Chapter 8
Creating a Simple Table Based App

> That's been one of my mantras - Focus and Simplicity. Simple can be harder than complex: You have to work hard to get your thinking clean to make it simple. But it's worth it in the end because once you get there, you can move mountains.
>
> - Steve Jobs

Now that you have a basic understanding of prototyping and our demo app, we'll work on something more interesting and build a simple table-based app using UITableView in this chapter. Once you master the technique and the table view customization (to be discussed in the next chapter), we'll start to build the Food Pin app.

First of all, what exactly is a table view in an iPhone app? A table view is one of the most common UI elements in iOS apps. Most apps (except games), in some ways, make use of table view to display content. The best example is the built-in Phone app. Your contacts are displayed in a table view. Another example is the Mail app. It uses table view to display your mail boxes and emails. Not only designed for listing textual data, table view allows you to present the data in the form of images. The TED, Google+ and Airbnb are also good examples. Figure 8-1 displays a few more sample table-based apps. Though they look different, all in some ways utilize a table view.

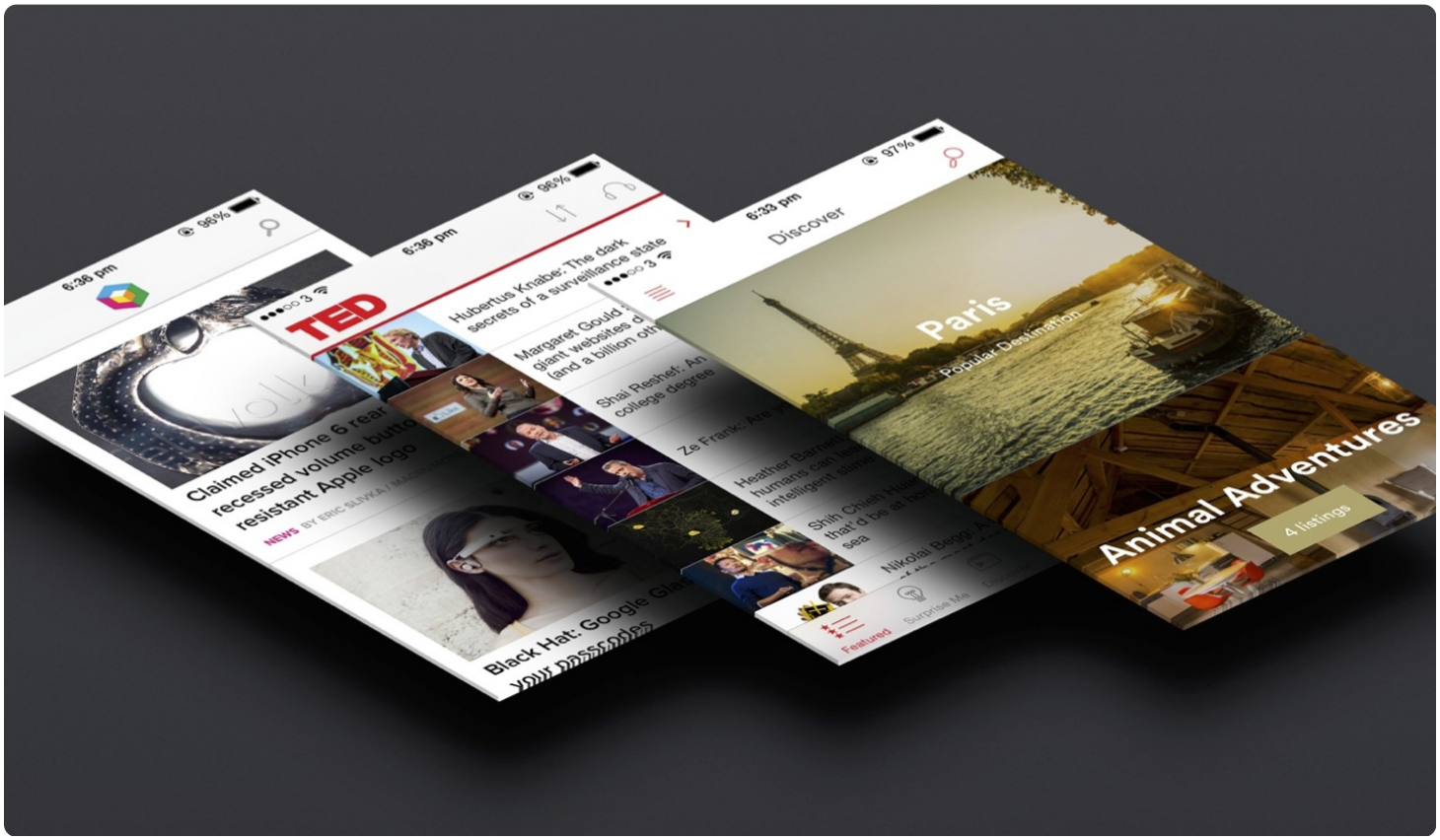

Figure 8-1. Sample table-based apps

Creating a SimpleTable Project

> Don’t just read the book. If you’re serious about learning iOS programming, stop reading. Open your Xcode and code! This is the best way to learn programming.

Let's get started and create a simple app. The app is really simple. We'll just display a list of restaurants in a plain table view. We'll polish it in the next chapter. If you haven't fired up Xcode, launch it, and create a new project using the "Single View application" template.

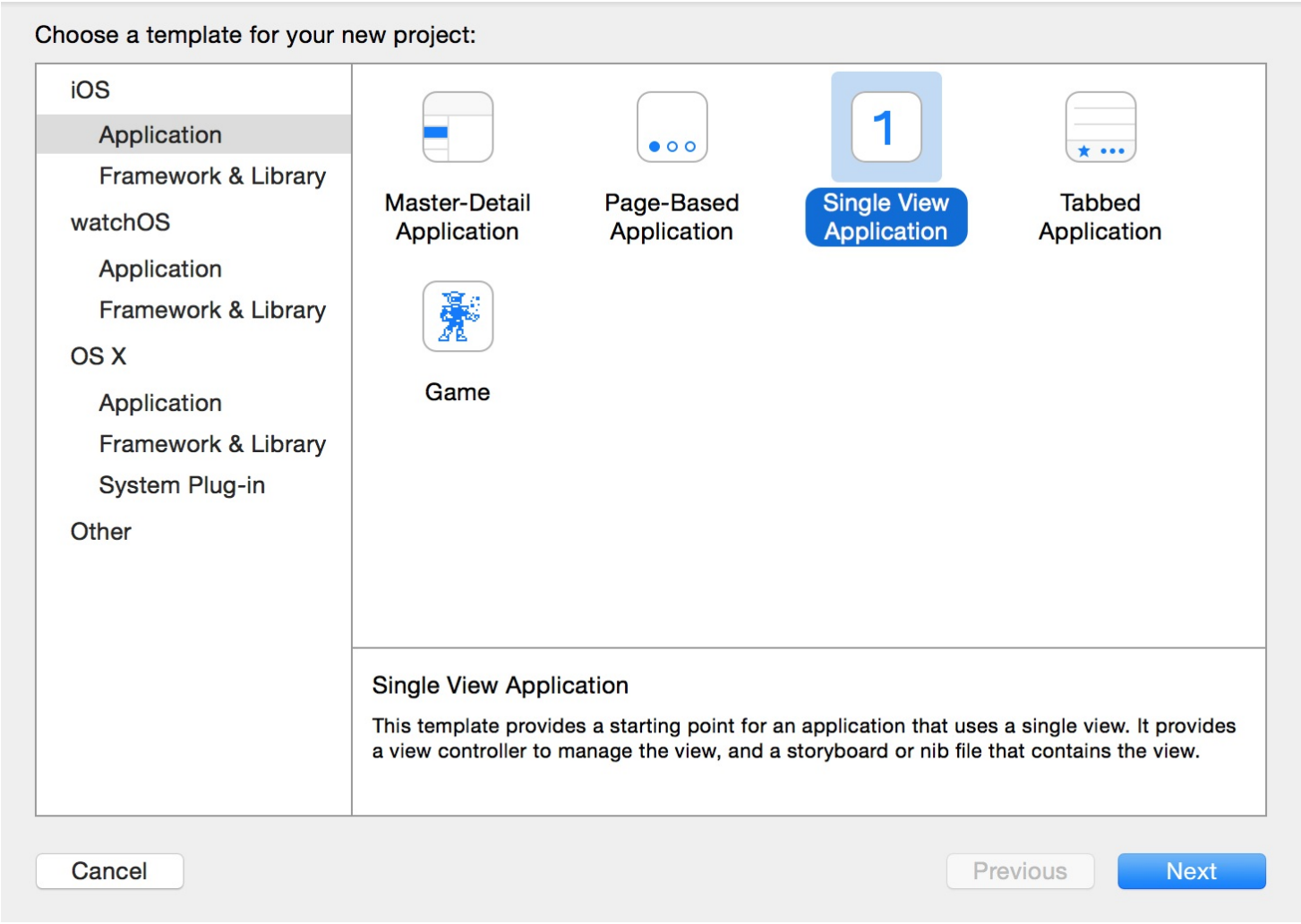

Figure 8-2. Xcode Project Template

Click "Next" to continue. Again, fill in all the required options for the Xcode project:

- **Product Name: SimpleTable** – This is the name of your app.
- **Organization Name: AppCoda** – It's the name of your organization.
- **Organization Identifier: com.appcoda** – It's actually the domain name written the other way round. If you have a domain, you can use your own domain * name. Otherwise, you may use "com.appcoda" or just fill in "edu.self".
- **Bundle Identifier: com.appcoda.StackViewDemo** - It's a unique identifier of your app, which is used during app submission. You do not need to fill in this option. Xcode automatically generates it for you.
- **Language: Swift** – We'll use Swift to develop the project.
- **Devices: iPhone** – Select "iPhone" for this project.

- **Use Core Data: [unchecked]** – Do not select this option. You do not need Core Data for this simple project.
- **Include Unit Tests: [unchecked]** – Do not select this option. You do not need unit tests for this simple project.
- **Include UI Tests: [unchecked]** – Do not select this option. You do not need UI tests for this simple project.

Click "Next" to continue. Xcode then asks you where to save the *SimpleTable* project. Pick a folder on your Mac. Click "Create" to continue.

User Interface Design

To present data in a table in an iOS app, all you need to use is the table view object. First, select `Main.storyboard` to switch to the Interface Builder editor. In the Object library, look for the `Table View` object and drag it into the view.

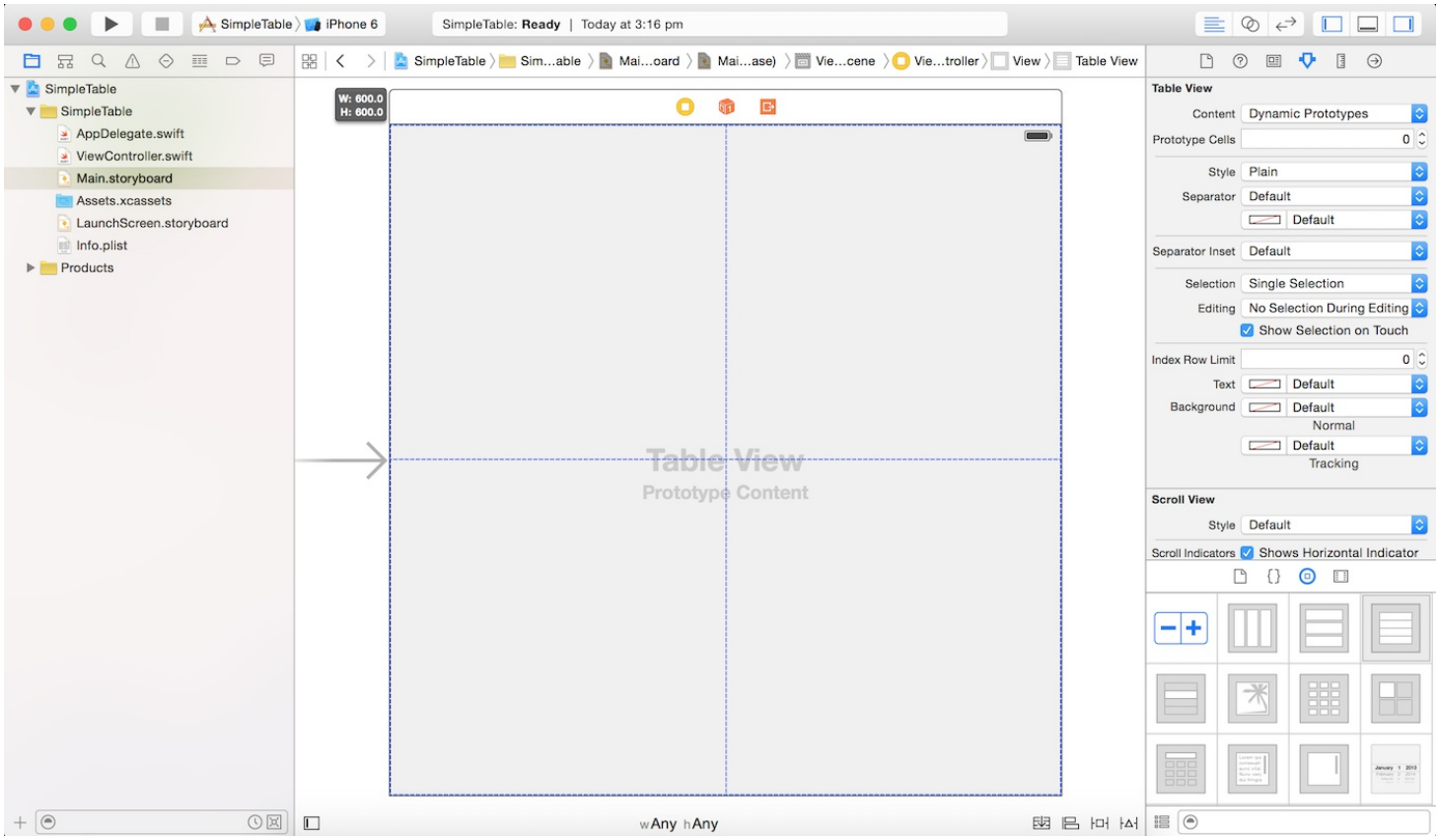

Figure 8-3. Drag table view from the Object library to the View

Select the table view. In the Attributes inspector (if it doesn't appear in your Xcode, choose View > Utilities > Show Attributes Inspector), change the number of Prototype Cells from 0 to 1. A prototype cell will appear in the table view. Prototype cells allow you to easily design the layout of your table view cell. It also comes with several standard cell styles including basic, right detail, left detail and subtitle for you to choose from. In this example, we will just use the basic style. For cell customizations, I will leave it to the next chapter.

Select the cell and open the Attributes inspector. Change the cell style to *Basic*. This style is good enough to display a cell with both text and image. Additionally, set the identifier to *Cell*. This is a unique key for identifying the prototype cell. We'll use it later in the code.

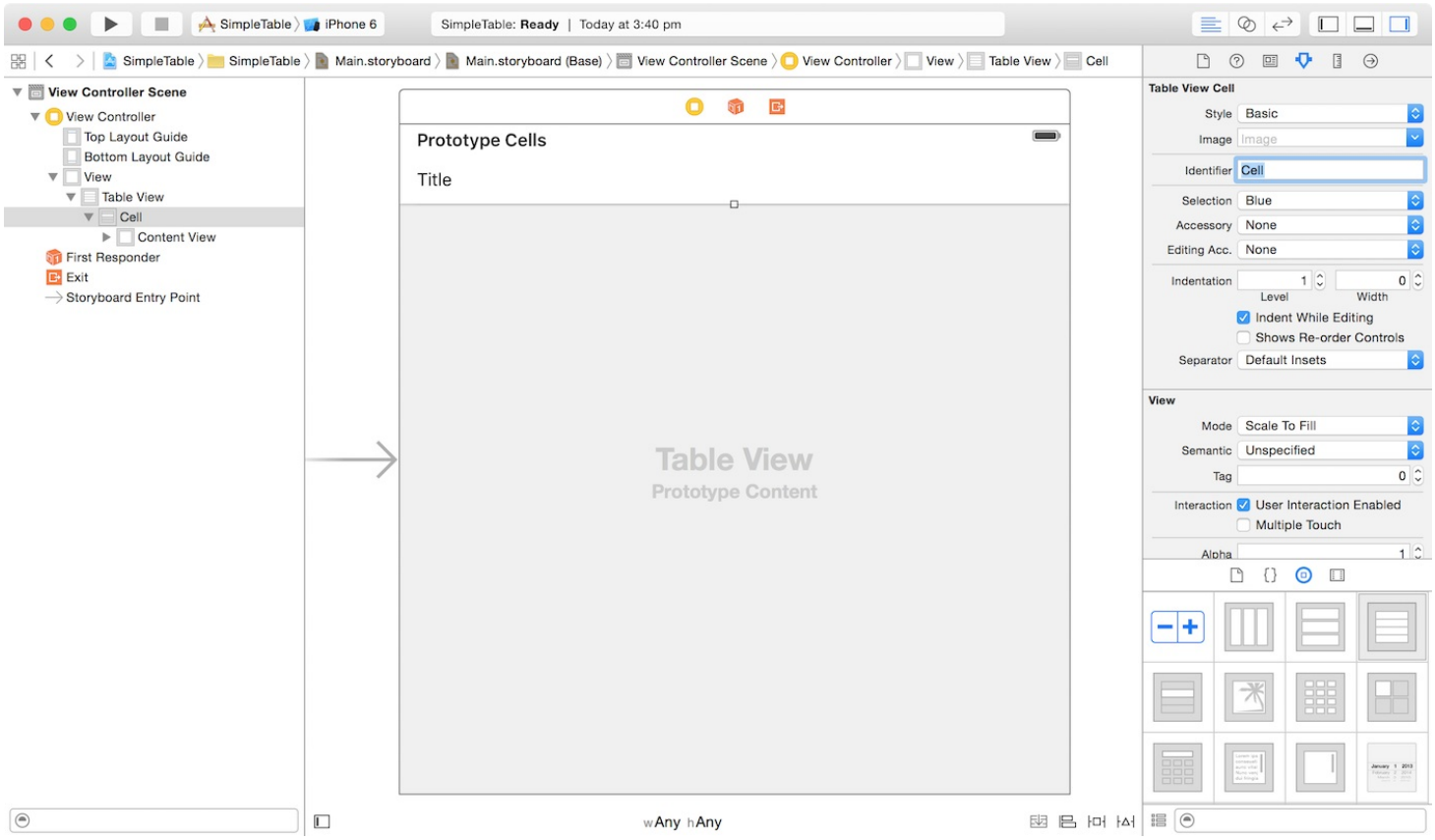

Figure 8-4. Prototype Cell in table view using Basic style

Run Your App Without Any Coding

Before moving on, try to run your app using the Simulator. Click the "Run" button to build your app and test it. The Simulator screen will look like the one in figure 8-5.

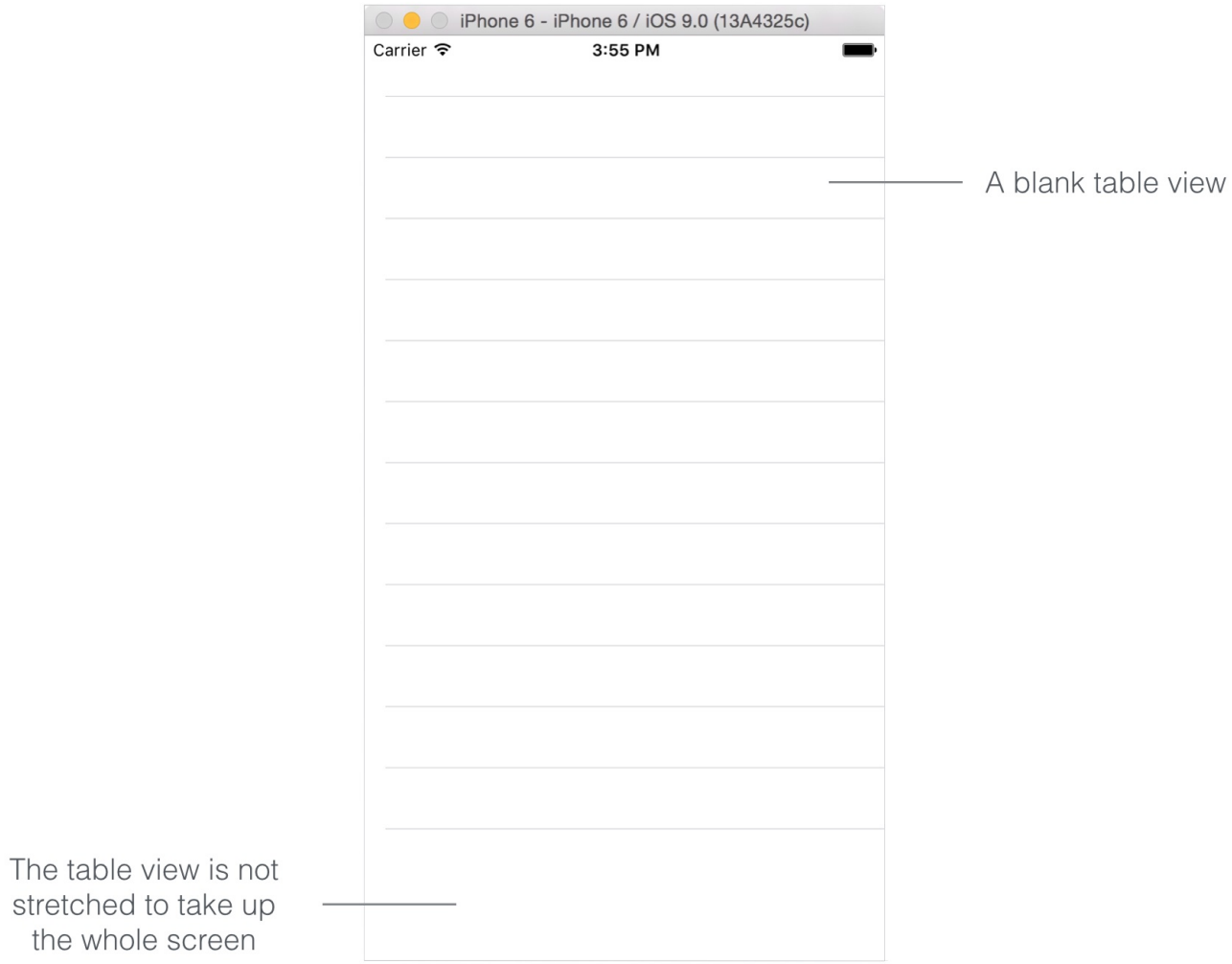

Figure 8-5. A blank table view

Easy, right? You have created the table view for your app. Meanwhile, it doesn't display any data. If you take a closer look at the table view, it is not perfectly stretched to take up the whole screen. I believe you should know the reason if you thoroughly understand auto layout.

So far we haven't defined any layout constraints for the table view. This is why it is not displayed properly. Now select the table view in Interface Builder, click the Pin button in the layout bar. Set the spacing of the top, left, right and bottom side

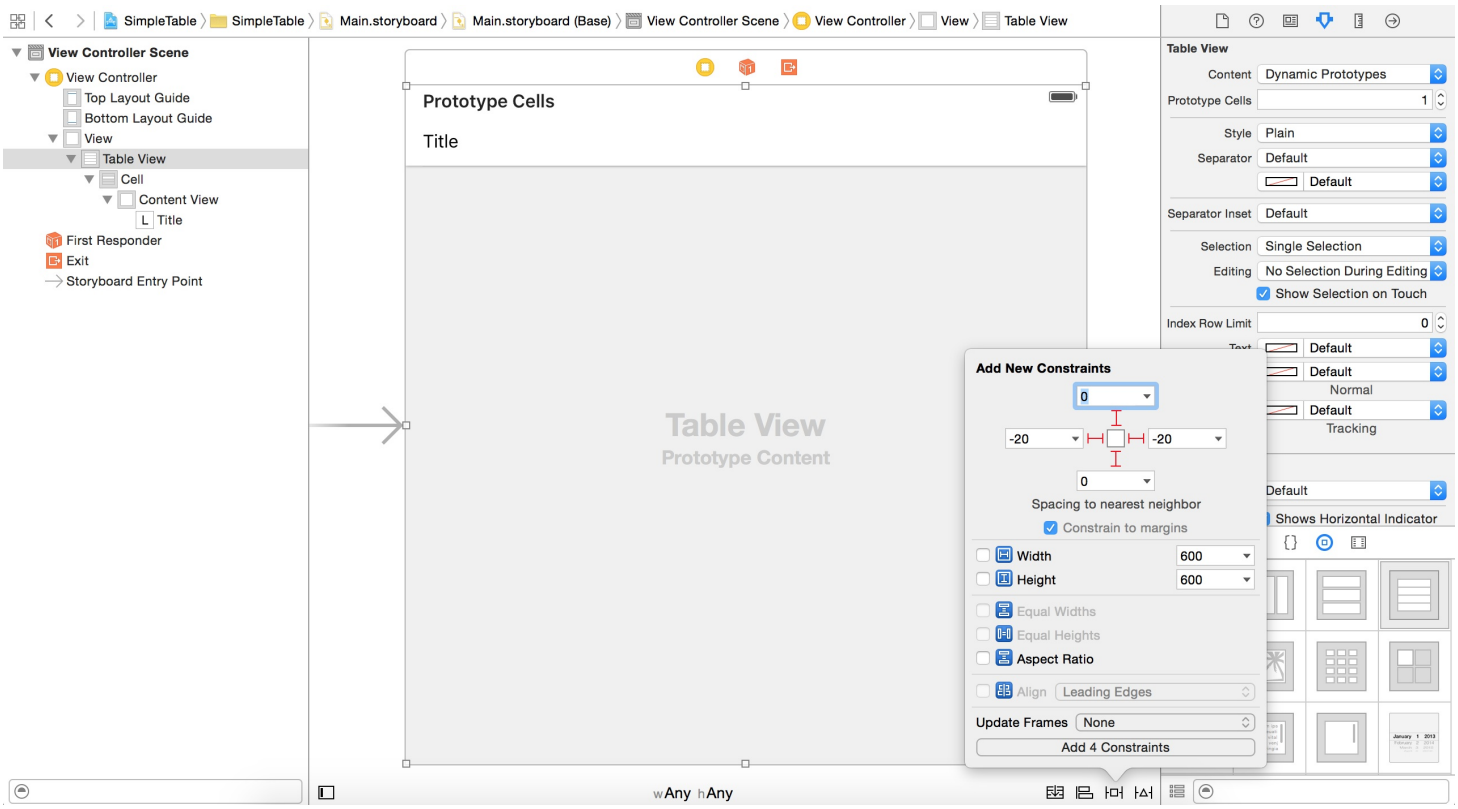

Figure 8-6. Adding layout constraint to the table view

Select each of the dashed red line symbols. Once selected, the dashed red line turns to a solid red line. Click the "Add 4 Constraints" button to add the constraints. Here we define 4 spacing constraints for each side of the table view. Here, we ensure that there is no space between the bottom of the UITableView and the Bottom Layout Guide. The top of UITableView is now aligned to the top of the view. The two horizontal space constraints ensures that the left & right sides of the table view will stretch to the edge of the view. In other words, your table view will automatically resize to fit all displays.

You can run the project again. The table view should now support all screen sizes. With the UI design ready, let's move onto the core part and write some code to insert the table data.

UITableView and Protocols

I mentioned before that we deal with foundation classes provided by the iOS SDK. These classes are organized into what-so-called *frameworks*. The UIKit framework is one of the most commonly used frameworks.

It provides classes to construct and manage your app's user interface. All objects listed in the Object library of Interface Builder are provided by the framework. The Button object you used in the HelloWorld app, and the Table View object we're now working on are from the UIKit framework. While we use the term *Table View*, the actual class is *UITableView*. Simply put, every UI component in the Object libary has a corresponding class. You can click any item in the Object Library and reveal the actual class name in the pop-over menu.

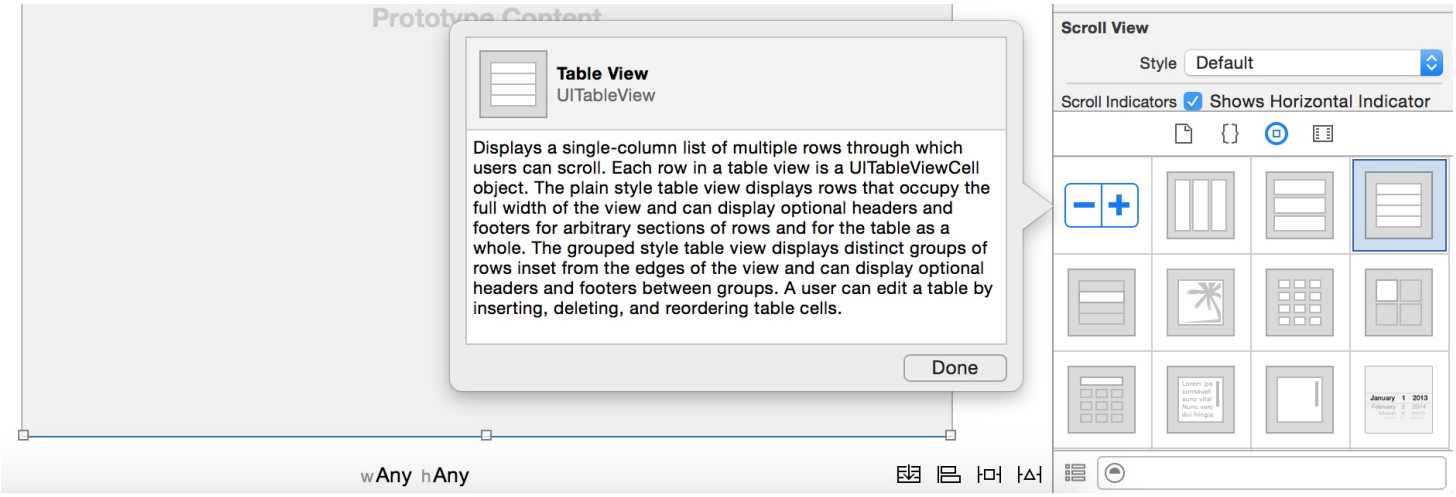

Figure 8-7. Click the object in the Object library to reveal the actual class name in UIKit framework

> I intentionally leave out the discussion of classes and methods to later chapters. If you have no ideas what a class is, don't worry. Just think of it as a code template. I will explain it to you in later chapters.

Now that you have an idea about the relationship between Table View and UITableView. We'll write some code to insert the table data. Select `ViewController.swift` in the project navigator to open the file in the editor pane. Append `, UITableViewDataSource, UITableViewDelegate` after `UIViewController` to adopt the protocols.

As soon as you insert the code after `UIViewController`, Xcode detects an error. The red exclamation mark indicates that there is a problem. Click the little exclamation mark on the left side of the editor. Xcode will highlight the line of code, and display a message telling you the details of the issue. The message is helpful showing the cause of the problem, but it doesn't show you the solution.

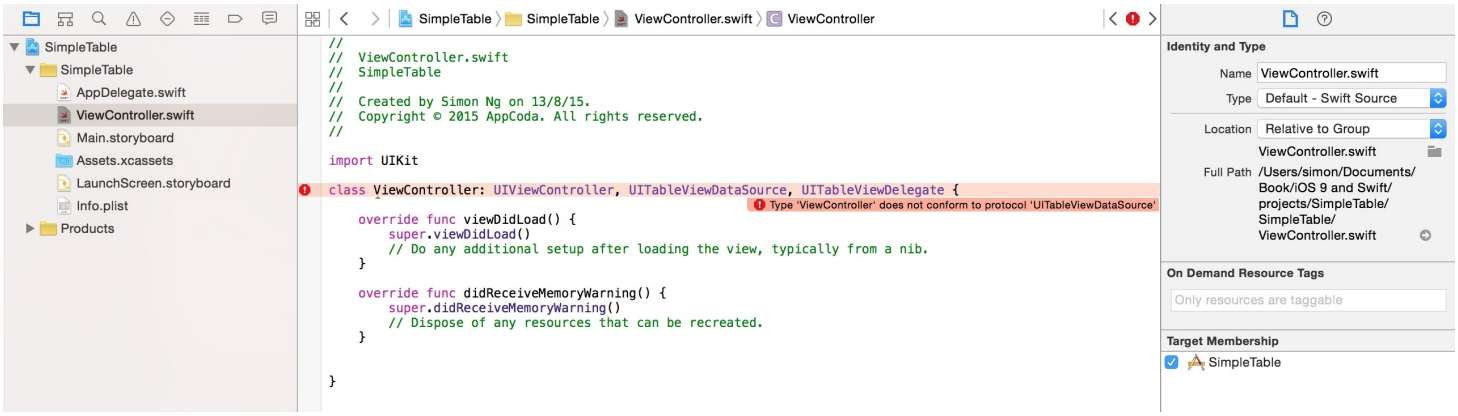

Figure 8-8. Implementing UITableViewDataSource and UITableViewDelegate protocols

So, what does it mean by "Type ViewController does not conform to protocol UITableViewDataSource"?

The `UITableViewDelegate` and `UITableViewDataSource` are known as protocols in Swift. To display data in a table view, we have to conform to a set of requirements, defined in the protocols. Here, the `ViewController` class is the one that adopts the protocol, and implements all the mandatory methods.

It may be confusing. What are these protocols? Why protocols?

Well, let's say you're starting a new business. You hire a graphic designer to design your company logo. He's a skilled designer capable of creating any kind of logo. But he can't start the logo design right away. In the least, you need to provide him with some requirements such as company name, color preference, business nature before he can create a logo. However, you're busy. You delegate the task to your personal assistant, and let her deal with the designer, providing him with the logo requirements.

In iOS programming, the `UITableView` class is just like the graphic designer. It's flexible enough to display various kinds of data (e.g. image, text) in table form. You can use it to display a list of countries or contact names. Or in our project, we'll be presenting a list of restaurants with thumbnails.

But before UITableView can display the data for you, it requires someone to provide some basic information like:

- how many rows do you want to display in the table view?
- what is the table data? For example, what do you want to display for row 2? What do you want to display in row 5?

That "someone" is known as delegate object. In the above analogy, the personal assistant is the delegate object. In iOS programming, it also applies the concept of delegation usually known as *delegation pattern*. An object relies on another object to perform a specific task. In our project, the ViewController is the delegate that provides the table data. Figure 8-9 illustrates the relationship of UITableView, the protocols and the delegate object.

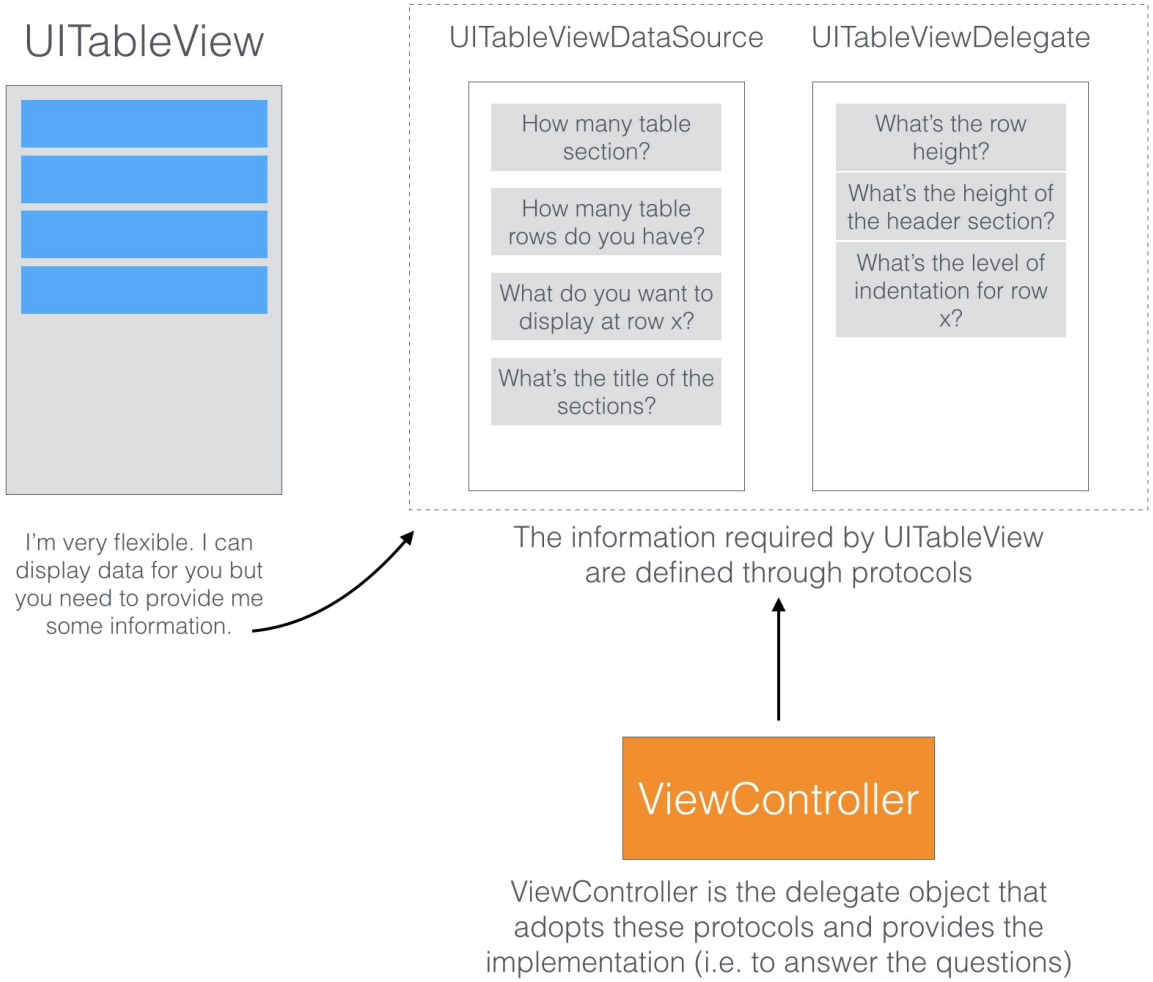

Figure 8-9. The relationship of UITableView, the protocols and the delegate object

So how do you tell UITableView what data to display? The `UITableViewDataSource` protocol is the key. It's the link between your data and the table view. The protocol defines a list of

methods that you have to adopt. Here are the two required methods for the `UITableViewDataSource` protocol:

- tableView(_:numberOfRowsInSection:)
- tableView(_:cellForRowAtIndexPath:)

All you need is to provide an object that implements the above methods, so that the UITableView knows the number of rows to display and the data for each row. The protocol also defines some optional methods, but we'll not discuss it here.

The `UITableViewDelegate` protocol, on the other hand, deals with the appearance of the UITableView. All methods defined in the protocols are optional. They let you manage the height of a table row, configure section headings and footers, re-order table cells, etc. We will not change any of these methods in this example. We will leave that for a later chapter.

With some basic knowledge of protocols, let's continue to code the app. Select `ViewController.swift` and declare a variable for holding the table data. Name the variable `restaurantNames` and insert the following line of code in the class:

```
var restaurantNames = ["Cafe Deadend", "Homei", "Teakha", "Cafe Loisl", "Petite Oyster", "For Kee Restaurant", "Po's Atelier", "Bourke Street Bakery", "Haigh's Chocolate", "Palomino Espresso", "Upstate", "Traif", "Graham Avenue Meats And Deli", "Waffle & Wolf", "Five Leaves", "Cafe Lore", "Confessional", "Barrafina", "Donostia", "Royal Oak", "CASK Pub and Kitchen"]
```

In this example, we use an array to store the table data. The syntax of declaring an array in Swift is similar to that in Objective C. The values are separated by commas and surrounded by a pair of square brackets.

> In Swift, use `let` to make a constant and `var` to make a variable. It's so simple, as compared to Objective-C. All items in an array are of the same type (e.g. String). And thanks to Swift's type inference feature, you do not need to specify the array type. It is automatically detected by Swift. Swift can infer that `String` is the type to use for the restaurantNames variable.

Arrays for Absolute Beginners

An array is a fundamental data structure in computer programming. You can think of an array as a collection of data elements. Consider the `restaurantNames` array in the above code, it represents a collection of `String` elements. You may visualize the array like this:

Figure 8-10. restaurantNames array

Each of the array elements is identified or accessed by an index. An array with 10 elements will have indices from 0 to 9. That means restaurantNames[0] returns the first item of the array.

Let's continue to code. We adopt the two required methods of the `UITableViewDataSource` protocol.

```
func tableView(tableView: UITableView, numberOfRowsInSection section: Int) ->
Int {
    // Return the number of rows in the section.
    return restaurantNames.count
}

func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath:
NSIndexPath) -> UITableViewCell {

    let cellIdentifier = "Cell"
    let cell = tableView.dequeueReusableCellWithIdentifier(cellIdentifier,
forIndexPath: indexPath)

    // Configure the cell...
    cell.textLabel?.text = restaurantNames[indexPath.row]

    return cell
}
```

The first method is used to inform the table view the total number of rows in a section (a table view can have multiple sections but there is only one by default). You can simply call the `count` method to get the number of items in the `restaurantNames` array.

The second method will be called every time a table row is displayed. By using the `indexPath` object, we can get the current row (`indexPath.row`). Here what we did is to retrieve the indexed item from the `restaurantNames` array, and assign it to the text label (`cell.textLabel.text`) for display.

Okay, but what is `dequeueReusableCellWithIdentifier` in the second line of code?

The `dequeueReusableCellWithIdentifier` method is used for retrieving a reusable table cell from the queue with the specified cell identifier. The `Cell` identifier is the one we defined earlier in Interface Builder.

You want your table-based app to be fast and responsive even when handling thousands of rows of data. If you allocate a new cell for each row of data instead of reusing one, your app will use excess memory and may result in a sluggish performance when user scrolls the table view. Remember every cell allocation has a performance cost, especially when the allocation happens over a short period of time.

The screen real estate of iPhone is limited. Even if your app needs to display 1,000 records, the screen may only be able to fit 10 table cells at most. Therefore, why on earth would the app allocate a thousand table view cells instead of creating 10 table cells and reuse them? This would save tons of memory and make table view work more efficiently. For performance reasons, you should reuse table view cells instead of creating new ones.

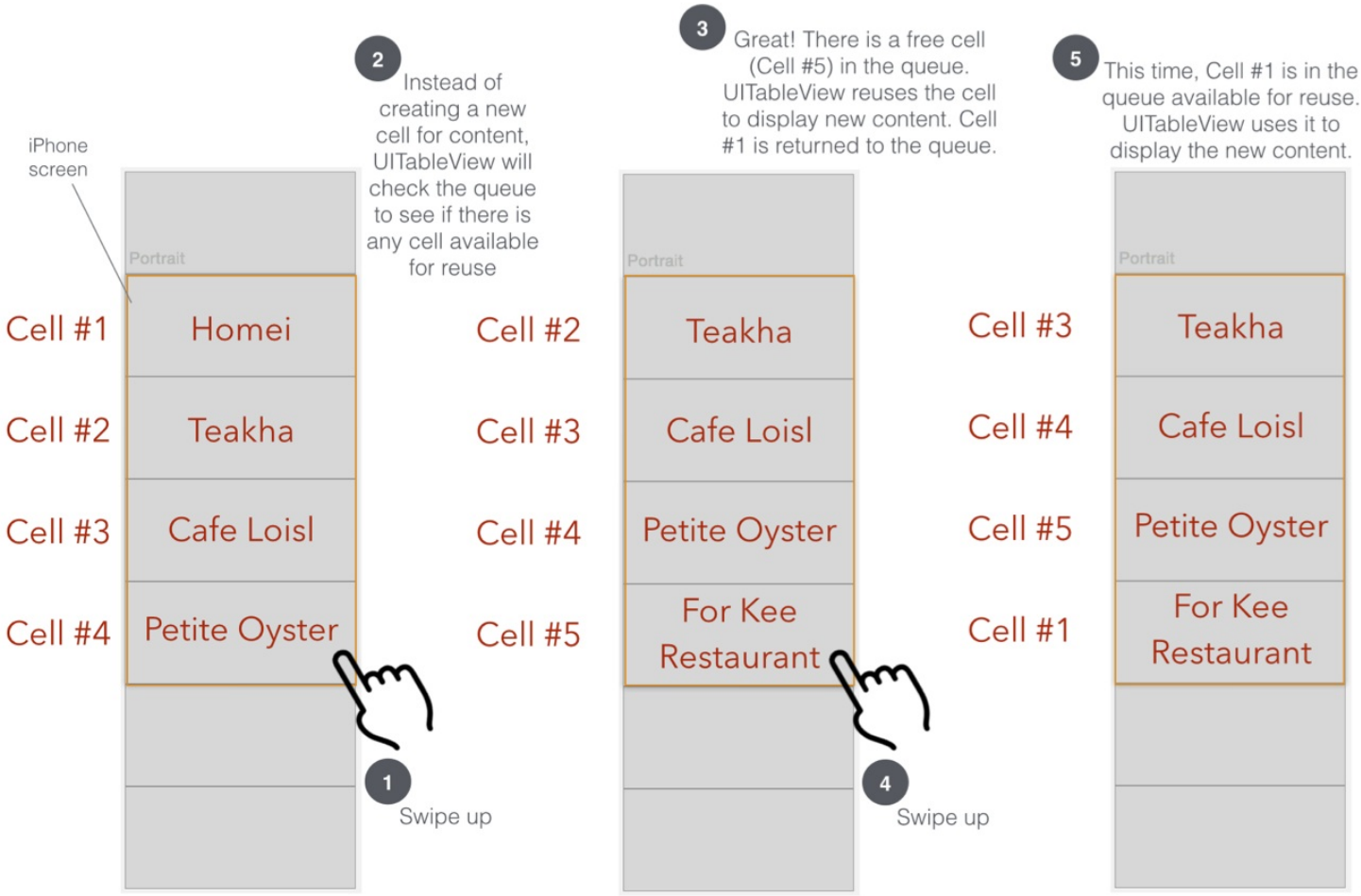

Figure 8-11. Illustration showing how UITableView reuses table view cells

Now, let's hit the "Run" button and test your app. Oops! The app still displays a blank table view just like before.

Why is it the table view still not showing the content as expected? We've written the code for displaying table data and implemented all required methods. But why the table view didn't show the content as expected?

There is still one thing left.

Connecting the DataSource and Delegate

Even though the `ViewController` class has adopted the `UITableViewDelegate` and `UITableViewDataSource` protocols, the `UITableView` object in storyboard has no idea about it.

We have to tell the `UITableView` object that `ViewController` is the delegate object for the data source.

Go back to Main.storyboard. Select the table view. Press and hold the control key, drag from the table view to the View Controller object in the Document Outline.

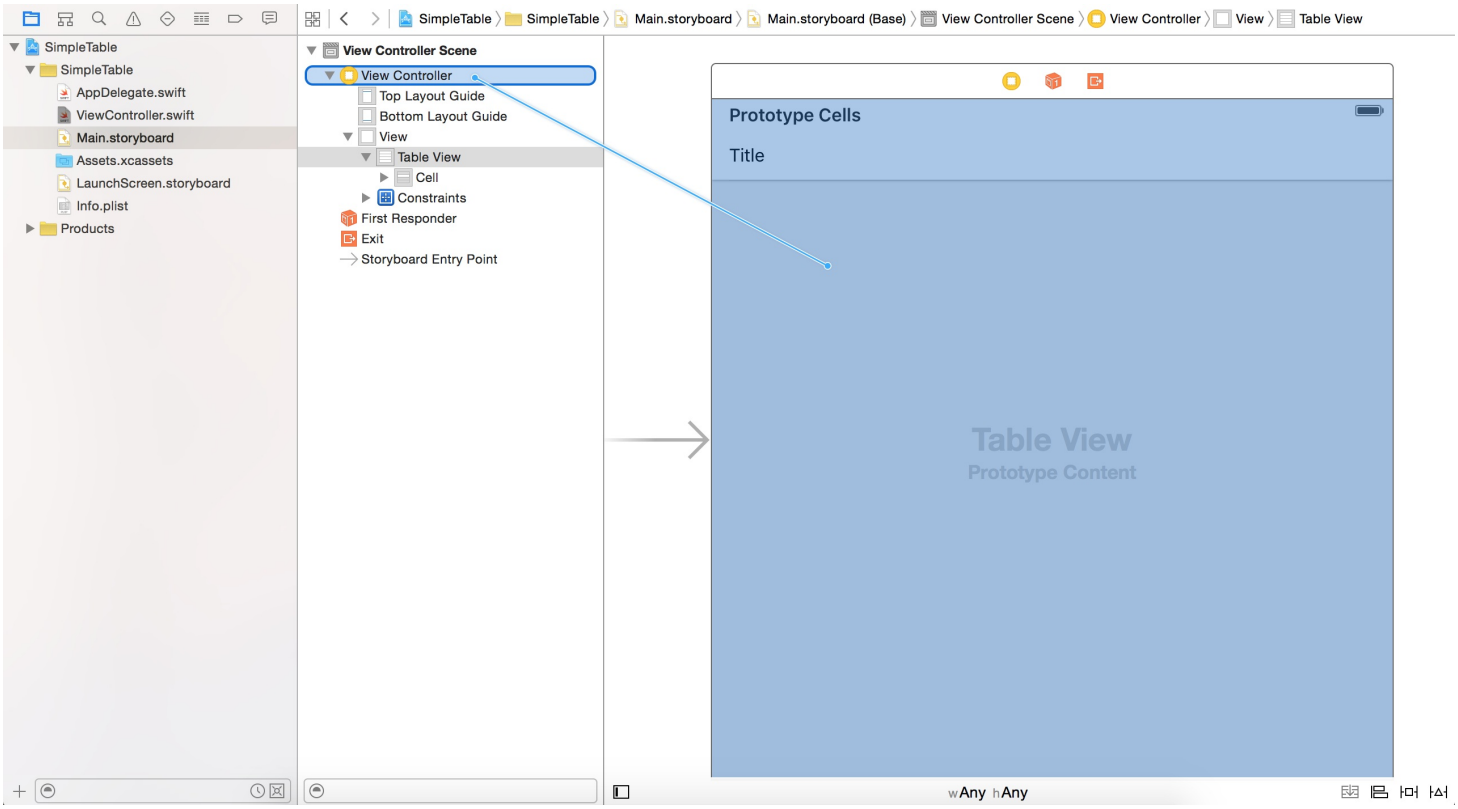

Figure 8-12. Connecting Table View with its Datasource and Delegate

Release both buttons. In the pop-over menu, select *dataSource*. Repeat the above steps and make a connection with *delegate*.

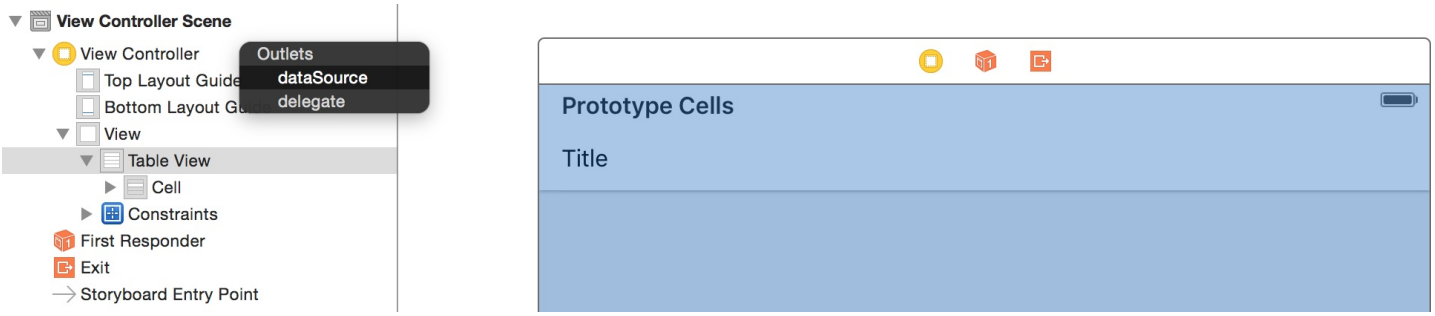

Figure 8-13. Selecting the dataSource and delegate outlets

That's it. To ensure the connections are linked properly, you can select the Table View again. Click the Connections Inspector icon in the Utility area to reveal the existing connections. Alternatively, you can right-click the table to reveal the connections as well.

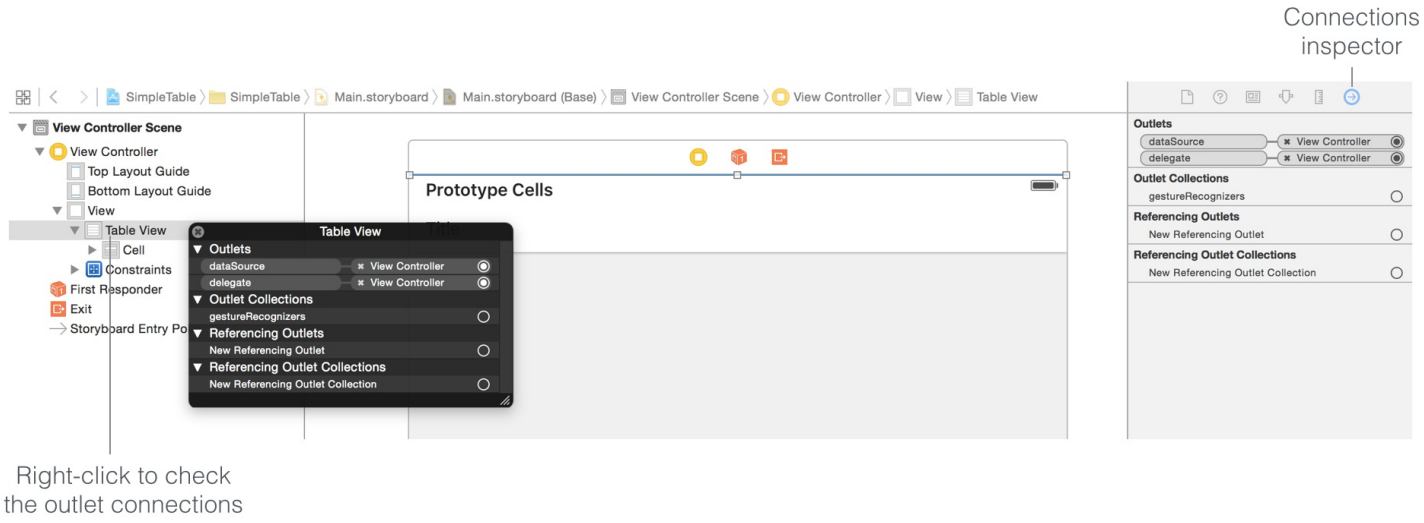

Figure 8-14. Two ways to reveal the connections

Test Your App

Finally, you're ready to test your app. Simply hit the "Run" button and load your app in the simulator. Your app should now show a list of restaurants.

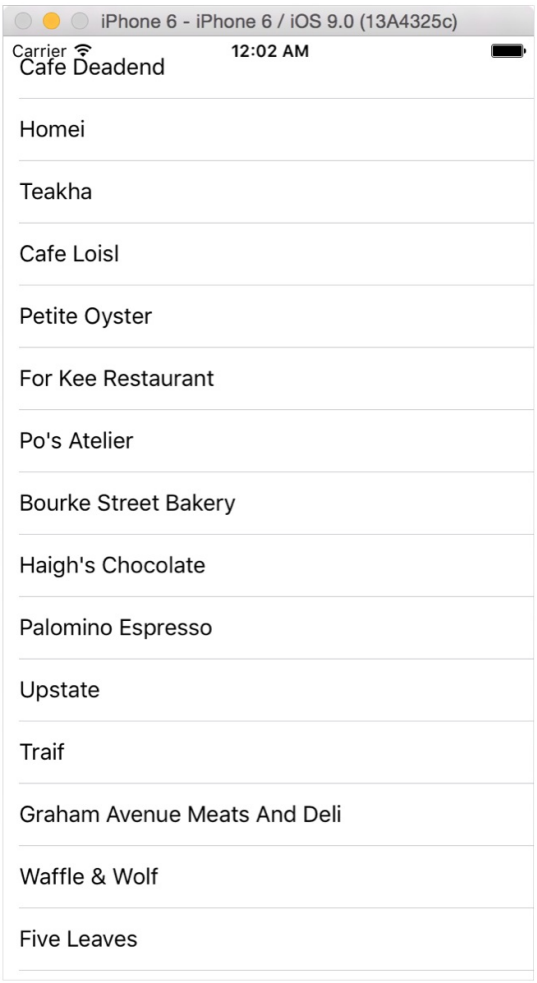

Figure 8-15. SimpleTable App

Add Thumbnail to Table View

Wouldn't it be great to add an image to each row? UITableView makes it extremely easy to do this. You just need to add a line of code to insert a thumbnail in each row.

First, download the sample images from https://www.dropbox.com/s/d1rwisj6pt89db3/restaurantimages.zip. The zipped archive contains three image files. Unzip the file, and drag the images from Finder to the asset catalog (Assets.xcassets).

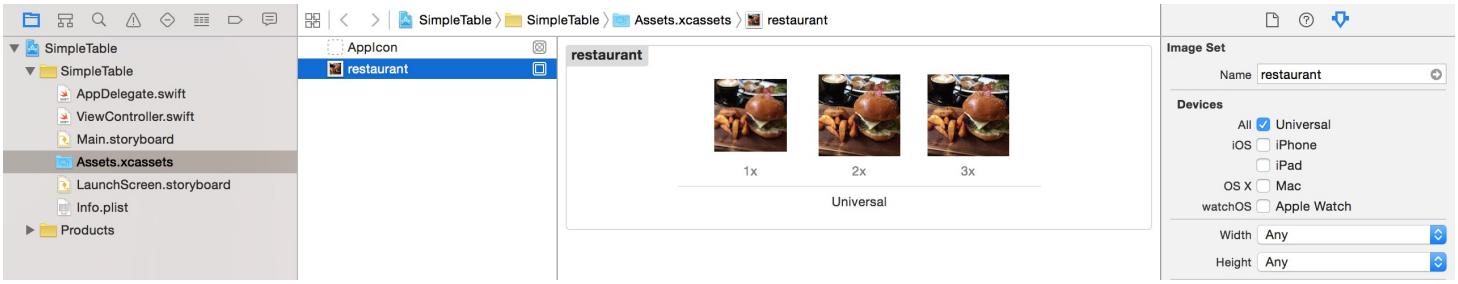

Figure 8-16. Adding images to the asset catalog

Now edit `ViewController.swift` and add the following line of code in the `tableView(_:cellForRowAtIndexPath:)` method. Put it right before `return cell`:

```
cell.imageView?.image = UIImage(named: "restaurant")
```

The UIImage class provided by the UIKit framework lets you create images from files. It supports various image formats such as PNG, GIF and JPEG. Simply pass the name of the image (file extension is optional) and the class will load the image for you.

Recall that we use the `Basic` cell style for the table view cell, which comes with a default area for showing images or thumbnails. This line of code instructs `UITableView` to load the image and display it in the image view of the table cell. Now, hit the "Run" button again and your SimpleTable app should display the image in each row.

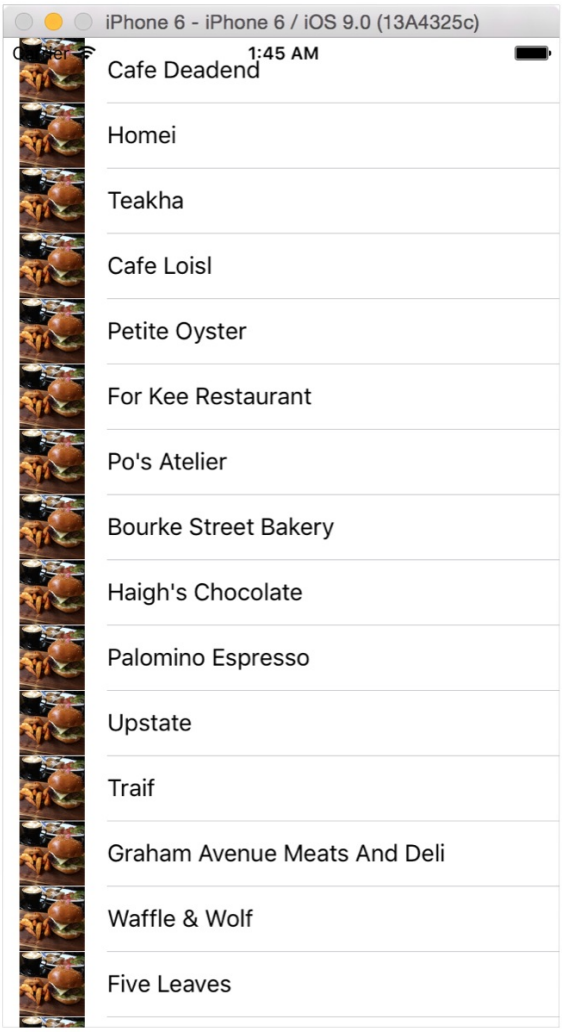

Figure 8-17. SimpleTable app with images

Hide the Status Bar

The content of table view now overlaps with the status bar. This doesn't look good. An easy remedy is to hide the status bar. You can control the appearance of the status bar on a per view controller basis. If you don't want to show the status bar in a particular view controller, simply add the following lines of code:

```
override func prefersStatusBarHidden() -> Bool {
    return true
}
```

Insert the code to ViewController.swift and test the app again. You should have a full-screen

table view without status bar.

Your Exercise

The demo app displays the same image for all table cells. Try to tweak the app such that it shows a different image in each cell. You can download the image pack from https://www.dropbox.com/s/d1rwisj6pt89db3/restaurantimages.zip.

```
Quick note: In case you do not know how to complete the exercise, no worries. I will go through it with you in the next chapter.
```

Summary

Table view is one of the most commonly used elements in iOS programming. If you thoroughly understood the materials and built the app, you should have a good idea of how to create your own table view. I tried to keep everything simple in the demo app. In a real world app, the table data is generally not 'hard-coded'.

Usually, it's loaded from a file, database or somewhere else. We'll talk about that later. Meanwhile, make sure you thoroughly understand how the table view works. Otherwise, go back to the beginning and study the chapter again.

For your reference, you can download the complete Xcode project from https://www.dropbox.com/s/kufb6373g1rrsn8/SimpleTable.zip.