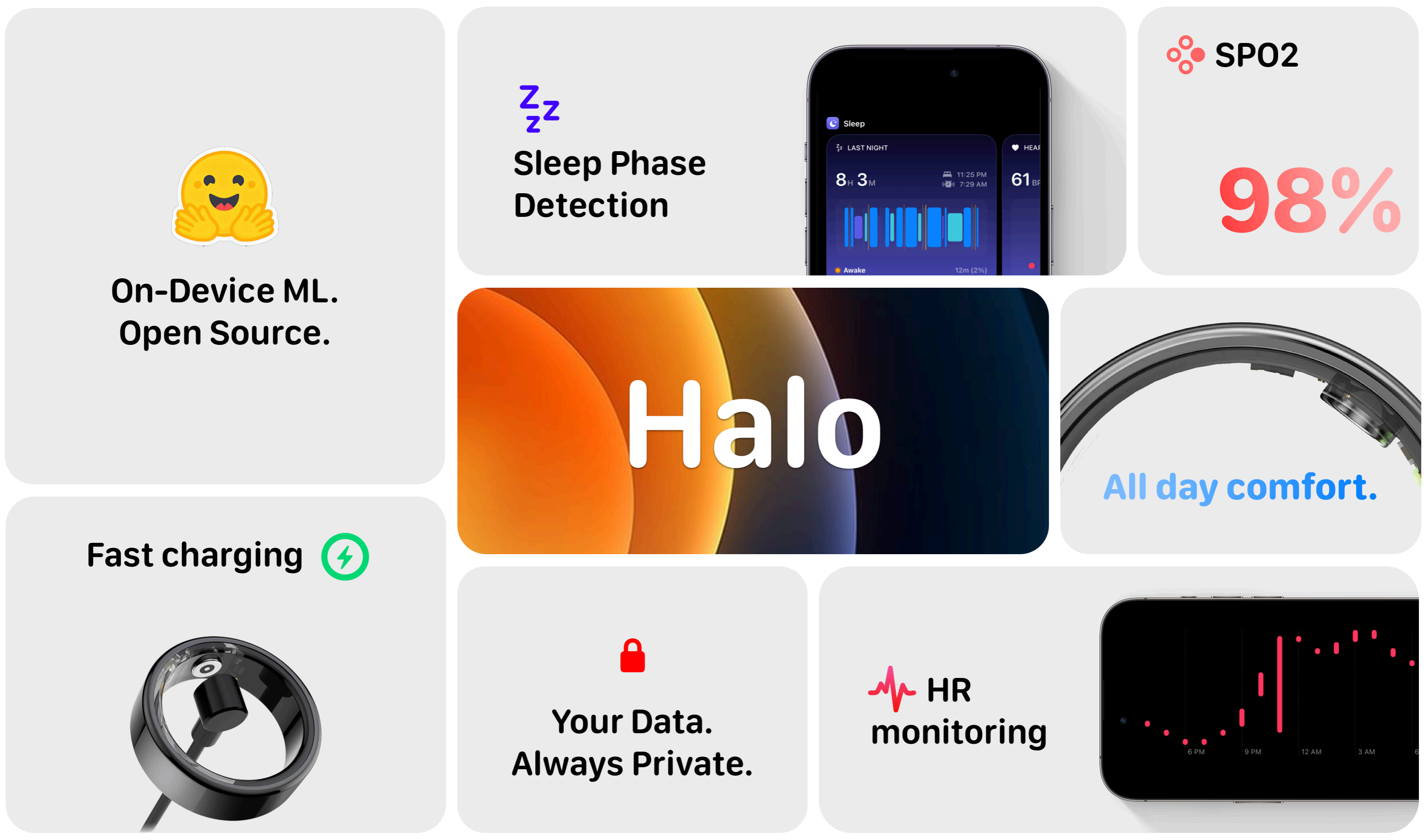
Introduction



In the rapidly evolving landscape of wearable technology, we find ourselves at a crossroads. The market is flooded with sleek, feature-packed devices promising to revolutionize our approach to health and fitness. Yet, beneath the polished exteriors and marketing hype lies a troubling reality: most of these devices are black boxes, their inner workings shrouded in proprietary code and closed-source hardware. As consumers, we're left in the dark about how our intimate health data is collected, processed, and potentially shared.

Enter **Halo**, an open-source alternative that aims to democratize health tracking. This series of articles will serve as your entry-level guide to building and using a fully transparent, customizable wearable device.

It's important to note that Halo is not intended to compete with consumer-grade wearables in terms of polish or feature completeness. Instead, it offers a unique, hands-on approach to understanding the technology behind health tracking devices.

We'll be using `Swift 5` to build the accompanying iOS interface and `Python >= 3.10`. Since the code for this project is 100% open-source, please don't hesitate to submit pull requests, or fork the project to take it in a whole new direction.

What You'll Need

- Physical access to the COLMI R02 which you can grab for $11-$30 at the time of writing.
- A development environment with Xcode 16 installed, and an optional membership to the Apple Developer Program
- `Python >= 3.10` with `pandas`, `numpy`, `torch` and of course, `transformers` installed.

Acknowledgements

This project is built on code and my learnings from the following Python repository.

Disclaimer

As a doctor first, I'm legally obligated to remind you: none of what you're about to read is medical advice. Now, let's go make some wearables beep!

Chapter 1: Pairing the Ring

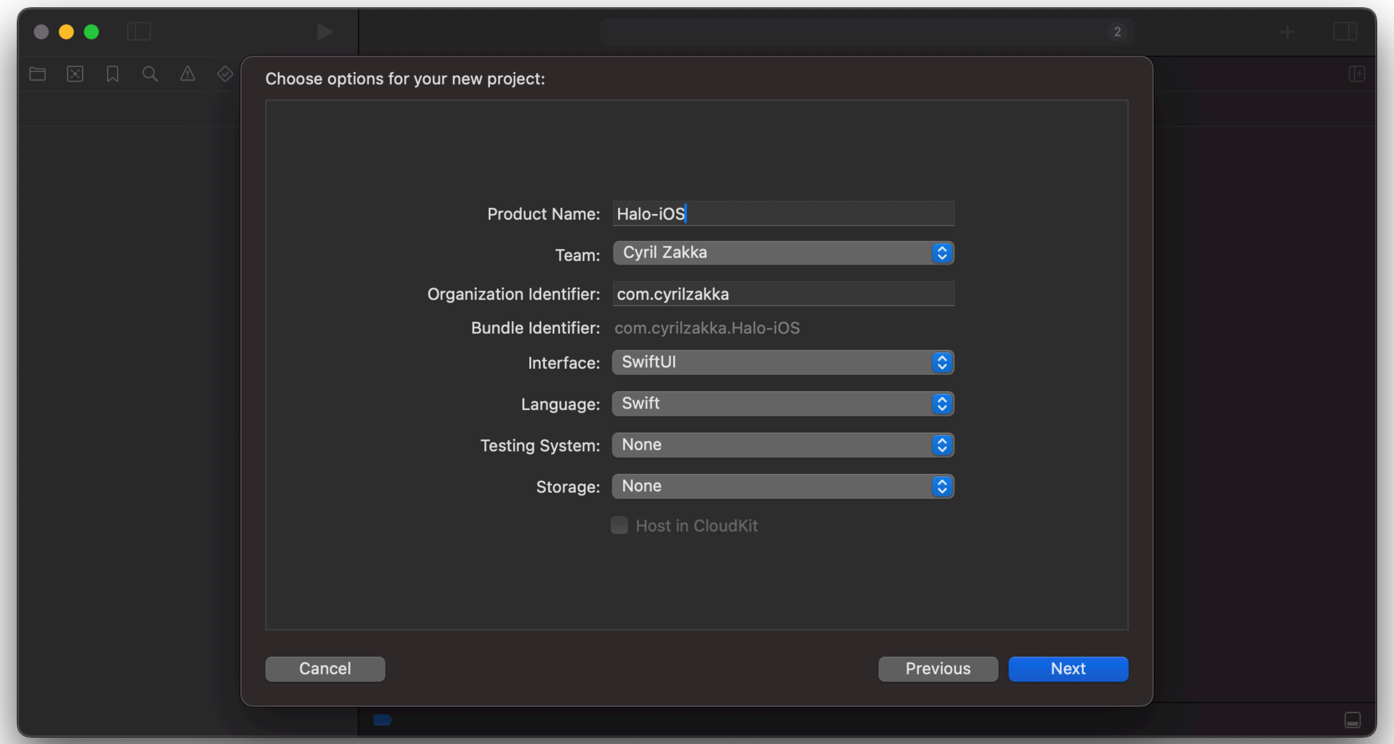
Before we dive into the code, let's understand some key specifications of Bluetooth Low Energy (BLE) first. BLE operates on a simple client-server model using three key concepts: *Centrals*, *Services*, and *Characteristics*. Let's break these down:

- A **Central device** (like your iPhone) initiates and manages connections with a **Peripheral device** (like our COLMI R02 ring). The ring broadcasts its presence, waiting for a phone to connect to it. Only one phone can connect to the ring at a time.
- **Services** are collections of related features on the ring. Think of them as categories - one service might handle heart rate monitoring, another handles battery status. Each service has a unique identifier (UUID) that the client uses to find it.
- **Characteristics** are the actual data points or control mechanisms within each service. They can be read-only (like getting sensor data), write-only (like sending commands), or both. Some characteristics can also notify your phone automatically when their values change, which is crucial for real-time health monitoring.

When your phone connects to the ring, it locates the services it needs and then interacts with specific characteristics to send commands or receive data. This structured approach ensures efficient communication while maintaining long battery life. With these details out of the way, let's get to building!

Setting up the Xcode Project

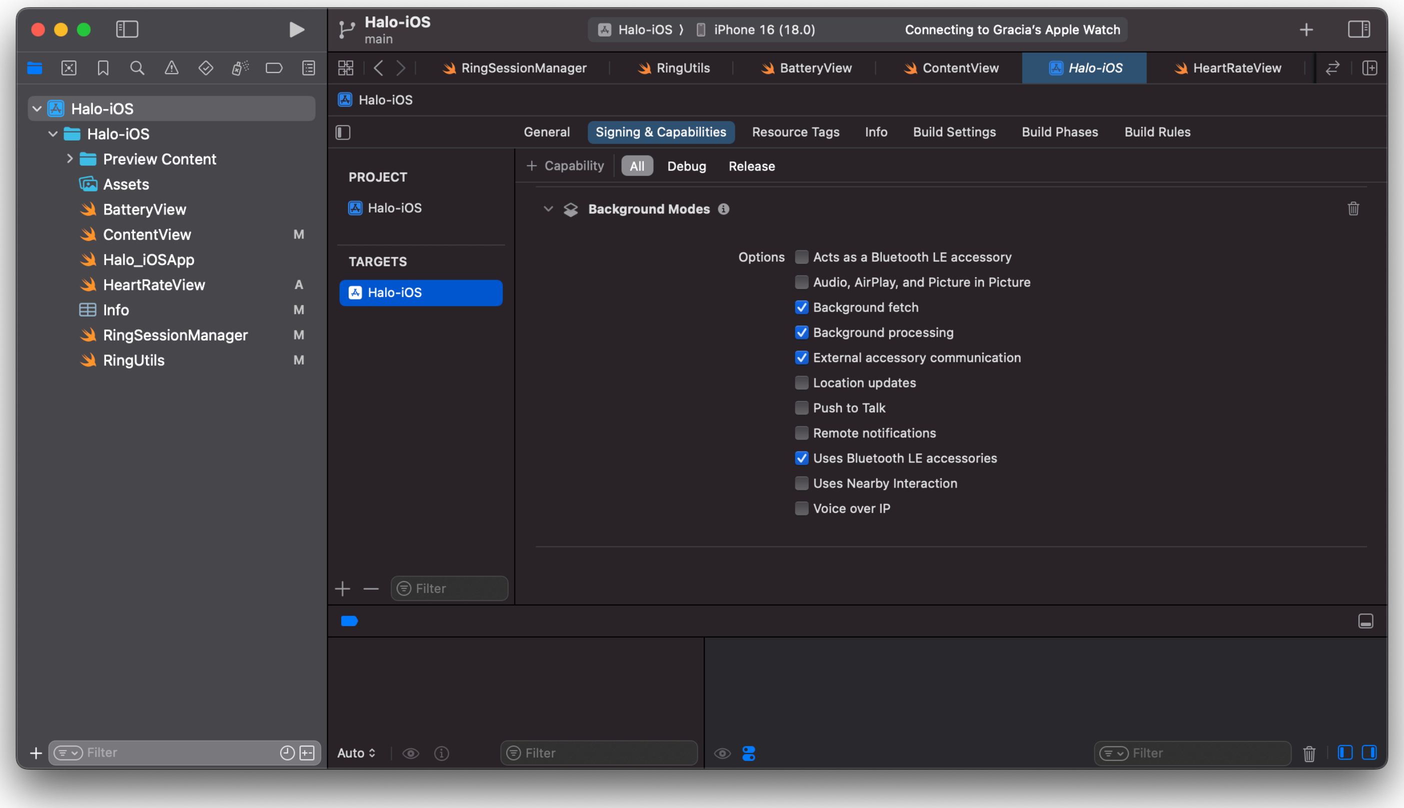
Let's create a new project called `Halo` targeting `iOS`. For the organization identifier, it's customary to use a reverse domain name (like `com.example`). We'll use `com.FirstNameLastName` for this project.


Choose options for your new project:

Product Name:	Halo-iOS
Team:	Cyril Zakka
Organization Identifier:	com.cyrilzakka
Bundle Identifier:	com.cyrilzakka.Halo-iOS
Interface:	SwiftUI
Language:	Swift
Testing System:	None
Storage:	None
	☐ Host in CloudKit

.

We now need to enable specific capabilities for our app. Go to the `Signing & Capabilities` tab in Xcode and enable the following `Background Modes`:



Background Modes

Options
- [ ] Acts as a Bluetooth LE accessory
- [ ] Audio, AirPlay, and Picture in Picture
- [x] Background fetch
- [x] Background processing
- [x] External accessory communication
- [ ] Location updates
- [ ] Push to Talk
- [ ] Remote notifications
- [x] Uses Bluetooth LE accessories
- [ ] Uses Nearby Interaction
- [ ] Voice over IP

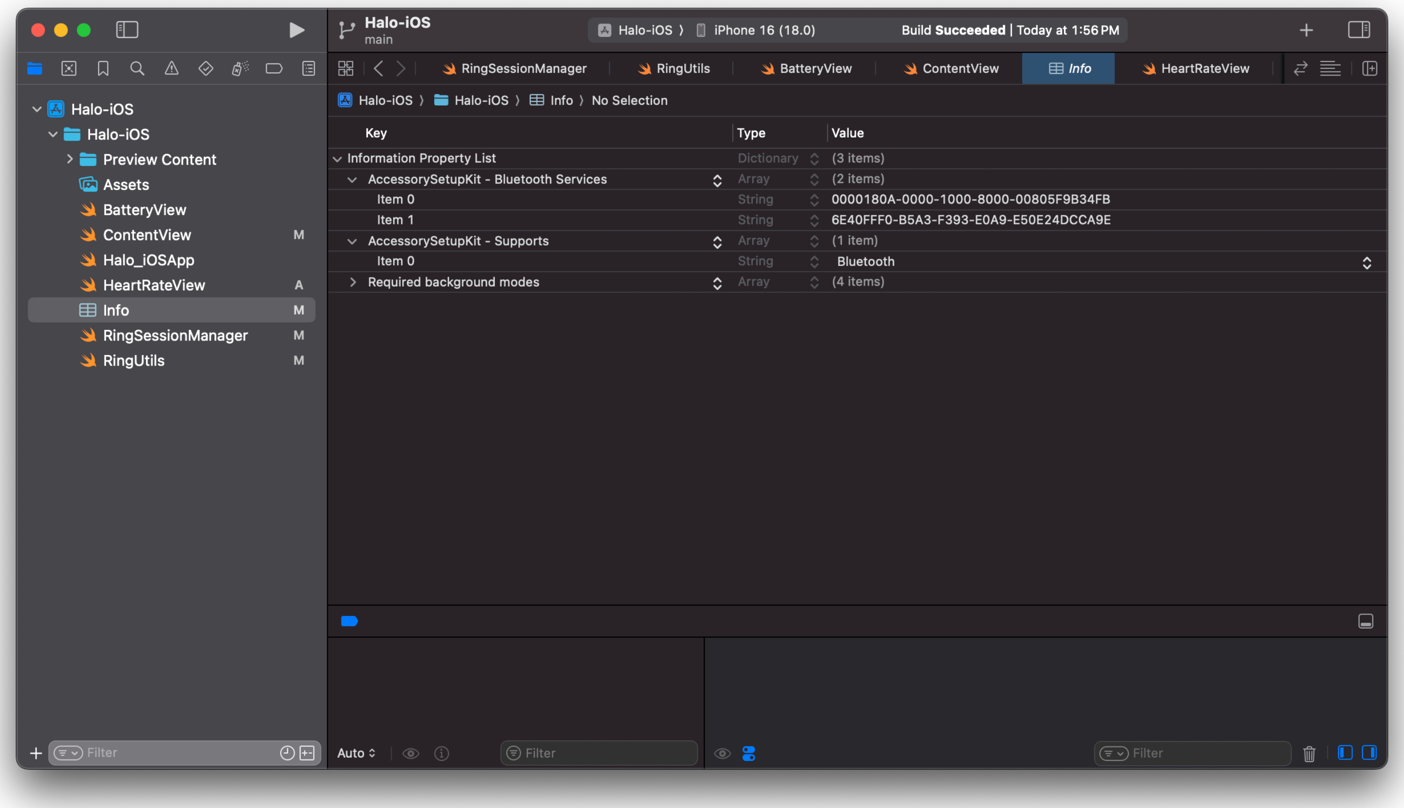
These settings ensure your app can maintain connections with the ring and process data even when it's not in the foreground.

In the next step, we'll make use of a framework called `AccessorySetupKit` - Apple's latest framework for connecting Bluetooth and Wi-Fi accessories to iOS apps. Released with iOS 18, it replaces the traditional method of requesting broad Bluetooth permissions with a more focused approach - your app only gets access to specific devices that users explicitly approve.

When a user wants to connect their COLMI R02 ring to our app, `AccessorySetupKit` presents a system interface showing only compatible nearby devices. Once selected, our app can communicate with the ring without requiring full Bluetooth access to the user's device. This means better privacy for users and simpler connection management for developers. Let's walk through setting up `AccessorySetupKit` for our ring.

First open up `Info.plist` which you can either find in the left sidebar, or by navigating to `Project Navigator (⌘1) > Your Target > Info`. Now enter the following key-value entries to work with our COLMI R02 ring:

- Add `NSAccessorySetupKitSupports` as an `Array` type and Insert `Bluetooth` as its first item
- Add `NSAccessorySetupBluetoothServices` as an `Array` type and these UUIDs as `String` items `6E40FFF0-B5A3-F393-E0A9-E50E24DCCA9E` and `0000180A-0000-1000-8000-00805F9B34FB`.



Key	Type	Value
Information Property List	Dictionary	(3 items)
AccessorySetupKit - Bluetooth Services	Array	(2 items)
Item 0	String	0000180A-0000-1000-8000-00805F9B34FB
Item 1	String	6E40FFF0-B5A3-F393-E0A9-E50E24DCCA9E
AccessorySetupKit - Supports	Array	(1 item)
Item 0	String	Bluetooth
Required background modes	Array	(4 items)

We should now be good to go! 🤗

If you prefer to skip the setup and jump straight to the code, you can find the full code below.

Ring Session Manager

First, we'll create a `RingSessionManager` class that handles all the ring communication. This class will be responsible for:

- Scanning for nearby rings
- Connecting to a ring
- Discovering services and characteristics
- Reading and writing data to the ring

Step 1: Create the RingSessionManager

Create a new Swift (⌘N) file named `RingSessionManager.swift`. I've highlighted the key properties and methods you'll need to implement below. You'll find the full class at the end of this section. Let's start by defining the class and its properties:

▶ Click to expand

Step 2: Discovering the Ring

The ring broadcasts itself using specific Bluetooth services. We need to tell iOS what to look for. We'll create a `ASDiscoveryDescriptor` object with the ring's Bluetooth service UUID. This descriptor will help `AccessorySetupKit` identify the ring when scanning for nearby devices:

▶ Click to expand

You can replace the `UIImage(named: "colmi")!` with your ring's image. Make sure to add the image to your project's asset catalog. I used the following product image.

Step 3: Showing the Ring Picker

When the user wants to connect their ring, we show Apple's built-in device picker:

▶ Click to expand

Step 4: Handling Ring Selection

When the user picks their ring from the list, we need to handle the connection:

▶ Click to expand

Step 5: Establishing the Connection

Once we have a ring selected, we initiate the actual Bluetooth connection:

▶ Click to expand

Step 6: Understanding the Delegate Methods

Our `RingSessionManager` implements two crucial delegate protocols that handle Bluetooth communication. Let's explore what each delegate method does:

The Central Manager Delegate

First, we implement `CBCentralManagerDelegate` to handle the overall Bluetooth connection state:

▶ Click to expand

This method gets called whenever the Bluetooth state changes on the device. When Bluetooth powers on, we check if we have a previously connected ring and try to reconnect to it. When we successfully connect to a ring, this method gets called:

▶ Click to expand

And when the ring disconnects (either intentionally or by going out of range):

▶ Click to expand

The Peripheral Delegate

The `CBPeripheralDelegate` methods handle the actual communication with our ring. First, we discover the ring's services:

▶ Click to expand

Once we find the services, we need to discover their characteristics - these are the actual data points we can read from or write to:

▶ Click to expand

When we receive data from the ring, this method gets called:

▶ Click to expand

And finally, when we send commands to the ring, this callback confirms if they were received:

▶ Click to expand

Full Code

The full `RingSessionManager` class should now look like this:

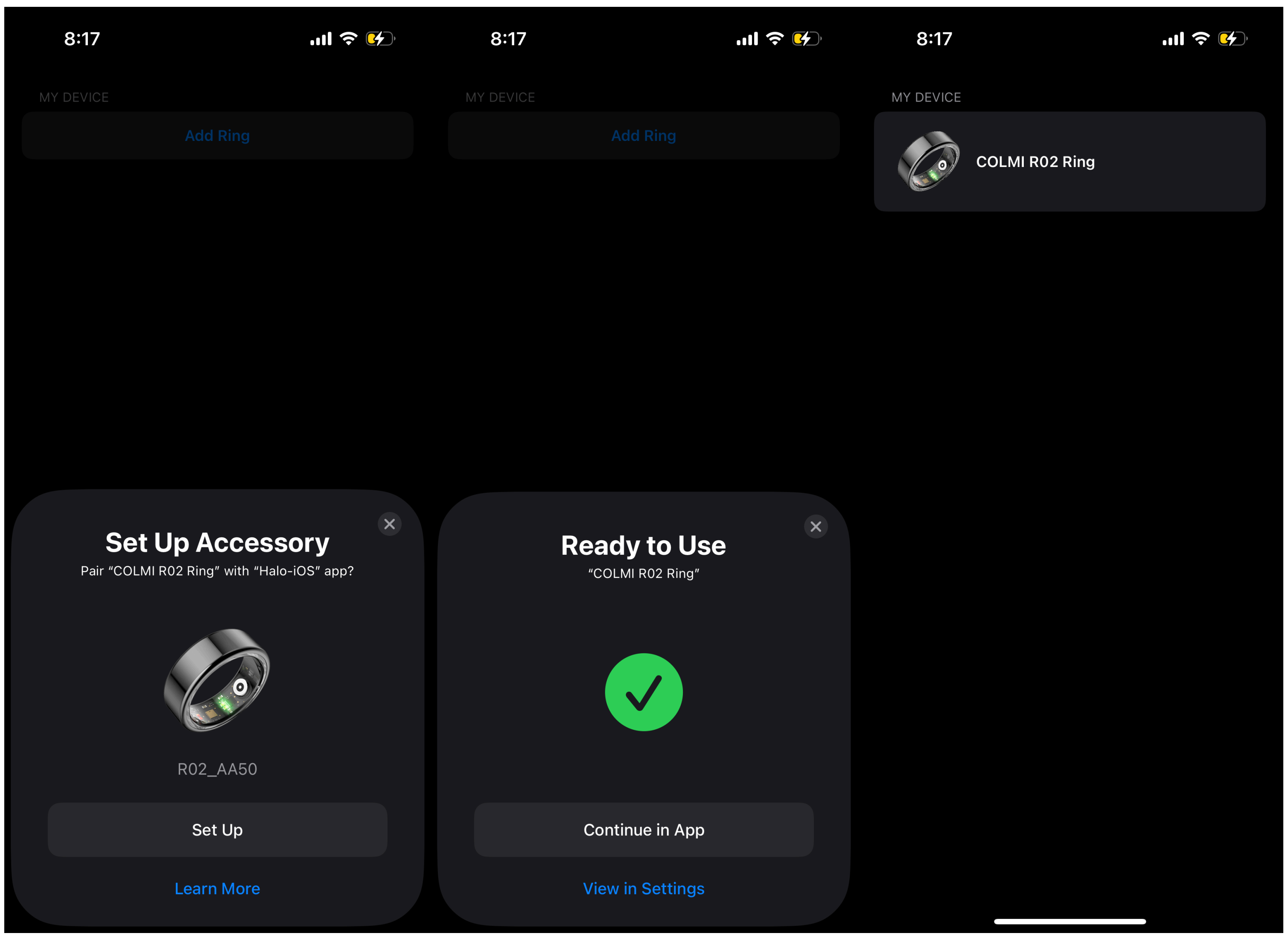
▶ Full Code

Step 7: Making it Work in Our App

Open up `ContentView.swift` and paste the following. Everything should now be in place!

▶ Click to expand

If all goes well, you should now be able to build and run your app. When you tap the `Add Ring` button, you'll see a pop-up of nearby devices including your COLMI R02 ring. Select it and the app will connect to it. 🎉



In the next chapter, we'll explore how to read and write data to the ring, starting with battery level, and working our way to raw sensor data (photoplethysmogram, accelerometer, etc). We'll then use this data to build features like real-time heart rate monitoring, activity tracking, sleep phase detection and more. Stay tuned!