Sample Code

Transferring data with Watch Connectivity

Transfer data between a watchOS app and its companion iOS app.

Download

iOS 17.0+ | iPadOS 17.0+ | watchOS 10.0+ | Xcode 16.0+

Overview

Some watchOS apps rely on their companion iOS app to perform complicated tasks, and need to exchange data with the companion app even when there's no internet connection. The Watch Connectivity framework provides APIs for this purpose. This sample demonstrates how to use the APIs, and how to handle Watch Connectivity background tasks.

Important

Use a physical iPhone and Apple Watch to test this sample code project.

Configure the sample code project

Before building the sample app, perform the following steps in Xcode:

1. Verify that the bundle identifiers for the watchOS app and widget targets are based on the iOS app target's bundle identifier. For example, if the iOS app target uses `com.YourCompany.ProductName`, the watchOS app and widget targets must use `com.YourCompany.ProductName.watchkitapp` and `com.YourCompany.ProductName.watchkitapp.SimpleWatchWidget`, respectively. To check this, select each target and click its Signing & Capabilities tab.

2. In the Signing & Capabilities tab for each target, set the developer team to let Xcode automatically manage the provisioning profile. See Assign a project to a team for details.

3. In the Info tab of the `SimpleWatchConnectivity Watch App` target, change the value of the `WKCompanionAppBundleIdentifier` key to the iOS app target's bundle identifier.

4. Replace the App Group container identifier `group.com.example.apple-samplecode.SimpleWatchConnectivity` with one specific to your team. The identifier occurs in the entitlements for the watchOS app and the widget, and in `WidgetSupport.swift`. See Configuring App Groups for more details.

Transfer data with Watch Connectivity

The Watch Connectivity framework provides APIs that accomplish the following tasks:

- Updating app context data
- Sending a message
- Transferring user info and managing outstanding transfers
- Transferring a file, observing the transfer progress, and managing outstanding transfers
- Updating an active complication from the companion iOS app

All APIs transfer a dictionary between the companion apps, with notable differences. `updateApplicationContext(_:)` sends a dictionary that represents the current app context to the companion app. It overwrites the context data currently existing in the pipeline, if any. `transferUserInfo(_:)` guarantees to deliver a dictionary. If an app performs another transfer before finishing the previous one, the system queues the transfers and delivers them in the order received. `sendMessage(_:replyHandler:errorHandler:)` sends a dictionary immediately. If the method encounters an error, it returns the error via the error handler.

An app can provide a reply handler to receive a response from its companion app. The reply handler runs asynchronously on a background thread, and returns quickly to avoid timeout. Sending a message from a watchOS app wakes up its companion iOS app, if the companion is reachable.

Update an active complication from the companion iOS app

This sample provides a WidgetKit complication that shows a timestamp. To activate the complication:

1. Choose a Modular watch face on the Apple Watch.
2. Press the watch face to show the customization screen, tap the Edit button, and swipe right to show the configuration screen.

3. Tap the large rectangular area, rotate the digital crown to find `SimpleWatchConnectivity Watch App`, tap it, and then select the complication.

4. Press the digital crown and tap the screen to finish the configuration.

To update the complication, the iOS app in this sample calls `transferCurrentComplicationUserInfo` if the complication is active. The system allows 50 transfers of this kind per day. Apps can use `remainingComplicationUserInfoTransfers` to retrieve the number of remaining times.

```
if WCSession.default.isComplicationEnabled {
    let userInfoTranser = WCSession.default.transferCurrentComplicationUserInfo(user
```

The watchOS app persists the data it receives to the shared `UserDefaults`, and calls `reloadTimelines(ofKind:)` for the system to reload the timelines for the widget:

```
WidgetCenter.shared.getCurrentConfigurations { result in
    switch result {
    case .success(let widgetInfoList):
        for widgetInfo in widgetInfoList where widgetInfo.kind == WidgetSupport.widg
            WidgetCenter.shared.reloadTimelines(ofKind: widgetInfo.kind)
        }
    case .failure(let error):
        print(error.localizedDescription)
    }
}
```

When the system requests the timeline, the widget retrieves the data from the shared `UserDefaults` and uses it to create and return a timeline entry:

```
func getTimeline(in context: Context, completion: @escaping (Timeline<Entry>) -> Voi
```

Handle Watch Connectivity background tasks

When using Watch Connectivity, apps must complete every background task (`WKWatchConnectivityRefreshBackgroundTask`). An uncompleted task consumes the background-task time budget that watchOS allocates to the app, which results in a crash when the budget runs out.

This sample retains the tasks in an array, and completes them when:

- The app finishes handling the tasks.

- The current `WCSession` turns to a state other than `.activated`.
- `hasContentPending` becomes `false`, indicating that there's no pending data received prior to `WCSession` activation waiting for processing.

The following code completes the tasks at the end of `handle(_:)`:

```
func handle(_ backgroundTasks: Set<WKRefreshBackgroundTask>) {
    for task in backgroundTasks {
        if let wcTask = task as? WKWatchConnectivityRefreshBackgroundTask {
            wcBackgroundTasks.append(wcTask)
            Logger.shared.append(line: "\(#function):\(wcTask.description) was apper
        } else {
            task.setTaskCompletedWithSnapshot(false)
            Logger.shared.append(line: "\(#function):\(task.description) was complet
        }
    }
    completeBackgroundTasks()
}
```

The following code completes the tasks in the other cases:

```
activationStateObservation = WCSession.default.observe(\.activationState) { _, _ in
    DispatchQueue.main.async {
        self.completeBackgroundTasks()
    }
}
hasContentPendingObservation = WCSession.default.observe(\.hasContentPending) { _, _
    DispatchQueue.main.async {
        self.completeBackgroundTasks()
    }
}
```

On watchOS, the system suspends an app when a person stops using the app and lowers their wrist. Later, when watchOS triggers a background task for the app, watchOS wakes the app from the suspended state. Using Xcode to run an app prevents the system from completing the suspension process, and may lead to different app behaviors. When encountering an issue related to background tasks, consider debugging it by launching the app directly from the Home Screen and analyzing the app logs. This sample uses `Logger` to write logs into a file, and transfers the file to the iOS app when a person taps the file transfer button in the watchOS app.