9 Simple Swift App

In this bonus chapter, we're going to make a simple Notes app using a storyboard, UIkit, and Swift.

Creating a simple app with Swift

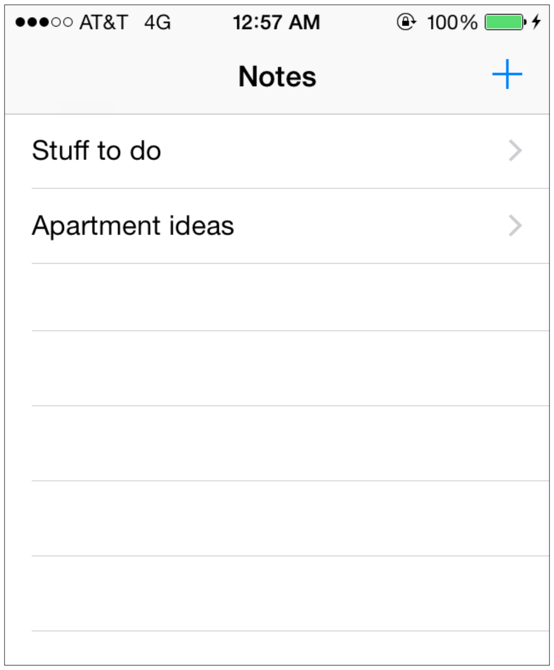
Apple's new programming language, Swift, is still in its infancy at the time of writing this book. However, there is still loads of documentation to reference, combined with an entire blog dedicated to Swift created by Apple. So, before we go ahead and start creating games, let's create a simple iOS app in Swift to get a feel of how the syntax works in conjunction with our familiar UIkit elements.

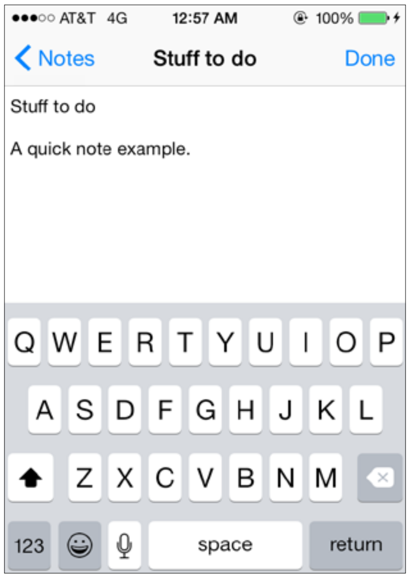
Goal of the app

Our app is going to be a very simple version of the Notes app on all iOS devices. It will be able to provide the following basic features of a note-taking app:

- Creating new notes
- Editing an existing note
- Saving changes to a note
- Recording changes to the device (data persistence)

Here are two screenshots of what the final version of the app is going to look like:





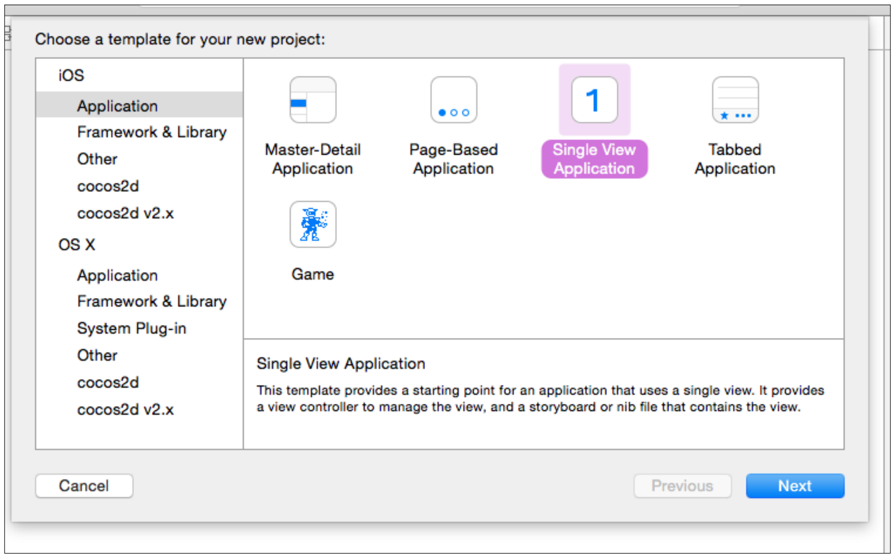
Stuff to do

A quick note example.

It won't be too hard, so let's get right into it by first creating a project to start from.

Creating a blank project

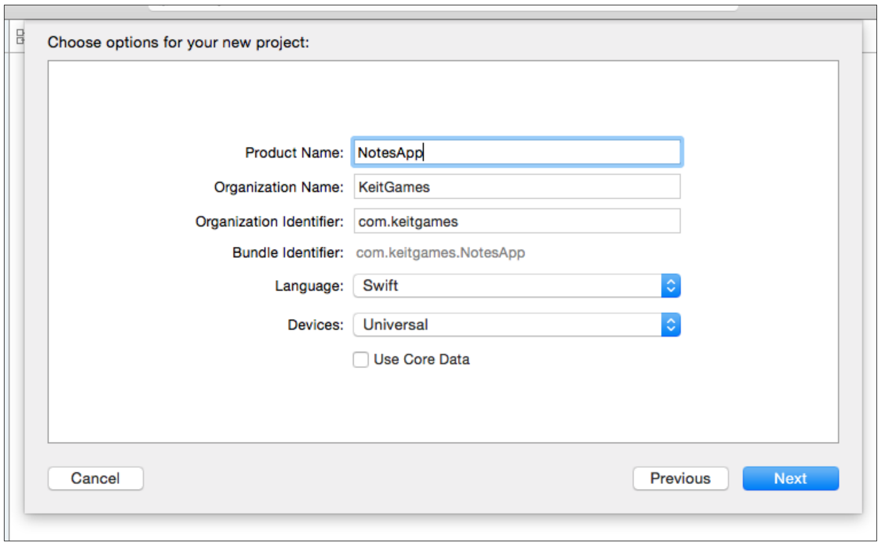
Open Xcode and go to **File** | **New** | **Project**. From the **iOS Application** section, select **Single View Application** and click on **Next**, as shown in the following screenshot:



Single View Application

This template provides a starting point for an application that uses a single view. It provides a view controller to manage the view, and a storyboard or nib file that contains the view.

Name it `NotesApp` (or whatever you wish), set the language to **Swift**, and set the devices to **Universal**. Then click on **Next**. This is also shown in the following screenshot for your reference:



Product Name:	NotesApp
Organization Name:	KeitGames
Organization Identifier:	com.keitgames
Bundle Identifier:	com.keitgames.NotesApp
Language:	Swift
Devices:	Universal

☐ Use Core Data

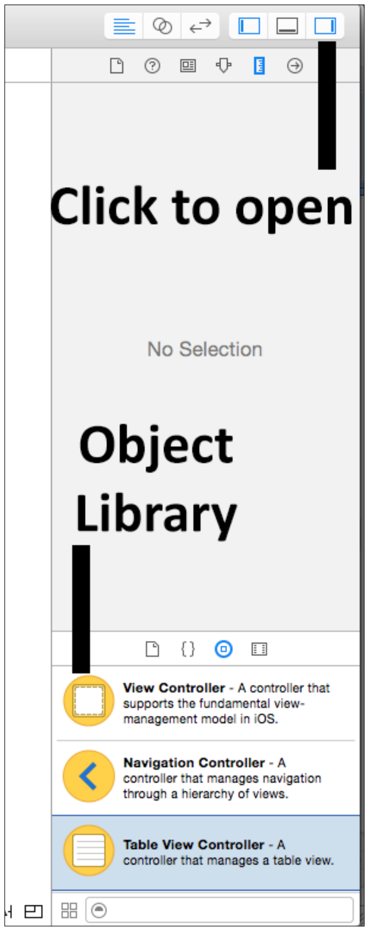
Save the project wherever you want; it doesn't particularly matter.

Now let's set up our UI so that it's not so boring.

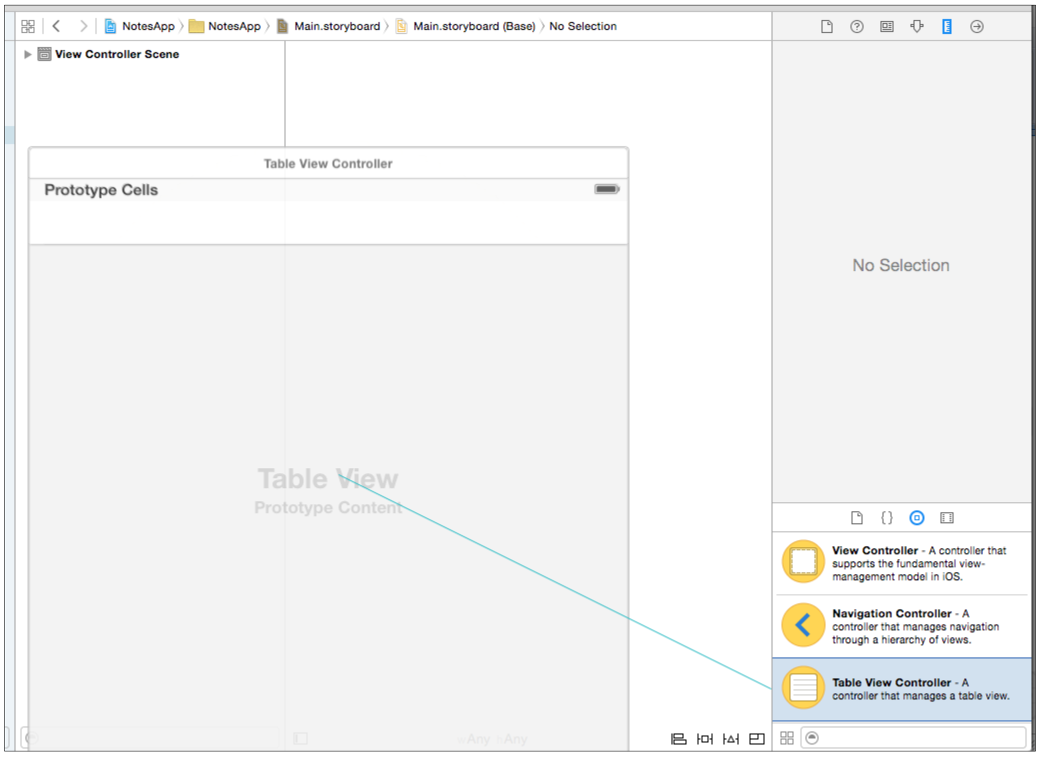
Setting up the view controllers in Interface Builder

Click on `Main.storyboard` to access Interface Builder, and you will see a visual representation of the app. Although we told Xcode that we wanted to create a **Single View Application**, we can still modify it however we want.

The first order of business is to make sure you can see the **Object Library** within Xcode. If you don't have the **Object Library** visible, click on the sidebar button in the top-right corner of Xcode to open it, as shown in the following screenshot:


Click to open
No Selection
Object Library
View Controller - A controller that supports the fundamental view-management model in iOS.
Navigation Controller - A controller that manages navigation through a hierarchy of views.
Table View Controller - A controller that manages a table view.

From there, drag a **Table View Controller** controller to any place within the storyboard. It doesn't really matter where you put it, but for clarity, it's probably best to place it to the left of the currently existing view controller.

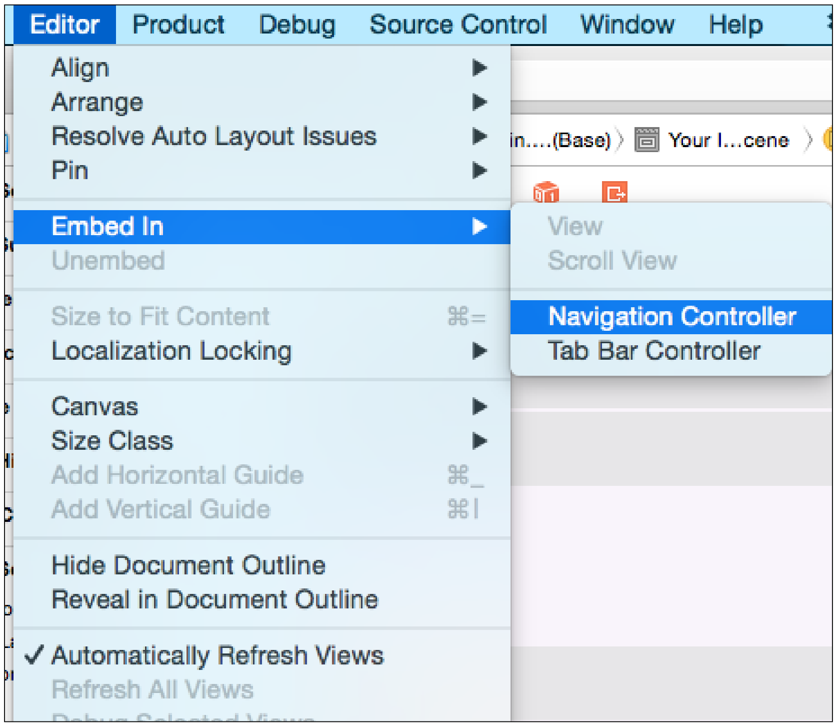


View Controller - A controller that supports the fundamental view-management model in iOS.

Navigation Controller - A controller that manages navigation through a hierarchy of views.

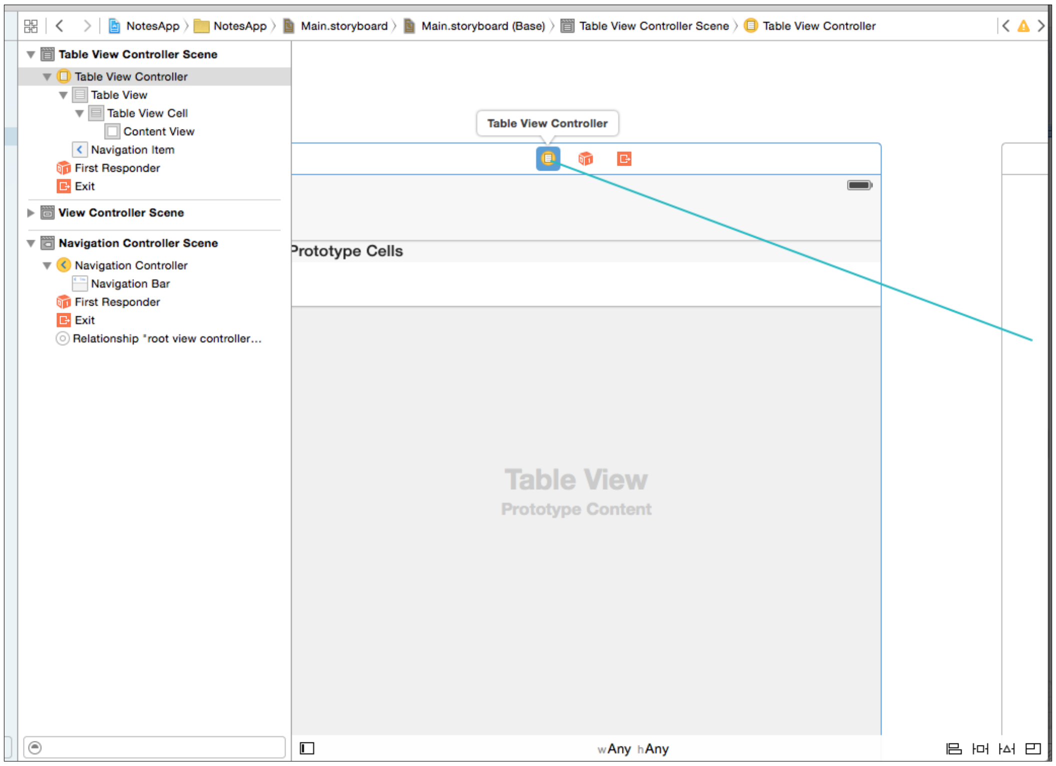
Table View Controller - A controller that manages a table view.

Next, select **Table View Controller** (it should be selected by default) and go to **Editor | Embed In | Navigation Controller**. This will add a navigation bar to the top of our table view, as well as **Navigation Controller** for us, and link up the two view controllers automatically so that we don't have to.



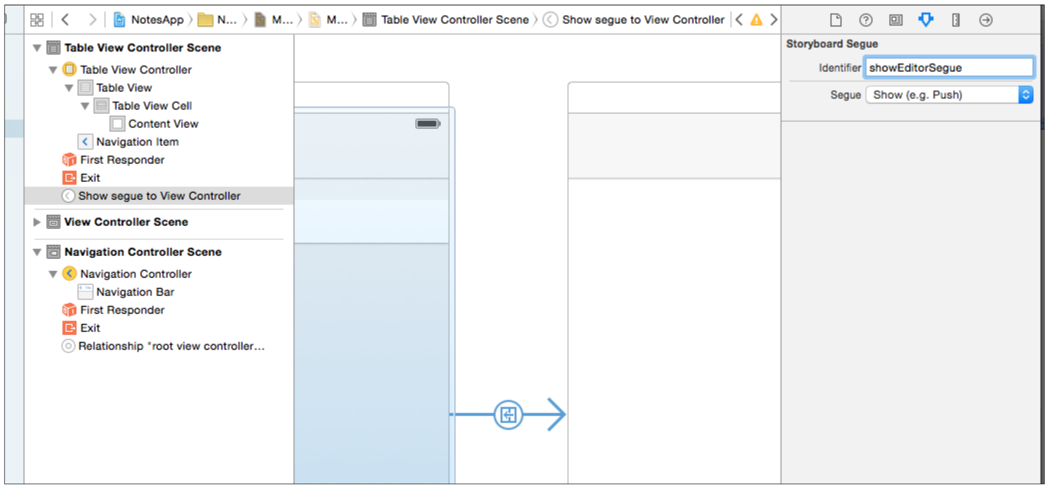
After that, select **Table View Controller**, press *Ctrl*, and drag it from the **View Controller** icon (the yellow box, either in the document outline or above the view controller), to the blank view controller that was initially created.

From the list that pops up, select the **Push** segue.



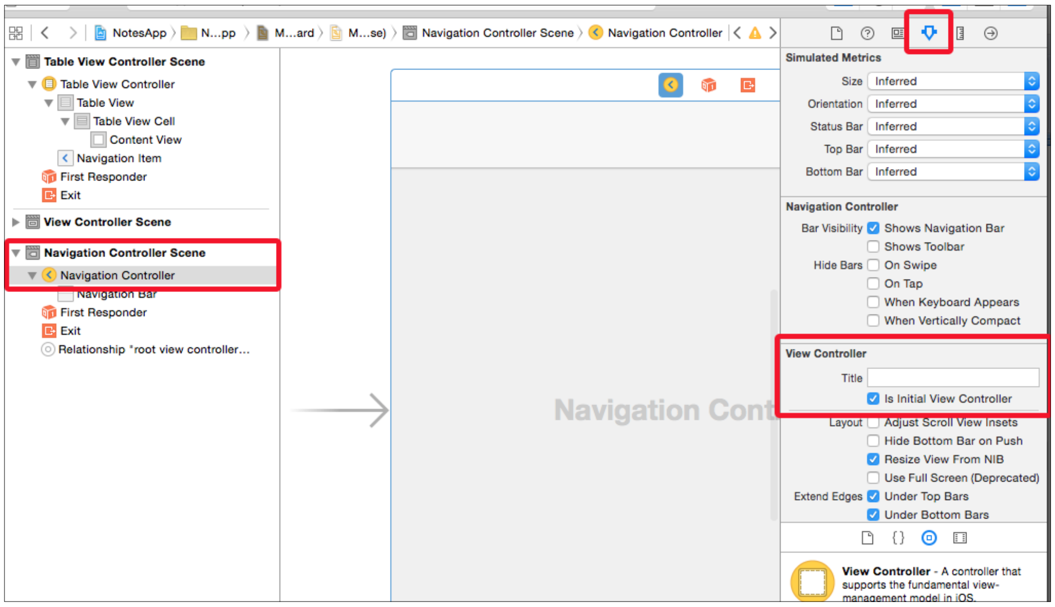
This creates a segue (the name for a transition between two `UIViewController` objects). Essentially, if you're unfamiliar, it will automatically create a navigation bar at the top of your view controller and allow you to transition to this view controller, with the navigation automatically built in.

With the segue created, click on the **Segue** icon between the table views on the blank view controller, and give it a name such as `showEditorSegue` or something relevant.


NotesApp
Table View Controller Scene
Show segue to View Controller
Table View Controller Scene
Table View Controller
Table View
Table View Cell
Content View
Navigation Item
First Responder
Exit
Show segue to View Controller
View Controller Scene
Navigation Controller Scene
Navigation Controller
Navigation Bar
First Responder
Exit
Relationship "root view controller...

Storyboard Segue
Identifier showEditorSegue
Segue Show (e.g. Push)

Finally, we want to set the initial view controller to be our navigation controller. So, click on **Navigation Controller**, and under the **Attributes Inspector**, check the box that says **Is Initial View Controller**.



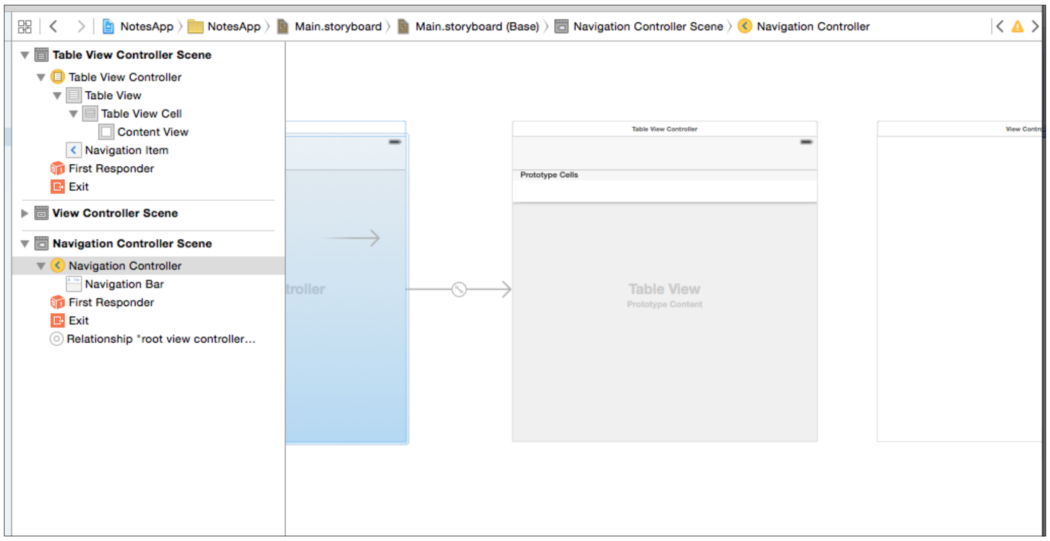
Simulated Metrics
Size Inferred
Orientation Inferred
Status Bar Inferred
Top Bar Inferred
Bottom Bar Inferred

Navigation Controller
Bar Visibility ✓ Shows Navigation Bar
☐ Shows Toolbar
Hide Bars ☐ On Swipe
☐ On Tap
☐ When Keyboard Appears
☐ When Vertically Compact

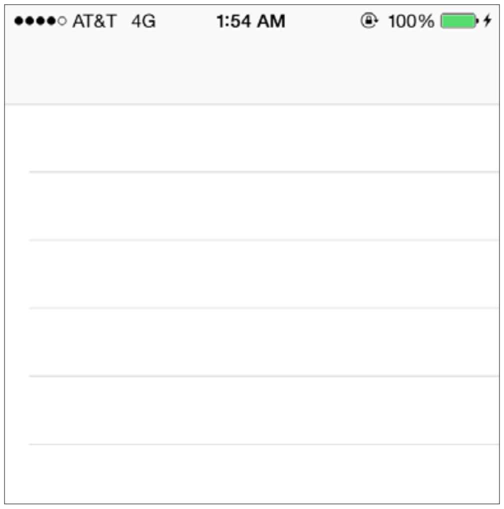
View Controller
Title
✓ Is Initial View Controller
Layout ☐ Adjust Scroll View Insets
☐ Hide Bottom Bar on Push
✓ Resize View From NIB
☐ Use Full Screen (Deprecated)
Extend Edges ✓ Under Top Bars
✓ Under Bottom Bars

View Controller - A controller that supports the fundamental view-management model in iOS.

Alternatively, wherever you have the initial view controller that you started with, there should be an arrow coming in from the left of it. Click and drag that arrow onto the **Navigation Controller** (feel free to zoom out a bit if you have to).



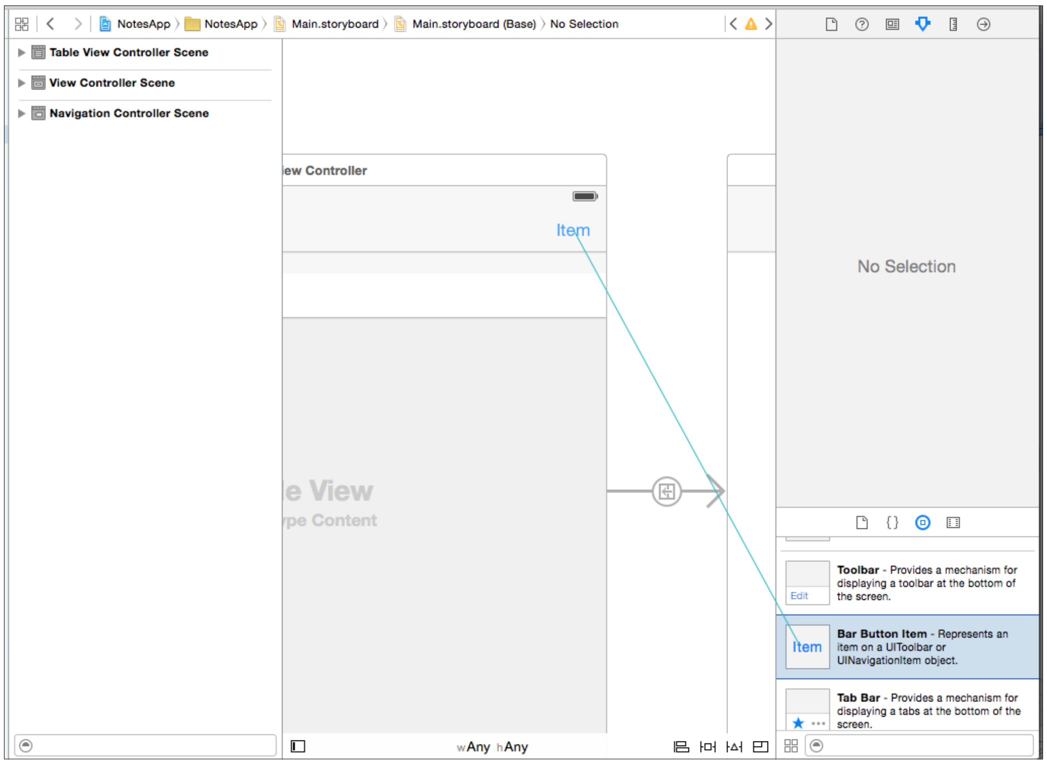
If you run the app at this point, you'll get to an app with a blank list of items. You will also get a navigation bar across the top that has no title or buttons. So we can't test our editor segue yet.



Set up the UI

With our view controllers set up properly, we can add to our views the necessary elements that will make up the core of our interaction.

First, drag a **Bar Button Item** from the **Object Library** onto the far right side of the navigation bar of **Table View Controller**. As you drag the button close to it, it will be highlighted in blue (not shown in the following screenshot):



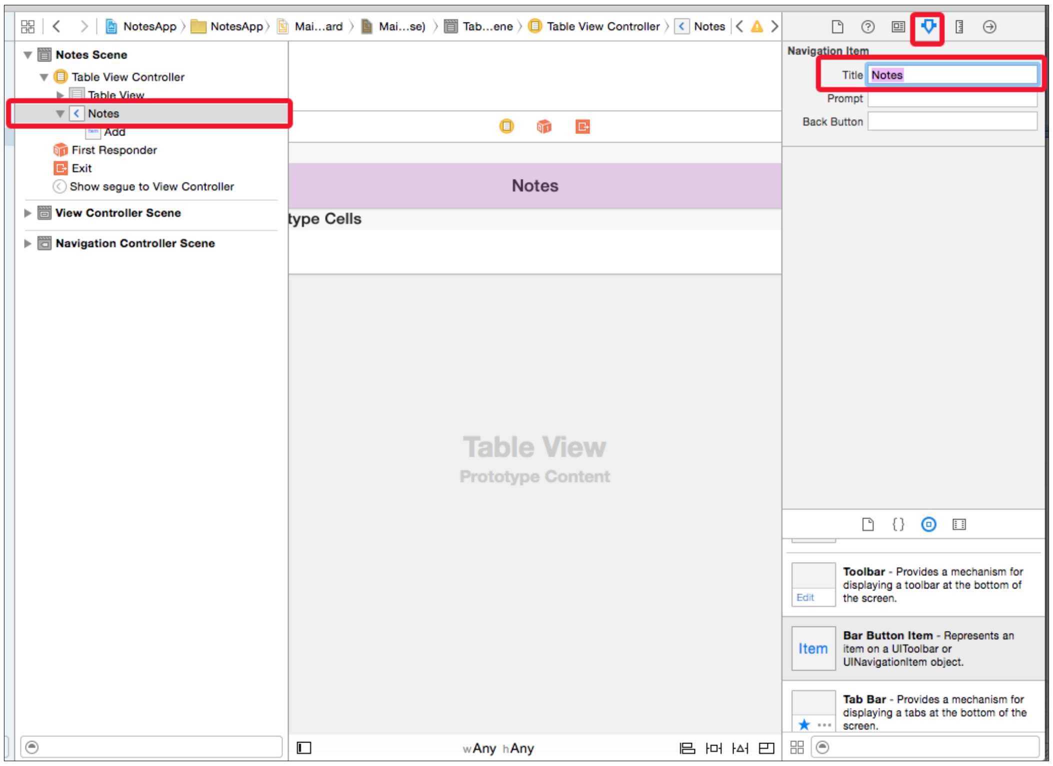
Toolbar - Provides a mechanism for displaying a toolbar at the bottom of the screen.

Bar Button Item - Represents an item on a UIToolbar or UINavigationItem object.

Tab Bar - Provides a mechanism for displaying a tabs at the bottom of the screen.

Then, change the button's identifier to **Add**, turning it into a **+** button.

Next, click on **Navigation Item** in **Table View Controller**, and change the **Title** property to **Notes**. When you do this, it'll change the name of the scene to **Notes Scene**.



Navigation Item

Title	Notes
Prompt	
Back Button	

Toolbar - Provides a mechanism for displaying a toolbar at the bottom of the screen.

Bar Button Item - Represents an item on a UIToolbar or UINavigationItem object.

Tab Bar - Provides a mechanism for displaying a tabs at the bottom of the screen.

Now click on **Table View Cell** (right below where it says **Prototype Cells**, or in the outline on the left-hand side), and change the cell's **Identifier** to **CELL**. The actual name doesn't matter, but we're just naming it something obvious for later reference:

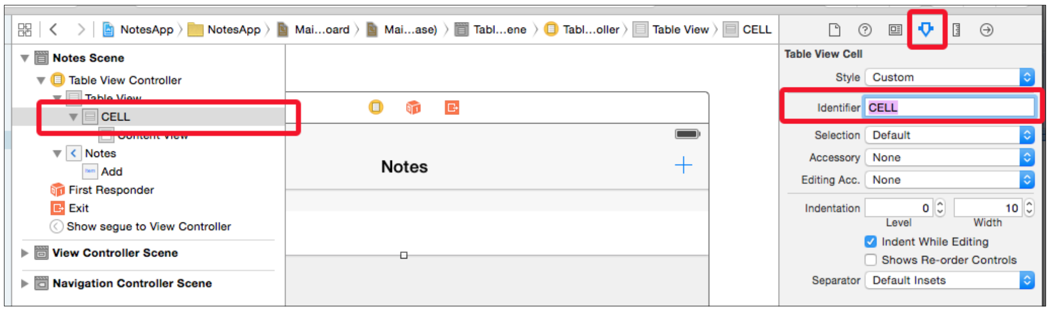
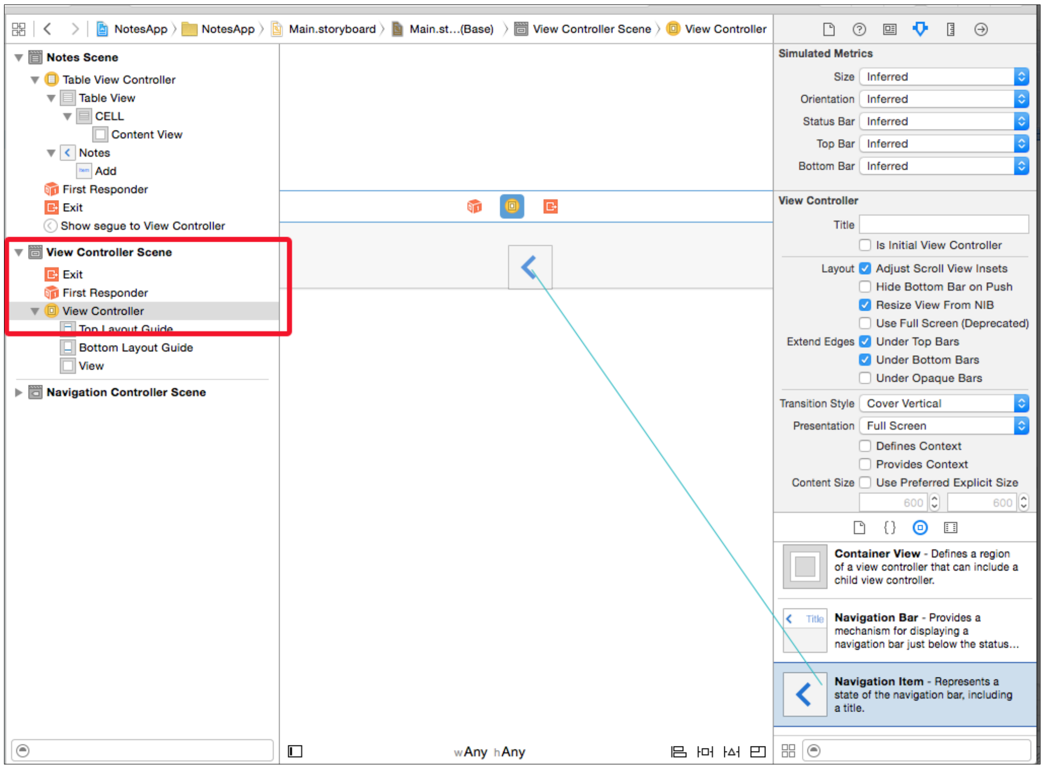

NotesApp 〉 NotesApp 〉 Mai…oard 〉 Mai…ase) 〉 Tabl…ene 〉 Tabl…oller 〉 Table View 〉 CELL
Notes Scene
Table View Controller
Table View
CELL
Content View
Notes
Add
First Responder
Exit
Show segue to View Controller
View Controller Scene
Navigation Controller Scene
Notes

Table View Cell

Style	Custom
Identifier	CELL
Selection	Default
Accessory	None
Editing Acc.	None
Indentation	0 Level, 10 Width
	☑ Indent While Editing
	☐ Shows Re-order Controls
Separator	Default Insets

With the **Table View Cell** still selected, change the **Accessory** property to **Disclosure Indicator** (which will give it a **<** arrow on the right-hand side of the cell).

That's all for the table view, so now select the **View Controller** scene, and drag a **Navigation Item** onto the view:



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

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
Content Size: ☐ Use Preferred Explicit Size
600 600

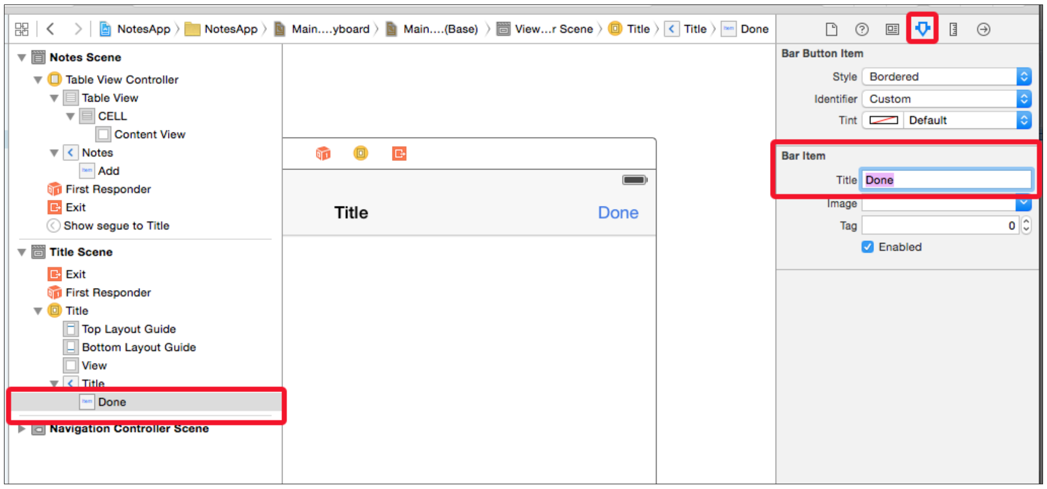
Container View - Defines a region of a view controller that can include a child view controller.

Navigation Bar - Provides a mechanism for displaying a navigation bar just below the status...

Navigation Item - Represents a state of the navigation bar, including a title.

Feel free to change the title (or leave it; it's up to you). It doesn't matter here because we're going to modify it in the code anyway.

Next, drag a **Bar Button Item** onto the right side of the newly created **Navigation Item** (similar to **Table View**), and change the **Title** property of the button to **Done**.



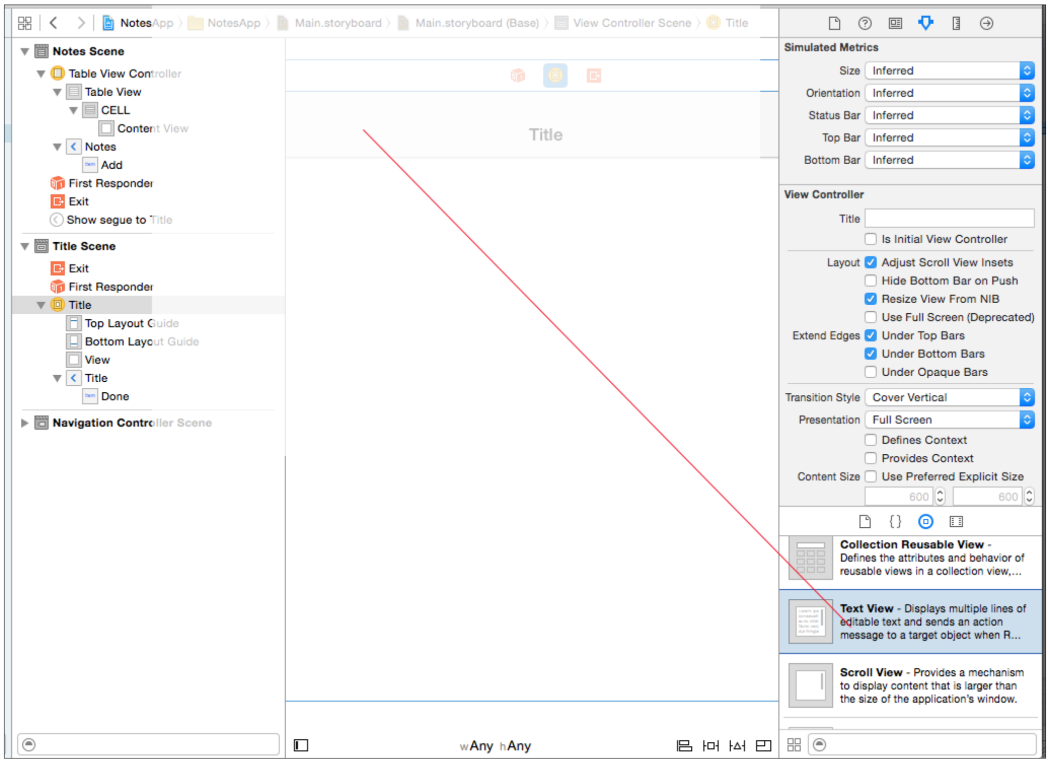
Bar Button Item
Style: Bordered
Identifier: Custom
Tint: Default

Bar Item
Title: Done
Image:
Tag: 0
✓ Enabled

Alternatively, you can change the button's **Identifier** to **Done**, which will make the text bold in addition to showing the word **Done**. Whichever way you do it, the button will still work as intended later on.

But what's a note-taking app without the ability to modify the notes? So, let's add a giant text view to the screen. With the main scene selected (**Title Screen** or whatever you ended up calling it), drag a **Text View** (not a **Text Field**) onto the view's **Navigation Bar**.

This ensures that the text view fills the entirety of the screen across all edges.



Simulated Metrics

Size	Inferred
Orientation	Inferred
Status Bar	Inferred
Top Bar	Inferred
Bottom Bar	Inferred

View Controller

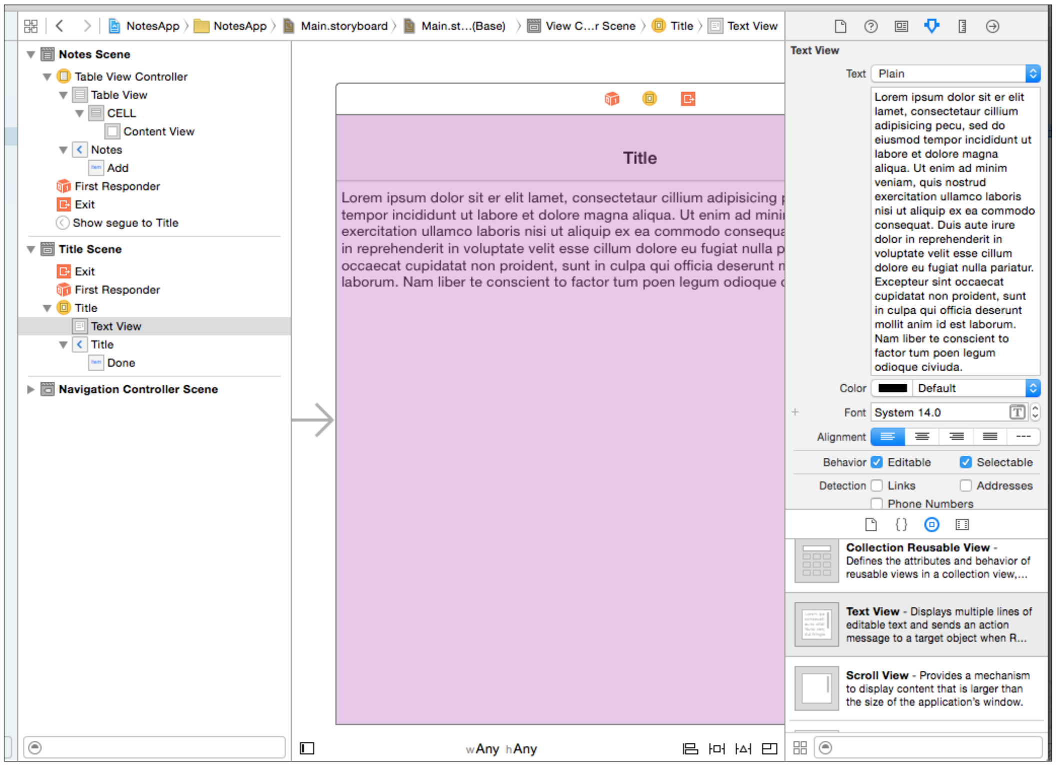
Title	
	☐ Is Initial View Controller
Layout	✓ Adjust Scroll View Insets
	☐ Hide Bottom Bar on Push
	✓ Resize View From NIB
	☐ Use Full Screen (Deprecated)
Extend Edges	✓ Under Top Bars
	✓ Under Bottom Bars
	☐ Under Opaque Bars
Transition Style	Cover Vertical
Presentation	Full Screen
	☐ Defines Context
	☐ Provides Context
Content Size	☐ Use Preferred Explicit Size
	600 600

Collection Reusable View - Defines the attributes and behavior of reusable views in a collection view,...

Text View - Displays multiple lines of editable text and sends an action message to a target object when R...

Scroll View - Provides a mechanism to display content that is larger than the size of the application's window.

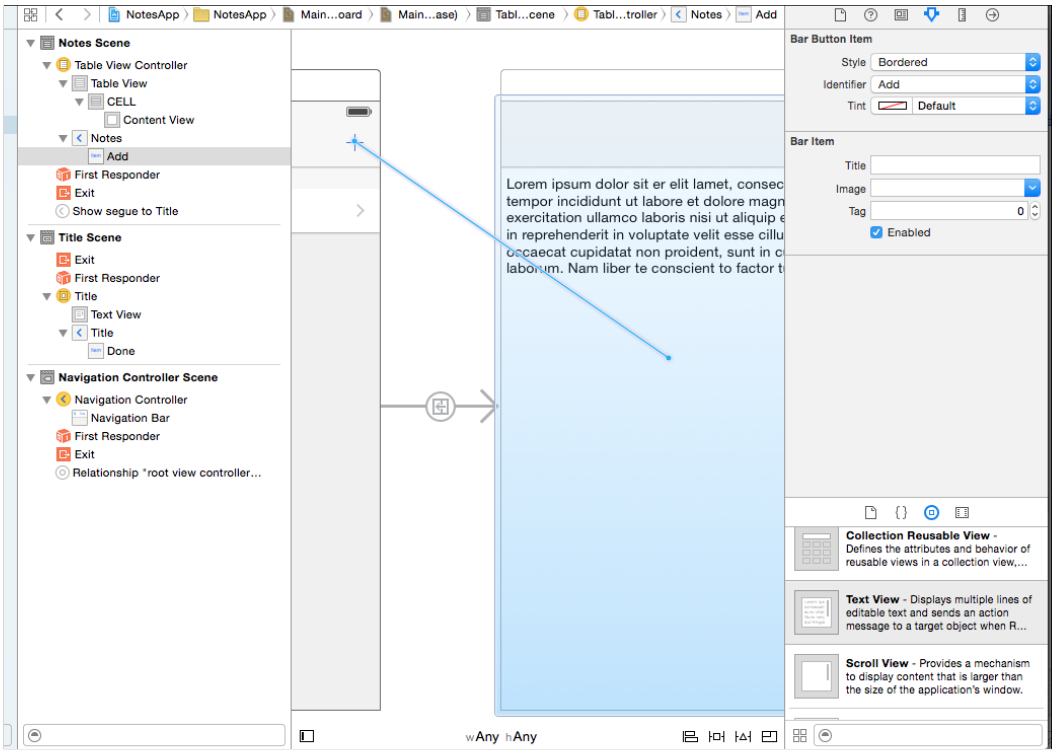
Here's what it looks like when properly placed:


NotesApp
NotesApp
Main.storyboard
Main.st…(Base)
View C…r Scene
Title
Text View
Notes Scene
Table View Controller
Table View
CELL
Content View
Notes
Add
First Responder
Exit
Show segue to Title
Title Scene
Exit
First Responder
Title
Text View
Title
Done
Navigation Controller Scene
Title
wAny hAny
Collection Reusable View - Defines the attributes and behavior of reusable views in a collection view,…
Text View - Displays multiple lines of editable text and sends an action message to a target object when R…
Scroll View - Provides a mechanism to display content that is larger than the size of the application's window.

Lorem ipsum dolor sit er elit lamet, consectetaur cillium adipisicing pecu, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum. Nam liber te conscient to factor tum poen legum odioque civiuda.

Text View
Text: Plain
Color: Default
Font: System 14.0
Alignment
Behavior: [✓ Editable] [✓ Selectable]
Detection: Links, Addresses, Phone Numbers

To make sure that the **Text View** is filling the screen properly, press *Ctrl* and drag from the **+** button of the **Table View** to the Title Scene, and select **Show** under **Action Segue**.



Lorem ipsum dolor sit er elit lamet, consec
tempor incididunt ut labore et dolore magn
exercitation ullamco laboris nisi ut aliquip e
in reprehenderit in voluptate velit esse cillu
occaecat cupidatat non proident, sunt in c
laborum. Nam liber te conscient to factor t

Bar Button Item

Style	Bordered
Identifier	Add
Tint	Default

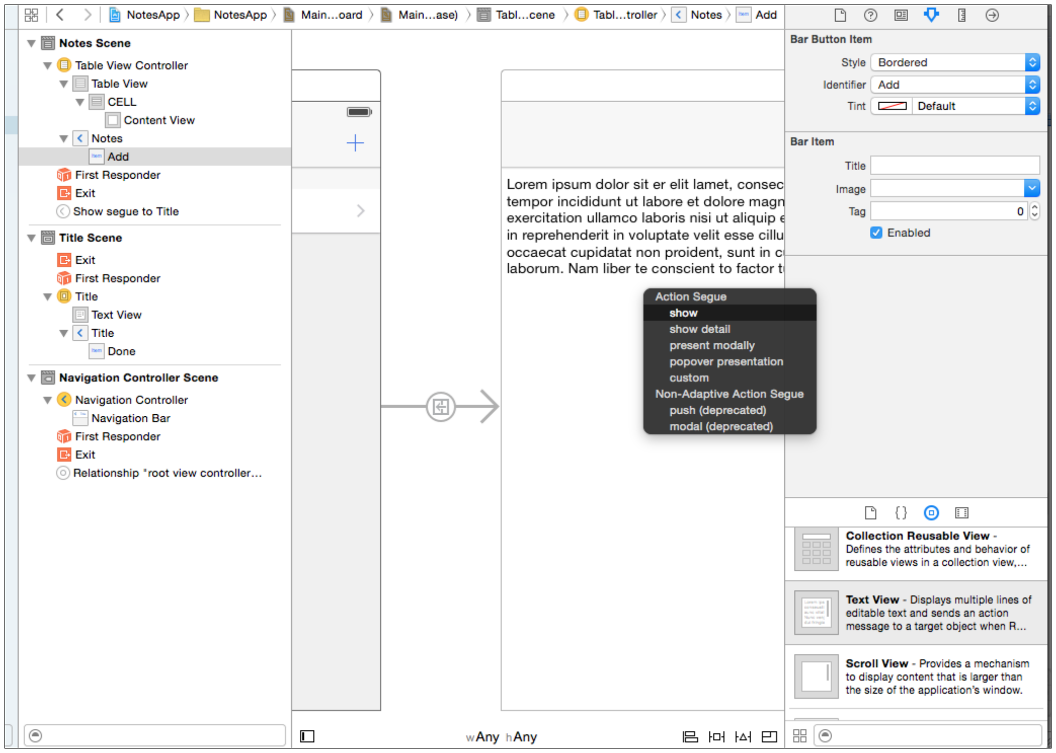
Bar Item

Title	
Image	
Tag	0
	✓ Enabled

Collection Reusable View - Defines the attributes and behavior of reusable views in a collection view,...

Text View - Displays multiple lines of editable text and sends an action message to a target object when R...

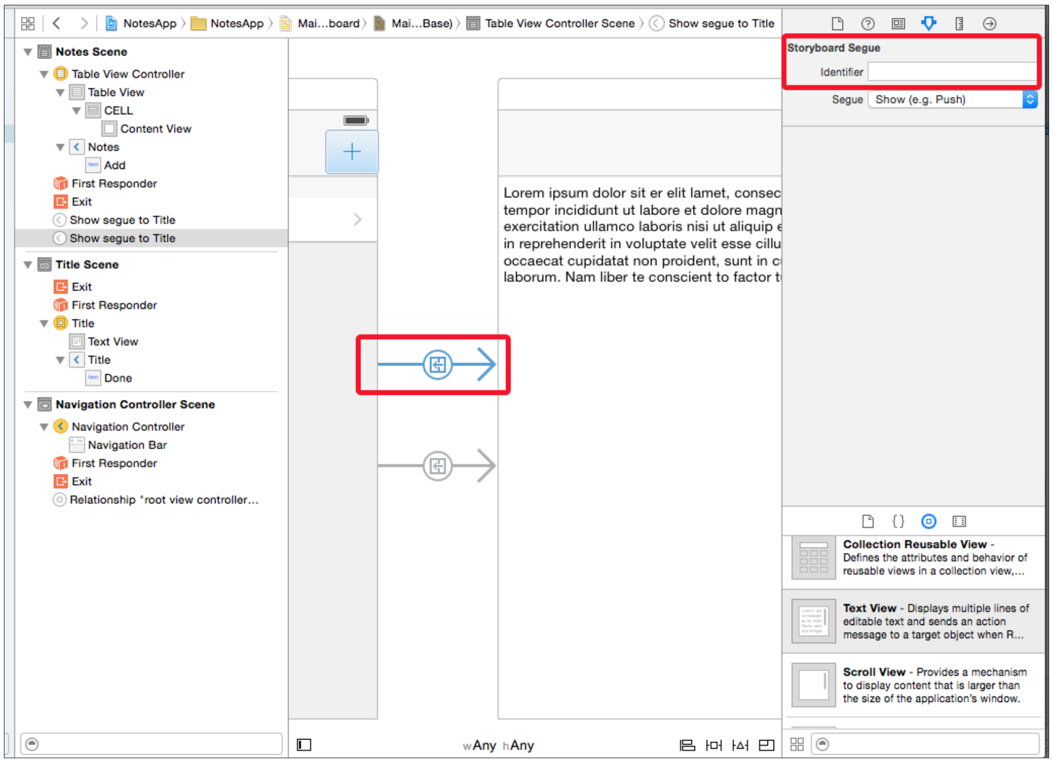
Scroll View - Provides a mechanism to display content that is larger than the size of the application's window.



NotesApp › NotesApp › Main...oard › Main...ase) › Tabl...cene › Tabl...troller › Notes › Add
Notes Scene
Table View Controller
Table View
CELL
Content View
Notes
Add
First Responder
Exit
Show segue to Title
Title Scene
Exit
First Responder
Title
Text View
Title
Done
Navigation Controller Scene
Navigation Controller
Navigation Bar
First Responder
Exit
Relationship "root view controller...
Lorem ipsum dolor sit er elit lamet, consec
tempor incididunt ut labore et dolore magn
exercitation ullamco laboris nisi ut aliquip e
in reprehenderit in voluptate velit esse cillu
occaecat cupidatat non proident, sunt in c
laborum. Nam liber te conscient to factor t
Action Segue
show
show detail
present modally
popover presentation
custom
Non-Adaptive Action Segue
push (deprecated)
modal (deprecated)
Bar Button Item
Style Bordered
Identifier Add
Tint Default
Bar Item
Title
Image
Tag 0
Enabled
Collection Reusable View - Defines the attributes and behavior of reusable views in a collection view,...
Text View - Displays multiple lines of editable text and sends an action message to a target object when R...
Scroll View - Provides a mechanism to display content that is larger than the size of the application's window.
wAny hAny

Then run the app and click on the **+** button. If you can read all of the default text (and it's not going off the screen), you're good. Otherwise, you'll have to delete the **Text View** and try again.

Once you know it's working, you can get rid of the segue you just created from the **+** button. Just click on either of the segues between the two view controllers, and whichever doesn't have an identifier is the segue you're going to delete.



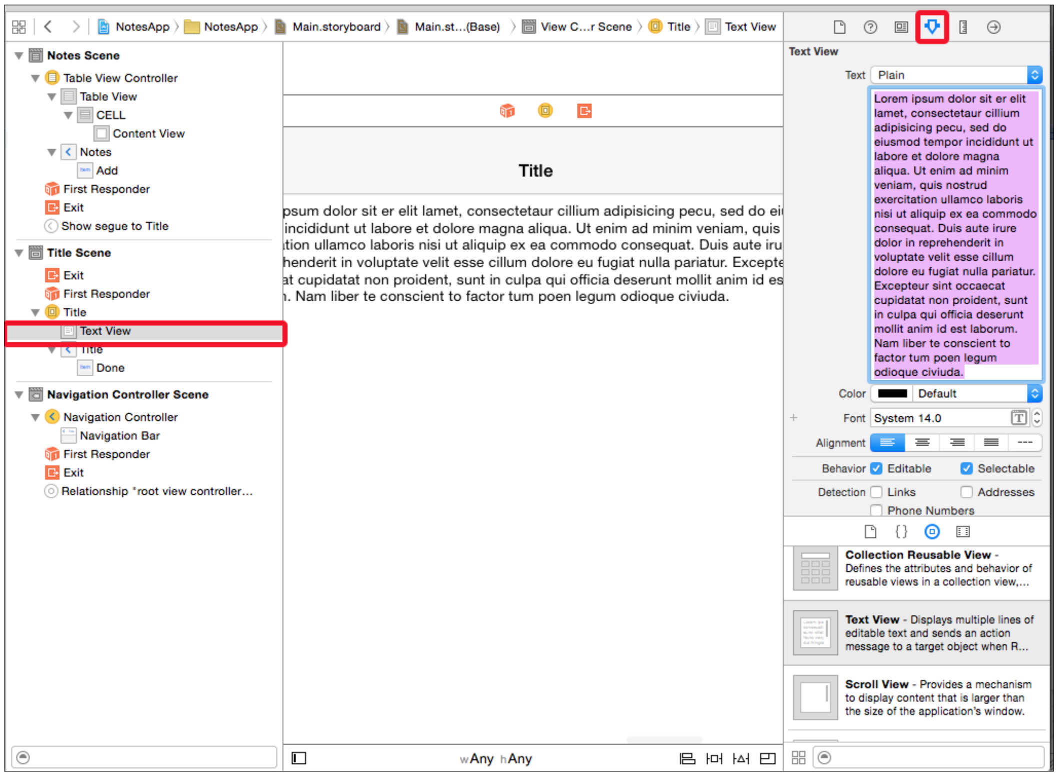
Lorem ipsum dolor sit er elit lamet, consec
tempor incididunt ut labore et dolore magn
exercitation ullamco laboris nisi ut aliquip e
in reprehenderit in voluptate velit esse cillu
occaecat cupidatat non proident, sunt in c
laborum. Nam liber te conscient to factor t

Collection Reusable View - Defines the attributes and behavior of reusable views in a collection view,...

Text View - Displays multiple lines of editable text and sends an action message to a target object when R...

Scroll View - Provides a mechanism to display content that is larger than the size of the application's window.

Now that you know the **Text View** is working properly, you can (if you wish) select the text view and just delete all of the text within it.


NotesApp > NotesApp > Main.storyboard > Main.st...(Base) > View C...r Scene > Title > Text View
Notes Scene
Table View Controller
Table View
CELL
Content View
Notes
Add
First Responder
Exit
Show segue to Title
Title Scene
Exit
First Responder
Title
Text View
Title
Done
Navigation Controller Scene
Navigation Controller
Navigation Bar
First Responder
Exit
Relationship "root view controller...
Title
wAny hAny
Text View
Collection Reusable View - Defines the attributes and behavior of reusable views in a collection view,...
Text View - Displays multiple lines of editable text and sends an action message to a target object when R...
Scroll View - Provides a mechanism to display content that is larger than the size of the application's window.

Text View

Text: Plain

Lorem ipsum dolor sit er elit lamet, consectetaur cillium adipisicing pecu, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum. Nam liber te conscient to factor tum poen legum odioque civiuda.

Color: Default

Font: System 14.0

Alignment

Behavior: [✓ Editable] [✓ Selectable]

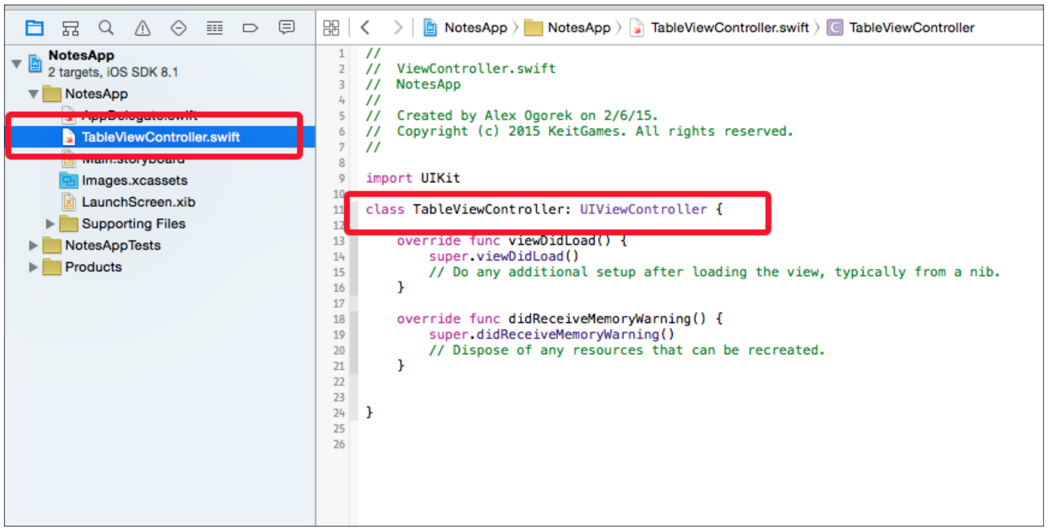
Detection: Links, Addresses, Phone Numbers

Now that we have all our user interface elements laid out, we can start coding the functionality. Yes, so far, all this has been the same as what happens when using Objective-C, but it's still important to know (and set up properly) when making iOS apps.

Set the TableViewController and displaying the data

Even though you could argue that creating a note might be the most important feature of your app, if you can't see the list of notes created, it's a useless app. So, we want to make sure we get that aspect working before anything else.

First, change `ViewController.swift` to `TableViewController.swift` (either by double-clicking slowly or clicking then hitting return). Also change the name of the class within the file (on line 11 or so).



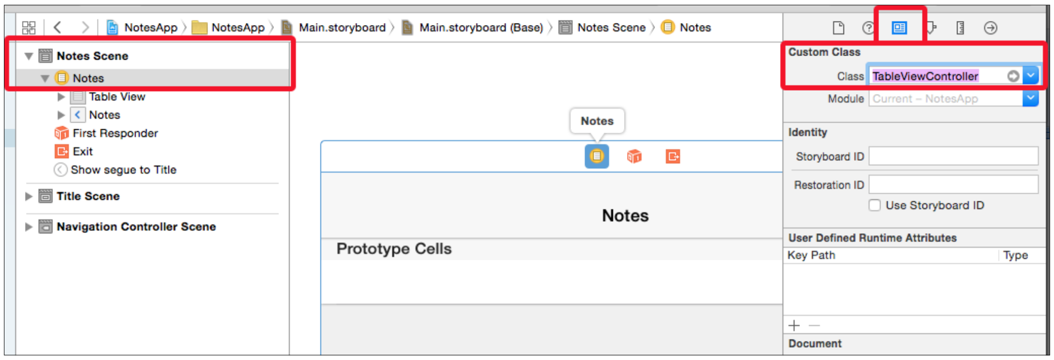
```
1  //
2  //  ViewController.swift
3  //  NotesApp
4  //
5  //  Created by Alex Ogorek on 2/6/15.
6  //  Copyright (c) 2015 KeitGames. All rights reserved.
7  //
8
9  import UIKit
10
11 class TableViewController: UIViewController {
12
13     override func viewDidLoad() {
14         super.viewDidLoad()
15         // Do any additional setup after loading the view, typically from a nib.
16     }
17
18     override func didReceiveMemoryWarning() {
19         super.didReceiveMemoryWarning()
20         // Dispose of any resources that can be recreated.
21     }
22
23
24 }
25
26
```

Since our base/initial view controller is going to be a table view controller, we need to change our main view controller's superclass (or parent class) to `UITableViewController`. So, in `TableViewController.swift`, change the class it inherits from:

```
class TableViewController: UITableViewController {
...
}
```

Now that it's of the `UITableViewController` type, go back to `Main.storyboard`, select the **Notes Scene** (make sure you select the controller and not any of the elements within it), and, under the **Identity Inspector**, change the **Class** property to **TableViewController**.

Make sure it's `TableViewController`, not `UITableViewController`. If you're only seeing `UITableViewController`, it's because you haven't changed the superclass (or parent class) to the type of `UITableViewController`. So, go back and make sure you change the name and inheritance of the class properly (and clear up any errors that exist, as this may also contribute to the storyboard issue).



Class: TableViewController
Module: Current – NotesApp

With the class and the controller linked, we can actually display some data. To do that, we're going to implement three methods that will help the `UITableViewController` object determine what it's going to display. In `TableViewController.swift`, add the following methods below the `didReceiveMemoryWarning` function:

```
override func tableView(tableView: UITableView,
numberOfRowsInSection section: Int) -> Int {

    // return the desired # of elements. In this case, 5
      return 5
}

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

    //grab the "default cell", using the identifier we set up in
      the Storyboard
    var cell = tableView.dequeueReusableCellWithIdentifier("CELL")
      as UITableViewCell

    // set the text to a test value to make sure it's working
    cell.textLabel!.text = "Test Value"
```

```
        //return the newly-modifed cell
        return cell
    }

    override func tableView(tableView: UITableView,
      didSelectRowAtIndexPath indexPath: NSIndexPath) {

        //do nothing at the moment
    }
```

These three methods are fairly self-explanatory, even if you've never used a `UITableView` object before. Plus, the comments should help if you get lost.

If you're wondering why we have to use `override` here, it's because of the inheritance from `UITableViewController`. Instead of simply defining a function implemented by a delegate, we're directly overriding the implementation of that function.

Also note the location and implementation of the return type on these functions (as previously shown in the *Goal of the App* section of this chapter).

> If you have to type these methods yourself, you can start typing `tableView` and it will suggest a list of options. Highlight the option you want using the arrow keys (or clicking on the item), and either hit return or double-click on the item. Then Xcode will generate the code for you, including the override keyword.

Running the app at this point will let you see the table view populated with some values. When you tap on any of the values, nothing happens just yet. We'll get to that soon. First, we need to implement the creation of a new note.

Create an array for notes

The key functionality of a note-taking app is the ability to create new notes. To do that, we just need to implement the functionality of the **+** button that we added to our main view controller. When we create a note, we will want to add it to a list of notes previously created. But since we don't yet have a list to add it to, we must create a list.

In `TableViewController.swift`, we add an instance variable to the top of the class that will store our notes. The structure we're going to use is an array of dictionaries. The keys we're going to use are `title` and `body`. The syntax looks a little wonky, but it's essentially combining the two types, and telling Swift that we're going to be using dictionaries within this array that have both their keys and values as a String type:

```
class TableViewController: UITableViewController {

    //an array of dictionaries
    //  keys = "title", "body"
    var arrNotes = [[String:String]]()

override func viewDidLoad() {
...
}
```

So, imagine a series of notes, each with a title and a body, all arranged within an array. This makes it easy for the table view (as well as for us) when determining which note to display or modify.

Now that we have a list to work with, we can change the hardcoded values in our `tableView` method to the following:

```
override func tableView(tableView: UITableView,
  numberOfRowsInSection section: Int) -> Int {

//we want "the number of elements in the array" to be the number
  of rows
    return arrNotes.count
}

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

    //grab the "default cell"
    var cell = tableView.dequeueReusableCellWithIdentifier("CELL")
      as UITableViewCell

    //set the text to the "title" value of the same index of the
      array
    cell.textLabel!.text = arrNotes[indexPath.row]["title"]

    return cell
}
```

Running the app at this point will show nothing in the table (which is intended, because we haven't added anything to the array yet).

Implement the + button

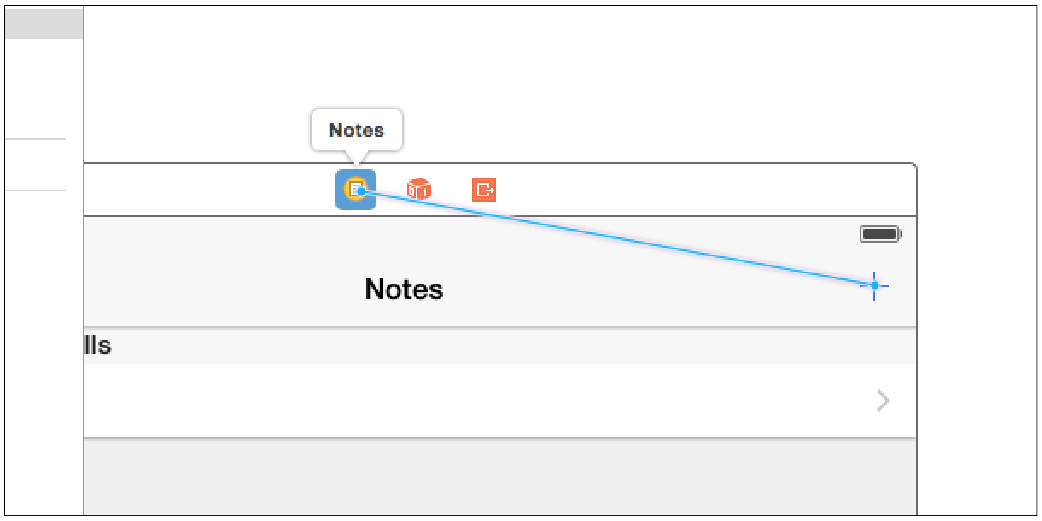
Now that we have an array being created and the table view using it to display any data within it, we can implement the **+** button's add mechanics.

So in `TableViewController.swift`, add the following method. It will be called when the **+** button is tapped:

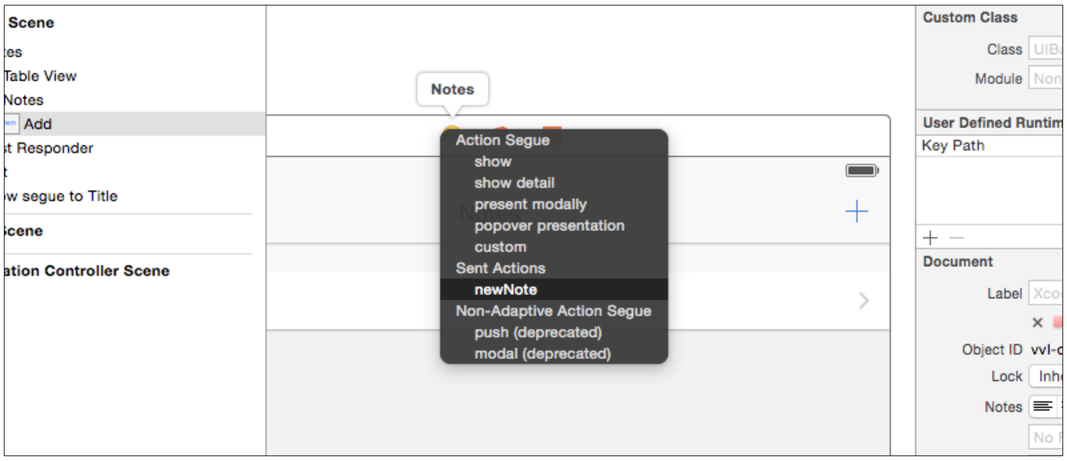
```
@IBAction func newNote() {

    //new dictionary with 2 keys and test values for both
    var newDict = ["title" : "TestTitle",
         "body" : "TestBody"]

    //add the dictionary to the front (or top) of the array
    arrNotes.insert(newDict, atIndex: 0)

    //reload the table ( refresh the view)
    self.tableView.reloadData()
}
```

But just because we've created the method doesn't mean it will work. We must link the button to the method in our storyboard.

Open `Main.storyboard`, and select the **+** button from the **Notes Scene**. Press *Ctrl* and drag from the **+** button to the yellow view controller icon (either on the left in the outline or above the view), and select **newNote** under the **Sent Actions** section.



When you let go of the mouse, the following menu will appear, asking you what you want to link the **+** button's button clicked event to. In this case, we want to link it to the **newNote** action that we created.

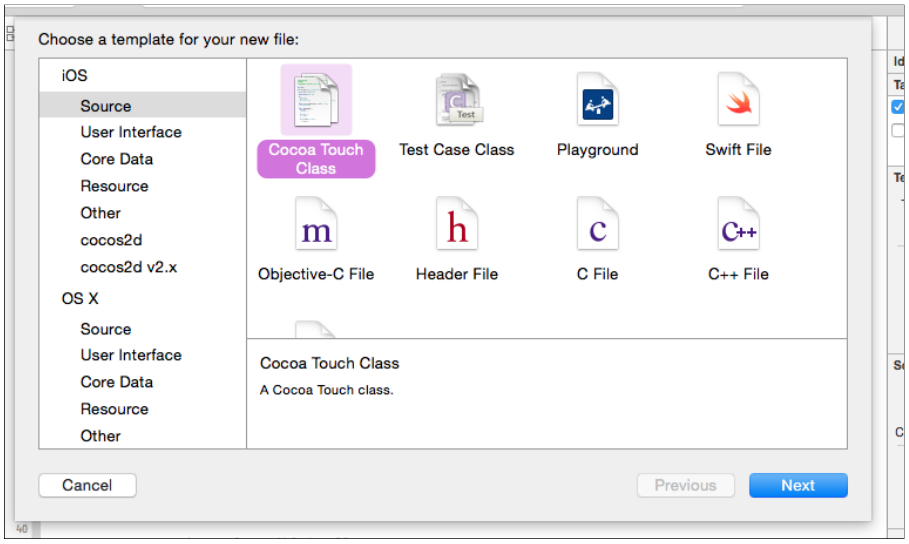


Now you should be able to run the app, and upon tapping the **+** button, you'll see an item get added to the table view every time. It looks like it's being added to the bottom of the list, but it's actually being added to the top and pushing everything else down.

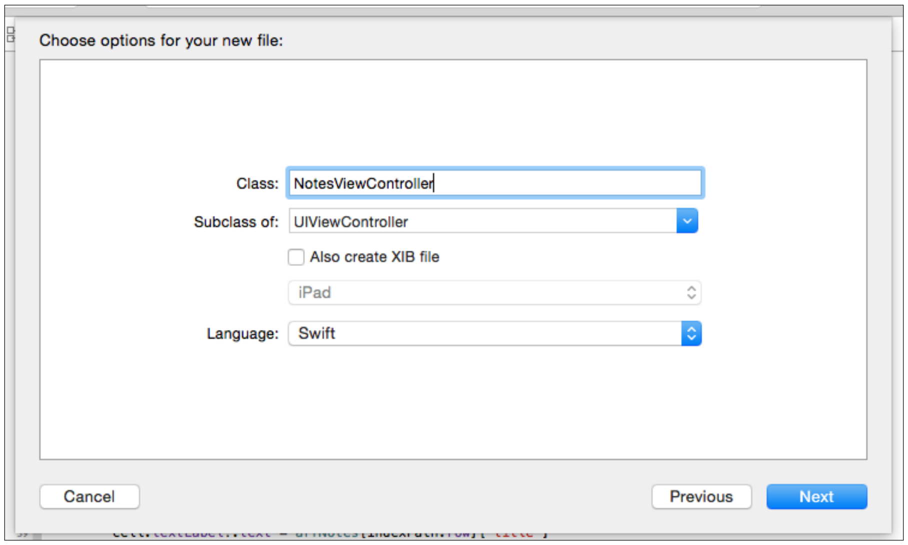
Create and link the notes editor class

We need to create a new class to modify the data in our editor view (the one with the textbox).

To do this, you need to right-click (or press *Ctrl* and click) on one of the files or folders in the project and select **New File**. From there, go to the **iOS Source** section, choose **Cocoa Touch Class**, and then click on **Next**.



We're going to name it `NotesViewController`, make it a subclass of `UIViewController`, and choose **Swift** as **Language**.



Choose options for your new file:

Class: NotesViewController

Subclass of: UIViewController

Also create XIB file

iPad

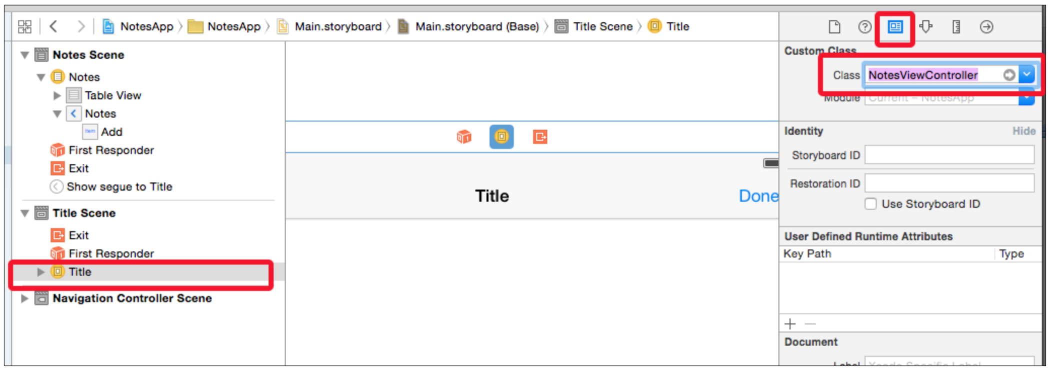
Language: Swift

Then click on **Create**, and it will take you to the file.

At the top of the `NotesViewController.swift` file, we're going to create an `IBOutlet` property for our `UITextView` object. The **!** mark at the end indicates that it's an optional type (similar to the **?** mark) and is implicitly unwrapped. In other words, you don't have to manually unwrap the variable to access it (but you would if it were defined as `UITextView!`):

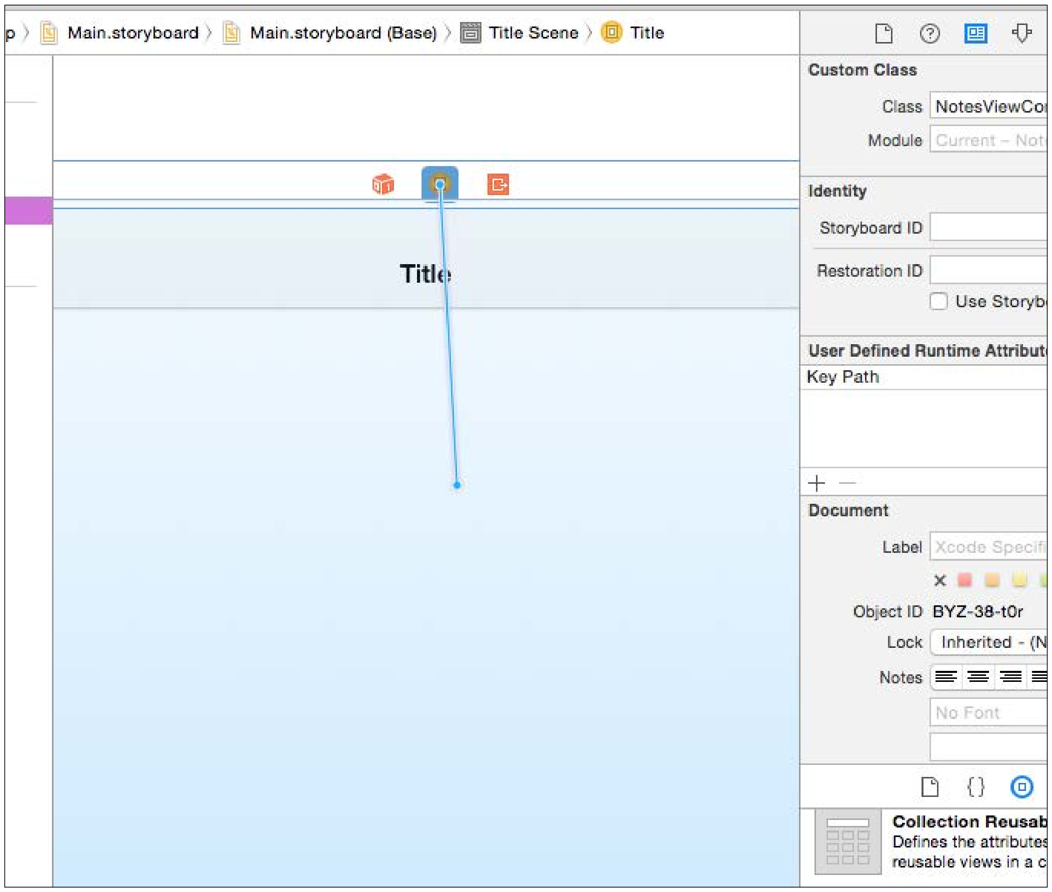
```
class NotesViewController: UIViewController {

    //a variable that links to the main body text view
    @IBOutlet weak var txtBody : UITextView!

    override func viewDidLoad() {
    ...
}
```

With the variable ready, we can go ahead and link the two together. In `Main.storyboard`, select the **Title Scene** (the scene with the textbox), and change its **Class** to **NotesViewController**.

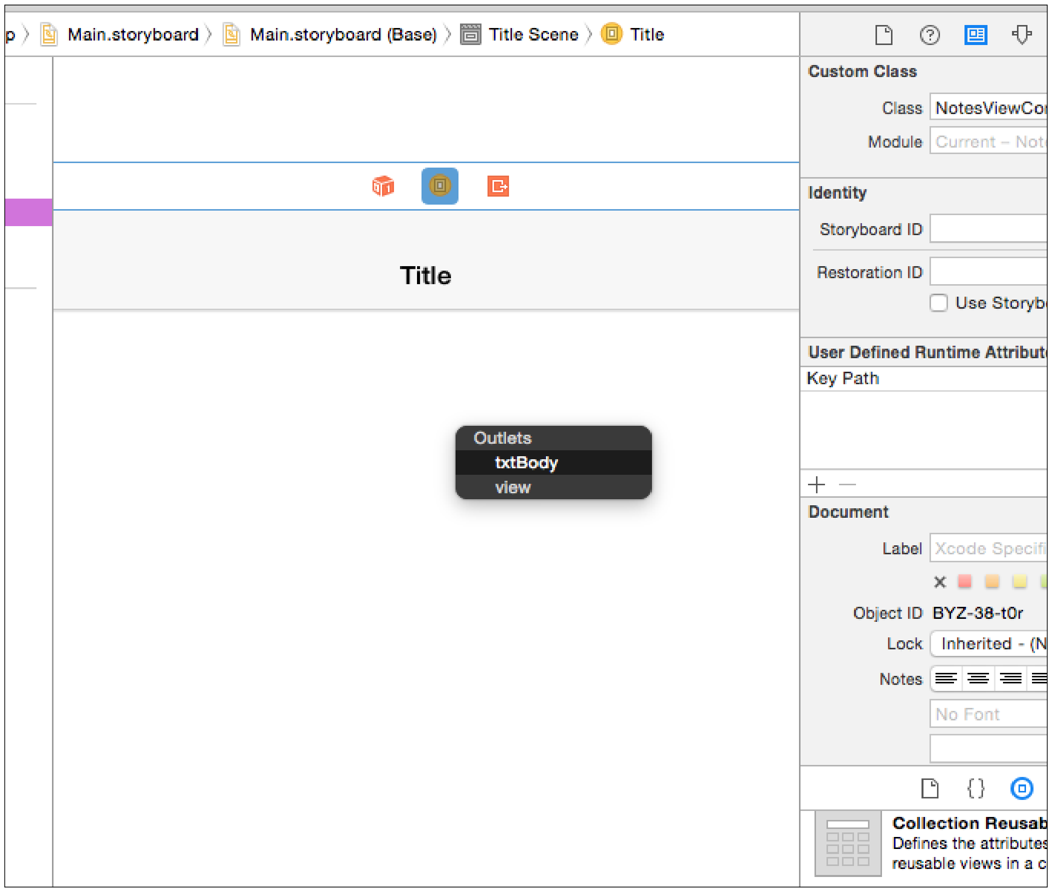


Class NotesViewController
Module Current - NotesApp

Then, with the view controller selected, press *Ctrl* and drag from the yellow icon to the text view.



Select the `txtBody` variable from the available options.



Transition to the notes editor

Now that we have our notes editor created and linked, let's transition to it using the segue identifier we created earlier. Because we have the identifier of the segue set, we can call that segue in the respective areas. We're calling the segue manually so that we can do some internal work before the segue happens.

In your `didSelectRowAtIndexPath` method as well as your `newNote` method, you need to call the `performSegue` method with the identifier you set up earlier:

```
override func tableView(tableView: UITableView,
  didSelectRowAtIndexPath indexPath: NSIndexPath) {

    //push the editor view using the predefined segue
    performSegueWithIdentifier("showEditorSegue", sender: nil)
}

@IBAction func newNote() {

    ...

    //reload the table (aka, refresh the view)
    self.tableView.reloadData()

    //push the editor view using the predefined segue
    performSegueWithIdentifier("showEditorSegue", sender: nil)
}
```

If you run the app at this point, you will be able to actually transition back and forth between the two views with no problem. It's just that there's no data being sent, so let's do that.

Modify the destination's title value

We first need to know which index is selected so that we can send the information to the `NotesViewController`. At the top of our `TableViewController` class, we add a variable to hold the selected index:

```
//selected index when transitioning (-1 as sentinel value)
var selectedIndex = -1

override func viewDidLoad() { ... }
```

Then, in the `tableView:didSelectRowAtIndexPath:` method, set the selected index value to the row that was selected:

```
override func tableView(tableView: UITableView,
  didSelectRowAtIndexPath indexPath: NSIndexPath) {

  //set the selected index before segue
  self.selectedIndex = indexPath.row


  ...
}
```

We're also going to add it to the `newNote` method, for when we tap the **+** button:

```
@IBAction func newNote() {


    ...

    //set the selected index to the most recently added item
    self.selectedIndex = 0

    //reload the table (refresh the view)
    self.tableView.reloadData()

    //push the editor view using the predefined segue
    performSegueWithIdentifier("showEditorSegue", sender: nil)
}
```

Now that the index is being set in the only two places that can change the index, let's actually perform our transition. If you're unfamiliar with navigation controllers, let me tell you that Apple provides a method called `prepareForSegue` that takes two parameters and allows us to modify the destination view controller before it appears on the screen.

That being said, add the `prepareForSegue` function to `TableViewController.swift`:

```
override func prepareForSegue(segue: UIStoryboardSegue, sender:
  AnyObject?) {

    //grab the view controller we're gong to transition to
    let notesEditorVC = segue.destinationViewController as
    NotesViewController
```

```
        //set the title of the navigation bar to the selectedIndex's title
        notesEditorVC.navigationItem.title =
        arrNotes[self.selectedIndex]["title"]
    }
```

The preceding method will set the title of our navigation bar to the `title` value in our dictionary at the same index that's being selected. For example, when the **+** button is tapped, we insert a new dictionary at index 0, select it, and set the title of the `NotesViewController` navigation bar to be the same as that of the newly created note.

If you run the app at this point and create a new note (or select a previously created note), you'll see that the title is now being shown as `TestTitle` (and not whatever you had before). But you also might be wondering why we aren't modifying the text view here either. Well, the way `prepareForSegue` works is that it can access the view controller just fine, but UI elements in our view controller have not been created yet, so we have to set a variable first.

Send the body text

Since we can't directly set the text view's text property in the `prepareForSegue` method, we must create a variable that will act as the "middleman" variable. Then, in the `viewDidLoad` method of `NotesViewController`, we can set the text accordingly.

So in `NotesViewController.swift`, add a string variable to hold the body text while the `UIViewController` loads. Then set the `txtBody` object's text property to the string value:

```
    //a string variable to hold the body text
    var strBodyText : String!

    override func viewDidLoad() {
        super.viewDidLoad()

        //set the body's text to the intermitent string
    self.txtBody.text = self.strBodyText
    }
```

Then, in `TableViewController.swift`, we want to set this variable in the `prepareForSegue` method:

```
    override func prepareForSegue(segue: UIStoryboardSegue, sender:
      AnyObject?) {
        ...

        //set the body of the view controller to the selectedIndex's
          body
        notesEditorVC.strBodyText =
          arrNotes[self.selectedIndex]["body"]
    }
```

Now if you run the app, you'll see that the body text's test value we assigned is being properly passed to `NotesViewController`.

Implement the Done button

If you recall, we added a navigation button with the word **Done**. Right now, it's not doing anything, so let's fix that. The functionality we want for the **Done** button is to hide the keyboard, but it will be visible only when the keyboard is also visible.

To make the keyboard visible automatically, we need to set the text view to become the first responder. This will enable editing on the text view, and the keyboard will pop up as soon as we transition to the `NotesViewController`.

In the `viewDidLoad` method, right after we set the text view's text to the string variable, we need to call the following:

```
//makes the keyboard appear immediately
self.txtBody.becomeFirstResponder()
```

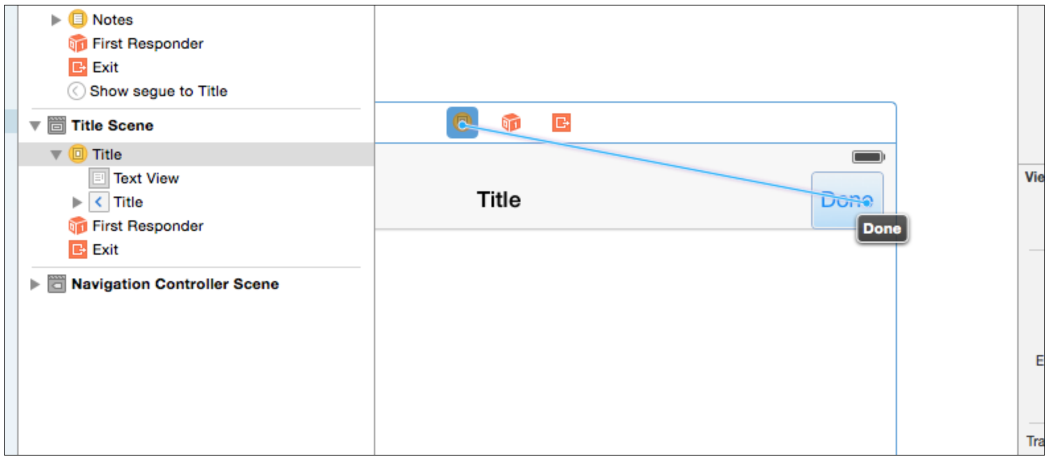
Now that the keyboard is showing, we need a way to hide it (as well as the **Done** button). So first, we need to create an `IBOulet` variable to store the button. At the top of `NotesViewController.swift`, add the following code:

```
//a variable to link the Done button
@IBOutlet weak var btnDoneEditing: UIBarButtonItem!
```

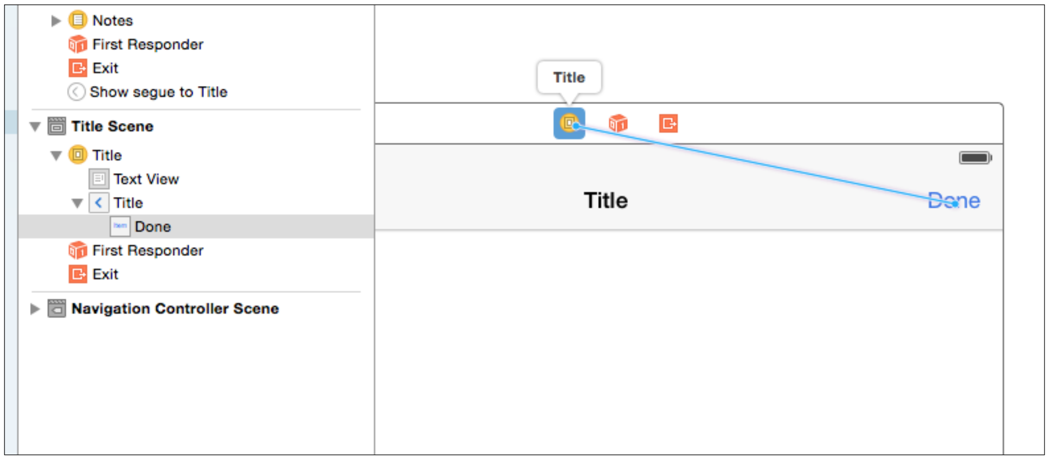
Also add the following method, which the **Done** button will call when tapped:

```
@IBAction func doneEditingBody() {

    //hides the keyboard
    self.txtBody.resignFirstResponder()

    //makes the button invisible (still allowed to be pressed, but
    that's okay for this app)
    self.btnDoneEditing.tintColor = UIColor.clearColor()
}
```

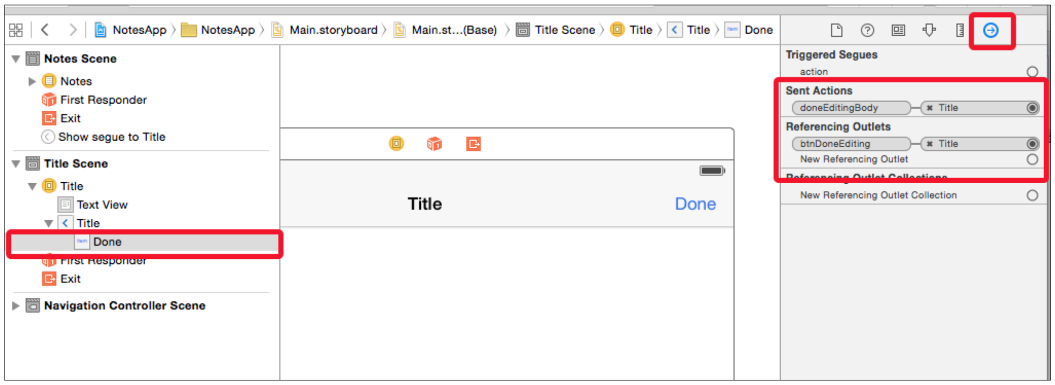
Now that you have a method for the **Done** button, open `Main.storyboard`, select the `NotesViewController` (**Title Scene** or whatever you called it), and link the button to the `IBOutlet` as well as the `IBAction`.



Again, make sure you link not only the view controller to the button (as shown in the preceding screenshot) but also the Done button's button clicked (touch up inside) event to the view controller (as shown in the following screenshot). Like the **+** button, we want to link our **Done** button to the `doneEditingBody` function we just created.



To verify that you've added both correctly, check the **Connections** tab of the **Done** button.



If you run the app at this point, you'll see the **Done** button and keyboard disappear as intended, but the **Done** button is never visible again whereas the keyboard appears again (but it does still work, which is fine for this sample app).

To fix this, we need to change the tint color of the **Done** button back to the default color when the text view begins editing. So in `NotesViewController.swift`, add the following method:

```
func textViewDidBeginEditing(textView: UITextView) {

    //sets the color of the Done button to the default blue
    //it's not a pre-defined value like clearColor, so we give it
      the exact RGB values
    self.btnDoneEditing.tintColor = UIColor(red: 0, green:
      122.0/255.0, blue: 1, alpha: 1)
}
```

But this method won't get called just yet. We need to tell the view controller that the text view's delegate methods (the preceding method is one of the delegate methods) should get called in this class.

To do that, add a comma, followed by the `UITextViewDelegate` keyword, after the class declaration:

```
class NotesViewController: UIViewController , UITextViewDelegate {
  ...
}
```

Then assign the delegate of the text view to `self`. You can do this by linking it either in the storyboard or in the `viewDidLoad` method, like this:

```
override func viewDidLoad() {

    ...

    //allows UITextView methods to be called (so we know when they
      begin editing again)
self.txtBody.delegate = self
}
```

Now you should be able to run the app and see the **Done** button working completely as intended. But still, the notes aren't being saved after they are modified.

Save the selected value

When the user is done editing their note, or wants to go back to the list of notes, they should notify the view controller that a change was made and the data in the table should be updated accordingly.

To do this, we're going to implement the protocol (or delegate) pattern.
If you're unfamiliar, it's exactly the same as the `UITextView` and the `UITableView` delegate methods that we've been implementing. But instead of using an Apple-defined method, we're going to create our own protocol, and have the `TableViewController` conform to that protocol.

So in `NotesViewController.swift`, add the following protocol to the top of the file:

```
//the protocol (or delegate) pattern, so we can update the table
  view's selected item
protocol NoteViewDelegate {

  //the name of the function that will be implemented
  func didUpdateNoteWithTitle(newTitle : String, andBody newBody :
    String)
}

class NotesViewController: UIViewController , UITextViewDelegate {
  ...
}
```

Now that we have a protocol to conform to, we need to know which class is going to be conforming to it. To know this information, we must create an optional variable (optional because of the **?** mark at the end, and optional here does not mean "not required"). So create a delegate variable in `NotesViewController.swift` that will determine what class to call the protocol's methods on:

```
class NotesViewController: UIViewController , UITextViewDelegate {

  //a variable to hold the delegate (so we can update the table
    view)
   var delegate : NoteViewDelegate?


   ...
}
```

With this delegate variable added, we can call the `didUpdateNoteWithTitle` method on the delegate class (but only if the delegate exists—is not equal to nil). So first in our **Done** button method, `doneEditingBody`, we're going to make a call to the method via the delegate, like this:

```
@IBAction func doneEditingBody() {


    ...


    //tell the main view controller that we're going to update the
      selected item
    //but only if the delegate is NOT nil
    if self.delegate != nil {

      self.delegate!.didUpdateNoteWithTitle( self.navigationItem.
title!, andBody: self.txtBody.text)

    }
}
```

Also, when the user hits the **Back** button (or in this case, **Notes**), we want to make a call to the `delegate` method. The easiest way to do this is by implementing the `viewWillDisappear` function in `NotesViewController`, as this is called only when the **Back** button is pressed:

```
override func viewWillDisappear(animated: Bool) {

    super.viewWillDisappear(animated)
```

```
        //tell the main view controller that we're going to update the
        selected item
        //but only if the delegate is NOT nil
        if self.delegate != nil {
          self.delegate!.didUpdateNoteWithTitle(
            self.navigationItem.title!, andBody: self.txtBody.text)
        }
    }
```

Before we can assign the delegate of the view controller, we need to tell `TableViewController` that it will be conforming to the `NoteViewDelegate`. So, in `TableViewController.swift`, set the delegate the same way as you set the text view:

```
    class TableViewController: UITableViewController ,
      NoteViewDelegate {
      ...
    }
```

It's going to give us an error, which is expected because we haven't added the protocol's required method, `didUpdateNoteWithTitle`, to the `TableViewController` class. Simply add that method, which will just modify the title and body of the selected index:

```
    func didUpdateNoteWithTitle(newTitle: String, andBody newBody:
      String) {

        //update the respective values
        self.arrNotes[self.selectedIndex]["title"] = newTitle
        self.arrNotes[self.selectedIndex]["body"] = newBody

        //refresh the view
        self.tableView.reloadData()
    }
```

Finally, we need to assign the delegate to the `TableViewController` in our `prepareForSegue` method:

```
    override func prepareForSegue(segue: UIStoryboardSegue, sender:
      AnyObject?) {

      ...

       //set the delegate to "self", so the method gets called here
    notesEditorVC.delegate = self
    }
```

Suppose you the run the app at this point. You'll see that when you edit a note, go back to the list of notes, and then tap on the note you recently edited, it shows the updated text.

In other words, it's properly saving the "note" portion of the note as intended.

But the title still isn't getting updated, so let's handle that.

Update the title accordingly

We want the title of our note to be the first set—everything from the first non-whitespace character up to the first newline character.

Since we're already setting the text view's delegate to the `NotesViewController`, all we have to do is add the following method to break the body into multiple sections separated by the newline character. Then it will go through each section, and the first section with text will be the title:

```
func textViewDidChange(textView: UITextView) {

    //separate the body into multiple sections
    let components = self.txtBody.text.componentsSeparatedByString("
\n")

    //reset the title to blank (in case there are no components
      with valid text)
    self.navigationItem.title = ""

    //loop through each item in the components array (each item is
      auto-detected as a String)
    for item in components {

      //if the number of letters in the item (AFTER getting rid of
        extra white space) is greater than 0...
      if countElements(item.stringByTrimmingCharactersInSet(NSCharact
erSet.
  whitespaceAndNewlineCharacterSet())) > 0 {

        //then set the title to the item itself, and break out of
          the for loop
        self.navigationItem.title = item
        break
      }
    }
}
```

Now run the app and watch the title change as you type!

However, when you first create a new note via the **+** button, it sets the title to something other than what's in the body, which is why it seems weird when it's first created.

So simply go to `TableViewController` and remove the test values (because we know it works now):

```
@IBAction func newNote() {
   //new dictionary with 2 keys and blank values for both
     var newDict = ["title" : "",
       "body" : ""]
  ...
}
```

Save and read the notes to and from the device

Although we have everything working as intended while the app is running, it's not a very good app unless it stores data in the device for use at a later date. Here, we're going to use `NSUserDefaults` as in the previous chapters throughout this book. Only this time, because it's Swift and not Objective-C, there's a slight syntax change when reading the value, but it's pretty straightforward.

So open `TableViewController` and add the following method that, when called, will save our notes array in the device using `NSUserDefaults`:

```
func saveNotesArray() {

    //save the newly updated array
    NSUserDefaults.standardUserDefaults().setObject(arrNotes,
      forKey: "notes")
    NSUserDefaults.standardUserDefaults().synchronize()
}
```

We're going to call this function twice—once in the `didUpdateNoteWithTitle` method and once in the `newNote` method:

```
func didUpdateNoteWithTitle(newTitle: String, andBody newBody:
  String) {

    ...
```

```
        //save the notes to the phone
        saveNotesArray()
    }

    @IBAction func newNote() {

        ...

        //save the notes to the phone
        saveNotesArray()

        //push the editor view using the predefined segue
        performSegueWithIdentifier("showEditorSegue", sender: nil)
    }
```

Although it might be saving on the phone, we have no idea whether it's working or not because we're not loading in any of the previously saved data when we relaunch the app. So, in the `viewDidLoad` function of `TableViewController`, we're going to read the array from `NSUserDefaults`. However, because the `arrayForKey` method returns `"AnyObject?"` as its type, we need to do something called **downcasting**, which is essentially attempting to convert the `AnyObject` type into the type we want. Downcasting doesn't always work, which is why we have the if-block to prevent crashes:

```
    override func viewDidLoad() {
        super.viewDidLoad()

        //read in the saved value. use "as?" to convert "AnyObject"
          (the type returned by NSUserDefaults) to the array of
          dictionaries
        //this is in an if-block so no "nil found" errors crash the
          app
        //this is known as downcasting
        if let newNotes = NSUserDefaults.standardUserDefaults().
          arrayForKey("notes") as? [[String:String]] {

          //set the instance variable to the newNotes variable
          arrNotes = newNotes

        }
    }
```

When you run the app at this point, everything will be in order. We're done here.

And that's it for the app

Congratulations! You've just made your first app using Swift. Not too bad, eh? It feels somewhat like Objective-C—only with a slightly different syntax — and it uses the same method and parameter names (which is good for those who are familiar with Objective-C and want to move on to a new language).

Summary

In this chapter, we created a simple nongame app using Swift and covered a lot of basic Swift elements such as optional, for-in loops, the delegate pattern, table views, and more.

As mentioned earlier, if you wish to learn more about Swift, there's a plethora of online resources available at your disposal, such as online courses, Apple-created content, as well as online communities that provide assistance as needed.