App Note’s

CoreData use krke k liye project banate time (use Core Data) file to check krna h .

Core Data – (Is a Framework) to store Data in Computer by creating Core Data File in project. To use Core Data you should have to import it .

-: It consists Three Layer for executing: -

1.**Manage Object Model Layer** : - Describe data here what you want to input

2. **Persistent Store Coordinator Layer**:- It helps to read and write in Database .

It also helps to select one or multiple data stores in Database.ye layer chek krti h ki jo aap data store krne wale ho us tarah ka structure h ki nahi database main.

3. **Managed Storage Context Layer** : - This is the Top layer the Database with which app code interact.

It manages data in application memory during read/write operations. It allows to perform insert/update/delete/select operations in the data.

Object **Manage Object** types are passed to and from this layer.Ye layer hame project m dikhti h jisse ham input dete h or manage krte h delete/modify etc.

**App Start - > UI Application Delegate - > Persistent Container - > Manage Object Context**

**Ye sara Code ham use krenge Save Button k sath {**

3no Layers App Deligate m Show hoti h - >Core Data Stack k naam se

Uske ander – persitent Container type ka code hota h jo ki 3no layer ko conain krta h . or agar 3no layers m se kise ko bhi acess krna h to persiatentContainer tak ana hi padega.isme viewContext nam ka Variable h jo top layer type ka h (viewContext:NSObjectViewContext).

Isme 3no layers k 3no variables h ..

User se data lane k baad usko save krne k liye manage object Context tak jana padega jo ki peristentContainer ka ander h jo ki UIApplicationDelegate k ander h jo ki UI Application k ander hai.

Ab ham getcontext nam ka ek function banayenge jo ki data ko import krega jo ki NSNanageObjectContext type ka hota h .

Iske bad ham ek object create krenge app nam ka jo ki UIApplication type ka hoga ab UIapplication Type ka ek Object add krenge jo ki shared naam ka ha

Ab ham UIApplicationDelegate ka type UIApplication Type ka hota h uske baad uske baad jiska object hame App se milega jiska type AppDeligate h

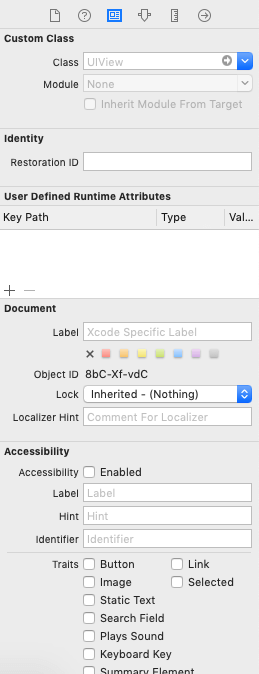
Uske baad hame ek objet create krenge context nam ka jo ki UINSObectCOntext type ka h jo ki hame Aplicationdelegate ka persistentContainer .Viewcontext

Return context **}**

ab object create kreng : - same function ka ander- jo ki hame textField se milne wale h

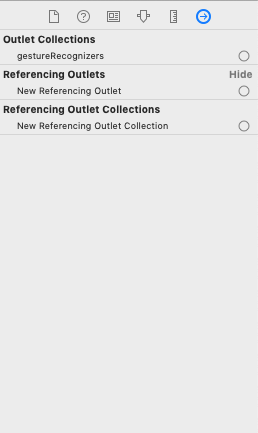
dentity Inspector

Identity Inspector is mainly used when we need to work with the storyboard. It shows information about storyboard components (View Controllers) and their corresponding Swift class files. To program the storyboard components accordingly, we need to assign class files to them. All this information is shown by Identity Inspector, as shown in the below image. It prompts the developer to assign class and module to the corresponding storyboard View Controller. It also prompts the developer to assign the identity name to the View Controller, which is used throughout the project to identify the View Controller. We can also give some Runtime constraint to the UIView shown in the storyboard.



Connections Inspector

It shows the information about the connections of the corresponding storyboard UIView to the swift class file. It contains all the connections of the storyboard to the swift class files.



# **Views and View Controllers**

In iOS development, the view controllers are the foundation of the Application's internal structure. The View Controller is the parent of all the views present on a storyboard. Each application has at least one ViewController. It facilitates the transition between various parts of the user interface.

The UIViewController is the parent class of all the ViewControllers. It defines all the methods and properties for managing our views. This class also manages the events and transitions from one view controller to another. It also coordinates between the different parts of the application.

# **Views and View Controllers**

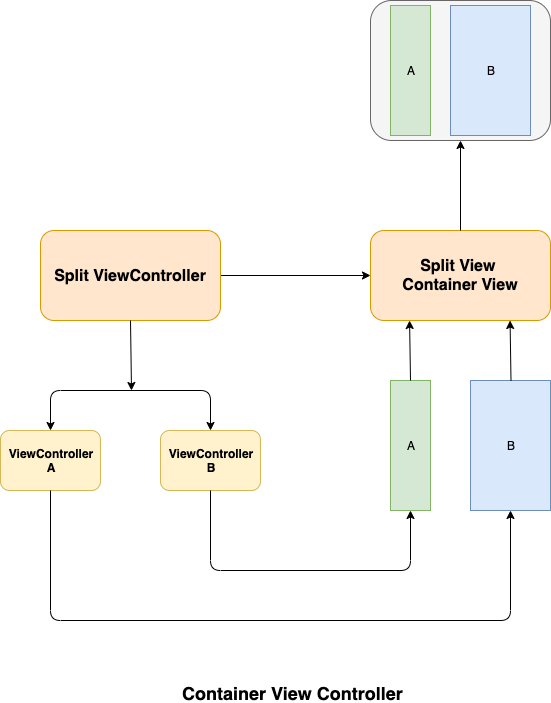
In iOS development, the view controllers are the foundation of the Application's internal structure. The View Controller is the parent of all the views present on a storyboard. Each application has at least one ViewController. It facilitates the transition between various parts of the user interface.

The UIViewController is the parent class of all the ViewControllers. It defines all the methods and properties for managing our views. This class also manages the events and transitions from one view controller to another. It also coordinates between the different parts of the application.

## Types of ViewControllers

There are two types of ViewControllers:

1. **Content ViewController:** Content ViewControllers are the main type of View Controllers that we create. The Content View Controllers holds the content of the Application screen. In other words, we can say that the Content View Controller manages the discrete piece of the application content. The Content ViewController manages all the Views itself.
2. **Container ViewController:** Container ViewController is different from content ViewController in the sense that it acts as a parent View Controller, which collects information from the child view controllers. The task of the container view controller is to present the collected information to facilitate the navigation to the child view controllers. The container ViewController only manages the RootView, which incorporates one or more Child ViewControllers.



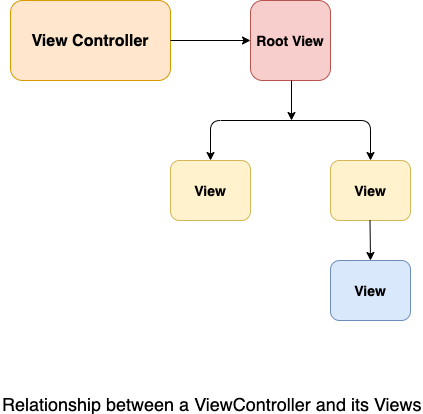
Most iOS applications are the mixture of both, Content ViewController and Container ViewController.

1.6K

How To Downgrade iOS 16 to 15 Without Losing Data - EASY METHOD????

## View Management

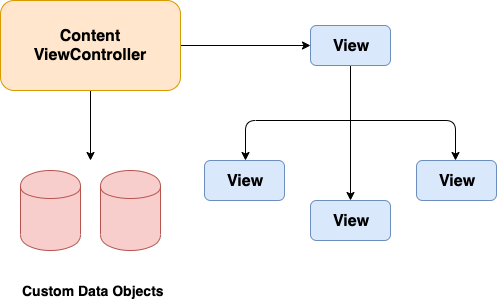
In iOS development, the ViewController manages the hierarchy of views. As shown in the below image, each ViewController contains a RootView which contains all the content of the view controller. All the custom views needed to maintain an iOS application is added to the root view to display the content. The following figure shows the relationship between ViewController, RootView, and its subviews. Each sub view is referred by a Super View which incorporates a chain of Views where a RootView acts as the Parent View of all the views present in the View Controllers.



## Data Marshaling

In iOS Development, a View Controller is responsible for displaying the data of our iOS application on the screen. It acts as an interface between its Views (created by the developer) and the application data. Each ViewController in the Storyboard is assigned a Class that inherits the UIViewController class.

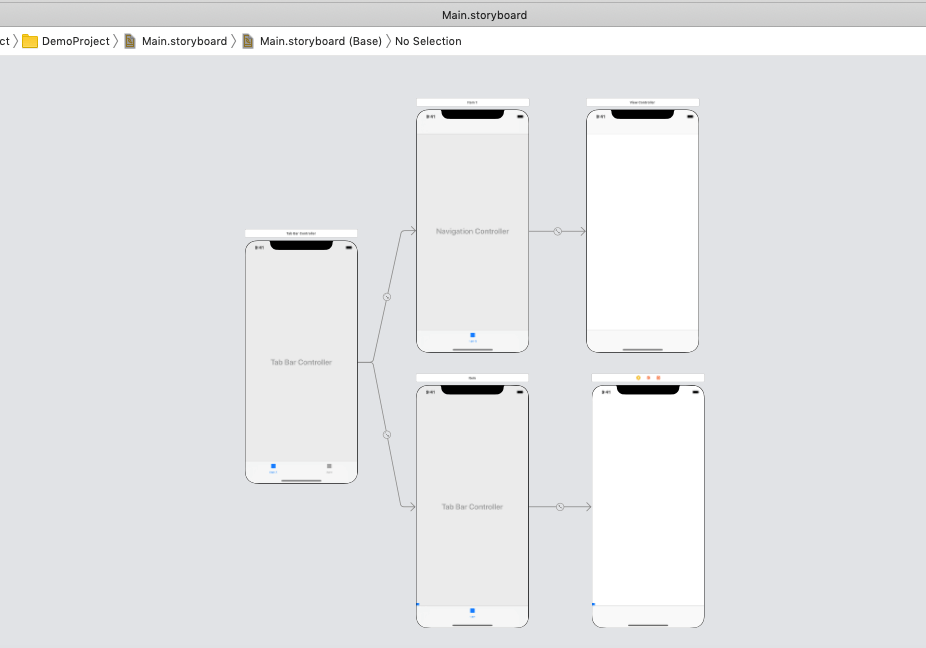
All of the properties and methods defined in the UIViewController is present in the class that we assign to the ViewController. However, for the development of our application, we need to define our properties and methods in the ViewController class. It helps us manage the visual representation of our application.



# **Storyboard and Interface Builder**

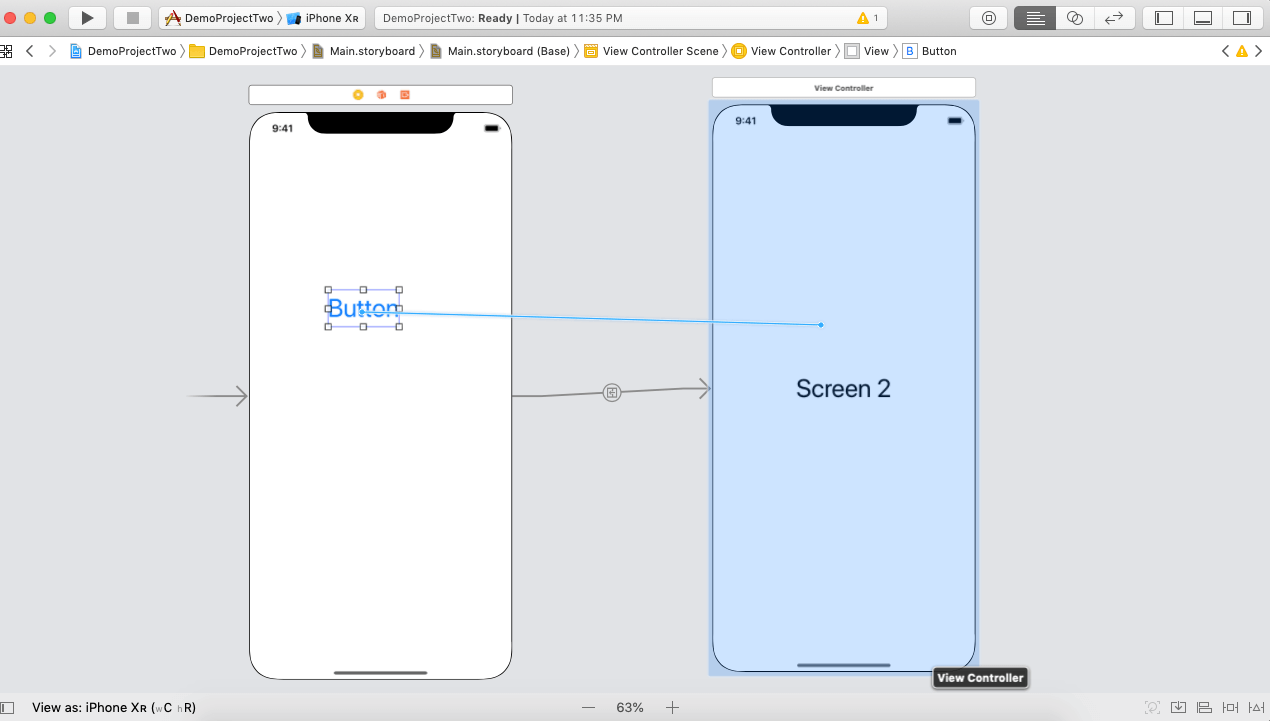
The storyboard is first introduced in iOS 5 to save time building user interfaces for the iOS applications. It is a visual representation of the user interface of an iOS app. It can be defined as the sequence of screens, each of which represents the ViewController and the Views. The transitions between two storyboard screens need a segue object, which represents a transition between two ViewControllers.

The Storyboard is built using a visual editor provided by XCode, in which we can layout and design the user interfaces of the application by adding the widgets from the media library such as buttons, views, table views, text fields, etc.



## Segues

Segues are used to make the transitions between two screens in the storyboard. We can set the type of transition like the model or push on the segue. In simple words, the segue is like an arrow defined on an object like a button or ViewController so that any user event on the object leads to the transition defined by the segue.



Sometimes, we may need to pass the data between the ViewControllers. It can be done by using prepareForSegue method, which is invoked on the View Controller when the segue is triggered. The segue can either be performed on the tap of the object on which the segue is defined or programmatically by using performSegue(withIdentifier: String) method on ViewController.

# **Label**

UILabel inherits the UIView class. It represents a class of views that display one or more lines of read-only texts. In iOS applications, the label is used in the association with UIControls to fulfill the Application requirements.

The syntax of the UILabel class is given as follows.

1. **class** UILabel : UIView
2. **import** UIKit
4. **class** ViewController: UIViewController {
6. @IBOutlet weak var textLbl: UILabel!
8. override func viewDidLoad() {
9. **super**.viewDidLoad()
10. // Do any additional setup after loading the view.
11. textLbl.text = "Hello World"
12. textLbl.font = .italicSystemFont(ofSize: 30)
13. textLbl.backgroundColor = UIColor.blue
14. textLbl.textAlignment = .center
15. textLbl.textColor = UIColor.white
16. textLbl.shadowColor = UIColor.black
17. textLbl.isHighlighted = **true**
18. }
19. }

### **Example 2**

**Making the Label tappable**

The following example will make the label created, in example 1 tappable. For this purpose, we will create an object of the class UITapGestureRecognizer.

1. **import** UIKit
3. **class** ViewController: UIViewController {
5. @IBOutlet weak var textLbl: UILabel!
6. var didTap = **true**
8. override func viewDidLoad() {
9. **super**.viewDidLoad()
10. // Do any additional setup after loading the view.
11. textLbl.text = "Hello World"
12. textLbl.font = .italicSystemFont(ofSize: 30)
13. textLbl.backgroundColor = UIColor.blue
14. textLbl.textAlignment = .center
15. textLbl.textColor = UIColor.white
16. textLbl.shadowColor = UIColor.black
17. textLbl.isHighlighted = **true**
18. let tap = UITapGestureRecognizer(target: self, action: #selector(didTextLabelTap(sender:)))
19. textLbl.isUserInteractionEnabled = **true**
20. textLbl.addGestureRecognizer(tap)
21. }
23. @objc func didTextLabelTap(sender: UITapGestureRecognizer){
24. **if**(didTap){
25. textLbl.backgroundColor = UIColor.brown
26. didTap = **false**
27. }
28. **else**{
29. textLbl.backgroundColor = UIColor.blue
30. didTap = **true**
31. }
33. }
34. }

# **TextField**

It can be defined as an object which is used to display an editable text area in the interface. Textfields are used to get the text-based input from the user.

1. **class** UITextField : UIControl

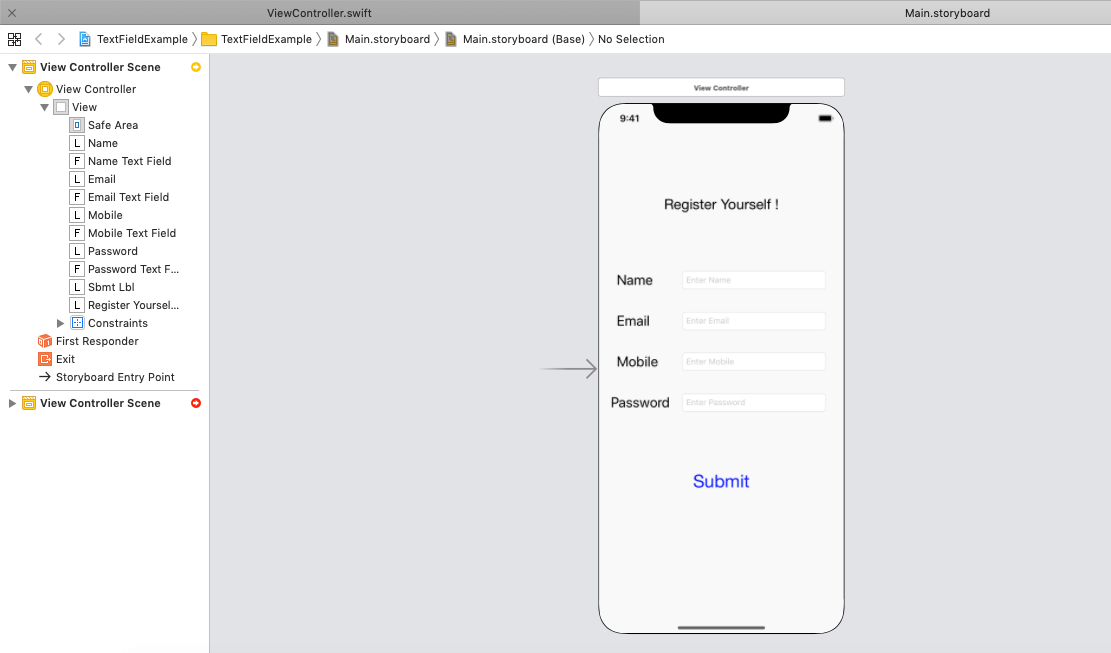
The following View is created in the project to get the information from the user.

1. **import** UIKit
3. **class** ViewController: UIViewController {
4. @IBOutlet **weak** **var** createUserSuccessView: UIView!
6. @IBOutlet **weak** **var** nameTextField: UITextField!
8. @IBOutlet **weak** **var** emailTextField: UITextField!
10. @IBOutlet **weak** **var** passwordTextField: UITextField!
12. @IBOutlet **weak** **var** mobileTextField: UITextField!
14. @IBOutlet **weak** **var** sbmtLbl: UILabel!
16. @IBOutlet **weak** **var** messageNameLbl: UILabel!
18. **override** **func** viewDidLoad() {
19. **super**.viewDidLoad()
20. // Do any additional setup after loading the view.
21. setInitViews()
22. sbmtLbl.isUserInteractionEnabled = **true**
23. **let** tap = UITapGestureRecognizer(target: **self**, action: #selector(sbmtBtnTapped(sender:)))
24. sbmtLbl.addGestureRecognizer(tap)
25. }
26. **func** setInitViews(){
27. nameTextField.becomeFirstResponder()
28. emailTextField.delegate = **self**
29. mobileTextField.delegate = **self**
30. nameTextField.delegate = **self**
31. passwordTextField.delegate = **self**
33. }

36. @objc **func** sbmtBtnTapped(sender: UITapGestureRecognizer){
37. **if**(nameTextField.text?.isEmpty ?? **false** || emailTextField.text?.isEmpty ?? **false** || passwordTextField.text?.isEmpty ?? **false** || passwordTextField.text?.isEmpty ?? **false**){
38. **let** alert = UIAlertController(title: **nil**, message: "Please fill all the details", preferredStyle: .alert)
39. **let** action = UIAlertAction(title: "OK", style: .**default**) { (action) **in**
40. **self**.dismiss(animated: **true**, completion: **nil**)
41. }
42. alert.addAction(action)
43. **self**.present(alert, animated: **true**, completion: **nil**)
44. }
45. **else**{
46. messageNameLbl.text = "Hi " + (nameTextField.text ?? "")
47. createUserSuccessView.isHidden = **false**
49. }
50. }
51. }

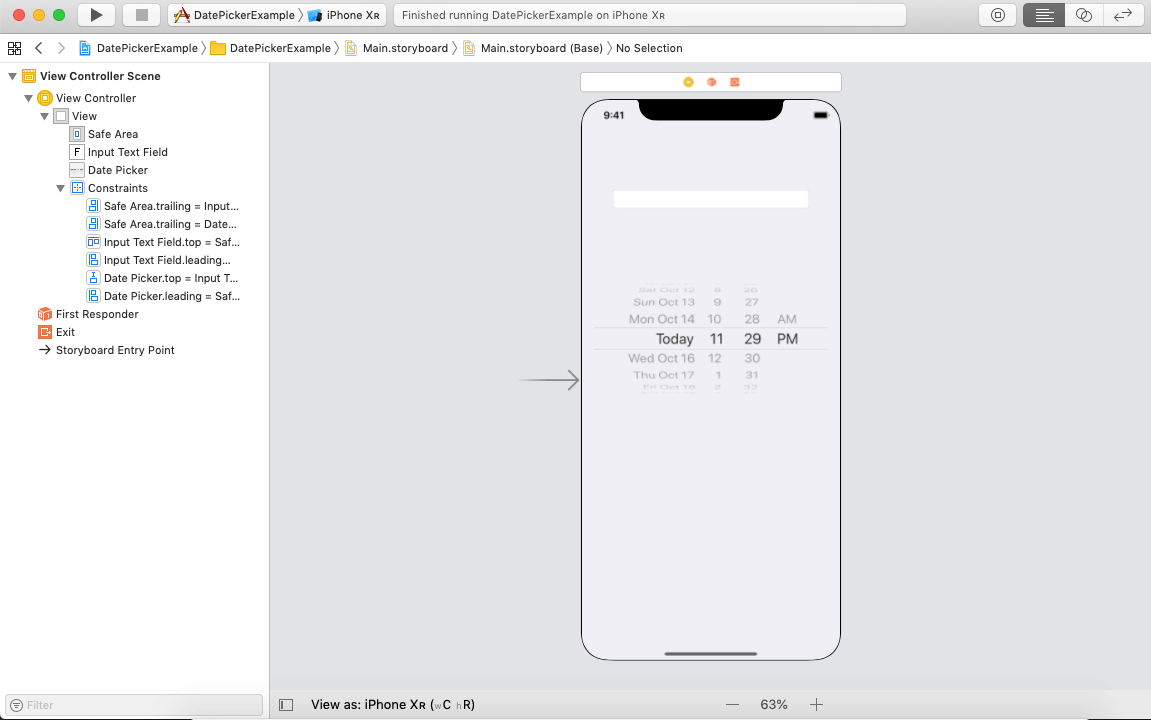
54. **extension** ViewController:UITextFieldDelegate{
56. **func** textField(\_ textField: UITextField, shouldChangeCharactersIn range: NSRange, replacementString string: String) -> Bool {
57. **if**(textField == mobileTextField){
58. **let** currrentCharacterCount = textField.text?.count ?? 0
59. **if** range.length + range.location > currrentCharacterCount {
60. **return** **false**
61. }
62. **let** newLength = currrentCharacterCount + string.count - range.length
63. **return** newLength <= 10
64. }
65. **else**{
66. **return** **true**
67. }
68. }
69. }

|  |  |  |
| --- | --- | --- |
| **SN** | **Method Signature** | **Description** |
| 1 | func textFieldShouldBeginEditing(UITextField) -> Bool | It asks the delegate if editing should begin the respective textfield. |
| 2 | func textFieldDidBeginEditing(UITextField) | It tells the delegate that the editing is started in the textfield. |
| 3 | func textFieldShouldEndEditing(UITextField) -> Bool | It asks the delegate to end the editing in the textfield. |
| 4 | func textFieldDidEndEditing(UITextField, reason: UITextField.DidEndEditingReason) | It tells the delegate that the editing has been stopped for the specified textfield. |
| 5 | func textFieldDidEndEditing(UITextField) | It is the overloaded methods which also does the same as the above. |
| 6 | func textField(UITextField, shouldChangeCharactersIn: NSRange, replacementString: String) -> Bool | It asks the delegate that if the text field's current content should be changed. |
| 7 | func textFieldShouldClear(UITextField) -> Bool | It asks the delegate if the text field's current content should be removed. |



# **DatePicker**

DatePicker is a control used in iOS applications to get the date and time values from the user. We can allow the user to either enter the time in point or time interval.

1. **class** UIDatePicker : UIControl
2. 
3. **import** UIKit
5. **class** ViewController: UIViewController {
7. @IBOutlet weak var inputTextField: UITextField!
9. @IBOutlet weak var datePicker: UIDatePicker!
11. let dateFormatter = DateFormatter()
13. override func viewDidLoad() {
14. **super**.viewDidLoad()
15. // Do any additional setup after loading the view.
16. dateFormatter.dateFormat = "MM/dd/yyyy"
17. inputTextField.inputView = datePicker
18. datePicker.datePickerMode = .date
19. inputTextField.text = dateFormatter.string(from: datePicker.date)
21. }
23. @IBAction func datePickerValueChanged(\_ sender: UIDatePicker) {
25. inputTextField.text = dateFormatter.string(from: sender.date)
26. view.endEditing(**true**)
27. }
28. }

**Core attributes**

|  |  |  |
| --- | --- | --- |
| **SN** | **Attribute** | **Description** |
| 1 | Mode | It represents the DatePicker Mode. It is used to determine whether the datepicker is going to display the date, time, date and time, or a countdown interval. This can be accessed at runtime using datePickerMode property. |
| 2 | Locale | This represents the locale associated with the datepicker. This property overrides the system default locale. This can be accessed at runtime using the local property. |
| 3 | interval | It represents the granularity of the minute's spinner. The default value is 1, and the maximum value is 30. This value must be a divisor of 60. It can be accessed at runtime using minuteInterval property. |

**Date Attributes**

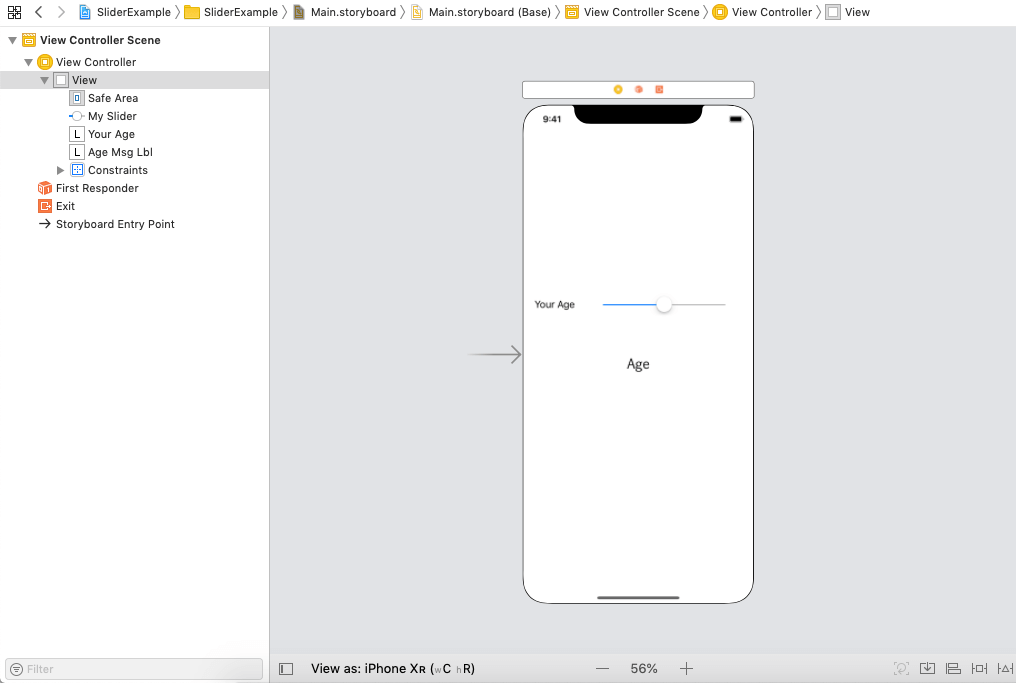
|  |  |  |
| --- | --- | --- |
| **SN** | **Attribute** | **Description** |
| 1 | Date | It represents the date that the date picker is going to display initially. We can set this property at runtime. |
| 2 | Constraints | It represents the range of the dates that can be selected. We can configure the minimumDate and maximumDate property to configure the range. |
| 3 | Timer | It is the initial value of the datepicker when it is shown in countdown timer mode. |

# **Slider**

A slider can be defined as a UIControl, which provides the contiguous range of values on a single scale to the user, out of which the user is prompted to select a single value.

The user is supposed to move the thumb on the slider. The slider is connected with the action method, which is notified every time the user moves the thumb onto a slider. The value of the slider can be retrieved each time the action method is called.

The slider is declared as follows.



1. **class** UISlider : UIControl
2. **import** UIKit
3. **class** ViewController: UIViewController {
4. @IBOutlet weak var mySlider: UISlider!
5. @IBOutlet weak var ageMsgLbl: UILabel!
6. override func viewDidLoad() {
7. **super**.viewDidLoad()
8. // Do any additional setup after loading the view.
9. mySlider.minimumValue = 0
10. mySlider.maximumValue = 60
11. }
12. @IBAction func sliderValueChanged(\_ sender: UISlider) {
13. let roundedValue = round(sender.value)
14. sender.value = roundedValue
15. ageMsgLbl.text = "Your Age is "+Int(sender.value).description
16. }
18. }

**Core Attributes**

|  |  |  |
| --- | --- | --- |
| **SN** | **Attribute** | **Description** |
| 1 | Value (minimum/ maximum) | It represents the float value, which is specified at the ends of the slider. The minimum value represents then leading end of the slider whereas the maximum represents then trailing end of the slider. |
| 2 | Value (current) | It represents the initial value of the slider, which is changed when the user interacts with the slider. It exists between the minimum and maximum values. This can be accessed at runtime by using value property on the slider object. |

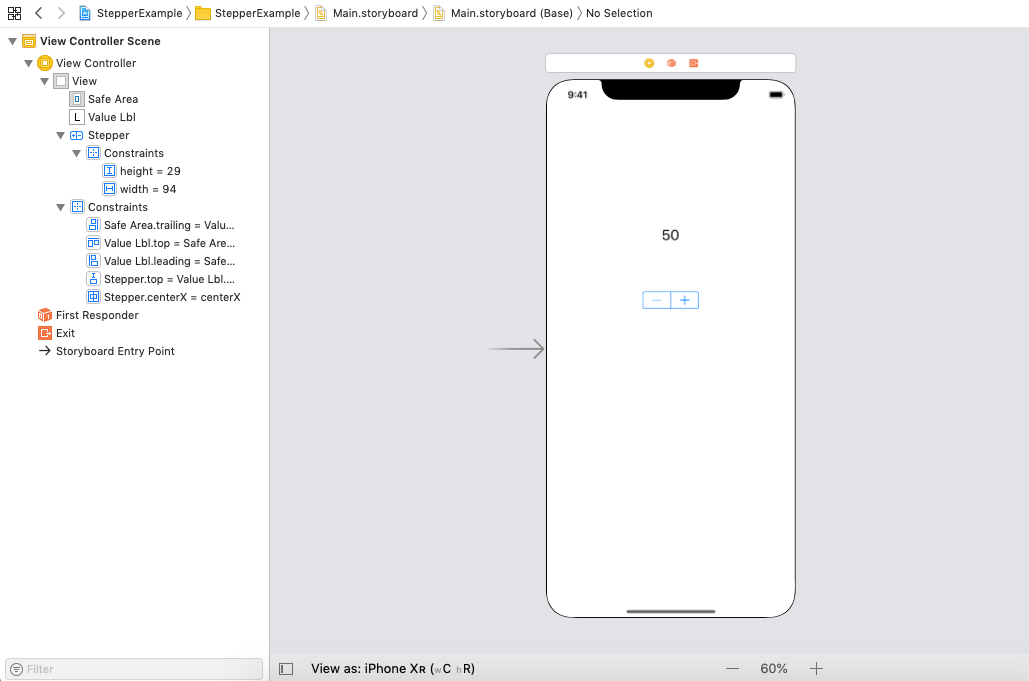
**Appearance attributes**

|  |  |  |
| --- | --- | --- |
| **SN** | **Attribute** | **Description** |
| 1 | Min Image | It represents the image specifies the leading end of the slider. This can be accessed by using minimumValueImage property at runtime. |
| 2 | Max Image | It represents the image specifies the trailing end of the slider. This can be accessed by using the maximumValueImage property at runtime. |
| 3 | Min Track Tint | It is the track tint color of the leading side of the slider. This can be accessed by using minimumTrackTintColor property at runtime. |
| 4 | Max Track Tint | It is the track tint color of the trailing side of the slider. This can be accessed by using the maximumTrackTintColor property at runtime. |
| 5 | Thumb Tint | It is the tint color of the slider's thumb. This can be accessed by using the thumbTintColor property at runtime. |

# **Stepper**

It is a type of UIControl which is used to increase and decrease value. The stepper consists of two buttons. It is associated with a value which gets repeatedly increased or decreased on holding down two of the buttons once at a time. The rate of the change depends on the duration the user presses the control.

1. **class** UIStepper : UIControl

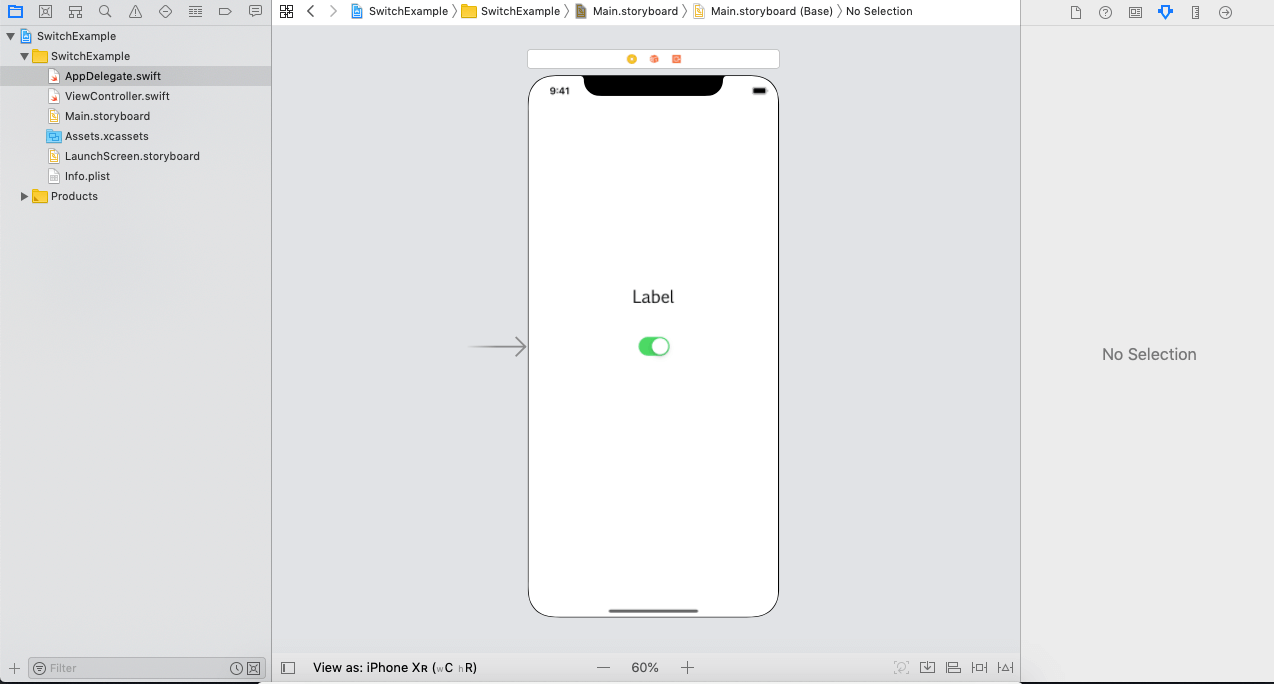


1. **import** UIKit
3. **class** ViewController: UIViewController {
5. @IBOutlet weak var valueLbl: UILabel!
6. @IBOutlet weak var stepper: UIStepper!
8. override func viewDidLoad() {
9. **super**.viewDidLoad()
10. // Do any additional setup after loading the view.
11. stepper.autorepeat = **true**
12. stepper.isContinuous = **true**
13. valueLbl.text = stepper.value.description
14. stepper.maximumValue = 20
15. stepper.minimumValue = -20
16. }
18. @IBAction func stepperValueChanged(\_ sender: UIStepper) {
19. valueLbl.text = sender.value.description
20. }
21. }

# **Switch**

The switch can be defined as the UIControl, which provides binary choices to the user either **on** or **off**. The state of a switch is managed by properties and methods defined in the UISwitch class, which is the subclass of UIControl.

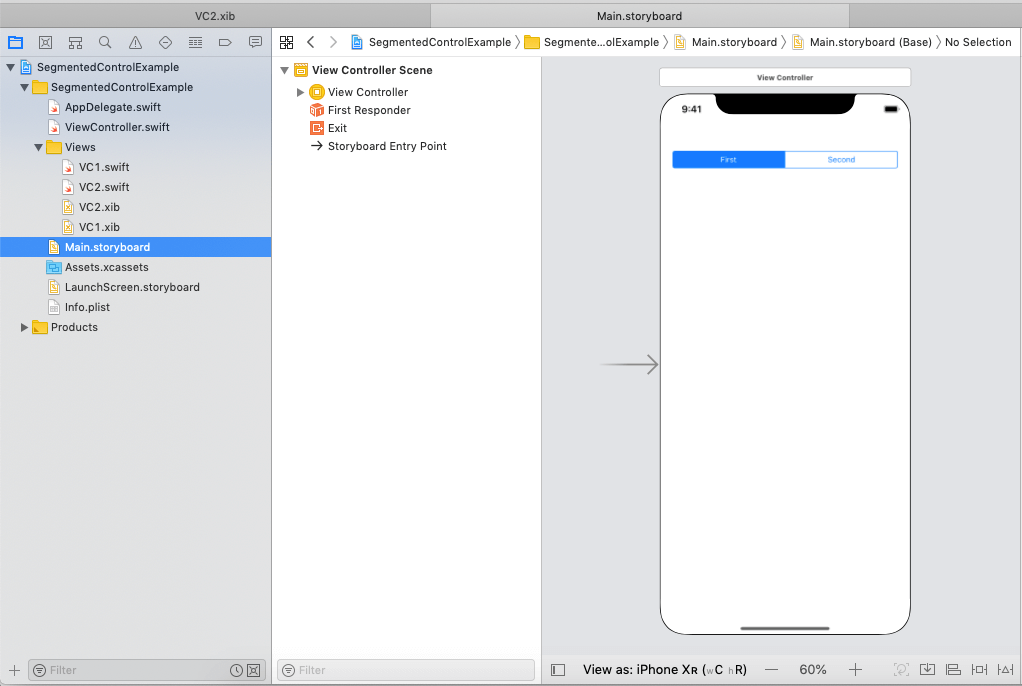
The UISwitch class is declared as follows.

1. **class** UISwitch : UIControl
2. 
3. **import** UIKit
5. **class** ViewController: UIViewController {
7. @IBOutlet weak var msgLbl: UILabel!
9. @IBOutlet weak var mySwitch : UISwitch!
11. override func viewDidLoad() {
12. **super**.viewDidLoad()
13. // Do any additional setup after loading the view.
14. }
16. @IBAction func switchValueChanged(\_ sender: UISwitch) {
17. **if**(mySwitch.isOn){
18. msgLbl.text = "Switch is On"
19. }
20. **else** {
21. msgLbl.text = "Switch is Off"
22. }
23. }

# **Segment Control**

Segment control can be defined as the horizontal control, which controls multiple segments where a discrete button controls each segment. A segment control can be used to display multiple views within a single view controller, where each view can be displayed by using a discrete button.

The segment control is declared as follows.

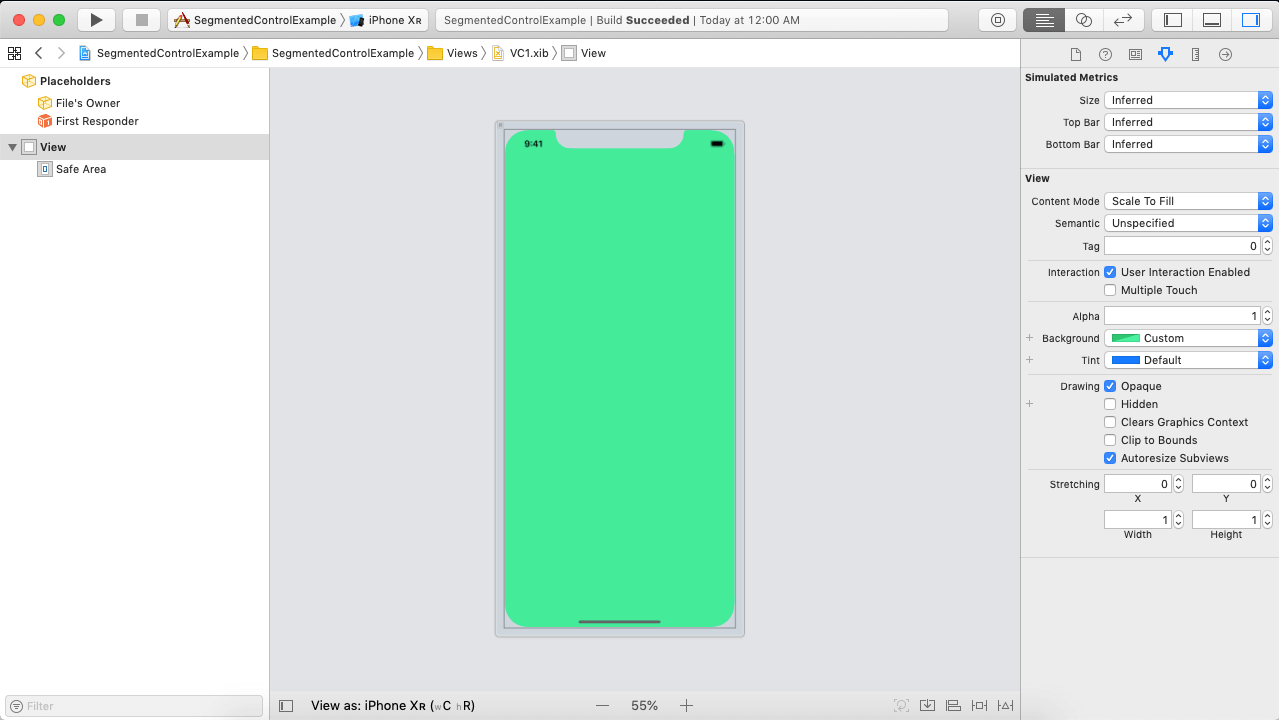
1. **class** UISegmentControl : UIControl
2. ccordingly.
3. 

### **What is a XIB file?**

XIB stands for an XML interface builder. The interface builder allows us to develop graphical user interfaces with the help of cocoa and carbon. XIB files are loaded at the runtime to provide the user interface for the application. The XIB files are stored as NIB or XIB files, which represent UIView.

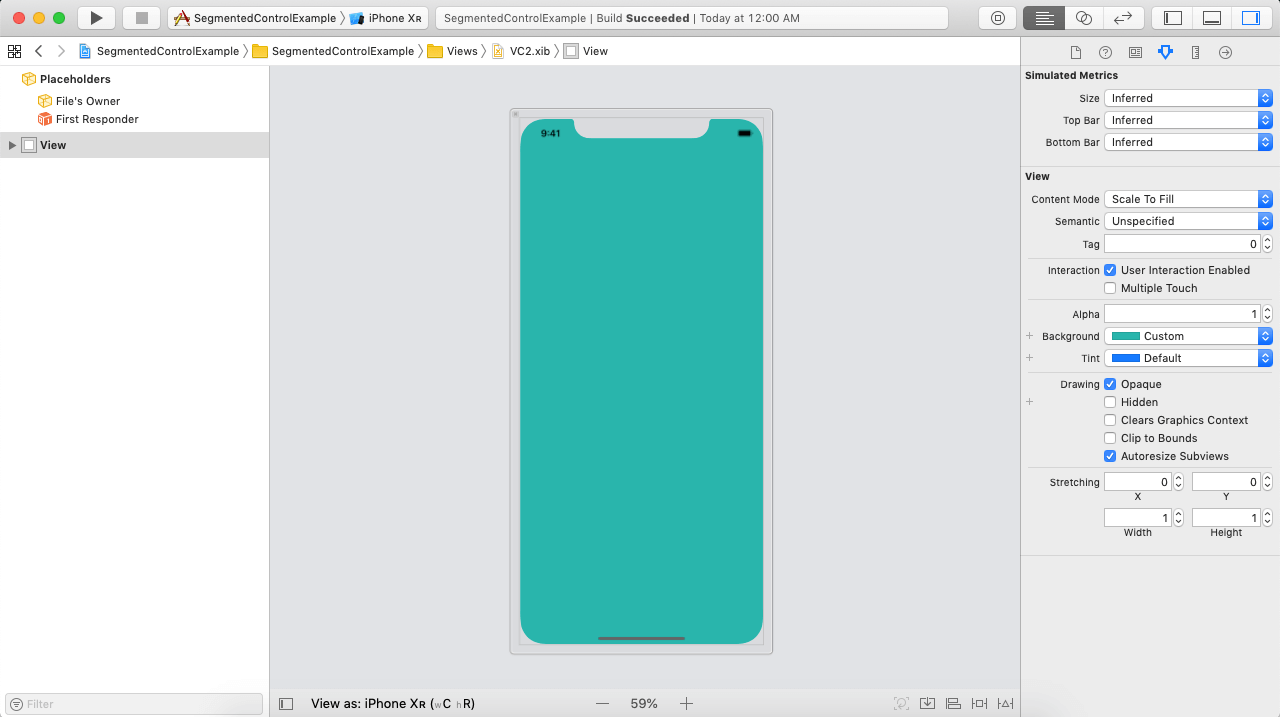
**VC1.xib**

The following image shows the VC.xib file.



**VC2.xib**

The following image shows the VC2.xib file.



**ViewController.swift**

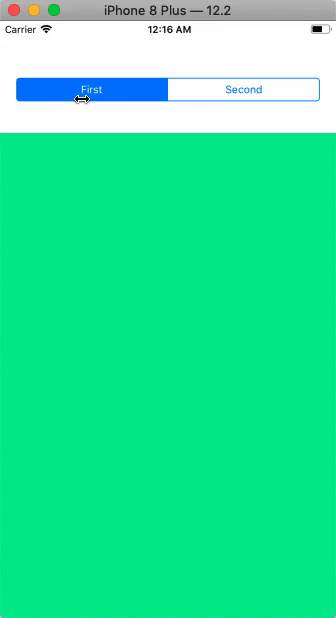
In ViewController.swift file, we will create the action connection for the segment control, which is notified each time the valueChanged event is triggered for segment control.

It toggles between the VC1.xib and VC2.xib on the valueChanged event of Segment Control.

1. **import** UIKit
3. **class** ViewController: UIViewController {

6. @IBOutlet weak var segmentedControl: UISegmentedControl!
8. @IBOutlet weak var viewContainer: UIView!
10. var views = Array<UIView>()
12. override func viewDidLoad() {
13. **super**.viewDidLoad()
14. // Do any additional setup after loading the view.
15. views.append(VC1().view!)
16. views.append(VC2().view!)
17. **for** v in views{
18. viewContainer.addSubview(v)
19. }
20. viewContainer.bringSubviewToFront(views[0])
21. }
23. @IBAction func switchViewAction(\_ sender: UISegmentedControl) {
24. viewContainer.bringSubviewToFront(views[sender.selectedSegmentIndex])
25. }
26. }

**Output:**



# **UIView**

UIView can be defined as an object by using which we can create and manage the rectangular area on the screen. We can have any number of views inside a view to create a hierarchical structure of the UIViews.

The UIView is managed by using the methods and properties defined in the UIView class that inherits UIKit. The declaration of UIView is given as follows.

class UIView : UIKit

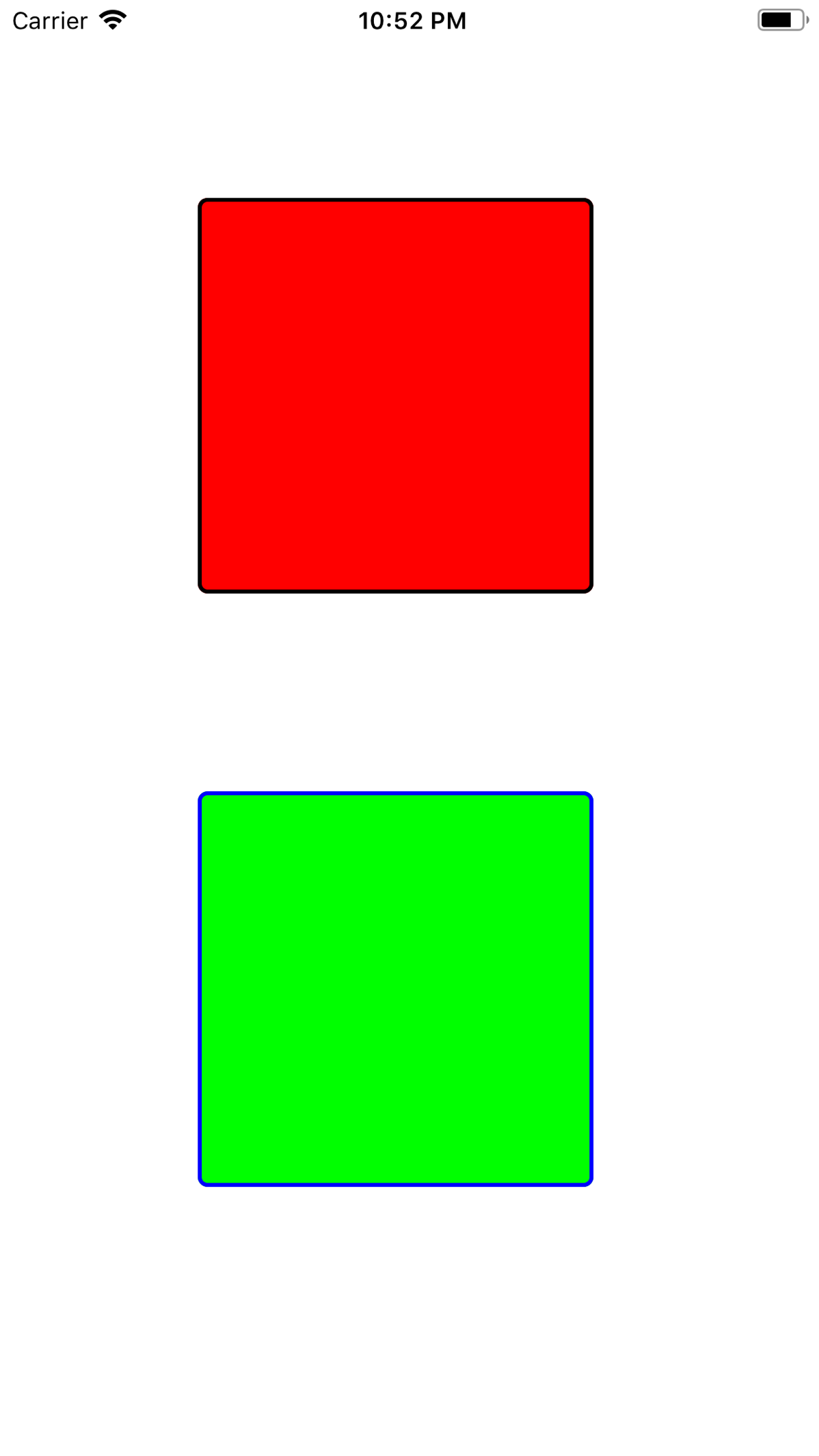
Views are the fundamentals of iOS application development, and that's why UIView is one of the most used object in the object library. The views are the basic building block of the iOS application, which renders the content within its bounds rectangle and also handles any interaction with the content.

UIViews are the fundamentals and the connection point of the iOS application with the user. There are several activities that are performed by the views in the iOS application.

* Drawing and animation
  + By using views, we can draw into the rectangular area of the screen.
* Layout and Sub view management
  + We can embed one or more subviews into the UIView. The appearance of the subviews can be managed by managing the appearance of the super view.
  + We can define the auto-layout rules to govern the size and positioning of the view hierarchy on different iOS devices.
* Event Handling
  + A view can respond to the touch and another kind of event since it is the subclass of UIResponder.
  + We can add the gesture recognizers for the uiview, such as UITapGestureRecognizer.

A View can be created programmatically by instantiating UIView class. We can pass the frame object inside the UIView constructor. In iOS application development, there are many objects like labels, text fields, etc. which directly inherits the UIView class to use the common properties and methods.

1. let rect = CGRect(x: 10, y: 10, width: 100, height: 100)
2. let myView = UIView(frame: rect)
3. **import** UIKit
5. **class** ViewController: UIViewController {
7. override func viewDidLoad() {
8. **super**.viewDidLoad()
9. // Do any additional setup after loading the view.
10. let frame1 = CGRect(x: 100, y: 100, width: 200, height: 200)
11. let myView1 = UIView(frame: frame1)
12. myView1.layer.shadowColor = UIColor.black.cgColor
13. myView1.layer.borderColor = UIColor.black.cgColor
14. myView1.layer.borderWidth = 2
15. myView1.layer.cornerRadius = 5
16. myView1.layer.shadowRadius = 2
18. let frame2 = CGRect(x:100, y:400, width: 200, height: 200)
19. let myView2 = UIView(frame: frame2)
20. myView2.layer.shadowColor = UIColor.blue.cgColor
21. myView2.layer.borderColor = UIColor.blue.cgColor
22. myView2.layer.borderWidth = 2
23. myView2.layer.cornerRadius = 5
24. myView2.layer.shadowRadius = 2
26. myView2.backgroundColor = .green
27. myView1.backgroundColor = .red
28. view.addSubview(myView1)
29. view.addSubview(myView2)
30. }
31. }

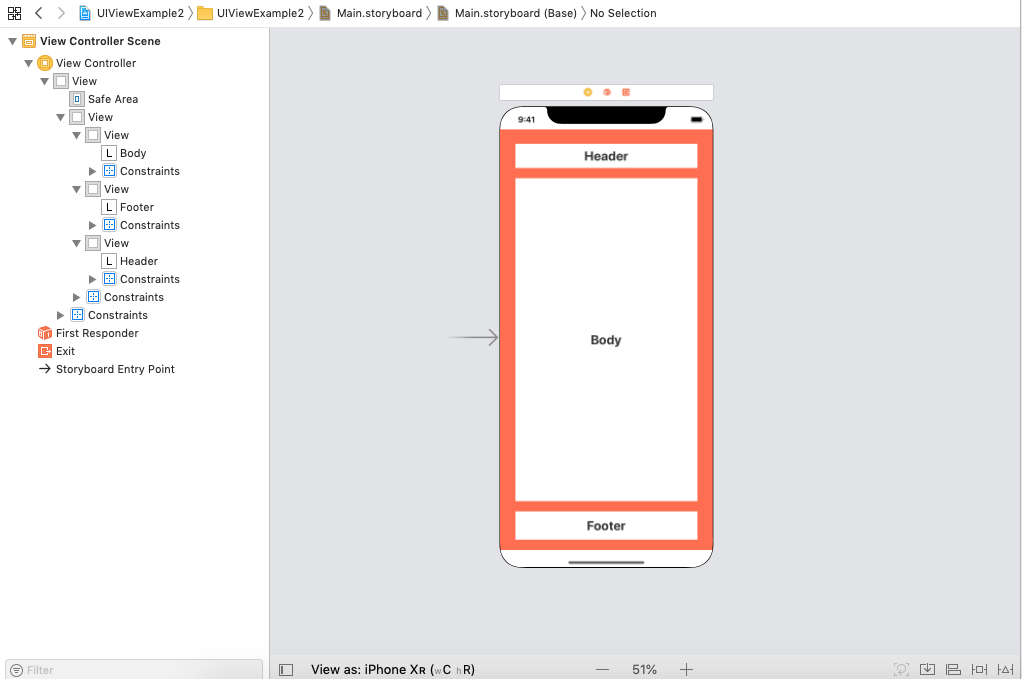


**Example 2**

In this example, we will simulate the structure of the web page on the iOS application. In this type of iOS application, where we need to display separate customizations, we will use UIViews in the iOs applications.

**Interface builder**

The following image shows the interface builder (main.storyboard) developed in the project. The left pane in the window shows the hierarchy of views and labels used in the project.



**ViewController.swift**

In the ViewController file, we will add the behavior for the labels so that we can change the look of the UIView associated with the label.

1. **import** UIKit
3. **class** ViewController: UIViewController {
5. @IBOutlet weak var headerView: UIView!
7. @IBOutlet weak var bodyView: UIView!
9. @IBOutlet weak var footerView: UIView!
11. @IBOutlet weak var headerLbl: UILabel!
13. @IBOutlet weak var bodyLbl: UILabel!
15. @IBOutlet weak var footerLbl: UILabel!
17. override func viewDidLoad() {
18. **super**.viewDidLoad()
19. // Do any additional setup after loading the view.
21. let headerTapGestureRecognizer = UITapGestureRecognizer(target: self, action: #selector(headerLblTapped))
22. headerLbl.isUserInteractionEnabled = **true**
23. headerLbl.addGestureRecognizer(headerTapGestureRecognizer)
25. let bodyTapGestureRecognizer = UITapGestureRecognizer(target: self, action: #selector(bodyLblTapped))
26. bodyLbl.isUserInteractionEnabled = **true**
27. bodyLbl.addGestureRecognizer(bodyTapGestureRecognizer)

30. let footerTapGestureRecognizer = UITapGestureRecognizer(target: self, action: #selector(footerLblTapped))
31. footerLb
32. l.isUserInteractionEnabled = **true**
33. footerLbl.addGestureRecognizer(footerTapGestureRecognizer)
34. }
36. @objc func headerLblTapped(){
37. headerView.backgroundColor = .orange
38. }
40. @objc func bodyLblTapped(){
41. bodyView.backgroundColor = .green
43. }
45. @objc func footerLblTapped(){
46. footerView.backgroundColor = .orange
47. }
48. }

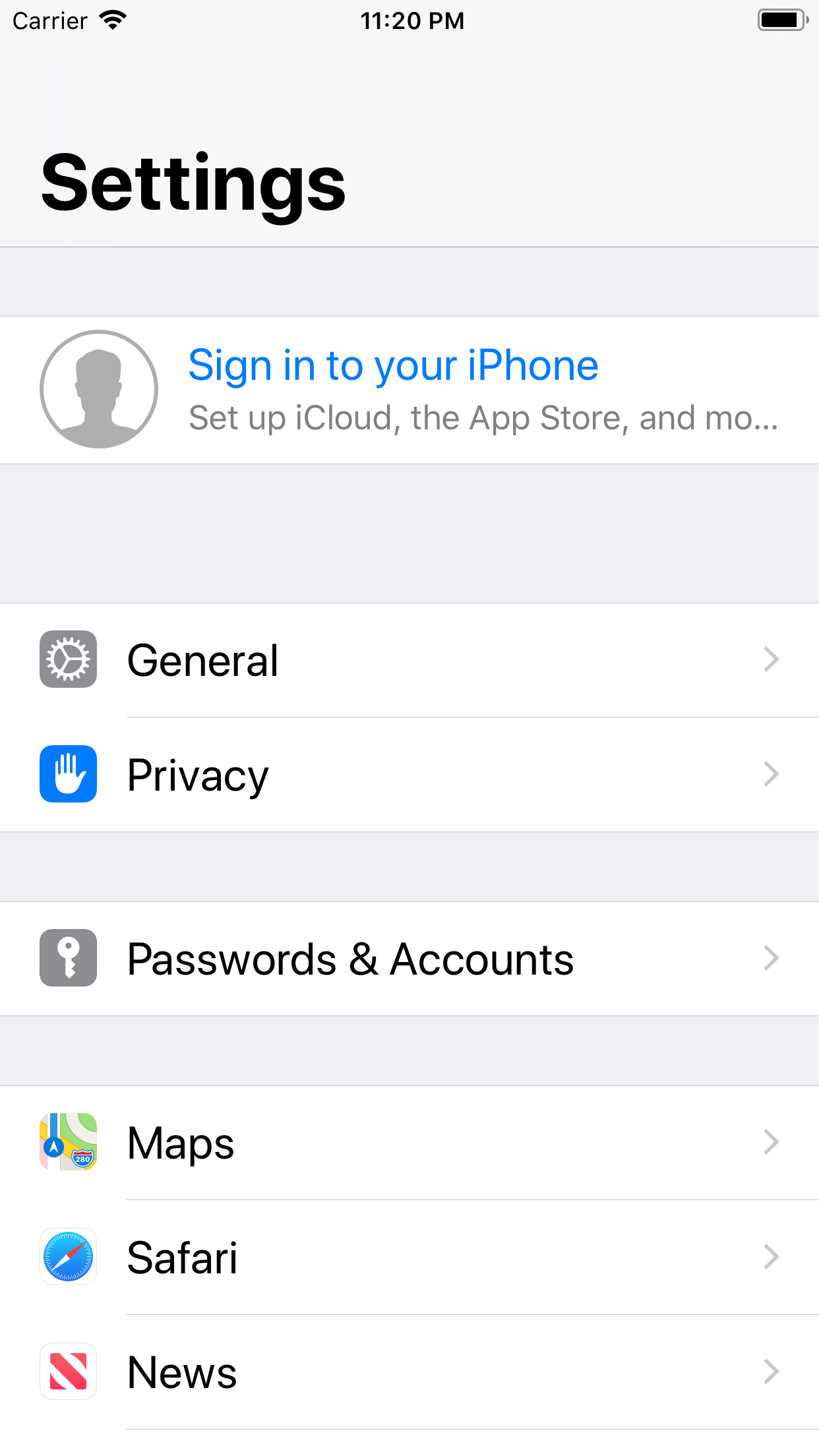
# **TableView**

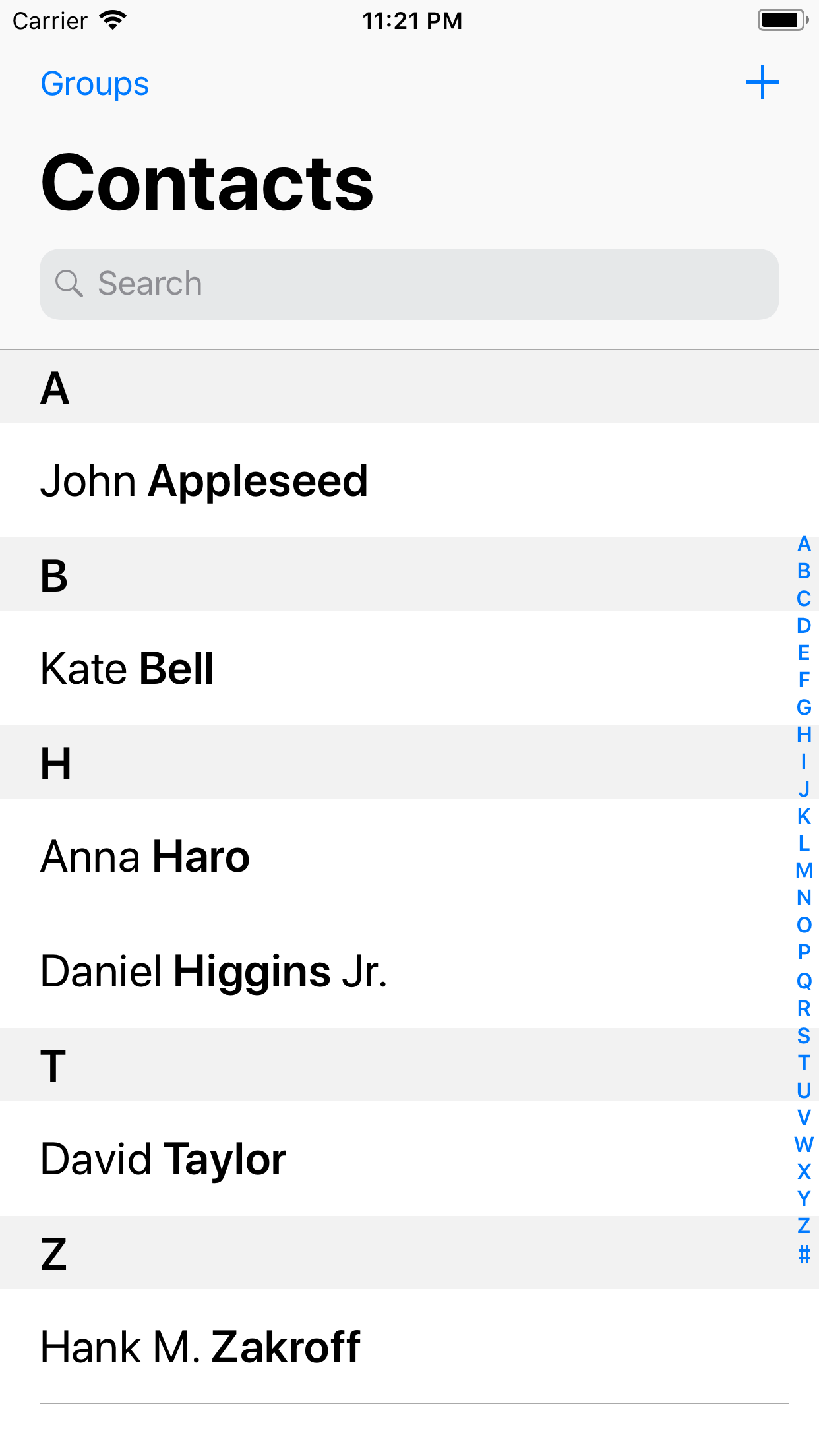
TableView can be defined as the view which can arrange the data using rows in a single column. It is used in almost every iOS application, for example, contacts, facebook, Instagram, etc. The tableview is the instance of the UITableView class, which inherits the UIScrollView class. We will discuss UIScrollView in the upcoming chapters of this tutorial.

1. **class** UITableView : UIScrollView

In iOS applications, whenever we need to display the single column containing vertically scrolling content, we use tableview. The tableview can display multiple records (divided into rows), which can be scrolled vertically if needed. Each row of the tableview presents each record of the data source. For example, in the contact app, we display each contact name in the separate row of the tableview, and we get the details related to the contact on the click to that row.

The below image shows how the tableview is used to display data in the settings app





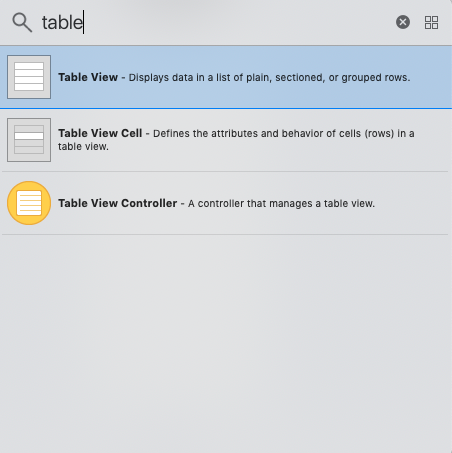
We can create the sections in the tableview to group the related rows, or we can show the long list of records in multiple rows without sections.

In iOS applications, the tableview is used in the association with the navigation controller to organize the data hierarchically. Here, we can navigate between different levels of hierarchy using the navigation controller.

The appearance of the tableview is managed by UITableView class, which inherits UIScrollView. In tableview, the row is simulated by the object of the UITableViewCell class, which can be used to display the actual content. We can customize the tableview cells to display any content in the iOS application.

Adding UITableView to interface

To add the tableview to the storyboard, search for the Tableview in the object library and drag the result to the storyboard.



To use the tableview, we need to set its delegate and data source properties. The tableview is a data-driven object, i.e., it gets the data to be shown from the data source object. In the real-world applications, the data source object contains the data which is returned by an API call from the database server.

The delegate and data source properties of the tableview can be set by using the following line of code in the viewDidLoad method of ViewController.

1. tableView.delegate = self
2. tableView.datasource = self

TableView Delegate Methods

The tableview delegate methods are defined to add the following features to the tableview.

* We can create the custom headers and footers for the sections in the tableview.
* We can specify the custom heights for rows, headers, and footers.
* Provide height estimates for the rows, headers, and footers.
* We can define the method which can handle the row selections.

|  |  |  |
| --- | --- | --- |
| **SN** | **Method** | **Description** |
| 1 | func tableView(UITableView, willDisplay: UITableViewCell, forRowAt: IndexPath) | The tableview notifies this delegate when it is about to draw a cell for a particular row. |
| 3 | func tableView(UITableView, willSelectRowAt: IndexPath) -> IndexPath? | The tableview notifies this delegate method when the specified row is about to be selected. |
| 4 | func tableView(UITableView, didSelectRowAt: IndexPath) | This delegate is notified when the specified row of the tableview is selected. |
| 5 | func tableView(UITableView, willDeselectRowAt: IndexPath) -> IndexPath? | This delegate is notified when the particular cell is about to be deselected. |
| 6 | func tableView(UITableView, didDeselectRowAt: IndexPath) | This delegate is notified when the particular row is deselected. |
| 7 | func tableView(UITableView, viewForHeaderInSection: Int) -> UIView? | This delegate method returns a UIView which represents the header of the tableview. |
| 8 | func tableView(UITableView, viewForFooterInSection: Int) -> UIView? | This delegate method returns the uiview, which represents the footer of the tableview. |
| 9 | func tableView(UITableView, willDisplayHeaderView: UIView, forSection: Int) | This delegate method is notified when the tableview is going to display the headerview for the particular section. |
| 10 | func tableView(UITableView, willDisplayFooterView: UIView, forSection: Int) | This delegate method is notified when the tableview is going to display the footer view for the particular section. |
| 11 | func tableView(UITableView, heightForRowAt: IndexPath) -> CGFloat | This delegate method returns the height for the row. |
| 12 | func tableView(UITableView, heightForHeaderInSection: Int) -> CGFloat | This delegate method returns the height of the header of section in the tableview. |
| 13 | func tableView(UITableView, heightForFooterInSection: Int) -> CGFloat | This delegate method returns the height for the footer of a particular section in the tableview. |
| 14 | func tableView(UITableView, estimatedHeightForRowAt: IndexPath) -> CGFloat | It asks the delegate for the estimated height of the row in a particular location. |
| 15 | func tableView(UITableView, estimatedHeightForHeaderInSection: Int) -> CGFloat | It asks the delegate for the estimated height of the header in a particular location. |
| 16 | func tableView(UITableView, estimatedHeightForFooterInSection: Int) -> CGFloat | It asks the delegate for the estimated height for the footer in the particular section. |

## TableView DataSource Methods

To maintain the data to be displayed by the tableview, we need to maintain a DataSource object that implements the UITableViewDataSource protocol. The datasource object manages the tableview data. The datasource object performs the following main tasks.

1. It reports the number of rows and sections to be displayed in the tableview.
2. It allocates the reusable cells for each row in the tableview.
3. It provides the titles for headers and footers in the tableview sections.
4. To perform the above-mentioned tasks, there are some functions defined in the UITableviewDataSource protocol. The following table contains the important methods defined in the protocol.

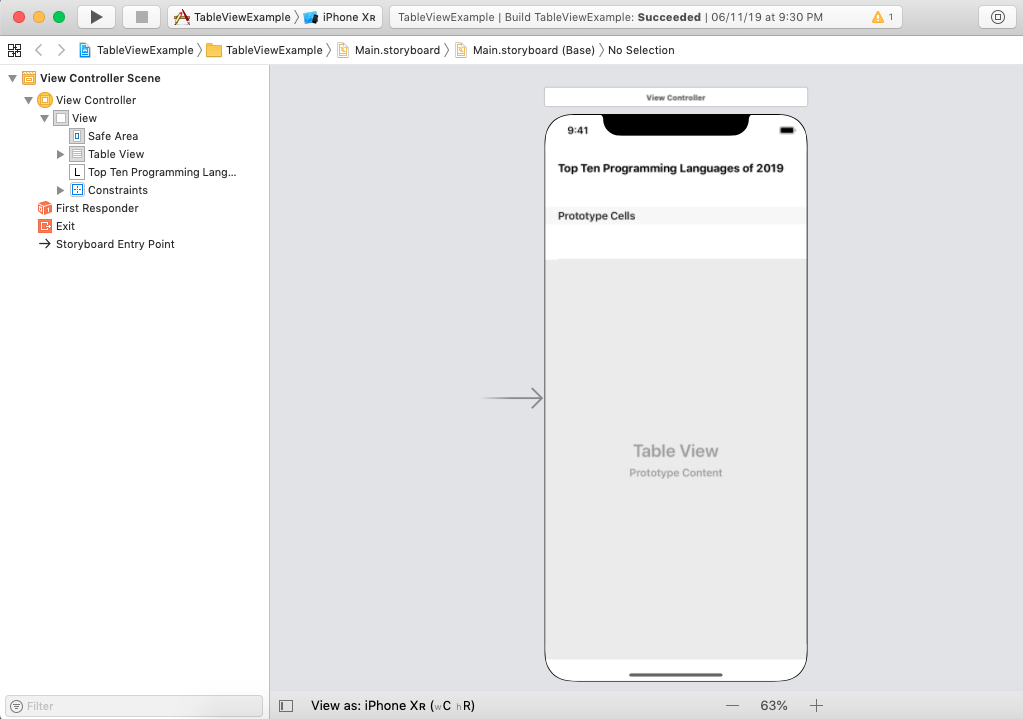
|  |  |  |
| --- | --- | --- |
| **SN** | **Method** | **Description** |
| 1 | func tableView(UITableView, numberOfRowsInSection: Int) -> Int | This method returns the number of rows to be displayed in the section of the tableview. |
| 2 | func numberOfSections(in: UITableView) -> Int | This method returns the number of sections to be displayed in the tableview. |
| 3 | func tableView(UITableView, cellForRowAt: IndexPath) -> UITableViewCell | This method returns the object of a UITableViewCell, which shows the actual content of a particular row in the tableview. This method inserts the cell for a particular row in the tableview. |
| 4 | func tableView(UITableView, titleForHeaderInSection: Int) -> String? | This method returns a string representing the title of the header in the section of the tableview. |
| 5 | func tableView(UITableView, titleForFooterInSection: Int) -> String? | This method returns a string representing the title of the footer in the section of the tableview. |
| 7 | func tableView(UITableView, canEditRowAt: IndexPath) -> Bool | It asks the DataSource to verify whether the particular row is editable or not. |
| 8 | func tableView(UITableView, canMoveRowAt: IndexPath) -> Bool | It asks the DataSource to verify whether the particular row can be moved to another location in the tableview. |
| 9 | func tableView(UITableView, moveRowAt: IndexPath, to: IndexPath) | This method moves the specific row to some other location in the tableview. |
| 10 | func sectionIndexTitles(for: UITableView) -> [String]? | It returns the array of the string containing the titles for the sections in the tableview. |

There are two methods that are required to be defined if the ViewController implements the UITableViewDatasource protocol, which is mentioned in the following code.

1. extension ViewController : UITableViewDataSource{
3. func tableView(\_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
4. **return** dataSourceArr.count
6. }
8. func tableView(\_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
9. let cell = tableView.dequeueReusableCell(withIdentifier: "cell", **for**: indexPath)
10. cell.textLabel?.text = "cell text"
12. **return** cell
13. }
15. }

## Interface Builder

In this example, we will create the following view controller by adding the tableview to the interface builder. We will also use the label object to show the title of the tableview. We will add a prototype cell to this tableview and assign ViewController.swift class for this ViewController.



**ViewController.swift**

In ViewController.swift, we will create the connection outlet of the tableview added to the storyboard. We will also define the delegate and datasource methods to display the tableview data.

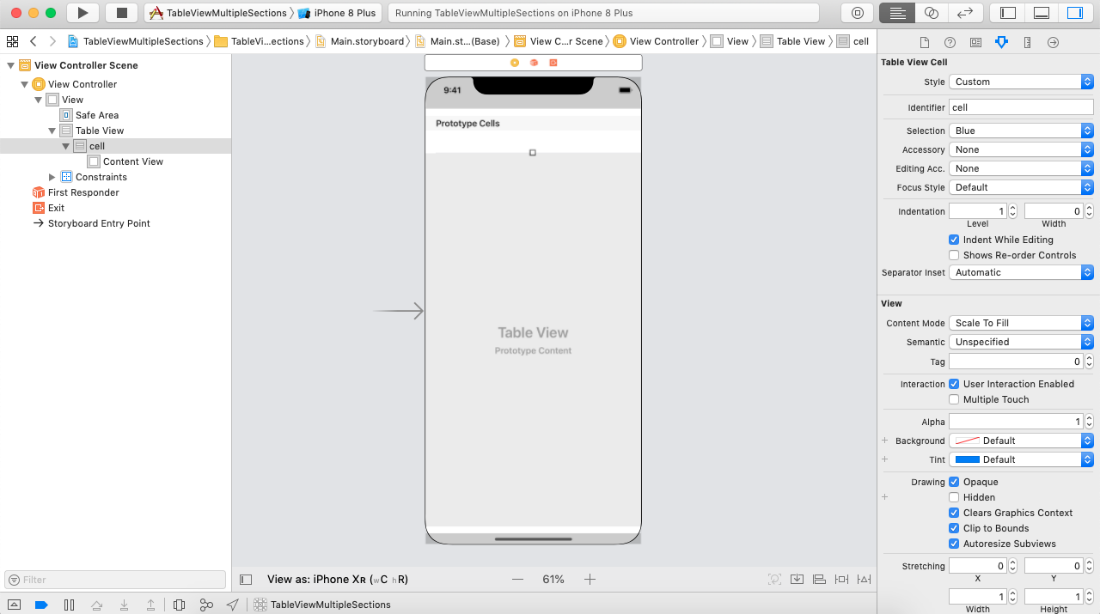
1. **import** UIKit
2. **class** ViewController: UIViewController {

5. @IBOutlet weak var tableView: UITableView!
6. var dataSourceArr = Array<String>()
8. override func viewDidLoad() {
9. **super**.viewDidLoad()
10. // Do any additional setup after loading the view.
11. tableView.delegate = self
12. tableView.dataSource = self
13. dataSourceArr = ["Python","JavaScript","Java","Swift","GoLang","C#","C++","Scala"]

16. }
17. }

20. extension ViewController : UITableViewDelegate{
22. }

25. extension ViewController : UITableViewDataSource{
27. func tableView(\_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
28. **return** dataSourceArr.count
30. }
32. func tableView(\_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
33. let cell = tableView.dequeueReusableCell(withIdentifier: "cell", **for**: indexPath)
34. cell.textLabel?.text = dataSourceArr[indexPath.row]
35. cell.textLabel?.textAlignment = .center
36. **return** cell
37. }
38. }
39. **Example: Handling multiple sections in the tableview**
40. In this example, we will create the multiple sections in the tableview, and we will define the variable number of rows and row content depending upon the particular section.
41. **Interface Builder**
42. **To create the interface builder for this example, we need to add a tableview and add a prototype cell for the tableview. The interface builder looks like the below image with a prototype cell.**



**ViewController.swift**

1. **import** UIKit

4. **class** ViewController: UIViewController {

7. @IBOutlet weak var tableView: UITableView!
9. override func viewDidLoad() {
10. **super**.viewDidLoad()
11. // Do any additional setup after loading the view.
12. tableView.delegate = self
13. tableView.dataSource = self
14. }
16. }

19. extension ViewController: UITableViewDataSource{
21. func numberOfSections(in tableView: UITableView) -> Int {
22. **return** 3
23. }
25. func tableView(\_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
26. **if** section == 0{
27. **return** 2
28. }
29. **else** **if** section == 1{
30. **return** 3
31. }
32. **else** **if** section == 2{
33. **return** 4
34. }
35. **return** 0
36. }
38. func tableView(\_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
39. let cell = tableView.dequeueReusableCell(withIdentifier: "cell", **for**: indexPath)
40. **if**(indexPath.section == 0){
41. cell.textLabel?.text = "Row in section 1"
42. }
43. **else** **if**(indexPath.section == 1){
44. cell.textLabel?.text = "Row in section 2"
45. }
46. **else** **if**(indexPath.section == 2){
47. cell.textLabel?.text = "Row in section 3"
48. }
49. **return** cell
50. }
51. }

54. extension ViewController : UITableViewDelegate{
56. func tableView(\_ tableView: UITableView, titleForHeaderInSection section: Int) -> String? {
57. **return** "Section " + (section+1).description
58. }
60. }

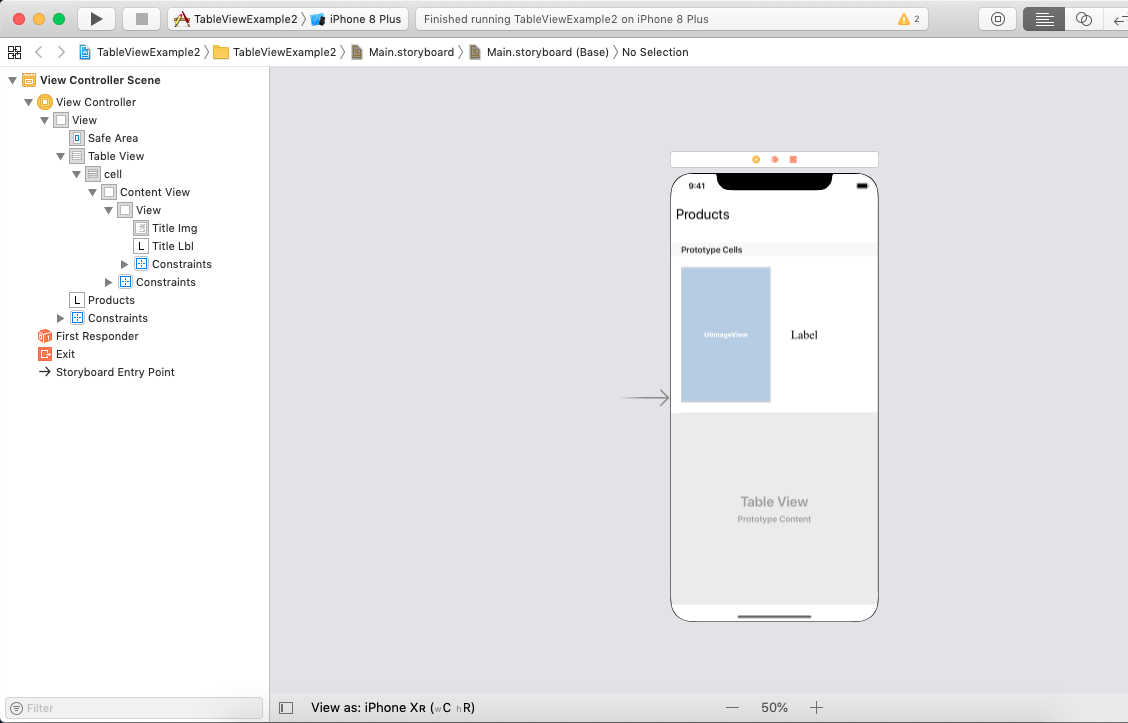
**Example 2: Customizing Table View cell**

In this example, we will customize the tableview cell by assigning it to a class and creating the outlets of the cell objects in that class. In most of the iOS applications, there are the requirements to customize the tableview class since we can-not always fulfill our requirements by just setting the label text of the cell.

This example simulates the list view of the products shown in the ECommerce Application.

**Interface Builder**

To create the interface builder for this example, we need to add the tableview to the view controller and add the prototype cell into it. In the content view prototype cell, we will add a uiview to which, we will add an UIImageView and UILabel objects. The following image shows the storyboard in the example.



**MyTableViewCell.swift**

MyTableViewCell inherits the UITableViewCell class, which is assigned to the prototype cell of the tableview. In this class, we can instantiate the image view and label objects.

1. **import** UIKit

4. **class** MyTableViewCell: UITableViewCell {

7. @IBOutlet weak var titleImg: UIImageView!
9. @IBOutlet weak var titleLbl: UILabel!

12. override func awakeFromNib() {
13. **super**.awakeFromNib()
14. // Initialization code
15. }

18. override func setSelected(\_ selected: Bool, animated: Bool) {
19. **super**.setSelected(selected, animated: animated)

22. // Configure the view for the selected state
23. }

26. }

**ViewController.swift**

1. **import** UIKit
3. **class** ViewController: UIViewController {
5. var imgArr = ["Product1","Product2","Product3","Product4","Product5","Product6","Product7","Product8"]
6. var lblTextArr = ["Powerbanks","Storage Devices","LED Bulbs","Laptop Bags","Keyboards","Routers","Shoes"]
8. @IBOutlet weak var tableView: UITableView!

11. override func viewDidLoad() {
12. **super**.viewDidLoad()
13. // Do any additional setup after loading the view.
15. tableView.delegate = self
16. tableView.dataSource = self
17. }
19. }



24. extension ViewController : UITableViewDataSource{
26. **public** func tableView(\_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
27. **return** imgArr.count - 1
28. }
30. **public** func tableView(\_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
31. let cell = tableView.dequeueReusableCell(withIdentifier: "cell", **for**: indexPath) as! MyTableViewCell
32. cell.titleImg?.image = UIImage(named: imgArr[indexPath.row])
33. cell.titleLbl.text = lblTextArr[indexPath.row]
34. **return** cell
35. }
36. }

39. extension UIViewController : UITableViewDelegate{
41. **public** func tableView(\_ tableView: UITableView, heightForRowAt indexPath: IndexPath) -> CGFloat {
42. **return** 150
43. }
45. }



S