Extracted from:

Developing for Apple Watch, Second Edition

Create Native watchOS 2 Apps with the WatchKit SDK

The Pragmatic Bookshelf

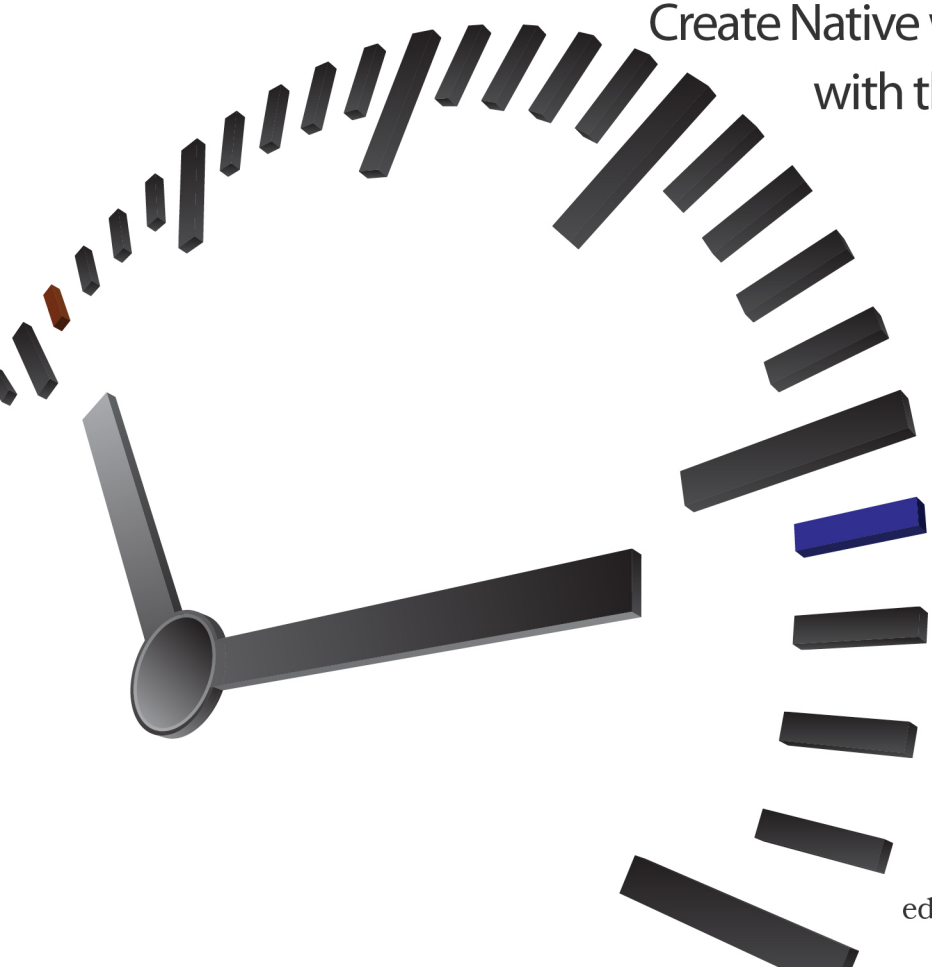
Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Second Edition

Developing for Apple Watch

Create Native watchOS 2 Apps with the WatchKit SDK



Jeff Kelley

edited by Rebecca Gulick

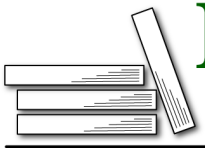
Developing for Apple Watch, Second Edition

Create Native watchOS 2 Apps with the WatchKit SDK

Jeff Kelley

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

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)
Potomac Indexing, LLC (index)
Linda Recktenwald (copyedit)
Gilson Graphics (layout)
Janet Furlow (producer)

For customer support, please contact *support@pragprog.com*.

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

Printed in the United States of America.
ISBN-13: 978-1-68050-133-9
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—May 2016

CHAPTER 5

Delivering Dynamic Content with Tables

Every piece of user interface we've seen so far in WatchKit has something in common: it's static. We create the watch app's UI in Interface Builder, set it up just so, and then use it in the app, but we can't add or remove interface objects. Sure, we can get clever and hide some portions of the UI, but there's one thing we haven't yet seen: creating dynamic user interfaces that respond to users' data. In our iPhone app, we can create these interfaces manually, perhaps placed inside a UIScrollView if there's more than a screenful of content, but it's much more common—and easier—to use either UITableView or UICollectionView. WatchKit targets much smaller screen sizes than UIKit, so it offers us WKInterfaceTable, a simpler, stripped-down version of UIKit's table views.

In this chapter, let's explore tables in WatchKit. We'll cover how they work, how they differ from UIKit, and how to use them the right way. By the end of this chapter, you'll know how to present your data in rows, how to make it perform well, and how to make it look fantastic.

Comparing WatchKit Tables and iOS Table Views

If table views on iOS have one thing going for them, it's that they're routine fodder for introductory materials. Table views are where most developers new to working with Apple platforms encounter the delegation design pattern. Delegation is a routine stumbling block, most often encountered in UITableView. WKInterfaceTable, by contrast, has no datasource to set and no delegate to implement, and—most importantly—it eschews UIKit's "don't call us; we'll call you" paradigm in favor of requiring you to tell it in advance all of the data it'll need to display. If you've never used UITableView and have no idea what I'm talking about, don't worry. You don't need to have used it to understand WKInterfaceTable.

The difference between the two, as with many features of the Apple Watch UI, is that the display is divorced from the data. Your users scroll through

content on the watch, and given its performance concerns, achieving the 60-frames-per-second scrolling users have come to expect from Apple devices requires tradeoffs. Instead of running code every time a new row comes onscreen, as you do with UITableView, you'll populate the entire table at once, allowing the watch's processor to take a break while the user moves the content up and down. This also helps battery life—another paramount concern for Apple Watch apps.

Architectural differences aside, you'll notice as soon as you drag a WKInterfaceTable onto the storyboard that each distinct type of row in the table is nothing more than a WKInterfaceGroup. Unlike UIKit's UITableViewCell, there's no built-in UI element to repurpose. Your table view rows will be fully custom groups, configured with whatever interface objects make sense. To serve as a bridge between your interface controller and your table rows, you'll create objects that coordinate the data, known as *row controllers*. Now, what to put in a table? To introduce using tables to show data, let's create a new screen in TapALap—one for displaying the user's run history in the app.

Row Types and Storyboard Groups

Every WKInterfaceTable delivers content to the user in vertically stacked rows. To demonstrate this, let's create a new interface controller in our running app. Once the user is finished with a run, she'll want to be able to quickly scroll through a history of her runs, comparing distances and paces as she goes. To that end, let's make an interface controller with a table in it that displays one row for each run, with the newest at the top.

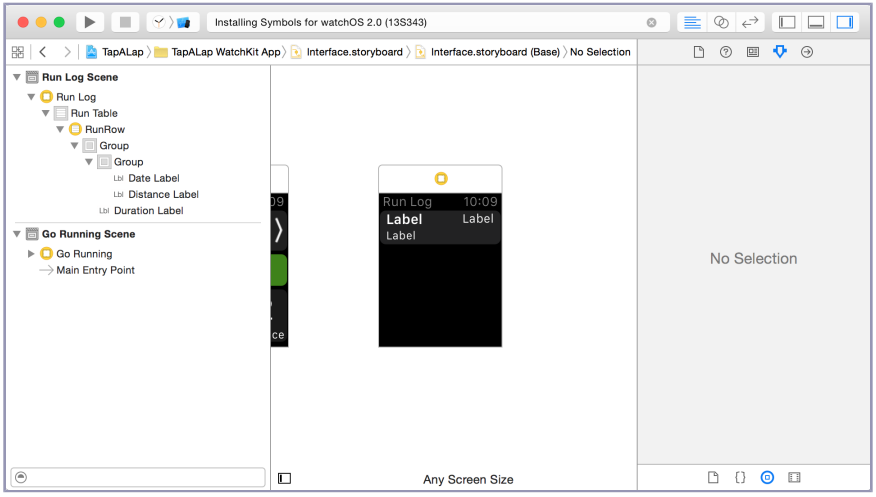
First, open your watch app's storyboard and drag in a new interface controller. Give it the title "Run Log." Next, create a new WKInterfaceController subclass and name it RunLogInterfaceController, being sure to add it to your WatchKit extension and not your iPhone app. You'll come back and implement some methods later on, but for now you just need the class to exist. Head back to the storyboard and select the interface controller you just made. In Xcode's Identity Inspector, under Custom Class, set the Class value to RunLogInterfaceController. Now your storyboard and your subclass are in sync. Finally, drag the Main arrow to the interface controller so the app starts right at that screen.

To implement the UI, you'll need to head over to the Object Library and drag a new WKInterfaceTable, listed simply as Table, onto the interface controller. By default, it has one group inside. Each row type in the table is represented by a group in your storyboard; the table will re-create that group for every row of that type. For this row, you want to show the user the run's date, distance, and duration. You'll use three WKInterfaceLabel objects for it and arrange them

in the group. As you saw in the chapter on groups, you can use subgroups to arrange these labels nicely, and that's what you're going to do.

First, create the three labels by dragging them from Xcode's Object Library to the row group. To help keep track of them as you lay them out, you'll give them custom titles. In the Document Outline of the storyboard (if you don't see it on the left side of the storyboard, click Editor → Show Document Outline), you can press ↩ and change the display name of a selected item. Instead of Label, name these "Date Label," "Distance Label," and "Duration Label." Now you can see what you're doing. The result you want is the date label in large, bold text above the distance label, with the duration label on the right side. To that end, select both date and distance labels and select Editor → Embed In → Vertical Group. The duration label disappears; since the vertical group is by default set to match the width of its container, it pushes everything else away. Select the new group and, in Xcode's Attributes Inspector, change its width to Size To Fit Content. Now you'll see all three on the same screen.

Select the duration label, change its font to the built-in Subhead style, and set its horizontal position to Right. Change the date label's font to the Headline style and the distance label's font to the Footnote style. Now your row is looking pretty good, if a bit stretched out. There's a lot of vertical space, so select the group that contains the left two labels, open the Attributes Inspector, and give it a custom spacing of 0. Now that you've done all this, your interface controller in the storyboard should look like the following image.



Snazzy! After admiring your handiwork for a bit, you can start putting real data into this row and see how it works. To do that, you need to create a row controller object.

Linking Content to the UI with Row Controllers

Looking at the interface object hierarchy in the storyboard's Run Log interface controller (if you don't see it, select Editor → Show Document Outline), you can see that your labels are nested inside a parent group, but that instead of the group being a child of the table (which would match how it appears in the interface editor), there's an intermediate object that looks different, called Table Row Controller. This is the row controller we've been talking about, and it's vital to understanding how tables work in WatchKit.

Row controllers are lightweight objects that sit between tables and row content. Their entire role is to ferry data from your interface controller to the interface objects you've set up in the storyboard. Your row controller is going to be incredibly simple, let's create it now and walk through what it needs to do.

You'll need a new class for the row controller, and since you'll be referencing it from the storyboard, it'll need to inherit from `NSObject`—storyboards use a lot of internal `NSObject` magic. Name the class `RunLogRowController`. Create a new Swift file called `RunLogRowController.swift` using the WatchKit Class template to create the class.

Next, let's set up outlets for your UI elements. You'll make one for each label (there's no need to reference their containing groups for now), and name them the same as they're named in the storyboard:

Chapter 5/TapALap/TapALap WatchKit Extension/RunLogRowController.swift

```
@IBOutlet weak var dateLabel: WKInterfaceLabel!
@IBOutlet weak var distanceLabel: WKInterfaceLabel!
@IBOutlet weak var durationLabel: WKInterfaceLabel!
```

With that set up, you need to connect these outlets to your UI so that you can reference the interface objects in code. Head back to the storyboard and select the row controller object in the Run Log interface controller's Document Outline. In Xcode's Identity Inspector, change its class to `RunLogRowController`. In the Attributes Inspector, set its Identifier to RunRow. If you fail to do this, the table won't be able to create the row, because it uses this identifier to, well, identify it. Now you can link it up with its outlets. For each label, Control-drag from the row controller in the Document Outline to the label, selecting the appropriate outlet.

Your UI is all hooked up and you need to put the appropriate data in it. To keep the row controller class separate from your model objects, the row controller will deal only with values: distance, duration, and date. Add a new method, `configure(date:distance:duration:)`, and it will take care of the UI for you:

Chapter 6/TapALap/TapALap WatchKit Extension/RunLogRowController.swift

```
var dateFormatter: NSDateFormatter?
var distanceFormatter: NSLengthFormatter?
var durationFormatter: NSDateComponentsFormatter?

func configure(date date: NSDate, distance: Double, duration: NSTimeInterval) {
    dateLabel.setText(dateFormatter?.stringFromDate(date))
    distanceLabel.setText(distanceFormatter?.stringFromMeters(distance))
    durationLabel.setText(durationFormatter?.stringFromTimeInterval(duration))
}
```

When a new RunLogRowController is created, the dateFormatter, distanceFormatter, and durationFormatter properties are all nil. These are all NSFormatter subclasses to help you convert numbers to strings, but instead of creating them in this class, you'll fill them in later—that way you can reuse your NSDateFormatter.

Perfect. Whenever you call configure(date:distance:duration:), you'll automatically put the data into your UI. That raises the next question: when does that happen? Let's look at the other side of this and get some data into your table.

Configuring the Content in Tables

Now that you've squared away how your row controller is going to behave, you can use it in your RunLogInterfaceController. Unlike UITableView from UIKit, where you'd return the number of rows from a data source method and then configure each row on demand, you'll set the number of rows manually and then iterate through them and configure them as you go. You'll also configure your NSFormatter subclasses to do formatting and then pass them to each row controller. You'll get everything set up in the interface controller's willActivate() method. First, however, you need the data to put in the rows. Add a new Swift file to your WatchKit extension, and name it Run.swift. In it, create a new class with some properties for a run:

Chapter 5/TapALap/TapALap WatchKit Extension/Run.swift

```
class Run {

    let distance: Double // in meters
    let laps: [NSTimeInterval]
    let startDate: NSDate

    init(distance: Double, laps: [NSTimeInterval], startDate: NSDate) {
        self.distance = distance
        self.laps = laps
        self.startDate = startDate
    }

}
```

Joe asks:
Why Do We Need to Reuse Date Formatters?

Creating a date formatter, according to Apple's Data Formatting Guide, "is not a cheap operation."[a] If we were to create a new date formatter for every row in the table, we'd be sitting here for a long time waiting for a table of any reasonable length to display. Speed is the name of the game with tables (see Performance Concerns, on page ?, for more), so we want our code to be as efficient as possible.

a. From Apple's Data Formatting Guide at developer.apple.com/library/prerelease/watchos/documentation/Cocoa/Conceptual/DataFormatting/Articles/dfDateFormatting10_4.html#//apple_ref/doc/uid/TP40002369-SW10.

You could have made Run a struct instead, but you'll be using it for some class-only functionality in the future, so marking it as a class now avoids future headaches. Now that you have the class to store the data, head back to RunLogInterfaceController.swift and add an array of runs to display:

Chapter 5/TapALap/TapALap WatchKit Extension/RunLogInterfaceController.swift

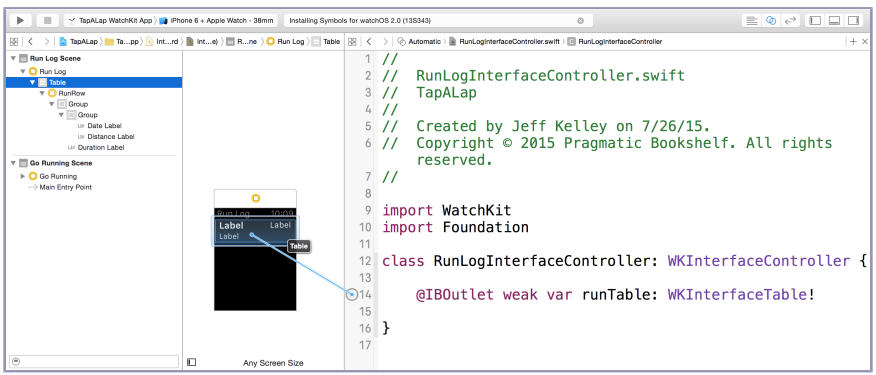
```
var runs: [Run]?
```

Before you can add rows to the table, you need a way to reference it. Open the storyboard and the Assistant Editor, and then connect the table in your Run Log interface controller to a new @IBOutlet property called runTable:

Chapter 5/TapALap/TapALap WatchKit Extension/RunLogInterfaceController.swift

```
@IBOutlet weak var runTable: WKInterfaceTable!
```

Connecting the @IBOutlet is especially easy using the Assistant Editor; simply click the circle to the left of its declaration; then drag to the table in your UI, just as in this image:



You're now ready to display run data in the table. In the interface controller's willActivate() method, you'll iterate over the runs array:

Chapter 5/TapALap/TapALap WatchKit Extension/RunLogInterfaceController.swift

```
Line 1  override func willActivate() {
        -     super.willActivate()
        -
        -     guard let runs = runs else { return }
        5
        -     runTable.setNumberOfRows(runs.count, withRowType: "RunRow")
        -
        -     for i in 0 ..< runTable.numberOfRows {
        -         guard let rowController = runTable.rowControllerAtIndex(i)
        10            as? RunLogRowController else { continue }
        -
        -         configureRow(rowController, forRun: runs[i])
        -     }
        - }
```

You get started with the table on line 6, when you set the number of rows to the number of runs in your runs array. Then you get to a for loop on line 8, and this is where the real fun begins. The table tells you how many row controllers it created with its numberOfRows() method, so you can loop through them. Inside each loop, you call the table's rowControllerAtIndex() method to get the row controller it's created for the row, and then you can modify it—but before you can use it, you need to make sure it's the right class, because rowControllerAtIndex() returns AnyObject?.

To do that, you'll create a method called configureRow(_:forRun:). In that method, you simply pass in your formatters and the proper Run instance, and the code you wrote in the section on row configuration, on page 9, runs to configure the row. Here's what the method should look like:

Chapter 6/TapALap/TapALap WatchKit Extension/RunLogInterfaceController.swift

```
func configureRow(rowController: RunLogRowController, forRun run: Run) {
    rowController.dateFormatter = dateFormatter
    rowController.distanceFormatter = distanceFormatter
    rowController.durationFormatter = durationFormatter

    rowController.configure(date: run.startDate,
        distance: run.distance,
        duration: run.duration)
}
```

That method was easy. If you try to build, you'll notice that you never declared the formatters, so let's do that now. You don't need them to be created until you actually use them, so lazy variables in Swift are a perfect match. Head up to the class extension and add some declarations for them:

Chapter 5/TapALap/TapALap WatchKit Extension/RunLogInterfaceController.swift

```
lazy var dateFormatter: NSDateFormatter = {
    let dateFormatter = NSDateFormatter()
    dateFormatter.dateStyle = .ShortStyle
    return dateFormatter
}()

lazy var distanceFormatter = NSLengthFormatter()

lazy var durationFormatter: NSDateComponentsFormatter = {
    let dateComponentsFormatter = NSDateComponentsFormatter()
    dateComponentsFormatter.unitsStyle = .Positional
    return dateComponentsFormatter
}()
```

Now that you've declared them, you can use them whenever they're needed, and because they're marked lazy, they'll be initialized the first time they're used. Note that two of them need more customization, so wrap the customizations in a closure, using its return value as the initialized value of the property. Next, you need a duration property for the Run class, which you can create as a computed property. The duration of the run is the same as the sum of all of its laps, so you can use Swift's reduce() method to add them:

Chapter 5/TapALap/TapALap WatchKit Extension/Run.swift

```
var duration: NSTimeInterval {
    return laps.reduce(0, combine: +)
}
```

You're finished! Build and run the app, and the content of your runs array will be displayed in the run log. Only there isn't anything in that array. For now, you can add some test data:

Chapter 5/TapALap/TapALap WatchKit Extension/RunLogInterfaceController.swift

```
override init() {
    srand48(time(UnsafeMutablePointer<time_t>(bitPattern: 0)))

    let randomRun: (NSDate) -> Run = { date in
        let lapCount = arc4random_uniform(20) + 5
        let lapDistance = arc4random_uniform(1000) + 1

        var laps: [NSTimeInterval] = []

        for _ in 0 ..< lapCount {
            // Pace is in m/s. 9 minutes per mile is about 3 meters per second.
            // Generate a random pace between ~7.5 min/mi and ~10.5 min/mi.
            let speed = 3.0 + (drand48() - 0.5) // in meters per second
            let lapDuration: NSTimeInterval = Double(lapDistance) / speed

            laps.append(lapDuration)
        }
```

```
        let run = Run(distance: Double(lapDistance * lapCount),
            laps: laps,
            startDate: date)

        return run
    }

    runs = []

    for i in 0 ..< 5 {
        runs?.append(randomRun(NSDate().dateByAddingTimeInterval(Double(i)
            * 24 * 60 * 60)))
    }
}
```

For real this time, you can display data. Build and run one more time and you'll see something like this:

As you can see, getting your data into the table is pretty straightforward. If you needed to use multiple types of rows with different row controllers, you'd use the table's setRowTypes() method instead of setNumberOfRows(_:withRowType:). To use that method, simply create an array of strings, one for each row in the table, set to the row type identifier. If you do that, keep in mind that the row controller class you get back from your table's rowControllerAtIndex() method will depend on that identifier! In your willActivate() you can assume you're getting a RunLogRowController, but in more complicated table layouts you won't be so lucky.

Another reason this code is so straightforward is that you have static content. You create the table in the storyboard and set the number of rows in your interface controller's code, and then you're finished with it. The watch takes care of scrolling for you, and everything is smooth as butter. If you need to *modify* the contents of the table, things get a bit trickier.