Programare dispozitive iOS

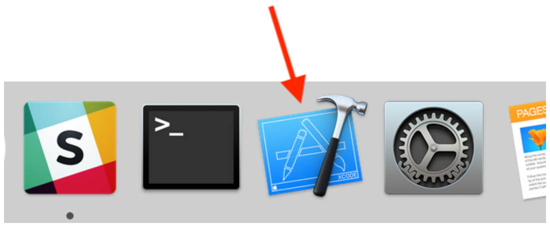
# iOS Lab - Lesson 1

## Lesson goals:

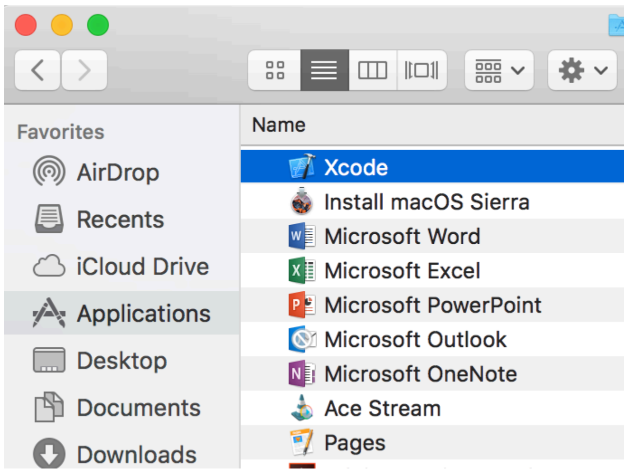
- Get familiar with Xcode and the Simulator;
- Introduction to storyboards and interface builder;
- Create and customize a few basic views;
- Create a basic navigation between two different screens;
- Understand some basic concepts of the MVC architecture.

## 1. Introduction to Xcode

Let’s get started by opening up **Xcode**. It should be in the dock:



If it’s not in the dock, open a **Finder** window, go to **Applications** and you should be able to find **Xcode** somewhere in the applications list.

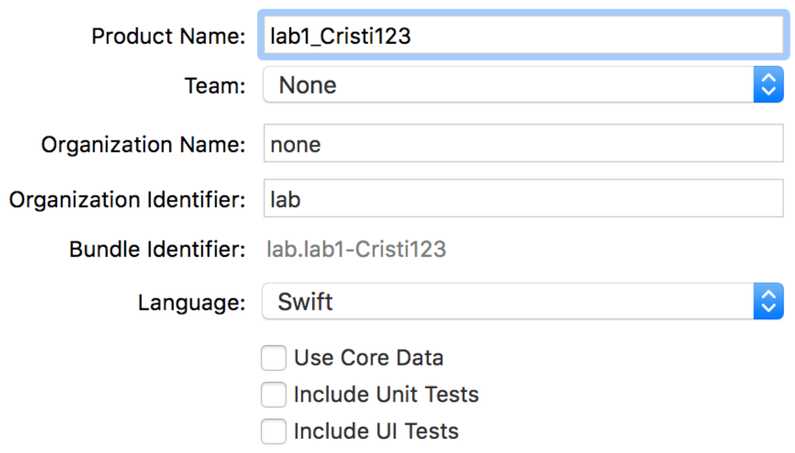


Go ahead and tap **Create a new Xcode project**. Select **Single View App** and press **Next**.

For the project options:

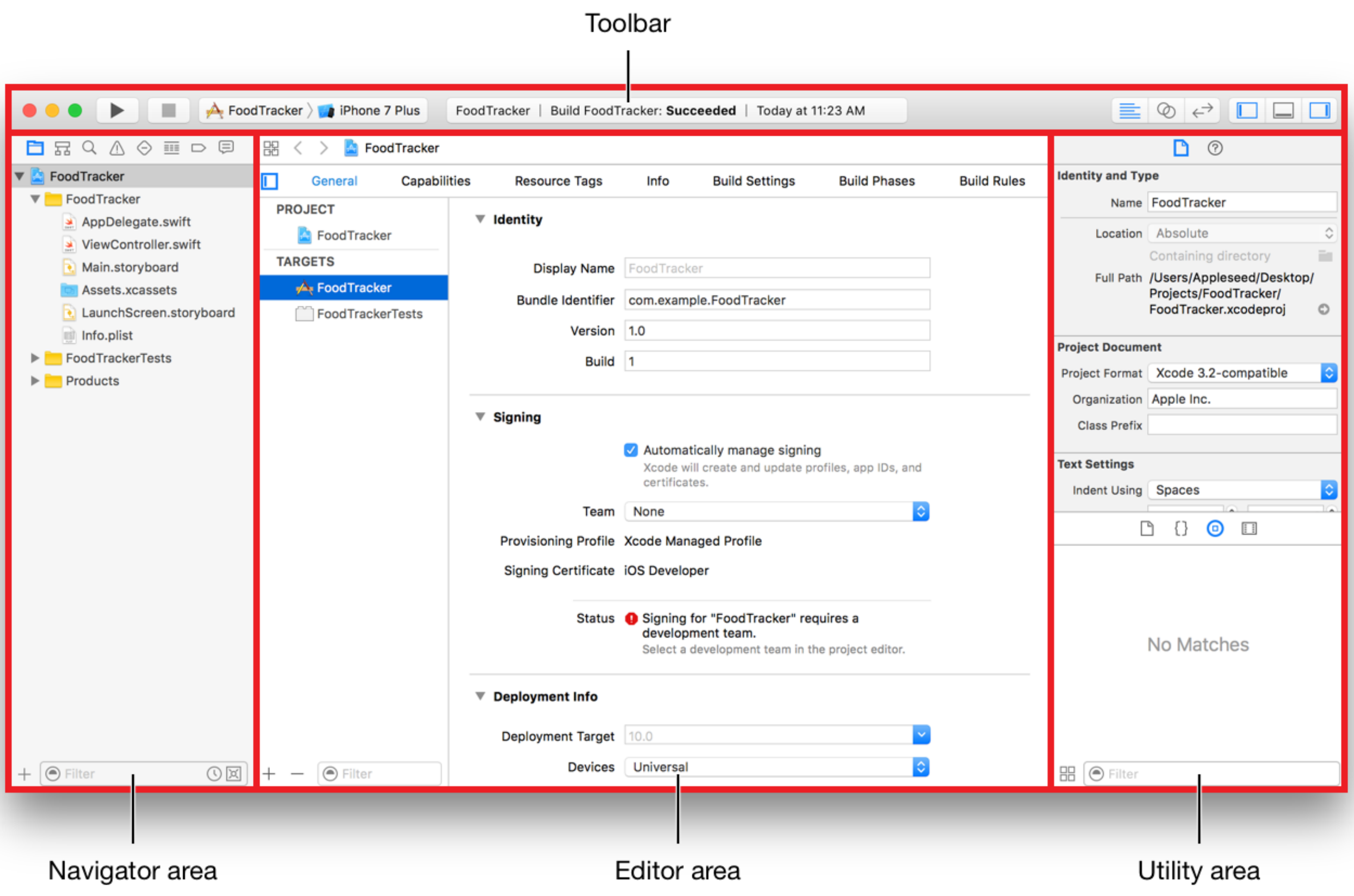
- **Product name:** Lab1_*your_name* (Or something that wouldn't be already taken)
- **Language:** Swift
- Make sure the last 3 checkboxes are unchecked
- The other settings don't really matter

Here are my options:



Product Name: lab1_Cristi123
Team: None
Organization Name: none
Organization Identifier: lab
Bundle Identifier: lab.lab1-Cristi123
Language: Swift
☐ Use Core Data
☐ Include Unit Tests
☐ Include UI Tests

Press **Next**, select the **Desktop** as a save location and press **Create**.
Here's what's in front of you:



FoodTracker 〉 iPhone 7 Plus | FoodTracker | Build FoodTracker: **Succeeded** | Today at 11:23 AM

FoodTracker
FoodTracker
AppDelegate.swift
ViewController.swift
Main.storyboard
Assets.xcassets
LaunchScreen.storyboard
Info.plist
FoodTrackerTests
Products

General | Capabilities | Resource Tags | Info | Build Settings | Build Phases | Build Rules

PROJECT
FoodTracker
TARGETS
FoodTracker
FoodTrackerTests

Identity
Display Name: FoodTracker
Bundle Identifier: com.example.FoodTracker
Version: 1.0
Build: 1

Signing
☑ Automatically manage signing
Xcode will create and update profiles, app IDs, and certificates.
Team: None
Provisioning Profile: Xcode Managed Profile
Signing Certificate: iOS Developer
Status: Signing for "FoodTracker" requires a development team.
Select a development team in the project editor.

Deployment Info
Deployment Target: 10.0
Devices: Universal

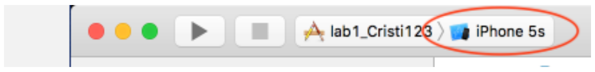
Identity and Type
Name: FoodTracker
Location: Absolute
Containing directory
Full Path: /Users/Appleseed/Desktop/Projects/FoodTracker/FoodTracker.xcodeproj

Project Document
Project Format: Xcode 3.2-compatible
Organization: Apple Inc.
Class Prefix:

Text Settings
Indent Using: Spaces

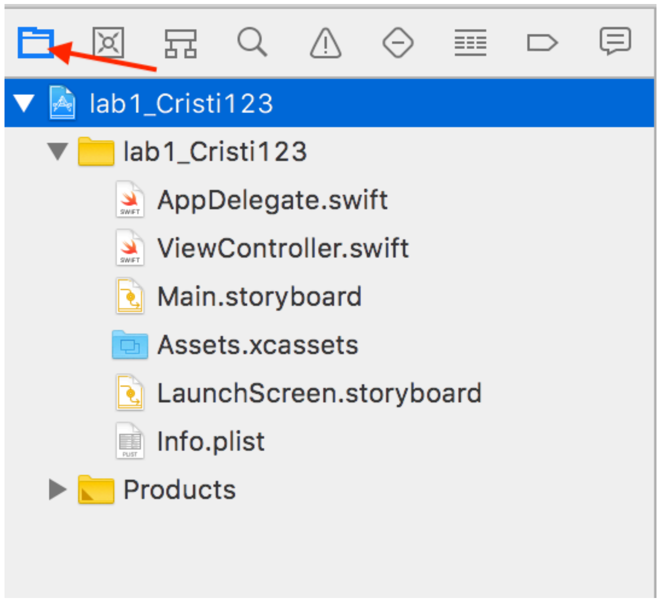
No Matches

Let’s run the app for the first time. First off, in the left side of the Toolbar, select the iPhone you want to run your app on. Let’s select an **iPhone X** or an **iPhone XS**.



Press the **Run** button. It will take a few moments.

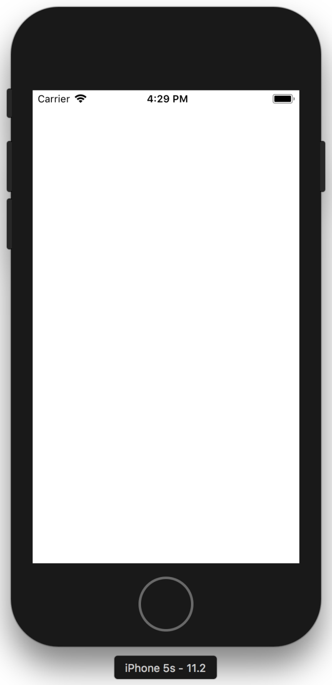
Until the Simulator opens up, let’s take a look at what is already in our project. In the left of Xcode’s window is the **Navigation area**. Make sure the first tab is selected.



That’s the **Project navigator**. It’s where you can see all the files included in your project. You can see the files that Xcode added for us when we created a new project.

- The **AppDelegate.swift** file is where we respond to app-wide events like the app being launched or the app entering background. We won’t be editing this file today.
- The **ViewController.swift** file is our app first screen. We’ll be customizing this file shortly.
- The **Main.storyboard** file is where you design all your app’s UI. We’ll open this file and check it out next.
- The **Assets.xcassets** folder is a good place to store and organize all your images used in the UI.

At this point, if your simulator finished starting up, you should be able to see your app:



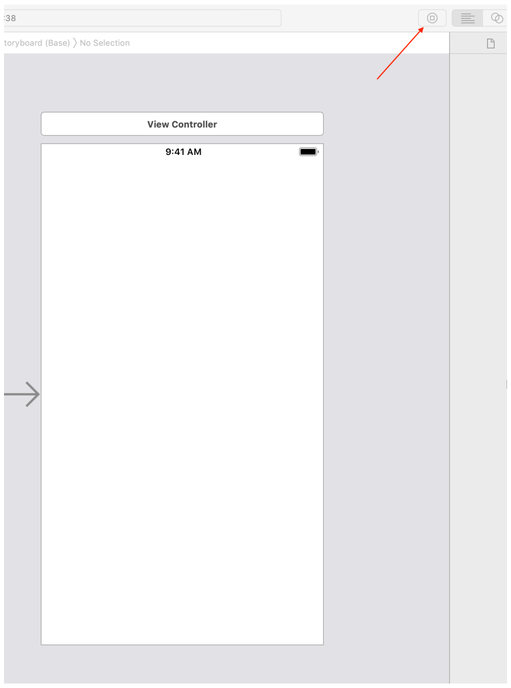
That’s it. A white screen. Good job!

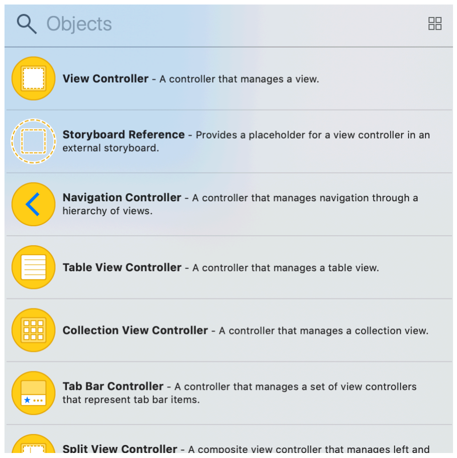
## 2. Views and Constraints

Let’s go back to Xcode and open up the **Main.storyboard** file.

You’ll see an empty, white screen. That’s the screen you saw when you ran the app on the simulator. You can see a grey arrow pointing to the screen. That tells you it’s you **Initial View Controller**. You can think of that screen as the entry point for your app. When you’ll start adding multiple screens, it’s going to be important to know what’s the first screen the user sees.

In the right side of the window, in the top corner you’ll be able to see the **Object library**.





**View Controller** - A controller that manages a view.

**Storyboard Reference** - Provides a placeholder for a view controller in an external storyboard.

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

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

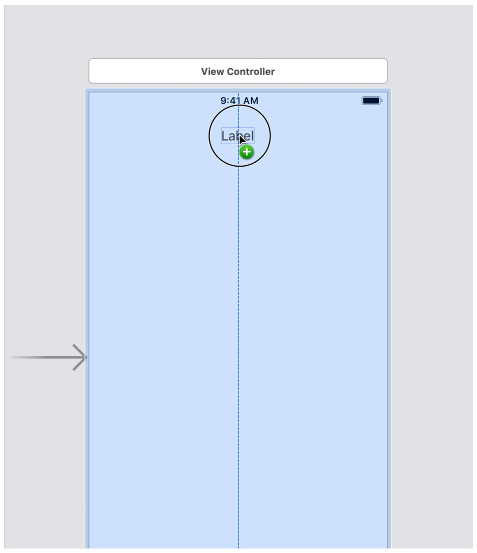
**Collection View Controller** - A controller that manages a collection view.

**Tab Bar Controller** - A controller that manages a set of view controllers that represent tab bar items.

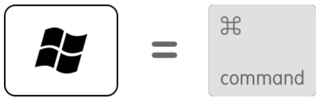
**Split View Controller** - A composite view controller that manages left and

In the Object library you can see all the UI controls (and other stuff) that you can add to you app.

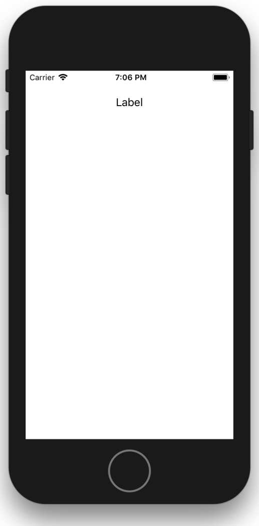
Try to find the **Label**. Drag it into your empty screen. Try and place it centered, near the top.

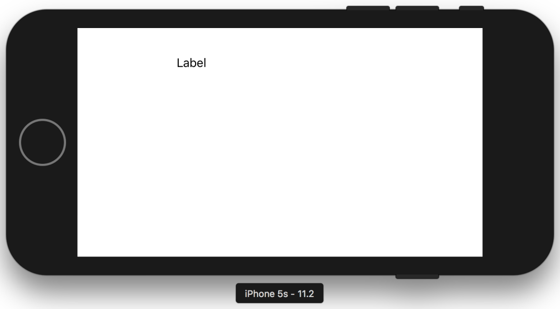


Let's run the app again and see our label. It seems like it looks fine, but let's see what happens when we rotate the screen. Press ⌘ + → on your keyboard. If you're using a Windows keyboard, that's the ***Windows symbol*** *key* and the ***right arrow*** key.



That shortcut should have rotated the screen of the emulator in landscape.





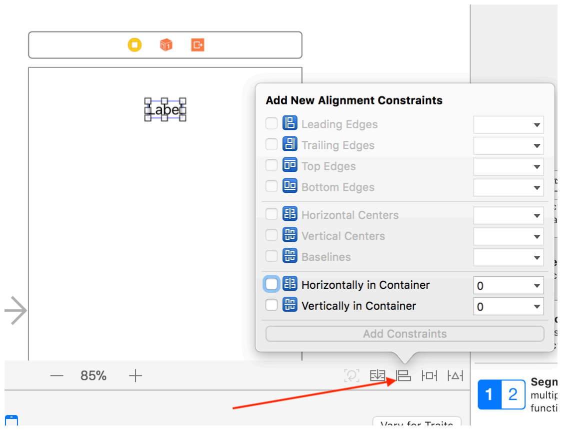
As you can see, we have an issue. When rotating the screen, the label doesn't stay in the center of the screen. That's because we only positioned it in the center. We never told Xcode it should always stay centered.

This isn't only the case when rotating the screen. It's also important when developing for different screen sizes (both iPhone 8 Plus and iPad have a larger screen then the iPhone 5)

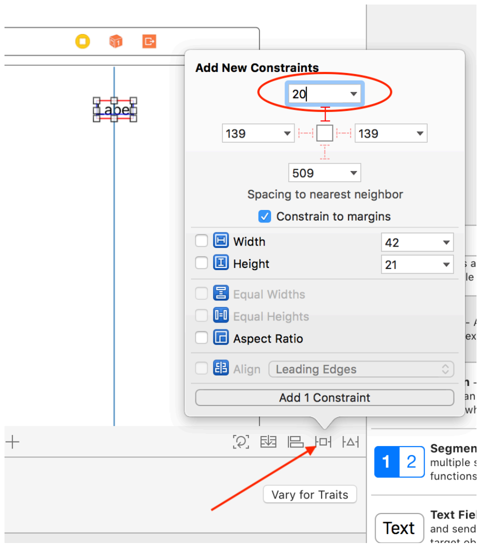
Let's go back to he storyboard. We need to tell Xcode 2 things:

- the label needs to be centered horizontally
- the label needs to be placed 20 points below the status bar.

Select the label. Press the button shown in the image. Press the checkbox next to **Horizontally in Container**. Press **Add 1 Constraint**.



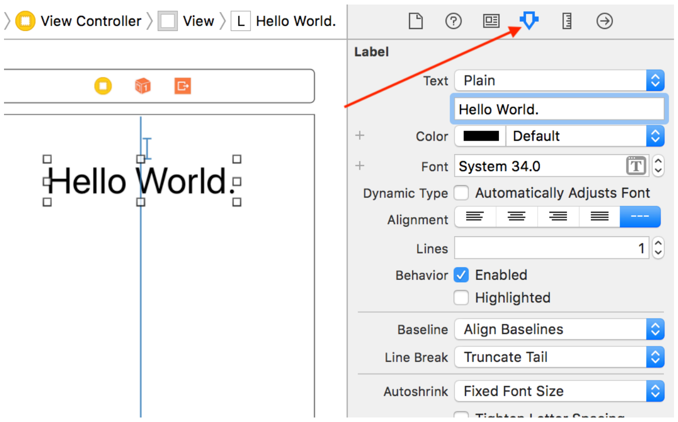
Next, with the label still selected, press the button shown in the image. Enter the value **20** for the top spacing of the label, below the status bar.



That's it. Xcode knows to set the height and width of label based on it's content.

Go ahead and make sure the label is still selected and go to the **Attributes Inspector** on the right side of the screen (see picture below).

In the Attributes Inspector you can change the appearance of our label. Change the text of the label (I went with the classic "Hello World"). And change the font size to something



View Controller 〉 View 〉 Hello World.

Label

Text: Plain

Hello World.

Color: Default

Font: System 34.0

Dynamic Type: Automatically Adjusts Font

Alignment

Lines: 1

Behavior: ✓ Enabled

Highlighted

Baseline: Align Baselines

Line Break: Truncate Tail

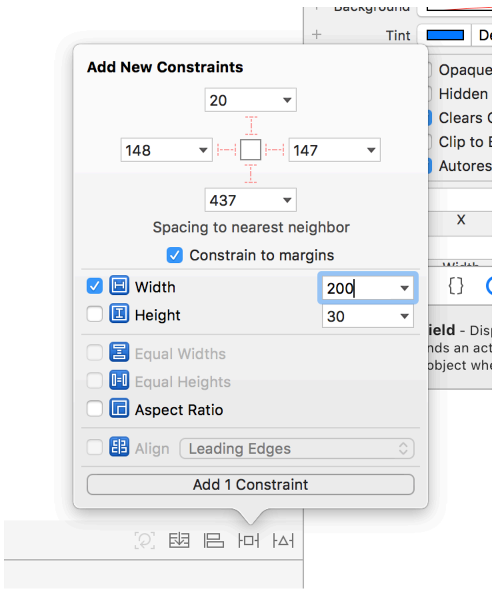
Autoshrink: Fixed Font Size

bigger. You'll see the label's frame will get bigger, but it will stay centered and the top spacing will remain the same.

Go ahead and run the app again, rotate the device. You'll see the label stays centered even on landscape mode.

## 3. Outlets and Actions

Add a **Text Field** underneath the label by dragging one from the bottom right.
Add a centered **horizontally in container** constraint like you already did for the label.
Add a top distance of **20** points below the label like you already did for the label.
The text field also needs a fixed width constraint so Xcode knows how wide it should be. Make sure the textfield is selected and add a fixed width constraint of 200. Don't forget to press **Add 1 Constraint**.



Add New Constraints

20

148 147

437

Spacing to nearest neighbor

[✓] Constrain to margins

[✓] Width 200

[ ] Height 30

[ ] Equal Widths

[ ] Equal Heights

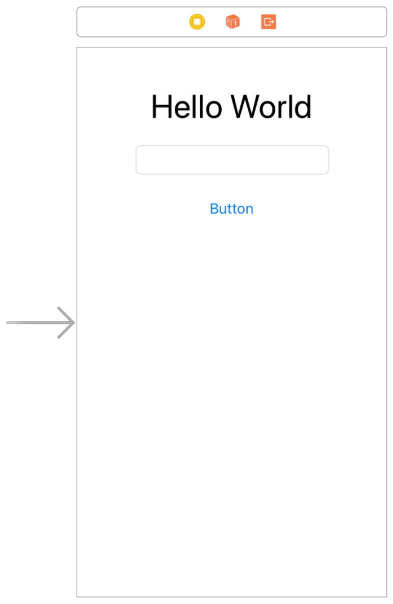
[ ] Aspect Ratio

[ ] Align Leading Edges

Add 1 Constraint

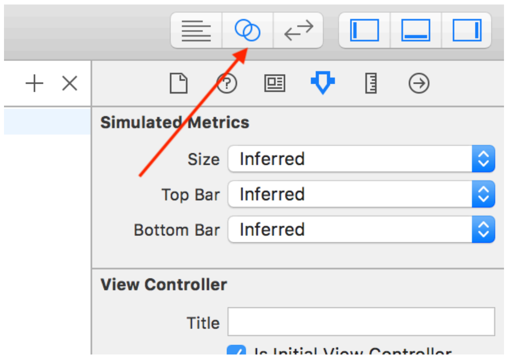
Add a button underneath the textfield.
Make it centered and add a spacing of 20 between it and the text field.

Your screen should look similar to this:

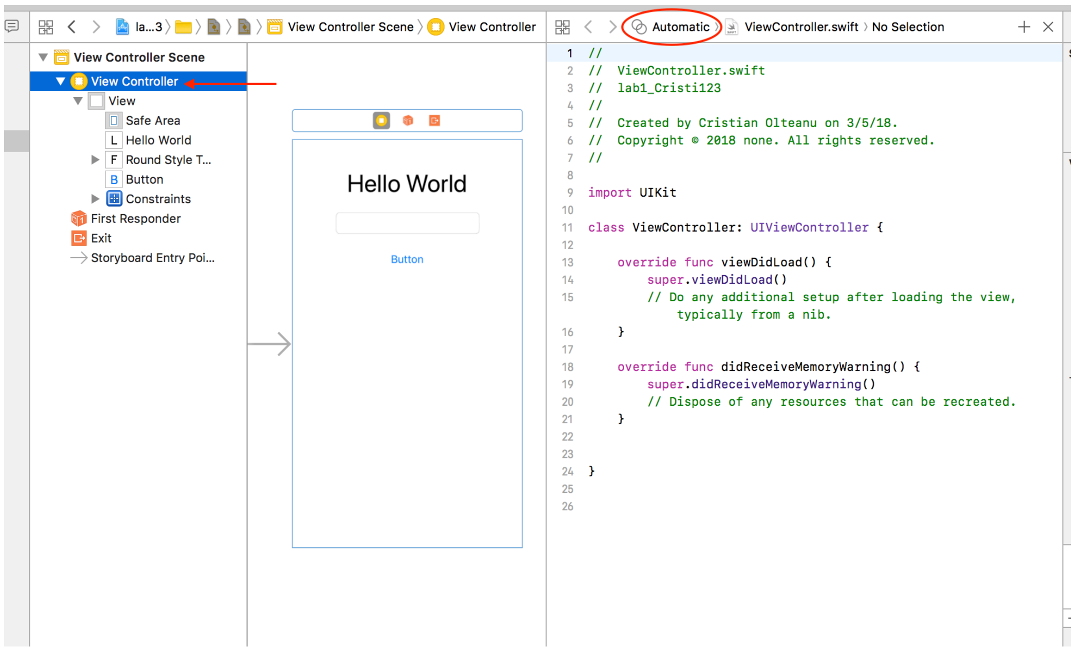


Run the app!
Try pressing the button. Nothing happens.
Let's show a alert when you press the button.

Back to the storyboard. On the top right of the window press the **Show the Assistant Editor** button (see image below).

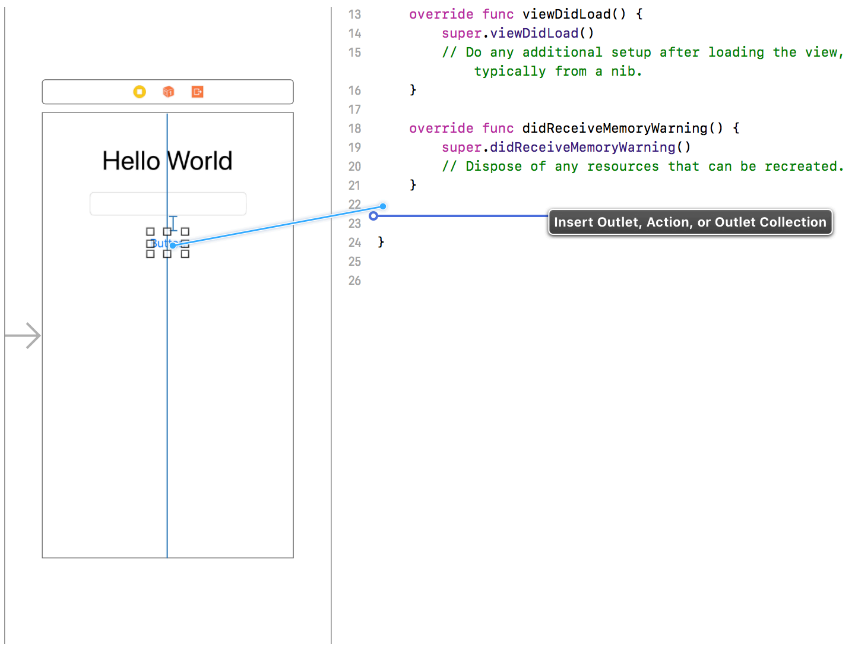


You should see the storyboard on the left and some code on the right.
First, make sure you select your view controller in the storyboard (see arrow below) and then make sure **Automatic** is selected in the right side editor (see red circle below)



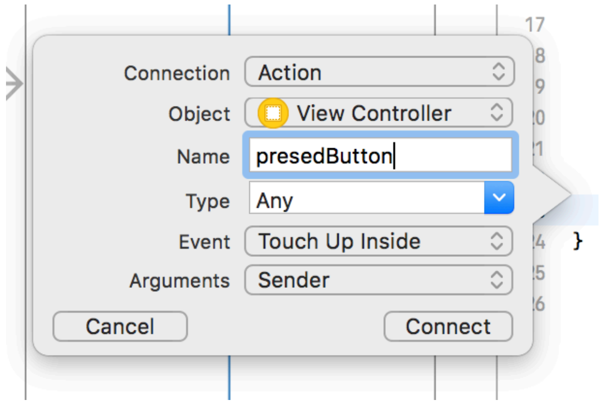
```
//
//  ViewController.swift
//  lab1_Cristi123
//
//  Created by Cristian Olteanu on 3/5/18.
//  Copyright © 2018 none. All rights reserved.
//

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.
    }


}
```

Next, hold down **control** on the keyboard, then click and drag the button from the storyboard to the code (see picture below, here’s a video if you’re having some trouble)



```
    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.
    }


}
```

When you finish dragging the button, you should see a pop-up like this:



| | |
|---|---|
| Connection | Action |
| Object | View Controller |
| Name | presedButton |
| Type | Any |
| Event | Touch Up Inside |
| Arguments | Sender |

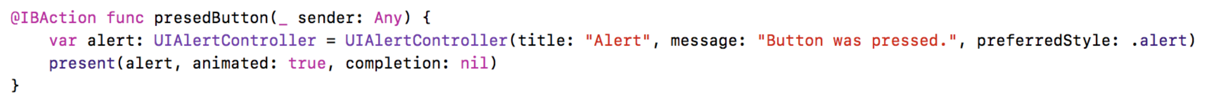
Make sure the connection type is **Action**, then name the action **pressedButton**. Press **Connect**.

Xcode just created a function for us that will get called every time you press that button. Let’s add some code to that function.

Let’s create an alert with a title and a message and show the alert when someone presses the button.

Here’s the code for that:

( when using a windows keyboard, the shortcut for copy is ***windows key*** + ***C***. Paste is ***windows key*** + ***V*** )

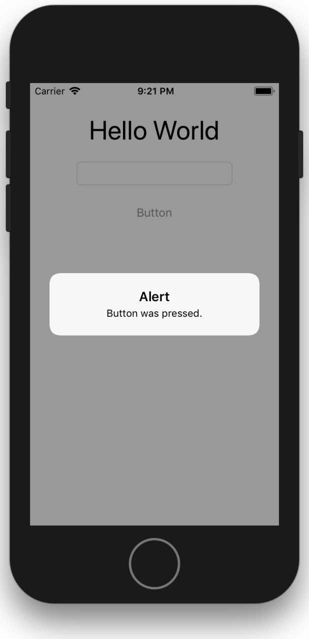


```
@IBAction func presedButton(_ sender: Any) {
    var alert: UIAlertController = UIAlertController(title: "Alert", message: "Button was pressed.", preferredStyle: .alert)
    present(alert, animated: true, completion: nil)
}
```

Update your function so it looks like this. Try to understand what you’re typing. Here are some notes:

- ***var*** is the keyword used to declare a variable. After it, is the variable’s name and the variable type.
- after declaring the variable, we initialize it using a constructor from the **UIAlertController** class.
- once the alert is instantiated, we need to present it.

Run the app and press the button.
There it is! The only issue is that we can’t close the alert.



Let’s update our function.
Here’s the code:
(First line is the same. Then, I created a new button on the second line and added the button to the alert on the third line.)
Run the app and see if it works!

```
@IBAction func presedButton(_ sender: Any) {
    var alert: UIAlertController = UIAlertController(title: "Alert", message: "Button was pressed.", preferredStyle: .alert)
    var cancelAction: UIAlertAction = UIAlertAction(title: "Cancel", style: .cancel, handler: nil)
    alert.addAction(cancelAction)
    present(alert, animated: true, completion: nil)
}
```

You might be wondering about those warnings you're seeing on your screen (The yellow highlighting). Xcode is trying to tell you that those two variables' values never change, so they should be declared as **constants**. To silence those warning change the **var** keyword to **let**.

Next, let's only show the alert if the text field is empty.

To do that we need to reference the text field from our code. We need to add an outlet to do the text field.

Similar to how you added an action to the button, we can add an outlet to the label. Hold down control on the keyboard, click and drag from the label to your code. Connection type is **outlet**. Name should be **textField** (here's a video if you can't figure it out).

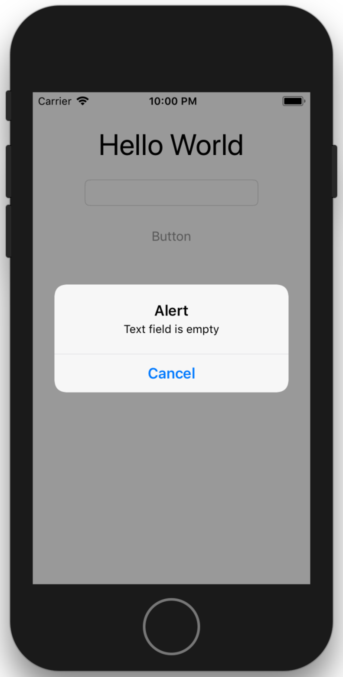
Now we can check if the text field is empty.

In your **pressedButton** function, add an if block to check if the textField's text is empty. If it is, then it should execute our 4 lines.

Here's the updated code:

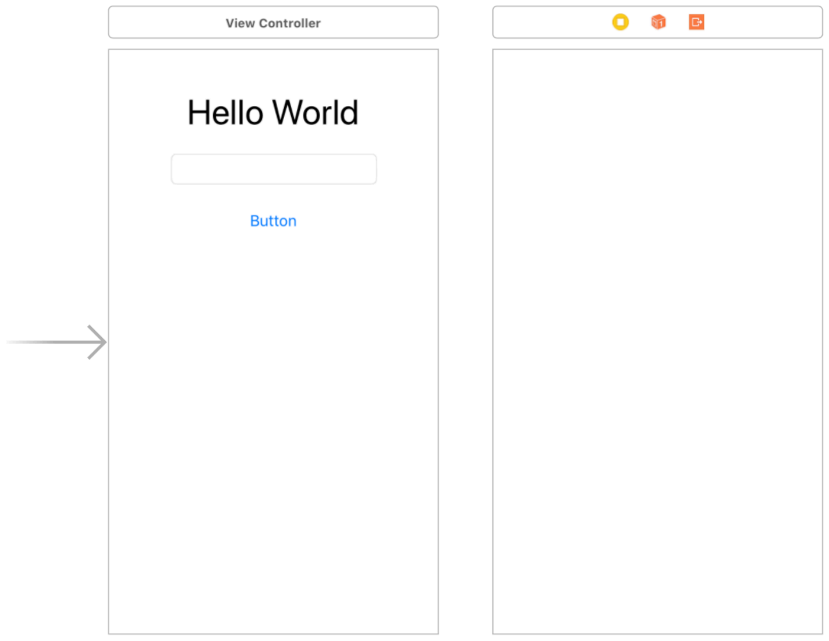
```
@IBAction func presedButton(_ sender: Any) {
    if textField.text!.isEmpty {
        let alert: UIAlertController = UIAlertController(title: "Alert", message: "Text field is empty", preferredStyle: .alert)
        let cancelAction: UIAlertAction = UIAlertAction(title: "Cancel", style: .cancel, handler: nil)
        alert.addAction(cancelAction)
        present(alert, animated: true, completion: nil)
    }
}
```

Don't worry about the **exclamation mark** we added in the if-condition for now. We'll get to that in another lesson.

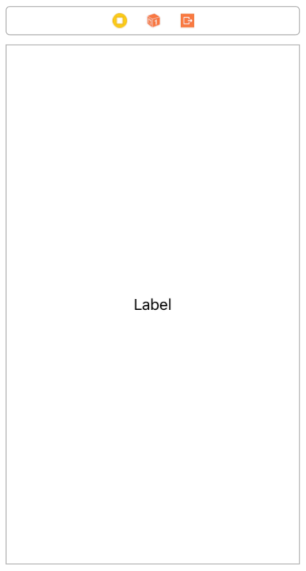


# 4. Navigation and View Controller lifecycle

Go back to the storyboard and add a new **View Controller** next to the old one. Like this:

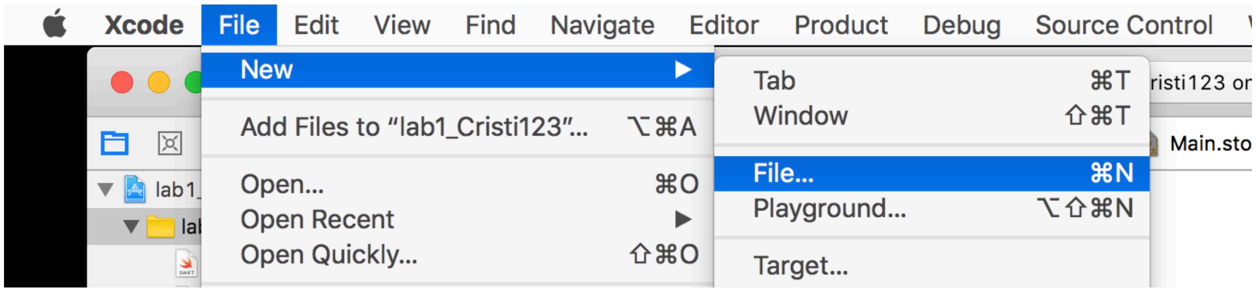


Add a new label to our second view controller, center it both vertically and horizontally this time.



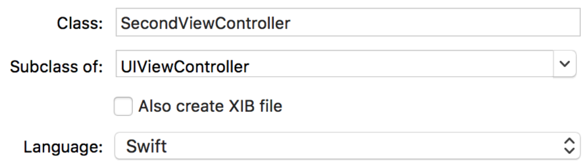
We'll need a new class for this new screen.

From the top bar in Xcode, go to **File** / **New** / **File…**



Select **Cocoa Touch Class** and press **Next**.

First, let's make sure we subclass **UIViewController** in the **Subclass of:** text field, then name it **SecondViewController**. Also, make sure the Language is set to **Swift**.



Class: SecondViewController

Subclass of: UIViewController

Also create XIB file

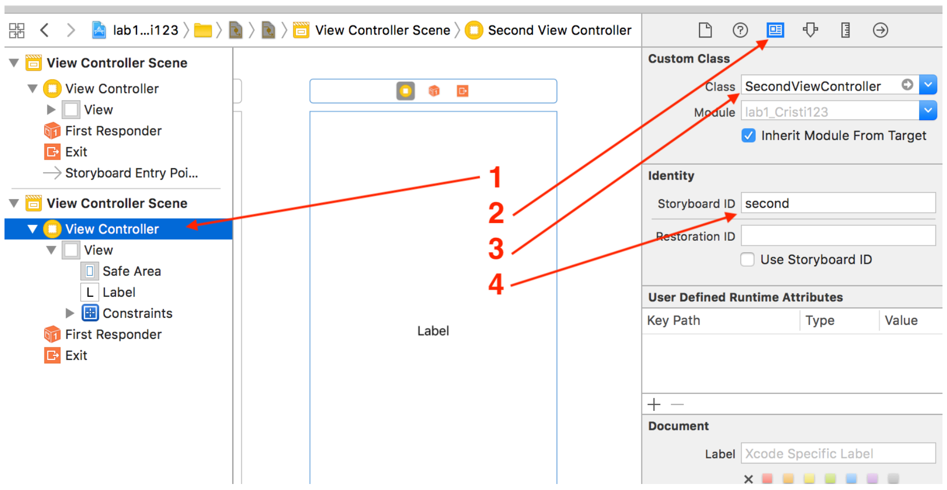
Language: Swift

Press **Next** and then **Create**.

Next, we need to tell Xcode that the view controller we added in the storyboard needs to be of type **SecondViewController**.

Go back to the storyboard.

1) Select the view controller
2) Press **Show the Identity Inspector** tab
3) Set **Class** to **SecondViewController**
4) Set **Storyboard ID** to **second**



Class SecondViewController

Module lab1_Cristi123

☑ Inherit Module From Target

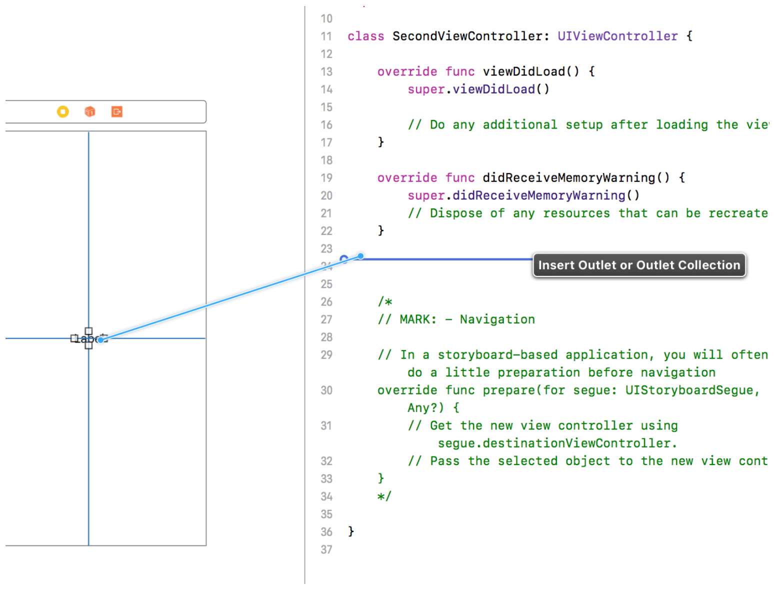
Storyboard ID second

Restoration ID

☐ Use Storyboard ID

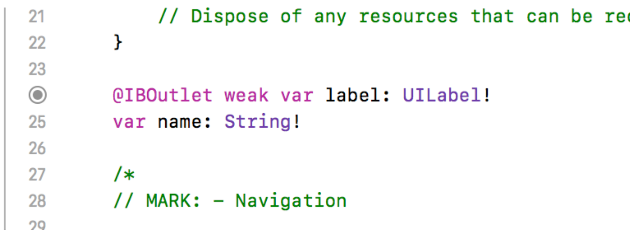
Label Xcode Specific Label

Create an outlet for the new label in the SecondViewController. Name it **label**.



```
class SecondViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()

        // Do any additional setup after loading the vie
    }

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


    /*
    // MARK: - Navigation

    // In a storyboard-based application, you will often
        do a little preparation before navigation
    override func prepare(for segue: UIStoryboardSegue,
        Any?) {
        // Get the new view controller using
            segue.destinationViewController.
        // Pass the selected object to the new view cont
    }
    */

}
```

We want the label to say “Hello ” + ***whatever you typed in the text field.***
We need to store that was typed in the text field in a variable.
Let’s create a new variable in the SecondViewControlled. Call it **name**. Here’s how you write that: ( line 25 in the picture below )



```
21          // Dispose of any resources that can be re
22      }
23
        @IBOutlet weak var label: UILabel!
25      var name: String!
26
27      /*
28      // MARK: - Navigation
29
```

Make sure you actually added it to the SecondViewController class.

Next, go back to the **ViewControler.swift** file. Let’s edit our **pressedButton** function.
We want to show the Second View Controller only if the textField is not empty.
Add an else block to our if condition. There, we need to instantiate a new SecondViewController object, set it’s name variable and present it. Here’s how we can do that: (only added the else block, the other parts should be identical)

```
@IBAction func presedButton(_ sender: Any) {
    if textField.text!.isEmpty {
        let alert: UIAlertController = UIAlertController(title: "Alert", message: "Text field is empty", preferredStyle: .alert)
        let cancelAction: UIAlertAction = UIAlertAction(title: "Cancel", style: .cancel, handler: nil)
        alert.addAction(cancelAction)
        present(alert, animated: true, completion: nil)
    } else {
        let second = storyboard?.instantiateViewController(withIdentifier: "second") as! SecondViewController
        second.name = textField.text
        present(second, animated: true, completion: nil)
    }
}
```

Here’s what we did:

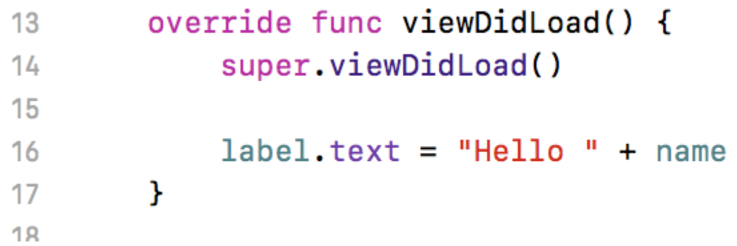
We created the **SecondViewController** based on the screen from the **storyboard** that has the **Storyboard ID** equal to “second”.

Them we tell that view controller that it’s name should be the text from the text field.

Last thing, we present the view controller.

You can try and run the app. You’ll see that the label from the second screen doesn’t change it’s text. That’s because we never coded that part.

Let’s go to the SecondViewController.swift file. In the **viewDidLoad()** function add this line of code:

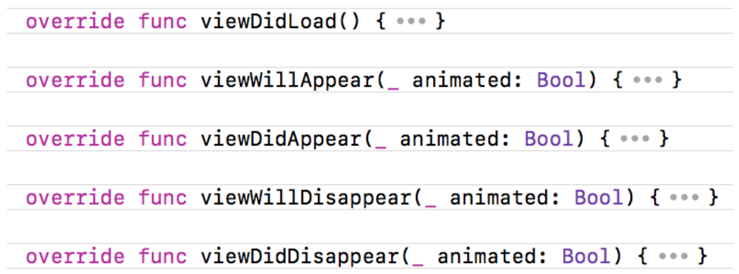


```swift
    override func viewDidLoad() {
        super.viewDidLoad()

        label.text = "Hello " + name
    }
```

Now you can run the app and see the label’s text change.

The **viewDidLoad()** function is automatically called when the screen gets created. It’s a good place to configure the way the screen looks if you need to do that in code.

There are other such functions, here is the order they get called:



```swift
override func viewDidLoad() { ••• }

override func viewWillAppear(_ animated: Bool) { ••• }

override func viewDidAppear(_ animated: Bool) { ••• }

override func viewWillDisappear(_ animated: Bool) { ••• }

override func viewDidDisappear(_ animated: Bool) { ••• }
```

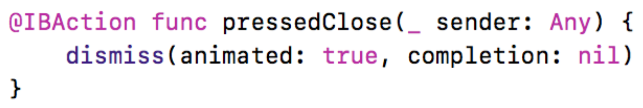
One last thing for the SecondViewController: there’s no way to close the screen.

Add a new button below the label, keep it centered and just below the label.

Change the text of the button to say **Close**

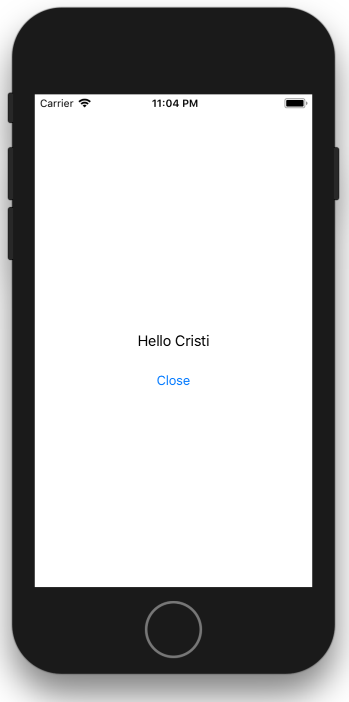
Add an **action** to that button, name it **pressedClose**.

Here’s the code to hide this screen:



```swift
@IBAction func pressedClose(_ sender: Any) {
    dismiss(animated: true, completion: nil)
}
```

Run the app and test the button.



## 5. Model-View-Controller (MVC)

You might of heard about MVC before. It's an architectural pattern that separates these 3 major parts of your app in an attempt to make your code easier to understand and work with.

Apple uses this design pattern in their API, so it's important to have a basic understanding of what it is.

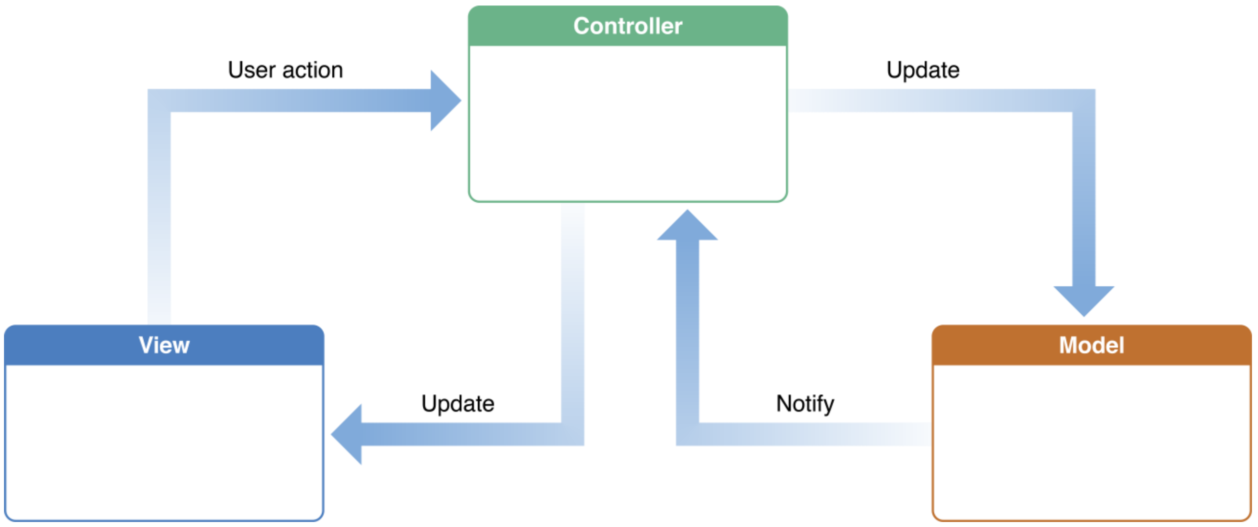
Here's a brief explanation of what those 3 components should do:

- A **model** represents your data. Either what should be displayed on the screen or what gets saved in memory. In our app, in the **SecondViewController**, we had a variable called **name**. That's a basic example of a model object. We used that variable to store what should be displayed on screen

- A **view** is any UI element that gets displayed on the screen. In our app we used labels, a text field and a button. There are more complicated views that we'll learn about in future lessons.

- A **controller** is the component that contains all your logic in the app. It links your models and the views together.

This is how Apple uses this pattern:



Here's a few examples from our app:

- The **view** lets the **controller** know when the user interacts with the app.
  -> The **pressedButton()** gets called when a user presses the button. The controller then does some logic (checks the text field's text) and makes a decision of what it should do next.

- The **controller** updates the **view**
  -> The **SecondViewController** sets the text for the label with the desired text.

- The **controller** updates the **model**
  -> Our first **ViewController** sets the value of the **name** variable based on it's defined logic.

- The **model** notifies the **controller** when it changes
  -> We didn't get to this part in this lesson. Imagine if the **name** variable changed it's value. We would need to let the controller know when this happened so that the controller knows it should update the view.