MODULE Deep Dive into iOS App Development

SESSION TITLE: [Working with Views, Outlets, and Actions]

Sources:

1. [Beginning IOS programming] [Chapter Number 5][9781118841471]
2. [Professional IOS programming] [Chapter Number 4][9781118661130]

MODULE Objective

At the end of this module, you will be able to:

- Work with views, alerts, and actions

Session Objectives

At the end of this session, you will be able to:

- Add table views into an app
- Implement the data source
- Implement sections and indexes
- Ask a user input form
- Respond to the user input
- Extend the `UIAlert` view

<H1> Introduction

In this session, you will add various functionalities to an app named **Bands**. This is a simple app that you can use to keep track of your favorite bands or take quick notes about your favorite music bands or groups.

The most common views in iOS applications are table views. A table view presents data in a scrollable list of multiple rows that may be divided into sections. For example, you want users to add as many bands as they like in your **Bands** app. The added bands should be displayed in a scrollable list, which provides details about the bands in distinct sections. You can use a table view to give a standard look and feel to your app, or you can customize it with varying cell heights and content for a complex user interface.

In this session, you will learn how to add table views, sections, and indexes in an app. You will also learn how to implement and present a `UIActionSheet` in iOS 7. The `UIActionSheet` is a preliminary

document for an application programming interface (API). The session also describes how to process the user response of a `UIActionSheet`.

<H1> [Exploring Table Views]

In the **Bands** app, you would be adding different items, and you need to add them in an ordered form. For example, if you are adding a contact in your phone, then you need to enter the contact details and save the contact. Now after saving the contact when you open your contacts list again, you will see the contacts in a table view. Similarly, when you open your phone settings, the options available are also created using a table view. In a similar fashion, you have to add names of music bands in the **Bands** application.

You can use a table view to perform the following actions:

- Enable users to navigate through hierarchically structured data
- Present an indexed list of items
- Display detail information and controls in visually distinct groupings
- Present a selectable list of options

A table view is an instance of the `UITableView` class. The `UITableView` class is a little different from what you would expect a table to be. It consists of a scrolling view with a single row of cells instead of a table with multiple columns and rows. A better way to think of a `UITableView` is a scrolling list of cells.

Some features about the `UITableView` are as follows:

- The `UITableViews` have their own delegates called the `UITableViewDelegates` to control their appearance. This is just as `UITextViews`, which have the `UITextViewDelegates` to control their look and feel.
- The `UITableViews` also have a data source protocol called the `UITableViewDataSource`, which interacts with the data model of the app.
- Because `UITableViews` are so prevalent in iOS applications, Apple supplies a `UITableViewController` class, a subclass of `UIViewController`. A major benefit of using the `UITableViewController` subclass is that it implements the `UITableViewDelegate` and `UITableViewDataSource` along with an `IBOutlet` to the `UITableView`. Therefore, using `UITableViewController` is much easier than adding all these three objects to another class.

- While working with **Swift** language, it is much cleaner to use `UITableViewDelegate` and `UITableViewDataSource`. Its new protocol syntax is as follows:

```
class ViewController: UIViewController, UITableViewDelegate, UITableViewDataSource{
      ----
                }
```

Big Picture

A table view is one of the most common user interface (UI) elements in iOS applications. Applications use table views to display data in scrollable lists. Examples of table views include the built-in iPhone apps, Contacts for displaying contacts and Mail for displaying email.

Real Life Connect

If you are looking to develop an iOS application, its view (or how it appears) plays a big role in application development. For that, you have to set up its user interface table view by using the **`UITableView -> UITableViewDelegate`** menu. The `UITableView` and `UITableViewDelegate` also have a data source protocol, the `UITableViewDataSource`, which interacts with the data model of the app. Therefore, you should place the position of a cell, which contains a value like a button, or a cell for an image, or a cell for text or information in an effective way to make an application impressive to users.

<H2> [Learning about Tables]

Source: [Beginning of iOS Programming] [Chapter No 5] [104]

The best way to learn about `UITableView` works is to start by adding one to your **Bands** app. You can use just a `UITableView` as the main view of the app.

A common way to add views in application is to use a `UINavigationController`. This is known as a Master-Detail application. The `UINavigationController` is a container controller that enables different `UIViewControllers` to display within it. It also adds a `UINavigation` Item at the top of the screen.

<H3> [Adding a `UITableView`]

Source: [Beginning of iOS Programming] [Chapter No 5] [104-106]

---

1. Open the **Bands** project in Xcode.
2. Select the `Main.storyboard` file from the **Project Navigator**.
3. Find and drag a new **Navigation Controller** from the Objects library onto the storyboard, as shown in **Figure 1**. The Navigation Controller that you added has a `UITableView` set as its root `UIViewController` by default. The Storyboard relationship with the two dots and a line signifies this.

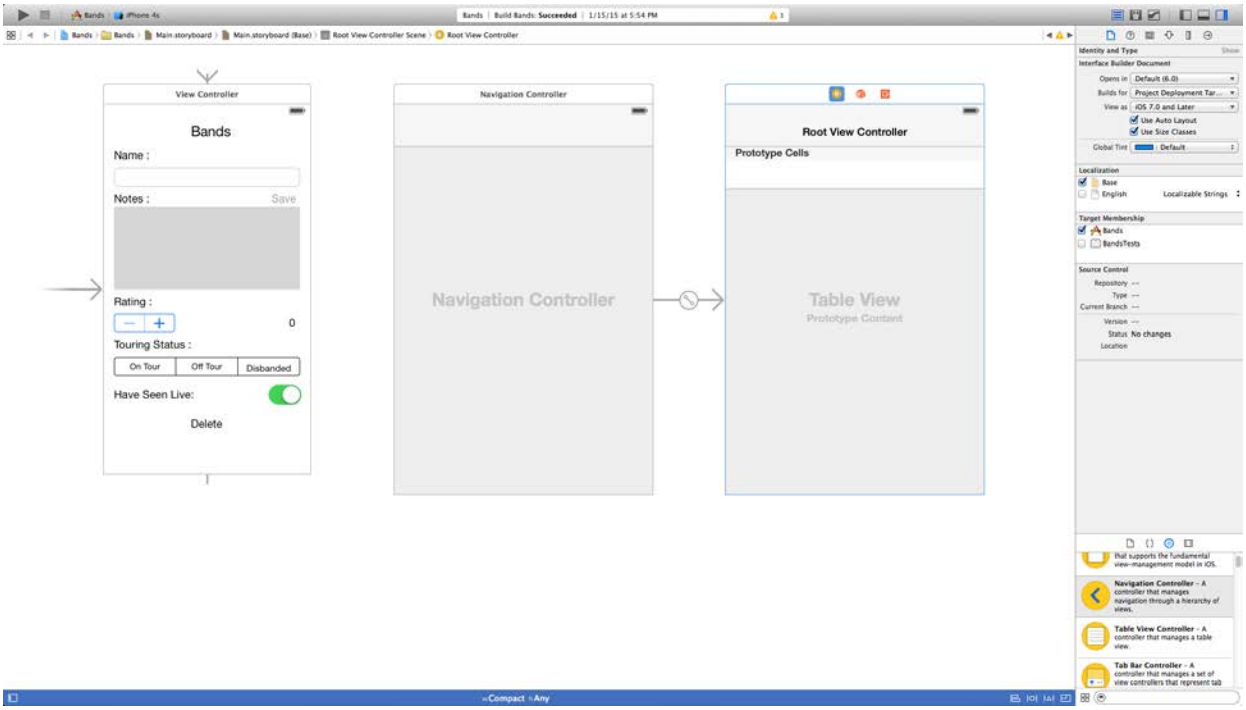

Figure 1: Drag a New Navigation Controller to the Storyboard

4. Move the arrow pointing to the left side of the **View Controller** and drag it to the Navigation Controller. This arrow will tell the Storyboard which scene to initially show when the app launches. By pointing it at the Navigation Controller, the `UITableView` is now shown in **Figure 1** on the launch instead of the View Controller.

5. Select the Navigation Item of the **Bands** List Table View Controller in the Storyboard hierarchy, as shown in **Figure 2**.

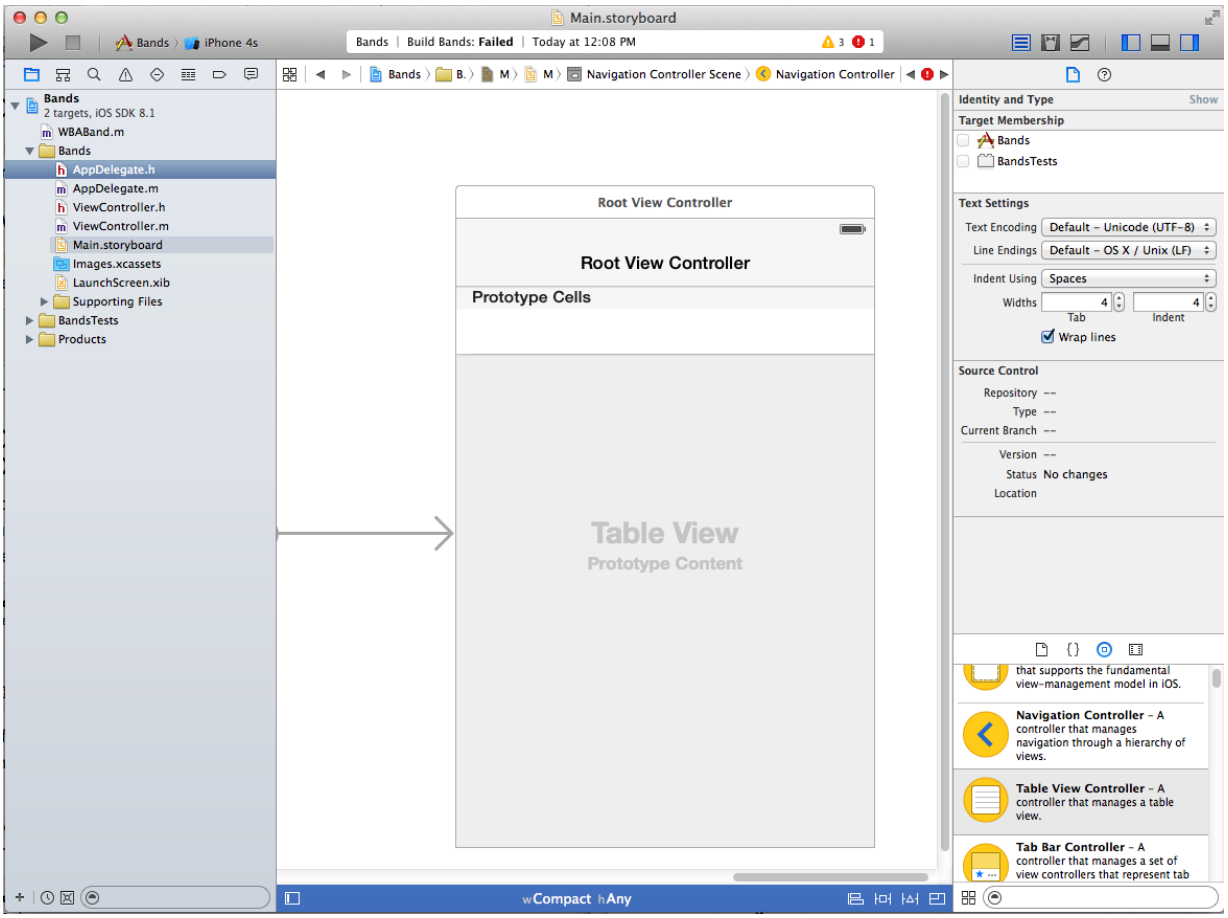

Figure 2: Select the Navigation Item of the Table View Controller

6. In the Attributes inspector, change the **Title** from **Root View Controller** to **Bands**, as shown in **Figure 3**.

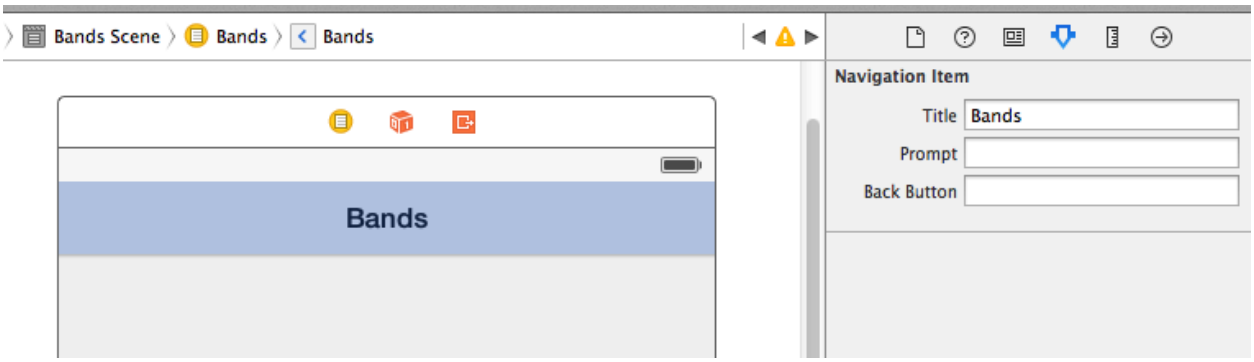

Figure 3: Specify the Title of the Navigation Item as Bands

7. Run the app in the iPhone 4-inch simulator. You can now see the table view, as shown in **Figure 4**.

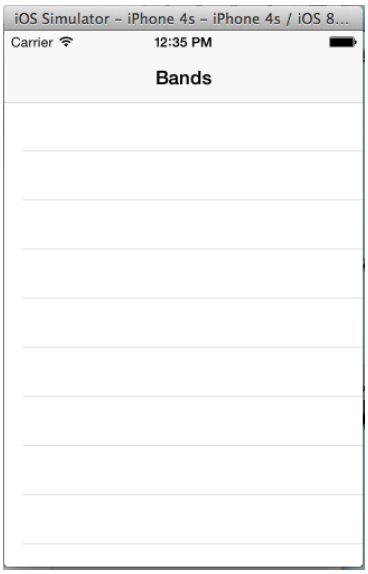

Figure 4: Run the App in Simulator

Additional Knowhow

Entities such as **Table View Controller, Table Data Source**, and **Table View Delegate** interact with each other to create a table view. In short, the View Controller, Data Source, and Delegate are the same objects. The Data Source adopts the `UITableViewDataSource` protocol and the Delegate adopts the `UITableViewDelegate` protocol. After it is set, the View Controller sends the `reloadData` message to the data source and delegates. The `reloadData` will dynamically change the updated message.

Quick Tip

Storyboards make it easy for you as a developer to visualize how the users can navigate through an app.

<H2> [Learning about Cells]

Source: [Beginning of iOS Programming] [Chapter No 5] [107]

The `UITableViewCell` object represents a cell in a `UITableView`, and is used to list entities in rows. In the **Bands** app, you can use the `UITableViewCell` feature in displaying a list of songs, categories, and genres.

Unlike the `UITableView`, the `UITableViewCell` objects do not have a delegate. If you use a predefined style for your cell, you will not need to add a code file. Predefined cells are versatile, so you

should consider using them first before creating a custom cell.

There are four predefined cell styles you can use: **basic, left detail, right detail**, and **subtitle**. All the predefined cells have a `textLabel`, which is a `UILabel`, and an `accessoryView`, which is a `UIView`. The right detail, left detail, and subtitle styles add a `UILabel` named `detailTextLabel`. The basic, right detail, and subtitles also include a `UIImageView` named `imageView`.

<H3> [Displaying a `UITableViewCell`]

Source: [Beginning of iOS Programming] [Chapter No 5] [109-110]

---

1. Select the `Main.storyboard` file from the Project Navigator.
2. Select the **Table View Cell** from the Storyboard hierarchy, as shown in **Figure 5**.

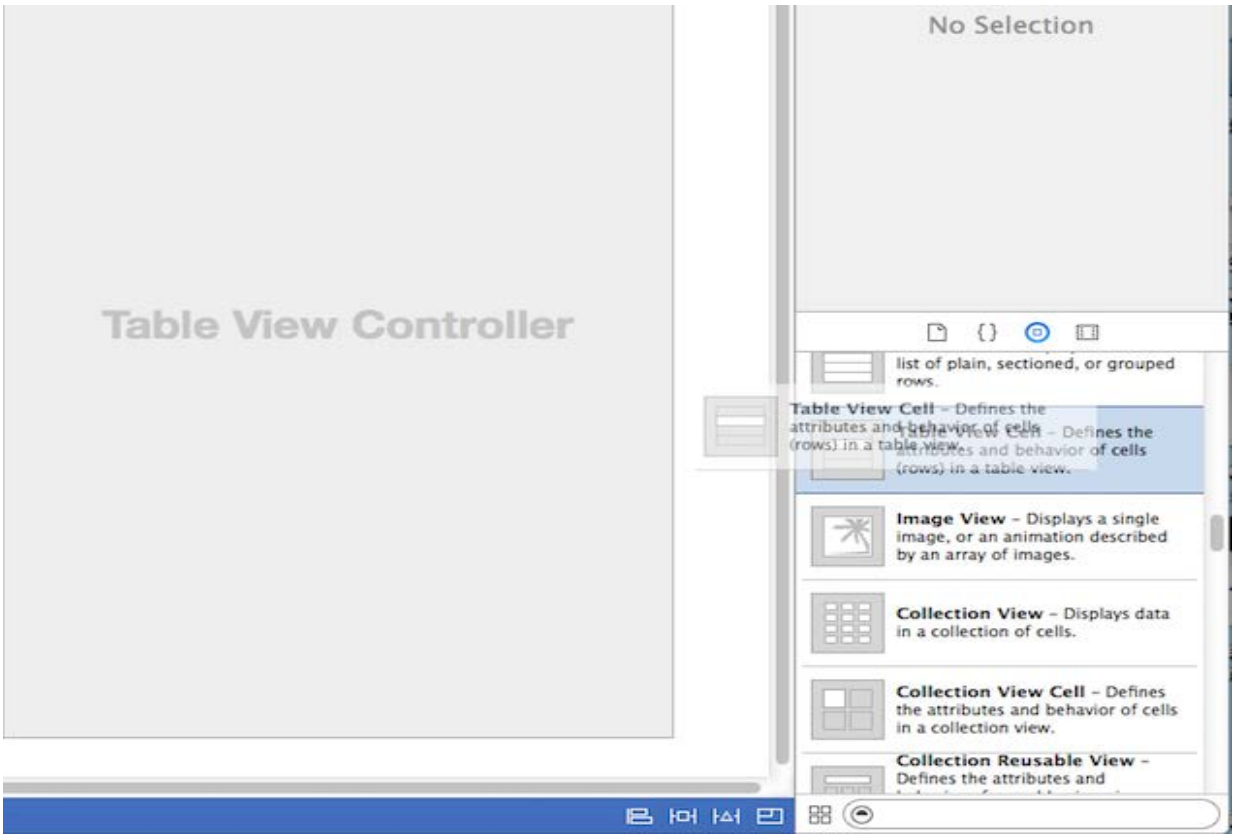

Figure 5: Add Table View Cell to the User Interface

3. In the Attributes inspector, set the **Style** of the cell to **Basic** and the **Identifier** to **Cell**, as shown in

Figure 6.

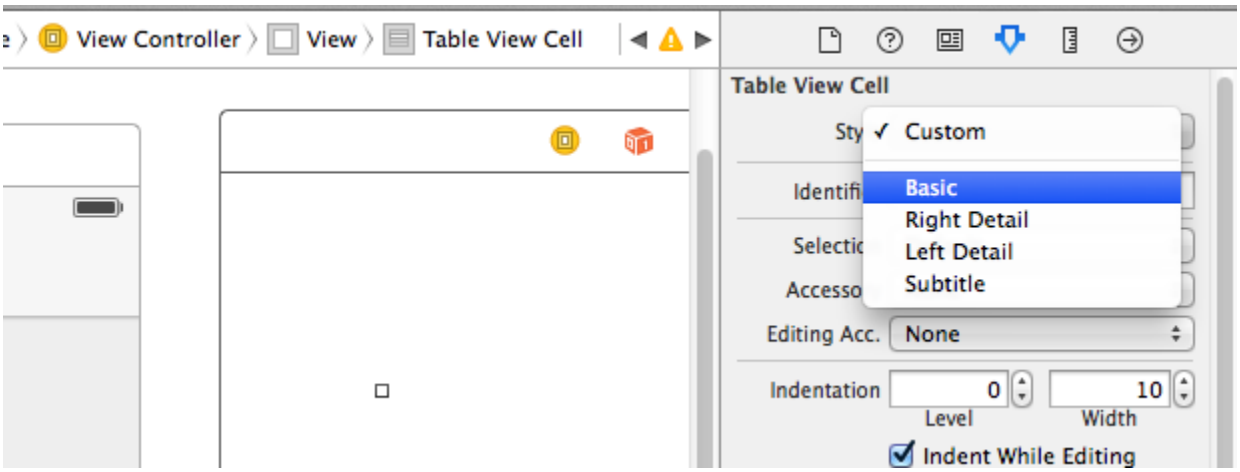

Figure 6: Set the Style as Basic and Identifier as Cell

4. Select the `WBABandsListTableViewController.m` file in the Project Navigator.

5. Find the `numberOfSectionsInTableView:` method and change the return value to `1`, as shown in the following code. This method tells the `UITableView` that there is one section.

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
      // Return the number of sections.
      return 1;
}
```

6. Find the `tableView:numberOfRowsInSection:` method and change the return value to `10`, as shown in the following code. This method tells the `UITableView` that there are 10 rows in that section.

```
- (NSInteger)tableView:(UITableView *)tableView
numberOfRowsInSection:(NSInteger)section
{
      // Return the number of rows in the section.
      return 10;
}
```

7. Find the `tableView:cellForRowAtIndexPath:` method and add the following code. This method dequeues a `UITableViewCell` with an identifier of "Cell" if one exists, or creates a new one. The code then sets the `textLabel` to the row number of the `indexPath`. An `NSIndexPath` simply has the section number and row number of the cell the table is going to display.

```
- (UITableViewCell *)tableView:(UITableView *)tableView
```

```
cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
      static NSString *CellIdentifier = @"Cell";
      UITableViewCell *cell =
      [tableView dequeueReusableCellWithIdentifier:CellIdentifier
      forIndexPath:indexPath];
      // Configure the cell...
      cell.textLabel.text = [NSString stringWithFormat:@"%d",
      indexPath.row];
      return cell;
}
```

8. Run the app in the **iPhone 4-inch simulator**. You can see 10 numbered cells (0–9) in the table, as shown in **Figure 7**.

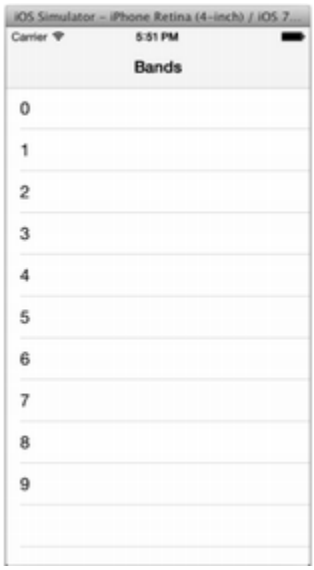

Figure 7: Run the App in Simulator

When you work with Swift, you create the `ViewController` class for creating cells, setting the number of cells, and handling table cells. The **Swift** format for these operations is as follows:

```
//code to set Number of Rows
class ViewController: UIViewController, UITableViewDelegate,
UITableViewDataSource
      {
      func tableView(tableView: UITableView , numberOfRowsInSection section:
Int) -> Int {
      return self.items.count;
```

```
      }

// To Create a cell
class       ViewController:       UIViewController,       UITableViewDelegate,
UITableViewDataSource
      {
      ….
func   tableView(tableView:   UITableView,   cellForRowAtIndexPath   indexPath:
NSIndexPath) -> UITableViewCell {
      var                        cell:UITableViewCell                        =
self.tableView.dequeueReusableCellWithIdentifier("cell") as UITableViewCell
      cell.textLabel?.text = self.items[indexPath.row]
      return cell
      }
// Adding String to cell
class       ViewController:       UIViewController,       UITableViewDelegate,
UITableViewDataSource {
      ….
      @IBOutlet var tableView: UITableView
      var items: [String] = ["One", "Two", "Three"]
```

Additional Knowhow

If you run an app and the `UITableView` does not display the cells correctly, make sure you have the identity of the table set to the `WBABandsListTableViewController` class and that the `dataSource` and delegate are connected correctly.

<H1> [Implementing the Bands Data Source]

Source: [Beginning of iOS Programming][Chapter No 5][110]

The **Bands** data source is a repository from which the `UITableView` gets data and reflects it in the table form. You probably already know how to add a single `WBABand` object and store it using `NSUserDefaults`. In this section, you will expand on that by storing as many `WBABand` objects as the user wants to add to the app. You also need to keep in mind how the bands will display in the `UITableView`. This way you can use the storage as the data source for a table.

<H2> [Creating the Bands Storage]

Source: [Beginning of iOS Programming][Chapter No 5][110]

The easiest way of storage is an NSMutableDictionary, which is a key/value data storage object. For the **Bands** storage, the first letter of the bands is the key, and the value is an NSMutableArray with all the bands that have that first letter.

Because you create sections of bands by the first letters of their names, it also makes sense to sort them in alphabetical order. To do this, you need a way to compare two bands by their first names. All subclasses of NSObject have the compare: method. You need to override this method in the WBABand class to compare bands by their names.

Finally, you also need to implement another NSMutableArray of the first letters used. You will learn its reason in the *Implementing Sections and Index* section of this session, but it makes sense to implement the code for it now while adding the code for the WBABand data storage.

<H3> [Adding Band Object Storage]

1. Select the WBABand.m file from the Project Navigator, and add the following code to the implementation:

```
- (NSComparisonResult)compare:(WBABand *)otherObject
{
      return [self.name compare:otherObject.name];
}
```

2. Select the WBABandsListTableViewController.h file from the Project Navigator, and add the following code:

```
@class WBABand;
@interface WBABandsListTableViewController : UITableViewController

@property(nonatomic,strong)NSMutableDictionary *bandsDictionary;
@property (nonatomic, strong) NSMutableArray *firstLettersArray;

- (void)addNewBand:(WBABand *)WBABand;
- (void)saveBandsDictionary;
- (void)loadBandsDictionary;

@end
```

3. Select the WBABandsListTableViewController.m file from the Project Navigator, and add the WBABand.h file to the imports with the following code:

```
#import “WBABand.h”
```

4. Add the following code before the implementation:
```
static NSString *bandsDictionarytKey = @"BABandsDictionarytKey";
```

5. Add the following code to the implementation:

```
- (void)addNewBand:(WBABand *)bandObject
{
      NSString *bandNameFirstLetter = [bandObject.name substringToIndex:1];
      NSMutableArray *bandsForLetter = [self.bandsDictionary
objectForKey:bandNameFirstLetter];

      if(!bandsForLetter)
            bandsForLetter = [NSMutableArray array];

      [bandsForLetter addObject:bandObject];
      [bandsForLetter sortUsingSelector:@selector(compare:)];
      [self.bandsDictionary                          setObject:bandsForLetter
      forKey:bandNameFirstLetter];

      if(![self.firstLettersArray containsObject:bandNameFirstLetter])
      {
            [self.firstLettersArray addObject:bandNameFirstLetter];
            [self.firstLettersArray sortUsingSelector:@selector(compare:)];
            }

            [self saveBandsDictionary];
      }

- (void)saveBandsDictionary
{
      NSData *bandsDictionaryData = [NSKeyedArchiver
archivedDataWithRootObject:self.bandsDictionary];
      [[NSUserDefaults standardUserDefaults] setObject:bandsDictionaryData
forKey:bandsDictionarytKey];
}

- (void)loadBandsDictionary
{
      NSData *bandsDictionaryData = [[NSUserDefaults standardUserDefaults]
objectForKey:bandsDictionarytKey];

      if(bandsDictionaryData)
      {
            self.bandsDictionary = [NSKeyedUnarchiver
      unarchiveObjectWithData:bandsDictionaryData];
            self.firstLettersArray = [NSMutableArray
      arrayWithArray:self.bandsDictionary.allKeys];
            [self.firstLettersArray sortUsingSelector:@selector(compare:)];
```

```
        }
        else
        {
                self.bandsDictionary = [NSMutableDictionary dictionary];
                self.firstLettersArray = [NSMutableArray array];
        }
}
```

6. Modify the `viewDidLoad` method with the following code:

```
- (void)viewDidLoad
{
        [super viewDidLoad];

        [self loadBandsDictionary];
}
```

Additional Knowhow

In Objective-C, `NSObject` is the base class, but **Swift** uses Objective-C class as its base class. In **Swift**, you provide Objective-C runtime metadata for the class implementation. If you subclass `NSObject` in your **Swift** project, you get the Objective-C runtime flexibility. It should be noted that if you avoid Objective-C flexibility, you can improve the performance of your **Swift** project. This happens so because the performance of **Swift** decreases when Objective-C libraries are used. However, if the requirement is not performance but quick application development, then Objective C libraries can be used in **Swift**. It should also be noted that the hardware of smartphones these days is very powerful and the performance issue is seen for very critical processes only.

Technical Stuff

To implement the `addNewBand:` method, you can use the following steps:

1. Get the first letter of the band name using the substring method, which takes a string value from `NSString`.
2. Next, you look in the `NSMutableDictionary` dictionary to see if you already have a band with that first letter. If so, then find an `NSMutableArray` for the letter; else you need to create a new one.
3. Then, add the band to the array and sort it.
4. Finally, look for the first letter in the `firstLettersArray`. If it is not there, you add it in and then sort that array as well.

Quick Tip

You can declare the `NSMutableDictionary` to hold all the bands and the `NSMutableArray` to store the first letters of the band names.

<H2> [Adding Bands]

Source: [Beginning of iOS Programming][Chapter No 5][113]

You already know how to build the user interface by adding a band. However, when you add the Navigation Controller to a project, the scene in the storyboard disappears. Instead you can present the scene from the `WBABandsListTableViewController`. When the users want to add a new band, they will tap a button that you will place on the `UINavigationItem`.

First, you need to make some modifications to the `ViewController` class, starting with renaming it according to the Apple naming conventions. Renaming a class can be tricky, since you need to find every place in the project where the name is used and change it. The Refactor feature in Xcode finds all of them for you. **Refactoring** is a feature of Xcode that renames the methods and classes and enables those changes to be applied across your entire codebase.

Additional Knowhow

In addition to renaming methods and classes, some of the major refactoring actions are as follows:

- **Extract:** You can extract a piece of selected code into a new method or function. Refactoring enables you to smartly determine what parameters and return values to give the new method based on the surrounding code.
- **Move Up:** Refactoring enables you to move a method, property, or instance variable up to its super class.
- **Move Down:** This is the inverse of Move Up, but it only works with instance variables.
- **Encapsulate**: If you want to convert all uses of an instance variable into accessor methods, you will be able to encapsulate that instance variable behind those methods.

<H3>[Renaming a Class Using Xcode Refactoring]

Source: [Beginning of iOS Programming][Chapter No 5][114]

---

1. Select the `ViewController.h` class from the Project Navigator.
2. Right click on the `ViewController` class name and select **Refactor ➪ Rename** from the right-click menu, as shown in **Figure 8**.

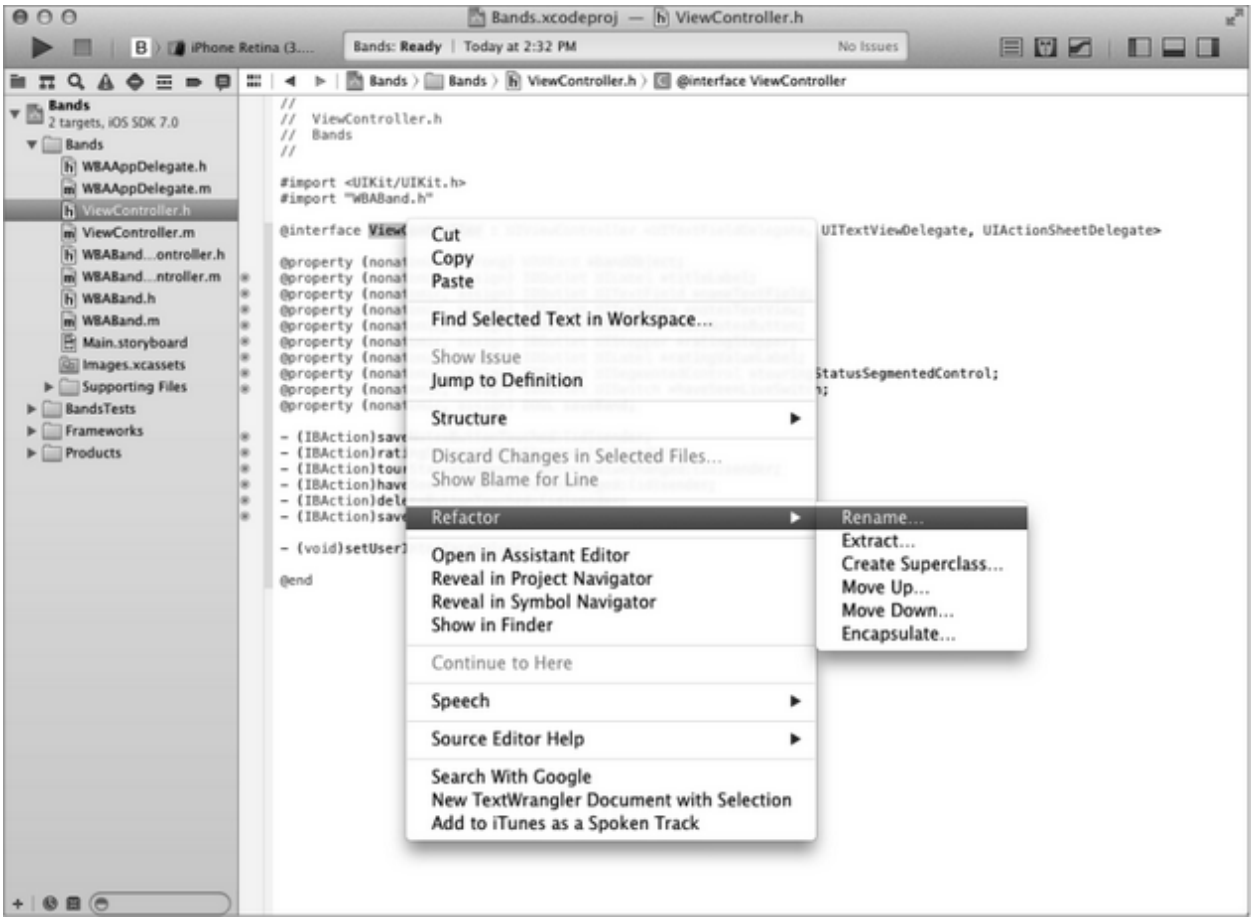

Figure 8: Select Refactor -> Rename from the `ViewController` Right-Click Menu

3. Rename the `WBABandDetailsViewController` class and click **Preview**. You can see how many places are being changed in the preview.
4. Review the changes and click **Save**.
5. Xcode will prompt you to enable snapshots, which are a lightweight type of version control. Whether or not you want to use them is up to you.
6. Compile the project and verify there are no errors.

<H2> [Displaying Bands]

Source: [Beginning of iOS Programming][Chapter No 5][119]

Now that you have implemented the **Bands** data source along with a way to add new bands, it is time to display the bands in the `UITableView`. Most of the hard work is done. You just need to modify the `UITableViewDataSource` methods to use the **Bands** dictionary and reload the `UITableView`

data when a new band is added.

<H3>[Displaying Band Names]

Source: [Beginning of iOS Programming][Chapter No 5][120]

---

1. Select the `WBABandsListTableViewController.m` file from the Project Navigator.
2. Modify the `numberOfSectionsInTableView:` method with the following code:

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
      // Return the number of sections.
      return self.bandsDictionary.count;
}
```

3. Modify the `tableView:numberOfRowsInSection:` method with the following code:

```
- (NSInteger)tableView:(UITableView *)tableView
numberOfRowsInSection:(NSInteger)section
{
      // Return the number of rows in the section.
      NSString *firstLetter = [self.firstLettersArray objectAtIndex:section];
      NSMutableArray *bandsForLetter = [self.bandsDictionary
objectForKey:firstLetter];
      return bandsForLetter.count;
}
```

4. Modify the `tableView:cellForRowAtIndexPath:` method with the following code:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath
{

static NSString *CellIdentifier = @"Cell";
UITableViewCell *cell = [tableView
dequeueReusableCellWithIdentifier:CellIdentifier forIndexPath:indexPath];

NSString *firstLetter = [self.firstLettersArray
objectAtIndex:indexPath.section];
NSMutableArray *bandsForLetter = [self.bandsDictionary
objectForKey:firstLetter];
WBABand *bandObject = [bandsForLetter objectAtIndex:indexPath.row];
```

```
// Configure the cell...
cell.textLabel.text = bandObject.name;

return cell;
}
```

5. Modify the `viewWillAppear:` method with the following code:

```
            - (void)viewWillAppear:(BOOL)animated
            {
                  [super viewWillAppear:animated];

                  if(self.bandDetailsViewController)
                  {
                        if(self.bandDetailsViewController.saveBand)
      {
            [self addNewBand:self.bandDetailsViewController.bandObject];
            [self.tableView reloadData];
      }
      self.bandDetailsViewController = nil;
      }
}
```

6. Run the app in the iPhone 4-inch simulator. When you add a new band, it displays in the table, as shown in **Figure 9**.

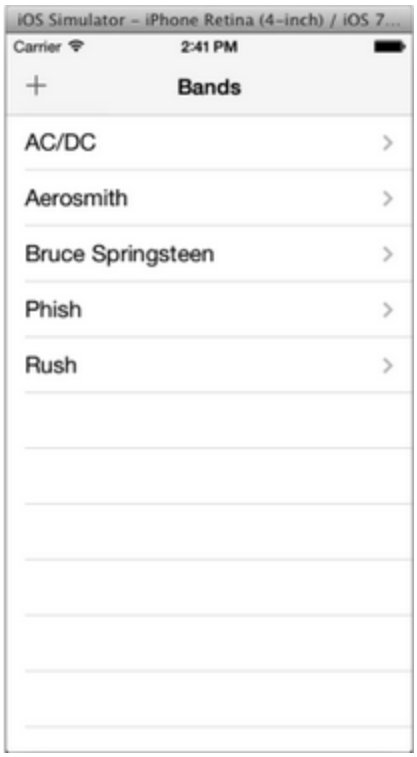

Figure 9: Run the App in Simulator

Additional Knowhow

In the `WBABandDetailsViewController`, you can change how it is dismissed based on if it has a `UINavigationController`. If it does, you can call the `popViewControllerAnimated:` method to return back to the `WBABandsListViewController`. When it appears, it looks to see if it has a row selected. If it does, it will either update a band if `saveBand` is true or delete the band if not.

<H1> [Implementing Sections and an Index]

Source: [Beginning of iOS Programming][Chapter No 5][121]

As users of the **Bands** app add more and more bands, it becomes harder to find the band they want in the `UITableView`. To help them out, you have already sorted the bands alphabetically. Adding sections and the section index can help them as well. The sections break up the bands visually, while the index gives the users a shortcut to jump to different sections.

<H2> [Adding Section Headers]

Source: [Beginning of iOS Programming][Chapter No 5][122]

Section headers are simple to add because of the data storage architecture you implemented using both the dictionary and the first letters array. There is a single method in the `UITableViewDataSource` protocol, which returns the section headers for the section.

<H3>[Displaying the First Letter Section Headers]

Source: [Beginning of iOS Programming][Chapter No 5][122]

---

1. Select the `WBABandsListTableViewController.m` file from the Project Navigator.
2. Add the following code to the implementation:

```
- (NSString *)tableView:(UITableView *)tableView
titleForHeaderInSection:(NSInteger)section
{
      return [self.firstLettersArray objectAtIndex:section];
}
```

In **Swift**, you can define a function for implementing headers as follows:

```
func tableView(tableView : UITableView , titleForHeaderInSection
section: Int) -> String
            {     ....
                  return [self.firstLettersArray objectAtIndex:section]
            }
```

3. Run the app in the iPhone 4-inch simulator. You will see section headers corresponding to the first letters of the band names, as shown in **Figure 10**.

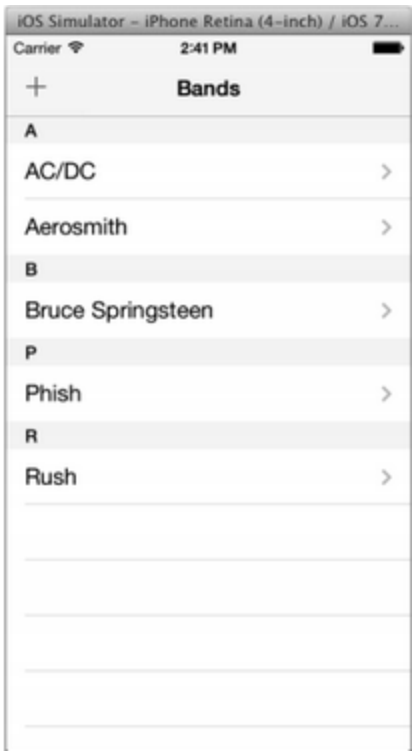

Figure 10: Run the App in Simulator

<H2> [Enabling Edit Mode]

Source: [Beginning of iOS Programming][Chapter No 5][124]

`UITableView` has a built-in edit mode. `UITableViewController` also has a built-in button, which you can add to the `UINavigationItem` to toggle into an edit mode named `editButtonItem`. Since the `WBABandsListTableViewController` is a subclass of the `UITableViewController`, you can access the `editButtonItem` from `self`.

When a table goes into edit mode, it asks its delegate which rows are editable. You can control which rows are editable by implementing the `tableView:canEditRowAtIndexPath:` method. If this method is not implemented, the table will not allow any of the rows to be editable.

<H3>[Implementing the Allow Edit Method]

Source: [Beginning of iOS Programming][Chapter No 5][124]

---

1. Select the `WBABandsListTableViewController.m` file from the Project Navigator.

2. Add the following code to the `viewDidLoad` method:

```
- (void)viewDidLoad
{
      [super viewDidLoad];
      [self loadBandsDictionary];
}
self.navigationItem.rightBarButtonItem = self.editButtonItem;
```

3. Add the following method to the implementation:

```
- (BOOL)tableView:(UITableView *)tableView canEditRowAtIndexPath:
(NSIndexPath *)indexPath
{
      return YES;
}
```

In **Swift**, you can find `viewDidLoad` as an override function in `ViewController.`**`Swift`** as follows:

```
override func viewDidLoad() {
      super.viewDidLoad()
self.navigationItem.rightBarButtonItem = self.editButtonItem

// For editing row
override func tableView (tableView: UITableView, canEditRowAtIndexPath
indexPath:NSIndexPath) -> Bool
{
      return YES;
}
```

4. Run the app in the iPhone 4-inch simulator. When you tap the **Edit** button, you will see the delete option in each cell next to the band name, as shown in **Figure 11**.

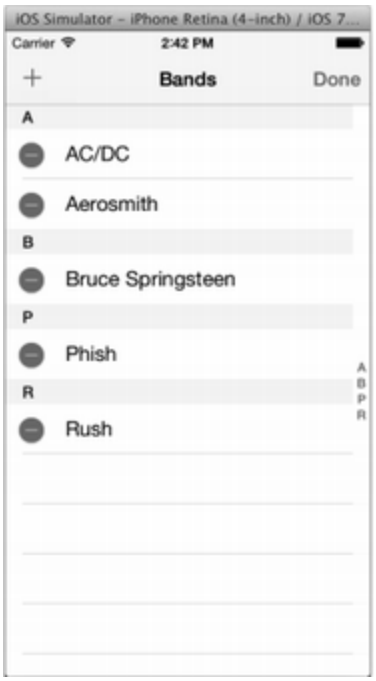

Figure 11: Run App in Simulator

<H2> [Modifying Data]

Source: [Beginning of iOS Programming][Chapter No 5][126]

Users of the **Bands** app might want to modify band information. The best way to do this is to show the Band Details scene when the user taps the band name in the `UITableView`. You can implement a segue in the storyboard to do this.

A **segue** is a way of transitioning from one view to the next. The two most common segues are:

- **Modal Segue:** A modal segue presents the view over the parent view. This is just as using the `presentViewController:animated:completion:` method, as you did with the Add button. You could have used a segue for this, but learning how to present and dismiss views in code is a valuable lesson.
- **Push Segue:** A push segue can be used with a `UINavigationController`. The view slides in from the right and adds a `UINavigationItem` to the top of the view with a Back button, which enables the user to return to the parent `UIViewController`. It does this by using a navigation stack. As you segue from one `UIViewController` to the next, the `UIViewControllers` get pushed onto the navigation stack. The Back button pops each `UIViewController` off until you get back to the `rootUIViewController`. This makes it

easy for users to know where they are in the app.

<H3> [Implementing a Push Segue]

1. Select the `Main.storyboard` file from the Project Navigator.
2. In the Band Details scene, move all the subviews down to 20 pixels**,** except for the Save and Delete buttons, as shown in **Figure 12**.

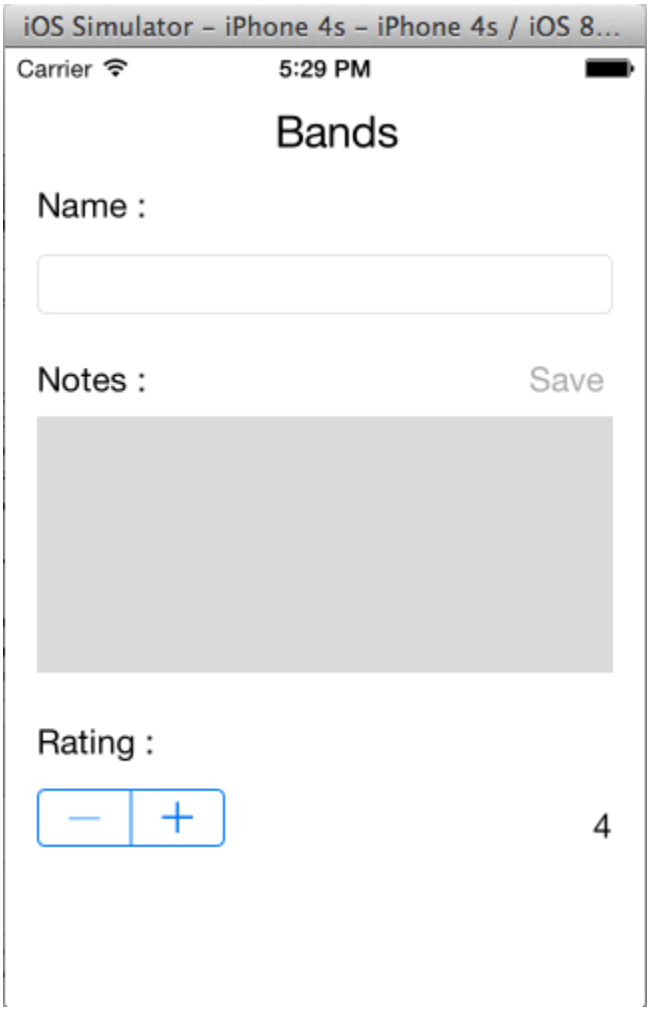

Figure 12: Move all Subviews to 20 Pixels

3. Control-drag from the **Prototype Cell** to the Band Details scene, as shown in **Figure 13**.

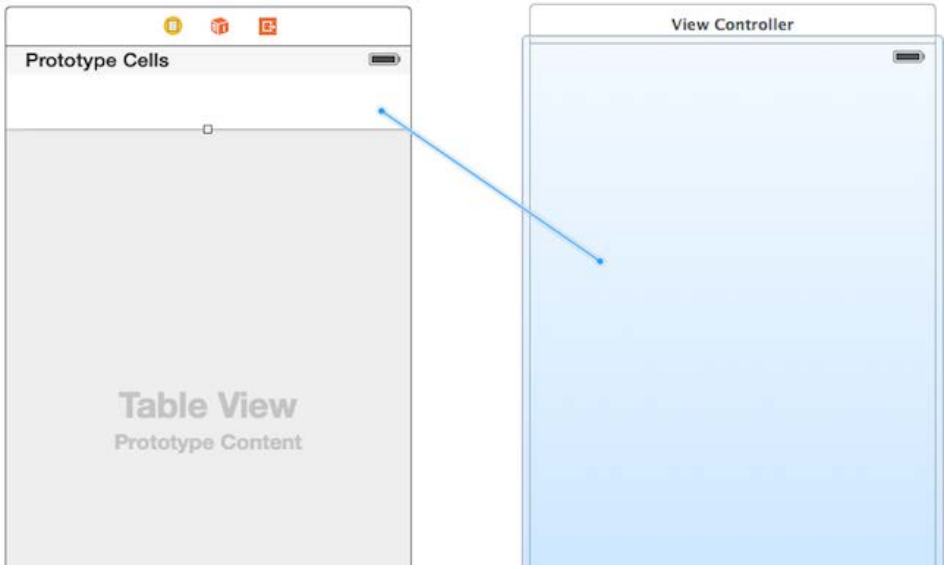

Figure 13: Control-Drag from the Prototype Cell to the Band Details Scene

4. Select **Push** from the segue pop-up menu that appears, as shown in **Figure 14**. The segue is represented by an arrow from the Bands List scene to the Band Details scene. Now the Band Details scene also has a `UINavigationItem`.

Figure 14: Select Push from the Segue Pop-Up Menu

5. Select the `UINavigationItem` and set its title to **Band** in the Attributes inspector.
6. Select the `WBABandListTableViewController.h` file from the Project Navigator and add the following code to the interface:

```
@class WBABand, WBABandDetailsViewController;
@interface WBABandsListTableViewController : UITableViewController

@property (nonatomic, strong) NSMutableDictionary *bandsDictionary;
@property (nonatomic, strong) NSMutableArray *firstLettersArray;
@property (nonatomic, strong) WBABandDetailsViewController
*bandDetailsViewController;

- (void)addNewBand:(WBABand *)bandObject;
- (void)saveBandsDictionary;
- (void)loadBandsDictionary;
- (void)deleteBandAtIndexPath:(NSIndexPath *)indexPath;
-(void)updateBandObject:(WBABand *)bandObject
atIndexPath: (NSIndexPath *)indexPath;
```

```
- (IBAction)addBandTouched:(id)sender;

@end
```

7. Select the `WBABandListTableViewController.m` file from the Project Navigator.
8. Add the following code to the `viewDidLoad` method:

```
- (void)viewDidLoad
{
      [super viewDidLoad];

      [self loadBandsDictionary];

      self.navigationItem.rightBarButtonItem = self.editButtonItem;
      self.clearsSelectionOnViewWillAppear = NO;
}
```

9. Modify the `viewWillAppear:` method with the following code:

```
- (void)viewWillAppear:(BOOL)animated
{
      [super viewWillAppear:animated];
      if(self.bandDetailsViewController)
      {
            NSIndexPath       *selectedIndexPath       =       [self.tableView
            indexPathForSelectedRow];

            if(self.bandDetailsViewController.saveBand)
            {
                  if(selectedIndexPath)
                  {
                        [self
                        updateBandObject:self.bandDetailsViewController.bandO
                        bject atIndexPath:selectedIndexPath];
                        [self.tableView
                        deselectRowAtIndexPath:selectedIndexPathanimated:YES]
                        ;
                  }
                  else
                        [self
                        addNewBand:self.bandDetailsViewController.bandObject]
                        ;
                  [self.tableView reloadData];
            }
            else if (selectedIndexPath)
```

```
            {
                  [self deleteBandAtIndexPath:selectedIndexPath];
            }
            self.bandDetailsViewController = nil;
      }
}
```

10. Add the following methods to the implementation:

```
- (void)updateBandObject:(WBABand *)bandObject
atIndexPath:(NSIndexPath *)indexPath
{
      NSIndexPath         *selectedIndexPath         =         [self.tableView
      indexPathForSelectedRow];
      NSString *sectionHeader = [self.firstLettersArray
objectAtIndex:selectedIndexPath.section];
      NSMutableArray *bandsForSection = [self.bandsDictionary
objectForKey:sectionHeader];
      [bandsForSection removeObjectAtIndex:indexPath.row];
      [bandsForSection addObject:bandObject];
      [bandsForSection sortUsingSelector:@selector(compare:)];
      [self.bandsDictionary setObject:bandsForSection forKey:sectionHeader];
      [self saveBandsDictionary];
}
- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender
{
      NSIndexPath         *selectedIndexPath         =         [self.tableView
      indexPathForSelectedRow];
      NSString *sectionHeader = [self.firstLettersArray
objectAtIndex:selectedIndexPath.section];
      NSMutableArray *bandsForSection = [self.bandsDictionary
objectForKey:sectionHeader];
      WBABand            *bandObject            =            [bandsForSection
      objectAtIndex:selectedIndexPath.row];
      self.bandDetailsViewController = segue.destinationViewController;
      self.bandDetailsViewController.bandObject = bandObject;
      self.bandDetailsViewController.saveBand = YES;
}
```

11. Select the `WBABandViewController.m` file from the Project Navigator.

12. Modify the `saveButtonTouched:` method with the following code:

```
- (IBAction)saveButtonTouched:(id)sender
{
      if(!self.bandObject.name || self.bandObject.name.length == 0)
      {
            UIAlertView *noBandNameAlertView = [[UIAlertView alloc]
initWithTitle:@"Error" message:@"Please supply a name for the band"
delegate:nil cancelButtonTitle:@"OK" otherButtonTitles:nil];
            [noBandNameAlertView show];
      }
      else
      {
            self.saveBand = YES;

            if(self.navigationController)
                  [self.navigationController popViewControllerAnimated:YES];
            else
                  [self dismissViewControllerAnimated:YES completion:nil];
      }
}
```

13. Modify the `actionSheet:clickedButtonAtIndex:` method with the following code:

```
- (void)actionSheet:(UIActionSheet *)actionSheet
clickedButtonAtIndex:(NSInteger)buttonIndex
{
      if(actionSheet.destructiveButtonIndex == buttonIndex)
      {
            self.bandObject = nil;
            self.saveBand = NO;

            if(self.navigationController)
                  [self.navigationController popViewControllerAnimated:YES];
            else
                  [self dismissViewControllerAnimated:YES completion:nil];
      }
}
```

14. Run the app in the iPhone 4-inch simulator (see **Figure 15** and **Figure 16**). You can now see the `accessoryView` for each row is set to a chevron. When a row is selected, you can segue to the Band Details scene with all the UI objects set according to the `bandObject`.

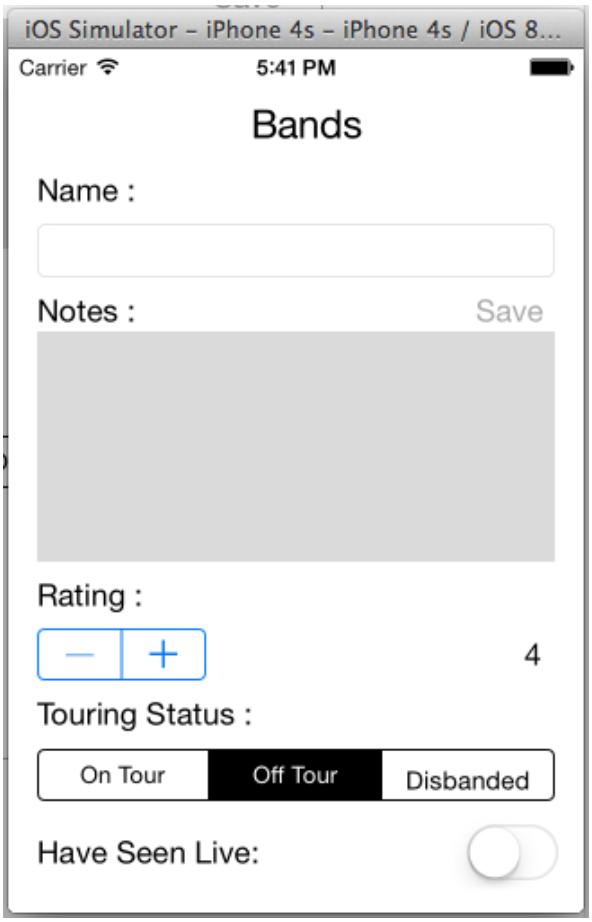

Figure 15: Run Bands on Simulator

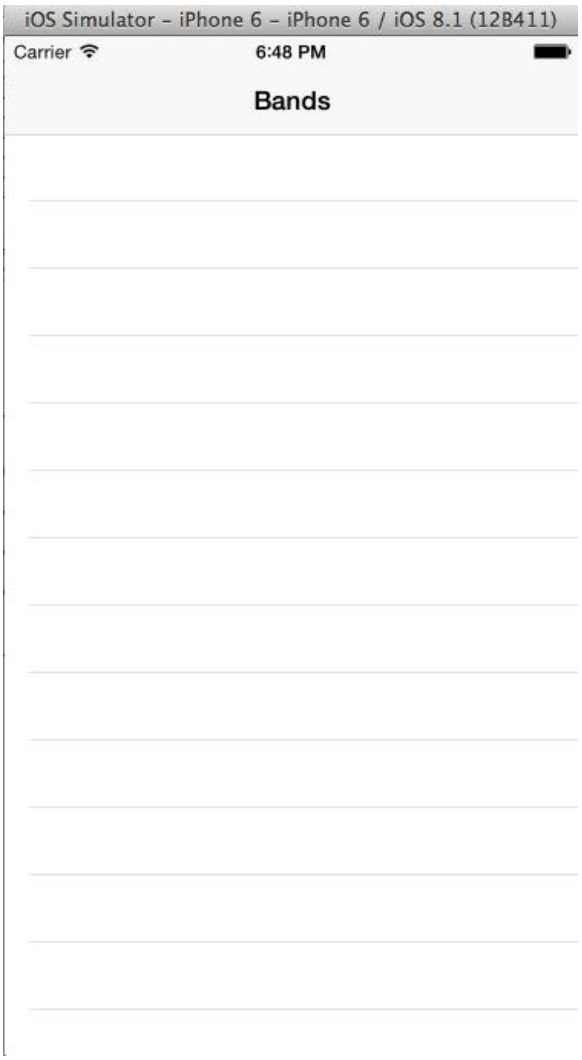

Figure 16: Bands List Represented as Table View

The Big Picture

An application is effective if it is performing well. If an app has interesting animation effects, then the users will use that app with much more interest. So the view part of an app is important. An app should also be good from the data storage perspective, as users do not want to enter values again and again. An app should have an effective form that will attract people to enter details; this is the impact of the user interface of an app.

Additional Knowhow

When an app runs, tapping a cell starts the segue, which first calls the `prepareForSegue:sender:` method. In this method, you get the `NSIndexPath` of the selected row in the `tableView` and set the `bandObject` of the `WBABandDetailsViewController` with the `bandObject` in the data source.

<H1> [Asking for User Input]

Source: [Professional IOS Programming][Chapter No 4][119]

Sometimes your application logic requires that you either attract the user's attention with a message he/she cannot ignore, or present the user with one or more options to choose from. Based on the choice the user has made, your application flow can continue. For example, you want to give a popup message in the **Bands** app, which indicates that a new song has been added to the playlist.

Within iOS, you can use two main objects for this purpose:

- **`UIActionSheet`:** As the name implies, this is a sheet that requires the user to select an action.
- **`UIAlertView`:** This notifies the user with a title and a message. The `UIAlertView` remains visible until the user presses one of the presented button(s) that will remove the `UIAlertView` from the presentation stack.

Both the `UIActionSheet` and the `UIAlertView` are presented in a modal state to the user interface, blocking the user interface until the user has made a selection.

<H2> [Creating a `UIActionSheet` with Multiple Options]

Source: [Professional IOS Programming][Chapter No 4][120]

When you want the user of your application to make a choice from a series of actions that can be performed on the `UIViewController`, you can create a `UIActionSheet` with the options you want the user to choose from. Normally, you present a `UIActionSheet` if there is more than one action to choose from. Do not forget to also present a cancel action to the user so the application can return to its previous state if the user does not want to make a selection.

For example, in the **Bands** app, you wish to include a feature in which the users can categorize the genre of music, such as Rock, Metal, Romantic, and Devotional. For this purpose, you will have to use the `UIActionSheet`.

<H3> [Creating a `UIActionSheet` with a Single Option]

1. Create a new project in **Xcode** using the **Single View Application** template, and name it **PresentActionSheet**. In Xcode version 6, you can choose the Language as Objective-C or **Swift**, and select the **Use Core Data** check box for creating a database, as shown in **Figure 17**.

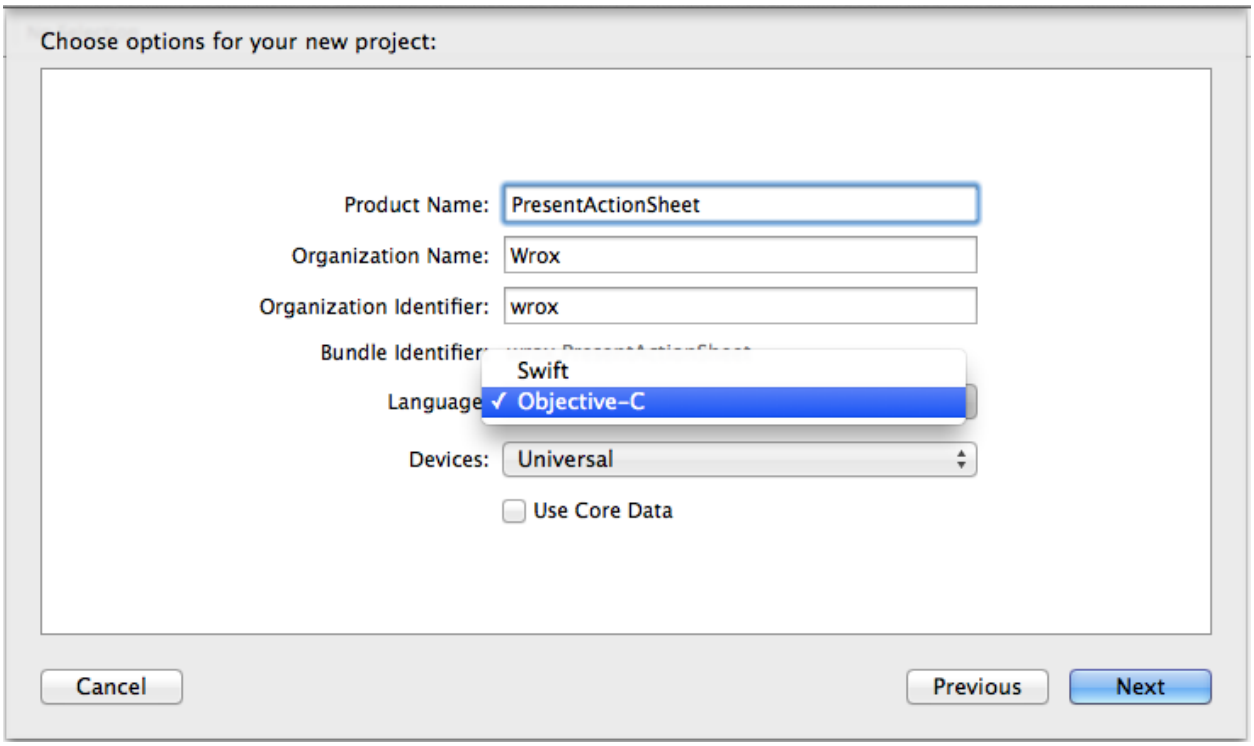

Figure 17: Create the PresentActionSheet Project

2. Open the `YDViewController.xib` file with Interface Builder and place a `UIButton` on top of the View as shown in **Figure 18**.

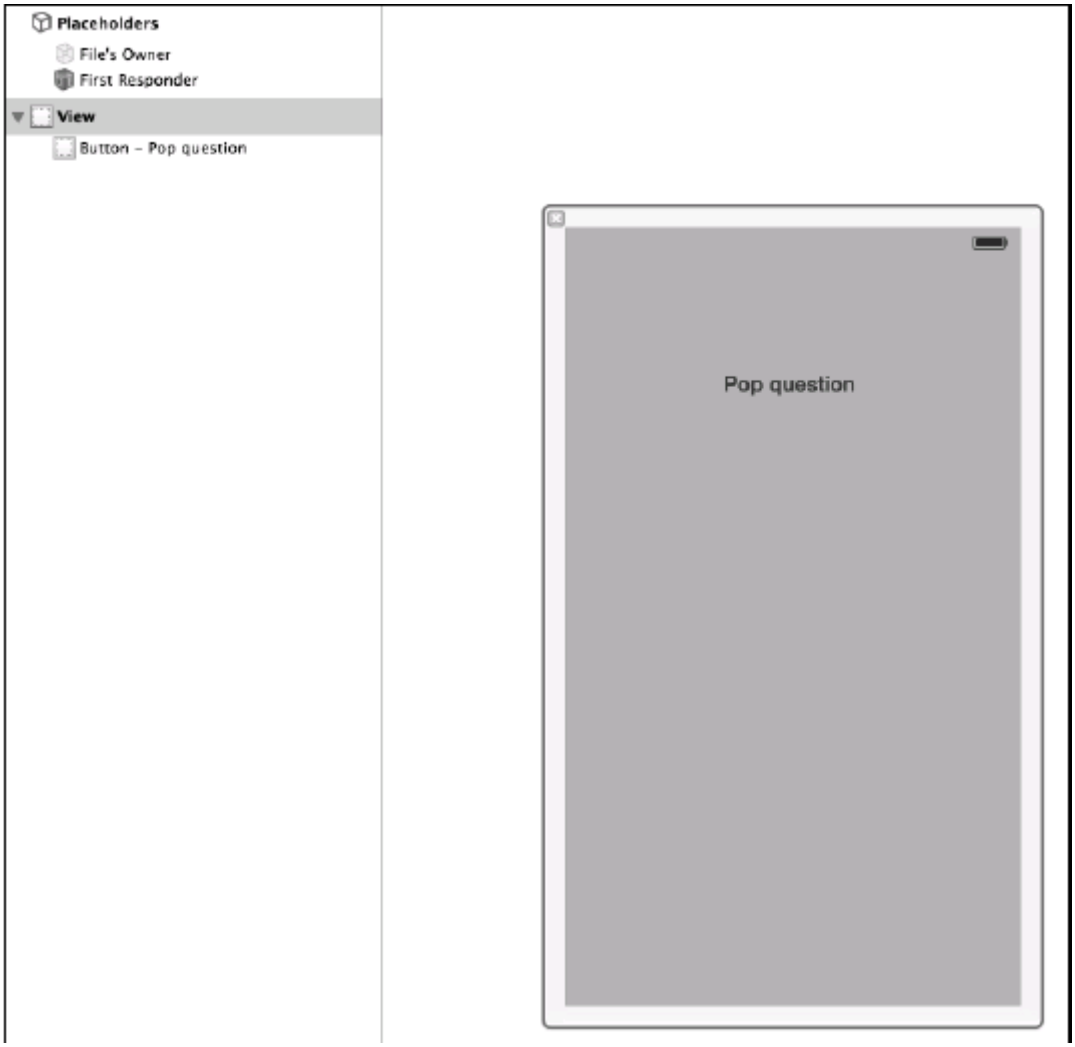

Figure 18: Place a `UIButton` on Top of the View

3. Using the Assistant Editor, create an `IBAction` declaration named **`popQuestion`** for this UIButton in the `YDViewController.m` file, as shown in **Figure 19**.

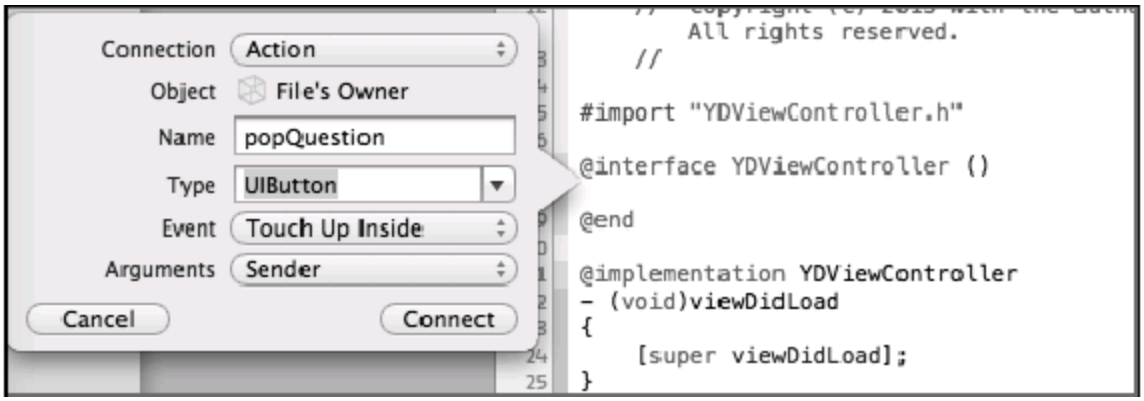

Figure 19: Create an `IBAction` Declaration in `YDViewController.m`

4. Open the `YDViewController.m` file and implement the code as shown below:

```
#import "YDViewController.h"

@interface YDViewController ()
- (IBAction)popQuestion:(UIButton *)sender;

@end

@implementation YDViewController
- (void)viewDidLoad
{
      [super viewDidLoad];
}

- (IBAction)popQuestion:(UIButton *)sender
{
      UIActionSheet *actionSheet = [[UIActionSheet alloc]
            initWithTitle:@"Choose action"
            delegate:nil
            cancelButtonTitle:nil
            destructiveButtonTitle:@"OK"
            otherButtonTitles:nil];
      actionSheet.actionSheetStyle = UIActionSheetStyleDefault;
      [actionSheet showInView:self.view]; // show from your view
}
- (void)didReceiveMemoryWarning
{
      [super didReceiveMemoryWarning];
```

```
        // Dispose of any resources that can be recreated.
}

@end
```

5. The `UIActionSheet` object is initialized in the `popQuestion:` method. For now, you do not have to set the delegate; you will learn how to do this in the section *Responding to User Input*. You present the action sheet by calling the `showInView:` method of the `UIActionSheet` class. (You will learn more about methods to present the `UIActionSheet` in the section *Presenting the `UIActionSheet`*).

6. Launch your application. When you click the **Pop question** button, you will get the result as shown in **Figure 20**.

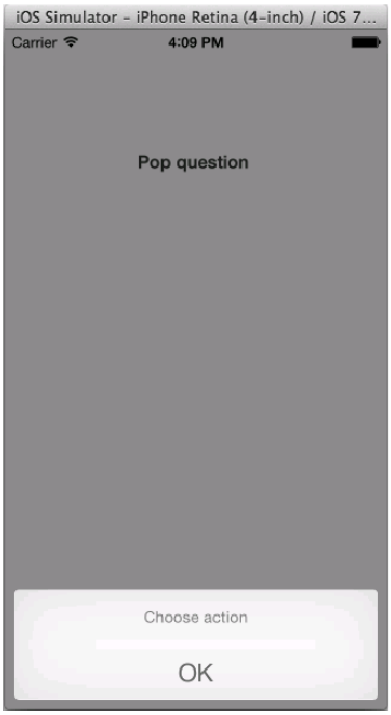

Figure 20: Run the Pop Question in Simulator

In this example, you created a `UIActionSheet` with a single button with the title @"OK" because that string has been assigned to the `destructiveButtonTitle`.

<H3> [Creating a `UIActionSheet` with Multiple Options]

In a real-world application, you would use the `UIActionSheet` to ask the user to choose an action from a series of possible options. To achieve this, create a new project in which you add more buttons to

the `UIActionSheet`. You can add buttons by calling the `addButtonWithTitle:` method of the `UIActionSheet` class. You have to make sure that you do not create more buttons than the application window can show, of course.

1. Start **Xcode** and create a new project using the **Single View Application** template. Name the project as **MultipleChoice**, as shown in **Figure 21**.

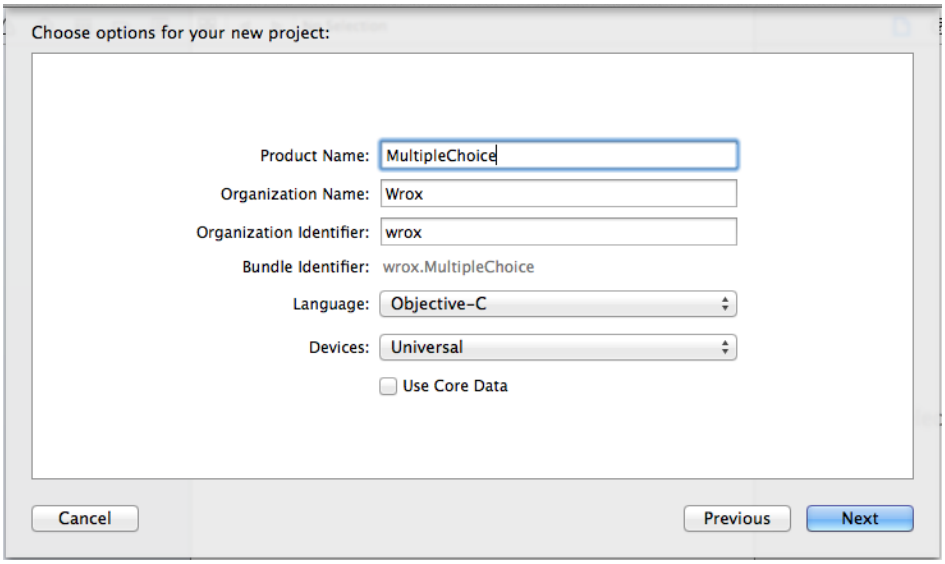

Figure 21: Create the MultipleChoice Project

2. Like you have done in the previous application, open the `YDViewController.xib` file with Interface Builder, create a `UIButton`, and use the Assistant Editor to create the `popQuestion:` declaration. The implementation of the `popQuestion:` method is different because you want to present multiple options to the user. For this, you use the `addButtonWithTitle:` method of the `UIActionSheet` class in Objective-C.

Swift does not approve of the `UIAlertView` and `UIActionSheet` classes and replaces them by `UIAlertController`. You can also use `UIAlertView`, `UIActionSheet`, and `UIAlertController` to provide interactions in the application.

3. Because you have multiple options, you need to set the `destructiveButtonIndex` to the index of the **Cancel** button. The index is determined based on the sequence of the `addButtonWithTitle:` methods you have called, where it is always a zero-based index. The implementation is shown below:

```
#import "YDViewController.h"
```

```
@interface YDViewController ()

- (IBAction)popQuestion:(UIButton *)sender;

@end

@implementation YDViewController

- (void)viewDidLoad
{
      [super viewDidLoad];
      // Do any additional setup after loading the view, typically from a
      nib.
}
-(IBAction)popQuestion:(id)sender
{
      UIActionSheet *actionSheet = [[UIActionSheet alloc]
                  initWithTitle:@"Choose action"
                  delegate:nil

            cancelButtonTitle:nil
            destructiveButtonTitle:nil
            otherButtonTitles:nil];
      [actionSheet addButtonWithTitle:@"Take picture"];
      [actionSheet addButtonWithTitle:@"Choose picture"];
      [actionSheet addButtonWithTitle:@"Take video"];
      [actionSheet addButtonWithTitle:@"Cancel"];
      //set the desctructiveButtonIndex to the button that is responsible for
      //the cancel function
      actionSheet.destructiveButtonIndex=3;
      [actionSheet showInView:self.view];
      // show from your view
}
- (void)didReceiveMemoryWarning
{
      [super didReceiveMemoryWarning];
      // Dispose of any resources that can be recreated.
}

@end
```

4. In **Swift** language, you use the same method for adding buttons, which is by dragging a UI button to the `ViewController.Swift` file and then adding the following code:

```
@IBAction func popQuestion(sender : UIButton) {
      var actionsheet: UIActionSheet
      actionsheet = UIActionSheet(title: “Choose action”, delegate: nil
,cancelButtonTitle: nil , destructiveButtonTitle= nil, otherButtonTitles: nil
)
      actionsheet.addButtonWithTitle(“Take picture”)
      actionsheet.addButtonWithTitle(“Choose picture”)
      actionsheet.addButtonWithTitle(“Take video”)
      actionsheet.addButtonWithTitle(“cancel”)
```

5. Launch the application. When you click the **Pop question** button, you will get the result as shown in **Figure 22**.

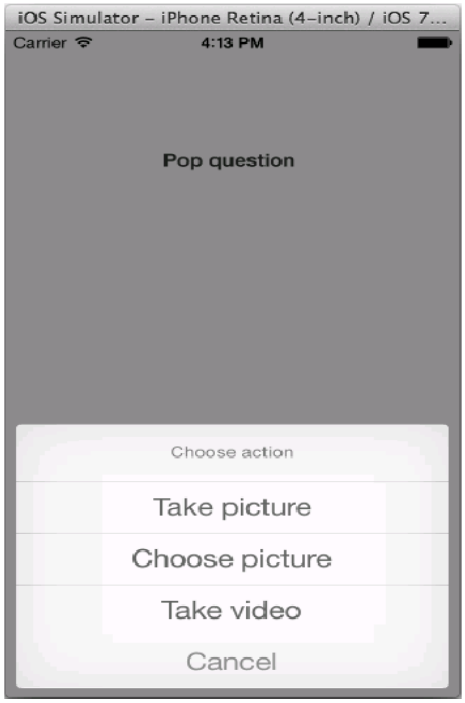

Figure 22: Run the App in Simulator

In this example, you learned how to create a `UIActionSheet` and how to add additional buttons to it. You used the `showInView:` method to present the `UIActionSheet`. In the next section, you will learn about different methods to present your `UIActionSheet`.

<H2> [Presenting the `UIActionSheet`]

Source: [Professional IOS Programming][Chapter No 4][125]

You can present a UIActionSheet to the user in several different ways related to the type of navigation your application is supporting.

The following methods are available for presenting the UIActionSheet:

- showInView
- showFromTabBar
- showFromBarButtonItem
- showFromRect inView
- showFromToolbar

All of the above methods are used to center the action sheet to the middle of a screen. In this section, you will learn about the showInView and the showFromTabBar methods in detail.

<H3>[Presenting with showInView]

Source: [Professional IOS Programming][Chapter No 4][125]

The showInView: method of the UIActionSheet class shows the actionSheet within the assigned View Container, as in the previous implementation (see **Figure 22**).

<H3>[Presenting with showFromTabBar]

Source: [Professional IOS Programming][Chapter No 4][125]

If your application is using a UITabBarController to manage its navigation stack, you can use the showFromTabBar: method to present your UIActionSheet.

1. Create a new project in Xcode, and choose the **Tabbed Application Project** template to create an application with a UITabBarController named **TabbedActionSheet** using the options shown in **Figure 23**. Make sure to select **Universal** from the **Devices** drop-down list because you need the ability to run this application on both an iPhone as well as an iPad.

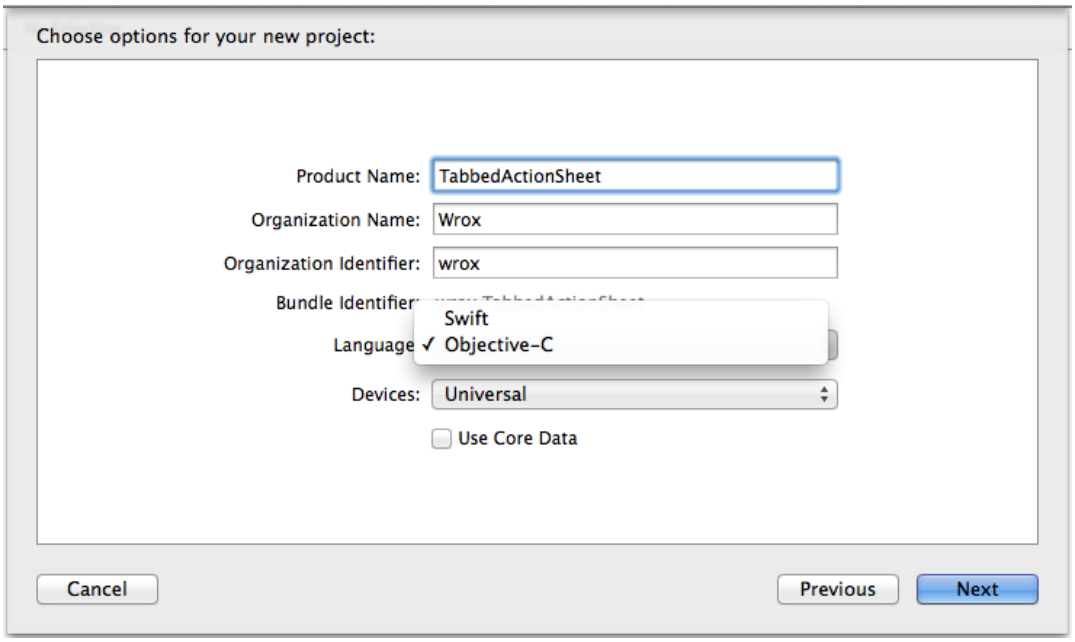

Figure 23: Create the TabbedActionSheet Project

2. The project template automatically creates the `FirstViewController` and `SecondViewController` classes as well as the Interface Builder files. Use Interface Builder and the Assistant Editor to place a `UIButton` with the title **Pop question** on the view for both `ViewControllers` and create the `popQuestion:` method declaration, like you've done in the previous samples.

3. In the following code, you will see the same logic as you have used before to create and initialize the `UIActionSheet` and add the buttons to it by calling the `addButtonWithTitle:` method of the `UIActionSheet` class.

```
#import "YDFirstViewController.h"

@interface YDFirstViewController ()
-(IBAction)popQuestion:(id)sender;
@end

@implementation YDFirstViewController

-(id)initWithNibName:(NSString *)nibNameOrNil bundle:(NSBundle
*)nibBundleOrNil
{
      self = [super initWithNibName:nibNameOrNil bundle:nibBundleOrNil];
      if (self) {
            self.title = NSLocalizedString(@"First", @"First");
```

```
            self.tabBarItem.image = [UIImage imageNamed:@"first"];
      }
      return self;
}

- (void)viewDidLoad
{
      [super viewDidLoad];
}
-(IBAction)popQuestion:(id)sender
{
      UIActionSheet *actionSheet = [[UIActionSheet alloc]
                  initWithTitle:@"Choose action"
                  delegate:nil
                  cancelButtonTitle:nil
                  destructiveButtonTitle:nil
                  otherButtonTitles:nil];
      [actionSheet addButtonWithTitle:@"Take picture"];
      [actionSheet addButtonWithTitle:@"Choose picture"];
      [actionSheet addButtonWithTitle:@"Take video"];
      [actionSheet addButtonWithTitle:@"Cancel"];
            //set   the   destructiveButtonIndex   to   the   button   that   is
            responsible
            //for the cancel function
      actionSheet.destructiveButtonIndex=3;
      [actionSheet showFromTabBar:self.tabBarController.tabBar];
}
- (void)didReceiveMemoryWarning
{
      [super didReceiveMemoryWarning];
      // Dispose of any resources that can be recreated.
}

@end
```

In the above code, you are presenting the `UIActionSheet` by calling the `showFromTabBar:` method, instead of the `showInView:` method of the `UIActionSheet` class. However, when you run the application on an iPhone or iPad device or simulator (see **Figure 24**), you will not see much difference compared to the `showInView:` method. This is because the `UIActionSheet` will cover the width of the device's screen and will be presented from the bottom.

Figure 24: Present `UIActionSheet` on the Simulator

So what is the reason for using the `showFromTabBar:` method instead of the `showInView:` method?

To know the answer, try pressing the **Pop question** button on either one of the `ViewControllers` and tap anywhere else on the screen. You will see that the `UIActionSheet` is removed from the screen and does not act as a modal View demanding the user to make a selection. Also, if you implement the delegate method to handle the user's selection, you will see that it is not responding. This is because the `UIActionSheet` will never become the first responder. The code, which you have seen above, presents the `UIActionSheet` as shown in **Figure 43**.

Quick Tip

If your application uses a `UIToolbar` or a `UINavigationController` for managing your navigation stack, you can use the `showFromToolbar:` method of the `UIActionSheet` class to present the `UIActionSheet`.

Additional Knowhow

The `showFromRect:inView:animated:` method accepts a `CGRect` for the coordinates from where it should show, as well as a `UIView` in which to present the `UIActionSheet`. In this implementation, you show the `UIActionSheet` directly from the frame.

<H1> [Responding to User Input]

Source: [Professional IOS Programming][Chapter No 4][133]

So far, you have learned how to present the `UIActionSheet` in different scenarios related to your application's navigation logic. You implemented several additional buttons, but did not yet implement anything to respond to the user selection.

<H2> [Processing the User Selection]

Source: [Professional IOS Programming][Chapter No 4][133]

The whole purpose of presenting a `UIActionSheet` is to allow a user to choose from the list of presented options and to process the response by performing the associated application logic.

1. Start Xcode and create a new project using the **Single View Application** template. Name the project as **ActionSheetResponding** using the project options shown in **Figure 25**.

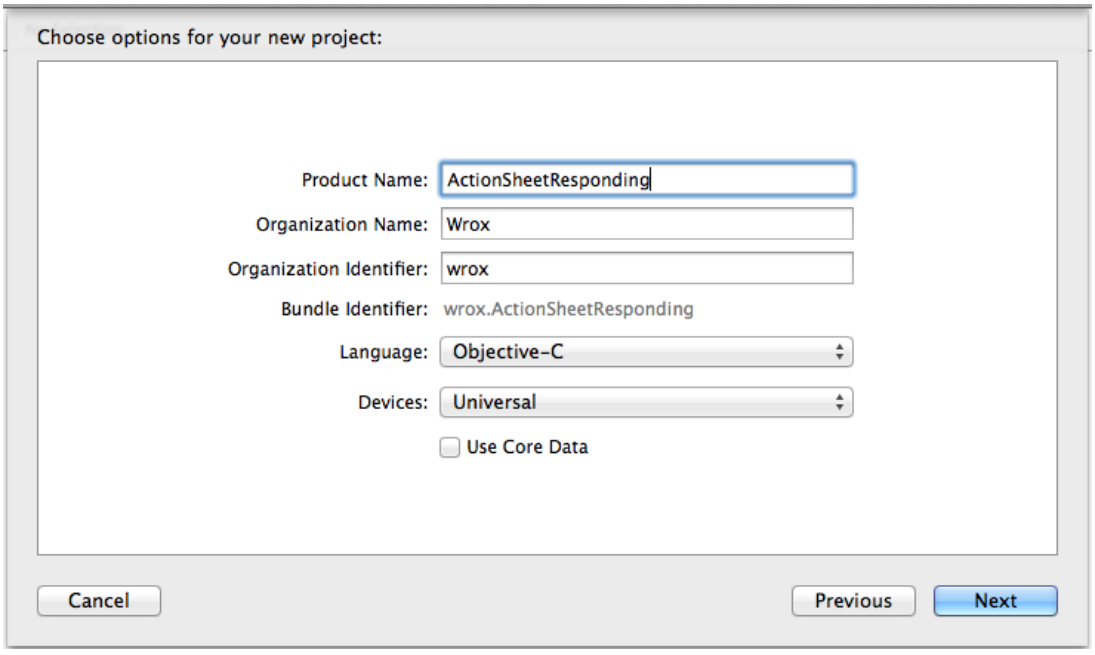

Figure 25: Create the ActionSheetResponding Project

2. To be able to respond to the user selection, you need to subscribe to the `UIActionSheetDelegate` protocol in your `YDViewController` class extension and implement the `actionSheet:clickedButtonAtIndex:` delegate method.

3. When you create the `UIActionSheet`, you also set the delegate to `self`. When the user clicks one of the buttons in the `UIActionSheet`, the `actionSheet:clickedButtonAtIndex:` delegate method is called. The index is used to identify what choice the user has made(in the same sequence as you have created the buttons starting with index 0). Since you have captured the user's choice, you can now perform the desired action. The complete `YDViewController` implementation is shown below:

```
#import "YDViewController.h"

@interface YDViewController ()<UIActionSheetDelegate>
- (IBAction)popQuestion:(id)sender;
@end

@implementation YDViewController

- (void)viewDidLoad
{
      [super viewDidLoad];
      //Do any additional setup after loading the view, typically from a nib.
}
- (IBAction)popQuestion:(id)sender;
{
      UIActionSheet *actionSheet = [[UIActionSheet alloc]
                        initWithTitle:@"Choose action"
                        delegate:nil
                        cancelButtonTitle:nil
                        destructiveButtonTitle:nil
                        otherButtonTitles:nil];
      [actionSheet addButtonWithTitle:@"Take picture"];
      [actionSheet addButtonWithTitle:@"Choose picture"];
      [actionSheet addButtonWithTitle:@"Take video"];
      [actionSheet addButtonWithTitle:@"Cancel"];
      //set the destructiveButtonIndex to the button that is responsible
        for the cancel function
      actionSheet.destructiveButtonIndex=3;
      actionSheet.delegate=self;
      [actionSheet showInView:self.view];
      //show from your view
}
```

```
- (void)actionSheet:(UIActionSheet *)actionSheet
      clickedButtonAtIndex:(NSInteger)buttonIndex
{
      int index = buttonIndex;

      [actionSheet dismisswithClickedButtonIndex:buttonIndex animated:YES];
      switch (index) {
            case 0:
            {
            NSLog(@"Take picture selected");
            }
                  break;
            case 1:
            {
            NSLog(@"Choose picture selected");
            }
                  break;
            case 2:
            {
            NSLog(@"Take video selected");
            }
                  break;
            case 3:
            {
            NSLog(@"Cancel selected");
            }
                  break;
            default:
                  break;
      }
}
- (void)didReceiveMemoryWarning
{
      [super didReceiveMemoryWarning];
      // Dispose of any resources that can be recreated.
}

@end
```

In the above code, the `dismissWithClickedButtonIndex:animated:` method is called to dismiss the `UIActionSheet`, and a switch on the index is used to identify what choice the user has made. In this implementation, you are just writing an `NSLog` with the result of the selected action.

You can apply similar logic in **Swift** language, but the syntax for `showInView` will be as follows:

```
//first add the button as per the code explained before
```

```
//code syntax for showInView

actionsheet!.showInView(self.view)
// format for Switch case will be as :
      Switch (index)          {
            case :0
            {
                  println(“Take picture selected”)
            }
            case :1
            {
                  println(“Choose picture selected”)
            }
            case :2
            {
                  println(“Take video selected”)
            }
            case :3
            {
                  println(“Cancel selected”)
            }
            default:
                        }
```

Quick Tip

Swift does not support `fallthrough` in the Switch statement. Therefore, there is no need to place a break after each statement if you have a command to execute, else you will use the `fallthrough` keyword explicitly for an empty case.

<H1> [Extending the `UIAlertView`]

Source: [Professional IOS Programming][Chapter No 4][136]

The `UIAlertView` object is an easy-to-use object to alert the user with a message containing a title, a message element, and one or more buttons, like the traditional OK and Cancel buttons.

Additional Knowhow

The capability to add other objects to the `UIAlertView` was introduced in iOS 5.

Real Life Connect

Suppose a user is filling a form and he/she has entered wrong values. As a result, the form is not

submitted. If the user does not know the reason behind the non-submission of the form, it will create problems for him/her. However, if there is an alert system in the application, it will give an alert when the user gives a wrong input. In this way, the user will be able to enter right inputs in another attempt. Finally, an alert informs that the values are submitted successfully. This scenario highlights the importance of `UIAlertView` in an iOS application.

<H2> [Adding a `UITextField` to a `UIAlertView`]

Source: [Professional IOS Programming][Chapter No 4][136]

In many application scenarios, you simply want the user to enter a username, a password, or a code, and confirm his/her input by clicking an OK or Cancel button. Often, you do not want to make a specific `UIViewController` object for this purpose. This is where the `UIAlertView` comes into play.

1. Create a new project in Xcode using the **Single View Application** template and name it **AlertViews** using the project options shown in **Figure 26**.

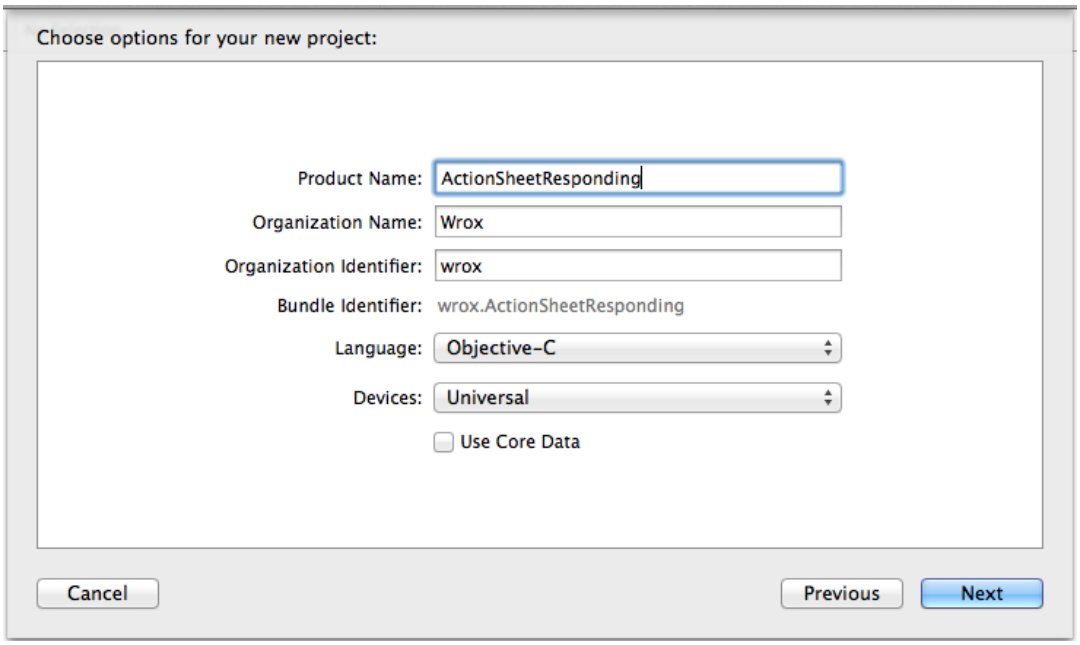

Figure 26: Create the ActionSheetResponding Project

2. Use Interface Builder and the Assistant Editor to create the user interface for the `YDViewController` as shown in **Figure 27**.
3. Open the `YDViewController.m` file and subscribe to the

`UIAlertViewDelegate` protocol. Use the Assistant Editor to create five different method declarations, one for each of the `UIButton` objects created.

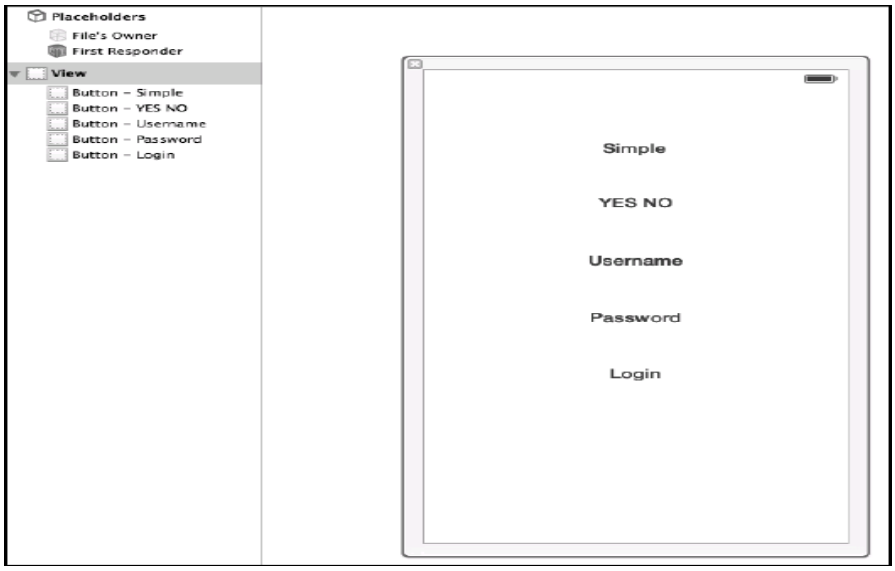

Figure 27: Create the `YDViewController` UI Using the Assistant Editor

The `alertViewStyle` property of the `UIAlertView` class determines what kind of `UITextField` is added to the `UIAlertView`. This can be a plain text field, a secure text field, or a combination of both. This last scenario is very useful to ask for a username and a password. The delegate property of the `UIAlertView` is set to `self`, and in the `alertView:clickedButtonAtIndex:` delegate method you can determine which button is pressed by the user and what input he/she made.

The `UIAlertView` class is available, but deprecated in iOS 8. So you should avoid using it in future versions because they might not support it. Instead, you can use the `UIAlertController` with a preferred style of the `UIAlertControllerStyleActionSheet` in future Versions of iOS.

Lab Connect

During the lab hour of this session, you will create views and alerts in a form while submitting.

Cheat Sheet

- A table view is one of the most common views in iOS applications. It enables users to perform the following actions:
    - Enable users to navigate through hierarchically structured data
    - Present an indexed list of items
    - Display detail information and controls in visually distinct groupings
    - Present a selectable list of options
- A table view is an instance of the `UITableView` class.
- The `UITableView` controls its appearance through the `UITableViewDelegates`.
- The `UITableView` interacts with the data model of an app through `UITableViewDataSource`.
- The `UITableViewController` class implements the `UITableViewDelegate` and `UITableViewDataSource` along with an `IBOutlet` to the `UITableView`.
- The `UINavigationController` is a `container` controller that enables different `UIViewControllers` to display within it. It also adds a `UINavigation` Item at the top of the screen.
- The `UITableViewCell` object represents a cell in a `UITableView`. The `UITableViewCell` does not have a delegate.
- The easiest way of storage is an `NSMutableDictionary`, which is a key/value data storage object.
- You can easily rename the methods and classes across your entire codebase using the refactor feature of Xcode.
- The sections and an index help to easily find items in the `UITableView`. The sections break up the items visually, while the index gives a shortcut to jump to the different sections.
- `UITableView` has a built-in edit mode. When a table goes into edit mode, it asks its delegate which rows are editable. You can control which rows are editable by implementing the `tableView:canEditRowAtIndexPath:` method.
- A segue is a way of transitioning from one view to the next. The two most common segues are modal segues and push segues.
- The `UIActionSheet` requires the user to select an action. You can create a `UIActionSheet` with a single option or multiple options.
- You can present the `UIActionSheet` using the following methods:
    - `showInView`
    - `showFromTabBar`
    - `showFromBarButtonItem`
    - `showFromRect inView`
    - `showFromToolbar`
- The purpose of presenting a `UIActionSheet` is to allow a user to choose from the list of presented options and to process the response by performing the associated application logic.

- The `UIAlertView` notifies the user with a title and a message. It remains visible until the user presses one of the presented button(s) that will remove the `UIAlertView` from the presentation stack.