CSCI 4100 Laboratory 3

Introduction to IOS

Introduction

This laboratory is an introduction to iOS development for the iPhone and the iPad. We will produce a simple application in the same way that we did for Android that you can run in the simulator or on one of our iOS devices. The process of getting an iOS application onto a device is more complex than an Android one, since you must establish credentials with Apple before you can install the application. The process of acquiring these credentials takes some time, so we will start it at the beginning of the lab and by the end of the laboratory everything should be ready to go. In the meantime you can create your first iOS application and test it in the simulator.

Apple Developer Program

There are two things that you need to install applications on a device. The first is a developer certificate, which identifies you as an Apple developer. The second is a provisioning profile that allows you to sign applications for use on our test devices. To get a developer certificate you must first be a member of our team. The university is part of the Academic Developer program at Apple, so we must add you to that program first. To get this started send an email to the TA, so he has your email address along with your first and last name. The TA will then send you an invitation to join the team. You will need to respond to this invitation immediately (we need this information for the next step that we want to complete before the end of the laboratory). You will need an Apple ID for this response, so if you don't already have one you should create one at apple.ca.

The rest of the process can be done within Xcode, the iOS development environment. To find Xcode start the finder and select Applications from the Favorites on the left side of the window. At the bottom of the right pane you will find the Xcode application. You can either launch it directly from there by double clicking on the icon, or you can drag the Xcode icon to the dock (the list of applications running across the bottom on the screen) so you can launch it from there in the future.

Once you have started Xcode, choose Organizer from the Windows menu and then press the Devices button at the top of the window that is displayed. From the Editor menu select "Refresh from Developer Portal". After doing that you will get a dialogue similar to the one shown in figure 1. You should enter your Apple ID and password at this point and press the "Log In" button. At this point Xcode will ask you if you want it to request a development certificate for you (see figure 2), you should press the "Submit Request" button at this point. After Xcode has retrieved the certificate it will ask you if you want to export them. I recommend that you do this by following the set of steps after pressing the Export button. It will ask you for a file to store the certificates on; you should save it on the server and also on a USB key. This serves as a backup for your certificates and makes it possible to use them outside of the lab.

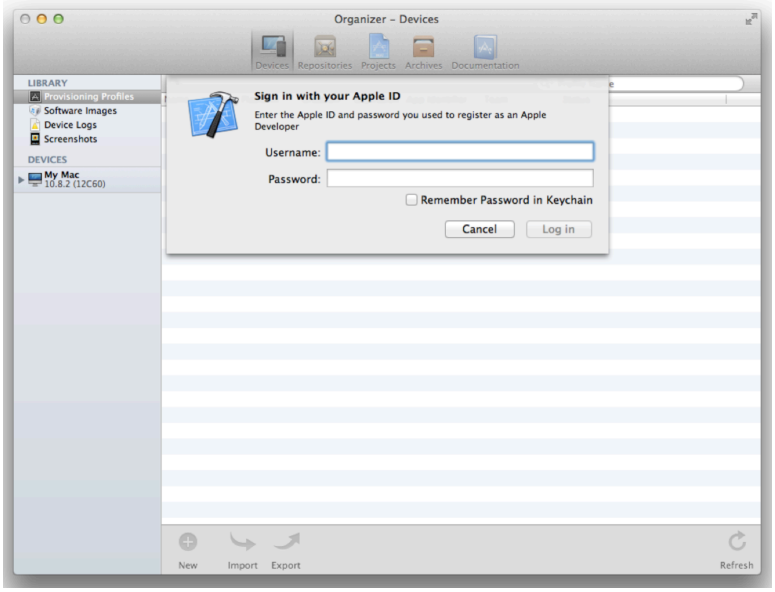

Figure 1

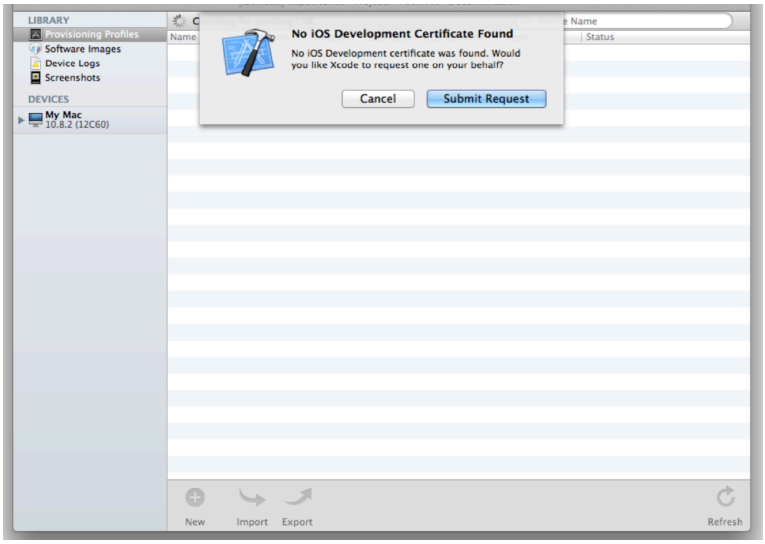

Figure 2

Our First iOS Application

We are now ready to start producing our first iOS application; figure 3 shows what this application will look like. Start by creating a new project, either from the Xcode start screen (see figure 4) or by selecting New from the File menu and then "Project …". This will produce the dialogue shown in figure 5 where you select "Single View Application".

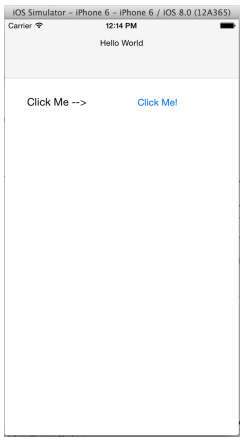

Figure 3 First iPhone Application

Figure 4 Xcode Start Screen

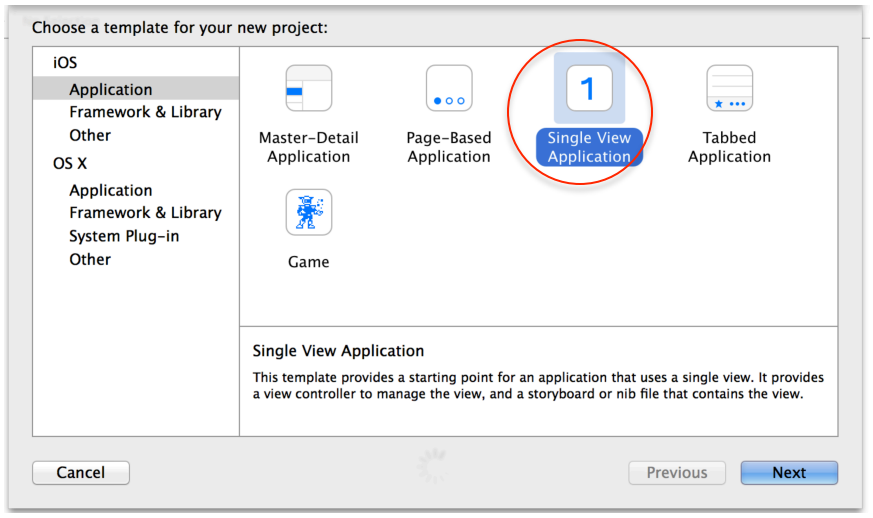

Figure 5 Choose Single View Application

Click Next and you will be taken to the dialogue in figure 6. Start by filling in the application name; I've used Mechanics, since this lab is all about the mechanics of building an iPhone application. Next fill in the organization name; you can just use your

own name. This is followed by company identifier, which is the prefix of the bundle identifier, which is similar to the namespace or package name used in Java. Under language select Swift (the default is Objective-C), since we will be using Swift for all of the labs. In devices make sure that iPhone is selected. You can now press Next and choose a location for your project. I suggest having a separate folder for each lab. Once you have done that you will be taken to the initial project window shown in figure 7.

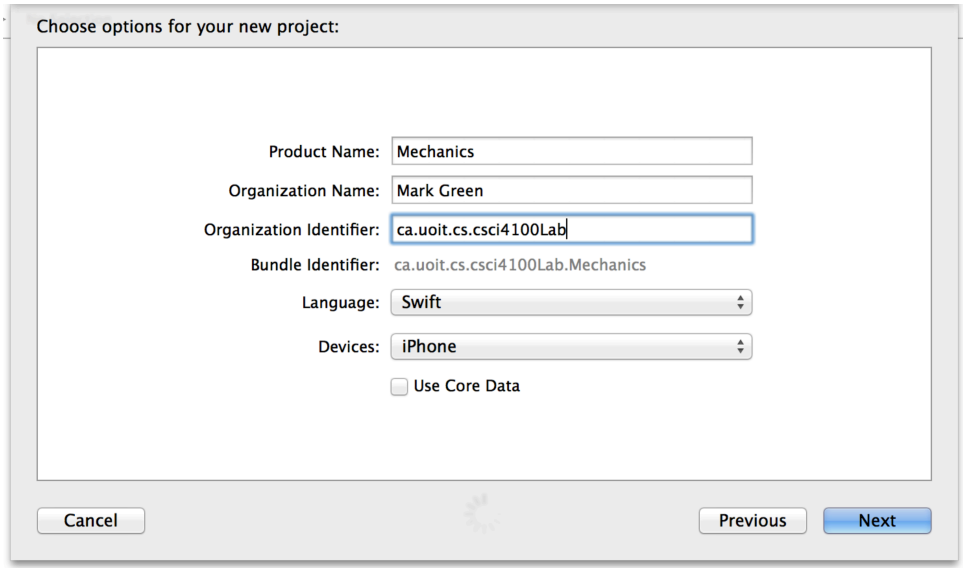

Figure 6 Project Options dialogue

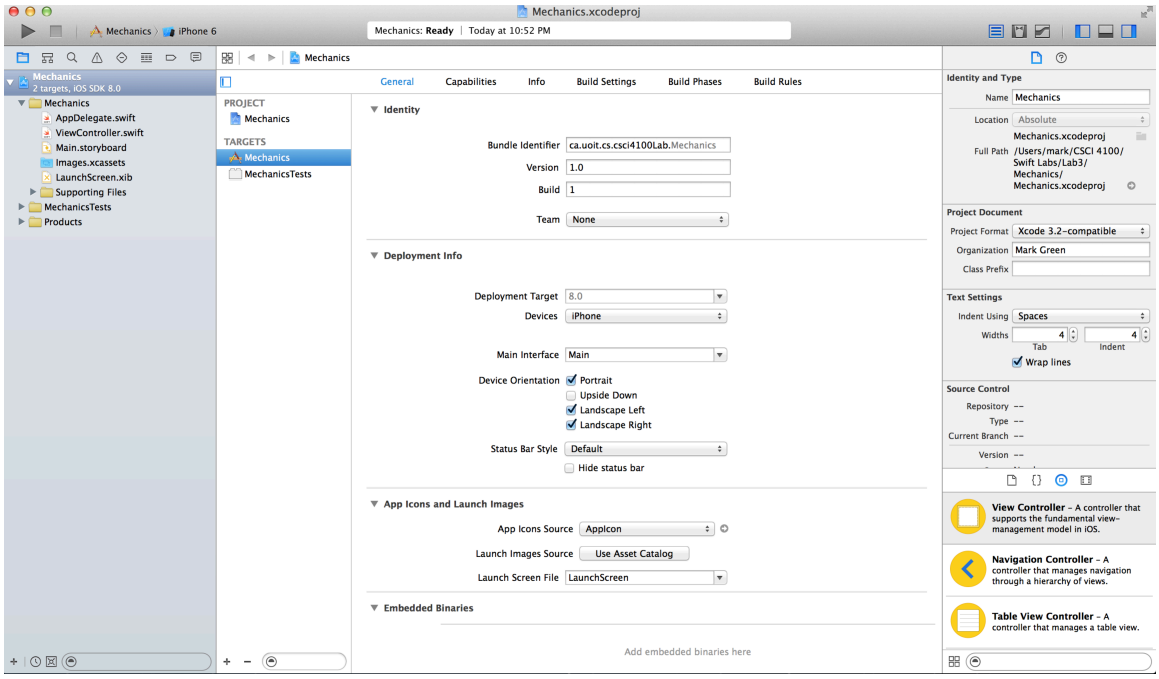

Figure 7 Initial project window

The initial project window displays the project properties and we can change many of them through the different tabs. We won't need to worry about this for now, but we will

come back to it later. Before making any changes, ensure that the project builds and runs properly. You can do this by clicking on the Run button in the toolbar, going to Project menu and selecting Run from the menu, or by pressing ⌘R (if using a non-Apple keyboard, substitute ⌘ with the Windows key). This will run our empty application in the iPhone simulator (again be patient waiting for it to start). We start off with an empty application, not as interesting as the initial Android one.

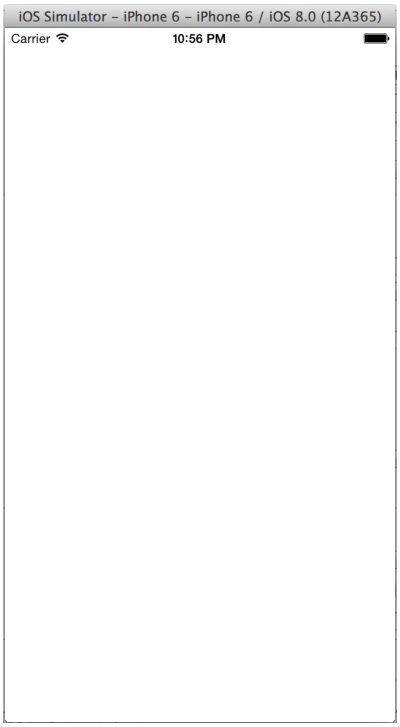

Figure 8 Initial application version in iPhone simulator

Outlets and Actions

In Cocoa UI elements are bound to the code via outlets and actions. Outlets allow UI elements to be referenced in code, and actions allow bits of code to be triggered whenever an action happens in the UI.

For example, an application could have a label that is updated when a button is pressed. In this case, the label is the UI element tied to an outlet in the code, and the button's click event is tied to an action in the code.

In code these are defined as **IBOutlet** and **IBAction**.

Program Organization

Objective-C was the original language used to develop iOS applications; it is also used to develop Mac OSX applications as well. This is an interesting programming language, but many programmers found it difficult to learn due to its syntax and its object model. In the summer of 2014 Apple introduced a new programming language call Swift that makes it easier to develop iOS and OSX applications. This is a more modern language

that is more familiar to most programmers. Applications can still be written on Objective-C and in a combination of the two languages, so there is a way of translating from one language to the other. In addition, the iOS APIs are all written on Objective-C. As a result, there are some Objective-C features that are reflected in Swift making it not as clean a language as I would like. If you are familiar with C++ or Java you will find the basics of Swift to be familiar; we will highlight some of the differences in the next lab.

Open up the MechanicsViewController.h file that Xcode created for the project. Inside you will find the following code:

On the left side of the XCode screen you will see the project explorer. Open the ViewController.swift file and take a quick look at its contents:

```
//
//  ViewController.swift
//  Mechanics
//
//  Created by Mark Green on 2014-09-18.
//  Copyright (c) 2014 Mark Green. All rights reserved.
//

import UIKit

class ViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view,
typically from a nib.
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated.
    }


}
```

In Swift there is no separate .h file as you would find with C++, everything is contained in the .swift file. The import statement brings in all the definitions from the UIKit module, which defines all of the user interface elements. Each screen in an iOS application has a view controller, which is a subclass of the UIViewController class, so XCode automatically creates a new view controller class for us and adds two of the required methods to this class. Both of these methods are inherited from UIViewController. In Swift we must explicitly state that we are overriding a method from the parent class, otherwise the compiler will generate and error. For this example we don't need to change any of these methods.

Adding Outlets and Actions to Interfaces

In our hello world application there are two UI elements: a button and a label. When the button is clicked, the label's text is changed. In order to support this functionality we need one outlet and one action.

Outlets are class properties in Swift and actions are class methods in Swift. We use a special syntax for adding both of them to our class. Add the following after the viewDidLoad method:

```
@IBOutlet var label : UILabel!;
@IBAction func didClickButton (sender : AnyObject) {
    self.label.text = "hello World"
}
```

Note you may get an error message at this point complaining about not having an initializer for the class, you can safely dismiss this error, or explore how initializers work in Swift.

Our outlet declaration is preceded by @IBOutlet, this tells XCode that we want to link this property to part of the user interface that we will design in the next section. All variable declarations start with the keyword var followed by the name of the variable, a : and then the type of the variable. If you assign an initial value to a variable you don't need to specify the type. The ! after the type indicates that this is an unwrapped optional; don't worry about this, we will talk about it next week.

Our action is called didClickButton and is called each time the user clicks the button. Again we use @IBAction at the start to tell XCode that we will link this action to the user interface. Action methods have a single parameter; the user interface object that called them. We use AnyObject as the type of this parameter. The implementation of this method is quite simple. The label variable references the label on the screen and it has a property called text that is the text string that it displays. All we need to do is set a new value for this property.

Constructing the UI

Now select the Main.storyboard file from the project view on the left side of the Xcode window. The Xcode window should now look something like figure 9. This is similar to the UI editor in Android, just with a slightly different arrangement of its components. The center is the view of the user interface that we are constructing. In the lower right is the list of widgets that we can add to the user interface, and in the upper right is an area where we can change the widget properties. On the left is a hierarchical view of the user interface.

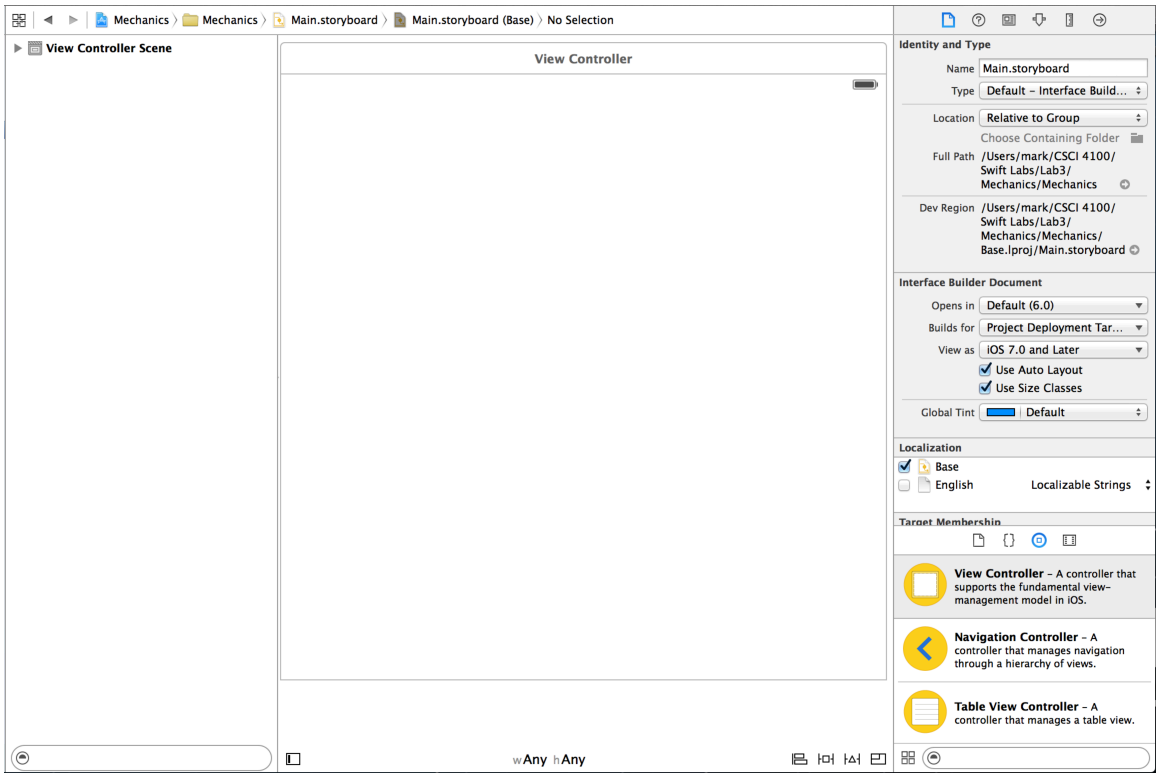

Figure 9 User interface designer

Open up the user interface by selection the arrow to the left of View Controller Scene and then select View Controller. Start designing the user interface by dragging a Label widget to the top left corner of the user interface. Above the properties window select the third icon from the right and the window should look like figure 10. Change the second line in the properties pane to "Click me →". I resized the widget so the text was clearer.

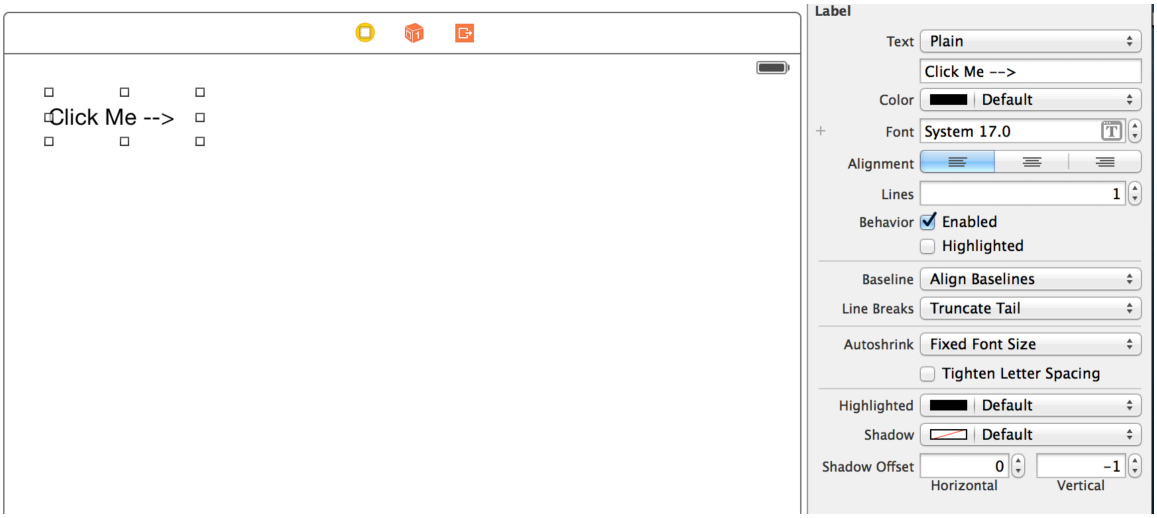

Figure 10 Label Widget Properties

Now drag a Button to the UI layout and change its Title property to “Click me!”. We have now finished the design of the user interface, now all we need to do is connect the widgets to the program code.

Now we need to connect the code in our view controller to the user interface elements that we have just added. Start by clicking the View Controller in the left pane and then in the top right pane select the rightmost tab. Figure 11 shows what you should see in this tab. In the Outlets section of this pane you will notice our label property that we added to our view controller. At the very bottom of the pane you will notice our didClickButton action. These are the items that we need to connect.

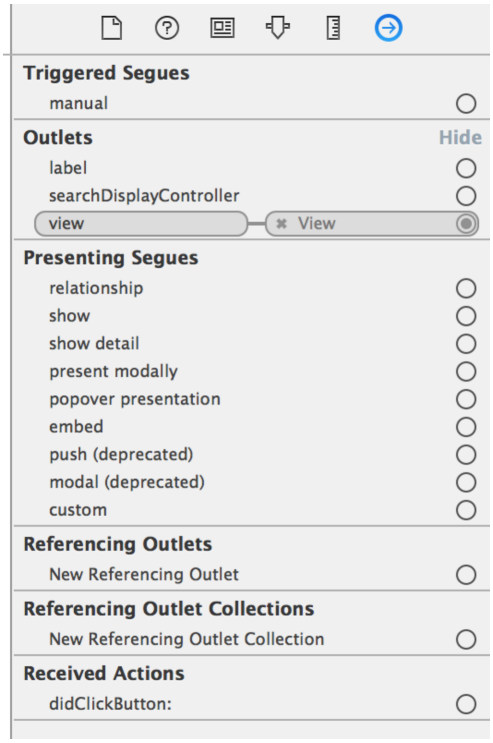

Figure 11 Outlets and actions tab

Put the mouse in the circle next to label, press the left button and drag over to the label widget. Once you are inside the label widget you can release the button. This performs the connection to the label. Similarly drag from the didCLickButton circle to the button widget. In this case when you release the mouse button a pop up menu will appear with a selection of events. From this menu select Touch Up Inside. When you are finished the outlets pane should look like figure 12.

Now that we have connected everything up we can run our application. It does look a bit boring and not quite what is presented in figure 3. We can fix this by adding a navigation controller that will give us a navigation bar. Start by moving the label and button down a bit so we have room for the navigation bar. Then make sure that the view controller is selected and the go to the Editor menu, select Embed In and then select Navigation Controller. Double click the navigation bar that appears at the top of the window and

enter “Hello World”. This can be done in the navigation bar, or the prompt property in the property view. Now run your application and see what it looks like.

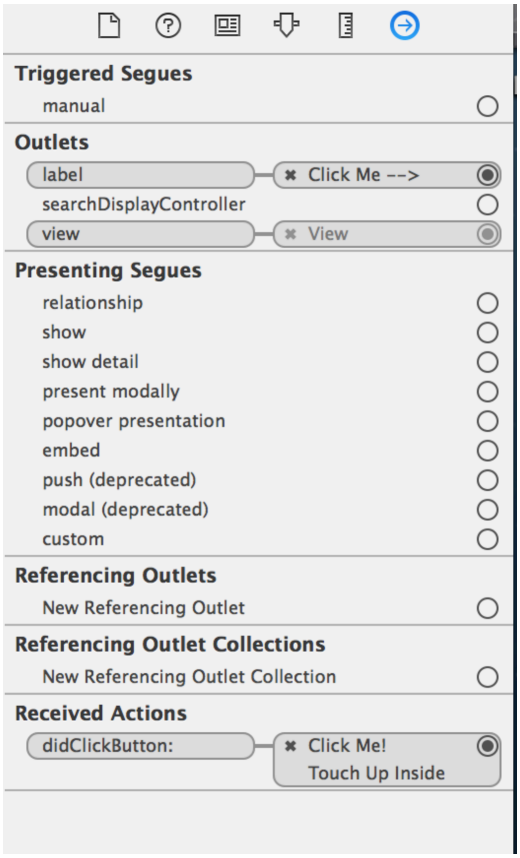

Figure 12 Completed outlets and actions tab.

This completes the application; try running it a see what happens.

Laboratory Report

For this laboratory add another label and button to the application and have it do something interesting. If you are adventurous you can consider using other widgets. Show the resulting application and program code to the TA.