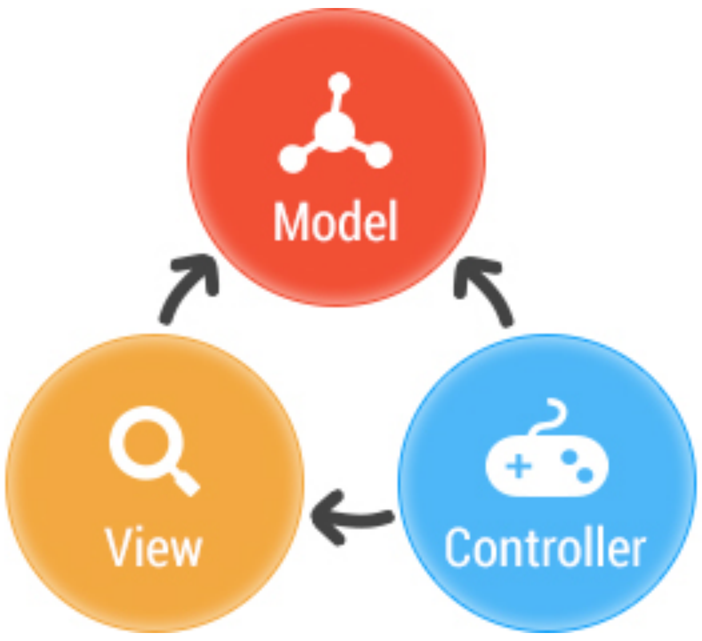
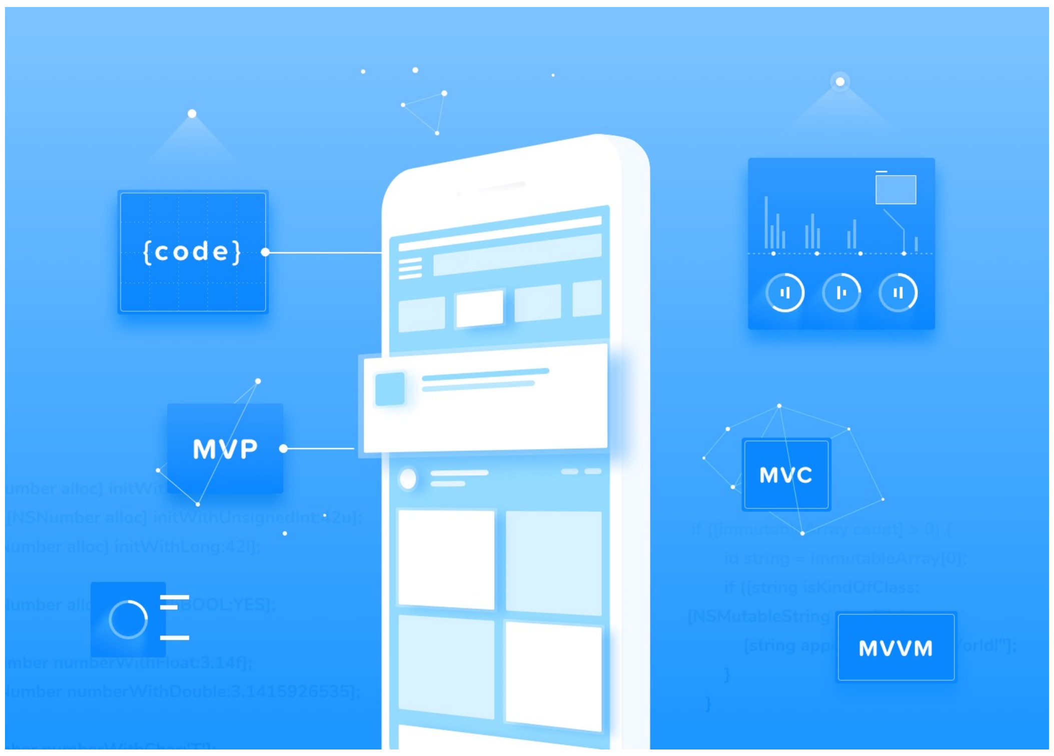
CHAPTER SIX : INTRODUCTION TO DESIGN PATTERN MVC





The **Model-View-Controller (MVC)** is an architectural pattern that separates an application into three main logical components: the model, the view, and the controller. Each of these components are built to handle specific development aspects of an application. MVC is one of the most frequently used industry-standard web development framework to create scalable and extensible projects.

Since this book is a practical tutorial in iOS development, we will not only focus on the theoretical part of software engineering and computer science. However, in order to develop an application in iOS, you have to understand and use this MVC design pattern. This Model-View-Controller design pattern is one of the most popular design pattern in Graphic User Interface design, and it will definitely make your task a lot easier..

I. OVERVIEW OF THE MVC PATTERN

No matter which programming language you learn, one important concept that you need to know is **Separation of Concerns** (SoC). The concept is pretty simple. Here, the Concerns are different aspects of software functionality. This concept encourages developers to break a complicated feature or program into several areas of concern such that each area has its own responsibility. The delegate pattern, that we explained in the earlier chapters, is one of the examples of SoC.

The **model-view-controller** (MVC) concept is another example of SoC. The core idea behind MVC is to separate a user interface into three areas (or groups of objects) that each area is responsible for a particular functionality. As the name suggests, MVC breaks a user interface into three parts:

- **Model** – a model is responsible for holding the data or any operations on the data. The model can be as simple as an array object that stores the table data. Add, update and delete are examples of the operations. In the business world, these operations are usually known as business rules.
- **View** – a view manages the visual display of information. For example, UITableView displays data in a list format.
- **Controller** – a controller is the bridge between the model and the view. It translates the user interaction from the view (e.g. tap) into the appropriate action to be performed in the model. For example, a user taps the delete button in the view. Consequently, the controller triggers a delete operation in the model. Once finished, the model requests the view to refresh itself so as to reflect the update of the data model.

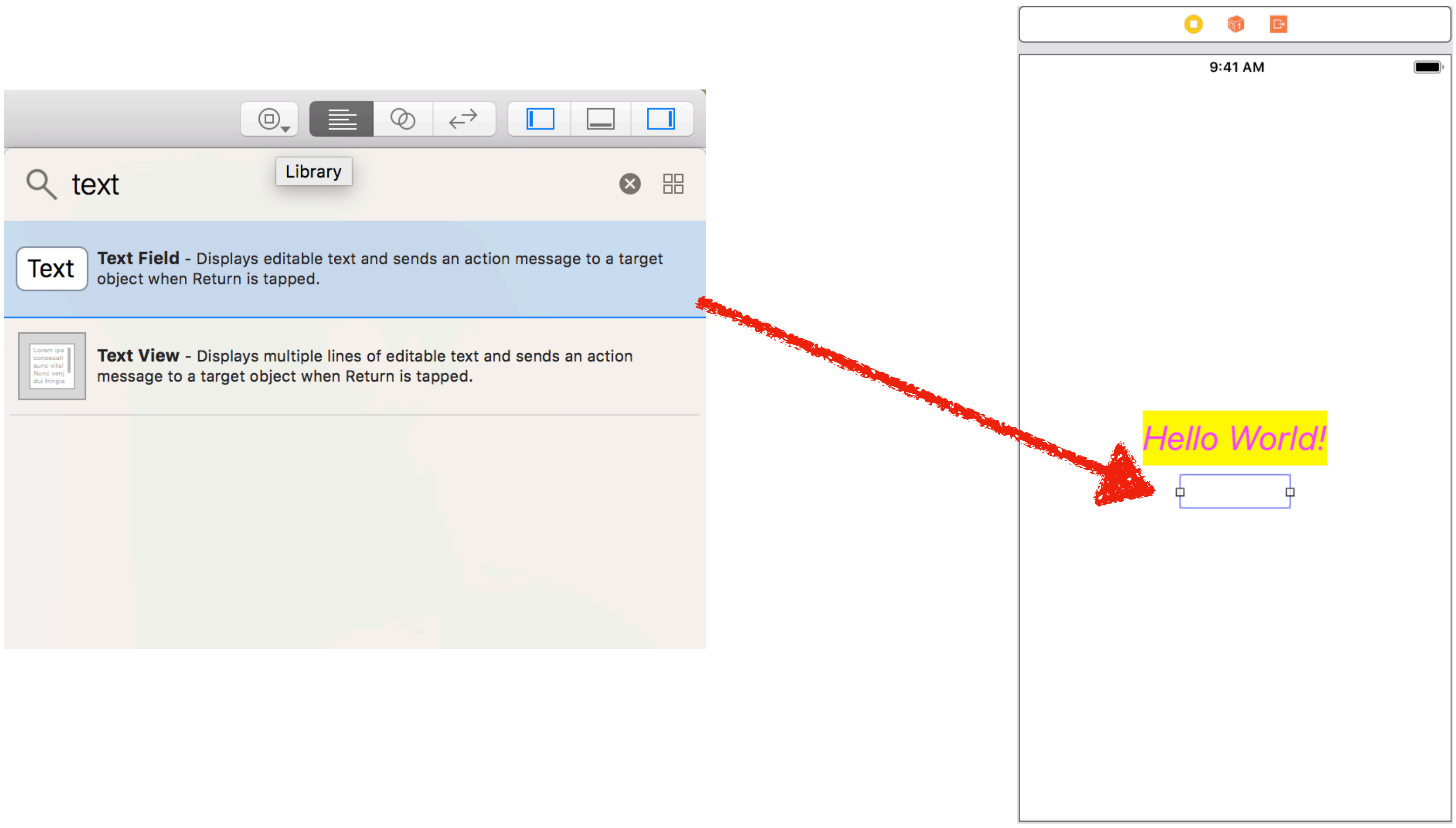
II. APPLY MVC TO HELLO WORLD APP: VIEW

Still remember our Hello World App? If you didn't finish the app, you can download the project from the following link to get started in this chapter.

https://github.com/CarlistleZ/MyiOSTutorial/blob/master/ChapterSix_Start_HelloWorld.zip

In the previous chapter, we added a button to our UI and made it functional by display an alert box. An extremely example as it is, it doesn't really follow the MVC design pattern. In this chapter, we will add a text field for username and display a "customized" information in the alert box.

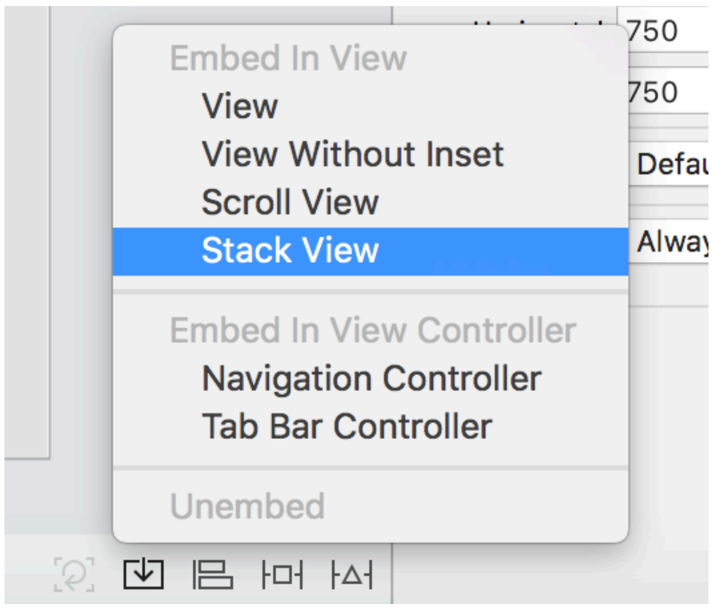
Go to **Library**, choose **Text Field**. This is the object that holds the user input text. Drag it to the Main.storyboard. The storyboard will look like this:

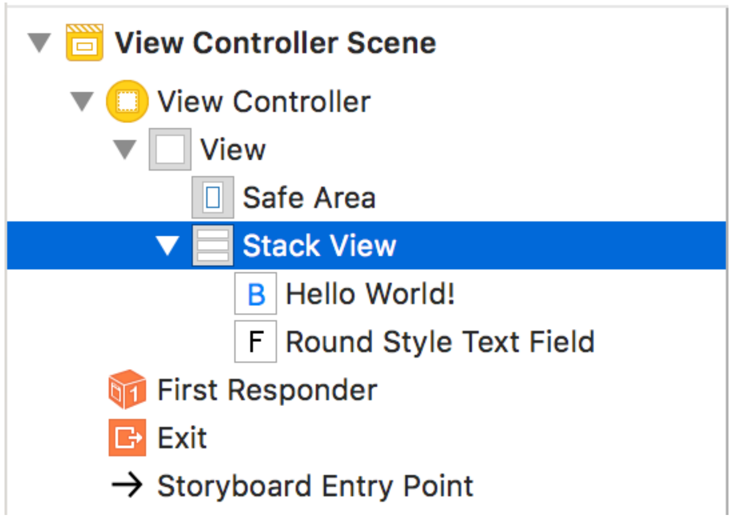


Text Field - Displays editable text and sends an action message to a target object when Return is tapped.

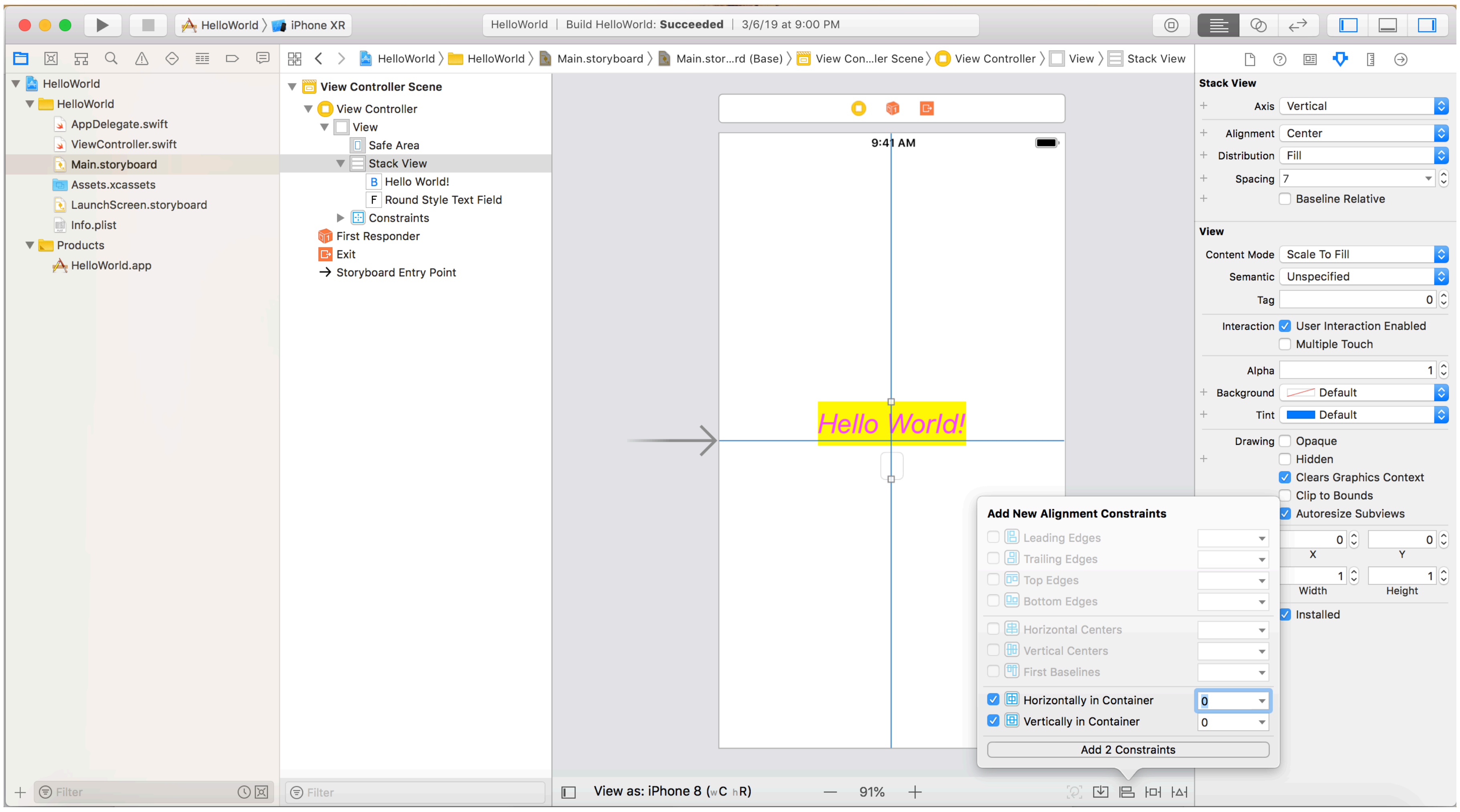
Text View - Displays multiple lines of editable text and sends an action message to a target object when Return is tapped.

In order to make our new UI display properly, we need to make some modifications on layout constraints. First go to **Document Outline** and delete the horizontal and the vertical constraints of the Button. Choose the button and the text field at the same time by command click. Choose **Embed in > Stack View**. In Document Outline, the vertical stack view now contains two objects.





Fix the stack view in the center vertically and horizontally just like what we did before. The UI will look like this:


HelloWorld 〉 iPhone XR — HelloWorld | Build HelloWorld: **Succeeded** | 3/6/19 at 9:00 PM

HelloWorld 〉 HelloWorld 〉 Main.storyboard 〉 Main.stor...rd (Base) 〉 View Con...ler Scene 〉 View Controller 〉 View 〉 Stack View

- HelloWorld
  - HelloWorld
    - AppDelegate.swift
    - ViewController.swift
    - Main.storyboard
    - Assets.xcassets
    - LaunchScreen.storyboard
    - Info.plist
  - Products
    - HelloWorld.app

- View Controller Scene
  - View Controller
    - View
      - Safe Area
      - Stack View
        - B Hello World!
        - F Round Style Text Field
      - Constraints
  - First Responder
  - Exit
  - Storyboard Entry Point

Stack View

Axis	Vertical
Alignment	Center
Distribution	Fill
Spacing	7
	☐ Baseline Relative

View

Content Mode	Scale To Fill
Semantic	Unspecified
Tag	0
Interaction	☑ User Interaction Enabled
	☐ Multiple Touch
Alpha	1
Background	Default
Tint	Default
Drawing	☐ Opaque
	☐ Hidden
	☑ Clears Graphics Context
	☐ Clip to Bounds
	☑ Autoresize Subviews
X	0
Y	0
Width	1
Height	1
	☑ Installed

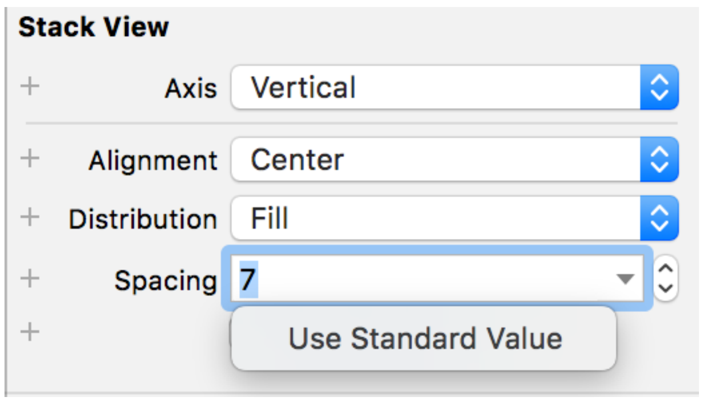
Add New Alignment Constraints

☐ Leading Edges	
☐ Trailing Edges	
☐ Top Edges	
☐ Bottom Edges	
☐ Horizontal Centers	
☐ Vertical Centers	
☐ First Baselines	
☑ Horizontally in Container	0
☑ Vertically in Container	0

Add 2 Constraints

View as: iPhone 8 (wC hR) — 91%

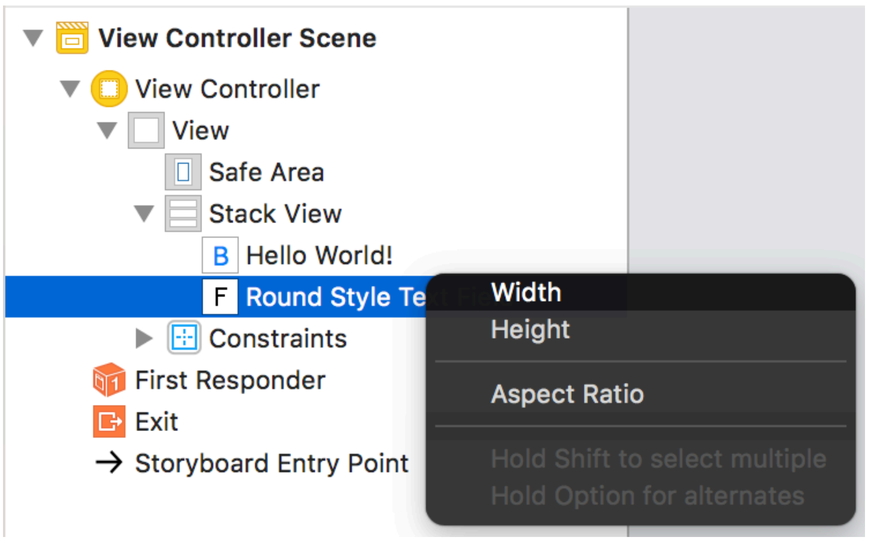
Whenever we create a stack view, we always need to change the **Spacing** between items in the stack view. Instead of using 7 points, choose **Use Standard Value**. With this option, the spacing will be adjusted automatically during each repaint. You will get a consistent spacing with the rest of the UI. If you want to have a robust and pretty stack view, you almost always have to choose this option.

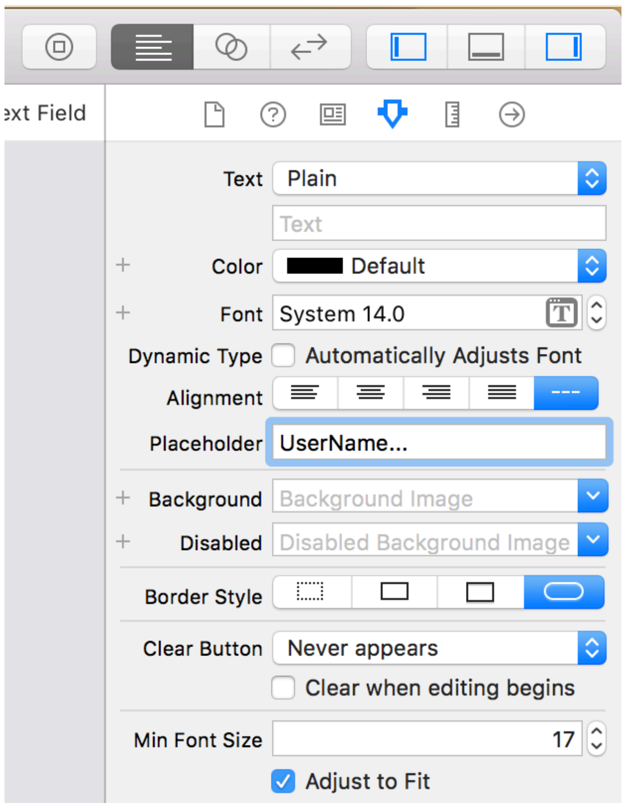

Stack View

Axis	Vertical
Alignment	Center
Distribution	Fill
Spacing	7

Now let's fix our text field. In Document Outline, **control-drag** on Round Style Text Field horizontally and choose **Width**. Change the **Width Constant** to **130** in Attribute Inspector. With this constraint: **Round Style Text Field.Width Equal 130 pts**, the width will have a prioritized constant value of 130 points during the repaint.

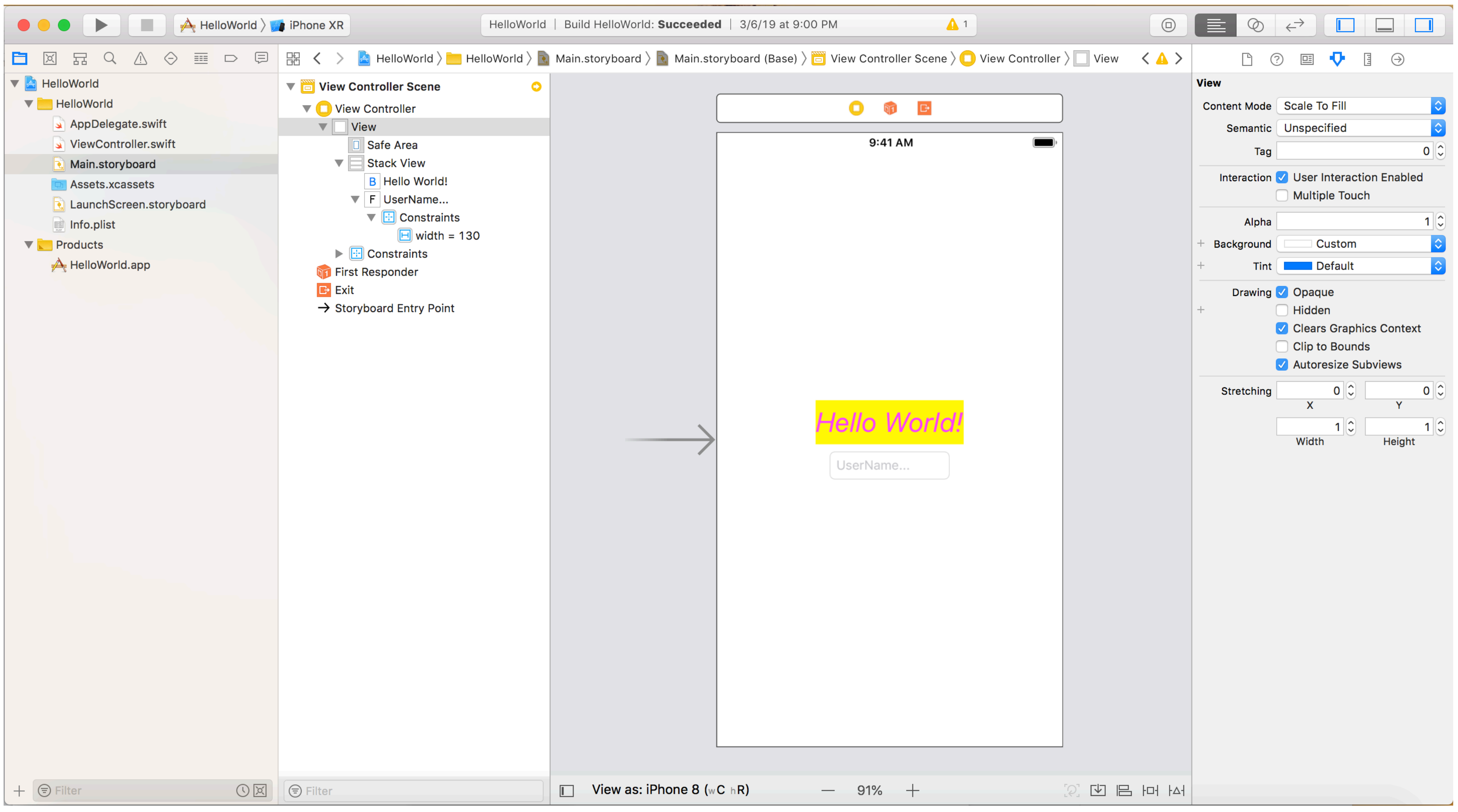
Choose the text field in **Document Outline** again, go to **Attribute Inspector**, set the content of **Placeholder** to **Username…** This text will show if the text field is empty and unselected.




ext Field

Text	Plain
	Text
Color	Default
Font	System 14.0
Dynamic Type	☐ Automatically Adjusts Font
Alignment	
Placeholder	UserName...
Background	Background Image
Disabled	Disabled Background Image
Border Style	
Clear Button	Never appears
	☐ Clear when editing begins
Min Font Size	17
	☑ Adjust to Fit

Check the final result of the Main.storyboard after all the modifications.



View

Content Mode	Scale To Fill
Semantic	Unspecified
Tag	0
Interaction	☑ User Interaction Enabled
	☐ Multiple Touch
Alpha	1
Background	Custom
Tint	Default
Drawing	☑ Opaque
	☐ Hidden
	☑ Clears Graphics Context
	☐ Clip to Bounds
	☑ Autoresize Subviews
Stretching	X 0, Y 0
	Width 1, Height 1

III. APPLY MVC TO HELLO WORLD APP: MODEL

In the previous section, we finished the **View** of MVC. In this section, we will define the **model**. If you have trouble identify what the model is, just ask yourself this question:

WHAT do I want to present?

Since our problem is simple, it is quite obvious that the model in our case is the username from the user input.

Go to ViewController.swift, in the `class ViewController`, add an optional string variable to our class.

```
var username: String?
```

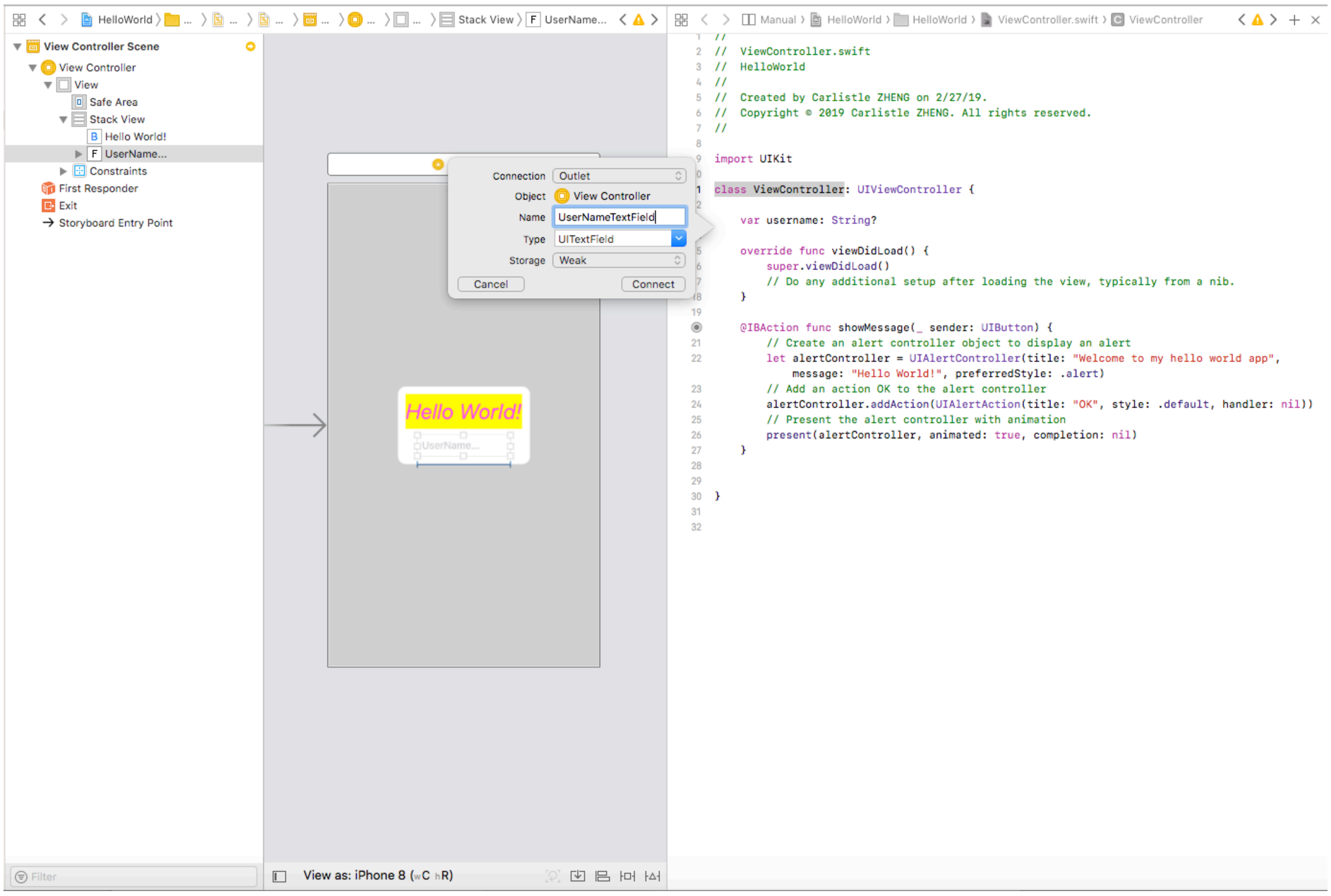
The keyword var indicates that we are declaring an variable. The identifier(or the name of the attribute) is username. Its type is `String?` , a String optional. If you don't fully understand what an optional is, I highly recommend you to go back to Chapter Five and go through the optional again.

We have to display the right message according to the user input. When an optional is not initialized, its value is **nil**. In order to avoid this case when we are trying to put the string optional in the alert box, we have to **unwrap the optional** in the controller.

VI. APPLY MVC TO HELLO WORLD APP: CONTROLLER

Click **Show Assistant Editor** on the upper right corner, choose `Main.storyboard` on the left and `ViewController.swift` on the right. In order to control the view and the model, the controller has to know both of them. First, let's make the controller get the information from the view.

The information that we want from the view is the text content that user inputs as the username. So **control-drag** from the text field to the `class ViewController`.



Connection: Outlet
Object: View Controller
Name: UserNameTextField
Type: UITextField
Storage: Weak
Cancel | Connect

```
//
//  ViewController.swift
//  HelloWorld
//
//  Created by Carlistle ZHENG on 2/27/19.
//  Copyright © 2019 Carlistle ZHENG. All rights reserved.
//

import UIKit

class ViewController: UIViewController {

    var username: String?

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

    @IBAction func showMessage(_ sender: UIButton) {
        // Create an alert controller object to display an alert
        let alertController = UIAlertController(title: "Welcome to my hello world app",
            message: "Hello World!", preferredStyle: .alert)
        // Add an action OK to the alert controller
        alertController.addAction(UIAlertAction(title: "OK", style: .default, handler: nil))
        // Present the alert controller with animation
        present(alertController, animated: true, completion: nil)
    }


}
```

Choose Outlet as connection type because we want to "connect" the text field to the controller so that we can get the changes in the `ViewController.class`. Use the name **UserNameTextField** as the name of the outlet. Leave everything else unchanged and click connect. Xcode will generate this line of code in the controller:

```
    @IBOutlet weak var UserNameTextField: UITextField!
```

The second connection is between the controller and the model. We have to synchronize the **UITextField outlet** and **username String optional**. It is very important to keep in mind that a optional can be `nil`. In our case when the text field has no content, we have to display a default message. So there are two cases to deal with:

```
        // Unwrapping the optional user name
        let greetingMessage = userNameTextField.text != nil ?
userNameTextField.text! : “user"
```

or a short-hand version like this:

```
        // Unwrapping the optional user name
        let greetingMessage = userNameTextField.text ?? "user"
```

Here we define a constant greetingMessage to hold the text we want. If userNameTextField.text is `nil`, the string is "user" used. If not, greetingMessage holds the user input from the storyboard.

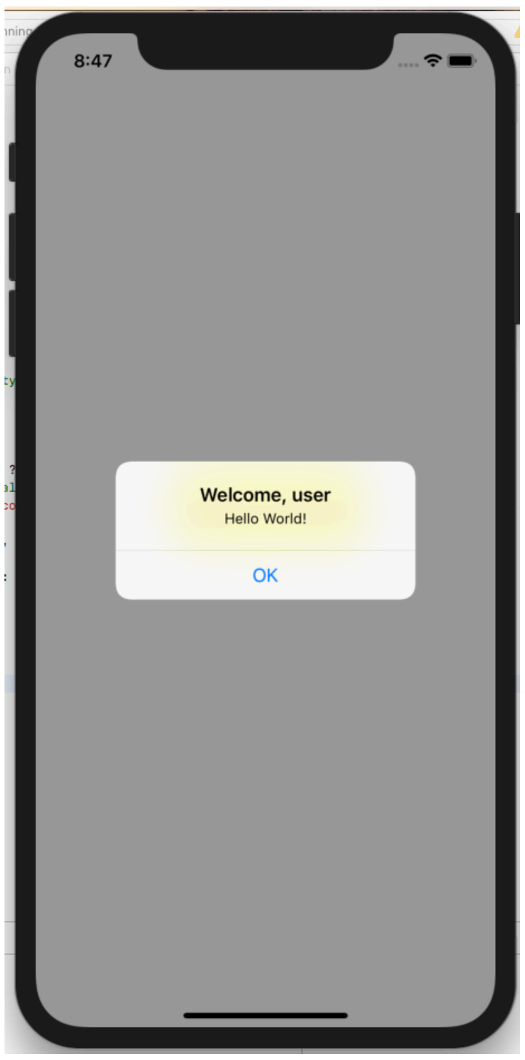
Instead of displaying `"Welcome to my hello world app"` as the title of alertController, we can inject a variable in a string using `\( )` syntax. Change the title string to `"Welcome, \(greetingMessage)"` like this:

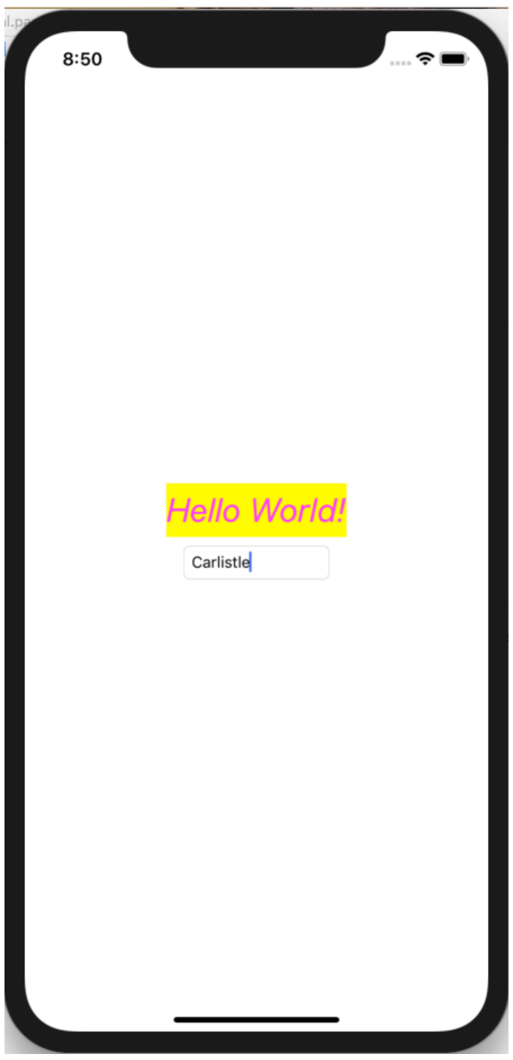
```
        // Create an alert controller object to display an alert
        let alertController = UIAlertController(title: "Welcome, \
(greetingMessage)", message: "Hello World!", preferredStyle: .alert)
```

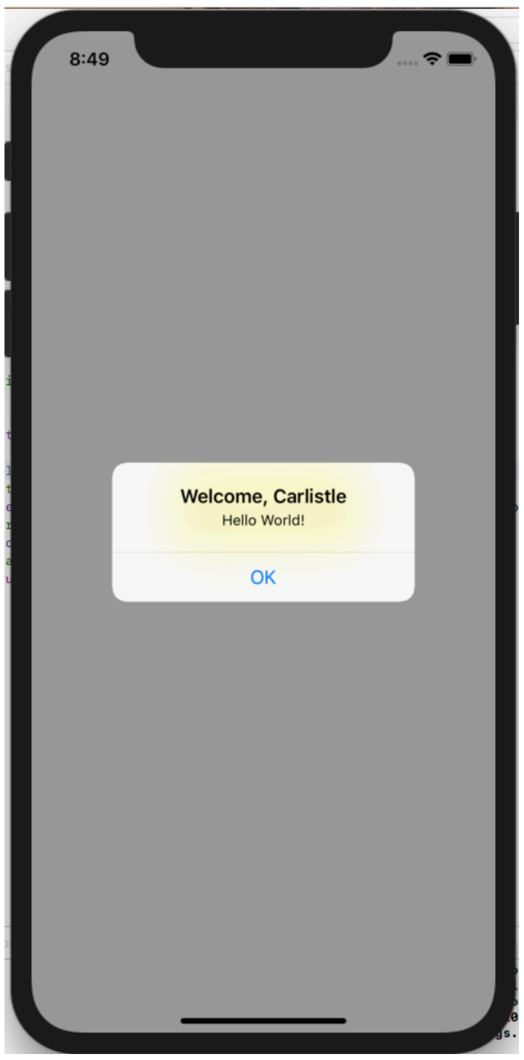
After the modifications, your **ViewController.swift** should look like this:

```
//
//  ViewController.swift
//  HelloWorld
//
//  Created by Carlistle ZHENG on 2/27/19.
//  Copyright © 2019 Carlistle ZHENG. All rights reserved.
//

import UIKit

class ViewController: UIViewController {

    var username: String?
    @IBOutlet weak var userNameTextField: UITextField!

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

    @IBAction func showMessage(_ sender: UIButton) {
        // Unwrapping the optional user name
        let greetingMessage = userNameTextField.text != nil ? userNameTextField.text! :
“user"
        // Create an alert controller object to display an alert
        let alertController = UIAlertController(title: "Welcome, \(greetingMessage)",
message: "Hello World!", preferredStyle: .alert)
        // Add an action OK to the alert controller
        alertController.addAction(UIAlertAction(title: "OK", style: .default, handler:
nil))
        // Present the alert controller with animation
        present(alertController, animated: true, completion: nil)
    }


}
```

Click build and run to see your modifications, make sure to test both cases: with and without a user name input.

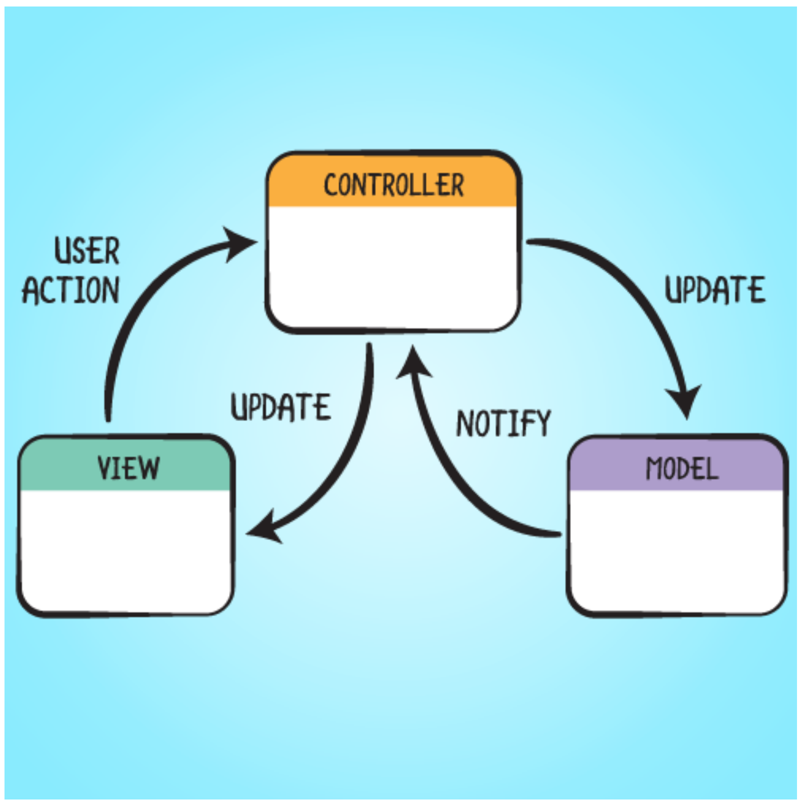






V. FURTHER UNDERSTANDING ON MVC

Object-oriented programs benefit in several ways by adapting the MVC design pattern for their designs. Many objects in these programs tend to be more reusable and their interfaces tend to be better defined. The programs overall are more adaptable to changing requirements—in other words, they are more easily extensible than programs that are not based on MVC. Moreover, many technologies and architectures in Cocoa—such as bindings, the document architecture, and scriptability—are based on MVC and require that your custom objects play one of the roles defined by MVC.



Model Objects Encapsulate Data and Basic Behaviors

Model objects represent special knowledge and expertise. They hold an application's data and define the logic that manipulates that data. A well-designed MVC application has all its important data encapsulated in model objects. Any data that is part of the persistent state of the application (whether that persistent state is stored in files or databases) should reside in the model objects once the data is loaded into the application. Because they represent knowledge and expertise related to a specific problem domain, they tend to be reusable.

Ideally, a model object has no explicit connection to the user interface used to present and edit it. For example, if you have a model object that represents a person (say you are writing an address book), you might want to store a birthdate. That's a good thing to store in your Person model object. However, storing a date format string or other information on how that date is to be presented is probably better off somewhere else.

In practice, this separation is not always the best thing, and there is some room for flexibility here, but in general a model object should not be concerned with interface and presentation issues. One example where a bit of an exception is reasonable is a drawing application that has model objects that represent the graphics displayed. It makes sense for the graphic objects to know how to draw themselves because the main reason for their existence is to define a visual thing. But even in this case, the graphic objects should not rely on living in a particular view or any view at all, and they should not be in charge of knowing when to draw themselves. They should be asked to draw themselves by the view object that wants to present them.

View Objects Present Information to the User

A view object knows how to display, and might allow users to edit, the data from the application's model. The view should not be responsible for storing the data it is displaying. (This does not mean the view never actually stores data it's displaying, of course. A view can cache data or do similar tricks for performance reasons). A view object can be in charge of displaying just one part of a model object, or a whole model object, or even many different model objects. Views come in many different varieties.

View objects tend to be reusable and configurable, and they provide consistency between applications. In Cocoa, the AppKit framework defines a large number of view objects and provides many of them in the Interface Builder library. By reusing the AppKit's view objects, such as `NSButton` objects, you guarantee that buttons in your application behave just like buttons in any other Cocoa application, assuring a high level of consistency in appearance and behavior across applications.

A view should ensure it is displaying the model correctly. Consequently, it usually needs to know about changes to the model. Because model objects should not be tied to specific view objects, they need a generic way of indicating that they have changed.

Controller Objects Tie the Model to the View

A controller object acts as the intermediary between the application's view objects and its model objects. Controllers are often in charge of making sure the views have access to the model objects they need to display and act as the conduit through which views learn about changes to the model. Controller objects can also perform set-up and coordinating tasks for an application and manage the life cycles of other objects.

In a typical Cocoa MVC design, when users enter a value or indicate a choice through a view object, that value or choice is communicated to a controller object. The controller object might interpret the user input in some application-specific way and then either may tell a model object what to do with this input—for example, "add a new value" or "delete the current record"—or it may have the model object reflect a changed value in one of its properties. Based on this same user input, some controller objects might also tell a view object to change an aspect of its appearance or behavior, such as telling a button to disable itself. Conversely, when a model object changes—say, a new data source is accessed—the model object usually communicates that change to a controller object, which then requests one or more view objects to update themselves accordingly.

Controller objects can be either reusable or nonreusable, depending on their general type. Types of Cocoa Controller Objects describes the different types of controller objects in Cocoa.

SUMMARY

In this chapter, we have worked on these topics:

- Concept of Separation of Concerns
- Design pattern MVC
- Functions of Models in MVC
- Functions of Views in MVC
- Functions of Models in MVC
- Implementation of Models in MVC
- Implementation of Views in MVC
- Implementation of Models in MVC

If any of these is unclear to you, please make sure to go back and read the related part or parts before moving on the the next chapter.

For reference, you can download the complete Xcode project from

https://github.com/CarlistleZ/MyiOSTutorial/blob/master/ChapterSix_HelloWorld.zip