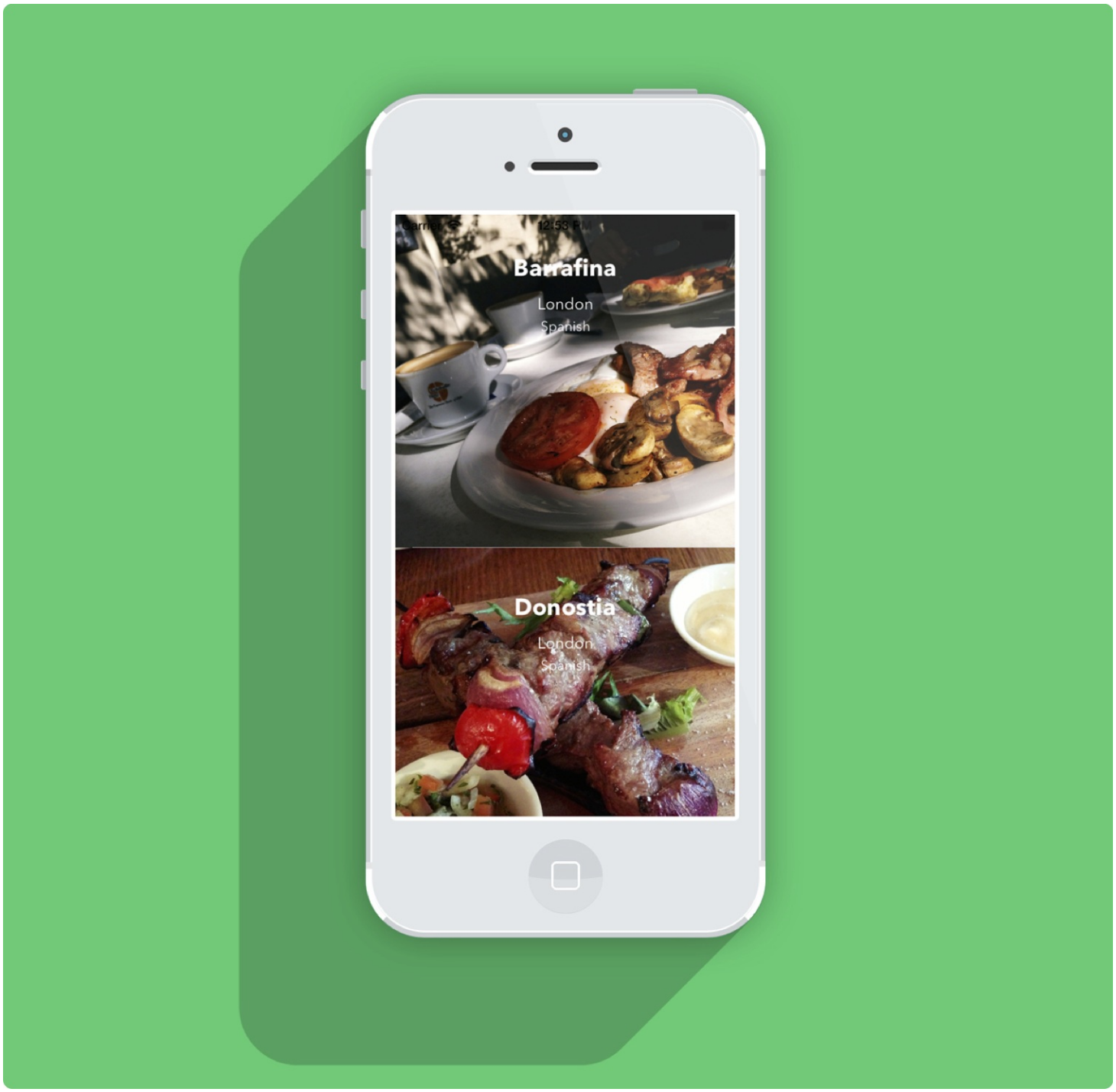
Chapter 9
Customize Table Views Using Prototype Cell



> I think that's the single best piece of advice: Constantly think about how you could be doing things better and questioning yourself.
>
> - Elon Musk, Tesla Motors

In the previous chapter, we created a simple table-based app to display a list of restaurants using the basic cell style. In this chapter, we'll customize the table cell and make it look more stylish. There are a number of changes and enhancements we are going to work on:

- Rebuild the same app using `UITableViewController` instead of `UITableView`
- Display a distinct image for each restaurant rather than showing the same thumbnail
- Design a custom table view cell instead of using the basic style of table view cell

You may wonder why we need to rebuild the same app. There are always more than one way to do things. Previously, we used `UITableView` to create the table view. In this chapter, we'll use `UITableViewController` to create a table view app in Xcode. Will it be easier? Yes, it's going to be easier. Recalled that we needed to explicitly adopt both `UITableViewDataSource` and `UITableViewDelegate` protocols, `UITableViewController` has already adopted these protocols and established the connections for us. On top of this, it has all the required layout constraints right out of the box.

Quick note: Starting from this chapter, you will learn how to develop a real-world app called FoodPin. It's gonna be fun!

Building Table View App Using UITableViewController

First, let's see how to use `UITableViewController` to re-create the same SimpleTable app. Fire up your Xcode, and create a new project using the "Single View application" template. Name the project *FoodPin* and fill in all the required options for the Xcode project, just like what you did in the previous chapter.

Quick note: We're building a real app, so let's give it a better name. Feel free to use other names if you want.

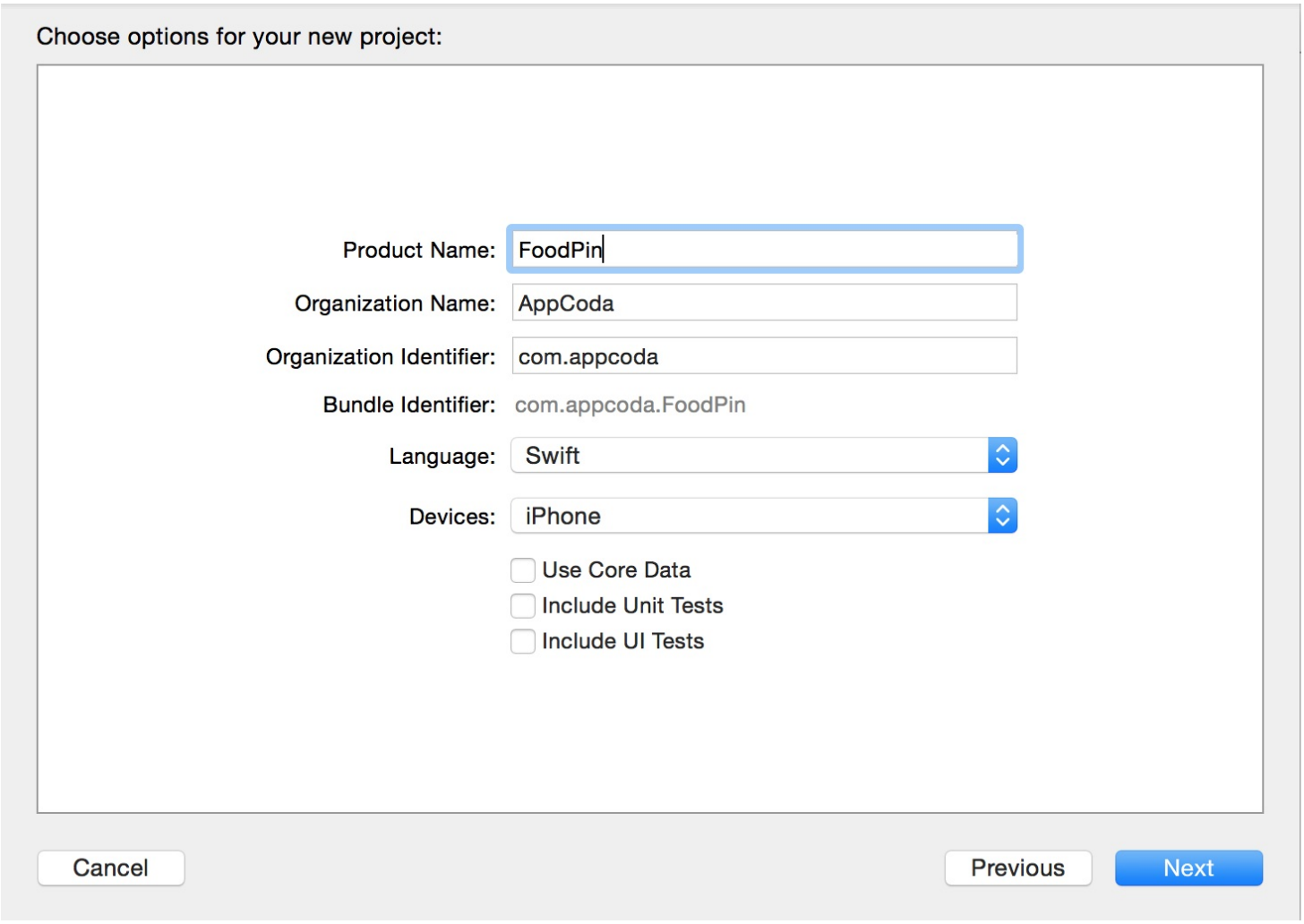


Figure 9-1. Create a FoodPin project

Once you create the Xcode project, select `Main.storyboard` and jump to the Interface Builder editor. As usual, a default view controller is generated by Xcode. This time, we will not use the default controller. Select the view controller, and press the `delete` key to delete it. The view controller is associated with `ViewController.swift`. We do not need it either. In the Project Navigator, select the file and hit the `delete` button. Select "Move to Trash" when prompted. This will completely delete the file.

Go back to Interface Builder. Drag a Table View Controller (i.e. UITableViewController) from the Object library, and place it in the storyboard. You have to designate that controller as the initial view controller. This tells iOS that the table view controller is the first view controller to be loaded. All you need to do is select the Attributes inspector, and check the `Is Initial View Controller` option. You'll then see an arrow pointing to the table view controller (see figure 9-

2).

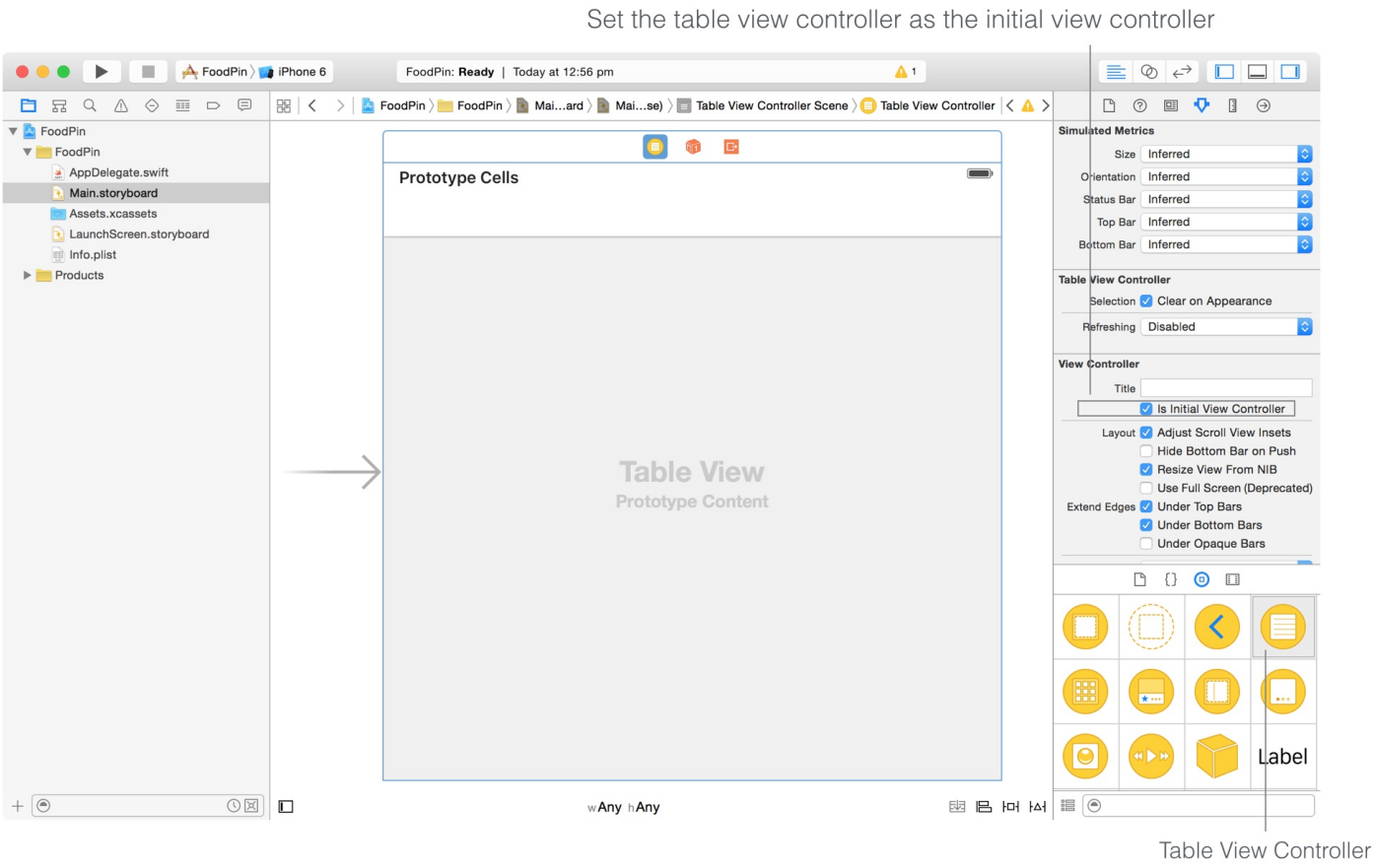


Figure 9-2. Drag a Table View Controller from the Object library and set it as the initial view controller

We haven't inserted any data into the table yet. So if you compile and run the app now, you'll end up with a blank table.

By default, the table view controller is associated with the `UITableViewController` class. In order to populate our own data, we have to associate it with our own class.

Go back to the Project Navigator and right-click the FoodPin folder. Select "New Files..." to create a new file.

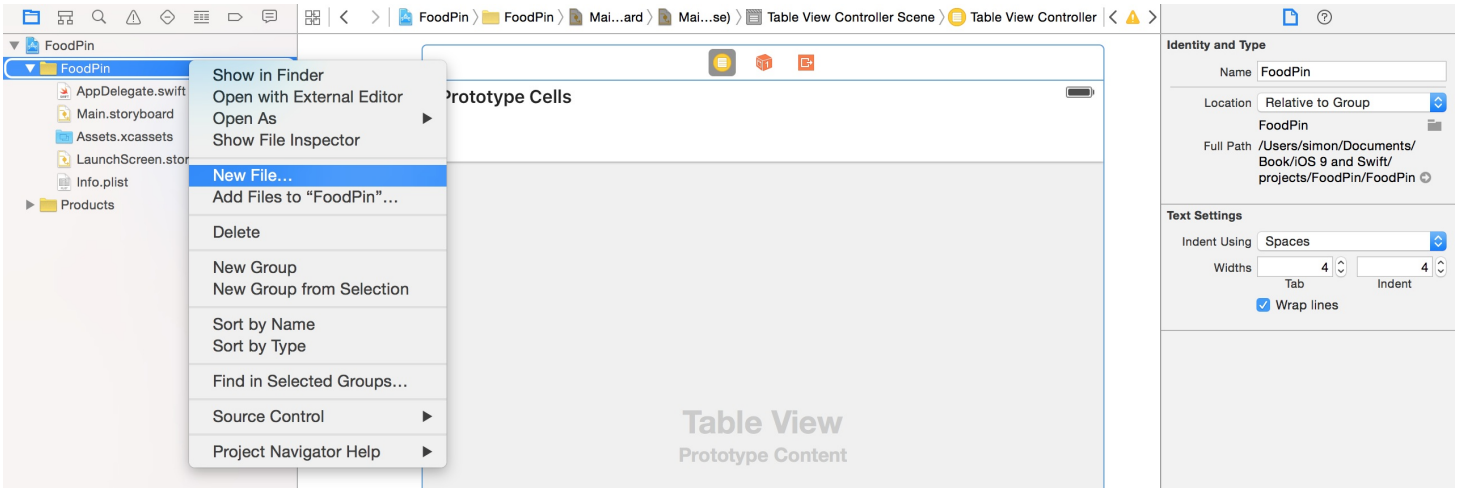


Figure 9-3. Create a new file

Choose Source (under iOS category) > Cocoa Touch Class as the template, and then click `Next`. Name the new class `RestaurantTableViewController`. Because we're working with a table view controller, change the value of "Subclass of" to `UITableViewController`. Keep the rest of the values intact, click "Next" and save it to the project folder. You should see the `RestaurantTableViewController.swift` file in the project navigator.

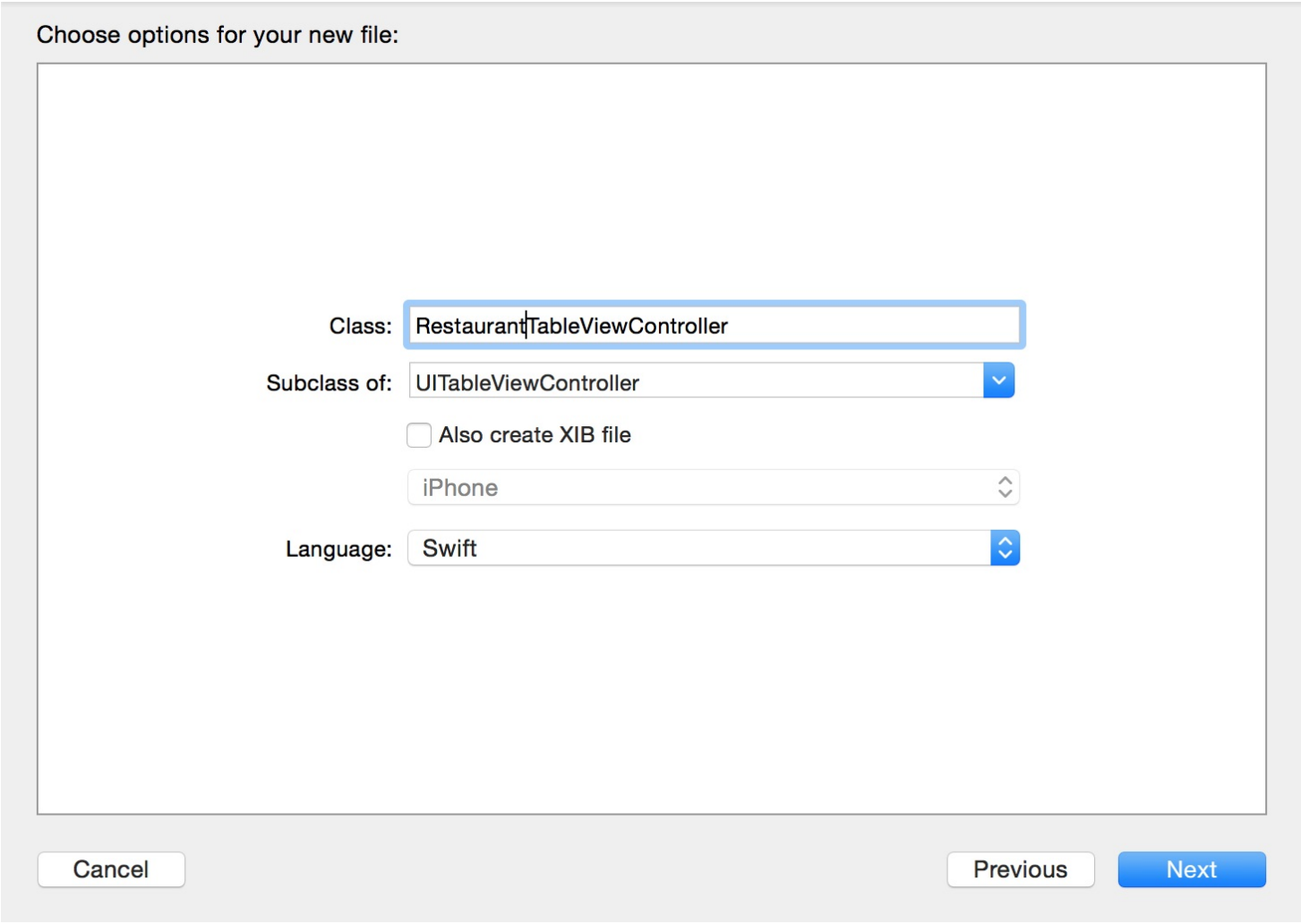


Figure 9-4. Create a subclass of UITableViewController

Superclass and Subclass

If you're new to programming, you may wonder what a subclass is. Swift is an object oriented programming (OOP) language. In OOP, a class can be inherited by another class. In the example, the RestaurantTableViewController class inherits from the UITableViewController class. It inherits all the states and functionalities provided by the UITableViewController class. The RestaurantTableViewController class is known as a subclass (or child class) of UITableViewController. In other words, the UITableViewController class is referred as the superclass (or parent class) of RestaurantTableViewController.

The table view controller in the storyboard has no idea about the `RestaurantTableViewController` class. So we have to assign the table view controller with the

new custom class. Go to the Interface Builder editor and select the table view controller. In the Identity inspector, set the custom class to `RestaurantTableViewController`. Now we have established a relationship between the table view controller in the storyboard and the new class.

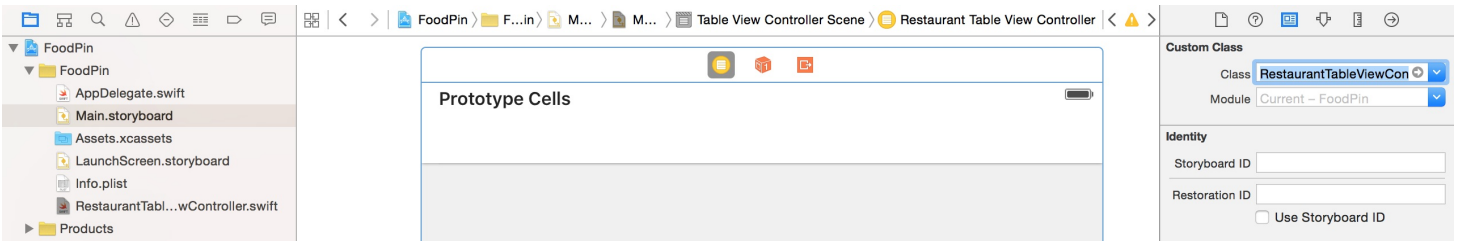


Figure 9-5. Set the custom class of the Table View Controller

There is one more thing to configure for the table view. Select the prototype cell. In the Attributes inspector, change the style to `Basic` and set the identifier to `Cell`. This is pretty much the same as what we have done in the previous chapter.

Okay, the user interface is ready. Let's move onto the code. Select the `RestaurantTableViewController.swift` file in the Project Navigator and declare an instance variable, which is used for holding the table data.

```
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", "Waffle & Wolf", "Five Leaves", "Cafe Lore", "Confessional", "Barrafina", "Donostia", "Royal Oak", "Thai Cafe"]
```

As said before, the `UITableViewController` class has already adopted the `UITableViewDataSource` and `UITableViewDelegate` protocols. Since the `RestaurantTableViewController` class is a subclass of `UITableViewController`, it has adopted these protocols too.

If you're not forgotful, we need to implement the following required methods of the `UITableViewDataSource` protocol in order to provide the table data:

- tableView(_:numberOfRowsInSection:)

- tableView(_:cellForRowAtIndexPath:)

The `UITableViewController` class provides a default implementation for these two methods. Usually, the default methods don't fit well in our own apps. We need to override the default ones and provide our own implementation. Insert the following code in `RestaurantTableViewController.swift`:

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

    return cell
}
```

In Swift, to override a method of a superclass, we simply add the `override` keyword at the very beginning of the method. The above code is exactly the same as the one we covered in the previous chapter, so I'll not go into the details.

Next, change the following code snippets in `RestaurantTableViewController`. The code is generated by Xcode when creating the class file. By default, it returns zero for both the number of section and the number of rows in section. In other words, it's telling the table view that there is no data in the table view. This is what we want. So we have to modify the code like this:

```
override func numberOfSectionsInTableView(tableView: UITableView) -> Int {
    return 1
}

override func tableView(tableView: UITableView, numberOfRowsInSection section:
Int) -> Int {
    return restaurantNames.count
}
```

Here we tell the table view that there is only one section and return the total number of restaurants as stored in the array. As a side note, the `numberOfSectionsInTableView` method is optional. If you remove it, the table view still works because the number of sections is set to 1

by default.

Lastly, download the image pack from https://www.dropbox.com/s/d1rwisj6pt89db3/restaurantimages.zip and drag all images to the Assets.xcassets folder. Now, hit the "Run" button and your FoodPin app should look the same as the one we built earlier.

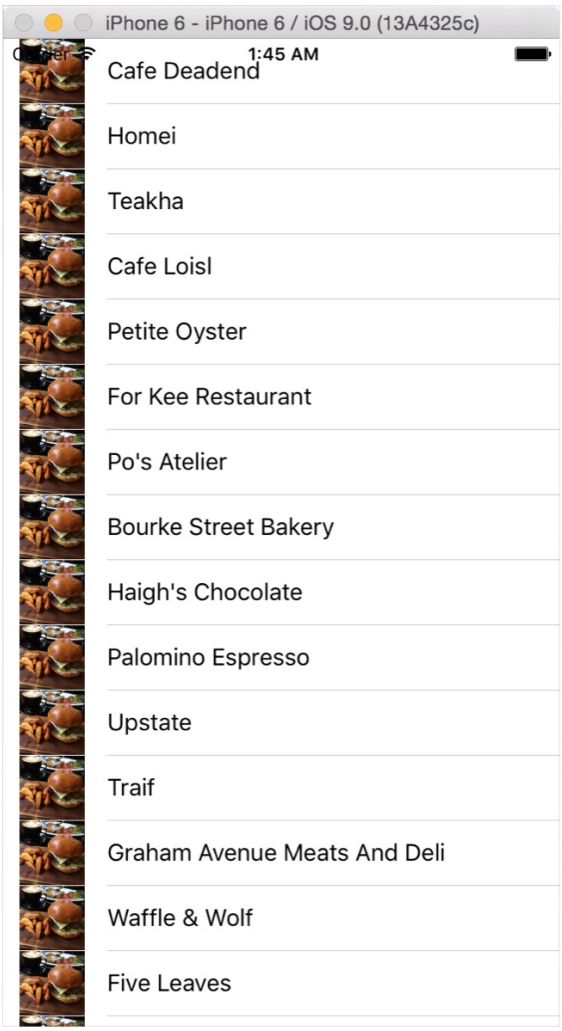


Figure 9-6. SimpleTable App

By now you should understand how to present data in a table view. I've showed you two approaches:

1. By using UITableView and View Controller

2. By using UITableViewController.

You may wonder which approach you should use. In general, approach #2 is good enough. `UITableViewController` has configured everything for you. You can simply override some methods to provide the table data. But what you lose is flexibility. The table view, embedded in `UITableViewController`, is fixed. You can't change it. If you want to layout a more complicated UI using table views, approach #1 will be more appropriate.

Display Different Thumbnails

Did you go through the exercise in the previous chapter? I hope you've put your effort in it. There are many ways to display different thumbnails in each table row. I'll show you the most straightforward way to do. First, download another image pack from https://www.dropbox.com/s/zzlsbz2c4p3pow1/restaurantimages.zip, and unzip it. Add all the images into the asset catalog (i.e. Assets.xcasset). The pack includes some food and restaurant images. If you like, you're free to use your own images.

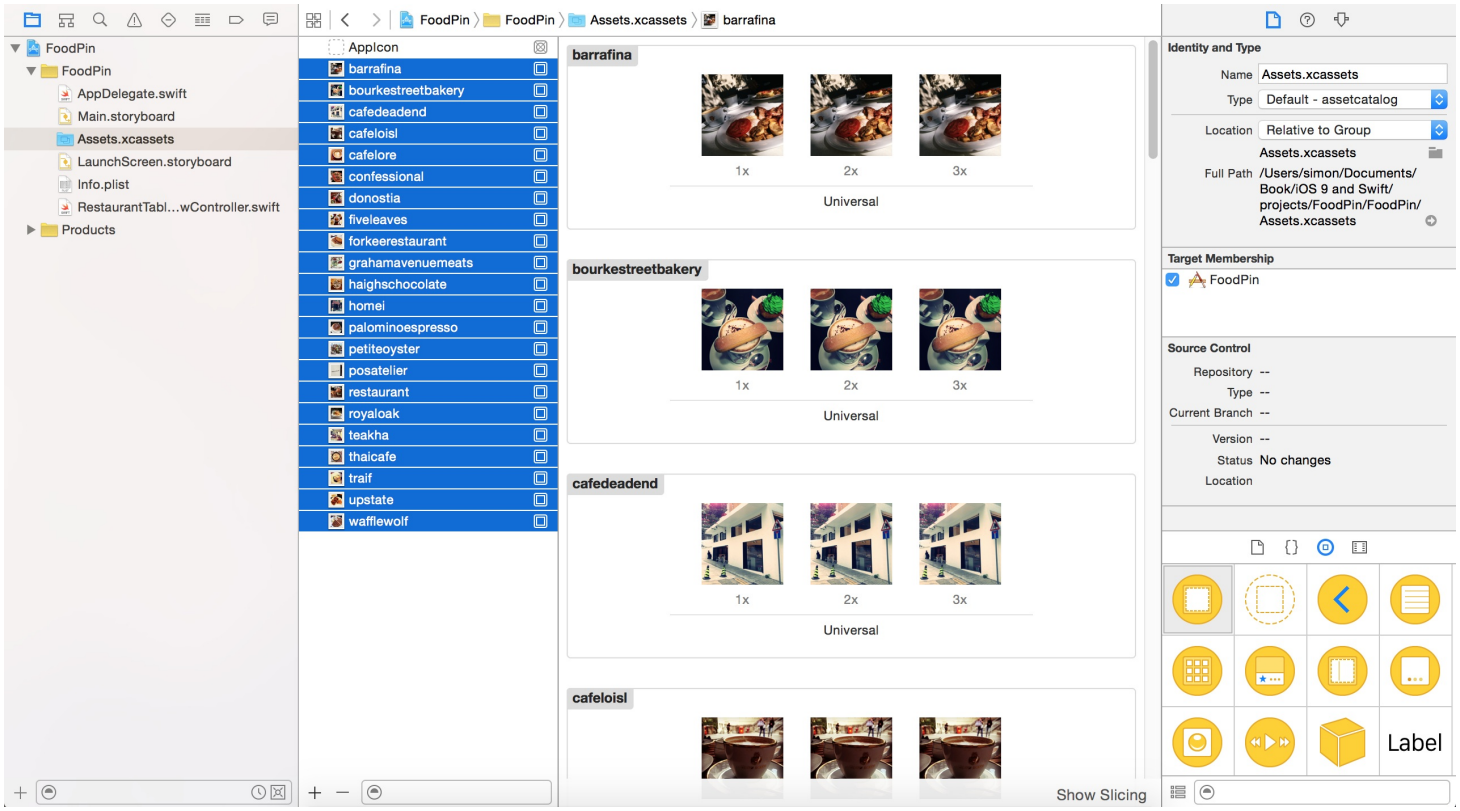


Figure 9-7. Adding restaurant images to the asset catalog

Before we move on to changing the code, let's revisit the code for displaying the thumbnails in table row.

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

The above line code in the `tableView(_:cellForRowAtIndexPath:)` method instructs `UITableView` to display restaurant.jpg in each cell. In order to display a different image, we need to alter this line of code.

As explained before, this method is called by iOS every time a particular table row is about to display. The current row number is embedded in the `indexPath` parameter. You can simply use `indexPath.row` to find out which row is now being processed.

Therefore, to display a different image in each row, all we need to do is add a new array to store the file name of thumbnails. Let's name the array `` `restaurantImages `` . Insert the following line of code in the `RestaurantTableViewController` class:

```
 var restaurantImages = ["cafedeadend.jpg", "homei.jpg", "teakha.jpg",
"cafeloisl.jpg", "petiteoyster.jpg", "forkeerestaurant.jpg", "posatelier.jpg",
"bourkestreetbakery.jpg", "haighschocolate.jpg", "palominoespresso.jpg",
"upstate.jpg", "traif.jpg", "grahamavenuemeats.jpg", "wafflewolf.jpg",
"fiveleaves.jpg", "cafelore.jpg", "confessional.jpg", "barrafina.jpg",
"donostia.jpg", "royaloak.jpg", "thaicafe.jpg"]
```

In the above code, we initialize the `restaurantImages` array with a list of image file names. The order of images are aligned with that of the `restaurantNames` .

In order to load the corresponding image of the restaurant, change the line of code in the `tableView(_:cellForRowAtIndexPath:)` method to:

```
cell.imageView?.image = UIImage(named: restaurantImages[indexPath.row])
```

After saving all the changes, try to run your app again. Each restaurant has its own image.

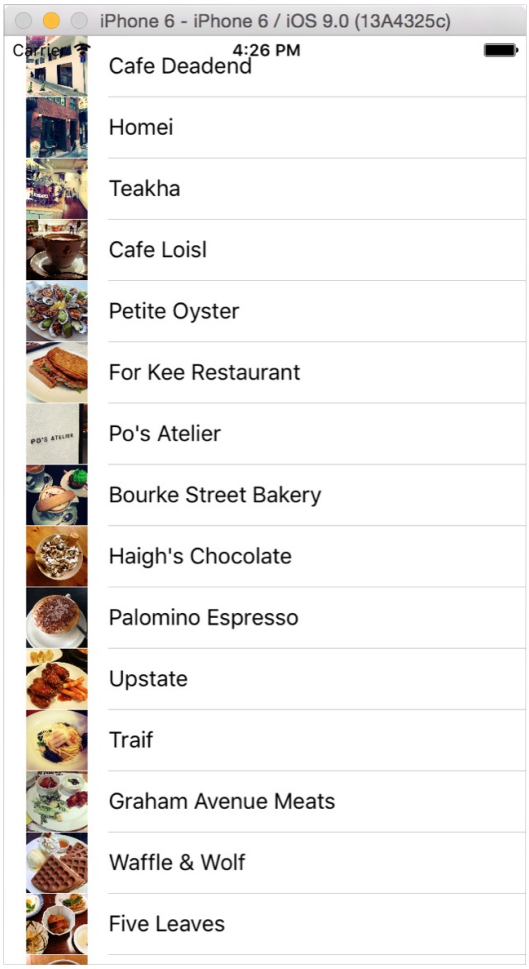


Figure 9-8. SimpleTable App with different images

Customize Table View Cells

Does the app look better? We're going to make it even better by customizing the table cells. So far we utilize the default style of the table view cell. The position and size of the thumbnail are fixed. What if you want to:

- Change the height of the cell
- Make the thumbnail a little bit bigger
- Show more information about the restaurant such as location and type
- Change the font type and size
- Display circular images instead of square images

To give you a better idea about how the cell is customized, take a look at figure 9-9. The cell looks awesome, right?

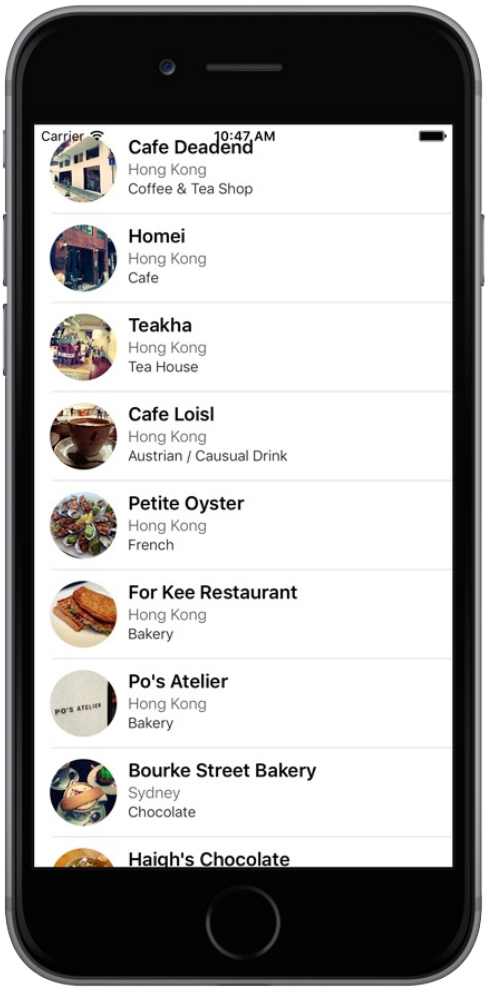


Figure 9-9. Redesigned table view cells

Designing Prototype Cells in Interface Builder

The beauty of prototype cells is that it allows developers to customize the cell right inside the table view controller. To build a custom cell, you simply add other UI controls (e.g. labels, image views) to the prototype cell.

Let's first change the style of the cell. You can't customize the cell when it's set to the `Basic` style. Select the prototype cell and change the style from `Basic` to `Custom` in the Attributes

inspector.

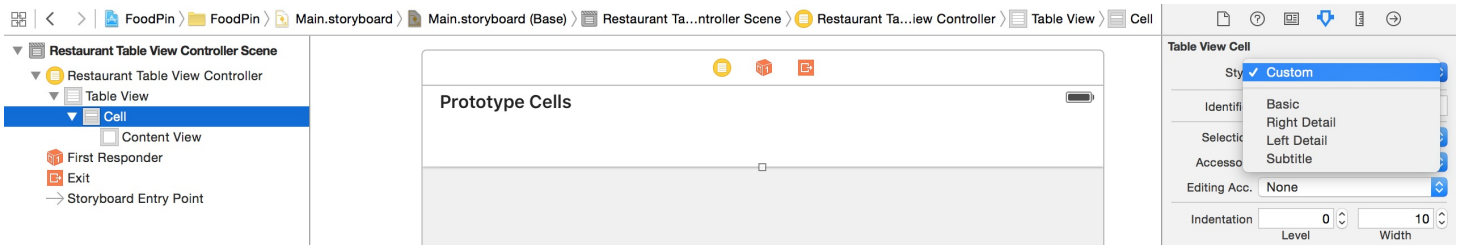


Figure 9-10. Changing the cell style from Basic to Custom

In order to accommodate a larger thumbnail, we have to make the cell a little bit bigger. You'll need to change both the row height of table view and prototype cell. Select the table view and change the row height to `80`.

Figure 9-11. Changing the row height of the table view

Then select the prototype cell and go to the Size inspector. Check the `custom` checkbox and change the row height to `80`.

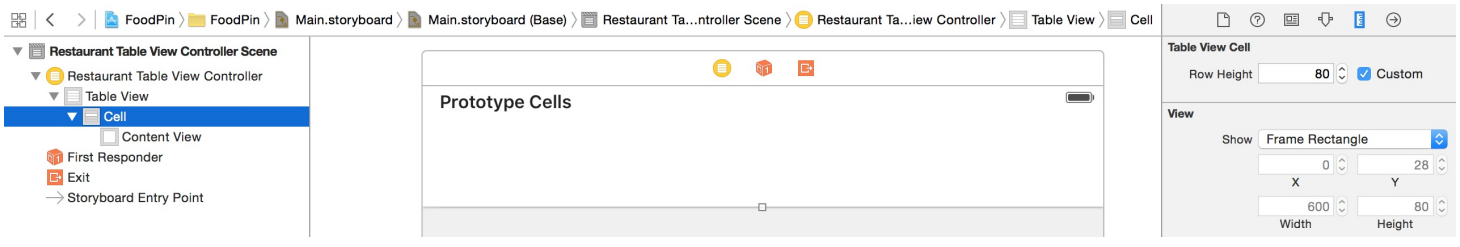


Figure 9-12. Changing row height of the prototype cell

After altering the row height, drag an Image View object from the Object library to the prototype cell. You're free to resize the image to fit your need. Figure 6-12 shows my recommended size. You can select the image view, click "Size" inspector and change the "X", "Y", "Width" and "Height" attributes.

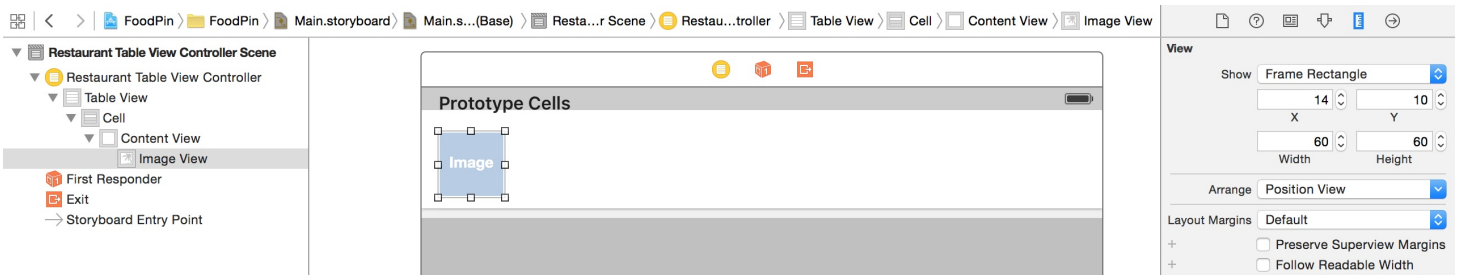


Figure 9-13. Adding an image view to the prototype cell

Next, we will add three labels to the prototype cell:

- Name - for restaurant name
- Location - for restaurant location (e.g. New York)
- Type - for restaurant type (e.g. tea house)

To add a label, drag a Label object from the Object library to the cell. Name the first label "Name". Instead of using the system font for the label, we'll use a text style. I will explain to you the difference of a fixed font and a text style in later chapters. For now, just go to the Attributes inspector, and change the font to `Text Styles - Headline`.

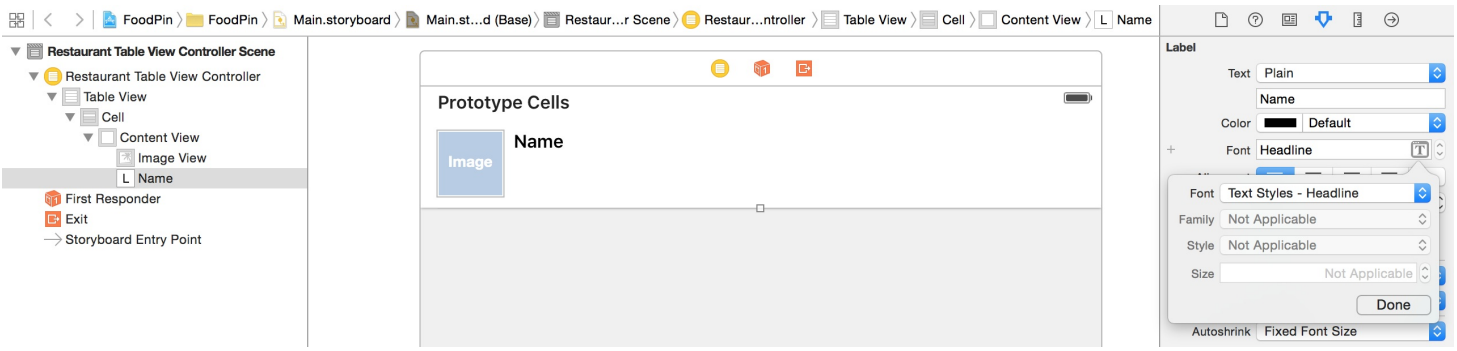


Figure 9-14. Adding name label to the prototype cell

Drag another label to the cell and name it *Location*. Change the font style to `Light` and set the font size to `14` points. Also set to the font color to `Dark Gray`. Lastly, create another label and name it *Type*. Similarly, change the font style to `Light` and set the font size to `13` points.

I've covered stack views in chapter 6. Not only can you use stack views in a view controller, you can also apply stack views to layout the components in a prototype cell. Therefore, instead of defining the layout constraints for the labels and image view, we will use stack views to manage them. First, hold the `command` key, and select the three labels. Click the `Stack` button in the layout bar to embed them in a vertical stack view. Go to the Attributes inspector, and change the spacing of the stack view from `0` to `1`. This will add a space between the labels.

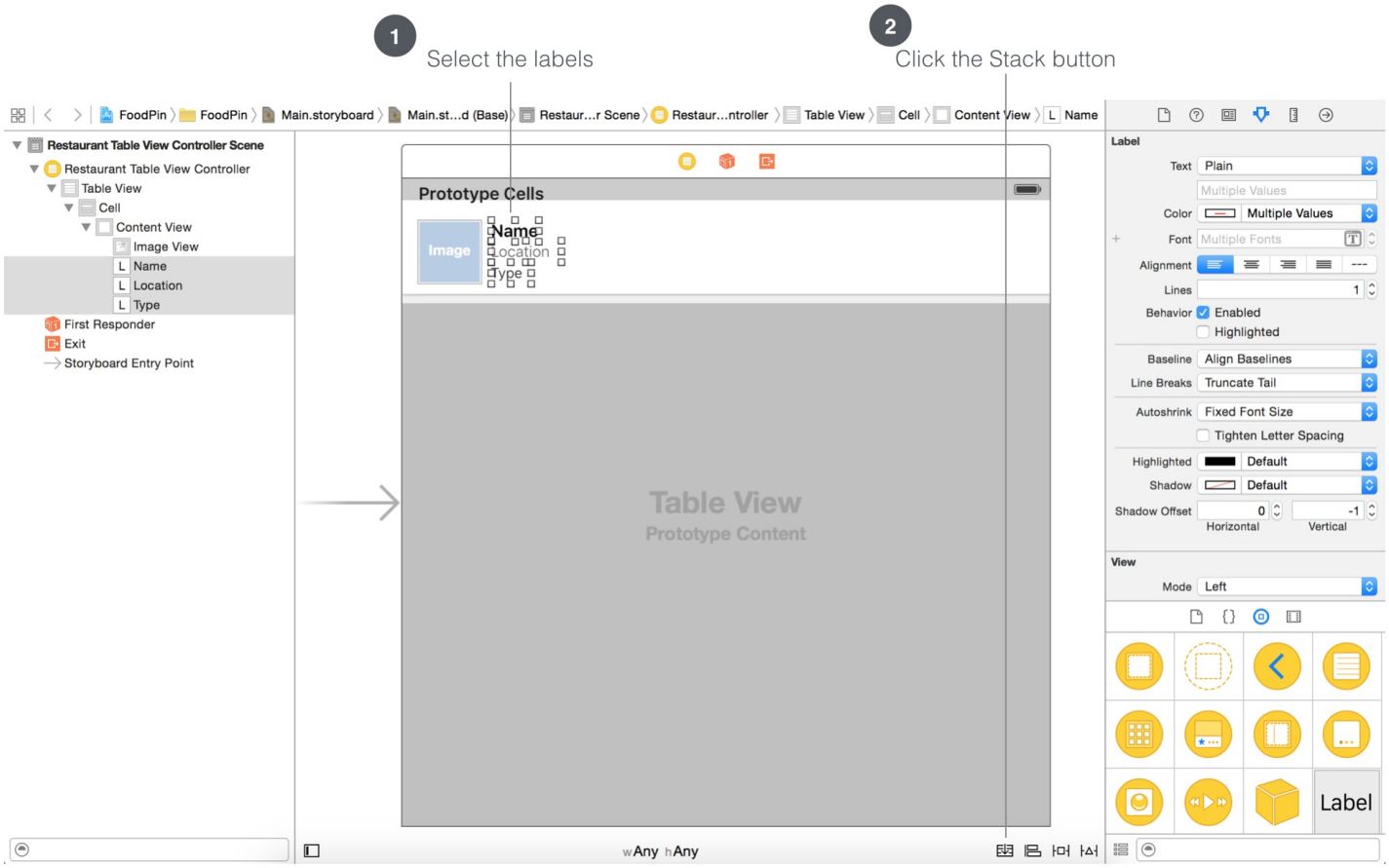


Figure 9-15. Adding the labels using stack view

Now select both the image view and the stack view that we have just created. Click the `Stack` button, Interface Builder will embed them into a horizontal stack view. By default, there is no

space between the image view and the "Label" stack view. You can just go up to the Attribute inspector, and change the spacing option from `0` to `10`. Awesome, right? You can nest stack views to create some complex layouts.

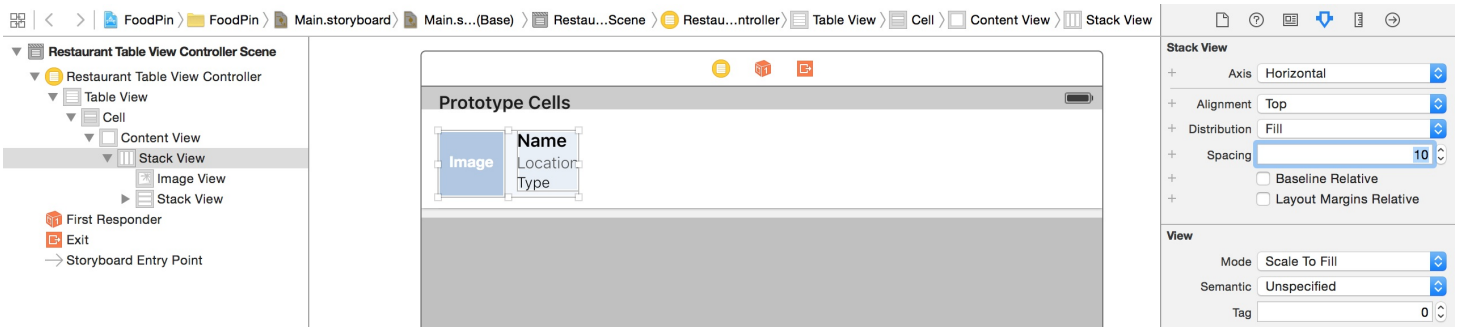


Figure 9-16. Combining the label stack view and the image view into a horizontal stack view

Again, it doesn't mean you do not need to use auto layout. We still need to define the layout constraints for the stack view. Figure 9-17 depicts the layout requirements of the cell.

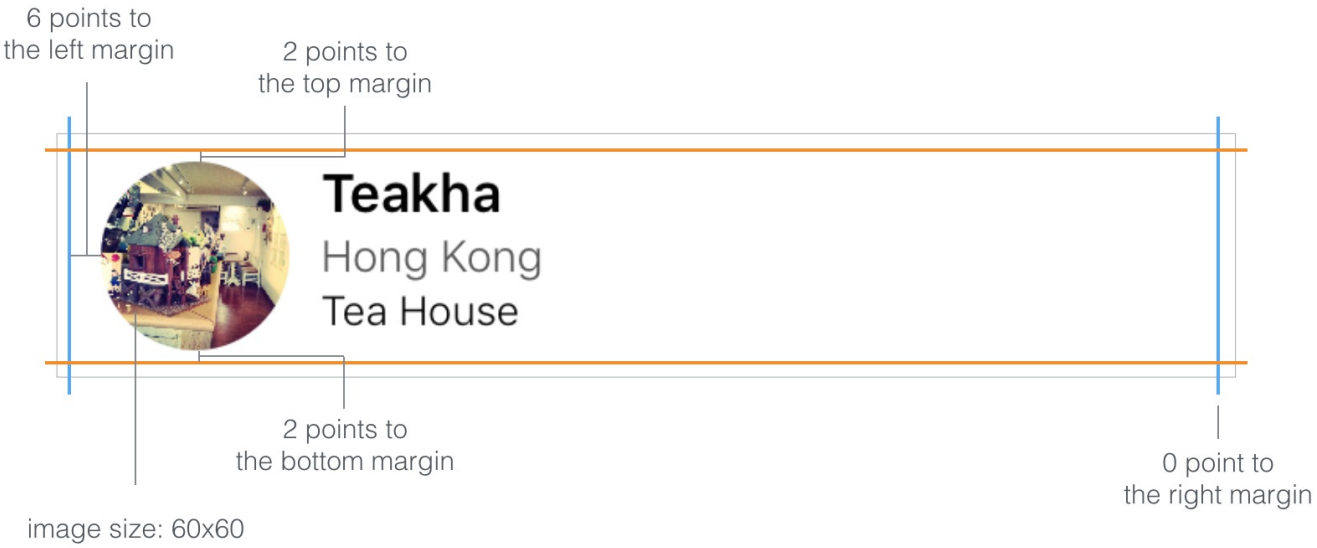


Figure 9-17. The layout requirement of the prototype cell

In brief, we want the cell content (i.e. the stack view) is confined to the viewable area of the cell. This is why we're going to define a spacing constraint for each side of the stack view. On top of this, the size of image view should be fixed to 60x60 points.

Now select the stack view, and click the Pin button in the layout bar. Set the values of the top, left, bottom and right sides to 2, 6, 1.5 and 0 respectively.

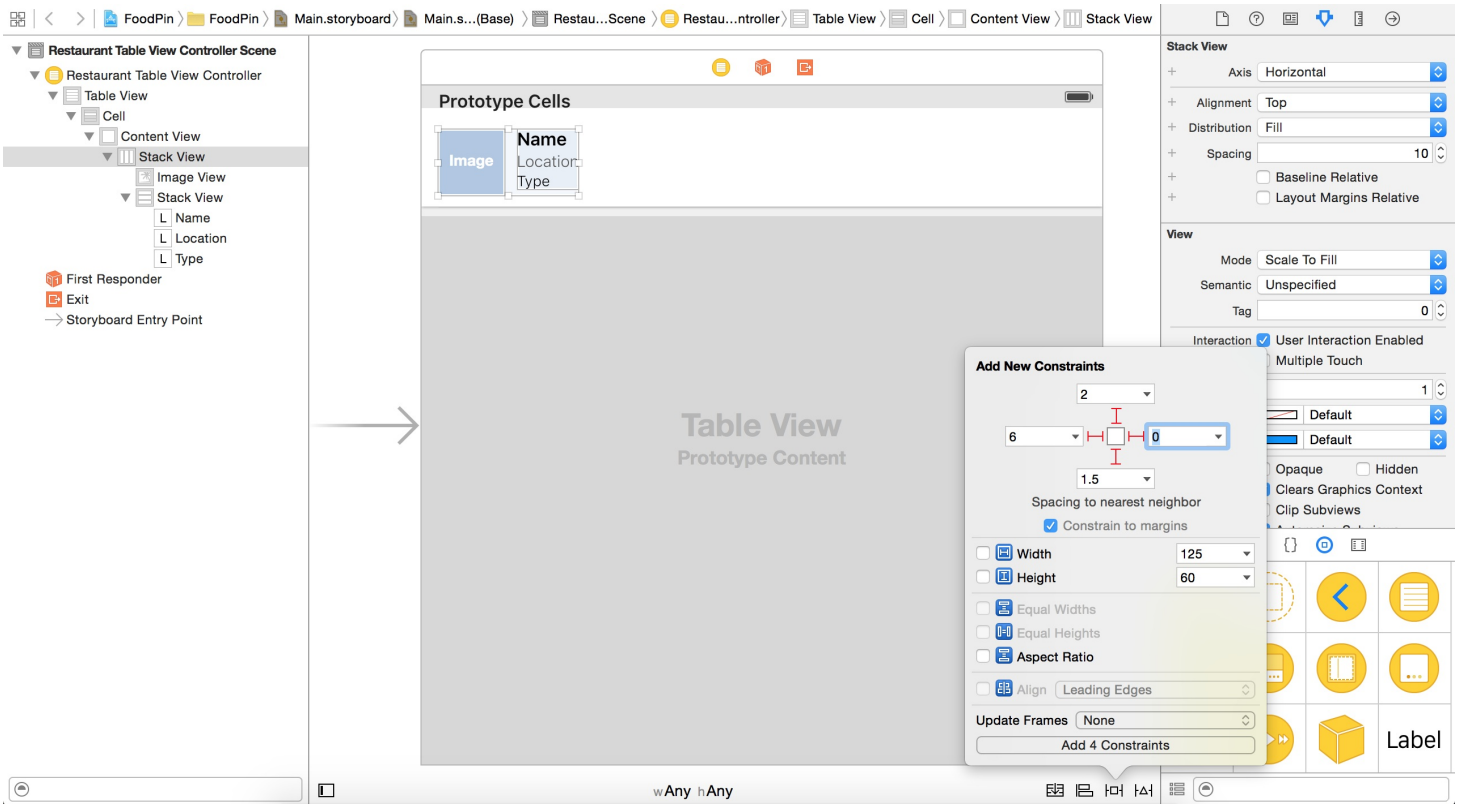


Figure 9-18. Adding spacing constraints for the stack view

Once you add the 4 constraints, the stack view should be resized automatically. Next, in the document outline, drag horizontally from the image view to itself. In the pop over menu, hold shift key and choose both "width" and "height" options. This ensures the size of the image view is fixed.

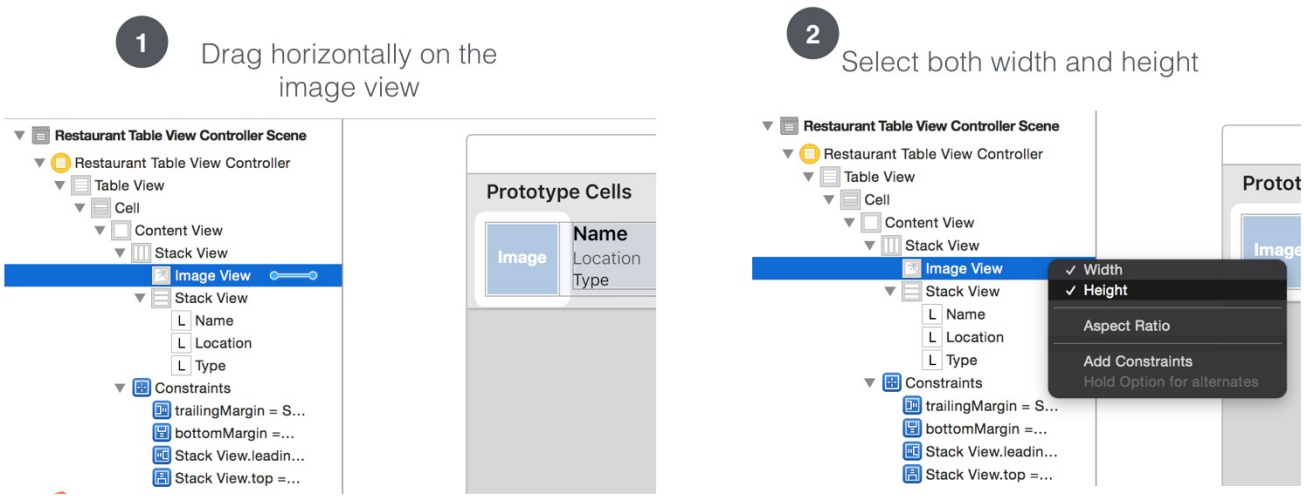


Figure 9-19. Adding the width and height constraints for the image view

Cool! You've completed the layout of the prototype cell. Let's move on and write some code.

Creating a Class for the Custom Cell

So far, we've designed the table cell. But how can we change the label values of the prototype cell? These values are supposed to be dynamic. By default, the prototype cell is associated with the `UITableViewCell` class. In order to update the cell data, we're going to create a new class, which extends from `UITableViewCell`, for the prototype cell. This class represents the underlying data model of the custom cell. As usual, right click the "FoodPin" folder in Project Navigator and select "New File...".

After selecting the option, Xcode prompts you to select a template. As we're going to create a new class for the custom table view cell, select "Cocoa Touch Class" and click "Next". Fill in `RestaurantTableViewCell` as the class name and set the value of "Subclass of" to `UITableViewCell`.

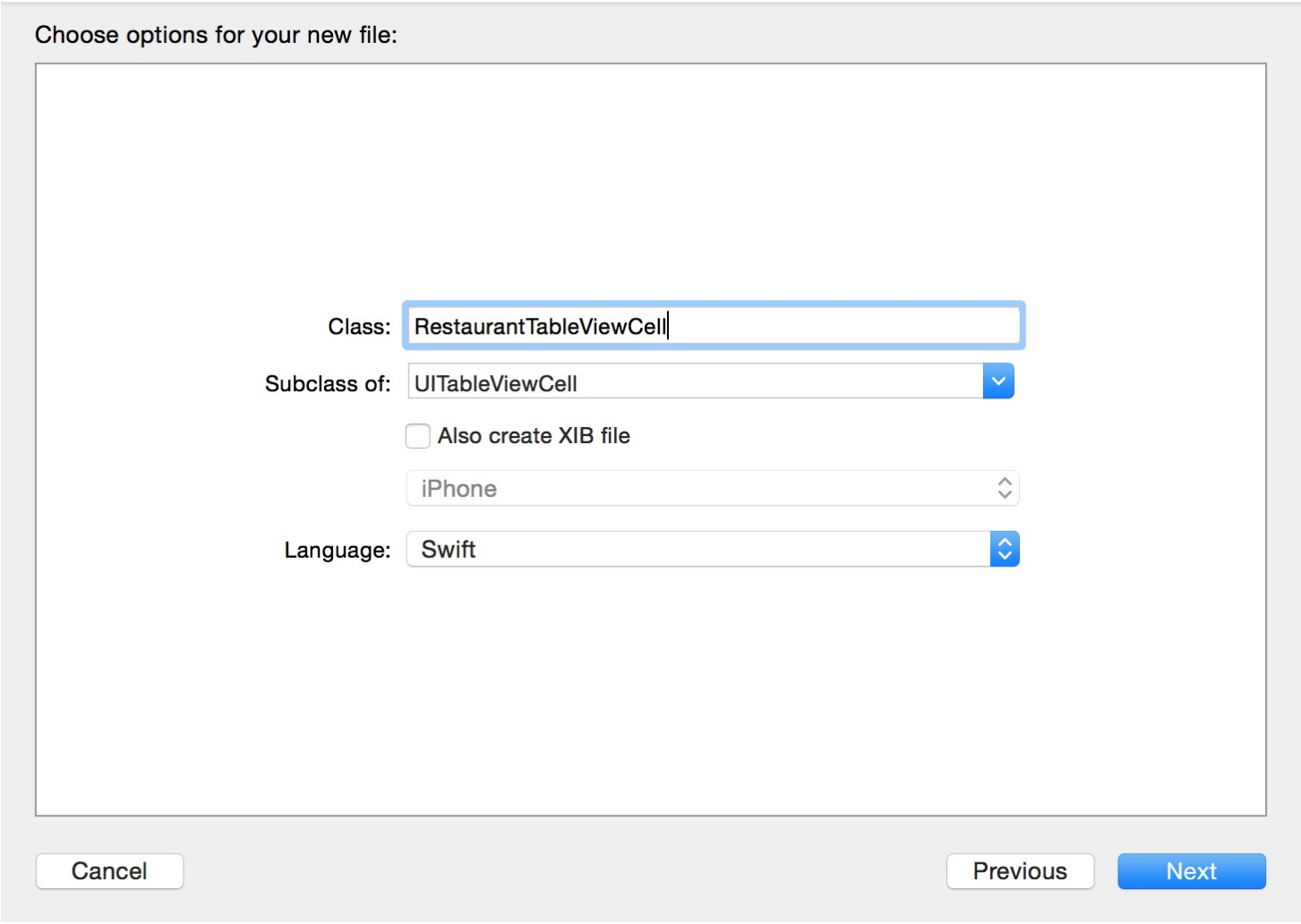


Figure 9-20. Creating a new class for the custom table view cell

Click "Next" and save the file in the FoodPin project folder. Xcode should create a file named `RestaurantTableViewCell.swift` in the Project Navigator.

Next, declare the following outlet variables in the `RestaurantTableViewCell` class:

```
@IBOutlet var nameLabel: UILabel!
@IBOutlet var locationLabel: UILabel!
@IBOutlet var typeLabel: UILabel!
@IBOutlet var thumbnailImageView: UIImageView!
```

The `RestaurantTableViewCell` class serves as the data model of the custom cell. In the cell, we have 4 properties that are changeable:

- thumbnail image view
- name label
- location label
- type label

The data model stores and provides the values for the cell to display. Each of them is required to connect with the corresponding user interface object in Interface Builder. By connecting the source code with the UI objects, we can change the values of UI objects dynamically.

This is a very important concept in iOS programming. Your UI in storyboard and code are separated. You create the UI in Interface Builder, and you write your code in Swift. If you want to change the value or properties of a UI element (e.g. label), you have to establish a connection between them so that an object in your code can obtain a reference to an object defined in a storyboard. In Swift, you use `@IBOutlet` keyword to indicate a property of a class, that can be exposed to Interface Builder. For properties annotated with the `IBOutlet` keywords, we call it *outlets*.

So in the above code, we declare four outlets. Each outlet is going to connect with its corresponding UI object. Figure 9-21 depicts the connections.

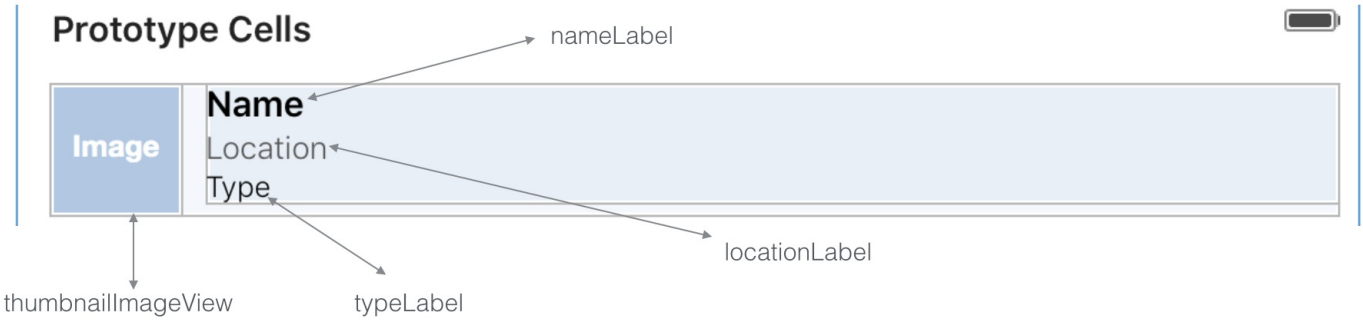


Figure 9-21. Outlet connections

@IBAction vs @IBOutlet

We've used @IBAction to indicate action methods when developing the HelloWorld app. What's the difference between @IBAction and @IBOutlet? @IBOutlet is used to indicate a property that can be connected with a view object in a

```
storyboard. For example, if the outlet is connected with a button, you can use
the outlet to change the color or title of the button. On the other hand,
@IBAction is used to indicate an action method that can be triggered by a
certain event. For example, when a user taps a button, it can trigger an action
method to do something.
```

Before we can establish a connection between the outlets of `CustomTableViewCell` class and the prototype cell in Interface Builder. We have to first set the custom class.

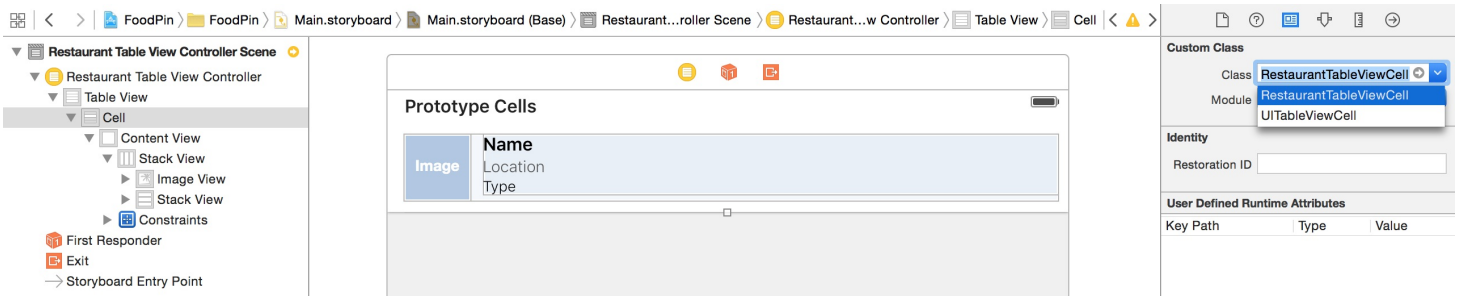


Figure 9-22. Set the custom class of the prototype cell

By default, the prototype cell is associated with the default `UITableViewCell` class. To assign the prototype cell with the custom class, select the cell in storyboard. In the Identity inspector, set the custom class to `CustomTableViewCell`.

Establishing the Connection

Next, we'll establish the connections between the outlets and UI objects in the prototype cell. In Interface Builder, right click the cell in the Document Outline view to bring up the Outlets inspector. Drag from the circle (next to thumbnailImageView) to the UIImageView object in the prototype cell (see figure 9-23). Xcode automatically establishes the connection when you release the button.

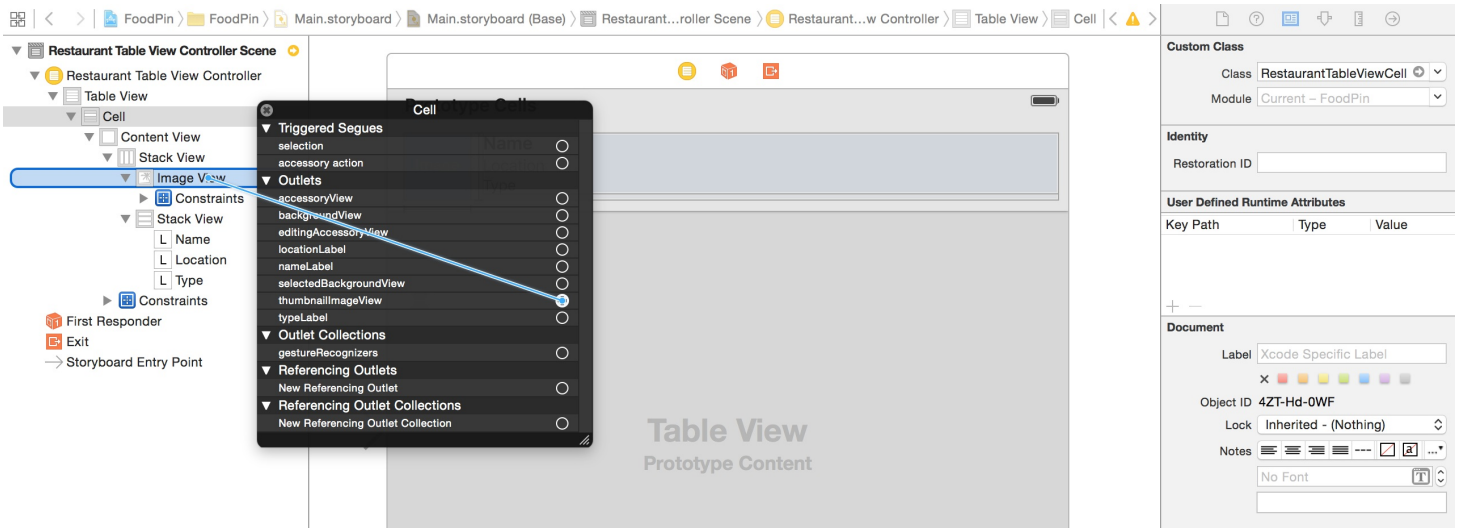


Figure 9-23. Connecting the outlet with the image view

Repeat the above procedures for the following outlets:

- locationLabel - connects to the Location label the cell
- nameLabel - connects to the Name label of the cell
- typeLabel - connects to the Type label of the cell

After you've made all the connections, the UI should look like the screenshot shown in figure 9-24.

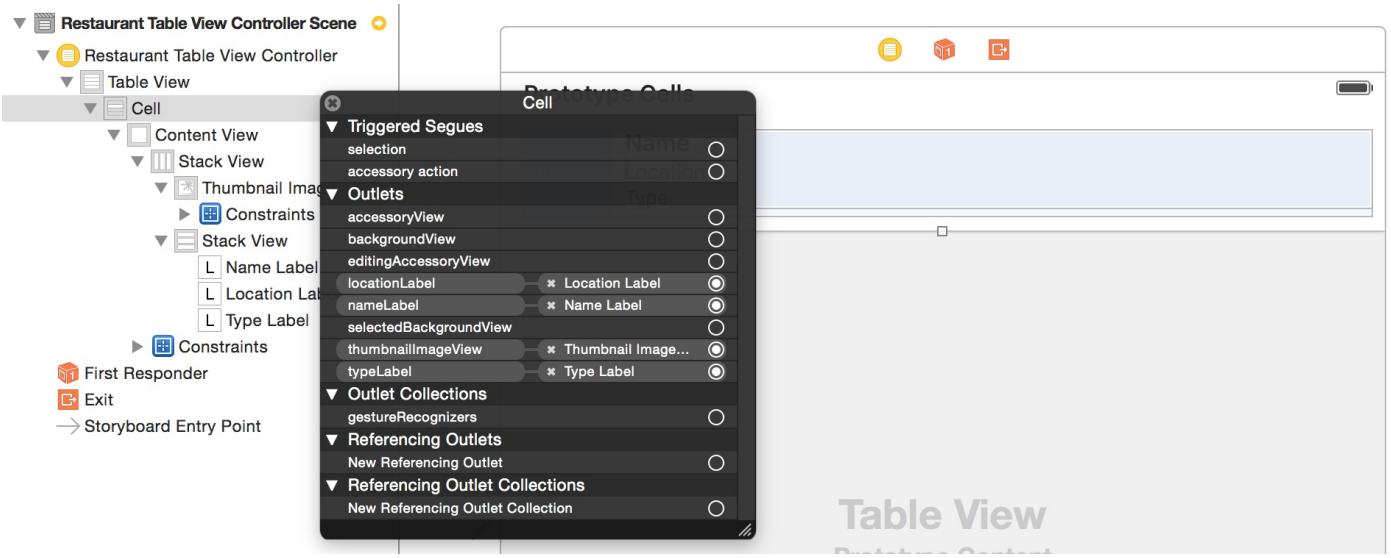


Figure 9-24. The outlet connections

Coding the Table View Controller

Finally, we come to the last part of the change. In the `RestaurantTableViewController` class, we're still using `UITableViewCell` (i.e. the default cell) to display the content. We need to modify a line of code in order to use the custom cell. If you look into the existing implementation of the `tableView(_:cellForRowAtIndexPath:)` method. The second line of code is:

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

I've explained the meaning of the `dequeueReusableCellWithIdentifier` method in the previous chapter. It is flexible enough to return any cell types from the queue. By default, it returns a generic cell of a `UITableViewCell` type. In order to use the `RestaurantTableViewCell` class, it's our responsibility to "convert" the returned object of `dequeueReusableCellWithIdentifier` to `RestaurantTableViewCell`. This conversion process is known as downcasting. In Swift, we use the `as!` keyword to perform a forced conversion. Therefore, change the above line of code to the following:

```
let cell = tableView.dequeueReusableCellWithIdentifier(cellIdentifier,
forIndexPath: indexPath) as! RestaurantTableViewCell
```

as! and as?

```
Downcasting allows you to convert a value of a class to its derived class. For
example, RestaurantTableViewCell is a child class of UITableViewCell. The
dequeueReusableCellWithIdentifier method always returns a UITableViewCell
object. If a custom cell is used, this object can be converted to the specific
cell type (e.g. RestaurantTableViewCell). Prior to Swift 1.2, you can just use
the "as" operator for downcasting. However, sometimes the object may not be
converted to a specified type. Therefore, from Swift 1.2 and onwards, Apple
introduced two more operators: as! and as?. If you're quite sure that the
downcasting can perform correctly, use "as!" to perform the conversion. In case
you're not sure if the value of one type can be converted to another, use "as?"
to perform an optional downcasting. You're required to perform additional
checking to see if the downcasting is successful or not.
```

I know you can't wait to test the app, but we need to change a few more lines of code. The lines of code below set the values of restaurant name and image:

```
// Configure the cell...
cell.textLabel?.text = restaurantNames[indexPath.row]
cell.imageView?.image = UIImage(named: restaurantImages[indexPath.row])
```

Both `textLabel` and `imageView` are properties of the default `UITableViewCell` class. Because we're now using our own `RestaurantTableViewCell`, we need to use the properties of the custom class. Change the above lines to the following:

```
// Configure the cell...
cell.nameLabel.text = restaurantNames[indexPath.row]
cell.thumbnailImageView.image = UIImage(named: restaurantImages[indexPath.row])
```

Now you're ready to go. Hit the Run button and test the app. Your app should look like the one shown in figure 9-25. Try to rotate the simulator. The app also works in landscape orientation.

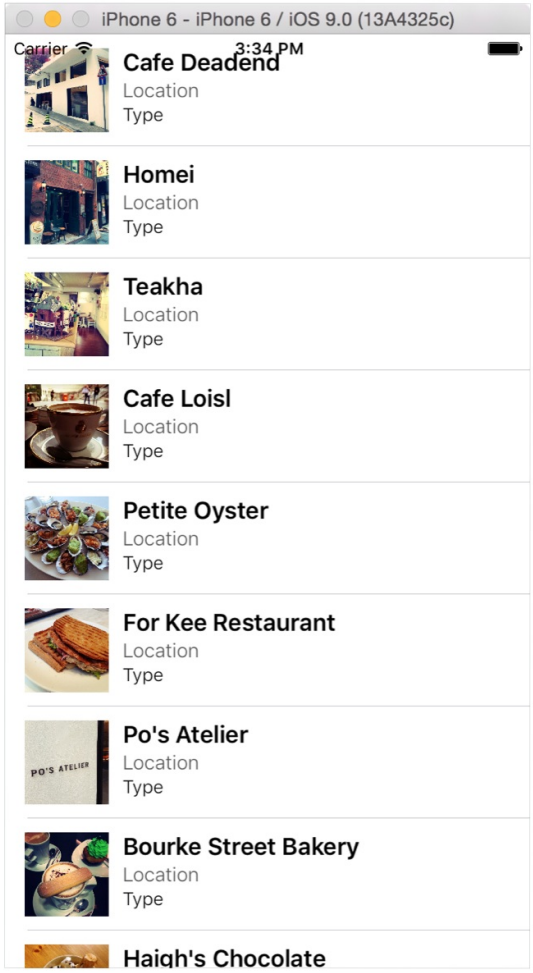


Figure 9-25. FoodPin app with custom table view cells

It is a huge improvement from the previous version of the app. We're going to make it even better by changing the thumbnail into circular image.

Circular Image

Since the release of iOS 7, iOS favors circular image over square image. You can find circular icons or images in stock apps such as Contacts and Phone. Wouldn't it be great to make all the restaurant images circular? You do not need Photoshop to tweak the images. What you need is just two lines of code. Every view in the UIKit (e.g. UIView, UIImageView) is backed by an instance of the `CALayer` class (i.e. layer object). The layer object is designed to manage the backing store for the view and handles view-related animations.

The layer object provides various attributes that can be set to control the visual content of the view such as:

- Background color
- Border and border width
- Shadow color, width, etc
- Opacity
- Corner radius

The corner radius is the attribute, which we use to draw rounded corners. Xcode provides two ways to edit the layer properties. You can directly update its properties through code. Here is the lines of code the following lines of code for changing the corner radius of the image view:

```
cell.thumbnailImageView.layer.cornerRadius = 30.0
cell.thumbnailImageView.clipsToBounds = true
```

An even easier way is to make the change through Interface Builder. First, select the image view in the stack view. Go to the Identity inspector, click the Add button (+) in the lower left of the user defined runtime attributes editor. A new runtime attribute appears in the editor. Double click on the Key Path field of the new attribute to edit the key path for the attribute. Set the value to `layer.cornerRadius` and hit Return to confirm. Click on the Type attribute and choose `Number`. Lastly, set the value to `30`. To make a circular image from a square image, the radius is set to half the width of the image view. Here, the width of square image is 60 points. Thus, the corner radius is set to 30 points.

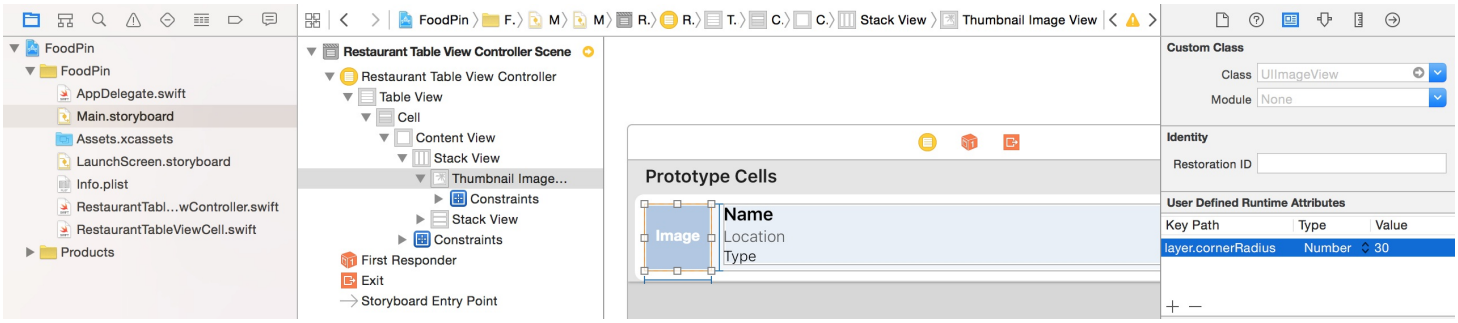


Figure 9-26. Set the runtime attribute to make the corner round

When the image view is initialized, this runtime attribute will be automatically loaded to make

the corner round. There is one more thing you have to configure before the circular image works properly. Select the image view and go to the Attributes inspector. In the Drawing section, enable the *Clip Subview* option. This causes the content to be clipped to the rounded corners.

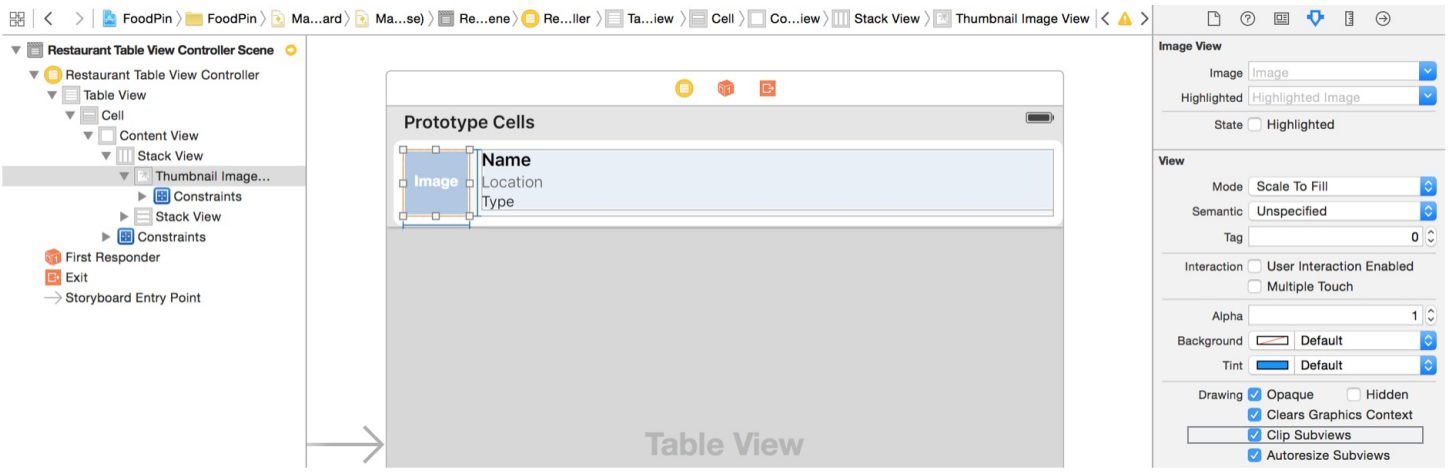


Figure 9-27. FoodPin app with circular images

Now compile and run the app. The UI looks even better, right? Without writing a line of code, we change the square images to circular one.

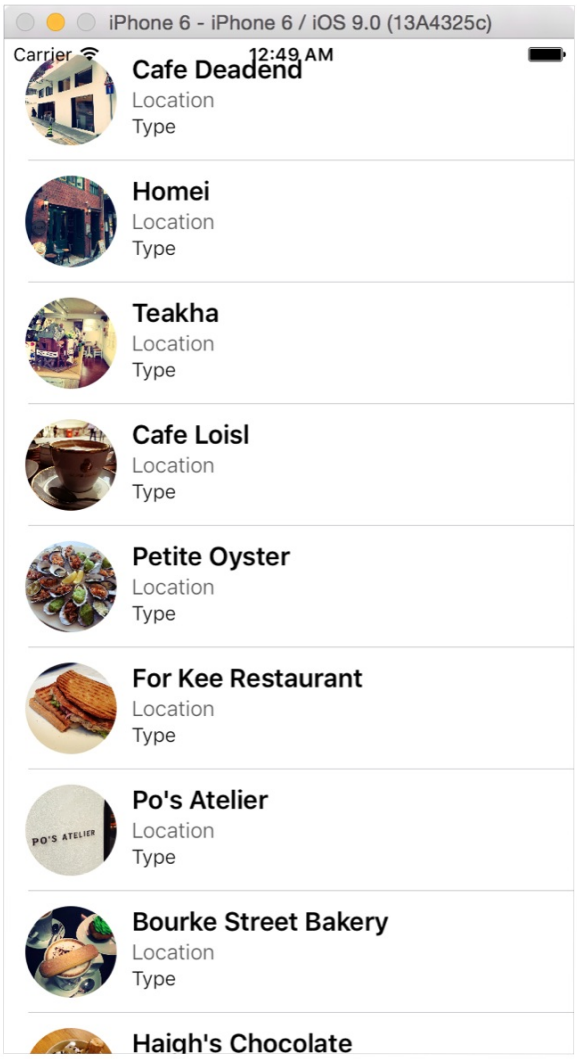


Figure 9-28. FoodPin app with circular images

You're free to alter the value of corner radius. Try to change the corner radius to 10 points and see what you'll get.

Your Exercise

As of now, the app simply displays "Location" and "Type" for all rows. As an exercise, I'll leave it to you to fix the issue. Try to edit the source code to update the location and type labels. Below are the two arrays you need:

```
var restaurantLocations = ["Hong Kong", "Hong Kong", "Hong Kong", "Hong Kong",
"Hong Kong", "Hong Kong", "Hong Kong", "Sydney", "Sydney", "Sydney", "New
```

```
York", "New York", "New York", "New York", "New York", "New York", "New York",
"London", "London", "London", "London"]

var restaurantTypes = ["Coffee & Tea Shop", "Cafe", "Tea House", "Austrian /
Causual Drink", "French", "Bakery", "Bakery", "Chocolate", "Cafe", "American /
Seafood", "American", "American", "Breakfast & Brunch", "Coffee & Tea", "Coffee
& Tea", "Latin American", "Spanish", "Spanish", "Spanish", "British", "Thai"]
```

The previous exercise may be too easy for you. Here is another challenge. Try to redesign the prototype cell and see if you create an app like this (see figure 9-29).

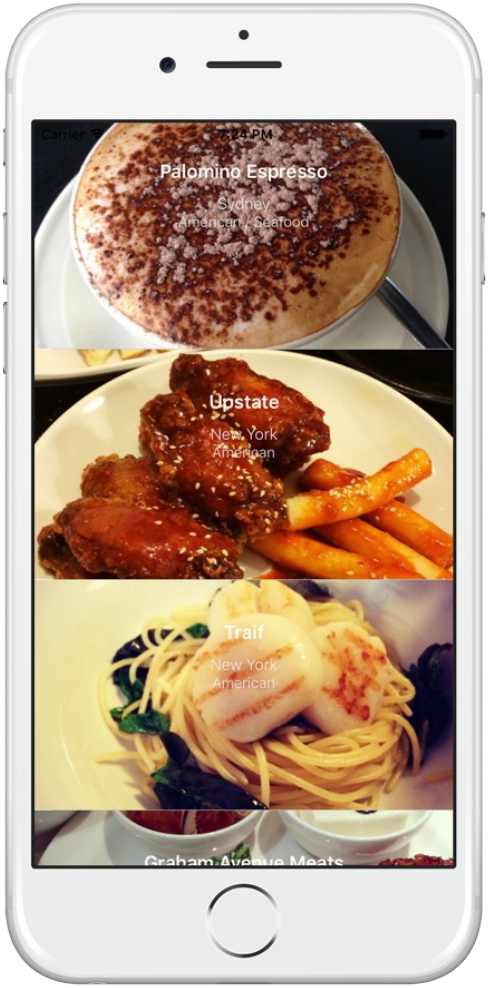


Figure 9-29. Redesigned Custom Table

Summary

Congratulations! You've made a huge progress. If you understand the ins and outs of cell

customization, you're ready to make some awesome UIs. Table view is the backbone for most iOS apps. Unless you're building a game, you'll probably implement a table view in one way or another when building your own apps. Table view customization may be a bit complex for some of you. So take some time to work on the exercise and play around with the code. Remember "learn by doing" is the best way to learn coding.

For reference, you can download the complete Xcode project from https://www.dropbox.com/s/xrvpamowqykwfrb/FoodPinCustomTable.zip?dl=0. If you can't figure out how to complete the exercise, you can download the solution from https://www.dropbox.com/s/nnsr863vyusxfdk/FoodPinCustomTableSolution.zip?dl=0.