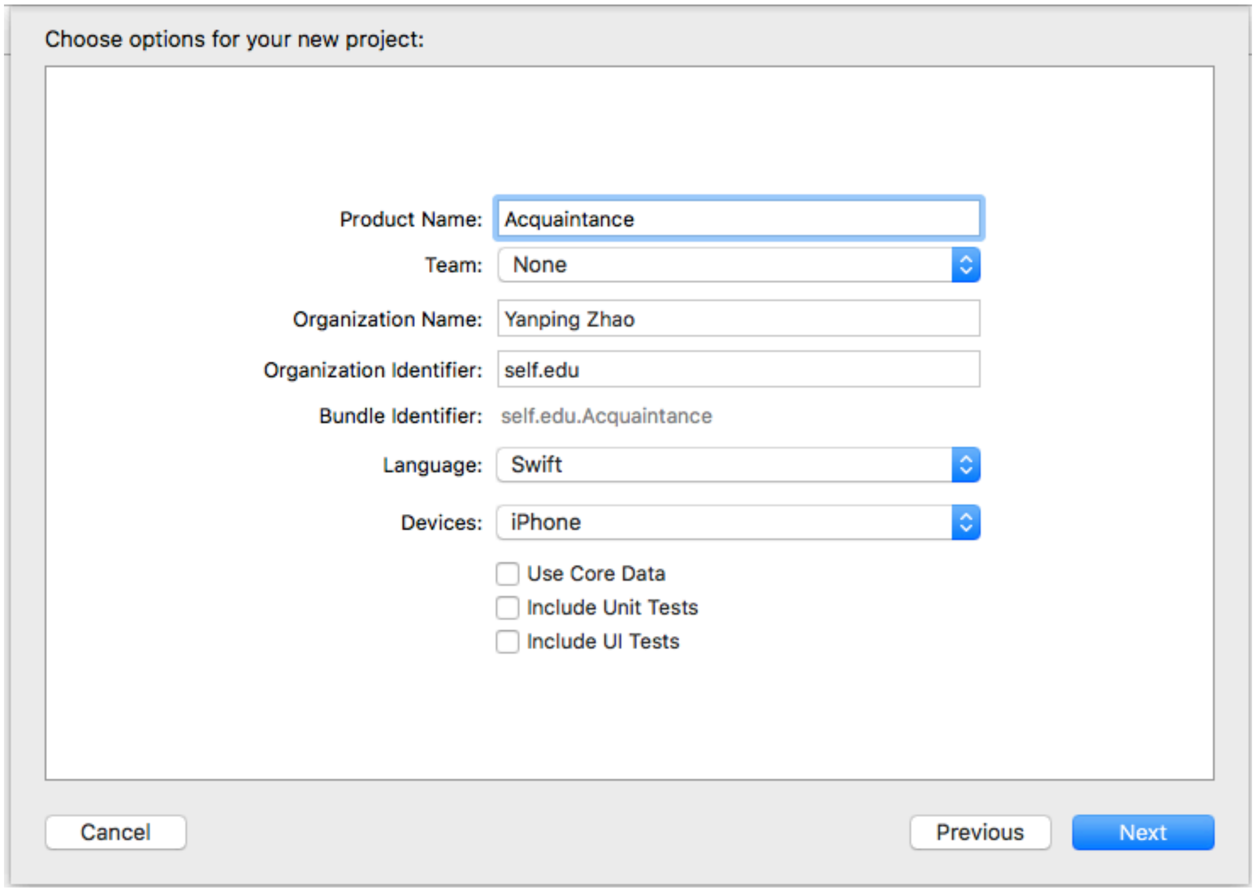
“Acquaintance” — a Table View Based App

“Acquaintance” is a simple app that keeps track of the acquaintance a user knows. When the app launches, it displays a list of acquaintance with summary information, including a name, a photo, and a short memo. The user can add a new acquaintance, delete a acquaintance, or update the information of an existing acquaintance.

Creating a Blank Table View

To get started, create a “Single View” application from the Xcode template and fill out the project information as follows:



Choose options for your new project:

Product Name:	Acquaintance
Team:	None
Organization Name:	Yanping Zhao
Organization Identifier:	self.edu
Bundle Identifier:	self.edu.Acquaintance
Language:	Swift
Devices:	iPhone

- [ ] Use Core Data
- [ ] Include Unit Tests
- [ ] Include UI Tests

The template app’s interface displays a plain view. We are going to replace it with a table view. Open “Main.storyboard” in the Interface Builder, select the “view controller scene” and delete it. Next, drag a “Table View Controller” from the Object Library to the storyboard.

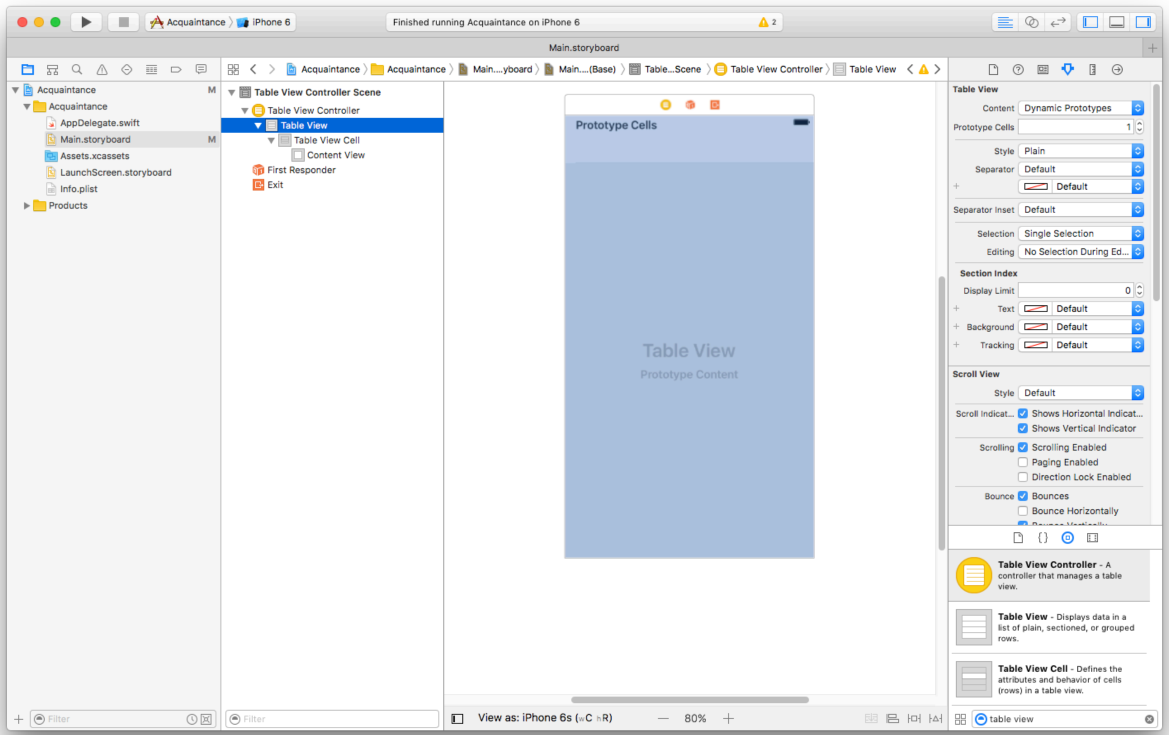


Table View
Content: Dynamic Prototypes
Prototype Cells: 1
Style: Plain
Separator: Default
Default
Separator Inset: Default
Selection: Single Selection
Editing: No Selection During Ed...
Section Index
Display Limit: 0
Text: Default
Background: Default
Tracking: Default
Scroll View
Style: Default
Scroll Indicat...: ✓ Shows Horizontal Indicat... ✓ Shows Vertical Indicator
Scrolling: ✓ Scrolling Enabled ☐ Paging Enabled ☐ Direction Lock Enabled
Bounce: ✓ Bounces ☐ Bounce Horizontally

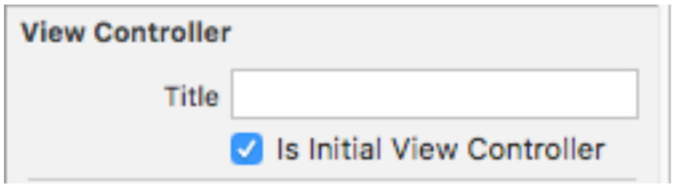
Table View Controller - A controller that manages a table view.

Table View - Displays data in a list of plain, sectioned, or grouped rows.

Table View Cell - Defines the attributes and behavior of cells (rows) in a table view.

table view

In the Document Outline pane, select "Table View Controller", and then in the Attributes Inspector, check "Is Initial View Controller".



View Controller
Title:
✓ Is Initial View Controller

In the Document Outline pane, select "Table View Cell", and then in the Attributes Inspector, change Style to "Subtitle" and set the Identifier to "ListCell". A "Subtitle" style table view cell contains an image and two text labels.

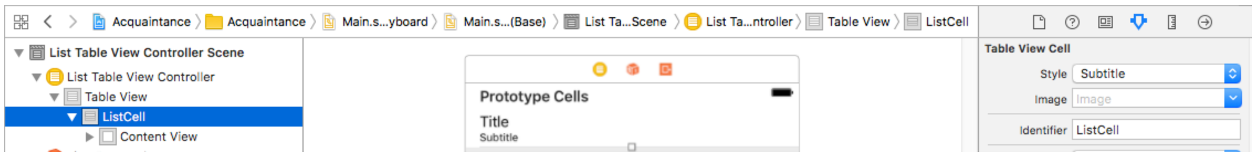
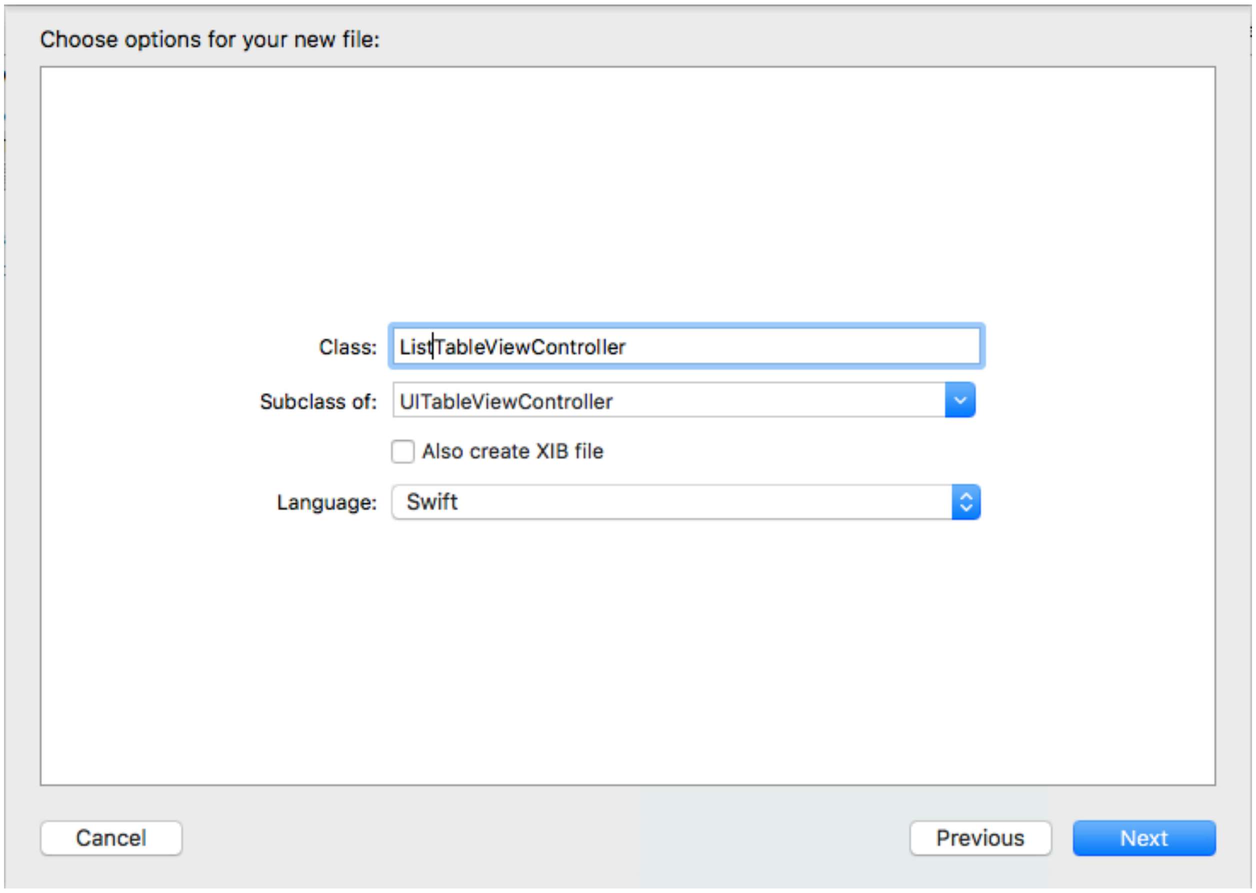


Table View Cell
Style: Subtitle
Image: Image
Identifier: ListCell

Next, in the "Project Navigator" pane, select ViewController.swift and delete it. From File->New->File…, select an iOS Source file of type "Cocoa Touch Class", and then choose the following options for this new file.


Choose options for your new file:

Class: ListTableViewController

Subclass of: UITableViewController

☐ Also create XIB file

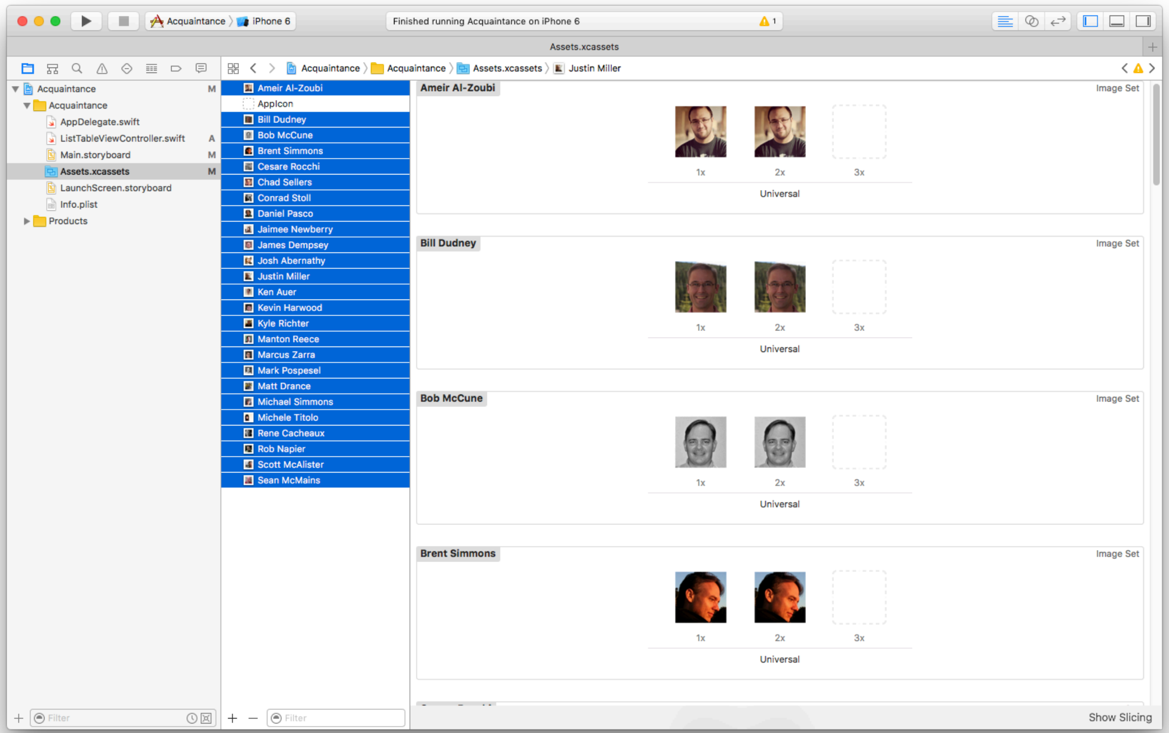
Language: Swift

Switch to the Interface Builder, select “Table View Controller”. In the “Identity Inspector”, change the Class name to ListTableViewController.

Build and run the app, it should show a blank table view.

Displaying a List of Acquaintance in the Table View

Download the sample photos. In the Project Navigator pane, select Assets.xcassets to open the asset catalog. Drag all the photos into the asset catalog.



Open *ListTableViewController.swift*, add an array instance variable *acqList*. This array contains a list of names.

```
var acqList = ["Ameir Al-Zoubi", "Bill Dudney", "Bob McCune", "Brent Simmons", "Cesare Rocchi", "Chad
Sellers", "Conrad Stoll", "Daniel Pasco", "Jaimee Newberry", "James Dempsey", "Josh Abernathy", "Justin
Miller", "Ken Auer", "Kevin Harwood", "Kyle Richter", "Manton Reece", "Marcus Zarra", "Mark Pospesel", "Matt
Drance", "Michael Simmons", "Michele Titolo", "Michael Simmons", "Rene Cacheaux", "Rob Napier", "Scott
McAlister", "Sean McMains"]
```

Next, start to work on the *UITableViewDataSource* protocol. First, remove the following optional method, since the number of sections in this table view is 1.

```
override func numberOfSections(in tableView: UITableView) -> Int {
        // #warning Incomplete implementation, return the number of sections
        return 0
    }
```

Secondly, provide the number of rows in the section, which equals to the number of items in the array "names".

```
override func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
        // #warning Incomplete implementation, return the number of rows
        return acqList.count
    }
```

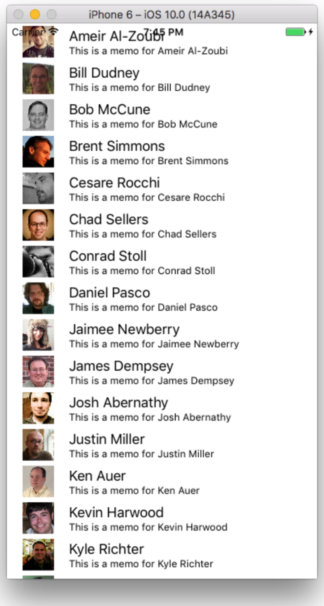
Lastly, configure the cells in the table view. This method *tableView(_:cellForRowAt)* is called every time a row is displayed.

```
override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
        let cell = tableView.dequeueReusableCell(withIdentifier: "ListCell", for: indexPath)

        // Configure the cell...
        cell.imageView?.image = UIImage(named: acqList[indexPath.row])
        cell.textLabel?.text = acqList[indexPath.row]
        cell.detailTextLabel?.text = "This is a memo for " + acqList[indexPath.row]

        return cell
    }
```

In the above method, the *IndexPath* is a structure that contains the section and the row index of a cell. Since this table view has only one section, the section index is always 0. We can use *indexPath.row* as an index to retrieve the name string from the *names* array and display it in the table view cell at the position *indexPath*.

In iOS, due to resource limitations and performance costs, a table view only allocates a certain number of cell objects and reuses them, since only a limited number of cells are visible on the screen at any moment. The *dequeueReusableCell* method is used to retrieve a reusable table cell from the recycle pool with the specified cell identifier. This "cell identifier" in code must be the same as the cell identifier specified in the Interface Builder.

Build and run the app. It should display a list of people, each with a photo, a name, and a short memo string.



Ameir Al-Zoubi
This is a memo for Ameir Al-Zoubi
Bill Dudney
This is a memo for Bill Dudney
Bob McCune
This is a memo for Bob McCune
Brent Simmons
This is a memo for Brent Simmons
Cesare Rocchi
This is a memo for Cesare Rocchi
Chad Sellers
This is a memo for Chad Sellers
Conrad Stoll
This is a memo for Conrad Stoll
Daniel Pasco
This is a memo for Daniel Pasco
Jaimee Newberry
This is a memo for Jaimee Newberry
James Dempsey
This is a memo for James Dempsey
Josh Abernathy
This is a memo for Josh Abernathy
Justin Miller
This is a memo for Justin Miller
Ken Auer
This is a memo for Ken Auer
Kevin Harwood
This is a memo for Kevin Harwood
Kyle Richter
This is a memo for Kyle Richter

Creating a Model Class

iOS apps always follow the Model-View-Controller design pattern. In this app, we are going to create a model class to store the information about an acquaintance.

In Xcode menu, select File->New->File…, and then choose the “Swift File” template under the “iOS” tab. Name this file “Person.swift”. This *Person* will be our model class.

Open *Person.swift*, define the class *Person* as follows:

```
import Foundation
import UIKit

class Person: NSObject {
    var name: String
    var photo: UIImage?
    var notes: String?

    init?(name: String, photo: UIImage?, notes: String?) {
        if name.isEmpty {
            return nil
        }

        self.name = name
        self.photo = photo
        self.notes = notes

        super.init()
    }

    convenience init?(_ name: String) {
        self.init(name: name,
                  photo: nil,
                  notes: nil)
    }
}
```

This *Person* class contains 3 stored properties. The property *name* is of type *String,* since every *Person* objects must have a name. The *photo* property is an optional since a person may not have a picture. The *notes* property is also an optional.

This *Person* class also defines a designated initializer and a convenience initializer. The convenience initializer takes in the name of a person and sets the photo and notes as nil.

After defining the *Person* class, we then re-define the *acqList* array as an array of the *Person* objects rather than an array of names in *ListTableViewController.swift*.

```
var acqList = [Person("Ameir Al-Zoubi"),  Person("Bill Dudney"), Person("Bob McCune"), Person("Brent
Simmons"), Person("Cesare Rocchi"), Person("Chad Sellers"), Person("Conrad Stoll"), Person("Daniel Pasco"),
Person("Jaimee Newberry"), Person("James Dempsey"), Person("Josh Abernathy"), Person("Justin Miller"),
Person("Ken Auer"), Person("Kevin Harwood"), Person("Kyle Richter"), Person("Manton Reece"),
Person("Marcus Zarra"), Person("Mark Pospesel"), Person("Matt Drance"), Person("Michael Simmons"),
Person("Michele Titolo"), Person("Michael Simmons"), Person("Rene Cacheaux"), Person("Rob Napier"),
Person("Scott McAlister"), Person("Sean McMains")]
```

Then, modify the method *tableView(_:cellForRowAt)* to configure the table view cells with the new *acqList* array. Note that if a *Person* object does not contain a photo, a default photo “photoalbum” will be shown. You need to add this photo to the asset catalog before using it.

```
override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
        let cell = tableView.dequeueReusableCell(withIdentifier: "ListCell", for: indexPath)

        let person = acqList[indexPath.row]

        // Configure the cell...
        if let photo = person.photo {
            cell.imageView?.image = photo
        } else {
            cell.imageView?.image = UIImage(named:"photoalbum")
        }
        cell.textLabel?.text = person.name
        cell.detailTextLabel?.text = person.notes

        return cell
    }
```

Also modify the *ViewDidLoad()* method to set the photos and notes for every object in the *Person* array:

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

        // Uncomment the following line to preserve selection between presentations
        // self.clearsSelectionOnViewWillAppear = false

        // Uncomment the following line to display an Edit button in the navigation bar for this view
controller.
        // self.navigationItem.rightBarButtonItem = self.editButtonItem()
        for person in acqList {
            if let name = person?.name {
                person?.photo = UIImage(named: name)
                person?.notes = "This is a memo for " + name
            }
        }
    }
```

Build and run the app. It should show a list of people, each with a photo and a name.

Deleting a Row

In iOS app, users normally swipe horizontally across a table row to reveal the Delete button. The method *tableView(_:commit:forRowAt:)* of the *UITableViewDataSource* protocol handles the deletion (or insertion) of a specific row.

To enable the swipe-to-delete feature of a table view, we just need to implement that method. In *ListTableViewController.swift*, add the following method:

```
// Override to support editing the table view.
override func tableView(_ tableView: UITableView, commit editingStyle: UITableViewCellEditingStyle,
forRowAt indexPath: IndexPath) {
     }
  }
```

Build and run the app. Even though this method does not contain any implementation, iOS detects its existence and will automatically displays a *Delete* button when the user swipes across a row. When the user taps the *Delete* button, we need to do two things: 1) remove the person from the data array *acqList;* 2) remove the row from the table view. So update the method with the code below:

```
// Override to support editing the table view.
  override func tableView(_ tableView: UITableView, commit editingStyle: UITableViewCellEditingStyle,
forRowAt indexPath: IndexPath) {
      if editingStyle == .delete {
         // Delete the row from the data source
         acqList.remove(at: indexPath.row)
         tableView.deleteRows(at: [indexPath], with: .fade)
      } else if editingStyle == .insert {
         // Create a new instance of the appropriate class, insert it into the array, and add a new row to the
table view
      }
  }
```

Build and the run the app. The Swipe-to-delete function should work as expected.

Creating a Table View to Display Detailed Information

The list table view displays a list of acquaintance with summary information. When the user taps a row, we would want the app to show a new page with detailed information about this acquaintance, e.g., name, email, phone number, etc. The user should also be able to modify those information on that page.

In this app, we will use a static table view to display detailed information. To get started, open *Main.storyboard* and add a new “Table View Controller” to the storyboard. In the Document Outline pane, select the “Table View” of the new table view controller. In the Attributes Inspector, choose the “Content” as “Static Cells”, “Sections” as 3, and “Style” as “Grouped”.

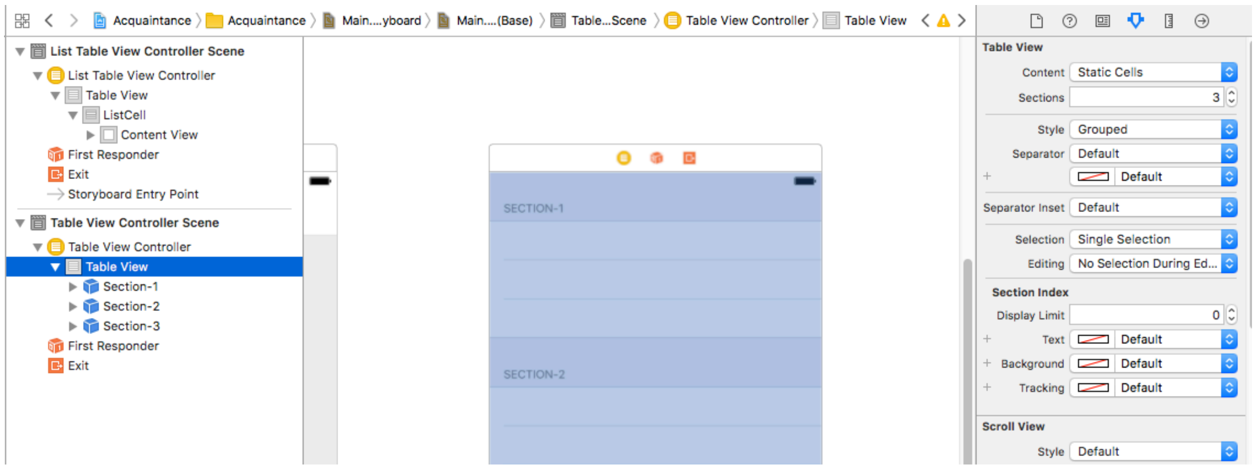


Table View

Content	Static Cells
Sections	3
Style	Grouped
Separator	Default
	Default
Separator Inset	Default
Selection	Single Selection
Editing	No Selection During Ed...

Section Index

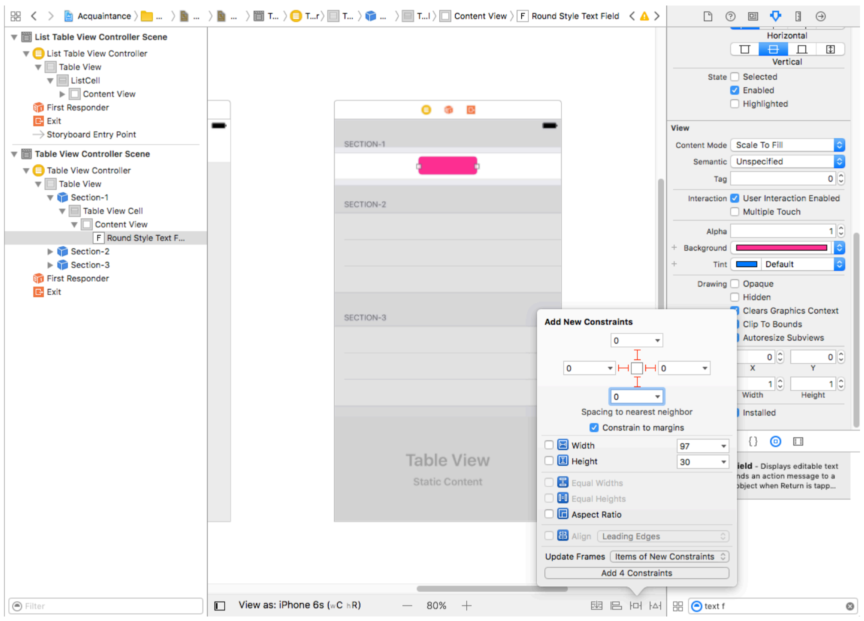
Display Limit	0
Text	Default
Background	Default
Tracking	Default

Scroll View

Style	Default

As configured above, this table view contains 3 sections. The first section will be used to display the name of the acquaintance. In the Document Outline pane, select Section-1. In the Attributes Inspector, change the Header to “Name”. Then expand Section-1. By default, this section contains 3 table view cells. Remove the last 2 cells.

Expand the remaining table view cell and select “Content View”. From the Object Library, drag a “Text Field” object and put it under the “Content View”. Set the following constraints so that the text field fills out the entire cell. You can change the background color of the text field to make it more visible.



List Table View Controller Scene
List Table View Controller
Table View
ListCell
Content View
First Responder
Exit
Storyboard Entry Point
Table View Controller Scene
Table View Controller
Table View
Section-1
Table View Cell
Content View
Round Style Text F...
Section-2
Section-3
First Responder
Exit

Add New Constraints
Spacing to nearest neighbor: 0, 0, 0, 0
Constrain to margins ✓
Width: 97
Height: 30
Equal Widths
Equal Heights
Aspect Ratio
Align: Leading Edges
Update Frames: Items of New Constraints
Add 4 Constraints

Content Mode: Scale To Fill
Semantic: Unspecified
Tag: 0
Interaction: ✓ User Interaction Enabled
Alpha: 1
Tint: Default

The second section will be used to show the photo of the acquaintance. In the Document Outline pane, delete 2 extra table view cells. Select the remaining cell, in the Size Inspector, change the “Row Height” to 200 (or whatever value you prefer).

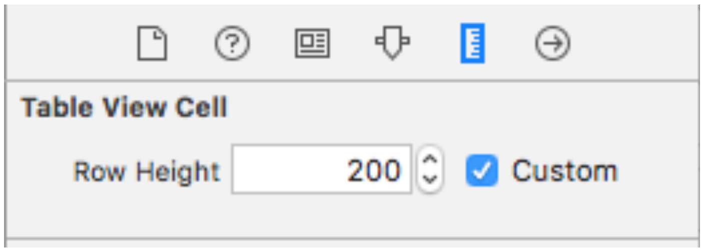


Table View Cell

Row Height: 200 ✓ Custom

Then drag an image view from the Object Library to this cell under *Content View*. Add the following 4 constraints to resize the image view. In the Attribute inspector, set the image to *photoalbum,* set the Content Mode to *Aspect Fill,* and check the *Clip to Bounds* checkbox*.*

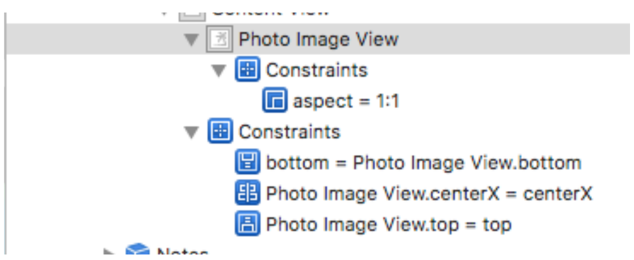


Photo Image View
Constraints
aspect = 1:1
Constraints
bottom = Photo Image View.bottom
Photo Image View.centerX = centerX
Photo Image View.top = top

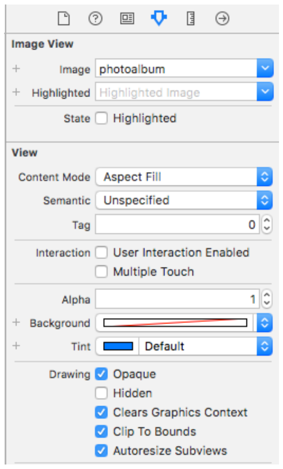
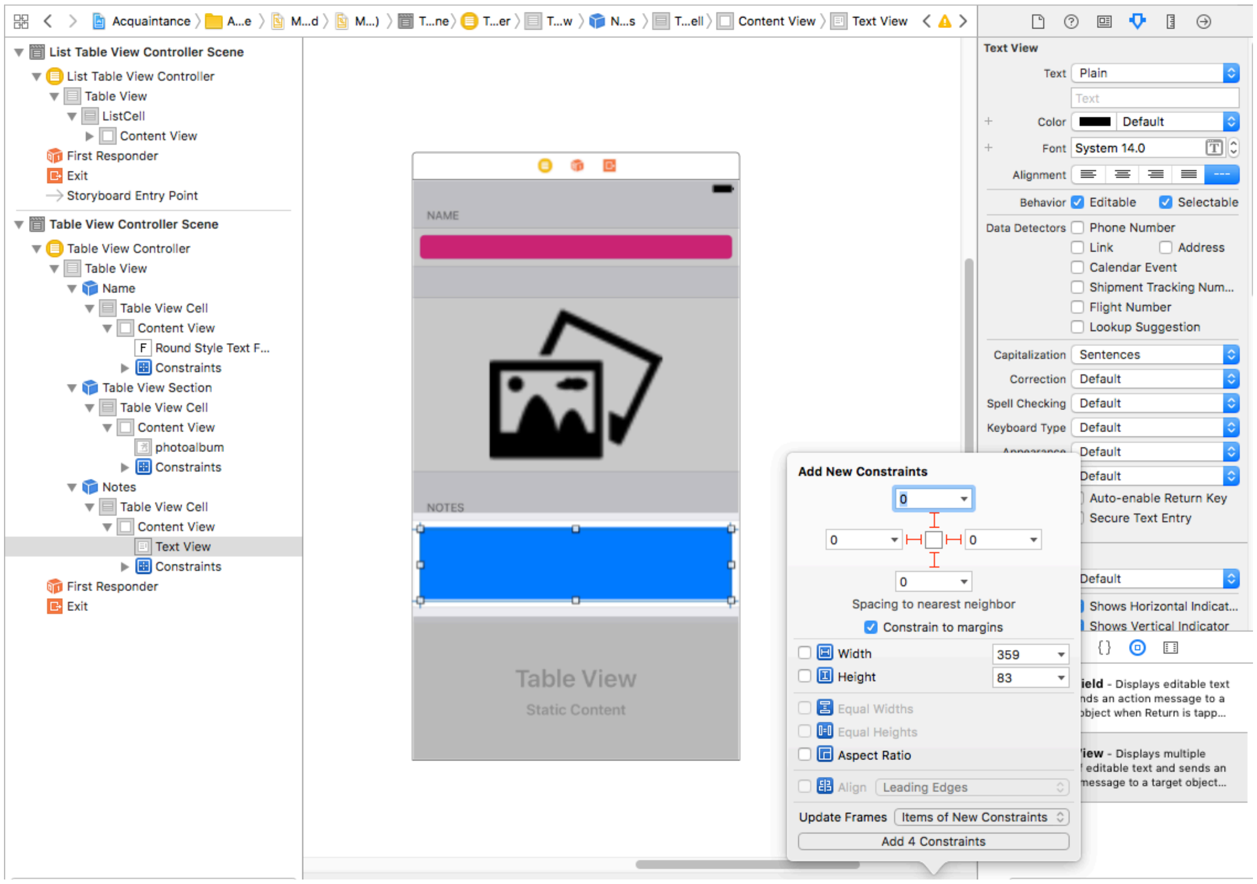


Image View
Image: photoalbum
Highlighted: Highlighted Image
State: Highlighted

View
Content Mode: Aspect Fill
Semantic: Unspecified
Tag: 0
Interaction: User Interaction Enabled
Multiple Touch
Alpha: 1
Background
Tint: Default
Drawing: ✓ Opaque
Hidden
✓ Clears Graphics Context
✓ Clip To Bounds
✓ Autoresize Subviews

The third section will contain one text view for the user to enter notes about this acquaintance. Change the section header to “Notes”. Also change the row height to 100, and then add a “Text View” object to the cell. You can also set a background color to make it more visible to the user.



Text View

Text	Plain
Color	Default
Font	System 14.0
Behavior	✓ Editable ✓ Selectable
Data Detectors	Phone Number, Link, Address, Calendar Event, Shipment Tracking Num..., Flight Number, Lookup Suggestion
Capitalization	Sentences
Correction	Default
Spell Checking	Default
Keyboard Type	Default
Appearance	Default
	Default
	Auto-enable Return Key
	Secure Text Entry
	Default
	Shows Horizontal Indicat...
	Shows Vertical Indicator

Add New Constraints

Top: 0, Left: 0, Right: 0, Bottom: 0

Spacing to nearest neighbor

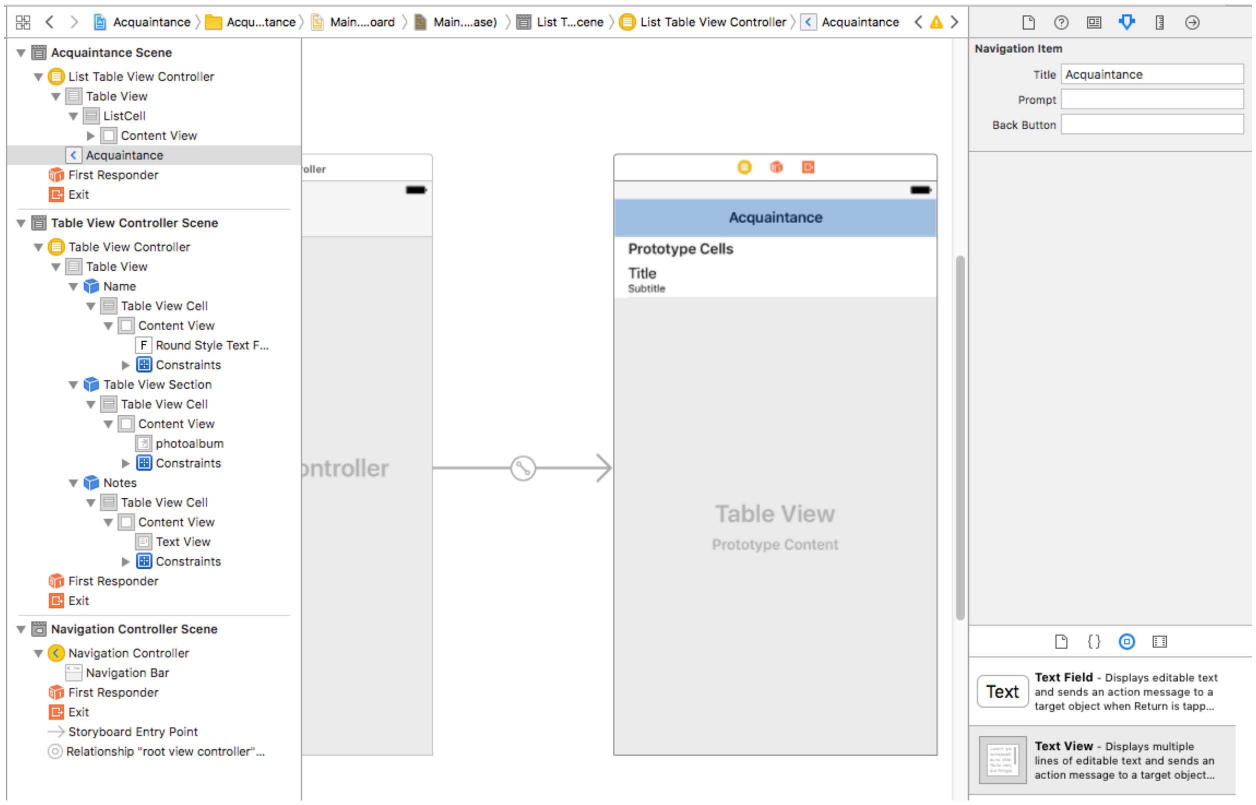
✓ Constrain to margins

Width	359
Height	83
Equal Widths	
Equal Heights	
Aspect Ratio	
Align	Leading Edges
Update Frames	Items of New Constraints

Connecting Views with Navigation Controllers and Segues

Now, it’s time to connect the two table views with navigation controllers and segues to form a hierarchical content structure.

Xcode provides an *embed* feature that makes it easy to embed any view controller in a navigation controller. Select the “list table view controller” and click *Editor* in the menu. Choose Embed in -> Navigation Controller. After the list table view is embedded in the navigator controller, select the navigation bar of the list table view controller. In the Attributes inspector, change the title of the navigation bar to *Acquaintance*.



Navigation Item

Title: Acquaintance

Prompt:

Back Button:

Text Field - Displays editable text and sends an action message to a target object when Return is tapp...

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

The list table view scene will transit to the detail scene when a user taps a cell. To implement this transition, we will add a segue to connect the prototype cell and the detail scene. Press and hold the *control* key, click on the prototype cell and drag to the *Detail View Controller* scene. Select "Show" for the segue style. You will see a connector between the two view controllers.

Segues can be triggered by multiple sources. As your storyboard becomes more complex, it is very likely that you'll have more than one segue between the same two view controllers. Therefore, the best practice is to give each segue a unique identifier. To assign an identifier to the segue created above, select the segue, and then go to the Attributes inspector. Set the value of the identifier to *ShowDetail*.

Build and run the app. When a row of the list table view is tapped, the app navigates to the detail page. The user can tap the *back* button to return to the list page.

Passing Data with Segues

Without writing a single line of code, we've made the app navigable between two table views. Next, we are going to write code to pass information between the list and the detail table views.

First, add a new class *DetailTableViewController* to the project. Open Main.storyboard, select the detail view controller and open the Identity inspector. Change the custom class to *DetailTableViewController*.

Open *DetailTableViewController.swift*, add an instance variable *person* of type *Person*. This object represents the person in the cell tapped by the user and passed from the *List table view controller* to the *Detail view controller*.

```
    var person: Person?
```

When the user taps a cell, the segue "ShowDetail" will be trigged. This segue will do the following things: 1) calling the *prepare(for:sender)* method in the source view controller. You can pass any relevant data to the destination controller in that method. 2) performing the visual transition based on the style of the segue. For example, if the segue style is "Show", the destination view controller will be pushed on top of the current view controller stack.

To pass data from the *List table view controller* to the *Detail view controller*, open *ListTableViewController.swift,* insert the following code in the *prepare(for:sender)* method.

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
        if segue.identifier == "ShowDetail",
            let indexPath = tableView.indexPathForSelectedRow,
            let detailViewController = segue.destination as? DetailTableViewController {
            detailViewController.person = acqList[indexPath.row]
        }
    }
```

Next, add some code in *DetailTableViewController.swift* to display the information. First, add the following outlet variables and connect them to the UI controls in the storyboard.

```
    @IBOutlet weak var nameTextField: UITextField!
    @IBOutlet weak var photoImageView: UIImageView!
    @IBOutlet weak var notesTextView: UITextView!
```

Next, remove the following two delegate methods since we do not need them for static table views.

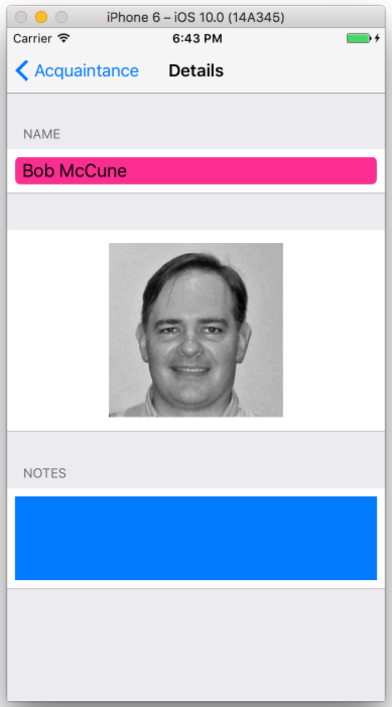
```
    override func numberOfSections(in tableView: UITableView) -> Int {
        // #warning Incomplete implementation, return the number of sections
        return 0
    }

    override func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
        // #warning Incomplete implementation, return the number of rows
        return 0
    }
```

Then, insert the following code in the *ViewDidLoad* method to display the information. This method is called when the view is loaded into the memory.

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

       nameTextField.text = person?.name
        if let photo = person?.photo {
            photoImageView.image = photo
        } else {
            photoImageView.image = UIImage(named:"photoalbum")
        }
        notesTextView.text = person?.notes

        navigationItem.title = "Details"
    }
```

The *navigationItem* represents the view controller that is pushed on top of the navigation controller. In this case, the *navigationItem* is the *Detail table view controller*. It's title is displayed in the center of the navigation bar.

Build and run the app. When tapping a cell in the List table view, the information will be shown in the Detail page. You can click the *name* text field and the *notes* text view to edit those information. However, if you click the back button to return to the List page, the changes are not transferred back. We will discuss how to pass data back and revisit segues in later sections.



Adding a New Acquaintance

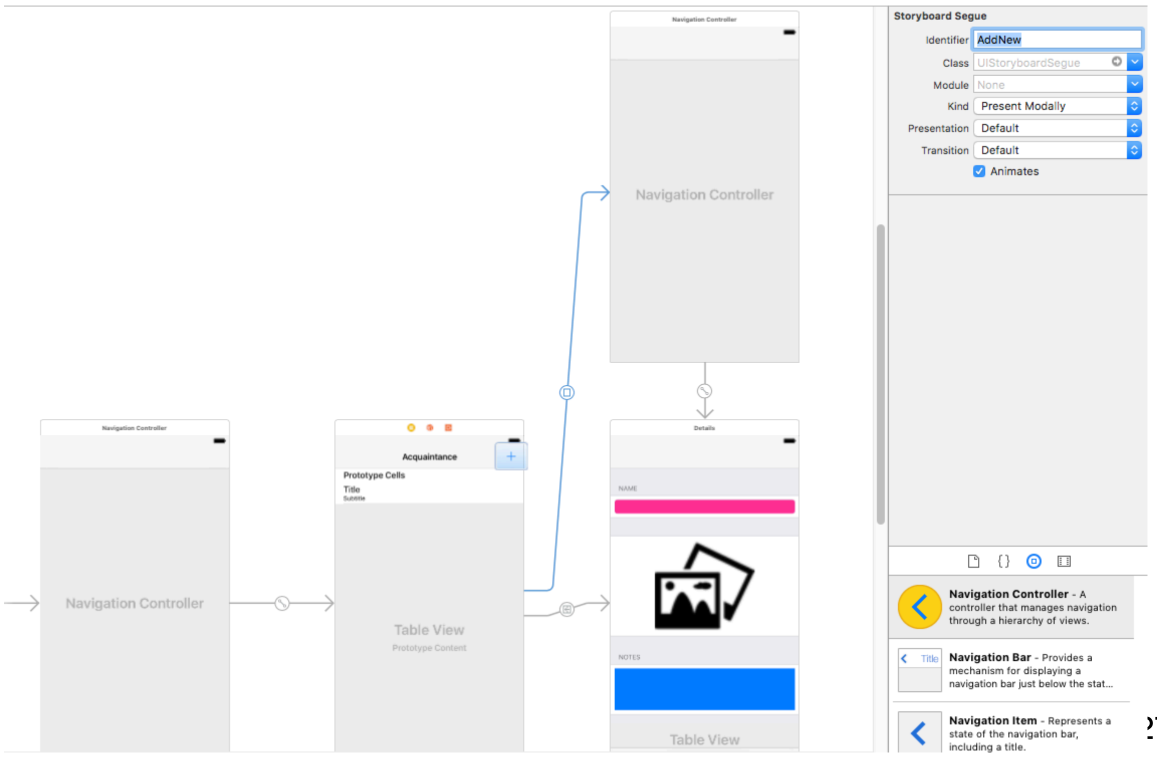
This app allows a user to add a new acquaintance. Since adding a acquaintance requires the same set of information as those displayed in the Detail page, we can reuse the Detail table view, but we will present this table view “modally” rather than pushing it on top of the current view.

We can add a + button at the top-right corner of the List table view. When the user taps this button, the Detail table view will be brought up. In the Interface Builder editor, drag a *Bar Button Item* from the Object library to the navigation bar of the List Table View controller. In the *Attribute inspector*, change the *System Item* to A*dd*, and you'll see a + icon. A bar button (UIBarButtonItem) is very similar to a standard button (UIButton). However, a bar button is specifically designed for navigation bars and toolbars.

Before connecting the two view controllers with a segue, we will follow the common practice to embed the Detail view controller into a navigation controller. Drag a Navigation Controller from the Object Library to the storyboard. Delete its Root View Controller scene. From the new navigation controller, hold down the *Control* key and drag to the Detail view controller. In the pop-up window, select *Root View Controller* relationship.

Press and hold the *Control* key, click the + button and drag to the navigation controller containing the Detail view controller. Select “Present Modally" as the segue style, and you'll see a segue connector between the two view controllers. Select this segue, in the *Attributes* inspector, set the identifier as *AddNew*. Since no data is needed when presenting the Detail scene for adding new acquaintance, we do not need to add any code in the *prepare(for:sender)* method for this segue.

Build and run the app. When the user taps the + button, a blank Detail page with will be animated up from the bottom.



Storyboard Segue

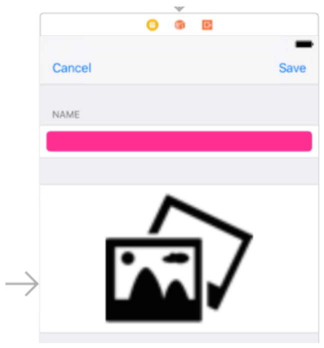
Identifier	AddNew
Class	UIStoryboardSegue
Module	None
Kind	Present Modally
Presentation	Default
Transition	Default
✓	Animates

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

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

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

The user can enter information in that page. We need to provide a way for the user to dismiss or save that information. A commonly used method is to add two buttons in the navigation bar. In the *Detail view controller*, drag a *Bar Button Item* to the top-left corner of its navigation bar. In the Attribute inspector, set its Title to *Cancel*. Drag another Bar Button Item to the top-right corner of the navigation bar, and name it *Save*.



When a user taps the Cancel button, the detail view will be dismissed, and the information entered by the user will be discarded. The app will transit to the List scene. This transition can be done in an action method. In *DetailTableViewController.swift*, define an *IBAction* method named *cancel* and connect it to the *Cancel* button in the Detail scene. Define the *cancel* method as follows.

```
@IBAction func cancel(_ sender: UIBarButtonItem) {
        // Depending on style of presentation (modal or push presentation), this view controller needs to be dismissed in two different ways.
        if presentingViewController is UINavigationController {
            dismiss(animated: true, completion: nil)
        } else {
            navigationController!.popViewController(animated: true)
        }
    }
```

This method first determines how the Detail scene was presented by checking the type of the property *presentingViewController*. If it was presented modally (using the Add button), i.e., the type of the *presentingViewController* property is *UINavigationController*, the *Detail* scene will be dismissed using *dismiss(animated:completion).* Otherwise, the Detail view was presented with push navigation (tapping a table view cell), it will be dismissed by the navigation controller that presented it.

Build and run the app. Regardless how the Detail scene is presented, when the user clicks the *Cancel* button, the app should navigate back to the List scene without taking any changes the user made in the Detail scene.

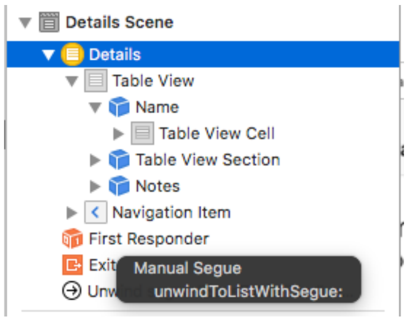
Passing Data from the Detail view to the List view with Segues

The Detail scene allows the user to add a new acquaintance or update the information of an existing acquaintance. In either case, the app needs to pass data from the Detail view controller to the List view controller when a user taps the *Save* button. This data passing can be implemented by a so-called *unwind* segue. An unwind segue can be used to navigate back through a modal or push segue.

To use an unwind segue, you need to do two things. First, declare an *IBAction* method in the destination view controller. In this case, it is the List table view controller. In *ListTableViewController.swift*, add the following action method. Note that this method must take a segue object as the parameter.

```
@IBAction func unwindToList(segue:UIStoryboardSegue) {
}
```

Secondly, in the Main.storyboard, select *Detail View Controller* in the Document Outline pane, press and hold the *Control* key and drag from the *Detail View Controlller* to the *Exit*. When prompted, select *unwindToList* for the action segue. Select this segue and set its identifier to *SaveToList*.



When the unwind segue is trigged, it does the following things: 1) calling the *prepare(for:sender)* method in the source view controller (Detail view controller in this case). You can prepare any relevant data you want to pass to the destination view controller (List table view controller in this case) in this method. 2) performing the visual transition back to the List view controller scene, depending on how the Detail scene was presented. 3) calling the unwind action method in the destination view controller (List table view controller in this case).

The data we want to pass to the List view controller is the information the user enters in the UI controls. In *DetailTableViewController.swift*, insert the following code to the *prepare(for:sender)* method.

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
        // Get the new view controller using segue.destinationViewController.
        // Pass the selected object to the new view controller.
        if segue.identifier == "SaveToList" {
            if person == nil {
                person = Person(nameTextField.text!)
            } else {
```

```
            person?.name = nameTextField.text!
        }

        person!.photo = photoImageView.image
        person!.notes = notesTextView.text
    }
}
```

If the person optional object is currently nil, we need to create a new object first. Since a person's name cannot be nil, we need to make sure that *nameTextField.text* is not nil.

Open *ListTableViewController.swift*, and write the unwind method as follows:

```
@IBAction func unwindToList(segue:UIStoryboardSegue) {
      if segue.identifier == "SaveToList",
        let detailViewController = segue.source as? DetailTableViewController,
        let person = detailViewController.person {
            if let selectedIndexPath = tableView.indexPathForSelectedRow {
                // Update an existing person.
                acqList[(selectedIndexPath as NSIndexPath).row] = person

                tableView.reloadRows(at: [selectedIndexPath], with: .none)
            } else {
                // Add a new person.
                let newIndexPath = IndexPath(row: acqList.count, section: 0)
                acqList.append(person)

                tableView.insertRows(at: [newIndexPath], with: .bottom)
            }
    }
}
```

This method first retrieves the person object, and then either updates an existing item with this object or adds this object to the acquaintance list array. After the acquaintance list is updated, the table view must also be updated.

Build an run the app. In the *Detail* page, after you make some changes and click the *Save* button, nothing happens. It is because the *SaveToList* unwind segue is from the *DetailTableViewController* to the *ListTableViewController*. Tapping the *Save* button will not get this segue triggered. We will need to manually trigger this segue to do the transition. The reason for making a manual segue is to give the app a chance the validate the user's input before passing it back to the List scene. For example, the user must enter a name for the acquaintance. If the name field is empty, the app should not move back to the List scene. This check can be done in the action method of the *Save* button. Define an IBAction method *save* and connect it to the *Save* button in the storyboard. Write the *save* action method as follows:

```
@IBAction func save(_ sender: UIBarButtonItem) {
      // Depending on style of presentation (modal or push presentation), this view controller needs to be
dismissed in two different ways.
      if nameTextField.text == nil || nameTextField.text!.isEmpty {
        let alertController = UIAlertController(title: "Invalid Data", message: "The name cannot be empty",
preferredStyle: .alert)
```

```
            alertController.addAction(UIAlertAction(title: "OK", style:.cancel, handler: nil))
            present(alertController, animated: true, completion: nil)
        } else {
            performSegue(withIdentifier: "SaveToList", sender: self)
        }
    }
```

If the name textfield is nil or empty, a error message box will be shown. If the name is not empty, the *performSegue(withIdentifier:send)* method will activate the unwind segue *SaveToList*.

Build and run the app. When the user taps the *Save* button, the data will be passed back to the *List* table view. If the name field is empty, the app will prompt an error message box.

Displaying the Photo Library Using UIImagePickerController

In this app, we want to allow the user to choose a photo from the built-in photo library for an acquaintance. The UIKit framework provides a convenient API called *UIImagePickerController* for picking photos from the photo library. This API can also be used to bring up a camera interface for taking a picture. Note that the simulator doesn't support the camera feature. If you want to test an app that utilizes the built-in camera, you'll need a real iOS device.

To implement this functionality, add the *UITableViewDelegate* method *tableview(_:didSelectRowAt)* to detect the touch and load the photo library:

```
override func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {
      if indexPath.section == 1, UIImagePickerController.isSourceTypeAvailable(.photoLibrary) {
         let imagePicker = UIImagePickerController()
         imagePicker.allowsEditing = false
         imagePicker.sourceType = .photoLibrary
         present(imagePicker, animated: true, completion: nil)
      }
   }
```

The *tableView(_:didSelectRowAt:)* method is called when a cell is selected. In this case, we only want to bring up the photo library when the photo cell is selected, which is the first cell in the second section. The index of that section is 1 since the index starts at 0.

Sometimes, the user may not allow you to access the photo library. As a good practice, you should always use the class method *isSourceTypeAvailable* to verify if a particular media source is available.

To load up the photo library, we create an instance of *UIImagePickerController* and set its sourceType to *.photoLibary* . You then call the *present*(_:*animated*:*completion*:) method to bring up the photo library.

Build and run the app, when tapping the photo cell, the app may crash with the following error message:

```
[access] This app has crashed because it attempted to access privacy-sensitive
data without a usage description.  The app's Info.plist must contain an
NSPhotoLibraryUsageDescription key with a string value explaining to the user
how the app uses this data.
```

In iOS 10 or later, for privacy reasons, you have to explicitly describe the reason your app accesses the user's photo library. If you fail to do so, you will see the above error.

To fix that, you need to add a key (*NSPhotoLibraryUsageDescription*) in the Info.plist file and provide your reason. In the Project Navigator pane, select Info.plist. Right click any blank area in the editor and select *Add Row*. Choose "*Privacy - Photo Library Usage Description*" for the key and set the value to: *You need to grant the app access to your photo library so you can pick a photo for the acquaintance*.

Build and run the app. This time, tapping the photo cell will bring up the built-in photo library. When prompted, tap OK to enable your app to access the photo library. However, if you select a photo from the photo library, it wouldn't show up in the image view. To retrieve the photo selected from the photo library, the *DetailTableViewController* class needs to adopt two delegates: *UIImagePickerControllerDelegate* and *UINavigationControllerDelegate*.

```
class DetailTableViewController: UITableViewController, UIImagePickerControllerDelegate,
UINavigationControllerDelegate
```

Add the following line of code in the *tableView(_:didSelectRowAt:)* method:

```
            ...
            imagePicker.allowsEditing = false
            imagePicker.sourceType = .photoLibrary
            imagePicker.delegate = self
            ...
```

When a user chooses a photo from the photo library, the *imagePickerController(_:didFinishPickingMediaWithInfo:)* delegate method will be called. Implement this delegate method as follows:

```
func imagePickerController(_ picker: UIImagePickerController, didFinishPickingMediaWithInfo info: [String :
Any]) {
        if let selectedImage = info[UIImagePickerControllerOriginalImage] as? UIImage {
            photoImageView.image = selectedImage
            photoImageView.contentMode = .scaleAspectFill
            photoImageView.clipsToBounds = true
        }

        dismiss(animated: true, completion: nil)
    }
```

The *info* dictionary contains the selected image. *UIImagePickerControllerOriginalImage* is the key of the image selected by the user. The above code assigns the image view with the selected image. We also change the content mode so that the image is displayed in aspect fit mode. Lastly, we call the dismiss method to dismiss the image picker.

Customizing Table View Cells

So far, we are using the built-in table view cell style in the *List table view* to display a list of acquaintance. In this section, we will customize the table view cell with customized table view cell class.

In Main.storyboard, select the prototype cell of the List table view. In the Attributes Inspector, change the Style to *Custom*.

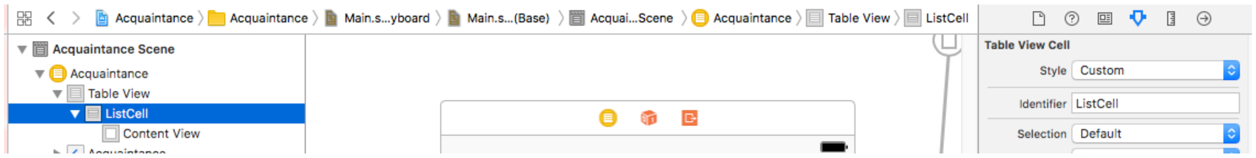


Table View Cell
Style: Custom
Identifier: ListCell
Selection: Default

Before we design the custom cell, we will need to set a custom row height for the table view cell. Select the *List Table View* in the Document Outline pane. In the Size Inspector, change *Row Height* to 100.

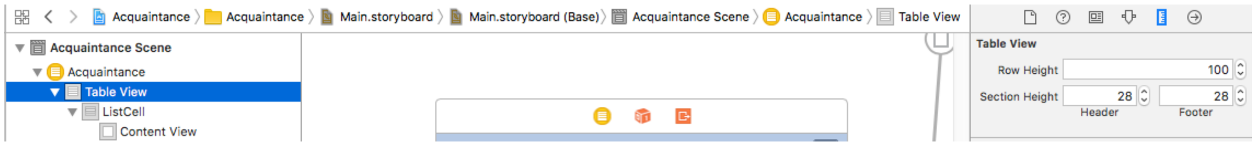


Table View
Row Height: 100
Section Height: 28 (Header) 28 (Footer)

Then select the prototype cell. In the Size Inspector, check the *custom* checkbox. The row height should be automatically set to *100*.

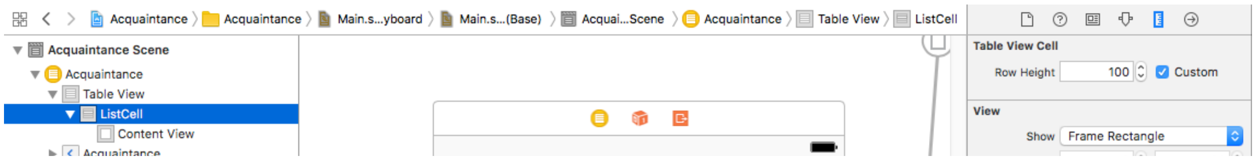
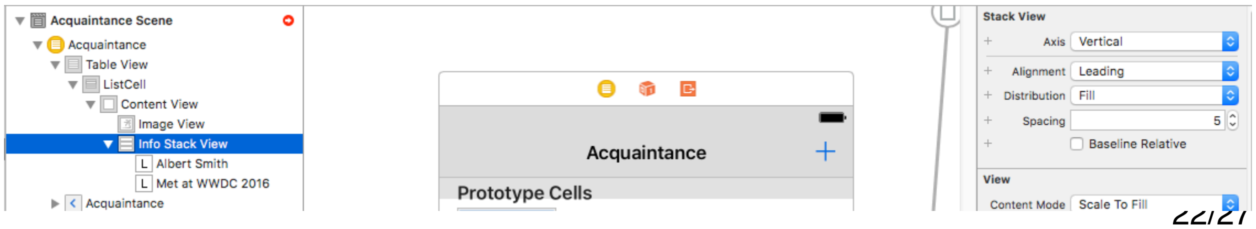


Table View Cell
Row Height: 100 ☑ Custom
View
Show: Frame Rectangle

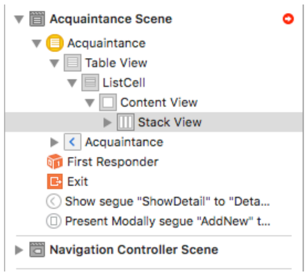
After altering the row height, drag an Image View object from the Object library to the prototype cell under *Content View*. Resize the image view and place it on the left side of the cell. Drag two labels and place them to the right of the image view. Make the font size of the top label bigger, e.g., 24pt.

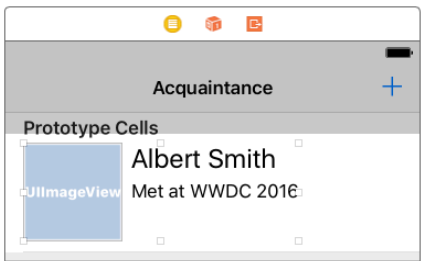
In iOS, we will make extensive use of stack views so that we can define much less layout constraints for individual UI components. We will first stack the two labels together. Hold the *command* key, and select the two labels. Click the *stack* button in the layout bar to embed them in a vertical stack view. Select the stack view, in the Attributes Inspector, set Alignment to *Leading*, Distribution to *Fill,* and change the Spacing to some value, e.g., 5 to add a space between the two labels.

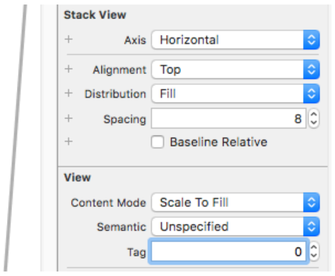


Stack View
Axis: Vertical
Alignment: Leading
Distribution: Fill
Spacing: 5
☐ Baseline Relative
View
Content Mode: Scale To Fill

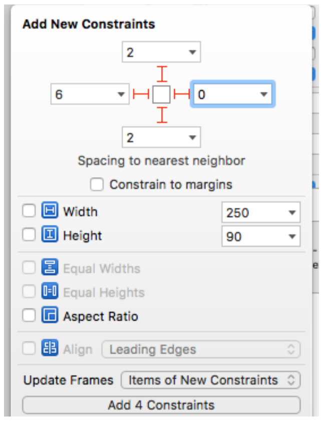
Next, hold the *command* key, and select the image view and the stack view containing the two labels. Click the *stack* button in the layout bar to embed them in a horizontal stack view. Select the stack view, in the Attributes Inspector, set Alignment to *Top*, Distribution to *Fill,* and add some spacing, e.g., 8.





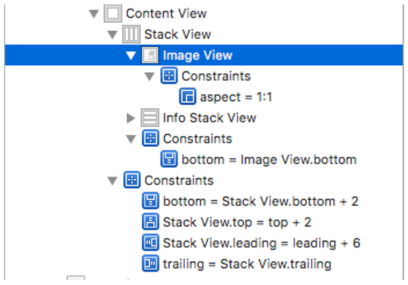

Stack View
Axis: Horizontal
Alignment: Top
Distribution: Fill
Spacing: 8
Baseline Relative
View
Content Mode: Scale To Fill
Semantic: Unspecified
Tag: 0

Now, we can begin to specify layout constraints, starting from the outermost stack view. Select that stack view, and define the leading, top, trailing, and bottom constraints as follows:

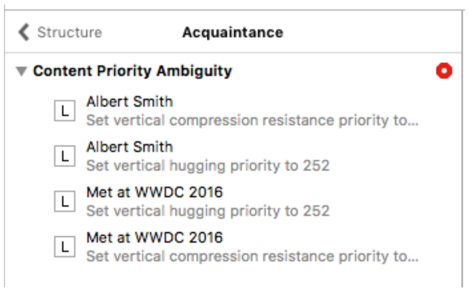

Add New Constraints
Top: 2
Leading: 6
Trailing: 0
Bottom: 2
Spacing to nearest neighbor
Constrain to margins
Width: 250
Height: 90
Equal Widths
Equal Heights
Aspect Ratio
Align: Leading Edges
Update Frames: Items of New Constraints
Add 4 Constraints

After the stack view is placed, we need to layout the subviews. The image view object already has 2 constraints set by the system when being embedded in the stack view. Its top equals to the top of the stack view, and its leading is aligned with the left edge of the stack view. So we only need to specify one more vertical constraint and one more horizontal constraint. We also want to make the image view a square. So we add the following two constraints: 1) Aspect Ration 1:1; 2) bottom space to the stack view = 0.

After setting those constraints, you may still see a red indicator at the top-right corner of the Document Outline pane. The red indicator means that Interface Builder has detected some layout constraint issues. Click the red indicator, you may see the issues like follows.



```
▼ Content View
  ▼ Stack View
    ▼ Image View
      ▼ Constraints
          aspect = 1:1
    ▶ Info Stack View
    ▼ Constraints
        bottom = Image View.bottom
  ▼ Constraints
      bottom = Stack View.bottom + 2
      Stack View.top = top + 2
      Stack View.leading = leading + 6
      trailing = Stack View.trailing
```



```
Structure        Acquaintance

▼ Content Priority Ambiguity
  Albert Smith
  Set vertical compression resistance priority to...
  Albert Smith
  Set vertical hugging priority to 252
  Met at WWDC 2016
  Set vertical hugging priority to 252
  Met at WWDC 2016
  Set vertical compression resistance priority to...
```

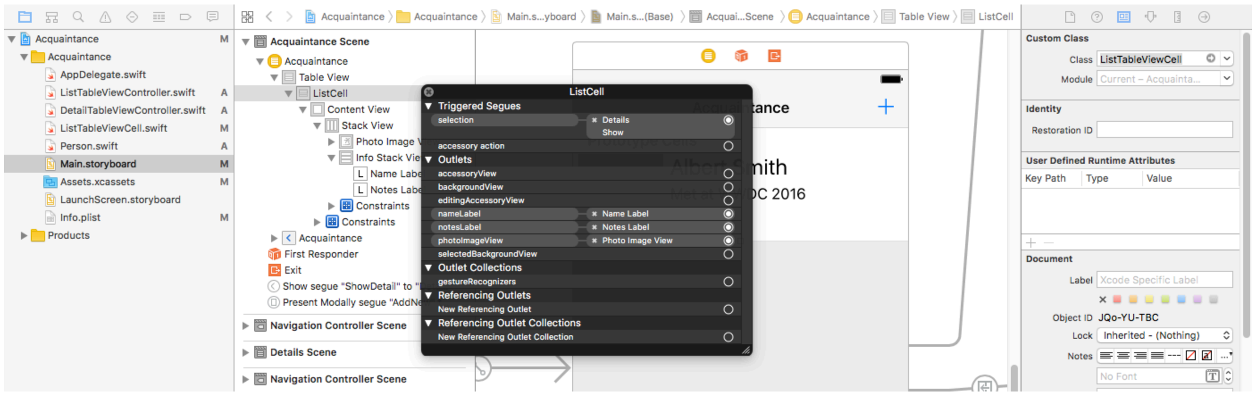
The reason to have those issues is because the stack view does not know how to layout the two labels vertically to fill the stack view. Which label should the stack view expand first if the stack view height is enlarged? Which label should the stack view shrink first if the stack view height is reduced?

The *Content Hugging Priority* describes how ease a control is allowed to grow. The default priority is 251. The lower the value, the easier its size can grow. The *Content Compression Resistance* priority defines how easy a control is allowed to be squished. The default value is 750. The higher the value, the harder its size can be shrunk. In this case, we want to keep the size of the name label unchanged and expand/shrink the notes label first. So we set the *Vertical Content Hugging Priority* of the notes label to 250, and the *Vertical Content Compression Resistance Priority* of the name label to 751. These settings should solve the constraint issues.

By default, the prototype cell is associated with the *UITableViewCell* class. In order to update the cell data, we need to create a subclass of *UITableViewCell* for the prototype cell. From menu, select File->New->File… to create a new "Cocoa Touch Class" and set the value of "Subclass of" to UITableViewCell. Named it *ListTableViewCell.* Open Main.storyboard, select the cell in the storyboard. In the Identity inspector, set the custom class to *ListTableViewCell*.

Next, declare the following outlet variables in the *ListTableViewCell* class, and connect these outlets to the controls in the storyboard.

```
@IBOutlet weak var photoImageView: UIImageView!
@IBOutlet weak var nameLabel: UILabel!
@IBOutlet weak var notesLabel: UILabel!
```



```
ListCell
▼ Triggered Segues
  selection                 × Details Show
  accessory action
▼ Outlets
  accessoryView
  backgroundView
  editingAccessoryView
  nameLabel                 × Name Label
  notesLabel                × Notes Label
  photoImageView            × Photo Image View
  selectedBackgroundView
▼ Outlet Collections
  gestureRecognizers
▼ Referencing Outlets
  New Referencing Outlet
▼ Referencing Outlet Collections
  New Referencing Outlet Collection
```

```
Custom Class
  Class   ListTableViewCell
  Module  Current – Acquainta...
Identity
  Restoration ID
Document
  Label      Xcode Specific Label
  Object ID  JQo-YU-TBC
  Lock       Inherited - (Nothing)
  Notes      No Font
```

Finally, in *ListTableViewController.swift,* modify the *tableView(:cellForRowAt:)* method with the following code:

```
override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
        let cell = tableView.dequeueReusableCell(withIdentifier: "ListCell", for: indexPath) as!
ListTableViewCell

        let person = acqList[indexPath.row]

        // Configure the cell...
        if let photo = person.photo {
            cell.photoImageView.image = photo
        } else {
            cell.photoImageView.image = UIImage(named:"photoalbum")
        }
        cell.nameLabel.text = person.name
        cell.notesLabel.text = person.notes

        return cell
    }
```

Build and run the app.

Beautifying the Appearance of Navigation Bar

So far, we have not refined the visual appearance of the app. Apple provides an Appearance API for customizing the appearance of UIKit controls across the entire application, through an appearance proxy of a specific class. For example, to customize the appearance of navigation bar, you use *UINavigationBar.appearance*() to get the appearance proxy of the *UINavigationBar* class.

All the customization code can all be put in AppDelegate's method *application(_:didFinishLaunchingWithOptions:)*.

We will start from customizing the appearance of the navigation bar. To change the background color of the navigation bar, set the *barTintColor* of the navigation bar.

```
UINavigationBar.appearance().barTintColor = UIColor(red: 25/255.0, green:
        56/255.0, blue: 135/255.0, alpha: 1.0)
```

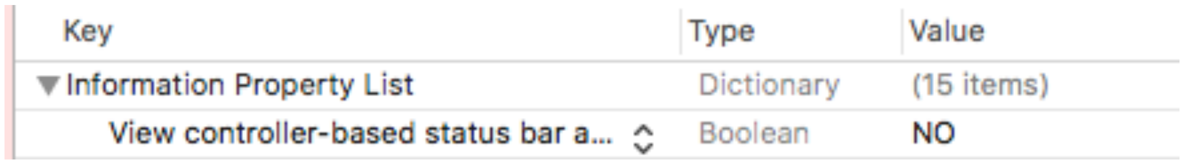
To change the color of the bar buttons on the navigation bar, set the *tiltColor*:

```
UINavigationBar.appearance().tintColor = UIColor.white
```

To change the navigation bar title color and font, set the *titleTextAttributes* property:

```
if let barFont = UIFont(name: "Futura", size: 24.0) {
        UINavigationBar.appearance().titleTextAttributes =
[NSForegroundColorAttributeName:UIColor.white, NSFontAttributeName:barFont]
}
```

Lastly, we can change the style of status bar from dark to light to make the app look better. One way is to set the status bar style through the statusBarStyle property of UIApplication, which will change the style of the status bar for the entire app. By default, however, the Xcode project is enabled to use "View controller-based status bar appearance". This means you can control the appearance of the status per view controller. If you want to change the style of the status globally, you need to first opt out the *View controller-based status bar appearance*. To opt out, select the Info.plist in project navigator. Right click any blank area and select the Add Row. Insert a new key named *View controller-based status bar appearance* and set the value to NO.



Key	Type	Value
▼ Information Property List	Dictionary	(15 items)
View controller-based status bar a...	Boolean	NO

Then add the following line in *application(_:didFinishLaunchingWithOptions:)*

```
UIApplication.shared.statusBarStyle = .lightContent
```

Customizing the Appearance of the Detail Table View

Most of the customizations to the *Detail* Table view can be done in the storyboard.

First, we can remove the separators for all rows. Select *Detail Table View*, in the *Attributes* inspector, change Separator to *None.*

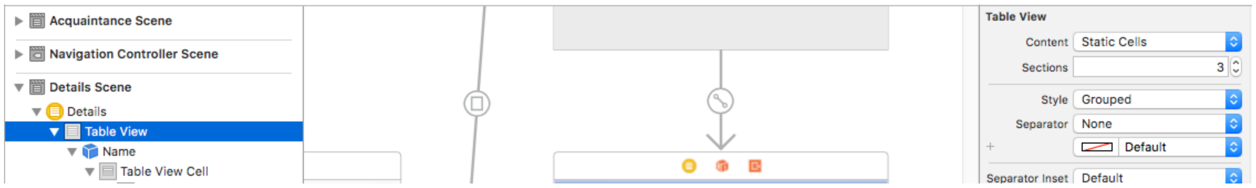
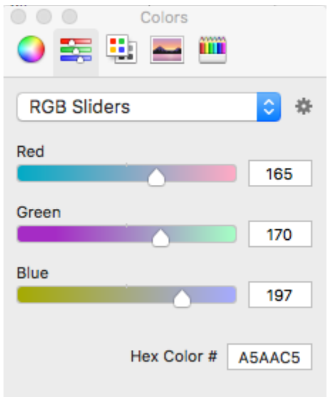


Table View
Content: Static Cells
Sections: 3
Style: Grouped
Separator: None
Default
Separator Inset: Default

Next, set the background color of the table view. Select *Detail Table View,* in the *Attributes* inspector, change the background color.



RGB Sliders
Red: 165
Green: 170
Blue: 197
Hex Color # A5AAC5

Next, select the table view cells under every sections, modify their background colors to the same color.
You can also change the background color, font, etc for every cells. Use your own imaginations.