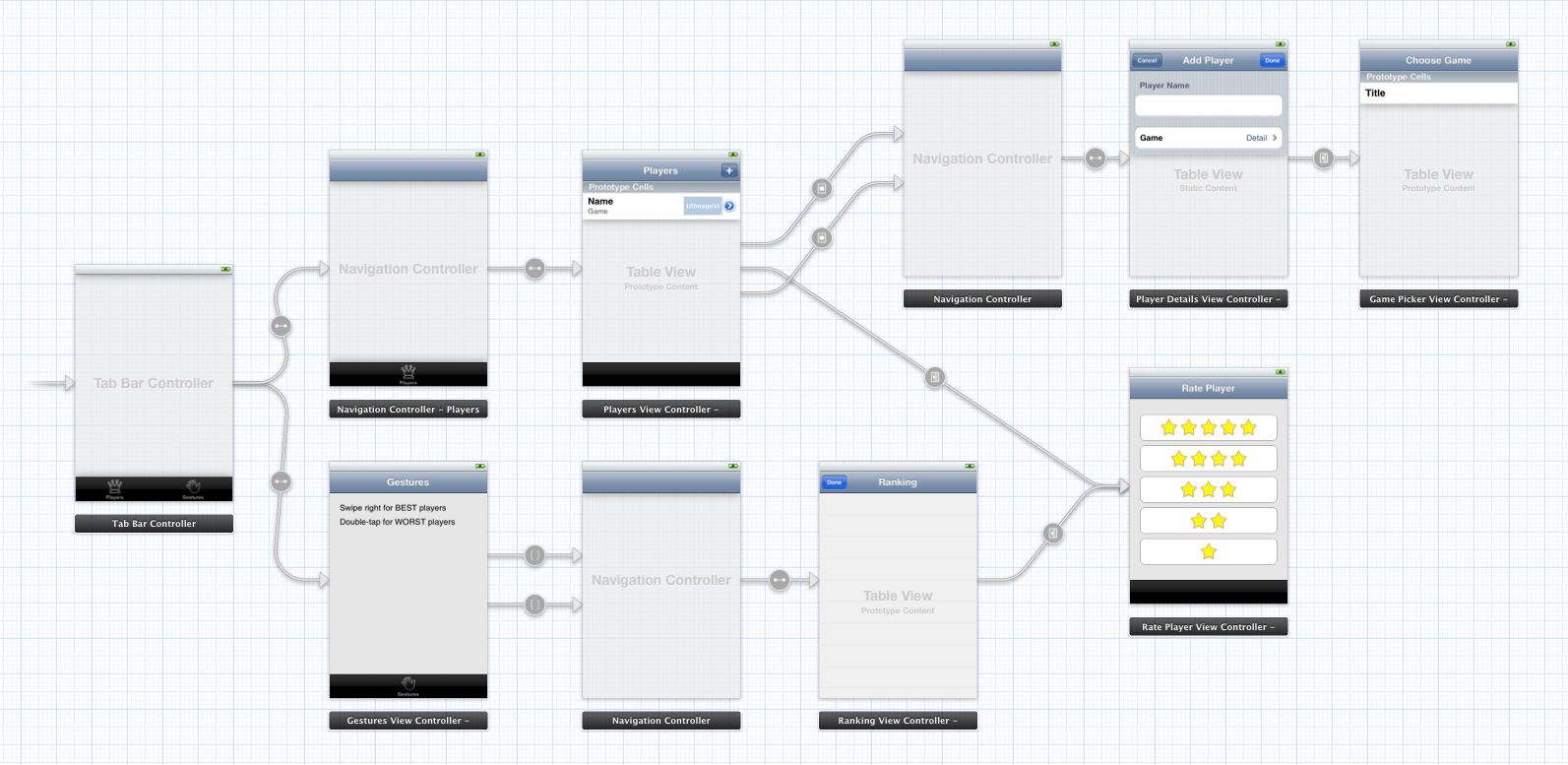
StoryBoards

Storyboarding is an exciting new feature in iOS 5 that will save you a lot of time building user interfaces for your apps. To show you what a storyboard is, I’ll let a picture do the talking. This is the storyboard that we will be building in this tutorial:



You may not know exactly yet what the app does but you can clearly see which screens it has and how they are related. That is the power of using storyboards.

If you have an app with many different screens then storyboards can help reduce the amount of glue code you have to write to go from one screen to the next. Instead of using a separate nib file for each view controller, your app uses a single storyboard that contains the designs of all of these view controllers and the relationships between them.

Storyboards have a number of advantages over regular nibs:

* With a storyboard you have a better conceptual overview of all the screens in your app and the connections between them. It’s easier to keep track of everything because the entire design is in a single file, rather than spread out over many separate nibs.
* The storyboard describes the transitions between the various screens. These transitions are called “segues” and you create them by simply ctrl-dragging from one view controller to the next. Thanks to segues you need less code to take care of your UI.
* Storyboards make working with table views a lot easier with the new prototype cells and static cells features. You can design your table views almost completely in the storyboard editor, something else that cuts down on the amount of code you have to write.

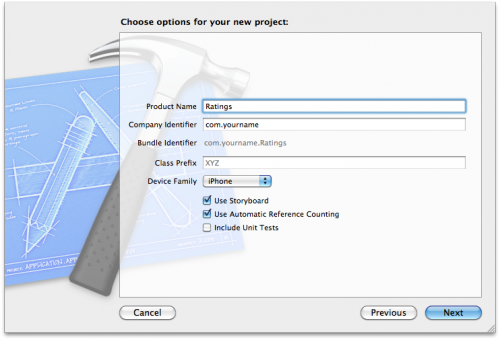
Not everything is perfect, of course, and storyboards do have some limitations. The Storyboard Editor isn’t as powerful as Interface Builder yet, there are a few things IB can do that the Storyboard Editor unfortunately can’t. You also need a big monitor, especially when you write iPad apps!

If you’re the type who hates Interface Builder and who really wants to create his entire UI programmatically, then storyboards are probably not for you. Personally, I prefer to write as little code as possible — especially UI code! — so this tool is a welcome addition to my arsenal.

You can still use nibs with iOS 5 and Xcode 4.2. Using Interface Builder isn’t suddenly frowned upon now that we have storyboards. If you want to keep using nibs then go right ahead, but know that you can combine storyboards with nibs. It’s not an either-or situation.

In this tutorial we’ll take a look at what you can do with storyboards.

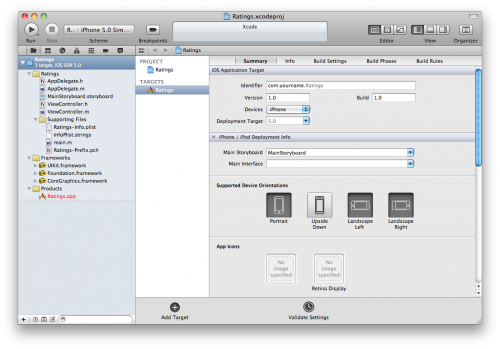
Fire up Xcode and create a new project. We’ll use the Single View Application template as our starting point and then build up the app from there.



Fill in the template options as follows:

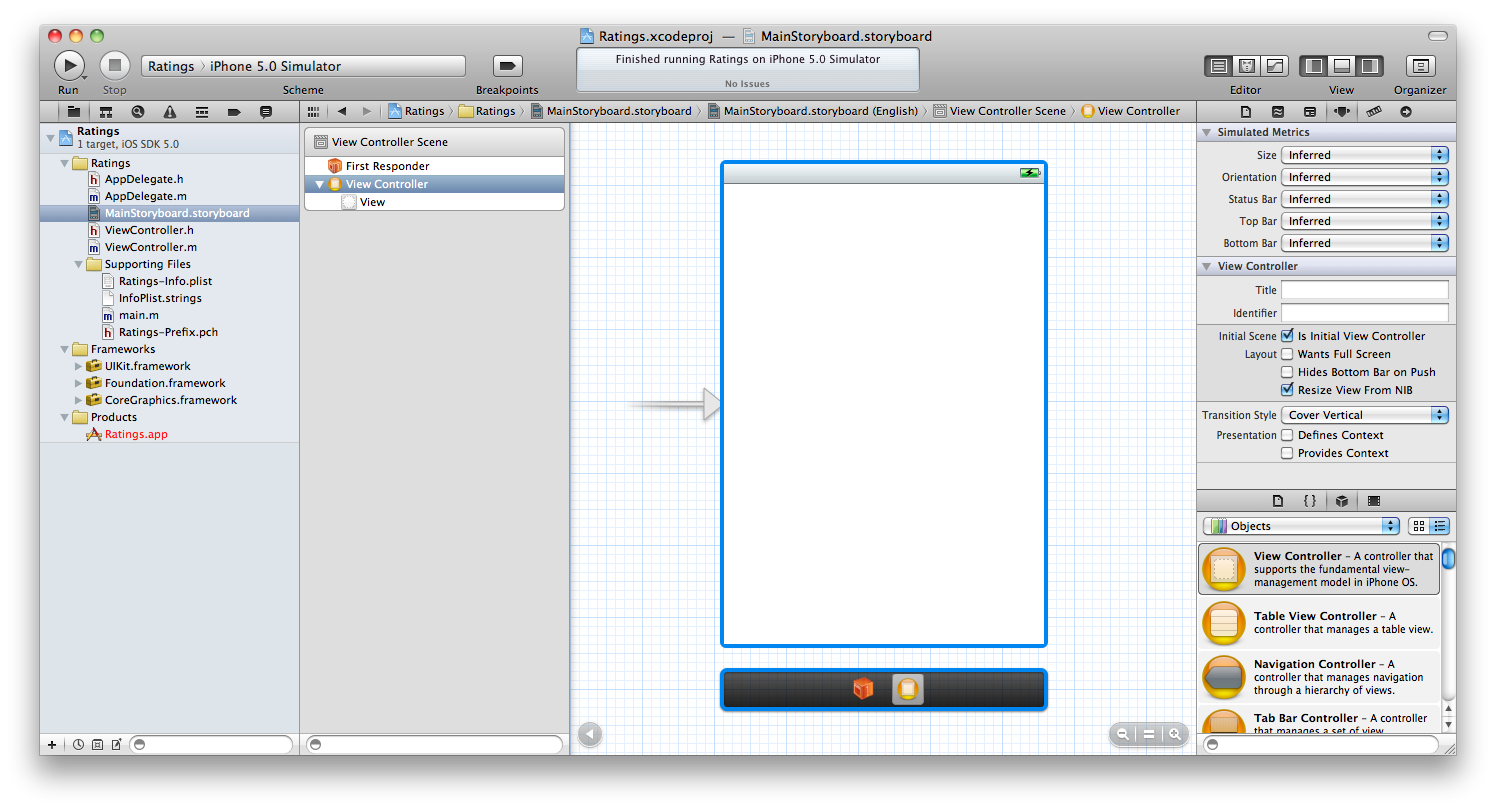
* *Product Name*: Ratings
* *Company Identifier*: the identifier that you use for your apps, in reverse domain notation
* *Class Prefix*: leave this empty
* *Device Family*: iPhone
* *Use Storyboard*: check this
* *Use Automatic Reference Counting*: check this
* *Include Unit Tests*: this should be unchecked

After Xcode has created the project, the main Xcode window looks like this:



Our new project consists of two classes, AppDelegate and ViewController, and the star of this tutorial: the MainStoryboard.storyboard file. Notice that there are no .xib files in the project, not even MainWindow.xib.

Let’s take a look at that storyboard. Click the MainStoryboard.storyboard file in the Project Navigator to open the Storyboard Editor:

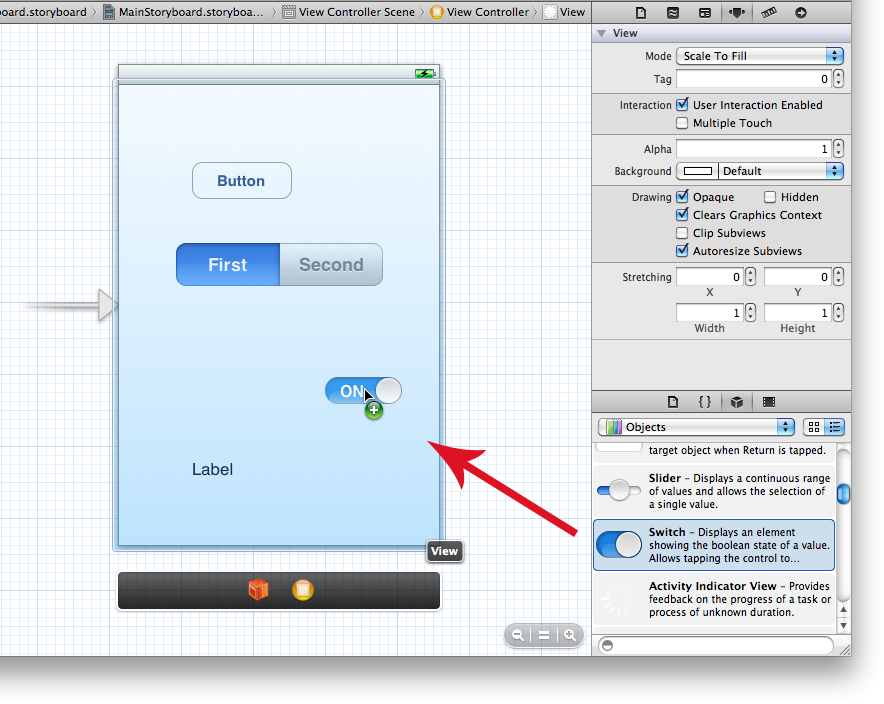


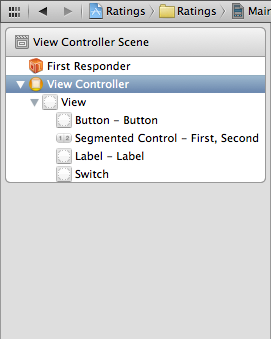
The Storyboard Editor looks and works very much like Interface Builder. You can drag new controls from the Object Library (see bottom-right corner) into your view controller to design its layout. The difference is that the storyboard doesn’t contain just one view controller from your app, but all of them.

The official storyboard terminology is “scene”, but a scene is really nothing more than a view controller. Previously you would use a separate nib for each scene / view controller, but now they are all combined into a single storyboard.

On the iPhone only one of these scenes is visible at a time, but on the iPad you can show several at once, for example the master and detail panes in a split-view, or the content of a popover.

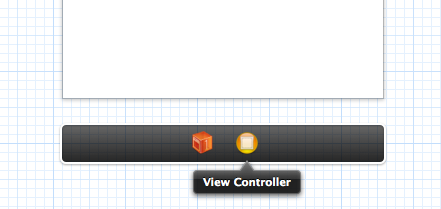
To get some feel for how the editor works, drag some controls into the blank view controller:



The sidebar on the left is the Document Outline: 

In Interface Builder this area lists just the components from your nib but in the Storyboard Editor it shows the contents of all your view controllers. Currently there is only one view controller in our storyboard but in the course of this tutorial we’ll be adding several others.

There is a miniature version of this Document Outline below the scene, named the Dock:



The Dock shows the top-level objects in the scene. Each scene has at least a First Responder object and a View Controller object, but it can potentially have other top-level objects as well. More about that later. The Dock is convenient for making connections. If you need to connect something to the view controller, you can simply drag to its icon in the Dock.

Note: You probably won’t be using the First Responder very much. This is a proxy object that refers to whatever object has first responder status at any given time. It was also present in Interface Builder and you probably never had a need to use it then either. As an example, you could hook up the Touch Up Inside event from a button to First Responder’s cut: selector. If at some point a text field has input focus then you can press that button to make the text field, which is now the first responder, cut its text to the pasteboard.

Run the app and it should look exactly like what we designed in the editor:

If you’ve ever made a nib-based app before then you always had a MainWindow.xib file. This nib contained the top-level UIWindow object, a reference to the App Delegate, and one or more view controllers. When you put your app’s UI in a storyboard, however, MainWindow.xib is no longer used.



The Xcode project contains another interesting file, **Main.storyboard**. The storyboard defines how the user interface of our application will look like. By default, the storyboard is named **Main.storyboard**. Select the storyboard to open it.

So how does the storyboard get loaded by the app if there is no MainWindow.xib file?

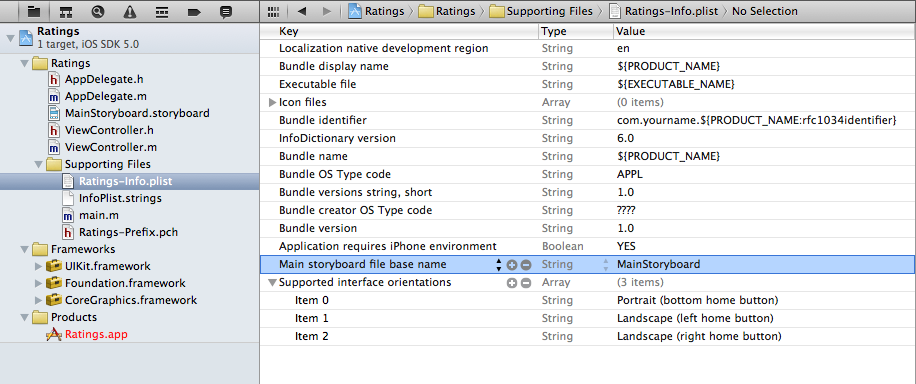
Let’s take a peek at our application delegate. Open up AppDelegate.h and you’ll see it looks like this:

|  |
| --- |
| #import <UIKit/UIKit.h>    @interface AppDelegate : UIResponder <UIApplicationDelegate>    @property (strong, nonatomic) UIWindow \*window;    @end |

It is a requirement for using storyboards that your application delegate inherits from UIResponder (previously it used to inherit directly from NSObject) and that it has a UIWindow property (unlike before, this is not an IBOutlet).

If you look into AppDelegate.m, you’ll see that it does absolutely nothing, all the methods are practically empty. Even application:didFinishLaunchingWithOptions: simply returns YES. Previously, this would either add the main view controller’s view to the window or set the window’s rootViewController property, but none of that happens here.

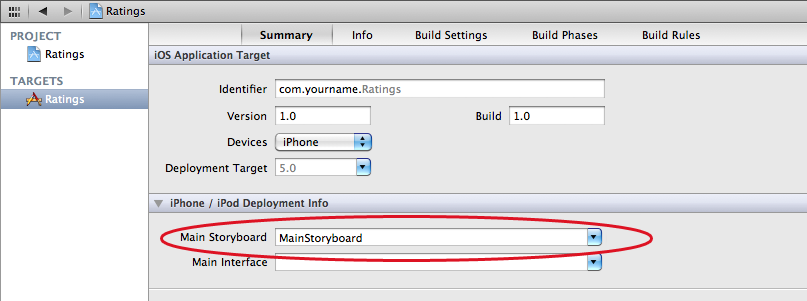
The secret is in the Info.plist file. Click on Ratings-Info.plist (it’s in the Supporting Files group) and you’ll see this



In nib-based projects there was a key in Info.plist named NSMainNibFile, or “Main nib file base name”, that instructed UIApplication to load MainWindow.xib and hook it into the app. Our Info.plist no longer has that setting.

Instead, storyboard apps use the key UIMainStoryboardFile, or “Main storyboard file base name”, to specify the name of the storyboard that must be loaded when the app starts. When this setting is present, UIApplication will load the MainStoryboard.storyboard file and automatically instantiates the first view controller from that storyboard and puts its view into a new UIWindow object. No programming necessary.

You can also see this in the Target Summary screen:



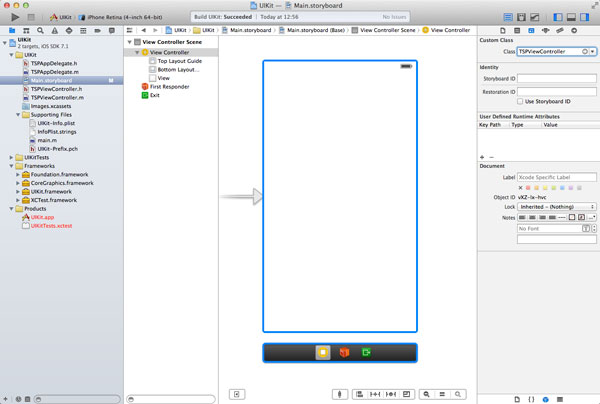
There is a new iPhone/iPod Deployment Info section that lets you choose between starting from a storyboard or from a nib file.

For the sake of completeness, also open main.m to see what’s in there:

|  |
| --- |
| #import <UIKit/UIKit.h>    #import "AppDelegate.h"    int main(int argc, char \*argv[])  {  @autoreleasepool {  return UIApplicationMain(argc, argv, nil,  NSStringFromClass([AppDelegate class]));  }  } |

Previously, the last parameter for UIApplicationMain() was nil but now it is NSStringFromClass([AppDelegate class]).

A big difference with having a MainWindow.xib is that the app delegate is not part of the storyboard. Because the app delegate is no longer being loaded from a nib (nor from the storyboard), we have to tell UIApplicationMain specifically what the name of our app delegate class is, otherwise it won’t be able to find it.

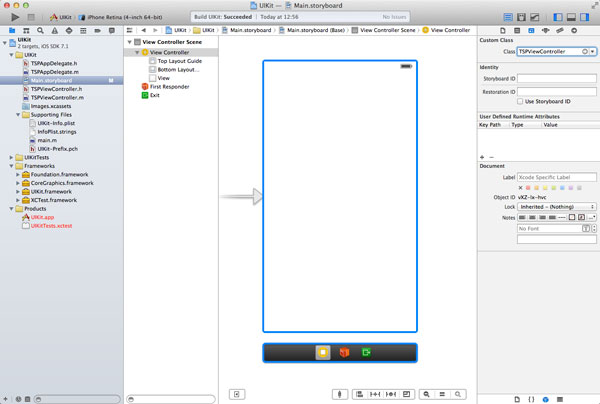


The storyboard currently contains one view in the central workspace. On the right of the **Project Navigator** you can see a list of items, which are the objects you see in the view. The top item is named **View Controller Scene**, which contains one child item labeled **View Controller**.

The **View Controller**object also has a number of child items, but there is one that is of special interest to us, the object named **View**. Remember our discussion about the MVC pattern. Here you can see the MVC pattern in action. The model is missing at the moment, but we do have a view, the **View** object, and a controller, the **View Controller** object.

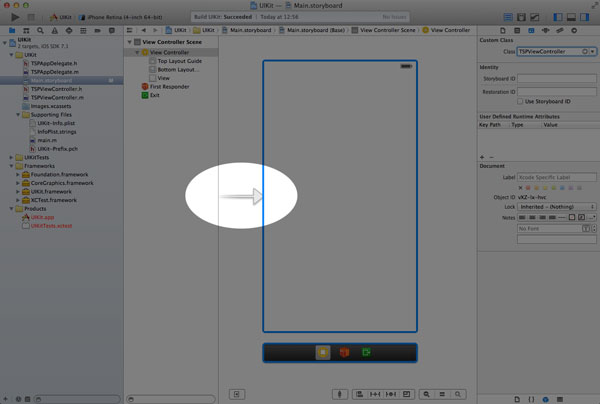
When our application launches, the storyboard is used to create the application's user interface. The view controller is automatically instantiated and so is the view controller's view. The **View** object in the storyboard is managed by the view controller.

Wait a minute. Where can I find the class of the view controller in the storyboard? How can I change its behavior to create a unique application? Select the **View Controller** object on the left in the storyboard and open the **Identity Inspector** on the right.



The **Identity Inspector** tells you everything you need to know. At the top, in the section **Custom Class**, you can see the name of the view controller's class, TSPViewController. Have you noticed that the two files we haven't talked about yet have that same name? We'll explore these files in a few moments.

The view controller is instantiated for us, because it is the initial view controller of the storyboard. This is indicated in the storyboard by the arrow pointing to the storyboard.

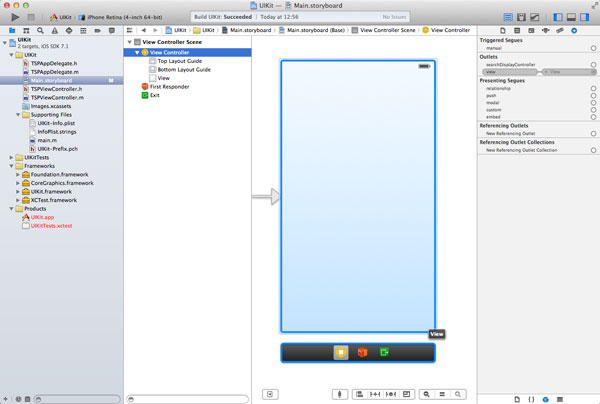


**UIViewController**

If you open **TSPViewController.h**, you'll notice that the TSPViewController class is a subclass of UIViewController. Like TSPAppDelegate, the UIViewController class is a subclass of UIResponder. View controllers, or subclasses thereof, fall in the **controller** category of the MVC pattern. As the name implies, they control a view, an instance of the UIView class, which falls in the **view** category of the MVC pattern.

A view controller manages a view and the view's subviews as we'll see later. To do this, the view controller needs to know about the view. In other words, it needs to have a reference to the view.

The view controller in the storyboard has a reference to the view. You can verify this by selecting the view controller in the storyboard and opening the **Connections Inspector** on the right.

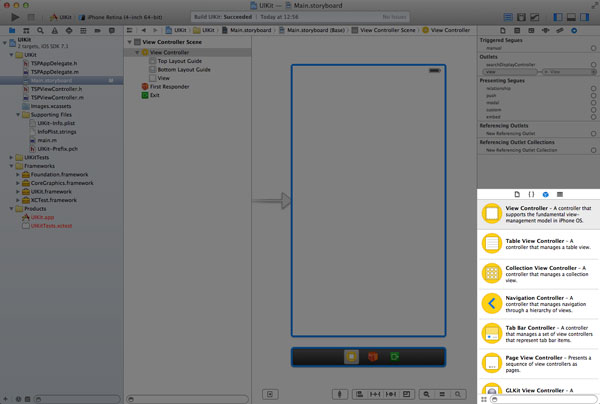


In the **Connections Inspector**, you should see a section named **Outlets**. The term outlet is a fancy word for a property, which you can set in the storyboard. Hover with your mouse over the outlet named **view** and observe how the view in the workspace is highlighted. That is the connection between the view controller and the view.

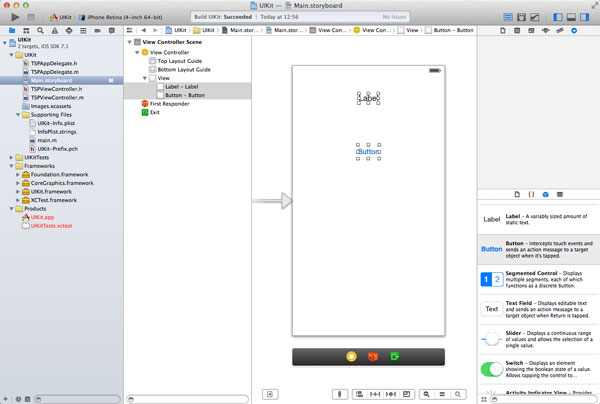
## UIView

Even though your application can have only one instance of UIWindow, it can have many views. The UIView class is an important component of the UIKit framework, because many classes inherit from it—either directly or indirectly.

Revisit **Main.storyboard** by selecting it and take a look at the **Object Library** at the bottom of the **Inspector**.



Browse the Object Library and drag a **label** and a **button** to the view in the workspace. It doesn't matter where you position them in the view as long as they are in the view controller's view.



You'll have noticed that two new objects have been added to the **Objects** section on the left. Both the label (UILabel) and the button (UIButton) inherit from UIView. Did you notice that the **Label** and **Button** objects are slightly indented compared to the **View** object? This indicates that the **Label** and **Button** objects are subviews of the **View** object. A view can have one or more subviews that it manages.

As I mentioned earlier, the UIView class is an important component of UIKit. A view manages a rectangular area or frame on the screen. It manages the contents of the area, the subviews, and any interactions with the view's contents. The UIView class is a subclass of UIResponder. You'll learn much more about views over the course of this series.

**Outlets**

Let's take a look at an example to illustrate the relationship between the storyboard, the view it contains, and the view controller. These three components are important and I want to make sure that you understand just how they work together.

A few minutes ago, you added a label and a button to the view controller's view. How does the view controller know about these objects? At the moment, they don't appear in the **Connections Inspector**, but we can change that by telling the view controller about them.

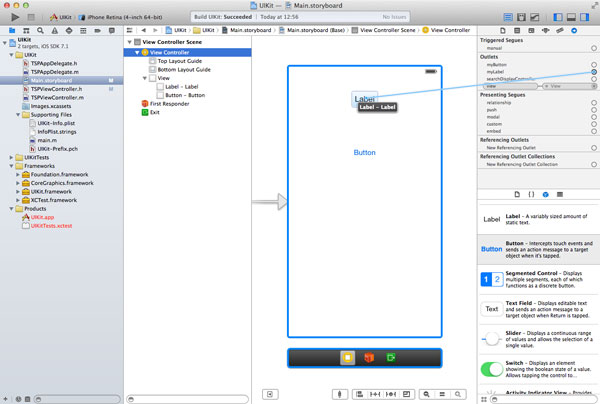
Open the view controller's header file (**TPSViewController.h**) and add a property for the label and for the button.

|  |  |
| --- | --- |
| 1  2 | @property IBOutlet UILabel \*myLabel;  @property IBOutlet UIButton \*myButton; |

By adding the IBOutlet keyword to the property declaration, the properties will appear in the Connections Inspector in the storyboard and that is what we want.

Head back to the storyboard, select the **View Controller** object in the **View Controller Scene**, and open the **Connections Inspector** on the right. The new properties are now listed in the list of **Outlets**. However, the view controller hasn't yet made the **connection** between the new properties and the objects in the storyboard.

This is easy to remedy. Drag from the empty circle on the left of the myLabel outlet to the label in the workspace. This will create that all-important connection so that the view controller knows about the label. Do the same for the button.



Even though we can change the text of the label in the storyboard, let's do this in the view controller to illustrate that the view controller has access to the label and button in the storyboard.

Open the view controller's implementation file (**TPSViewController.m**) and look for the viewDidLoad method. Modify the viewDidLoad method to reflect the implementation below. The comments have been omitted for clarity.

|  |  |
| --- | --- |
| 1  2  3  4  5 | - (void)viewDidLoad {      [super viewDidLoad];        [self.myLabel setText:@"This is an instance of UILabel."];  } |

We can send messages to the label property by asking the view controller, self, for its myLabel property. By sending the myLabel property a message of setText: and passing a string literal, we update the label's text.

Note that setText: is an accessor or setter. Even though it's possible to use Objective-C's dot notation (see below), I've used the more traditional syntax as it better shows you what's actually happening.

|  |  |
| --- | --- |
| 1 | self.myLabel.text = @"This is an instance of UILabel"; |

Run your application in the iOS Simulator by clicking the **Run** button in the top left and notice that the label's text is indeed updated.

**Actions**

We've explored a lot of new things in this article. I want to end this installment by talking about actions. Just like outlets, actions are nothing more than methods that you can see in the storyboard.

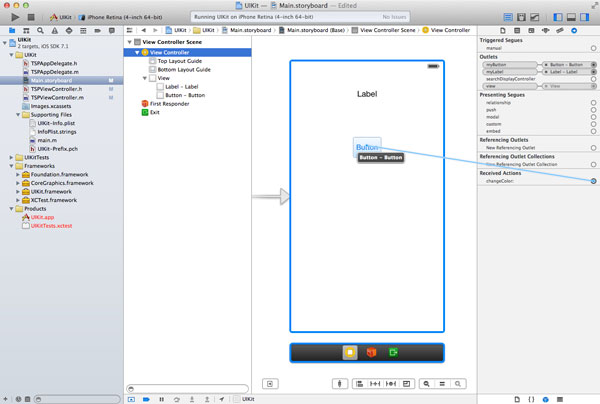
Let's see how this works. Open the view controller's header file (**TPSViewController.h**) and add the following method declaration somewhere in the view controller's interface block.

|  |  |
| --- | --- |
| 1 | - (IBAction)changeColor:(id)sender; |

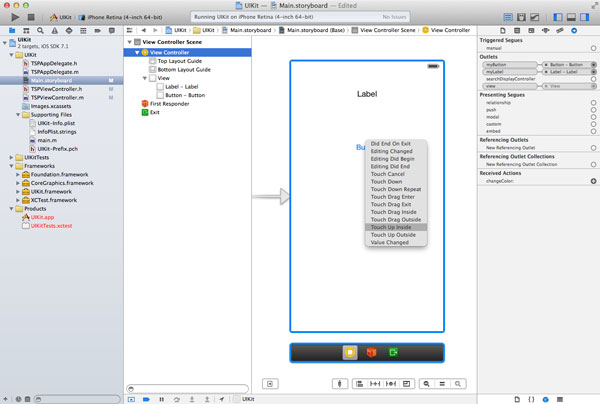
Don't be confused by the IBAction keyword. IBAction is identical to void, which means that the method returns no value. If we take a closer look at the method declaration, we can see that it takes one argument of type id, a reference to an Objective-C object.

As the argument name implies, the argument of the method or action is the object that sent the message to the view controller. I will explain this in more detail in just a bit.

Revisit the storyboard, select the **View Controller** object in the **View Controller Scene**, and open the **Connections Inspector**. A new section has appeared in the **Connections Inspector** named **Received Actions** and the action we just added is listed in this section.



Drag from the empty circle on the left of the action to the button in the workspace. A small window with a list of options should appear. The list contains all the possible events that the button can respond to. The one that interests us is **Touch Up Inside**. This event is triggered when a user touches the button and lifts their finger while still inside the button

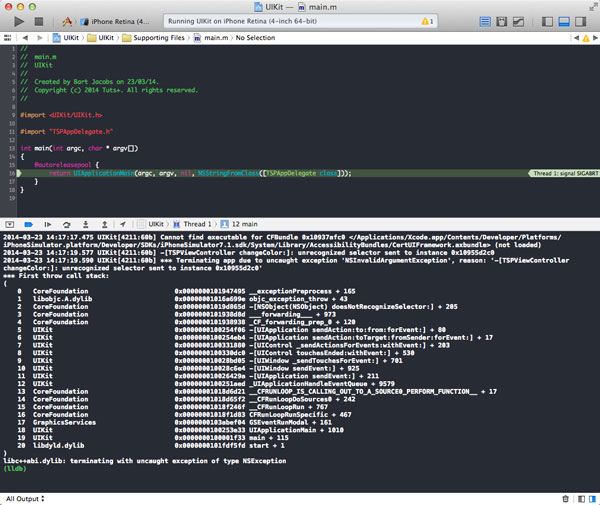


Build and run your application once again and tap the button. Did the application crash for you as well? How did this happen? When you tapped the button, it sent a message of changeColor: to the view controller. Even though the view controller declares a changeColor: method, it does not implement this method yet.

Whenever a message is sent to an object that does not implement a corresponding method, an exception is raised and the application crashes if the exception isn't caught. You can verify this by running the application once more, tapping the button, and inspecting the output in the console window.

Build and run your application once again and tap the button. Did the application crash for you as well? How did this happen? When you tapped the button, it sent a message of changeColor: to the view controller. Even though the view controller declares a changeColor: method, it does not implement this method yet.

Whenever a message is sent to an object that does not implement a corresponding method, an exception is raised and the application crashes if the exception isn't caught. You can verify this by running the application once more, tapping the button, and inspecting the output in the console window.



To remedy this, we need to implement the changeColor: method. Open the view controller's implementation file (**TPSViewController.m**) and add the following method implementation somewhere in the implementation block.

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12 | - (IBAction)changeColor:(id)sender {      NSLog(@"Sender Class > %@", [sender class]);      NSLog(@"Sender Superclass > %@", [sender superclass]);        int r = arc4random() % 255;      int g = arc4random() % 255;      int b = arc4random() % 255;        UIColor \*color = [UIColor colorWithRed:(r/255.0) green:(g/255.0) blue:(b/255.0) alpha:1.0];        [self.view setBackgroundColor:color]; |

The implementation of the changeColor: method is identical to the one we used [earlier in this series](http://mobile.tutsplus.com/tutorials/iphone/creating-your-first-ios-application/). However, I've added two extra NSLog calls to its implementation to show you that the sender of the message is indeed the button that we added to the storyboard.

The method itself is pretty simple. We generate three random integers between 0 and 255, pass these values to colorWithRed:green:blue:alpha: to generate a random color, and update the background color of the view controller's view with the randomly generated color.

Note that self.view references the view that the view controller manages and that we saw earlier in the storyboard.

Build and run your application one more time, tap the button, and don't forget to inspect the output in Xcode's console window. You'll notice that the sender is an instance of UIButton and its superclass is UIControl.

References:

<http://www.raywenderlich.com/5138/beginning-storyboards-in-ios-5-part-1>

<http://kurrytran.blogspot.in/2011/07/simple-ios-5-tutorial-using-storyboard.html>

New Tutorial: <http://kurrytran.blogspot.com/2011/10/ios-5-storyboard-and.html>

New Tutorial: <http://kurrytran.blogspot.com/2011/10/ios-5-storyboard-uitableview-tutorial.html>

New Tutorial: <http://kurrytran.blogspot.com/2013/06/fast-scrolling-customized-uitableviews.html>

<http://www.raywenderlich.com/50317/beginning-auto-layout-tutorial-in-ios-7-part-1>

http://www.tutorialspoint.com/ios/ios\_memory\_management.htm