Extracted from:

Developing for Apple Watch

Your App on Their Wrists

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Pragmatic exPress

Developing for Apple Watch

Your App on Their Wrists

Jeff Kelley

edited by Rebecca Gulick

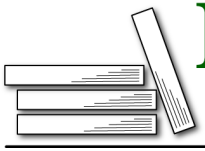
Developing for Apple Watch

Your App on Their Wrists

Jeff Kelley

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *https://pragprog.com*.

The team that produced this book includes:

Rebecca Gulick (editor)
Candace Cunningham (copyeditor)
Dave Thomas (typesetter)
Janet Furlow (producer)
Ellie Callahan (support)

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-068-4
Encoded using the finest acid-free high-entropy binary digits.
Book version: P2.0—July 2015

Sharing Data Between Apps

All iOS apps are sandboxed, and WatchKit extensions are no different. The unparalleled security this offers means that we need to do a little bit of extra work to allow the WatchKit extension and iPhone app to share data; by default, their sandboxes cannot communicate. To solve this, we'll create a new *App Group*, essentially a shared container for file storage for our app. Just like we added the HealthKit capability, head to the TapALap target settings and enable App Groups under Capabilities. Click the + button in the App Groups section to add a new one. We'll provide a name to Xcode's Add A New Container dialog box. Much like the bundle ID, this uses reverse DNS-style names as convention, but with `group` prepended to it; the example for this project would be `group.com.pragprog.tapalap`, but feel free to make it match your bundle ID if it differs. When we have an active group, we'll *also* need to add it to the WatchKit extension target settings. Enable App Groups, and we'll see a check box for the new group. Check it, and we're done.

With App Groups enabled, we can share files as well as use a shared `NSUserDefaults` domain. If your app uses Core Data, you can save your persistent store to this location, allowing you to access it from either place. Obtaining the place to store data is very easy:

```
let manager = NSFileManager.defaultManager()
let group = "group.com.pragprog.tapalap"
let url: NSURL = manager.containerURLForSecurityApplicationGroupIdentifier(group)
```

Once you have the `NSURL` for the container, you can save files to it and read files from it just like you would any other filesystem path. All you need to do is ensure that you pass the correct identifier to `containerURLForSecurityApplicationGroupIdentifier()`. For TapALap, since we're not using Core Data or storing files to disk, we'll go a simpler route and use `NSUserDefaults` to save the data.

If you've used `NSUserDefaults` in the past, you've likely used its `standardUserDefaults()` method to save things to the default location. This works great in the sandbox, but it has the same problem as file storage: calling `standardUserDefaults()` in your iPhone app will save to a *different* user defaults store than calling `standardUserDefaults()` in your WatchKit extension. While this is acceptable for settings that don't apply to other parts of your app, it won't work for saving data and loading it from other parts of our app. To that end, we need to use our App Group to save defaults to a common store. We'll want to update the list of saved runs when ending one, much like saving it to HealthKit, so return to `RunTimerInterfaceController.swift` and add some code to `endRun()`, this time a new method called `saveRunToUserDefaults()`:

tapalap/tapalap WatchKit Extension/RunTimerInterfaceController.swift

```
func endRun() {
    let contexts: [AnyObject]?

    if let lapCount = laps?.count {
        let run = Run(distance: track.lapDistance * Double(lapCount),
            laps: laps!,
            startDate: startDate!)

        saveRunToUserDefaults(run)
        saveRunToHealthKit(run)

        contexts = [NSNull(), run]
    }
    else {
        contexts = nil
    }

    let names = ["GoRunning", "RunLog"]

    WKInterfaceController.reloadRootControllersWithNames(names,
        contexts: contexts)
}
func saveRunToUserDefaults(run: Run) {
    let defaults = NSUserDefaults(suiteName: "group.com.pragprog.tapalap")

    if let defaults = defaults {
        var savedRuns = defaults.objectForKey("Runs") as? [[NSObject: AnyObject]]

        if savedRuns != nil {
            savedRuns!.append(run.dictionaryRepresentation())
        }
        else {
            savedRuns = [run.dictionaryRepresentation()]
        }

        defaults.setObject(savedRuns, forKey: "Runs")
        defaults.synchronize()
    }
}
```

Once again, we'll use the dictionaryRepresentation() method of the Run to convert the run into a Dictionary. The saveRunToUserDefaults() method adds this dictionary to an array of dictionaries in the user defaults—creating an empty array if needed—and then saves the new array back to the user defaults. The next time we run the app, these will persist. Now that we're saving the runs into NSUserDefaults, we can use that data in the iPhone app.

For the iPhone app, a simple table view displaying the user's run history is a good demonstration of saving data and using it in both locations. The ViewController class is as good a place as any for it, so open ViewController.swift and add a runs() computed property to retrieve the runs:

tapalap/tapalap/ViewController.swift
```
var runs: [[NSObject: AnyObject]]? {
    get {
        let defaults = NSUserDefaults(suiteName: "group.com.pragprog.tapalap")!

        return defaults.objectForKey("Runs") as? [[NSObject: AnyObject]]
    }
}
```

This runs() getter simply creates a new NSUserDefaults using the same group ID as the WatchKit app. Next, declare that the ViewController class conforms to the UITableViewDataSource protocol:

tapalap/tapalap/ViewController.swift
```
class ViewController: UIViewController, UITableViewDataSource {
```

We don't need the class to conform to the UITableViewDelegate protocol, as we're merely displaying the data, not interacting with it. To populate the table view with our saved runs from NSUserDefaults, implement the UITableViewDataSource protocol methods as follows:

tapalap/tapalap/ViewController.swift
```
func numberOfSectionsInTableView(tableView: UITableView) -> Int {
    return 1
}

func tableView(tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
    if let count = runs?.count {
        return count
    }
    else {
        return 0
    }
}

func tableView(tableView: UITableView,
    cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {
    let cellIdentifier = "RunCell"

    var cell: UITableViewCell? =
    tableView.dequeueReusableCellWithIdentifier(cellIdentifier) as?
    UITableViewCell

    if cell == nil {
        cell = UITableViewCell(style: .Value1, reuseIdentifier: cellIdentifier)
```

```
    }

    if let runDictionary = runs?[indexPath.row] {
        cell?.textLabel?.text =
            distanceFormatter.stringFromMeters(runDictionary["distance"] as!
                Double)

        cell?.detailTextLabel?.text =
            runDateFormatter.stringFromDate(runDictionary["startDate"] as!
                NSDate)
    }

    return cell!
}
```

Just like in the WatchKit table, we want to reuse our formatters to avoid re-creating them for every cell. Again, we can use lazy properties to create them only as needed:

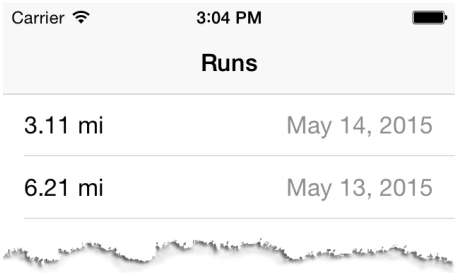
tapalap/tapalap/ViewController.swift

```
lazy var distanceFormatter: NSLengthFormatter = {
    let distanceFormatter = NSLengthFormatter()
    distanceFormatter.numberFormatter.maximumSignificantDigits = 3
    return distanceFormatter
}()

lazy var runDateFormatter: NSDateFormatter = {
    let runDateFormatter = NSDateFormatter()
    runDateFormatter.dateStyle = .MediumStyle
    runDateFormatter.timeStyle = .NoStyle
    return runDateFormatter
}()
```

This is all pretty standard UITableView stuff. Quite different from WKInterfaceTable! As a final step, we need to *create* the table view. Luckily, we can do that entirely in our Main.storyboard file in the iPhone app's group. Drag a new UITableView onto the view controller's view, filling it. Use the Pin button at the bottom of the canvas to pin the top, bottom, left, and right edges of the table view to its parent view. Next, control-drag from the table view to the view controller and connect it to the dataSource outlet. Since we aren't making this table view interactive, we don't need to set the delegate. Finally, to avoid the table view running behind the status bar, select the view controller and then select Editor → Embed In → Navigation Controller. This places a navigation bar at the top of the screen. We can add a title by double-clicking the navigation bar above the table view; I chose Runs.

Build and run the watch app, save some runs, and then build and run the iPhone app. We'll see our runs in the table view we created, as shown.



Just like that, we're sharing data from the watch app to the iPhone app and vice versa. `NSUserDefaults` is a powerful class for sharing simple data like runs and preferences, and App Groups make it easy to share.