Extracted from:

iOS 9 SDK Development

Creating iPhone and iPad Apps with Swift

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

iOS 9 SDK Development

Creating iPhone and iPad Apps with Swift

Chris Adamson
with Janie Clayton

edited by Rebecca Gulick

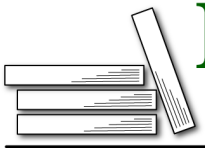
iOS 9 SDK Development

Creating iPhone and iPad Apps with Swift

Chris Adamson with Janie Clayton

The Pragmatic Bookshelf

Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Rebecca Gulick (editor)
Potomac Indexing, LLC (index)
Liz Welch (copyedit)
Dave Thomas (layout)
Janet Furlow (producer)

For sales, volume licensing, and support, please contact *support@pragprog.com*.

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

Printed in the United States of America.
ISBN-13: 978-1-68050-132-2
Encoded using the finest acid-free high-entropy binary digits.
Book version: P2.0—August 2016

CHAPTER 7

Working with Tables

For organizing and presenting many of the kinds of data we see in iPhone and iPad apps, it's hard to beat a table view. Thanks to the intuitive flick-scrolling provided by iOS, it's comfortable and convenient to whip through lists of items to find just the thing we need, with each item visually presented in whatever way makes sense for the app. In many apps, the table view is the bedrock of the app's presentation and organization.

In this chapter, we're going start turning our Twitter application into one that's based around a table view. However, it's going to take us a few chapters to completely move away from the web view. First, we'll put some fake data into a table view, and then in the following chapters we'll get real data from the Twitter API and load it into the table view.

Tables on iOS

Coming from the desktop, one might expect a `UITableView` to look something like a spreadsheet, with rows and columns presented in a two-dimensional grid. Instead, the table view is a vertically scrolling list of items, optionally split into sections.

The table view is essential for many of the apps that ship with the iPhone, as well as popular third-party apps. In Mail, tables are used for the list of accounts, the mailboxes within each account, and the contents of each mailbox. The Reminders app is little more than a table view with some editing features, as are the alarms in the Clock app. The Music app shows lists of artists or albums, and within them lists of songs. Even the Settings app is built around a table, albeit one of a different style than is used in most apps (more on that later).

And while our Twitter app currently displays a web view of all the tweets we've parsed, pretty much every Twitter app out there (including the official Twitter app, as well as *Twitterrific*, *Tweetbot*, and *Echofon*) uses a table view to present tweets.

So our task now is to switch from the web view to a table view–based presentation of the tweets. We'll build this up slowly, as our understanding of tables and what they can do for us develops.

Table Classes

To add a table to an iOS app, we use an instance of `UITableView`. This is a `UIScrollView` subclass, itself a subclass of `UIView`, so it can either be a full-screen view unto itself or embedded as the child of another view. It cannot, however, have arbitrary subviews added to it, as it uses its subviews to present the individual cells within the table view.

The table has two properties that are crucial for it to actually do anything. The most important is the `dataSource`, which is an object that implements the `UITableViewDataSource` protocol. This protocol defines methods that tell the table how many sections it has (and optionally what their titles are) and how many rows are in a given section, and provides a cell view for a given section-row pair. The data source also has editing methods that allow for the addition, deletion, or reordering of table contents. There's also a `delegate`, an object implementing the `UITableViewDelegate` protocol, which provides method definitions for handling selection of rows and other user interface events.

These roles are performed not by the table itself—whose only responsibility is presenting the data and tracking user gestures like scrolling and selection—but by some other object, often a view controller. Typically, there are two approaches to wiring up a table to its contents:

- Have a `UIViewController` implement the `UITableViewDataSource` and `UITableViewDelegate` protocols.
- Use a `UITableViewController`, a subclass of the `UIViewController` that is also defined as implementing the `UITableViewDataSource` and `UITableViewDelegate` protocols

It's helpful to use the second approach when the *only* view presented by the controller is a table, as this gives us some nice additional functionality like built-in pull-to-refresh, or scrolling to the top when the status bar is tapped. But if the table is just a subview, and the main view has other subviews like buttons or a heads-up view, then we need to use the first approach instead.

Model-View-Controller

The careful apportioning of responsibilities between the view class and the controller comes from UIKit's use of the *model-view-controller* design pattern, or MVC. The idea of this design is to split out three distinct responsibilities of our UI:

- *Model*—The data to be presented, such as the array of tweets
- *View*—The user interface object, like a text view or a table
- *Controller*—The logic that connects the model and the view, such as how to fill in the rows of the table, and what to do when a row is tapped

This pattern explains why the class we've been doing most of our work in is a "view controller"; as a controller, it provides the logic that populates an onscreen view, and updates its state in reaction to user interface events. Notice that it is not necessary for each member of the design to be have its own class: the view is an object we created in the storyboard, and the model can be a simple object like an array. At this point in our app's evolution, only the controller currently requires a custom class. Still, some developers prefer the clarity of each role having its own class, so sometimes you'll see a class that exists only to implement UITableViewDataSource for a given table.

Creating and Connecting Tables

We're going to need to make some major changes to our user interface to switch to a table-driven approach. In fact, we're going to blow away our original view entirely. We'll get all our functionality back eventually, and we'll be in a better position to build out deeper and more interesting features. Eventually, we'll have an app that looks and feels like a real Twitter client.

We'll start by preparing our view controller to supply the table data. We can do this by either declaring that we implement UITableViewDataSource, or by becoming a subclass of UITableViewController. Since the table will be the only thing in this view, let's do the latter. In ViewController.swift, rewrite the declaration like this:

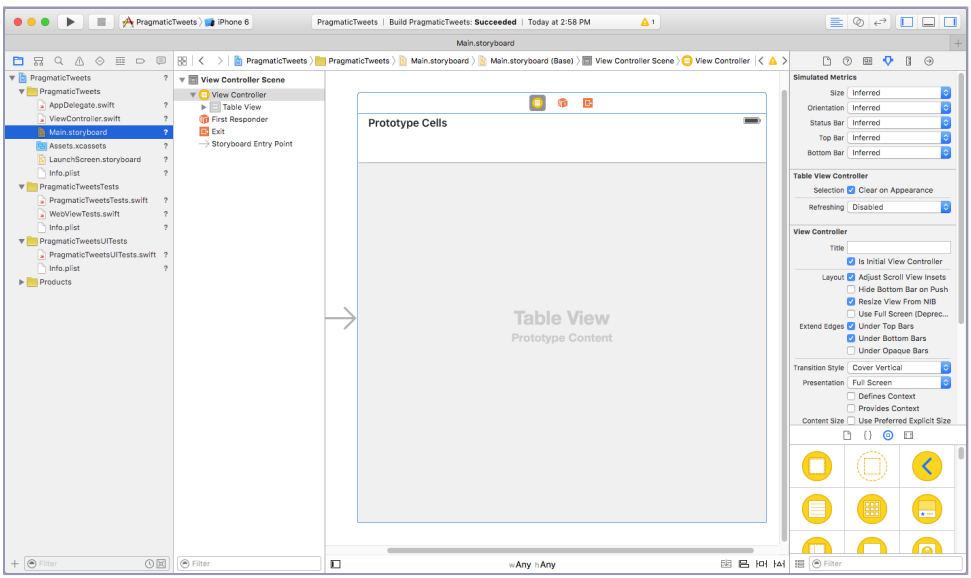
tables/PragmaticTweets-7-1/PragmaticTweets/ViewController.swift

```
class ViewController: UITableViewController {
```

Adding a Table View to the Storyboard

Now switch to Main.storyboard and look through the Object area at the bottom right for the Table View Controller object, shown in this figure. Drag one into the storyboard, anywhere where it won't collide with the existing view controller. This adds a new Table View Controller Scene to the list of scenes in the storyboard.

Select the view controller from the previously existing scene and press ⌫ to delete the old scene. This leaves the storyboard with no entry point. Select the Table View Controller, bring up its Attributes Inspector (⌥⌘4), and select the Is Initial View Controller check box. The view gets an arrow on its left side, showing our app once again has a place to start. The view itself shows a status bar that says Prototype Cells above a Table View that has a single Table View Cell as a subview, as seen in the following figure:



We can run this app…but it shows an empty table! That's because the table is not yet connected to a data source that can provide it with cells or even a count of how many sections and rows there are. Let's get to work on that.

Providing a Temporary Table Data Source

As it is, the table in the storyboard doesn't know to use our class; it expects to create a generic `UITableViewController` for the table. We want it to use our `ViewController` instead. So, while still in `Main.storyboard`, choose the Table View Controller and visit its Identity Inspector in the right-side pane (⌥⌘3). In the Custom Class section, for the Class, enter `ViewController`. This should autocomplete, since we declared that our `ViewController` class is a valid `UITableViewController`, although we've done nothing to implement that behavior yet.

While here, Control-click on the table view, or visit its Connections Inspector (⌥⌘6), and notice that table view's connections to the `dataSource` and `delegate` properties are already wired up, connected to the view controller.

As a warm-up, let's provide a trivial implementation of the data source methods, just to ensure the new storyboard and its connections are good to go. To do this, our data source needs to provide a minimum of three things: the number of sections, the number of rows in a given section, and a cell for a given section and row. In ViewController.swift, provide the following trivial implementations of the UITableViewDataSource methods numberOfSectionsInTableView(), tableView(numberOfRowsInSection:), and tableView(cellForRowAtIndexPath:), as well as the optional tableView(titleForHeaderInSection:), which will let us see the section breaks.

tables/PragmaticTweets-7-1/PragmaticTweets/ViewController.swift

```
override func numberOfSectionsInTableView(tableView: UITableView)
  -> Int {
    return 5
}

override func tableView(tableView: UITableView,
  titleForHeaderInSection section: Int) -> String? {
    return "Section \(section)"
}

override func tableView(tableView: UITableView,
  numberOfRowsInSection section: Int) -> Int {
    return section + 1
}

override func tableView(tableView: UITableView,
  cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {
    let cell = UITableViewCell(style: .Default, reuseIdentifier: nil)
    cell.textLabel?.text = "Row \(indexPath.row)"
    return cell
}
```

Notice that in our quick-and-dirty table code, three of our methods are called tableView(). The reason these methods don't get confused with one another is because they're differentiated by their named parameters: one takes titleForHeaderInSection, another takes cellForRowAtIndexPath, and so on.

By convention, all these methods take the table view in question as their first argument, so if we had multiple tables, a method would be able to figure out which table it's working with.

But as for why it has to be the *first* parameter, that's more of a legacy of Objective-C, where it was somewhat more natural to incorporate the name of your first parameter into the method name, and differentiate with the rest of the parameters. Swift came later, so we're stuck with the old naming schemes, at least for now.

In this book, when we encounter cases where the method name by itself isn't unique, we'll include the parameters for clarity. That way, we'll call out the

difference between tableView(numberOfRowsInSection:) and tableView(cellForRowAtIndexPath:), but we won't feel the need to write viewWillAppear(animated:) when there's only one method that starts like that, so it can be written as just viewWillAppear().

Chained Optionals

One other new thing to notice in the tableView(cellForRowAtIndexPath:) implementation is this line:

```
cell.textLabel?.text = "Row \(indexPath.row)"
```

This particular use of the ? operator is new, and quite handy. The textLabel property of UITableViewCell is an optional, so ordinarily we would want to test it with an if let or guard let. But in the middle of a chain of dot accessors, this is burdensome. And to just force-unwrap with ! would be dangerous.

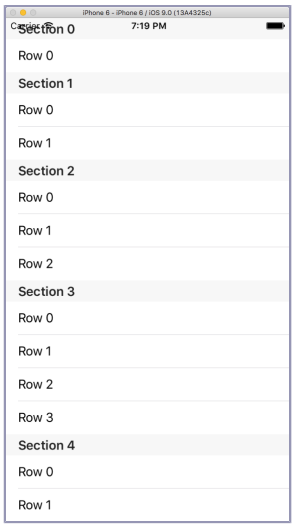
One alternative is to use the ? right before a dot operator. This syntax is called the *chained optional*, and it works like this: the expression is evaluated left-to-right, and all optionals marked with ? are tested against nil. If any optional is nil, processing stops and the whole expression evaluates to nil. In an assignment like this, it's OK for the left side to be nil, because assigning a value to nil (instead of to a real variable) just quietly does nothing.

If none of the optionals are nil, then we can get the value at the end of the chain, albeit with one caveat: its type becomes optional, even if the last type in the chain wasn't optional. Again, that's fine here, because the text property of the textLabel is also an optional type: String?.

Lots of the changes since Swift 1.0 have made dealing with optionals easier. This is one we'll get a lot of mileage out of.

Anyway, while we're in the ViewController.swift file, let's delete the line that declares the twitterWebView that no longer exists, and all of the handleShowMyTweetsTapped() methods that populated it. We won't need those anymore. Also, delete the contents of reloadTweets(), but leave the method definition; we'll rebuild that one shortly. Finally, with no twitterWebView, there's no need for the WebViewTests test class, so delete that entire file.

In this implementation, we are telling the table that there are five sections, that each section has one more row than the section index (that is to say, there's one row in section 0, two rows in section 1, etc.), and that any time a new cell is needed, it should create a new UITableViewCell, get its textLabel property (a UILabel), and set the text property of the label to a string that shows the row number. When run, the table will look like the figure on page 125.



You may be wondering why the status bar overlaps the table. This is one of the more controversial aspects of the iOS 7 visual design—view controllers default into a full-screen mode. In fact, the property `wantsFullScreenLayout` was deprecated in iOS 7, and since then view controllers are assumed to *always* fill the screen with their views, even the space under the status bar.

It looks horrible at first, but the idea is that once we start scrolling and see content go under the status bar, the transparency of the status bar gives us a visual cue about information that is about to come fully into view. In later chapters, we'll add a navigation bar at the top and then it'll look and feel a lot better.

Notice that `tableView(cellForRowAtIndexPath:)` passes in an `NSIndexPath`. This is a class originally intended for representing paths in tree structures, things like "the third child of the second child of the root node." In iOS, it is pressed into service representing table entries. `NSIndexPath` is extended to add the properties `section` and `row` (which are implemented as just the first and second entries in the path), and this combination of section and row can uniquely identify any cell in `UITableView`.

Now we have a table and a way to get data into it. What we need to do next is provide a nontrivial implementation of the data source, one that actually shows some tweets.