Sample Code

Updating an App to Use Swift Concurrency

Improve your app's performance by refactoring your code to take advantage of asynchronous functions in Swift.

Download

iOS 15.0+ | iPadOS 15.0+ | Xcode 14.0+ | watchOS 8.0+

Overview

Note

This sample code project is associated with WWDC21 session 10194: Swift Concurrency: Update a Sample App.

Swift concurrency provides a standard set of language tools and techniques for concurrent programming. However, you may already have an existing project built with concurrency that uses other frameworks and techniques. You don't have to convert all of your code all at once; instead, you can use specific refactoring techniques to convert your code one piece at time.

This sample provides two separate versions of the Coffee Tracker app:

- The original version uses completion handlers to query the HealthKit SDK and to respond to `CLKComplicationDataSource` calls, and dispatch queues to isolate concurrent access to memory. For more information on the original version, see Creating and updating a complication's timeline.
- The updated version uses Swift's concurrency features to provide clearer code with better error checking at compile time. It replaces the completion handlers with `async` functions, and uses

actors to guarantee safe access to data.

Watch the session to see the process step by step, and then compare the two projects to see the differences.

Configure the Sample Code Project

To add the complication to an active watch face, start by building and running the sample code project in the simulator, and follow these steps:

1. Click the Digital Crown to exit the app and return to the watch face.
2. Using the trackpad, firmly press the watch face to put the face in edit mode, then tap Customize.
3. Swipe left until the configuration screen highlights the complications. Select the complication to modify.
4. Scroll to the Coffee Tracker complication, and then click the Digital Crown again to save your changes.
5. Tap the Coffee Tracker complication to go back to the app.

For more information on setting up watch faces, see Change the watch face on your Apple Watch.

After configuring and running the Coffee Tracker app, you can test the background updates. Make sure the Coffee Tracker complication appears on the active watch face. Then build and run the app in Simulator, and follow these steps:

1. Add one or more drinks using the app’s main view.
2. Click the Digital Crown to send the app to the background.
3. Open Settings, and scroll down to Health > Health Data > Nutrition > Caffeine to see all of the drinks you added to the app.
4. Click Delete Caffeine Data to clear all of the caffeine samples from HealthKit.
5. Navigate back to the watch face.

Coffee Tracker updates the complication within 15 minutes; however, the update may be delayed based on the system’s current state.

Convert Completion Handlers to Use Asynchronous Methods

The `HealthKitController` type contains several calls to the HealthKit SDK. In SDKs that support Swift concurrency, frameworks add `async-await` versions of most functions that previously took completion handlers. You can remove completion handlers by updating these calls to use the `async-await` versions. You suspend the `store.save()` operation by adding the `await` keyword. Execution resumes after the `await` completes. An `async` function can also be a throwing function, which you call by prepending `try await` to the function call. Wrap the call in a `do-catch` statement instead of using an `Error?` type as a parameter to the completion handler.

```
// Save the sample to the HealthKit store.
do {
    try await store.save(caffeineSample)
    self.logger.debug("\(mgCaffeine) mg Drink saved to HealthKit")
} catch {
    self.logger.error("Unable to save \(caffeineSample) to the HealthKit store: \(er
}
```

In some cases an SDK call requires using a completion handler. For example, a call to `init(type:predicate:anchor:limit:resultsHandler:)` takes a completion handler, but the call that needs to `await` is the call to `execute(_:)`.

To `await` the results of a completion handler in these cases, add a `continuation`:

```
private func queryHealthKit() async throws -> ([HKSample]?, [HKDeletedObject]?, HKQu
    return try await withCheckedThrowingContinuation { continuation in
        // Create a predicate that only returns samples created within the last 24 h
        let endDate = Date()
        let startDate = endDate.addingTimeInterval(-24.0 * 60.0 * 60.0)
        let datePredicate = HKQuery.predicateForSamples(withStart: startDate, end: e

        // Create the query.
        let query = HKAnchoredObjectQuery(
            type: caffeineType,
            predicate: datePredicate,
            anchor: anchor,
            limit: HKObjectQueryNoLimit) { (_, samples, deletedSamples, newAnchor, e

            // When the query ends, check for errors.
            if let error = error {
                continuation.resume(throwing: error)
            } else {
                continuation.resume(returning: (samples, deletedSamples, newAnchor))
            }

        }
        store.execute(query)
    }
}
```

To protect the stored properties on the controller when accessed asynchronously, change `HealthKitController` from a `class` type to an `actor`:

```
actor HealthKitController {
```

Calls to `async` functions from synchronous functions are made by creating new asynchronous tasks, which can use `await` to wait for completion:

```
// Handle background refresh tasks.
case let backgroundTask as WKApplicationRefreshBackgroundTask:

    Task {
        // Check for updates from HealthKit.
        let model = CoffeeData.shared

        let success = await model.healthKitController.loadNewDataFromHealthKit()

        if success {
            // Schedule the next background update.
            scheduleBackgroundRefreshTasks()
            self.logger.debug("Background Task Completed Successfully!")
        }

        // Mark the task as ended, and request an updated snapshot, if necessary.
        backgroundTask.setTaskCompletedWithSnapshot(success)
    }
```

Put the Coffee Data Class on the Main Actor

The `CoffeeData` class implements `ObservableObject` and has an `@Published` property to feed the SwiftUI views. To ensure that all updates to this property are made on the main thread, place the type on the main actor:

```
@MainActor
class CoffeeData: ObservableObject {
```

Two methods that perform synchronous IO — the `load` and `save` methods — are factored out into a separate `CoffeeDataStore` actor, which performs these activities away from the main thread. The model type on the main actor must use `await` to call methods on the `CoffeeDataStore` actor, which allows other work to run on the main thread during the synchronous IO operations.

The two types communicate by passing an array of `Drink` values, which is a value type because `Drink` is a structure. Loading returns an array of drinks, and saving takes an array of drinks as an argument.

To perform all methods asynchronously, replace the `currentDrinks` property's `didSet` operation with `private(set)` and add a new `async` method named `drinksUpdated`. Move the code from the setter into the new method. Call the `drinksUpdated` after any code that sets the `currentDrinks` property, using an `await` call.

Update the `drinksUpdated()` method to call the `CoffeeDataStore` actor using an `await` call. The `CoffeeDataStore` actor saves the data on a background thread.

Calls to the `CoffeeData` object from SwiftUI views don't require any use of `await` as these views are also on the main actor due to their use of `@EnvironmentObject`.

Replace Delegates and Completion Handlers with Async Methods

Several methods on the `CLKComplicationDataSource` protocol used to configure the app's timeline take completion handlers, which you can replace with their `async` equivalents:

```
// Define whether the complication is visible when the watch is unlocked.
func privacyBehavior(for complication: CLKComplication) async -> CLKComplicationPriv
    // This is potentially sensitive data. Hide it on the lock screen.
    .hideOnLockScreen
}
```

See Also

Standard Library

`struct Int`

A signed integer value type.

`struct Double`

A double-precision, floating-point value type.

`struct String`

A Unicode string value that is a collection of characters.

`struct Array`

An ordered, random-access collection.

`struct Dictionary`

A collection whose elements are key-value pairs.

Swift Standard Library

Solve complex problems and write high-performance, readable code.

{} TicTacFish: Implementing a game using distributed actors

Use distributed actors to take your Swift concurrency and actor-based apps beyond a single process.