SocialApp

Practice

PART 3

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 the 2nd part of the practice we focuses on the table 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 have seen how the segue identifiers specified in 1st part of the practice allow you to share data between view controllers, and you learnt about a variety of ways to obtain data from the Internet and display that in an application.

In this part of the practice we focuses on the collection view.

Now let's move on to the final view controller in this application: the Compose view controller.

Configuring the Compose View

The final view for this application is the Compose view controller. This is where the user can compose a message and post it to Twitter. Let's enforce the 140-character limit for tweets by using a UITextView delegate method and then animate the activity indicator when it's sending the tweet data to Twitter. Here are the steps:

1. You've created the visual element and set up the outlets, so open ComposeViewController.swift. As previously mentioned, you're using a UITextView delegate method, so the first thing you need to do is implement the UITextViewDelegate protocol. Add the highlighted code to the class line so that it looks like this:

class ComposeViewController: UIViewController**, UITextViewDelegate** {

2. You need the Social and Accounts frameworks for this view controller, so add their import statements beneath the line that says import UIKit:

import UIKit
import Accounts
import Social

3. To create a property to receive the ACAccount object for the selected account from the Feed view controller, after the class line, add the following highlighted code:

```
class ComposeViewController: UIViewController, UITextViewDelegate {
   var selectedAccount : ACAccount!
```

4. Go back to FeedViewController.swift to pass the selected account details over to your newly created property.

5. Scroll down until you see the prepareForSegue method. You have an empty if statement set up for the ComposeTweet segue; modify it so that it passes the selectedAccount object to the Compose view controller, as shown next:

```
if(segue.identifier == "ComposeTweet")
{
   let targetController = segue.destination as! ComposeViewController
   targetController.selectedAccount = selectedAccount
}
```

6. Switch back to ComposeViewController.swift.

7. You need to create a custom function and a delegate method as well as two actions in this view controller. The good news is that none of them require a great deal of code. Scroll down until you see the viewDidLoad method. All you need to do here is specify the delegate property of the text view, which as we've mentioned previously is this view controller, so it's set to self. Although you can add this in the storyboard, let's do it here for the sake of variety. After the *super.viewDidLoad()* line, add the following code:

```
override func viewDidLoad() {
   super.viewDidLoad()
   self.tweetContent.delegate = self
}
```

8. You need to create a function called *postContent* to handle the transmission of the tweet text to the chosen Twitter account. Create the function stub just below the *viewDidLoad* method:

```
func postContent(post : String) {
}
```

9. As you can see, the function takes one parameter, a String object called post. The first thing you want to do when this function is called is to start animating the *postActivity* activity indicator. You do this by sending the *startAnimating* message. Due to the way the activity indicator was configured in part 1 of the practice, it's at this point that it appears and begins its animation:

```
func postContent(post : String){
   postActivity.startAnimating()
}
```

10. You're ready to prepare the Twitter request. Just as you did in the Feed view controller, let's create an *NSURL* object with the appropriate API URL for the type of request you want to make, which is used when instantiating the *SLRequest* object. A key difference this time is that you use a different HTTP method. You no longer issue a get request but instead issue a post request. When a post request is made with the SLRequest object, you supply the required parameters in an *NSDictionary* object. If you refer to the Twitter documentation for the status update API at https://dev.twitter.com/docs/api/1.1/post/statuses/ update, you see that the only required parameter is called *status*. The *status* parameter should be the textual content of the status update, which is the contents of the post parameter the function is supplied with. Drop down a line in the function and add the following code:

```
func postContent(post : String){
   postActivity.startAnimating()
   if let account = selectedAccount {
      let requestURL = URL(string: "https://api.twitter.com/1.1/statuses/update.json")
      if let request = SLRequest(forServiceType: SLServiceTypeTwitter,
                        requestMethod: SLRequestMethod.POST,
                        url: requestURL,
                        parameters: NSDictionary(object: post, forKey: "status" as
                        NSString) as [NSObject : AnyObject]) {
         request.account = account
      }
   }
}
```

11. Access the perform method of the *SLRequest* object just as you did in the *Feed view controller*. This time, however, when you receive a successful response code, you want to stop animating the activity indicator and dismiss the view controller. Drop down a line and add the following highlighted code to complete the method:

```
request.account = account
request.perform() {
   responseData, urlResponse, error in
   if(urlResponse?.statusCode == 200) {
      print("Status Posted")
      DispatchQueue.main.async
      {
         self.postActivity.stopAnimating()
         self.dismiss(animated: true, completion: nil)
      }
   }
}
```

```
func postContent(post : String) {
   postActivity.startAnimating()
```

```
    if let account = selectedAccount {
        let requestURL = URL(string: "https://api.twitter.com/1.1/statuses/update.json")
        if let request = SLRequest(forServiceType: SLServiceTypeTwitter,
                        requestMethod: SLRequestMethod.POST,
                        url: requestURL,
                        parameters: NSDictionary(object: post, forKey: "status") as
                        [NSObject : AnyObject]) {
            request.account = account
            request.perform()
            {
                responseData, urlResponse, error in
                if(urlResponse?.statusCode == 200) {
                    print("Status Posted")
                    DispatchQueue.main.async
                    {
                        self.postActivity.stopAnimating()
                        self.dismiss(animated: true, completion: nil)
                    }
                }
            }
        }
    }
}
```

12. It's time to address the two action methods: dismissView and postToTwitter. These are both one-liners; dismissView is a duplicate of the method used in the Tweet view controller, and postToTwitter simply calls the postContent method you just finished writing. Implement them both as follows:

```
@IBAction func dismissView(_ sender: AnyObject) {
    dismiss(animated: true, completion: nil)
}

@IBAction func postToTwitter(_ sender: AnyObject) {
    postContent(post: self.tweetContent.text)
}
```

13. You need to implement a *UITextView* delegate method that restricts the text view's content to 140 characters. This is done by using the *textView(_:should ChangeTextIn:)* method, which is called every time a character is typed. The method checks that the text view's content isn't greater than 140 characters and that it won't exceed 140 characters if someone pastes in some text. If the content is too large, the method returns false, and no more text can be typed. Add the following method just below *viewDidLoad*:

```
func textView(_ textView: UITextView,
        shouldChangeTextIn range: NSRange,
        replacementText text: String) -> Bool {
```

```
    let targetlength : Int = 140
    return textView.text.characters.count <= targetlength
}
```

That was the last line of code for this application. Run it. You should be able to successfully access your Twitter accounts, view the Twitter feed, see a tweet in detail, and even post your own.

Importantly, in this practice, you've learned all about configuring table views and the different methods and properties of the UITableView class, which you will use in your own applications.

Discovering the Collection View

A collection view is a great class that Apple introduced in iOS 6 and is relatively new (compared to most other objects, which have existed since the first version). Collection views offer developers a flexible way to display large amounts of data just like their cousin the table view, with the difference being that you can display data in columns as well as rows. Another neat feature is that collection views can scroll either vertically or horizontally, giving you that extra dimension as a developer.

Although structurally they're quite similar, one of the largest differences between the collection view and the table view is that the collection view's layout is completely separate from the view. It can be set to either a default or a custom *UICollectionViewLayout*, giving you a massive amount of flexibility over your design.

To demonstrate the implementation and configuration of a collection view, let's make some changes to SocialApp. First, let's turn it into a tabbed application, and then look at storing user preferences to automate account selection.

Embedding a Tab Bar Controller

The change we're aiming for here is to have a *Feed tab* and a *Following tab* in the application, with the *Feed tab* obviously being the *Feed view controller*. Let's create the Following view controller, which is a collection view controller that shows the avatars of all the users that the selected Twitter account follows. To turn SocialApp into a tabbed application, you need to add a tab bar controller between the *Accounts view* and the *Feed view*:

1. Open *Main.storyboard* from the *Project Navigator*. Navigate around the storyboard until you're able to see the segue connection from the *Accounts* scene to the *Feed* scene. Highlight the segue and delete it, as shown in Figure 34.

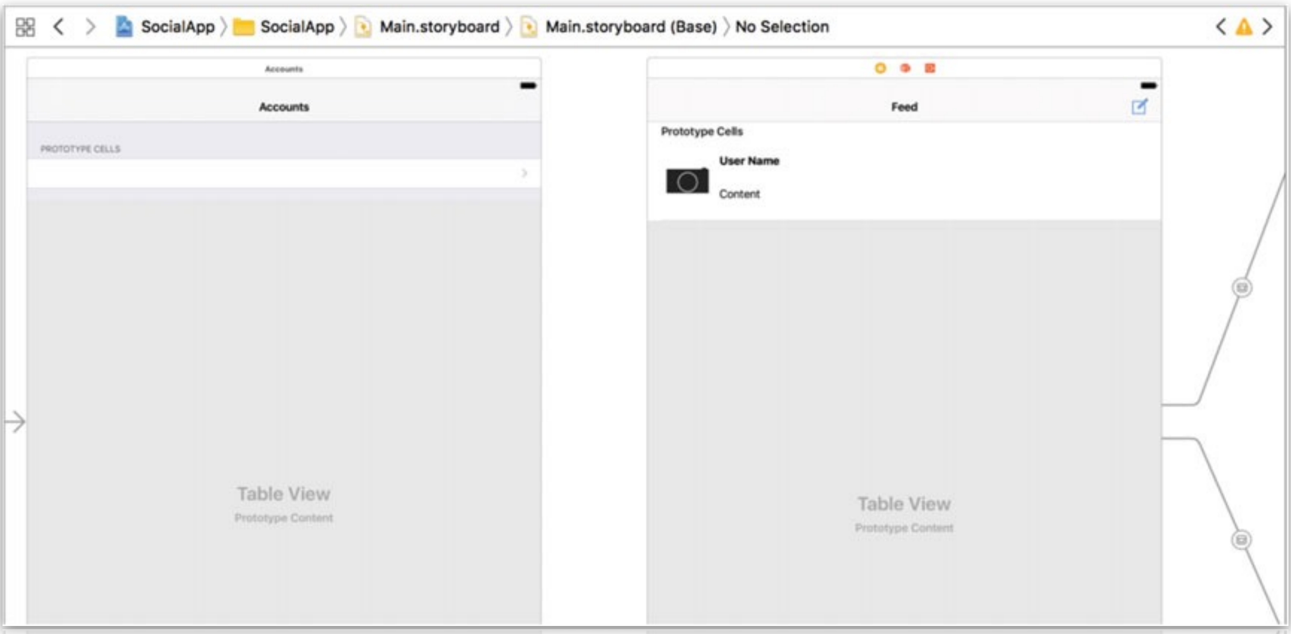


Figure 34. *The SocialApp storyboard with the ShowTweets segue removed*

2. Add a tab bar controller, with the Feed scene as one of the tabs. You could drag in a tab bar controller and manually link it up, but instead let's allow Xcode do the hard work for you. Select the *Feed view controller* either by clicking it directly in the storyboard or by selecting *Feed* from beneath *Feed Scene* in the *Document Outline*.

3. From the menu bar, choose *Editor* ➤ *Embed In* ➤ *Tab Bar Controller*. This adds a tab bar controller to your storyboard, sets the *Feed scene* as the first tab, and arranges the views to suit your needs.

4. You need to recreate the *ShowTweets* segue, but this time you're doing things differently by linking from the *Accounts view controller* rather than the table cell to the tab bar controller. This is called a manual segue because it isn't tied directly to an object that can be triggered by the user. Instead, the segue is triggered programmatically, because it's not possible to segue from a table cell to a tab bar controller and then on to the default view controller. Zoom out and Control+drag a connection from the yellow Accounts icon to the tab bar controller, as shown in Figure 35.

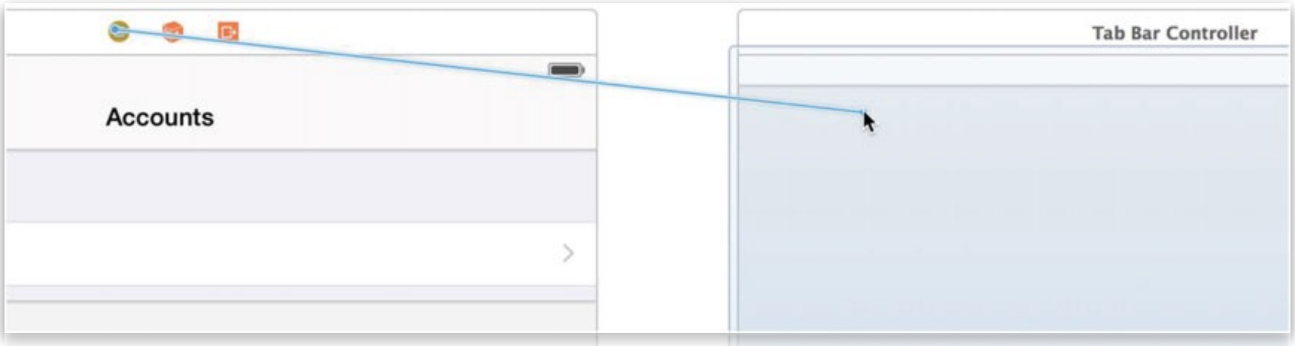


Figure 35. *Connecting the Accounts view controller to the tab bar controller with a manual segue*

5. When you release the mouse button, select Show as the segue type.

6. Select the newly created segue and open the *Attributes Inspector*. Set Identifier back to *ShowTweets*. You're finished with the storyboard for now, but before you add the *Collection view controller*, let's get the application back to a working condition. This involves executing a manual segue and storing the user's account selection so that whenever the application runs in the future, it will default to the account that the user selected and go straight to the feed.

Persisting User Preferences with UserDefaults

In a real-world scenario, a user picking up *SocialApp* would find it slightly annoying to have to select their account every time the application runs. Fortunately, iOS gives you a number of ways to persist user preferences, including iCloud and Core Data. In this instance, you're using the *UserDefaults* class.

UserDefaults allows the app to store values or certain types of objects against a textual key and can't be accessed from other applications. Whenever the application is closed and rerun, the preferences stored in *UserDefaults* are preserved, but the user can access and change the saved preferences to their heart's content. The *UserDefaults* class has methods that make it easy to both store and access a range of common types such as *Booleans*, *floats*, *integers*, *doubles*, and *URLs*, and it also supports the storage of the following objects:

- Data
- String
- Number
- Date
- Array
- Dictionary

The object you want to store here is an *ACAccount*, so you have to convert it to a Data object. Follow these steps:

1. Open *AccountsViewController.swift* from the *Project Navigator*. You need to create a *UserDefaults* instance variable to allow you to access the preferences from different

methods without having to instantiate the method each time. After the line *var accountStore : ACAccountStore?*, drop down a line and add the highlighted code:

```
import UIKit
import Accounts
class AccountsViewController: UITableViewController {
var twitterAccounts : NSArray?
var accountStore : ACAccountStore?
var userDefaults : UserDefaults?
```

2. Scroll down to the **viewDidLoad** method. The first thing you need to do when the view loads is to initialize the **userDefaults** object and then determine whether a preference called **selectedAccount** has already been saved; if so, you execute the manual segue with the perform method and go straight to the *Feed view controller*. After the line **accountStore = ACAccountStore()**, add the highlighted code:

```
override func viewDidLoad() {
   super.viewDidLoad()
   accountStore = ACAccountStore()
   userDefaults = UserDefaults.standard
   if (userDefaults?.object(forKey: "selectedAccount") != nil) {
      performSegue(withIdentifier: "ShowTweets", sender: self)
   }

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

3. That's it for the **viewDidLoad** method. It's time to address what happens when you tap on a cell. In the past, the segue from the cell was triggered, and the prepare method then passed the selected account across to the feed. This time, however, the application is going to save the selection before moving away. To do this, you use another commonly used **UITableView** method called **didSelectRowAt**, which is triggered every time a table cell is selected. Before you implement this method, delete the entire **prepareForSegue** method; there is no longer a segue associated with the cell, and the method won't be needed from here on out.

4. To create the **didSelectRowAt** method stub, after the **cellForRowAt** method, type the following highlighted code:

```
   return cell
 }
override func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {

}
```

5. In this method, you use the indexPath object to allocate an **ACAccount** object from **twitterAccounts** array based on the selected cell's index. Add the following highlighted code to the **didSelectRowAt** method:

```
override func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {
   if let account : ACAccount = self.twitterAccounts![indexPath.row] as? ACAccount {
   }
```

6. As mentioned previously, you need to convert the account object into something that can be stored in the **NSUserPreferences**; in this case, it will be converted to an **NSData** object using the **NSKeyedArchiver** class. After the previous line, add the following highlighted code:

```
override func tableView(tableView: UITableView, didSelectRowAtIndexPath indexPath:
NSIndexPath) {
   if let account : ACAccount = self.twitterAccounts![indexPath.row] as? ACAccount {
      let accountData = NSKeyedArchiver.archivedData(withRootObject: account) as Data
```

7. The selected account is in a compatible format, so it can be saved to the **NSUserPreferences** instance, **userDefaults**. The process for saving a preference comes in two parts: first, use the **setObject: forKey:** method, which associates the **accountData** object with a key; then, call the **synchronize** method, which saves the preference to the system. Add the following code to the method:

```
let accountData = NSKeyedArchiver.archivedData(withRootObject: account) as Data
userDefaults?.set(accountData, forKey: "selectedAccount")
userDefaults?.synchronize()
```

8. Now that you've saved the user's selection, you can manually trigger the segue. Just as you did in the **viewDidLoad** method, you need to call the **performSegueWithIdentifier** method. Add this code to complete the method:

```
override func tableView(tableView: UITableView, didSelectRowAtIndexPath indexPath:
NSIndexPath) {
   if let account : ACAccount = self.twitterAccounts!.object(at: indexPath.row) as? ACAccount {
      let accountData = NSKeyedArchiver.archivedData(withRootObject: account) as Data
      userDefaults?.set(accountData, forKey: "selectedAccount") userDefaults?.synchronize()

      performSegue(withIdentifier: "ShowTweets", sender: self)
   }
}
```

That's it for the Accounts view controller. You've removed the previous mechanisms for selecting an account, considered the overall user experience, and replaced those mechanisms with something that will be much more user friendly.

Now that you've saved the user's selection, you need to implement the retrieval of that selection when the application moves to the Feed view controller:

1. Open **FeedViewController.swift** and scroll down to the **viewDidLoad** method.

2. When the view loads, you create an instance of **NSUserDefaults**; there is no point in creating an instance variable, because this is the only time you need to access it in this file. You then retrieve the selected account from the preferences using the **object(forKey:)** method, which retrieves the object associated with the key that is supplied: in this case, **selectedAccount**. Finally, you reverse the conversion process with the **NSKeyedUnarchiver** class, which allows the conversion of the Data object back into an **ACAccount** object. After the line **super.viewDidLoad()**, add the following highlighted code:

```
override func viewDidLoad() {
   super.viewDidLoad()
   let userDefaults = UserDefaults.standard
   let accountData = userDefaults.object(forKey: "selectedAccount") as! Data
   selectedAccount = NSKeyedUnarchiver.unarchiveObject(with: accountData) as!
ACAccount
   self.navigationItem.title = selectedAccount.accountDescription
   retrieveTweets()
}
```

3. Run the application in the Simulator. Select an account, and you segue across to the newly tabbed Feed view. Quit the Simulator and then rerun the application. If everything has been done correctly, you should start facing the **Feed view** instead of the **Accounts view**. As a user, this is a much more favorable situation to be in. There is, however, one small issue you need to address: after embedding a tab bar controller, the table view positioning changed, and now the first row renders underneath the navigation bar and the title has vanished, as shown in Figure 36. This is far from ideal, and unfortunately Xcode doesn't give you an easy way to fix this; it has to be done in code.

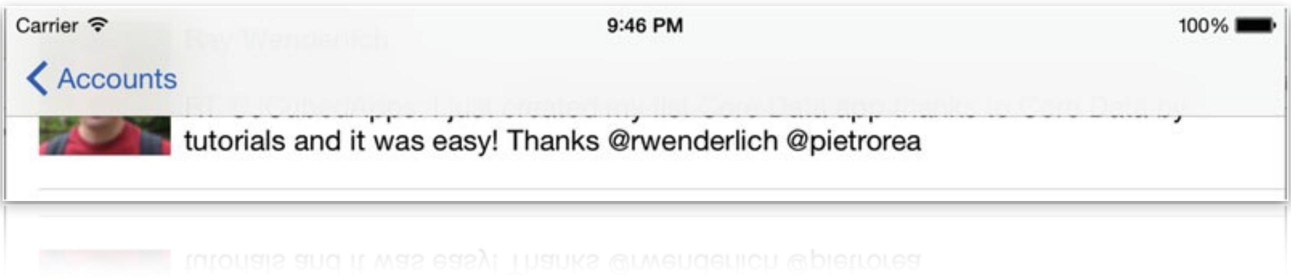


***Figure 36.** The first table row obscured by the navigation bar*

4. The title no longer applies itself because when you embedded the tab bar controller, you effectively inserted another level between the view and the navigation bar controller.

Ensure that you still have **FeedViewController.swift** open. Then, in the **viewDidLoad** method, add the highlighted code below:

```
self.tabBarController?.navigationItem.title = selectedAccount.accountDescription
```

5. Drop down another line and add the following highlighted code to resolve the positioning issue:

```
self.tabBarController?.navigationItem.title = selectedAccount.accountDescription
self.tabBarController?.edgesForExtendedLayout = []
```

6. Rerun the application; it should function perfectly. You've successfully implemented a user preferences system that will make life much easier for your users. You're now ready to start creating the **Collection view controller**.

Adding a Collection View Controller

You've successfully pulled apart and reassembled your application. It's time to turn your attention back to the storyboard and, in particular, to add a Collection view controller to the application:

1. Open Main.storyboard from the Project Navigator and position the scenes as shown in Figure 37.

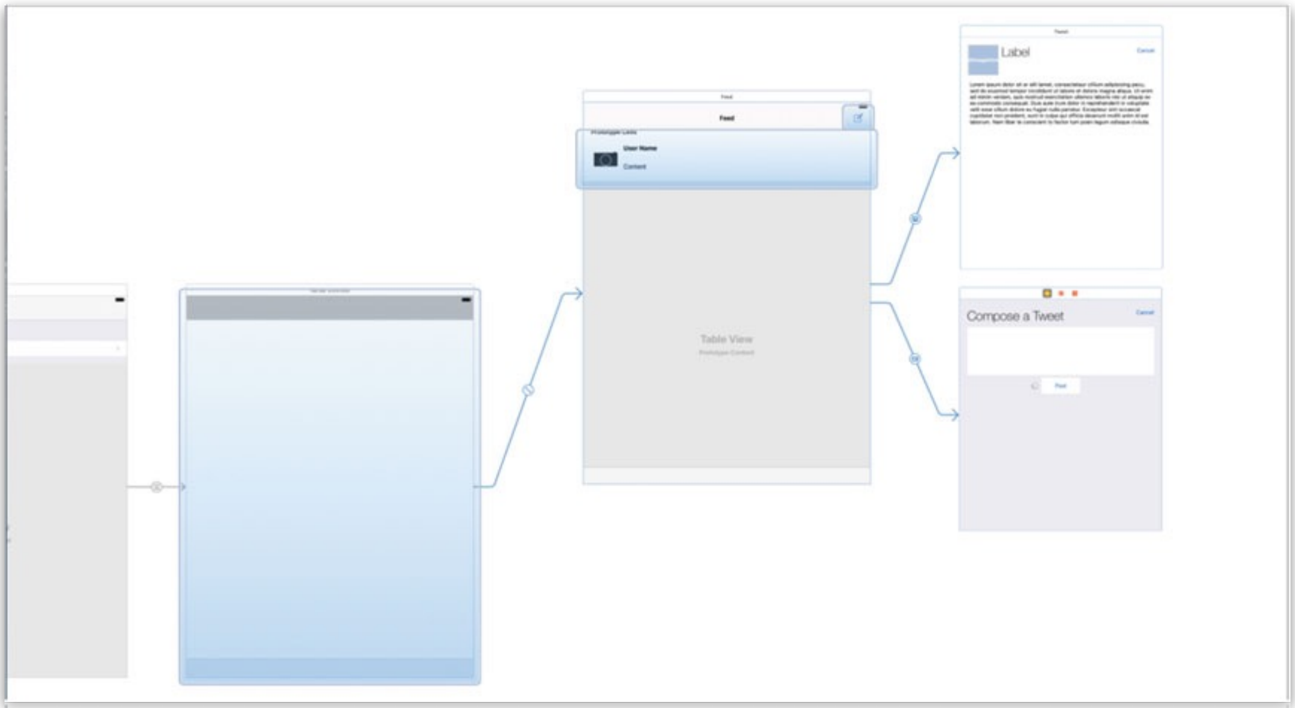
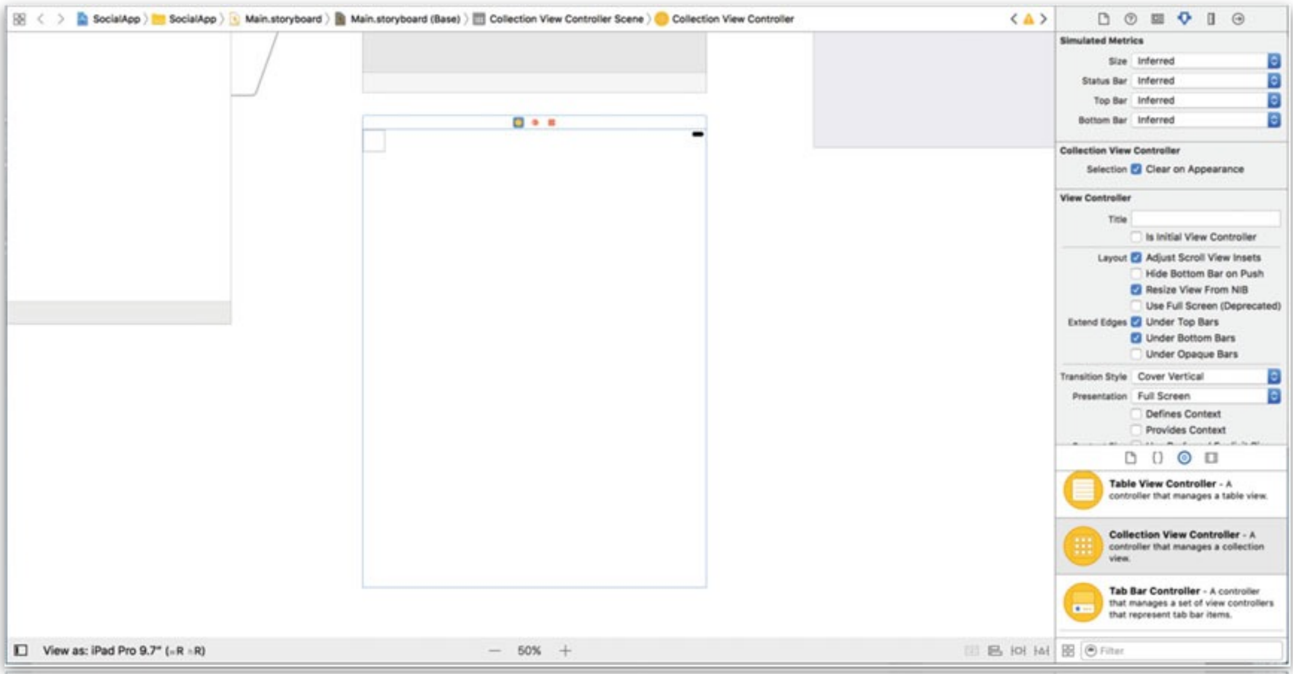


Figure 37. *Arranging the story board in anticipation of the Collection view controller*

2. Drag in a Collection view controller from the Object Library and position it below the **Feed scene**, as shown in Figure 38.


SocialApp
SocialApp
Main.storyboard
Main.storyboard (Base)
Collection View Controller Scene
Collection View Controller
View as: iPad Pro 9.7" (wR hR)
50%
Filter

Simulated Metrics
Size: Inferred
Status Bar: Inferred
Top Bar: Inferred
Bottom Bar: Inferred

Collection View Controller
Selection: [✓] Clear on Appearance

View Controller
Title:
[ ] Is Initial View Controller
Layout: [✓] Adjust Scroll View Insets
[ ] Hide Bottom Bar on Push
[✓] Resize View From NIB
[ ] Use Full Screen (Deprecated)
Extend Edges: [✓] Under Top Bars
[✓] Under Bottom Bars
[ ] Under Opaque Bars
Transition Style: Cover Vertical
Presentation: Full Screen
[ ] Defines Context
[ ] Provides Context

Table View Controller - A controller that manages a table view.

Collection View Controller - A controller that manages a collection view.

Tab Bar Controller - A controller that manages a set of view controllers that represent tab bar items.

Figure 38. *Dragging in a Collection view controller from the Object Library*

3. Create a relationship between the tab bar controller and the Collection view controller. To do this, select the tab bar controller and then Control+drag a connection to the **Collection view controller**, as shown in Figure 39.

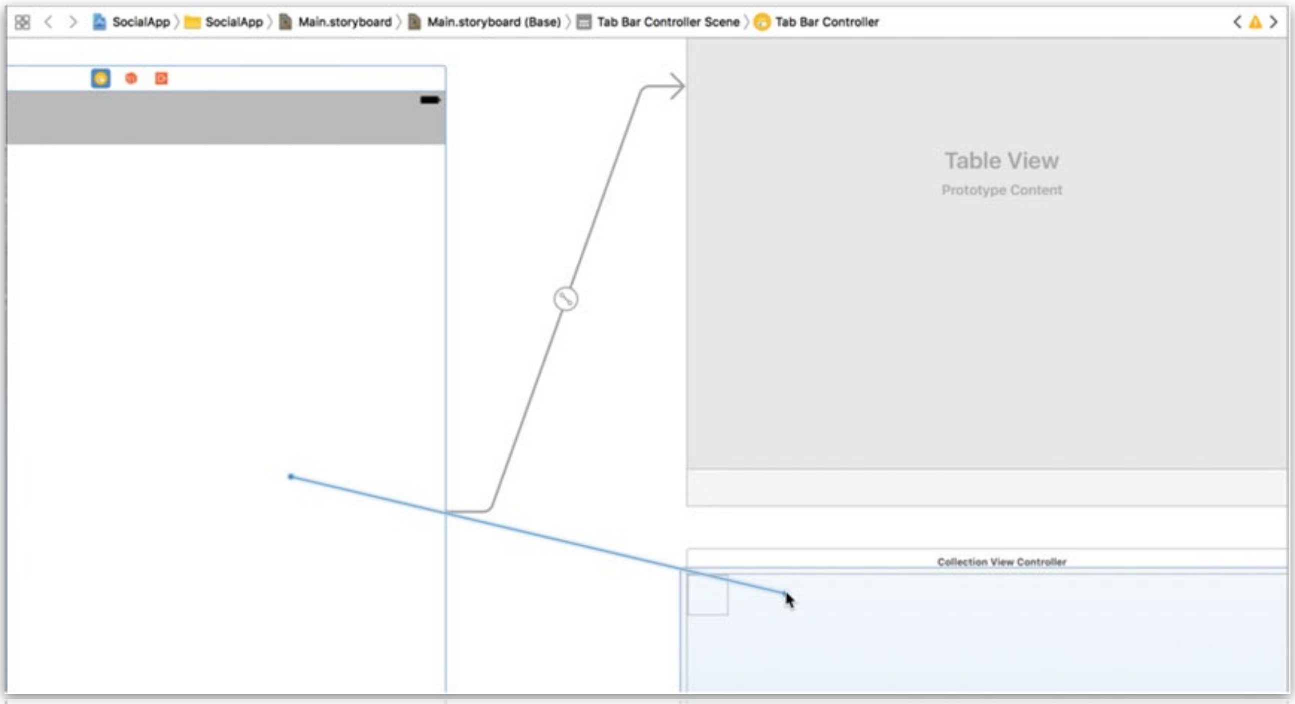


Figure 39. *Control+dragging a connection from the tab bar controller to the Collection view controller*

4. When you release the mouse, select **View Controllers** under **Relationship Segues**. You now have two tabs but no icons or title.

5. To set the **Feed tab bar** button title, select **Item** from the **Document Outline** within the **Feed scene** and open the **Attributes Inspector**, as shown in Figure 40.

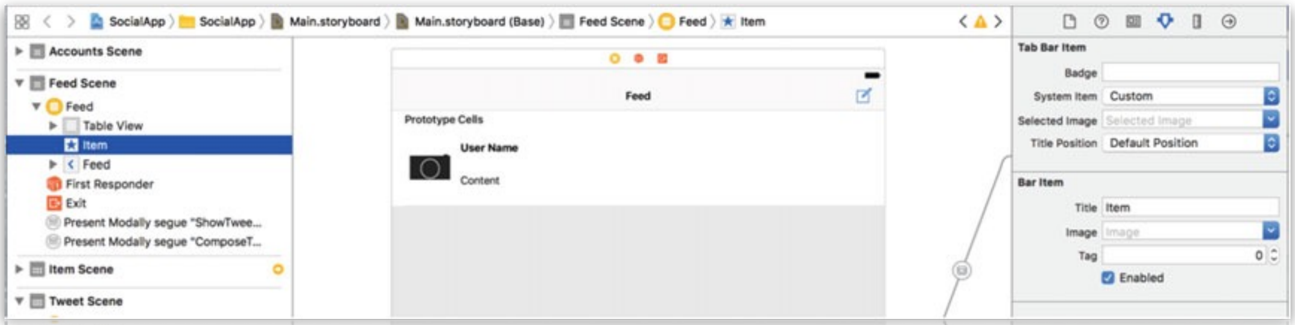


Figure 40. *Selecting the Feed tab bar button*

6. Change the **System Item** drop-down list from Custom to Recents.

7. Next, expand the **Collection View Controller Scene** in the **Document Outline**, again select **Item** and, in the **Attributes Inspector**, change **System Item** from **Custom** to **Contacts**. Your tab bar for the collection view should now look like the one shown in Figure 41.

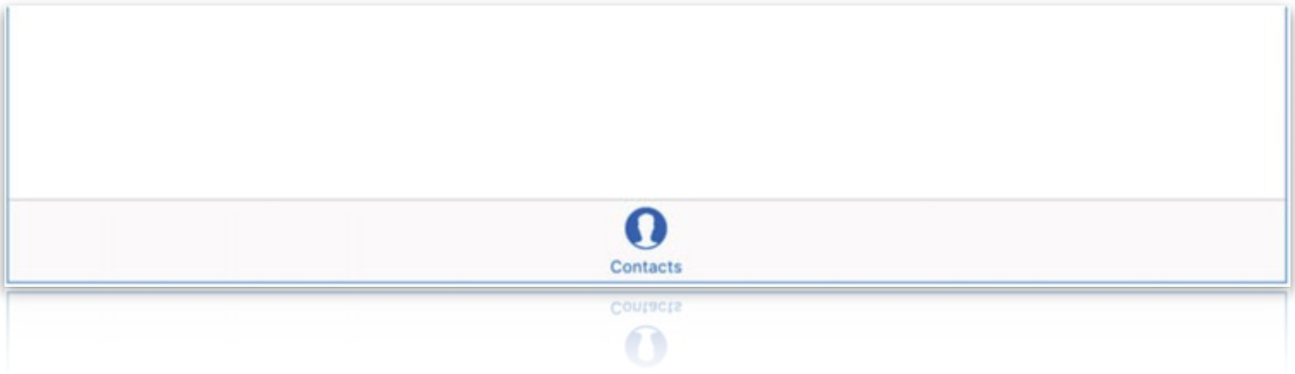


Figure 41. *The tab bar controller featuring a pre-built contacts tab*

8. As you did with the other view controllers, you need to create a customer view controller class file. In this instance, you're subclassing **UICollectionViewController**. Right-click the **View Controllers** group in the **Project Navigator** and select **New File**.

9. When prompted, select the **Cocoa Touch Class** option and click Next. Set the **Subclass** value to **UICollectionViewController** and the **Class** value to **FollowingViewController** and then click **Next**. Accept the default folder Xcode suggests to save the file and click **Create**.

Now that you've successfully created the last custom view controller for this application, you're ready to configure the visual aspects of the collection view before fetching the user details followed by the selected account from Twitter.

Configuring a Collection View

We've already mentioned that the **UICollectionView** class is very similar to the **UITableView** class in terms of methods and the fact that they both use cells to present large amounts of data to the users. They also have sections with independent headers and footers. Yet despite these similarities, the collection-view configuration in Xcode is significantly different from that of the table view.

To begin, open **Main.storyboard** from the **Project Navigator** and position the storyboard so that you can see the collection view, as shown in Figure 42. Structurally, what you're looking at isn't really any different from what you started with in the table view, as shown in Figure 21 of the 2nd part of the practice. The white-bordered box in the top-left corner of the view is the prototype cell.

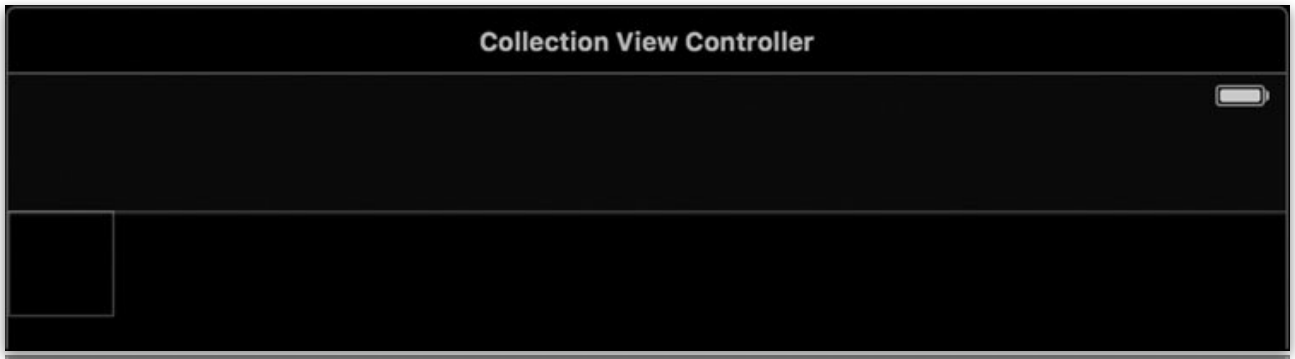


Figure 42*. The top of the Collection view controller showing the prototype cell*

Select the collection view so that you can look at the key attributes in the **Attributes Inspector**. To do this, click the main area of the collection view or select **Collection View** from the **Document Outline**, as shown in Figure 43.

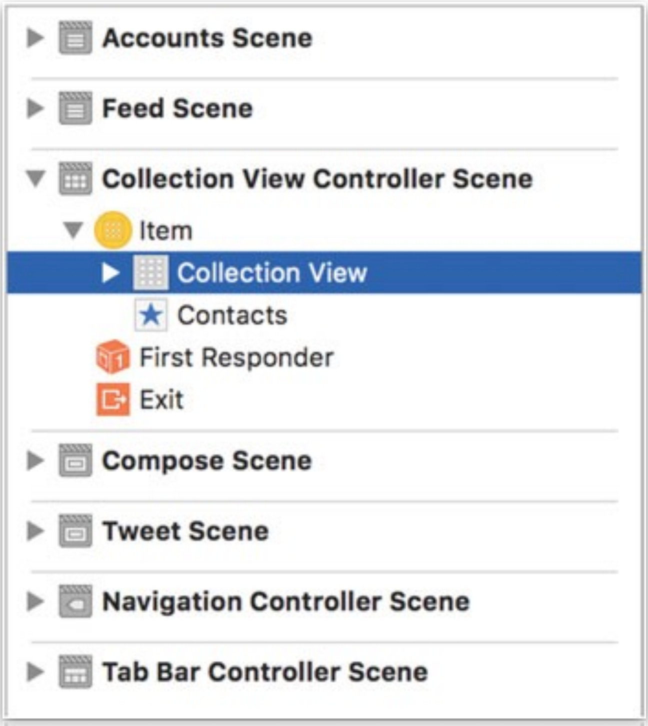


Figure 43*. Selecting the collection view from the Document Outline*

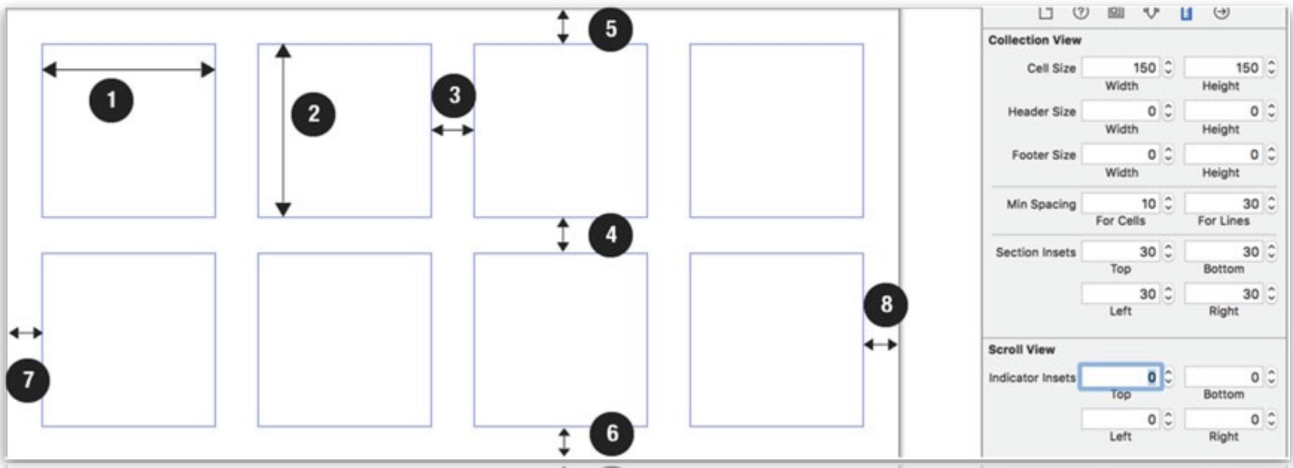
Open the **Attributes Inspector**. Let's take a closer look at the key options available to you:

- *Items*: Unlike the table view, collection views don't have a static mode. The **Items** attribute increases and decreases the number of prototype cells. No matter how many items you have to display, if they have a single type of appearance, then you only need one cell, which you reuse.

- *Layout*: In a collection view, the layout is a separate entity from the view. The default layout is **Flow**, which provides a grid of items continuing uninterrupted in a fixed direction. Changing the attribute to **Custom** exposes a class selector whereby you can specify a custom **UICollectionViewLayout**.

- *Scroll Direction*: As you might expect, this attribute controls the direction in which the cells are positioned for scrolling.

- *Accessories:* The **Section Header** and **Section Footer** options allow you to add a prototype header and footer to the section. Unlike with table views, you can't manually specify any text in either container; it must be set programmatically. Unlike in other views, much in collection views depends on the settings of the **Size Inspector**. When you open the **Size Inspector**, you see many configurable values; Figure 8-44 shows the different sizes and where they take effect. In this example, the number of items is set to 8 to help you visualize how the cells react to one another.



Collection View

Cell Size	150 (Width)	150 (Height)
Header Size	0 (Width)	0 (Height)
Footer Size	0 (Width)	0 (Height)
Min Spacing	10 (For Cells)	30 (For Lines)
Section Insets	30 (Top)	30 (Bottom)
	30 (Left)	30 (Right)

Scroll View

Indicator Insets	0 (Top)	0 (Bottom)
	0 (Left)	0 (Right)

Figure 44. *The different sizes you can alter in a collection view*

- *Cell Width and Height*: Points 1 and 2 represent the width and height of the cell, respectively. The cell doesn't have to be square; the two values can change independently of each other.

- *Minimum Spacing*: The first value (highlighted by point 3), For Cells, sets a minimum value for the horizontal spacing between cells. This is useful because, by default, the cells are spaced nicely, and the horizontal gap is far greater than 10 points. However, you know it won't slip below 10 points if the size of the view changes. The **For Lines** value shown by point 4 sets a minimum width between the rows of cells.

- *Section Insets*: These four values control the spacing around the outside of the section of cells, so the cells function as a collective entity. When you increase the value of any of the sizes illustrated by points 5–8, you move the cells farther from that side of the view. By default, the **Section Insets** values are set to 0, which can leave content feeling squashed; set a nice inset value to bring the cells in from the edge, which is more visually appealing.

Follow these steps:

1. For the **Followers** collection view, set the **Cell Size Width** and **Height** values to 75, set **Minimum Spacing** to 10 For Cells and 30 For Lines, and set all **Section Insets** to 30.

2. Before you start adding code to the **Followers** view controller, you need to specify the class the view controller uses. Select the **Collection view controller** by either clicking the top bar of the view controller or selecting **Item** under **Collection View Controller Scene** in the **Document Outline**.

3. Open the **Identity Inspector** and set the class to **FollowingViewController**, as shown in Figure 45.

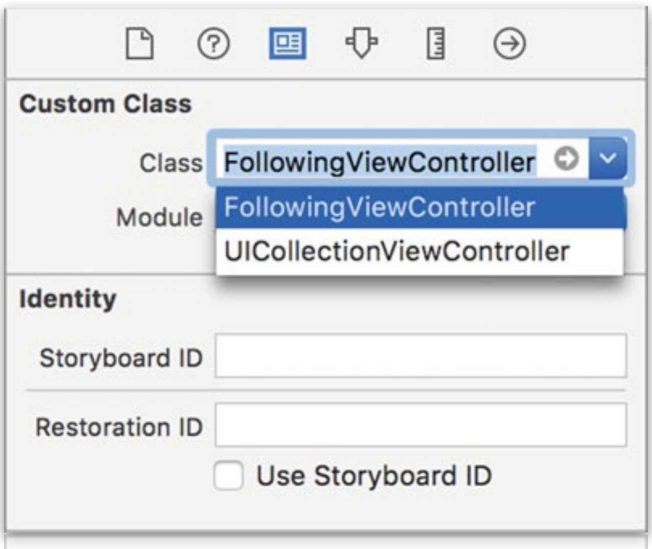

Custom Class
Class: FollowingViewController
Module
Options: FollowingViewController, UICollectionViewController
Identity
Storyboard ID
Restoration ID
Use Storyboard ID

Figure 45. *Setting Class to Following View Controller*

4. Select the single cell in the **Following view controller**. Open the **Attributes Inspector** and set the **Identifier** value to **FollowerCell**.

Displaying Items in a Collection View

You've configured the collection view in Xcode, but to finish the view you need to write the code that retrieves the list of users the app follows. We won't focus too much on the code for retrieving the list of followed users, but rather on the key methods of the **UICollectionViewController** class:

1. Open **FollowingViewController.swift** from the **Project Navigator**. The first thing you need to do is import the **Social** and **Accounts** frameworks. After the line import **UIKit**, add the following highlighted import statements:

```
import UIKit
import Accounts
import Social
```

2. Set the reuse identifier and declare a number of instance variables, just as you did with the **Feed view controller**. You need an **NSMutableArray** instance called following to store the details of each user the account follows and an **NSCache** object called **imageCache**. The start of your file, with these items added, should look like this:

```
import UIKit
import Accounts
import Social

private let reuseIdentifier = "FollowerCell"

class FollowingViewController: UICollectionViewController {

   var following : NSMutableArray?
   var imageCache : NSCache<AnyObject, AnyObject>?
```

3. Move down to the **viewDidLoad** method; in this method, just as you did with the **Feed view controller**, you want to initialize the queue object, set the navigation bar title to *Following*, and then call the **retrieveUsers** function, which you'll add shortly. Add the highlighted code to your **viewDidLoad** method:

```
override func viewDidLoad() {
   super.viewDidLoad()
   // Register cell classes
   self.collectionView!.register(UICollectionViewCell.self, forCellWithReuseIdentifier:
   reuseIdentifier)
   self.tabBarController?.navigationItem.title = "Following"
   retrieveUsers()
}
```

4. Look at the retrieveUsers function. Create a stub for the function below the **viewDidLoad** method, as shown next:

```
   retrieveUsers()

}

func retrieveUsers() {
}
```

5. Much of this code is the same as that used in the **Feed view controller**, so we don't present the code in any detail. Clear the following array and then retrieve **selectAccount** from the stored user preferences, as shown next:

```
func retrieveUsers() {
   following?.removeAllObjects()
   let userDefaults = UserDefaults.standard
   let accountData = userDefaults.object(forKey: "selectedAccount") as! Data
   let selectedAccount = NSKeyedUnarchiver.unarchiveObject(with: accountData) as!
ACAccount
}
```

6. Declare an **SLRequest** object and instantiate it using the URL specified by the Twitter API for retrieving a list of "friends," as Twitter refers to the API that returns "up to 200 users," followed by the supplied account:

```
let accountData = userDefaults.object(forKey: "selectedAccount") as! Data
let selectedAccount = NSKeyedUnarchiver.unarchiveObject(with: accountData) as! ACAccount
let requestURL = URL(string: "https://api.twitter.com/1.1/friends/list.json?count=200")
if let request = SLRequest(forServiceType: SLServiceTypeTwitter,
                          requestMethod: SLRequestMethod.GET,
                          url: requestURL,
                          parameters: nil) {
   request.account = selectedAccount
}
```

Note For more information on configuring the Twitter Friends/List API, visit https://dev.twitter.com/ docs/api/1.1/get/friends/list.

7. You need to call the **performRequestWithHandler** method of the **SLRequest**. Just as before, you check for a valid status code and parse the **JSON** response before picking the "*users*" array from the parsed code and assigning it to the following array. Calling the **UICollectionView** method **reloadData** causes three methods to be called. Add the following code to complete this method:

```
request.account = selectedAccount
request.perform()
{
   responseData, urlResponse, error in
   if(urlResponse?.statusCode == 200) {
      do {
         let followingData = try JSONSerialization.jsonObject(with: responseData!,
           options: JSONSerialization.ReadingOptions.mutableContainers) as! NSDictionary

         self.following = followingData.object(forKey: "users") as? NSMutableArray
      }
      catch let error as NSError {
         print("json error: \(error.localizedDescription)")
      }
   }
```

```
    DispatchQueue.main.async {
        self.collectionView!.reloadData()
    }
}
```

8. On to the delegate methods. Just as with the table view, you have to specify the number of sections via the **numberOfSections(:in:)** method. Set it to return **1**, as shown next:

```
override func numberOfSections(in collectionView: UICollectionView) -> Int {
    return 1
}
```

9. You need to specify how many of the potential 200 cells to render should appear **viathecollectionView(_ :numberOfItemsInSection:)** method, which returns the number of rows in the following array:

```
override func collectionView(_ collectionView: UICollectionView, numberOfItemsInSection
section: Int) -> Int {
    if let followCount = following?.count {
        return followCount
    } else {

        return 0
    }

}
```

10. Scroll down and locate the **cellForItemAtIndexPath** method. It's used to initialize the cell and set its content, just as its **UITableView** equivalent does.

11. Use the highlighted code to pull the relevant data for the current index from the following array and store it in an **NSDictionary** object before extracting the URL for the user's profile image, as you did in the **Feed view controller**:

```
override func collectionView(_ collectionView: UICollectionView, cellForItemAt indexPath:
IndexPath) -> UICollectionViewCell {
    let cell = collectionView.dequeueReusableCell(withReuseIdentifier: reuseIdentifier, for:
    indexPath)
    let userData = following?.object(at: indexPath.row) as! NSDictionary
    let imageURLString = userData.object(forKey: "profile_image_url") as! String
    return cell
}
```

12. You need to set up the image that is programmatically added to the cell using the **addSubview** method, because no **UIImageView** exists in the cell, and then return the cell object. Complete the method with this highlighted code:

```
override func collectionView(_ collectionView: UICollectionView, cellForItemAt indexPath:
IndexPath) -> UICollectionViewCell {
   let cell = collectionView.dequeueReusableCell(withReuseIdentifier: reuseIdentifier, for:
   indexPath)
   let userData = following?.object(at: indexPath.row) as! NSDictionary
   let imageURLString = userData.object(forKey: "profile_image_url") as! String

   let operationQueue = OperationQueue.main
   operationQueue.maxConcurrentOperationCount = 4
   if let image = imageCache?.object(forKey: imageURLString as AnyObject) as? UIImage {
      let imageView = UIImageView(image: image) as UIImageView
      imageView.bounds = cell.frame
      cell.addSubview(imageView)
   } else {

      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 imageView = UIImageView(image: downloadedImage)
                     imageView.bounds = cell.frame
                     if let cell = self.collectionView!.cellForItem(at: indexPath) as
                     UICollectionViewCell! {
                        cell.addSubview(imageView)
                     }
                  }

                  self.imageCache?.setObject(downloadedImage, forKey: imageURLString
                  as AnyObject)
               }
            }
         }

         catch let error as NSError {
            print("parse error: \(error.localizedDescription)")
         }
      }

   }

   return cell
}
```

That completes the collection view and the practice. Run your application. As long as you're following some other Twitter users, your collection view should populate with user avatars, as shown in Figure 46. Note that we've used the Xcode logo instead of some of the faces, for privacy reasons.

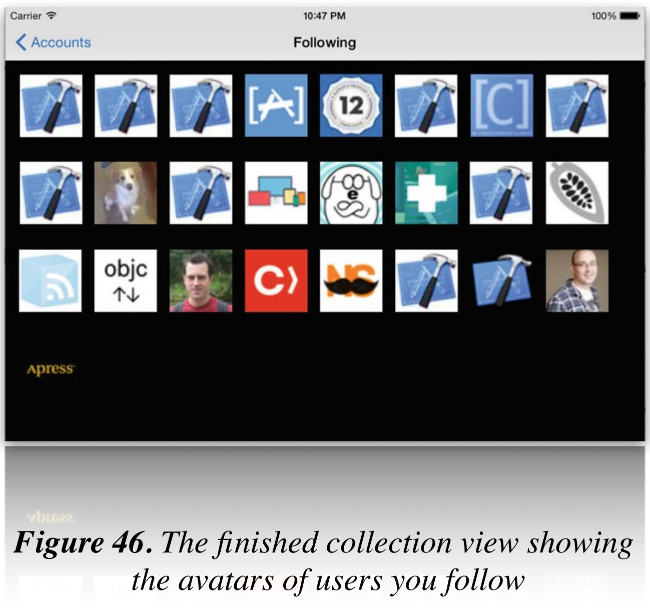


***Figure 46.** The finished collection view showing the avatars of users you follow*

Summary

You covered a lot in this practices. The things you learned:

- The difference between static and prototype table cells
- When to use a grouped or a plain style table view
- How to create a custom table cell
- How to fetch data from the Internet
- How to parse JSON data
- How to embed a tab bar controller into your application
- How to persist user preferences even when the application has closed

***