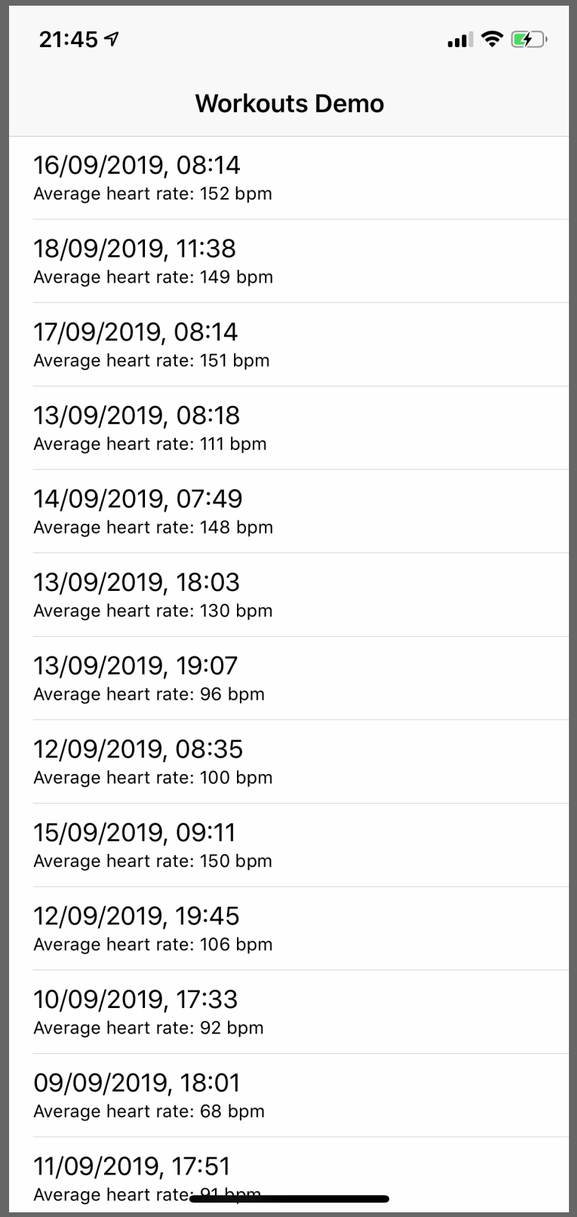
I've got a brand new Combine Publisher

(I'll give you the key)

What is this all for? 


21:45
Workouts Demo
16/09/2019, 08:14
Average heart rate: 152 bpm
18/09/2019, 11:38
Average heart rate: 149 bpm
17/09/2019, 08:14
Average heart rate: 151 bpm
13/09/2019, 08:18
Average heart rate: 111 bpm
14/09/2019, 07:49
Average heart rate: 148 bpm
13/09/2019, 18:03
Average heart rate: 130 bpm
13/09/2019, 19:07
Average heart rate: 96 bpm
12/09/2019, 08:35
Average heart rate: 100 bpm
15/09/2019, 09:11
Average heart rate: 150 bpm
12/09/2019, 19:45
Average heart rate: 106 bpm
10/09/2019, 17:33
Average heart rate: 92 bpm
09/09/2019, 18:01
Average heart rate: 68 bpm
11/09/2019, 17:51
Average heart rate: 91 bpm

Workouts sample query

A query to retrieve all workouts stored in health kit

```
extension HKSampleQuery {

    public static func workouts(
        completion: @escaping (HKSampleQuery, [HKSample]?, Error?) -> Void
    ) -> HKSampleQuery {

        let sampleType = HKWorkoutType.workoutType()
        let start = Date.distantPast
        let end = Date()
        let predicate = HKQuery.predicateForSamples(withStart: start, end: end, options: HKQueryOptions())
        let sortDescriptor = NSSortDescriptor(key: HKSampleSortIdentifierStartDate, ascending: false)
        let limit = HKObjectQueryNoLimit
        return HKSampleQuery(sampleType: sampleType,
                             predicate: predicate,
                             limit: limit,
                             sortDescriptors: [sortDescriptor],
                             resultsHandler: completion)
    }
}
```

Average heart rate statistics query

A query to retrieve the average heart rate for a given workout

```
extension HKStatisticsQuery {

    public static func averageHeartRate(
        for workout: HKWorkout,
        completion: @escaping (HKStatisticsQuery, HKStatistics?, Error?) -> Void
    ) -> HKStatisticsQuery {

        let predicate = HKQuery.predicateForSamples(during: workout.activeIntervals)
        return HKStatisticsQuery(quantityType: .heartRate,
                                 quantitySamplePredicate: predicate,
                                 options: [.discreteAverage],
                                 completionHandler: completion)
    }
}
```

Workout item

Provides both the workout and average heart rate information

```
struct WorkoutItem {
    let start: Date
    let end: Date
    let averageHeartRate: Double
}
```

Traditional fetching 

Fetch method with a completion block

```
extension HKHealthStore {

    func fetchWorkouts(completion: @escaping (Result<[WorkoutItem], Error>) -> Void) {

        requestAuthorization(toShare: nil, read: sampleTypes) { success, error in

            guard success else {
                completion(.failure(error!))
                return
            }

            let workoutsQuery = HKSampleQuery.workouts { _, result, error in

                guard let samples = result else {
                    completion(.failure(error!))
                    return
                }

                guard let workouts = samples as? [HKWorkout] else {
                    print("Samples are not workouts!")
                    return
                }

                let group = DispatchGroup()
                var result: [WorkoutItem] = []

                for workout in workouts {

                    group.enter()

                    let averageHeartRateQuery = HKStatisticsQuery.averageHeartRate(for: workout) {
                        query, averageHeartRateResult, error in

                        defer { group.leave() }

                        guard let averageHeartRate = averageHeartRateResult?.averageQuantity() else {
                            print("Can't get average heart rate! \(error!)")
                            return
                        }

                        result.append(WorkoutItem(workout: workout, averageHeartRate: averageHeartRate))
                    }

                    self.execute(averageHeartRateQuery)
                }

                group.notify(queue: .main) {
                    completion(.success(result))
                }
            }

            self.execute(workoutsQuery)
        }
    }
}
```

Fetch method with a completion block

Request authorization from HealthKit to read the data

```
extension HKHealthStore {

    func fetchWorkouts(completion: @escaping (Result<[WorkoutItem], Error>) -> Void) {

        requestAuthorization(toShare: nil, read: sampleTypes) { success, error in

            guard success else {
                completion(.failure(error!))
                return
            }
```

Fetch method with a completion block

Fetch all the workouts

```
let workoutsQuery = HKSampleQuery.workouts { _, result, error in

    guard let samples = result else {
        completion(.failure(error!))
        return
    }

    guard let workouts = samples as? [HKWorkout] else {
        print("Samples are not workouts!")
        return
    }
```

Fetch method with a completion block

Fetch the associated average heart rate data for each workout

```
let group = DispatchGroup()
var result: [WorkoutItem] = []

for workout in workouts {

    group.enter()

    let averageHeartRateQuery = HKStatisticsQuery.averageHeartRate(for: workout) {
        query, averageHeartRateResult, error in

        defer { group.leave() }

        guard let averageHeartRate = averageHeartRateResult?.averageQuantity() else {
            print("Can't get average heart rate! \(error!)")
            return
        }

        result.append(WorkoutItem(workout: workout, averageHeartRate: averageHeartRate))
    }

    self.execute(averageHeartRateQuery)
}

group.notify(queue: .main) {
    completion(.success(result))
}
```

Fetch method with a completion block

Oh and don't forget to execute the workouts query

😅

```
                self.execute(workoutsQuery)
            }
        }
}
```

Fetch method with a completion block

```
extension HKHealthStore {

    func fetchWorkouts(completion: @escaping (Result<[WorkoutItem], Error>) -> Void) {

        requestAuthorization(toShare: nil, read: sampleTypes) { success, error in

            guard success else {
                completion(.failure(error!))
                return
            }

            let workoutsQuery = HKSampleQuery.workouts { _, result, error in

                guard let samples = result else {
                    completion(.failure(error!))
                    return
                }

                guard let workouts = samples as? [HKWorkout] else {
                    print("Samples are not workouts!")
                    return
                }

                let group = DispatchGroup()
                var result: [WorkoutItem] = []

                for workout in workouts {

                    group.enter()

                    let averageHeartRateQuery = HKStatisticsQuery.averageHeartRate(for: workout) {
                        query, averageHeartRateResult, error in

                        defer { group.leave() }

                        guard let averageHeartRate = averageHeartRateResult?.averageQuantity() else {
                            print("Can't get average heart rate! \(error!)")
                            return
                        }

                        result.append(WorkoutItem(workout: workout, averageHeartRate: averageHeartRate))
                    }

                    self.execute(averageHeartRateQuery)
                }

                group.notify(queue: .main) {
                    completion(.success(result))
                }
            }

            self.execute(workoutsQuery)
        }
    }
}
```

Can Combine help? 

Create the publisher

```
public struct SamplePublisher {

    public typealias Output = [HKSample]
    public typealias Failure = Error

    fileprivate let store: HKHealthStore
    fileprivate let sampleType: HKSampleType
    fileprivate let predicate: NSPredicate?
    fileprivate let limit: Int
    fileprivate let sortDescriptors: [NSSortDescriptor]?
}
```

Conform to the Publisher protocol

```
extension SamplePublisher: Publisher {

    public func receive<Subscriber>(subscriber: Subscriber)
        where
        Subscriber: Combine.Subscriber,
        Subscriber.Failure == Failure,
        Subscriber.Input == Output
    {
        let subscription = Subscription(subscriber: subscriber,
                                        store: store,
                                        sampleType: sampleType,
                                        predicate: predicate,
                                        limit: limit,
                                        sortDescriptors: sortDescriptors)
        subscriber.receive(subscription: subscription)
    }
}
```

```
extension SamplePublisher {

    fileprivate final class Subscription<Subscriber>
        where
        Subscriber: Combine.Subscriber,
        Subscriber.Failure == Failure,
        Subscriber.Input == Output
    {
        private let store: HKHealthStore
        private let query: HKSampleQuery

        fileprivate init(subscriber: Subscriber,
                         store: HKHealthStore,
                         sampleType: HKSampleType,
                         predicate: NSPredicate?,
                         limit: Int,
                         sortDescriptors: [NSSortDescriptor]?) {

            self.store = store
            self.query = HKSampleQuery(sampleType: sampleType,
                                       predicate: predicate,
                                       limit: limit,
                                       sortDescriptors: sortDescriptors ) { _, samples, error in

                if let samples = samples {
                    _ = subscriber.receive(samples)
                    subscriber.receive(completion: .finished)
                } else {
                    subscriber.receive(completion: .failure(error!))
                }
            }
        }
    }
}
```

Conform to the Subscription protocol

```
extension SamplePublisher.Subscription: Subscription {

    func request(_ demand: Subscribers.Demand) {
        store.execute(query)
    }

    func cancel() {
        store.stop(query)
    }
}
```

Add an extension to provide the publisher

```
extension HKHealthStore {

    public func samplePublisher(
        sampleType: HKSampleType,
        predicate: NSPredicate?,
        limit: Int,
        sortDescriptors: [NSSortDescriptor]?
    ) -> SamplePublisher {

        SamplePublisher(store: self,
                                sampleType: sampleType,
                                predicate: predicate,
                                limit: limit,
                                sortDescriptors: sortDescriptors)
    }
}
```

Wait! We already had a convenience initializer for a sample query! 

Workouts sample query

A query to retrieve all workouts stored in health kit

```
extension HKSampleQuery {

    public static func workouts(
        completion: @escaping (HKSampleQuery, [HKSample]?, Error?) -> Void
    ) -> HKSampleQuery {

        let sampleType = HKWorkoutType.workoutType()
        let start = Date.distantPast
        let end = Date()
        let predicate = HKQuery.predicateForSamples(withStart: start, end: end, options: HKQueryOptions())
        let sortDescriptor = NSSortDescriptor(key: HKSampleSortIdentifierStartDate, ascending: false)
        let limit = HKObjectQueryNoLimit
        return HKSampleQuery(sampleType: sampleType,
                             predicate: predicate,
                             limit: limit,
                             sortDescriptors: [sortDescriptor],
                             resultsHandler: completion)
    }
}
```

Create the publisher

```
public struct SamplePublisher2 {

    public typealias Output = [HKSample]
    public typealias Failure = Error

    typealias Completion = (HKSampleQuery, [HKSample]?, Error?) -> ()

    fileprivate let store: HKHealthStore
    fileprivate let query: (@escaping Completion) -> HKSampleQuery
}
```

Conform to the Publisher protocol

```
extension SamplePublisher2: Publisher {

    public func receive<Subscriber>(subscriber: Subscriber)
        where
        Subscriber: Combine.Subscriber,
        Subscriber.Failure == Failure,
        Subscriber.Input == Output
    {
        let subscription = Subscription(subscriber: subscriber, store: store, query: query)
        subscriber.receive(subscription: subscription)
    }
}
```

```
extension SamplePublisher2 {

    fileprivate final class Subscription<Subscriber>
        where
        Subscriber: Combine.Subscriber,
        Subscriber.Failure == Failure,
        Subscriber.Input == Output
    {
        private let store: HKHealthStore
        private let query: HKSampleQuery
        fileprivate init(subscriber: Subscriber,
                         store: HKHealthStore,
                         query: (@escaping Completion) -> HKSampleQuery) {

            self.store = store
            self.query = query { _, output, failure in

                if let output = output {
                    _ = subscriber.receive(output)
                    subscriber.receive(completion: .finished)
                } else {
                    subscriber.receive(completion: .failure(failure!))
                }
            }
        }
    }
}
```

Conform to the Subscription protocol

```
extension SamplePublisher2.Subscription: Subscription {

    func request(_ demand: Subscribers.Demand) {
        store.execute(query)
    }

    func cancel() {
        store.stop(query)
    }
}
```

Add an extension to provide the publisher

```
extension HKHealthStore {

    public func publisher(
        for query: @escaping (@escaping (HKSampleQuery, [HKSample]?, Error?) -> ()) -> HKSampleQuery
    ) -> SamplePublisher2 {
        SamplePublisher2(store: self, query: query)
    }
}
```

Using SamplePublisher2

```
let store = HKHealthStore()

let workoutsPublisher = store.publisher(for: {
    HKSampleQuery.workouts(completion: $0)
})
```

Using SamplePublisher2

```
let store = HKHealthStore()

let workoutsPublisher = store.publisher(for: HKSampleQuery.workouts)
```

Let's take a gander at the HealthKit queries 

A look at the HealthKit queries

```
init(sampleType: HKSampleType,
     predicate: NSPredicate?,
     limit: Int,
     sortDescriptors: [NSSortDescriptor]?,
     resultsHandler: @escaping (HKSampleQuery, [HKSample]?, Error?) -> Void)

init(quantityType: HKQuantityType,
     quantitySamplePredicate: NSPredicate?,
     options: HKStatisticsOptions,
     completionHandler: @escaping (HKStatisticsQuery, HKStatistics?, Error?) -> Void)

init(sampleType: HKSampleType,
     samplePredicate: NSPredicate?,
     completionHandler: @escaping (HKSourceQuery, Set<HKSource>?, Error?) -> Void)
```

The completion closures

```
(HKStatisticsQuery, HKStatistics?,  Error?  ) -> Void

(HKSourceQuery,     Set<HKSource>?, Error?  ) -> Void

(HKSampleQuery,     [HKSample]?,    Error?  ) -> Void
```

The completion closures

```
(HKStatisticsQuery, HKStatistics?,  Error?  ) -> Void

(HKSourceQuery,     Set<HKSource>?, Error?  ) -> Void

(HKSampleQuery,     [HKSample]?,    Error?  ) -> Void
```

```
(Query,
```

The completion closures

```
(HKStatisticsQuery, HKStatistics?,  Error?  ) -> Void

(HKSourceQuery,     Set<HKSource>?, Error?  ) -> Void

(HKSampleQuery,     [HKSample]?,    Error?  ) -> Void
```

```
(Query,             Output?,
```

These aren't three different closures!

```
(HKStatisticsQuery, HKStatistics?,  Error?  ) -> Void

(HKSourceQuery,     Set<HKSource>?, Error?  ) -> Void

(HKSampleQuery,     [HKSample]?,    Error?  ) -> Void
```

```
(Query,             Output?,        Failure?) -> Void
```

Let's make the publisher

generic! 

Specific

```
public struct SamplePublisher2 {

    public typealias Output = [HKSample]
    public typealias Failure = Error

    typealias Completion = (HKSampleQuery, [HKSample]?, Error?) -> ()

    fileprivate let store: HKHealthStore
    fileprivate let query: (@escaping Completion) -> HKSampleQuery
}
```

Generic

```
public struct QueryPublisher<Query: HKQuery, Output, Failure: Error> {

    typealias Completion = (Query, Output?, Failure?) -> ()

    fileprivate let store: HKHealthStore
    fileprivate let query: (@escaping Completion) -> Query
}
```

Specific

```
extension SamplePublisher2: Publisher {

    public func receive<Subscriber>(subscriber: Subscriber)
        where
        Subscriber: Combine.Subscriber,
        Subscriber.Failure == Failure,
        Subscriber.Input == Output
    {
        let subscription = Subscription(subscriber: subscriber, store: store, query: query)
        subscriber.receive(subscription: subscription)
    }
}
```

Generic

```
extension QueryPublisher: Publisher {

    public func receive<Subscriber>(subscriber: Subscriber)
        where
        Subscriber: Combine.Subscriber,
        Subscriber.Failure == Failure,
        Subscriber.Input == Output
    {
        let subscription = Subscription(subscriber: subscriber, store: store, query: query)
        subscriber.receive(subscription: subscription)
    }
}
```

Specific

```
extension SamplePublisher2 {

    fileprivate final class Subscription<Subscriber>
        where
        Subscriber: Combine.Subscriber,
        Subscriber.Failure == Failure,
        Subscriber.Input == Output
    {
        private let store: HKHealthStore
        private let query: HKSampleQuery
        fileprivate init(subscriber: Subscriber,
                         store: HKHealthStore,
                         query: (@escaping Completion) -> HKSampleQuery) {

            self.store = store
            self.query = query { _, output, failure in

                if let output = output {
                    _ = subscriber.receive(output)
                    subscriber.receive(completion: .finished)
                } else {
                    subscriber.receive(completion: .failure(failure!))
                }
            }
        }
    }
}
```

Generic

```
extension QueryPublisher {

    fileprivate final class Subscription<Subscriber>
        where
        Subscriber: Combine.Subscriber,
        Subscriber.Failure == Failure,
        Subscriber.Input == Output
    {
        private let store: HKHealthStore
        private let query: Query
        fileprivate init(subscriber: Subscriber,
                         store: HKHealthStore,
                         query: (@escaping Completion) -> Query) {

            self.store = store
            self.query = query { _, output, failure in

                if let output = output {
                    _ = subscriber.receive(output)
                    subscriber.receive(completion: .finished)
                } else {
                    subscriber.receive(completion: .failure(failure!))
                }
            }
        }
    }
}
```

Specific

```
extension SamplePublisher2.Subscription: Subscription {

    func request(_ demand: Subscribers.Demand) {
        store.execute(query)
    }

    func cancel() {
        store.stop(query)
    }
}
```

Generic

```
extension QueryPublisher.Subscription: Subscription {

    func request(_ demand: Subscribers.Demand) {
        store.execute(query)
    }

    func cancel() {
        store.stop(query)
    }
}
```

Specific

```
extension HKHealthStore {

    public func publisher(
        for query: @escaping (@escaping (HKSampleQuery, [HKSample]?, Error?) -> ()) -> HKSampleQuery
    ) -> SamplePublisher2 {
        SamplePublisher2(store: self, query: query)
    }
}
```

Generic

```
extension HKHealthStore {

    public func publisher<Query: HKQuery, Output, Failure: Error>(
        for query: @escaping (@escaping (Query, Output?, Failure?) -> ()) -> Query
    ) -> QueryPublisher<Query, Output, Failure> {
        QueryPublisher(store: self, query: query)
    }
}
```

Using the generic QueryPublisher

```
let store = HKHealthStore()

let workouts = store.publisher(for: HKSampleQuery.workouts)
```

Using the generic QueryPublisher

```
let store = HKHealthStore()

let workouts = store.publisher(for: HKSampleQuery.workouts)

let averageHeartRate = store.publisher(for: {
    HKStatisticsQuery.averageHeartRate(for: workout, completion: $0)
})
```

The final reveal 

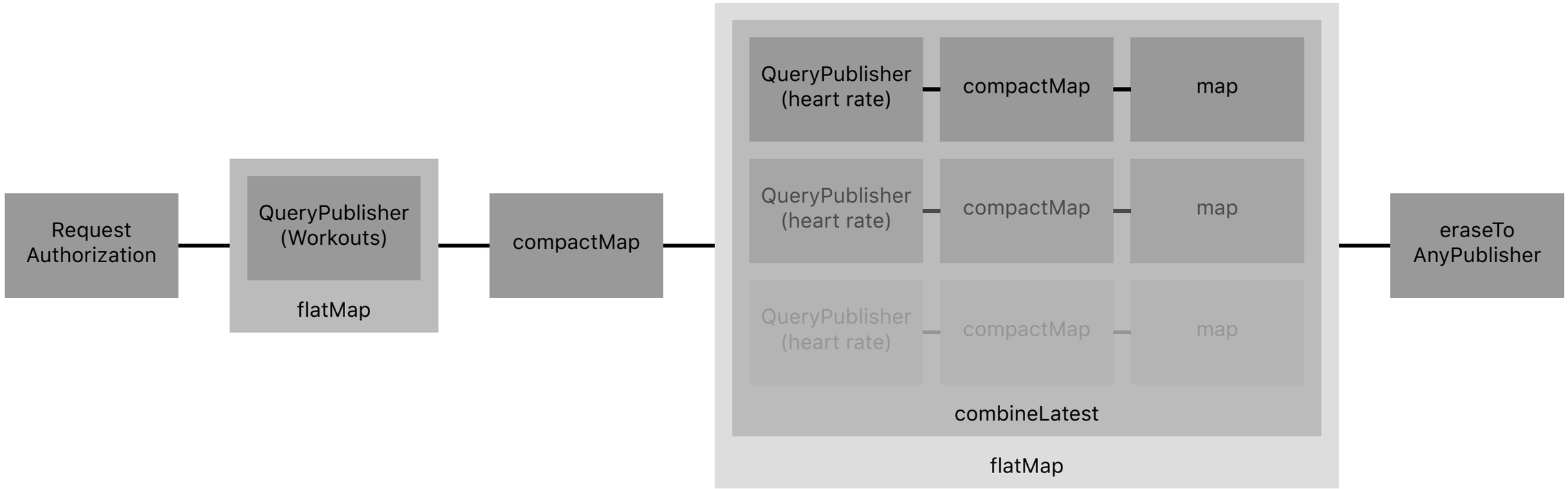
Fetch method with a completion block

```
extension HKHealthStore {

    func fetchWorkouts(completion: @escaping (Result<[WorkoutItem], Error>) -> Void) {

        requestAuthorization(toShare: nil, read: sampleTypes) { success, error in

            guard success else {
                completion(.failure(error!))
                return
            }

            let workoutsQuery = HKSampleQuery.workouts { _, result, error in

                guard let samples = result else {
                    completion(.failure(error!))
                    return
                }

                guard let workouts = samples as? [HKWorkout] else {
                    print("Samples are not workouts!")
                    return
                }

                let group = DispatchGroup()
                var result: [WorkoutItem] = []

                for workout in workouts {

                    group.enter()

                    let averageHeartRateQuery = HKStatisticsQuery.averageHeartRate(for: workout) {
                        query, averageHeartRateResult, error in

                        defer { group.leave() }

                        guard let averageHeartRate = averageHeartRateResult?.averageQuantity() else {
                            print("Can't get average heart rate! \(error!)")
                            return
                        }

                        result.append(WorkoutItem(workout: workout, averageHeartRate: averageHeartRate))
                    }

                    self.execute(averageHeartRateQuery)
                }

                group.notify(queue: .main) {
                    completion(.success(result))
                }
            }

            self.execute(workoutsQuery)
        }
    }
}
```

Using the publishers together

```
extension HKHealthStore {

    var workoutsPublisher: AnyPublisher<[WorkoutItem], Error> {

        self.requestAuthorization(read: sampleTypes)
            .flatMap { self.publisher(for: HKSampleQuery.workouts) }
            .compactMap { $0 as? [HKWorkout] }
            .flatMap { workouts in

                workouts.map { workout in

                    self.publisher(for: { HKStatisticsQuery.averageHeartRate(for: workout, completion: $0) })
                        .compactMap { $0.averageQuantity() }
                        .map { WorkoutItem(workout: workout, averageHeartRate: $0) }
                }
                .combineLatest
            }
            .eraseToAnyPublisher()
    }
}
```

Using the publishers together



Thank you

@danielctull