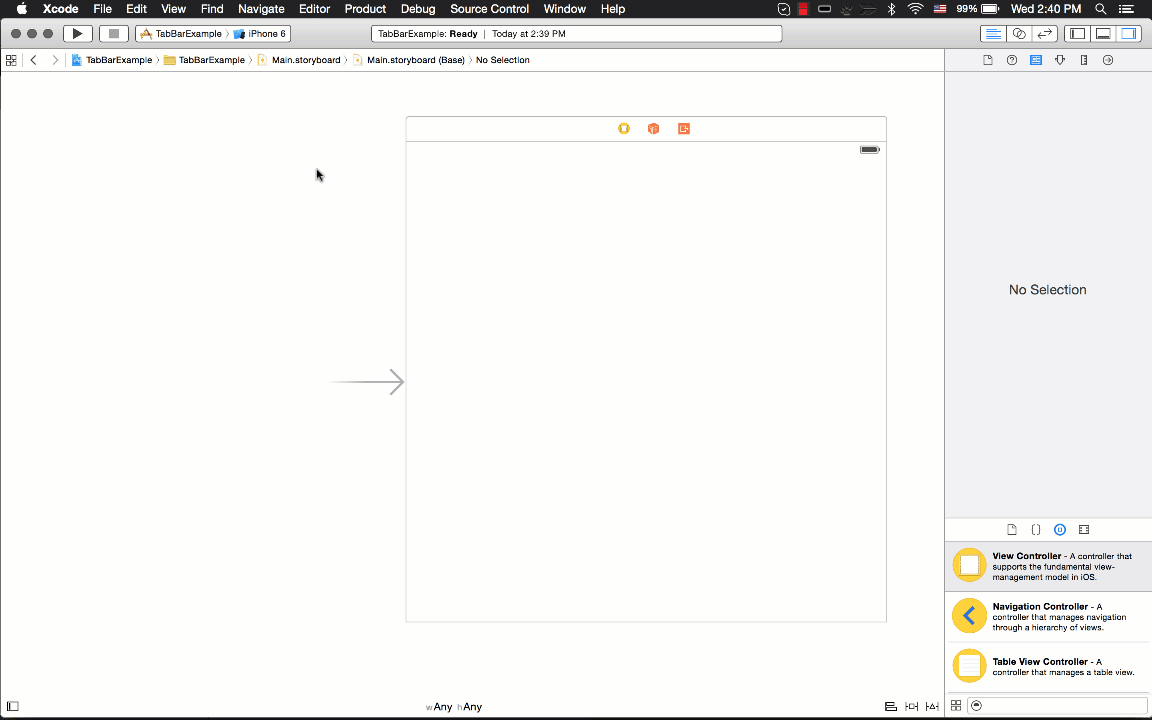
1. Human Interface Guidelines <https://developer.apple.com/design/human-interface-guidelines/>
2. Tab Bar Controller documentation <https://developer.apple.com/documentation/uikit/uitabbarcontroller>

**Tab Bar Controller**

There are other ways to get around an app too. One way is using a Tab Bar Controller. This is just a quick intro to tab bars, don't feel like you need to master them now.

Tab Bar Controller combined with the Navigation Controller gives our application that classic iOS feeling. It's amazing how Apple has made integrating these core features so fun. Like with many things in iOS... just control drag.



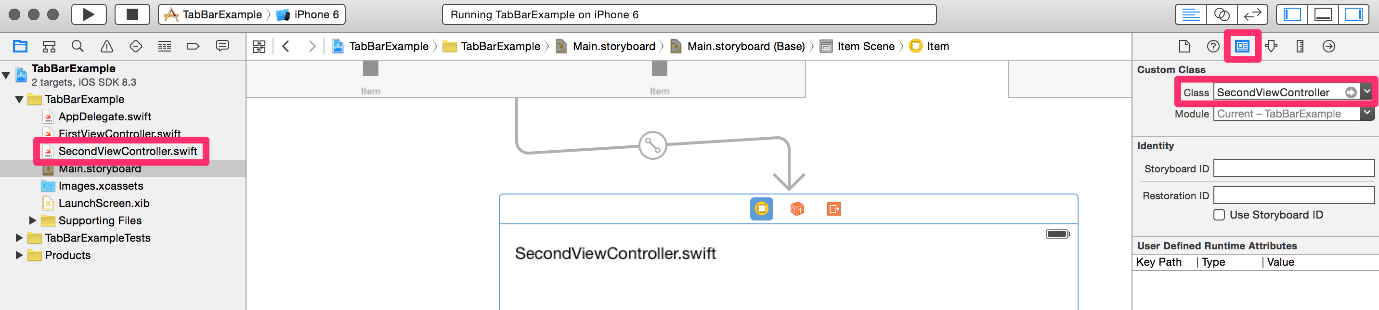
**Connecting our View Controllers**

Let's hook up our View Controllers into code. Currently, only one of the View Controllers has a code backing it (ViewController.swift). Naming a file ViewController.swift is so vague and can make things confusing sometimes. Let's have two View Controller files. **One will be called FirstViewController.swift that controls the left View Controller**(the one that shows up first when the application is launched) and the **other will be called SecondViewController.swift that controls the right View Controller.**

**FirstViewControllerصورة تحتوي على نص

تم إنشاء الوصف تلقائياً**

**SecondViewController**



**viewWillAppear**

Go ahead and add the following code and observe what gets logged as we click on the two different tabs.

**FirstViewController.swift**

import UIKit

class FirstViewController: UIViewController {

override func viewDidLoad() {

super.viewDidLoad()

print("FirstViewController viewDidLoad")

}

override func viewWillAppear(\_ animated: Bool) {

super.viewWillAppear(animated)

print("FirstViewController viewWillAppear")

}

}

**SecondViewController.swift**

import UIKit

class SecondViewController: UIViewController {

override func viewDidLoad() {

super.viewDidLoad()

print("SecondViewController viewDidLoad")

}

override func viewWillAppear(\_ animated: Bool) {

super.viewWillAppear(animated)

print("SecondViewController viewWillAppear")

}

}

copy

As we can see, the view controllers get loaded the first time we visit them. After that, they exist in their own view hierarchy and will not be loaded again.

# Table View Controller

First, let's read the description in Apple's  [UITableViewController Class Reference](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UITableViewController_Class/index.html" \t "_blank). In our previous application, we used a normal View Controller and dragged a Table View object onto it. To populate each cell **we first had to conform to the dataSource protocol by declaring it in the class declaration**, then **we had to implement the appropriate methods** and then we also had to **set the Table View's dataSource property as self**. To respond to events happening on the cell we had to do the same thing but this time, **we set the Table View's delegate property as self**then implemented didSelectRowAtIndexPath method.

**Most of the time we use a Table View Controller**instead of using the default View Controller if we are dealing with Table Views. **Table View Controller is optimized for using Table Views in a View Controller**. For example, it **already has set its tableView's dataSource and delegate as itself**. There are also other convenient methods that are useful for using Table Views. However, **we can only use UITableViewController when the Table View takes up the entire screen**. This is not much of a restriction because **most of the time this is exactly what we want to do.**

# Navigation Controller

Table View Controller is often used with a Navigation Controller. First, let's read the description in Apple's description of  [Navigation Controllers](https://developer.apple.com/library/ios/documentation/WindowsViews/Conceptual/ViewControllerCatalog/Chapters/NavigationControllers.html). Navigation controller serves as the navbar for your application. **There is usually a button on the top left and the top right to help you navigate through different View Controllers in your application**.

# Bucket List

### Setting Up The TableView Controller

### Navigation Controllers / Segues

NOTE: If the below video does not properly work, please use the following [**link**](https://vimeo.com/199587108/8889e0fa50).

We will be building an app similar to Beast List. **However, this time, we are going to utilize multiple scenes along with using the Table View Controller and the Navigation Controller**. First create a new project with the Single Page Application template.

## 1. Set up the Table View Controller

Delete the default View Controller in our storyboard. Then look for a Table View Controller from the Object Library and place it on the screen. **Then we have to specify that this is the initial View Controller that Xcode should load when the application first starts.**Our Single Page Application initially sets the empty View Controller as the initial View Controller, but Xcode no longer knows which View Controller to load because we deleted it.

صورة تحتوي على نص

تم إنشاء الوصف تلقائياً

**Now let's change our View Controller to have a more accurate name**. We should never name our View Controller ViewController unless we are testing something real quick. The name is too generic for anyone to understand what that View Controller is responsible for. **We are going to change the name of our file to BucketListViewController.swift and in our declaration, we will specify that we want BucketListViewController**. Also, instead of inheriting from a UIViewController, **we will be inheriting from UITableViewController which is a subclass of UIViewController.** **UITableViewController automatically has a Table View taking up the whole screen with its delegate and dataSource already set to the view controller. UITableViewController already specifies that it conforms to the dataSource and delegate protocol.**

صورة تحتوي على نص

تم إنشاء الوصف تلقائياً

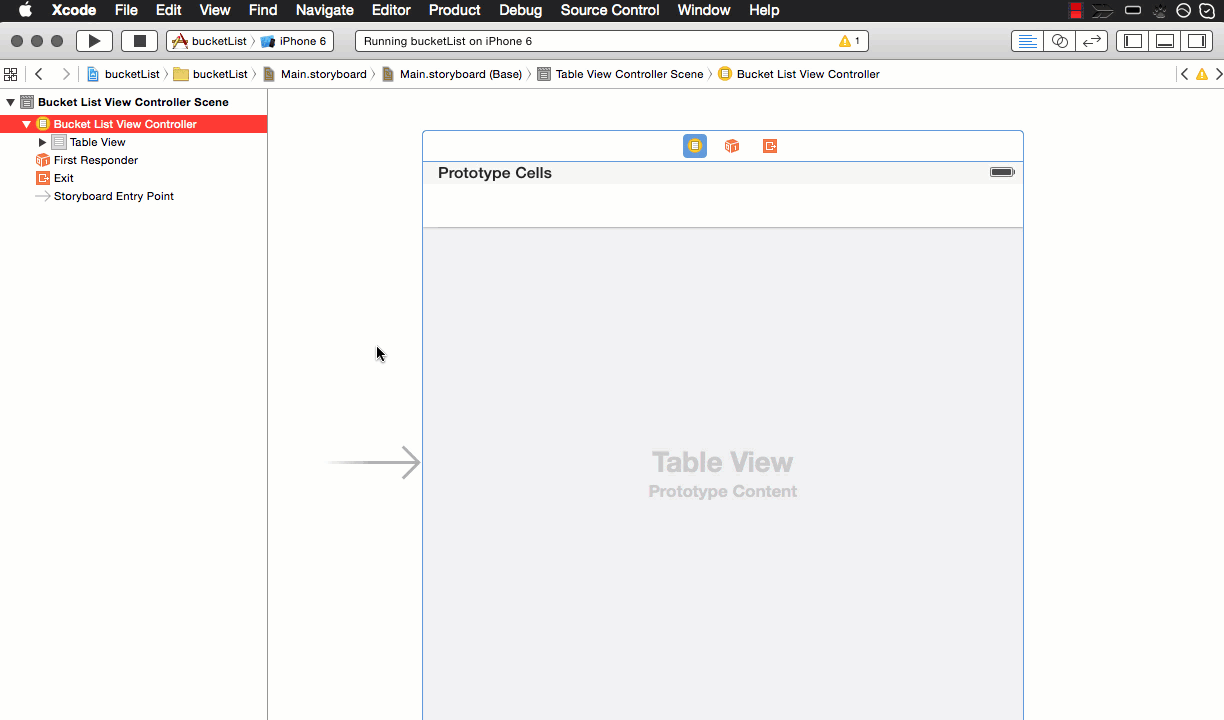
Now we have to hook up our code with the scene in the storyboard.

صورة تحتوي على نص

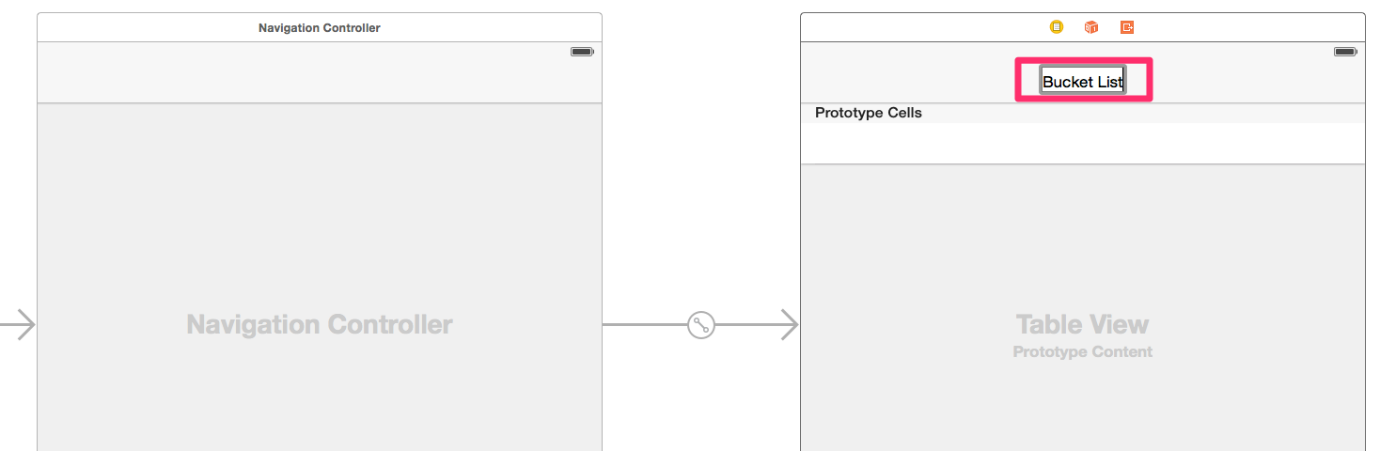
تم إنشاء الوصف تلقائياً

## 2. Embed the Navigation Controller

First go ahead and run the application. We should just see an empty Table View that takes up the first screen. Let's embed in a Navigation Controller because our application is going to have more than one scene.



Then double click on the navbar on top of the table to change the title.



## 3. Implement the Data Source protocol

We must implement the two required methods in the UITableViewDataSource Protocol. For now, we have an array of strings to serve as our Model object. We also have to make sure we set the reuse identifier of our prototype cell in the storyboard.

var items = ["Sky diving", "Live in Hawaii"]

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

return items.count

}

override func tableView(\_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {

// dequeue the cell from our storyboard

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

// All UITableViewCell objects have a build in textLabel so set it to the model that is corresponding to the row in array

cell.textLabel?.text = items[indexPath.row]

// return cell so that Table View knows what to draw in each row

return cell

}

We now have a working Bucket List application that looks very similar to our Beast List application but with some key differences. Let's recap:

1. We used a **UITableViewController**class along with a **Table View Controller.**
   * The **Table View Controller** has a Table View UI element as it's "Super View" vs the previous Beast List assignment where the "Super View" is a regular UIView and we added the Table View inside of that UIView. The Super View is the outermost UIView in the View Controller.
   * The UITableViewController class inherits from UIViewController and itself conforms to UITableViewDataSource and UITableViewDelegate so you do not have to explicitly conform to these in your own class
   * We had to specify that we were using a custom class in our User Interface to actually hook up the class to the UI.
2. We embedded a **Navigation Controller** into our Table View Controller. A Navigation Controller gives us access to the top NavBar and "controls" displaying and manipulating that top NavBar. In addition the Navigation Controller helps us segue between view controllers using the NavBar. We'll be adding a segue in this fashion in the next section
3. We conformed to the **UITableViewDataSource** protocol to help us display the data similar to how we did in the Beast List.

# Adding Items to the Bucket List

In our Bucket list application (unlike our Beast List App) we have no space to add a Bucket list item. A typical iOS user flow is to have a "+" button in the NavBar to add a particular item. This "+" button should segue to another view where we can add an item and then save it (which should bring us back to the main view controller).

## 1. Add the Bar Button

We are going to drag and drop a Bar Button from the Object Library to the top right section of our Navigation Controller. Then we are going to give it an identifier of 'Add' so that we get a + sign on our Bar Button.

## 2. Have the Bar Button segue to another View Controller

First, we are going to drag a new Table View Controller onto the storyboard. Then we are going to control drag from the + bar button to the new View Controller and click the show segue.

Go ahead and run the application. When we click on the + it segues into the new View Controller we created. **It automatically created a bar button in the top left for us to segue back into the scene that we came from.**

## 3. Connecting our new View Controller to Code

We are going to create a new Swift file to serve as the code behind the new TableViewController we added to our application. **The file name should match the name of the class and the class needs to inherit from UITableViewController.**

صورة تحتوي على نص

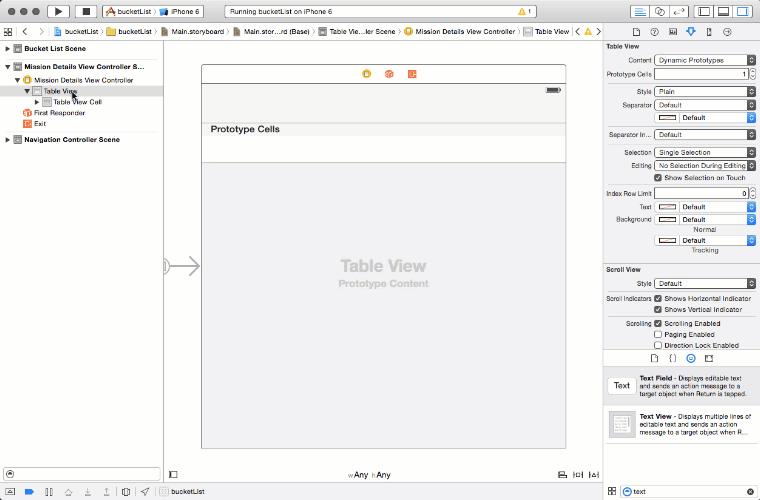
تم إنشاء الوصف تلقائياً

After we create the file we are going to connect it to the scene in our storyboard.

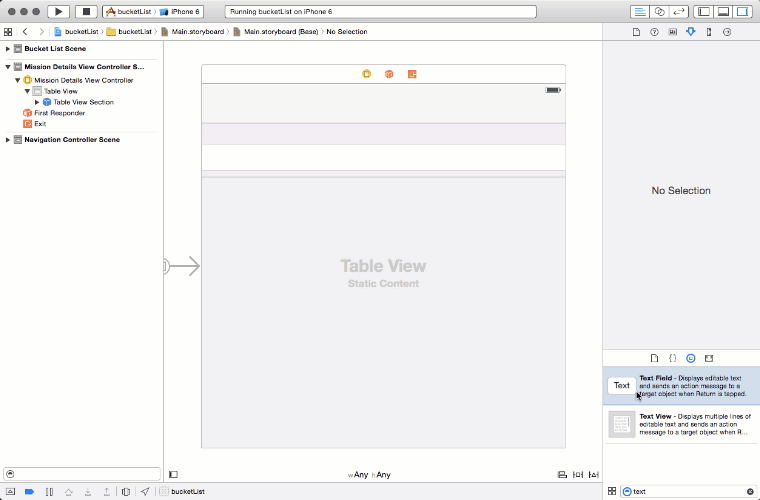
صورة تحتوي على نص

تم إنشاء الوصف تلقائياً

We will get an error in Xcode saying that we need to give a reusable identifier for the Prototype table cell in our new View Controller. In this scene, we only need one table row to enter in the new item. **This means that we are going to have static content.**Go ahead and make the following changes:



Then drag and drop the text input into the cell with the following constraints and attributes:



## 4. Embedding a new Navigation Controller

Let's say we want more control over what the Bar Button says in our new controller. **Since we will be adding a new item on this page, it might be clearer to the users if there is a cancel button on the top left and a done button on the top right**. Clicking on 'Cancel' seems clearer than clicking on 'BucketList' to cancel the item entry.

First, we are going to change our show segue to present modally. A show segue doesn't allow us to alter what it says in the top left bar button. **We are going to get rid of our show segue and do a modal segue instead.**

Now we are going to embed a new Navigation Controller.  **We still want a navbar on the top so we are going to embed in another Navigation Controller**. Then we are going to drag a Bar Button to the top left with the identifier of Cancel and another Bar Button to the top right with the identifier of Done.

### 5. Implementing the CancelButtonDelegate Protocol

### Delegation

Delegation is a common design pattern in iOS that involves having **one object perform actions "on behalf of, or in coordination with, another object".** A delegate is an object that conforms to a protocol which describes what methods it must implement which will then be invoked by the second object. Here are the key aspects of using Delegation:

* **Object A must conform to a protocol** that describes what methods it has that are relevant to Object B (the "delegate" protocol)
* **Object B must have a reference** to a type that conforms to the protocol (this will be a reference to Object A)
* **Object A must implement the protocol methods**
* **Object B will call upon Object A's protocol methods** through its reference

For our purposes we will be using Delegation to handle the interactions between our AddItemTableViewController and our BucketListViewController. Specifically we will be using the delegate to communicate specific events that occur on the AddItemTableViewController to the BucketListViewController (Cancel button pressed, Done button pressed). This way we can have the BucketListViewController handle the functionality for adding a new list item (since the array of list items already exists in the BucketListViewController).

Currently, when we press the Cancel button, nothing happens. In fact, if our users get to this page, they are trapped inside. **We want the presenting View Controller to get dismissed when the user pressed the Cancel button**. We are going to use the Delegation design pattern to accomplish this.

Go ahead and create a new Swift file with the following protocol. We'll call the protocol "CancelButtonDelegate" because the protocol will help distinguish any View Controller that can be a CancelButtonDelegate and handle the action of a Cancel bar button being pressed. Remember that we are creating this protocol from scratch, we can name the functions anything we want, and can pass any parameters we need.

import UIKit

protocol CancelButtonDelegate: class {

func cancelButtonPressed(by controller: UIViewController)

}

This protocol defines that any class that conforms to CancelButtonDelegate must implement the cancelButtonPressed(by controller: UIViewController) -> () method.

Now which View Controller needs to conform to this protocol? Since the cancel button is being pressed from the MissionDetailsViewController we would want the BucketListViewController to be its CancelButtonDelegate so that the BucketListViewController can handle dismissing the MissionDetailsViewController.

So now when we are segueing into the AddItemTableViewController, **we are going to set the BucketListViewController to be its delegate**. To do this, we must first declare that we are going to be conforming to the CancelButtonDelegate Protocol and then implement the required methods for this protocol.

class BucketListViewController: UITableViewController, CancelButtonDelegate {

// ...

func cancelButtonPressed(by controller: UIViewController) {

dismiss(animated: true, completion: nil)

}

// ...

}

Now that BucketListViewController conforms to CancelButtonDelegate, **we need to add a property in our AddItemViewController so that BucketListViewController can set itself as the cancelButtonDelegate**. Also, control drag from the Cancel button to create an Action that will get triggered when a user clicks on it. Once the user clicks on the button we want to send a message cancelButtonPressed to our cancelButtonDelegate...if we have one.

import UIKit

class AddItemTableViewController: UITableViewController {

// ...

weak var delegate: CancelButtonDelegate?

@IBAction func cancelBarButtonPressed(\_ sender: UIBarButtonItem) {

delegate?.cancelButtonPressed(by: self)

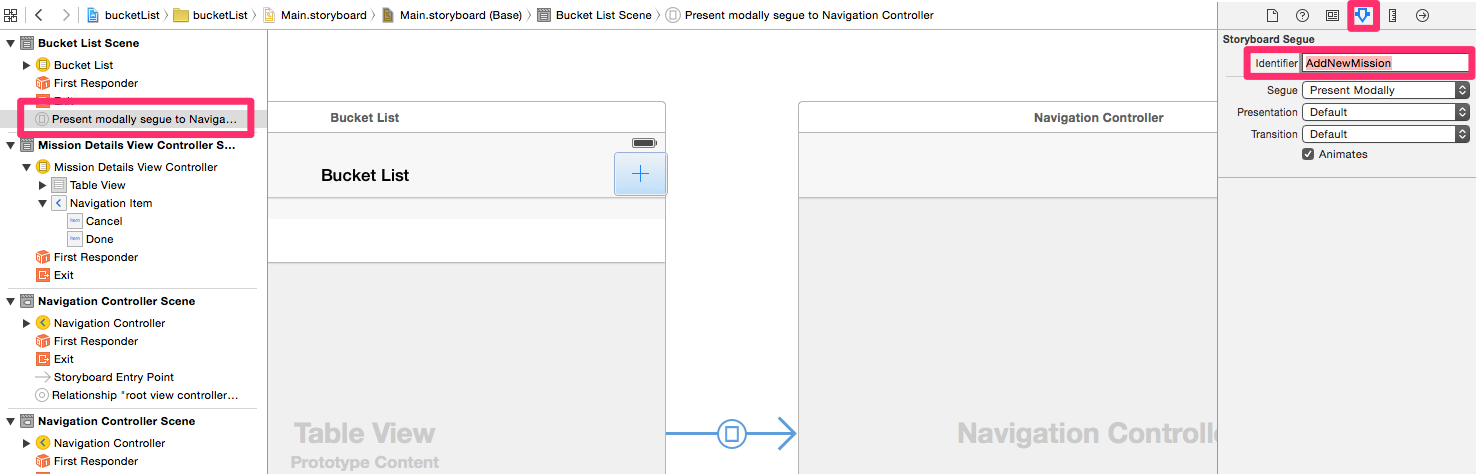
}

// ...

}

Now in the AddItemTableViewController we have an action for the Cancel Bar Button and inside of that action, we call the delegate's cancelButtonPressed method (passing self as the UIViewController). Note that we have a **weak var cancelButtonDelegate** which is of the "CancelButtonDelegate" type. We now need to make sure that **when we segue to the AddItemViewController we set this property to point to the BucketListViewController.**

We have to first give our modal segue an identifier.



Now when we are segueing into the AddItemViewController, we are going to set ourselves as its cancelButtonDelegate.

class BucketListViewController: UITableViewController, CancelButtonDelegate {

// ...

override func prepare(for segue: UIStoryboardSegue, sender: Any?) {

if segue.identifier == "AddNewMission" {

let navigationController = segue.destination as! UINavigationController

let controller = navigationController.topViewController as! AddItemTableViewController

controller.delegate = self

}

}

// ...

}

The prepareForSegue method above is a built in method that runs when a segue is triggered and before the segue actually occurs. We will be using this method a lot in the coming chapters -- its value comes from the fact that **from within the method we can reference both the ViewController that we are segueing from and the ViewController that we are segueing to.**

Now when we run the application, **we can safely navigate backward when a user presses the Cancel button.**

### CancelButtonDelegate Recap

1. We **defined the CancelButtonDelegate Protocol**
   * The protocol has one method -- cancelButtonPressed(by controller: UIViewController) -> ()
2. We **made the BucketListViewController conform to the protocol.**  
   * The implementation of cancelButtonPressed simply dismisses the top ViewController
3. We **gave the AddItemTableViewController a reference to a type that conforms to the CancelButtonDelegate protocol** and defined where we were going to use the cancelButtonDelegate (when the cancel bar button action is triggered).
4. We **assigned the reference from AddItemTableViewController to the BucketListViewController in the prepareForSegue method**: "controller.cancelButtonDelegate = self"

## 6. Implementing the AddItemTableViewControllerDelegate Protocol

Now that we can cancel when we are on the AddItemTableViewController page let's implement the functionality to actually add the new Item!

To do this we are going to implement another protocol for when the Done button is pressed. When the Done button is pressed we want to grab what is in the text label and send it to our delegate. Then our delegate will append it to its Array of Strings.

Create a new Swift file with the following Protocol definition in it:

import UIKit

protocol AddItemTableViewControllerDelegate: class {

func addItemViewController(\_ controller: AddItemTableViewController, didFinishAddingItem item: String)

func addItemViewController(\_ controller: AddItemTableViewController, didPressCancelButton button: UIBarButtonItem) // Taken from CancelButtonDelegate file, and altered to match pattern.

// NOTE: You will need to update AddItemTableViewController to make the Cancel Button work

}

Notice that we called the function inside of AddItemTableViewControllerDelegate "addItemTableViewController". This is a convention often used in the iOS framework (You may have seen the "tableView" methods that are all different because of their parameters).

First let's modify the AddItemTableViewController. **We want it to have a property called addItemTableViewControllerDelegate so that it can be set later on similar to how the cancelButtonDelegate was set**. We also want to invoke our delegate's method passing along what is in the Text Field once the user presses "Done." This means that we need to have an Outlet to the UITextField as well. AddItemTableViewController should look like this afterwards:

class AddItemTableViewController: UITableViewController {

weak var delegate: AddItemTableViewControllerDelegate?

@IBOutlet weak var itemTextField: UITextField!

@IBAction func cancelBarButtonPressed(\_ sender: UIBarButtonItem) {

delegate?.addItemViewController(controller: self, didPressCancelButton: sender)

}

@IBAction func doneBarButtonPressed(\_ sender: UIBarButtonItem) {

delegate?.addItemViewController(controller: self, didFinishAddingItem: itemTextField.text!)

}

}

Now let's declare that our BucketListViewController conforms to this protocol and also implement the necessary function. Here we are going to append the item that we get to our items array and then reload our table view.

import UIKit

class BucketListViewController: UITableViewController, CancelButtonDelegate, AddItemViewControllerDelegate {

// ...

func addItemViewController(controller: AddItemViewController, didFinishAddingItem item: String) {

dismiss(animated: true, completion: nil)

items.append(item)

tableView.reloadData()

}

// ...

}

### AddItemViewControllerDelegate Recap

1. We **defined the AddItemViewControllerDelegate Protocol**
   * The protocol has one method -- addItemViewController(controller: AddItemViewController, didFinishAddingItem: String) -> ()
2. We **gave the AddItemViewController a reference to a type that conforms to the AddItemViewControllerDelegate protocol** and defined where we were going to use the delegate (when the done bar button action is pressed).
3. We **made the BucketListViewController conform to the protocol.**  
   * The implementation of the didFinishAddingItem method saves the item, that was passed through the method and dismisses the AddItemViewController. It also reloads the tableView data so that we can see the new item!
4. We **assigned the reference from AddItemViewController to the BucketListViewController in the prepareForSegue method**: "controller.delegate = self"