SocialAPP

Practice

PART 2

In practice part 1, you began work on SocialApp, a Twitter client; we presented an in-depth look at building an application structure with storyboards, explaining how to tie scenes together with segues. You also learned about the principles of combining the View element with the Controller element. In addition, you prepared the scenes for SocialApp and tried them all together with segues.

In this age of big data, as developers, we often find ourselves needing a way to display large amounts of data to users in a concise and structured manner. In iOS, Apple has provided the table view and collection view for this purpose.

This 2nd part of the practice focuses on the table view. Then the 3rd part of the practice focuses on the collection view. You explore how each view is structured and how you can use Xcode to alter their appearance. Additionally, you learn about creating custom cells, where you subclass UITableViewCell to customize the elements in your cells. You will see how the segue identifiers specified in 1st part of the practice allow you to share data between view controllers, and you learn about a variety of ways to obtain data from the Internet and display that in an application.

Because this practice is reliant on your having access to at least one Twitter account, if you don't have one, it would be a good idea to register at www.twitter.com. Ensure that you've created the account and that you've "followed" some other Twitter users; whatever your personal feelings are about Twitter, remember that you don't have to use it beyond this practice, and you can delete your account when you're ready.

After you finish SocialApp, you will enhance the application using collection views, a close cousin of the table view. Collection views have methods and properties similar to those of table views and much of what you learn about table views can be directly applied to collection views.

What Is a Table View?

A table view represents an instance of the UITableView class; it presents the user with a single column of cells listed vertically. Table views provide developers with a great way of displaying a large number of options or data to the user, such as in a Twitter application where you can scroll through potentially hundreds or even thousands of tweets. They can also be used to neatly lay out application settings, exploiting the table view's hierarchical structure to take the user from high-level categories right down to granular details and microsettings. The flexibility of table views means you can find one in nearly every application, but you may not recognize them right away. Figure 1 shows some of the table views in use through iOS and the default applications.

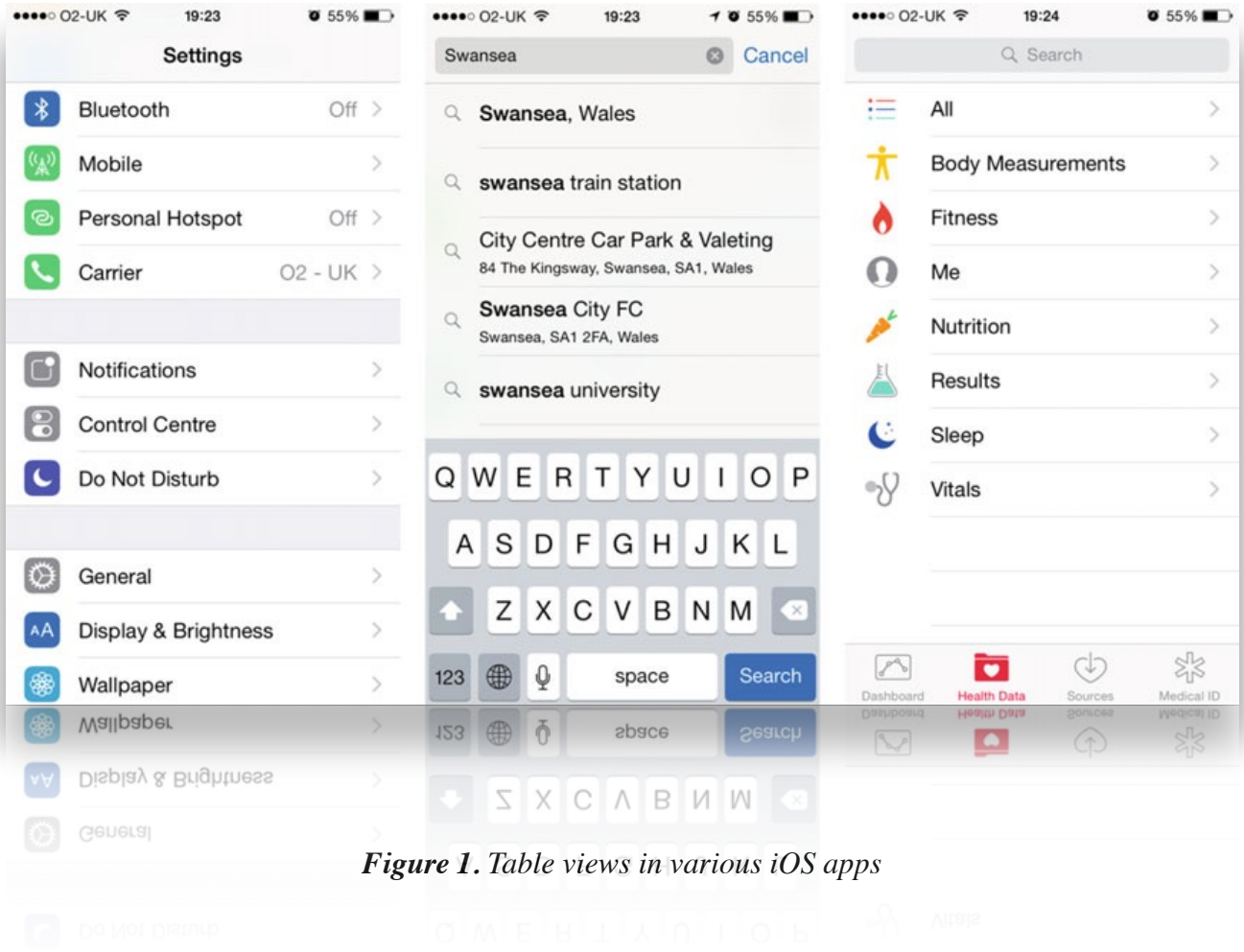

Figure 1. *Table views in various iOS apps*

Because of their popularity, Apple spends a lot of time improving the flexibility and feature set of table views with each successive release of iOS. This makes it easy to add features found in many of Apple's own applications such as Pull To Refresh.

Table View Composition

Before you get into the configuration of table views, it's good to have a basic idea of their composition and key components, which are layered on top of each other. Figure 2 shows a visual breakdown of the different elements in a table view. Let's examine these components in more detail:

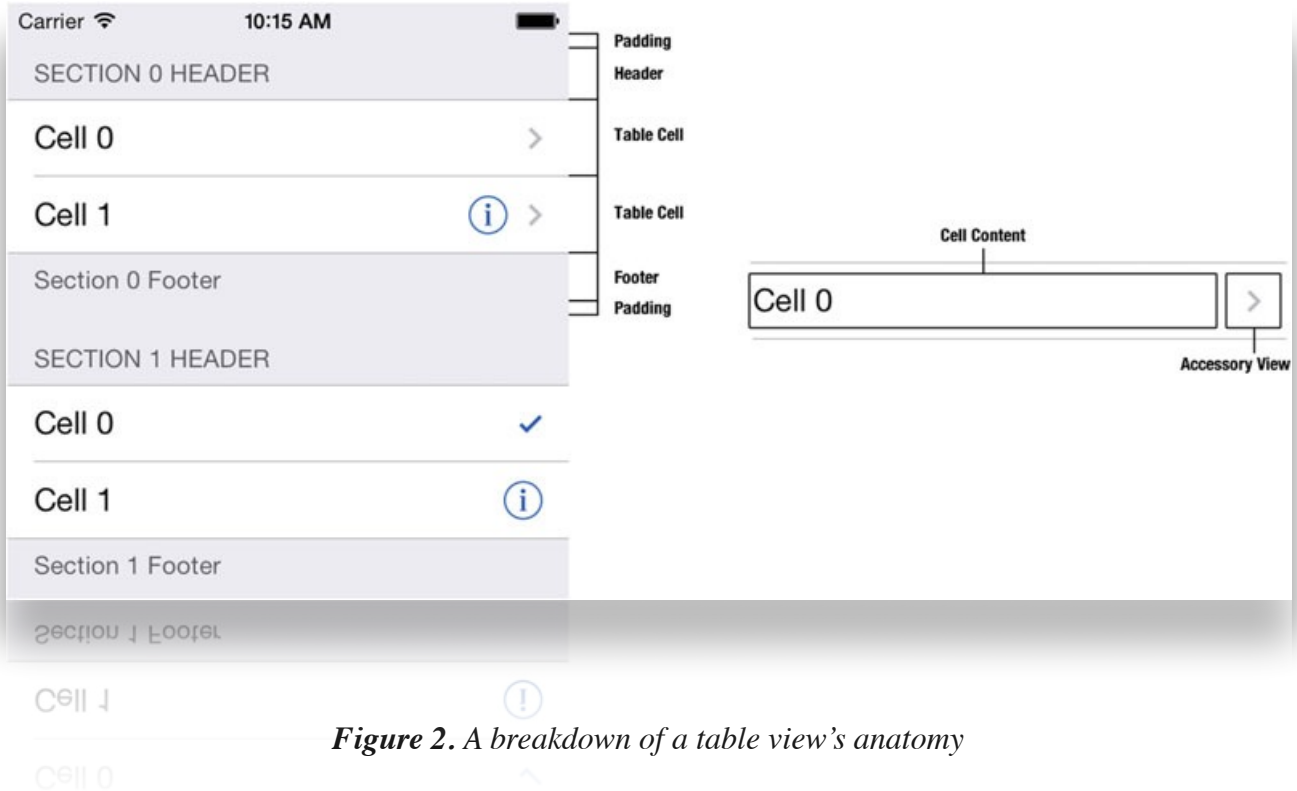

Figure 2. *A breakdown of a table view's anatomy*

- *View*: The foundation in the hierarchy of elements. It sits at the bottom of the stack, coordinating all the child components. The View element controls the overall look and feel of the table and anchors all the delegate methods together.

- *Section*: The next item in the table view stack. Sections are useful for breaking up tables, grouping cells together, and providing a header and footer for the group.

- *Cell*: Represents a row in the table view and can have varying states, such as when it's being edited, that affects the number of key areas in a cell. The two default areas of a table cell are the cell content and the accessory view. You may think the cell content area is self-explanatory, but it's actually a varied element and, depending on the style of cell, can contain an image, a title, and a subtitle. The accessory view can contain
a disclosure indicator, such as an arrow, a detail accessory for providing more information about the row, or both.

Table View Styles

Table views haven't changed visually since they were overhauled in iOS 7. It's worth noting that when Apple released iOS 7, most areas received a visual makeover, but one of the greatest changes was seen in table views. There are two separate styles for table views, but they are no longer as visually distinct as they once were. Figure 3 shows a plain style table view on the left and a grouped style table view on the right.

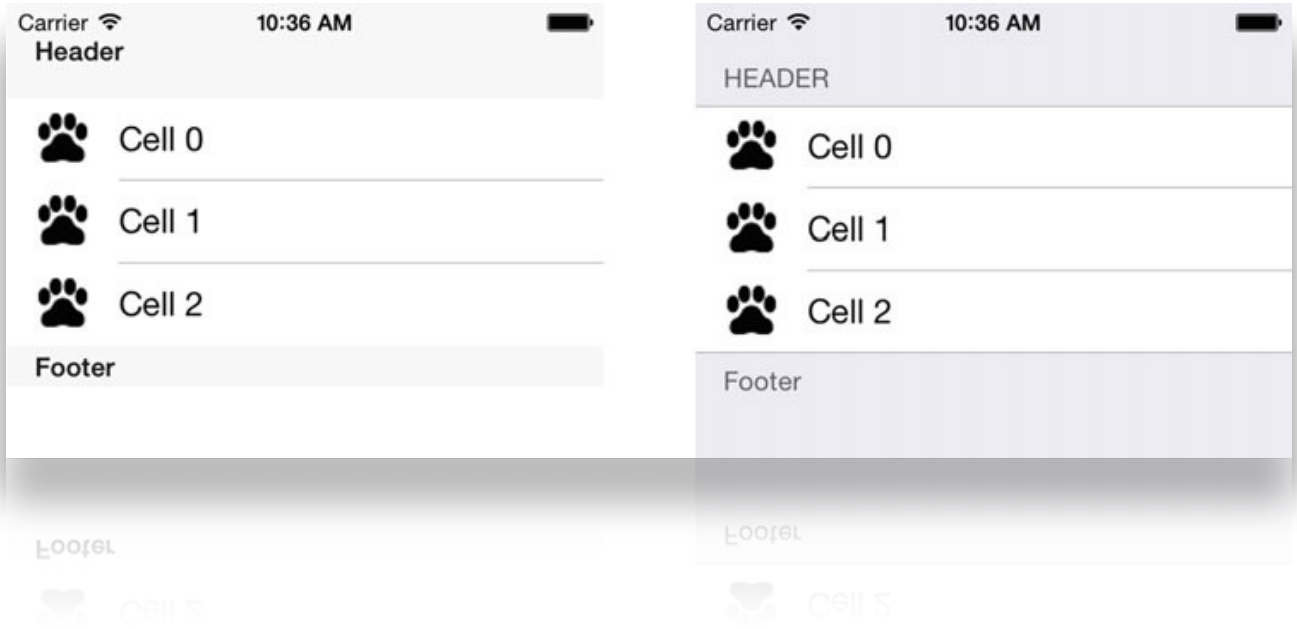

Figure 3. *A plain table view (left) and a grouped table view (right)*

In practical terms, you would typically use the plain style to list large amounts of data, usually in a single section with no header or footer. The grouped style is ideal for situations where you have a static list of data and you want to group similar items together, such as the different configuration options in the Settings application in iOS (first image in Figure 1).

Configuring the Accounts View

In the Twitter client SocialApp, the Accounts view is one of the simplest to set up, because it uses a plain table view with very little customization. However, you also get your first introduction to the Social and Accounts frameworks, which allow you to access details about accounts set up on the device and then use the account to authorize requests to social media sites such as Twitter or Facebook.

This is the first time you've had to go back to work on an existing project. If you haven't already opened the SocialApp project that you started in part 1 of the practice, start by opening Xcode and choosing File ➤ Open (+O) to locate SocialApp.xcodeproj in the folder where you created it. Click Open, as shown in Figure 4.

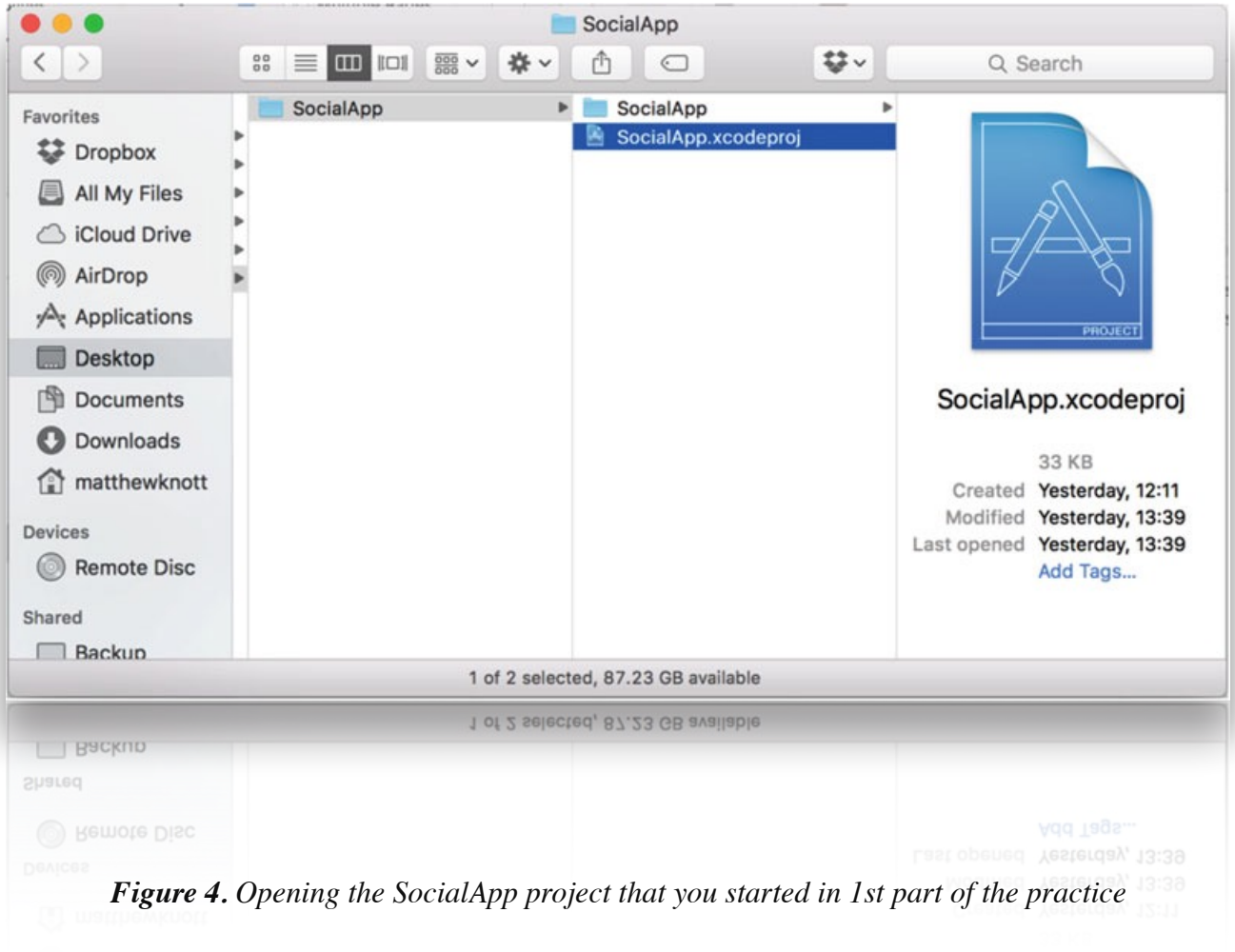

Figure 4. *Opening the SocialApp project that you started in 1st part of the practice*

Before you configure the table view for the Accounts scene, let's explore the options available in Xcode for altering the table view itself; later in this practice, you customize the cells. With the project open, select Main.Storyboard from the Project Navigator and position the storyboard so that you're looking at the Accounts scene, as shown in Figure 5.

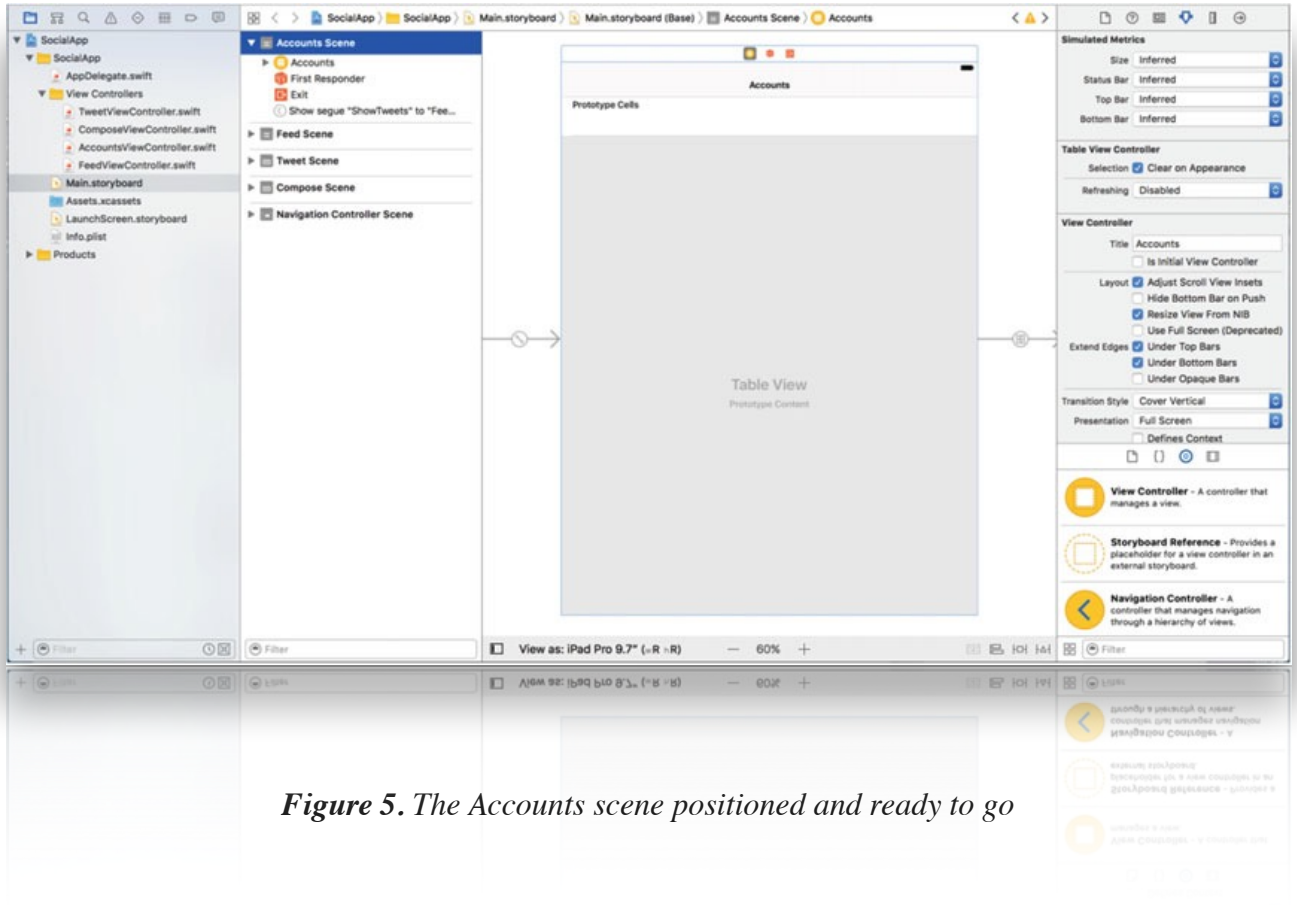

Figure 5. *The Accounts scene positioned and ready to go*

The Key Attributes of Table Views

Select the table view in the Accounts scene; you can do this by either clicking the words Table View Prototype Content in the middle of the view or selecting Table View from the Accounts scene in the Document Outline. Now open the Attributes Inspector. If you're looking at the right object, the first segment in the Attributes Inspector is Table View. Figure 6 shows the default attributes of a table view in Xcode 8.

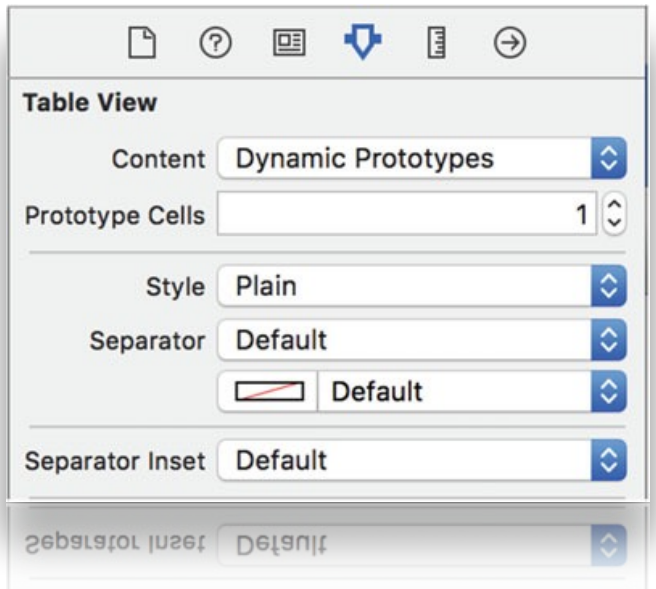

Figure 6. *The default attributes of a table view in Xcode7*

When configuring a table view, most of the time you're after one of these first five attributes, because they have the largest influence on the table view's structure and style. Let's look at these options in more detail.

- *Content (Dynamic Prototypes)*: Unlike many of the attributes you can configure in Xcode, the content type can't be set programmatically. It's purely an Xcode thing. Selecting Dynamic Prototypes as the content type gives you a single table cell by default. The idea is that often, you have only one cell style in a table, so you configure the one cell, and in code you reuse it for each row. Using the Prototype Cells attribute that is available for this content type, you can increase the number of prototypes and have one for each distinct cell type. If you're using dynamic prototypes, you must customize the code to be able to view any cells in your view. This is the content type you'll use for all the table views in this application.

- *Content (Static Cells)*: When using static cells, you use Xcode to specify how many sections exist in your table view and how many cells appear in each section. You can then create segues from these individual cells to other scenes in your storyboard. One of the biggest differences between static cells and dynamic prototypes is that, although the delegate methods that table views use to specify the number of sections and cells *can* be used to influence the table view, they can also be completely removed, leaving the attributes specified in Xcode as the controlling factor.

- *Style*: This is where you choose between plain or grouped styles. As you can see in Figure 3, there is very little difference between the two in iOS 9. Both styles can be extensively customized using code to change sizes and colors.

- *Separator*: The separator in a table view is the line that appears between cells. This attribute gives you four options to choose from: Default, Single Line, Single Line Etched, and None. In reality, these four options are only two. The Default style is the same as the Single Line style, and Single Line Etched is the same as None, because as it was deemed not compatible with the flat design approach Apple took with iOS 7; effectively, this style and its code equivalent UITableViewCellSeparatorStyleSingleLineEtched have been deprecated.

- *Separator Insets*: This was introduced in iOS 7 and Xcode 5. In earlier versions of iOS, the separator spanned the full width of the cell; however, in iOS 10, there is a small indent on the left side of the cell by default. By setting the Separator Insets attribute to Custom, you can specify a custom left and right indent, depending on the style you want to achieve.

Note longer considered acceptable for use. Often when a class is deprecated, it remains available for use; however, because it's unsupported, it may have consequences with other areas of your application and may cause unexpected issues with other classes.

Manipulating Static Table Views

Let's take a look at how Xcode allows you to manipulate a static layout, before going back and implementing the dynamic prototype system you're using for this scene:

1. Select the Accounts table view and then open the Attributes Inspector. Set the Content attribute to Static Cells; you should notice that the single cell you had becomes three cells, and the second attribute becomes Sections.

2. Let's increase the number of sections so that there are two groups of cells to work with: change the Sections attribute to 2. Your table should now resemble the one shown in Figure 7.

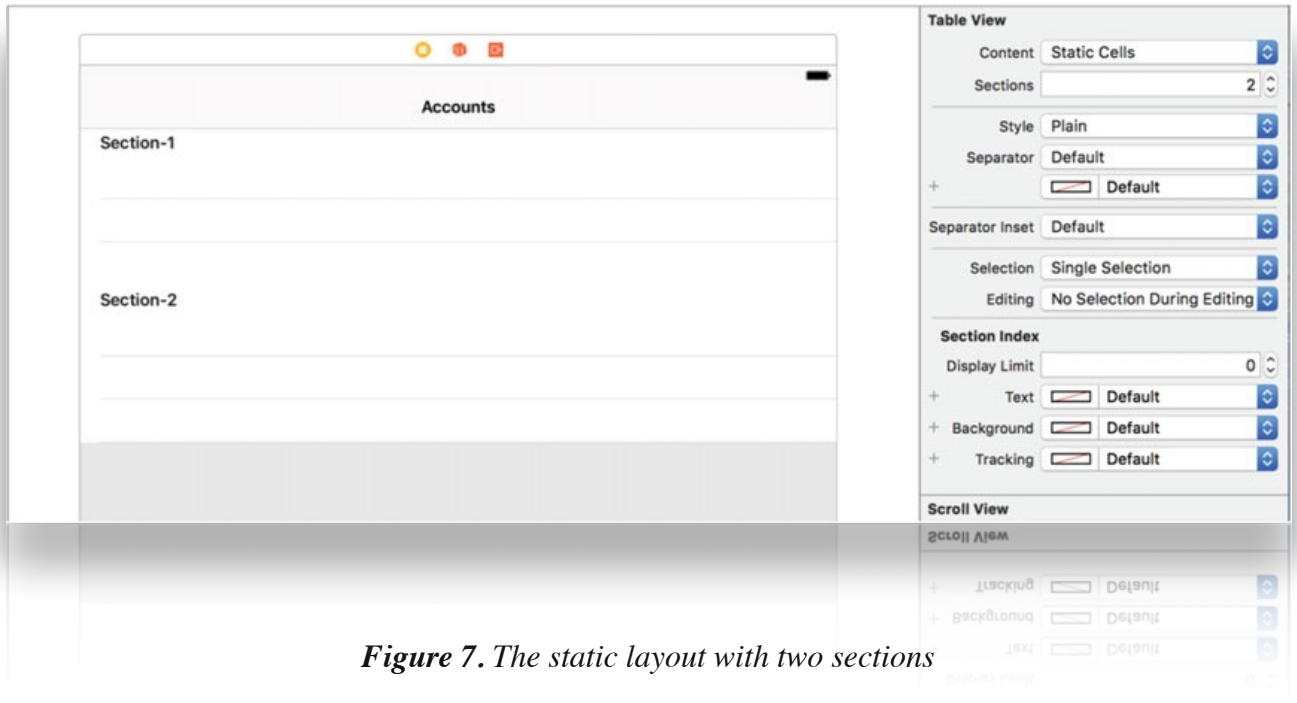

Figure 7. *The static layout with two sections*

3. The number of cells in a section is controlled directly by the section attributes. Select the first section by clicking Section-1 in the view or by expanding Table View in the Document Outline and selecting Section-1, as shown in Figure 8.

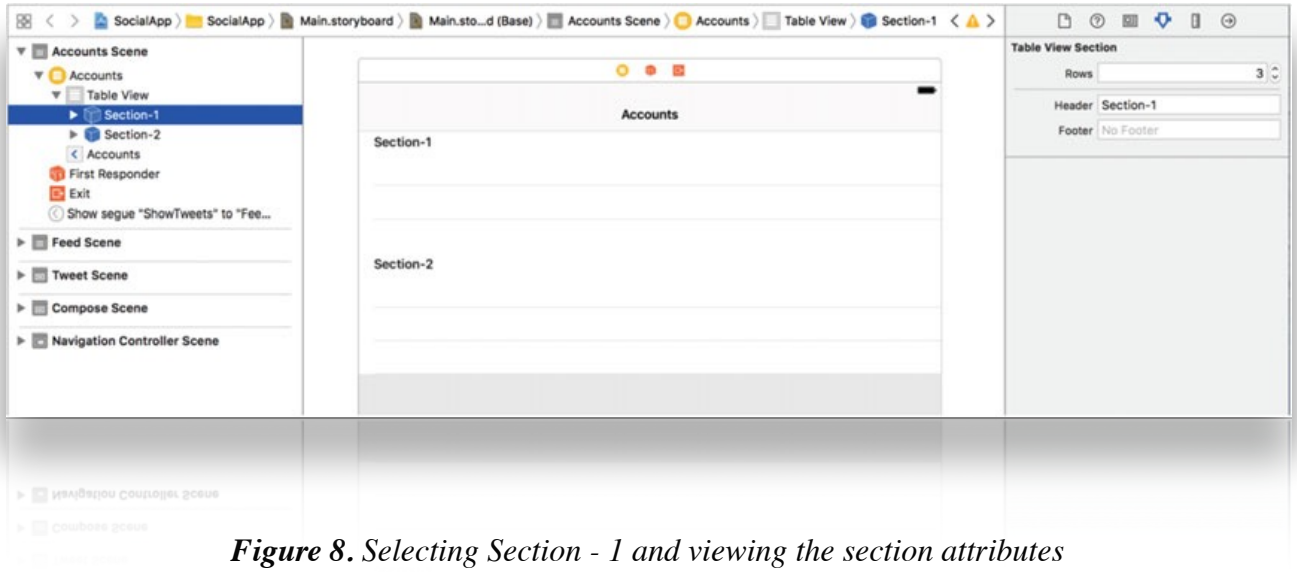

Figure 8. *Selecting Section - 1 and viewing the section attributes*

4. In the Attributes Inspector, the terminology changes slightly from cells to rows. Looking at the attributes, you can see that here you specify the header and footer value of your section and also the number of rows. Feel free to try setting your own values for these attributes and see how the table changes.

5. You can also delete individual cells and move them around. In Section-2, delete two cells; click the cells individually and then press the Backspace key to remove them. Select the single remaining cell.

6. In the Attributes Inspector, change Style to Basic; doing so adds the word Title to the cell.

7. Double-click the word Title to edit it, as shown in Figure 9, and change it to readCell 1.

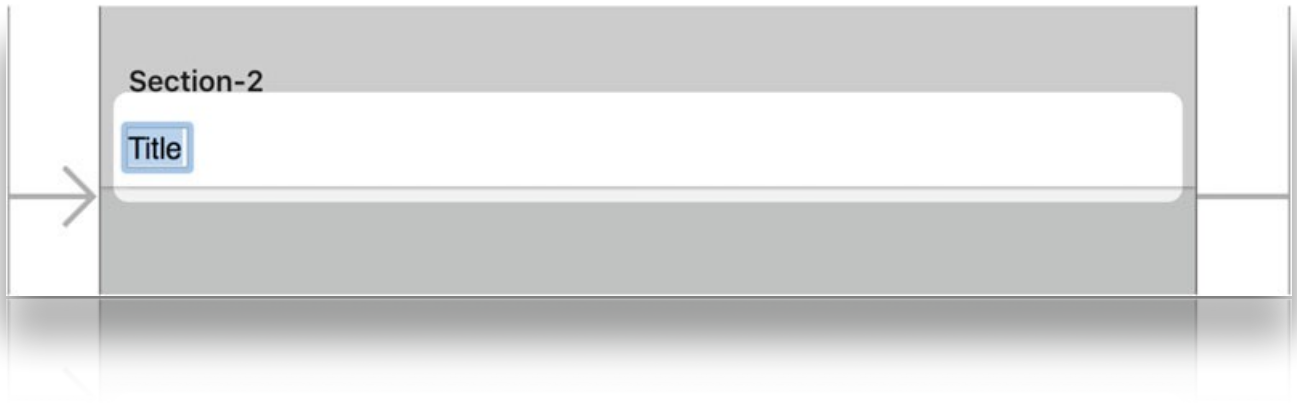

Figure 9. *Changing the title of the basic cell*

8. Press the Return key to commit the change and then reselect Section-2. Now increase the number of rows from one to three by changing the Rows attribute to 3. This allows you to clone your row three times. This is a really handy way of duplicating a custom cell when using static cells instead of dynamic prototypes.

9. Rename each of your new cells by double-clicking Cell 1 and changing them to 2 and 3, respectively, so that Section-2 resembles Figure 10.

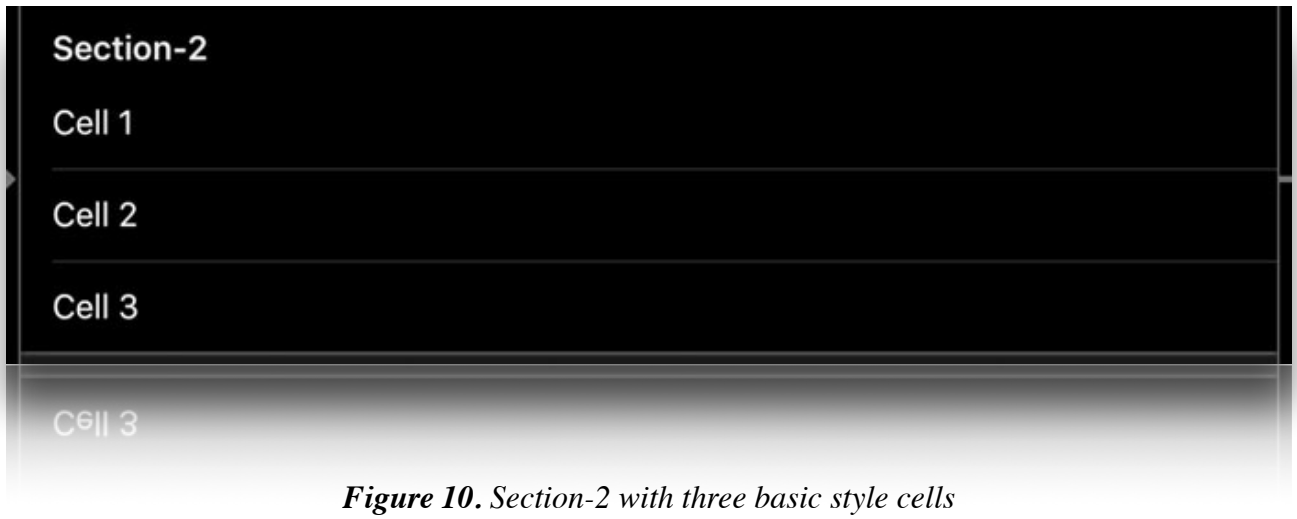

Figure 10. *Section-2 with three basic style cells*

10. To demonstrate how easy it is to reorder a static table view, highlight Cell 3 with a single click and then drag it to the top of Section-2. A solid blue line appears, as shown in Figure 8-11. You can also move cells between sections, meaning that changing your layout needn't be a chore.

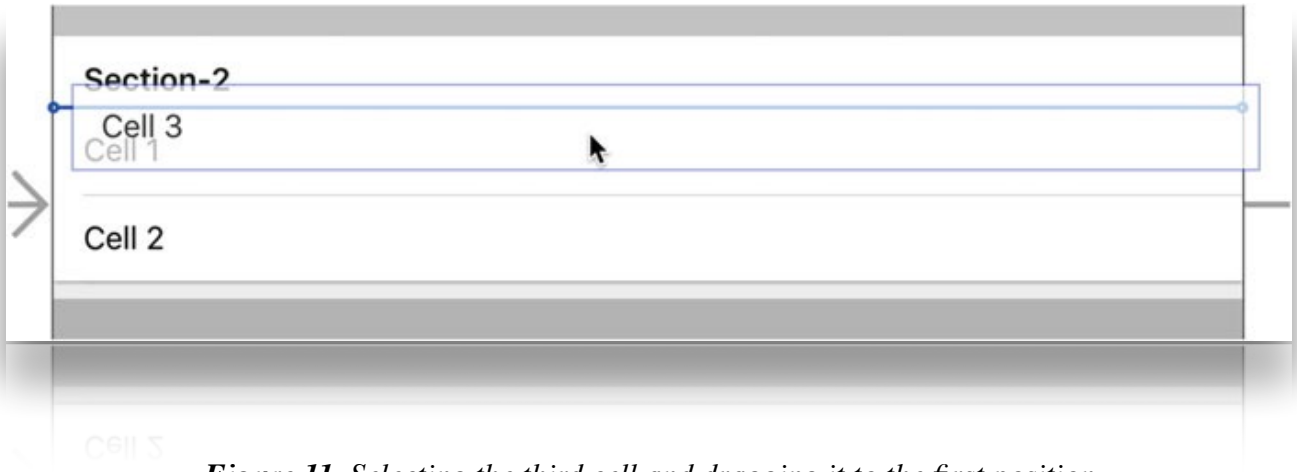

Figure 11. *Selecting the third cell and dragging it to the first position*

11. Now that you've seen the various ways Xcode lets you manipulate static layouts, you're ready to get the Accounts scene built up and working. Reselect the table view, change the Content attribute to Dynamic Prototypes, and ensure that Style is set to Grouped. This will leave you with three cells in the single section.

12. You only want one prototype cell, so delete any excess cells by manually highlighting them and pressing the Backspace key until you're left with a single cell, ready for customizing.

13. Highlight the one remaining cell and go to the Attributes Inspector. The style should currently be set to Custom, which is fine because you'll be setting the content programmatically.

14. Set the Accessory attribute to Disclosure Indicator, which adds the indicator arrow on the right side of the cell.

15. Give your cell an identifier so that you can refer to it in code and reuse it efficiently. Set the Identifier attribute to AccountCell. The cell's attributes should resemble those shown in Figure 12.

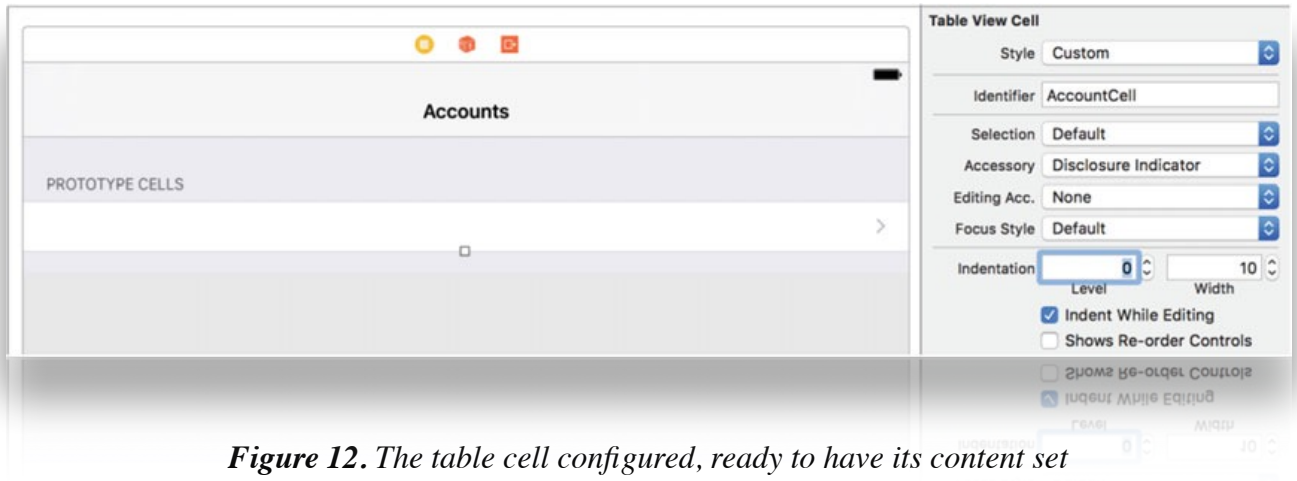

Figure 12. *The table cell configured, ready to have its content set*

You're finished with the layout and design of your table view, and it's time to add the code to access the Twitter accounts on the device and display them in the table. To do this, you need to add the Accounts and Social frameworks to the project.

The Accounts and Social Framework

In previous part, you took advantage of some of the Apple-provided frameworks, and in this part you use two more: Accounts.framework and Social.framework. Like the other frameworks you've used, they make potentially complex and intensive tasks much simpler. Based on the fact that this is the first time you've worked with these two frameworks, it's also worth noting that they work really well together, and you sometimes need to use both in order to make the most of their functionality. Accounts framework classes are prefixed with AC; in this project, you create an instance of the ACAccount class to hold the details of a selected Twitter account. You hand that ACAccount object to other view controllers as you navigate through the project and then combine it with one of the Social framework's classes, SLRequest, which uses the ACAccount object to authenticate requests with Twitter.

If you've worked with Twitter's APIs in the past, you know that authentication involves a process called *three-legged OAuth* that sends a number of requests back and forth with Twitter. The combination of the ACAccount object with the SLRequests means you don't have to do any of that. iOS does all the running around so that you're free to focus on functionality and how you handle the requests to Twitter.

Now that you understand the roles of the frameworks you need to add to the project, you're ready to begin writing the code that will display Twitter accounts in the table view.

Retrieving and Displaying Twitter Accounts

To finish this scene, you need to achieve two objectives: retrieve an array of Twitter accounts registered on the device or in the Simulator and display them for users to choose from. Let's start by setting up the view controller's header, importing the frameworks, and creating the properties that are needed for this scene. As in previous practice, we explain what needs to be done bit by bit and then review the code at the end of the process:

1. Open AccountsViewController.swift. You first need to import the Accounts framework that is used for this scene. After the line import UIKit, add the following highlighted line:

Note API stands for *application programming interface*. An API is a mechanism that specifies how different pieces of computer software interact with one another. In iOS, Apple uses the term *API* to describe new classes in frameworks. When we talk about the Twitter API, we don't mean classes in the Social framework, but rather the Twitter REST API, which you can learn more about at http://dev.twitter.com.

```
import UIKit
import Accounts
```

2. You need to create an array to store the retrieved accounts and make them available to all functions in this class. Add the highlighted code:

```
import UIKit
import Accounts
class AccountsViewController: UITableViewController {
    var twitterAccounts : NSArray?
```

3. You need to declare an instance of ACAccountStore. The ACAccountStore class is the gateway to the list of Twitter and other social media accounts stored in iOS; you declare it and then try to get permission from the user to access the Twitter account. If the user grants permission, the object becomes initialized. Add the highlighted code after the NSArray:

```
var twitterAccounts : NSArray?
var accountStore : ACAccountStore?
```

So far, you've added a reference to the Accounts framework, created an array called twitterAccounts, and created an ACAccountStore object to manage the retrieval of Twitter accounts. Your initial code should look like this:

```
import UIKit
import Accounts
class AccountsViewController: UITableViewController {
    var twitterAccounts : Array<Any>?
    var accountStore : ACAccountStore?
```

4. To get into the nitty-gritty, scroll down until you see the viewDidLoad function. The first thing you want to do when the view loads is initialize the ACAccountStore instance. That doesn't mean you're accessing the accounts; you're just initializing the object so that it can be interacted with. Drop down a line after super.viewDidLoad() and type the following highlighted line:

```
super.viewDidLoad()
accountStore = ACAccountStore()
```

5. You'll use the requestAccessToAccountsWithType method of the accountStore object. This method needs to be told the type of account to which you're requesting access. You do this by creating an ACAccountType object and then using another accountStore method: accountTypeWithAccountTypeIdentifier. Drop down a line, and add this highlighted code (as a single line):

```
super.viewDidLoad()
accountStore = ACAccountStore()
let accountType : ACAccountType = accountStore!.accountType(withAccountTypeIdentifier:
ACAccountTypeIdentifierTwitter)
```

6. You're at the stage where you want to ask the user for permission to use their Twitter accounts in the application using the requestAccessToAccountsWithType method. When this method is accessed, it creates an alert for the users to either grant or deny the request to access their Twitter account. Add the following highlighted code:

```
let accountType : ACAccountType = accountStore!.accountType(withAccountTypeIdentifier:
ACAccountTypeIdentifierTwitter)

accountStore?.requestAccessToAccounts(with: accountType, options: nil,
    completion: { granted, error in
})
```

7. Notice that you pass the accountType object into the method to specify that it wants Twitter account access, and you handle completion using a code block into which you add the logic as to whether granted returned yes or no. Because you only want to look at the accounts available *if* access was granted, that should be the next thing you check. To do so, add the highlighted if statement in the code block, as shown next:

```
let accountType : ACAccountType = accountStore!.accountType(withAccountTypeIdentifier:
ACAccountTypeIdentifierTwitter)

accountStore?.requestAccessToAccounts(with: accountType, options: nil,
   completion: { granted, error in
   if(granted)
   {

   }

})
```

8. With access granted, you need to populate the twitterAccounts object with all the available Twitter accounts on the device. Here you use the accountsWithAccountType method of the accountStore and reuse the accountType object to restrict the request to Twitter accounts. In the if statement, add the following highlighted code:

```
if(granted)
{
   self.twitterAccounts = self.accountStore!.accounts(with: accountType)
}
```

9. Although the user has granted access, you need to check whether user has added any Twitter accounts to the device. This is done by checking the twitterAccounts count property in an if else statement. After the previous line, drop down and add this code:

```
if(granted)
{
   self.twitterAccounts = self.accountStore!.accounts(with: accountType)
   if (self.twitterAccounts!.count == 0)
   {

   } else {

   }

}
```

10. If there are no Twitter accounts stored in iOS, you want to summon an alert view and tell the user that no accounts were found. Because you're running on an arbitrary thread and all interface changes need to be executed on the main thread, you add a Grand Central Dispatch

call to execute the display of the alert view. Add the following highlighted code in the first set of braces:

```
if (self.twitterAccounts!.count == 0)
{
   let noAccountsAlert : UIAlertController = UIAlertController(title: "No Accounts Found",
      message: "No Twitter accounts were found.",
      preferredStyle: UIAlertControllerStyle.alert)
   let dismissButton : UIAlertAction = UIAlertAction(title: "Okay",
      style: UIAlertActionStyle.cancel) {
      alert in
      noAccountsAlert.dismiss(animated: true, completion: nil)
   }
   noAccountsAlert.addAction(dismissButton)
   DispatchQueue.main.async() {

       self.present(noAccountsAlert, animated: true, completion: nil)

   }

}

else {

}
```

11. If there are Twitter accounts, they're added to the twitterAccounts object, and you just need to tell the table view to reload the data shown in the table. This is done by accessing the reloadData method. In the second set of braces for the else statement, add this highlighted lines of code:

```
else {

      DispatchQueue.main.async() {
          self.tableView.reloadData()
      }

}
```

You've added the code that pulls together a dataset for the table view in the shape of the twitterAccounts array. Now you need to get that data into the cells by using two of the UITableView class's delegate methods: numberOfSectionsInTableView and numberOfRowsInSection. These two methods control the number of sections displayed in the table and also the number of table cells per section by returning an NSInteger, a numeric value that the table view interprets. Setting up these

two methods is pretty simple in this application: you only ever have one section, and the number of cells is the count property from the twitterAccounts object. Furthermore, Apple has already added stubs for these two methods.

12. Scroll down until you find the numberOfSections(in:) method; or click AccountsViewController in the jump bar and select numberOfSections(in:), as shown in Figure 13.

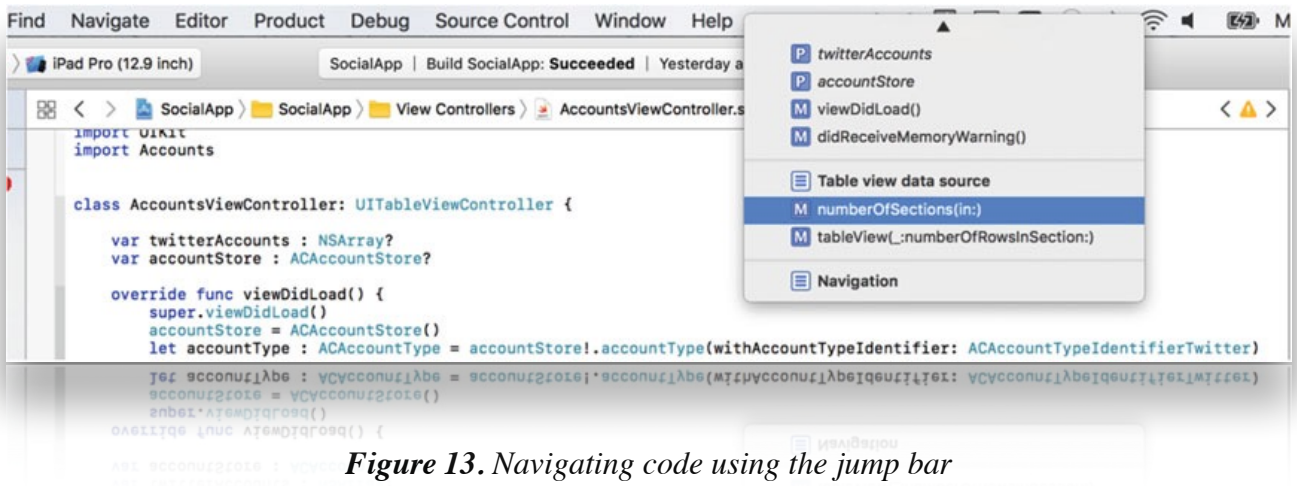

Figure 13. *Navigating code using the jump bar*

13. As previously mentioned, the numberOfSections(in:) method returns a value that determines how many sections appear in the table view. In this instance, you only want a single section, so change the return value to 1 as shown next:

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

14. You need to program the **tableView(_:numberOfRowsInSection:)** method to return the number of items held in the array holding the Twitter accounts configured on this device so that the application knows how many cells to create in the table view, this is done returning the value of cellCount. Because there may be no items in the array, and you don't want an exception, add the highlighted if statement as shown next:

```
override func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int
{
   if let cellCount = self.twitterAccounts?.count {
      return cellCount
   } else {

      return 0

   }

}
```

15. The application will try to create an instance of your prototype cell if there are any Twitter accounts set up in the Simulator, so it's important to set the correct reference name for the cell before trying to run the application. Look for the tableView(_:cellForRowAt:) method: it should be just below the last method you changed. It's commented out by default, so remove the /* from the start and the */ from the end of the method.

16. Within the method you have an object called cell, which is initialized and then before being returned; this object is a UITableViewCell. When initialized, it's passed a value that currently says "reuseIdentifier" but that needs to say "AccountCell", which is the name you gave the table cell when you configured it in the storyboard. Go ahead and change the highlighted value as shown next:

```
let cell = tableView.dequeueReusableCell(withIdentifier: "AccountCell", for: indexPath)
```

17. It's been a very long time coming, but you're at a point where you can run the application in the Simulator. Choose Product ➤ Run (+R) to launch the application.

The first thing you should see is a prompt for access to the device's Twitter accounts, as shown in Figure 14. It's important to click OK at this point to grant access; you're then presented with one of the two outcomes, depending on the number of Twitter accounts you have. If there are no Twitter accounts installed, you get the alert view warning you that no accounts were found; otherwise, you see a single row with an arrow in your table view.

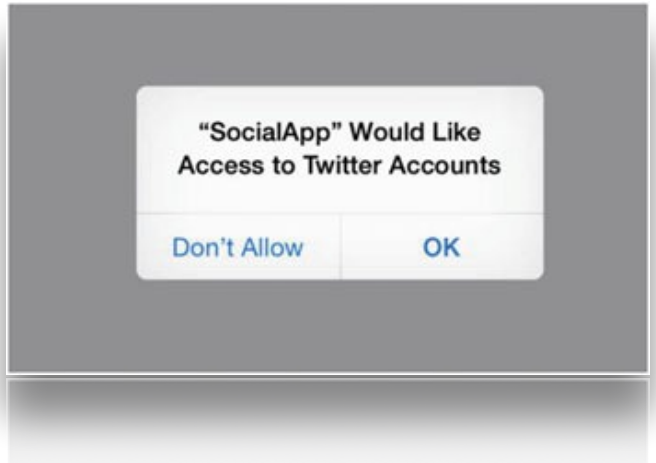

Figure 14. *The security prompt as king for access to the Twitter accounts for the application*

18. If you don't have any Twitter accounts added in iOS and you saw the alert, adding your Twitter account is very easy. In the Simulator, return to the home screen by choosing Hardware ➤ Home (+Shift+H). Navigate to the first page of icons, click the Settings icon, scroll down, and select the Twitter option from the left column, as shown in Figure 16.

Note If you accidentally refuse permission to the application, you can grant permission from the Twitter settings. SocialApp is listed at the bottom of the Twitter settings with a switch beside it, as shown in Figure 15. Turn this to the on position, and permission will be granted to access the accounts.

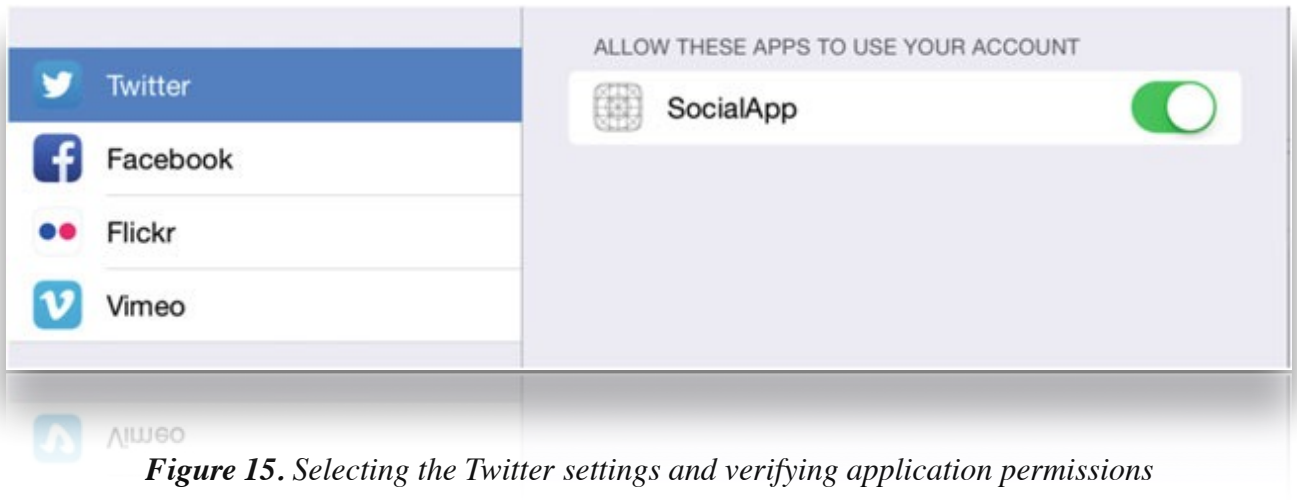

Figure 15. *Selecting the Twitter settings and verifying application permissions*

19. Type in your Twitter account name and password and click Sign In. You can repeat this step to add more Twitter accounts if you want. When you're done, go back to Xcode and rerun your application; this time your view should resemble Figure 16.

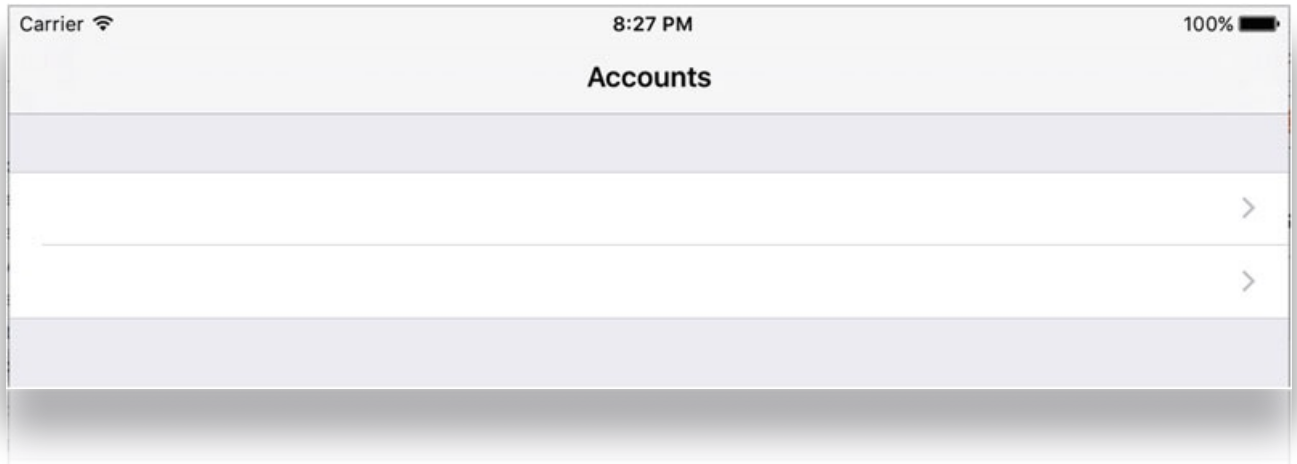

Figure 16. *The Accounts view with hopefully one or more rows that will show account details*

Although the application runs, you still have to make the table view display the account name in the cell (currently it's empty); and to do that, you need to add some code to the delegate method **tableView(_:cellForRowAt:)**. This is the method you quickly altered before running the application. All delegate methods are called as the result of an event occurring. In this case, it's the table view responding to the number of rows it has been told to display and then calling the **tableView(_:cellForRowAt:)** method to allow it to populate the specific cell's contents. To display the correct account information in the cell, the application needs to know which row it's on so that the corresponding entry can be fetched from the array. You can establish this by looking at the method variable **indexPath**. This object has properties indicating the cell's section and row numbers: both follow the array format for positioning, starting at 0 and incrementing by 1. Because there is only one section, the row value corresponds to the position of elements in the **twitterAccounts** array. For example, if the indexPath row property is 0, the application will fetch the account at position 0 in the array.

20. Remove the comment **// Configure the cell…** and in its place create an **ACAccount** object based on the account stored at the supplied position in the array:

```
if let account : ACAccount = self.twitterAccounts?[indexPath.row] as? ACAccount {
}
```

21. Put a value in the table view cell that shows the name of the account in the array. The table cell is currently the default **UITableViewCell**, which has three controls that can be manipulated: a label called **textLabel**, a subtitle label called **detailTextLabel**, and an image view called **imageView**. Let's take the **accountDescription** property of the account object and use it to set the **textLabel**'s **Text** property, as shown in the highlighted code:

```
if let account : ACAccount = self.twitterAccounts?[indexPath.row] as? ACAccount {
   cell.textLabel!.text = account.accountDescription
}
```

22. The complete method should look like this:

```
override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) ->

UITableViewCell {
    let cell = tableView.dequeueReusableCell(withIdentifier: "AccountCell", for: indexPath)
    if let account : ACAccount = self.twitterAccounts?[indexPath.row] as? ACAccount {
        cell.textLabel!.text = account.accountDescription
    }

    return cell

}
```

23. Run your application again; this time the table view should be populated as shown in Figure 17. What's more, if you select an account, you're taken to the FeedViewController scene.

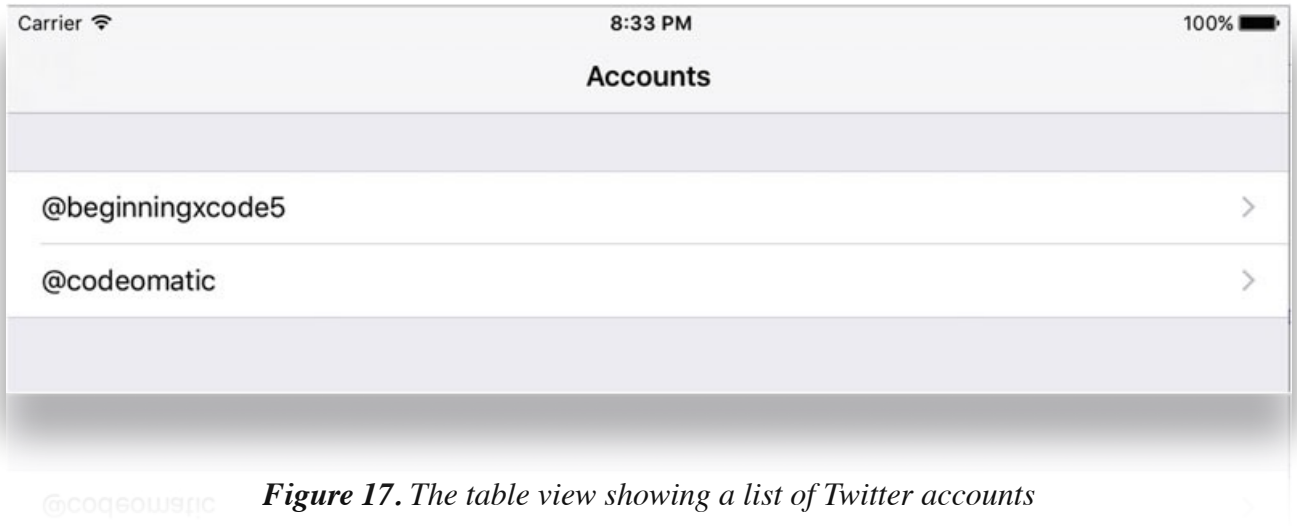

Figure 17. *The table view showing a list of Twitter accounts*

Caution Clicking the Compose button at this stage will cause the view appear and, while it looks great, you haven't yet written the code to dismiss the view controller and are stuck unless you rerun the application from Xcode.

One great feature to take note of here is that as you show the Feed view controller, you're automatically given a button that takes you back to the Accounts view controller without having to write any code.

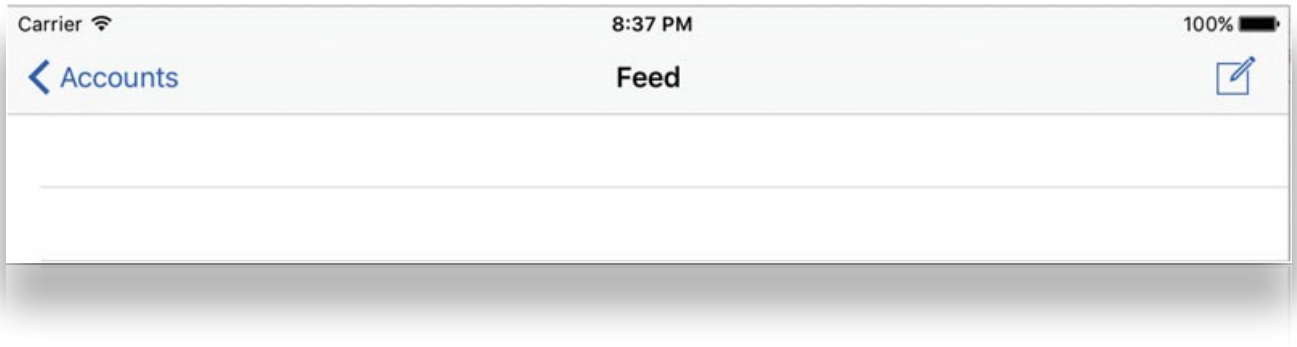

Figure 18. *The navigation controller allows you to navigate back to the Accounts view without adding any code*

Before you focus on the next scene—the Feed view controller—you need to think ahead slightly. When an account is selected, that selection made in the Accounts view controller needs to be passed on, a bit like a baton in a relay, to the Feed view controller when the segue is triggered. To do this, you need to create a custom initializer in the Feed view controller to receive the selected **ACAccount** object.

24. Open FeedViewController.swift from the Project Navigator. Because you need to refer to both the Accounts *and* the Social framework in this view controller, add the following two highlighted import statements after import UIKit:

```
import UIKit

import Accounts
import Social
```

25. Right after the class definition, type the following highlighted line:

```
class FeedViewController: UITableViewController {

  var selectedAccount : ACAccount!
```

You've now created a property in your **FeedViewController** class that can receive the baton, or in this case the selected **ACAccount**, from the Accounts view controller. All that remains is to pass the object across when the segue is called.

26. Open **AccountsViewController.swift** once more from the Project Navigator.

27. If you weren't using segues to navigate between view controllers, you would use the **didSelectRowAtIndexPath** method to determine what to do next; but because you're using a segue, you use the **prepareForSegue** method. Yet again, the folks at Apple have already written a basic implementation of the method for you, but it's currently commented out. At the bottom of the **AccountsViewController.swift** file, you should see the method you need,

commented out in green (assuming you haven't deviated from the default color scheme). Before the line // **MARK: - Navigation** are the characters /* that index the start of a commented block of code; remove them. Next, look for */ just before the last }; this signifies the end of the comment block. Remove it as well. The method is now uncommented and ready for use.

28. The **prepare(for:sender:)** method is called when a segue is about to be triggered. It gives you a chance to perform any actions that need to be processed before the view changes. One of the parameters passed to this method is a **UIStoryboardSegue** object called segue; you can use this to check the segue identifier and then take appropriate action. The identifier for this segue to the Feed view controller scene is **ShowTweets**. To check for this in an if statement, add the highlighted code to the method:

```
override func prepare(for segue: UIStoryboardSegue, sender: AnyObject?) {
   if(segue.identifier == "ShowTweets") {
   }

}
```

29. You need to find out which account was selected. Create an **ACAccount** object from the selection and pass that to the Feed view controller's selected account property. The first task is to determine which row was selected. You do this by creating an **NSIndexPath** object called path based on the result of the **indexPathForSelectedRow** property. Add the following highlighted line to your if statement:

```
if(segue.identifier == "ShowTweets")
{
   if let path : IndexPath = self.tableView.indexPathForSelectedRow {
   }
}
```

30. Create the ACAccount object, called account. This is almost an exact duplicate of when you instantiated an ACAccount object in the cellForRowAtIndexPath method, as the highlighted code shows:

```
if(segue.identifier == "ShowTweets") {
   if let path : IndexPath = self.tableView.indexPathForSelectedRow {
      if let account : ACAccount = self.twitterAccounts![path.row] as? ACAccount {
      }

   }

}
```

31. Pass the account object to the Feed view controller. When a segue is triggered and this method is called, the destination view controller is stored in a property of the segue object called **destinationViewController**. Because you know that the destination view controller is a Feed view controller, you cast **destinationViewController** from a generic **AnyObject** type to be a **FeedViewController** type. Once you've created a **FeedViewController** object, you simply take the account object and give it to **selectedAccount**. All of these actions are done in the following highlighted code:

```
if(segue.identifier == "ShowTweets") {
   if let path : IndexPath = self.tableView.indexPathForSelectedRow {
      if let account : ACAccount = self.twitterAccounts![path.row] as? ACAccount {
         let targetController = segue.destination as! FeedViewController
         targetController.selectedAccount = account
      }

   }

}
```

Now you've finished writing the code for the Accounts view controller and even added a bit to the Feed view controller. That's one down and five to go. Let's move on to the Feed view controller, where you build on your table view skills and learn about creating custom cells and subclassing **UITableViewCell** to take customizations to another level.

Configuring the Feed View

The Feed view is the center point of SocialApp; it lists the 20 latest tweets using some of the methods and classes used in the previous Accounts view, along with many that haven't been encountered in the app so far. In the sections that follow, you learn how to:

- Use the SLRequest class to fetch the JSON-formatted data from the Internet
- Use an NSCache object to handle some basic caching
- Use an NSOperationQueue to streamline the retrieval of Twitter avatar images
- Subclass UITableViewCell to create a custom class for the cells in the table view

Because we want to focus on the table view and how to populate it, we won't go into much detail about the code used for retrieving the data from the Twitter API or processing it.

Figure 19 shows what the rows in the finished table view will look like. Before we get into the code, you need to build the cell's interface and link it to a custom **UITableViewCell** class. Here

are the steps:

1. You subclass UITableViewCell the same way you did UITableViewController. Right-click the SocialApp group in the Project Navigator and choose New File (+N), as shown in Figure 20.

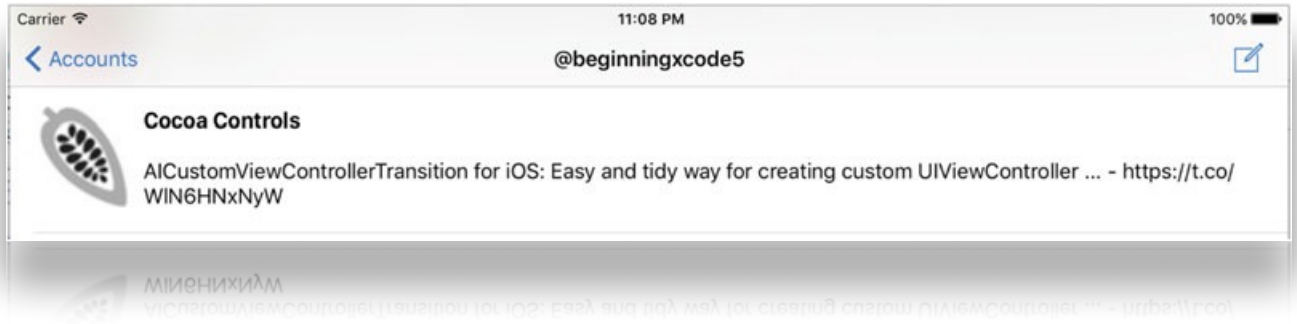

Figure 19. *A look a head at arow from the finished Feed view*

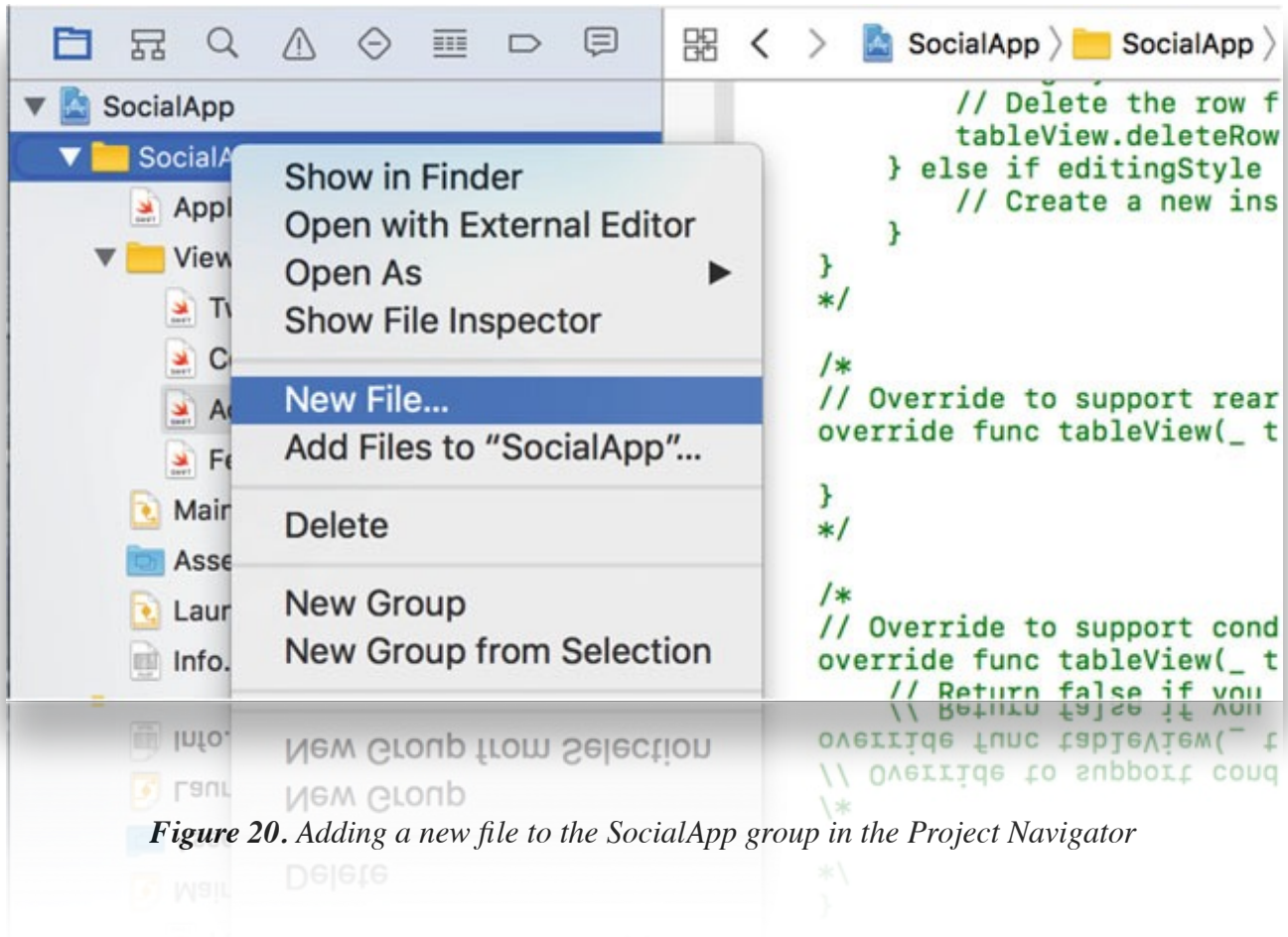

Figure 20. *Adding a new file to the SocialApp group in the Project Navigator*

2. Select the Cocoa Touch Class option, which should be selected by default, and click Next. Set the Subclass... value to **UITableViewCell** and the Class value to **TweetCell**. There is no need to create an XIB file, so leave that unchecked and click Next. As always, you want to save the file in the project folder. Click Create to create the file and add it to the project.

You're now ready to set up the visual elements of the table view.

3. Open **Main.storyboard** from the Project Navigator and arrange the view so you can see the Feed scene, as shown in Figure 21.

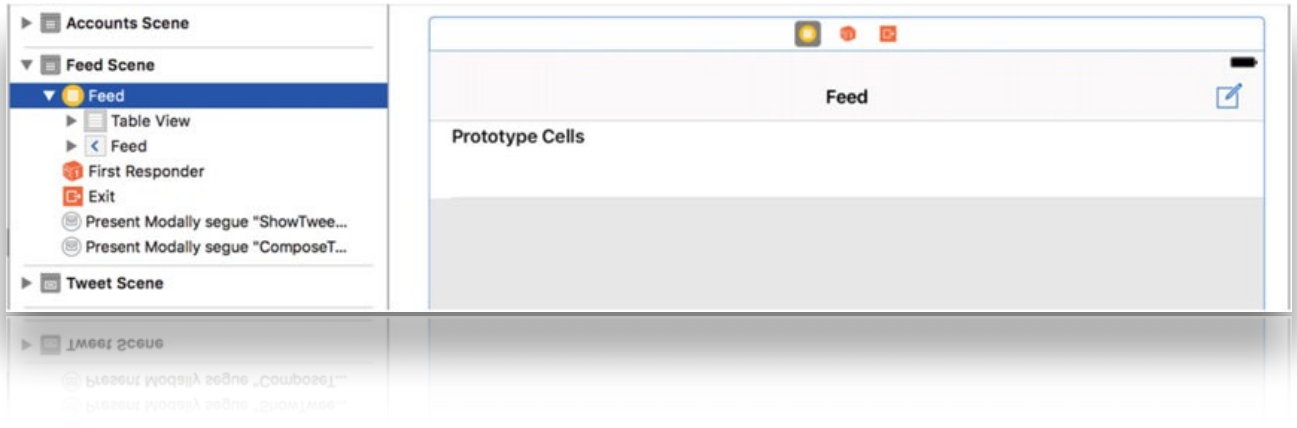

Figure 21. *The storyboard file, open and ready for you to build the interface*

4. The default row height of the cell is far too small to display everything nicely. To resize the cell, select the table view as you did in the previous scene by clicking the view where it says Table View Prototype Content or by selecting Table View from the Feed view controller scene in the Document Outline. Next, open the Size Inspector and set the Row Height value to 120, as shown in Figure 22.

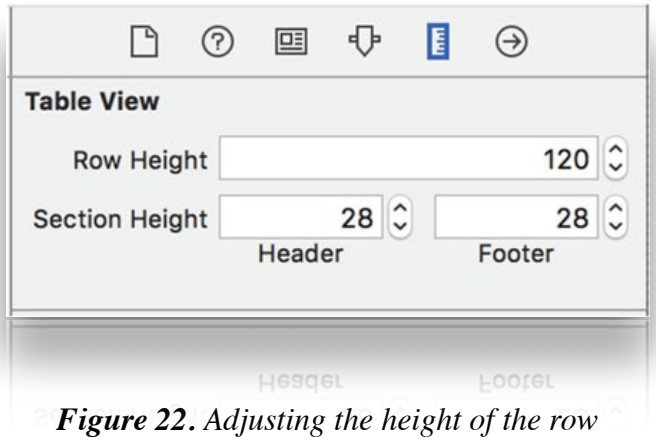

Figure 22. *Adjusting the height of the row in the Size Inspector*

5. Before you start creating the interface of the cell, you need to specify that the cell is controlled by the new TweetCell class. Select the cell, open the Identity Inspector, and, from the Class drop-down list, select TweetCell.

6. Open the Attributes Inspector for the cell. You need to specify a reuse identifier here, so in the Identifier attribute, type TweetCell.

7. Now that there is plenty of room to work, you can add the controls: an image view and two labels. Start by dragging in an image view from the Object Library onto the cell. It tries to fill the view, but don't worry; put it anywhere and go back the Size Inspector. Set the X and Y values to 20 and the Height and Width values to 79. The view should resemble that shown in Figure 23.

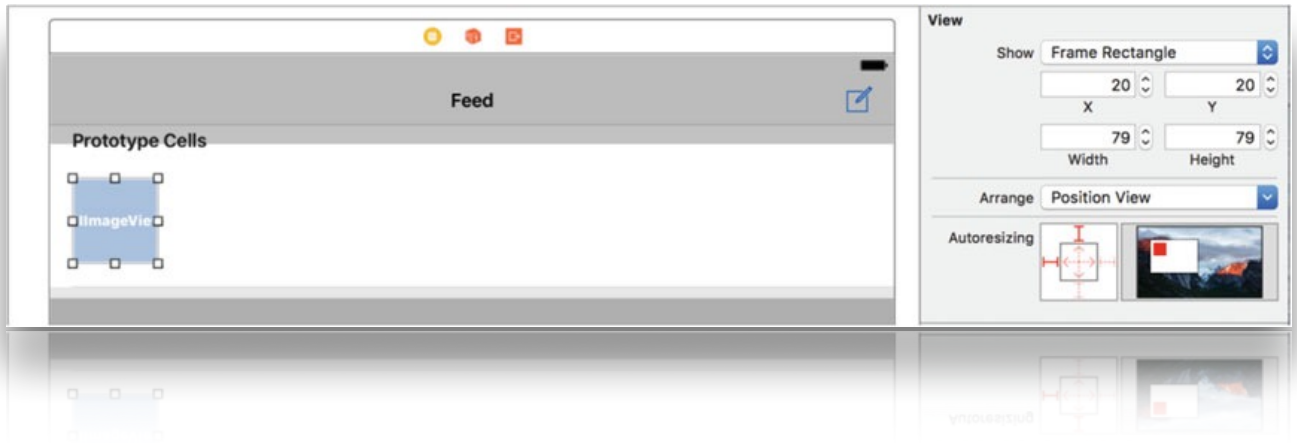

Figure 23. *Manually sizing and positioning the image view*

8. This image view will display the Twitter photo of the tweet's author. Before you leave the image view, you need to set a default image to act as a placeholder while the image is downloading from the Internet. Open **Assets.xcassets** from the Project Navigator: in the project files for this practice create a folder called images that must contain a file called camera.png. Drag that file from Finder to the Asset Catalog sidebar, as shown in Figure 24.

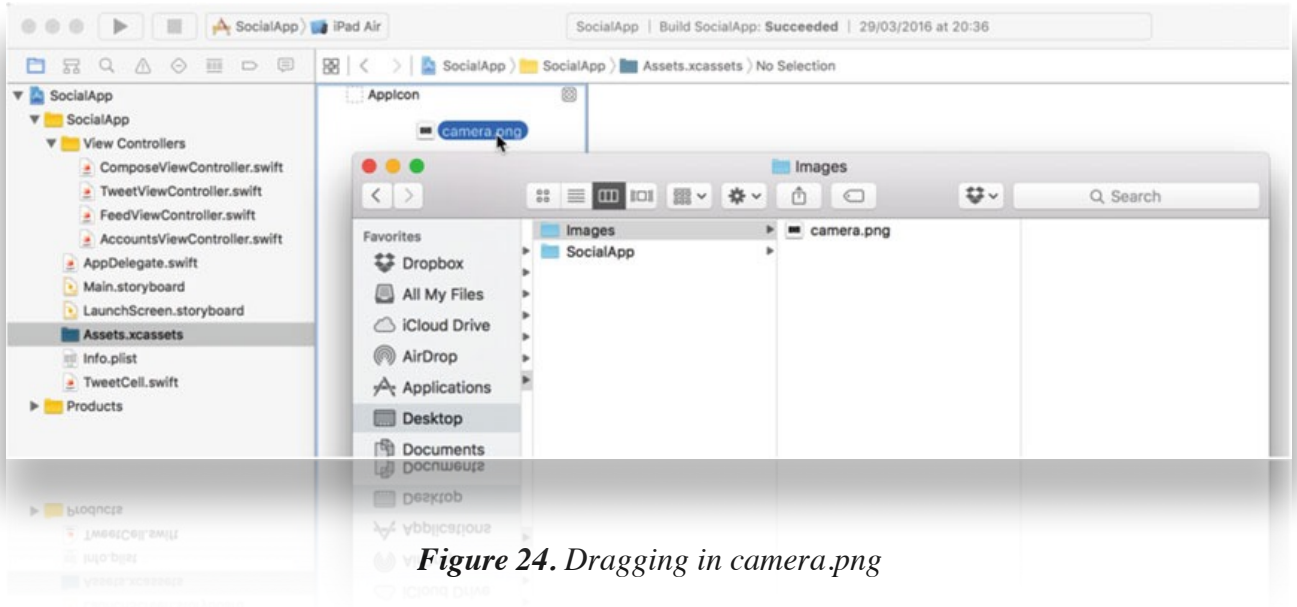

Figure 24. *Dragging in camera.png*

9. When you release the file, it automatically creates an image set named camera. Switch back to the storyboard, select the image view if it isn't already selected, and open the Attributes Inspector.

10. Set the Image attribute to camera, which should be shown in the list of available images. Your image view should now resemble Figure 25.

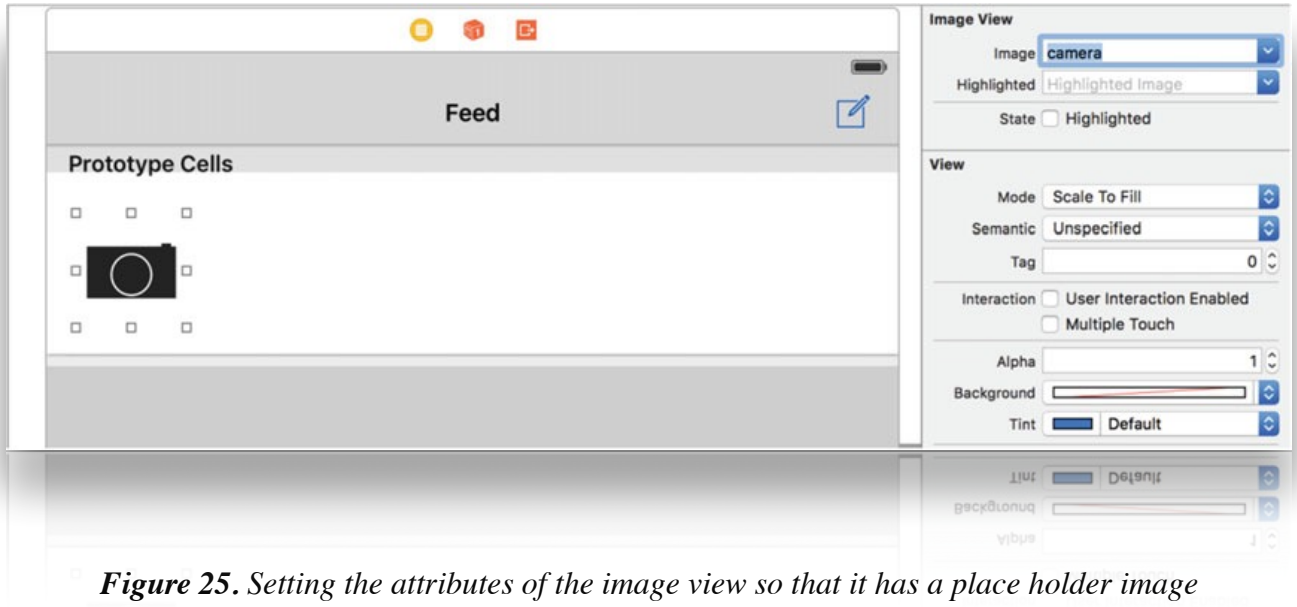

Figure 25. *Setting the attributes of the image view so that it has a place holder image*

11. Add a label for the username of the author of the tweet. To do so, drag in a label from the Object Library and align it with the top of the image view. In the Attributes Inspector, change the Text attribute to **User Name**. Using the arrow icon, change the Font attribute to System, Style to Bold and Size to 17.

12. With the font and the placeholder text set correctly, size the label appropriately. Keep it at a single-line height, but increase the width to the right until the blue margin appears. It should resemble Figure 26.

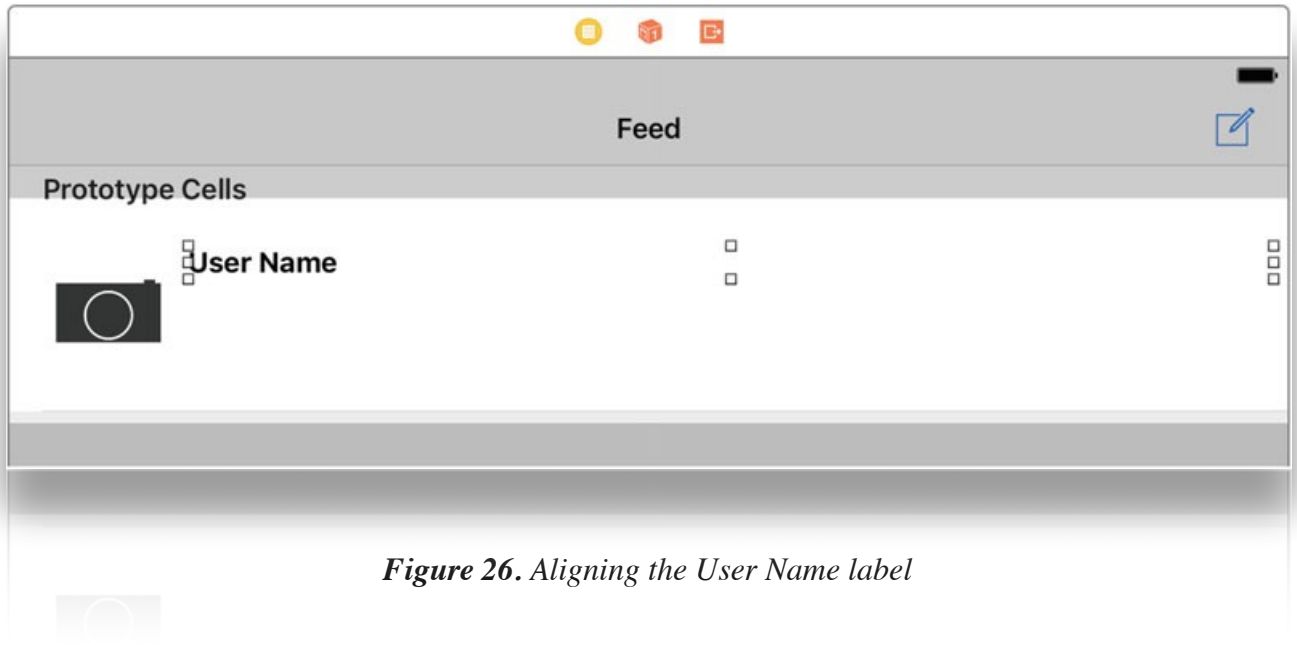

Figure 26. *Aligning the User Name label*

13. The final object that needs to be added to the cell is a label for the tweet content. Drag in a label and position it just below the User Name label. Make it the same width as the User Name label and then increase the height until the margin guidelines at the bottom of the cell appear. Change the default text to read **Content**. Your cell should resemble Figure 27.

Figure 27. *Aligning the tweet Content label*

14. In the Attributes Inspector, set the Lines attribute to 2. Xcode will wrap the text onto a second line if needed.

Note If the text length exceeds what will fit on two lines, the Line Breaks attribute will determine what will happen. The default option is Truncate Tail, which cuts the text short and appends an ellipsis (or ...) to the end of the text.

15. To align the interface elements, click the Resolve Auto Layout Issues button. Under All Views in TweetCell, choose Add Missing Constraints.

16. You've now built the interface for the custom cell. Next you need to create the outlets for the three objects. Open the Assistant Editor with the cell selected, and this time ensure that the code file that is loaded is **TweetCell.swift**; it's likely that the file loaded is actually **FeedViewController.swift**. Click the filename on the jump bar, as shown in Figure 28. and then choose **TweetCell.swift**.

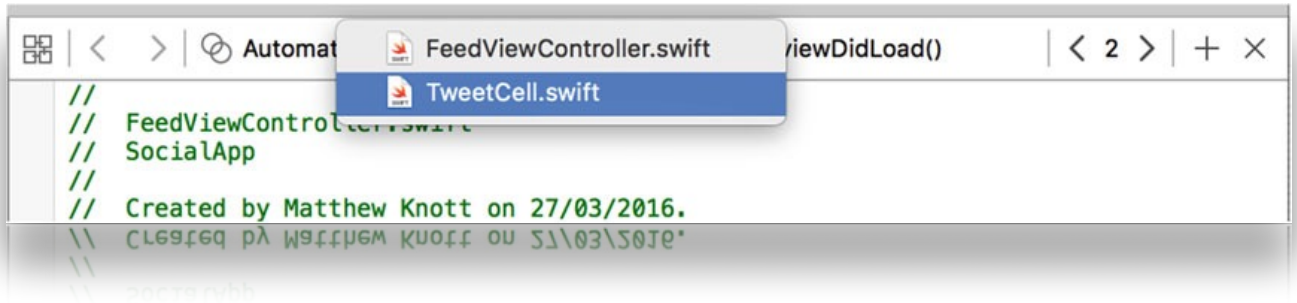

Figure 28. *Ensuring that TweetCell.swift is loaded before creating the outlet*

17. Control+drag a connection from the image view to just below class **TweetCell** and release the mouse. Name this outlet **tweetUserAvatar**, as shown in Figure 29.

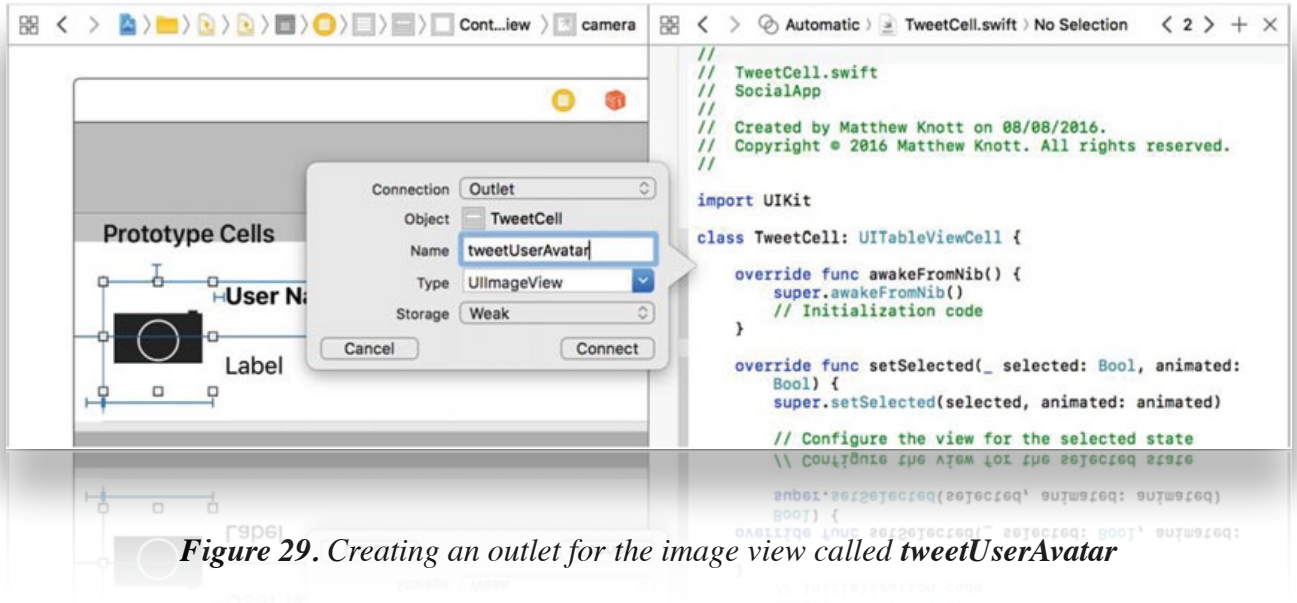

Figure 29. *Creating an outlet for the image view called* ***tweetUserAvatar***

18. Create an outlet in the same way for the **User Name** label, naming it **tweetUserName**. Finally, create an outlet for the Content label named **tweetContent**. The start of your custom **UITableViewCell** class should resemble the following code:

```
import UIKit

class TweetCell: UITableViewCell {
    @IBOutlet weak var tweetUserAvatar: UIImageView!
    @IBOutlet weak var tweetUserName: UILabel!
    @IBOutlet weak var tweetContent: UILabel!
```

Now that you've created the interface and the outlets for the objects in the cell's view, you can begin to bring all the different elements and classes together in the Feed view controller. Here you write the code that fetches the Twitter feed and then parses the returned data before displaying it in the custom table cell:

1. To get started, prepare the header file. Switch back to the Standard Editor and then open **FeedViewController.swift** from the Project Navigator. You need to create some instance variables, just as you did for **AccountsViewController**, but this time you're creating three. After **var selectedAccount : ACAccount!**, add the following highlighted code:

```
class FeedViewController: UITableViewController {
    var selectedAccount : ACAccount!
    var tweets : NSMutableArray?
    var imageCache : NSCache<AnyObject, AnyObject>?
```

Note You have a lot of code to get through for this view controller. As already mentioned, we won't be going into a huge amount of detail, but be assured that much of the code you're writing in this practice is reusable for any kind of application that fetches data from the Internet.

2. Scroll down to the **viewDidLoad** method. Because this method is called when the view loads, you perform a few key tasks here. First, clear out all the green commented lines of code so you're left with just **super.viewDidLoad()**.

3. You need to set the title of the view to the Twitter account name that was passed to the view from the previous Accounts view controller, initialize **OperationQueue** and configure its basic settings, and finally add a call to a function that hasn't been written yet called **retrieveTweets**. Add the highlighted code to your **viewDidLoad** method:

```
override func viewDidLoad() {
    super.viewDidLoad()
    self.navigationItem.title = selectedAccount.accountDescription
    retrieveTweets()
}
```

4. Xcode correctly flags the last line of the **viewDidLoad** method as being in error. This is because you haven't written that function yet. Drop down a few lines after the **viewDidLoad** method and declare the function stub as shown next:

```
func retrieveTweets() {

}
```

5. Xcode is happy that everything is back in order. You still need to write this method's substantial code. We'll go through each major block of code and explain the code. First you need to clear the **tweets** array to remove any previously stored tweets. Add the highlighted code to start the function:

```
func retrieveTweets() {
   tweets?.removeAllObjects()
}
```

6. You need to check that you do indeed have a valid **ACAccount** object. If so, you declare and initialize an **SLRequest** object with the URL to the Twitter API that provides the home timeline data you want to display in the table view. You then authenticate the request using the **selectedAccount ACAccount** object. Drop down a line and add this highlighted code:

```
func retrieveTweets() {
   tweets?.removeAllObjects()
   if let account = selectedAccount {
      let requestURL =
         URL(string: "https://api.twitter.com/1.1/statuses/home_timeline.json")
      if let request = SLRequest(forServiceType: SLServiceTypeTwitter,
         requestMethod: SLRequestMethod.GET,
         url: requestURL, parameters: [:]) {
         request.account = account
      }
   }

}
```

7. For the final block of this method, you've given the request object all the parameters it needs, and now you execute the **performRequestWithHandler** method. This method accesses the supplied URL and returns the response from the request to a code block. If the request is successful, it returns a status code of 200. When this happens, you parse the JSON code into an array and use that as the contents of the tweets array. Finally, you call the **reloadData** method of the table view to update the information shown on the screen. Add the highlighted code after the last line you wrote:

```
request.account = account
request.perform()
{
   responseData, urlResponse, error in
   if(urlResponse?.statusCode == 200) {
      if(urlResponse?.statusCode == 200) {
          do {
                self.tweets = try JSONSerialization.jsonObject(with: responseData!,
                options: JSONSerialization.ReadingOptions.mutableContainers) as?
                NSMutableArray
          } catch let error as NSError {
              print("json error: \(error.localizedDescription)")
          }
      }

   }

   DispatchQueue.main.async() {
      self.tableView.reloadData()
   }

}
```

Note If you aren't familiar with http response codes, it may be worth looking up the common codes online. Even if you've never heard the term before, you've almost certainly come across them while browsing the Internet. Errors 404 and 500 are two of the more visible error codes that you may have seen on a web site in the past, but there are many others, and it's worth doing some research on them if you intend to use web APIs to get data into your application.

8. The completed code for the retrieveTweets method should look like this:

```
func retrieveTweets() {
    tweets?.removeAllObjects()
    if let account = selectedAccount {
        let requestURL = URL(string: "https://api.twitter.com/1.1/statuses/home_timeline.json")
        if let request = SLRequest(forServiceType: SLServiceTypeTwitter,
            requestMethod: SLRequestMethod.GET, url: requestURL, parameters: [:]) {
            request.account = account
            request.perform()
            {
                responseData, urlResponse, error in
                if(urlResponse?.statusCode == 200) {
                    if(urlResponse?.statusCode == 200) {
                        do {
                            self.tweets = try JSONSerialization.jsonObject(with: responseData!,
                                options: JSONSerialization.ReadingOptions.mutableContainers)
                                as? NSMutableArray
                        } catch let error as NSError {
                                print("json error: \(error.localizedDescription)")
                        }
                    }
                }
                DispatchQueue.main.async() {
                    self.tableView.reloadData()
                }
            }
        }
    }
}
```

9. This is a good point at which to run your application to check for errors. The application should compile and allow you to select a Twitter account. On the feed screen, you don't see anything yet; but more importantly, you shouldn't see any errors. If you do, check things such as the correctness of the name of the segue in the storyboard and whether you typed the URL correctly.

10. Back in **FeedViewController.swift**, it's time to look at the table view delegate methods. Starting with **numberOfSections(in:)** and **tableView(_:numberOf RowsInSection:)**, you need to return 1 for the single section you want to have and the number of tweets in the array to set the number of rows in the table. The completed methods should look like this:

```
override func numberOfSectionsInTableView(tableView: UITableView) -> Int {
    return 1
}
override func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int
{
    // Return the number of rows in the section.
    if let tweetCount = self.tweets?.count {
        return tweetCount
    } else {
        return 0
    }
}
```

11. It's time to pair the data you've received and stored in the array with the custom **TweetCell** table cell using the **tableView(_:cellForRowAt:)** method. Delete the comments surrounding the method so you're left with just the stub and change the highlighted values:

```
override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) ->
UITableViewCell {
    let cell = tableView.dequeueReusableCell(withIdentifier: "TweetCell", for: indexPath)
            as! TweetCell
    // Configure the cell...
    return cell
}
```

12. You need to create two **NSDictionary** objects to store different parts of the Twitter feed data. One stores the main message data; the other stores the portion that relates directly to the user who created the tweet. An **NSDictionary** is a type of array that uses a key-value pairing system to store and access data. This means instead of asking for the value at position 0, as you would with an array, you ask for the value that corresponds to **"name."** Add the following highlighted code into your method:

let cell = tableView.dequeueReusableCell(withIdentifier: "TweetCell", for: indexPath) as!

TweetCell

let tweetData = tweets?.object(at: indexPath.row) as! NSDictionary
let userData = tweetData.object (forKey: "user") as! NSDictionary

return cell

13. Let's take data from those **NSDictionary** objects and populate the interface. Add the following highlighted code to set the values of the two labels and then return the cell object to stop the error from being reported in Xcode:

let tweetData = tweets?.object(at: indexPath.row) as! NSDictionary

let userData = tweetData.object(forKey: "user") as! NSDictionary

cell.tweetContent.text? = tweetData.object(forKey: "text") as! String

cell.tweetUserName.text? = userData.object(forKey: "name") as! String

return cell

14. Because you returned the **cell** object, you're now error free and can run the application. You haven't set the image yet, but you should be able to select your Twitter account and see the user and content values in each cell, as shown in Figure 30. If you get an exception when you go to the Feed view controller, check that you specified the correct reuse identifier on the cell in the storyboard as well as in the code.

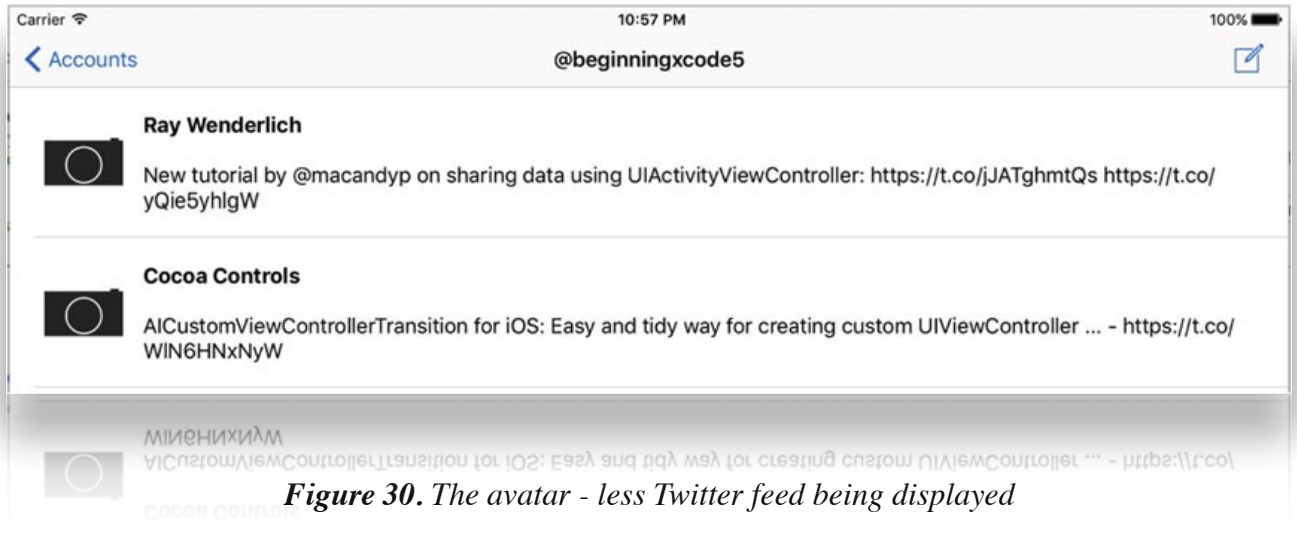

Figure 30. *The avatar - less Twitter feed being displayed*

15. I hope you have a huge sense of satisfaction at seeing your application finally come to life as it reads live data from the Internet. There is one final block of code for this method, which focuses on retrieving, caching, and displaying Twitter users' avatars. First you try to retrieve the image from the cache; if that fails, you create an operation for the **OperationQueue** queue object to fetch the image's data and create an image from it before displaying it and then caching it for future use. Add the following highlighted code:

```
cell.tweetUserName.text? = userData.object(forKey: "name") as! String
let operationQueue = OperationQueue.main
operationQueue.maxConcurrentOperationCount = 4
let imageURLString = userData.object(forKey: "profile_image_url_https") as! String
let image = imageCache?.object(forKey: imageURLString as AnyObject) as? UIImage
if let cachedImage = image {
   cell.tweetUserAvatar.image = cachedImage
} else {
   cell.tweetUserAvatar.image = UIImage(named: "camera.png")
   operationQueue.addOperation() {
      let imageURL = URL(string: imageURLString)
      do {
         if let imageData : Data = try Data(contentsOf: imageURL!) {
            let image = UIImage(data: imageData) as UIImage?
```

```
            if let downloadedImage = image {
                OperationQueue.main.addOperation() {
                    let cell = tableView.cellForRow(at: indexPath) as! TweetCell
                    cell.tweetUserAvatar.image = downloadedImage
                }
                self.imageCache?.setObject(downloadedImage, forKey: imageURLString as
                    AnyObject)
            }
        }
    } catch let error as NSError {
        print("parse error: \(error.localizedDescription)")
    }
  }
}
return cell
```

16. Finally, open the **info.plist** file from the Project Navigator. Hover over the first line **information property list** and click the plus symbol. In the row that appears, choose **App Transport Security Settings**. Next mouse over the row you just added and click the plus icon, this time select **Allow Arbitrary Loads** and change its value from **No** to **Yes**. Your **info.plist** should resemble Figure 31.

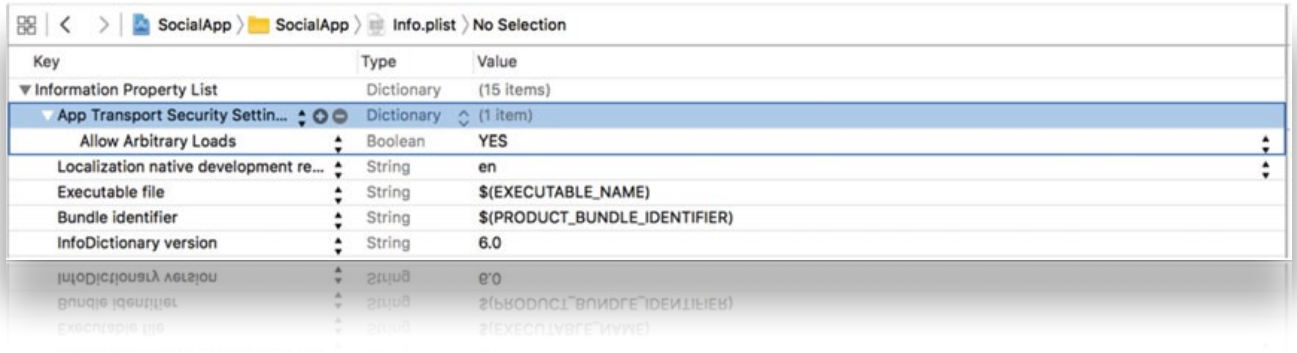

Figure 31. *Changing the info.plist settings for the application*

17. Take this opportunity to rerun the application. This time, after a brief delay, images should appear instead of the placeholder camera.png image, as shown in Figure 32.

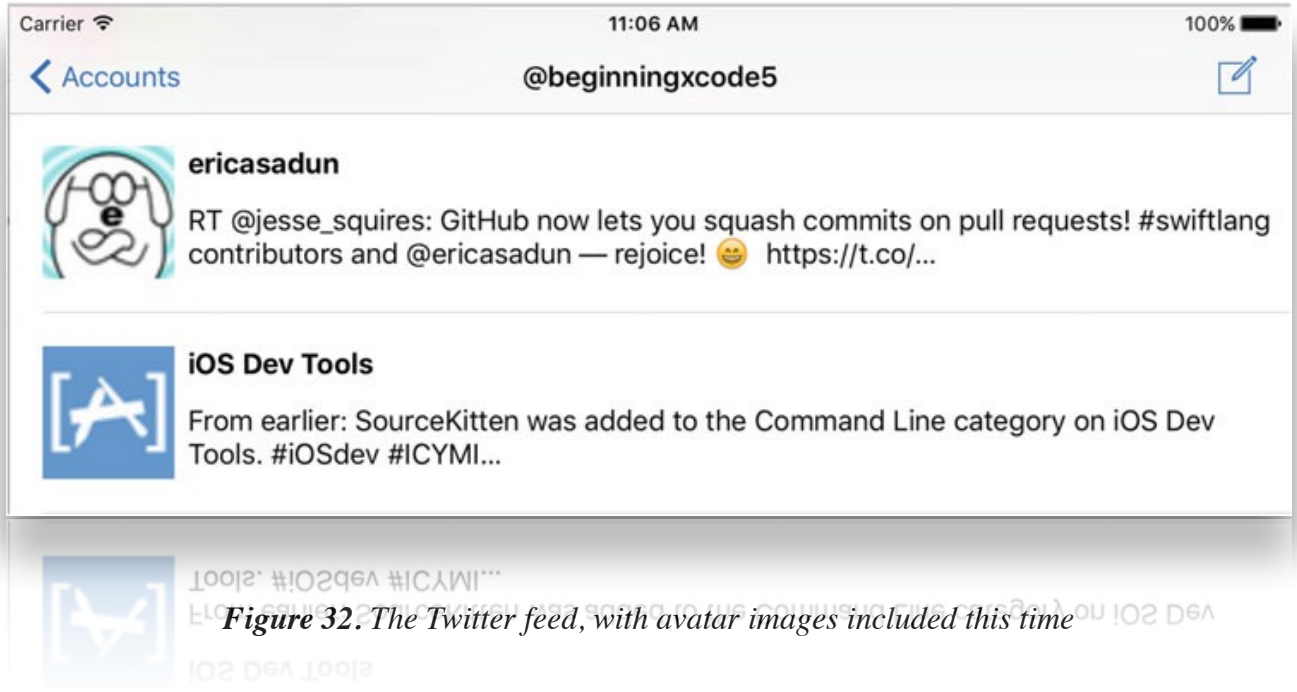

Figure 32. *The Twitter feed, with avatar images included this time*

Note There is always a delay when fetching data from the Internet. But because you're using the **OperationQueue** object and efficiently switched between the main and arbitrary threads, there is no slowdown in the application, which would have guaranteed you negative app store reviews. Notice how quickly you can scroll up and down the list of tweets, all because the **NSCache** stores them for later use.

Before moving on, here is the full code for the table**View(_:cellForRowAt:)** method:

```
override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) ->
    UITableViewCell {
    let cell = tableView.dequeueReusableCell(withIdentifier: "TweetCell", for: indexPath)
        as! TweetCell
    let tweetData = tweets?.object(at: indexPath.row) as! NSDictionary
    let userData = tweetData.object(forKey: "user") as! NSDictionary
    cell.tweetContent.text? = tweetData.object(forKey: "text") as! String
    cell.tweetUserName.text? = userData.object(forKey: "name") as! String
    let operationQueue = OperationQueue.main
    operationQueue.maxConcurrentOperationCount = 4
    let imageURLString = userData.object(forKey: "profile_image_url_https") as! String
```

```
    let image = imageCache?.object(forKey: imageURLString as AnyObject) as? UIImage
    if let cachedImage = image {
        cell.tweetUserAvatar.image = cachedImage
    } else {
        cell.tweetUserAvatar.image = UIImage(named: "camera.png")
        operationQueue.addOperation() {

            let imageURL = URL(string: imageURLString)
            do {
                if let imageData : Data = try Data(contentsOf: imageURL!) {
                    let image = UIImage(data: imageData) as UIImage?
                    if let downloadedImage = image {
                        OperationQueue.main.addOperation(){
                            let cell = tableView.cellForRow(at: indexPath) as! TweetCell
                            cell.tweetUserAvatar.image = downloadedImage
                        }
                        self.imageCache?.setObject(downloadedImage, forKey: imageURLString
                            as AnyObject)
                    }
                }
            } catch let error as NSError {
                print("parse error: \(error.localizedDescription)")
            }
        }
    }
    return cell
}
```

18. Next let's create the stubs for the two segues away from this view controller: **ComposeTweet** and **ShowTweet**. Scroll down to the bottom of the file and uncomment the **prepare(for:sender:)** method.

19. Handle the two possible segues by adding the following highlighted code:

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    if(segue.identifier == "ComposeTweet") {


    } else if(segue.identifier == "ShowTweet") {


    }
}
```

That's it for the Feed view controller for now. Once the other views have been configured, you'll come back to this file and add the code to pass the baton when the segue is triggered. For now, let's move on to the third view controller: Tweet view controller.

Because the final two views in the application subclass the **UIViewController**, in the following section we only focus on adding the code.

Configuring the Tweet View

The Tweet view controller allows users to see the full text of the tweet they selected from the Twitter feed, as well as any associated metadata. This is useful if the text in the table view cell has been truncated. If you were making a complete Twitter client, you would have to add numerous bits of additional information, such as how many times the tweet has become a favorite or been retweeted. For SocialApp, however, you configure the Feed view controller to pass across an **NSDictionary** object containing the data of the selected tweet; you then pick relevant information from that object to be displayed in the view.

First, as you did with the Feed view controller, let's configure the Tweet view controller to receive the **NSDictionary**. Because you're just pulling information from an **NSDictionary** object and not interacting with Twitter or the Internet, you don't need the Accounts or Social framework in this view controller. Follow these steps:

1. Open **TweetViewController.swift** from the Project Navigator. After the **@IBOutlets** and **@IBActions**, create an **NSDictionary** property called **selectedTweet** with this line of code:

var selectedTweet : NSDictionary?

This creates the global property that will be set when the view is initialized—or, to think of it another way, it's a runner on the track waiting to receive the **NSDictionary** baton from the runner or view controller before it. Now you need to go back to Feed view controller and pass across an **NSDictionary** of tweet data.

2. Back in **FeedViewController.swift**, scroll down to the **prepareForSegue** method. You've already created the if statement that checks for the **ShowTweet** segue, but in that if statement you need to determine the selected row's index, retrieve the relevant entry from the **tweets** array , and pass it to the Tweet view controller by setting its **selectedTweet** property. Following is the if statement and all the required highlighted code:

```
else if(segue.identifier == "ShowTweet") {
    if let path : IndexPath = self.tableView.indexPathForSelectedRow {
        if let tweetData = self.tweets?.object(at: path.row) {
            let targetController = segue.destination as! TweetViewController
            targetController.selectedTweet = tweetData as? NSDictionary
        }
    }
}
```

3. Now that the information is being passed across, it's easy to access the information you want to display. Open **TweetViewController.swift** again.

4. All the processing of data is done in the **viewDidLoad** method. You fetch the user's avatar directly from the Internet, rather than from the cache. You should be familiar with the rest of the code from the previous view controller. Add the highlighted code to the **viewDidLoad** method:

```
override func viewDidLoad() {
    super.viewDidLoad()
    let userData = selectedTweet?.object(forKey: "user") as! NSDictionary
    tweetText.text? = selectedTweet?.object(forKey: "text") as! String
    tweetAuthorName.text? = userData.object(forKey: "name") as! String
```

```
    let imageURLString = userData.object(forKey: "profile_image_url_https") as! String

    let imageURL = URL(string: imageURLString)

    if let imageData = NSData(contentsOf: imageURL!) {

        DispatchQueue.main.async {

            self.tweetAuthorAvatar.image = UIImage(data: imageData as Data)

        }

    }

}
```

5. To make the tweet close with the **Cancel** button, you implement the **dismissView** method, which uses the **UIViewController** method of dismiss. Because it's a **UIViewController** method and this is the implementation file for a class that subclasses **UIViewController**, you access your base class methods by using self, although it isn't always necessary with Swift. Add the highlighted line of code to your action:

```
@IBAction func dismissView(sender: AnyObject) {

    self.dismiss(animated: true, completion: nil)

}
```

Now run your application. Select a tweet from the feed, and the tweet should be expanded in the modal dialog. For the first time, you can see the form sheet presentation style in effect, as shown in Figure 33. In the 3rd part of the practice we will move on to the final view controller in this application: the Compose view controller.

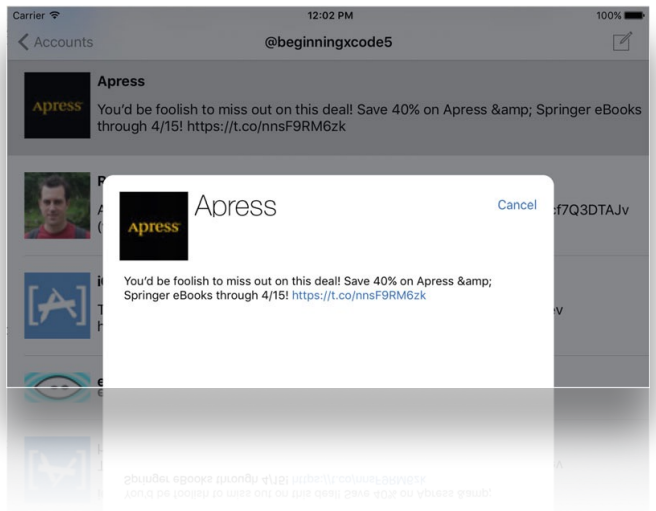

Figure 33. *The Tweet view controller showing the data that was passed to it along with an image pulled directly from the Internet. Notice the link detection in effect.*