Private CloudKit Sharing for Heart Rate Data

Daniel Zhang

Contents

An app without data is like a car without fuel or power. It won't go anywhere. Let's talk about some of our most pertinent data, our heart rate. We have a unified data store thanks to gadgets such as the Apple Watch, Oura ring, or other health trackers syncing to the Health app on an iPhone. Moreover, if we own multiple Apple computing devices, anyone? Sharing and accessing our health data across all of them is possible with CloudKit, which is compatible with iOS, watchOS, and macOS.

The latest updates to Swift offer numerous options for working with data, especially asynchronous data streams. I'll discuss a couple topics:

- Transforming legacy completion handlers into Publishers using Futures and Subjects in Combine, and AsyncStreams in Swift Concurrency.
- Setting up Core Data for durable storage and sharing data between devices using CloudKit.

```swift
import HealthKit

let healthStore = HKHealthStore()
let heartRateType = HKQuantityType.quantityType(forIdentifier: .heartRate)!

func fetchData(completion: @escaping () -> Void)
{
    healthStore.requestAuthorization(toShare: nil, read: Setting.healthReadTypes)
    { (success, error) in // ((Bool, Error?)) -> Void
        if success
        {
            let calendar = Calendar.current
            let now = Date()
            let startOfDay = calendar
                .startOfDay(for: Date().daysBefore(numberOfDays: 3))
            let predicate = HKQuery.predicateForSamples(
                withStart: startOfDay,
                end: now,
                options: .strictStartDate
            )
            let sortDescriptor = NSSortDescriptor(
                key: HKSampleSortIdentifierStartDate,
                ascending: false
            )
            let query = HKSampleQuery(
                sampleType: self.heartRateType,
                predicate: predicate,
                limit: HKObjectQueryNoLimit,
                sortDescriptors: [sortDescriptor]
            )
            { _, results, _ in // (Void) -> (HKQuery, [HKSample]?, NSError)
                self.handleSamples(results, completion: completion)
            }
            self.healthStore.execute(query)
        } else
        {
```

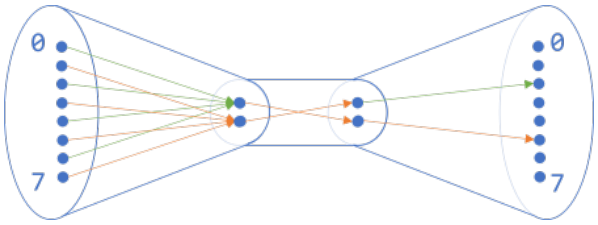

Figure 1:

```
            print("Failed to request authorization to access heart rate data.")
        }
    }
}
```

1 Combine versus Swift Concurrency

It's common for individuals to believe that newer things are automatically better. Take for instance, the trend of people gravitating towards async/await simply because it's newer than Combine. However, Combine is actually a higher level of abstraction than async/await and it's not going out of style anytime soon. In fact, Combine is built on top of async/await. Swift Concurrency and Combine also work well in tandem, and can be used to enhance older APIs that involve completion handlers. We'll explore more on how they intersect later on.

1.1 What's not to like about monads?

When programming, people often prefer to work with a higher level of abstraction, which is where Combine comes in handy. However, it's important to note that Combine is an exception due to its use of monads. If you're not familiar with the term, it might be helpful to first understand what isn't a monad. For example, a completion handler isn't one and cannot be chained like a Publisher and instead requires a nested structure of callbacks to continue executing tasks.

I have an interesting concept to share with you. Imagine a completion handler

```
healthStore.authorizePublisher().flatMap // (Bool) -> Publisher
{ _ in
    healthStore.queryPublisher() // (HKQuantitySample) -> Publisher
}.sink(receiveCompletion:
    { completion in
        switch completion
        {
        case .failure(let error):
            XCTFail("🚨 \(error)")
            expect.fulfill()
        case .finished:
            print("🏁 Complete")
            expect.fulfill()
        }
    }, receiveValue:
    { value in
        let sample = value.toSample()
        do { try sample.saveToCoreData(managedContext: context) }
        catch { XCTFail(error.localizedDescription) }
    }).store(in: &cancellables)
```

Figure 2:

as a bag, but inverted so that it's turned inside out. Now, picture this inverted bag having a surface that can be linked to the inner dimension of another bag. That's essentially what a Publisher is. It opens and exposes structure to enable piping between domains while keeping the details hidden away on the other side. If this is still confusing, don't worry. Understanding monads can take some time.

1.2 Linking together Publishers

When retrieving heart rate data, completion handlers are commonly used. However, instead of using them, we can represent the operations with a Publisher. They can be readily chained, allowing for a simplified, linear ordering of operations with less code. Although error handling and completion require more boilerplate, the logic is organized neatly away from the surface, inside-out. By mapping completion handlers to Publishers, we can simplify our process and effectively utilize Combine.

Heart rate data can be effectively modeled using reactive streams, which consist of a series of time-stamped data points. With the ability to capture high-quality data,

including heart rate variability, watches can provide useful insights into stress levels and other health indicators. Despite the large amount of data involved, the use of Combine allows for filtering and transformation into a manageable form.

1.3 Reuseable patterns

To summarize, this process involves condensing a large amount of code into one line and including boilerplate, which allows for the convenient chaining of operations, as well as effective error handling and cancellation. It also provides significant control over timing and other aspects of the data stream. If we only require a single value, we can easily convert a completion handler to a Publisher using a Future. However, if we need multiple values, a Subject is the better option.

Combine is a versatile tool that goes beyond data processing. It can also model the user interaction in an app as a stream of data, making it useful for automating testing and identifying elusive, difficult to reproduce, bugs. I've shared a project on GitHub that demonstrates how to use Combine to model user interactions using patterns of state changes in view models.

As noted before, Combine and Swift Concurrency work great in tandem. Starting with iOS 15, we have the option to transform Publishers into AsyncStreams and extract their values by utilizing a for-await loop. Rather than choosing between the two, the key is finding the ideal way to integrate them. For the purposes of this article, I focused specifically on data streams, and didn't delve into other potential applications.

1.4 One-size does not fit all

I would like to offer another example of an intersection that could save you time and frustration. In the following code sample, we convert NSManagedObjects to model structs, a common task in many applications. However, using the Combine pattern of wrapping a data series into a Subject within the same function can lead to a tricky problem that might not be easily detected. If the operation involves a network call,

any issues may remain hidden due to natural delays. We don't want to end up with non-deterministic outcomes caused by timing differences. Hopefully, this example will help.

```
func heartRateSamplePublisher() -> AnyPublisher<HeartRateSample, Error>
{
    let subject = PassthroughSubject<HeartRateSample, Error>()
    do
    {
        let fetchResult = try context.fetch(fetchRequest())
        let models = (fetchResult as? [NSManagedObject])?
            .compactMap { $0.toHeartRateSample() }
        models?.forEach { model in subject.send(model) }

        /// Combine doesn't give us threading or backpressure control in
        /// this scenario. The above send() might lose one or more values if the
        /// subscriber is not already connected.
    }
    catch { return Fail(error: error).eraseToAnyPublisher() }
    subject.send(completion: .finished)
    return subject.eraseToAnyPublisher()
}
```

Combine doesn't give us threading or backpressure control when sending from a Subject. One solution could be to enclose the affected code in a dispatch group. However, with the advent of Swift Concurrency, we can now opt for a cleaner alternative. This entails wrapping our code with a Task whenever we need to execute it as a single unit. By taking advantage of each tool's unique capabilities, we can maximize the potential benefits. Swift Concurrency is a lower level API that grants us more control, which can benefit everyone, especially when working with Combine.

```
func heartRateSamplePublisher() -> AnyPublisher<HeartRateSample, Error>
{
    let subject = PassthroughSubject<HeartRateSample, Error>()
    do
```

```
    {
        let fetchResult = try context.fetch(fetchRequest())
        let models = (fetchResult as? [NSManagedObject])?
            .compactMap { $0.toHeartRateSample() }
        Task
        {
            models?.forEach { subject.send($0) }
            subject.send(completion: .finished)
        }
    }
    catch { return Fail(error: error).eraseToAnyPublisher() }
    return subject.eraseToAnyPublisher()
}
```

2 Data modeling

Next, let's talk about Core Data.

It might help to have a refresher on setting up a data model. Creating one is a little tricky, but it's not too bad once you get the hang of it. The data model is represented by a graph of entities and their relationships. Entities are like tables in a database. They have attributes, which are like columns in a database. Entities can also have relationships to other entities. If we create a HeartRate entity, we can add a relationship to a Workout entity.

I'm giving this example because it's like the one I'm using in my app. It will illustrate how to set up relationships between entities. It's simple to understand that a workout will have many heart rates, and a heart rate belongs to a single workout. Therefore, we can create a one-to-many relationship between a workout and heart rate samples. We can also create a one-to-one relationship between heart rate samples and a workout.

For an entity, the destination is the entity it's related to. The relationship is either to-one (one to one) or to-many (one to many.) The name for the relationship matches

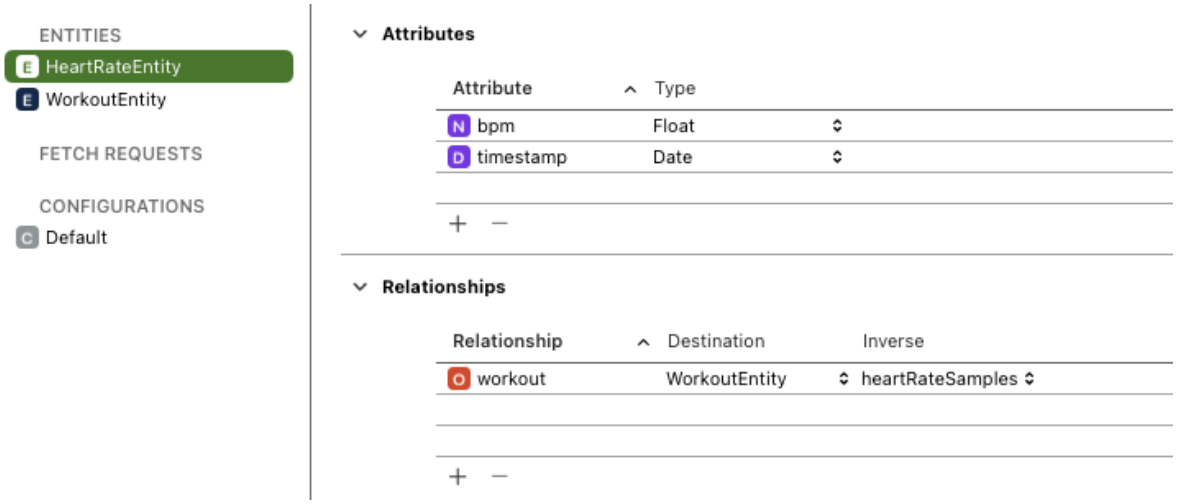

Figure 3: Heart Rate entity, simplified to help illustrate creating relationships.

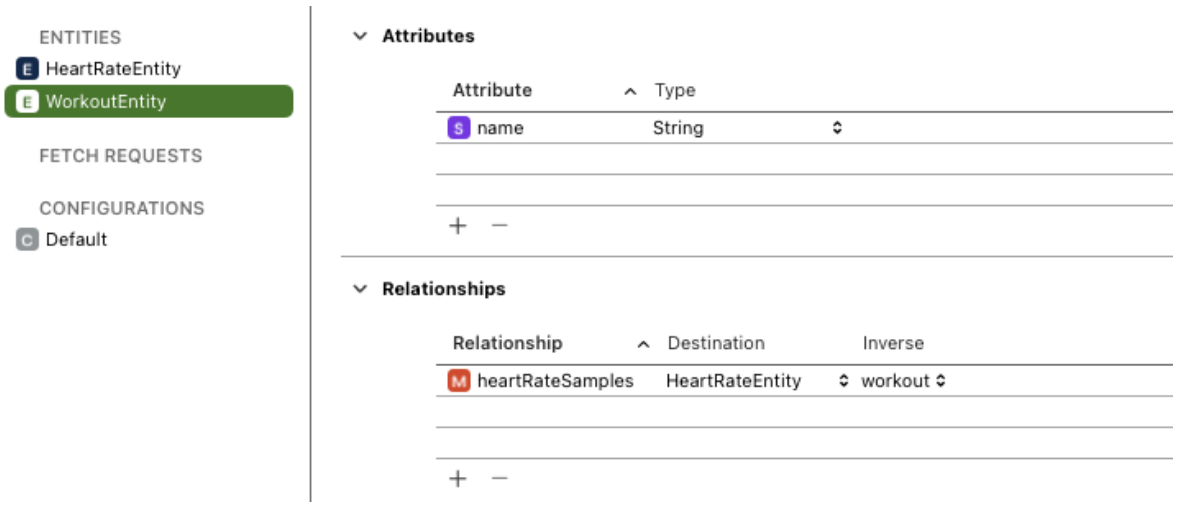

Figure 4: Workout entity, simplified to help illustrate creating relationships.

the destination entity except we don't call it an entity. The inverse is the relationship from the destination entity back to the source entity.

3 CloudKit dashboard

Once we've set everything up, we can begin running tests to confirm that our data is being saved correctly. The CloudKit dashboard is available to us to check that our data is being stored in the private database. If your developer account differs from the one you use on your devices, you may use the "Act As iCloud Account" button to log in. It could also help to have the app open on a device.

The most critical piece of information is when the zone appears as

$$\text{“com.apple.coredata.cloudkit.zone,”} \tag{1}$$

instead of "_defaultZone," for the private database. I am providing a visual representation of what to seek so that you can quickly identify it. Not knowing what to anticipate may result in missing certain things. Also, it's worth noting that it may take some time for records to appear after erasing and deleting all data.

4 Conclusion

One of the advantages of using Swift is that we can effectively retrieve data collected by an Apple Watch, as it is automatically saved in the Health app on our iPhone. By utilizing functional reactive programming, Combine, and Swift Concurrency, we can efficiently manage the flow of data. Additionally, we can utilize Core Data and CloudKit to save and synchronize our data across multiple devices without the need to add a server. As everything is overseen by Apple, we can concentrate on improving our app without worrying about extra security or privacy concerns.

In my upcoming article, I will be covering the topic of optimizing data retrieval and presenting it in a visually appealing manner through charts. Stay tuned!

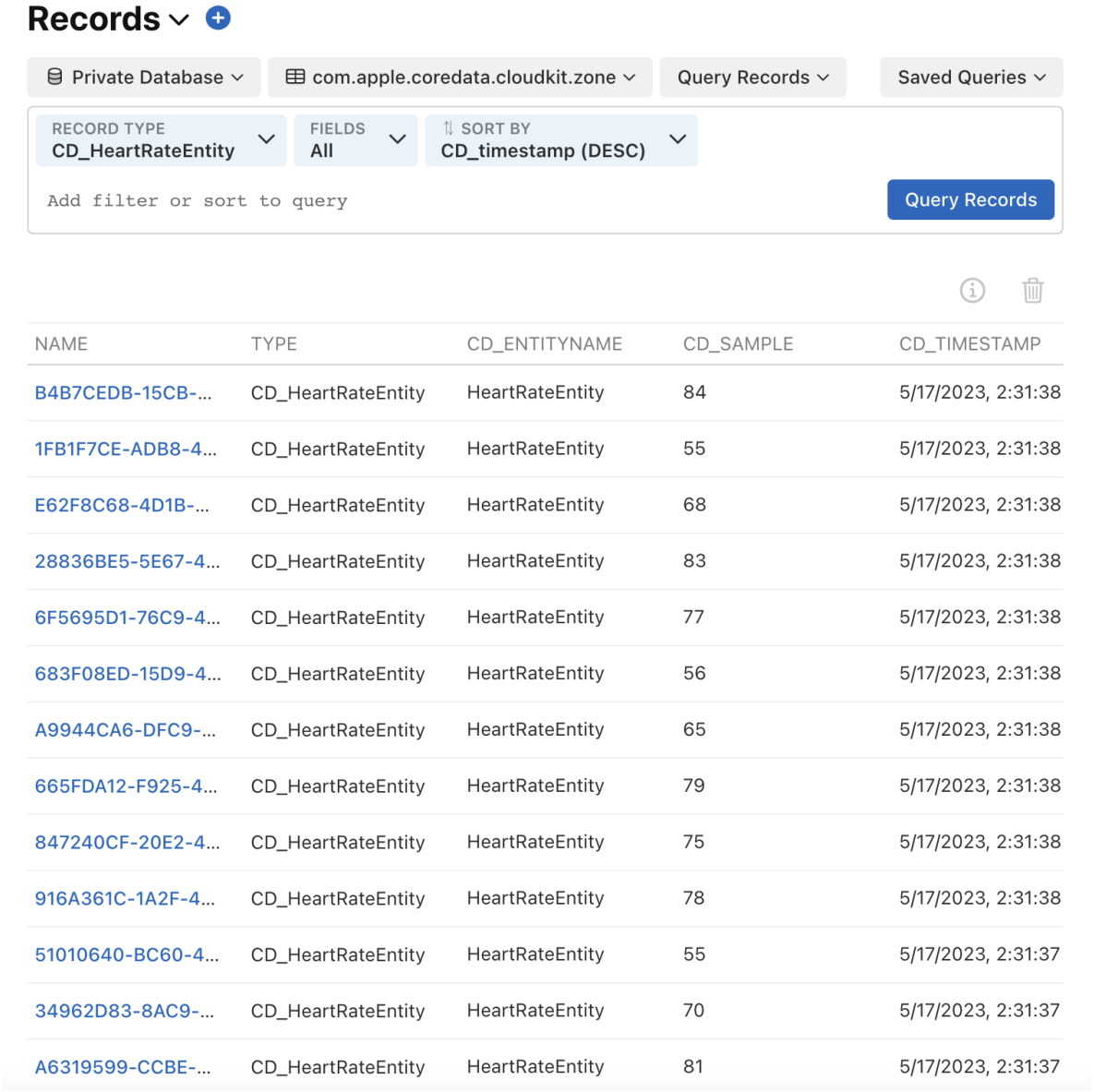
Records
Private Database
com.apple.coredata.cloudkit.zone
Query Records
Saved Queries
RECORD TYPE
CD_HeartRateEntity
FIELDS
All
SORT BY
CD_timestamp (DESC)
Add filter or sort to query
Query Records
NAME
TYPE
CD_ENTITYNAME
CD_SAMPLE
CD_TIMESTAMP
B4B7CEDB-15CB-...
CD_HeartRateEntity
HeartRateEntity
84
5/17/2023, 2:31:38
1FB1F7CE-ADB8-4...
CD_HeartRateEntity
HeartRateEntity
55
5/17/2023, 2:31:38
E62F8C68-4D1B-...
CD_HeartRateEntity
HeartRateEntity
68
5/17/2023, 2:31:38
28836BE5-5E67-4...
CD_HeartRateEntity
HeartRateEntity
83
5/17/2023, 2:31:38
6F5695D1-76C9-4...
CD_HeartRateEntity
HeartRateEntity
77
5/17/2023, 2:31:38
683F08ED-15D9-4...
CD_HeartRateEntity
HeartRateEntity
56
5/17/2023, 2:31:38
A9944CA6-DFC9-...
CD_HeartRateEntity
HeartRateEntity
65
5/17/2023, 2:31:38
665FDA12-F925-4...
CD_HeartRateEntity
HeartRateEntity
79
5/17/2023, 2:31:38
847240CF-20E2-4...
CD_HeartRateEntity
HeartRateEntity
75
5/17/2023, 2:31:38
916A361C-1A2F-4...
CD_HeartRateEntity
HeartRateEntity
78
5/17/2023, 2:31:38
51010640-BC60-4...
CD_HeartRateEntity
HeartRateEntity
55
5/17/2023, 2:31:37
34962D83-8AC9-...
CD_HeartRateEntity
HeartRateEntity
70
5/17/2023, 2:31:37
A6319599-CCBE-...
CD_HeartRateEntity
HeartRateEntity
81
5/17/2023, 2:31:37

Figure 5:

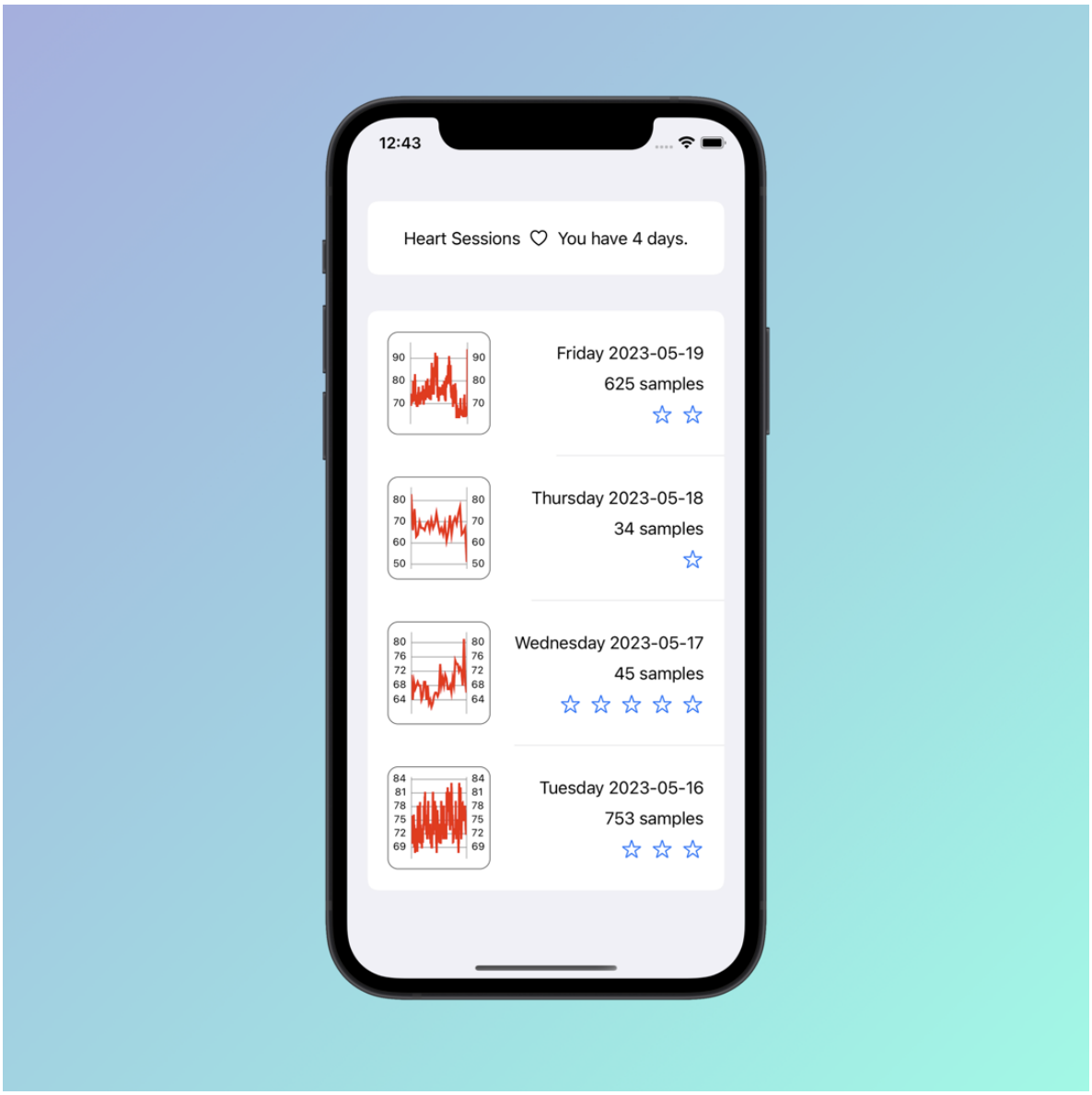
12:43
Heart Sessions ♡ You have 4 days.
90 80 70
Friday 2023-05-19
625 samples
80 70 60 50
Thursday 2023-05-18
34 samples
80 76 72 68 64
Wednesday 2023-05-17
45 samples
84 81 78 75 72 69
Tuesday 2023-05-16
753 samples

Figure 6:


12:21
Heart Sessions ♡ You have 4 days.
130
118
105
92
80
68
55
Friday
2023-05-19
625 samples
{First} to {Last}
{Total}
Thursday
2023-05-18
34 samples
{First} to {Last}
{Total}
Wednesday
2023-05-17
45 samples
{First} to {Last}
{Total}
Tuesday
2023-05-16
753 samples
{First} to {Last}

Figure 7: