Sample Code

Creating and updating a complication’s timeline

Create complications that batch-load a timeline of future entries and run periodic background sessions to update the timeline.

Download

iOS 13.0+ | iPadOS 13.0+ | watchOS 8.0+ | Xcode 14.2+

Overview

The Coffee Tracker app records a user’s caffeine intake. Each time the user adds a drink, the app recalculates the current caffeine levels and the equivalent cups of coffee consumed. It then updates the complication timeline and estimates the decrease in the user’s caffeine level over the next 24 hours.

This sample demonstrates the basic steps to set up and fill the complication’s timeline, including setting up support for complications, creating entries to fill the complication’s timeline, and then updating the timeline every time the user makes a change.

The app also updates the complications based on external changes that occur when the app isn’t running. Coffee Tracker saves and reads caffeine samples to HealthKit, so the app must respond to any external changes, such as another app adding or deleting a caffeine sample from HealthKit. Coffee Tracker uses a background observer query to monitor HealthKit for changes, and updates the app’s data and the complication timeline.

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 Edit.

3. Swipe left until the edit 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 watch screen to exit the edit screen.

6. 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 the 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. Settings should show all the drinks you added to the app.

4. Click Delete Caffeine Data to clear all the caffeine samples from HealthKit.

5. Navigate back to the watch face.

Coffee Tracker updates the complication within 1 minute; however, the system may delay updates based on it's current state.

Set up support for complications

The app declares the `ComplicationController` class as the complication's data source. Xcode saves this setting in the WatchKit Extension's `Info.plist` file.

Next, the Coffee Tracker app implements the `CLKComplicationDataSource` protocol's methods to configure the app's timeline. The app declares the supported complications by implementing the `getComplicationDescriptors(handler:)` method.

```
func complicationDescriptors() async -> [CLKComplicationDescriptor] {
    logger.debug("Accessing the complication descriptors.")
    let descriptor = CLKComplicationDescriptor(identifier: "Coffee_Tracker_Caffeine_
                                               displayName: "Caffeine Dose",
                                               supportedFamilies: CLKComplicationFam
    return [descriptor]
}
```

ClockKit identifies complications based on their `identifier` and their `family`. Apps can declare support for multiple complications by giving each complication a unique identifier. Coffee Tracker declares support for a single complication, returning a `CLKComplicationDescriptor` object that declares support for the `Coffee_Tracker_Caffeine_Dose` identifier for all families.

Because every copy of Coffee Tracker supports the same set of `CLKComplicationDescriptor` objects, the app doesn't need to implement the data source's `handleSharedComplicationDescriptors(_:)` method. For more information on defining the supported families, see Declaring complications for your app.

Load Future Entries

Because the app can easily calculate caffeine levels in the future, Coffee Tracker declares that it can batch-load future timeline entries by implementing the `getTimelineEndDate(for:withHandler:)` method and setting the end date for the timeline to 24 hours in the future.

ClockKit can request batches of timeline entries up to that deadline. After that point, the caffeine level drops to `0.0`. Because the data stops changing at that point, ClockKit won't need any additional timeline entries until the user adds another drink.

```
// Define how far into the future the app can provide data.
func timelineEndDate(for complication: CLKComplication) async -> Date? {

    // Indicate that the app can provide timeline entries for the next 24 hours.
    Date().addingTimeInterval(24.0 * 60.0 * 60.0)
}
```

Finally, the app sets the privacy behavior by implementing the `getPrivacyBehavior(for:withHandler:)` method, hiding the complication data on the user's caffeine intake when the watch is locked.

```
// 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
}
```

Display current data

Coffee Tracker uses three techniques to keep the complications up to date.

- The app provides future timeline entries in five-minute increments that ClockKit uses to update the complications automatically.
- While the app is running, Coffee Tracker updates its complications whenever the user adds a drink. This updates not just the current complication, but also reloads the entire complication timeline.
- Finally, the app uses a background observer query to monitor HealthKit for any updates to its caffeine samples. The app then updates its data based on any changes.

For more information, see Keeping your complications up to date.

Create timeline entries

If there's an active complication on the watch face, ClockKit calls the data source's methods to keep the complication's timeline filled. ClockKit calls the `getCurrentTimelineEntry(for:withHandler:)` method to get the current complication.

```
// Return the current timeline entry.
func currentTimelineEntry(for complication: CLKComplication) async -> CLKComplicatio
    logger.debug("Accessing the current timeline entry.")
    return createTimelineEntry(forComplication: complication, date: Date())
}
```

Then it calls the `getTimelineEntries(for:after:limit:withHandler:)` method to batch load future timeline entries.

```
// Return future timeline entries.
func timelineEntries(for complication: CLKComplication,
                     after date: Date,
                     limit: Int) async -> [CLKComplicationTimelineEntry]? {
    logger.debug("Accessing timeline entries for dates after \(DateFormatter.localiz

    let fiveMinutes = 5.0 * 60.0
    let twentyFourHours = 24.0 * 60.0 * 60.0

    // Create an array to hold the timeline entries.
    var entries: [CLKComplicationTimelineEntry] = []

    // Calculate the start and end dates.
    var current = date.addingTimeInterval(fiveMinutes)
    let endDate = date.addingTimeInterval(twentyFourHours)

    // Create a timeline entry for every five minutes from the start time.
    // Stop after you reach the limit or the end date.
    while current < endDate && entries.count < limit {
        entries.append(createTimelineEntry(forComplication: complication, date: curr
        current = current.addingTimeInterval(fiveMinutes)
    }

    return entries
}
```

ClockKit automatically calls these methods when it needs to refill the timeline. In Coffee Tracker, both of these methods call the `createTimelineEntry(forComplication:date:)` method to create the event. `createTimelineEntry(forComplication:date:)` then calls `createTemplate(forComplication:date:)` to build the template, and wraps the template in a `CLKComplicationTimelineEntry` object.

For more information, see Loading future timeline events.

Create and fill a complication template

ClockKit uses a template-driven user interface. The system divides the complications into a number of families based on their size and capabilities, and each family then provides a number of templates that define its layout. When ClockKit asks the data source for a timeline entry, the app needs to instantiate a template for the specified identifier and family, and then fill the template with the required data, before wrapping it in a `CLKComplicationTimelineEntry` object and returning it.

Because Coffee Tracker uses only a single identifier, it starts by creating a `switch` statement covering all the families that the app supports.

```
// Select the correct template based on the complication's family.
private func createTemplate(forComplication complication: CLKComplication, date: Dat
    switch complication.family {
    case .modularSmall:
        return createModularSmallTemplate(forDate: date)
    case .modularLarge:
        return createModularLargeTemplate(forDate: date)
    case .utilitarianSmall, .utilitarianSmallFlat:
        return createUtilitarianSmallFlatTemplate(forDate: date)
    case .utilitarianLarge:
        return createUtilitarianLargeTemplate(forDate: date)
    case .circularSmall:
        return createCircularSmallTemplate(forDate: date)
    case .extraLarge:
        return createExtraLargeTemplate(forDate: date)
    case .graphicCorner:
        return createGraphicCornerTemplate(forDate: date)
    case .graphicCircular:
        return createGraphicCircleTemplate(forDate: date)
    case .graphicRectangular:
        return createGraphicRectangularTemplate(forDate: date)
    case .graphicBezel:
        return createGraphicBezelTemplate(forDate: date)
    case .graphicExtraLarge:
        return createGraphicExtraLargeTemplate(forDate: date)

    @unknown default:
        logger.error("Unknown Complication Family")
        fatalError()
    }
}
```

The app calls a helper method for each family that creates a template the family supports. The helper method also creates all the data providers needed to fill the template. For example, the following helper method creates a graphical corner template.

```
// Return a graphic template that fills the corner of the watch face.
private func createGraphicCornerTemplate(forDate date: Date) -> CLKComplicationTempl
    // Create the data providers.
    let leadingValueProvider = CLKSimpleTextProvider(text: "0")
    leadingValueProvider.tintColor = data.color(forCaffeineDose: 0.0)

    let trailingValueProvider = CLKSimpleTextProvider(text: "500")
    trailingValueProvider.tintColor = data.color(forCaffeineDose: 500.0)

    let mgCaffeineProvider = CLKSimpleTextProvider(text: data.mgCaffeineString(atDat
    let mgUnitProvider = CLKSimpleTextProvider(text: "mg Caffeine", shortText: "mg")
    mgUnitProvider.tintColor = data.color(forCaffeineDose: data.mgCaffeine(atDate: d
    let combinedMGProvider = CLKTextProvider(format: "%@ %@", mgCaffeineProvider, mg

    let percentage = Float(min(data.mgCaffeine(atDate: date) / 500.0, 1.0))
    let gaugeProvider = CLKSimpleGaugeProvider(style: .fill,
                                               gaugeColors: [.green, .yellow, .red],
                                                          gaugeColorLocations: [0.0, 300.0 / 50
                                                          fillFraction: percentage)

    // Create the template using the providers.
    return CLKComplicationTemplateGraphicCornerGaugeText(gaugeProvider: gaugeProvide
                                                                    leadingTextProvider: leadir
                                                                    trailingTextProvider: trail
                                                                    outerTextProvider: combinec
}
```

This example creates a curved gauge with text outside it. The gauge is a graphical element, like a thermometer or progress bar. To fill the template, the app supplies a gauge provider, which specifies the gauge's start value, end value, current value, and the color gradient it uses. The app also provides two text providers for the labels at the start and end of the gauge. Finally, it adds another text provider for the main text. Depending on the watch face, the gauge and text may use the specified colors to provide additional information.

Reload the timeline

The `CoffeeData` model object declares `currentDrinks` as a `@Published` property. The system alerts subscribers to any change to the `currentDrinks` array. For example, SwiftUI uses the publisher to trigger updates to the main view.

However, the app also needs to update the complications and the app's saved data after the value is changed. To control this, the app declares the `currentDrinks` setter as `private`, and create an asyncronous `drinkDataUpdated()` method that it calls whenever it updates the `currentDrinks` property.

```
// The list of drinks consumed.
// Because this is @Published property,
// Combine updates the app's main interface when a change occurs.
@Published public private(set) var currentDrinks: [Drink] = []

// Asynchronously update any active complications and save
// the list of drinks after the current drinks property changes.
private func drinkDataUpdated() async {
    logger.debug("Updating the system based on the new current drinks property.")

    // Save the app's data.
    await store.save(currentDrinks)

    // Update any complications on active watch faces.
    let server = CLKComplicationServer.sharedInstance()
    let complications = await server.getActiveComplications()

    for complication in complications {
        server.reloadTimeline(for: complication)
    }
}
```

The `drinkDataUpdated()` method starts by saving the app's data. Next, it accesses the list of active complications from the complication server, and tells the complication to reload its timeline — deleting the existing timeline and loading new data.

However, if the system launches the app to handle a background update, Coffee Tracker may call the `drinkDataUpdated()` method before the complication server reattaches to the active complications. Therefore, the app uses the `getActiveComplication()` to wait, if necessary, until the server connects.

```
extension CLKComplicationServer {

    // Safely access the server's active complications.
    @MainActor
    func getActiveComplications() async -> [CLKComplication] {
        return await withCheckedContinuation { continuation in

            // First, set up the notification.
            let center = NotificationCenter.default
            let mainQueue = OperationQueue.main
            var token: NSObjectProtocol?
            token = center.addObserver(forName: .CLKComplicationServerActiveComplica
                center.removeObserver(token!)
                continuation.resume(returning: self.activeComplications!)
            }

            // Then check to see if we have a valid active complications array.
            if activeComplications != nil {
                center.removeObserver(token!)
                continuation.resume(returning: self.activeComplications!)
            }
        }
    }
}
```

This method starts by setting up an observer for the CLKComplicationServerActiveComplicationsDidChange notification. Then it checks the value of the `activeComplications` property. If the property has a non-`nil` value, it cancels the observer and returns the value. Otherwise, it waits for the observer, and then returns the value.

Schedule background observer queries

Before Coffee Tracker can communicate with HealthKit, it needs to authorize HealthKit and set up the background observer query.

```swift
// Authorize HealthKit and set up the background observer query.
public func setUpHealthKit() {

    // Make sure HealthKit is available and authorized.
    guard isAvailable else { return }
    guard store.authorizationStatus(for: caffeineType) == .sharingAuthorized else {

    // Return if an observer query is already running.
    guard backgroundObserver == nil else { return }

    logger.debug("Setting up the background observer queries.")

    // Set up the background delivery rate.
    store.enableBackgroundDelivery(for: caffeineType, frequency: .immediate) { succe
        guard success else {
            self.logger.error("Unable to set up background delivery from HealthKit:
            fatalError()
        }
    }

    // Set up the observer query.
    backgroundObserver =
    HKObserverQuery(sampleType: caffeineType,
                    predicate: nil,
                    updateHandler: processUpdate(query:completionHandler:error:))

    if let query = backgroundObserver {
        logger.debug("Starting the background observer query.")
        store.execute(query)
    }
}
```

Coffee Tracker requests both read and write access to `.dietaryCaffeine` samples. Then it enables the background delivery for observer queries. Finally, it creates and executes an observer query for the `.dietaryCaffeine` data type.

Coffee tracker creates the background observer query immediately after it launches. Setting up an observer query quickly after launch enables the observer to respond promptly when the system launches it in the background because of a change in HealthKit.

The query runs the `processUpdate(query:,completionHandler:,error:)` method whenever it recieves an update from the observer query.

```
func processUpdate(query: HKObserverQuery,
                   completionHandler: @escaping () -> Void,
                   error: Error?) {

    logger.debug("Received an update from the background observer query.")

    // Check for any errors that occur while setting up the observer query.
    guard error == nil else {
        logger.error("Unable to set up a background observer query: \(error!.localiz
        fatalError()
    }

    logger.debug("Responding to a background query.")

    Task {

        // Load the updated data from the HealthKit Store.
        let success = await loadNewDataFromHealthKit()

        // Check for any errors.
        guard success == true else {
            logger.error("Unable to query for new or deleted caffeine samples.")
            fatalError()
        }

        // Call the completion handler when done.
        completionHandler()
    }
}
```

This method checks the update for errors. If there aren't any errors, it asynchronously loads the new data from HealthKit, calling the update's completion handler as soon as it's done processing the results.

To preserve battery life and maintain performance, WatchKit carefully budgets each app’s time for background tasks. In general, if an app has a complication on the active watch face, it can safely use four updates per hour, shared between both the app’s background tasks and it’s background observer queries. However, HealthKit further limits the update on caffeine samples to a maximum of one per hour. And the system may further limit background activity as needed.

This means, the first update from HealthKit should trigger within a minute. However, the system may delay additional updates for up to an hour or more. For more information, see `enableBackgroundDelivery(for:frequency:withCompletion:)`.

The app also loads any new data from HealthKit whenever it enters the foreground.

See Also

Sample Code

{} Providing Multiple Complications

Present multiple complications for a single complication family using descriptors.