifications/gatt/services and the values for characteristics at www.bluetooth.com/specifications/gatt/characteristics. While the GATT do not cover all use cases, they cover a wide range, which should be sufficient for most projects. For advertising the battery level, you can use the Battery Service identifier (0x180F) and the Battery Level State characteristic (0x2A1B). The door is a bit harder, as there is no door service; however, the Alert Notification Service identifier (0x1811) and Alert Status characteristic (0x2A3F) estimate the use case pretty well and are appropriate here. As shown in Listing 6-5, modify your solution to include these new values.

Listing 6-5. Arduino Solution with Valid GATT UUIDs

```
#include <BLEDevice.h>
#include <BLEUtils.h>
#include <BLEServer.h>
#include <BLE2902.h>

#define RED_LED_PIN 14
...
#define LOCK_SERVICE_UUID "83b41811-6e9c-4b25-89cf-871cc74cc68e"
#define BATT_SERVICE_UUID "7d69180F-6e19-48c6-a503-05585abe761e"
#define LOCK_CHARACTERISTIC_UUID "4b612A3F-2e29-4fdf-a74a-7b8bf70ecd9a"
#define BATT_CHARACTERISTIC_UUID "8e622A1B-0275-4f80-bb64-58f2b2771cba"

void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600);
  Serial.println(" Program start");
```

```

pinMode(RED_LED_PIN, OUTPUT);
...
}
...

```

Next, you can begin setting up the Bluetooth server. The `Arduino_ESP32_BLE` library exposes these functions through the `BLEServer` class. The class is straightforward to use, as the setup primarily requires registering the characteristics, services, and callback handlers for the server events (for example, connection initiated, disconnected). To begin, initialize the server, characteristics, and services, as shown in Listing 6-6.

Listing 6-6. Initializing the Bluetooth LE Server from the Arduino Solution

```

...
#define BATT_CHARACTERISTIC_UUID "134c298f-7d6b-4f64-8496-8965e0851d03"
BLECharacteristic *lockCharacteristic;
BLECharacteristic *battCharacteristic;

void setup() {
    ...
    pinMode(BATTERY_PIN, INPUT);
    startBLE();
}

void startBLE() {
    // Create the BLE Device
    BLEDevice::init("IOTDoor");

    // Create the BLE Server
    BLEServer *bleServer = BLEDevice::createServer();

    // Create the BLE Service
    BLEService *lockService = bleServer->
        createService(LOCK_SERVICE_UUID);

    // Create a BLE Characteristic
    lockCharacteristic = lockService->createCharacteristic(
        LOCK_CHARACTERISTIC_UUID,
        BLECharacteristic::PROPERTY_READ |

```

```

        BLECharacteristic::PROPERTY_WRITE |
        BLECharacteristic::PROPERTY_NOTIFY |
        BLECharacteristic::PROPERTY_INDICATE
    );

    lockCharacteristic->addDescriptor(new BLE2902());

    BLEService *battService = bleServer->
        createService(BATT_SERVICE_UUID);

    battCharacteristic = lockService->createCharacteristic(
        BATT_CHARACTERISTIC_UUID,
        BLECharacteristic::PROPERTY_READ |
        BLECharacteristic::PROPERTY_WRITE |
        BLECharacteristic::PROPERTY_NOTIFY |
        BLECharacteristic::PROPERTY_INDICATE
    );
    battCharacteristic->addDescriptor(new BLE2902());

    // Start the service
    lockService->start();
    battService->start();

    // Start advertising
    bleServer->getAdvertising()->start();
}

```

In accordance with the execution flow of Arduino solutions, you must start the server from the `setup()` method, as it is the only area of the program that is executed once. To send messages later on, you will have to reuse the characteristic objects, thus I declared them as global variables. When setting up the services and characteristics, be careful to attach them correctly, or they will not work. As with the Bluetooth specification, services can only be discovered when they are attached to a characteristic. The properties for the characteristics do not have to include the full list in my example (READ, WRITE, NOTIFY, INDICATE). For future projects, you can pare them down as you need to.

The final requirement to start the server is to register the connection callbacks. To perform this operation, you will have to implement the `BLEServerCallbacks` protocol. As C++ and Swift share common language design ancestry, the process of implementing

the protocol should be very familiar to you. Just as in Swift, define a class that inherits the protocol, and implement the required named methods, `onConnect(BLEServer * pServer)` and `onDisconnect(BLEServer * pServer)`. As the names suggest, these are called when the server establishes or destroys a connection. In Listing 6-7, I have modified the solution to include this code and added the call to attach the callbacks to the server object. Right now, the only actions you must perform on the callbacks are setting a status object to true or false and turning the blue LED on or off. Later, you will use this status object again, to help filter outgoing messages.

Listing 6-7. Arduino Solution Including Valid Bluetooth LE Callbacks

```
...
BLECharacteristic *lockCharacteristic;
BLECharacteristic *battCharacteristic;
bool deviceConnected = false;

class MyServerCallbacks: public BLEServerCallbacks {
    void onConnect(BLEServer* pServer) {
        deviceConnected = true;
        digitalWrite(BLUE_LED_PIN, deviceConnected);
    };

    void onDisconnect(BLEServer* pServer) {
        deviceConnected = false;
        digitalWrite(BLUE_LED_PIN, deviceConnected);
    }
};

...
void startBLE() {
    // Create the BLE Device
    BLEDevice::init("IOTDoor");

    // Create the BLE Server
    BLEServer *bleServer = BLEDevice::createServer();
    bleServer->setCallbacks(new MyServerCallbacks());

    // Create the BLE Service
    BLEService *lockService = bleServer->createService(LOCK_SERVICE_UUID);
```

```

...
}
...

```

This step completes all of the setup required for the Bluetooth LE server component of the Arduino solution. To verify that the server was set up correctly, I like to use a Bluetooth LE scanner app to check that the Arduino is advertising the correct information. My preferred app is LightBlue Explorer, available on the App Store at <https://itunes.apple.com/us/app/lightblue-explorer/id557428110?mt=8>. In Figure 6-5, I have included screenshots of the app, including the status of my door sensor. On the home screen of the app, you should see the advertising name you specified in Listing 6-6 (“IOTDoor”), and on the detail page, you see the four UUIDs you used to set up the server. To find the device, simply download the solution to the HUZZAH32, press the Reset button on the chip, and wait a few seconds.

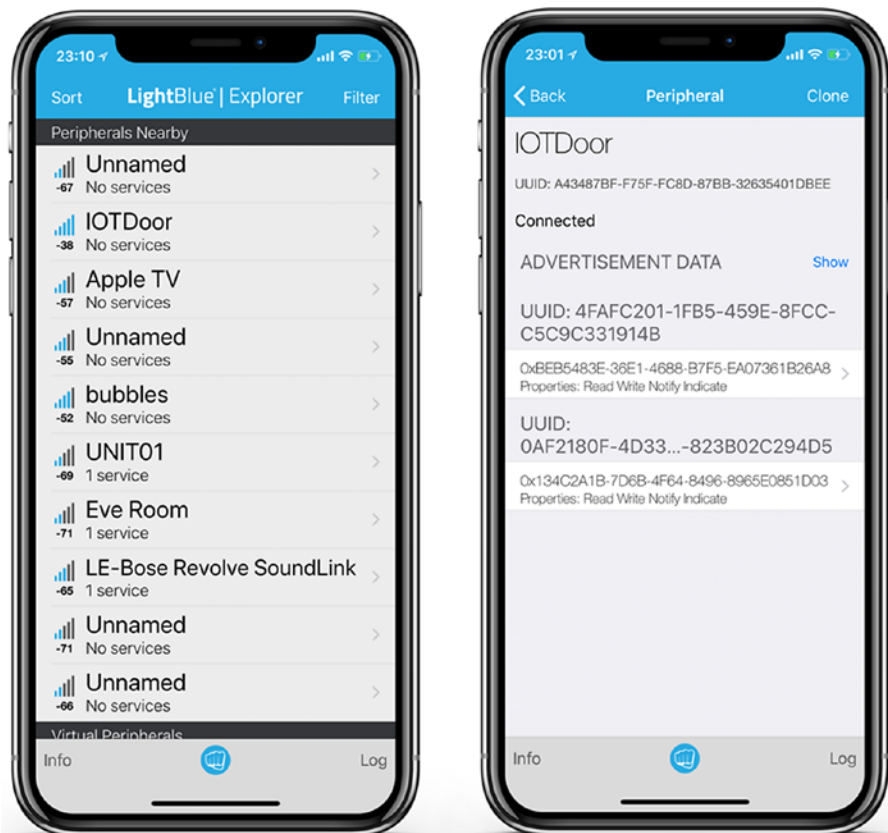


Figure 6-5. Verifying Bluetooth LE advertising data using LightBlue Explorer

Caution After adding the Bluetooth server initialization code, your Arduino solution will start up much slower. You can verify this by opening up the Arduino IDE serial monitor. You should see log messages from the Bluetooth server code every few sections, while it is starting up.

Sending Data Updates via Bluetooth LE

Believe it or not, setting up the Bluetooth server for advertising the device was the hardest part of the Arduino solution. To send data updates via Bluetooth LE, you simply have to use the saved characteristics (Battery Power Level, Alert Status) and push new updates when the state changes. Although the Arduino solution is set to poll for updates every second, in practice, it is not a wise idea to transmit Bluetooth updates every second. In addition to draining power, constantly transmitting the status will prevent you from being able to notify the user when the device has an update (such as when the magnetic sensor detects that the door has been opened).

Begin by modifying the `checkSensor()` method, which you used in Chapter 5 to detect if the door sensor was closed or open. In the old implementation of the method, you checked if the pin was high (connected) or low (disconnected) and used that value to turn the LED on or off. While this implementation is still valid, you must save the old value to trigger the update.

To send updates using the ESP32 BLE Arduino library, you set a new value on the characteristic, using the `setValue()` method. Next, you tell the characteristic object to notify all connected devices using the `notify()` method. In Listing 6-8, I have updated the solution to save the old sensor value as a global variable and to trigger the Bluetooth update only if a device is connected to the peripheral and the value has changed since the last update.

Listing 6-8. Posting Magnetic Sensor Updates from the Arduino Solution

```
void loop() {
  // put your main code here, to run repeatedly:
  checkSensor();
  checkBattery();
  delay(1000);
}
```

```

...
void checkSensor() {
    int currentState = digitalRead(SWITCH_PIN);
    if (currentState != switchState) {
        updateLockBLE(currentState);
    }
    switchState = currentState;
    digitalWrite(RED_LED_PIN, switchState);
    Serial.print("Sensor state: ");
    Serial.println(switchState);
}
...
void updateLockBLE(bool currentState) {
    uint8_t value = 0;
    if (deviceConnected) {
        value = currentState ? 1 : 0;
        lockCharacteristic->setValue(&value, 1);
        lockCharacteristic->notify();
    }
}

```

For the `checkBattery()` method, you can use the same process; however, you will have to adjust the logic for detecting changes in status. In practice, you will notice the battery level continuously decreasing as the sensor stays powered on. This is normal, but for the sake of users, it is best to notify them only about noticeable changes. In Listing 6-9, I have updated the solution to save the old battery value (just as with the magnetic sensor), but instead of sending a notification every time the value changes, I only send it when the battery level is at least 5% lower than the previous reading.

Listing 6-9. Posting Battery Updates When the Level Decreases More Than 5%

```

void checkBattery() {
    float currentLevel = analogRead(BATTERY_PIN);
    currentLevel = ((currentLevel / 4095) * 2 * 3.3 *
        1.1) * 100 / 4.3;
    if (currentLevel + 5.0 < batteryLevel) {

```



```

    updateBatteryBLE(currentLevel);
  }
  batteryLevel = currentLevel;
  Serial.print("Battery Level: ");
  Serial.print(batteryLevel);
  Serial.println("%");
}
...
void updateBatteryBLE(float currentLevel) {
  if (deviceConnected) {
    char string[8];
    dtostrf(currentLevel, 3, 1, string);
    battCharacteristic->setValue(string);
    battCharacteristic->notify();
  }
}

```

With these steps, the Arduino solution in this chapter is complete. From here on out, you will focus on how to build an app that connects to the sensor you created and read the values it transmits.

Using Core Bluetooth to Communicate with Bluetooth LE Devices

As mentioned at the beginning of the chapter, Apple provides the Core Bluetooth framework to help you connect to Bluetooth devices quickly and safely, using their APIs. Instead of focusing on pouring over the Bluetooth specification to implement basic communication yourself, you can focus on the business logic directly, just as you did with the ESP32 BLE library for the Arduino solution. In this half of the chapter, I will quickly walk you through the setup of the IOTHome project, which will be used as the base for the remaining projects in this section, and then introduce the following steps for setting up an app that connects to a Bluetooth LE accessory:

- Adding the Bluetooth background service permission to the project
- Discovering and connecting to Bluetooth devices

- Monitoring Bluetooth characteristic updates
- Responding to Bluetooth characteristic updates while the app is backgrounded

As with previous projects in the book, the goals here are not only to learn some useful skills for yourself but also how to apply them to provide a convenient user experience in the future.

Setting Up the IOTHome Project

Jumping back to Figure 6-1 for a second, the IOTHome project for this section will provide a tab-based user interface to allow users to monitor various data points about their home, using IoT technologies. Although this chapter will focus only on the screen for the door sensor, it is a good idea to start the project out on the right foot, by creating a new project in Xcode using the Tab Bar Controller template, just as you did in Chapter 1 for the IOTFit project. To shift the focus to the Bluetooth portion of the project, I strongly recommend referring back to Chapter 1 for a detailed review of this operation. However, for everyone's benefit, the summary is

1. Choose Tab Bar Controller from the New Project option in Xcode's File menu.
2. Navigate to the `Main.storyboard` file, to verify that the tab-based project was created successfully.
3. Rename the tabs "Door" and "Home" by double-clicking them in the storyboard, then choose new names and icons from the Attributes Inspector Xcode's right-hand side panes.
4. Rename the classes representing the tabbed view controllers to `HomeViewController` (first tab) and `DoorViewController` (second tab).

When complete, your storyboard and project structure should resemble the screenshot in Figure 6-6.

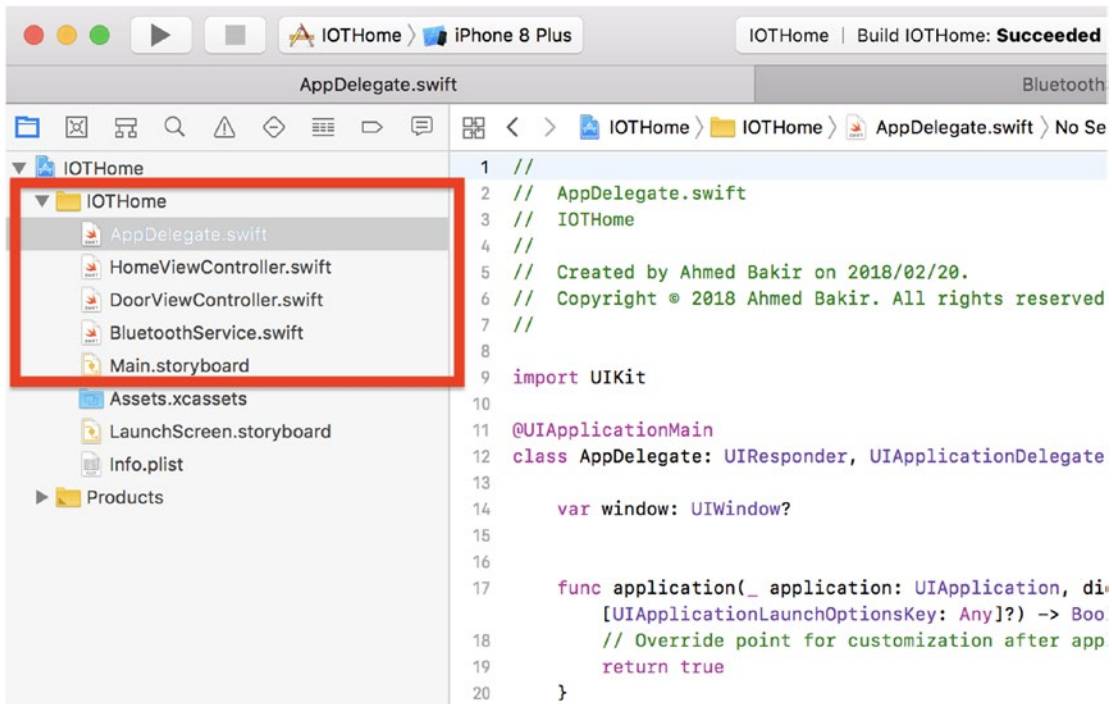


Figure 6-6. New IOTHome project, including modified storyboard and view controller names

In order to initiate the Bluetooth connection and display the data from the sensor, you will have to set up the user interface with the use of code and Interface Builder. In Listing 6-10, I have provided the modified DoorViewController class, which includes the properties and stub method for the user interface. As this screen will be used mostly for displaying data, the user interface primarily consists of UILabel objects; however, you will have to use a UIButton to initiate and destroy the connection. The user can only access the app from the “connected” or “disconnected” state, so my implementation shares the same button for the connection management actions.

Listing 6-10. DoorViewController Class, Including User Interface Properties and Stub Methods

```
import UIKit

class DoorViewController: UIViewController {
    @IBOutlet var statusLabel: UILabel?
```

```

@IBOutlet var batteryLabel: UILabel?
@IBOutlet var lastUpdatedLabel: UILabel?
@IBOutlet var connectButton: UIButton?

let dateFormatter = DateFormatter()

override func viewDidLoad() {
    super.viewDidLoad()

    dateFormatter.dateStyle = .medium
    dateFormatter.timeStyle = .short
}

@IBAction func connect() {
}
}

```

To help you theme the view controller, in Table 6-1, I have provided the styling options I used for each element. As with the first section, I used Apple’s font classes to reduce the complexity of managing fonts and give the user an experience more consistent with Apple’s iOS design guidelines.

Table 6-1. *Styling for Door View Controller User Interface Elements*

Element Name	Text Style	Height	Top Margin	Bottom Margin	Left Margin	Right Margin
Navigation bar	Prefers large text	—	—	—	—	—
“Status” title label	Title 2	24	40	—	30	20
“Status” value label	Title 2	24	40	—	20	≥30
“Battery Level” title label	Title 2	24	8	—	30	20
“Battery Level” value label	Title 2	24	8	—	20	≥30
“Last Updated” label	Body	25	8	—	20	20
“Press to Connect” label	Body	25	—	20	20	20
“Connect” button	Title 1	60	20	30	20	20

After the elements are styled, remember to drag and drop outlets, to connect the code to the storyboard file. Your completed storyboard and its Connection Inspector should resemble the output shown in Figure 6-7.

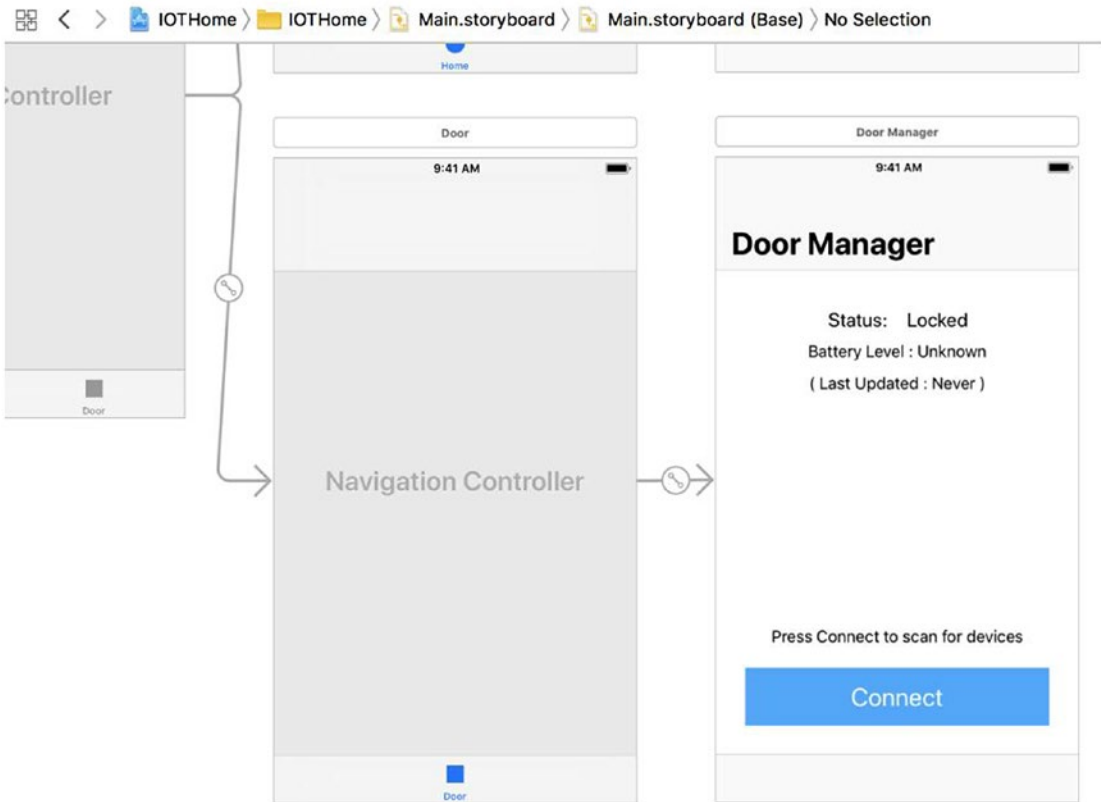


Figure 6-7. *Styled storyboard and Interface Builder connections for Door View Controller*

Enabling Bluetooth Accessory Background Updates

Although iOS is notorious for pausing applications while they are backgrounded, Bluetooth LE peripheral updates are one of the few exceptions, much like the background locations updates you enabled in the first section of the book. In the same manner as background location updates, to enable Bluetooth updates, you must declare them as capabilities in your application’s project settings.

To begin the process of declaring the capability, click the `.xcodeproj` file for the IOTHome project (IOTHome.xcodeproj). As shown in Figure 6-8, click the Capabilities tab. Click on the Background Modes and then Uses Bluetooth LE accessories.

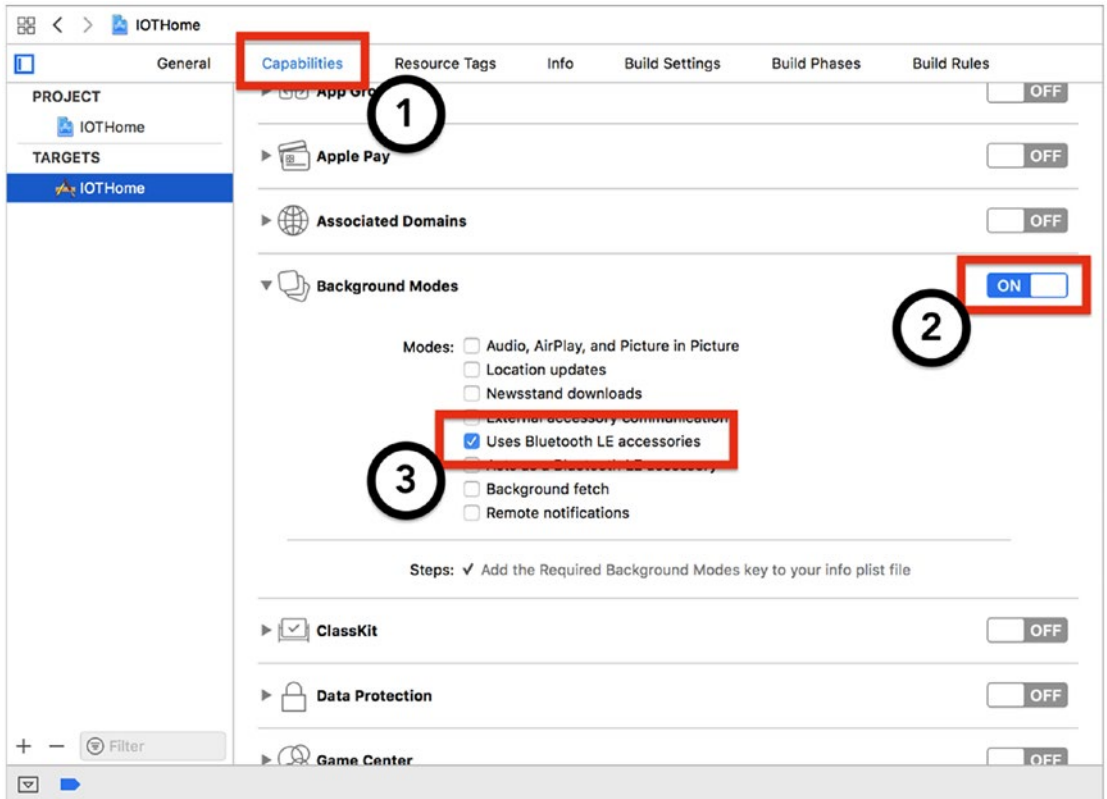


Figure 6-8. Enabling the Bluetooth accessory background mode

For background location updates, you are required to update the capabilities for your project and define a message for the location permission alert in your project's `Info.plist` file. For background Bluetooth projects, the “and” is what you require to define the role the iOS device will play in the Bluetooth operation. For this project, you only have to discover and connect to Bluetooth LE devices. Thus, you will only perform the central manager role. Click the `Info.plist` file for your project and enable the `NSBluetoothPeripheralUsageDescription` key-value pair, as shown in Figure 6-9. For the description, I used the following message: “IOTHome would like to use Bluetooth to help you monitor Bluetooth-based IOT sensors in your home. This information will not be shared outside of the app.”

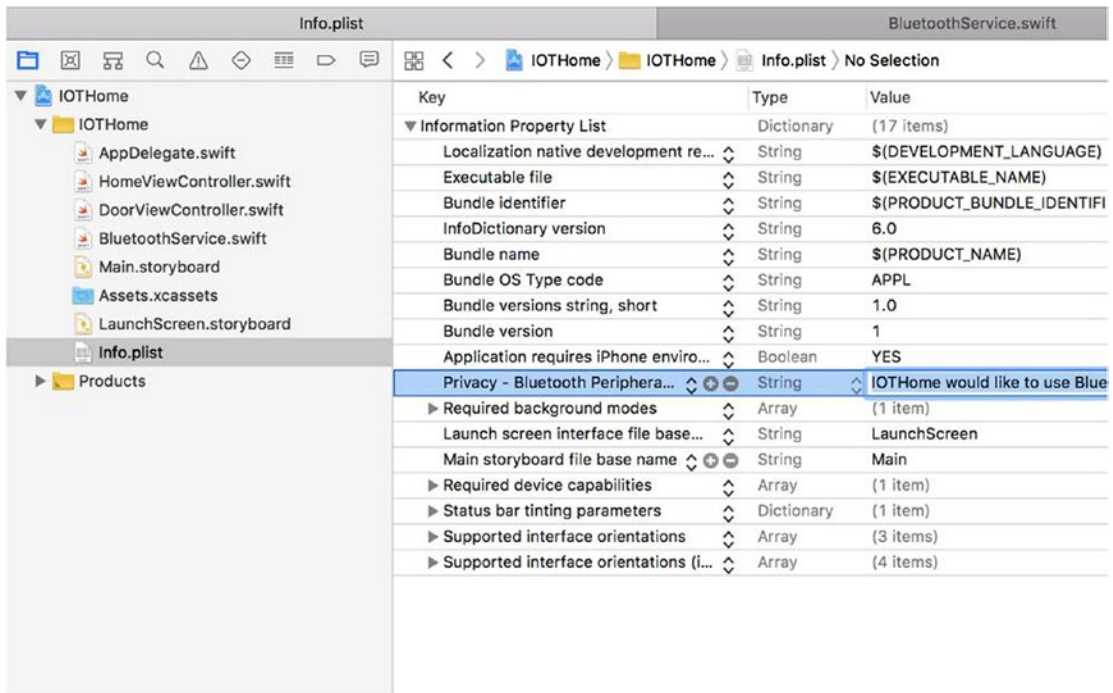


Figure 6-9. Specifying the central manager alert message in the project's Info.plist

These steps complete the project setup for this iteration of the IOTHome project. Now you can begin to flesh out the project, by using Core Bluetooth to discover and interact with the door sensor.

Setting Up the App As a Central Manager

With the rough user interface and capability permissions in place, you are now ready to begin scanning for Bluetooth devices from the IOTHome app. To follow the vocabulary of Bluetooth LE, in this section, you will enable the iOS app to act as a central manager and connect to the Arduino-based peripheral.

To help keep the code manageable, you will wrap the Bluetooth operations into a class called BluetoothService. This will deliver state messages back to the class that instantiated it, via a protocol you will define, called BluetoothServiceDelegate. While it is possible to put everything into one file, the logic required to implement Bluetooth is a bit heavy, and sticking that many functions into one view controller will introduce the *Massive View Controller* anti-pattern many developers cite as a weakness in Apple's Model-View-Controller architecture for iOS applications. The recent

adoption of protocols as a way to alleviate this is called *Protocol Oriented Programming*. It gives you an easy path to split up code, by defining key behaviors in protocols and grouping the implementations via *extensions*. If you have ever used extensions before to add functionality to classes or split up large delegate-based APIs (such as `UITableViewDelegate`), you are already familiar with the basics of protocol oriented programming.

For the IOTHome app, you must expose the following events to consumers of the `BluetoothService` class:

- Bluetooth connection state change
- Door state change
- Battery level change

Following these requirements, you can declare the `BluetoothServiceDelegate` protocol, with the signature provided in Listing 6-11. This should go in a new file named `BluetoothService.swift`. Bluetooth is a serial-based protocol, like the RS-232 serial port of legacy computing, meaning it delivers data as a continuous stream of bytes. Core Bluetooth exposes this data using the `Data` class. In the Arduino solution, the state data was transmitted as string values, so you can safely convert the input data from Core Bluetooth to `String` objects, to make message passing easier. On a similar note, Core Bluetooth does not provide a terse object to describe the state of a connection, so I defined the `ConnectionStatus` enum to represent this.

Listing 6-11. Declaration for `BluetoothServiceDelegate` Protocol

```
enum ConnectionStatus {
    case unknown
    case scanning
    case connecting
    case connected
    case disconnected
}

protocol BluetoothServiceDelegate: class {
    func didUpdateConnection(status: ConnectionStatus)
    func didReceiveDoorUpdate(value: String)
    func didReceiveBatteryUpdate(value: String)
}
```


Although the protocol specifies the functions that will be used, it does not perform any logic by itself. To implement the behavior for the `BluetoothServiceDelegate` protocol, you must provide the method definitions in `BluetoothService.swift`, as shown in Listing 6-12. The main points of this initial implementation include convenience initializers, a stub for the `connect()` method, and `CBUUID` objects to represent the Bluetooth LE service and characteristic UUIDs you will have to interact with later.

Listing 6-12. Initial Definition for `BluetoothService` Class

```
import Foundation
import CoreBluetooth

let doorServiceUUID = CBUUID(string: "83b46845-6e9c-4b25-89cf-
871cc74cc68e")
let battServiceUUID = CBUUID(string: "7d6925f3-6e19-48c6-a503-
05585abe761e")
let doorCharUUID = CBUUID(string: "4b61d6b9-2e29-4fdf-a74a-7b8bf70ecd9a")
let battCharUUID = CBUUID(string: "8e628af6-0275-4f80-bb64-58f2b2771cba")

class BluetoothService: NSObject {
    let doorServices = [doorServiceUUID]
    let doorCharacteristics = [doorCharUUID,
        battCharUUID]

    weak var delegate: BluetoothServiceDelegate?

    convenience init(delegate:
        BluetoothServiceDelegate) {
        self.init()
        self.delegate = delegate
    }

    private override init() {
        super.init()
    }
}
```

```

func connect() {
    delegate?.didUpdateConnection(status: connectionStatus)
}
}

```

Having declared the `BluetoothService` class and its corresponding protocol, switch back to `DoorViewController.swift`. As shown in Listing 6-13, create a property to represent an instance of the `BluetoothService` class and instantiate it in the `viewDidLoad()` method. To allow compilation to complete successfully, add a *stub* (blank) extension of the `BluetoothServiceDelegate` protocol beneath the definition for the `DoorViewController` class. In the `connect()` method you defined earlier to represent the *Connect* button being pressed, call the corresponding method from the `BluetoothService` class.

Listing 6-13. Initial Implementation of the `BluetoothServiceDelegate` Protocol

```

class DoorViewController: UIViewController {
    ...
    var bluetoothService: BluetoothService?

    override func viewDidLoad() {
        super.viewDidLoad()
        ...
        bluetoothService = BluetoothService(delegate: self)
    }

    @IBAction func connect() {
        bluetoothService?.connect()
    }
}

extension DoorViewController: BluetoothServiceDelegate {
    func didReceiveDoorUpdate(value: String) {
    }

    func didReceiveBatteryUpdate(value: String) {
    }

    func didUpdateConnection(status: ConnectionStatus) {
    }
}

```

Following the logical progression of the application, you should now begin to develop the `connect()` method for the `BluetoothService` class. The `ConnectionState` enum I provided earlier defines five states of a Bluetooth connection: `unknown`, `scanning`, `connecting`, `connected`, and `disconnected`. During the `unknown` and `disconnected` states, you can safely assume that the app does not have a connection to a Bluetooth peripheral, so you can use these to begin the process of scanning for a device. If the state is `connected` or `connecting`, you can assume the user is attempting to establish a connection with a device, and this is the cue to disconnect. Finally, the `scanning` state indicates that the app is scanning for a device, and you can use this to stop scanning. In Listing 6-14, I have included an initial implementation of the `connect()` method, which updates the connection state, based on the previous one. To set the initial state correctly, I have included a `connectionStatus` property in the class, with a default value of `unknown`. As you progress in this section, you will connect these state changes to CoreBluetooth API calls.

Listing 6-14. Initial Implementation of the `connect()` Method

```
class BluetoothService: NSObject{
    func connect() {
        switch connectionStatus {
        case .unknown, .disconnected :
            connectionStatus = .scanning
        case .connected, .connecting:
            connectionStatus = .disconnected
        default:
            connectionStatus = .disconnected
        }
        delegate?.didUpdateConnection(status: connectionStatus)
    }
}

extension DoorViewController: BluetoothServiceDelegate {
    ...
    func didUpdateConnection(status: ConnectionState) {
        switch status {
        case .connecting:
            connectButton?.setTitle("Connecting", for: .normal)
        }
    }
}
```

```

    case .connected:
        connectButton?.setTitle("Disconnect", for: .normal)
    case .scanning:
        connectButton?.setTitle("Scanning", for: .normal)
    default:
        connectButton?.setTitle("Connect", for: .normal)
    }
}
}

```

For the IOTHome app, the only Bluetooth role the BluetoothService class must provide is that of a central manager. You can accomplish this via the CBCentralManager class. To use it in your program, you must instantiate a CBCentralManager object and set its delegate. In Listing 6-15, I have added this object as a property to the class and provided stubs for the CBCentralManagerDelegate protocol methods you must use in this project.

Listing 6-15. Initializing a CBCentralManager Object

```

class BluetoothService: NSObject {
    ...
    weak var delegate: BluetoothServiceDelegate?
    var centralManager: CBCentralManager?
    ...
    private override init() {
        super.init()
        centralManager = CBCentralManager.init(delegate: self,
            queue: nil)
    }
}

extension BluetoothService: CBCentralManagerDelegate {
    func centralManagerDidUpdateState(_ central:
        CBCentralManager) {
    }

    func centralManager(_ central: CBCentralManager,

```

```

    didDiscover peripheral: CBPeripheral, advertisementData:
    [String: Any], rssi RSSI: NSNumber) {
}

func centralManager(_ central: CBCentralManager, didConnect
    peripheral: CBPeripheral) {
}
}

```

Connecting to a Bluetooth LE Peripheral

Now that the `CBCentralManager` object is initialized correctly, you are ready to establish a connection to a Bluetooth LE peripheral. To begin, update the `connect()` method, as shown in Listing 6-16, to start or stop scanning for what state the app is in when the Connect button is pressed.

Listing 6-16. Starting or Stopping Scanning for Bluetooth LE Peripherals

```

class BluetoothService: NSObject {
    ...
    var centralManager: CBCentralManager?
    var connectedPeripheral: CBPeripheral?
    ...
    func connect() {
        switch connectionStatus {
        case .unknown, .disconnected :
            centralManager?.scanForPeripherals(withServices:
                nil, options: nil)
            connectionStatus = .scanning
        case .connected, .connecting:
            if let connectedPeripheral = connectedPeripheral {
                centralManager?.cancelPeripheralConnection(
                    connectedPeripheral)
                connectionStatus = .disconnected
            }
        default:
            centralManager?.stopScan()
        }
    }
}

```

```

        connectionStatus = .disconnected
    }
    delegate?.didUpdateConnection(status: connectionStatus)
}
}

```

You initiate the scan by calling the `scanForPeripherals(withServices:options:)` method on your Central Manager object. You can specify UUIDs to filter by, but in my experience, this makes scanning for devices harder, if your advertising code is not set up correctly on the Arduino, so I have omitted it in this example. Stopping scanning is equally easy, as it simply requires you to call the `stopScan()` method. To disconnect, you must call `cancelPeripheralConnection()` with a saved `CBPeripheral` object, which you will have after completing the connection.

When your Central Manager has found Bluetooth LE peripherals, it will reply back with your implementation of the `centralManager(didDiscover:advertisementData:RSSI:)` method. This method returns a `CBPeripheral` object for each device it finds, including the device's name and services from its advertising data. After you have identified the device you want to connect to, you should save its `CBPeripheral` object and attempt to connect to it. In Listing 6-17, I have updated the `BluetoothService` class's implementation of the `centralManager(didDiscover:advertisementData:RSSI:)` method, to save the device that matches the name "IOTHome" to a property in this class, and then try to connect to it.

Listing 6-17. Discovering and Saving a Bluetooth LE Peripheral

```

extension BluetoothService: CBCentralManagerDelegate {
    ...
    func centralManager(_ central: CBCentralManager,
    didDiscover peripheral: CBPeripheral, advertisementData:
    [String: Any], rssi RSSI: NSNumber) {
        if peripheral.name == "IOTDoor" {
            self.connectedPeripheral = peripheral
            self.connectedPeripheral?.delegate = self
            centralManager?.connect(peripheral, options: nil)
        }
    }
    ...
}

```

While it would be ideal to start using the peripheral after finding it, there is still a bit more effort required to retrieve data from the device. Once the connection is established, the `centralManager(didConnect:)` method will fire. Inside this method, you must begin scanning for the services you want to use on the peripheral (door and battery status). After you have connected to the IOTHome peripheral, you no longer have to scan for devices, so you should tell the Central Manager to stop scanning and update the status of the connection status property for the `BluetoothService` class.

Similar to the `CBCentralManager`, the `CBPeripheral` sends its updates back through delegate methods. In Listing 6-18, I have updated the `BluetoothService` class to include the completed `centralManager(didConnect:)` method and a stub for the `CBPeripheralDelegate` protocol.

Listing 6-18. Sending Updates After Connecting to a Peripheral

```
extension BluetoothService: CBCentralManagerDelegate {
    func centralManagerDidUpdateState(_ central:
        CBCentralManager) {
        switch(central.state) {
            case .poweredOn:
                NSLog("It's showtime")
            default:
                NSLog("Device is not ready")
        }
    }
    ...

    func centralManager(_ central: CBCentralManager, didConnect
        peripheral: CBPeripheral) {
        central.stopScan()
        connectedPeripheral?.discoverServices(doorServices)
        self.connectionStatus = .connecting
        delegate?.didUpdateConnection(status: connectionStatus)
    }
}
```

```

extension BluetoothService: CBPeripheralDelegate {
  func peripheral(_ peripheral: CBPeripheral,
    didDiscoverServices error: Error?) {
  }

  func peripheral(_ peripheral: CBPeripheral,
    didDiscoverCharacteristicsFor service: CBService,
    error: Error?) {
  }

  func peripheral(_ peripheral: CBPeripheral,
    didUpdateValueFor characteristic: CBCharacteristic,
    error: Error?) {
  }
}

```

After connecting to the device, the `CBPeripheral` object will respond with the services it has discovered. Reflecting on the hierarchical nature of Bluetooth LE communication, after finding the services, you must inquire about the characteristics they reveal. As shown in Listing 6-19, this is accomplished by calling `discoverCharacteristics()` on the `CBPeripheral` object.

Listing 6-19. Interrogating Services and Characteristics for a Bluetooth LE Peripheral

```

extension BluetoothService: CBPeripheralDelegate {
  func peripheral(_ peripheral: CBPeripheral,
    didDiscoverServices error: Error?) {
    guard let services = peripheral.services
    else { return }
    for service in services {
      peripheral.discoverCharacteristics(
        doorCharacteristics, for: service)
    }
  }

  func peripheral(_ peripheral: CBPeripheral,
    didDiscoverCharacteristicsFor service: CBService, error:

```



```

    Error?) {
        self.connectionStatus = .connected
        delegate?.didUpdateConnection(status: connectionStatus)
    }
    ...
}

```

For now, inside the `peripheral(didDiscoverCharacteristicsFor:error:)` method, you can set the connection state to `connected` and notify the `BluetoothService` delegate that the connection has been established.

Monitoring Characteristic Updates

Having discovered what the peripheral is capable of, the last remaining steps are to indicate that the `BluetoothService` object should receive updates when the characteristic values change and to pass these changes along to the `BluetoothService` delegate. To register for the updates, you will have to modify the `peripheral(didDiscoverCharacteristicsFor:error:)` method. The method returns a `CBSERVICE` object containing valid references to all of the available services on the device. Use the `characteristics` property to extract these values from the service, then use the `setNotifyValue(enabled:For:)` method on the `CBPeripheral` object to indicate that you want to receive the updates, as shown in Listing 6-20. To limit the number of messages that are received, use the known UUIDs for the battery and lock characteristics.

Listing 6-20. Registering for Characteristic Updates

```

extension BluetoothService: CBPeripheralDelegate {
    ...
    func peripheral(_ peripheral: CBPeripheral,
        didDiscoverCharacteristicsFor service: CBSERVICE,
        error: Error?) {
        guard let characteristics = service.characteristics
            else { return }
        for characteristic in characteristics {
            if characteristic.uuid == doorCharUUID {
                self.connectedPeripheral?.setNotifyValue(true,

```

```

        for: characteristic)
    }

    if characteristic.uuid == battCharUUID {
        self.connectedPeripheral?.setNotifyValue(true,
            for: characteristic)
    }
}
self.connectionStatus = .connected
delegate?.didUpdateConnection(status:
    connectionStatus)
}
}

```

To handle the characteristic updates, implement the `peripheral(didUpdateValueFor:error:)` method. This method returns a `CBCharacteristic` object, including the raw binary data from the characteristic object. Since the updates were transmitted from the Arduino device as character strings, in this method, you should try to convert the raw data to strings. The easiest way to do this is with the `String` class's `String(data:encoding:)` constructor method. Generally, UTF-8 is the safest encoding to use when parsing plain text English, but you can also try Unicode encoding, if you plan on working with non-English characters.

Because the `IOTHome` app should be able to handle door or battery updates independently of each other, you should pass their updates via separate methods in the `BluetoothServiceDelegate` protocol. In Listing 6-21, I have implemented the `peripheral(didUpdateValueFor:error:)` method with this logic. When the updates come in, I check which characteristic UUID they are originating from, convert the data to a string, and then call the `didReceiveDoorUpdate(value:)` method or `didReceiveBatteryUpdate(value:)` method.

Listing 6-21. Handling Characteristic Updates

```

extension BluetoothService: CBPeripheralDelegate {
    ...
    func peripheral(_ peripheral: CBPeripheral,
        didUpdateValueFor characteristic: CBCharacteristic,
        error: Error?) {

```

```

guard let characteristicData = characteristic.value
    else { return }
if characteristic.uuid == doorCharUUID,
    let stringValue = String(data: characteristicData,
        encoding: String.Encoding.utf8) {
        delegate?.didReceiveDoorUpdate(value: stringValue)
    }
if characteristic.uuid == battCharUUID,
    let stringValue = String(data: characteristicData,
        encoding: String.Encoding.utf8) {
        delegate?.didReceiveBatteryUpdate(value:
            stringValue)
    }
}
}
}

```

Finally, to update the user interface when the updates come in, switch to the `DoorViewController` class. As shown in Listing 6-22, inside the `didReceiveDoorUpdate(value:)` and `didReceiveBatteryUpdate(value:)` methods, update the status labels with the new values.

Listing 6-22. Updating the User Interface After Receiving Characteristic Updates

```

extension DoorViewController: BluetoothServiceDelegate {
    func didReceiveDoorUpdate(value: String) {
        let statusString = (value == "1") ? "Locked" :
            "Unlocked"
        let dateString = dateFormatter.string(from: Date())
        statusLabel?.text = statusString
        lastUpdatedLabel?.text = "(Last Updated: \(dateString))"
    }
    func didReceiveBatteryUpdate(value: String) {
        batteryLabel?.text = "Battery Level: \(value)%"
    }
}
}

```

Monitoring Updates While the App Is in the Background

As a nice finishing touch on the IOTHome app, you can make the app present a notification on the user's phone when characteristic updates come in while the app is backgrounded. This way, even if users do not have the IOTHome app open, they can still find out if their door was opened or the sensor's battery level is low. To add this feature, you must take advantage of two features from iOS's notification framework, `UserNotifications`, requesting notification permission from the user and scheduling a notification when the Bluetooth characteristic updates come in.

To present the notification permission dialog, you must call the `requestAuthorization(options:completionHandler:)` method on the `UNUserNotificationCenter` object, which represents the user's device. Similar to location services, this is a singleton object shared by all iOS apps. You can access it by calling the `current()` method on the `UNUserNotificationCenter` class. In Listing 6-23, I have made the call to request the permission, by adding a `viewWillAppear()` implementation to the `DoorViewController` class. The message for the permission prompt comes from the value you specified in the project's `Info.plist` file for the `NSBluetoothPeripheralUsageDescription` key-value pair.

Listing 6-23. Requesting Notification Permission from the Door View Controller

```
import UIKit
import UserNotifications

class DoorViewController: UIViewController {
    ...
    override func viewWillAppear(_ animated: Bool) {
        super.viewWillAppear(animated)

        let center =
            UNUserNotificationCenter.current()

        center.requestAuthorization(
            options: [.alert, .sound]) { (completed: Bool,
            error: Error?) in
                NSLog("Notification request completed with status:
                \(completed)")
            }
        }
    }
}
```

The `viewWillAppear()` method is called when the view controller is presented, either by switching tabs or navigating back to the view controller from another view controller in a navigation stack. The permission dialog will only appear the first time the user opens the Door View Controller. It should look like the screenshot in Figure 6-10.

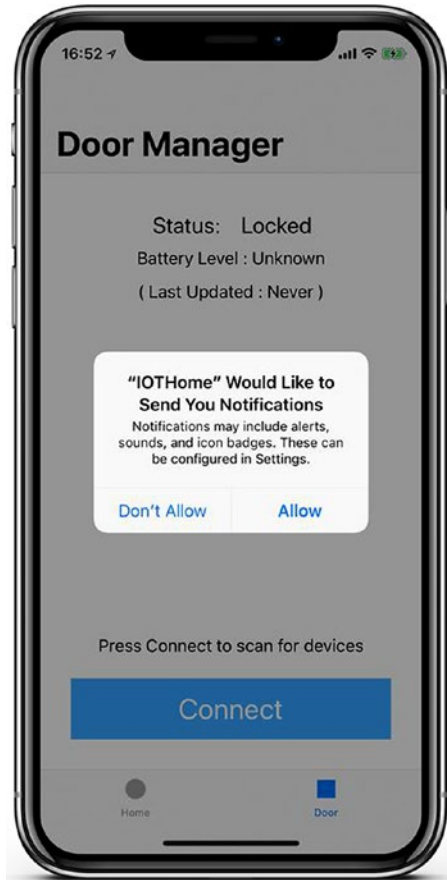


Figure 6-10. Notification permission dialog for the IOTHome app

To present the notification, you must create a notification request, using content information, including a title, message, and sound, and a trigger, such as time or a location update. For the `DoorViewController` class, I have enclosed this logic in a method called `scheduledLocalNotification()`, which takes strings indicating the update type and value as its input. The trigger is set to fire half a second after the value is received, as I want the user to know immediately when his or her door has opened.

Because you requested Bluetooth central manager permission for the app at the beginning of this section, you can use that to your advantage to schedule the notification request. The background permission allows the IOTHome app a couple of seconds of executions upon every characteristic update. As shown in Listing 6-24, in the `DoorViewController` class's `BluetoothService` delegate methods, you can add a call to schedule a local notification if the app is backgrounded. The check for whether the app is backgrounded is good for user experience, as the user does not have to see notifications when the app is open. The notifications will continue to be delivered until the Bluetooth connection is disconnected by the user or a physical limitation (for example, distance or insufficient power).

Listing 6-24. Scheduling a Background Notification When Characteristic Updates Are Received

```
extension DoorViewController: BluetoothServiceDelegate {
    func didReceiveDoorUpdate(value: String) {
        ...
        let state = UIApplication.shared.applicationState
        if state == .background {
            scheduleLocalNotification(updateType: "Door",
                updateValue: value)
        }
    }
}

func didReceiveBatteryUpdate(value: String) {
    ...
    let state = UIApplication.shared.applicationState
    if state == .background {
        scheduleLocalNotification(updateType:"Battery level",
            updateValue: "\(value)%")
    }
}
...

```

```

func scheduleLocalNotification(updateType: String,
updateValue: String) {
    let center = UNUserNotificationCenter.current()

    let content = UNMutableNotificationContent()
    content.title = "IOTHome device update"
    content.body = "\{updateType\} is now \{updateValue\}"
    content.sound = UNNotificationSound.default()

    let trigger =
        UNTimeIntervalNotificationTrigger(
            timeInterval: 0.5,
            repeats: false)

    let request =
        UNNotificationRequest(identifier:
            "IOTHomeNotification", content: content,
            trigger: trigger)

    center.add(request) { (error: Error?) in
        if let errorObject = error {
            NSLog("Error scheduling notification:
                \{errorObject.localizedDescription\}")
        } else {
            NSLog("Notification scheduled successfully")
        }
    }
}
}

```

Summary

In this chapter, you learned how to implement both roles of Bluetooth LE: the peripheral (the device that offers information) and the central manager (the device that fetches information from one or more peripherals). The ESP32-based circuit you started in Chapter 5 served as the peripheral, powered by the open source ESP32_BLE_Arduino library, and the IOTHome iOS served as the central manager, using the Core Bluetooth framework. For both roles, after going through some lengthy setup steps, it became quite easy to transfer data using Bluetooth LE characteristic updates. To help the user reduce his/her battery usage, you also configured both devices to send and respond to updates only when the device experienced a noticeable state change (for example, a 10% reduction in the door opening or battery level since the last update).

Bluetooth continues to be one of the most popular protocols for communicating with hardware accessories. My hope is that you will be able to reuse most of what you learned in this chapter for future updates to the protocol and on other devices you may choose to build!

CHAPTER 7

Setting Up a Raspberry Pi and Using It As a HomeKit Bridge

At this point in your journey through Internet of Things (IoT) app development, you have learned how to use sensors on iOS devices (GPS, motion) and how to interact with third-party hardware devices, such as Arduino, using open communication protocols, including Bluetooth. In this chapter, you will learn about an Apple-specific IoT technology that lies somewhere in between both of these: HomeKit.

HomeKit was introduced in 2014 with iOS 8, as Apple's proprietary standard to enable iOS devices to communicate with certified third-party IoT accessories for the home, such as smart lightbulbs and air conditioners. The sales pitch was that through Apple's hardware certification process, special encryption chip for IoT devices, and deep integration with iOS, it would be able to deliver the strongest, most secure platform for IoT in the home. Rather than buying a special hardware device to serve as the gateway for your devices, you could use an iPad, HomePod, or Apple TV in your home to serve this purpose. Additionally, you would be able to use Siri to check on the status of your devices.

Unfortunately, the certification process and hardware chip proved to be too expensive and too late for many third-party hardware manufacturers, and the platform never achieved the momentum that was expected of it. While HomeKit has not proven itself to be the instant commercial status Apple hoped for, more compatible devices are being released for it every day. And, more important, Apple now allows hobbyists to create noncertified HomeKit devices for their personal use, through its release of a noncommercial version of the HomeKit Accessory Protocol (HAP) specification. This license allows you to create accessories for your personal use, which will appear as

“non-certified” when you connect to them through an iOS device. To make things even better, your personal devices can be built on anything that can implement HAP, from a Raspberry Pi to a Mac.

On the software side, as a part of iOS, Apple provides a HomeKit framework that allows you to manage rooms and devices registered in the system’s HomeKit database. Released at the same time as HealthKit, HomeKit aims to operate in the same manner, by providing a protected, system-wide database of HomeKit devices that any user-permitted application can access. However, one major component Apple does not provide is an implementation of the HAP for use on hardware devices. To fill this niche, in this chapter, you will use an excellent open source project, HomeBridge (<https://github.com/nfarina/homebridge>), which implements most of the HAP specification and provides a plug-in system that makes it easy to connect other popular IoT services and accessories to your project.

Learning Objectives

In this chapter, you will use a Raspberry Pi to build a HomeKit bridge for the door sensor you created in Chapters 5 and 6 and you will learn how to register it as a valid HomeKit device in iOS. The Raspberry Pi will connect to the sensor via Bluetooth to read its status and report it back via the HomeKit Accessory Protocol (HAP). As the name implies, a HomeKit bridge can connect multiple devices to HomeKit through a single interface. To demonstrate the bridge functionality of a device, you will also connect a temperature sensor to the Raspberry Pi and report its status over HAP.

While there is an API that allows you to manage HomeKit devices and rooms within your app, after the publication of the first edition of this book, I found those features being adopted less in HomeKit devices, as they mirror functions in Apple’s Home app. For this edition, I decided to forego those topics in favor of expanding on configuring the Raspberry Pi, as it is such an involved process.

In deploying the HomeKit bridge, you will learn the following key concepts for iOS IoT application development:

- Setting up a Raspberry Pi
- Installing Linux packages, such as Node.js, and their dependencies
- Installing and configuring HomeBridge
- Registering devices through the Home app
- Debugging HomeBridge

For this chapter, you will use the Raspberry Pi as your hardware development platform. Like the Arduino, the Raspberry Pi is a popular open source hardware platform. However, unlike the Arduino, there are only a few officially supported Raspberry Pi devices, manufactured by the Raspberry Pi Foundation (www.raspberrypi.org), and a Raspberry Pi is intended to run Linux and provide desktop computer-like functionality, whereas an Arduino is intended to power sensors. For this reason, the Raspberry Pi is called a *single-board computer*. As a full Linux computer, you can run most Linux packages. For this chapter, you will take advantage of Node.js to run the HomeBridge service and its plug-ins. Node.js is a runtime environment that allows you to run JavaScript programs from the command line and is frequently used today as a replacement for web servers and startup utilities written in such compiled languages as C.

Depending on your needs, you may eventually want to look into using a BeagleBone or Asus Tinker Board as a replacement for the Raspberry Pi, but I find the Raspberry Pi's setup process and user community to be the friendliest for beginners.

As always, you can find the iOS code for this project under the Chapter 7 folder of the GitHub page for this book (<https://github.com/Apress/program-internet-of-things-w-swift-for-ios>). The configuration scripts for the Raspberry Pi part of the project are included in the Pi subfolder under the Chapter 7 folder.

Setting Up the Raspberry Pi HomeKit Bridge

In this chapter, you will use a Raspberry Pi to serve as the HomeKit bridge for the IOTHome door sensor and a temperature sensor you will connect directly to the Raspberry Pi. This will allow the user to access both statistics via voice commands in Siri and the companion app you will build later in the chapter.

While there will be a hardware component to this chapter, much of it will piggyback off of what you learned in Chapter 5. Unlike the Arduino, you will not have to install a special IDE to connect to the Raspberry Pi. However, you will have to spend a little bit more time on the setup, to get HomeBridge working correctly.

Putting Together the Hardware

For this project, the hardware requirements are a Raspberry Pi capable of running HomeBridge and a temperature sensor to read the temperature of the room the Pi is located in. In Table 7-1, I have included the list of parts that I used when putting together the circuit for this project.

Table 7-1. *List of Parts for the HomeBridge Project*

Part Name	Quantity	Mouser Part #
Raspberry Pi 3 (or newer)	1	RPI3-MODBP-BULK
Solderless breadboard	1	854-BB170-WH
Breadboard jumper wire pack	1	713-110990049
DHT22 temperature sensor	1	485-385
microSD USB reader/writer	1	485-939
16GB (or larger) microSD memory card	1	467-SDSDQAD-016G

To revisit Chapters 5 and 6 for a second, one of the greatest benefits to come out of the popularity of IoT applications in the last few years is the increased availability to hobbyists of low-cost sensors. As you will notice in the part list, I selected the DHT22 temperature/humidity sensor module for our project. In one package, it bundles the integrated circuit for the humidity sensor, its supporting parts (for example, capacitors, resistors), and provides a simple three-pin interface consisting of a power pin, ground pin, and data pin. While you can hook up the sensor directly to the Pi’s header pins, I recommend using a breadboard to make the circuit easier to move around and reconfigure in the future.

In the part list, I have specified “any modern, wireless-enabled Raspberry Pi.” In order to run HomeBridge, not only must your host device be able to run a Node.js server, it also requires Wi-Fi and Bluetooth, to communicate with your HomeKit hub (an iPad, HomePod, or Apple TV in your home). While the scripts and instructions provided in this chapter will run on any Raspberry Pi capable of running a recent distribution of the Raspbian Linux distribution, the Raspberry Pi 3 and Raspberry Pi Zero W were the first devices to include Bluetooth and Wi-Fi onboard. If you have a Raspberry Pi 2 or Pi Zero, on which you have already configured USB Bluetooth and Wi-Fi modules, you are more than welcome to use them here.

On a related note, if you are not familiar with how the Raspberry Pi works, it requires a microSD card to run its system image. It comes with a small bootloader, but it is only capable of starting up the image on your microSD card. For the power supply, you can use any device capable of outputting 5V DC, such as the USB port on a computer, a portable battery, or a USB wall wart power adapter.

When collected, your parts should resemble those in the photo in Figure 7-1. I used a Raspberry Pi 3 for my implementation, as I think it is the easiest to acquire, and its large size makes it very easy to work with.

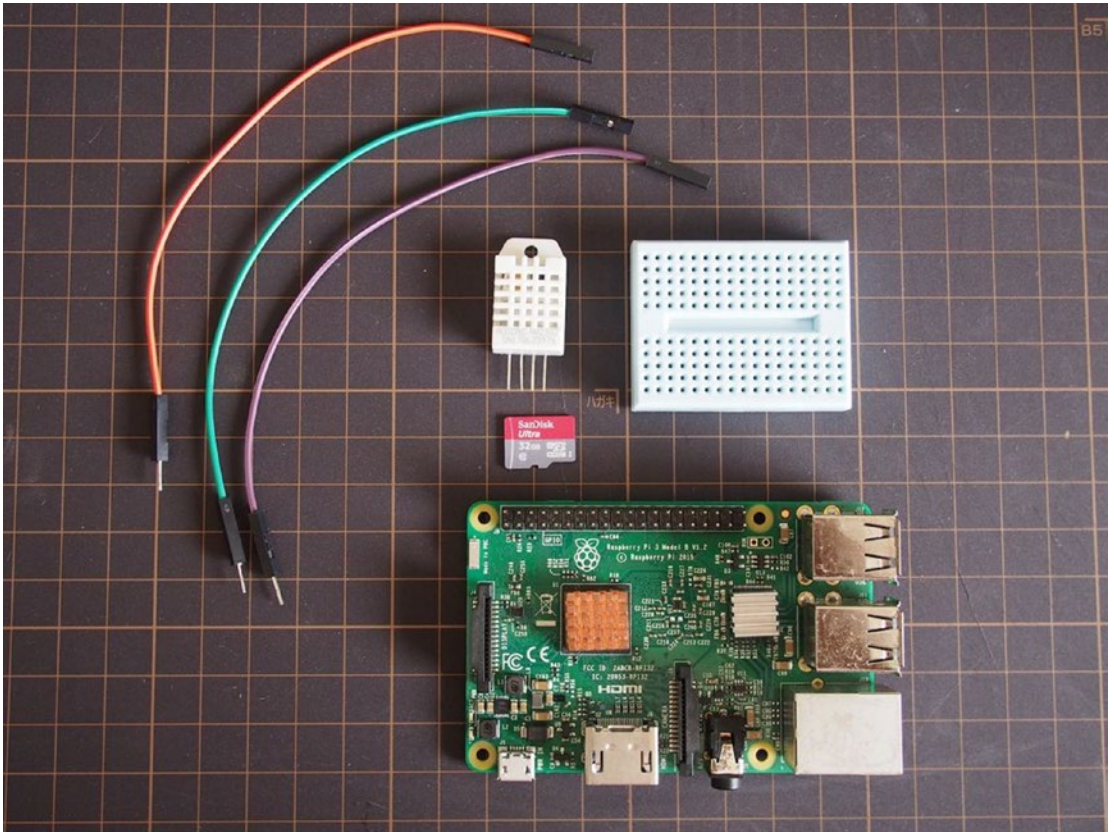


Figure 7-1. Collected parts for the IOTHome project

Assembling the Circuit

Putting together the circuit for this project is very straightforward. As shown in Figure 7-2, you primarily have to connect the temperature sensor's VCC and GND pins to those on the Pi, then connect the DATA pin to any available general-purpose input/output (GPIO) pin on the Raspberry Pi. For this project, I chose GPIO21 (Pin 40).

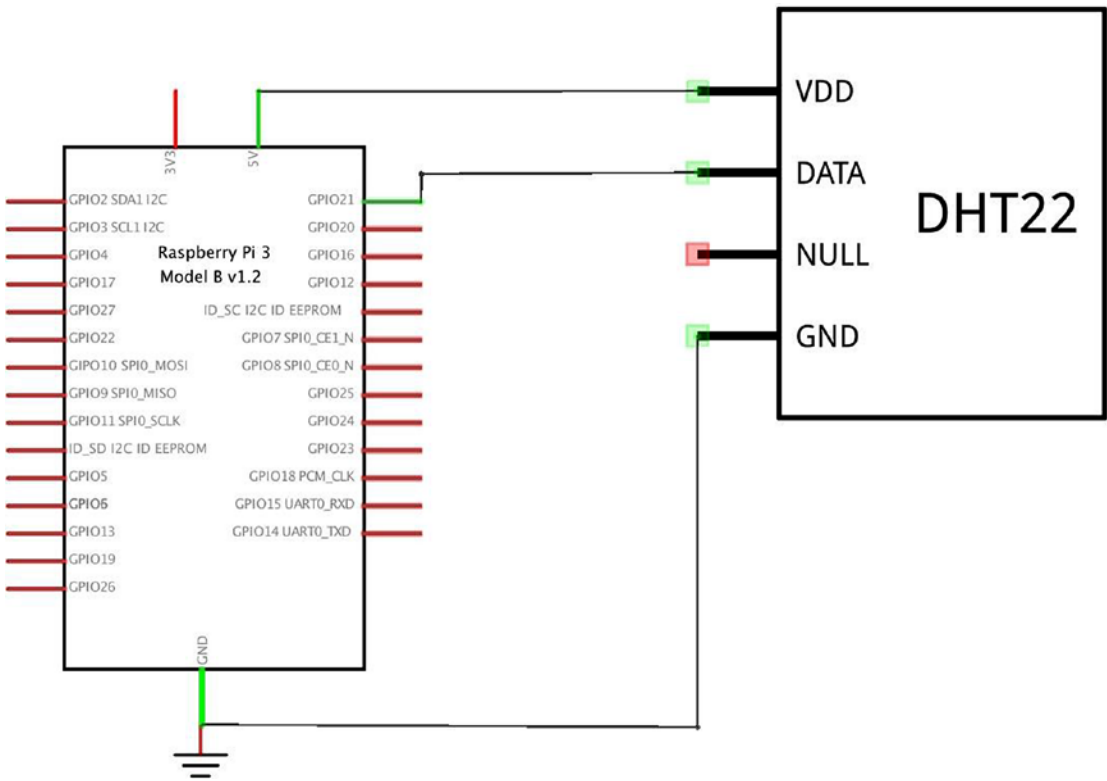


Figure 7-2. Schematic for the IOTHome project

To make the connections, I used male-to-female header cables. I connected the female ends to the header on the Pi and the male ends into the breadboard. From there, I connected the DHT22 temperature sensor directly to the breadboard. I have attached a photograph of my completed circuit in Figure 7-3. Once again, I used color-coded header cables and breadboard jumper wires to help me quickly debug the circuit.

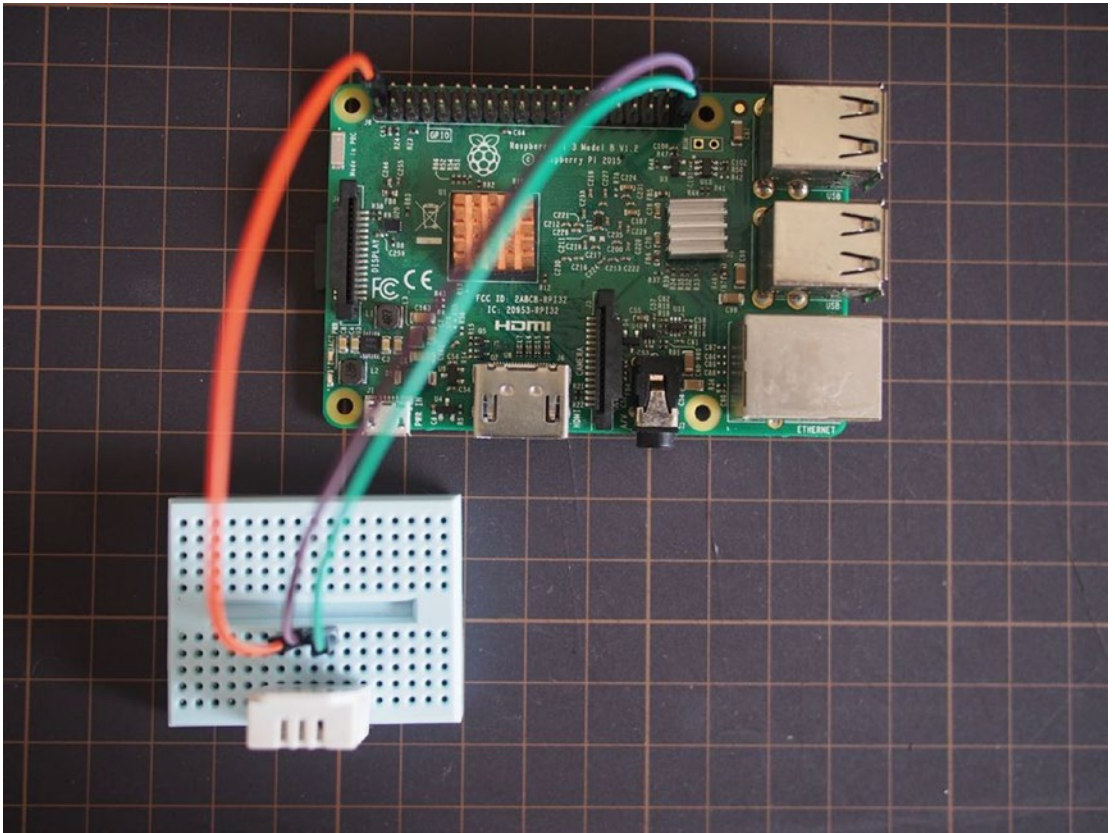


Figure 7-3. Photograph of completed circuit for the IOTHome project

Bootstrapping the Raspberry Pi

Now that the helper circuit is completed for the project, you can begin to bootstrap the Raspberry Pi. *Bootstrapping* is a term used in Linux and embedded systems to refer to preparing a system to turn on for the first time (“bringing it up by the bootstraps”).

As I mentioned earlier, the bootloader on the Raspberry Pi is only capable of booting up whatever is on the microSD card inserted into the device. For this project, you will use the Raspbian Linux distribution, provided by the Raspberry Pi Foundation, as the operating system for the Raspberry Pi. If you have ever used Ubuntu on a desktop


computer before, you will be very familiar with how Raspbian works, as it shares the same common relative: Debian Linux. Both distributions share the same package manager and architecture as Debian, with slight modifications for their intended use cases.

To install Raspbian, the first step is to download a pre-built image file containing a pre-built, bootable instance of Raspbian from the Raspberry Pi Foundation's web site (www.raspberrypi.org/downloads/raspbian/). As shown in Figure 7-4, from the Downloads page, select Raspbian, Raspbian Stretch with Desktop, and then Download ZIP. The major difference between the Desktop and Lite distributions is that the Lite distribution sacrifices the GNOME Desktop graphical user interface, providing you with a command-line interface only. If you are comfortable with the command line, you are welcome to install the Lite version, but I recommend the full Desktop version, because GNOME makes the Pi easier to configure and reuse for other purposes later.


1

DOWNLOADS

Raspbian is our official operating system for **all** models of the Raspberry Pi. Download it here, or use **NOOBS**, our easy installer for Raspbian and more.



NOOBS



RASPBIAN

2

Raspberry Pi Desktop (for PC and Mac)

Debian with Raspberry Pi Desktop is the Foundation's operating system for PC and Mac. You can create a live disc, run it in a virtual machine, or even install it on your computer.


3

RASPBIAN

Raspbian is the Foundation's official supported operating system. You can install it with [NOOBS](#) or download the image below and follow our [installation guide](#).

Raspbian comes pre-installed with plenty of software for education, programming and general use. It has Python, Scratch, Sonic Pi, Java, Mathematica and more.


The Raspbian with Desktop image contained in the ZIP archive is over 4GB in size, which means that these archives use features which are not supported by older unzip tools on some platforms. If you find that the download appears to be corrupt or the file is not unzipping correctly, please try using [ZZip](#) (Windows) or [The Unarchiver](#) (Macintosh). Both are free of charge and have been tested to unzip the image correctly.



RASPBIAN STRETCH WITH DESKTOP
Image with desktop based on Debian Stretch

Version: April 2018
Release date: 2018-04-18
Kernel version: 4.14
Release notes: [Link](#)

[Download Torrent](#) [Download ZIP](#)



RASPBIAN STRETCH LITE
Minimal image based on Debian Stretch

Version: April 2018
Release date: 2018-04-18
Kernel version: 4.14
Release notes: [Link](#)

[Download Torrent](#) [Download ZIP](#)

SHA-256: 0e2922e551a895b136f2ea83d1bc0ca71e016e6d50244ba3da52bd73b242dc9664df5d1b6

SHA-256: 5a0747b2bfb8c8664192831b7dc5b22847718a1cb77639a1f3db368

Figure 7-4. Downloading a Raspbian image from the Raspberry Pi web site

Once the zip file has completed downloading, you will have to write it to the microSD card in a way that allows it to be recognized by the bootloader as a valid disk image. Disk images package the files for an operating system and any documents and configuration files that the creator feels are useful to consumers of the image. For Linux distributions, they provide a useful way to share a distribution with end users without forcing them to go through a lengthy, sensitive installation process. A skilled engineer builds a system he or she feels is an appropriate starting place for most users and then shares it. However, if the disk image is not unpacked and installed in the correct way, it will be unusable by the target computer. The tool I prefer for burning Raspberry Pi images to microSD cards is Etcher (<https://etcher.io>). It has an easy-to-understand user interface and high reliability rate. As shown in Figure 7-5, after installing Etcher, insert the microSD card into the adapter on your Mac and open Etcher. From its single-screen user interface, select the ZIP file for the Raspbian image and the drive corresponding to the microSD card, then select Flash! to begin the image-burning process. If your microSD card is not appearing in Etcher, please check that both the microSD card and adapter are securely attached to your Mac.

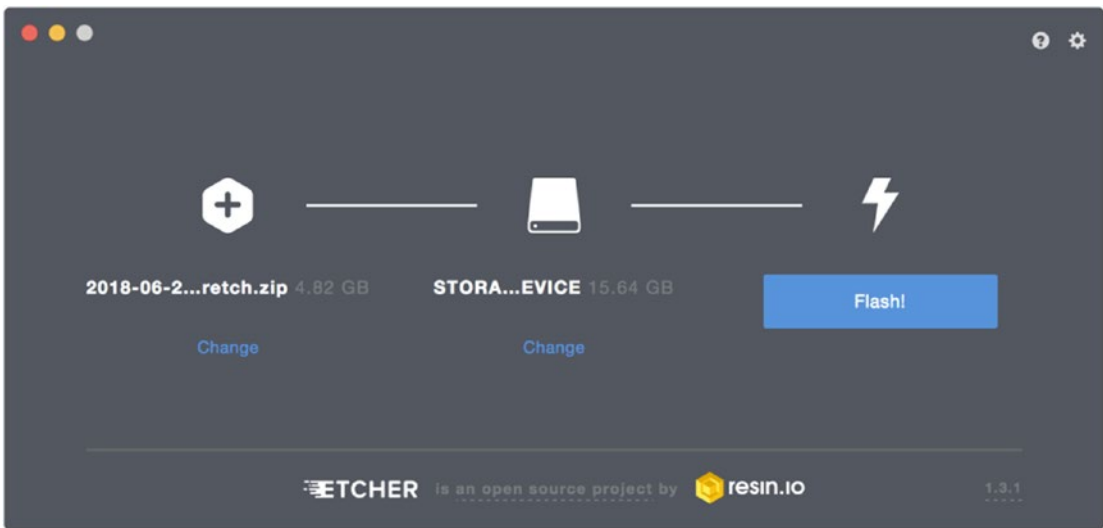


Figure 7-5. *Burning a Raspbian image to a microSD card using Etcher*

Etcher will play a sound and present a pop-up to inform you that the disk image has been written successfully. At this time, it is safe to remove the microSD card from your Mac and insert it into the slot at the bottom of the Raspberry Pi, as shown in Figure 7-6. At this time, you should also plug an HDMI-based monitor and USB keyboard/mouse into the Raspberry Pi.

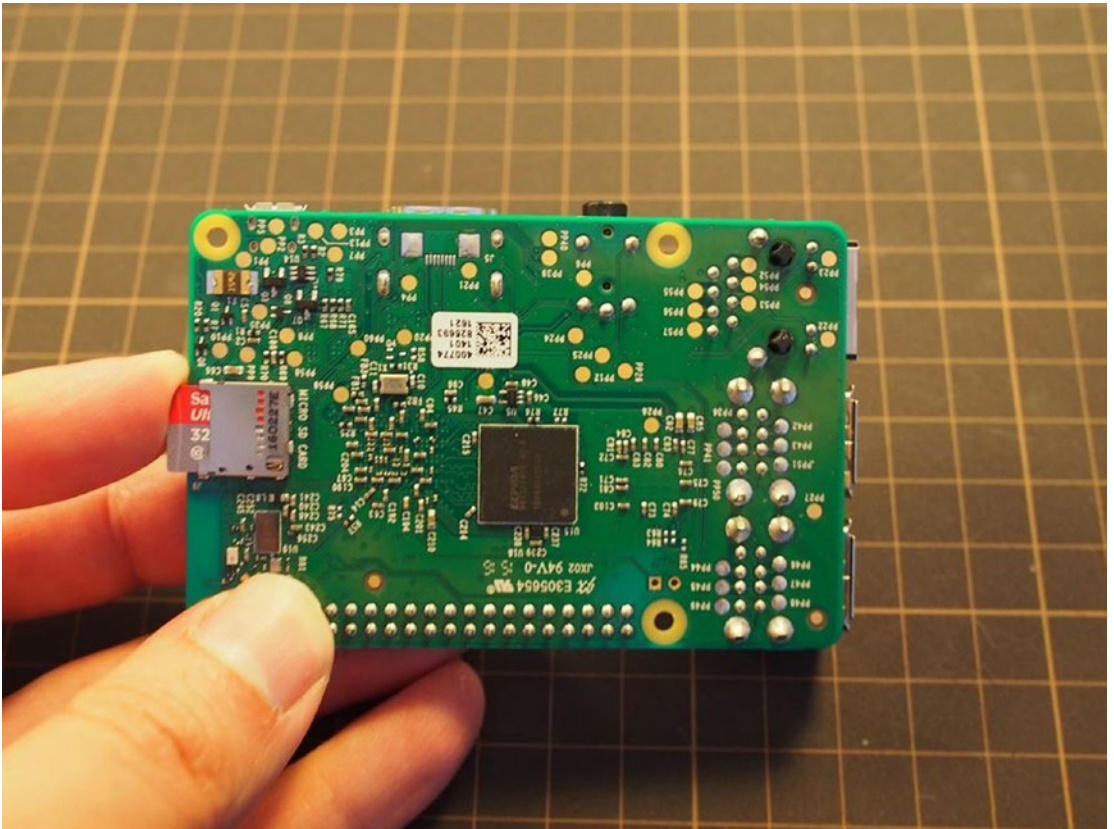


Figure 7-6. *Inserting a microSD card into the Raspberry Pi*

Once all of these preparations are complete, connect your power source to the USB port marked PWR and wait a minute or two for the GNOME desktop to boot up. When the desktop is ready, the image on your monitor should resemble Figure 7-7.

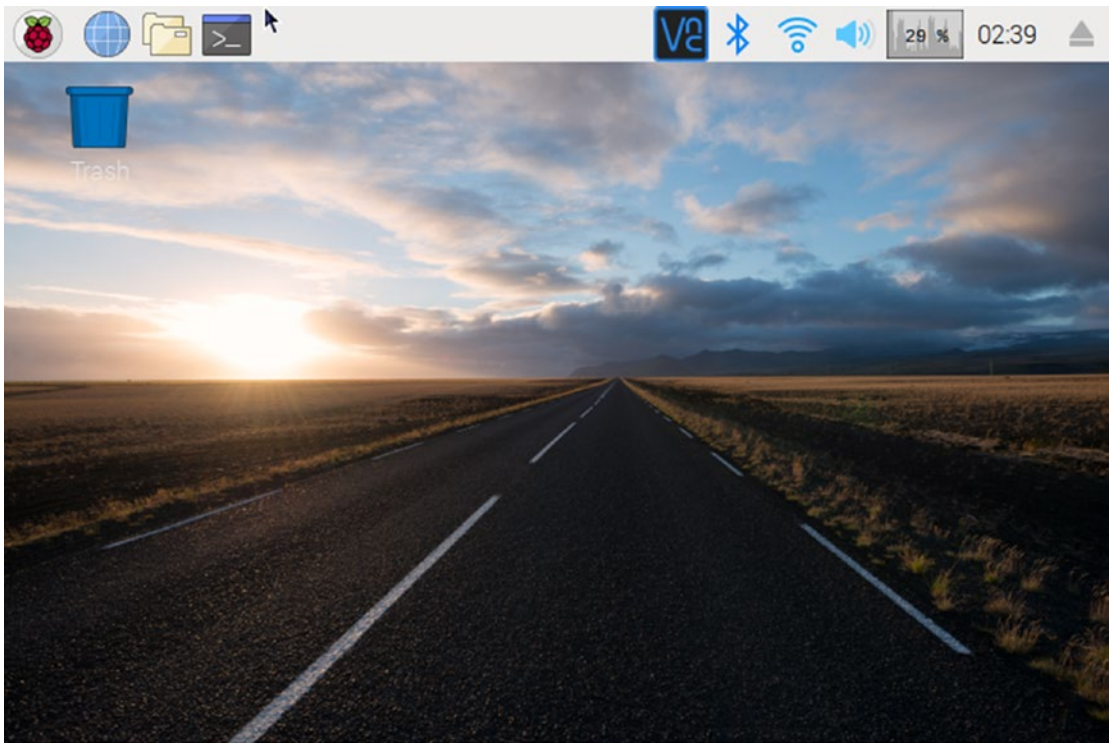


Figure 7-7. Screenshot of desktop for freshly installed Raspbian distribution

As shown in Figure 7-8, find the Wi-Fi icon at the top right of the system tray, then click it to present the network selection drop-down menu. Select your network, then enter in the password when prompted.

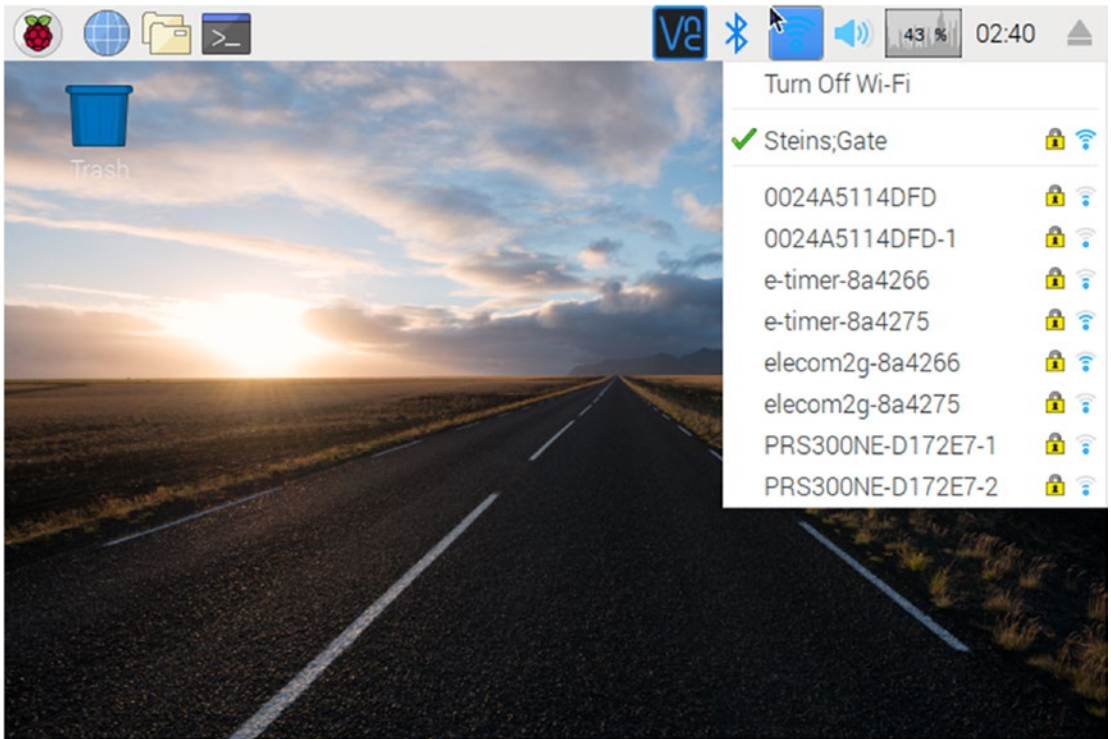


Figure 7-8. *Selecting a WiFi network from the Raspbian desktop*

For the final configuration step, you must enable the hardware interfaces on the Pi, using the Raspberry Pi configuration tool. This will allow you to access the I2C, SPI, and other hardware communication libraries that are preinstalled on Raspbian but disabled by default for security reasons. As shown in Figure 7-9, to find this tool, click the raspberry icon at the top-left of the system tray, navigate to Preferences, and then select Raspberry Pi Configuration. From the application that appears, click the Interfaces tab, then click the SPI, I2C, and 1-Wire check boxes, to enable them. After you click OK, you will be asked to restart the Pi to save the settings.

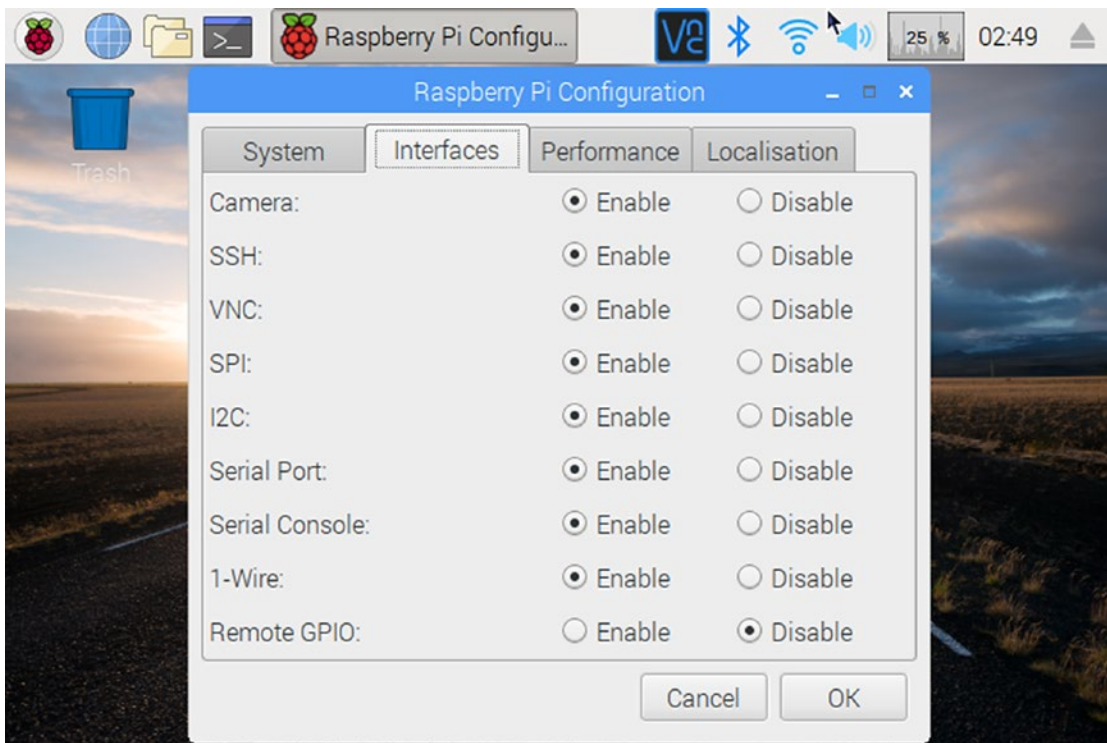


Figure 7-9. Enabling hardware interfaces on the Raspberry Pi

If you want to use a remote desktop client such as RealVNC or TightVNC to view your Pi's desktop without connecting to a monitor, you can enable this in the configuration tool as well. The next time the Pi restarts, the RealVNC server application will start up and prompt you to configure the VNC settings for the device.

Installing HomeBridge

Having completed the hardware setup phase of the project, you can now shift your focus to installing HomeBridge and its dependencies, the second-to-last step before you can start using the Raspberry Pi as a HomeKit bridge.

As mentioned at the beginning of the chapter, HomeBridge and its core dependency, HAP-NodeJS (the library that implements the HomeKit Accessory Protocol), run as Node.js applications. You must begin to set up this process by installing Node.js. Although the distribution of Node.js provided by the default package manager (`apt-get`) is very stable and fits most average use cases, HomeBridge requires a slightly newer, more powerful version of Node.js, which you will have to install yourself. To begin this

process, navigate to the Node.js distribution web site (<http://nodejs.org/dist/>) and select the latest version of Node.js 8 (<http://nodejs.org/dist/latest-v8.x/>). Under this directory, you will be presented with several different files that you can download whose names vary by operating system and processor architecture (for example, x86, ARMv6). These indicate what the targets the enclosing binaries were compiled for. For the Raspberry Pi, you should search for the latest Linux/ARMv6L archive. At the time of writing, the URL for this file is <http://nodejs.org/dist/latest-v8.x/node-v8.11.4-linux-armv6l.tar.gz>.

Although you can download the archive via a browser, I recommend using the OS X Terminal. By default, your Raspberry Pi will advertise its hostname (`raspberrypi.local`) over Bonjour. You can connect to it in the Terminal by attempting to log in to SSH with the `pi` user.

```
ssh pi@raspberrypi.local
```

As shown in Figure 7-10, after connecting to the device, download the file using the `wget` command.

```
wget http://nodejs.org/dist/latest-v8.x/node-v8.11.4-linux-armv6l.tar.gz
```

```

abakir — pi@raspberrypi: ~ — ssh pi@raspberrypi.local — 85x24
pi@raspberrypi:~ $ wget http://nodejs.org/dist/latest-v8.x/node-v8.11.4-linux-armv6l
[.tar.gz
--2018-08-17 10:55:42-- http://nodejs.org/dist/latest-v8.x/node-v8.11.4-linux-armv6l
.tar.gz
Resolving nodejs.org (nodejs.org)... 104.20.22.46, 104.20.23.46, 2400:cb00:2048:1::68
14:162e, ...
Connecting to nodejs.org (nodejs.org)|104.20.22.46|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 17039444 (16M) [application/gzip]
Saving to: 'node-v8.11.4-linux-armv6l.tar.gz'

node-v8.11.4-linux- 100%[=====] 16.25M 1.27MB/s in 14s

2018-08-17 10:55:56 (1.19 MB/s) - 'node-v8.11.4-linux-armv6l.tar.gz' saved [17039444/
17039444]

pi@raspberrypi:~ $
pi@raspberrypi:~ $
pi@raspberrypi:~ $
pi@raspberrypi:~ $
pi@raspberrypi:~ $
pi@raspberrypi:~ $
pi@raspberrypi:~ $
pi@raspberrypi:~ $

```

Figure 7-10. Downloading Node.js via the Raspberry Pi terminal

Note At the time of this writing, I observed that the Node.js 8 binary for Raspberry Pi had greater compatibility with HomeBridge and the plug-ins for this chapter than the Node.js 9 binary. You are welcome to try either one!

After the download is complete, you must unpack the archive using the `tar` command. For the file indicated previously, the command I used was

```
tar -xvf node-v8.11.4-linux-armv6l.tar.gz
```

After you have unpacked the file, you must copy the Node binaries to the default location for user-installed software: `/usr/local`. This will allow HomeBridge and other Node applications you may write later to use the binaries as if they were installed by the `apt-get` package manager. For my version of Node, I used the following command to copy the files:

```
sudo cp -R node-v8.11.4-linux-armv6l/*/usr/local/
```

You can verify that your files were copied correctly by restarting your Raspberry Pi and attempting to run the Node command that queries the currently installed version:

```
node -v
```

The result should print out the version number you manually downloaded.

Before installing HomeBridge, you must install a few other packages it depends on, specifically, the latest version of C++, the development tools for Python, WiringPi, AVAHI, PiGPIO, and BCM2835. These will allow HomeBridge to interface with the GPIO pins and Bluetooth from within Node. You can easily install C++ and the Python development tools, using the `apt-get` package manager, as follows:

```
sudo apt-get install g++
sudo apt-get install libavahi-compat-libdnssd-dev
sudo apt-get install python-dev
sudo apt-get install pigpio python-pigpio
```

After running these commands, you must enable PiGPIO as a service and restart the Pi, as shown following:

```
sudo systemctl enable pigpiod.service
```


The BCM2835 package is the C library that allows WiringPi to access the GPIO pins on the Raspberry Pi and will have to be installed first. Begin by finding the latest version of the package from www.airspayce.com/mikem/bcm2835/, using the `wget` command to download it, and then `tar`, to unzip, just as you did for Node.

```
wget http://www.airspayce.com/mikem/bcm2835/bcm2835-1.56.tar.gz
tar -xvf bcm2835-1.56.tar.gz
```

Unlike Node, you will have to run BCM2835's build scripts to install it correctly. Run the following commands, in order to configure the library for your hardware, verify it, and then install it:

```
cd bcm2835-1.56
./configure
make
sudo make check
sudo make install
```

You can verify that the installation is successful from the absence of a build failure while running the last `make` command.

To install WiringPi, you will have to download it from GitHub. You can perform this option from the command line, using the following `git pull` command, which makes a local clone of the project:

```
git clone git://git.drogon.net/wiringPi
```

As with BCM2835, you will have to run build scripts to install the package on your Raspberry Pi. For WiringPi, these commands are

```
cd wiringPi/
./build
```

To verify that the packages were installed correctly, attempt to run the `gpio readall` command to echo the status of all pins on the Raspberry Pi. Your output should contain a table of pin status, similar to that in the screenshot in Figure 7-11.

```

abakir — pi@raspberrypi: ~ — ssh pi@raspberrypi.local — 98x27
pi@raspberrypi:~ $ gpio readall
-----Pi 3-----
| BCM | wPi | Name | Mode | V | Physical | V | Mode | Name | wPi | BCM | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 8 | 3.3v | | | 1 | 2 | | | 5v | | |
| 3 | 9 | SDA.1 | ALT0 | 1 | 3 | 4 | | | 5v | | |
| 4 | 7 | SCL.1 | ALT0 | 1 | 5 | 6 | | | 0v | | |
| 4 | 7 | GPIO. 7 | IN | 0 | 7 | 8 | 1 | ALT5 | TxD | 15 | 14 |
| | | 0v | | | 9 | 10 | 1 | ALT5 | RxD | 16 | 15 |
| 17 | 0 | GPIO. 0 | IN | 0 | 11 | 12 | 1 | IN | GPIO. 1 | 1 | 18 |
| 27 | 2 | GPIO. 2 | IN | 0 | 13 | 14 | | | 0v | | |
| 22 | 3 | GPIO. 3 | IN | 0 | 15 | 16 | 0 | IN | GPIO. 4 | 4 | 23 |
| | | 3.3v | | | 17 | 18 | 0 | IN | GPIO. 5 | 5 | 24 |
| 10 | 12 | MOSI | ALT0 | 0 | 19 | 20 | | | 0v | | |
| 9 | 13 | MISO | ALT0 | 0 | 21 | 22 | 0 | IN | GPIO. 6 | 6 | 25 |
| 11 | 14 | SCLK | ALT0 | 0 | 23 | 24 | 1 | OUT | CE0 | 10 | 8 |
| | | 0v | | | 25 | 26 | 1 | OUT | CE1 | 11 | 7 |
| 0 | 30 | SDA.0 | IN | 1 | 27 | 28 | 1 | IN | SCL.0 | 31 | 1 |
| 5 | 21 | GPIO.21 | IN | 1 | 29 | 30 | | | 0v | | |
| 6 | 22 | GPIO.22 | IN | 1 | 31 | 32 | 0 | IN | GPIO.26 | 26 | 12 |
| 13 | 23 | GPIO.23 | IN | 0 | 33 | 34 | | | 0v | | |
| 19 | 24 | GPIO.24 | IN | 0 | 35 | 36 | 0 | IN | GPIO.27 | 27 | 16 |
| 26 | 25 | GPIO.25 | IN | 0 | 37 | 38 | 0 | IN | GPIO.28 | 28 | 20 |
| | | 0v | | | 39 | 40 | 0 | IN | GPIO.29 | 29 | 21 |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| BCM | wPi | Name | Mode | V | Physical | V | Mode | Name | wPi | BCM |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
-----Pi 3-----

```

Figure 7-11. Querying GPIO status using the `gpio readall` command

Now, you are finally ready to install HomeBridge! To install HomeBridge, download it using the Node Package Manager (NPM).

```
sudo npm install -g --unsafe-perm homebridge
```

The installation should take several minutes to complete. Because iOS’s Home app manages a database of HomeKit devices, the easiest way to verify that you created a valid device is by attempting to register the device after all of the plug-ins you want to use have been configured. In the next two sections, I will explain the steps required to do this for reading from a temperature sensor and Bluetooth device (the door sensor you created in Chapter 6).

Caution The `-g` and `--unsafe-perm` flags ensure that Node packages are installed globally (for all applications) and without restrictions on who can execute them. The global setting is required to run HomeBridge; however, you may want to look at finding alternatives to the unsafe permissions setting, if you are worried about security threats in your home/office network.

Configuring HomeBridge to Read Data from a Temperature Sensor

As mentioned at the beginning of the chapter, one of the great things about HomeBridge is that it has a very actively maintained, easy-to-use plug-in system. To read the temperature from the DHT22 temperature/humidity sensor, you can use the `homebridge-dht` plug-in. Just as with HomeBridge itself, it is available as an NPM package. Install the plug-in globally, using the following command:

```
sudo npm install -g homebridge-dht --unsafe-perm
```

To help verify that the DHT22 sensor and plug-in are working correctly, I suggest making the binary for reading the status of the sensor available on the command line, similar to how you installed Node earlier.

```
sudo cp /usr/local/lib/node_modules/homebridge-dht/dht22 /usr/local/bin/dht22
```

Unlike Node, you must explicitly give the `dht22` binary executable permissions to run it from the command line. You can perform this operation by using the `chmod` command, along with the `a+x` parameter (add executable).

```
sudo chmod a+x./usr/local/bin/dht22
```

You can now use the `dht22` command to verify that HomeBridge will be able to access the sensor. To test this functionality, attempt to run the `dht22` command, specifying that the sensor is connected to GPIO pin 21, as shown following:

```
dht22 -g 21
```

If your circuit and HomeBridge packages are set up correctly, three numbers should print out on the terminal containing the reading count, temperature, and humidity. For my sensor, the result was as follows:

```
0 28.2 C 41.5 %
```

Note When I was debugging my sample for this project, I found that errors on this step were most often caused by incomplete PiGPIO and WiringPi installations or the DHT22 sensor not receiving enough power. I found that 5V worked best for the DHT22, while its lower-spec sibling, the DHT11, worked better with 3.3V.

After verifying that the DHT22 sensor and binary are operating correctly, you can begin writing the configuration file for HomeBridge. In addition to its plug-in system, HomeBridge allows you to easily configure the device, using a JSON file. To begin, create an empty configuration file using the following:

```
mkdir .homebridge
touch .homebridge/config.json
```

HomeBridge requires that a configuration file named `config.json` be in the invisible `.homebridge` folder of the user whose account will be employed to run the application, so please be careful to type this command exactly as written previously.

Next, use your favorite command-line text editor to open the `config.json` file. Enter the text in Listing 7-1 into this file as the configuration.

Listing 7-1. Configuration JSON File for HomeBridge (`config.json`)

```
{  "bridge": {
    "name": "IOTHome",
    "username": "CC:22:3D:E3:CD:31",
    "port": 51826,
    "pin": "031-45-154"
  },
  "description": "IOTHome HomeBridge",
  "platforms": [],
  "accessories": [
    { "accessory": "Dht",
      "name": "Indoor Comfort Sensor",
      "name_temperature": "Temperature",
      "name_humidity": "Humidity",
      "gpio": "21",
      "service": "dht22" }
  ]
}
```

To comment on this file, the bridge dictionary specifies the values that are used to expose the Raspberry Pi as a valid HomeKit bridge, specifically its device name, port, PIN, and HomeKit “username.” You may change the values as you wish, but be careful to maintain the port number and hexadecimal number format of the username.

The `accessories` dictionary specifies an array of configurations for each accessory that will be attached to the bridge. The specific key-value pairs have to come from the documentation for the plug-in you are using. For the HomeBridge DHT sensor, this documentation can be found at www.npmjs.com/package/homebridge-dht. Of special interest to the configuration of this project are the `gpio` and `service` key-value pairs. As you may have noted, `"gpio": "21"` reflects that the sensor is connected to GPIO pin 21; the `"service": "dht22"` key-value pair specifies that you are using the DHT22 sensor, as opposed to other chips in the family, such as the DHT11.

With these steps, the temperature sensor is ready to interface with your Raspberry Pi and HomeBridge. If do not wish to set up the Bluetooth connection to the door sensor or startup options for the device, you can skip to the “Connecting to Your New HomeKit Bridge” section, to start using the sensor right away.

Configuring HomeBridge to Connect to a Bluetooth LE Accessory

Another powerful plug-in available to use with HomeBridge is `homebridge-bluetooth`, which allows you to connect to a Bluetooth peripheral through HomeKit. The plug-in acts as a Bluetooth client, such as the IOTHome app from Chapter 6, and in addition to reading status, can also be used to control a device. This functionality allows you to ask Siri if “the entrance door sensor is on” and get a response based on data from a sensor you built.

Before installing the plug-in, you will have to resolve a few dependencies. First, install the `node-gyp` and `noble` Node.js modules via `npm`.

```
sudo npm install -g --unsafe-perm node-gyp
sudo npm install -g --unsafe-perm noble
```

Next, following a tip from Noble’s GitHub documentation (<https://github.com/noble/noble>), you must enable non-administrator accounts to use the Bluetooth LE utilities on the Raspberry Pi, as follows:

```
sudo setcap cap_net_raw+eip $(eval readlink -f `which node`)
```

For security purposes, it is always a good idea to lock network and hardware access only to trusted users. However, for prototyping, the extra security can add hurdles, which make debugging hard. For a HomeKit bridge that is intended to be used for an extended period, I recommend rolling back this setting after you have become comfortable with HomeBridge.

Now, you can finally install `homebridge-bluetooth` using `npm`.

```
sudo npm install -g --unsafe-perm homebridge-bluetooth
```

As with `homebridge-dht`, in order to use the plug-in as part of your bridge, you will have to add its configuration information to your HomeBridge configuration file (`~/homebridge/config.json`), following the specification provided by the developers at www.npmjs.com/package/homebridge-bluetooth. For my implementation, I chose to follow their `switch` example, which registers the door sensor as a smart switch. Although there is a `lock` example, it is geared toward controlling popular IoT smart locks. Unfortunately, the `IOTDoor` project does not meet that specification yet. In Listing 7-2, I have provided the configuration I used for my door sensor.

Listing 7-2. Configuration JSON File for Homebridge (`config.json`), Including Bluetooth

```
{  "bridge": {
    "name": "IOTHome",
    "username": "CC:22:3D:E3:CD:31",
    "port": 51826,
    "pin": "031-45-154"
  },
  "description": "IOTHome HomeBridge",

  "platforms": "platforms": [
    {
      "platform": "Bluetooth",
      "accessories": [
        {
          "name": "IOTDoor",
          "address": "30:AE:A4:28:73:76",
          "services": [
            {
              "name": "Door Sensor",
              "type": "Switch",
              "UUID": " 83b46845-6e9c-4b25-89cf-
                871cc74cc68e",
```

```

    "characteristics": [
      {
        "type": "On",
        "UUID": " 4b61d6b9-2e29-4fdf-a74a-
          7b8bf70ecd9a"
      }
    ]
  }
]
}
]
}
],
"accessories": [
  { "accessory": "Dht",
    "name": "Indoor Comfort Sensor",
    "name_temperature": "Temperature",
    "name_humidity": "Humidity",
    "gpio": "21",
    "service": "dht22" }
]]

```

There are three aspects of the configuration that you must pay careful attention to. First, `homebridge-bluetooth` implements its interface via a platform configuration, rather than an accessory.

Next, in order to bridge the correct Bluetooth functionality, you must put the exact service and characteristic UUIDs for the lock status from the Arduino program into the `config.json` file. Finally, in order to find the device, you must provide its name and hardware address, as they are advertised. In Chapter 6's Arduino setup code, you did not have to set up a hardware address, as it is generated by the Bluetooth LE module in the ESP32 microcontroller. However, you can use the scanning utilities on the Raspberry Pi to find the hardware address. After verifying that your door sensor is not currently connected to any devices, run the following command:

```
sudo hcitool lescan
```

Your output should contain a list of nearby Bluetooth LE devices that are advertising in the physical vicinity of your Raspberry Pi, as shown in Figure 7-12.



```

[pi@raspberrypi:~ $ sudo hcitool lescan
LE Scan ...
DC:A9:04:6E:62:4E (unknown)
DC:A9:04:6E:62:4E (unknown)
30:AE:A4:28:73:76 IOTHome
70:4D:8A:24:33:67 (unknown)
70:4D:8A:24:33:67 (unknown)
55:82:A5:86:E6:A8 (unknown)
55:82:A5:86:E6:A8 (unknown)
5E:ED:2F:C6:83:12 (unknown)
62:E4:F1:F9:09:03 BLE Device
62:E4:F1:F9:09:03 (unknown)

```

Figure 7-12. Results of using the `hcitool` utility to scan for Bluetooth LE devices

After copying the hexadecimal hardware address for your IOTDoor sensor into the HomeBridge configuration file, save your result. Your HomeKit bridge is now fully configured!

Configuring HomeBridge to Run at Startup (Experimental)

To make your new HomeKit bridge meet user expectations, namely, that it will work when you plug it into power, you will have to configure the Raspberry Pi to start HomeBridge on startup. I have marked this section as experimental, since, during my research, I found it very difficult to find a configuration method that would be appropriate by Linux administration standards, easy to explain, and stable (i.e., boots up without errors every time). The solution I provide here is heavily based on the official HomeBridge GitHub documentation (<https://github.com/nfarina/homebridge/wiki/Running-HomeBridge-on-a-Raspberry-Pi#running-homebridge-on-bootup>) and Tim Leland’s supplementary guide (<https://timleland.com/setup-homebridge-to-start-on-bootup/>). In my experiments, these instructions worked best through one or two debugging sessions. After significant trial-and-error, I noticed much of the stability began to degrade. At that point, resetting the Raspberry Pi and starting over from scratch worked best for me.

To begin the setup process, start by creating a new system called `homebridge`, using the `useradd` command:

```
sudo useradd --system homebridge
```

For startup scripts, it is a common practice to run them as their own user. This allows you to separate users by their intended roles and adds security, by preventing jobs from running with full administrator permissions.

Next, you must specify the startup options for the job. Create a new file called `/etc/default/homebridge` using `nano` or your other favorite text editor.

```
sudo nano /etc/default/homebridge
```

For the contents of the configuration file, use the sample from Listing 7-3. The most important configuration option is `HOMEBRIDGE_OPTS`, which specifies where the HomeBridge configuration file is located. For this project, you will copy the configuration to `/var/lib/homebridge`, to make it more accessible by the `homebridge` user.

Listing 7-3. Startup Options File for HomeBridge

```
# Defaults / Configuration options for \homebridge
# The following settings tells homebridge where \to find the config.json
# file and where to \persist the data (i.e. pairing and others)
HOMEBRIDGE_OPTS=-U /var/lib/homebridge

# If you uncomment the following line, \homebridge will log more
# You can display this via systemd's journalctl: \journalctl -f -u
# homebridge
DEBUG=*
```

After saving the startup options file, you must define the service itself. Again, using your favorite text editor, create a file called `/etc/systemd/system/homebridge.service`.

```
sudo nano /etc/systemd/system/homebridge.service
```

Use the contents of Listing 7-4 as your guide to populating the service definition. In this file, be careful of the `User` key, which defines which user account will be employed to run the service, and the `ExecStart` key, which defines where HomeBridge is installed.

Listing 7-4. Startup Service Definition for HomeBridge

```
[Unit]
Description=Node.js HomeKit Server
After=syslog.target network-online.target

[Service]
Type=simple
User=homebridge
EnvironmentFile=/etc/default/homebridge
# Adapt this to your specific setup (could be \usr/bin/homebridge)
# See comments below for more information
ExecStart=/usr/local/bin/homebridge \${HOMEBRIDGE_OPTS}
Restart=on-failure
RestartSec=10
KillMode=process

[Install]
WantedBy=multi-user.target
```

Next, you must copy the local HomeBridge configuration files from your `.homebridge` directory to a new directory for the `homebridge` user. As shown in Listing 7-5, create a new directory, called, `/var/lib/homebridge`, copy all of your old files there, and then make the files executable.

Listing 7-5. Moving the HomeBridge Configuration Files

```
sudo mkdir /var/lib/homebridge
sudo cp ~/.homebridge/config.json /var/lib/homebridge/
sudo chmod -R 0777 /var/lib/homebridge
```

Finally, when the files have been moved over, register the service using the `systemctl` command.

```
sudo systemctl daemon-reload
```

Your HomeBridge installation is now ready to start up when your Raspberry Pi is powered on!

Connecting to Your New HomeKit Bridge

After the long and arduous process of setting up the Raspberry Pi, HomeBridge, and its plug-ins and creating the configuration file, you can now finally start HomeBridge and try to find your new HomeKit bridge in the iOS Home app.

To begin, start HomeBridge on your Raspberry Pi. A normal installation of HomeBridge is accomplished by running the command `homebridge -D &` (these options give you more debugging information and let the app run in the background). If you completed setting up the startup service, this is accomplished by running the command `sudo systemctl start homebridge`. Next, monitor the console output, to make sure HomeBridge does not report any errors while starting up. For a normal installation, these messages will appear in the terminal as HomeBridge starts up. For a startup service installation, you will have to monitor the status through the command `sudo journalctl -au homebridge`. When HomeBridge has started successfully, you should see log messages indicating that the Bluetooth and DHT sensor interfaces were established successfully, and, more important, a large QR code that you can use to set up the bridge from the Home app, as shown in Figure 7-13.



Figure 7-13. HomeBridge-generated QR code for HomeKit configuration

Next, open the Home app on your iPhone or iPad. It should start with the Home tab selected, showing an overview of your currently configured HomeKit Home and devices. As shown in Figure 7-14, press the Add button (+ sign) at the top right, then select Add Accessory from the action sheet that appears. This will take you to a screen with a camera view that allows you to scan in a HomeKit setup QR code. At this time, scan the QR code that appears in the HomeKit console output.

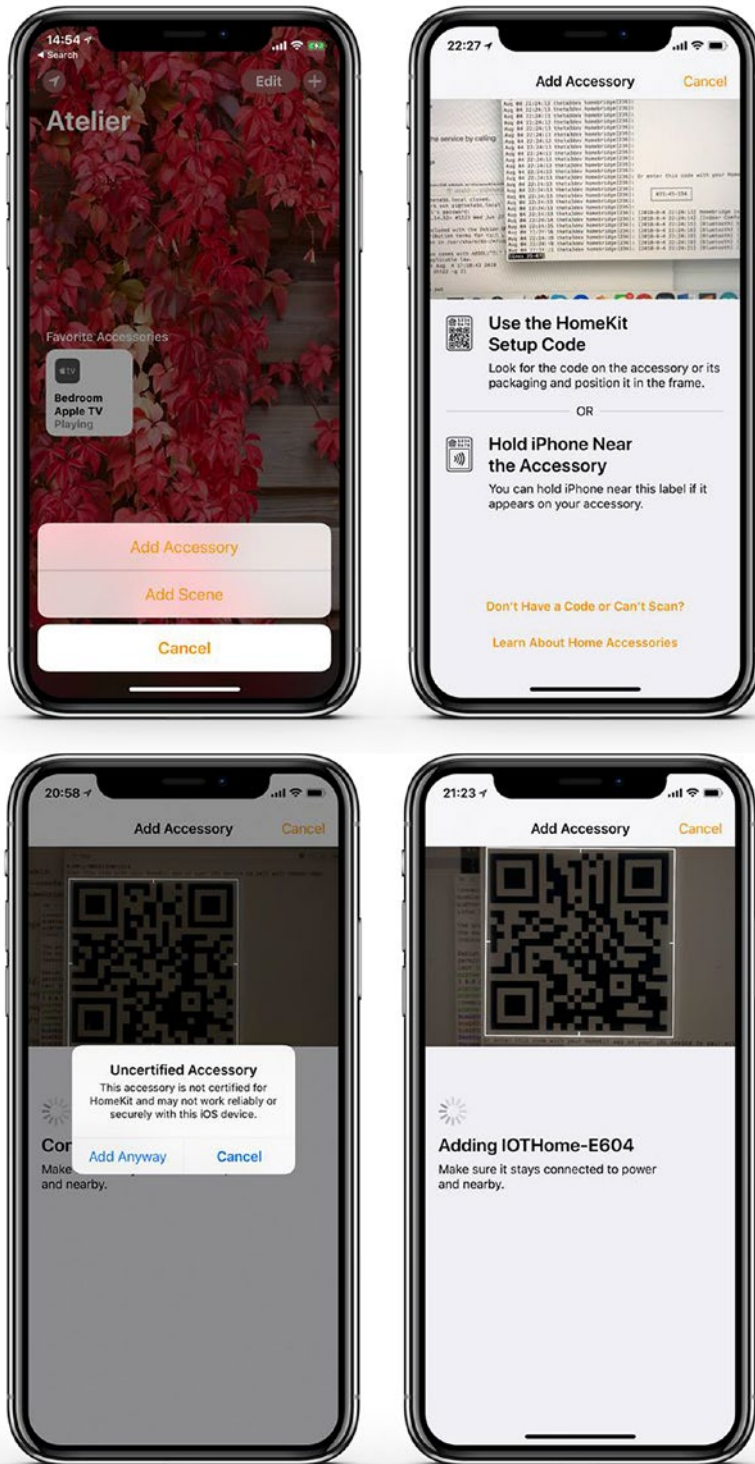


Figure 7-14. Adding a HomeKit accessory using the Home app

After the QR code has been recognized by the Home app, you will receive a pop-up asking you to confirm that you want to add an unofficial HomeKit device. This lets you know that HomeBridge is successfully advertising your bridge as a valid HomeKit device. If the Home app will not recognize your QR code, you can use the Don't Have a Code or Can't Scan button on the Add Accessory screen, to manually enter the PIN number from your HomeBridge configuration file. If your device is correctly set up, successful PIN entry should find your device and present the same pop-up.

After confirming your selection, an Adding message will appear on the screen for a few seconds while the bridge is being registered in the HomeKit database. As shown in Figure 7-15, once registration is complete, you will be asked to assign the room and display name for each service that is exposed through the bridge. For my configuration, I assigned the temperature and humidity services to my bedroom and the door sensor to the entrance. This will allow me to say, "Is the door switch on at the entrance?" or "What is the temperature in the bedroom?"

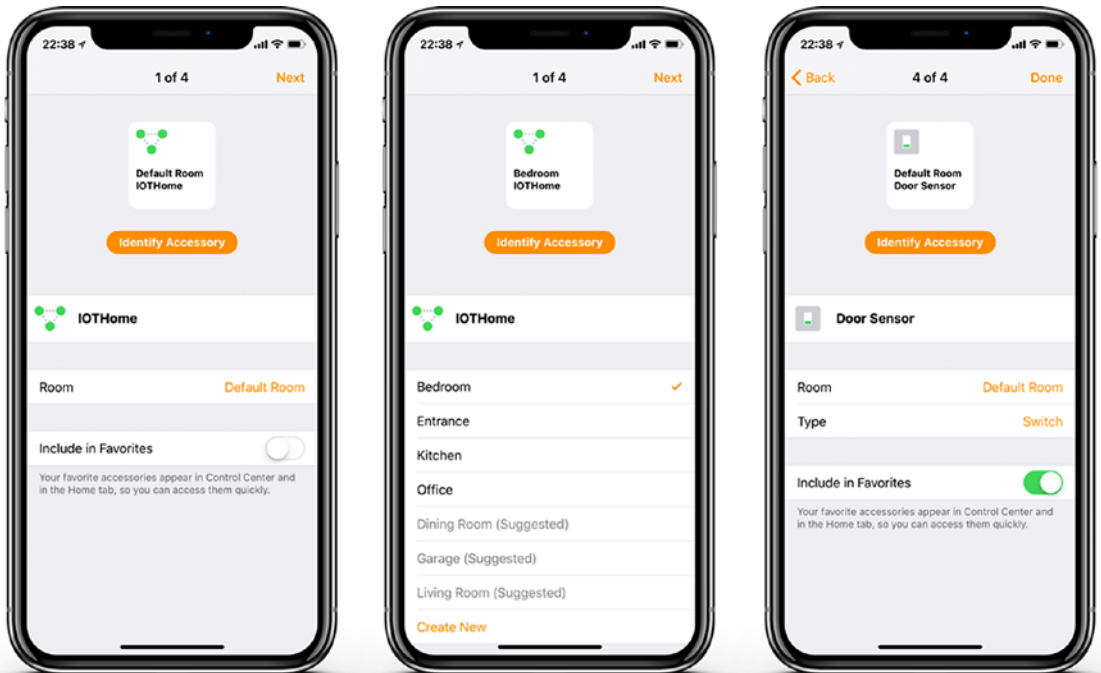


Figure 7-15. Assigning details to a new HomeKit device

After setup is complete, you will be returned to the Home tab. As shown in Figure 7-16, it will now include the services from your HomeKit bridge, as well as the latest values for each. As one final test, say “Hey Siri,” to activate Siri, then ask for the temperature in your bedroom. Siri should read back the latest value from your Raspberry Pi after a few seconds.

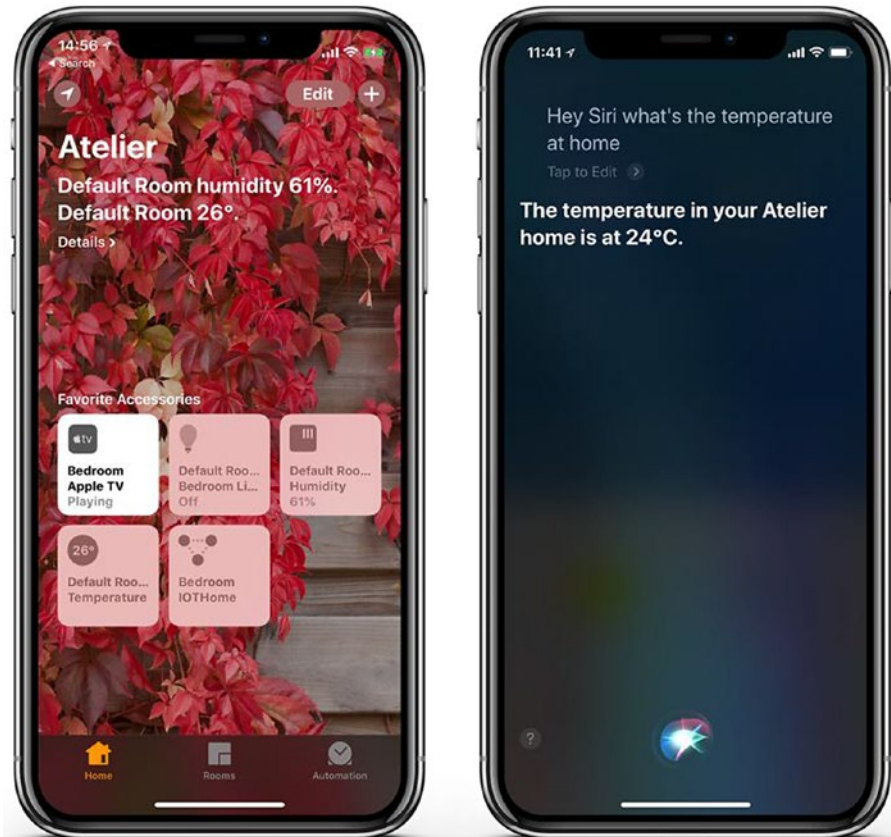


Figure 7-16. Using the Home app and Siri to confirm that the HomeKit bridge is operational

Troubleshooting Configuration Problems

While I was developing this chapter, the most common debugging problem I ran into was that after changing the HomeBridge configuration, HomeKit would not find the new services, even after restarting all of my devices. To alleviate this, I found the quickest solution in two simple tricks: changing the username of the HomeKit bridge and emptying the persist folder in HomeBridge’s configuration directory (`~/homebridge/persist` or `/var/lib/homebridge`, depending on your startup method).

As mentioned at the beginning of the chapter, HomeKit serves as a secure database of devices, managed by your Apple devices and HomeKit hub. Complementing this, HomeBridge keeps information on saved devices in its `persist` folder. Emptying this folder resets the saved information on your Raspberry Pi. Although you can delete and re-add devices through the Home app on your iOS devices, you cannot fully wipe out its database yourself. However, by changing some of the bytes of the username (for instance, changing the last two digits from 30 to 31), you can make it think you are connecting to a new device, allowing you to enter the provisioning process again.

Summary

In this chapter, you learned how to start harnessing the power of the Raspberry Pi single-board computer to run HomeBridge as a Node.js-powered HomeKit bridge for the door sensor you built in Chapters 5 and 6, as well as a thermometer sensor. Not only did this expand the functionalities of the smart functionalities of your IOTHome, it also allowed you to harness Siri commands for getting the statistics of your home sensors.

The most grueling part of this chapter was bootstrapping the Raspberry Pi, HomeBridge, and all of their dependencies, but I hope you will be able to use this chapter again in your projects as a quick setup guide. In later chapters, you will expand the capabilities of your Raspberry Pi even further, to make it as a web server. Not bad for a \$35 computer!

The instructions in this chapter were optimized for Raspberry Pi but can be modified slightly to work on a BeagleBone, Mac, or any other Linux-based system that can run Node.js.

CHAPTER 8

Building a Web Server on a Raspberry Pi

Back in the early days of the World Wide Web (the browser-based *Internet* you know today), the idea of a web server was transparent to most developers. You would install a program to host your HTML files (or open a free account on a service such as GeoCities) and spend most of your time thinking about what size to make your text or where you could find the best GIFs to put. To most people, a web server was just a place where you would upload your files.

As web development technologies continued to progress, more people found themselves needing the ability to control their web hosting themselves, leading to greater awareness of web server technologies. Instead of relying on your hosting provider's guarantees of security or uptime, you could begin to take matters into your own hands by configuring the server yourself.

Additionally, more advanced applications such as e-commerce and electronic medical records (EMR) systems began taking off, with their own requirements for user personalization, database integration, and security. To meet these needs, scripting languages, such as PHP and Ruby began to rise in popularity, allowing programmers to create pages that could perform all of the logic of processing data and generating a static page immediately after a user requested the page from his/her browser. Instead of static web pages, people were beginning to develop *web applications*.

In the world of the Internet of Things (IoT), you can take advantage of web application technology to help you expose data from your IoT devices over *HTTP*, the Hypertext Transfer Protocol. Whenever you use a web browser or an app that connects to a server, it is using HTTP to transfer that data. As you may remember from earlier chapters, protocols such as Bluetooth or HomeKit are optimized for specific use cases and sometimes require a massive amount of domain-specific knowledge to be deployed

successfully. Web technologies, on the other hand, are a well-known platform with a strong development community and low hardware requirements (you just need something that can power a web server).

In this chapter, you will revisit the Raspberry Pi and Node.js development environment from earlier chapters and expand both to advertise the same data over HTTP and HTTPS, using the Express module for Node.js. Due to its small binary size, potential for scaling, and huge developer base, Node has been taking over as a first-class web application scripting language.

Learning Objectives

In this chapter, you will learn how to set up a web server on the Raspberry Pi you have been using throughout this book, expose its sensor data over HTTPS end points, and connect to the server from the IOTHome app for iOS. You will use Node.js to create the web application and Swift for the iOS app. While some projects to get Swift running on Raspberry Pi are starting to gather steam, in today's environment, Python and Node.js are the most well-documented and -supported options for building web-based applications on a Raspberry Pi.

For this chapter, rather than depending on a full web server application, such as Apache, to listen for HTTPS connections and route them to Node.js, you will learn how to use the Express module for Node.js, to accomplish the same task. For projects for which you will be running multiple web applications on the same server, you may want to look at Apache or NGINX, to better manage your connections. However, the project in this chapter aims to replicate the use case for a single-purpose IoT device or a single-purpose web microservice, such as one you would host on Heroku or an Amazon Web Services Lambda.

In building the Raspberry Pi web server, you will learn the following key concepts for iOS IoT application development:

- Web application development core concepts
- Setting up Express to expose web services through Node.js
- Reading data from a temperature sensor in a Node.js application
- Reading data from Bluetooth sensors within a Node.js application
- Providing security with HTTPS

The project in this chapter is heavily based on the Express module for Node.js. If you would like to learn more about the module or find more support on it, please visit its official home page at www.expressjs.com.

Creating a Web Server to Share Data over HTTPS

In this section, I will focus on the process of setting up a web server using Node.js and Express, as well as how to use Node modules to expose the data generated by the IOTHome sensors (temperature, humidity, status of a switch) over HTTPS. The projects in this section assume that Node.js and the temperature sensor from Chapter 7 have been set up correctly. If you still feel uncomfortable with either topic, or are coming into this chapter directly, I highly recommend going back to the setup sections of Chapter 7 before progressing with this chapter.

In this section, you will start by implementing a web server that transfers data over HTTP, then you will learn how to apply an SSL certificate to the server, to make it an HTTPS server—the new, secure standard for web applications. After the web server is set up successfully, you will use it to provide the data for the Home screen on the IOTHome app.

Using Express to Expose Web Services

Unlike the HomeBridge project, where the primary task was installing and configuring a Node application that was developed by another party, in this chapter, you will develop your own Node application from scratch. To start developing the project, begin by deciding on a location for the project. After turning on your Raspberry Pi, open the terminal and create two new directories under your home directory: `sites` and `iothome`. These directories are the `sites` or `/var/www/` directory that most web hosting Linux distributions use as the *document root* (starting point) for web applications, and the location of the application itself. As shown in Listing 8-1, create the new directories, then use the `cd` command to change your working directory to the `iothome` directory.

Listing 8-1. Creating a Folder for the Express Project

```
mkdir ~/sites
mkdir ~/sites/iothome
cd ~/sites/iothome
```

Next, you must use `npm` to install Express in the directory for the `iothome` project. In Chapter 7, you installed the modules for HomeBridge globally, meaning all Node-based applications on the Raspberry Pi could use them. However, for application development, I recommend installing modules only in the project you are currently working on. In multi-application environments, this will prevent side effects on other projects. To install Express for the IOTHome project, run `npm install`, without the global flag.

```
npm install express
```

To verify the installation, you can write a simple Node application, which echoes Hello World across HTTP. To begin the development process, create a new JavaScript file, called `app.js`, using `nano` or your favorite text editor.

```
nano app.js
```

Unlike iOS or Arduino programming, in which application execution begins from a very specific method (for example, `viewDidLoad` for iOS, `setup` for Arduino), Node applications begin executing immediately. As Node implements object-oriented programming (OOP) concepts, you should use them to control the flow of your program in a predictable manner. To use Express, you must include the module, then instantiate an object of it. You can continue other operations while Express is running, but all HTTP requests will have to be handled through the Express object. To create your blank Express application, implement the code sample in Listing 8-2.

Listing 8-2. Creating a New Express-Based Node Application

```
var express = require('express');  
var app = express();
```

A QUICK INTRODUCTION TO HOW EXPRESS WORKS

Express works by listening for HTTP requests on a port, then responding to them, based on the *end point* that is specified. End points are defined as functions that are available to consumers of your API, made up of a *route*, the path component appended to the server's address and the HTTP method that is used to make the request (for example, GET, POST). Throughout web application development, you will see these two terms used interchangeably. To reduce confusion, I think the clearest explanation is that *end point* is primarily used as a term to describe external (client-facing) interactions with your server, and *route* is used primarily to describe the internal logic of your Express application.

For example, if you wanted to get a list of movie titles from a server with the IP address 10.0.1.5, the request would be `GET 10.0.1.5/movie/titles`. The `GET` method is frequently used for reading data. In this example, the route is `/movie/titles` and `GET` is the end point. In Express, the code for this end point would be look like this:

```
app.get('/movie/titles', function (req, res) { ... }
```

Creating a record for a new movie, on the other hand, may look something like this: `POST 10.0.1.5/movie/new`. The `POST` method is often used to add a new record. In database programming, the term *CRUD* is used to describe the four major operations for any type of data: create, read, update, and delete. Many back-end developers like to use this same model for naming their routes, and they will append `/new`, `/delete`, or `/update` to a route, to indicate it is a create, update, or delete operation. In Express, the code for this end point would be

```
app.post('/movie/new', function (req, res) { ... }
```

As you can tell, the method called on the `app` object changed, as well as the string for the route.

Finally, two of the other most frequently used HTTP methods are `PUT` and `DELETE`, which are used to update and delete records. You will not use them in this chapter, but they may be helpful to you in your own projects in the future.

By default, when you load a web page in a browser, it will try to make a `GET` request to the document root of the server on TCP port 80 (the port reserved for HTTP traffic). To represent this in Express, use the `get()` method on your object, to specify the code that should execute for the root directory. To make Express listen for requests, use the `listen()` method on the `app` object. The implementation for both the route and listener is shown in Listing 8-3.

Listing 8-3. Creating a New Express-Based Node Application

```
var express = require('express');
var app = express();

app.get('/', function (req, res) {
  res.send('Hello World');
});

app.listen(3000);
```

In this example, I instructed Node to send the text, `Hello World`, upon receiving a GET request for the document root. The `res` object is built into Node and specifies that you want to pipe output as an HTTP response. In this example, I asked Express to listen on port 3000 instead of the standard HTTP port 80, because Node considers port 80 a *privileged* port. Unless you are logged in to a root or system user account, you are not able to run applications from this port. Later in this section, you will switch to port 80, but for initial testing, it is best to use a non-privileged port.

To verify that the Express application is working, you must tell Node to start executing the new script, then try to make the request from a web browser.

Before starting the server, you must get the IP address for the Raspberry Pi. Inside the terminal, use the `ifconfig` command to view the information for your device. As shown in Figure 8-1, the IP address will appear next to the `inet` field, under the record for the `wlan0` interface (representing the built-in Wi-Fi interface).

```

pi@raspberrypi:~ $ ifconfig
eth0: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500
    ether b8:27:eb:24:31:c4 txqueuelen 1000 (Ethernet)
    RX packets 0 bytes 0 (0.0 B)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 0 bytes 0 (0.0 B)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    inet6 ::1 prefixlen 128 scopeid 0x10<host>
    loop txqueuelen 1 (Local Loopback)
    RX packets 133 bytes 7884 (7.6 KiB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 133 bytes 7884 (7.6 KiB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

wlan0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.43.237 netmask 255.255.255.0 broadcast 192.168.43.255
    ether b8:27:eb:71:64:91 txqueuelen 1000 (Ethernet)
    RX packets 32867 bytes 29936473 (28.5 MiB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 25905 bytes 5621109 (5.3 MiB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

pi@raspberrypi:~ $ █

```

Figure 8-1. Getting the IP address for a Raspberry Pi

Next, begin execution of the Node application, by calling the `node` command with your script's file name.

```
node app.js
```

Finally, open your favorite browser on another computer in your network and type in the URL for the Raspberry Pi, with the port number (3000) appended to it. Your "Hello World" string should appear as plain text in the browser, as shown in Figure 8-2.



Figure 8-2. Verifying the Hello World example in a browser

Congratulations on creating your first Node application! It will keep executing until you kill the Node process on your Raspberry Pi.

Reading Values from the DHT Temperature Sensor

Now that you have the hang of adding a Node module and Express route, you can begin to refine your simple example to mirror the status of the temperature sensor via HTTP. To begin, enter Ctrl+C into the terminal, to kill Node.js, then install the `node-dht-sensor` module using `npm`.

```
npm install node-dht-sensor
```

For my example, I thought it would make sense to use the temperature path extension to represent the temperature and humidity sensor. As shown in Listing 8-4, include the `node-dht-sensor` module in your application and add a route and end point for GET requests to the temperature path extension.

Listing 8-4. Defining a New Route for the Temperature Sensor

```
var express = require('express');
var dht = require('node-dht-sensor');
var app = express();
app.get('/temperature', function (req, res) {
```

```

//Your cool code will go here
});
app.listen(80);

```

At this point, I recommend making two changes: removing the end point for the earlier example and changing the port to 80. Many web servers are easily hacked when debugging end points are left active. Vigilant code maintenance is an easy way to reduce this risk in your applications. By using port 80, you will also be able to make your server behave closer to the HTTP specification, which states that HTTP traffic is transmitted on TCP/IP port 80.

Next, you must use the `node-dht-sensor` module to retrieve the data from the sensor. Referring to the documentation for the module available at its GitHub repository (<https://github.com/momenso/node-dht-sensor>), you will learn that you can perform this operation by calling the `read()` method on your `dht` object, specifying the sensor type (22 for DHT22 or 11 for DHT11), the general-purpose input/output (GPIO) pin its data line is connected to, and a callback method that will execute when the reading is complete. In my example in Chapter 7, I used GPIO 21 for the DHT22 sensor. In Listing 8-5, I have expanded the example to include reading the data from the temperature sensor.

Listing 8-5. Reading DHT22 Data from a Node Application

```

var express = require('express');
var dht = require('node-dht-sensor');
var app = express();

app.get('/temperature', function (req, res) {
  dht.read(22, 21, function(err, temperature, humidity) {
    res.type('json');
    if (!err) {
      res.json({
        'temperature': temperature.toFixed(1),
        'humidity': humidity.toFixed(1)
      });
    }
  });

```



```

    } else {
      res.status(500).json({error: 'Could not access
        sensor'});
    }
  });
});
app.listen(80);

```

In this example, you will notice that I used the `json()` method on the `res` object, to send back the temperature data. While plain text was sufficient for the Hello World example, using JSON (JavaScript Object Notation) is a widely adopted practice to represent dictionaries and hierarchical data in web application development. Additionally, most web and mobile frameworks these days provide built-in JSON validation and encoding/decoding, making it much easier to work with than custom data types. Following this line of reasoning, you will notice that I also used the `status()` method to return the error as a standard HTTP 500 server error. This allows you to use built-in HTTP error handling.

Next, you must restart the Node application. Kill the existing process, then run the script again with superuser permission.

```
sudo node app.js
```

Tip If you would like your Node application to automatically restart whenever you change its source code, I recommend looking into the nodemon tool, available via npm and GitHub at <https://github.com/remy/nodemon>. While this tool can be convenient during the development phase, I recommend disabling it in production.

To verify that the new route is working, attempt to load it in a web browser, by appending `/temperature` to the old URL. You should receive a plain text response containing the JSON-encoded temperature and humidity, as shown in Figure 8-3.

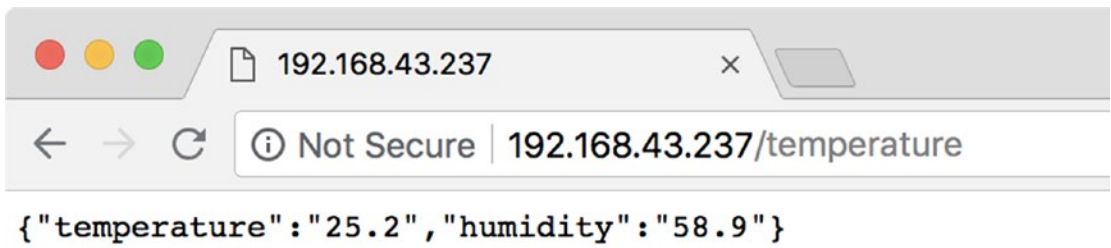


Figure 8-3. Verifying the temperature route in a browser

Caution It takes approximately two seconds for the DHT22 temperature and humidity sensor to get an accurate reading. Keep this in mind if you are getting stale data or time-outs when pinging the temperature route.

Reading Information from Bluetooth Devices

In the last section, you exposed the data from the temperature sensor over HTTP by reading its value directly from a Node application and then echoing it via Express. In this section, you will expose the Bluetooth door sensor’s data over HTTP, by making the Node application act like a Bluetooth central manager, using the Noble module for Node (<https://github.com/noble/noble>). You may remember from previous chapters that one of the riskiest and time-consuming parts of Bluetooth communication is discovering the device and establishing a connection to receive data. To help with this operation, in this section, you will add end points for managing the connection state and transmit data based on the last update (rather than making a new connection each time data is requested).

To begin, you must add Noble to the project, using the npm package manager.

```
npm install noble
```

Next, you must modify the Node application to include Noble and the UUIDs for Bluetooth services and characteristics for the door sensor, as shown in Listing 8-6. Just as with the apps in previous chapters, you will need these values to help identify the device and its data notifications.

Listing 8-6. Adding Noble and Bluetooth UUIDs to the Node Application

```
var express = require('express');
var dht = require('node-dht-sensor');
var noble = require('noble');
var app = express();

const LOCK_SERVICE_UUID = "4fafc2011fb5459e8fccc5c9c331914b";
const BATT_SERVICE_UUID = "0af2d54c4d334fc79e34823b02c294d5";
const LOCK_CHARACTERISTIC_UUID = "beb5483e36e14688b7f5ea07361b26a8";
const BATT_CHARACTERISTIC_UUID = "134c298f7d6b4f6484968965e0851d03";
...

```

Remember: These UUIDs were defined in Chapter 6 as unique, random hexadecimal values that identify the device. Just as with Chapter 6's Bluetooth app and Chapter 7's HomeBridge configuration, you need these values to find and identify the device. Because the values will not change while the application is executing, you can define them as constants, using the `const` keyword.

Just as with connecting to a hardware protocol like Bluetooth or I2C, it is common for flow-based web server operations, such as creating a new user account, to require the client developer (for example, mobile app developer, front-end web developer) to follow a specified flow of API calls to complete the operation. For the IOTHome Node application, the client developer must POST the `/door/connect/` end point before attempting to request data from the device. Similarly, after they have finished their session, they must POST to the `/door/disconnect/` end point, to close the connection and allow other applications to use the hardware.

In my implementation, I decided to start the connection process from the Express end point. In Listing 8-7, I have expanded the Node application to include a `/door/connect/` end point that uses Noble to scan for the door sensor. In this example, I also saved a reference to the response object from Express, so that I could complete the HTTP request at the same time the Bluetooth connection was established.

Listing 8-7. Discovering a Bluetooth Peripheral Using Noble

```

var response;
...
app.post('/door/connect', function (req, res) {
    console.log("start connect");
    response = res;
    noble.startScanning();
});
...
noble.on('discover', function(peripheral) {
    console.log("discovered");
    console.log("peripheral name "+peripheral.id+" "+peripheral.address +
" | " + peripheral.advertisement.localName);
var advertisement = peripheral.advertisement;
if (PERIPHERAL_NAME == advertisement.localName) {
    noble.stopScanning();
console.log('peripheral with name ' +
advertisement.localName + ' found');
console.log('ready to connect');
}
});

```

The arrangement of the connection may seem a bit odd at first. In the Arduino code for the door sensor, you had to implement completion handlers to progress through the connection flow for the Bluetooth server. In the iOS app, you had to implement delegate methods. To receive messages with Noble, you must respond to `discover` events, which are triggered by initiating a scan for devices. To implement an efficient Bluetooth LE connection process, you should scan only for the devices advertising the services you need. However, at the time of writing, I noticed that the results of the scan API that specifies service UUIDs were hard to predict, so, instead, I decided to filter discovered devices by the name specified in their advertisement data.

After you have confirmed that the device is within range, you must try to connect to it. In Listing 8-8, I have expanded the application to include the connection process. Just as when you implemented a Bluetooth central manager on iOS, after finding a device, you must connect to it and save a reference to it, so you can disconnect from it later.

Listing 8-8. Connecting to a Bluetooth Peripheral Using Noble

```

...
var savedPeripheral;
...
noble.on('discover', function(peripheral) {
  console.log("discovered");
  var advertisement = peripheral.advertisement;
  if (PERIPHERAL_NAME == advertisement.localName) {
    noble.stopScanning();
    console.log('attempting to connect');
    connect(peripheral);
  }
});

function connect(peripheral) {
  peripheral.connect(function(error) {
    if (error) {
      console.log('error = ' + error);
      response.status(500).json({error: 'Could not
      find sensor'});
    } else {
      console.log('connected');
      response.json({'status': 'connected'});
      savedPeripheral = peripheral;
    }
  });
}

```

For the final step in the connection process, you must find the characteristics for the data you want to observe and set up their completion handlers. In Listing 8-9, I call the `discoverAllServicesAndCharacteristics()` method, then subscribe to events only for the characteristics matching the desired UUIDs.

Listing 8-9. Observing and Responding to Characteristic Updates with Noble

```

function connect(peripheral) {
    peripheral.connect(function(error) {
        if (error) {
            ...
        } else {
            ...
            discoverServices();
        }
    });
}

function discoverServices() {
    if (savedPeripheral) {
        savedPeripheral.discoverAllServicesAndCharacteristics(
            function(error, services, characteristics) {
                if (error) {
                    console.log('error = ' + error);
                }
                console.log('services = ' + services);
                console.log('characteristics = ' + characteristics);
                for (characteristic in characteristics) {
                    if (characteristic.uuid ==
                        LOCK_CHARACTERISTIC_UUID ||
                        characteristic.uuid == BATT_CHARACTERISTIC_UUID) {
                        observeCharacteristic(characteristic);
                    }
                }
            }
        });
    }
}

function observeCharacteristic(characteristic) {

```

```

//Fires when data comes in
characteristic.on('data', (data, isNotification) => {
  console.log('data: ' + data + '');
  lastUpdateTime = date.getTime();

  if (characteristic.uuid == BATT_CHARACTERISTIC_UUID) {
    batteryStatus = data;
  }

  if (characteristic.uuid == LOCK_CHARACTERISTIC_UUID) {
    lockStatus = data;
  }
});

//Used to setup subscription
characteristic.subscribe(error => {
  if (error) {
    console.log('error setting up subscription = ' + error +
      'for uuid:' + characteristic.uuid);
  } else {
    console.log('subscription successful for uuid:' +
      characteristic.uuid);
  }
});

```

The subscription process is initiated through the `subscribe` method, but the data must be observed through the `on` method. Because it is impractical to make the user wait until the first update has been delivered, I save the values to global variables that can be queried later.

Caution While researching this chapter, I noticed that the Bluetooth utility on the Raspberry Pi became unable to maintain a connection after several connection debugging sessions. If you are having issues with the door sensor not reporting a successful connection via its blue status LED, try restarting the Pi and then trying again.

To expose the data over HTTP, create a `/door/status` end point. When the end point is called, echo the saved values from the global variables and wrap them in a JSON dictionary, as shown in Listing 8-10. To help enforce the connection flow for the API, return an error if the device connection has not been established yet.

Listing 8-10. Reading Data from a Bluetooth Peripheral, Using Noble

```

app.get('/door/status', function (req, res) {
  console.log("start connect");
  if (savedPeripheral) {
    res.json({
      'lockStatus': lockStatus,
      'batteryStatus': batteryStatus,
      'lastUpdateTime': lastUpdateTime
    });
  else {
    res.status(500).json({error: 'Not connected to a
      sensor. Please re-connect and try again.'});
  }
});

```

To wrap up the Node application, you must create the `/door/disconnect/` end point. As shown in Listing 8-11, in my implementation, I disconnect from the device when the end point is called. To be safe, I also stop scanning for the BLE device, just in case this method is called before the connection has been fully established.

Listing 8-11. Disconnecting from a Bluetooth Peripheral, Using Noble

```

app.post('/door/disconnect', function (req, res) {
  noble.stopScanning();
  console.log("stop scan");
  if (savedPeripheral) {
    console.log('disconnected');
    savedPeripheral.disconnect();
  }
  res.json({
    'status': 'disconnected'
  });
});

```

Before moving on, you may be wondering how to test the POST-based end points. For this task, I recommend downloading the Postman OS X app from www.getpostman.com/. As shown in Figure 8-4, to begin your debugging session, simply click the Send button

after configuring your request. The results of your request will appear in the large text field at the bottom of the window. You can access a list of your past requests (and their configurations) from the left sidebar of the window.

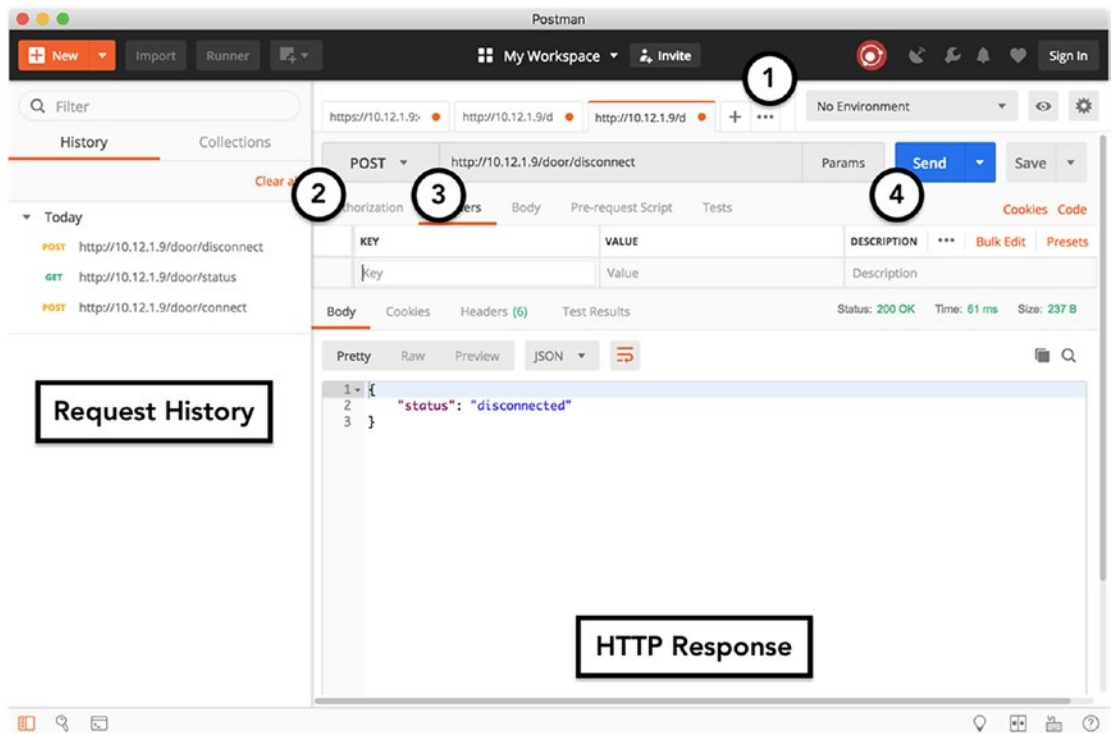


Figure 8-4. Using Postman to verify POST requests

Using HTTPS to Provide Secure HTTP Connections

In a move to increase the privacy of users' data and reduce phishing (false identity) attacks on the Internet, starting in 2016, Apple, Google, and other major technology companies announced that their platforms would be moving toward primarily supporting servers that implement HTTPS, an extension of HTTP that requires all data to be encrypted with Transport Layer Security (TLS). TLS is implemented by adding a Secure Sockets Layer (SSL) certificate to your server, which has been issued by a provider that is trusted by the major browsers and your platform (for example, iOS).

In Google Chrome, some of the most obvious implications of not using HTTPS are that your site will show up lower in Google's search ranking. Additionally, sites with untrusted TLS certificates will be marked as Not Secure and present users a warning

page when they are loaded in the browser. In iOS, Apple enforces HTTPS by making all HTTP requests fail inside of an app, unless the developer manually re-enables them. Additionally, all untrusted HTTPS requests will fail.

To work around these limitations and improve the security of the IOTHome device, you should extend the Node application to support HTTPS. As with the other functionality in this project, you can take advantage of Node modules and tools that have been developed for web apps, to easily add HTTPS to the IOTHome project.

There are three major options for implementing HTTPS in your project.

1. If you are developing for a production environment, you must request an SSL certification from a service that is trusted by the Internet Engineering Task Force (IETF), the organization that maintains the HTTPS standard. I recommend using Comodo, Verisign, or a certificate from Amazon Web Services (AWS). Certificates from these providers have the greatest compatibility, clear instructions, and support.
2. If you would like to develop a production-level prototype and already have a domain, you can use the Let's Encrypt trust authority (www.letsencrypt.org) and its accompanying tool, certbot-auto (<https://certbot.eff.org/docs/install.html>), to generate a free, trusted SSL certificate for testing.
3. For pure prototype purposes, you can generate your own SSL certificate, using OpenSSL on your Raspberry Pi.

For the purposes of this book, I have chosen option #3. If you would like to use options #1 or #2, I suggest creating those certificates on the server attached to your domain name, then copying it over to your Raspberry Pi (granted your SSL provider allows this capability).

GENERATING AN OPENSSL SELF-SIGNED SSL CERTIFICATE

With OpenSSL, you act as your own trust provider and generate an SSL certificate that meets the basic encryption requirements of HTTPS. This is referred to as a *self-signed certificate*. Because it is not generated by one of the trusted vendors I mentioned above, most browsers and iOS will initially reject it, until you perform some steps to trust it on the device, which I will explain after you are done generating the certificate.

If you have ever created an Apple Developer Program iOS development certificate or Push Notification certificate, you are already familiar with the process of generating SSL certificates (although the delivery method is different). In Apple's model, you create a private key (a unique hex value that is used as the base for the encryption/decryption of communications), create a Certificate Signing Request (CSR) file using the Keychain Access tool to serve as a receipt for your application for a new certificate, and then submit the CSR file to Apple's web site, which will refresh with a new SSL certificate you can download once your request has been processed. With OpenSSL, you can have complete control of this flow, to the point where you can even import existing private keys or hand CSRs to another service. For this project, though, you will act as your own Certificate Authority (CA), so you do not require a CA; you simply have to create a private key and a certificate. To do this in one command in OpenSSL, input the following command:

```
openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout express.  
key -out express.crt
```

The preceding command specifies that you want to create a private key based on RSA 2048-bit encryption and a certificate based on that key, which is good for 365 days. As with your iPhone developer certificate, make sure that you save the private key and do not share it with others. Losing the key will result in being unable to use the certificate. Sharing the key will allow others to break your encryption.

Now that you have a valid SSL certificate, you can begin using it in your Node application. To begin, add the `https` and `fs` (filesystem) modules to your project, as shown in Listing 8-12. These modules are provided with the standard Node.js distribution and do not require any additional installation steps.

Listing 8-12. Adding the `https` and `fs` Modules to the Node Project

```
var express = require('express');  
var dht = require('node-dht-sensor');  
var fs = require('fs');  
var https = require('https');  
var app = express();  
  
app.get('/temperature', function (req, res) {  
  ...  
});
```

Earlier in the chapter, `app.listen(80)` was used to instruct Express to listen for HTTP traffic on port 80. To use HTTPS in place of HTTP, you must disable this line and, instead, instruct an `https` object to listen for traffic. To initialize the `https` object, you must provide it with the paths to the SSL certificate and its private key on the Raspberry Pi. If you used Let's Encrypt to generate these, they will be under the folder that was output by the `certbot-auto` tool. If you generated the SSL certificates with your own provider, you will have to save the files to your Raspberry Pi, either by downloading them via the Chromium browser on the Pi itself or configuring another tool, such as `avahi-daemon`, to help make the Raspberry Pi discoverable by your Mac over Bonjour.

Once you have verified the location of the SSL certificate and private key, create a dictionary to store the file paths, and initialize a new `https` object, as shown in Listing 8-13.

Listing 8-13. Configuring the Node Project to Use HTTPS Instead of HTTP

```
var express = require('express');
var dht = require('node-dht-sensor');
var fs = require('fs');
var https = require('https');
var app = express();

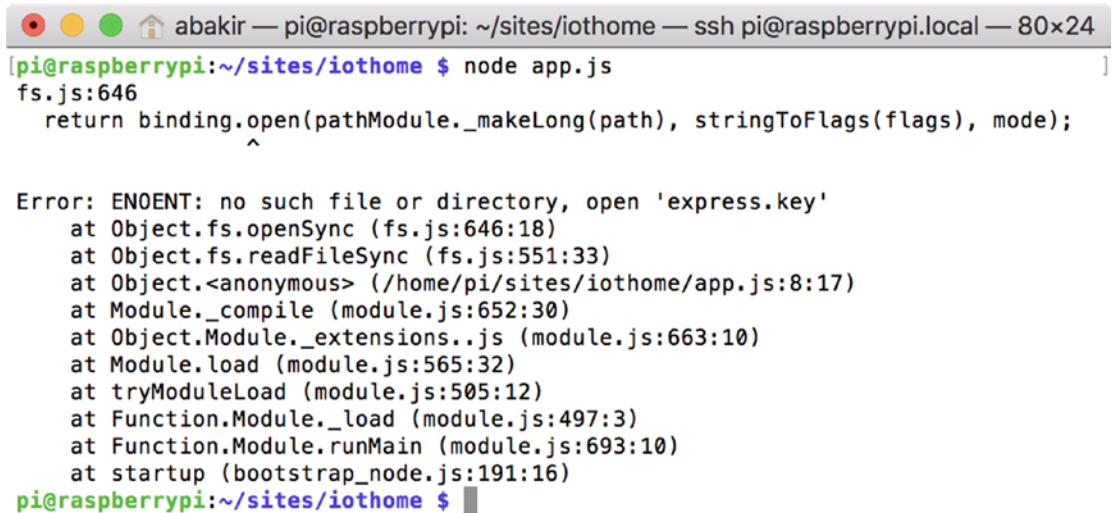
var sslOptions = {
  key: fs.readFileSync('express.key'),
  cert: fs.readFileSync('express.crt')
}

https.createServer(sslOptions, app);
https.listen(4443);
//app.listen(80);

app.get('/temperature', function (req, res) {
  ...
});
```

As with the earlier HTTP example, for the first HTTPS test, I suggest listening for traffic on port 4443 instead of the protected port for HTTPS, 443. To test that your SSL configuration was successful, kill your old Node process and reload the file for the application. If there is a problem loading the SSL certificate, you will see an error

message in the terminal at this point, similar to the example in Figure 8-5. As they are well-adopted technologies, you can find a great deal of data to help you resolve OpenSSL and Node HTTPS issues, based on these error messages.



```

[pi@raspberrypi:~/sites/iothome $ node app.js
fs.js:646
  return binding.open(pathModule._makeLong(path), stringToFlags(flags), mode);
                    ^
Error: ENOENT: no such file or directory, open 'express.key'
    at Object.fs.openSync (fs.js:646:18)
    at Object.fs.readFileSync (fs.js:551:33)
    at Object.<anonymous> (/home/pi/sites/iothome/app.js:8:17)
    at Module._compile (module.js:652:30)
    at Object.Module._extensions..js (module.js:663:10)
    at Module.load (module.js:565:32)
    at tryModuleLoad (module.js:505:12)
    at Function.Module._load (module.js:497:3)
    at Function.Module.runMain (module.js:693:10)
    at startup (bootstrap_node.js:191:16)
pi@raspberrypi:~/sites/iothome $

```

Figure 8-5. Sample error message for failed Node HTTPS configuration

Next, change the URL for the /temperature end point to include port number 4443 and https as the protocol. Attempt to load the URL in your browser. If you are using Google Chrome, you will receive a security warning about the page being insecure, similar to the one I received in Figure 8-6.

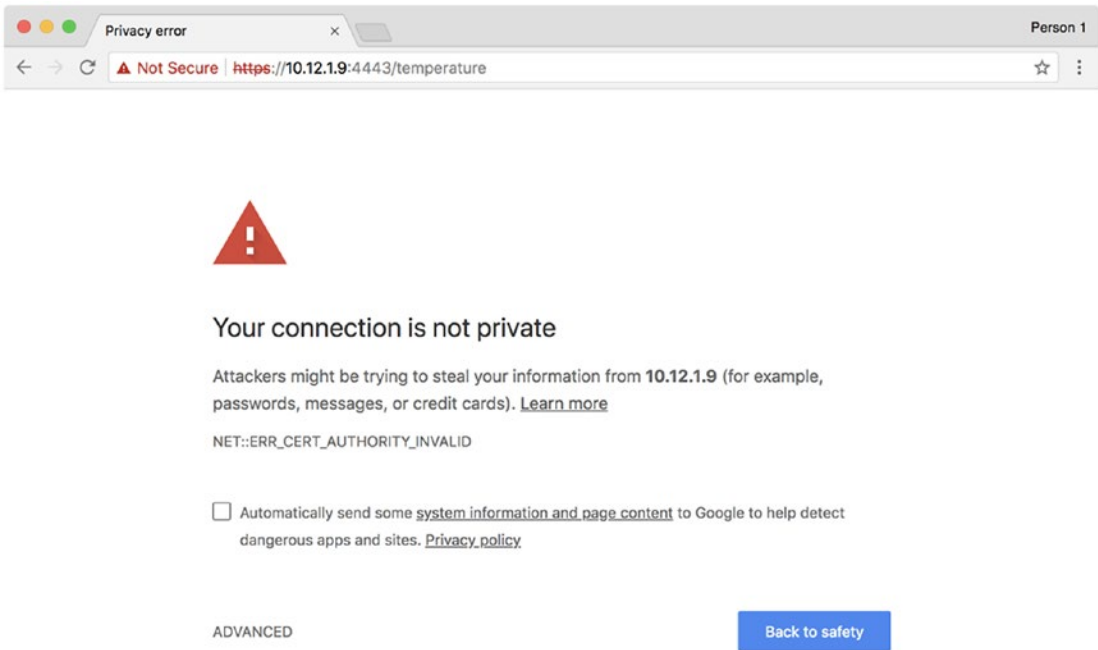


Figure 8-6. Google Chrome warning for pages with untrusted SSL certificates

To resolve this error, click the ADVANCED link at the bottom of the page, and then click the Proceed to... link, as shown in Figure 8-7.

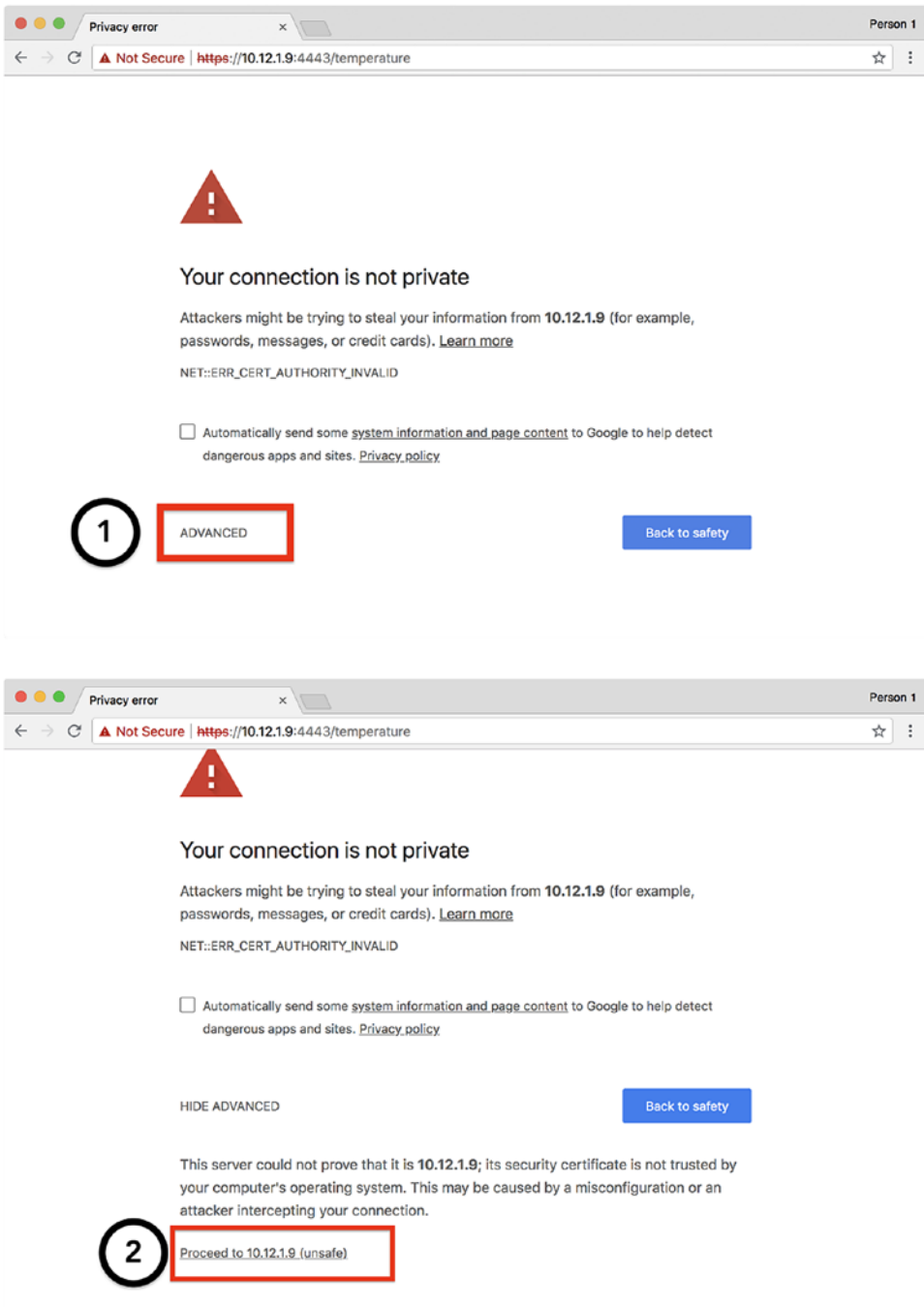


Figure 8-7. Enabling trust for a page in Google Chrome

After verifying that you want to load the page, you should now be able to see the temperature JSON data in the browser, just like you did when it was exposed through normal HTTP.

At this point, it is safe to change your application to listen on port 443. Just remember that you will have to run Node as a superuser, and that you will have to trust the :443 endpoint in Chrome. For HTTPS requests to port 443, you do not have to append the port number in iOS or your web browser.

To enable Postman to connect to self-signed certificates, click the Wrench icon at the top-right of the screen, as shown in Figure 8-8, then set SSL certificate verification to OFF.

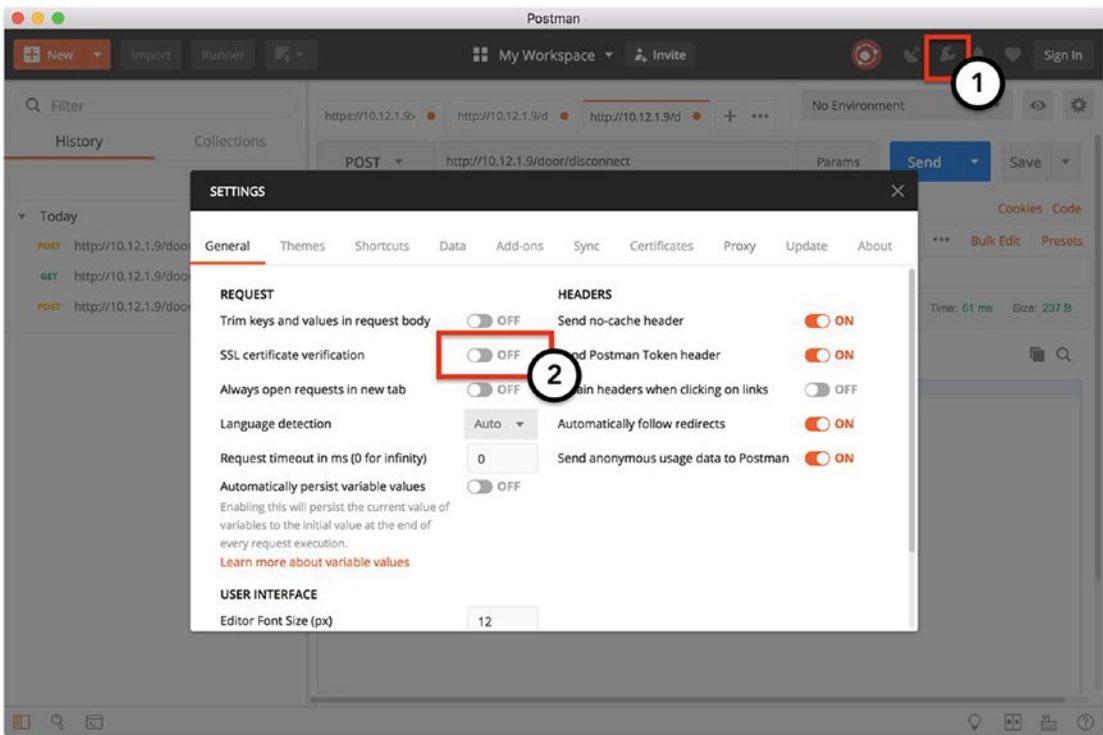


Figure 8-8. Trusting self-signed certificates in Postman

Configuring the Server to Start Up with the Raspberry Pi

As the final step in setting up the web server, you should make the Node application start with the Raspberry Pi on boot. This will prevent you from having to manually start and run the Node application every time you want to access its data through HTTPS. If you implemented this step for the HomeBridge project from the last chapter, this setup

process should be extremely familiar to you, as you will create a service using the `systemd` tool, to manage this operation. Unlike HomeBridge, however, the IOTHome web server is much easier to set up as a service.

To begin, you must create a service definition. First, create a file named `iothome.service` in your home directory using your favorite text editor. Within this file, you will have to specify

- The name of the service
- The working directory for the script that will be run as a service
- The location of the script
- The user permissions for the script
- The failure behavior for the script

In Listing 8-14, I have provided the service definition file for my implementation of the project. To mimic the development environment, note that the user is set to `root`, and the working direction is set as the `sites/iothome` folder for the `pi` user.

Listing 8-14. Service Definition for the IOTHome Node Application

```
[Service]
WorkingDirectory=/home/pi/sites/iothome
ExecStart=node app.js
Restart=always
StandardOutput=syslog
StandardError=syslog
SyslogIdentifier=iothome
User=root
Group=root

[Install]
WantedBy=multi-user.target
```

Next, you will have to copy the definition file to the default directory for `systemd` and enable read and execute permissions on the file.

```
sudo cp ~/iothome.service \ /etc/systemd/system/iothome.service
sudo chmod u+rwx /etc/systemd/system/iothome.service
```

As with the HomeBridge service, you must register the service with `systemd`.

```
sudo systemctl enable iothome
```

To start the service, call the `systemctl` tool again, this time with the `start` command.

```
sudo systemctl enable iothome
```

Your script is now set up to restart with the Raspberry Pi! To confirm that the operation was successful, restart your Raspberry Pi and try to call the `/temperature` end point from your browser. To view error messages, call the `systemctl` utility with the `status` command.

```
sudo systemctl status iothome
```

From here on out, if you have to modify your service definition or would like to modify the script itself, stop the service before performing your changes, then restart it once you are done.

Connecting to Your Server from an iOS App

At this point, you are able to access all of the data from the IOTHome system, using the web server on the Raspberry Pi. You also learned many different tools to debug the connection, including Google Chrome, the command line, and Postman. However, this is an iOS book, so it is only natural to learn how to apply these skills to iOS apps.

In this section, you will expand the IOTHome app from previous chapters to add a screen that allows users to access the sensors in the system via HTTP instead of Bluetooth. While the UI code for Apple platforms are mostly single-use, the networking code can be reused among all platforms. In Chapter 9, you will reuse the networking code from this chapter to power an Apple TV-based dashboard for the IOTHome system.

Setting Up the User Interface

For this project, the user interface plays a supporting role to the networking code. As such, I do not want to focus too much on creating a new user interface for the Home Manager screen (the one intended to show data for the entire system). As opposed to the Door Manager, it should show information from the temperature system and connect to the web server instead of Bluetooth to retrieve data. To accomplish this, you will subclass

the `DoorViewController` class (the backbone for the Door Manager screen), add the new properties for displaying temperature, and override the `connect()` method to initiate a call to the HTTPS web server, instead of a Bluetooth device.

For the user interface, I have provided the updated wireframes in Figure 8-9. I used the same basic layout as the Door Manager screen, except I added the new labels for the temperature and humidity data above the door sensor information. I also changed the description text for the Update button.

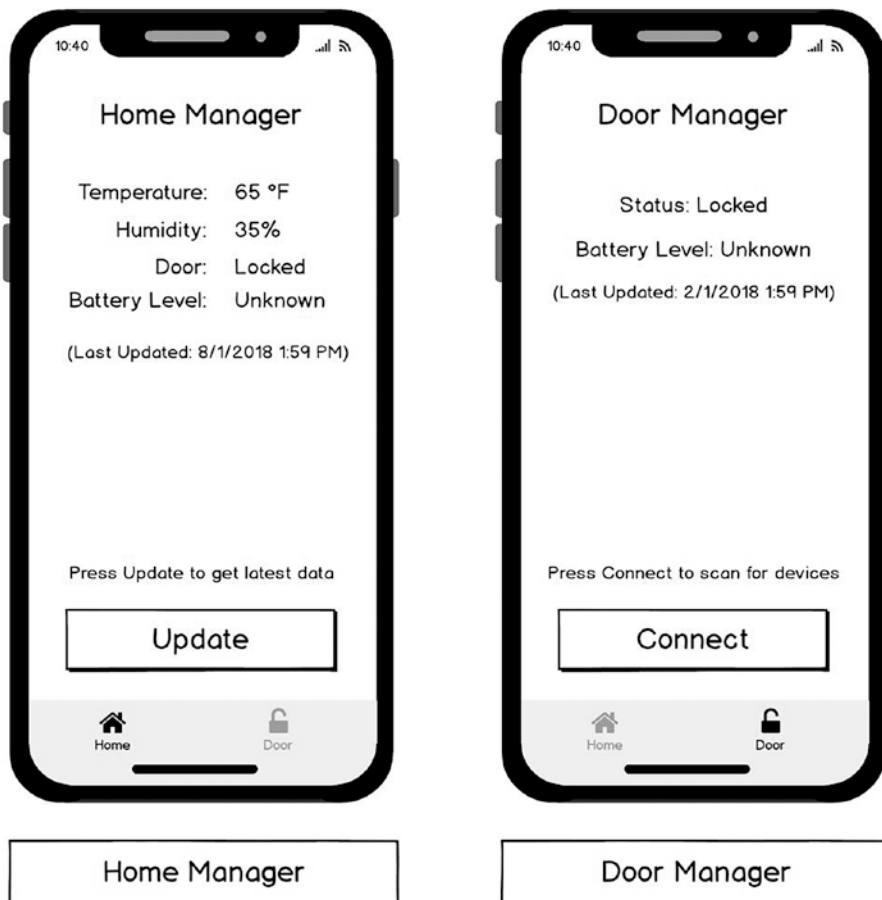


Figure 8-9. Updated wireframes for the IOTHome app

To start implementing the project, make a clone of the IOTHome app from Chapter 6. You can copy your project files or a fresh copy from the GitHub project for this book (<https://github.com/Apress/program-internet-of-things-w-swift-for-ios>).

In Table 8-1, I have provided the property names and constraints for the user interface elements. If you need a refresher on applying constraints, I recommend reviewing Chapters 1 and 6.

Table 8-1. *Styling for Home View Controller User Interface Elements*

Element Name	Text Style	Height	Top Margin	Bottom Margin	Left Margin	Right Margin
Navigation bar	Prefers large text	—	—	—	—	—
“Temperature” title label	Title 2	24	40	—	30	20
“Temperature” value label	Title 2	24	40	—	20	≥30
“Humidity” title label	Title 2	24	8	—	30	20
“Humidity” value label	Title 2	24	8	—	20	≥30
“Door” title label	Title 2	24	8	—	30	20
“Door” value label	Title 2	24	8	—	20	≥30
“Battery Level” title label	Title 2	24	8	—	30	20
“Battery Level” value label	Title 2	24	8	—	20	≥30
“Last Updated” label	Body	25	8	—	20	20
“Press to Connect” label	Body	25	—	20	20	20
“Connect” button	Title 1	60	20	30	20	20

Your final Interface Builder storyboard file (Main.storyboard) should be similar to my example in Figure 8-10.

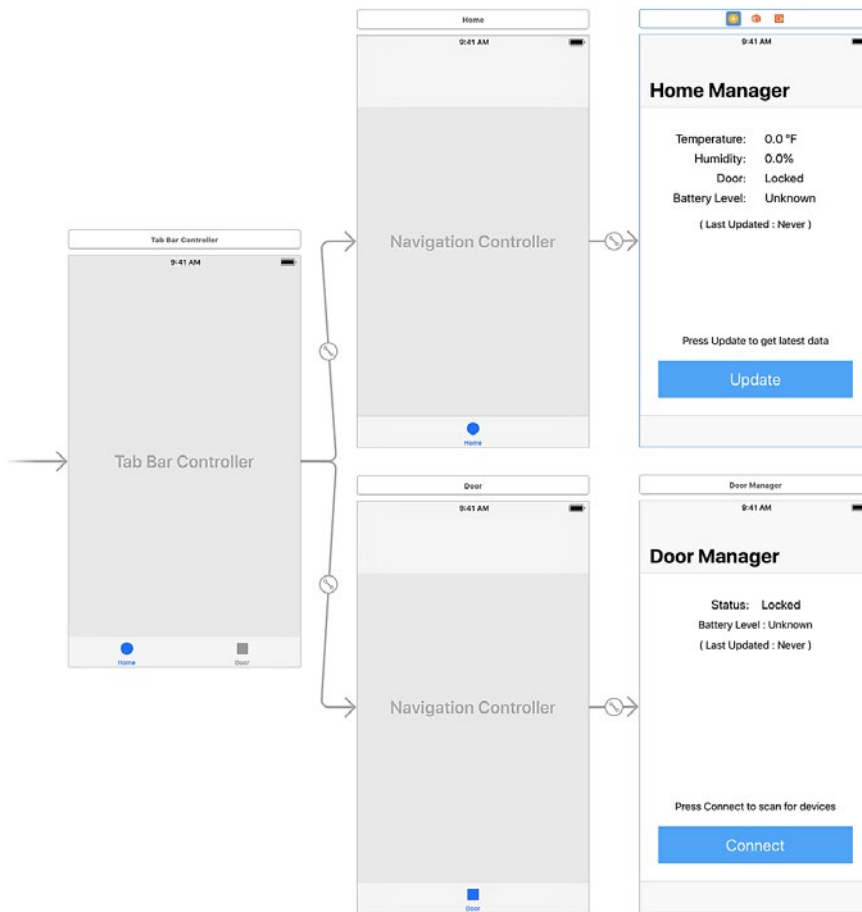


Figure 8-10. Updated storyboard for the IOTHome app

From the previous implementation in Chapter 6, the owner for the Home Manager screen was set to the HomeViewController class. As shown in Listing 8-15, update the HomeViewController.swift file to use the DoorViewController class as its parent. Additionally, add the properties for the new labels and create an empty connect() method, with the override keyword, to indicate you will be overriding a method from the parent class.

Listing 8-15. Updated HomeViewController Class, Including User Interface Scaffolding

```
import UIKit

class HomeViewController: DoorViewController {

    @IBOutlet var temperatureLabel: UILabel?
    @IBOutlet var humidityLabel: UILabel?

    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the
        // view, typically from a nib.
    }

    @IBAction override func connect() {
        //Put network init code here
    }
}
```

For the final step, in the user interface setup process, connect all of the outlets for the labels and Update button handler to the view controller via Interface Builder. Your Connection Inspection output for the class should resemble my implementation in Figure 8-11. If you need a refresher on making the connections, please review Chapters 1 and 6.

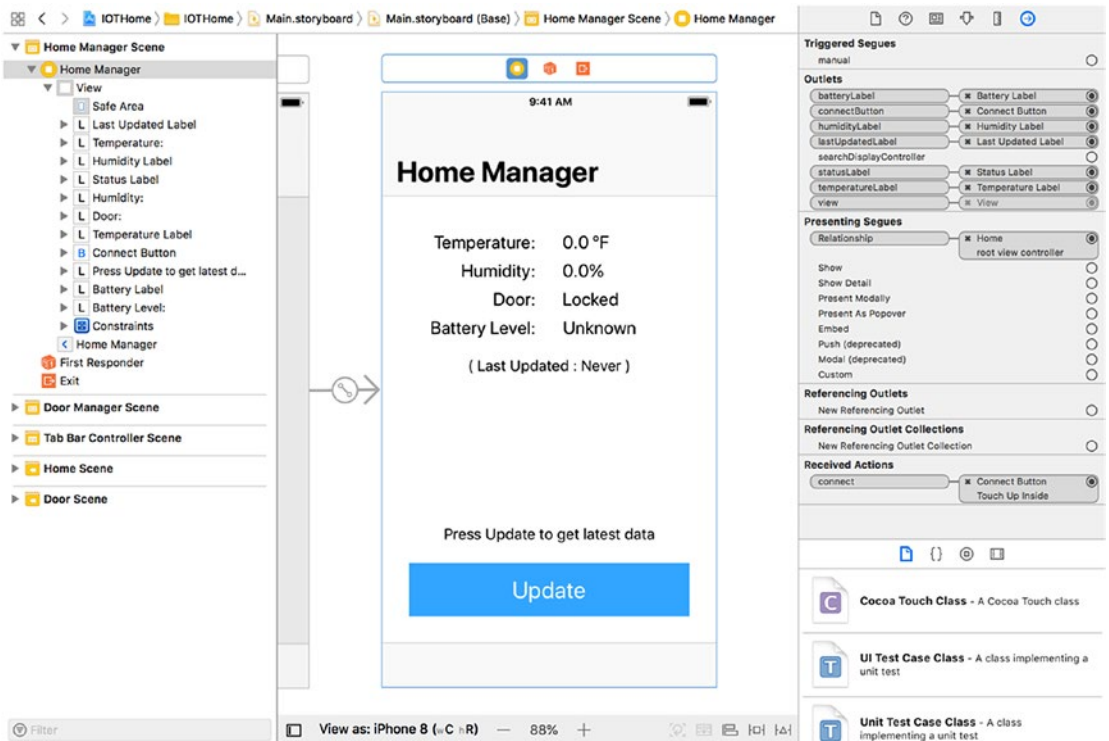


Figure 8-11. Updated storyboard connections for the Home View Controller

Making and Responding to HTTPS Requests

Now that the user interface is ready for the Home View Controller, you can begin working on the networking code for the project. In Chapter 6, you created a `BluetoothService` class to streamline Bluetooth connections in the Door View Controller. For this project, you will use a similar pattern, by creating a `NetworkManager` class to manage network connections in the app. Unlike the Bluetooth Service, the Network Manager will have to be accessed by the Home Manager and `AppDelegate` class for the app, and the state will have to be the same, regardless of who is calling it. For the sake of simplicity, in this project, I will implement this behavior via a singleton (a lazy-loaded, global object). I have provided my initial implementation of the class in Listing 8-16.

Listing 8-16. Initial implementation of Network Manager as a Singleton

```
import Foundation

class NetworkManager: NSObject {

    static let shared = NetworkManager(urlString:
        "https://raspberrypi.local")
    let baseUrl: URL

    init(urlString: String) {
        guard let baseUrl = URL(string: urlString) else {
            fatalError("Invalid URL string")
        }
        self.baseUrl = baseUrl
    }
}
```

Singletons are a contentious subject in the Apple developer community, owing to their nature of being shared globally, but for this chapter, a singleton is a convenient choice, because there are no side effects, and I want to re-create Apple’s approach to accessing hardware APIs (using a single object throughout iOS to manage one resource, for example, GPS, camera). If you are interested in alternatives to singletons, I recommend researching dependency injection. Dependency injection is not a first-class design pattern from Apple, so you should exercise some caution in picking a library or implementation that suits your application.

Singletons are implemented in Swift by adding a static property to an object, which returns an initialized instance of that class. If the object was initialized before, the existing object will be returned; otherwise, a new object will be initialized. This is referred to as *lazy-loading*. For my initializer, all I needed to do was initialize the class with the base URL for the network requests. For the base URL, use the Bonjour name of the device, as I did in my example. As of this writing, Raspbian ships with Bonjour enabled by default. Bonjour allows Apple devices to find devices on a network by their domain name, instead of an IP address.

For the network implementation, start with the simplest end point first (temperature). To query the temperature, all you have to do is make a GET request to the /temperature end point. In iOS, this operation is accomplished using the URLSession class. Just like the Network Manager, this object is a singleton. The URLSession class

exposes network operations in three main categories: data tasks, upload tasks, and download tasks. As the names suggest, upload and download tasks are intended for long-running file uploads or downloads. For short-running operations, such as web server API calls, data task is the most appropriate operation to use. Because all of the API responses from the Raspberry Pi return JSON data, you can wrap the network calls in a single method. In Listing 8-17, I have created the base method for these calls: `request(endpoint:httpMethod:completion:)`. As does its parameters, it takes the end point extension and string representing the method types, and it returns a JSON dictionary containing the response from the server (or an error).

Listing 8-17. Network Manager Method for Making HTTP Requests

```
class NetworkManager: NSObject {
    func request(endpoint: String, httpMethod: String,
        completion: @escaping (_ jsonDict: [String: Any]) ->
        Void) {

        guard let url = URL(string: endpoint, relativeTo:
            baseUrl) else {
            return completion(["error": "Invalid URL"])
        }

        var urlRequest = URLRequest(url: url)
        urlRequest.httpMethod = httpMethod

        let session = URLSession.default

        let task = session.dataTask(with: urlRequest) { (data:
            Data?, url: URLResponse?, error: Error?) in
            if error == nil {
                do {
                    guard let jsonData = data else {
                        return completion(["error": "Invalid
                            input data"])
                    }
                    guard let result = try
                        JSONSerialization.jsonObject(with: jsonData,

```

```

        options: []) as? [String : Any] else {
            return completion(["error": "Invalid
                JSON data"]) }
        completion(result)
    } catch let error {
        return completion(["error":
            error.localizedDescription])
    }
} else {
    guard let errorObject = error else { return
        completion(["error": "Invalid error
            object"]) }
    return completion(["error":
        errorObject.localizedDescription])
}
}
}
task.resume()
}
}
}

```

The basic flow of the method is to create a URL request object using the end point and HTTP method string, then create a completion handler for the data task, and execute the task using the `resume()` method. You may notice a significant amount of error handling in this method. Although the `JSONSerializer` and `URLSession` classes abstract a lot of logic for you, they are prone to failure from incorrect configuration. Adding detailed error handling will make it easier for you to find which step failed later. Because the method returns its result through a completion handler, you can pass along the error object, instead of the result from the server.

In Listing 8-18, I use this method to get the temperature, by calling from the Home View Controller's `connect()` method, via a new Network Manager method called `getTemperature(completion:)`. By using completion handler-based logic, you can quickly pass the result object through the entire flow, without having to reprocess it at every step.

Listing 8-18. Using the Network Manager to Get the Temperature from the Server

```

class NetworkManager: NSObject {
    ...
    func getTemperature(completion: @escaping (_ jsonDict:
        [String: Any]) -> Void) {
        request(endpoint: "temperature", httpMethod: "GET") {
            (resultDict: [String: Any]) in
                completion(resultDict)
        }
    }
}

class HomeViewController: DoorViewController {
    ...
    @IBAction override func connect() {
        NetworkManager.shared.getTemperature { [weak self]
            (resultDict: [String: Any]) in

                if let error = resultDict["error"] as? String {
                    self?.displayError(errorString: error)
                } else {
                    DispatchQueue.main.async {
                        if let temperature =
                            resultDict["temperature"] as? String {
                            self?.temperatureLabel?.text = "\(temperature) C"
                        }

                        if let humidity = resultDict["humidity"]
                            as? String {
                            self?.humidityLabel?.text = "\(humidity)%"
                        }
                    }
                }
            }
        }
    }
}

```

```

func displayError(errorString: String) {
    let alertView = UIAlertController(title: "Error",
        message: errorString, preferredStyle: .alert)
    let alertAction = UIAlertAction(title: "OK", style:
        .default, handler: nil)
    alertView.addAction(alertAction)

    DispatchQueue.main.async { [weak self] in
        self?.present(alertView, animated: true,
            completion: nil)
    }
}
}

```

Caution When you must access a class’s properties from within a completion handler, always perform the operation through a weak reference. Accessing `self` directly creates what is referred to as a *retain cycle*: a memory leak resulting from strong references to a class never being completely released.

Next, run the app and press the Update button, to attempt the network request. The request should fail with an SSL similar to my result in Figure 8-12. This is owing to the Raspberry Pi using a self-signed certificate, just like the issues you faced with Postman and Google Chrome.

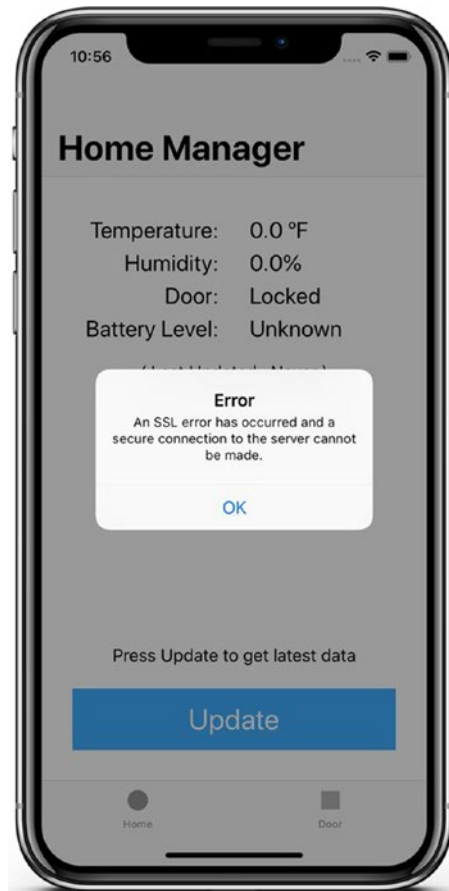


Figure 8-12. Error message for self-signed certificates

As mentioned earlier in the chapter, Apple wants to enforce verified SSL certificates as the default setting for network operations in iOS apps, to help ensure the safety of users' data. To enable self-signed certificates for the Raspberry Pi, you can add a *whitelist* entry, or exception, for the server in the IOTHome app's `Info.plist` file. Find the file in your project explorer, then secondary-click (long-press or right-click) it, to select `Open As` ► `Source File`. When the text editing window for the file appears, append the snippet in Listing 8-19, to enable self-signed certificates for the Raspberry Pi's domain only.

Listing 8-19. Info.plist Entry for Enabling Self-Signed SSL

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/
DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>CFBundleDevelopmentRegion</key>
    <string>$(DEVELOPMENT_LANGUAGE)</string>
    ...
<key>NSAppTransportSecurity</key>
<dict>
    <key>NSExceptionDomains</key>
    <dict>
        <key>raspberrypi.local</key>
        <dict>
            <key>NSExceptionAllowsInsecureHTTPLoads</key>
            <true/>
            <key>NSIncludesSubdomains</key>
            <true/>
        </dict>
    </dict>
</dict>
</dict>
</plist>

```

For the final change to enable self-signed certificates, you must override the `NSURLSessionDelegate` protocol method for HTTPS authentication challenges. In Listing 8-20, I have implemented this by creating a new `NSURLSession` object, which takes a delegate, in the `request()` method. Within the authentication method, I whitelist the Raspberry Pi's domain only. After this change, when you try to run the app again, your network requests should now complete successfully.

Listing 8-20. Enabling Self-Signed SSL Certificates Through URLSessionDelegate

```

class NetworkManager: NSObject, URLSessionDelegate {

    func request(endpoint: String, httpMethod:
        String, completion: @escaping (_ jsonDict:
            [String: Any]) -> Void) {
        ...
        var urlRequest = URLRequest(url: url)
        urlRequest.httpMethod = httpMethod

        let session: URLSession = URLSession(configuration:
            URLSessionConfiguration.default, delegate: self,
            delegateQueue: OperationQueue.main)

        let task = session.dataTask(with: urlRequest)
            { (data: Data?, url: URLResponse?, error:
                Error?) in
            ...
        }
        task.resume()
    }
    ...
func urlSession(_ session: URLSession, didReceive
    challenge: URLAuthenticationChallenge, completionHandler: @escaping
    (URLSession.AuthChallengeDisposition, URLCredential?) -> Void) {

    let method = challenge.protectionSpace.
        authenticationMethod
    let host = challenge.protectionSpace.host
    NSLog("Received challenge for \(host)")
    switch (method, host) {
    case (NSURLAuthenticationMethodServerTrust,
        "raspberrypi.local"):
        let trust = challenge.protectionSpace.serverTrust!
        let credential = URLCredential(trust: trust)
        completionHandler(.useCredential, credential)
    }
}

```

```

        default:
            completionHandler(.performDefaultHandling, nil)
        }
    }
}

```

To read the door status for the IOTHome system, you must call the `/door/connect` end point and then the `/door/status` end point. In Listing 8-21, I implemented this behavior by nesting the call for the `/door/status` end point within the completion handler for the `/door/connect` end point. Just as with the temperature reading, this network call should be initiated when you press the Update button in the app.

Listing 8-21. Getting the Status of the Door Sensor Through the Network Manager

```

class NetworkManager: NSObject, URLSessionDelegate {
    ...
    func getDoorStatus(completion: @escaping (_ jsonDict:
        [String: Any]) -> Void) {
        connectDoor { [weak self] (result: [String: Any]) in
            if (result["error"] as? String) != nil {
                return completion(result)
            } else {
                self?.request(endpoint: "door/status",
                    httpMethod: "GET") { (resultDict: [String:
                    Any]) in
                        completion(resultDict)
                    }
            }
        }
    }

    func connectDoor(completion: @escaping (_ jsonDict:
        [String: Any]) -> Void) {
        request(endpoint: "door/connect", httpMethod: "POST") {
            (resultDict: [String: Any]) in

```



```

        completion(resultDict)
    }
}

class HomeViewController: DoorViewController {
    ...
    @IBAction override func connect() {
        ...
        NetworkManager.shared.getDoorStatus{ [weak self]
            (resultDict: [String: Any]) in
            if let error = resultDict["error"] as? String {
                self?.displayError(errorString: error)
            } else {
                DispatchQueue.main.async {
                    if let doorStatus =
                        resultDict["doorStatus"] as? String {
                        self?.statusLabel?.text =
                            "\(\doorStatus)"
                    }

                    if let batteryStatus =
                        resultDict["batteryStatus"] as? String {
                        self?.batteryLabel?.text =
                            "\(\batteryStatus)"
                    }

                    if let lastUpdate =
                        resultDict["lastUpdate"] as? String {
                        self?.lastUpdatedLabel?.text =
                            "\(\lastUpdate)"
                    }
                }
            }
        }
    }
}

```

Finally, to implement the final API calls for the app, you should call the `/door/disconnect` end point when the app is backgrounded or when the Home screen is navigated away from. This will allow other devices to connect to the door sensor when the app is inactive. As shown in Listing 8-22, you can implement this by creating a `disconnectDoor()` method in the `NetworkManager` and calling it from the Home View Controller's `viewWillDisappear()` method, as well as the App Delegate's `applicationWillResignActive()` method.

Listing 8-22. Automatically Disconnecting from the Door Sensor

```
class NetworkManager: NSObject, URLSessionDelegate {
    ...
    func disconnectDoor(completion: @escaping (_ jsonDict:
    [String: Any]) -> Void) {
        request(endpoint: "door/disconnect", httpMethod:
        "POST") { (resultDict: [String: Any]) in
            completion(resultDict)
        }
    }
}

class AppDelegate: UIResponder, UIApplicationDelegate {
    ...
    func applicationWillResignActive(_ application:
    UIApplication) {
        NetworkManager.shared.disconnectDoor { (resultDict:
        [String: Any]) in
            NSLog("Disconnect result:
            \ \(resultDict.description)")
        }
    }
}

class HomeViewController: DoorViewController {
    ...
    override func viewWillDisappear(_ animated: Bool) {
        super.viewWillAppear(animated)
        NetworkManager.shared.disconnectDoor { (resultDict:

```

```

    [String: Any]) in
    NSLog("Disconnect result:
        \(resultDict.description)")
    }
}
}

```

Summary

In this chapter, you were able to build a classic IoT device by expanding the Raspberry Pi from earlier chapters to act as a web server and expose its data via HTTPS end points. During this process, you also learned how HTTP requests work, how Express and Noble can offload the hard work of implementing the HTTP and Bluetooth stacks for you, and how to connect to the end points using an iOS app. In a similar manner to setting up HomeKit, many of these tasks were not as much iOS- or Raspberry Pi-specific as they were implementations of established Linux and web application development practices.

Before single-board computers such as the Raspberry Pi achieved commercial success, many proprietary system-on-a-chip solutions were providing this same core functionality, GPIO and a web server, with a much higher price tag and learning curve. Thanks to the streamlining of technologies such as these, the IoT continues to expand, but as you will learn later in the book, you should also remember to add HTTPS or other security measures, to help make it a safe IoT.

PART 3

Building Apps Using Apple's Advanced Internet of Things Technologies

CHAPTER 9

Using tvOS to Build an Apple TV Dashboard App

Although the Apple TV was introduced in 2007, its adoption rate was slow for several years, and it was famously described as “a hobby” by Steve Jobs. At that time, its main purpose was to serve as a set-top box for media purchased on iTunes. In 2012, a lower-priced, iOS-inspired Apple TV 2 came out and took off immediately. Instead of being limited to iTunes content, users could suddenly watch content from Netflix and popular TV channels via apps that were packaged with system updates. This led many people to suspect that the device was, in fact, based on a fork of iOS, and the hope was that Apple would one day offer its software development kit to all developers.

In 2015, these predictions came true when Apple announced the Apple TV 4, the public release of the software development kit for its operating system (now called tvOS) and an App Store for the device. As many people speculated, tvOS is based on iOS, and developers can write apps for it using Swift and close forks of frameworks from iOS, including Cocoa Touch and Core Location. As of this writing, the notable exceptions are MapKit and WebKit—meaning you cannot embed web pages or maps directly into your native tvOS apps.

With this in mind, for this chapter, you will bridge the two worlds of tvOS and the Internet of Things (IoT) by taking what you know about iOS and using it to build an Apple TV dashboard for the IOTHome hardware. To make the user experience compelling, you will include data from both the IOTHome web server and the OpenWeatherMap.org weather API, allowing you to show climate inside and outside a user’s home.

Note Although you can use the Apple TV simulator and any Linux-based device to develop the project in this chapter, the code samples and explanations are optimized for an Apple TV 4 and Raspberry Pi 3 or newer versions.

Learning Objectives

In this chapter, you will learn how to build an Apple TV dashboard app by adding an Apple TV target to the IOTHome project, creating tvOS-resources for the target, and writing code to make the tvOS app render its user interface and make network requests on its own. In creating the tvOS dashboard app, you will learn the following key concepts for iOS IoT application development:

- Adding a tvOS target to an existing iOS app
- Creating a user interface for a tvOS project
- Making HTTP requests within a tvOS app
- Requesting user location from a tvOS app
- Connecting to the OpenWeatherMap.org public weather API
- Running a tvOS app on an Apple TV

One of Apple's great accomplishments with tvOS was the large extent to which it preserved iOS development tools and practices for the platform. You will find yourself forgetting what platform you are writing code for, compared to watchOS, in which many APIs are absent, or macOS, in which there is little overlap in workflows or shared frameworks. As you work through this chapter, I will highlight these similarities and how you can apply them to tvOS. As an added bonus, although you will be writing new code for the user interface, you will be able to reuse the network code from Chapter 8 with almost no changes.

In this chapter, I assume you are familiar with the IOTHome sensors (Chapters 5-7) and the web server (Chapter 8). If you are unfamiliar with either of these, I highly recommend reviewing them before continuing with this chapter. As with previous chapters, you can find the completed project for this chapter on the GitHub repository for this book (<https://github.com/Apress/program-internet-of-things-w-swift-for-ios>), under the Chapter 9 folder.

Setting Up the tvOS Target

If you have ever downloaded any games or weather apps for iOS, and happen to also own an Apple TV, you may have noticed some of these apps magically appearing on your Apple TV home screen. You can re-create this experience for your users and increase adoption of your tvOS experience by adding your tvOS app as a new target to your IOTHome iOS app. To get started, make a copy of the IOTHome app from Chapter 8, or download a fresh copy from the GitHub repository for the book. Next, open the project in Xcode and then select **New** ► **Target** from the File menu, as shown in Figure 9-1.

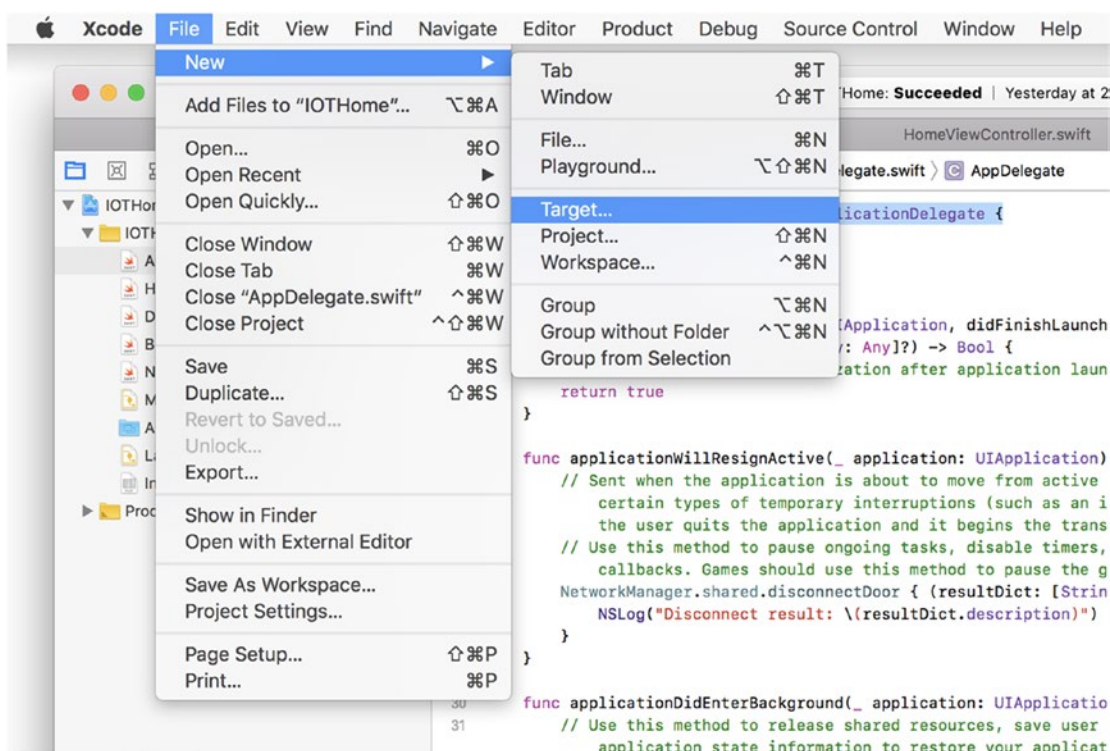


Figure 9-1. Adding a tvOS target to an existing iOS project

In the Template Picker window that appears after clicking the menu item, select tvOS and then Single View App, as shown in Figure 9-2.

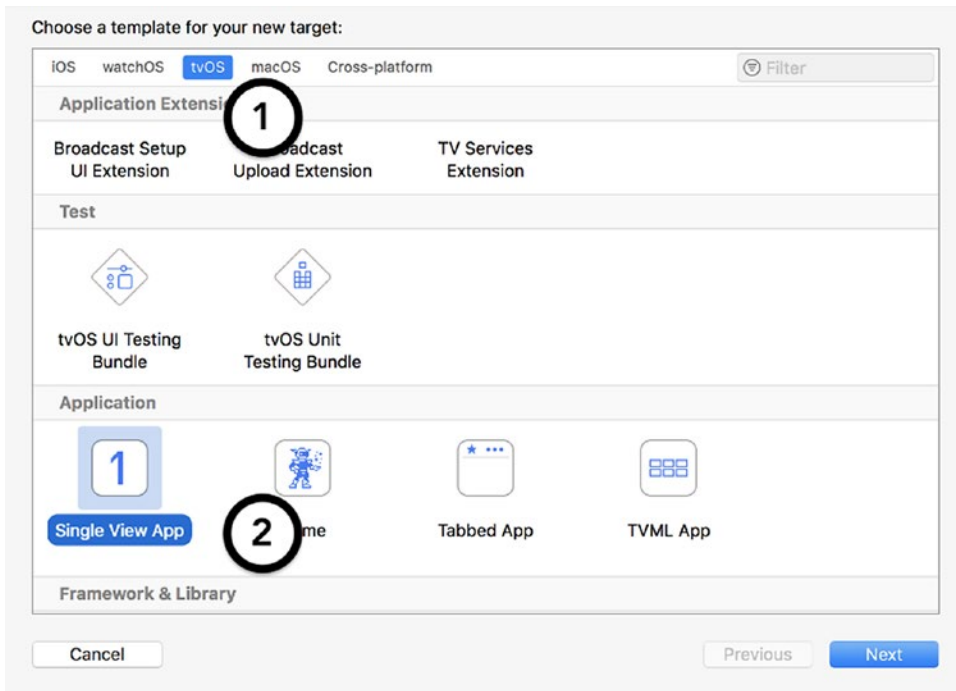


Figure 9-2. *Selecting the Single View App tvOS template*

When asked to name the product, call it “IOTHomeTV.” Your organization identifier should be the same as that you used for the previous iterations of the applications (reverse domain notation for your name or web site domain). Your project should now contain a new scheme and new folders for the IOTHomeTV target, as shown in Figure 9-3.

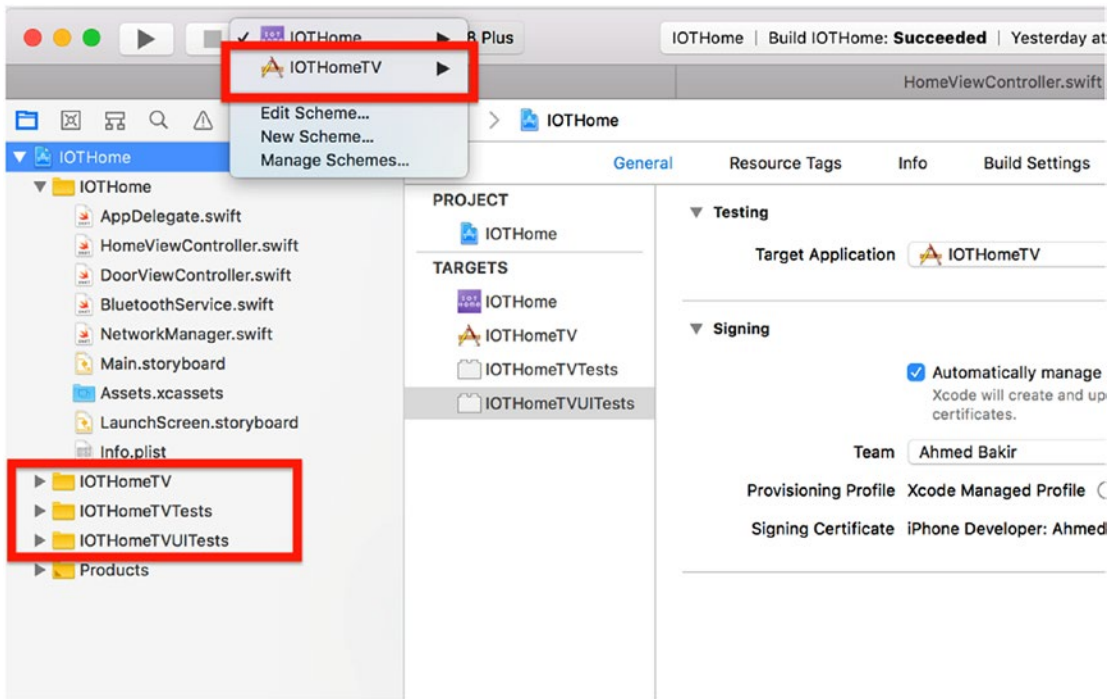


Figure 9-3. *IOTHome project hierarchy, including the new tvOS scheme and files*

Next, you must copy over the network whitelist from the iOS project, in order to connect to the self-signed HTTPS end points. Open the `Info.plist` file for the IOTHome iOS app (in the IOTHome folder) and secondary-click (right-click) the key named App Transport Security Settings. Select Copy to copy the contents of the key-value pair. Next, open the `Info.plist` file for the IOTHomeTV tvOS app (in the IOTHomeTV folder), and from the secondary-click (right-click) menu, select Paste, to paste the key-value pair. Your resulting `Info.plist` file should look similar to my result in Figure 9-4.

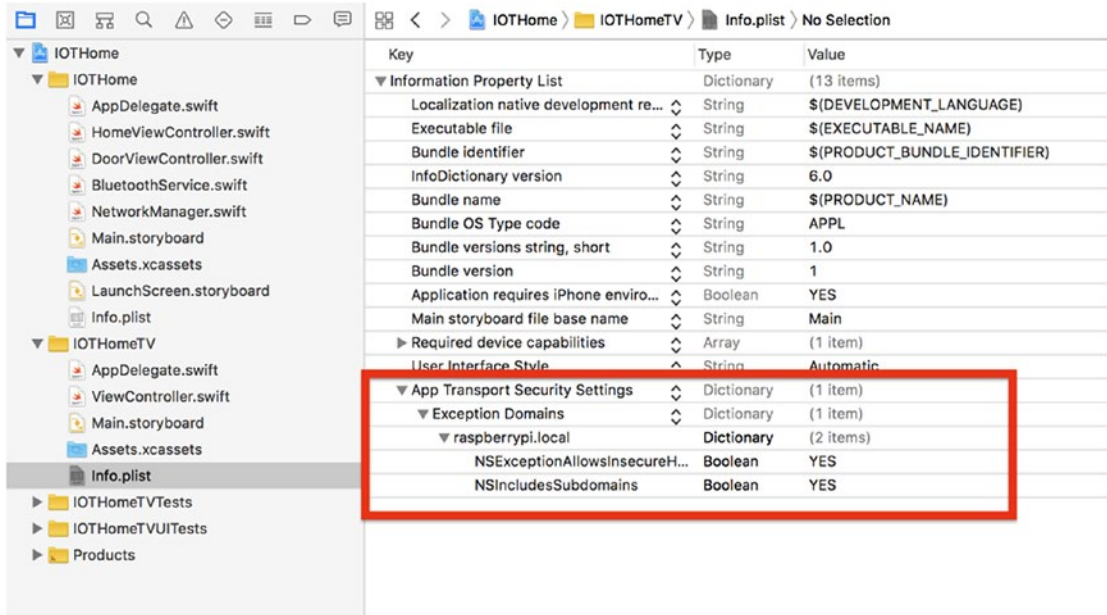


Figure 9-4. Info.plist file for the IOTHomeTV target, including the network whitelist

For the final project setup step, you have to make the NetworkManager class from the iOS target available to the tvOS target. One of the benefits of using the same project to manage both targets is that you can accomplish this task by using the same source file in both projects. As shown in Figure 9-5, to enable this capability, click the NetworkManager.swift file in the Project Navigator, then select the check box next to IOTHomeTV.

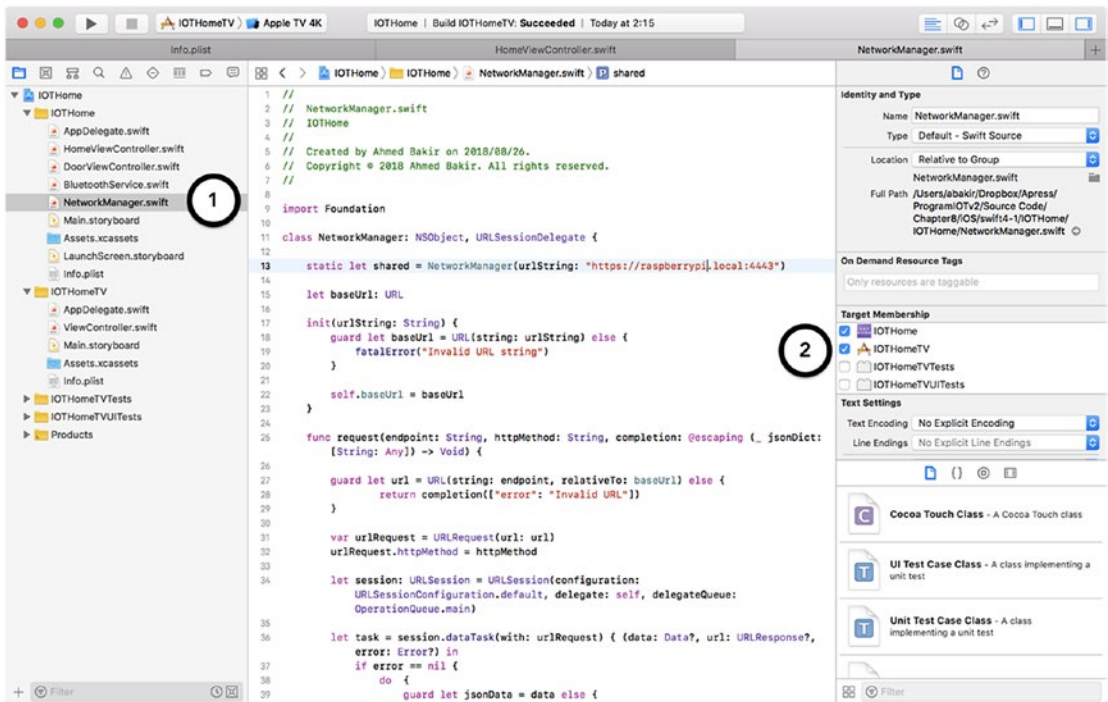


Figure 9-5. Sharing a source code between multiple targets in a project

To verify that the source file was added successfully and is compatible with tvOS, select the IOTHomeTV scheme and Apple TV 4K target device from the drop-down menu next to the Run button at the top of Xcode. You should be able to compile the target successfully with the new file included.

Creating the User Interface

Now that the new target for the tvOS is set up and compiles, you can begin building the user interface for the dashboard application. A dashboard is widely expected to be displayed for an extended period and should be easy to read. To achieve these goals, in Figure 9-6, I provide a wireframe for a tile-based user interface that displays the door status and inside temperature and humidity data from the IOTHome sensors, a three-day weather forecast, and the current outdoor temperature. The weather data is provided by querying OpenWeatherMap.org, employing the user's current location. You will import the icons from the `FontAwesome.swift` (<https://github.com/thii/FontAwesome.swift>) open source library.

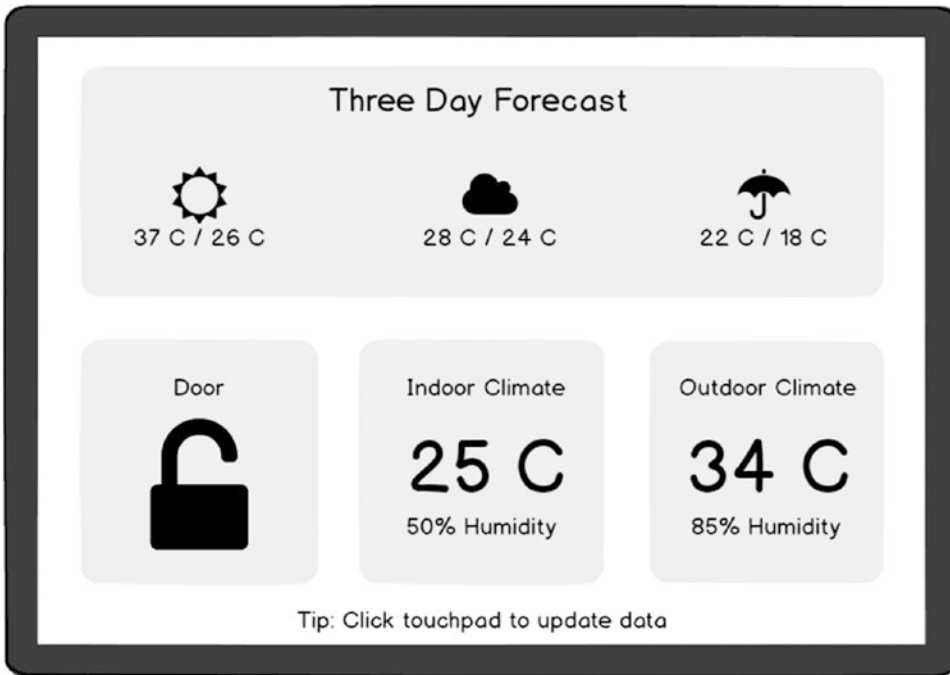


Figure 9-6. Wireframe for IOTHomeTV dashboard

One of the most convenient aspects of tvOS is that Apple allows you to reuse a significant majority of the development tools and practices you are familiar with from iOS development to help you build your apps. For the IOTHomeTV app, you can quickly build the user interface using Interface Builder and simple UIView, UIImage, and UILabel objects. To get started, open the Main.storyboard file for the IOTHomeTV target. The initial display of the empty storyboard should resemble the screenshot in Figure 9-7.

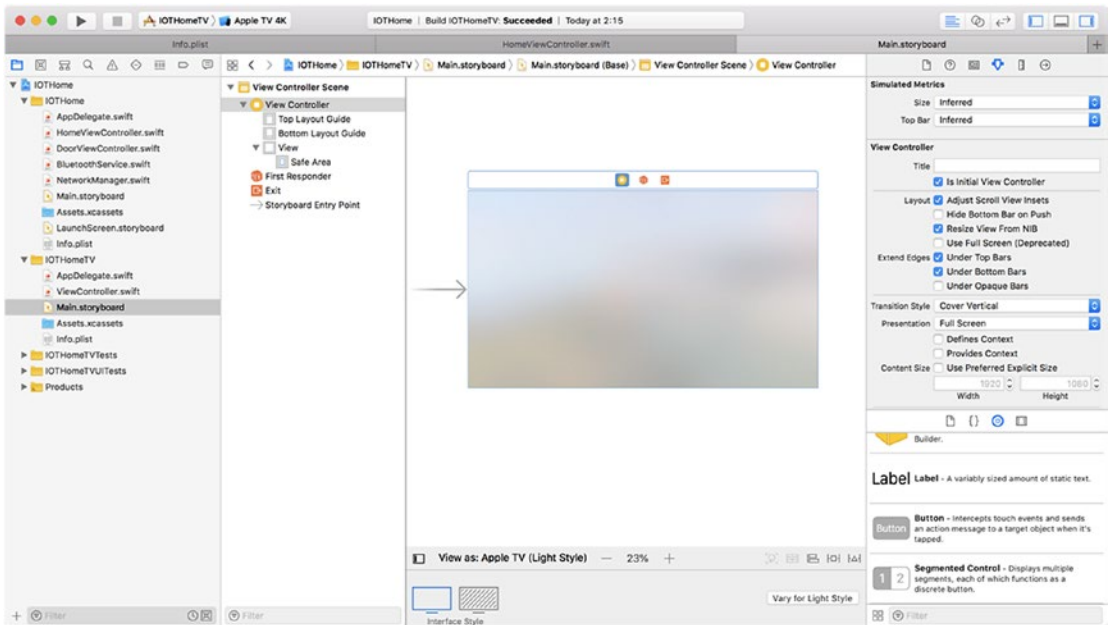


Figure 9-7. Blank storyboard for an Apple TV app

In the same manner as an iOS app, you can compose your user interface by dragging and dropping items from the Object Library at the bottom right of Interface Builder onto the storyboard. To further control the placement of elements, you can also use Auto Layout in the same manner as you would an iOS app. Using Table 9-1 as a guide, lay out the user interface. The labels and graphics should be placed over the tiles. Do not worry about the rounded borders on the views, as you will implement those programmatically later in this section.

Table 9-1. Styling for Main View Controller User Interface Elements

Element Name	Text Style	Align Relative to	Top Margin	Bottom Margin	Left Margin	Right Margin
“Three Day Forecast” background view	—	View	8	60	8	8
“Bottom Row” background views	—	View	60	40	60	60
“Tip” label	Title 3	View	40	20	8	8
“Title” labels	Title 2	Parent (center X)	30	—	—	—
“Icon” image views	—	Parent (center X, Y)	—	—	—	—
“Detail” text labels	Headline	“Icon” image views (center X)	20	—	—	—

As of this writing, there is only one aspect ratio supported by tvOS, so you can choose to skip setting the Auto Layout constraints for this project, if you would like to.

For the final step of setting up the user interface, you must link the storyboard to your source. In Listing 9-1, I have expanded the ViewController class for the IOTHomeTV project to include Interface Builder-accessible properties for all of the user interface elements.

Listing 9-1. View Controller Definition, Including User Interface Properties

```
class ViewController: UIViewController {

    @IBOutlet weak var forecastView: UIView?
    @IBOutlet weak var indoorView: UIView?
    @IBOutlet weak var outdoorView: UIView?
    @IBOutlet weak var lockView: UIView?

    @IBOutlet weak var firstDayLabel: UILabel?
    @IBOutlet weak var secondDayLabel: UILabel?
    @IBOutlet weak var thirdDayLabel: UILabel?

    @IBOutlet weak var indoorTempLabel: UILabel?
    @IBOutlet weak var indoorHumidityLabel: UILabel?
}
```

```

@IBOutlet weak var outdoorTempLabel: UILabel?
@IBOutlet weak var outdoorHumidityLabel: UILabel?

@IBOutlet weak var tipLabel: UILabel?

@IBOutlet weak var lockImageView: UIImageView?
@IBOutlet weak var firstDayImageView: UIImageView?
@IBOutlet weak var secondDayImageView: UIImageView?
@IBOutlet weak var thirdDayImageView: UIImageView?

override func viewDidLoad() {
    super.viewDidLoad()
}
}

```

After modifying the source file, link the properties to the storyboard, using the Connection Inspector in Xcode. In the same manner as in previous examples, you can review Chapter 1 for a refresher course on using Interface Builder. After making the connections, your completed storyboard should resemble my implementation in Figure 9-8.

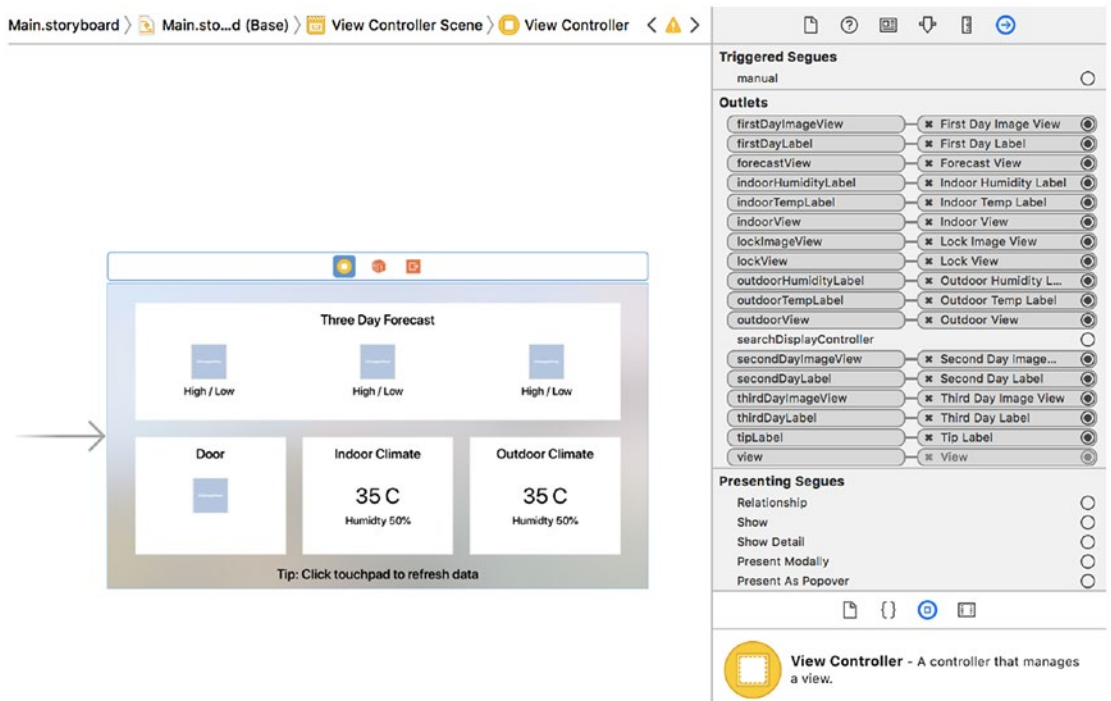


Figure 9-8. Completed storyboard for the IOTHomeTV app

Programmatically Styling Elements to Match the tvOS Design Language

Having laid out the user interface, you may have noticed it is a bit...plain. By adding soft shadows, rounded corners, and a blur effect to the backgrounds of the tiles, you can make a user interface that more closely matches what users are familiar with from the Apple TV home screen. These are changes you must make using code, but luckily, they are not very complicated to implement.

You can begin by adding the blur effect to the view. To achieve this, you can use the `UIVisualEffectView` class, which allows you to apply complicated visual effects, pre-built by Apple, to any `UIView`. To apply a modern iOS blur effect, use a `UIBlurEffect` object. In Listing 9-2, I applied the visual effect to the tiles by creating a helper method called `applyEffects()`.

Listing 9-2. Adding a Blur Effect to a View

```
class ViewController: UIViewController {
    ...
    override func viewDidLoad() {
        super.viewDidLoad()
        applyEffects(to: [forecastView, indoorView,
                        outdoorView, lockView])
    }

    func applyEffects(to views: [UIView?]) {
        for view in views {
            addBlurEffect(to: view)
        }
    }

    func addBlurEffect(to targetView: UIView?) {
        guard let targetView = targetView else { return }
        view.backgroundColor = UIColor.clear
        let blurEffect = UIBlurEffect(style: .regular)
        let blurView = UIVisualEffectView(effect:
            blurEffect)
        blurView.frame = view.bounds
    }
}
```



```

    targetView.addSubview(blurView)
    targetView.sendSubview(toBack: blurView)
}
}

```

Although you must add the visual effect as a sub-view, it does not have to block the content. By calling the `sendSubview(toBack:)` method, you can blur the effect to the back of the tile without affecting the previous layout of the view from the storyboard. The logic for applying the effects is called from the `viewDidLoad()` method for the view controller, as that method is executed after the view has been laid out from the storyboard and the view controller is ready to use.

Adding the corner radius is a more straightforward process. To apply a corner radius, provide a numerical value to apply to the `CALayer` for the view (the object that represents how the view is drawn) and specify that you want to clip the area under the rounded corners. In Listing 9-3, I have expanded the `applyEffects()` method to include the corner radius effect.

Listing 9-3. Adding a Rounded Corner to a View

```

class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        applyEffects(to: [forecastView, indoorView,
            outdoorView, lockView], cornerRadius: 20)
    }

    func applyEffects(to views: [UIView?],
        cornerRadius: CGFloat) {
        for view in views {
            addBlurEffect(to: view)
            addRoundedCorners(to: view, cornerRadius:
                cornerRadius)
        }
    }

    func addRoundedCorners(to targetView: UIView?,
        cornerRadius: CGFloat) {
        guard let targetView = targetView else { return }

```

```

        targetView.layer.cornerRadius = cornerRadius
        targetView.layer.masksToBounds = true
    }
}

```

For the shadow, you need to combine concepts from both of these effects. A shadow by itself is created by applying a shadow color, blur radius, and shadow position to a CALayer. This serves to replicate soft or hard light in real life. Unfortunately, the required masking to enable the rounded corners would clip the shadows, if applied to the same view. You can work around this by creating another view with the same position, applying the shadow effect to this new view and placing it underneath the content view. In Listing 9-4, I have expanded the view controller to implement these steps and add a shadow effect to the tiles.

Listing 9-4. Adding a Shadow Under a View with Rounded Corners

```

class ViewController: UIViewController {
    ...
    func applyEffects(to views: [UIView?],
        cornerRadius: CGFloat) {
        for view in views {
            addBlurEffect(to: view)
            addRoundedCorners(to: view,
                cornerRadius: cornerRadius)
            addShadow(to: view, cornerRadius: cornerRadius)
        }
    }
}

func addShadow(to targetView: UIView?, cornerRadius:
    CGFloat) {
    guard let targetView = targetView else { return }
    let shadowView = UIView(frame: targetView.frame)
    shadowView.layer.cornerRadius = cornerRadius

    shadowView.layer.shadowOffset = CGSize.zero
    shadowView.layer.shadowOpacity = 0.2
    shadowView.layer.shadowRadius = 10.0
    shadowView.layer.shadowColor = UIColor.black.cgColor
}

```

```

let shadowPath = UIBezierPath(roundedRect:
    shadowView.bounds, cornerRadius: cornerRadius)
shadowView.layer.shadowPath = shadowPath.cgPath

view.addSubview(shadowView)
view.bringSubview(toFront: targetView)
}
}

```

Note The `frame` property of a `UIView` includes its `x` and `y` positions. The `bounds` property sets these to (0,0). When adding a sub-view to a view, try to use `bounds`. When copying or moving a view, try to use `frame`.

After applying both visual effects and running the application in the Apple TV simulator, your output should resemble the screenshot in Figure 9-9.

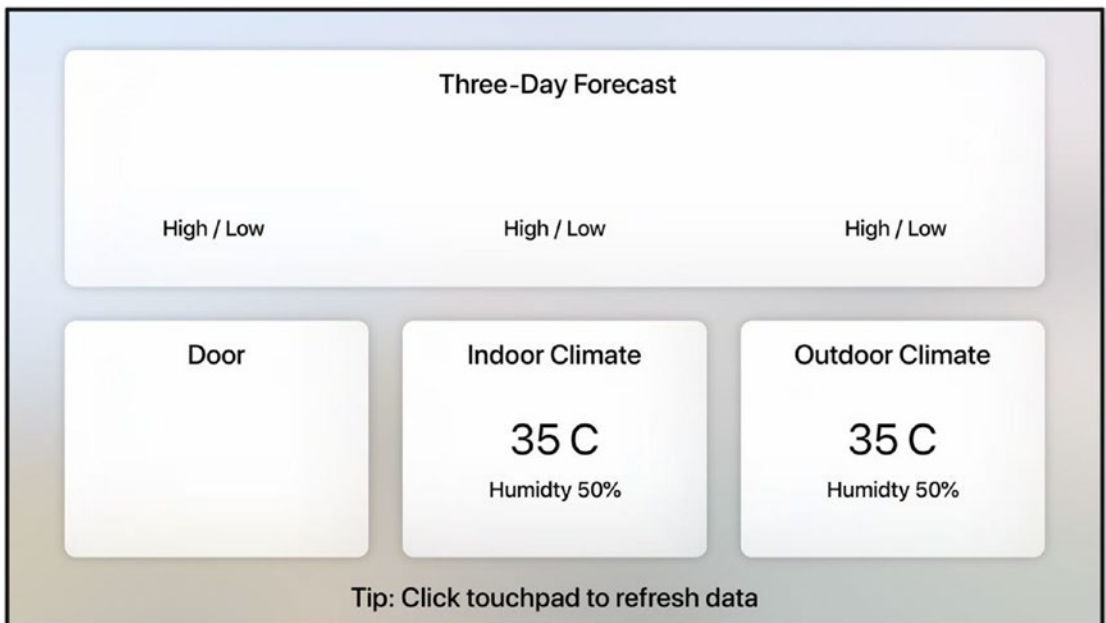


Figure 9-9. Stylized user interface for the *IOTHomeTV* app

Using Font Awesome for Font-Based Graphics

For the final piece of the user interface puzzle, you must add graphics to the application. Instead of importing graphics files directly, for this section, I would like to introduce a very popular font-based alternative, Font Awesome, and its Swift implementation, FontAwesome.swift. Font Awesome is a font file that provides a massive collection of icons. It is a popular choice in web development for helping reduce page size and the amount of work you have to hire a graphic designer for. On iOS, it provides the same benefits, in addition to allowing you to remove the burden of managing and scaling images yourself. The implementation you will use in this section, FontAwesome.swift, allows you to create UIImage objects based on the font, making its use exactly the same as if you were using graphics from an Assets Catalog or other source.

To get started, download or clone the repository for FontAwesome.swift from its GitHub page: <https://github.com/thii/FontAwesome.swift>. Next, copy all of the files in the archive's FontAwesome folder to your project, except for the Info.plist file. You can perform this operation by using the Add Files to IOTHome option in the File menu. As shown in Figure 9-10, when the file section pop-up window appears, select IOTHome and IOTHomeTV as the targets to include the files in both the iOS and tvOS apps. Make sure you also select Copy Items If Needed, to copy the files to add a copy of the files to the project.

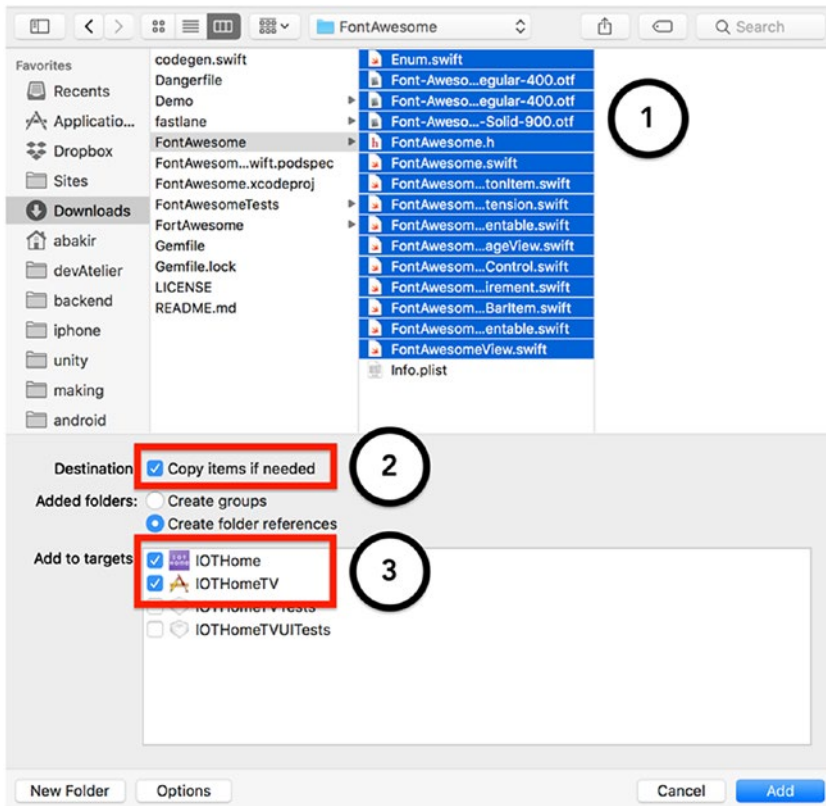


Figure 9-10. Importing the files for *FontAwesome.swift* into the *IOTHome* project

After the files for *FontAwesome.swift* have been added to your project, select all of them, then secondary-click (right-click), to present the context menu. Select *New Group from Selection*, to place all of the files in a single folder in your project. Your Xcode Project Navigator should now resemble my example in Figure 9-11.

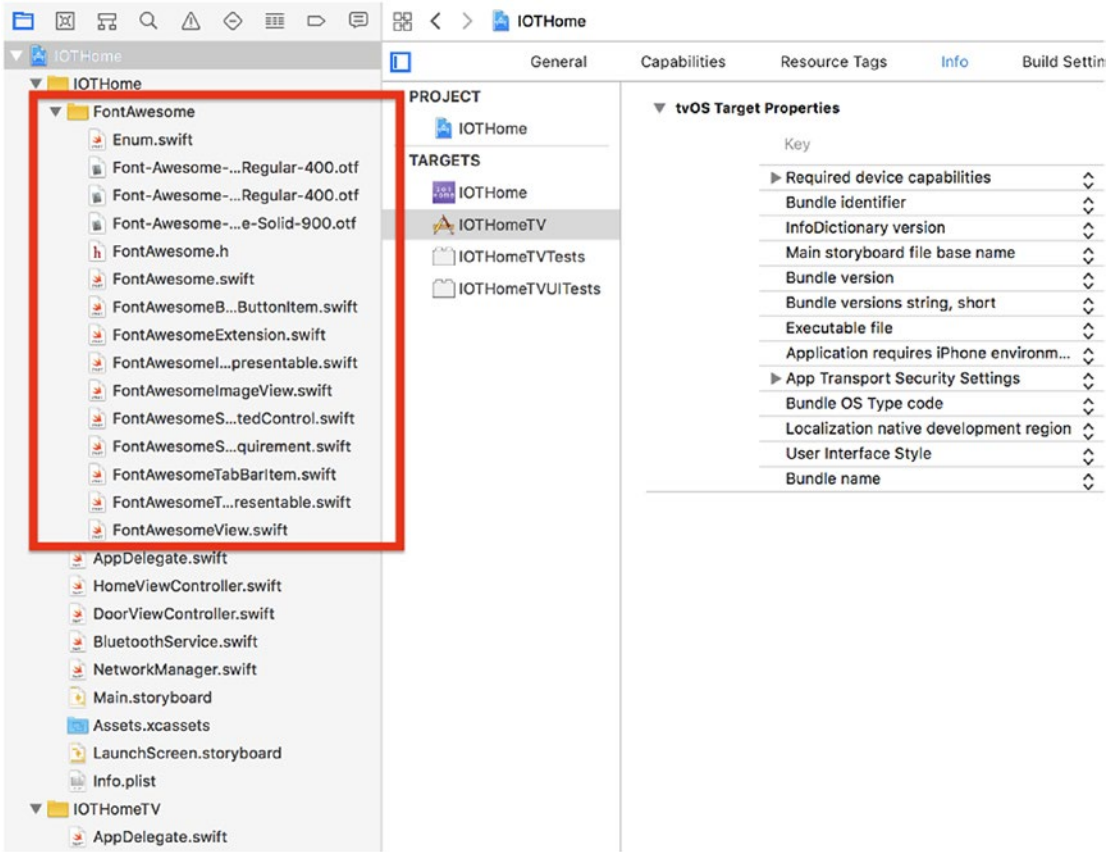


Figure 9-11. Project Navigator after adding FontAwesome.swift to the project

Finally, to use the library, you must create UIImage objects based on icons available in the font. Rather than pure Unicode Hex codes alone, Font Awesome provides names for each icon. To discover what icons are available, and their names, I like to use the official search tool for the font, available at: <https://fontawesome.com/icons?d=gallery&m=free>. To start the project off, use lock for the lock graphic and sun for the sunny weather graphic.

To create the graphic, you can use FontAwesome.swift’s extension for the UIImage class, UIImage.fontAwesome(name:style:textColor:size:). As the name suggests, you have to call the API using the icon’s name, display style, a tint color, and the size. In Listing 9-5, I have added these calls to the ViewController class.

Listing 9-5. Adding FontAwesome–Based Images to Image Views

```

class ViewController: UIViewController {
    ...
    override func viewDidLoad() {
        super.viewDidLoad()
        applyEffects(to: [forecastView, indoorView,
            outdoorView, lockView], cornerRadius: 20)

        addFontAwesomeImage(to: lockImageView, name: .lock)
        addFontAwesomeImage(to: firstDayImageView, name: .sun)
        addFontAwesomeImage(to: secondDayImageView, name: .sun)
        addFontAwesomeImage(to: thirdDayImageView, name: .sun)
    }

    func addFontAwesomeImage(to imageView: UIImageView?,
        name: FontAwesome) {
        guard let imageView = imageView else { return }
        imageView.image = UIImage.fontAwesomeIcon(name: name,
            style: .solid,
            textColor: UIColor.black,
            size: imageView.bounds.size)
    }
}

```

After adding the image views, running the application in the Apple TV simulator should provide output resembling that in the screenshot in [Figure 9-12](#).

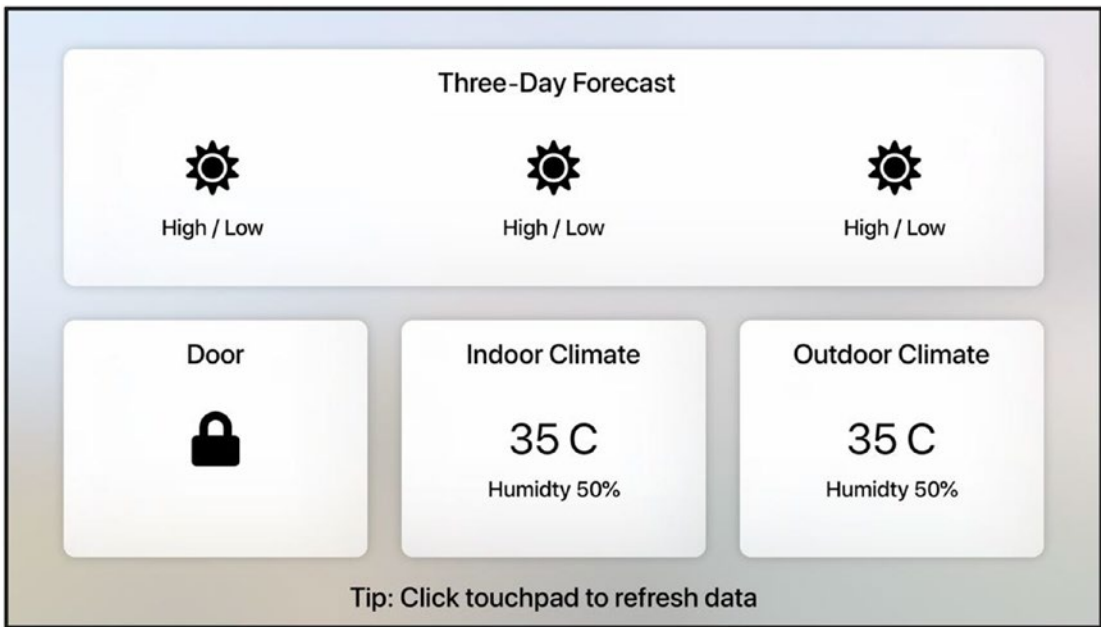


Figure 9-12. IOTHomeTV with Font Awesome images

Adding Data Sources to the tvOS App

Now that the user interface for the IOTHomeTV app is ready, you can begin integrating its data sources and populating it with real data from the IOTHome sensors you built in Chapters 5–7 and OpenWeatherMap.org. Although you can connect directly to Bluetooth devices from an Apple TV, for this system, it is more efficient and reliable to take advantage of the heavy lifting on the web server. In addition to being able to use a consistent interface, removing the need to add additional clients will free up the sensors to accept more connections.

At the beginning of the chapter, you were able to verify that the network client code from Chapter 8 compiles without issue on tvOS. To integrate the client, you simply have to call it from the IOTHomeTV app. In Listing 9-6, I began this integration by using the `NetworkManager` class to request the indoor climate data from the sensor on the Raspberry Pi.

Listing 9-6. Fetching Temperature Data from the IOTHome Web Server in the tvOS App

```

class ViewController: UIViewController {
    ...
    override func viewDidLoad() {
        super.viewDidLoad()
        applyEffects(to: [forecastView, indoorView,
            outdoorView, lockView], cornerRadius: 20)
        ...
        fetchNetworkData()
    }
    ...
    func fetchNetworkData() {
        NetworkManager.shared.getTemperature { [weak self]
            (resultDict: [String: Any]) in

            if let error = resultDict["error"] as? String {
                self?.tipLabel?.text = "Error: \(error)"
            } else {
                DispatchQueue.main.async {
                    if let temperature =
                        resultDict["temperature"] as? String {
                        self?.indoorTempLabel?.text =
                            "\(temperature) C"
                    }

                    if let humidity = resultDict["humidity"]
                        as? String {
                        self?.indoorHumidityLabel?.text =
                            "Humidity \(humidity)%"
                    }
                }
            }
        }
    }
}

```

Looking back at Chapter 8, you will remember that the Network Manager is responsible for managing the address of the server, using `NSURLSession` to make the network connection, and validating the JSON response. When a network request is complete, the manager executes the completion handler specified by the user, whose return parameter is a JSON dictionary containing an `error` key-value pair or the response from the target end point. Inside the completion handler, I used the temperature and humidity key-value pairs to update the text for the Indoor Temperature and Indoor Humidity labels. The network request is triggered from the `viewDidLoad()` method, so that the app can update the data when it is launched.

In Listing 9-7, I have expanded the `fetchNetworkData()` method further, to include the status from the door sensor. Instead of updating a label, in that sample, I updated the image for the door status.

Listing 9-7. Fetching Door Sensor Data from the IOTHome Web Server in the tvOS App

```
class ViewController: UIViewController {
    ...
    func fetchNetworkData() {
        NetworkManager.shared.getTemperature { [weak
            self] (resultDict: [String: Any]) in
            ...
        }
        NetworkManager.shared.getDoorStatus{ [weak self]
            (resultDict: [String: Any]) in
            if let error = resultDict["error"] as? String {
                self?.titleLabel?.text = "Error: \(error)"
            } else {
                DispatchQueue.main.async { [weak self] in
                    if let doorStatus =
                        resultDict["doorStatus"] as? String {
```

```

if doorStatus == "0" {
    self?.addFontAwesomeImage(to:
        self?.lockImageView, name:
            .lockOpen)
} else {
    self?.addFontAwesomeImage(to:
        self?.lockImageView, name: .lock)
}
}
}
}
}
}
}
}

```

Requesting User Location

At the beginning of the chapter, I described how I thought it would be useful to display a three-day forecast and current weather conditions in the IOTHomeTV dashboard. Although users are often aware of the conditions inside their home, having data on the weather outside allows them to better prepare for when they must leave. In order to request location-based forecasts, you must request the user’s current location, just as in Chapter 2, when you used location tracking to build a workout app.

As with the project in Chapter 2, before you can request a user’s location, you must set a description for the tvOS location permission request dialog. As shown in Figure 9-13, open the `Info.plist` file for the IOTHomeTV target, then add the Privacy – Location When in Use key-value pair. For my description string, I used the text: “IOTHomeTV would like to use your location to show you weather information for your area.”

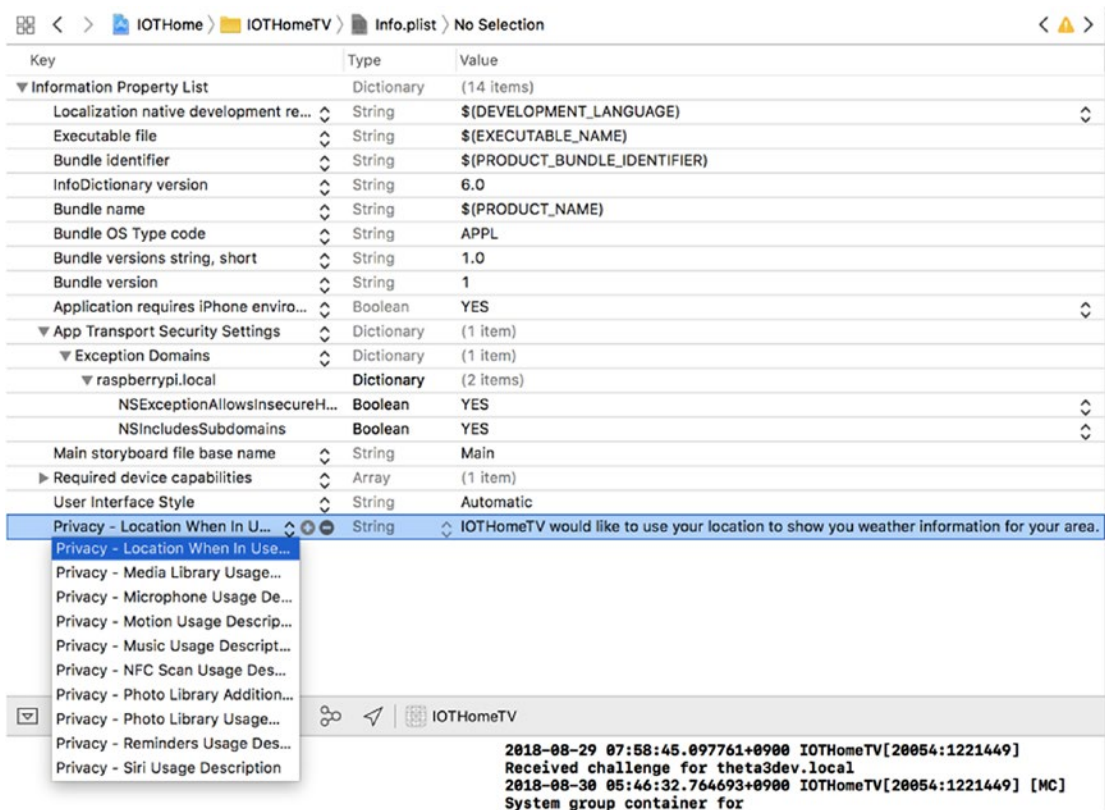


Figure 9-13. Adding a user location permission description to the IOTHomeTV app

Next, you must add the permission request after the app launches. Just as on iOS, user location on tvOS is managed through a CLLocationManager object. As shown in Listing 9-8, initialize an object, to manage this request; implement the viewWillAppear() method, to make the request; and the locationManager(didChangeAuthorization status:) delegate method to handle the result.

Listing 9-8. Requesting User Location Permission from the IOTHomeTV App

```
import UIKit
import CoreLocation

class ViewController: UIViewController {
    ...
    let locationManager = CLLocationManager()
    ...
}
```

```

override func viewDidLoad() {
    super.viewDidLoad()
    locationManager.delegate = self
    ...
}
override func viewWillAppear(_ animated: Bool) {
    super.viewWillAppear(animated)

    if CLLocationManager.authorizationStatus() ==
    .authorizedWhenInUse {
        locationManager.requestLocation()
    } else {
        locationManager.requestWhenInUseAuthorization()
    }
}
    ...
}

extension ViewController : CLLocationManagerDelegate {
    func locationManager(_ manager: CLLocationManager,
    didChangeAuthorization status: CLAuthorizationStatus) {
        NSLog("Authorization state: \(status)")
    }
    func locationManager(_ manager: CLLocationManager,
    didFailWithError error: Error) {
        let errorString = "Location error:
            \(error.localizedDescription)"
        tipLabel?.text = errorString
        NSLog(errorString)
    }
}

```

To verify the result, run the app in the Apple TV 4K simulator. You should receive a full-screen alert asking you for your user location permission, as shown in Figure 9-14.

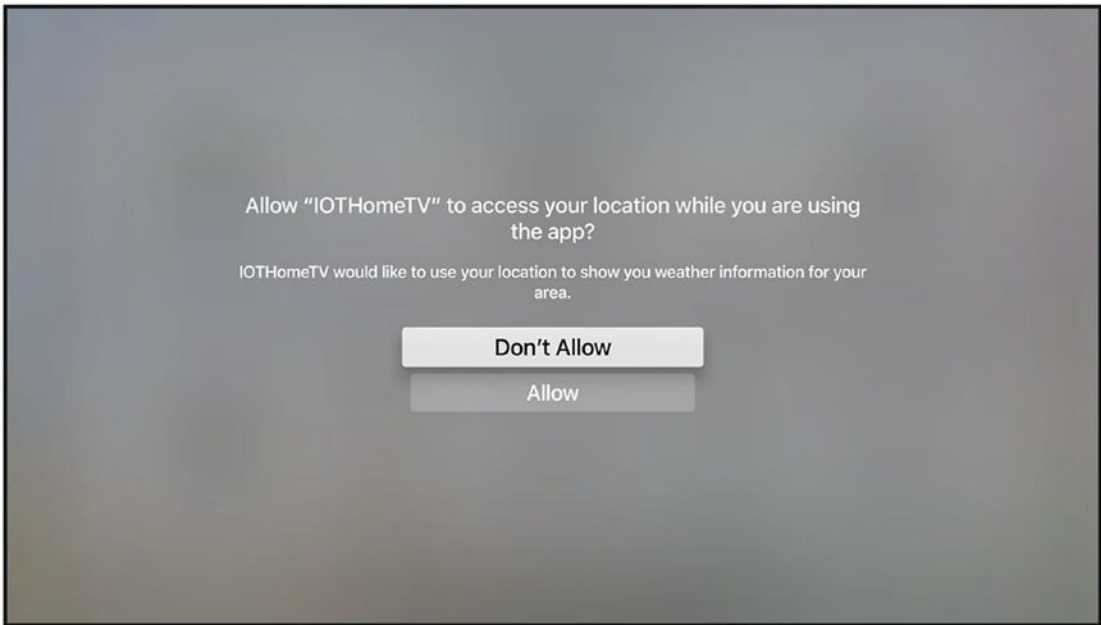


Figure 9-14. User Location permission pop-up for the IOTHomeTV app

Finally, to request the current user location, modify the `locationManager(didChangeAuthorization status:)` delegate method to request the current user location, after you have verified that permission has been granted. As shown in Listing 9-9, add another delegate method, `locationManager(didUpdateLocations locations:)`, to handle the location update, and then save the result in a property you can use later.

Listing 9-9. Requesting Current User Location from the IOTHomeTV App

```
import UIKit
import CoreLocation

class ViewController: UIViewController {
    ...
    let locationManager = CLLocationManager()
    var lastSavedLocation : CLLocation?
    ...
    override func viewDidLoad(_ animated: Bool) {
        ...
    }
}
```

```

extension ViewController : CLLocationManagerDelegate {
    func locationManager(_ manager:
        CLLocationManager, didChangeAuthorization
        status: CLAuthorizationStatus) {
        NSLog("Authorization state: \(status)")
        if status == .authorizedWhenInUse {
            locationManager.requestLocation()
        }
    }
    func locationManager(_ manager: CLLocationManager
    didUpdateLocations locations: [CLLocation]) {
        lastSavedLocation = locations.first
    }
}

```

Connecting to the OpenWeatherMap API

With the user interface, network functionality, and user location in place, you are almost ready to start using OpenWeatherMap's public weather database, to add outside weather information to the IOTHomeTV app. For the final piece in the setup puzzle, you must request an API key from OpenWeatherMap.org. Many services that allow others to use their data through a web API often require an API key to authenticate requests or enforce data access, according to membership level. For example, OpenWeatherMap.org allows up to 60 requests a minute on a free account, but anything above this requires a paying membership. As shown in Figure 9-15, to request a free account, navigate to www.openweathermap.org in your browser, then click the Sign Up link.

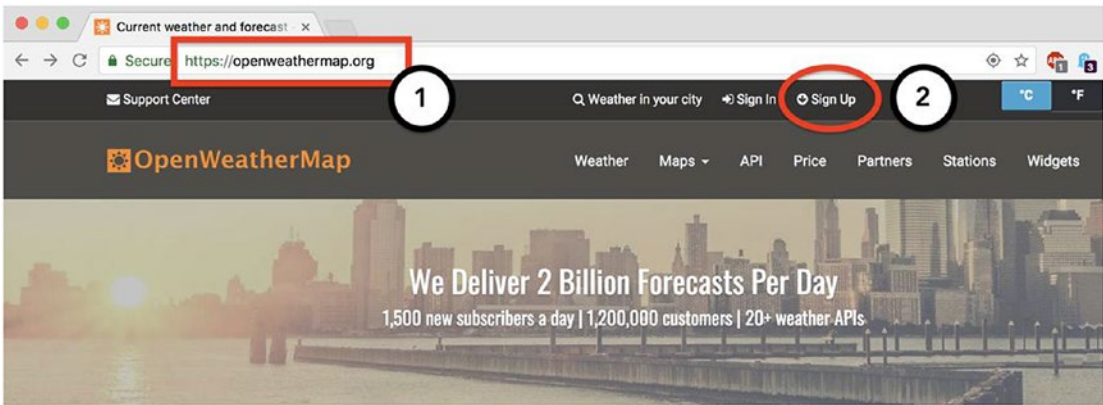


Figure 9-15. *Creating an account on OpenWeatherMap.org*

After you have completed creating your account, you will be redirected to the API keys page, pictured in Figure 9-16, which lists all of the OpenWeatherMap API keys available to you. Make a note of the Default key and do not share it with others, or you may risk losing access to your account.

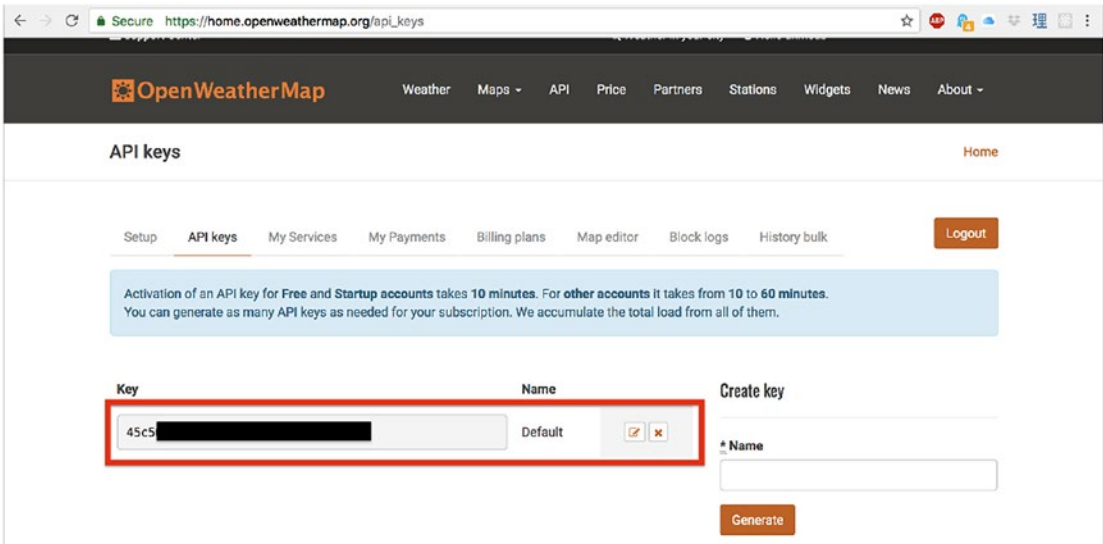


Figure 9-16. *OpenWeatherMap.org API keys page*

Now that you have an OpenWeatherMap API key and the user's current location, you can begin using the service to fetch weather data. To populate the weather fields on the app, you will use two end points from OpenWeatherMap: /weather (for current weather conditions) and /forecast (for the three-day forecast). Looking at the documentation

for the /weather end point (<https://openweathermap.org/current>), you will notice that you can use URL parameters to call the end point with geographic coordinates. URL parameters are a way of passing values to an end point by appending them to the end of the URL. For the coordinate (35.730534, 139.705001), you would use the URL:

```
https://api.openweathermap.org/data/2.5/weather?lat=35.730534&lon=139.705001&units=metric&appid=YOUR_APP_ID
```

In Listing 9-10, I have included the JSON response for this API call. For the Outdoor Conditions label, you need only the temperature and humidity fields, which you can extract from the temp and humidity key-value pairs inside the main dictionary.

Listing 9-10. Sample JSON Response for the OpenWeatherMap Current Conditions End Point

```
{
  "coord": {
    "lon": 139.71,
    "lat": 35.73
  },
  "weather": [{
    "id": 803,
    "main": "Clouds",
    "description": "broken clouds",
    "icon": "04n"
  }],
  "base": "stations",
  "main": {
    "temp": 27.64,
    "pressure": 1009,
    "humidity": 74,
    "temp_min": 26,
    "temp_max": 29
  },
  ...
}
```

The `NetworkManager` class so far has always used end points relative to the domain for the IOTHome web server. To support OpenWeatherMap, you will have to expand the Network Manager to use multiple base URLs and append parameters to the end of the URL. One special requirement of URL parameters is that the first parameter must be prefixed with `?`, and every consecutive parameter must be prefixed with `&`. Rather than writing the logic for this yourself, you can take advantage of the `URLComponents` class to build a properly formatted URL. In Listing 9-11, I have expanded the `request(endpoint: httpMethod: completion:)` method in the Network Manager to accept a base URL and URL parameter dictionary as input parameters. Additionally, I have modified the remaining methods with the new parameters and added a `formattedURL(baseUrl: endpoint: parameters:)` method to build the formatted URL.

Listing 9-11. Expanding the `NetworkManager` Class to Support Multiple Base URLs and URL Parameters

```
class NetworkManager: NSObject, URLSessionDelegate {
    let deviceBaseUrl = "https://raspberrypi.local"
    let opmBaseUrl = "https://api.openweathermap.org/data/2.5"
    let opmApiKey = "YOUR_API_KEY"
    static let shared = NetworkManager()

    func formattedUrl(baseUrl: String, endpoint: String,
        parameters: [String: String]? ) -> URL? {
        guard var urlComponents = URLComponents(string:
            "\(baseUrl)/\(endpoint)") else {
            return nil
        }

        urlComponents.queryItems = parameters?.compactMap({
            pair in
                URLQueryItem(name: pair.key, value: pair.value)
        })

        return urlComponents.url?.absoluteURL
    }
}
```

```

func request(baseUrl: String, endpoint: String,
  httpMethod: String, parameters: [String: String]? =
  nil, completion: @escaping (_ jsonDict: [String: Any]) -> Void) {

  guard let url = self.formattedUrl(baseUrl: baseUrl,
    endpoint: endpoint, parameters: parameters) else {
    return completion(["error": "Invalid URL"])
  }

  var urlRequest = URLRequest(url: url)
  urlRequest.httpMethod = httpMethod
  ...
}
...
func getTemperature(completion: @escaping (_
  jsonDict: [String: Any]) -> Void) {
  request(baseUrl: deviceBaseUrl, endpoint:
    "temperature", httpMethod: "GET") {
    (resultDict: [String: Any]) in
    completion(resultDict)
  }
}
...
func getDoorStatus(completion: @escaping (_
  jsonDict: [String: Any]) -> Void) {
  connectDoor { [weak self] (result: [String:
    Any]) in
    if (result["error"] as? String) != nil {
      return completion(result)
    } else {
      guard let deviceBaseUrl =
        self?.deviceBaseUrl else {
        return completion(["error":
          "Invalid device URL"])
      }
      self?.request(baseUrl: deviceBaseUrl,
        endpoint: "door/status",

```

```

        httpMethod: "GET") { (resultDict
                               [String: Any]) in
        completion(resultDict)
    }
}
}

func connectDoor(completion: @escaping (_
    jsonDict: [String: Any]) -> Void) {
    request(baseUrl: deviceBaseUrl, endpoint:
        "door/connect", httpMethod: "POST") {
        (resultDict: [String: Any]) in
        completion(resultDict)
    }
}

func disconnectDoor(completion: @escaping (_
    jsonDict: [String: Any]) -> Void) {
    request(baseUrl: deviceBaseUrl, endpoint:
        "door/disconnect", httpMethod: "POST") {
        (resultDict: [String: Any]) in
        completion(resultDict)
    }
}
}
}

```

In this example, rather than using a for loop to build the array of `URLQueryItem` objects, I used the new `compactMap()` method. This API takes cues from functional programming languages (such as Haskell) and has been taking off in popularity recently as a way of performing a calculation based on iterating through a set.

Now that the Network Manager is ready to handle your new requirements, you can add a `getOutdoorTemperature()` method to the Network Manager, to make the request and handle the response in the main view controller of the app, as shown in Listing 9-12.

Listing 9-12. Requesting Current Weather Conditions Using the Network Manager

```

class NetworkManager: NSObject, URLSessionDelegate {
    ...
    func getOutdoorTemperature(latitude: String, longitude:
        String, completion: @escaping (_ jsonDict: [String: Any])
        -> Void) {
        let parameters = ["appid": opmApiKey,
            "lat": latitude,
            "lon": longitude,
            "units": "metric"]
        request(baseUrl: opmBaseUrl, endpoint: "weather",
            httpMethod: "GET", parameters: parameters) {
            (resultDict: [String: Any]) in
                completion(resultDict)
        }
    }
}

class ViewController: UIViewController {
    ...
    @IBAction func fetchNetworkData() {
        ....
        fetchOutdoorTemperature()
    }
    ...
    func fetchOutdoorTemperature() {
        guard let latitude =
            lastSavedLocation?.coordinate.latitude,
            let longitude =
            lastSavedLocation?.coordinate.longitude else {
            return
        }

        NetworkManager.shared.getOutdoorTemperature(latitude:
            "\(latitude)", longitude: "\(longitude)") { [weak

```


in the location delegate will enable you to display valid data the first time the user loads the application.

OpenWeatherMap provides a free end point (`/forecast`) and a paid end point (`/forecast/daily`) for getting the daily forecast. The paid end point is significantly more convenient to use, but to expand the access of this book, I will cover using the free end point. The API documentation for the `/forecast` end point is available at <https://openweathermap.org/forecast5>. In Listing 9-13, I have expanded the Network Manager to include a method for calling the `/forecast` end point. There are no surprises here; it closely resembles the `/weather` end point.

Listing 9-13. Adding a Method for Requesting the Forecast from OpenWeatherMap

```
class NetworkManager: NSObject, URLSessionDelegate {
    ...
    func getForecast(latitude: String, longitude: String,
        completion: @escaping (_ jsonDict: [String: Any]) ->
        Void) {
        let parameters = ["appid": opmApiKey,
                "lat": latitude,
                "lon": longitude,
                "units": "metric"]
        request(baseUrl: opmBaseUrl, endpoint: "forecast",
                httpMethod: "GET", parameters: parameters) {
            (resultDict: [String: Any]) in
                completion(resultDict)
        }
    }
}
```

The response provides the daily forecast for five days, in three-hour blocks. Within the record for each block, you can find a human-readable description of the condition (for example: Sunny) and detailed statistics ranging from minimum temperature to barometric pressure. In Listing 9-14, I have provided an extremely abbreviated sample of one of these responses.

Listing 9-14. Sample Response for OpenWeatherMap Forecast End Point

```

{
  "cod": "200",
  "message": 0.1654,
  "cnt": 38,
  "list": [{
    "dt": 1535781600,
    "main": {
      "temp": 287.67,
      "temp_min": 287.67,
      "temp_max": 288.554,
      "pressure": 1019.41,
      ...
    },
    "weather": [{
      "id": 800,
      "main": "Clear",
      "description": "clear sky",
      "icon": "01n"
    }],
    ...
  }],
  "city": {
    "id": 5391959,
    "name": "Tokyo",
    ...
  }
}

```

Unfortunately, this API is much more complicated to use than the /weather end point. The data for each three-hour block is inside an array of dictionaries. As you can guess, you will have to traverse the sub-dictionaries to extract the values you require (list, main, weather). A common way of accomplishing this is by nesting guard-let statements.

To get an accurate average maximum and minimum temperature, you will have to average the results for each day. However, to demonstrate API usage, I will abbreviate this and use three samples that are spaced 24 hours apart. In Listing 9-15, I have expanded the view controller to include the request for the forecast.

Listing 9-15. Requesting the Forecast from the View Controller

```
class ViewController: UIViewController {
    ...
    func fetchOutdoorTemperature() {
        ...
        NetworkManager.shared.getForecast(latitude:
            "\(latitude)", longitude: "\(longitude)") {
            [weak self] (resultDict: [String: Any]) in

                if let error = resultDict["error"] as? String {
                    self?.titleLabel?.text = "Error: \(error)"
                } else {
                    guard let resultList = resultDict["list"]
                        as? [Any] else {
                        self?.titleLabel?.text = "Error: Invalid
                            response"
                        return
                    }
                    guard resultList.count > 15 else { return }
                    //today
                    self?.setupForecastView(dictionary:
                        resultList[0], index: 0)
                    //tomorrow
                    self?.setupForecastView(dictionary:
                        resultList[7], index: 1)
                    //the day after
                    self?.setupForecastView(dictionary:
                        resultList[15], index: 2)
                }
            }
        }
    }
}
```

I have offloaded processing the request and mapping it to the views in the `setupForecastView(dictionary:index:)` method, provided in Listing 9-16.

Listing 9-16. Processing the Forecast Request in the View Controller

```
class ViewController: UIViewController {
    ...
    func setupForecastView(dictionary: Any, index: Int) {
        guard let dayDict = dictionary as? [String: Any],
            let mainDict = dayDict["main"] as? [String: Any],
            let weatherArray = dayDict["weather"] as? [Any],
            let weatherDict = weatherArray.first as? [String:
                Any],
            let minTemp = mainDict["temp_min"] as? Double,
            let maxTemp = mainDict["temp_max"] as? Double,
            let conditionCode = weatherDict["id"] as? Int
            else { return }

        let icon: FontAwesome
        switch conditionCode {
            case 300...599: icon = .umbrella
            case 600...699: icon = .snowflake
            case 700...799: icon = .exclamationTriangle
            case 800: icon = .sun
            default: icon = .cloud
        }

        switch index {
        case 0:
            guard let imageView = firstDayImageView else { return }
            firstDayLabel?.text = "\(maxTemp) / \(minTemp)"
            firstDayImageView?.image =
                UIImage.fontAwesomeIcon(name: icon, style:
                    .solid, textColor: UIColor.black, size:
                    imageView.bounds.size)
        }
    }
}
```

```

case 1:
    guard let imageView = secondDayImageView else {
        return }
    secondDayLabel?.text = "\(\maxTemp) / \(\minTemp)"
    secondDayImageView?.image =
        UIImage.fontAwesomeIcon(name: icon, style: .solid,
            textColor: UIColor.black, size: imageView.bounds.size)
    default:
        guard let imageView = thirdDayImageView else {
            return }
        thirdDayLabel?.text = "\(\maxTemp) / \(\minTemp)"
        thirdDayImageView?.image =
            UIImage.fontAwesomeIcon(name: icon, style:
                .solid, textColor: UIColor.black, size:
                    imageView.bounds.size)
    }
}

```

This method contains a lot of logic to extract the nested values from the dictionaries for each block of data and map them to the user interface. Unfortunately, this is a common design oversight in many big data APIs. If possible, use this experience as an opportunity to advocate for simple network API responses in your back-end projects.

Handling Touch Input from the Siri Remote

For the final step in developing the IOTHomeTV app, you should take advantage of the primary human interface device for the Apple TV, the Siri Remote. Every Apple TV has included a remote control with the device; however, the fourth-generation Apple TV introduced the Siri Remote, with a touchpad for app-like gestures and a microphone for accepting Siri voice commands. In this section, I will focus on how to accept button and touchpad click input from the remote, to refresh the data in the application.

To begin, you can implement a button press. The basic interface of every Apple TV has relied on using directional buttons for moving between items, using the Play/Pause button to navigate to the detail page for an item, and the Menu button to navigate out of the item. For the IOTHomeTV, the natural interaction to implement would be to use the Play/Pause button to refresh the data in the application.

You can implement support for button presses on tvOS, using the `UITapGestureRecognizer` class. If you have implemented your own custom gesture recognizer on iOS (for example, making a view swipeable or clickable without using `UIButton` or `UIPageController`), you are already familiar with the `UITapGestureRecognizer` class and its parent class, `UIGestureRecognizer`. Their implementations are mostly the same on tvOS. The way gesture recognizers work is that you specify a gesture to observe (for example, tap, touch, swipe, pinch), a view to observe these events on, and a selector (method signature) to call when these events are recognized. You specify your event by instantiating a single-purpose subclass of `UIGestureRecognizer`, such as `UITapGestureRecognizer` or `UISwipeGestureRecognizer`. `UIGestureRecognizer` is an abstract class and cannot be instantiated directly.

To implement the Play/Pause button handler, instantiate a `UITapGestureRecognizer` object. As shown in Listing 9-17, specify the `fetchNetworkData()` method as the handler for the tap event by creating a selector with its method signature. In the same manner as on iOS, use the `addGestureRecognizer()` method from the `UIView` class, to attach the gesture to the main view for the view controller. Unlike iOS, you can limit button presses to only the Play/Pause button, using the `allowedPressTypes` property.

Listing 9-17. Adding a Play/Pause Button Gesture Recognizer

```
class ViewController: UIViewController {
    ...
    override func viewDidLoad() {
        super.viewDidLoad()
        ...
        setupGestureHandlers()
    }

    func setupGestureHandlers() {
        let tapRecognizer = UITapGestureRecognizer(target:
            self, action:
            #selector(ViewController.fetchNetworkData))
        tapRecognizer.allowedPressTypes = [NSNumber(value:
            UIPressType.playPause.rawValue)]
        self.view.addGestureRecognizer(tapRecognizer)
    }
}
```

To test the gesture recognizer, run the Apple TV simulator. As shown in Figure 9-17, under the Hardware menu, select Show Apple TV Remote. Click the onscreen remote control, and verify that the `fetchNetworkData()` method is called, via an update in the displayed data or a breakpoint.



Figure 9-17. Using the onscreen remote in the Apple TV simulator

Implementing a touchpad click in tvOS takes advantage of another concept from iOS, press events. However, these are implemented in a different fashion from gesture recognizers. Subclasses of `UIViewController` have default handler methods for `UITouch` and `UIPress` events, which are triggered by any touch or press event. To add touchpad click support, override the default delegate method for when a press is completed, `pressesEnded(presses:with Event:)`, and insert a call to refresh the network data only when the `Select` event has been recognized, as shown in Listing 9-18.

Listing 9-18. Adding Touchpad Click Support

```

class ViewController: UIViewController {
    ...
    override func pressesEnded(_ presses: Set<UIPress>, with
        event: UIPressesEvent?) {
        for pressEvent in presses {
            if pressEvent.type == .select {
                fetchNetworkData()
            }
        }
    }
}

```

To test the touchpad click event, open the Apple TV simulator again. Press down on the Option key on your keyboard to enable touchpad support in the simulator, and click inside the touchpad again. Once again, the call to `fetchNetworkData()` should be triggered.

Debugging the App on an Apple TV

Having completely developed the `IOTHomeTV` app and verified that it works through the Apple TV simulator, there is just one step left to round out your journey into tvOS development: running the app on Apple TV hardware. In the same vein as iOS devices, starting with Xcode 9, if you have a valid Apple Developer account, you can connect to an Apple TV and debug it wirelessly from your Mac, via your home or office network connection.

To begin, verify that your Mac and Apple TV are connected to the same wireless network. Next, open the Settings app on your Apple TV. As shown in Figure 9-18, select Remotes and Devices, and then Remote App and Devices. If you pair your Apple TV with an iPhone or iPad via the Remote app, it should show up on this screen; otherwise, it should be empty.

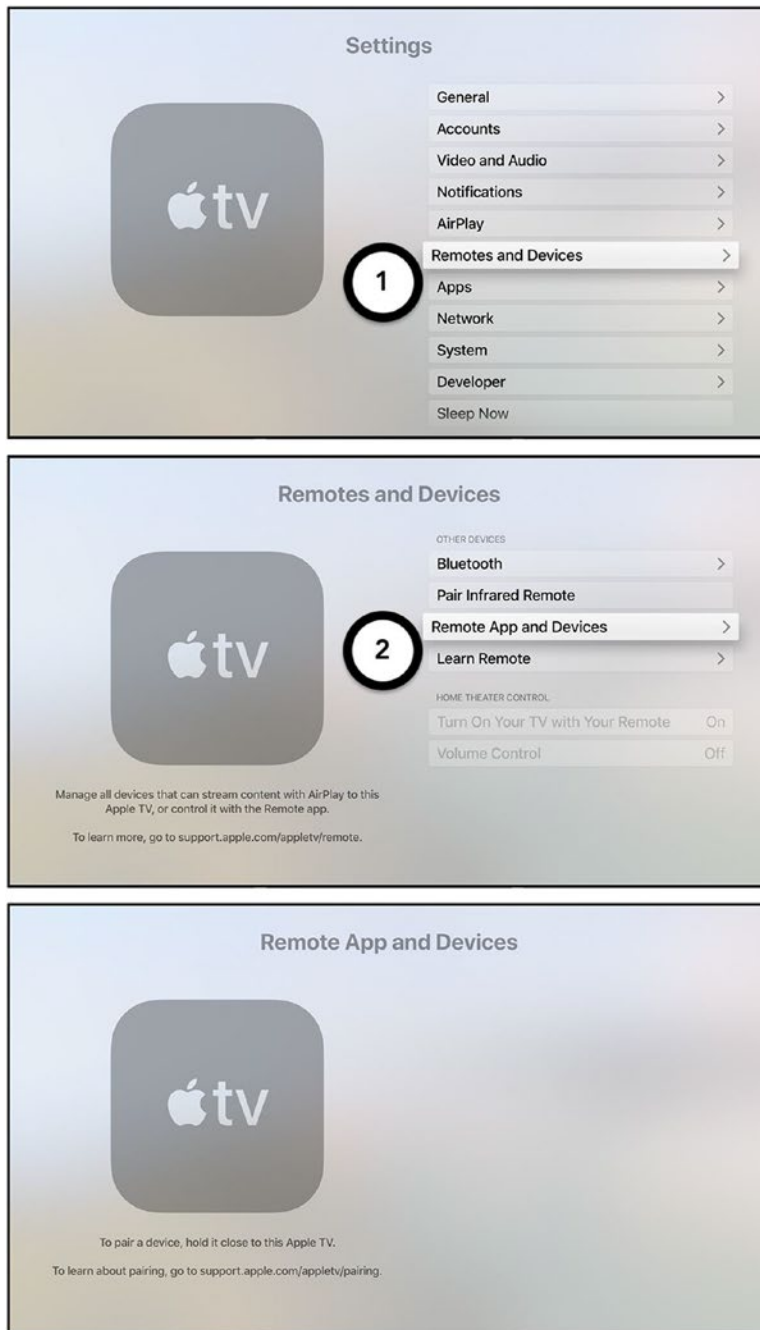


Figure 9-18. Preparing an Apple TV for pairing with a Mac

Leave the Remote App and Devices screen open on your Apple TV and open up Xcode. Within Xcode, click the Window menu and then Devices and Simulators. Inside the Devices and Simulators window, you should see a graphic of an Apple TV and a Pair button that includes the name of your Apple TV in its title text. As shown in Figure 9-19, after clicking this button, you will be presented with an authentication dialog that asks you to type in the identification code displayed on your Apple TV. Enter this code and then press Connect.

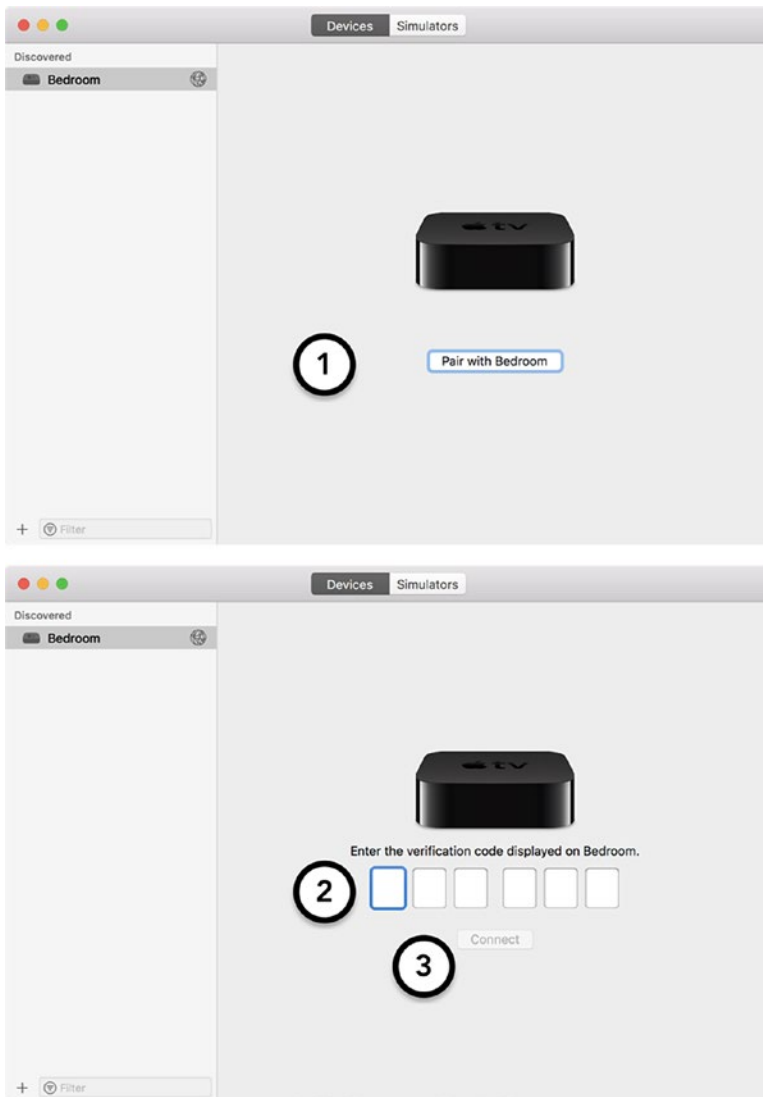


Figure 9-19. Pairing an Apple TV from a Mac

After the pairing process has been completed successfully, the Devices and Simulators window will update to show statistics about your Apple TV, as shown in Figure 9-20.

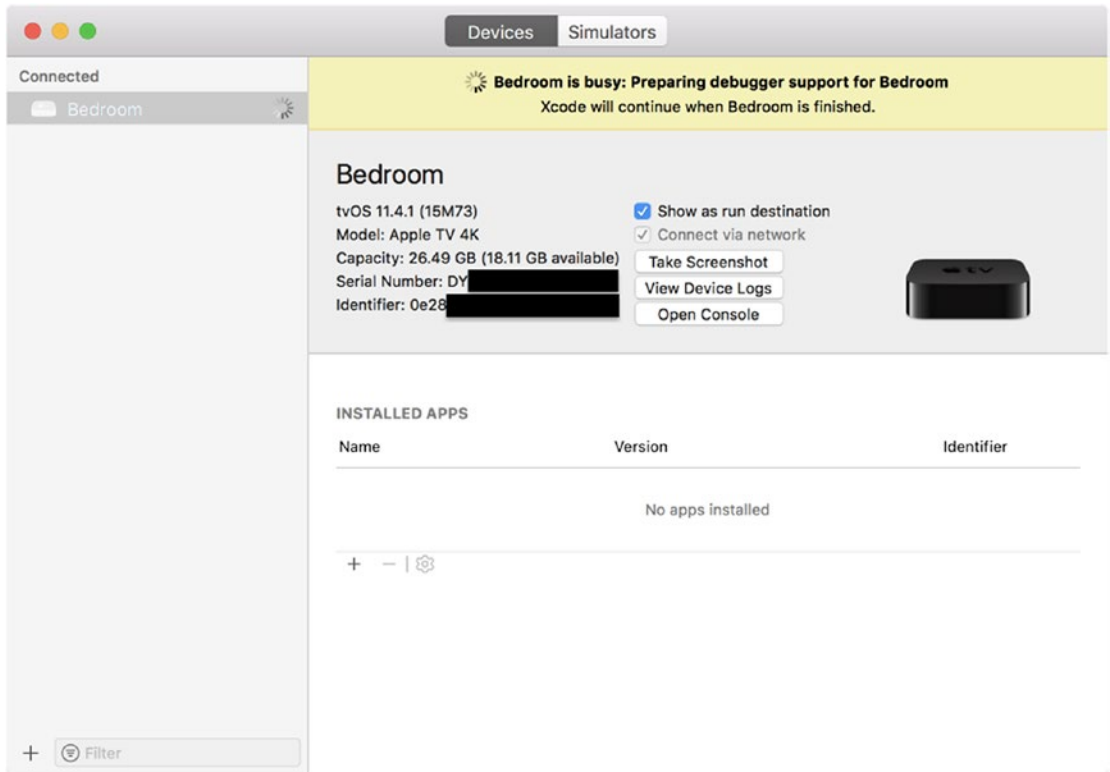


Figure 9-20. Confirmation of successful pairing with an Apple TV

After waiting several minutes for Xcode to download the debugging symbols for your device, you will be able to select your Apple TV as a Debugging Target from the target selection menu next to the Run button in Xcode, as shown in Figure 9-21. The first time you connect a device, you will also be prompted to add it to your Apple Developer Program account. Select the confirm action in this pop-up, to complete the device setup process.

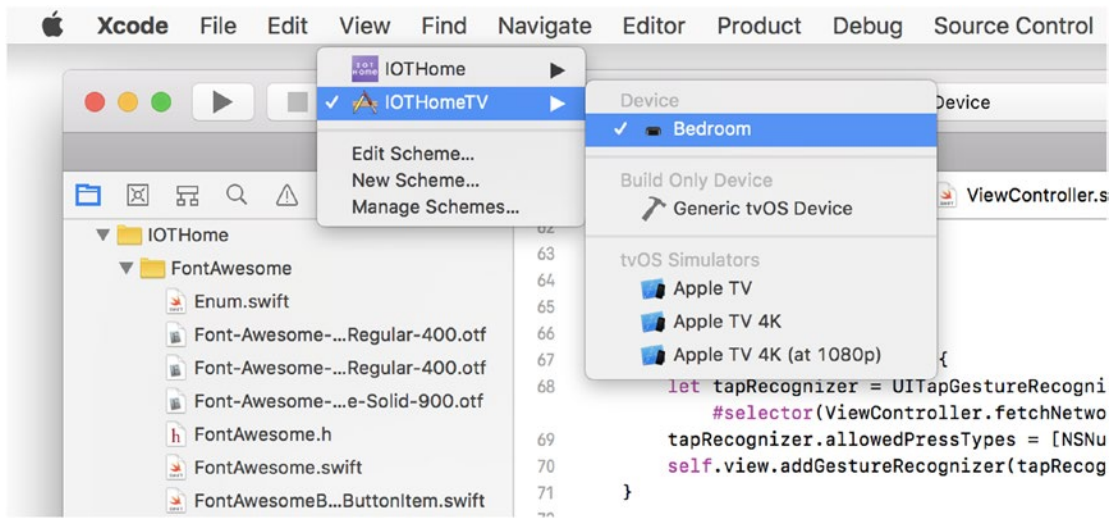


Figure 9-21. *Selecting an Apple TV as a debugging target*

At this point, you can press the Run button to run the app on the device, and you should be able to access debugging features you are used to on iOS devices, including breakpoints and stack inspection. Congratulations on building and running your first Apple TV app!

Summary

In this chapter, you learned how to leverage your knowledge of iOS programming to create a tvOS dashboard app for the IOTHome hardware. Although you had to make some special exceptions for tvOS, the development flow and APIs that were available for use closely resembled their counterparts in iOS. By adding the tvOS app as a target to the IOTHome project, you were able even to share code with the iOS app, including the Network Manager. To add that last bit of useful polish to the app, you learned how to apply visual effects to views, use Font Awesome for font-based icons, and OpenWeatherMap.org for public weather data.

As fun as it was building the app, it also gave you an opportunity to make the data from the IOTHome sensors more useful and easier to access for users. The biggest recurring challenge I kept seeing when consulting for companies implementing IoT sensors and big data was how to use all the data that has been generated. By giving your users a visualization of their data and augmenting it with other useful, related information data, you were able to reduce the friction of using the hardware and increase its value to your users. By making systems that improve a user’s life, you can help transform IoT devices from toys to tools.

CHAPTER 10

Using watchOS to Build an Apple Watch App

From Dick Tracy's watch phone in the 1940s to the Pip-Boy in today's extremely popular *Fallout* video game series, the idea of a watch as an information device on your wrist has been in the public consciousness long before Apple introduced the Apple Watch in 2015. Starting in the 1980s, such consumer electronics companies as Casio and Radio Shack began taking the leap forward and introduced the first smart watches, by adding calculators, FM radios, and contact books to digital watches. This trend continued through the 1990s and 2000s, with smart watches beginning to gain more and more connected features, such as cellular phone calling and PC data synchronization. However, as someone whose first part-time job was fixing watches in the early 2000s, I can tell you, the only smart watch I ever saw during that time was the single camera watch we had on display and never sold for several years. At that time, the only smart watches that were being sold were not compelling purchases, stemming from their high price tags, gimmicky features, and run-of-the-mill design.

This all began to change in 2012, when Pebble introduced their Pebble smart watch. Rather than being a calculator glued to the bottom of a regular digital watch, it was a Bluetooth-enabled timepiece that used an E Ink screen to display the time and notifications from your phone. The cherry on the cake was that you could use Pebble's software development kit to make your own watch faces and full-fledged applications. It quickly became the most funded project on the Kickstarter crowdfunding web site and helped push Google, Apple, and Samsung (among countless others) to get serious about connected watches.

In 2015, when Apple finally released the Apple Watch, with its own watchOS software development kit and App Store, many predicted it would be Apple's next huge market and instantly dominate the smart watch market, just as the iPhone revolutionized

smartphones in 2007. As of this writing, in 2018, the Apple Watch did fulfill the prophecy of becoming the number-one smart watch; however, it was a long, slow process of attrition. Although feature-wise, they were mostly the same. The Apple Watch provided a higher build quality, full-color screen, and health sensors that the Pebble did not have. Compared to Android Wear, the Apple Watch had a more passionate user base and hardware manufacturers that provided the supply and demand to keep improving the platform.

Most important of all, the Apple Watch was able to become the number-one smart watch by expanding into mainstream consciousness and addressing markets that were previously underserved. Although the platform did not expand to the popularity of smartphones, it has proved to be extremely popular in health care, the service sector, and fitness communities, as it provides nonintrusive access to snippets of information in environments in which smartphones are inappropriate or impractical. I know I personally am lost without my Apple Watch when I get on crowded trains in Tokyo. In these respects, the Apple Watch is a true Internet of Things (IoT) success story.

In this chapter, you will learn how to take advantage of the Apple Watch, to build apps that can run in tandem with your iOS apps. As with the tvOS project in Chapter 9, you will take the IOTFit fitness app from Chapters 1-4 and add a watchOS target that allows users to view their past workout history and create new workouts. You will port over the motion (accelerator) and location-tracking features from the iOS app, to generate all of the same data in the watchOS app. As an added bonus, you will also take advantage of the HealthKit store from the iOS project, which will automatically sync data between the user's watch and phone.

Learning Objectives

In this chapter, by building the watchOS version of the IOTFit fitness-tracking application, you will learn the following key concepts for IoT application development:

- Adding a watchOS target to an iOS project
- Building a table-based user interface on a watchOS app
- Adding Force Touch support to a watchOS app
- Using Core Motion, Core Location, and HealthKit on watchOS
- Populating a table view in watchOS

Similar to tvOS, watchOS provides a stripped-down development environment based on the development principles and frameworks from iOS. Starting with watchOS 2.0, Apple began exposing more of these frameworks to developers, so that Apple Watch apps could run without being tethered to an iPhone or Internet connection. Although this started as a way to improve performance on the device, it was one of the key factors that helped solidify the platform's adoption in the markets described previously. The IOTFit watchOS app will be able to run without being paired with an iPhone, except for when the user must enable permissions the first time the app is loaded. As mentioned in the introduction, all of the data in the app will be stored in HealthKit, which Apple automatically syncs between the user's watch and his/her iPhone.

Unlike tvOS, due to the limited memory and processing power of watchOS, many of these frameworks are stripped down to the core features required for smart watch use cases. For most frameworks, this is manifested by seeing a smaller subset of APIs available. However, in the area of user interface development, the toll is quite significant. Rather than being able to use the full set of UIView subclasses and Auto Layout, you will have to learn how to use stack views and the more limited WKInterface* classes. In this chapter, I will focus on both, with careful attention to making stack views work for practical user interfaces and implementing a WKInterfaceTable-based table view whose implementation concepts differ considerably from those of the UITableView class on iOS.

This chapter is based heavily on the foundation of the IOTFit fitness app from Chapters 1-4. If you are still uncomfortable with user interface development in Xcode, I highly recommend reviewing Chapter 1. If you would like to review the details of implementing location services and health APIs in iOS, I highly recommend reviewing Chapters 2-4.

Setting Up the Project

In the same manner as the tvOS project from Chapter 9, in this chapter, you will implement the watchOS version of the IOTFit project by adding a new target to the last iteration of the app from Chapter 4. Begin by copying your completed project from that chapter or downloading a fresh copy from the official GitHub repository for this book at <https://github.com/Apress/program-internet-of-things-w-swift-for-ios>.

When a watchOS target is added to an iOS project, not only will you be able to share code with the iOS app, but the watchOS target will be tagged as an Apple Watch version on the App Store. As shown in Figure 10-1, when a watchOS version is available

for an iOS app in the App Store, users are provided with a link on the App Store description page that shows them information on the watchOS version and the option to automatically install it, along with the iOS version.

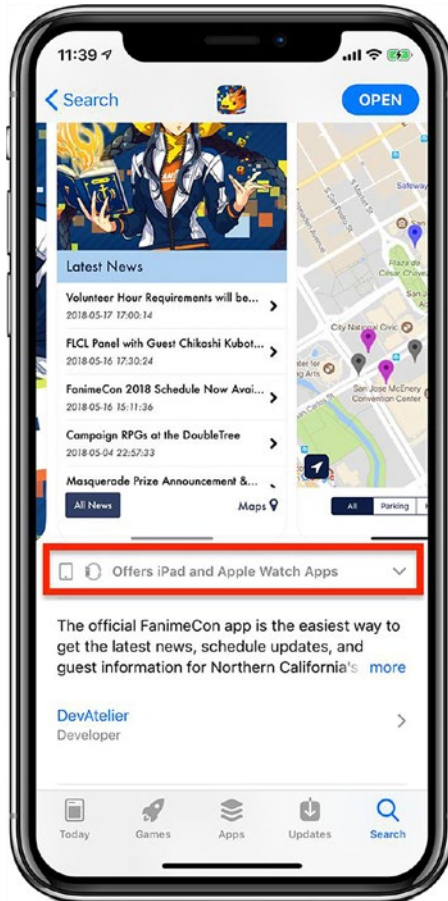


Figure 10-1. App Store description page for an iOS app that offers a watchOS version

After you have prepared a copy of the original project, go to the File menu and select New ► Target, just as you did in Chapter 9 for the tvOS target. From the template picker pop-up window, select WatchKit App, as shown in Figure 10-2.

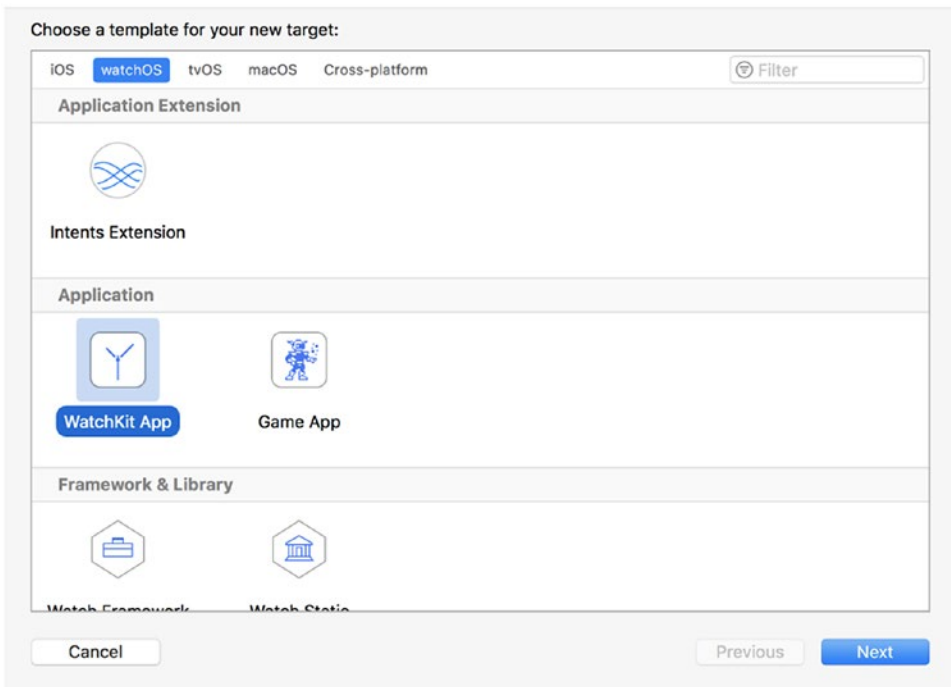


Figure 10-2. *Selecting a WatchKit app template*

When asked to configure the target, name it IOTFitWatch. As shown in Figure 10-3, confirm that the product is set to build against the credentials for your developer account and that the Include Notification Scene and Include Complication check boxes are not selected. For the scope of this project, you do not have to enable either option.

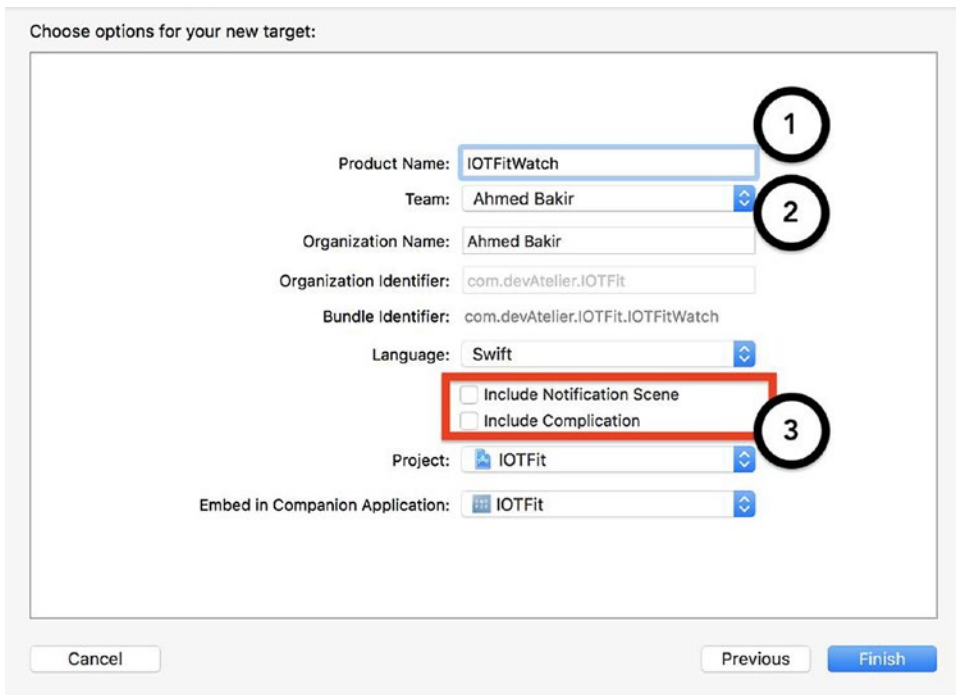


Figure 10-3. *Configuring the watchOS target*

The first time you add a watchOS target to a project, you will be asked to activate its target, as shown in Figure 10-4. Select Activate, to dismiss the pop-up, and continue with the setup process.

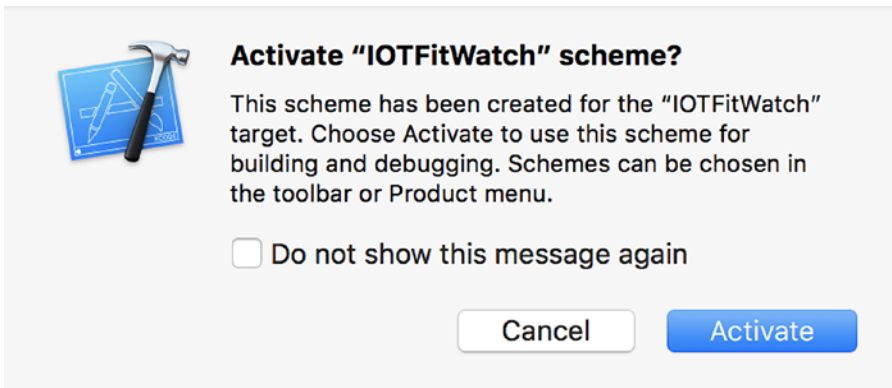


Figure 10-4. Activating the scheme for the watchOS app

After you have finished setting up the target, your project should now include two new folders, IOTFitWatch and IOTFitWatch Extension, as indicated in Figure 10-5.

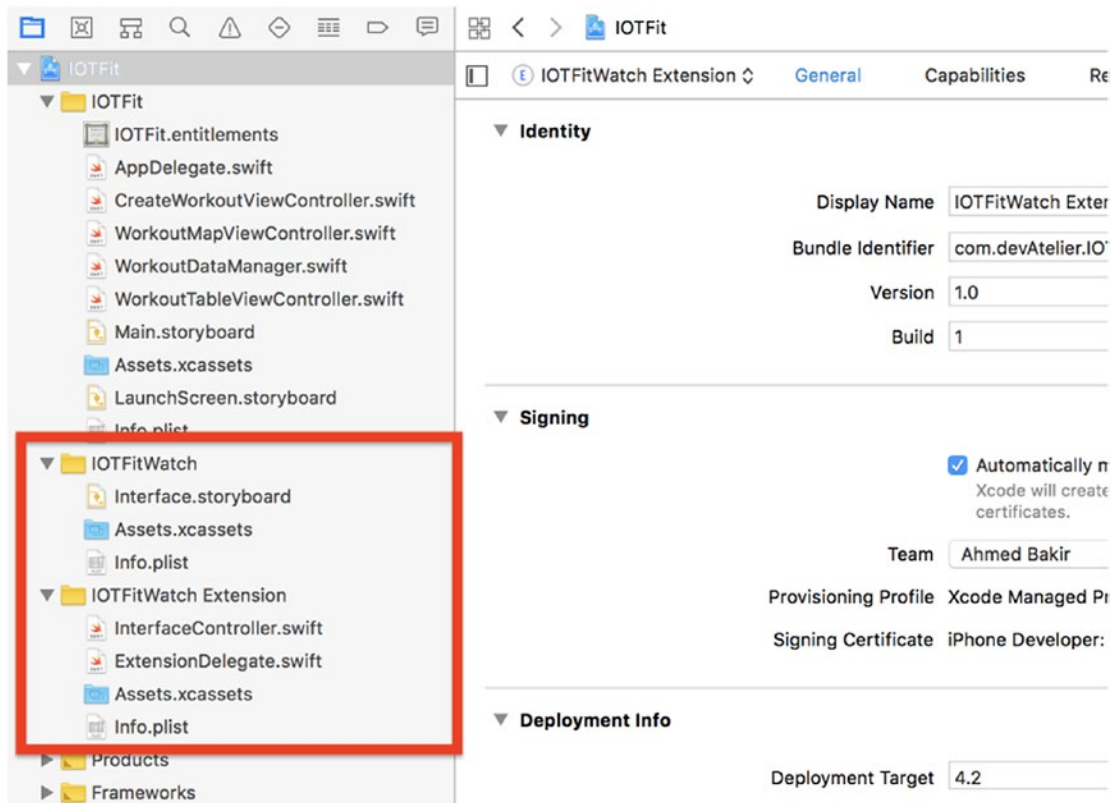


Figure 10-5. IOTFit project after including the watchOS target

Since watchOS 1.0, Apple has split watchOS apps into two parts: an app containing the static resources only (storyboards, assets) and an extension containing the code. In watchOS 1.0, Apple did not expose the watchOS frameworks on the watch hardware, and the extension was used by the iOS app to update the views that were specified by the WatchKit app component. Today, both pieces run on the watch hardware, but this logic separation pattern is still preserved by Apple.

Next, you must set the descriptions for the permission-restricted features the watchOS app will have to use: user location, HealthKit, and Core Motion. The easiest way to do this is to copy the Privacy description key-value pairs from the iOS project's Info.plist file into the Info.plist file in the IOTFitWatch Extension folder. The key-value pairs you need to copy are provided as source code in Listing 10-1. You can use the Open As ► Source Code secondary-click (right-click) option for the Info.plist file, to copy them into the file as text, or you can look for the Privacy... key-value pairs in the Property List Inspector and add them manually.

Listing 10-1. Privacy Permission Description Key-Value Pairs for the IOTFitWatch Extension Info.plist File

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/
DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>WKCompanionAppBundleIdentifier</key>
    <string>com.devAtelier.IOTFit</string>
    ...
    <key>NSHealthShareUsageDescription</key>
    <string>IOTFit would like to use HealthKit permission to
import workout data </string>
    <key>NSHealthUpdateUsageDescription</key>
    <string>IOTFit would like to use HealthKit permission to
export workout data to the Health app. </string>
    <key>NSLocationAlwaysAndWhenInUseUsageDescription</key>
    <string>IOTFit would like to use location permission to plot your
location during workouts. </string>
    <key>NSLocationAlwaysUsageDescription</key>
```

```

<string>IOTFit would like to use location permission to
plot your location during workouts. </string>
<key>NSLocationUsageDescription</key>
<string>IOTFit would like to use location permission to
plot your location during workouts. </string>
<key>NSLocationWhenInUseUsageDescription</key>
<string>IOTFit would like to use location permission to
plot your location during workouts.</string>
<key>NSMotionUsageDescription</key>
<string>IOTFit would like to use motion permission to
help you measure the step count and altitude of your
workouts. </string>
</dict>
</plist>

```

For the final step in the setup process, you must enable background modes for the WatchKit App, in order to track the user's location. As shown in Figure 10-6, select the IOTFit project file at the top of the Project Navigator, then navigate to the IOTFitWatch Extension target and click the Capabilities tab. In the Capabilities detail screen, enable Background Modes and Workout Processing.

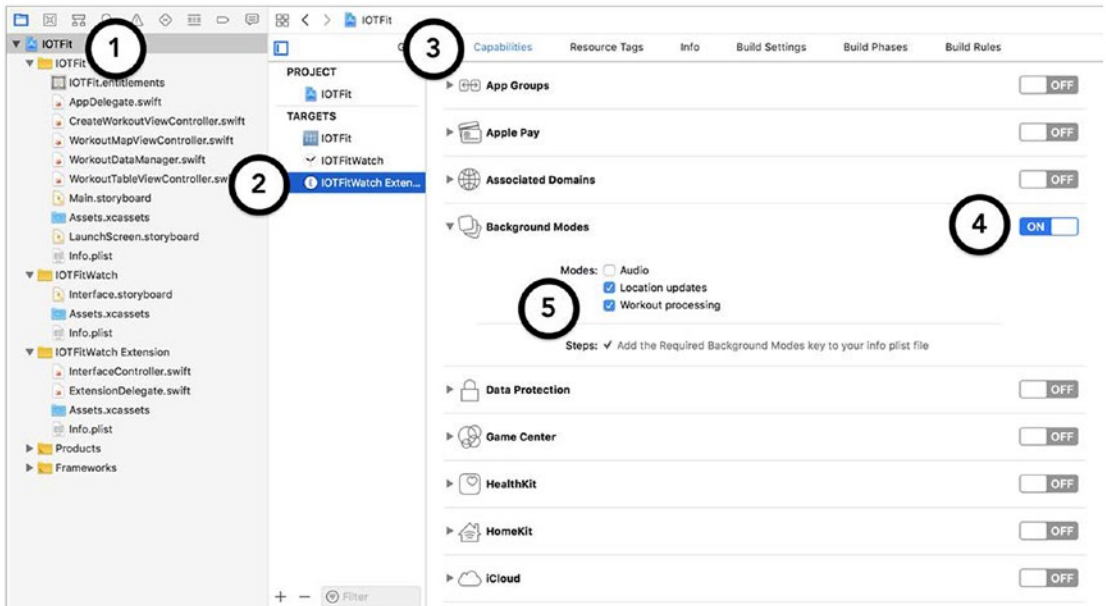


Figure 10-6. Enabling Background Modes for the IOTFitWatch app

Building a watchOS User Interface

Now that the watchOS target has been configured completely, you can begin putting together the user interface. For this project, I have adapted the IOTFit app to an Apple Watch form factor, as shown in Figure 10-7. Because most users interact with their Apple Watches when they are on the go or unable to access their phones, you should design your interface so that it is easy to view the most commonly used information from the app and equally easy to create a new record in the app.

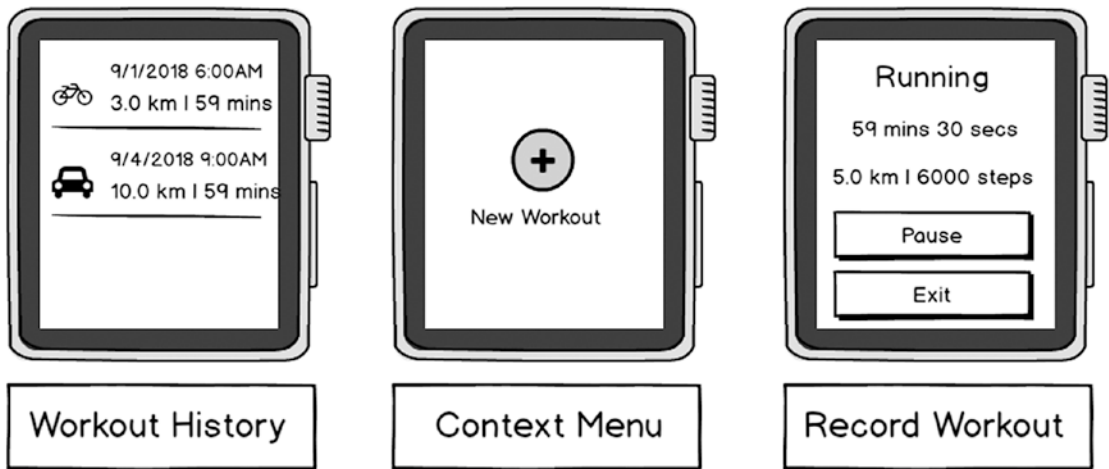


Figure 10-7. Design wireframes for the IOTFitWatch app

In the IOTFitWatch app, I expose the user’s workout history on the Home screen and provide a context menu that allows him/her to launch a Detail screen for recording a new workout. Users can bring up the context menu by performing a force touch (hard press) on the Workout Screen. The Workout History and Record Workout screens display the user’s activity type to help him/her identify workouts.

To get started developing the user interface, open the `Interface.storyboard` file under the `IOTFitWatch` folder. As shown in Figure 10-8, you will be greeted with a storyboard containing the blank interface for the first view controller in the app (`InterfaceController.swift`).

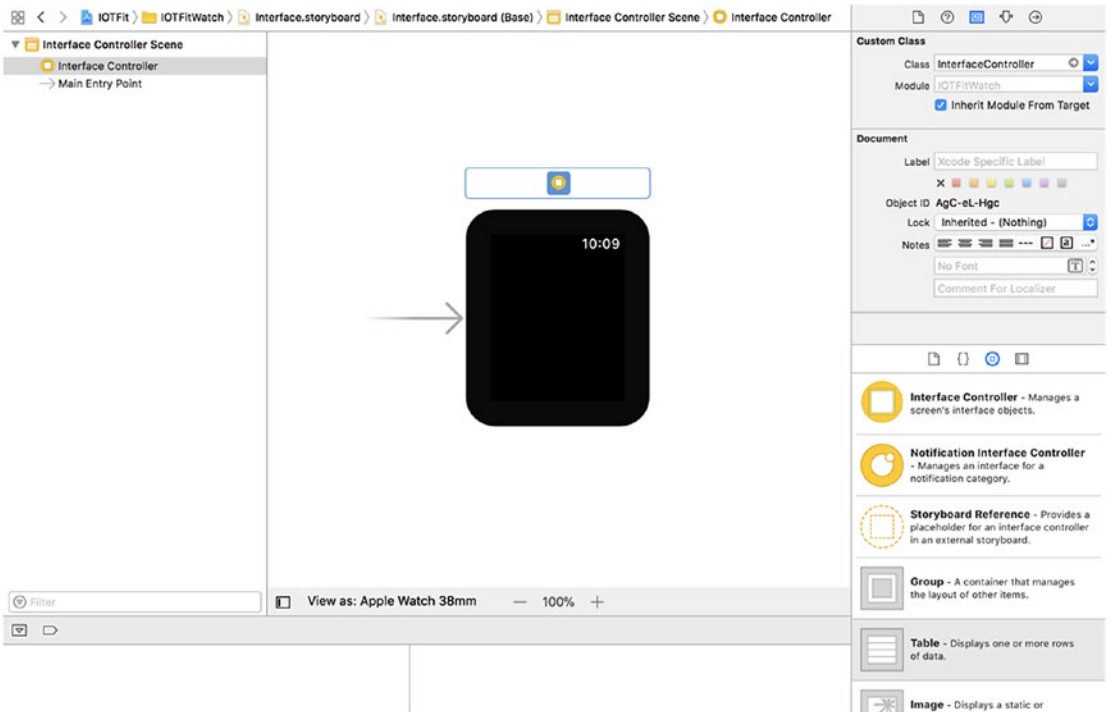


Figure 10-8. Initial storyboard for the IOTFitWatch project (*Interface.storyboard*)

Begin by laying out the Workout History screen. In the same manner as an iOS or tvOS app, drag a Table object from the Object Library to the view controller, as shown in Figure 10-9.

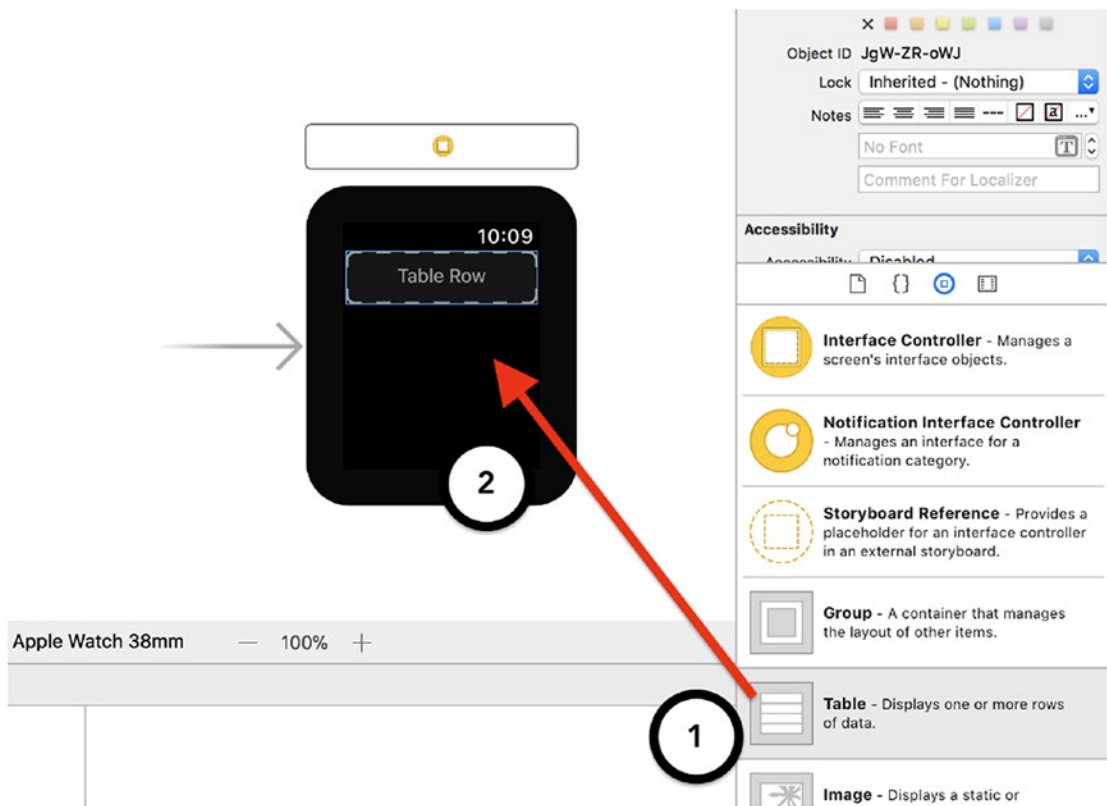


Figure 10-9. Adding a table to the Record Workout view controller

If you have upgraded to Xcode 10, your user interface may hide the Object Library by default. To open the Object Library on Xcode 10, click the Library button (the one with an iPhone Home button icon) at the top-right of Xcode, next to the options for toggling the side bars in Interface Builder, as shown in Figure 10-10.

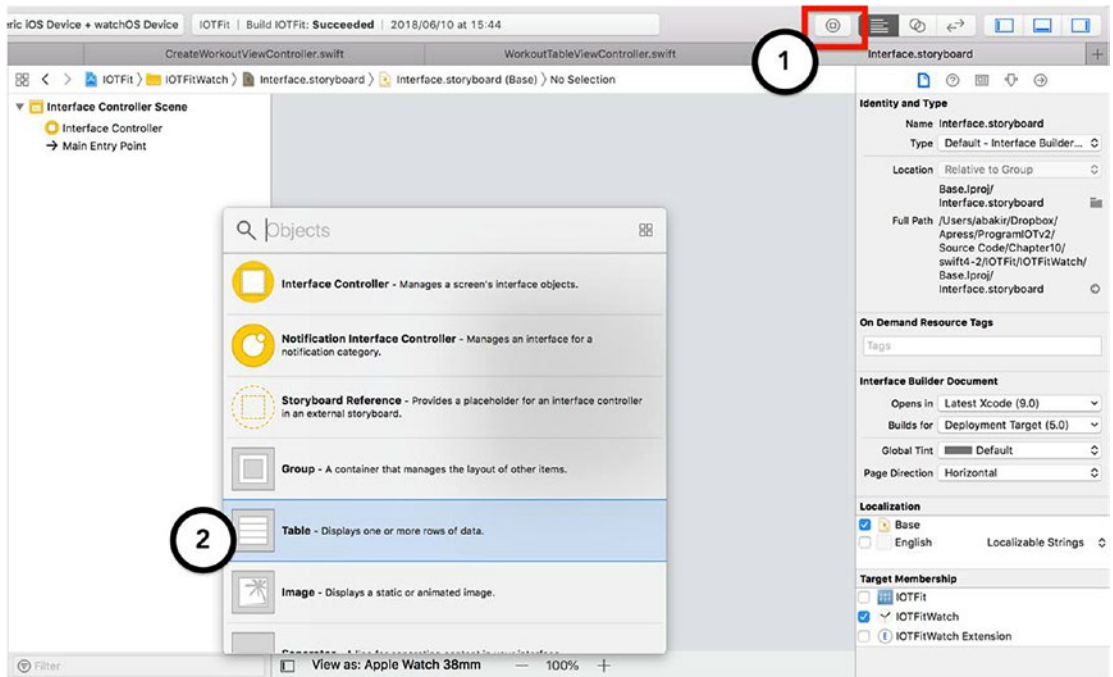


Figure 10-10. Finding the Object Library in Xcode 10

Next, you must configure the height of the table view cell (known as a *table row* in watchOS). As shown in Figure 10-11, click the table row, then in the Attributes inspector (the right detail pane of Interface Builder), click the Height drop-down, and select Fixed. This will allow you to make all of the rows a consistent height. For my implementation, I chose 50 pixels for the height.



Figure 10-11. *Configuring cell height for a table row*

As mentioned at the beginning of the chapter, one of the unfortunate side effects of watchOS’s small subset of iOS is that you must use stack views to lay out your user interfaces. Stack views manage a container view and layout each item vertically or horizontally in the container, according to a few configuration parameters (e.g., equal heights, equal spacing). In order to implement the user interface from Figure 10-7, in which there is an image to the left of two vertically-aligned labels, you will have to nest stack views. The parent stack view will be configured to stack items horizontally and contain the image view, and stack view for the labels. The stack view for the labels will be configured to stack items vertically.

To begin implementing the stack views, drag a Group from the Object Library onto the table row, as shown in Figure 10-12. The groups will be wider than the width of the Apple Watch at first. To fix this, click the right edge of each group and resize it so that the group for the image view is smaller than the group for the labels.

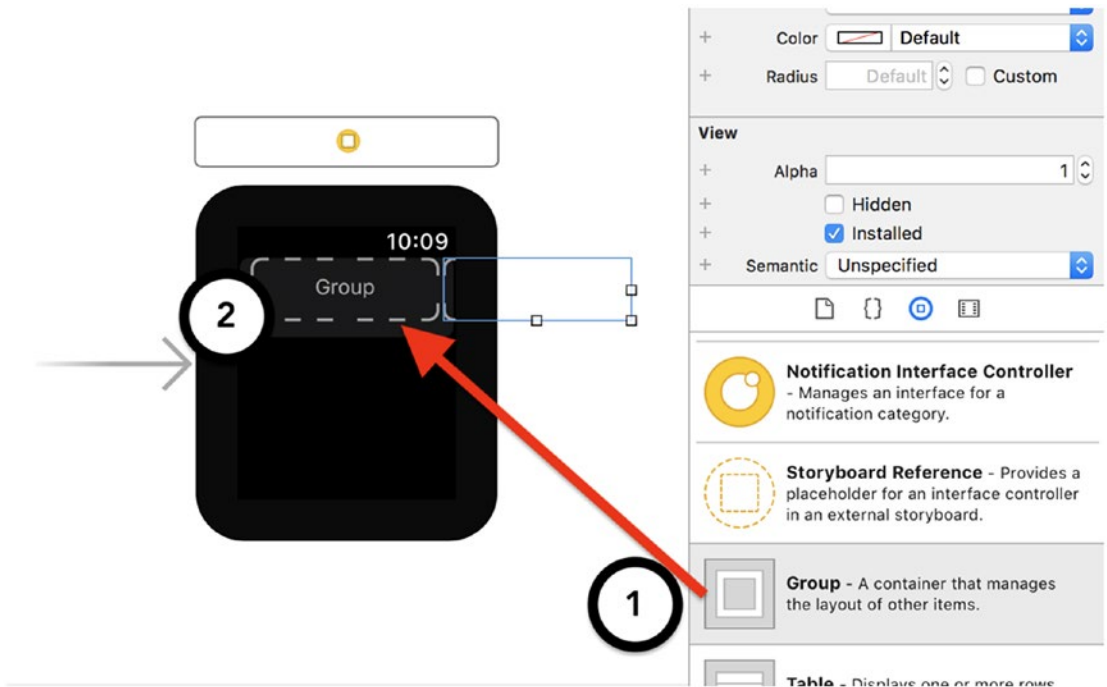


Figure 10-12. Adding an additional group to a table row

Next, drag Image and Label items from the Object Library onto the view controller. Your table row should look like my implementation in Figure 10-13. Although the image view is displayed correctly, the labels require adjustment.

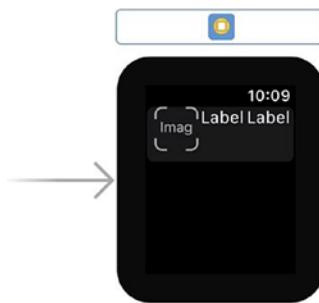


Figure 10-13. Table row with default stack view configuration

To fix the layout of the labels, you must configure the stack view to use vertical layout. As shown in Figure 10-14, select the group, then, inside the Layout drop-down menu, select Vertical.

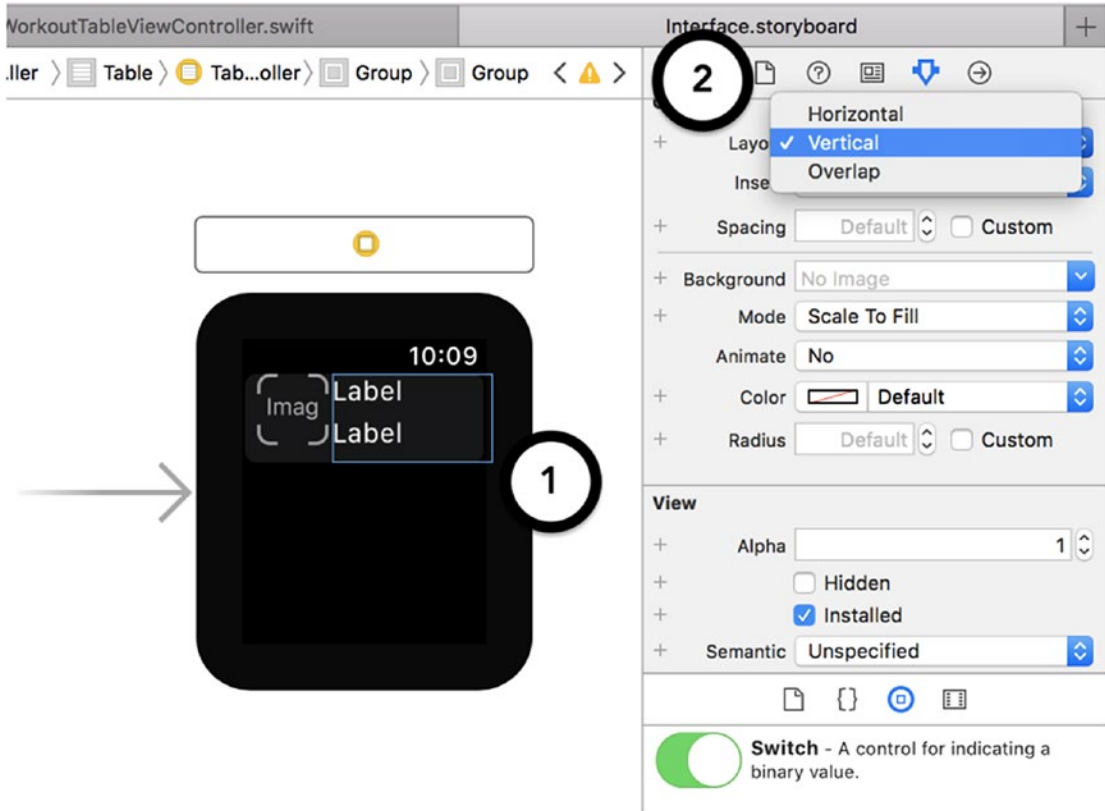


Figure 10-14. *Configuring vertical layout for a stack view*

To configure the font size, label text, or other options for a view, select it and use the drop-downs in the Attributes inspector to make the adjustments, as shown in Figure 10-15. After modifying the font size for the top label to Subhead, your table row should now match the design specified in the original design.

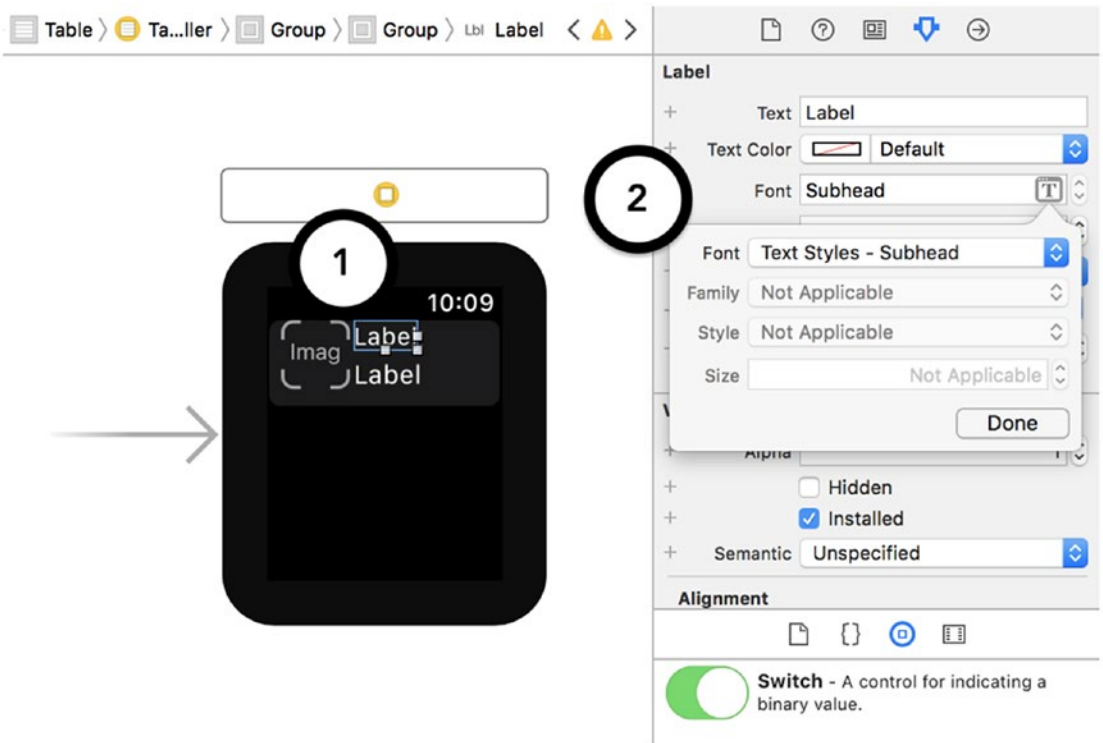


Figure 10-15. *Configuring text size and color for a stack view*

Having completed the Workout History view controller’s layout, you can lay out the Workout Detail view controller. Using the object library, drag an Interface Controller onto the storyboard, configure its single group to use a vertical layout, and then drag Label and Button items onto the group. Before configuring the views, your layout should look similar to my result in Figure 10-16. You can use the view hierarchy, highlighted on the left, to help you determine if you assigned sub-views to a stack view or normal UIView correctly.

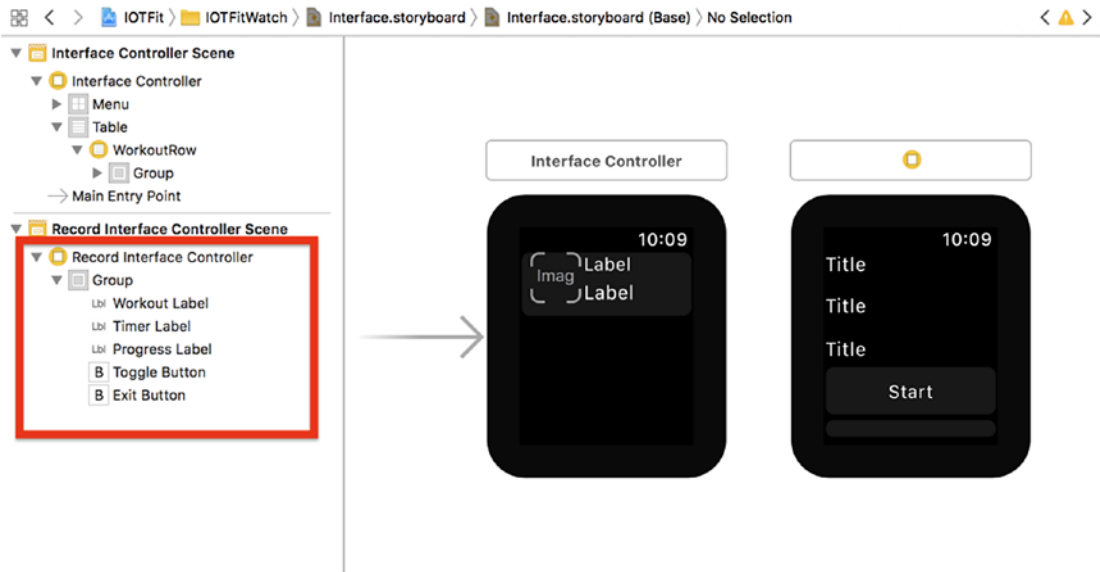


Figure 10-16. Record Workout view controller with initial sub-view arrangement

After configuring text sizes, alignment, and colors, your final storyboard should look similar to my implementation in Figure 10-17. I set all of the views to use the Body or Headline text styles and text centering. I set the color property of the buttons to Red and Blue, to make the buttons easier to differentiate.

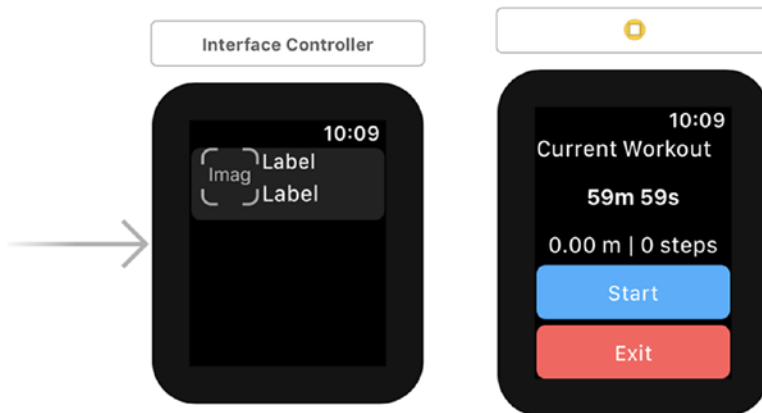


Figure 10-17. Final storyboard arrangement for the IOTFitWatch project

Just as you were required to do for your iOS apps, to finish the user interface setup for a watchOS project, you must connect the storyboard to your interface controller (view controller) classes. As its layout is more straightforward, begin with the Record Workout interface controller class. Click the `IOTFitWatch Extension` folder in the Project Navigator, then select `New File` from the secondary-click (right-click) context menu. As shown in Figure 10-18, navigate to watchOS from the template picker pop-up window, then select `WatchKit Class`. This is the equivalent of choosing the Cocoa Touch class template for an iOS project.

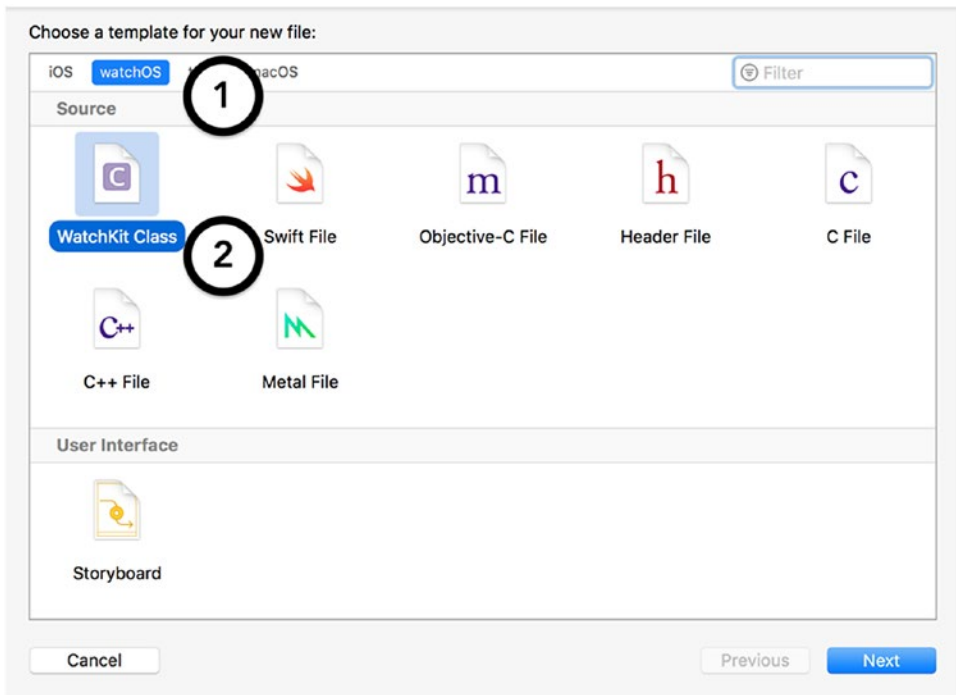


Figure 10-18. *Creating a WatchKit class from the File Template window*

When asked to choose your parent class, type in “`WKInterfaceController`.” Enter `RecordInterfaceController` as the class name. Use my sample in Listing 10-2 for the contents of the class. Each user interface item is represented by an `@IBOutlet` property. Unlike an iOS app, all of the elements are `WKInterface` classes.

Listing 10-2. Record Interface Controller Class Definition
(RecordInterfaceController.swift)

```
class RecordInterfaceController: WKInterfaceController {  
  
    @IBOutlet var timerLabel: WKInterfaceLabel?  
    @IBOutlet var workoutLabel: WKInterfaceLabel?  
    @IBOutlet var progressLabel: WKInterfaceLabel?  
    @IBOutlet var toggleButton: WKInterfaceButton?  
    @IBOutlet var exitButton: WKInterfaceButton?  
  
    override func willActivate() {  
        super.willActivate()  
    }  
  
    override func didDeactivate() {  
        super.didDeactivate()  
    }  
}
```

In the same manner as an iOS application, to connect the storyboard to the RecordInterfaceController class, you have to set it as the parent class. As shown in Figure 10-19, select the Interface Controller on the storyboard, then navigate to the Identity Inspector (the tab with the ID card icon), and enter RecordInterfaceController in the Class text field.

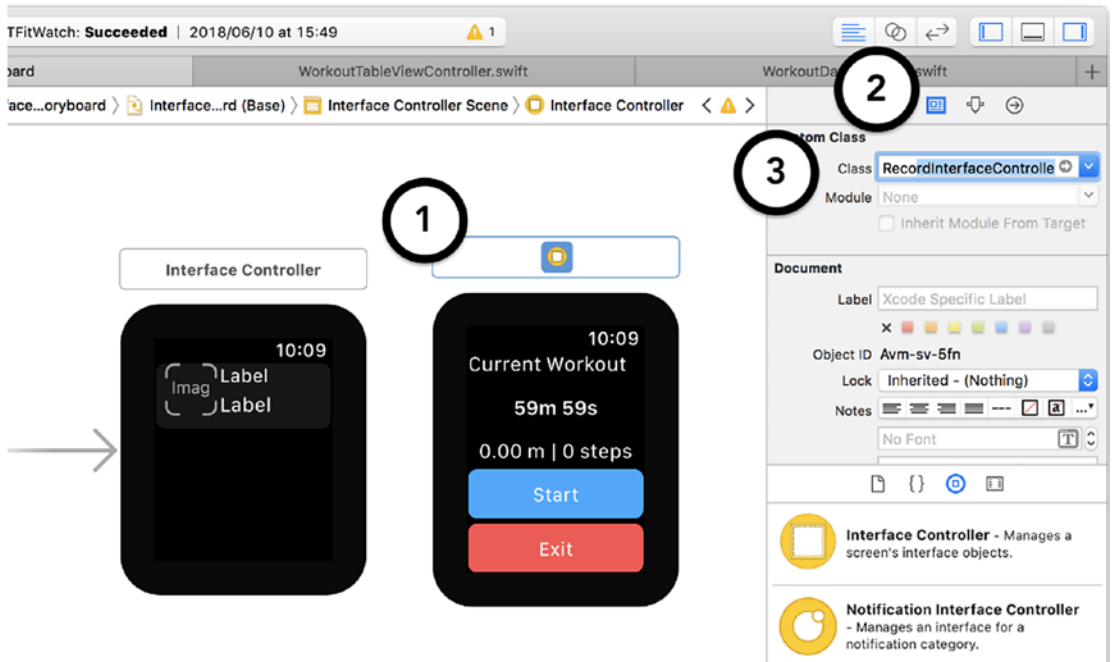


Figure 10-19. Setting ownership for a watchOS Interface Controller

Just as with an iOS storyboard, drag and drop connections from the Attributes inspector onto elements in the Interface Controller, to complete the connection process, as shown in Figure 10-20.

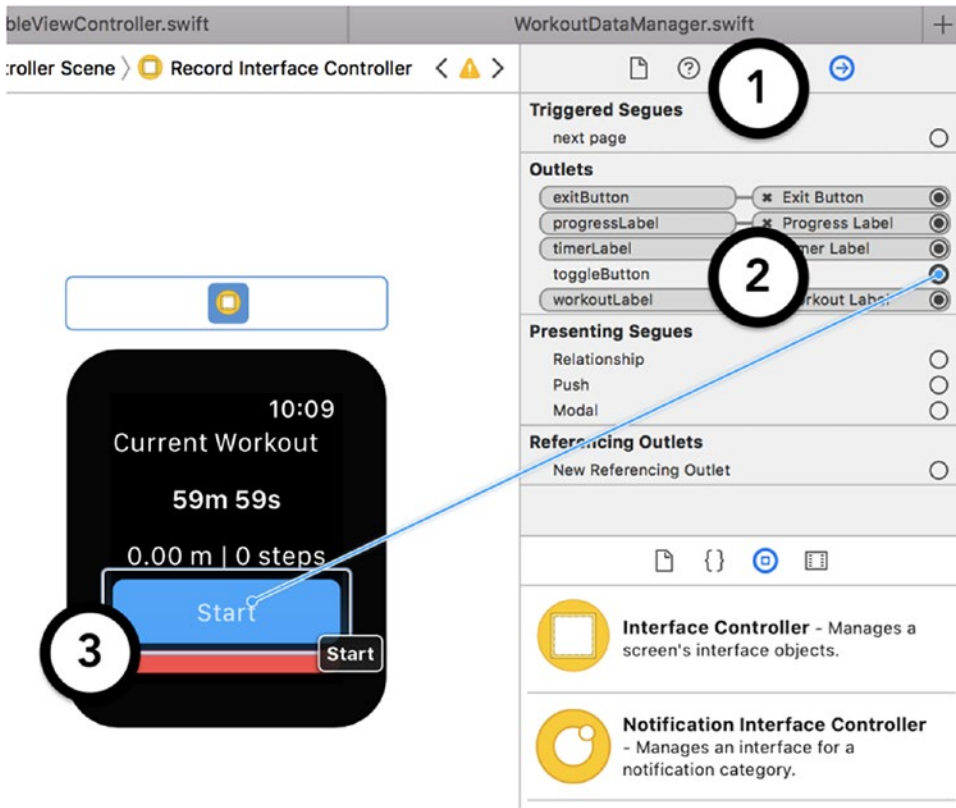


Figure 10-20. Connecting Outlets to Interface Controller elements

Setting Up a Table View Using the WKInterfaceTable Class

The setup process for the Workout History Interface Controller is more complicated, owing to the table view. To begin, modify the definition of the default InterfaceController class to add the outlet for the table view, as shown in Listing 10-3. After updating the definition, connect the outlet in Interface Builder.

Listing 10-3. Workout History Interface Controller Class Definition (InterfaceController.swift)

```
class InterfaceController: WKInterfaceController {
    @IBOutlet var workoutTable: WKInterfaceTable?
    ...
}
```


Next, you must create a class to define each table row otherwise known as a *row controller*, in watchOS terminology. Unlike a table view cell on iOS, in which you can use `UITableViewCell` as your parent class, watchOS does not define a parent class for table rows. Instead, you must use `NSObject` to define a generic object that is compatible with both Swift and Objective-C. Aside from this restriction, the rest of the setup process is straightforward, as you include the user interface elements as properties of the new class. In Listing 10-4, I have defined my table row as the `WorkoutRowController` class. The definition includes the properties for the image view and labels. Add a new file named `WorkoutRowController.swift` to the `IOTFitWatch Extension` target to contain this class.

Listing 10-4. Workout Row Controller Class Definition
(`WorkoutRowController.swift`)

```
import WatchKit

class WorkoutRowController: NSObject {
    @IBOutlet var icon: WKInterfaceImage?
    @IBOutlet var dateLabel: WKInterfaceLabel?
    @IBOutlet var durationLabel: WKInterfaceLabel?
}
```

Next, navigate back to Interface Builder and select the table row. As shown in Figure 10-21, begin by clicking the Attributes inspector (the second-to-last tab in the right pane) and setting an identifier for the row. Make a note of this, as you will use it later when you populate the data for the table.

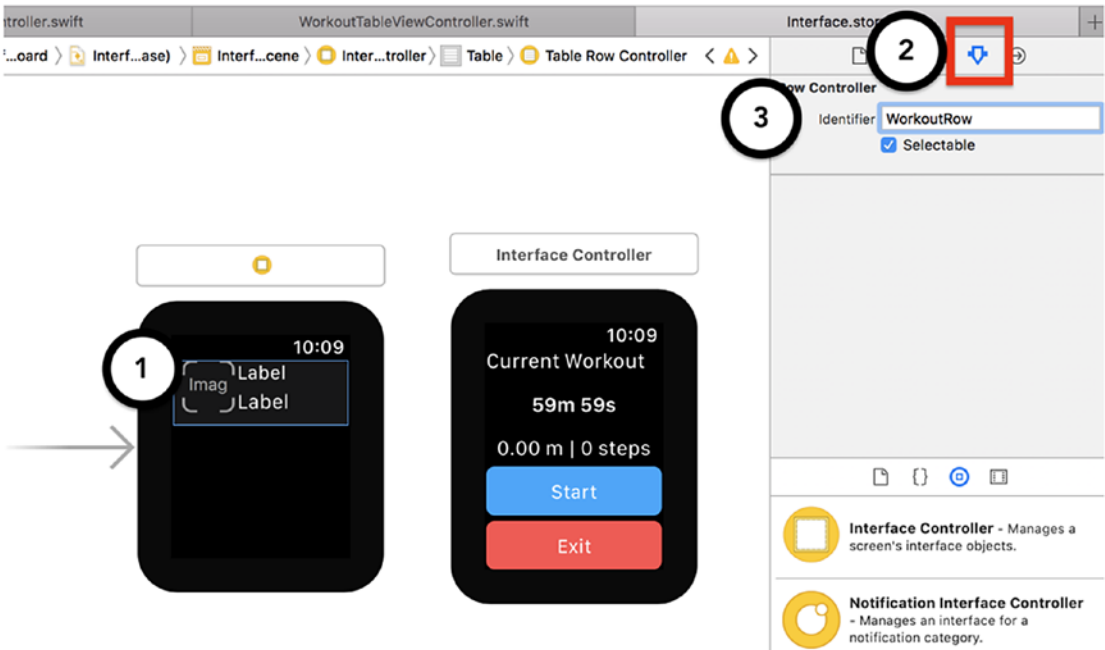


Figure 10-21. *Setting the identifier for a row controller*

After setting the identifier, click the Identity Inspector (the third tab) and set the class to `WorkoutRowController`, in the same manner you connected the `WorkoutInterfaceController` to its definition. If the operation was successful, you should be able to see the properties for the row controller in the Connections Inspector (the last tab) and connect them to the storyboard successfully, as shown in Figure 10-22.

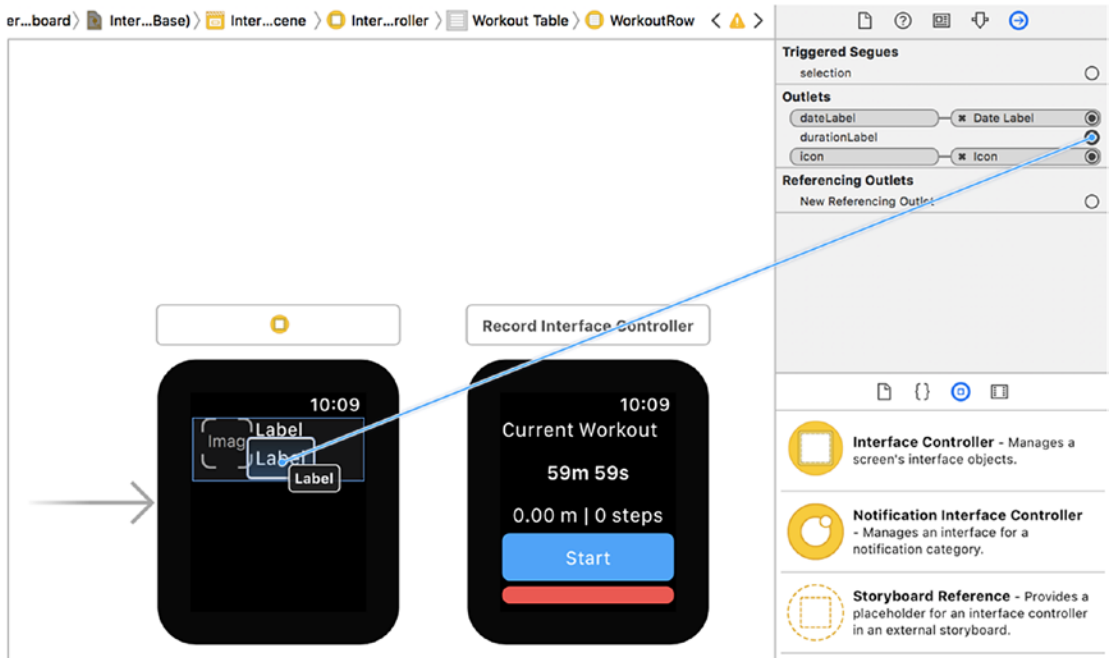


Figure 10-22. Connecting outlets to a row controller

Adding Force Touch Support

For the final piece in the user interface puzzle, you will learn how to use Force Touch to present a context menu to initiate the transition to the Record Workout Interface Controller. Although you can use segues, just as you would in an iOS app, using Force Touch provides a more native experience for the Apple Watch. *Force Touch* (or 3D touch on the iPhone) is the term Apple uses to describe the pressure sensitivity of their touch pads, starting with the iPhone 6S and eventually moving into the MacBook Pro and Apple Watch. These devices provide support for two kind of touch events: a *shallow press*, a normal touch event, which is used to select an item, and a *deep press*, a long, forceful press on the screen, which is used to bring up context menus.

You can enable Force Touch in your watchOS apps by adding a context menu to an Interface Controller. To add a context menu to an Interface Controller, drag a Menu object from the Object Library and drop it on the target Interface Controller. As shown in Figure 10-23, after adding a context menu to the Workout History Interface Controller, it will be added to its view hierarchy.

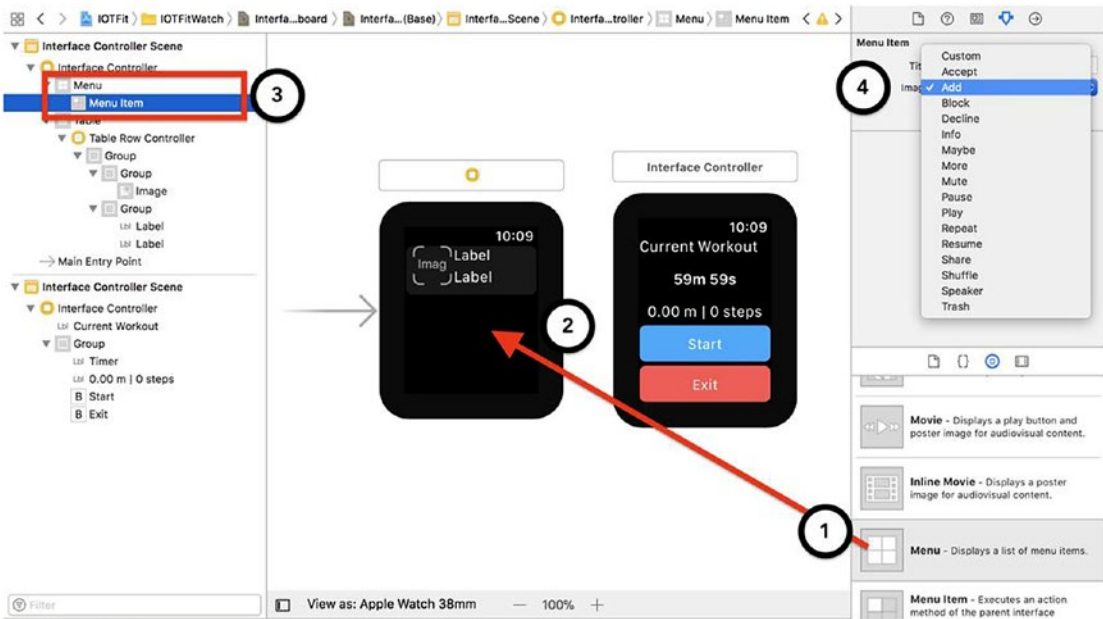


Figure 10-23. Adding and configuring a context menu

By default, a context menu will include one menu item. For the IOTFitWatch app, you only need an Add button, for presenting the Record Workout Interface Controller. To customize the menu item, select it from the view hierarchy and use the Attributes inspector to set a title and icon. For my implementation, I selected the Add icon and named the menu item, New Workout.

To enable the transition to the Record Workout Interface Controller, or perform any other action based on the menu selection, you must declare a method with the `@IBAction` keyword, to indicate that should be discoverable by Interface Builder. In Listing 10-5, I have added a method named `presentRecordInterface()` to handle the transition.

Listing 10-5. Adding a Method for Presenting the Record Workout Screen (InterfaceController.swift)

```
class InterfaceController: WKInterfaceController {
    @IBOutlet var workoutTable: WKInterfaceTable?
    ...
}
```

```

@IBAction func presentRecordInterface() {
    presentController(withName:
        "RecordInterfaceController", context: nil)
}
}

```

watchOS provides a `presentController(withName:context:)` for presenting one Interface Controller from another, similar to the `present(animated:completion:)` method on iOS. To use this method on watchOS, you must provide the Identifier string for the Interface Controller you want to target and a context object, which can be used to share data between the two Interface Controllers. In the earlier setup steps, although you set the class for the Record Workout Interface Controller, you did not have to set an identifier. To add the identifier, select the Record Workout Interface Controller and then navigate to the Attributes inspector, as shown in Figure 10-24.

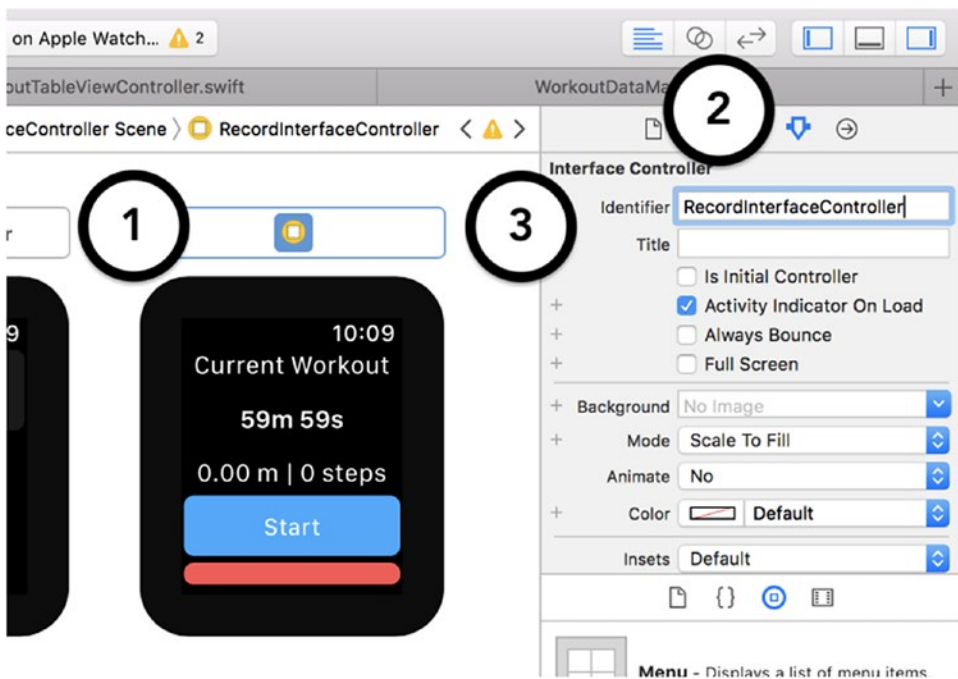


Figure 10-24. Adding an identifier string to an Interface Controller

Now that all the prerequisites for the `presentRecordInterface()` method have been met, you can connect to the storyboard. To perform this operation, select the menu item in the view hierarchy, navigate to the Connection Inspector, and drag a connection from

the selector item to the Record Workout Interface Controller. As shown in Figure 10-25, the `presentRecordInterface()` method signature should appear as a selectable option.

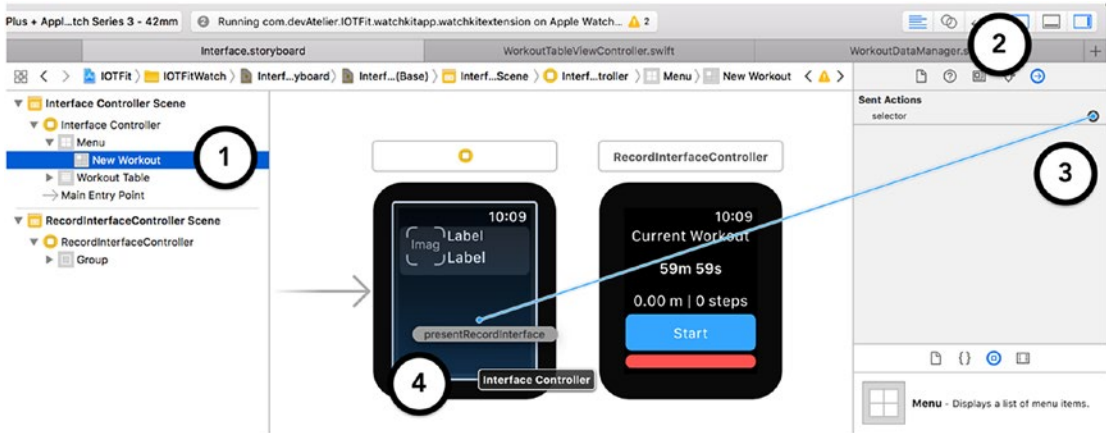


Figure 10-25. Connecting a selector to a method in an Interface Controller

To test the context menu, select one of the iOS simulators that is paired with an Apple Watch simulator from the menu next to the run button in Xcode (for example, iPhone 8 Plus + Apple Watch Series 3). Although the Workout History screen of the watch app will initially be empty, as shown in Figure 10-26, you can perform a deep press on your Mac’s touchpad, to bring up the context menu in the simulator. Alternatively, you can enable deep presses in the simulator by going to the Hardware menu and selecting Touch Pressure ► Deep Press.



Figure 10-26. Debugging the context menu from the Apple Watch simulator

To dismiss the Record Workout Interface Controller, create an `exit()` function, as shown in Listing 10-6, which will call the `dismiss()` method to dismiss the Interface Controller. This method behaves in the same fashion as the `dismissViewController()` method in iOS, by dismissing modal Interface Controllers from the view hierarchy. Use the Connection Inspector in Interface Builder to set the `exit()` method as the selector for the Exit button, in the same fashion you connected the selector for the context menu.

Listing 10-6. Dismissing the Record Workout Screen
(RecordInterfaceController.swift)

```
class RecordInterfaceController: WKInterfaceController {  
    ...  
    @IBAction func exit() {  
        dismiss()  
    }  
}
```

Creating a New Workout Using Core Location and Core Motion

The IOTFit app started as a simple location-based workout tracking app, and as you learned more sensor-based frameworks, it expanded into an accurate, detailed application through the adoption of Core Motion and HealthKit. Complementing its wide adoption in fitness applications, Apple included both of these frameworks, as well as Core Location, among the set of binaries that can run untethered on an Apple Watch.

To begin implementing the stand-alone features of the Apple Watch app, start with the Record Workout Interface Controller. It serves the same purpose as the Create Workout View Controller from the iOS project, allowing the user to start and stop a workout session. As mentioned at the beginning of the chapter, one of the goals of adding the IOTFitWatch app to the IOTFit iOS project is to share code between applications. Quickly skimming through the CreateWorkoutViewController class, you will notice that much of the heavy lifting, including logging locations, saving the workout to HealthKit, and formatting the time strings was taken care of by the WorkoutDataManager class. Because a great deal of this functionality is also applicable to the watchOS app, you should try to add it to the watchOS target. To perform this action, click the WorkoutDataManager.swift file in the Project Navigator, select the File Inspector tab (the one with the Document icon) in the right pane of Xcode, and click the check box next to IOTFitWatch Extension, as shown in Figure 10-27.

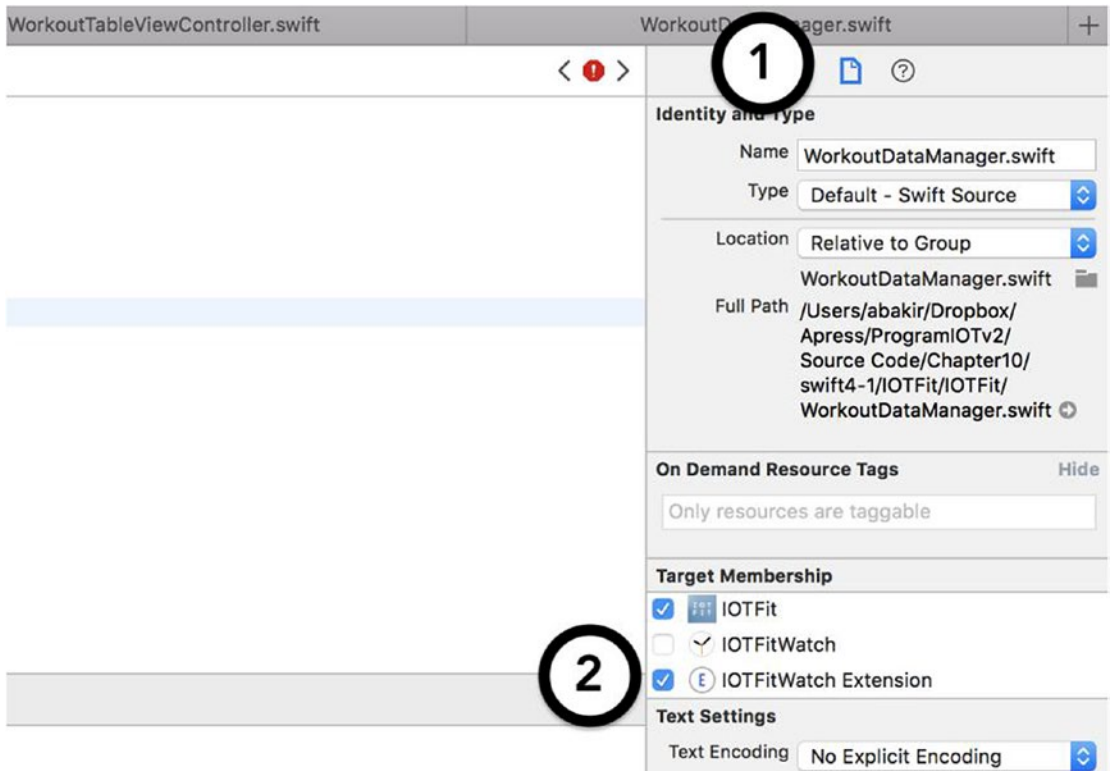


Figure 10-27. Adding the *WorkoutDataManager* class to the *IOTFitWatch Extension* target

With Xcode still set to compile for the iPhone + Apple Watch simulator, attempt to compile the watchOS app with the *WorkoutDataManager* class now included in the target. Compilation should fail, owing to being unable to resolve the definitions for *WorkoutType* and *WorkoutState*. In the iOS-only implementation of the *IOTFit* project, these types were defined inside the *CreateWorkoutViewController* class, which cannot be included in the watchOS target. To fix the compilation issue, move these values to the top of the *WorkoutDataManager* class, as shown in Listing 10-7. The watchOS app should now compile successfully.

Listing 10-7. Completing the *WorkoutDataManager* Class

```
import Foundation
import CoreLocation
import HealthKit
```

```

enum WorkoutState {
    case inactive
    case active
    case paused
}

struct WorkoutType {
    static let automotive = "Driving"
    static let running = "Running"
    static let bicycling = "Bicycling"
    static let stationary = "Stationary"
    static let walking = "Walking"
    static let unknown = "Unknown"
}

let timerInterval : TimeInterval = 1.0
...
class WorkoutDataManager {
    static let sharedManager = WorkoutDataManager()
    ...
}

```

To continue with the implementation of the Record Workout Interface Controller, port over the state initialization code from the Create Workout View Controller. Looking back at the iOS version, you will notice that the view controller maintained its state via a series of properties that were originally set to 0 values. When the view controller was loaded, you used these values to initialize the labels in the user interface. In Listing 10-8, I have expanded the Record Workout Interface Controller class to include these properties and the `updateUserInterface()` method. In particular, you will notice that watchOS interface controllers are awakened via the `willActivate()` method (vs. `viewWillAppear()` on iOS) and that the convenience method for setting button titles is much shorter.

Listing 10-8. Adding State to the RecordInterfaceController Class

```

class RecordInterfaceController:
    WKInterfaceController {
        ...
        var currentWorkoutState = WorkoutState.inactive
        var currentWorkoutType = WorkoutType.unknown

        var workoutStartTime : Date?
        var lastSavedTime : Date?
        var workoutDuration : TimeInterval = 0.0
        var workoutTimer : Timer?

        var workoutAltitude : Double = 0.0
        var workoutDistance : Double = 0.0
        var averagePace : Double = 0.0
        var floorsAscended : Double = 0.0
        var workoutSteps : Double = 0.0
        var lastSavedLocation : CLLocation?
        var isMotionAvailable : Bool = false
        ...
        override func willActivate() {
            super.willActivate()
            updateUserInterface()
        }
        ...
        func updateUserInterface() {

            switch(currentWorkoutState) {
            case .active:
                toggleButton?.setTitle("Stop")
            case .paused:
                toggleButton?.setTitle("Resume")
            default:
                toggleButton?.setTitle("Start")
            }
        }
    }
}

```

Next, you have to implement the methods to start or stop the workout. As shown in Listing 10-9, enable this by porting over the methods to start, stop, and reset the workout. These methods work by using the `currentWorkoutState` variable to determine the current state of the Interface Controller and then using that to request device permissions, start the measurements, or stop them and reset the Interface Controller back to the initial state. These methods were almost directly copied from the iOS version. Make sure you connect the `toggleWorkout()` method to the Start button on the storyboard to enable it.

Listing 10-9. Toggling State Within the `RecordInterfaceController` Class

```
import UIKit
import CoreLocation
import CoreMotion

class RecordInterfaceController: WKInterfaceController {
    ...
    var pedometer : CMPedometer?
    var motionManager : CMMotionActivityManager?
    var altimeter : CMAltimeter?
    let locationManager = CLLocationManager()
    ...
    func resetWorkoutData() {
        lastSavedTime = Date()
        workoutDuration = 0.0
        workoutDistance = 0.0
        workoutAltitude = 0.0
        workoutSteps = 0
        floorsAscended = 0
        averagePace = 0.0
        currentWorkoutType = WorkoutType.unknown
    }

    func startWorkout() {
        currentWorkoutState = .active
        UserDefaults.standard.setValue(true, forKey:
            "isConfigured")
    }
}
```

```

UserDefaults.standard.synchronize()
workoutTimer = Timer.scheduledTimer(timeInterval:
    timerInterval, target: self, selector:
    #selector(updateWorkoutData), userInfo: nil,
    repeats: true)
locationManager.startUpdatingLocation()
lastSavedTime = Date()
workoutStartTime = Date()
WorkoutDataManager.sharedManager.createNewWorkout()

if (CMMotionManager().isDeviceMotionAvailable &&
    CMPedometer.isStepCountingAvailable() &&
    CMAltimeter.isRelativeAltitudeAvailable()) {
    isMotionAvailable = true

    startPedometerUpdates()
    startActivityUpdates()
    startAltimeterUpdates()
} else {
    NSLog("Motion acitivity not available on device.")
    isMotionAvailable = false
}
}

func stopWorkoutTimer() {
    workoutTimer?.invalidate()
}

@IBAction func toggleWorkout() {

    switch currentWorkoutState {
    case .inactive:
        requestLocationPermission()
    case .active:
        currentWorkoutState = .inactive
        stopWorkoutTimer()
        pedometer?.stopUpdates()
    }
}

```

```

motionManager?.stopActivityUpdates()
altimeter?.stopRelativeAltitudeUpdates()

if let workoutStartTime = workoutStartTime {
    let workout = Workout(startTime:
        workoutStartTime, endTime: Date(), duration:
        workoutDuration, locations: [], workoutType:
        self.currentWorkoutType, totalSteps: workoutSteps,
        flightsClimbed: floorsAscended, distance: workoutDistance)
        WorkoutDataManager.sharedManager.
        saveWorkout(workout)
    }
default:
    NSLog("toggleWorkout() called out of context!")
}
updateUserInterface()
}
}

```

To start monitoring the user's location, motion activity, and to gain access to HealthKit, port over the methods for requesting permission and starting updates, as shown in Listing 10-10. Compared to the iOS version, the biggest difference here is that you must limit some of the configuration options for the hardware manager objects, as they are not available on watchOS.

Listing 10-10. Requesting and Monitoring Hardware Updates in the RecordInterfaceController Class

```

class RecordInterfaceController: WKInterfaceController {
    ...
    func startPedometerUpdates() {
        guard let workoutStartTime = workoutStartTime
else { return }

        pedometer = CMPedometer()
        pedometer?.startUpdates(from: workoutStartTime,
            withHandler: { [weak self] (pedometerData : CMPedometerData?,
                error: Error?) in

```

```

NSLog("Received pedometer update!")
if let error = error {
    NSLog("Error reading pedometer data")
    return
}

guard let pedometerData = pedometerData,
    let distance = pedometerData.distance as?
        Double,
    let averagePace =
        pedometerData.averageActivePace as? Double,
    let steps = pedometerData.numberOfSteps as?
        Int,
    let floorsAscended =
        pedometerData.floorsAscended as? Int else {
        return
    }
    self?.workoutDistance = distance
    self?.floorsAscended = Double(floorsAscended)
    self?.workoutSteps = Double(steps)
    self?.averagePace = averagePace
}}
}

func startActivityUpdates() {

    motionManager = CMMotionActivityManager()
    motionManager?.startActivityUpdates(to:
    OperationQueue.main, withHandler: { [weak self]
        (activity : CMMotionActivity?) in
        guard let activity = activity else {
        return
        }
        if activity.walking {
            self?.currentWorkoutType = WorkoutType.walking
        }
    }
}

```

```

    } else if activity.running {
        self?.currentWorkoutType = WorkoutType.running
    } else if activity.cycling {
        self?.currentWorkoutType =
            WorkoutType.bicycling
    } else if activity.stationary {
        self?.currentWorkoutType =
            WorkoutType.stationary
    } else {
        self?.currentWorkoutType = WorkoutType.unknown
    }
})
}

func startAltimeterUpdates() {
    altimeter = CMAltimeter()
    altimeter?.startRelativeAltitudeUpdates(to:
        OperationQueue.main, withHandler: { [weak self]
            (altitudeData : CMAltitudeData?, error: Error?) in
            if let error = error {
                NSLog("Error reading altimeter data")
                return
            }

            guard let altitudeData = altitudeData,
                let relativeAltitude =
                    altitudeData.relativeAltitude as? Double else {
                return
            }
            self?.workoutAltitude = relativeAltitude
        })
}

func requestLocationPermission() {
    if CLLocationManager.locationServicesEnabled() {

```



```

locationManager.desiredAccuracy =
    kCLLocationAccuracyNearestTenMeters
locationManager.distanceFilter = 10.0
locationManager.allowsBackgroundLocationUpdates =
true
locationManager.delegate = self

switch(CLLocationManager.authorizationStatus()) {
case .notDetermined:
    locationManager.requestWhenInUseAuthorization()
case .authorizedWhenInUse :
    requestAlwaysPermission()
case .authorizedAlways:
    resetWorkoutData()
    startWorkout()
default:
    NSLog("Unable to request location")
}
} else {
    NSLog("Unable to init location")
}
}

func requestAlwaysPermission() {
    if let isConfigured =
        UserDefaults.standard.value(forKey:
            "isConfigured") as? Bool, isConfigured == true {
        startWorkout()
    } else {
        locationManager.requestAlwaysAuthorization()
    }
}
}

```

Finally, to complete the setup process, you must extend the Record Interface Controller to implement the `CLLocationManagerDelegate` protocol. As shown in Listing 10-11, you can port this over directly from the iOS app, with the exception of the pause/resume delegate methods, which are not available on watchOS.

Listing 10-11. Adding Support for the `CLLocationManagerDelegate` Protocol to the `RecordInterfaceController` Class

```
class RecordInterfaceController: WKInterfaceController {
    ...
}
extension RecordInterfaceController : CLLocationManagerDelegate {
    func locationManager(_ manager: CLLocationManager,
        didChangeAuthorization status: CLErrorAuthorizationStatus) {
        switch status {
            case .authorizedWhenInUse:
                requestAlwaysPermission()
            case .authorizedAlways:
                resetWorkoutData()
                startWorkout()
            case .denied:
                NSLog("location permission denied")
            default:
                NSLog("Unhandled Location Manager Status:
                    \(status)")
        }
    }
    func locationManager(_ manager: CLLocationManager,
        didUpdateLocations locations: [CLLocation]) {
        guard let mostRecentLocation = locations.last else {
            NSLog("Unable to read most recent location")
            return
        }
    }
}
```

```
lastSavedLocation = mostRecentLocation  
NSLog("Most recent location: \(String(describing:  
    mostRecentLocation))")  
  
WorkoutDataManager.sharedManager.addLocation(coordinate:  
    mostRecentLocation.coordinate)  
    }  
}
```

To test the application, select the iPhone + Apple Watch simulator or your iPhone + Apple Watch and attempt to run the IOTFitWatch target. The first time you run the Apple Watch target, you will be prompted to open the IOTFit app on your iPhone, to accept the Health, Motion, and Location permissions, as shown in Figure 10-28. Press the Start button in the IOTFit app for iOS, to request these permissions. Although the app can run without being paired to the phone for most of its functions, accepting permissions requests on the phone is still a limitation of watchOS.

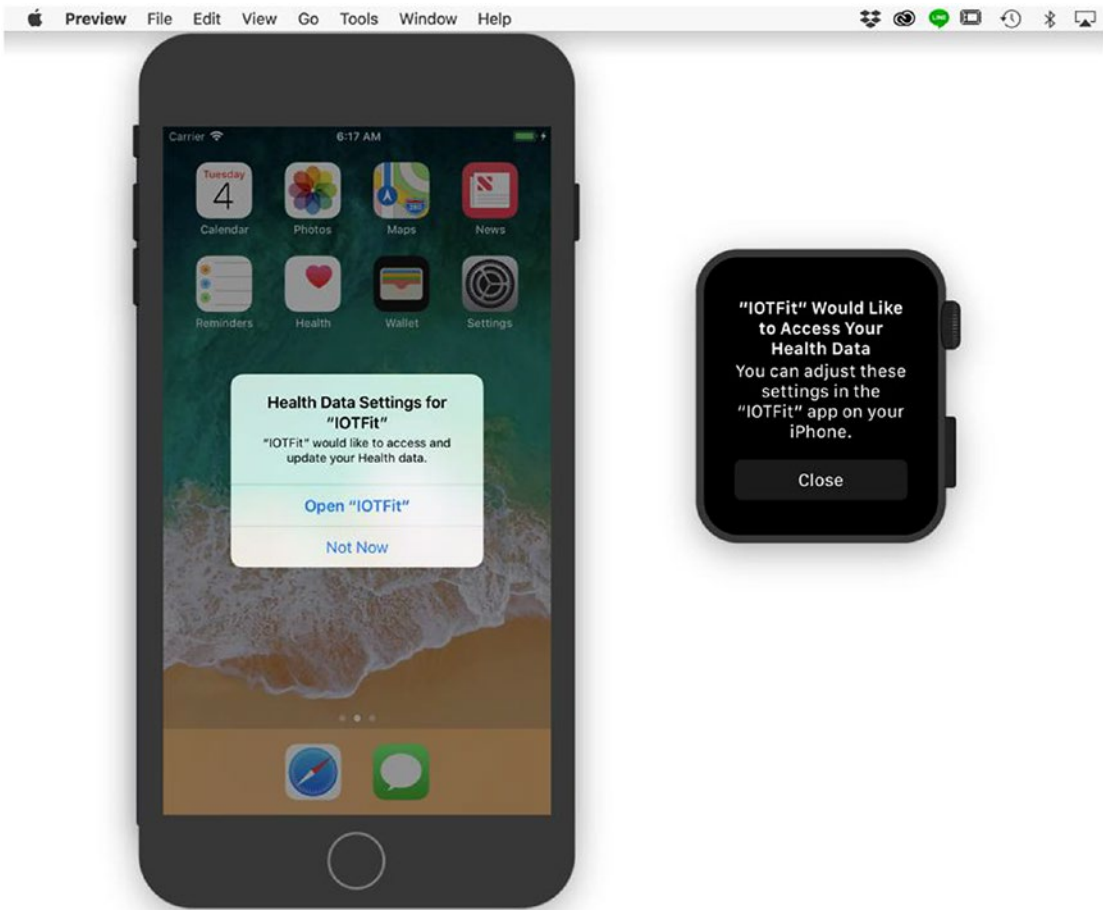


Figure 10-28. *Permission request dialog boxes for an Apple Watch app*

After you have accepted all of the permissions, you can press the Close button on the Apple Watch app to return to the Workout History Interface Controller. Now, when you present the Record Workout Interface Controller from the context menu, you will be able to start and stop a workout via the user interface.

Using HealthKit to Populate the Workout History Table

To complete the core functionality of the IOTFitWatch app, you must populate the Workout History Interface Controller with data on the user's past workouts. In the same manner as the Record Workout Interface Controller, you can leverage code from the IOTFit iOS app, to help with you with this task. In the iOS app, the Workout History was managed by the `WorkoutTableViewController` class. Looking at this class, you will remember that you fetched by the data for the table view by calling the `loadWorkoutsFromHealthKit()` method on the shared Workout Data Manager, which queried HealthKit for all the workouts the user saved on their device. Because you successfully set up HealthKit permission and imported the Workout Data Manager into the IOTFitWatch target in the last section, you can leverage them to load the table view data here.

To begin, port over the logic for loading the table. In the IOTFit iOS app, the data for the Workout Table View Controller was refreshed whenever the view controller was presented, via the `viewWillAppear(animated:)` method. In watchOS, the parallel to this method is the `willActivate()` method. As shown in Listing 10-12, expand the `InterfaceController` class to fetch data every time it becomes active. The `WKInterfaceTable` class does not define a protocol for initializing its data, so include a stub method for refreshing the table view's data.

Listing 10-12. Fetching Data When the Workout History Interface Controller Becomes Active (`InterfaceController.swift`)

```
class InterfaceController: WKInterfaceController {
    @IBOutlet var workoutTable: WKInterfaceTable?
    var workouts : [Workout]?
    let dateFormatter = DateFormatter()

    override func awake(withContext context: Any?) {
        super.awake(withContext: context)
        // Configure interface objects here.
        dateFormatter.dateStyle = .medium
    }
}
```

```

override func willActivate() {
    // This method is called when watch view
    // controller is about to be visible to user
    super.willActivate()

    WorkoutDataManager.sharedManager.
loadWorkoutsFromHealthKit { [weak self]
    (fetchedWorkouts: [Workout]?) in
        if let fetchedWorkouts = fetchedWorkouts {
            self?.workouts = fetchedWorkouts
            DispatchQueue.main.async {
                self?.refreshTable()
            }
        }
    }
}

func refreshTable() {
    //TODO: build table here
}
}

```

There are three primary tasks required to set up a table view in watchOS: specify the number of rows in the table, looking up cells by their identifier string, and initializing the discovered cells. In Listing 10-13, I have expanded the `refreshTable()` method to include this logic. Pay special attention to use the exact same identifier string for the row controller that you used in the storyboard.

Listing 10-13. Fetching Data When the Workout History Table View (InterfaceController.swift)

```

class InterfaceController: WKInterfaceController {
    ...
    func refreshTable() {
        guard let workouts = workouts else { return }
        workoutTable?.setNumberOfRows(workouts.count,
        withRowType: "WorkoutRow")
    }
}

```

```

for index in 0..workouts.count {
    guard let row = workoutTable?.rowController(at:
index) as? WorkoutRowController else { return }

    let selectedWorkout = workouts[index]
    let dateString = dateFormatter.string(from:
selectedWorkout.startTime)
    let durationString =
WorkoutDataManager.stringFromTime(timeInterval:
selectedWorkout.duration)

    let detailText = String(format: "%.0f m | %@",
arguments: [selectedWorkout.distance,
durationString])

    row.dateLabel?.setText(dateString)
    row.durationLabel?.setText(detailText)
}
}
}

```

To add the icons for the app, import `FontAwesome.swift` (<https://github.com/thii/FontAwesome.swift>) by dragging and dropping its Swift and font (.otf) files into the project, just as you did in the tvOS project in Chapter 9. After copying the files into the project, you must change the file ownership to expose it to watchOS correctly. For the font files, make sure that they are part of all three targets (IOTFit, IOTFitWatch, and IOTFitWatch Extension). As mentioned earlier, this change is required, because watchOS uses the Watch App target to store static files. Among the Swift files, remove the UIView subclasses from the IOTFitWatch Extension target, including `FontAwesomeBarButtonItem.swift`, `FontAwesomeTabBarItem.swift`, and `FontAwesomeSegmentedControl.swift`. After making these changes, the IOTFitWatch app should now compile again.

To set the image for each row, you can adapt the logic from the IOTHomeTV app: map the workout type to a Font Awesome icon name and then use that name to create a UIImage object. Just as with the IOTHomeTV app, use the official search page for Font Awesome to find the symbol names: www.fontawesome.com/icons?d=gallery&m=free. In Listing 10-14, I have updated the `refreshTable()` method with this logic.

Listing 10-14. Fetching Data When the Workout History Table View (InterfaceController.swift)

```

class InterfaceController: WKInterfaceController {
    ...
    func refreshTable() {
        guard let workouts = workouts else { return }
        workoutTable?.setNumberOfRows(workouts.count,
            withRowType: "WorkoutRow")

        for index in 0..let icon: FontAwesome

            switch selectedWorkout.workoutType {
                case WorkoutType.walking:
                    icon = FontAwesome.walking
                case WorkoutType.bicycling:
                    icon = FontAwesome.bicycle
                case WorkoutType.automotive:
                    icon = FontAwesome.car
                default:
                    icon = FontAwesome.dumbbell
            }

            let faImage = UIImage.fontAwesomeIcon(name: icon,
                style: .solid, textColor: UIColor.white, size:
                CGSize(width: 50, height: 50))
            row.icon?.setImage(faImage)
        }
    }
}

```


To test that the Workout History Interface Controller is now populated with data from HealthKit, run the app in the iPhone + Apple Watch Simulator or on your Apple Watch hardware. The iOS and Apple Watch apps should now both show records for the sample workouts you generated while testing the Record Workout Interface Controller, as shown in Figure 10-29.



Figure 10-29. *Workout History screens on both apps, including populated data*

Summary

In this chapter, you learned how to take advantage of watchOS to build an Apple Watch version of the IOTFit app, which was able to share code and workout data with the original iOS version of the app. In some places, the setup process for the Apple Watch app was more complicated than the iOS version, for example, putting together the user interface using stack views only. However, other parts of the watchOS app, like creating the table view, used stripped down versions of their iOS counterparts and were easier to

implement than their iOS counterparts. Much like the Apple TV project in Chapter 9, you were able to take advantage of running frameworks natively on the hardware and were able to make an Apple Watch app with all of the same workout tracking capabilities as the iOS version.

In keeping with the theme of the book, this was a perfect example of an IoT app, because users were able to use a “thing” (the Apple Watch) as another way of providing data for a bigger system. In this chapter, the large system you were able to take advantage of was HealthKit, which Apple automatically secures and syncs between devices for users.

CHAPTER 11

Using Face ID, Touch ID, and Keychain Services to Secure Your Apps

As you have learned so far in this book, Internet of Things (IoT) technologies can be used to expand the conveniences in people's lives. Compared to a few years ago, these devices have increased in their capabilities, as have the barriers of entry to creating a device. In this book alone, you have already taken advantage of an Arduino, Raspberry Pi, and iPhone as IoT sensors.

However, as they say, new medicines come at the cost of new side effects. For the IoT, the most unfortunate, unexpected side effect of its adoption has been an increase in security exploits, originating from improperly secured IoT devices. As mentioned throughout the book, much of the popularity of IoT has been powered by the availability of affordable system-on-a-chip (SoC) solutions, such as the ESP32 you used in earlier chapters. Unfortunately, when a widely used system's design shortcomings are exploited, its high adoption rate can lead to devastating damage.

This was precisely what has occurred with the Mirai botnet, which exploited a known set of commonly used default passwords on IoT devices to automatically install, replicate, and run a program to use infected devices for a coordinated attack on target systems, otherwise known as a *distributed denial-of-service* (DDoS) attack. Having affected more than 600,000 devices, in 2017 it took down several Domain Name System (DNS) servers, which form the backbone of routing Internet traffic, and was cited as

the largest DDoS attack publicly recorded.¹ As of this writing (late 2018), the botnet has continued to increase in sophistication and find new targets.

In Chapter 8, when you learned how to build a web server using a Raspberry Pi, one of the methods you learned for increasing security on a device was adding an SSL certificate to the web server, so that all of the data transferred between the devices and external clients (such as the iOS app) could be secured via HTTPS. Although not a perfect or complete solution, this was one step in making it harder for your users' data to be intercepted via simple traffic sniffing (using a program to monitor packets transferred on a network). For a complete solution, you should always try to research all of the paths to accessing an administrator account on a system and secure them, including changing passwords, using two-factor authentication, adding input validation to your web application APIs, and installing security patches and helper utilities to prevent known attacks.

In this chapter, you will learn three techniques on the iOS app side that you can use to help prevent your users' information from being leaked or stolen from their devices: Face ID, Touch ID, and Keychain Services. As an iPhone user, you may already be familiar with the first two services from your Home screen. They allow you to unlock your Home screen using a 3D surface scan of your face (on iPhone X and newer versions) or using a fingerprint (on iPhone 5S–iPhone 8 Plus, and iPad Air 2 and later versions). However, you may not be aware that you can use Keychain Services in your own apps to restrict access to parts of your application. This is a feature popular in many password repository applications (such as 1Password). Similarly, Keychain Services allows you to create an encrypted database for your app's data, which can only be accessed while your app is active and in the foreground on the user's devices. Keychain Services is Apple's recommended method for storing user passwords, API keys, and SSL certificates, which could be exploited greatly, if they were leaked from your app.

Learning Objectives

In this chapter, you will learn how to use Face ID, Touch ID, and Keychain Services to secure data, by expanding the IOTFit application from earlier chapters to use these APIs, to restrict access to the application to only the owner of the iOS device or via a password

¹Elie Bursztein, "Inside the infamous Mirai IoT Botnet," Cloudflare, <https://blog.cloudflare.com/inside-mirai-the-infamous-iot-botnet-a-retrospective-analysis/>, September 8, 2018.

stored in the keychain. By implementing these improvements to the IOTFit app, you will learn the following key concepts for IoT application development:

- Creating a lock screen user interface
- Determining Face ID and Touch ID availability on a device
- Using Face ID and Touch ID to restrict access to an application
- Storing and retrieving values from the iOS keychain
- Detecting when an app returns from the background

The workout data stored in the HealthKit store is encrypted and managed by the Health app on the device; however, you can add further security by disabling the portion of the `WorkoutDataManager` class that was used to backup data to a `.plist` file in the Documents folder for the application. Although the data in an application's bundle is not available to other applications, it is possible to extract this data via backup applications on a Mac or PC.

As with previous projects in this book, the source code for the completed project is available from the GitHub page for this book (<https://github.com/Apress/program-internet-of-things-w-swift-for-ios>). If you would like to review how the health features of the IOTFit app were created, please refer to Chapters 2–4. If you would like to review how HTTPS was added to a web server application, please refer to Chapter 8.

Setting Up the Project

To get started, create a duplicate of the IOTFit app from Chapter 10, either by copying your old project or downloading a copy from the GitHub repository for this book. As Face ID is easier to access unintentionally than Touch ID or typing a password into an iPhone, Apple requires permission from the user to enable it in your app. As in the case of HealthKit (Chapter 4), this permission prompt will appear the first time your users try to use Face ID in the app. Another similarity to HealthKit is that the permission prompt is enabled by defining a message string for it in your app's `Info.plist` file.

As shown in Figure 11-1, to set the Face ID permission prompt message for your app, open the `Info.plist` file for the IOTFit target in the Project Navigator, click the (+) button in any column, and add a new entry for the Privacy – Face ID Usage Description key-value pair. In my implementation, I used the message, “IOTFit would like to use Face ID to secure your workout data.”

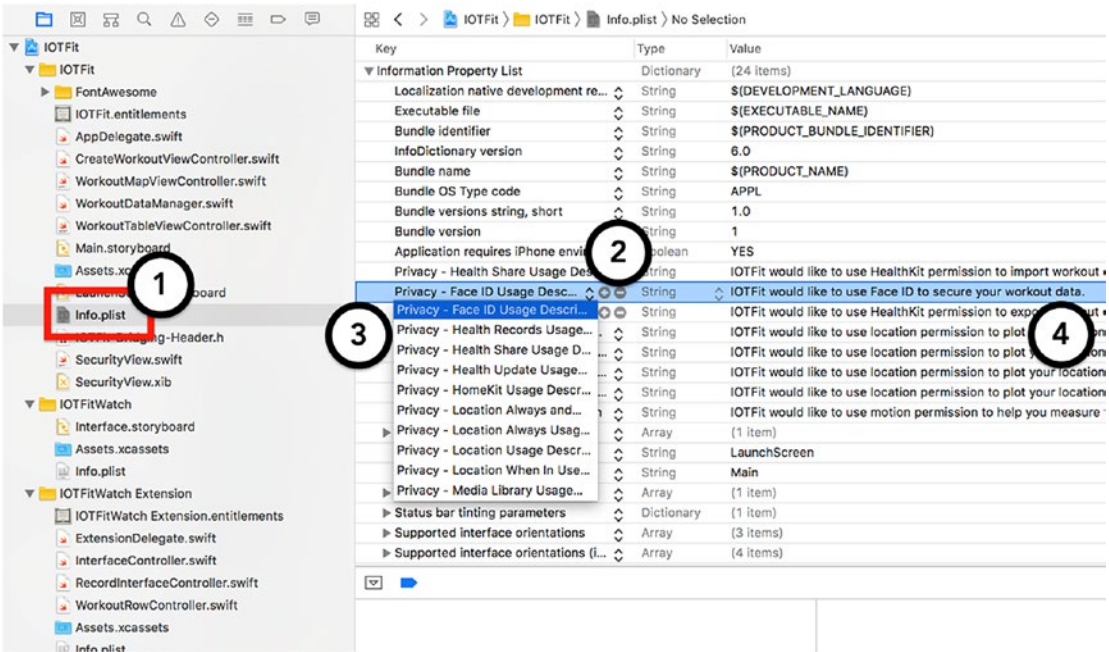


Figure 11-1. Adding a message string for the Face ID permission prompt

Creating a Lock Screen User Interface

When I was thinking about how to design a project for demonstrating Face ID, the best example that popped into my head was the lock screen for the 1Password password manager app. As shown in Figure 11-2, before you can access the user interface for the app, 1Password asks you to type in a password or press a Face ID button to unlock the app with Face ID. When verification is successful, the lock screen performs an animation that mimics a door opening, revealing the main user interface when it is complete.

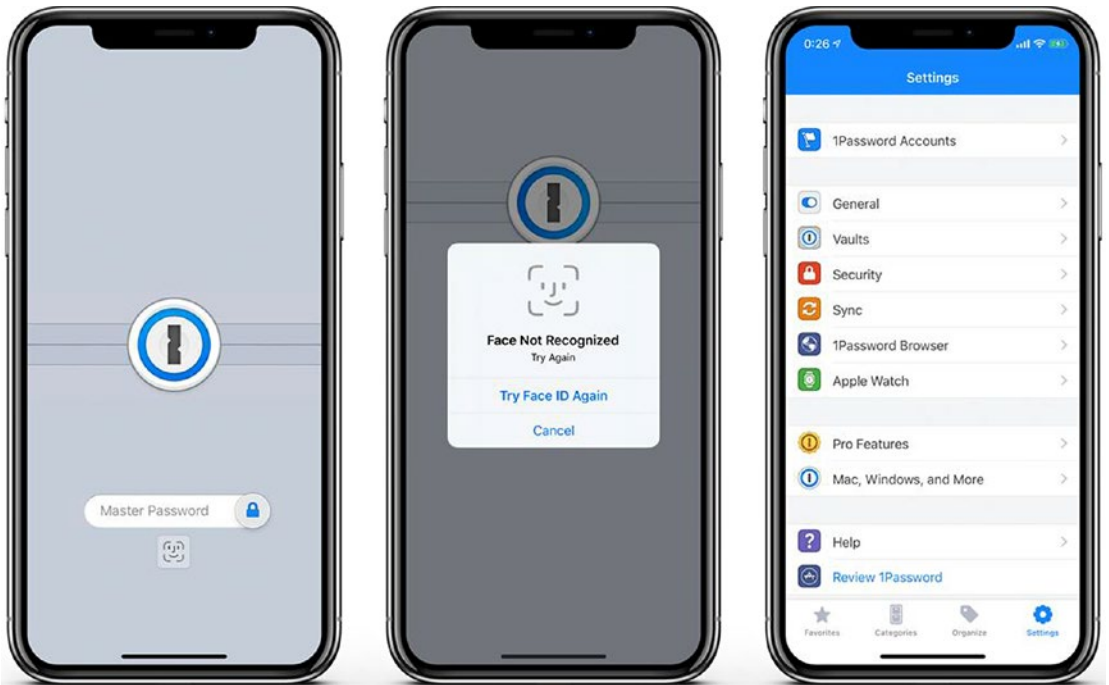


Figure 11-2. Lock screen for 1Password password manager

Although the full animation is a bit beyond the scope of this project, I thought the idea of showing a full-screen overlay to block sensitive features would be universally appropriate. In the IOTFit app, the most sensitive pieces of data are the user's past location and workout data. As shown in Figure 11-3, I have expanded the wireframes for the app to show a simple lock screen over the Workout History and Last Run screens. For the sake of simplicity, you will reveal the lock screen whenever the screens are navigated to within the Tab bar and when the app becomes active.



Figure 11-3. Design wireframes for the IOTFit app, including lock screen

As with 1Password, I have included a button to initiate Face ID (or Touch ID), and a password text field as a backup authentication method. In the app, by detecting which sensor is available on the device, you can update the text to show Touch ID or Face ID. As opposed to 1Password, I only applied the lock screen over the screens that display data, as the primary purpose of the app is recording workouts, and it is a good design practice to reduce the friction leading up to this activity.

To prevent the lock screen from blocking access to the Tab bar, I suggest implementing the lock screen as a `UIView` that will be displayed over the contents of the Workout History and Last Run view controllers. After authentication is successful, you will use a simple animation to dismiss the view and present the data. To begin this process, you must create a new `UIView` subclass and NIB (`.xib`) file to represent the lock screen.

As with previous examples in the book, create the `UIView` subclass by going to the File menu and selecting New ► File... . From the template picker that appears, select iOS ► Cocoa Touch Class. As shown in Figure 11-4, create a subclass of `UIView` named `SecurityView`.

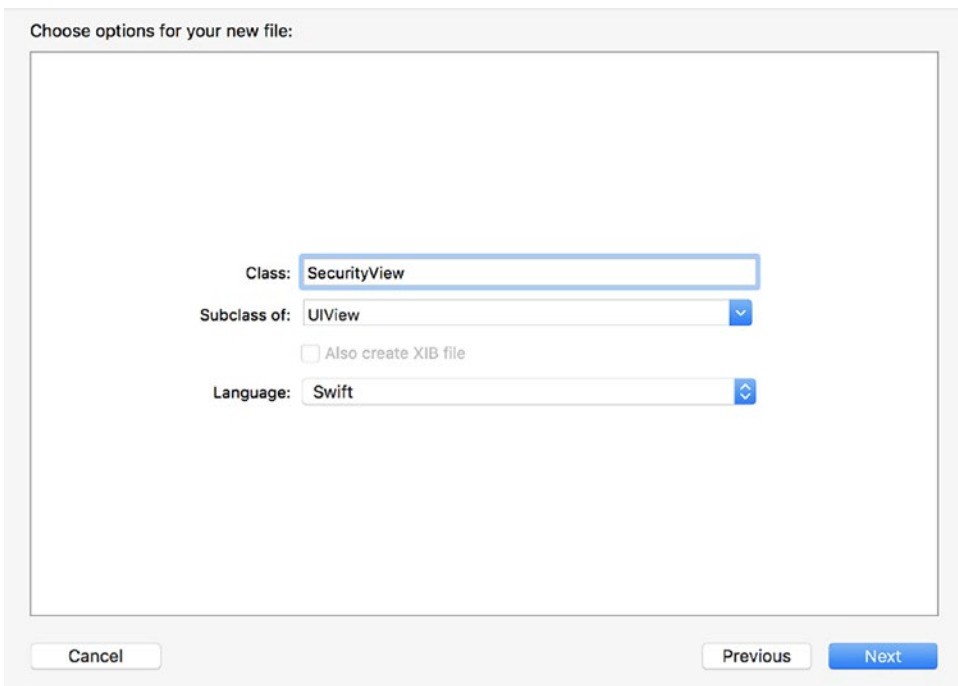


Figure 11-4. Adding a `UIView` subclass to the project

Because `UIView` subclasses can be used without modification in Objective-C classes, the first time you add a new one to a Swift project, you will be asked to add an Objective-C bridging header to the project, as shown in Figure 11-5. Select Create Bridging Header to create the bridging header and continue with the setup process.

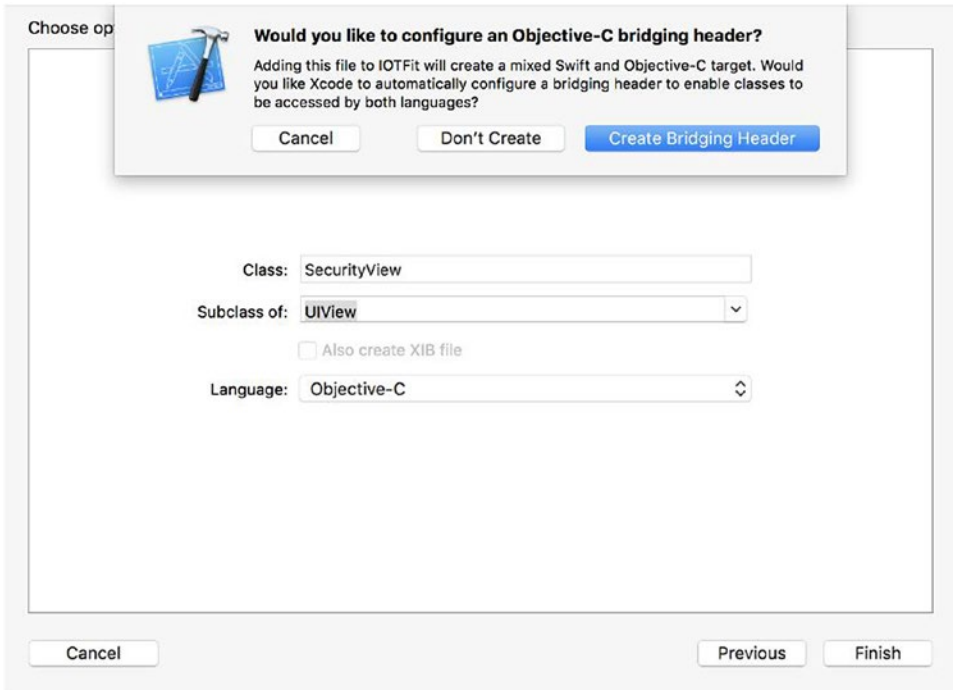


Figure 11-5. Prompt for adding an Objective-C bridging header to a Swift project

Next, you must create a NIB (.xib) file to manage the visual layout of the `SecurityView` class. While previous examples in the book managed all of these user interfaces on a single storyboard file, when you are creating a view that will be reused multiple places in an app, the common practice in the industry is to manage the single view in its own NIB file. NIBs are the original visual layout tool for Interface Builder and are intended to manage single views or view controllers, as opposed to storyboards, which are designed to manage common segues between multiple view controllers. To create a new NIB, open the File menu again and select New ► File... one more time. This time, scroll further down in the iOS template picker, as shown in Figure 11-6, and select View. When asked to name the file, use the name `SecurityView.xib`, to match the `UIView` subclass.

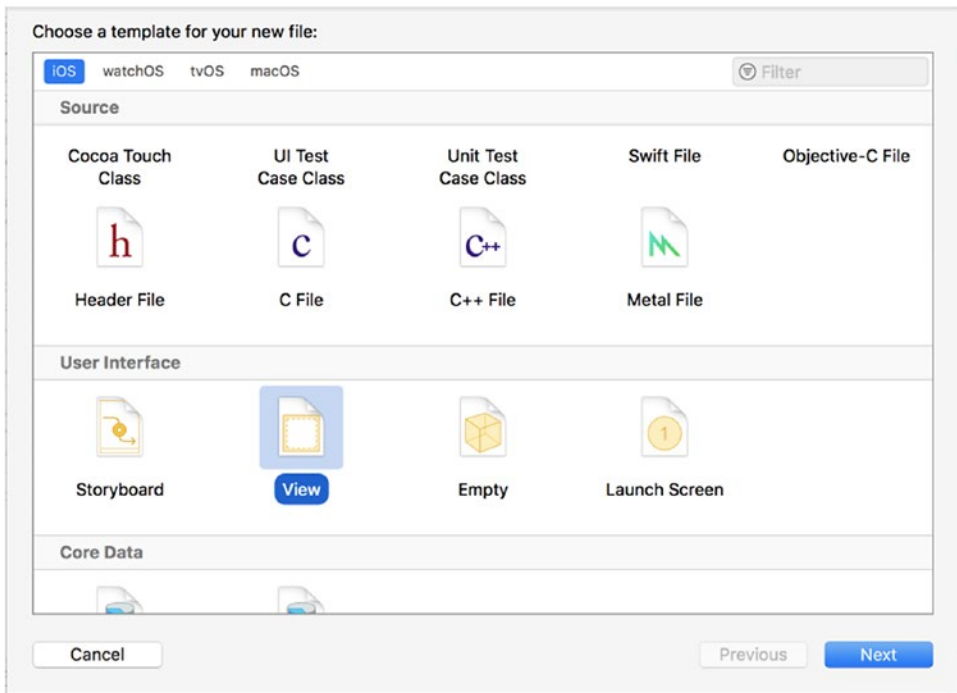


Figure 11-6. *Selecting a View NIB from the Xcode template picker*

After clicking the NIB file in the Project Navigator, you should see a single blank view inside Interface Builder, similar to when you created a new storyboard file. Using Table 11-1 as a guide, lay out the user interface, paying particular attention to the UITextField for password entry and the UIButton for presenting Face ID.

Table 11-1. *Styling for Security View User Interface Elements*

Element Name	Text Style	Align Relative to	Top Margin	Bottom Margin	Left Margin	Right Margin
“Verification Required” label	Title 1	View	60	40	40	40
“Description” label	Footnote	“Verification” label	40	40	40	40
“Password” text field	Body	“Description” label	40	8	80	80
“or” label	Footnote	“Password” text field	8	8	80	80
“Use Face ID” button	Body	“or” label	8	—	80	80

The NIB file for your completed security view should resemble the screenshot in Figure 11-7, after applying these styles. In the same manner as in previous chapters, your next step should be to define the SecurityView class, including its properties.

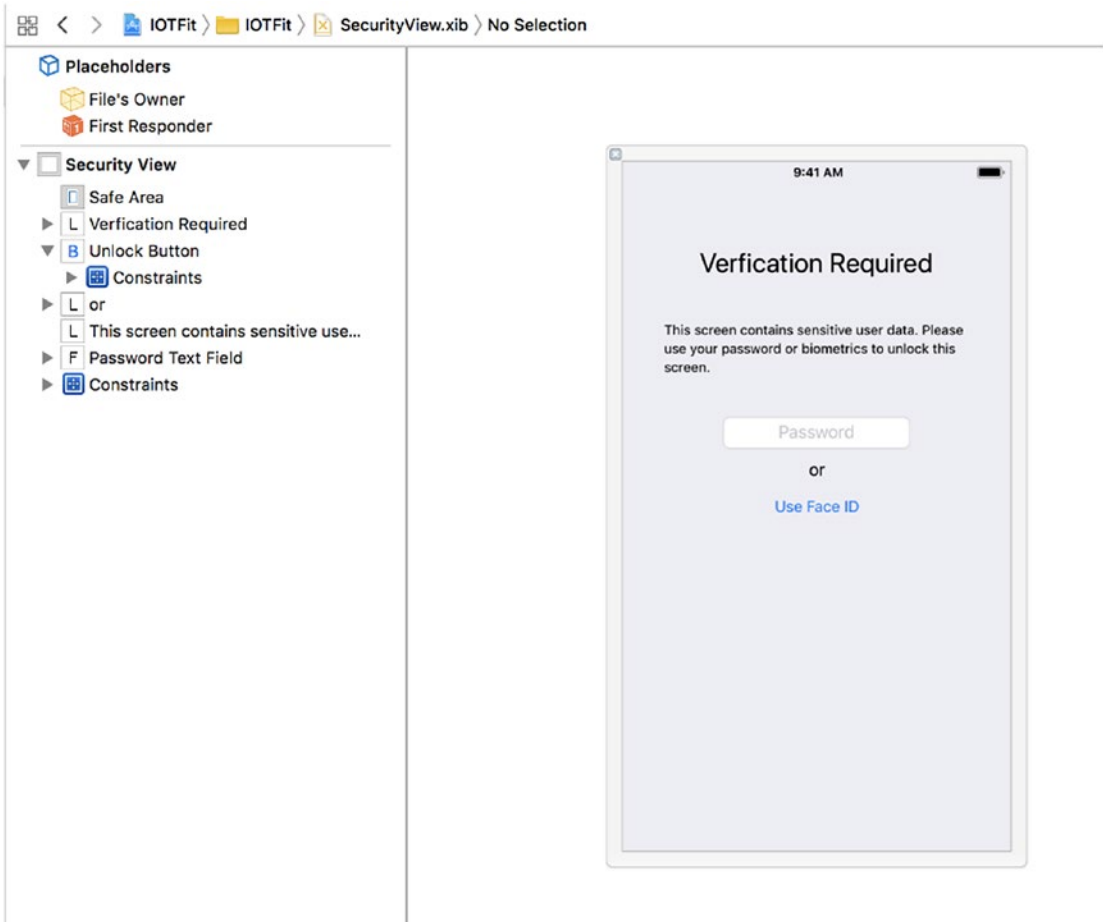


Figure 11-7. Final layout for the security view

Switch back to the SecurityView.swift class and use my example in Listing 11-1 as a starting point for your implementation. Pay careful attention to declare the button and text field with the @IBOutlet keyword and the button handlers with the @IBAction keyword, so that both will be compatible with Interface Builder.

Listing 11-1. Initial Definition for the SecurityView Class

```
import UIKit

class SecurityView: UIView {

    @IBOutlet var unlockButton: UIButton?
    @IBOutlet var passwordTextField: UITextField?

    @IBAction func validatePassword(sender:
        UITextField) {
        //password handling will go here
    }

    @IBAction func validateBiometrics(sender:
        UIButton) {
        //biometrics handling will go here
    }
}
```

To round out the SecurityView class, you will have to connect its outlets to the parent class. As with previous examples, you can find detailed instructions for this process in Chapter 1. In particular, do not forget to

- Set the SecurityView class as the parent class in the Identity Inspector (the third tab from the left).
- Connect the unlockButton and passwordTextField on the NIB file to their respective properties in the class (using the Connections Inspector).
- Connect the passwordTextField's delegate property to the SecurityView class (so you can handle its events).
- Connect the unlockButton's Touch Up Inside event to the validateBiometrics(sender:) method.

After making all of the connections, the Connections Inspector for the security view should resemble the screenshot in Figure 11-8.

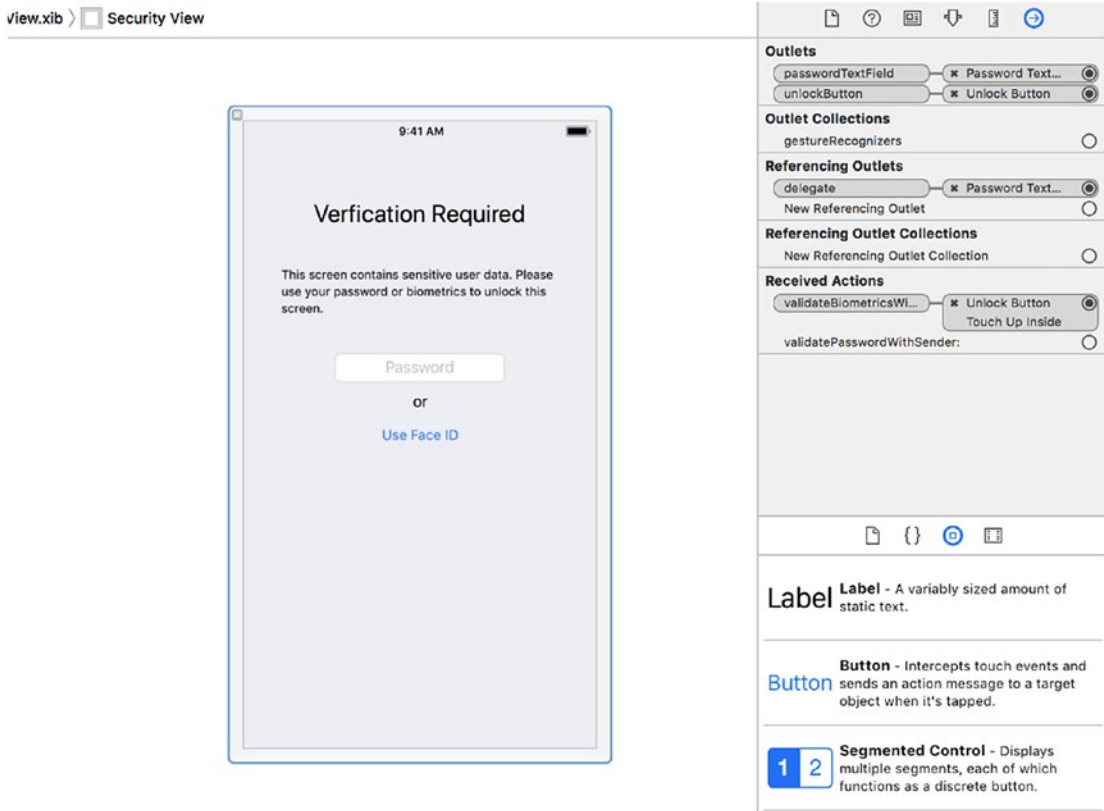


Figure 11-8. Connections Inspector for completed security view

For the final user interface setup tasks, you must display the security view when the `WorkoutMapViewController` and `WorkoutTableViewController` classes become visible. To begin, you must load the security view from the NIB file and make it available to the calling class. Unlike the Main storyboard, which is loaded as part of the app’s initialization process, to load a view from a NIB file, you must attempt to load the file via its bundle path (its relative path in the .app bundle) and verify that the contained class matches your expectations. In Listing 11-2, I have implemented this via a `setupSecurityView()` method, in which I use the `Bundle` class to load this view and then use the main view for calling the view controller to set the size and destination for the security view.

Listing 11-2. Adding the Security View to a View Controller
(WorkoutTableViewController.swift)

```
class WorkoutTableViewController: UITableViewController {
    ...
    var securityView: SecurityView?

    override func viewDidLoad() {
        super.viewDidLoad()
        ...
        setupSecurityView()
    }
    ...
    func setupSecurityView() {
        guard let securityNibItems =
            Bundle.main.loadNibNamed("SecurityView", owner:
                nil, options: nil),
            let securityView = securityNibItems.first as?
                SecurityView else { return }

        securityView.frame = view.frame
        securityView.autoresizingMask = [.flexibleWidth,
            .flexibleHeight]
        self.securityView = securityView
        view.addSubview(securityView)
    }
}
```

For the sake of brevity, my code listings for this chapter will be for the `WorkoutTableViewController` class only, but all of the same logic can be copied directly into the `WorkoutMapViewController` class, with a few exceptions, which I will point out throughout the chapter.

To make the security view visible to the user, you must bring it to the top of the view hierarchy when the `WorkoutTableViewController` or `WorkoutMapViewController` classes become active. In Listing 11-3, I perform this step via the `showSecurityView()` method, which is called from the `viewWillAppear()` method.

Listing 11-3. Presenting the Security View When a Tab Becomes Active
(WorkoutTableViewController.swift)

```

class WorkoutTableViewController: UITableViewController {
    ...
    override func viewWillAppear(_ animated: Bool) {
        super.viewWillAppear(animated)
        ...
        showSecurityView()
    }
    ...
    func showSecurityView() {
        if let securityView = self.securityView,
        securityView.isHidden == true {
            tableView.reloadData()
            securityView.alpha = 1.0
            securityView.isHidden = false
            view.bringSubview(toFront: securityView)
        }
    }
}

```

As the name *view hierarchy* suggests, views are presented as a stack, with the topmost view being the one that the user sees. When you built your user interfaces in Interface Builder, you simply added child views to the main view. As all of the child views were on the same presentation level (z-order), there was no need to manage view visibility. In the case of the security view, it should be presented over every other view in its calling class, to obstruct the user interface during the locked state. I was able to perform this operation in the `showSecurityView()` method. I used the `bringSubview(toFront:)` method on the main view. To avoid side effects, I also used the `isHidden` property on the security view, to prevent the view from being presented twice, if it was already active.

Querying for Sensor Availability

In the same manner you used to implement the other hardware-based APIs in this book, before attempting to use Touch ID or Face ID, you must first check if either is available on the device and that your app has access to them. To query this information, and eventually access the sensor, you will have to establish a *security context* for the app using the LocalAuthentication framework. To maintain its device-based security model, Apple performs all encryption on the iPhone via a discrete microprocessor called the Security Enclave. The LocalAuthentication framework allows you to access the Security Enclave through sessions referred to as contexts, in which you can query the availability of a security policy (for example, authentication via a biometric sensor) and attempt to request validation through that security policy. At no time does your app have access to the user’s personal information or encryption keys, maintaining the security of the device. Success or failure is returned through a Boolean return value and an error object, which will be set to a non-nil value describing the failure reason.

You can establish the security context for the security view using the LAContext class from the LocalAuthentication framework, and you can perform the availability query using the `canEvaluatePolicy(policy:error:)` method on the LAContext object. In Listing 11-4, I have expanded the SecurityView class to include this functionality by maintaining the context and authentication type as properties that can be reused when the authentication request is made later.

Listing 11-4. Detecting the Availability of Biometrics Using a Context (SecurityView.swift)

```
import UIKit
import LocalAuthentication

class SecurityView: UIView {
    ...
    let context = LAContext()

    override func awakeFromNib() {
        super.awakeFromNib()
        commonInit()
    }
}
```

```

private func commonInit() {
    let error: ErrorPointer = nil

    if context.canEvaluatePolicy(
        .deviceOwnerAuthenticationWithBiometrics, error: error) {
        //success!
    } else {
        NSLog("Biometrics unavailable on device")
        unlockButton?.isEnabled = false
    }
}
}

```

As with the `viewDidLoad()` method on a view controller, a view has an `awakeFromNib()` method that is called after it is loaded from a NIB file. As of this writing, Apple's primary security policies are Biometrics and Device Passcode or Biometrics only. Because, the app will use its own passcode, I have chosen to skip the device passcode option for the IOTFit app.

Although the policy query does not return information on the sensor type, after you have determined that the app has access to biometrics, you can use the `biometryType` property of the `LAContext` object to determine this information. In Listing 11-5, I have expanded the `commonInit()` method to check for this information and update the title of the Use Face ID button to properly reflect the sensor type. If biometrics are unavailable, I disable the button, so that the user cannot press it by accident.

Listing 11-5. Accessing the Sensor Type from a Context (`SecurityView.swift`)

```

import UIKit
import LocalAuthentication

enum AuthenticationType : String {
    case faceID
    case touchID
    case password
    case notAvailable
}

```

```

class SecurityView: UIView {
    ...
    var authenticationType: AuthenticationType?
    ...
    private func commonInit() {
        let error: ErrorPointer = nil

        if context.canEvaluatePolicy(
            .deviceOwnerAuthenticationWithBiometrics,
            error: error) {
            switch (context.biometryType) {
            case LABiometryType.faceID:
                authenticationType = AuthenticationType.faceID
                unlockButton?.setTitle("Use Face ID", for:
                    .normal)
            case LABiometryType.touchID:
                authenticationType = AuthenticationType.touchID
                unlockButton?.setTitle("Use Touch ID", for:
                    .normal)
            default:
                authenticationType =
                AuthenticationType.notAvailable
                unlockButton?.isEnabled = false
            }
        } else {
            NSLog("Biometrics unavailable on device")
            unlockButton?.isEnabled = false
        }
    }
}

```

Using Face ID or Touch ID to Restrict Access to Features

Now that you have established the availability of biometrics on the device and determined the sensor type, you can use this information to make the authentication request via the context property on the SecurityView class. To make this call, you will use the `evaluatePolicy(policy:localizedReason:)` method on the `LAContext` object. As described earlier, in keeping with Apple's security restrictions, it will return a Boolean value indicating success or failure and a non-null `Error` object, if the request failed. In Listing 11-6, I make this call from the `validateBiometrics(sender:)` method.

Listing 11-6. Requesting Biometric Authorization (SecurityView.swift)

```
class SecurityView: UIView {
    ...
    @IBAction func validateBiometrics(sender: UIButton) {
        passwordTextField?.resignFirstResponder()
        let permissionString = "Unlock with biometrics to
            reveal workout data"
        context.evaluatePolicy(
            .deviceOwnerAuthenticationWithBiometrics,
            localizedReason: permissionString) { [weak self]
                (success: Bool, error: Error?) in

            guard let authenticationType =
                self?.authenticationType else { return }

            if success == true {
                self?.delegate?.didFinishWithAuthenticationType(
                    authenticationType)
            } else {
                self?.delegate?.didFinishWithError(description:
                    error.debugDescription)
            }
        }
    }
}
```

In addition to making calls to the Security Enclave for you, the `LocalAuthorization` context also provides the initial permission pop-up to enable Face ID for your app and the pop-up that appears when you make the preceding authorization request. Unfortunately, it is up to you, as the developer, to handle the result yourself. One limitation of the view-based implementation is that views exist outside of view controllers, so by default, you will not be able to present any user interface updates outside of the view. To resolve this, you can establish a protocol to pass along information back to the presenting view controller when the security request has completed with success or failure.

In Listing 11-7, I have defined this protocol as `SecurityViewDelegate`, to reflect that classes that implement it are delegates of the protocol. Its methods are `didFinishWithAuthenticationType(type:)`, which is called when the request completes successfully, and `didFinishWithError(description:)`, which is called in case of a failure. This is similar to the design of the `UIImagePickerControllerDelegate` protocol used for the iPhone’s image picker and allows success and failure to be treated as discrete events.

Listing 11-7. Defining a Protocol to Pass Messages from the Security View (`SecurityView.swift`)

```
import UIKit
import LocalAuthentication
...
protocol SecurityViewDelegate {
    func didFinishWithAuthenticationType(_ type: AuthenticationType)
    func didFinishWithError(description: String)
}

class SecurityView: UIView {
    ...
    var delegate: SecurityViewDelegate?
    ...
    @IBAction func validateBiometrics(sender:
        UIButton) {
        ...
        context.evaluatePolicy(
            .deviceOwnerAuthenticationWithBiometrics,
            localizedReason: permissionString) {
```

```

        [weak self] (success: Bool, error:
        Error?) in
            ...
        if success == true {
            self?.delegate?.didFinishWithAuthenticationType(
                authenticationType)
        } else {
            self?.delegate?.didFinishWithError(description:
                error.debugDescription)
        }
    }
}
}

```

Messages are passed back to the presenting view controller via the `delegate` property on the security view. It is defined as an optional value to prevent crashes if the developer did not choose to implement a delegate.

The final steps to complete the authentication process are to now declare the `WorkoutTableViewController` and `WorkoutMapViewViewController` classes as delegates that implement the `SecurityViewDelegate` protocol and to implement the methods that will be called when the success or failure events are triggered. In Listing 11-8, I implemented this by setting the `delegate` property when presenting the security view from the `WorkoutTableViewController` class. For the success event, I dismissed the security view and for the failure event, and I presented a `UIAlertController` with the error description over all other views in the view controller. To implement a smoother transition, such as `IPassword`'s unlock animation, I used the `UIView` class's `animate()` method to animate the alpha level (transparency) of the security view fading to zero.

Listing 11-8. Implementing the `SecurityViewDelegate` Protocol to Receive Messages from the Security View (`WorkoutTableViewController.swift`)

```

class WorkoutTableViewController: UITableViewController {
    ...
    func setupSecurityView() {
        ...
    }
}

```

```

    securityView.delegate = self
    ...
}
}

extension WorkoutTableViewController: SecurityViewDelegate {
    func didFinishWithError(description: String) {
        let alert = UIAlertController(title: "Authentication
            Error", message: description, preferredStyle:
                .alert)
        let alertAction = UIAlertAction(title: "OK", style:
            .default, handler: nil)
        alert.addAction(alertAction)
        present(alert, animated: true)
    }

    func didFinishWithAuthenticationType(_ type:
        AuthenticationType) {
        UIView.animate(withDuration: 0.3, animations: { [weak
            self] in
            DispatchQueue.main.async {
                self?.securityView?.alpha = 0.0
                self?.securityView?.isHidden = true
                self?.securityView?.passwordTextField?.text =
                    nil
                guard let securityView = self?.securityView
                else { return }
                self?.view.sendSubview(toBack: securityView)
            }
        })
    }
}
}

```

When I was implementing this application, I noticed that the `UITableViewController` class expects to be the primary view in the hierarchy and can sometimes display under the views that are presented over it. To alleviate this problem, I modified the `UITableViewDataSource` delegate methods shown in Listing 11-9 to show

data only from HealthKit when the security view is not active. I made this call after every call that affected the presentation state of the security view.

Listing 11-9. Showing or Hiding Table View Data Based on the State of the Security View (`WorkoutTableViewController.swift`)

```
class WorkoutTableViewController: UITableViewController {
    ...
    // MARK: - Table view data source
    ...
    override func tableView(_ tableView: UITableView,
        numberOfRowsInSection section: Int) -> Int {
        guard let securityView = securityView else { return 0 }
        if securityView.isHidden {
            return self.workouts?.count ?? 0
        } else {
            return 0
        }
    }
    ...
    func showSecurityView() {
        if let securityView = self.securityView,
            securityView.isHidden == true {
            tableView.reloadData()
            securityView.alpha = 1.0
            securityView.isHidden = false
            view.bringSubview(toFront: securityView)
        }
    }
}

extension WorkoutTableViewController: SecurityViewDelegate {
    ...
    func didFinishWithAuthenticationType(_ type:
        AuthenticationType) {
        UIView.animate(withDuration: 0.3, animations:
            { [weak self] in
```



```

DispatchQueue.main.async {
    self?.securityView?.alpha = 0.0
    ...
    self?.view.sendSubview(toBack:
securityView)
    self?.tableView.reloadData()
}
})
}
}
}

```

If you attempt to test the application now, the first time you click on the Use Face ID button in either the Workout History or Last Run screens, you will be presented with the system dialog to accept Face ID authentication, then the Face ID scan dialog. After you have validated, the security view will disappear, as shown in Figure 11-9.

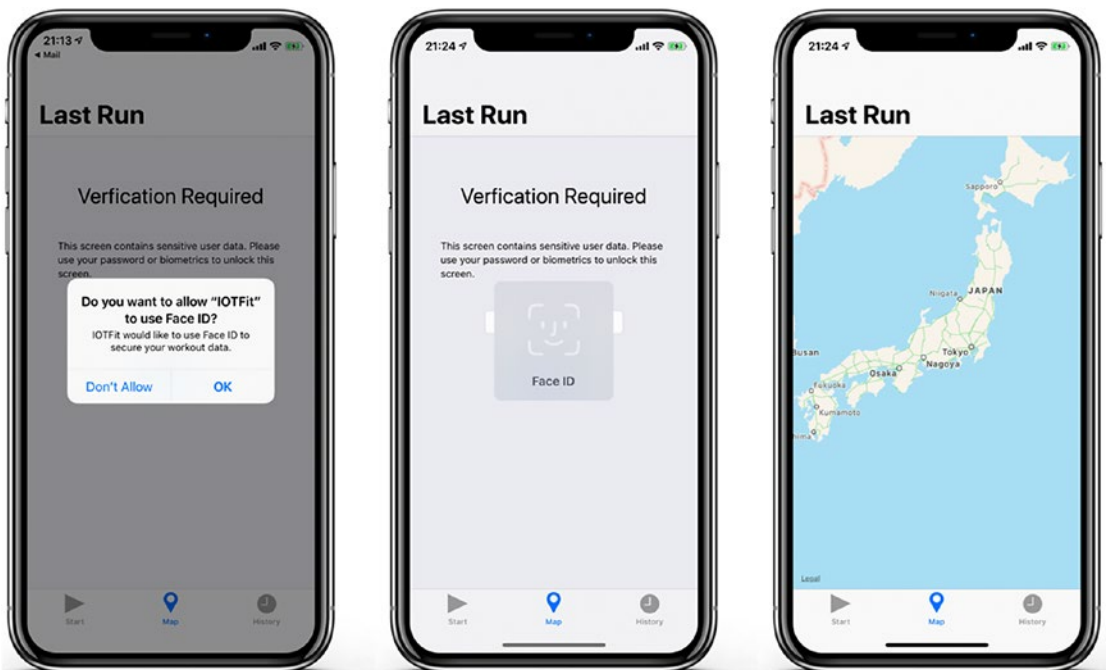


Figure 11-9. *Unlocking the Last Run screen using Face ID*

Using Keychain Services to Secure Data

Now that the app can authorize users via biometrics, you must implement the password field as a backup, in case the user is having trouble with Face ID or Touch ID. As mentioned at the beginning of the chapter, you will store the password in the device's Secure Enclave via Keychain Services. This data is only unencrypted and available when your app is active and in the foreground.

To begin, you must implement the `UITextFieldDelegate` protocol, to handle events from the password text field. Although the protocol defines several events, such as when editing begins or the entered text changes, the event you will want to observe is when the Return key is pressed on the onscreen keyboard. In Listing 11-10, I have implemented this event in the security view, by implementing the `textFieldShouldReturn()` delegate method. Although it is intended for enabling or disabling the Return key, many developers augment this by calling other methods before this one completes. For the IOTFit app, when the Return key is pressed, you should attempt to validate the password and clear the text field.

Listing 11-10. Performing an Action for a Text Field When the Return Key Is Pressed (`SecurityView.swift`)

```
extension SecurityView: UITextFieldDelegate {
    func textFieldShouldReturn(_ textField: UITextField) ->
        Bool {
        textField.resignFirstResponder()
        validatePassword(sender: textField)
        return true
    }
}
```

Next, you must validate the password. Unfortunately, the implementation of the application so far has not prompted the user for an initial password yet nor stored it anywhere. You can perform this operation in a similar manner to accessing values from the `UserDefaults` for your app, namely by checking if a key-value pair exists in the Secure Enclave and setting it if it does not. When using Keychain Services, this is accomplished by attempting to extract values based on a *search query*.

Despite its name, a Keychain Services search query is closer to a predicate than a simple key-value pair lookup. To perform a search query, you must specify the type of value you are trying to interrogate (for example, web site password, general password, SSH secret), the number of matches you want to investigate, whether you want to access the data from the query, and identifying information for the data (such as the app name). For the IOTFit app, the data you will want to store is a general password, identified by the app's name, in place of an account name. In Listing 11-11, I have expanded the `SecurityView` class to make this query, via a `checkPasswordExistence()` method. Because the presenting view controller will have to present the user interface for requesting the password, I have expanded the `SecurityViewDelegate` protocol to include new methods reflecting the password state of the application.

Listing 11-11. Querying If a Password Exists in the Secure Enclave (`SecurityView.swift`)

```
protocol SecurityViewDelegate {
    func didFinishWithAuthenticationType(_ type:
        AuthenticationType)
    func didFinishWithError(description: String)
    func needsInitialPassword()
}

class SecurityView: UIView {
    ...
    let ACCOUNT_NAME: String = "IOTFit"
    ...
    func checkPasswordExistence() {

        guard let accessControl = accessControl else { return }

        let query: [String: Any] = [kSecClass as String:
            kSecClassGenericPassword,
            kSecAttrAccount as String: ACCOUNT_NAME,
            kSecMatchLimit as String: kSecMatchLimitOne,
            kSecReturnAttributes as String: true,
            kSecReturnData as String: true]

```

```

let queryStatus = SecItemCopyMatching(query as
    CFDictionary, nil)

if queryStatus != errSecSuccess {
    delegate?.needsInitialPassword()
} else {
    NSLog("Password has already been set")
}
}
}

```

Although it breaks the pattern of many of Apple's other APIs, the only way to execute the query is by attempting to perform an operation on the Keychain (copy, update, add, or delete). Because the `SecItemCopyMatching(query:result:)` method does not return an error object, you can verify the result of the operation via the `OSStatus` value that is returned after performing the operation. Any value other than `errSecSuccess` indicates that the operation failed. When I was debugging this application, I noticed that searching for the error code in Google was effective in determining the failure reason. Apple uses the `OSStatus` type in both iOS and OS X, so there is a wealth of information on what its possible values represent.

Note The terms *Security Enclave* and *Keychain* are used interchangeably on iOS, as Keychain Services borrows its design from Keychain Services on macOS. In macOS, the secure store is referred to colloquially as *the Keychain*.

Next, you must make the call to check if the passcode exists and prompt the user to enter a password if it does not. It would make sense that before presenting the security view, which contains a password text field, you should check if there is a password to validate against. In my implementation, I performed this logic by adding a call to check the password state, after presenting the security view in the `showSecurityView()` methods in the `WorkoutMapViewController` and `WorkoutTableViewCellController` classes, as shown in Listing 11-12.

Listing 11-12. Checking If a Password Has Been Set When Presenting the Security View (`WorkoutTableViewController.swift`)

```
class WorkoutTableViewController:
    UITableViewController {
    ...
    func showSecurityView() {
        if let securityView = self.securityView,
            securityView.isHidden == true {
            ...
        }
        securityView?.checkPasswordExistence()
    }
}
```

To handle the case in which the password does not exist, you must create a method for saving a string value to the Keychain. As mentioned earlier, all Keychain Services operations must be executed with a query. When it comes to adding a new item to the Keychain, the query is almost exactly like that for looking up a value, except that you need to include the new value as binary data. In Listing 11-13, I have added this logic to the `SecurityView` class via the `savePassword(password:)` method. Pay careful attention to the `kSecValueData` key-value pair and `SecItemAdd(query:result:)`, as they are responsible for implementing the `add` value operation.

Listing 11-13. Saving a Value to the Keychain (`SecurityView.swift`)

```
protocol SecurityViewDelegate {
    ...
    func needsInitialPassword()
    func didSavePassword(success: Bool)
}

class SecurityView: UIView {
    ...
    func savePassword(password: String) {
        guard let passwordData = password.data(using:
            String.Encoding.utf8) else { return }

```

```

let query: [String: Any] = [kSecClass as String:
    kSecClassGenericPassword,
    kSecAttrAccount as String: ACCOUNT_NAME,
    kSecValueData as String: passwordData]

let queryStatus = SecItemAdd(query as CFDictionary,
    nil)

if queryStatus == errSecSuccess {
    delegate?.didSavePassword(success: true)
} else {
    NSLog("Error saving passcode: \(queryStatus)")
    delegate?.didSavePassword(success: false)
}
}
}

```

As with checking the password status, I expanded the `SecurityViewDelegate` protocol to include a method for indicating whether the password was saved successfully. To complete the process of saving the password, implement the `needsInitialPassword()` and `didSavePassword(success:)` methods in the `WorkoutTableViewController` and `WorkoutMapViewController` classes, as shown in Listing 11-14. In my implementation, I chose to present a `UIAlertController` with a text field to accept the new password and logged the result of the operation to the console using `NSLog()`.

Listing 11-14. Prompting for a New Password (`WorkoutTableViewController.swift`)

```

extension WorkoutTableViewController: SecurityViewDelegate {

    func needsInitialPassword() {
        let alert = UIAlertController(title: "Initial
            installation", message: "Please set a passcode for
            your data", preferredStyle: .alert)
        alert.addTextField { (textField: UITextField) in
            textField.placeholder = "Password"
            textField.isSecureTextEntry = true
        }
    }

```

```

let okAction = UIAlertAction(title: "OK", style:
    .default) { [weak self] (action: UIAlertAction) in
    guard let textField = alert.textFields?.first,
        let password = textField.text else { return }
    self?.securityView?.savePassword(password:
        password)
    }
let cancelAction = UIAlertAction(title: "Cancel",
    style: .cancel, handler: nil)
alert.addAction(okAction)
alert.addAction(cancelAction)
present(alert, animated: true)
}

func didSavePassword(success: Bool) {
    NSLog("Password save status: \(success)")
}
...
}

```

For the final step in the password validation process, you must implement the `validatePassword()` method in the `SecurityView` class, which should compare the saved password to the text that was entered into the text field on the security view. The query for extracting the value from the Keychain is exactly like that for detecting the presence of a value; however, after performing the copy operation, you should inspect its result to extract the password. In Listing 11-15, I have implemented this by adding a `getSavedPassword()` method to the `SecurityView` class.

Listing 11-15. Validating a Password Against a Saved Value in the Keychain (`SecurityView.swift`)

```

class SecurityView: UIView {
    ...
    private func getSavedPassword() -> String? {

        let query: [String: Any] = [kSecClass as String:
            kSecClassGenericPassword,
            kSecAttrAccount as String: ACCOUNT_NAME,

```

```

    kSecMatchLimit as String: kSecMatchLimitOne,
    kSecReturnAttributes as String: true,
    kSecReturnData as String: true]

var keychainItemRef: CTypeRef?

let queryStatus = SecItemCopyMatching(query as
    CFDictionary, &keychainItemRef)

guard queryStatus == errSecSuccess,
    let keychainItem = keychainItemRef as? [String:
        Any],
    let passwordData = keychainItem[kSecValueData as
        String] as? Data,
    let password = String(data: passwordData, encoding:
        String.Encoding.utf8)
    else { return nil }
return password
}

@IBAction func validatePassword(sender: UITextField) {
    guard let input = sender.text,
        let savedPassword = getSavedPassword(),
        input == savedPassword else {
        delegate?.didFinishWithError(description:
            "Invalid password")
        return
    }
    delegate?.didFinishWithAuthenticationType(.password)
}
}
}

```

Just as you had to serialize strings to binary data to save them in the keychain, to use stored values for string comparisons, you must reassemble them from binary data, using the `String(data:encoding:)` constructor.

Using Biometrics or an App Password to Lock Keychain Items

As an added bonus security feature, you can lock Keychain items even further, by limiting them to a specific security context. For example, if you were building a password manager, you could use this feature to make a specific set of passwords only available via Face ID or the device's passcode. This is used as a technique to replace building your complete security overlay, as you did in this chapter.

To add this extra layer of security to your Keychain items, you simply have to add a security context and access control settings to each search query. Similar to how the `LocalAuthentication` framework presents the initial Face ID permission prompt for you, it will also present the system's biometrics or password prompts when you try to access values using access control-enabled search queries.

For the extra security settings, you can use the same security context you used through the `SecurityView` class. However, you will have to define a separate access control policy via the `SecAccessControl` class. In Listing 11-16, after initializing the `SecurityView` class, I specified a security policy that will keep the Keychain items available only when the device is unlocked, and the user has validated his/her presence via Face ID or an app-specific password.

Listing 11-16. Adding an Access Control Policy (`SecurityView.swift`)

```
class SecurityView: UIView {
    ...
    private func commonInit() {
        let error: ErrorPointer = nil
        if context.canEvaluatePolicy(
            .deviceOwnerAuthenticationWithBiometrics,
            error: error) {
            ..
        } else {
            NSLog("Biometrics unavailable on device")
            ...
        }
    }
}
```

```

        accessControl = SecAccessControlCreateWithFlags(nil,
            kSecAttrAccessibleWhenUnlocked, .userPresence,
            nil)
    }
}

```

To use the access policy and context to protect the Keychain items, simply modify the copy and add queries from before, to include the context and access control policies for the SecurityView class, as shown in Listing 11-17.

Listing 11-17. Using the Access Control Policy and Context in Keychain Search Queries (SecurityView.swift)

```

class SecurityView: UIView {
    ...
    func checkPasswordExistence() {

        guard let accessControl = accessControl else { return }

        let query: [String: Any] = [kSecClass as
            String: kSecClassGenericPassword,
            kSecAttrAccount as String: ACCOUNT_NAME,
            kSecMatchLimit as String: kSecMatchLimitOne,
            kSecReturnAttributes as String: true,
            kSecReturnData as String: true,
            kSecAttrAccessControl as String: accessControl as Any,
            kSecUseAuthenticationContext as String: context]
        ...
    }

    func savePassword(password: String) {

        guard let accessControl = accessControl,
            let passwordData = password.data(using:
                String.Encoding.utf8) else { return }
    }
}

```

```

let query: [String: Any] = [kSecClass as
    String: kSecClassGenericPassword,
    kSecAttrAccount as String: ACCOUNT_NAME,
    kSecAttrAccessControl as String: accessControl as
        Any,
    kSecUseAuthenticationContext as String: context,
    kSecValueData as String: passwordData]
...
}

private func getSavedPassword() -> String? {
    guard let accessControl = accessControl else {
        return nil
    }

    let query: [String: Any] = [kSecClass as
        String: kSecClassGenericPassword,
        kSecAttrAccount as String: ACCOUNT_NAME,
        kSecMatchLimit as String: kSecMatchLimitOne,
        kSecReturnAttributes as String: true,
        kSecAttrAccessControl as String:
            accessControl as Any,
        kSecUseAuthenticationContext as String:
            context,
        kSecReturnData as String: true]
    ...
    return password
}
}

```

Now, if you load the IOTFit app on your device and attempt to access the Last Run or Workout History tabs, you will be presented with an App Password dialog before the tab loads, as shown in Figure 11-10. After entering the password, you will have a minute or two in which to navigate between the tabs freely, before you are asked to re-validate the app. Although the extra password prompt is a bit excessive for the IOTFit app, it may be helpful to you in your other projects.



Figure 11-10. App password prompt for Access Control-protected Keychain items

Caution After setting a security policy for Keychain items, you will always be prompted for the original security settings whenever you try to access those values again. To reset these values, you will have to add a `delete` operation to your app or reset your device. At the time of writing, Keychain items are retained, even after an app is deleted.

Detecting When an App Returns to the Foreground

For the final security enhancement to the IOTFit app, you will display the security view over the Last Run and Workout History screens when the app returns to the foreground from the background. If you use a password manager on a regular basis, you will recognize this as one of the functions it provides to prevent your information from being stolen after you initially unlock the app.

If you have ever looked into the `AppDelegate.swift` file in any of your projects, you may have noticed `applicationDidEnterBackground()` and `applicationDidEnterForeground()` methods, which handle when your app enters the foreground or background. These are intended to give you an opportunity to start or stop background tasks, such as network calls or database writes, when your app's state changes. One of the ways Apple saves battery power is through a scheduler that only gives apps' background execution times based on when they are used most often. Unfortunately, these times can be unpredictable, and these methods give you a few seconds of execution time to prepare or wind down your app before it gets sent to the background, and all tasks are paused.

For the IOTFit app, however, there are no globally running tasks or objects that you can pause from the app delegate. Instead, you must observe the state changes from the individual view controllers. To implement this, you can use iOS's Notification Center to observe the `UIApplicationWillResignActive` event within the Workout History and Last Run view controllers. When observing notifications, whether they originate from system events, internal messages, or push notifications, you always specify the notification name and a selector (method signature), to handle the notification. In Listing 11-18, I have updated the `WorkoutTableViewController` class to call the `showSecurityView()` method when the background event has been detected.

Listing 11-18. Using a Notification Observer to Detect When the App Is Backgrounded (`WorkoutTableViewController.swift`)

```
class WorkoutTableViewController: UITableViewController {
    ...
    override func viewDidLoad() {
        super.viewDidLoad()
        ...
        setupSecurityView()
    }
}
```

```

let notificationCenter = NotificationCenter.default
notificationCenter.addObserver(self, selector
    #selector(showSecurityView), name:
    Notification.Name.UIApplicationWillResignActive,
    object: nil)
}
}

```

You should call the notification observer only once, as multiple observers will cause the selector to be called multiple times. In my example, I ensured it would be called once by adding the observer to the `viewDidLoad()` method for the view controller.

When you try to compile the app, you will receive a compiler error about the `showSecurityView()` method being unfit as a selector. To fix this, add the `@objc` keyword before the function definition, as shown in Listing 11-19. This is owing to the fact that selectors are a concept ported over from Objective-C, requiring Swift methods to be defined as compatible with Objective-C, in order to be used as selectors. After the modification, the app should now compile successfully.

Listing 11-19. Defining a Method As Compatible with Objective-C
(`WorkoutTableViewCell.swift`)

```

class WorkoutTableViewCell: UITableViewController {
    ...
    @objc func showSecurityView() {
        ...
        securityView?.checkPasswordExistence()
    }
}

```

If you try to run the app on your device now, after unlocking the Last Run or Workout History screens and backgrounding the app, when you reopen the app, the security view will reappear. You have now created one of the most secure workout applications out there. Congratulations!

Summary

In this chapter, you learned how to leverage Face ID, Touch ID, and Keychain Services to secure the sensitive user data in the IOTFit app. Using iOS's LocalAuthentication framework, you were able to detect if biometrics were available to the app, modify the user interface accordingly, and unlock the security view, using the device's biometric sensor. As a backup option, in case biometrics were unavailable or failed, you learned how to store a password for the app in the device's Secure Enclave and how to perform add and lookup operations for this data, using Keychain Services. To go the extra mile, you learned how you could sidestep having to write your own security view, using access control for Keychain Services, and figured out how to lock the screens when the app was backgrounded.

When designing this project, I took inspiration from password managers, because I feel the security user experience they provide is appropriate for any case in which user data must be protected beyond the initial lock screen for the device. My hope is that with the lessons in this chapter, you can now build apps that protect against data theft that results from simply having access to an unlocked device or plugging it into a computer and reading the user data on the device.

Index

A

Adaptive user interface

appearance item

- background color, 31
- constraints and text properties, 28
- editor menu, 33
- embedding navigation controller, 34
- layout, 32
- map view, 37
- navigation controller, 35–36
- text color, 30
- title editing, 35
- view controller creation, 29
- workout time label, 28

auto layout issues

- constraint issues, 38
- context menu, 39
- pop-up, 38
- size inspector, 40
- storyboard, 41

base template

- contextual menu, 20
- editor preview, 22
- refactor menu, 21
- storyboard, 19–20

constraints

- auto layout, 24
- different iPhone sizes, 25
- screenshot, 27

tool adding, 26

workout view controller, 27

devices, 18

features, 17

interface builder device, 17

iPhone X and iPad Pro, 18–19

laying out, 22–23

Apple's bezel-less devices, 4

Apple TV dashboard app, *see* tvOS

dashboard application

Application programming

interfaces (APIs), 7

Arduino-based peripherals

Adafruit HUZZAH32

microcontroller, 165

battery status, 187

Bluetooth chips, 163

door-sensor hardware

IOTHome project, 167–168

part list, 166

GPIO

empty Arduino solution, 185

LED on/off, 186

pin modes, 185

hardware assembling

breadboards, 169

characteristics, 169

circuit, 178

HUZZAH32 chip, 171–173

LED, 176

INDEX

Arduino-based peripherals (*cont.*)

- physical connections, [170](#)
- power/ground connections, [175](#)
- red LED, [177](#)
- schematic diagram, [174](#)
- shared connections, [169](#)

LiPo and ADC Pin 13, [187](#)

objectives, [164](#)

process of, [165](#)

programming environment

- command-line instructions, [182](#)
- connection, [184](#)
- default display, [181](#)
- download, [179](#)
- installation, [180](#)
- software, [179](#)
- target hardware, [182](#)
- USB port, [183](#)

run and monitor solution, [188–189](#)

troubleshooting compilation, [190](#)

B

Bluetooth LE (low energy)

accessory, [255](#), [257](#)

device

- config.json file, [257](#)
- connection process, [279](#)
- discoverAllServicesAnd
Characteristics() method, [279](#)
- flow connection, [281](#)
- HomeBridge, [255](#)
- implementation, [282](#)
- node application, [277](#)
- npm package manager, [276](#)
- peripheral mode, [278](#)
- POST requests, [283](#)

hcitool utility, [258](#)

HomeBridge, [258](#)

Bluetooth LE (low energy) hardware

accessories, [195](#)

advertising packets, [196](#)

Arduino solution, [197](#)

background updates

- background notification, [231](#)
- DoorViewController class, [229](#)
- IOTHome app, [229](#)
- notification permission dialog, [230](#)
- scheduledLocalNotification(), [230](#)
- viewWillAppear() method, [230](#)

communication

- accessory background updates, [214](#)
- BluetoothService class, [217–218](#)
- BluetoothService
Delegate, [216–217](#), [219](#)
- CBCentralManager object, [221](#)
- central manager, [216](#)
- connect() method, [220](#)
- DoorViewController class, [212](#)
- elements, [213](#)
- IOTHome project, [211](#)
- properties and stub methods, [212](#)
- protocol oriented
programming, [217](#)
- steps of, [210](#)
- storyboard and interface
builder connections, [214](#)
- storyboard and view controller, [212](#)
- .xcodeproj file, [215](#)

companion app, [193](#)

data updates

- battery updates, [209](#)
- checkBattery() method, [209](#)
- checkSensor() method, [208](#)

- magnetic sensor updates, 208
 - notify() method, 208
 - setValue() method, 208
- design wireframes, 194
- ESP32_BLE_Arduino library
 - context menu, 199
 - ESP32 BLE library, 200
 - GitHub, 198
 - server features, 201
 - zip archive, 199
- iteration process, 194
- monitoring characteristic updates
 - BluetoothService object, 226
 - DoorViewController class, 228
 - handling method, 227
- objectives, 193
- peripheral device, 196, 201
 - BLEServer class, 204
 - filter outgoing messages, 206
 - GATT UUIDs, 203
 - LightBlue explorer, 207
 - OS X command line, 202
 - setup() method, 205
 - uuidgen tool, 202
- peripherals
 - cancelPeripheralConnection()
 - method, 223
 - characteristics, 225
 - connect() method, 222
 - interrogating services, 225
 - sending updates, 224
 - stop scanning, 222
- role, 196
- scanning, 196
- Bootstraps
 - Etcher, 244
 - hardware interfaces, 248
 - Linux and embedded systems, 241

- microSD card, 245
- pre-built image file, 242
- Raspbian image, 243
- screenshot of, 246
- WiFi network, 247
- zip file, 244

C

- CLLocationManager.authorizationStatus()
 - method, 67
- Control output pins, 184
- Core motion
 - activity-type updates
 - CMMotionActivityManager class, 114
 - currentActivity property, 117
 - OperationQueue object, 114
 - startActivityUpdates()
 - method, 114–115
 - stop pedometer updates, 118
 - toggleWorkout() method, 118
 - updateWorkoutData() method, 117
 - framework, 97
 - handling altimeter updates
 - design pattern, 118
 - screenshot of, 121
 - workout view controller, 118–120
 - motion permission
 - components, 99
 - description, 100
 - IOTFit project, 100, 104
 - CreateWorkoutViewController
 - class, 101
 - step count request, 102–103
 - objectives, 98
 - real-time step count updates
 - closure, 104
 - pedometer, 112

INDEX

Core motion (*cont.*)

- pedometer handling tool, 105

- startPedometerUpdates()
 - method, 112

- stopUpdates() method, 111

- user interface, 108–109

user interface

- auto layout constraints, 109

- UILabel property, 110

- updateWorkoutData() method, 111

- workout view controller, 108–109

CreateWorkoutViewController() class

- class definition, 43

- CLLocationManager property, 64

- contents, 42

- interface builder-compatible, 44

- service querying, 63

- state tracking, 61

D

DHT temperature sensor

- DHT22 sensor, 274

- json() method, 275

- temperature path extension, 273

- web browser, 276

didChangeAuthorizationStatus()

- method, 67, 69

Distributed denial-of-service (DDoS), 407

Dual in-line package (DIP), 169

E

Express project

- app.js, 270

- expose web services, 269

- folder creation, 269

- Hello World verification, 273

- IP address, 272

- listen() method, 271

- node application, 270

- working process, 270

External accessory communication, 53

F

Face ID, Touch ID and Keychain Services

- devices and external clients, 408

- face/fingerprint, 408

foreground

- AppDelegate.swift file, 441

- showSecurityView()

- method, 441–442

- viewDidLoad() method, 442

- WorkoutTableViewController

- class, 441

- key concepts, 408

lock screen user interface

- design wireframes, 412

- NIB (.xib) file, 414

- Objective-C bridging

- header, 414

- password manager app, 410

- screenshot, 417

- security view, 415–416

- SecurityView class, 417

- setupSecurityView() method, 418

- showSecurityView()

- method, 419–420

- UIView subclass, 413

- View NIB, 415

- viewWillAppear() method, 419

- WorkoutMapViewController

- class, 419

- WorkoutTableView

- Controller.swift, 419–420

- message string, 410
- restrict access
 - animate() method, 426
 - biometric authorization, 424
 - run screen, 429
 - security view, 428
 - SecurityViewDelegate protocol, 426
 - table view data, 428
 - UIImagePickerControllerDelegate protocol, 425
- secure data (*see* Keychain Services)
- security context
 - commonInit() method, 422
 - context and authentication type, 421
 - LocalAuthentication framework, 421
 - SecurityView class, 421
 - sensor type, 422
 - viewDidLoad() method, 422
- web server, 408

G

General-purpose input/output (GPIO), 239

H

HealthKit, *see* Workout History table

- history tab, 124
- HKQuantitySample class, 132
- key concepts, 124
- reading workout data
 - conversion logic, 144–146
 - execute() method, 144
 - HKSample object, 143
 - loadWorkoutsFromHealthKit() method, 143
 - table view controller, 146, 148
- represents data, 132

- requesting permission
 - capabilities, 126
 - features, 125
 - HKHealthStore object, 128
 - information property list, 127
 - IOTFit app, 131
 - reading/writing health data, 129
- save data and creation, 133
 - CreateWorkoutView Controller, 134–135
 - HKWorkout object, 136
 - iOS health app, 141–142
 - source code, 137–138
 - step count and flight objects, 139–140
 - steps of, 133
 - workout distance, 140–141
 - WorkoutDataManager classes, 134–135
- writing data, 132

HomeBridge

- apt-get package manager, 250
- Bluetooth LE
 - accessory, 255, 257–258
- configuration files, 260
- experimental configuration, 258
- git pull command, 251
- gpio readall command, 252
- HAP-NodeJS, 248
- installation, 248
- make command, 251
- Node.js applications, 248
- options file, 259
- service definition, 260
- tar command, 250
- temperature sensor, 253–255
- terminal command, 249
- wget command, 249, 251

INDEX

HomeKit

- assigning details, [264](#)
 - bridge, [261](#)
 - configuration, [262](#)
 - device, [264](#)
 - home app, [263](#)
 - Raspberry Pi, [235](#)
 - troubleshooting configuration, [265](#)
- HomeKit Accessory Protocol (HAP), [235](#)
- Hyper Text Transfer protocol (HTTP)
- connections
 - error message, [287](#)
 - Google Chrome, [288](#)
 - https and fs Modules, [285](#)
 - node project, [286](#)
 - options, [284](#)
 - self-signed certificates, [284](#), [290](#)
 - TLS and SSL, [283](#)
 - iSO apps
 - connect() method, [300](#), [302](#)
 - disconnectDoor() method, [308](#)
 - /door/status end point, [306–307](#)
 - error message, [303](#)
 - Info.plist file, [303–304](#)
 - initial implementation, [297](#)
 - network manager method, [299](#)
 - singletons, [298](#)
 - URLSessionDelegate, [305](#)

I, J

- Integrated circuits (ICs), [169](#)
- Internet Engineering Task
 - Force (IETF), [284](#)
- iOS (iPhone OS) application, [292](#)
 - HTTPS requests, [297](#)
 - connect() method, [300](#), [302](#)

- disconnectDoor() method, [308](#)
 - door/status, [306–307](#)
 - error message, [303](#)
 - Info.plist file, [303–304](#)
 - initial implementation, [297](#)
 - network manager method, [299](#)
 - singletons, [298](#)
 - URLSessionDelegate file, [304–305](#)
 - viewWillDisappear() method, [308](#)
- tvOS dashboard app, [314](#)
- user interface, [292](#)
- DoorViewController class, [293](#)
 - elements, [294](#)
 - home view controller, [297](#)
 - HomeViewController class, [296](#)
 - storyboard file, [294](#)
 - wireframes, [293](#)
- IOTFit application
- adaptive user interface (*see* Adaptive user interface)
 - Apple developer account-Xcode sign-in prompt, [15–16](#)
 - team selection, [14](#)
 - Xcode installation, [13–14](#)
- default project, [13](#)
- development, [12](#)
- features, [7](#)
- initial options (project name), [9–10](#)
- iOS APIs, [7](#)
- objectives, [4](#)
- storyboard (*see* Storyboard)
- tabbed app template, [9](#)
- wireframe diagrams, [6](#)
- Xcode
- editor window, [12](#)
 - iPhone testing, [50](#)
 - welcome screen, [8](#)

K

Keychain Services

- biometrics/app password
 - access control policy, 437
 - app password prompt, 440
 - search queries, 438
 - security policy, 440
- checkPasswordExistence()
 - method, 431
- getSavedPassword() method, 435
- NSLog() method, 434
- password exists, 431
- return key, 430
- savePassword() method, 433
- search query, 431
- SecurityViewDelegate protocol, 431
- textFieldShouldReturn() method, 430
- UITextFieldDelegate protocol, 430
- validatePassword() method, 435
- WorkoutTableViewCell
 - class, 432

L

Lithium polymer (LiPo) battery, 187

loadWorkoutsFromHealthKit()

- method, 128, 143

Location-based functions

- background modes
 - capabilities tab, 54
 - editor view, 54
 - Info tab, 56
 - iOS apps, 53
 - key-value pair, 57
 - location updates, 55
 - permission prompts, 57–58

handling updates, 72

- app changes state, 74–76
- background updates, 80
- timer class, 76
- pauseWorkout() method, 79
- request and delegate
 - handler method, 73
- startWorkout() method, 78
- timer class, 77–78
- updateUserInterface(), 74
- updateWorkoutData() method, 78
- workout distance, 79

location permission

- authorization states, 66
 - authorization status, 67–69
 - CLLocationManager property, 64
 - CLLocationManagerDelegate
 - protocol, 65
 - create workout view controller, 59
 - CreateWorkoutViewController
 - class, 60
 - didChangeAuthorizationStatus()
 - method, 69
 - flowchart, 60
 - hardware availability, 62
 - IOTFit, 70
 - presentPermissionErrorAlert()
 - method, 70
 - request flow, 72
 - requestLocationPermission()
 - method, 69
 - service querying, 63
 - state tracking, 61–62
 - user interface, 58–59
 - user page, 70
- map (*see* Map location)
- objectives, 52

M

Map location

- CLLocation class, [81](#)
 - codable protocol
 - compatible data types, [85](#)
 - file-based data storage, [82](#)
 - file templates, [84](#)
 - I/O implementation, [86–87, 89](#)
 - loadFromPlist() method, [87](#)
 - map() method-serialize data, [89](#)
 - property-list (.plist) file, [82](#)
 - PropertyListDecoder method, [87](#)
 - saveToList() method, [88](#)
 - saveWorkout() and
 - getLastWorkout() methods, [92](#)
 - WorkoutDataManager
 - class, [82, 84, 91](#)
 - Xcode creation, [83](#)
 - generate and displays
 - annotations, [93, 95](#)
 - IOTFit app, [96](#)
- Monitor input pins, [184](#)

N

Node Package Manager (NPM), [252](#)

O

- OpenWeatherMap API, [339](#)
 - account creation, [340](#)
 - compactMap() method, [344](#)
 - forecast method, [347](#)
 - getOutdoorTemperature()
 - method, [344–346](#)
 - JSON response, [341](#)

- keys page, [340](#)
- NetworkManager class, [342–344](#)
- processing request, [350](#)
- response, [348](#)
- view controller, [349](#)

P, Q

- pauseWorkout() methods, [79](#)
- presentPermissionErrorAlert()
 - method, [70](#)
- presentRecordInterface() method, [385](#)

R

Raspberry Pi

- hardware component
 - bootstraps, [241](#)
 - circuit, [239, 241](#)
 - HomeBridge, [248](#)
 - HomeKit bridge, [261](#)
 - IOTHome project, [239](#)
 - requirements, [237](#)
 - HomeKit, [235](#)
 - key concepts, [236](#)
 - objectives, [236](#)
 - server configuration
 - service definition, [291](#)
 - systemctl tool, [292](#)
 - systemd tool, [291](#)
 - single-board computer, [237](#)
 - web server, [268](#)
- RecordInterfaceController.swift, [378](#)
- requestLocationPermission()
 - method, [69, 72, 80](#)
 - resetWorkoutData() method, [105](#)

S

- saveWorkoutToHealthKit()
 - method, 128–129
- Secure Sockets Layer (SSL), 283
- Single-board computer, 237
- Siri Remote
 - addGestureRecognizer()
 - method, 352
 - allowedPressTypes() method, 352
 - fetchNetworkData() method, 352
 - handling touch input, 351
 - onscreen remote, 353
 - play/pause button, 352
 - touchpad, 354
 - UITapGestureRecognizer class, 352
- startPedometerUpdates()
 - method, 105
- startWorkout() methods, 72
- Storyboard
 - connection inspector
 - CreateWorkoutViewController
 - class, 46
 - Main.storyboard file, 45
 - pauseWorkout() method, 49
 - pause workout button, 47–48
 - property name, 47
 - testing, 50
 - user interface event, 49
 - interface builder-compatible
 - CreateWorkoutViewController
 - class, 42–44
 - MapKit framework and map
 - view property, 45
 - properties and methods, 41–42
- System-on-a-chip (SoC)
 - solutions, 407

T

- Table View Controller
 - cocoa touch class, 147
 - UITableViewController class, 146, 148
 - UITableViewDataSource
 - protocol, 156–157
 - UITableViewDelegate protocol, 158
 - user interface
 - context menu, 150
 - delegate outlets, 155
 - identifier, 156
 - ownership, 154
 - screenshot, 153
 - storyboard, 149–150
 - subtitle view, 152
 - tab bar item, 152
 - WorkoutTableViewController
 - class, 148
- toggleWorkout() method, 60, 64, 79
- Transport layer security (TLS), 283
- tvOS dashboard application
 - Apple TV and debug
 - confirmation of, 357
 - debugging target, 358
 - device setup process, 357
 - pair button, 356
 - remote app and devices, 354–355
 - data sources
 - fetchNetworkData() method, 334
 - NetworkManager class, 332–333
 - OpenWeatherMap API, 339
 - request location, 335
 - viewDidLoad() method, 334
 - Info.plist file, 317–318
 - key concepts, 314
 - scheme and files, 317

INDEX

tvOS dashboard application (*cont.*)

Siri Remote, [351](#)

source code, [319](#)

target, [315](#)

template picker window, [315](#)

user interface

blank storyboard, [321](#)

creation, [319](#)

design language, [324](#)

elements, [322](#)

font-based graphics, [328](#)

source file, [323](#)

ViewController class, [322](#)

wireframe of, [320](#)

U

UITableViewController class, [146](#)

UITableViewDataSource protocol

methods, [156](#)

UITableViewDelegate protocol, [158](#)

Universally unique identifiers (UUIDs), [197](#)

updateUserInterface() method, [74](#)

updateWorkoutData() method, [111](#)

User interface

blank storyboard, [321](#)

creation, [319](#)

design language, [324](#)

applyEffects() method, [325](#)

applyEffects() method, [324](#)

screenshot, [327](#)

shadow effect code, [326](#)

UIVisualEffectView class, [324](#)

viewDidLoad() method, [325](#)

elements, [322](#)

font-based graphics, [328](#)

FontAwesome.swift, [329](#)

Info.plist file, [328](#)

project navigator, [330](#)

screenshot, [331](#)

ViewController class, [330](#)

source file, [323](#)

ViewController class, [322](#)

watchOS, [368](#)

context menu, [384](#)

controller, [378](#)

debugging menu, [387](#)

default stack view

configuration, [373](#)

design wireframes, [368](#)

dismissViewController()

method, [387](#)

elements, [380](#)

file template window, [377](#)

force touch, [383](#)

InterfaceController.swift, [380](#), [384](#)

object library, [371](#)

presentRecordInterface()

method, [385](#)

RecordInterfaceController.swift,

[378](#), [388](#)

record workout view controller, [370](#)

storyboard arrangement, [369](#), [376](#)

sub-view arrangement, [376](#)

table row, [372–373](#)

text size and color, [375](#)

vertical layout, [374](#)

WKInterfaceTable class, [380](#)

WorkoutRowController.swift, [381](#)

wireframe of, [320](#)

V

viewWillAppear() method, [157](#)

W, X, Y, Z

watchOS application

- Apple Watch, 359

- core location and motion, 388

- CLLocationManagerDelegate

- protocol, 398

- hardware manager, 394, 396–397

- RecordInterfaceController class, 391

- stand-alone features, 388

- testing, 399

- toggleWorkout() method, 392–393

- updateUserInterface() method, 390

- WorkoutDataManager class, 389–390

IOTFit project

- app store description page, 361–362

- background modes, 367

- configuration, 364

- fitness app, 361

- Info.plist file, 366

- IOTFitWatch and IOTFitWatch

- extension, 365

- scheme activation, 365

- WatchKit app template, 363

- key concepts, 360

- memory and processing power, 361

- stripped-down development, 360

- user interface

- default stack view configuration, 373

- design wireframes, 368

- elements, 379

- file template window, 377

- force touch, 383

- image view, 372

- object library, 371

- RecordInterfaceController

- class, 378–379

- RecordInterfaceController.swift, 378

- record workout view controller, 370

- stack view, 372

- storyboard arrangement, 369, 376

- sub-view arrangement, 376

- table row, 371

- text size and color, 375

- vertical layout, 374

- WKInterfaceTable class, 380

Workout History table

- InterfaceController.swift, 401–402

- IOTHomeTV app, 403

- loadWorkoutsFromHealthKit()

- method, 401

- refreshTable() method, 402–403

- screenshot, 405

Web server

- iOS app, 292

- HTTPS requests, 297

- user interface, 292

- key concepts, 268

- objectives, 268

- share data (HTTPS)

- Bluetooth devices, 276

- DHT temperature sensor, 273

- express project, 269

- HTTP connections, 283

- IOTHome sensors, 269

- Raspberry Pi, 290

- willActivate() method *vs.*

- viewWillAppear() method, 390

- Wireframes *vs.* mockups, 7

- WorkoutDataManager class, 128

Workout History table

- InterfaceController.swift, 401–402

- IOTHomeTV app, 403

- loadWorkoutsFromHealthKit()

- method, 401

- refreshTable() method, 402–403

- screenshot, 405

- WorkoutMapViewViewController class, 45