

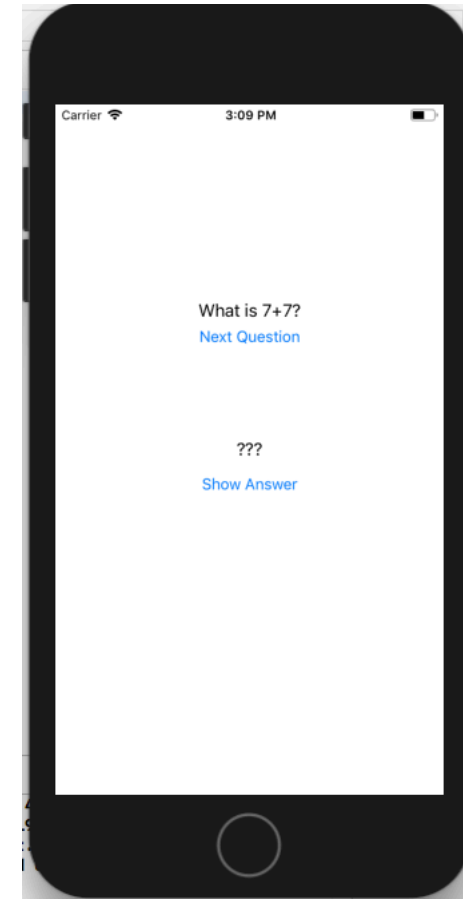


iOS: First Look at an App

SNEAK PEEK

Overview

- ▶ Create a simple iOS app
- ▶ Learn the basics of setting up a project
- ▶ Have a first look at the Swift code required to drive an iOS app



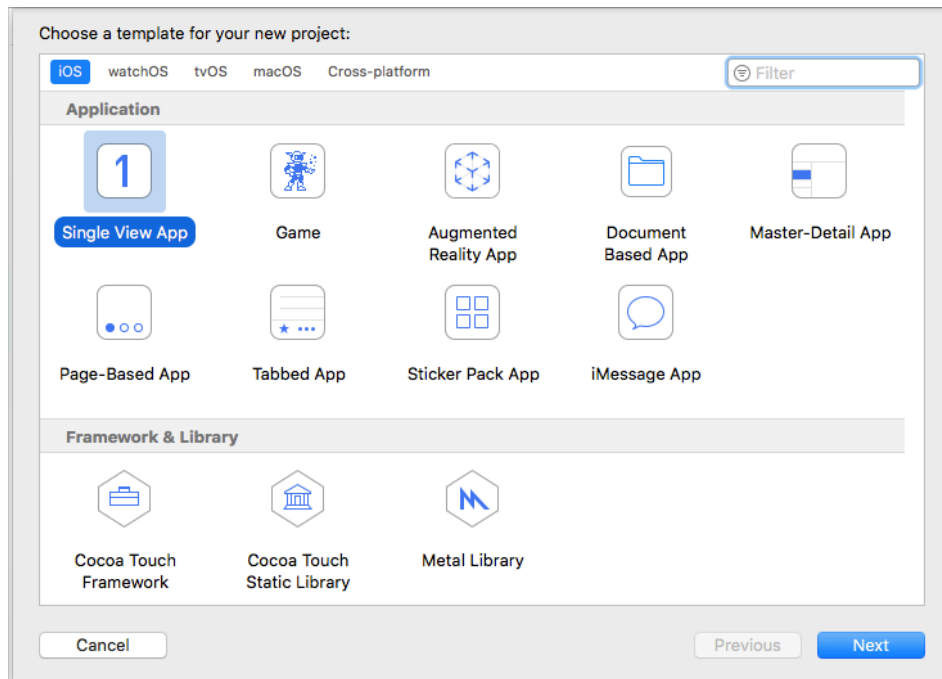
Quiz Project

This is Chapter One in the Big Nerd Ranch Guide iOS book

It's a good way to start working with Xcode and iOS apps

Create an Xcode Project

- ▶ From the File menu in Xcode, do New -> Project
- ▶ Choose an iOS project, and choose Single View App



Creating a New Project

- ▶ Fill in information for your new project
- ▶ You can set Team to “None” or else put in your Apple ID
- ▶ You can use your netid for the Organization Name
- ▶ The Organization Identifier is a way that will let Xcode create a unique Bundle Identifier for your project
- ▶ I like to use com.example.<netid>
- ▶ **Change User Interface to “Storyboard”**
- ▶ The other fields (Language, etc.) will have the defaults we want to use

Choose options for your new project:

Product Name: Quiz

Team: jason hibbeler (Personal Team)

Organization Name: jhibbele

Organization Identifier: com.example.jhibbele

Bundle Identifier: com.example.jhibbele.Quiz

Language: Swift

User Interface: Storyboard

☐ Use Core Data

☐ Use CloudKit

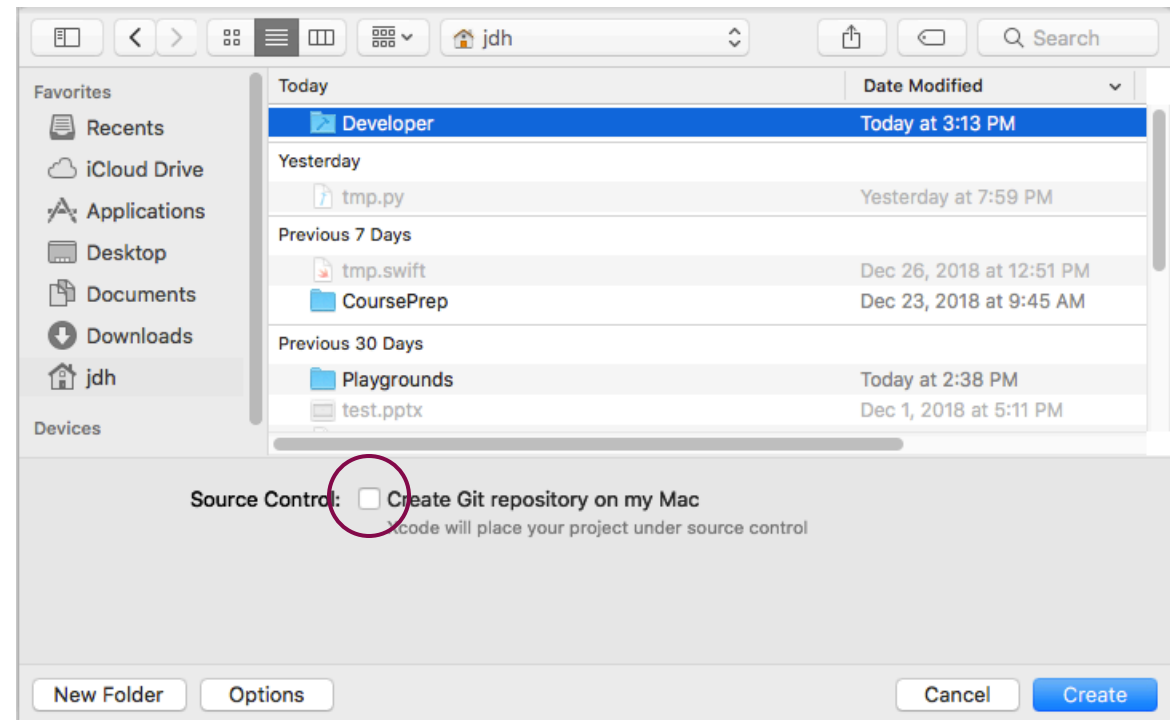
☒ Include Unit Tests

☒ Include UI Tests

Cancel Previous Next

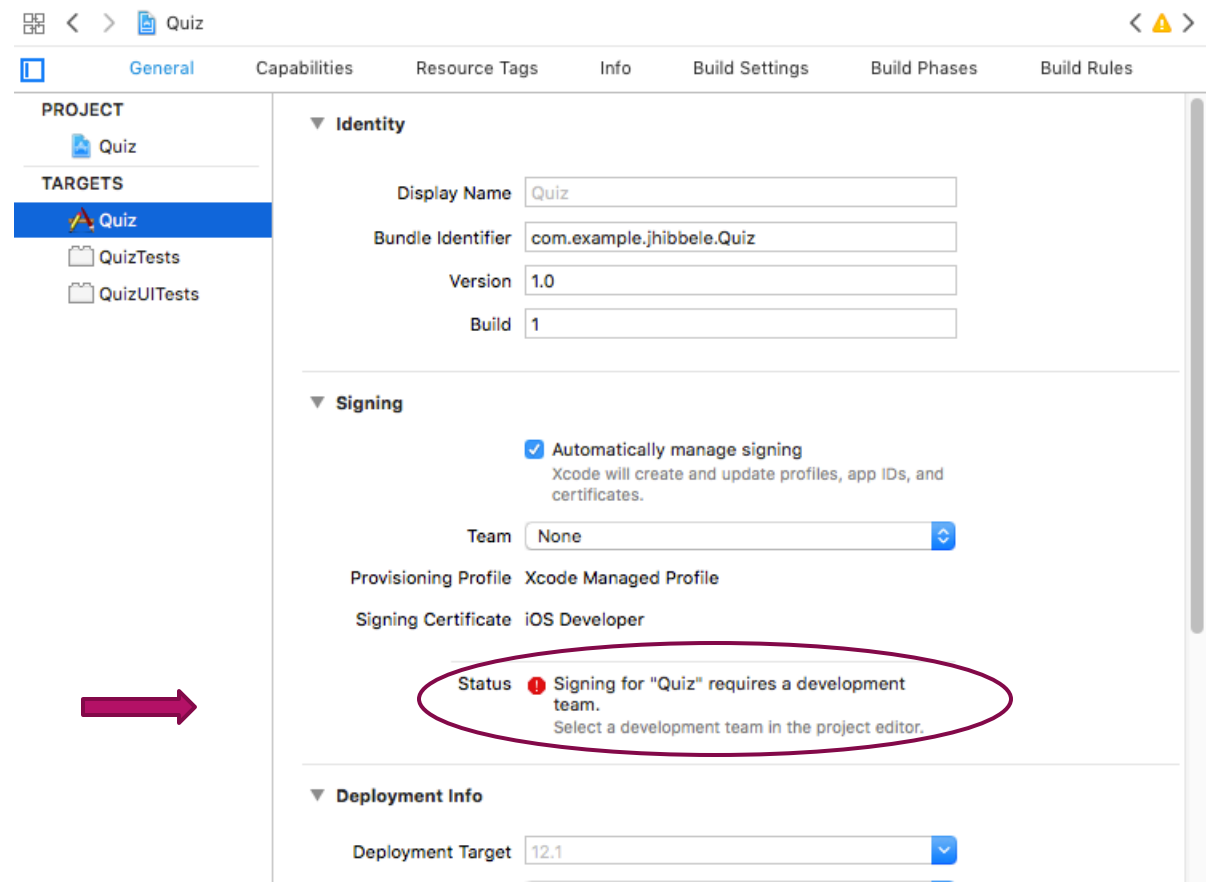
Creating a New Project

- ▶ By convention, Apple developers put their projects in a folder named Developer
- ▶ If you don't already have a Developer folder, create one in your home directory
- ▶ Then pick that folder as the place to save your new project
- ▶ Un-check the source control box; version control isn't important for this project but will be for your group project



Creating a New Project

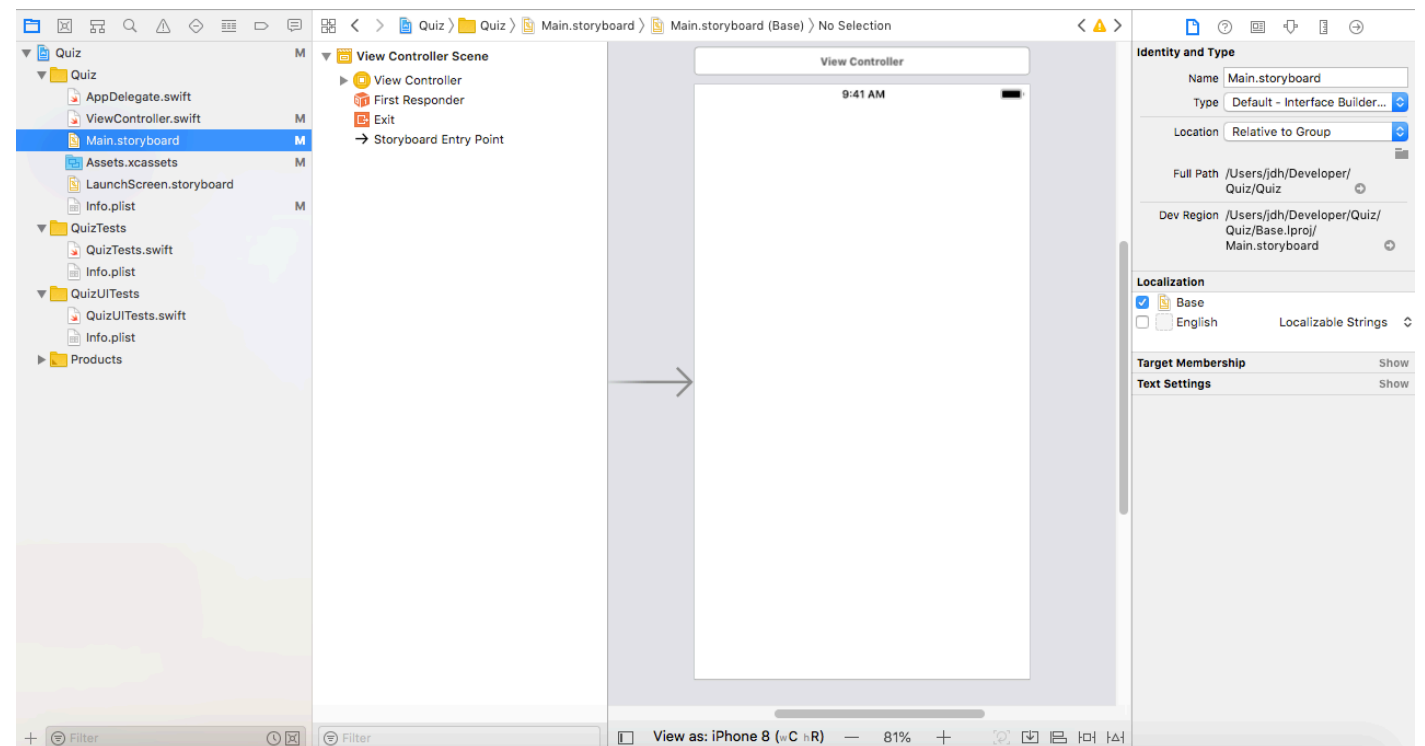
OK to ignore this error for now



Setting up a New Project

The book says "View as: iPhone 7 (wC hR)"; but iPhone 7 isn't available (in latest version of Xcode) – default will be iPhone 8

- This is OK...we won't worry about details of devices and screen appearance in this first project



Model-View-Controller

From the beginning, it's important to understand the Model-View-Controller (MVC) software architectural pattern



Model-View-Controller

From the beginning, it's important to understand the Model-View-Controller (MVC) software architectural pattern



- ▶ Key thing is the *separation of concerns*: the view does not know anything about the model
- ▶ And the model does not know anything about the view
- ▶ The controller sits in the middle and "translates" the model info to view info
 - and manages the view based on interactions that the user has with the view

Model-View-Controller

Model: this is Swift code that you create: it's where you keep track of whatever it is that your app is doing (game state, data that you're collecting, etc.)



Model-View-Controller

Controller: this is Swift code that you create; it has a certain formal structure through which to interact with the GUI elements

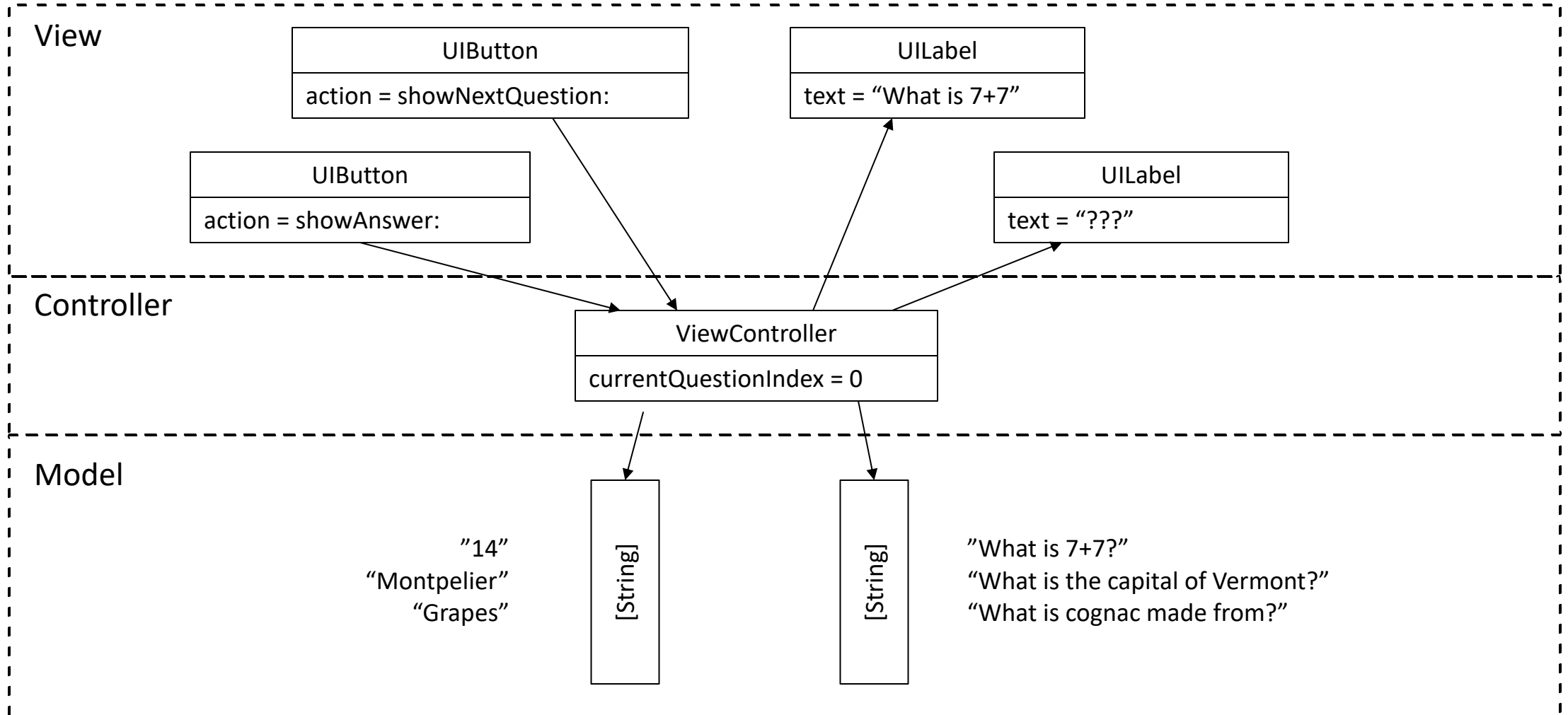


Model-View-Controller

View: this is code from the UIKit library (and from other iOS libraries); you use it by creating your user interface



The Quiz App

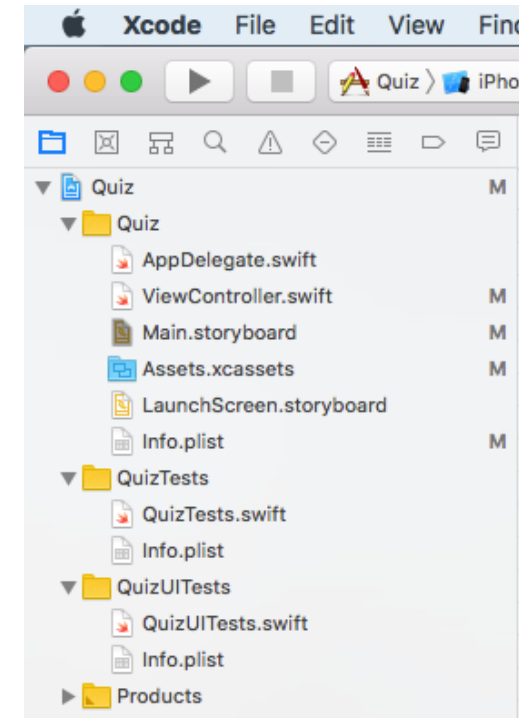


Components of an iOS App Project

Some of the components of an iOS App are:

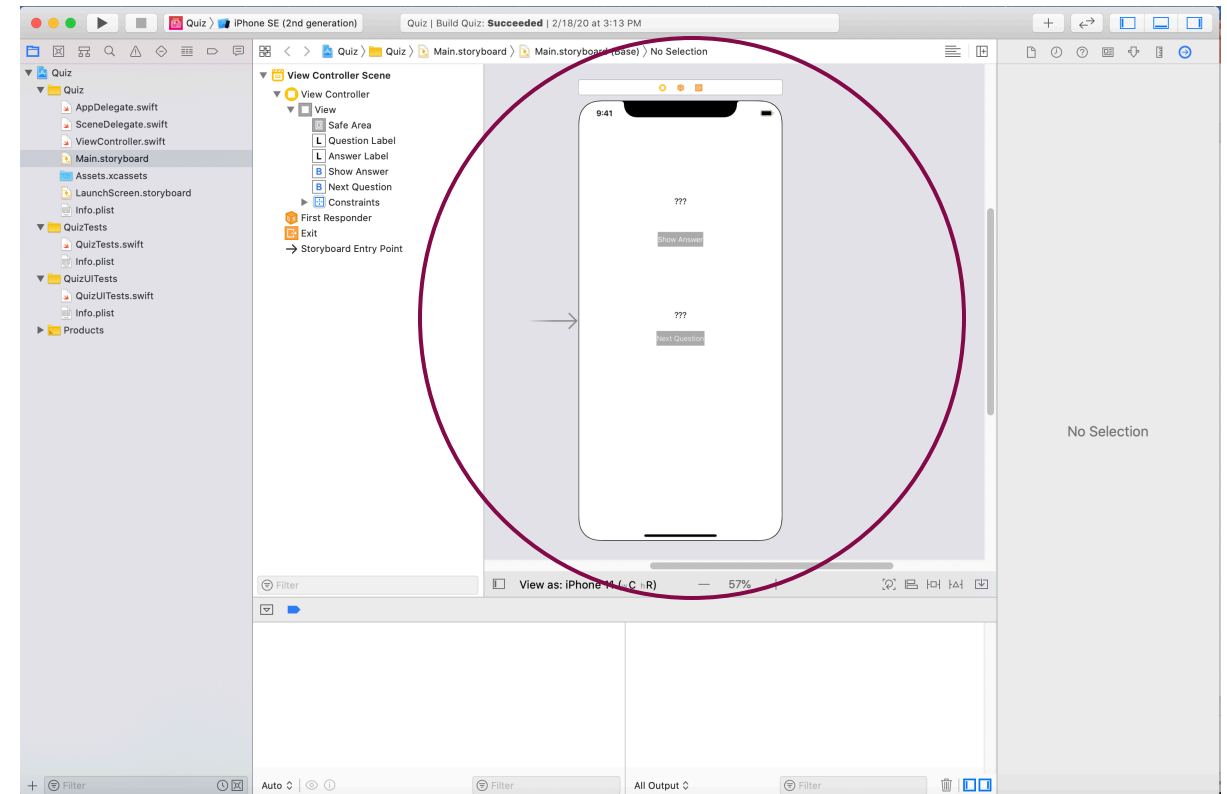
1. Main.storyboard: a representation of the UI
2. ViewController.swift: the code that runs a UI
3. Info.plist: high-level information about your app; tells the OS everything it needs to run your app

There are various other files and folders in a project, but these are the important components for now



The Storyboard

- ▶ In Xcode, the Storyboard is the graphic representation of an app's user interface
- ▶ This is how you build/modify your user interface
- ▶ In reality, the storyboard is a cryptic xml file, but Xcode "renders" the xml into something that looks like an iOS screen



Storyboard vs. SwiftUI

Apple is gradually trying to replace the Storyboard interface with a new UI framework called SwiftUI

SwiftUI is interesting, and we'll talk about it later in the semester

- ▶ it uses a different software architecture called Model-View-View Manager (MVVM)
- ▶ Apple introduced it in order to solve a fundamental difficulty/weakness in iOS design: keeping the interface and the code in sync
- ▶ it essentially jams the creation of the UI into the code so that the interface is created dynamically

SwiftUI interfaces don't yet have all of the functionality of Storyboard-based interfaces

- ▶ For this class, you should probably stick with Storyboards for your group project
- ▶ if you want to try SwiftUI for your individual project, that's fine

Storyboard vs. Code-based UI

Even without SwiftUI, it's possible to create an interface entirely in code

'The drawback of doing this is that the only way you can see your UI is by actually running your app

Advice from experts

- ▶ design your interface using Storyboards
- ▶ use code when necessary to tweak the interface dynamically a little bit if necessary

View Controllers

View controller

- ▶ the part of your code that handles interactions with the interface
- ▶ each screen of your interface has its own view controller
- ▶ listens for button presses; reads information from text fields, etc.

For each screen in your app

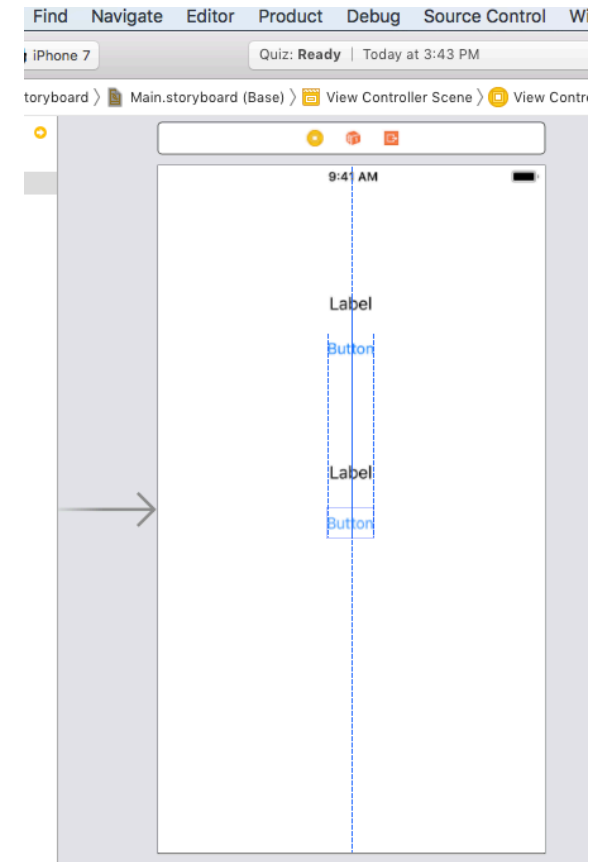
- ▶ you'll create your own VC class that is a subclass of `UIViewController` (from `UIKit`)
- ▶ and this is where you'll implement the code to "control" your app

Quiz App

Next, from the book (Chapter One), some details of building Storyboard UI

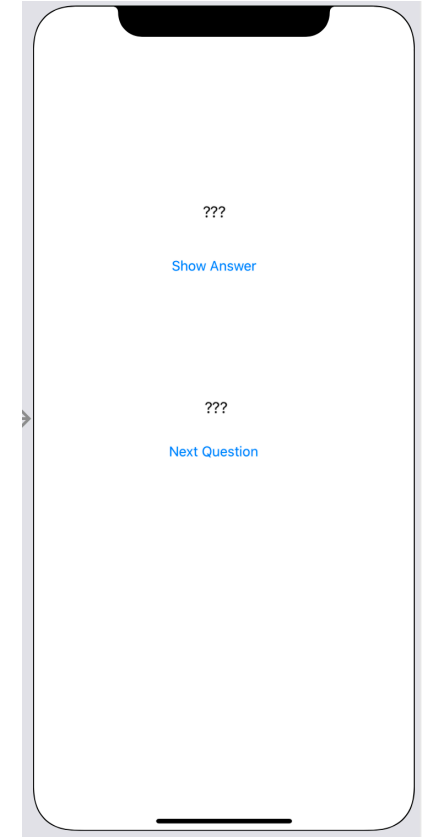
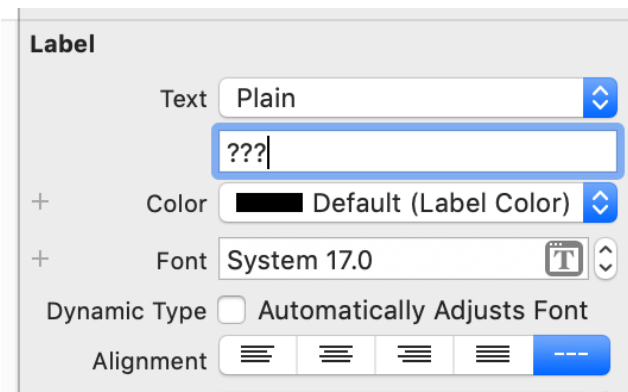
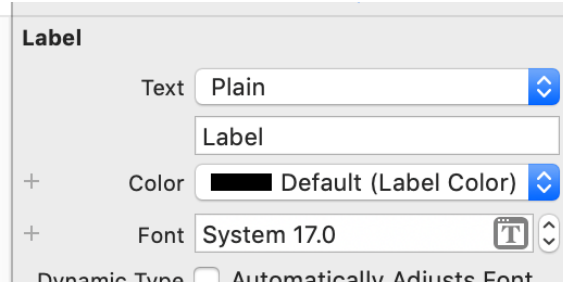
Adding Labels and Buttons

- ▶ For example: drag and drop a Label on to the canvas, and put it near the top
- ▶ The dashed blue line is the horizontal center of the view—useful for alignment between items or on the Storyboard itself
- ▶ Then drag a second label and put it in near the middle
- ▶ Then add two buttons—one underneath each label



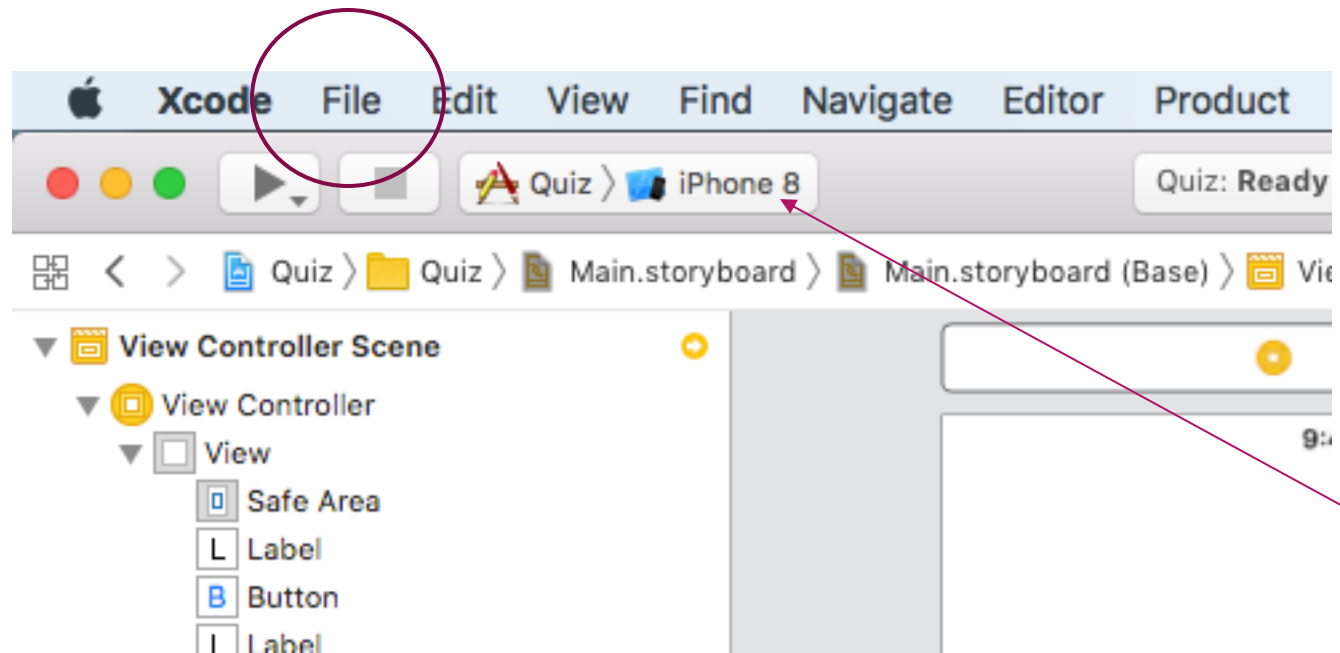
Change the Names of the Labels

- ▶ Change the text for the labels to ???
- ▶ Double-click on a label to do this
- ▶ Change the text for the buttons also



Build and Run

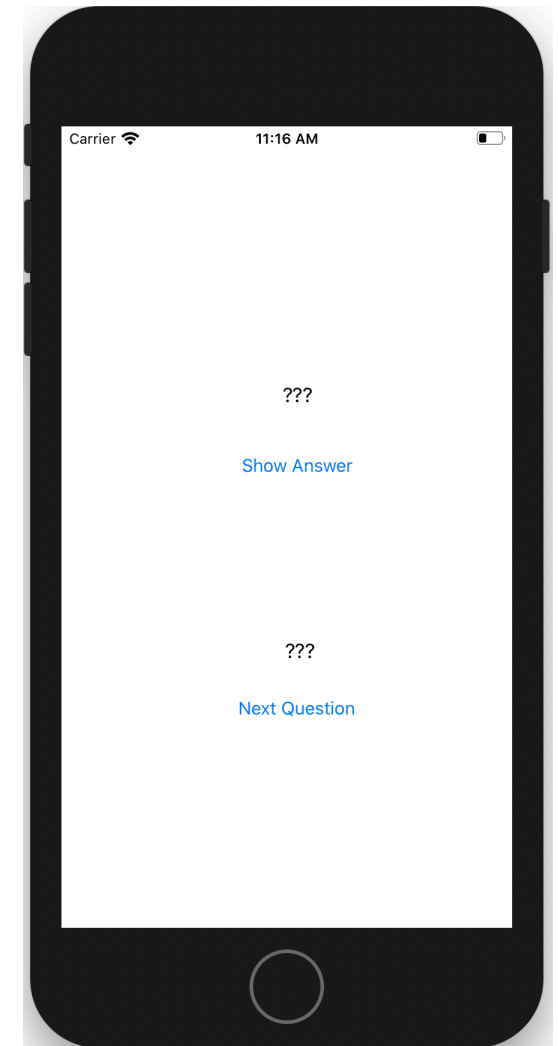
The first time you do this, it will take a while



OK to have iPhone 8 here (or some other hardware version)

The App in the Simulator

- ▶ The Simulator is part of Xcode
- ▶ It's good for rapid prototyping and development
- ▶ Apple's documentation:
 - Simulator is a great tool for rapid prototyping and development of your app allowing you to see the results of changes quickly, debug errors, and run tests.
 - It is also important to test your app on physical devices as there are hardware and API differences between a simulated device and a physical one.
 - In addition to those differences, Simulator is an app running on a Mac and has access to the computer's resources, including the CPU, memory, and network connection.
 - These resources are likely to be very different in capacity and speed than those found on a mobile device requiring tests of performance, memory usage, and networking speed to be run on physical devices.



Simulator vs. Actual iPhone

General differences:

- ▶ Performance testing for processing, graphics, and networking must be done on physical devices for accurate results.
- ▶ Simulator is an app running on a Mac using the computer's resources including the CPU, memory, and network connection. As a result, the simulator is not an accurate test of an app's performance, memory usage, and networking speed. Performance testing results in Simulator can be used only for relative differences in app functionality.
- ▶ User testing should be done on physical devices when real world results are required.
- ▶ User interaction with a pointer and keyboard is different from using fingers on iOS and watchOS, or from the focus-based model used on tvOS.

Simulator vs. Actual iPhone

The following hardware is not supported in Simulator:

- ▶ ambient light sensor
- ▶ audio input, except for using Siri by choosing Hardware > Siri.
- ▶ barometer
- ▶ bluetooth
- ▶ camera
- ▶ motion support (accelerometer and gyroscope)
- ▶ proximity sensor

more info: <https://help.apple.com/simulator/mac/current/#/devb0244142d>

Bottom Line

The simulator is fine for developing and most of the testing of CS275 apps

- ▶ but you really need to run and test your app on an actual iPhone

Important rule: design and test for a smaller screen! Don't start with iPhone X or iPhone 11 if you think that you will have to run on an iPhone 7 or 8

Auto Layout and Interface Builder

Auto Layout (AL), part of Xcode—it assists UI development by computing constraints between UI elements

Interface Builder (IB) is the graphic UI builder in Xcode—for our purposes, it's the best way to create an interface

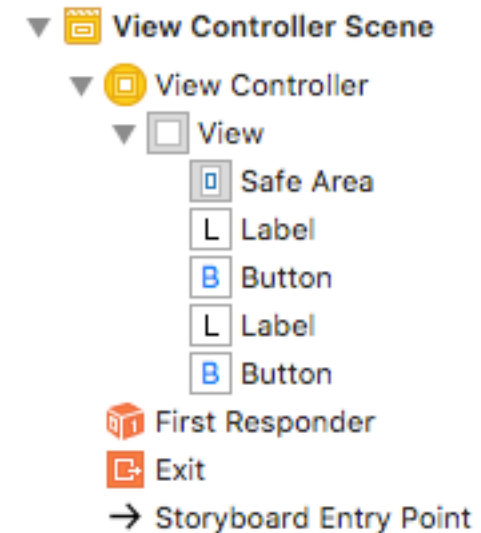
Here's an interesting discussion:

<https://www.toptal.com/ios/ios-user-interfaces-storyboards-vs-nibs-vs-custom-code>

Auto Layout

The book is slightly out of date in some places

- ▶ example: the Safe Area is the portion of the view that is unobscured by bars and other content
- ▶ the book (Figure 1.17 on p. 14) shows Top Layout Guide and Bottom Layout Guide
- ▶ but starting with iOS 11, Apple has replaced this with a single Safe Area layout guide

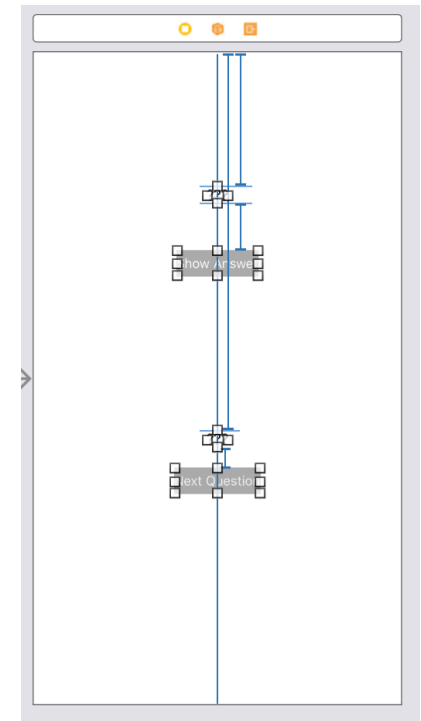


Constraints

- ▶ In order to render a layout, the UI system (Cocoa Touch) has to know how to place each element
- ▶ Each element (widget) must be full constrained in the horizontal and in the vertical directions
- ▶ Constraints can exist between elements
- ▶ Or from an element to its container (think of the whole screen itself)
- ▶ Or to an anchor point such as the horizontal or vertical centerline
- ▶ If you don't fully constraint each element, then the UI system will make assumptions
 - and these assumptions could be wrong
 - meaning that your interface might not actually look the way you want it to when you run it
 - worst case, one of your widgets might not even be on screen

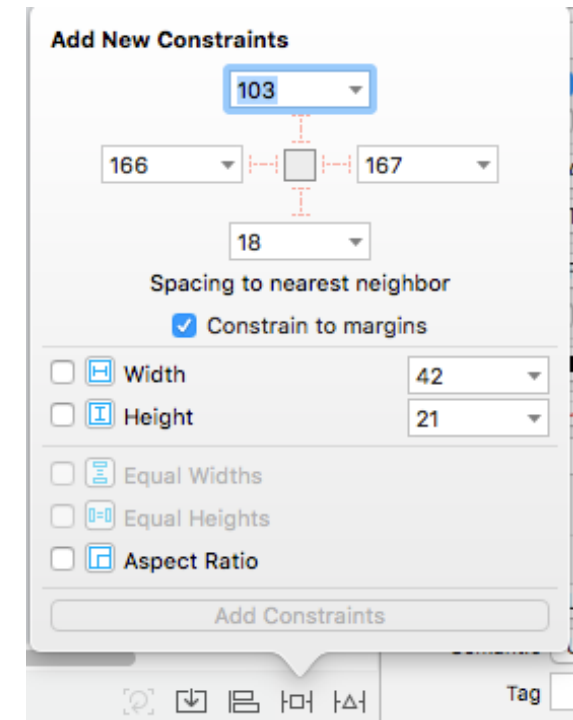
Constraints

- ▶ Here's the Quiz App
- ▶ The x and y position of every widget is completely specified
- ▶ Each widget is either "tied" to something above or below it
- ▶ Or to the vertical centerline of the view itself



Adding Constraints

- ▶ Figure 1.20 shows a previous menu for Adding New Constraints
- ▶ Here's the new version
- ▶ There's no longer an explicit "Update Frames" selection—Update Frames is no longer necessary
- ▶ In newer Xcode versions, Auto Layout updates frames automatically
- ▶ Basic idea: select a layout element, click the “Add New Constraints” button, and this dialog box will appear



Constraints and Auto Layout

- ▶ In general, it's tricky to build a fully constrained layout that looks like you want it to
- ▶ In my experience: building the interface is the most difficult part of mobile-app development
- ▶ It takes practice

Connections

A **connection** lets one object know where another object is in memory so that the two objects can communicate

There are two kinds of connections we can make in Interface Builder

1. *outlet*: a reference in your code to an object on your storyboard
2. *action*: a method in your code that gets triggered by an action in the UI
 - for example, touching a button on the screen

Making a Connection

Here's what an outlet looks like:

```
import UIKit

class ViewController: UIViewController {
    ● @IBOutlet var questionLabel: UILabel!
    ● @IBOutlet var answerLabel: UILabel!

    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view.
    }
}
```

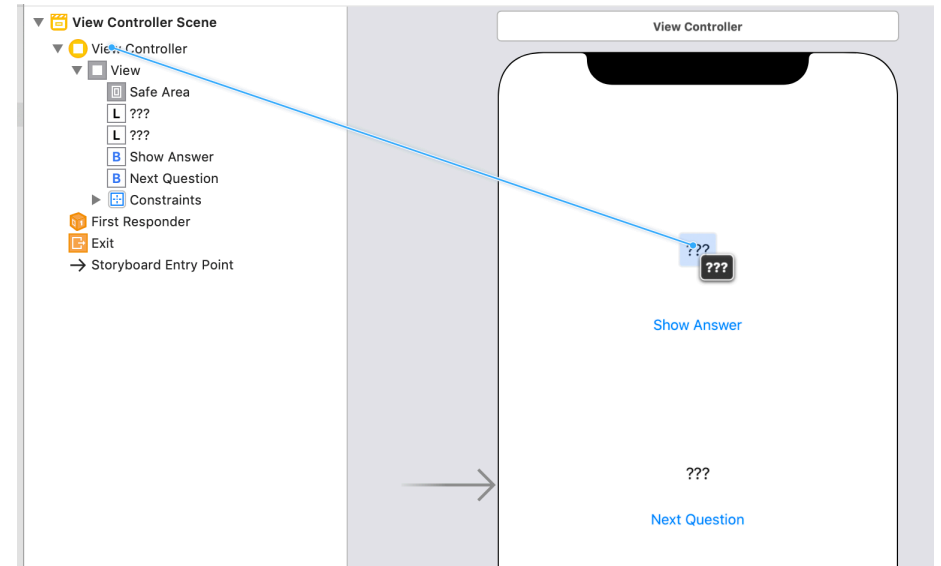
The filled-in circles, and the @IBOutlet, signify the connections from the code to the UI

- the exclamation point: we'll discuss soon why this is necessary
- the at sign: <https://docs.swift.org/swift-book/ReferenceManual/Attributes.html>

Making a Connection

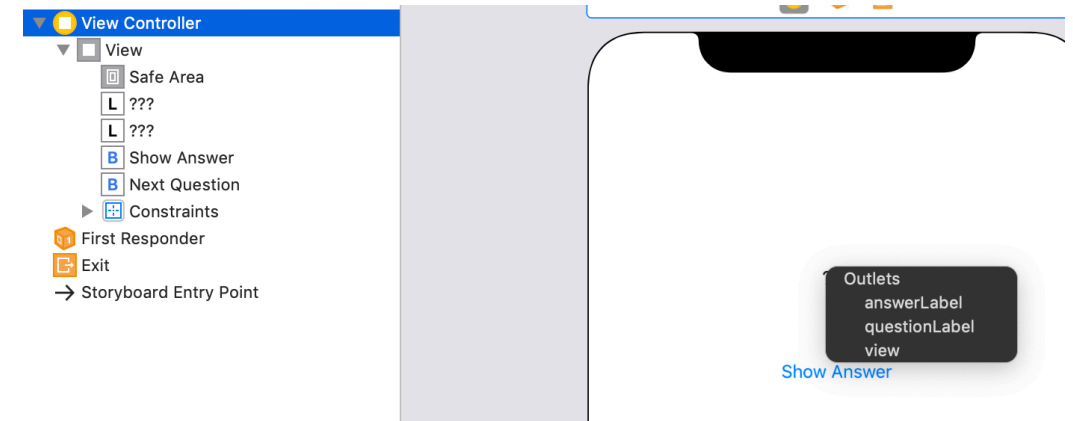
Then you have to tie the variable in ViewController.swift to the UI object (one of the labels)

- control-drag (or right-click and drag) from ViewController to one of the UILabels



Making a Connection

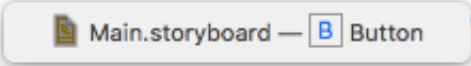
- ▶ Then release, and you'll see the names of the variables that you've marked with @IBOutlet
- ▶ Then pick one of them (for the top label, pick answerLabel)
- ▶ Then connect questionLabel to the bottom UILabel, using the same technique



Connections

- ▶ After you've made a connection, the little round button on the line number will be filled in
- ▶ And if you click on the button, Xcode will show you to which UI object that part of the code is connected
- ▶ This can help in debugging—if the circle is not filled in, then the UI element is not connected to anything in your code

```
11 class ViewController: UIViewController {  
12  
13     @IBOutlet var questionLabel: UILabel!  
14     @IBOutlet var answerLabel: UILabel!  
15  
16     @IBAction func question(_ sender: UIButton) {  
17  
18     }  
19  
20     @IBAction func showAnswer(_ sender: UIButton) {  
21  
22     }  
23 }  
24
```

A screenshot of the Xcode interface. A tooltip is visible, showing a connection from a UIButton (labeled 'B Button') in 'Main.storyboard' to the 'showAnswer' method in the code. The code is for a 'ViewController' class, and the 'showAnswer' method is currently selected and highlighted in blue.

Defining Action Methods

- ▶ Action method: essentially, a callback
- ▶ When a UIButton is tapped, a method is called on another object—the target
- ▶ The method that is triggered is called the action
- ▶ Here, the target for the two buttons will be the instance of ViewController
- ▶ Each button will have its own action (method on ViewController)

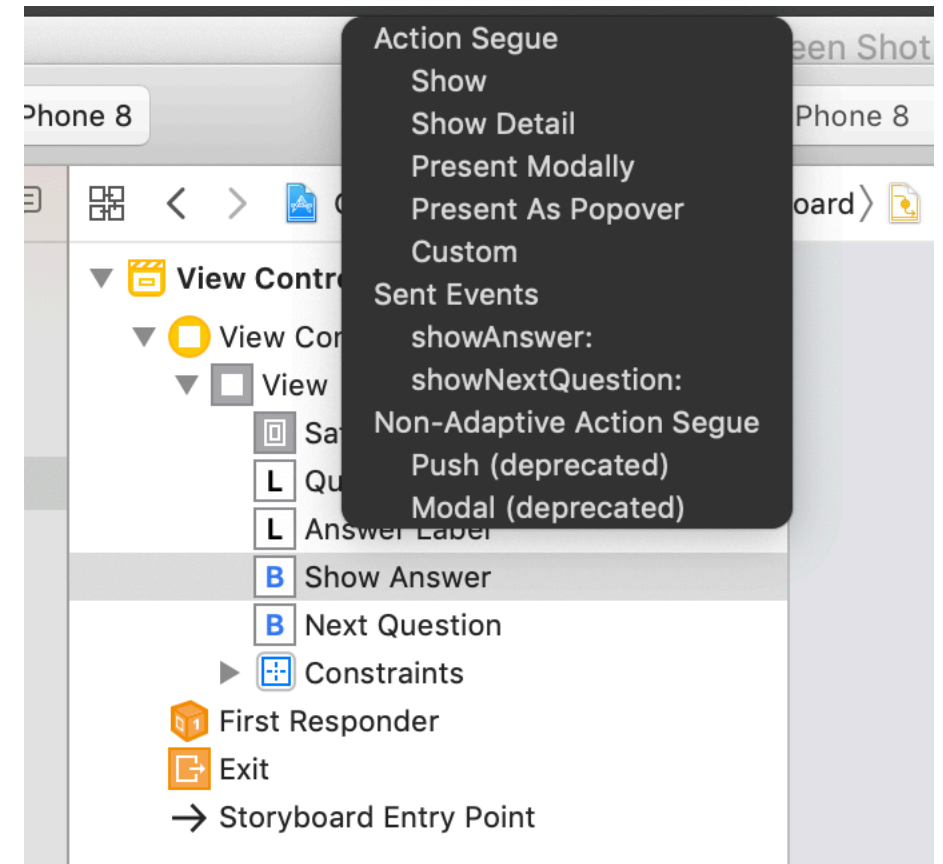
Defining Action Methods

Action methods:

```
class ViewController: UIViewController {  
    ● @IBOutlet var questionLabel: UILabel!  
    ● @IBOutlet var answerLabel: UILabel!  
  
    ● @IBAction func showNextQuestion(_ sender: UIButton) {  
        // implementation  
    }  
  
    ● @IBAction func showAnswer(_ sender: UIButton) {  
        // implementation  
    }  
}
```


Defining Targets and Action Methods

- ▶ Control-drag (or right-click-drag) from the UI object to its target
- ▶ Control-drag from the top button to the ViewController
- ▶ Release, and you'll see a list of possible actions
- ▶ Pick showAnswer: for the top button
- ▶ Same thing for bottom button: pick showNextQuestion:



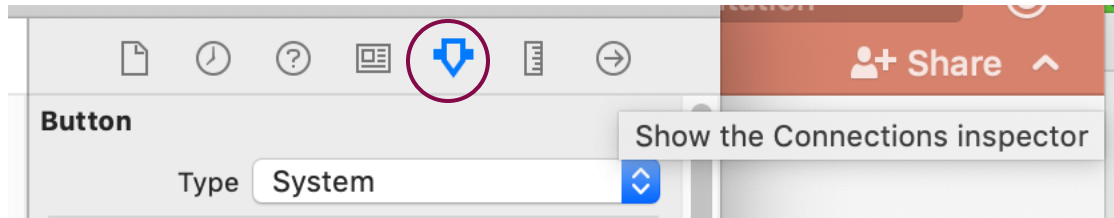
Defining Targets and Action Methods

Then, in the code, you'll see the bubbles next to the two methods filled in:

```
11 class ViewController: UIViewController {  
12     @IBOutlet var questionLabel: UILabel!  
13     @IBOutlet var answerLabel: UILabel!  
14  
15     override func viewDidLoad() {  
16         super.viewDidLoad()  
17         // Do any additional setup after loading the view.  
18     }  
19  
20     @IBAction func showNextQuestion(_ sender: UIButton) {  
21  
22     }  
23  
24     @IBAction func showAnswer(_ sender: UIButton) {  
25  
26     }  
27 }
```

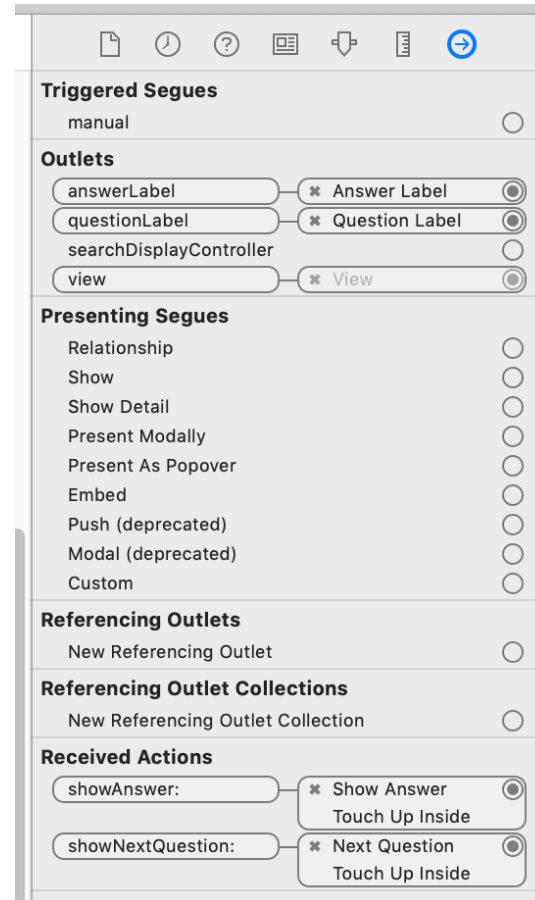
Connections Inspector

The Connections Inspector for the Storyboard will show you all of the connections



Connections Inspector

Here are the four connections



Creating the Model Layer

Here's the model code:

```
let questions: [String] = [  
    "What is 7+7?",  
    "What is the capital of Vermont?",  
    "What is cognac made from?"]
```

```
let answers: [String] = [  
    "14",  
    "Montpelier",  
    "Grapes"]
```

```
var currentQuestionIndex = 0
```

Implement the Action Methods

```
@IBAction func showNextQuestion(_ sender: UIButton) {  
    currentQuestionIndex += 1  
    if currentQuestionIndex == questions.count {  
        currentQuestionIndex = 0  
    }  
  
    let question: String = questions[currentQuestionIndex]  
    questionLabel.text = question  
    answerLabel.text = "???"  
}  
  
@IBAction func showAnswer(_ sender: UIButton) {  
    let answer: String = answers[currentQuestionIndex]  
    answerLabel.text = answer  
}
```

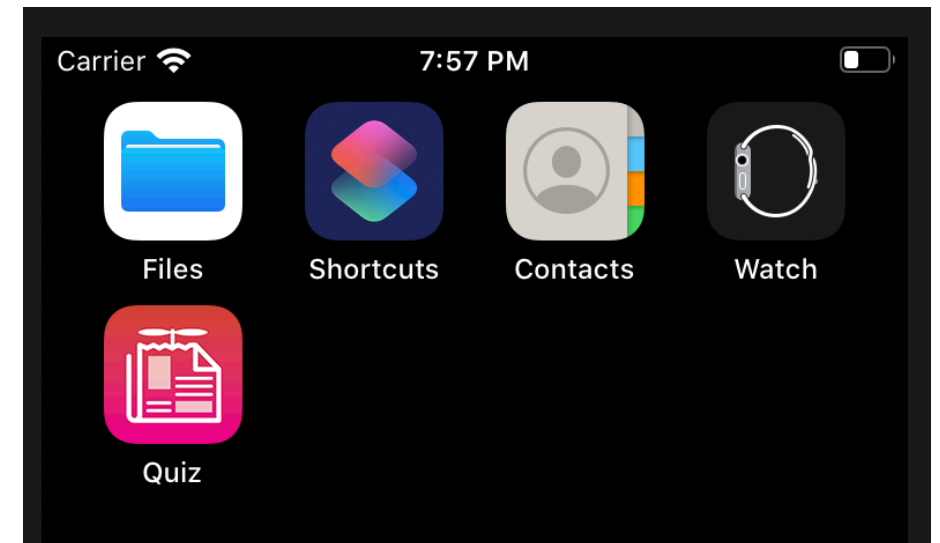
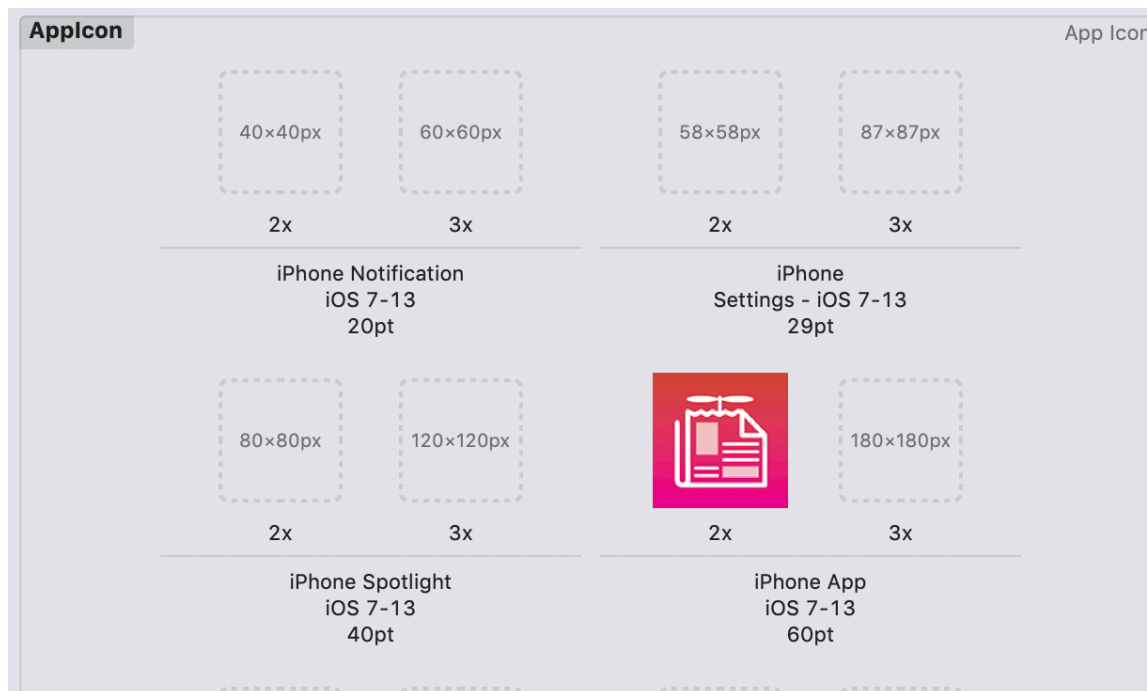
Load the First Question

- ▶ Every instance of `UIViewController` inherits a `viewDidLoad()` method
- ▶ This is called when the view is loaded (i.e., when the view for that view controller is first put into memory)
- ▶ This gives us a hook to do initializations for the view
- ▶ So in this case, here's where we can put the first question in the view

```
override func viewDidLoad() {  
    super.viewDidLoad()  
    questionLabel.text = questions[currentQuestionIndex]  
}
```

Adding Icons

- ▶ Assets.xcAssets -> AppIcons
- ▶ Book shows how to get the file Quiz-120.png



Last Thing: Launch Screen

We can select the image that will show while an application is in the process of being launched

Choose the top-level Quiz in the project navigator

► then choose Main

▼ App Icons and Launch Images

App Icons Source

AppIcon



Launch Screen File

LaunchScreen



▼ Frameworks, Libraries, and Embeddables

LaunchScreen

Main

Launch Screen

Setting an image to display during app launch is good UI design

- ▶ it lets the user know that something is happening

The Quiz app launches very fast

- ▶ and so the launch image will appear very briefly

UILabel!

When To Use An Implicitly Unwrapped Optional

There are two main reasons that one would create an Implicitly Unwrapped Optional. All have to do with defining a variable that will never be accessed when nil because otherwise, the Swift compiler will always force you to explicitly unwrap an Optional.

1. A Constant That Cannot Be Defined During Initialization

Every member constant must have a value by the time initialization is complete. Sometimes, a constant cannot be initialized with its correct value during initialization, but it can still be guaranteed to have a value before being accessed.

Using an Optional variable gets around this issue because an Optional is automatically initialized with nil and the value it will eventually contain will still be immutable. However, it can be a pain to be constantly unwrapping a variable that you know for sure is not nil. Implicitly Unwrapped Optionals achieve the same benefits as an Optional with the added benefit that one does not have to explicitly unwrap it everywhere.

A great example of this is when a member variable cannot be initialized in a UIView subclass until the view is loaded.

Exercise

- ▶ Try this yourself
- ▶ Go through Chapter 1 and build and run the simple app