

iOS SWIFT 3

Welcome to this course. The idea is to learn the fundamentals concepts that you need when you develop an app in iOS.

About Me

My name is Marlon David Ruiz. I am from Colombia. I've worked as iOS developer since 2013. I've been professor at Universidad de Medellín - Colombia.

Actually I am Projects Director at Integ.ro and I am working as Freelance in iOS projects.

You can contact me through the email (mrui723@gmail.com)

About Apple Inc

Apple is an American company headquartered in Cupertino, California that designs, develops and sells consumer electronics, computer software and online services. The company's hardware products are: iPod, iPhone, iPad, Mac Personal computer, Apple watch, Apple TV and the HomePod smart speaker. In software offers macOS, iOS operating systems, iTunes media Player, the safari web browser and the iLife and iWork creativity and productivity suites. In online services include iTunes Store, the iOS App Store and Mac App Store, Apple music and iCloud.

The company was founded by Steve Jobs, Steve Wozniak and Ronald Wayne in April 1976 to develop and sell personal computers.

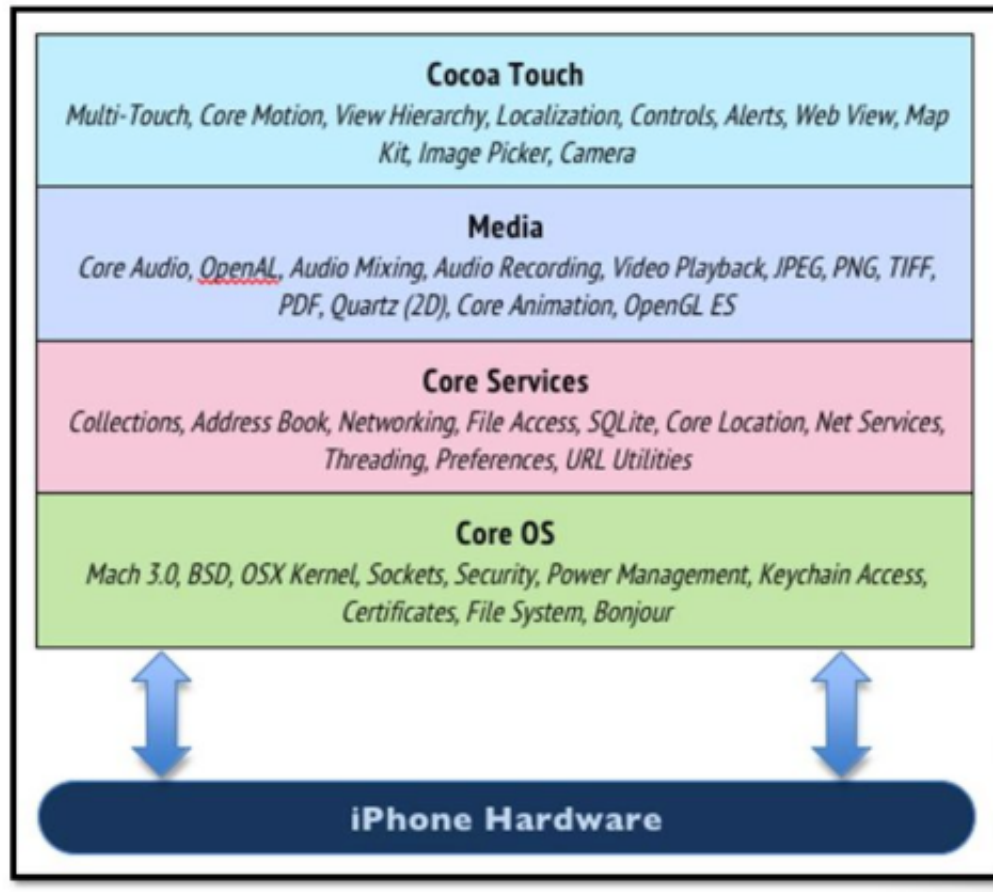
iOS

Operating system derived from Mac OSX. Based on Darwin BSD. It's a Unix certificate.

First release of iOS was on June 26th 2009

First iPhone was 2007 and It worked with iPhone OS

iOS operating system layers



There are two programming languages for developing native apps for iOS: Objective C and Swift.

Objective C

Objective C is a superset of C

1980 Brad J. Cox, based on SmallTalk

1988 NeXTSTEP offers a license for Objective C

2007 There is a big modification of Objective C

Swift

Chris Lattner started the development in 2010

Apple release the first version in the WWDC 2014

Swift works with Cocoa and Cocoa Touch

Swift can work with Objective C

Swift becomes an Open source code in 2015

Tools

Xcode is the IDE

Create an apple ID

Create an apple developer account

First Steps ([MyPlayground.playground](#))

We are going to create a playground. Open Xcode - File - New - Playground - Next - Create.

Playground allows us to learn and explore coding in Swift.

Please review the file `MyPlayground.playground` where you find:

1. Value type (Enumerations, structures or tuples - Array, String and Dictionary) is that copying — the effect of assignment, initialization, and argument passing — creates an *independent instance* with its own unique copy of its data.
2. Reference type (Class). implicitly creates a shared instance. After a copy, two variables then refer to a single instance of the data, so modifying data in the second variable also affects the original
3. Constants and Variables. Constant and variable names can't contain whitespace characters, mathematical symbols, arrows, private-use (or invalid) Unicode code points, or line- and box-drawing characters. Nor can they begin with a number, although numbers may be included elsewhere within the name.
4. Comments
5. Integers (8, 16, 32, 64 bits), Floating-Point Numbers (Float and Double), Boolean (Swift has a basic *Boolean* type, called `Bool`)
6. Typealiases. Define an alternative name for an existing type. You define type aliases with the `typealias` keyword.
7. Tuples. Group multiple values into a single compound value. The values within a tuple can be of any type and don't have to be of the same type as each other.
8. Optionals, If Statements and Forced Unwrapping, Optional Binding and Implicitly Unwrapped Optionals.
9. Error Handling (During execution). Error handling allows you to determine the underlying cause of failure, and, if necessary, propagate the error to another part of your program.
10. Assertions and Preconditions. *Assertions* and *preconditions* are checks that happen at runtime. You use them to make sure an essential condition is satisfied before executing any further code.
11. Basic Operators. An *operator* is a special symbol or phrase that you use to check, change, or combine values.

Hello World ([HelloWorld](#))

We are going to create our first iOS app.

1. Open Xcode

2. File -> New -> Project -> Single View Application.

Product Name: HelloWorld

Your new product's bundle identifier

Organization Name: Eafit

Organization Identifier: co.edu.eafit

Bundle Identifier: co.edu.eafit.HelloWorld

Language: Swift

Devices: iPhone

☐ Use Core Data

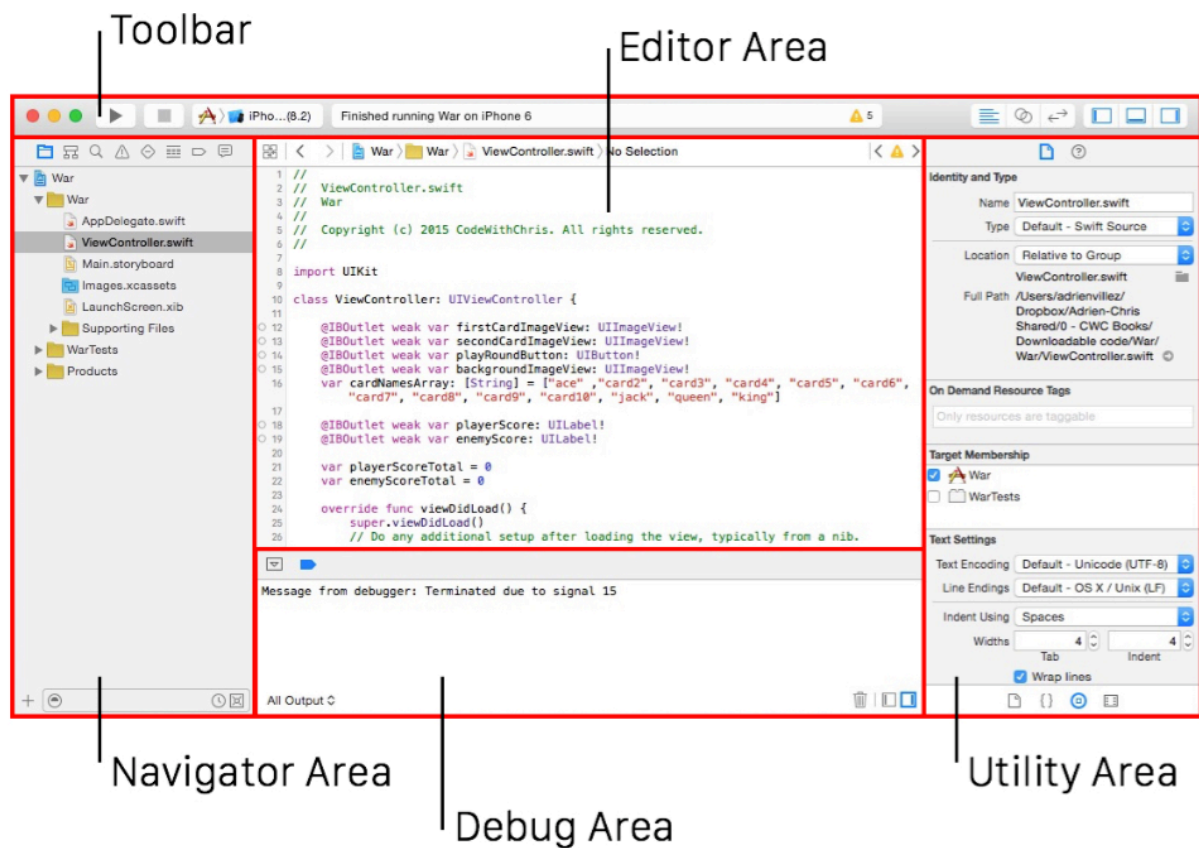
☐ Include Unit Tests

☐ Include UI Tests

3. See image:

4. Finding the place where you want to save the project.

OK We have created our Hello World project. Now it's time to understand the interface that Xcode offers us:

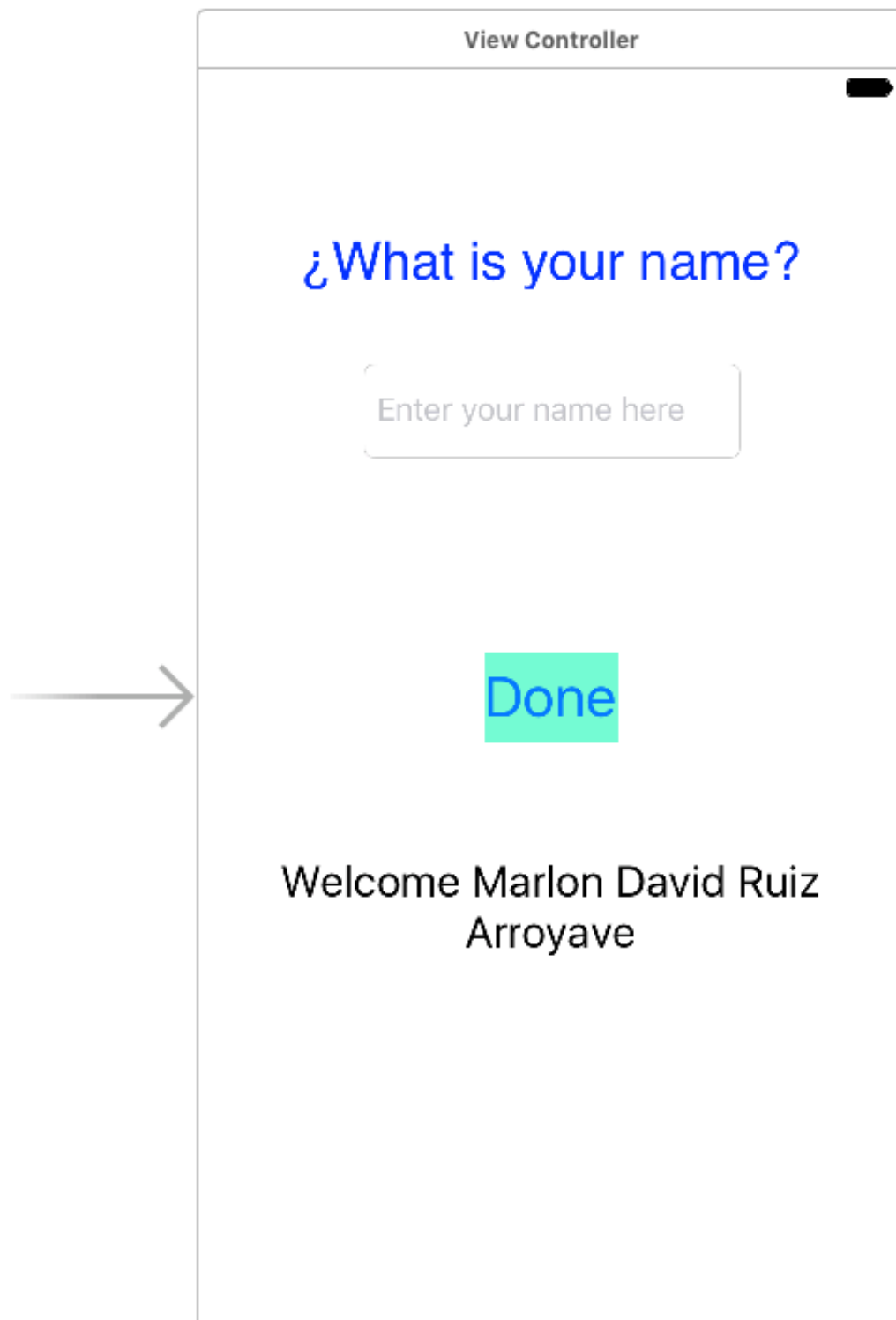


As you can see from the diagram, there are 4 major areas: the Navigator, Editor, Debug Area and Utility Area.

You can also show and hide the various areas as needed via the “View” buttons in the upper right hand corner:



Getting back to our HelloWorld project. Let's go to Main.storyboard and we start to play with some views:



After add these views is time to connect with our controller:

1. IBOutlets:

```
class ViewController: UIViewController {  
  
@IBOutlet weak var nameTextField: UITextField!
```

```
@IBOutlet weak var welcomeLabel: UILabel!
```

1. IBActions:

```
@IBAction func sendName(_ sender: Any) {  
    }  
}
```

When the user enter some information in the nameTextField and the user does tap over the button then welcomeLabel will appear with the message "Welcome" + the value of the TextField.

If the user does tap in the button and the TextField is empty then we need to show an alert with the title error and the message "Debes digitar tu nombre".

The result is the following:

```
@IBAction func sendName(_ sender: Any) {  
  
    if (nameTextField.text?.characters.count)! > 0 {  
  
        welcomeLabel.text = "Welcome \(nameTextField.text!)!!!"  
  
        welcomeLabel.isHidden = false  
  
    }else {  
  
        welcomeLabel.isHidden = true  
  
        let alertController = UIAlertController(title: "Error", message: "Debes digitar tu nombre", preferredStyle: .alert)  
  
        let okAction = UIAlertAction(title: "OK", style: .default, handler: nil)  
  
        alertController.addAction(okAction)  
  
        present(alertController, animated: true, completion: nil)  
  
    }  
  
    view.endEditing(true) //Hide the keyboard  
  
}
```

We have learned:

1. How to create a project in iOS.
2. How to declare IBOutlets and IBActions.
3. How to create an UIAlertController with UIAlertAction.
4. How to validate if the TextField is empty.
5. how to add views to Main.storyboard

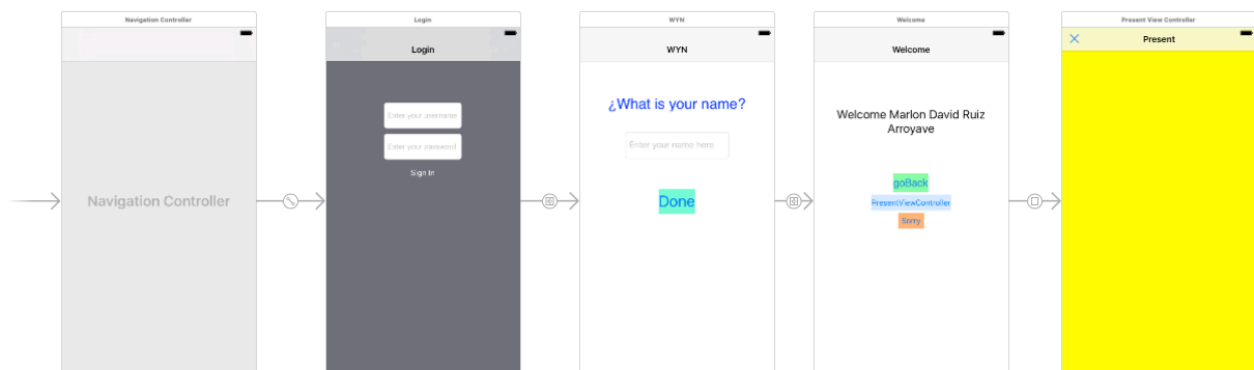
PresentNewViewController([PresentNewViewController](#))

We are going to create our second iOS app called PresentANewViewController.

When you think to create an app you should have:

1. *Wireframes. The best way that we have for consolidating our ideation. The intention is to know what the screen does, and not exactly what it looks like. You can do it with a piece of paper or with an application.*
2. *Mockups. A mockup is a prototype of our app with the color and fonts that it should have.*
3. *When you have the mockup of your app it's time to start coding. The order should be:*
 - 3.1 *Create the screen in the Main.storyboard.*
 - 3.2 *Create the controller (UIViewController, TableViewController, etc) that handles the view (screen)*
 - 3.3 *Connect the views that you need to handle in the controller (IBoutlets)*
 - 3.4 *Connect the views that handle events with your controller (IBActions, Delegates, DataSources)*
 - 3.5 *Develop the logic for each event that your screen should handle.*
 - 3.6 *Test the screen with the simulator or real device. If all is ok you can continue with the next screen doing the points 3.1 to 3.6 again. But if you find an error, you need to fix it before continuing with the next screen (this is not mandatory).*

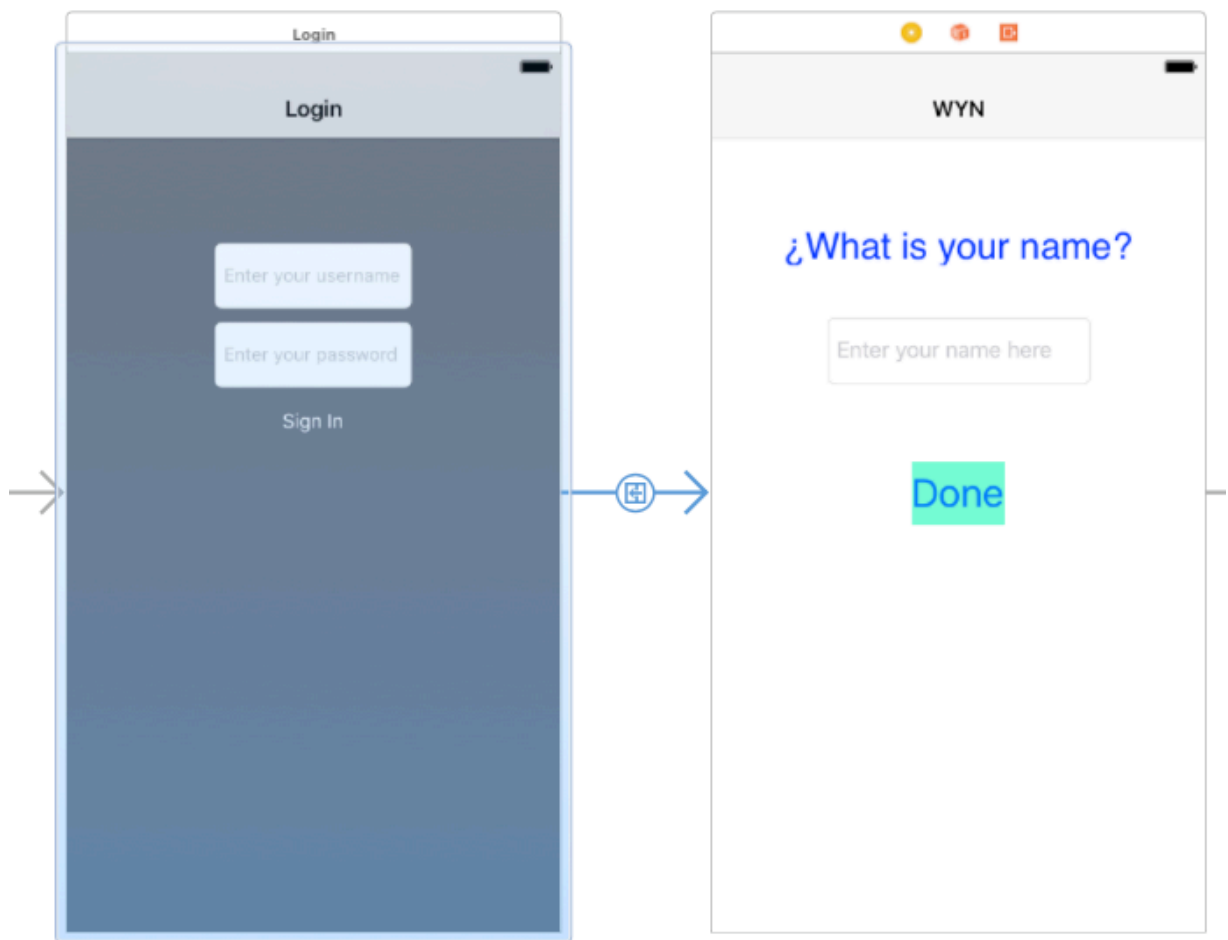
For this app the storyboard is the following:



We have the following screens:

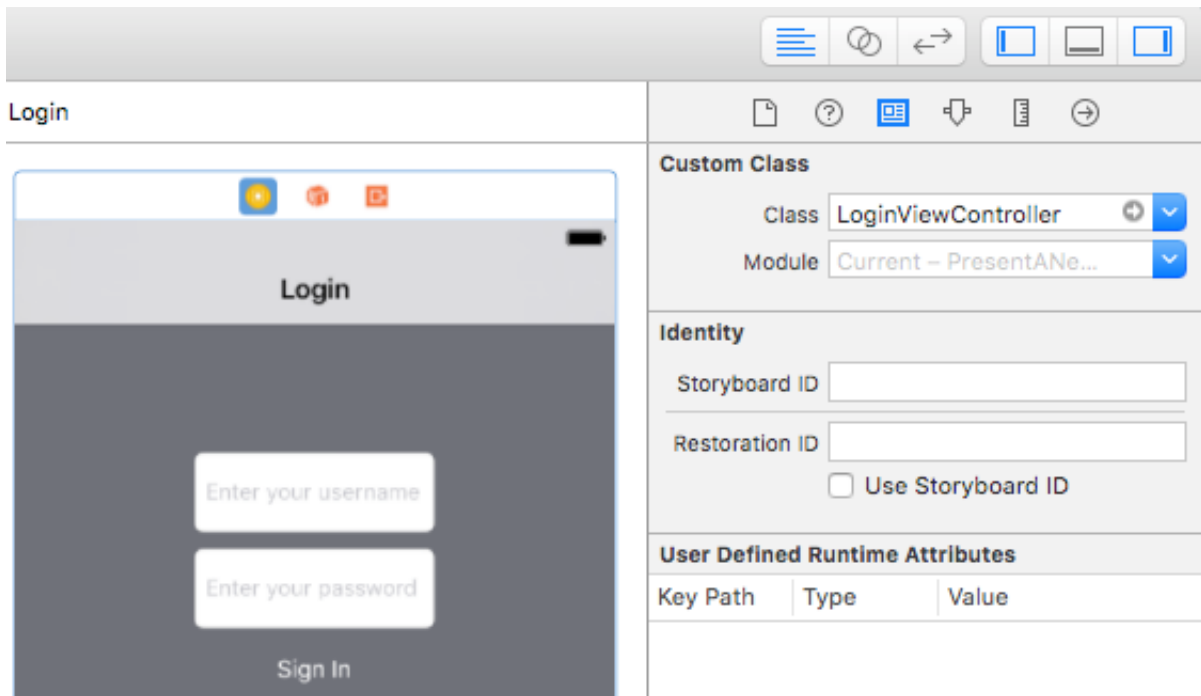
1. Login Scene. We have 2 TextFields and one button. if the user enters the correct username and password then the app should display the following screen (WYN).

The connection between Login and WYN is through a segue and the kind is Show. The Login screen is embedded in a navigation controller and this one allows us to stack the screen when the connection kind is Show.



This segue is from scene to scene and requires an identifier that in our code will be called programmatically.

We create a controller that inherits from `UIViewController`. We select the Login screen and in the identity inspector, we put `LoginViewController` in the class property field.



Screen, ViewController, TableViewController and Scene have the same meaning when someone wants to refer to a scene in the Main.storyboard.

We need to connect the TextFields as IBOutlet and the Button as IBAction.

```
// MARK: - IBOutlet
```

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

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

We create two properties:

```
// MARK: - Properties
```

```
let username = "mruiz"
```

```
let password = "mruiz"
```

The value for each property can be whatever thing. These are used for comparing the two text fields.

The implementation in our IBAction Button is to validate the text fields aren't empty and they are compared with the corresponding property. If all are OK then we invoke the next segue, calling to the function `performSegueWithIdentifier:sender`, we send the `wyn` identifier and `sender` in nil.

```
// MARK: - IBActions
```

```
@IBAction func login(_ sender: Any) {
```

```
if !(userTextField.text?.isEmpty)! {
```

```
if !(passwordTextField.text?.isEmpty)! {
```

```

if userTextField.text == username && passwordTextField.text == password {

performSegue(withIdentifier: "wyn", sender: nil)

}

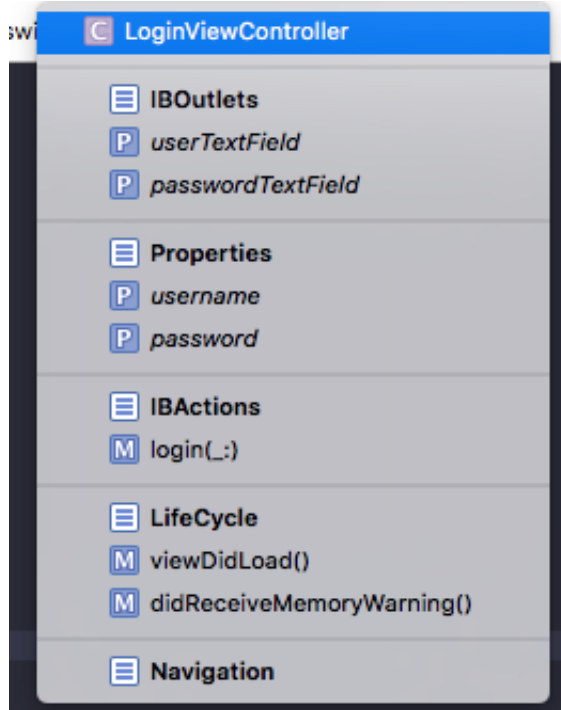
}

}

}

```

The MARK directive allows us to organize our code in groups.

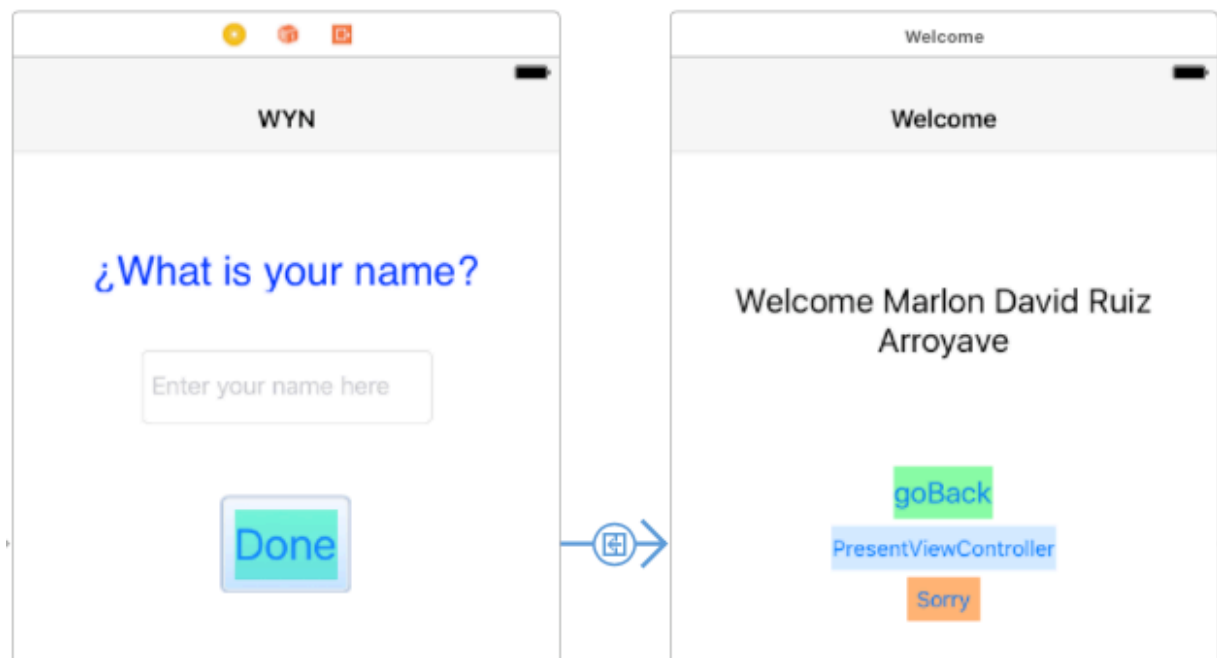


1. WYN Scene. We create a blue label with the name ¿What is your name?, a Text Field and one Button. The button is connected with the Welcome scene (segue with the kind show). The controller for this scene is the ViewController class and we connect the text field as an IBOutlet.

```

@IBOutlet weak var nameTextField: UITextField!

```



This segue is from button to scene.

When the user presses the done button we can control if we want that the app show or not the next scene. The method is called `shouldPerformSegue`.

```
override func shouldPerformSegue(withIdentifier identifier: String, sender: Any?)
-> Bool {

    return !(nameTextField.text?.isEmpty)!

}
```

For our case, if `nameTextField` field is empty the app won't show the next scene.

When we need to pass data between scene we use the `prepare for segue` method.

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {

    if segue.destination is GreetingsViewController {

        let greetingsVC = segue.destination as! GreetingsViewController

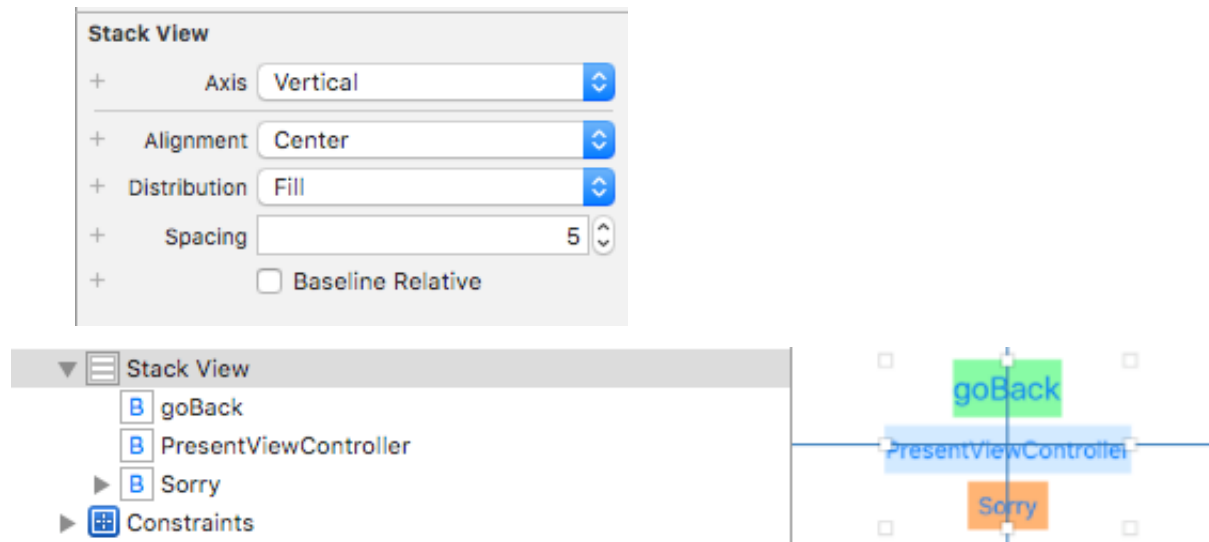
        greetingsVC.names = nameTextField.text!

    }

}
```

The code above validates that the next scene is a `GreetingsViewController` and we get the next scene for setting the `names` property with the value of the `nameTextField` field.

1. Welcome scene. We have a label and three buttons. We are going to use a Stack view for organizing the buttons. When we implement a Stack view we need to define if it is Vertical or Horizontal, alignment, distribution and spacing.



When you use a stack view you don't need to define constraints for every view's element inside of stack view, but we need to define constraints for the stack view.

The controller that handles the scene is GreetingsViewController. We create the IBOutlet for the label control and we declare a names variable with an empty String.

```
// MARK: - IBOutlets
```

```
@IBOutlet weak var welcomeLabel: UILabel!
```

```
// MARK: - Properties
```

```
var names: String = ""
```

We connect the goBack button as IBAction to our controller. When the user presses it the app go back to the before scene.

```
// MARK: - IBActions
```

```
@IBAction func goBack(_ sender: Any) {
```

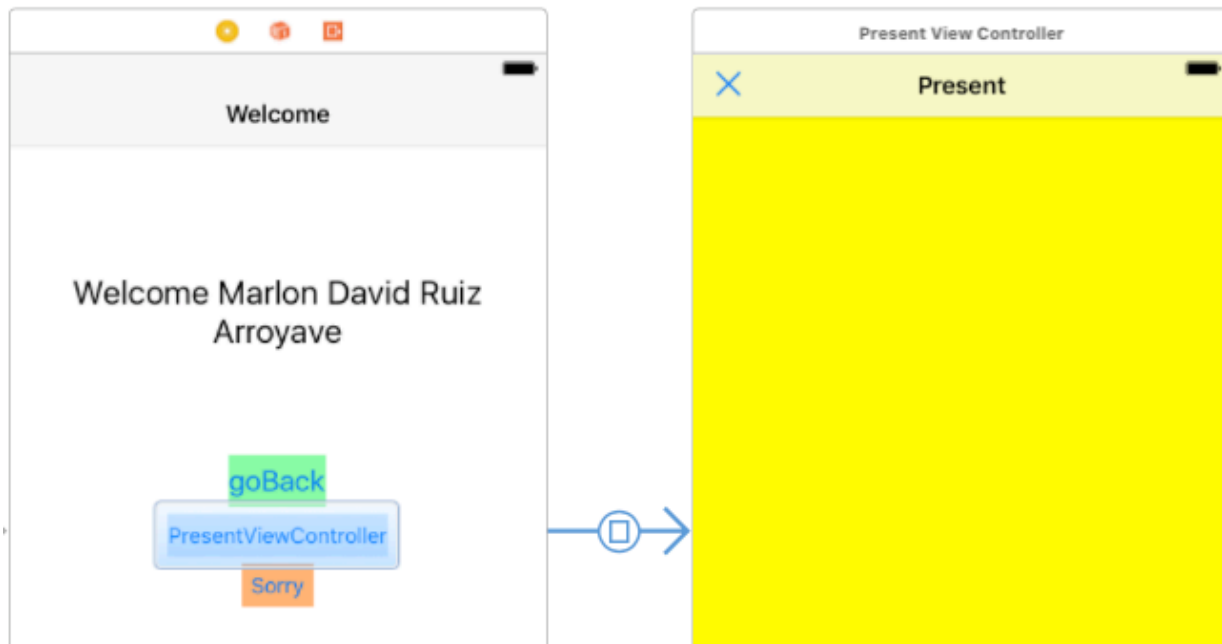
```
//dismiss(animated: true, completion: nil)
```

```
_ = navigationController?.popViewController(animated: true)
```

```
}
```

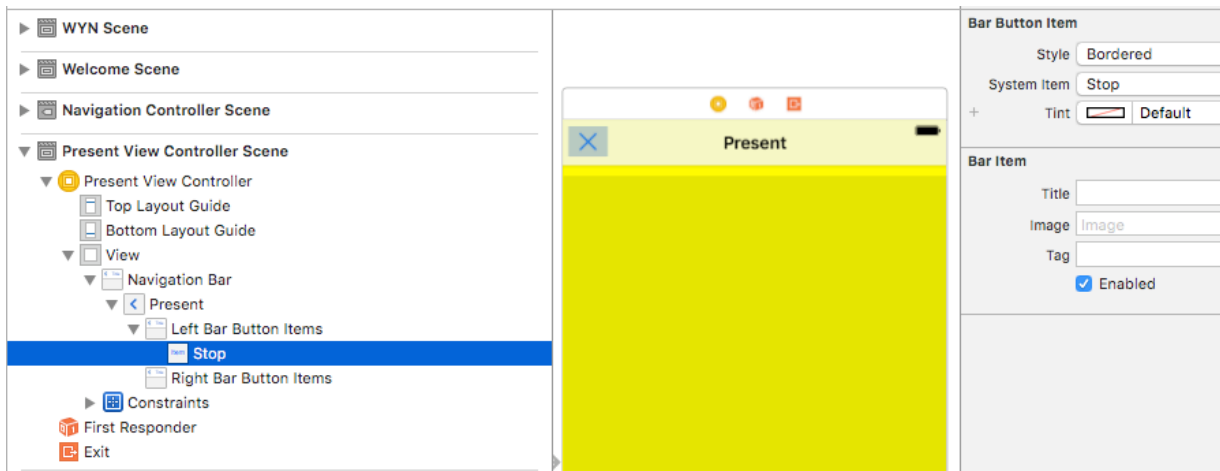
The navigation controller has the popViewController:animated method that remove the current controller.

The PresentViewController button has a connection with the next scene. This segue is the kind Present Modally.



When you present a new controller with the segue kind Present Modally, the new controller isn't in the navigation controller.

1. Present scene. The view has a Yellow background color, we have a navigation bar with a left bar button item and the type system item is Stop.



The controller that handles this scene is the PresentViewController. We connect the stop bar button item as IBAction to our controller. When user presses this button the current controller will disappear.

```
// MARK: - IBActions

@IBAction func dismiss(_ sender: Any) {

dismiss(animated: false, completion: nil)

}
```

When you present a controller as Present Modally you need to use the dismiss method for removing it.

We have learned:

1. How to Connect two scenes and Calling a segue programmatically.
2. How to use the MARK directive for organizing the code.
3. How to use the Stack View.
4. Kinds of segues: Show and Present Modally.
5. How to remove the current controller, depending on the way that controller has been presented.

Concepts

MVC

The Model-View-Controller (MVC) design pattern assigns objects in an application one of three roles: model, view, or controller.

Model Objects

Model objects encapsulate the data specific to an application and define the logic and computation that manipulate and process that data. When a model object changes (for example, new data is received over a network connection), it notifies a controller object, which updates the appropriate view objects. The model can communicate with the controller through the delegate pattern or notifications. The model never needs to communicate with the view objects.

View Objects

A major purpose of view objects is to display data from the application's model objects and to enable the editing of that data. Despite this, view objects are typically decoupled from model objects in an MVC application. View objects learn about changes in model data through the application's controller objects. The view can communicate with the controller through Target-Action pattern (Button) or delegate pattern.

Controller Objects

A controller object acts as an intermediary between one or more of an application's view objects and one or more of its model objects. A controller object interprets user actions made in view objects and communicates new or changed data to the model layer. When model objects change, a controller object communicates that new model data to the view objects so that they can display it.

Delegate pattern

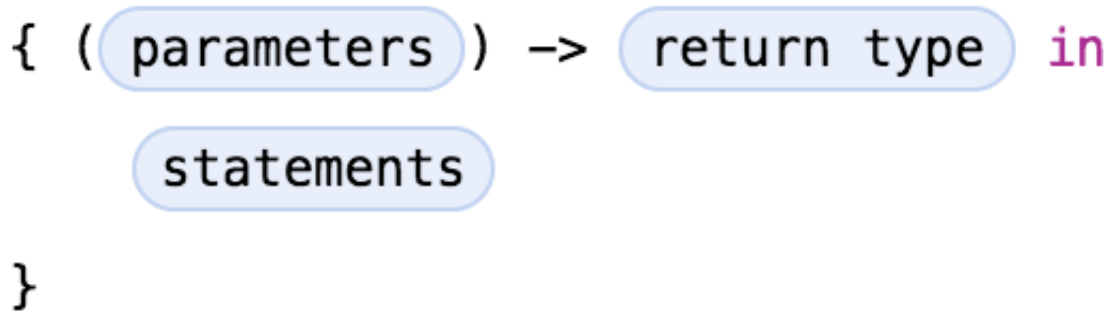
The idea behind delegates is that instead of class A executing some code, it tells its delegate to execute that code. The delegate is tied to another class (let's call it class B). In order to facilitate this, class A creates something called a protocol. This protocol has a list of methods in it (with no definition). Class A then has an instance of the protocol as a property. Class B has to implement the protocol defined in class A. Lastly, when class A is created, its delegate property is set to class B.

Closures

Closures are self-contained blocks of functionality that can be passed around and used in your code. Closures in Swift are similar to blocks in C and Objective-C and to lambdas in other programming languages.

Closures can capture and store references to any constants and variables from the context in which they are defined. This is known as *closing over* those constants and variables. Swift handles all of the memory management of capturing for you.

(Apple definition)



The diagram illustrates the syntax of a Swift closure. It consists of an opening curly brace followed by a pair of parentheses containing the word "parameters". This is followed by a right-pointing arrow and the words "return type". The word "in" is written in a pink color. Below this line is a light blue rounded rectangle containing the word "statements". The entire structure is enclosed in a closing curly brace.

Image from Apple

Escaping Closures

A closure is said to *escape* a function when the closure is passed as an argument to the function, but is called after the function returns. Marking a closure with `@escaping` means you have to refer to `self` explicitly within the closure.

(Apple definition)

Recommended Links

- Try git <https://try.github.io/levels/1/challenges/1>
- https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/index.html
- <https://www.raywenderlich.com>
- <https://www.appcoda.com>