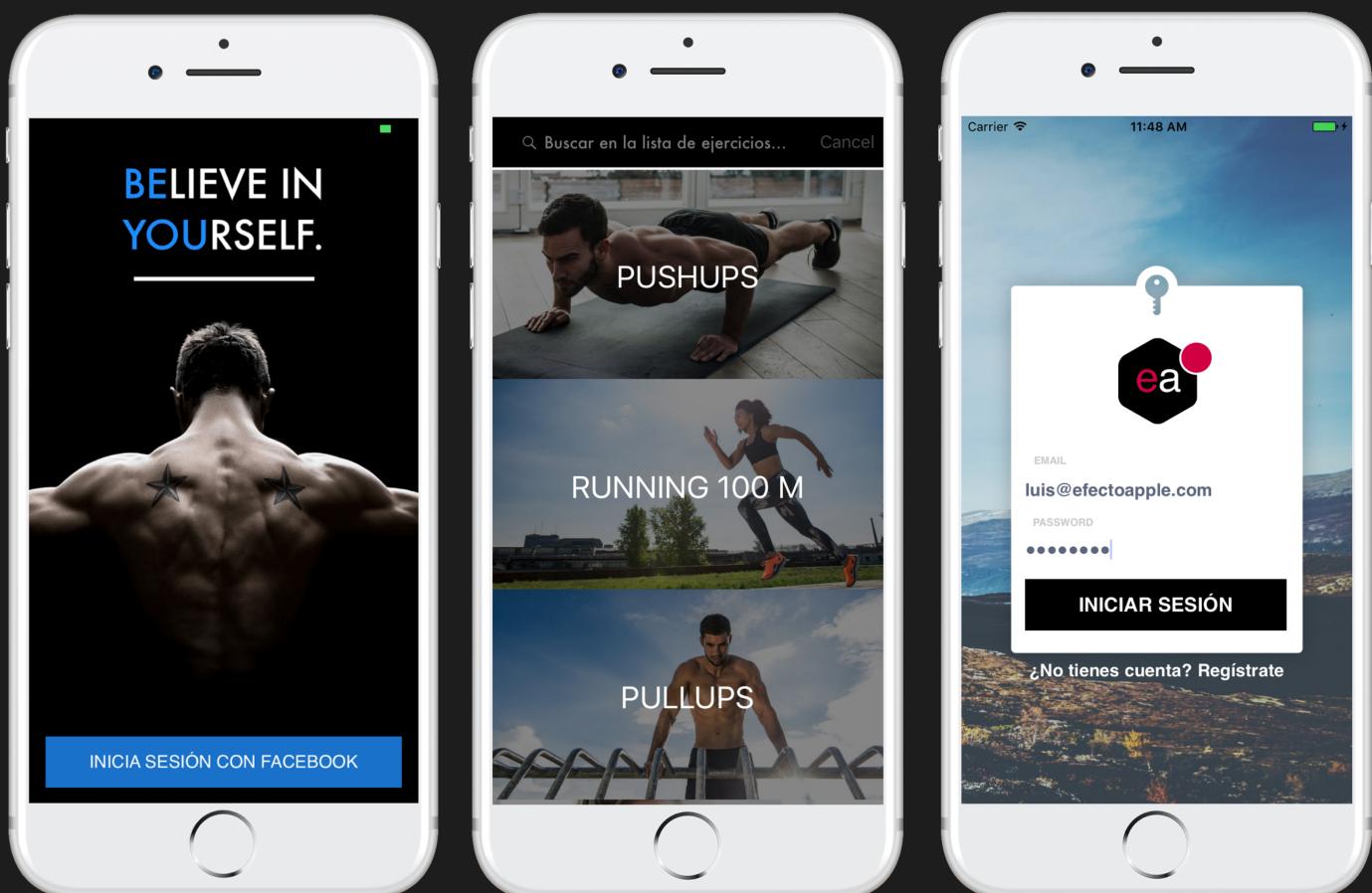




Incluye
CÓDIGO FUENTE

APRENDE DESARROLLO iOS



- Tutoriales PASO a PASO -

CREANDO 10 APLICACIONES

Por Luis Rollón.

Creador de www.efectoapple.com

Índice

Módulo 1: Introducción	4
Bienvenida	5
¿Este libro es para ti?	7
Requisitos para seguir los Tutoriales	8
Módulo 2: Tutoriales	9
Aplicación 1: Introducción a TableView	11
Aplicación 2: Aplicación con Menú Lateral	31
Aplicación 3: Trabajando con StackView	50
Aplicación 4: Introducción a Core Data	69
Aplicación 5: Creando Aplicación Multivista	89
Aplicación 6: Pantalla de Login con Email	152
Aplicación 7: Como trabajar con CollectionView	175
Aplicación 8: Utilizando Celdas Personalizadas	203
Aplicación 9: El Framework UserNotifications	231
Aplicación 10: Pantalla de Login con Facebook	265
Módulo 3: Código Fuente	298
Descarga de Código Fuente y Despedida	299



Módulo 1

Introducción

¡Hola y bienvenido al mundo de las Apps!

Estoy **encantado** de volver a dirigirme a ti (si ya nos conocemos) o de saludarte por primera vez, en caso de que este sea nuestro primer contacto.

Y sobre todo estoy encantado de poder hacerlo mediante este nuevo **ebook**.

Un ebook enfocado en el **Desarrollo de Aplicaciones iOS**.

Muchas personas piensan que la revolución de las aplicaciones móviles (Coloquialmente conocidas como Apps) llegó con el lanzamiento del primer iPhone.

Sin embargo, esto no es cierto.

El inicio de esta revolución vino con otro “invento” desarrollado por Apple:

La App Store.

El **10 de Julio de 2008**, se produjo la inauguración de esta tienda de aplicaciones.

A partir de ahí, todo un nuevo ecosistema se fue desarrollando a una velocidad de vértigo.

El impacto que produjo en los siguientes años fue de una magnitud enorme.

Actualmente hay cerca de **2 millones** de aplicaciones a la venta.

Solo el año pasado, los desarrolladores iOS obtuvieron más de **20.000 millones de dólares** por la venta de sus aplicaciones.

Actualmente el sueldo de un Desarrollador iOS con un par de años de experiencia está cerca de los **35.000 euros** y puede llegar a los **70.000**

euros en función de la experiencia del programador.

En Silicon Valley su salario se multiplica, pudiendo llegar en algunos casos hasta los **225.000 dólares** al año.

Esta es la situación actual.

La App Store ha eliminado las barreras de entrada a un mercado de venta de aplicaciones, que cada año que pasa mejora ampliamente las cifras el ejercicio anterior.

Hoy en día **cualquier persona** con algo de esfuerzo y conocimientos muy básicos de programación **puede crear su propia aplicación**.

En este libro, voy a intentar enfocar este aprendizaje de forma totalmente práctica, para que mientras desarrollas **10 aplicaciones iOS diferentes**, puedas ir adquiriendo conocimientos fundamentales que te servirán en el futuro.

Veamos primero, si estás en el sitio adecuado.

¿Este libro es para ti?

Odio perder el tiempo.

Tampoco me gusta hacer perder el tiempo a nadie.

Por este motivo, antes de comenzar, me gustaría responder a dos de tus preguntas:

¿Estoy realmente en el sitio correcto?

¿Hay algo de interés para mí?

Es muy sencillo, tienes entre tus manos el libro adecuado, si:

- Te interesa el Desarrollo de Aplicaciones iOS
- Tienes una idea para una app pero no sabes como llevarla a cabo
- Estás cansado de libros teóricos, donde no consigues avanzar
- Siempre has tenido la ilusión de crear tu propia aplicación
- Eres programador pero nunca antes has programado para la plataforma de Apple
- Te dedicas a otra cosa pero siempre te ha interesado la programación

Si te sientes identificado con alguna de estas situaciones, este libro es para ti.

La formación a la que vas a tener acceso te va a enseñar, paso a paso, como desarrollar 10 aplicaciones diferentes, a través de las cuales revisarás los conceptos fundamentales del Desarrollo iOS.

Además, al final del libro, podrás acceder a la descarga del código fuente completo de las 10 aplicaciones con las que vamos a trabajar.

Una vez que sabes, que este libro es para ti, veamos que necesitas para poder aprovecharlo al máximo.

Requisitos para poder seguir los Tutoriales

Antes de comenzar con este libro, es importante saber que necesitas para poder seguir los tutoriales que veremos aquí.

Recomiendo disponer de un **Mac** para poder trabajar de la mejor forma posible.

En el caso de que no dispongas de uno y no puedas adquirir uno ahora mismo, aquí tienes una **opción alternativa**.

Por menos de 1 € la hora ó 19 € al mes podrás utilizar un Mac compartido a través de internet. No hablamos de máquinas virtuales sino ordenadores Mac con Intel Core i5 y entre 8GB y 16GB de RAM.

Unicamente necesitas un navegador web para acceder a la plataforma y una conexión a internet.

Este servicio te ofrece **todo lo necesario** para el desarrollo de aplicaciones iOS. Es una opción perfectamente válida para quien no tiene ordenador Mac y no quiere hacer un desembolso grande pero busca comenzar con la Programación iOS.

Puedes echar un vistazo y comprobarlo por ti mismo aquí: [Mac in Cloud](#).

Además de disponer de un Mac o utilizar este servicio, necesitarás instalar **Xcode**.

Xcode es una app que utilizarás para desarrollar apps. Puedes descargarla totalmente gratis [desde aquí](#).

Desde Xcode desarrollarás tanto la interfaz de tu aplicación como todo lo que sucede de forma invisible al usuario. Además dispone de un simulador en el que podrás probar tus apps sin necesidad de disponer un iPhone, un iPad o un Apple Watch.

Xcode como cualquier IDE tiene sus defectos, pero después de probar varios entornos de desarrollo, a mi me parece que funciona muy muy bien.

Y aparte de un Mac y Xcode, lo último que necesitas es **interés y curiosidad**. El desarrollo de apps es un mundo apasionante, pero para avanzar en necesitarás poner también de tu parte.

Esto es todo, como ves, **no necesitas mucho** para comenzar.



Módulo 2

Tutoriales

Tutorial de Introducción a UITableView

COMO UTILIZAR TABLAS EN TUS APLICACIONES iOS

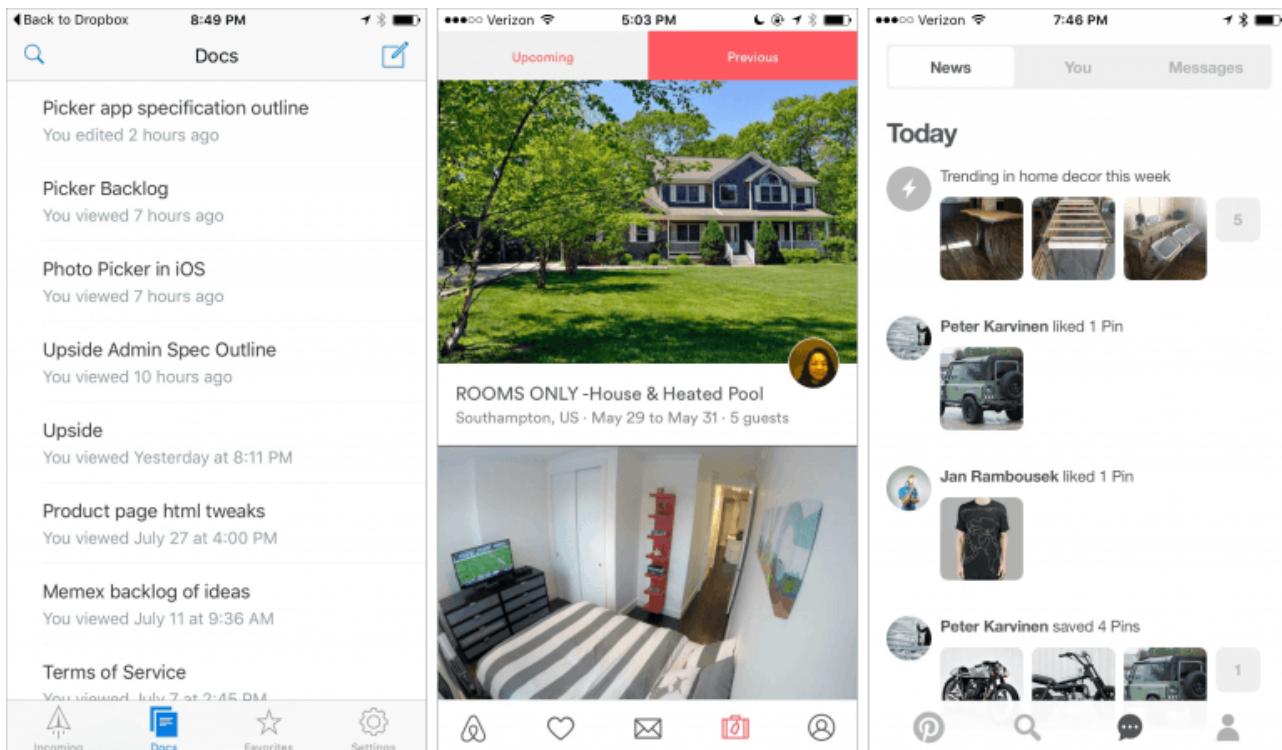
Lenguaje Swift | Nivel Principiante

1. Introducción

Las **Tablas** son un **componente fundamental** en cualquier aplicación iOS.

En este Tutorial vamos a desarrollar una **aplicación** desde cero, que utilizará una UITableView para mostrar información en pantalla.

Antes de comenzar, me gustaría que vieras una imagen, donde aparecen **tres aplicaciones** muy conocidas que utilizan **UITableViews** en sus interfaces:



En esta imagen, aparecen las apps de **Dropbox**, **Airbnb** y **Pinterest**. En todas se utilizan varias UITableViews y como puedes ver, el diseño de las aplicaciones no se parecen en nada. Esto se debe a que las UITableViews nos permiten una **gran flexibilidad**. Podemos desarrollar aplicaciones muy diferentes utilizando únicamente tablas.

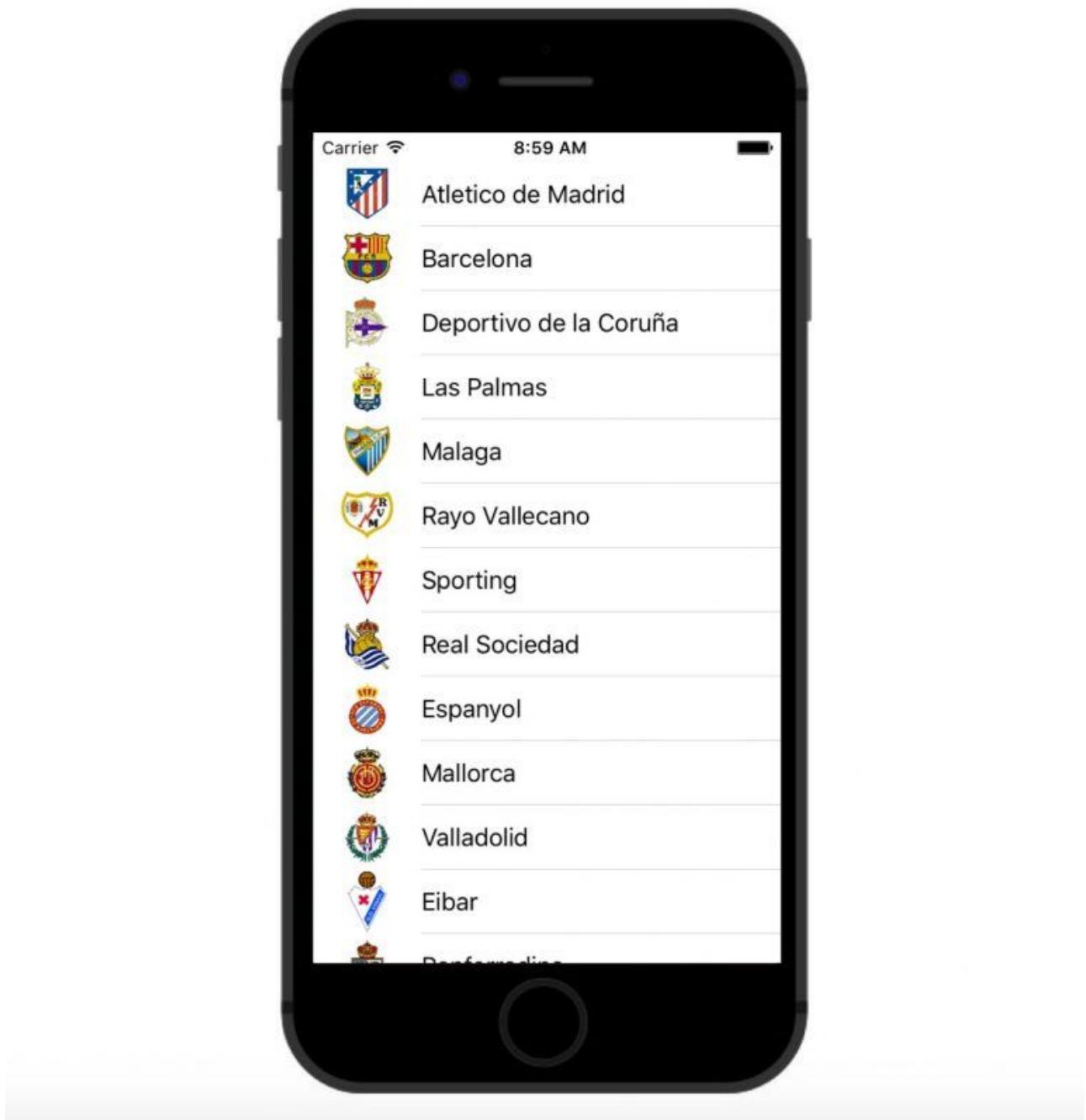
2. ¿Qué vamos a ver en este tutorial?

Aunque desarrollar una aplicación completa utilizando una única UITableView para mostrar la información en pantalla puede parecer algo excesivamente sencillo, estos son todos los temas que vamos a tocar:

- ¿En qué consiste un **Protocolo** en iOS?
- ¿Qué es el protocolo **UITableViewDataSource**?
- ¿Qué es el protocolo **UITableViewDelegate**?
- ¿En qué consisten los **delegados** en iOS?
- ¿Qué es el **NSIndexPath**?
- ¿Cómo funcionan las **UITableViews**?
- ¿Cómo puedo **añadir imágenes** a mis tablas?

Como puedes ver, una sencilla aplicación con una tabla nos permitirá trabajar con conceptos fundamentales del Desarrollo iOS como **Protocolos y Delegados**.

Al terminar el tutorial, nuestra aplicación tendrá este aspecto:



Nota Legal: El material utilizado para la realización de este Tutorial obedece únicamente a propósitos didácticos. EfectoApple reconoce la propiedad de dicho contenido por parte de LaLiga | Liga de Fútbol Profesional.

Como ves, lo que haremos será mostrar una **lista de equipos** de fútbol españoles con sus respectivos escudos.

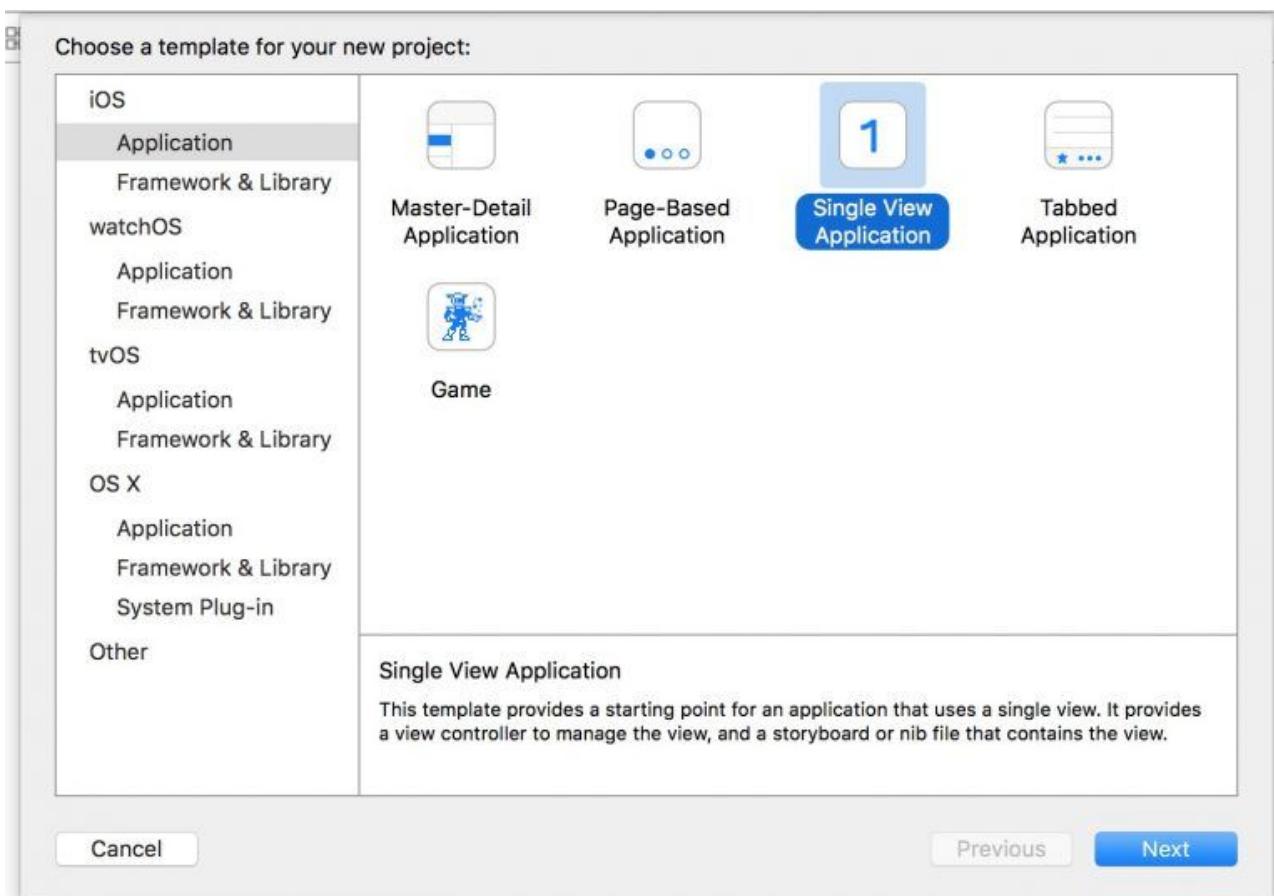
Si realmente tienes interés en aprender como funcionan las aplicaciones iOS, te recomiendo que no te limites a leer este tutorial, sino que abras Xcode y vayas siguiendo cada uno de los pasos hasta que programmes la **aplicación completa**. Es la mejor forma de aprender.

3. Creando nuestro proyecto

Abre Xcode y crea un nuevo proyecto, haciendo clic en la opción “**Create a new Xcode project**”.



Elige la plantilla “**Single View Application**” y haz clic en Next para continuar.

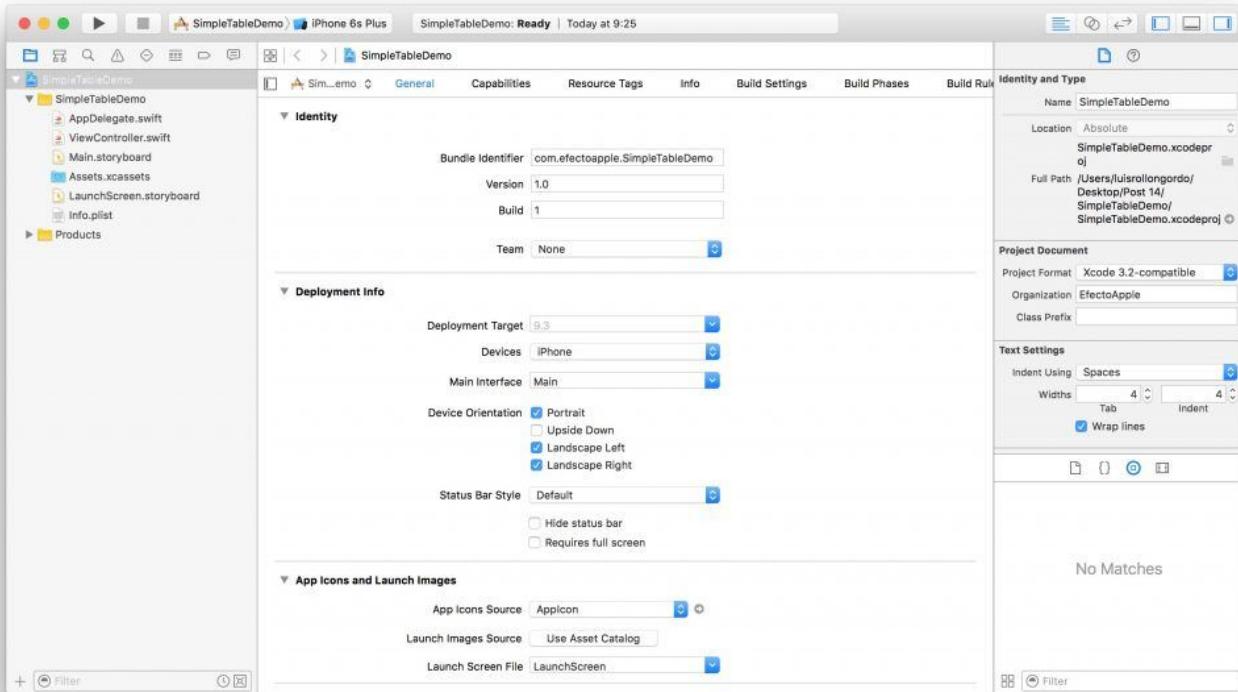


A continuación **rellena los campos** que Xcode te solicita:

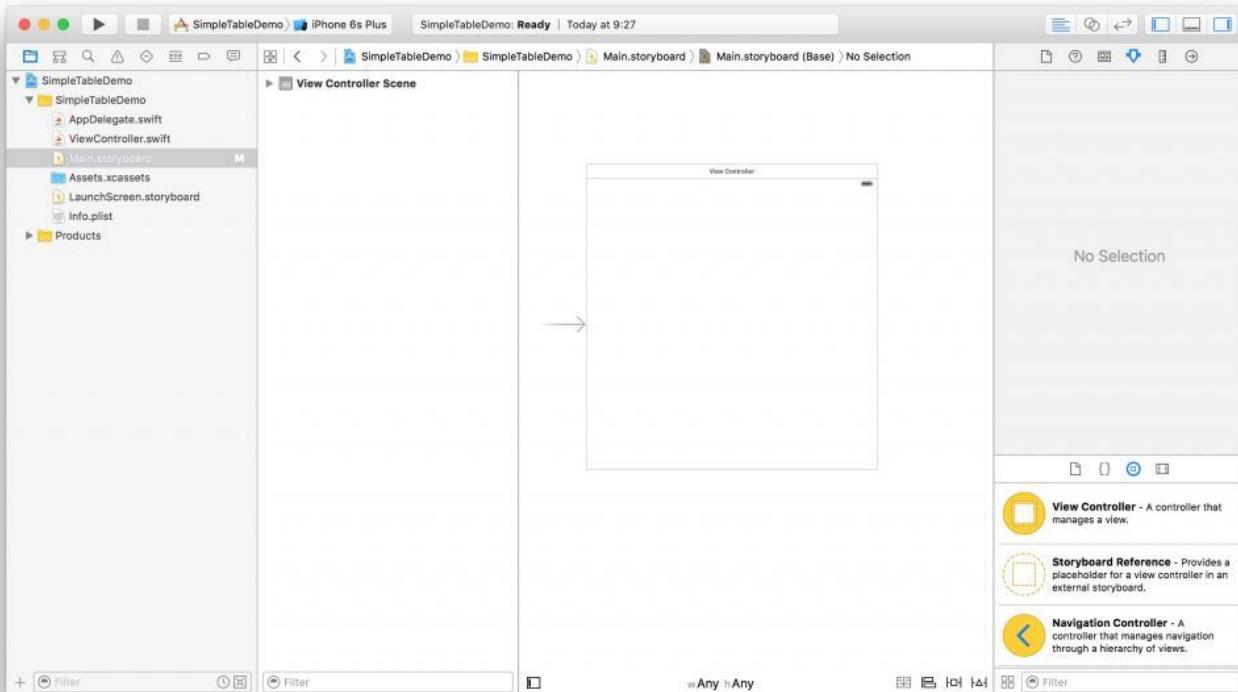
- **Product Name:** Es el nombre de la app, puedes usar el mismo que yo: **SimpleTableDemo**
- **Organization Name:** Yo utilizo EfectoApple, pero tu puedes usar tu nombre por ejemplo.
- **Organization Identifier:** Aquí se suele utilizar la notación de **nombre de dominio inverso**. No es más que si tienes un dominio propio, escribir primero el .com y luego el nombre de tu dominio. Por eso yo utilizo com.efectoapple. Si no tienes ningún dominio propio o prefieres no utilizarlo puedes poner el identificador que quieras.
- **Lenguaje:** Te recomiendo Swift, ya que en este tutorial utilizaremos Swift
- **Devices:** Selecciona iPhone.
- **Use Core Data:** No es necesario que actives esta opción.
- **Include Unit Tests:** No es necesario que actives esta opción.
- **Include UI Tests:** No es necesario que actives esta opción.

Una vez que has introducido esta información, pulsa en **Next** y Xcode te

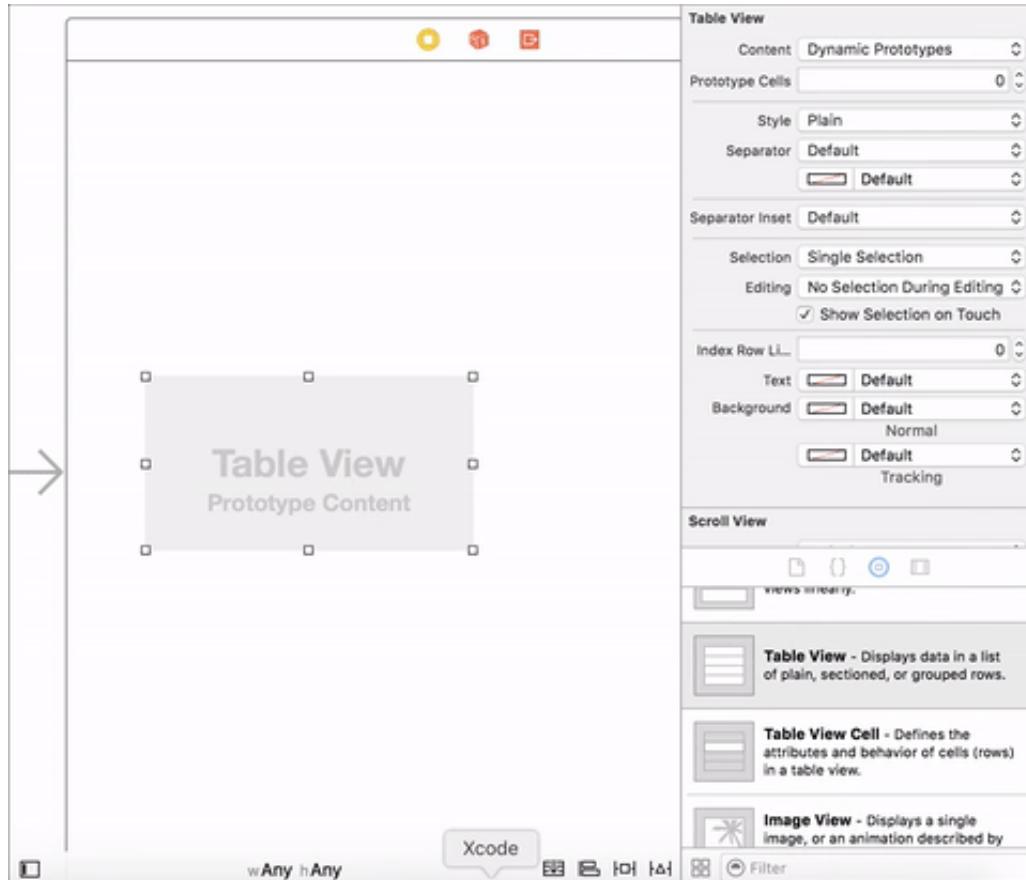
preguntará donde quieres guardar tu proyecto. Elige una ubicación de tu ordenador y haz clic en el botón **Create**. Xcode creará tu proyecto y te mostrará tu entorno de trabajo:



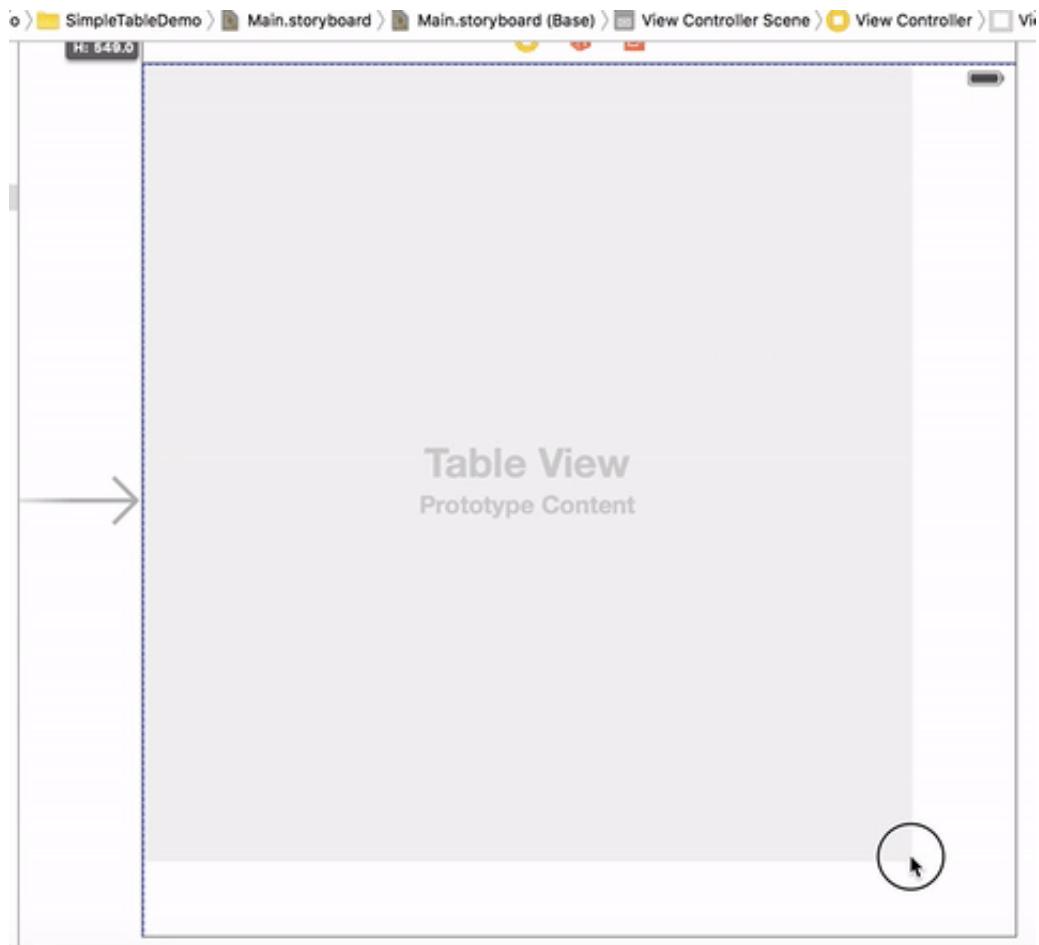
Lo primero que haremos será crear la interfaz de nuestra aplicación. Abre el fichero **Main.storyboard**.



Haz doble clic en el único **ViewController** que tenemos en nuestro storyboard, para seleccionarlo. Después, en la librería de objetos, situada en la parte inferior derecha, desplázate hacia abajo hasta encontrar el objeto **Table View** y arrástralolo al interior de la view de nuestro ViewController.



Ahora **redimensiona** la UITableView para que ocupe toda la view principal:

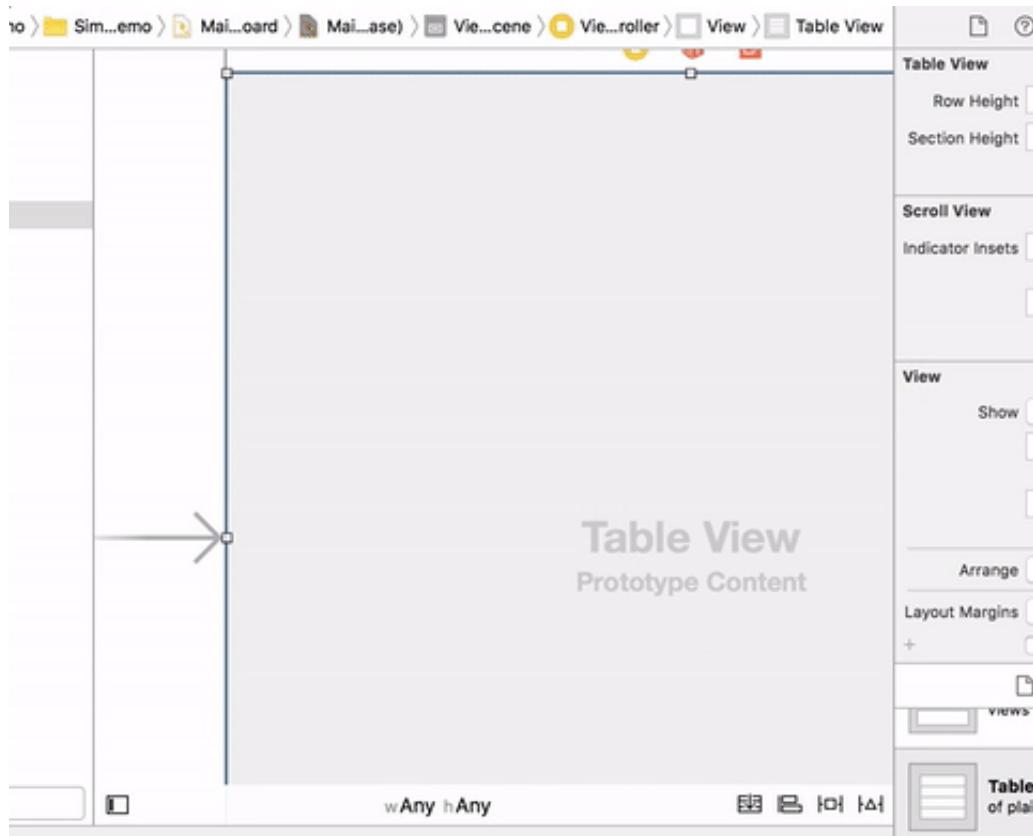


Perfecto. Acabas de añadir un UITableView a la interfaz de tu aplicación. Esta tabla será la que utilicemos para mostrar los datos de nuestra app.

Lo siguiente que tienes que hacer es **añadir las constraints** de esa tabla. Las constraints nos permiten asegurarnos de que al ejecutar nuestra app en cualquier dispositivo, nuestra tabla aparecerá **correctamente centrada**.

Si quieres aprender más sobre **Constraints** y **Auto Layout**, te recomiendo que visites nuestra **serie de tutoriales** sobre este tema: [Introducción a Auto Layout](#)

Para especificar que nuestra tabla aparezca siempre centrada en pantalla, haremos que mantenga una **distancia de 0 puntos** en cada uno de sus cuatro márgenes con respecto a la view principal. Para ello, lo único que tenemos que hacer es añadir estas cuatro constraints a nuestra UITableView:



Genial, ahora nuestra tabla se mostrará correctamente en cualquier dispositivo que ejecutemos nuestra aplicación.

Una tabla vacía no sirve para nada, así que nuestro siguiente paso será **añadir datos** a nuestra UITableView. Pero antes, debemos explicar algunos **conceptos básicos** que debes conocer para comprender completamente el comportamiento de las tablas.

4. Conceptos fundamentales de UITableView

PROTOCOLO UITABLEVIEWDATASOURCE

Apple nos ofrece la clase **UITableView** para que podamos trabajar con tablas en nuestras aplicaciones. Esta clase ha sido diseñada para poder mostrar tipos de datos muy diferentes. En una tabla puedes mostrar prácticamente **cualquier dato**, desde un conjunto de opciones básicas hasta un listado de restaurantes con la información completa de cada uno, clasificados por ciudades.

La pregunta es:

¿Cómo le decimos a la clase UITableView que información debe

mostrar?

La respuesta es, utilizando el protocolo **UITableViewDataSource**.

Un **protocolo** es la declaración de un conjunto de propiedades y métodos agrupados bajo un **nombre común**, que pueden ser implementados por cualquier clase.

Además de entender los protocolos, es importante que tengas en cuenta esto:

Las TableViews se dividen en Secciones y Celdas

También debes saber que las **tablas** se dividen en **diferentes secciones**, donde cada sección puede contener un determinado **número** de **celdas**.

El protocolo UITableViewDataSource **es el enlace** entre los datos que queremos mostrar y nuestra UITableView. Este protocolo declara **dos métodos** obligatorios:

- tableView:numberOfRowsInSection()
- tableView:cellForRowAtIndexPath()

El primero de los métodos: **numberOfRowsInSection()**, especifica el número de celdas que mostraremos en nuestra tabla y se ejecutará tantas veces como secciones tenga nuestra tabla. En nuestro caso, como no hemos indicado el número de secciones, iOS entenderá que nuestra tabla tiene **una única sección**, por lo que este método se ejecutará una única vez.

El segundo método: **cellForRowAtIndexPath()**, nos permite especificar **los datos que mostraremos en cada celda** y se ejecutará tantas veces como celdas tenga la única sección de nuestra tabla.

Al tratarse de **métodos obligatorios**, tendremos que implementarlos en nuestra clase si queremos utilizar el protocolo **UITableViewDataSource**.

PROTOCOLO UITABLEVIEWDELEGATE

Por otro lado tenemos el protocolo **UITableViewDelegate**. Este protocolo es el encargado de **determinar la apariencia** de nuestra UITableView. Todos sus métodos son opcionales, por lo que no estamos obligados a implementar ninguno. Sin embargo, **ofrece funcionalidades muy útiles** como especificar el height de las celdas, configurar tanto el header como el footer de una tabla, reordenar las celdas, etc.

En este tutorial no utilizaremos ninguno de estos métodos, ya que nos vamos a centrar únicamente en **mostrar los datos en nuestra aplicación** pero aún así es importante que conozcas las opciones que te ofrece este protocolo.

NSINDEXPATH

Recuerda lo que hemos comentado en el apartado anterior. Las tablas se dividen en diferentes **secciones**, donde a su vez cada sección puede contener un determinado número de **celdas**.

La clase **NSIndexPath** es fundamental para que entiendas completamente el trabajo con tablas en aplicaciones iOS. Siempre que quieras utilizar alguna TableView en tu aplicación, tendrás que implementar el método **cellForRowAtIndexPath** y gran parte del funcionamiento de este método se basa en un objeto de la clase **NSIndexPath**.

Este objeto tiene dos propiedades:

- .section
- .row

A través de estas dos propiedades, **podremos situarnos en un punto concreto de una tabla**. Mediante la propiedad .section especificaremos en que **sección** de nuestra tableView nos encontramos, mientras que a través de la propiedad .row determinaremos, dentro de esa sección, en que

celda nos encontramos.

Por tanto si especificáramos lo siguiente:

- `.section = 0`
- `.row = 4`

Nos encontraríamos en la quinta celda de la primera sección de nuestra tabla. Fácil, ¿no?

5. Añadiendo datos a nuestra tabla

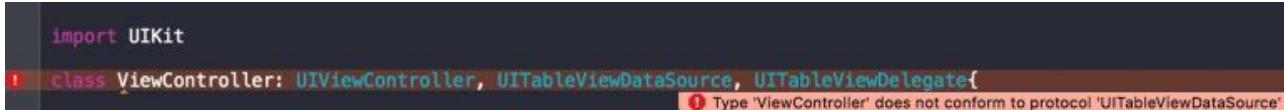
Ahora que ya conoces el concepto de **IndexPath** y ya entiendes para que sirven los protocolos **UITableViewDataSource** y **UITableViewDelegate**, podemos utilizarlos para añadir datos a nuestra aplicación.

Lo primero que tenemos que hacer es especificar que la clase **ViewController.swift** se ajustará a los protocolos **UITableViewDataSource** y **UITableViewDelegate**. Este es el primer paso siempre que trabajes con protocolos, determinar **que clase** será la que implemente el protocolo.

Para hacer esto únicamente tendrás que añadir **el nombre de los protocolos** a continuación de la declaración de nuestra clase, es decir, tu clase **ViewController.swift** deberá tener este aspecto:

```
class ViewController: UIViewController, UITableViewDataSource,  
UITableViewDelegate{  
  
    override func viewDidLoad() {  
  
        super.viewDidLoad()  
  
    }  
  
    override func didReceiveMemoryWarning() {  
  
        super.didReceiveMemoryWarning()  
  
    }  
}
```

Después de añadir estos dos protocolos, Xcode te mostrará por pantalla **el siguiente error:**



```
import UIKit
! class ViewController: UIViewController, UITableViewDataSource, UITableViewDelegate {
    // ...
    // A red underline is under 'ViewController' with a tooltip: 'Type 'ViewController' does not conform to protocol 'UITableViewDataSource''
```

Este error es completamente normal. Xcode te está avisando que **no has implementado** los dos métodos obligatorios, que debes utilizar, si quieres que tu clase se ajuste al protocolo UITableViewDataSource. Por ahora, **no prestes atención a este error**, más adelante, al añadir los dos métodos, haremos que desaparezca.

El siguiente paso será declarar la variable donde **almacenaremos la información** que queremos que muestre nuestra tabla.

Como los datos que mostraremos en nuestra **UITableView** será un listado de equipos de fútbol, utilizaremos un array llamado **teams** para almacenar ese listado. Por tanto, tendrás que añadir la siguiente variable justo antes del método **viewDidLoad()**:

Una vez que hemos declarado nuestro array, tendremos que rellenarlo con los **nombres de los equipos**. Para ello, tendrás que **añadir** una linea de código a tu método viewDidLoad():

```
override func viewDidLoad(){
    super.viewDidLoad()
    teams = ["Atletico de Madrid", "Barcelona", "Deportivo de la Coruña", "Las Palmas", "Malaga", "Rayo Vallecano", "Sporting", "Real Sociedad", "Espanyol", "Mallorca", "Valladolid", "Eibar", "Ponferradina", "Albacete"]
}
```

Perfecto. Ahora que ya tenemos nuestro array con todos los equipos que queremos mostrar en nuestra tabla, es hora de hacer que nuestra tabla **muestre estos datos**. Para ello utilizaremos los dos métodos del protocolo **UITableViewDataSource**:

- tableView:numberOfRowsInSection()

- `tableView:cellForRowIndexPath()`

Añade la implementación de estos dos métodos a tu clase **ViewController.swift**:

```
func tableView(tableView: UITableView, numberOfRowsInSection section:Int) -> Int {  
    return teams.count  
}  
  
func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {  
    let cell:UITableViewCell=UITableViewCell(style:  
    UITableViewCellStyle.Subtitle, reuseIdentifier: "mycell")  
  
    cell.textLabel?.text = teams[indexPath.row]  
  
    return cell  
}
```

Recuerda que hemos comentado antes, que estos dos métodos **no** se ejecutan una única vez.

A continuación tienes la **explicación del código** de cada uno de ellos.

El método **numberOfRowsInSection()**, como hemos dicho, es el encargado de especificar el **número de celdas** que mostraremos. En nuestro caso, como queremos mostrar tantas celdas como elementos hayamos almacenado en nuestro array teams, utilizaremos la propiedad **count**, que lo que hace es devolver el número de elementos que hay almacenados en un array. De esta forma, si nuestro array characters **tiene 14 elementos**, nuestro método numberOfRowsInSection devolverá 14. Por tanto nuestra tabla mostrará 14 celdas con contenido.

Por otro lado, el método **cellForRowIndexPath()** crea un objeto

celda, que hemos llamado **cell**, de tipo **UITableViewCell**, al que se le asigna el identificador “**mycell**”. Posteriormente, lo que hace es asignar a su propiedad **textLabel.text**, el texto que queremos mostrar, que en nuestro caso, será el elemento que esté contenido en el array, en la posición **indexPath.row**. Recuerda que la propiedad **indexPath.row** indica **la celda en la que nos encontramos**, por lo que la primera vez que se ejecute el método **cellForRowAtIndexPath()**, **indexPath.row** será igual a 0, la segunda será igual a 1, la tercera será igual a 2, de esta forma podremos recorrer nuestro array **teams** y mostraremos cada vez un elemento distinto del array. No olvides que el método **cellForRowAtIndexPath()** se ejecuta tantas veces como celdas vayamos a mostrar en nuestra tabla.

Ahora, prueba a ejecutar tu aplicación y...

...comprobarás que nuestra tabla se muestra completamente vacía.

¿Por qué ha sucedido esto?

Si te das cuenta, uno de los protocolos que hemos utilizado se llama **UITableViewDelegate**, lo que nos da la pista de que se trata de un **Protocolo de Delegado**.

Si no has trabajado antes con ellos, aquí puedes consultar todo lo que necesitas sobre el concepto de [Delegado](#).

Como has podido ver, hay que seguir **3 pasos** para implementar un delegado. Nosotros solo hemos realizado 2 pasos:

- **Paso 1: OK** – En la clase que actuará como Delegado lo hemos indicado específicamente añadiendo el protocolo **UITableViewDelegate** en su cabecera.

```
class ViewController: UIViewController, UITableViewDataSource, UITableViewDelegate{
```

- **Paso 2: PENDIENTE** – En la clase que delegará sus funciones tendremos que indicar quien será su clase Delegado. En este caso, no

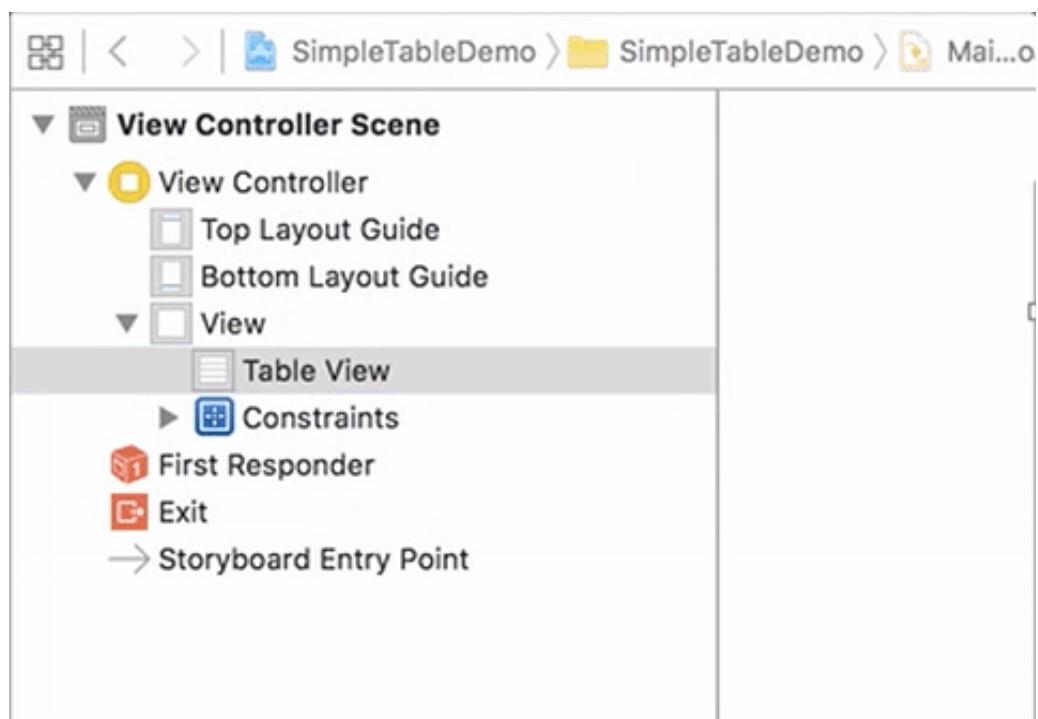
hemos especificado todavía en nuestra clase **UITableView** quien será su clase delegado.

- **Paso 3: OK** – Hemos implementado las funciones que queremos que realice el delegado a través de los métodos **numberOfRowsInSection()** y **cellForRowAtIndexPath()**

Para que el delegado funcione correctamente no podemos saltarnos el Paso 2. Así que vamos a ello. Vamos a especificar en nuestra clase UITableView **quien será su clase delegado**.

Accede al **Main.storyboard** y dejando la tecla **Ctrl** pulsada haz clic en el objeto **Table View** y arrastra hasta soltar justo encima del **View Controller**. En el menú flotante que aparece, selecciona **delegate**.

Repite el proceso de nuevo y selecciona también **dataSource**.



Ahora ya puedes **ejecutar la aplicación** y comprobar que nuestra tabla muestra correctamente los **datos** contenidos en el array teams.

Atletico de Madrid

Barcelona

Deportivo de la Coruña

Las Palmas

Malaga

Rayo Vallecano

Sporting

Real Sociedad

Espanyol

Mallorca

Valladolid

Eibar

Deportivo Alaves

6. Añadiendo imágenes a nuestra tabla

Nuestra aplicación quedaría **demasiado sencilla** si la termináramos aquí.

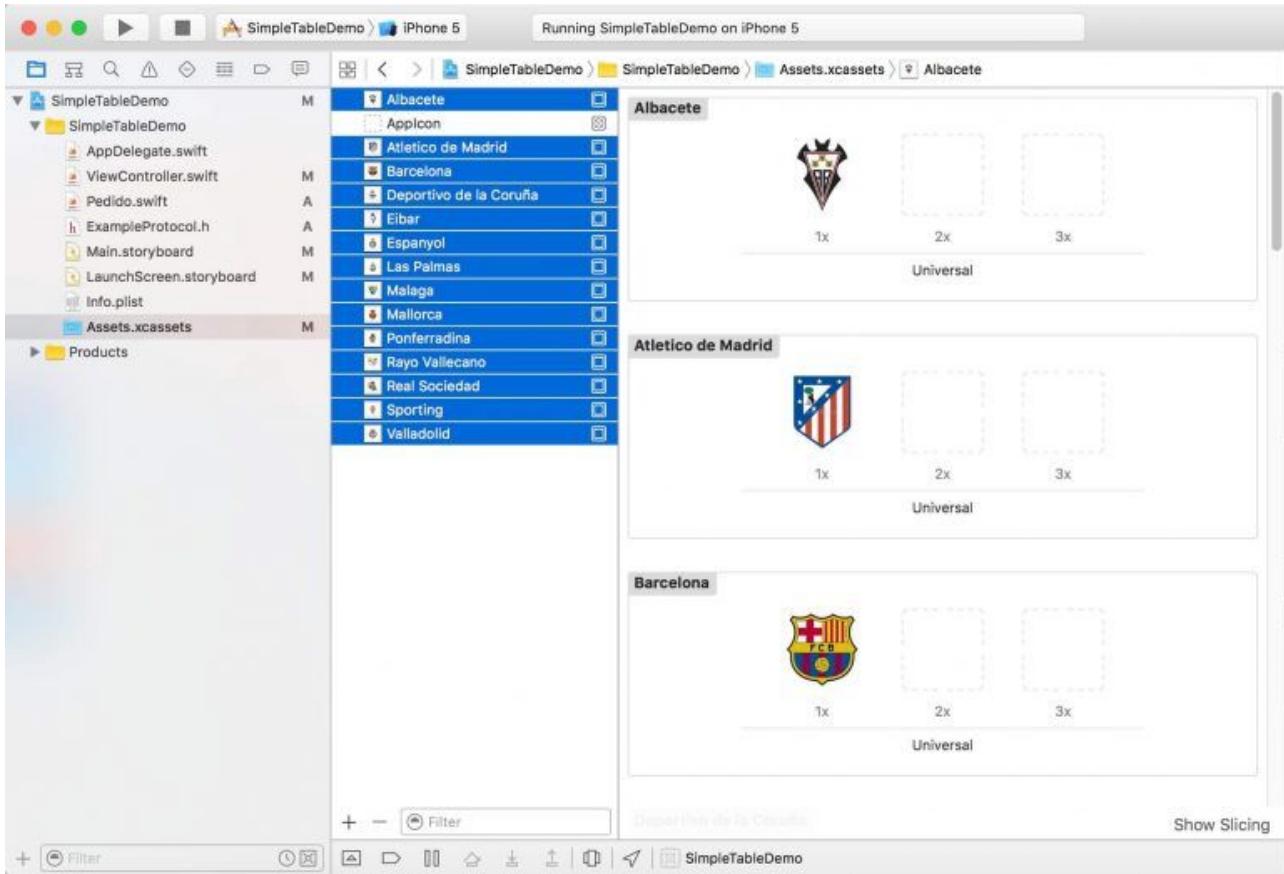
Lo que vamos a hacer es **añadir una imagen** diferente a cada una de las celdas.

Para que no pierdas tiempo buscando imágenes que añadir, **he preparado un .zip** con 14 escudos de equipos españoles de futbol.
[Descarga el fichero desde aquí.](#)

Una vez que lo hayas descargado, **descomprímelo** y comprueba que dentro aparecen las 14 imágenes.

Para añadir las imágenes al proyecto haz clic en el fichero

Assets.xcassets y arrastra las 14 imágenes al espacio donde está situado el AppIcon:



Con las imágenes añadidas a nuestro proyecto, simplemente tendremos que decirle a nuestro **UITableView** que muestre **estas imágenes** en cada una de las celdas.

Para ello tendrás que añadir la siguiente linea en el método **cellForRowIndexPath()**, justo antes del **return cell:**

```
cell.imageView!.image = UIImage(named: teams[indexPath.row])!
```

Lo que hacemos con esta linea de código es asignar a la imageView de la celda, una **UIImage** que contendrá para cada celda un escudo diferente.

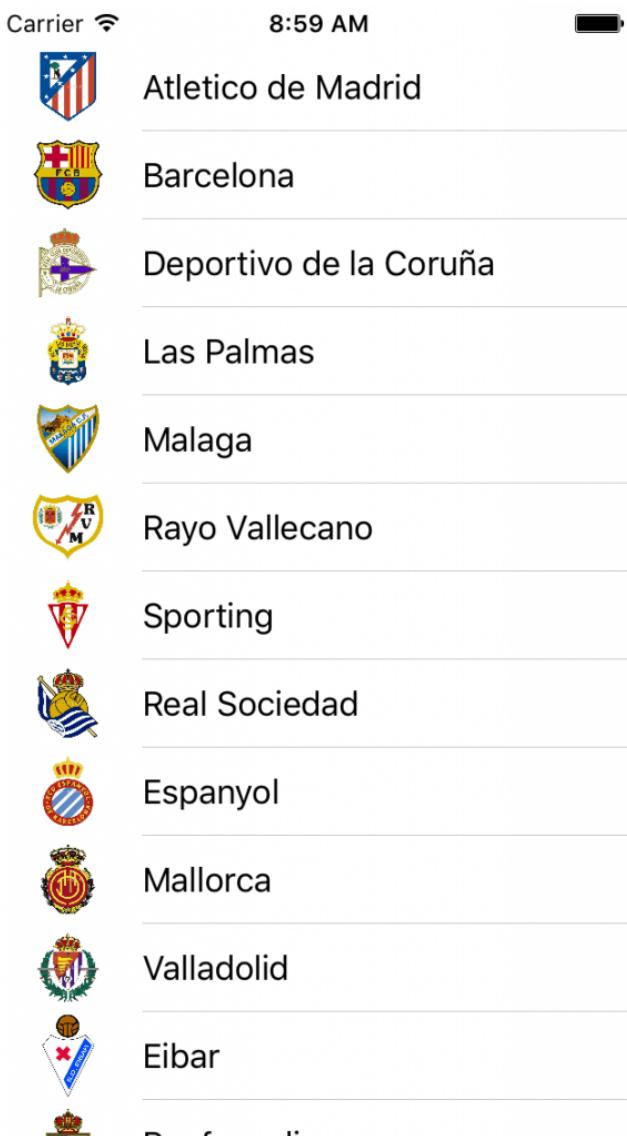
Date cuenta que para que esto funcione, hemos hecho que los títulos de las imágenes se llamen **exactamente igual** que el texto que estamos mostrando en cada una de las celdas de la tabla. Lo hemos hecho así para no tener que crear otro array con los nombres de las imágenes.

Este debería ser el aspecto de tu método **cellForRowIndexPath()** en

este momento:

```
func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell
{
    let cell:UITableViewCell=UITableViewCell(style:
    UITableViewCellStyle.Subtitle, reuseIdentifier: "mycell")
    cell.textLabel?.text = teams[indexPath.row]
    cell.imageView!.image = UIImage(named: teams[indexPath.row]!)
    return cell
}
```

Ahora ya puedes **ejecutar de nuevo la aplicación** y comprobar que se muestran los nombres de los equipos cada uno con un escudo diferente.



7. Resumen final

Espero que este tutorial te haya servido para entender como funcionan las **UITableViews** en iOS. Se trata de un elemento fundamental, por lo que es interesante que seas capaz de manejarlas perfectamente.

Aquí tienes un **pequeño resumen** de los puntos más importantes que hemos visto:

1. Crear desde cero una aplicación con una UITableView
2. Como funcionan los protocolos UITableViewDataSource y UITableViewDelegate
3. Utilizar los métodos numberOfRowsInSection() y cellForRowAtIndexPath()
4. Como utilizar el IndexPath
5. Mostrar datos almacenados en un array a través de nuestra tabla
6. Como añadir imágenes a las celdas de nuestra tabla

Crea tu Aplicación con Menú Lateral

INTEGRANDO LIBRERÍAS EXTERNAS EN TUS PROYECTOS

Lenguaje Swift | Nivel Avanzado

1. Introducción

En este tutorial, vamos a **desarrollar una aplicación en Swift**, la cual tendrá un tipo muy común de interfaz: **Un menú lateral**. Es probable que hayas visto este menú en muchas apps.

Podríamos programar esta aplicación completamente desde cero, pero en este caso, lo que vamos a hacer es utilizar una **librería de terceros** que nos permitirá incorporar el menú lateral a nuestra app. Así, además de ver como desarrollar la app, podrás entender el proceso a la hora de **añadir librerías externas a tu aplicación**.

El uso de librerías de terceros en tus aplicaciones es una opción **muy válida**. Permite incorporar funcionalidades completas que ayudan a minimizar nuestros tiempos de desarrollo. Por el contrario, en otras ocasiones, el incorporar librerías en tu aplicación puede suponer **aumentar** excesivamente **la complejidad de la app** y perder control sobre la misma. Por esto, debes ser tu mismo, como desarrollador quien evalúe **cual es la opción recomendada** para cada proyecto:

- Utilizar librerías de terceros
- Realizar el desarrollo de forma íntegra por tu cuenta

Otro punto interesante de este proyecto, es que vamos a desarrollar nuestra aplicación iOS con menú lateral **utilizando Swift**, mientras que **la librería** del menú lateral que vamos a incorporar ha sido desarrollada **completamente en Objective-C**.

Esto te servirá para comprobar lo fácil que puede ser integrar en tus

proyectos Swift cualquier código en Objective-C.

La librería que vamos a utilizar para nuestro menú lateral se llama **SWRevealViewController** y está disponible para que puedas incorporarla en cualquiera de tus proyectos **de forma gratuita**. Si quieres echarle un vistazo a la librería [accede aquí](#).

Para que te hagas una idea, al terminar el tutorial, **este será el aspecto** de nuestra aplicación iOS con menú lateral:



2. ¿Qué vamos a ver en este Tutorial?

Estas podrían ser las partes más interesantes de este Tutorial:

- Desarrollarás una aplicación funcional **en Swift**

- Aprenderás a **integrar librerías** de terceros en tus proyectos
- Verás como **integrar código en Objective-C** en tus proyectos Swift

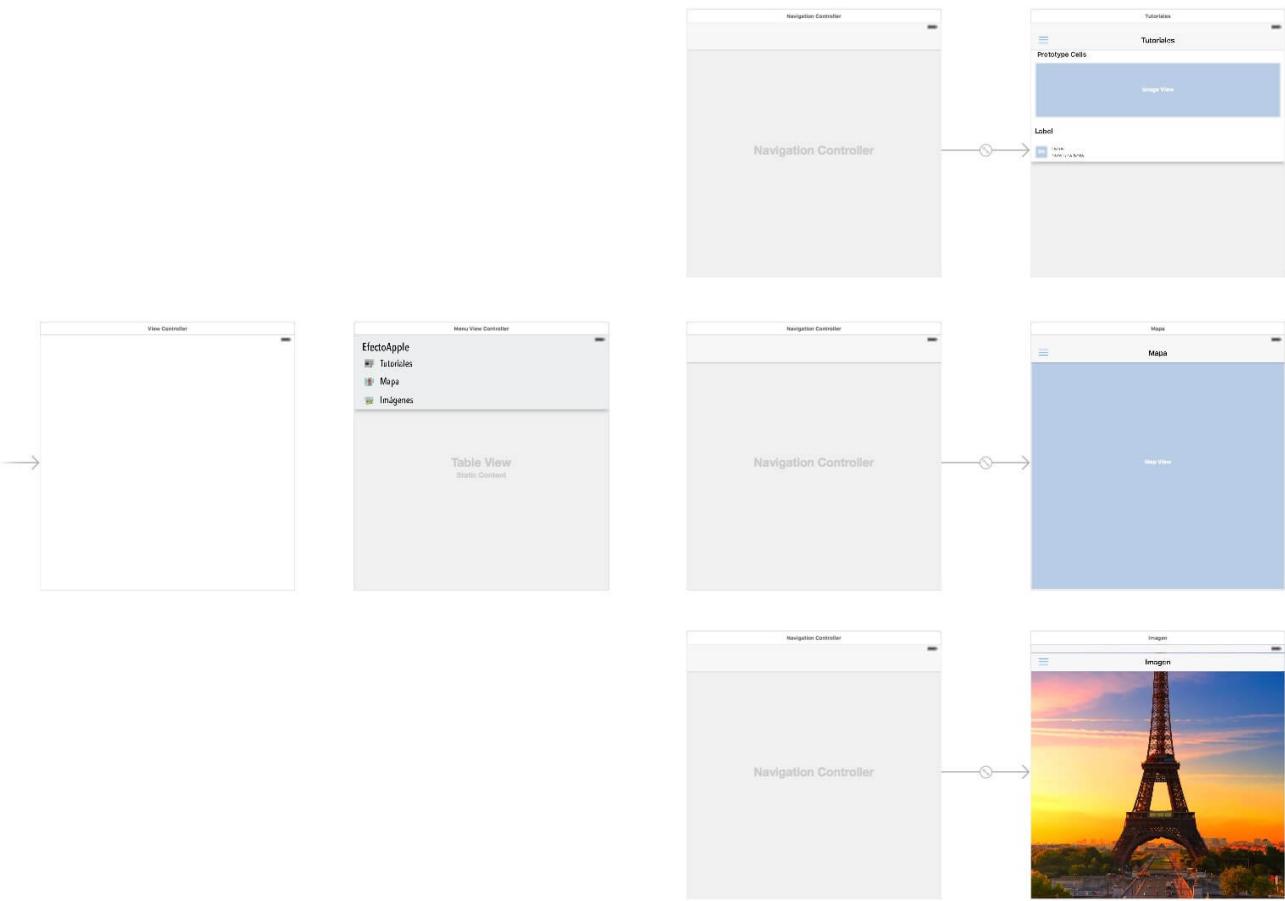
Si crees que alguno de estos puntos puede ser interesante para ti, sigue leyendo, porque ¡comenzamos!

3. Proyecto de Inicio

Para que no tengas que perder el tiempo creando gran parte de la interfaz de la aplicación, he creado **un proyecto de inicio** que puedes [descargar desde aquí](#). Trabajaremos a partir de este proyecto en las partes **realmente importantes** de la aplicación.

Para que comprendas el funcionamiento completo de la aplicación, vamos a echar **un vistazo rápido** al contenido del proyecto de inicio.

Descarga el proyecto, ábrelo y accede al fichero **Main.storyboard**. Ésta es la interfaz completa de nuestra aplicación y aquí puedes ver que están creados **todos los viewControllers** que vamos a necesitar en nuestra aplicación:



No te preocunes, si de primeras te parece una interfaz compleja, en cuanto la expliquemos verás que **es muy sencilla**.

Para poder utilizar la librería SWRevealViewController para construir nuestro menú lateral, he creado un **containerViewController**, que únicamente es un viewController vacío, que será el encargado de **contener nuestro menuViewController y el resto de viewController de la aplicación**. Este containerViewController es el ViewController que se encuentra más a la izquierda en la imagen.

Justo a la derecha de él, puedes ver que ya está creado el **menuViewController**. Simplemente se trata de una tabla estática con 3 elementos de menú. Cada uno de estos 3 elementos nos llevará a un **navigationController**, que contendrá un viewController diferente. Podremos ver una sección de **Tutoriales**, una sección donde se nos muestra un **Mapa** y una sección donde podremos ver una **Imagen**.

En la **sección Tutoriales** he creado un **tableView**, con una custom cell, donde mostraré algunos tutoriales de ejemplo de EfectoApple.

En la **sección Mapa**, únicamente hay un **mapView**, en la que no he especificado ninguna localización concreta.

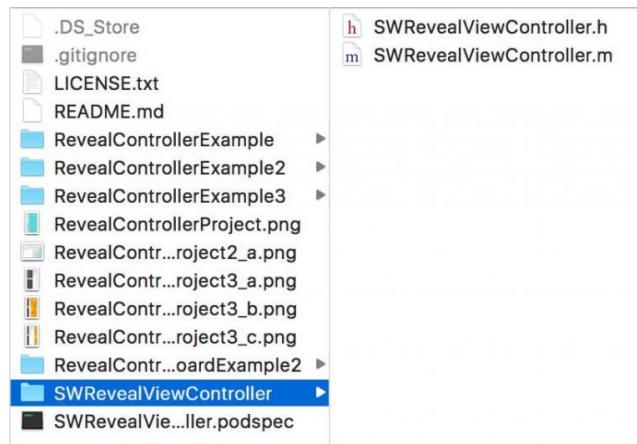
Por último, en la **sección Imágenes**, he añadido un **imageView** que muestra una fotografía de París.

Para este ejemplo, mostraremos contenido estático, pero si quieres modificar el proyecto y añadir otro tipo de contenido o más opciones al menú, únicamente tendrás que **añadir mas viewControllers** dentro del storyboard.

Como puedes ver, se trata de una **aplicación muy sencilla**.

4. Como integrar la librería externa

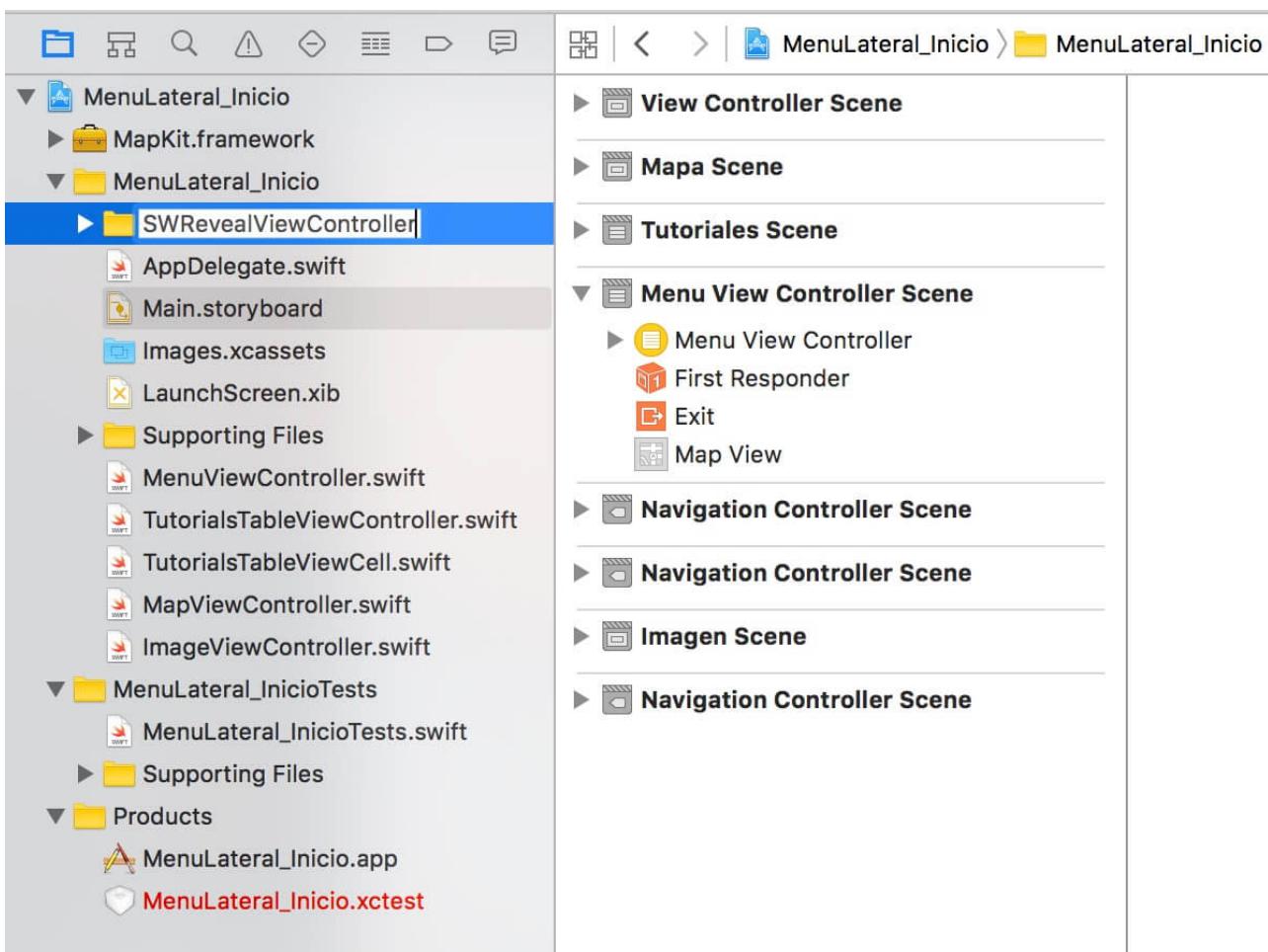
Como hemos dicho antes, vamos a utilizar la librería **SWRevealViewController** para implementar nuestro menú lateral. Así que lo primero que debes hacer es descargarla [desde aquí](#). Una vez que la hayas descargado, debes descomprimir el fichero descargado y dentro de la carpeta principal encontrarás otra carpeta llamada **SWRevealViewController**. Si accedes a ella, encontrarás 2 ficheros llamados **SWRevealViewController.h** y **SWRevealViewController.m**.



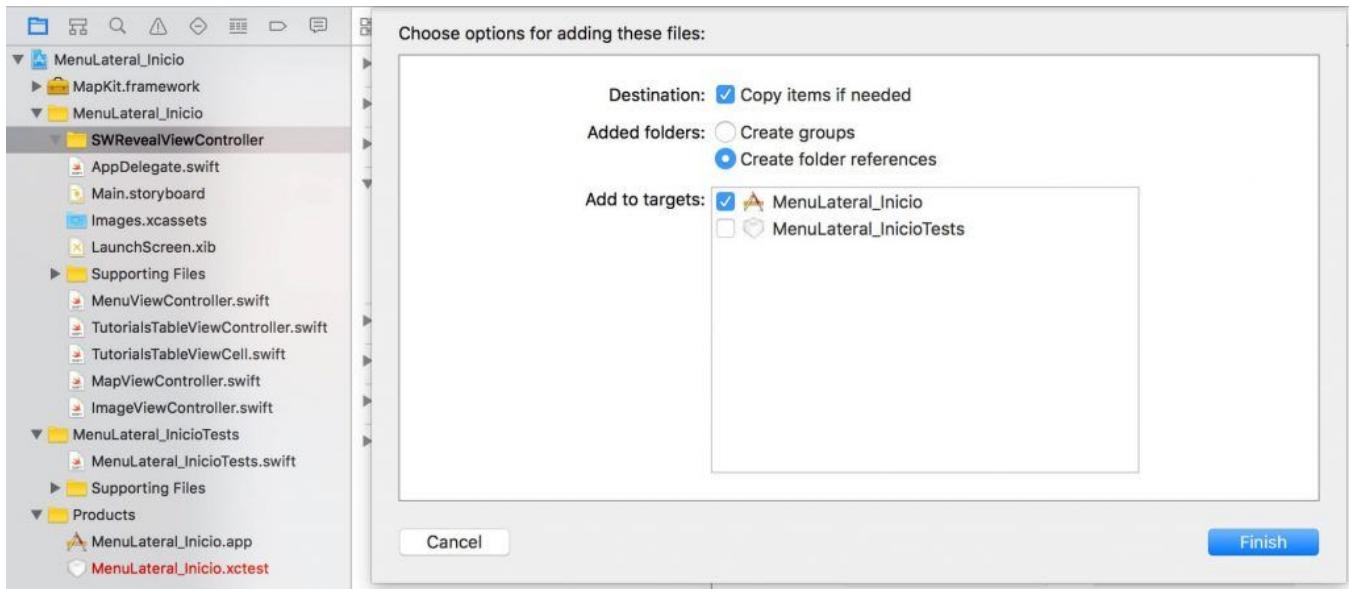
Si no has trabajado antes con Objective-C, tal vez te preguntes porque estos ficheros tienen extensiones **.h** y **.m** y no **.swift**. Esto se debe a que en Objective-C, cada clase está dividida en dos ficheros diferentes, un **fichero de cabecera** (**.h**) y un **fichero de implementación** (**.m**).

Estos dos ficheros, forman la librería SWRevealViewController, así que **vamos a añadirlos** a nuestra aplicación.

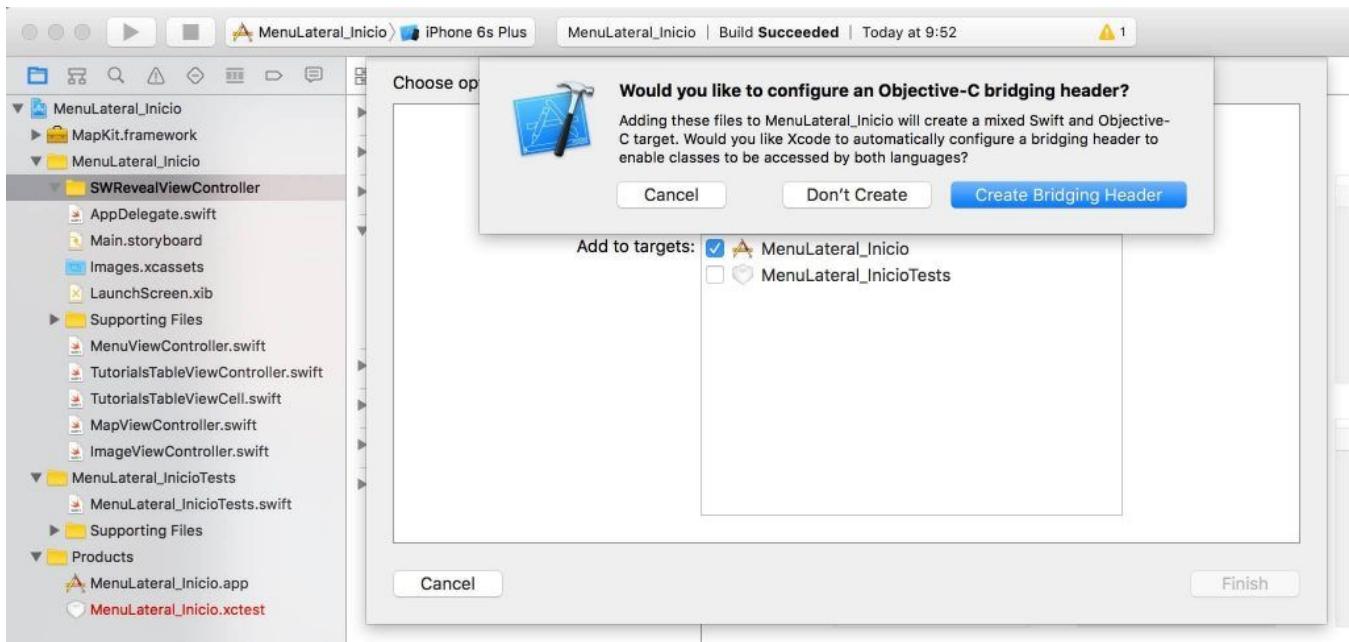
Lo primero que debes hacer, para mantener organizado tu código es **crear una carpeta en tu proyecto**, que se llame **SWRevealViewController**. Para ello, colócate en el navegador del proyecto, situado en la parte izquierda de Xcode, haz clic con el botón derecho sobre la carpeta **MenuLateral_Inicio** y selecciona “**New Group**”. Dale el nombre SWRevealViewController.



A continuación arrastra los **2 ficheros (.h y .m)** de la librería hasta dentro de esta carpeta. Xcode te pedirá que confirmes que quieres añadir estos ficheros, asegurate que la opción “**Copy items if needed**” esté seleccionada y pulsa en el botón **Finish**.



Al pulsar en este botón, Xcode te preguntará si quieres crear un **Objective-C bridging header**. Haz clic en el botón Create Bridging Header. Lo que estás haciendo es crear un **fichero header** que te permitirá **acceder a código Objective-C desde Swift**.



A continuación Xcode generará un fichero header llamado **SidebarMenu-Bridging-Header.h**, dentro de nuestra carpeta SWRevealViewController. Abre ese fichero SidebarMenu-Bridging-Header.h y añade la siguiente linea:

```
#import "SWRevealViewController.h"
```

5. Especificando nuestro ViewController principal y

secundario

Para entender la librería con la que estamos trabajando, debemos tener claro que nuestra aplicación consta de **dos partes diferenciadas**:

- **Un viewController Principal:** Donde mostraremos el **contenido** que seleccione el usuario: Puede ser la vista de Tutoriales, la de Mapa o la de Imágenes.
- **Un viewController Secundario:** Donde mostraremos nuestro **menú** de navegación.

La configuración de SWRevealViewController es muy simple, lo único que tenemos que hacer es asociar la clase SWRevealViewController con nuestro viewController principal y con nuestro viewController secundario **utilizando segues**.

Si no tienes experiencia utilizando storyboards, tal vez no sepas lo que son los segues. Un **segue** es una **transición entre las escenas** de nuestro storyboard. Es decir, la forma que tenemos de controlar, como pasar de un viewController de nuestra aplicación a otro.

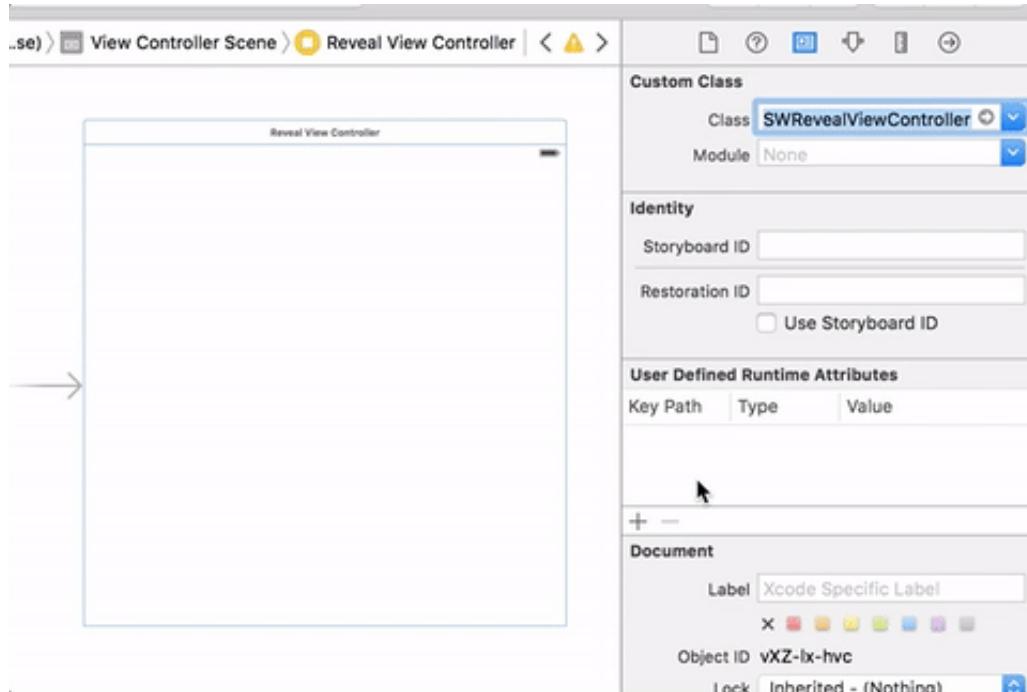
En nuestro proyecto, el viewController principal será el Navigation Controller de la **sección de Tutoriales**, de esta forma, esta será **la vista por defecto** que se mostrará al lanzar la aplicación, cuando el usuario todavía no ha seleccionado ninguna opción del menú.

En cambio, nuestro viewController secundario será **Menu View Controller**, que es quien gestiona el menú de navegación.

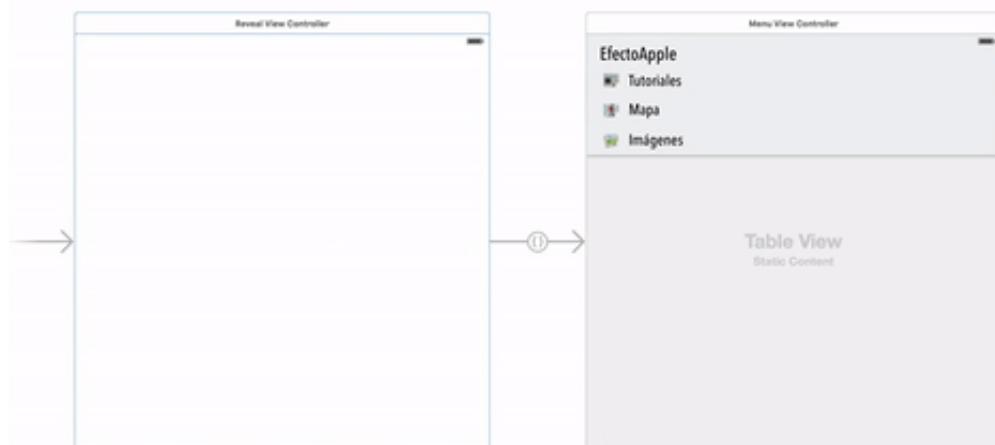
Una vez que tenemos esto claro, es el momento de especificar en la librería quien será el **viewController principal** y el **viewController secundario**.

Para ello, **abre el storyboard** y selecciona el viewController inicial de nuestra aplicación. Siempre puedes saber cual es el viewController inicial de una aplicación fijándote en la **flecha de selección** situada en la parte izquierda.

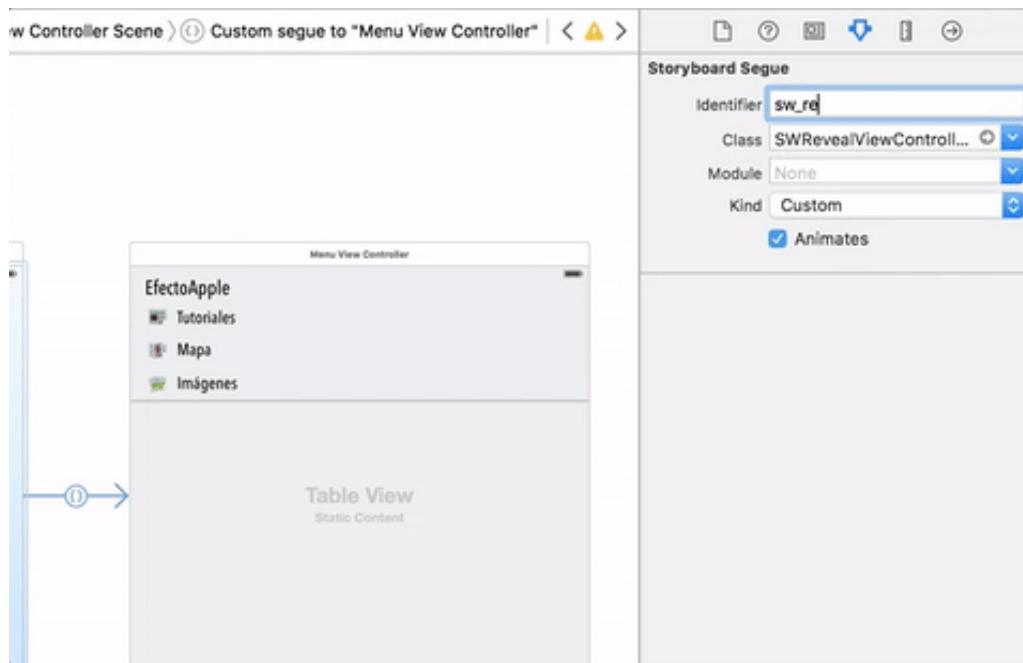
Una vez que lo has seleccionado accede al **Inspector de Identidad de Xcode** y cambia su clase a **SWRevealViewController**:



Después, deja pulsada la **tecla ctrl** y **arrastra** desde este viewController y suelta el ratón justo encima del viewController que tienes a la derecha: **Menu View Controller**. Verás un menú contextual que te ofrece varias opciones. Elige la opción “**reveal view controller set controller**”.



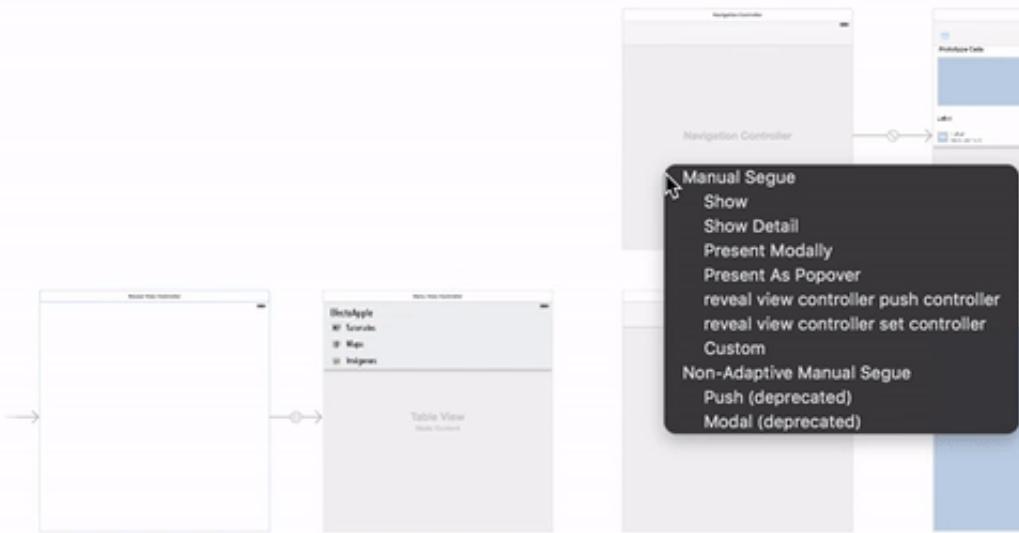
Lo que hemos hecho es definir un segue personalizado: “**SWRevealViewControllerSetSegue**”. Selecciona este segue y cambia su identificador a “**sw_rear**” utilizando el **Inspector de Identidad**.



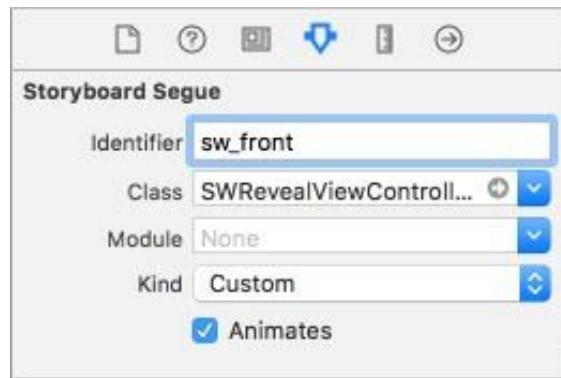
Especificando este identificador, lo que hacemos es decirle a SWRevealViewController, que **Menu View Controller** es el viewController **secundario** de nuestra aplicación. Es decir, el viewController que usaremos **para mostrar nuestro menú**.

Ahora que ya hemos definido nuestro viewController secundario, deberemos definir nuestro viewController principal. Para ello, **repetiremos el proceso**.

Arrastra desde **Reveal View Controller**, con la **tecla ctrl pulsada** y suelta justo encima del Navigation Controller situado en la parte superior. Después selecciona la opción “**reveal view controller set controller**”, exactamente igual que antes.



Selecciona el segue que acabas de crear y cambia su identificador a “**“sw_front”**”. De esta forma, le decimos a SWRevealViewController que el navigationController de la sección de Tutoriales **es el viewController principal** de nuestra aplicación.



Antes de continuar, lo mejor es que **hagas una prueba**, para comprobar si has seguido todos los pasos correctamente. **Ejecuta la aplicación** en el simulador de Xcode y comprueba que se muestra correctamente la vista Tutoriales. Verás, que por ahora, el botón de menú (También llamado **Hamburguer Button** por su forma) no funciona todavía. Este será nuestro siguiente paso, hacer que nuestro Hamburguer Button funcione.



Como convertirte en Desarrollador iOS

 Luis Rollón Gordo
hace una hora



Aprende a manejar Sketch

 Luis Rollón Gordo
hace una hora



Crea tu Aplicación iOS Cliente-Servidor

Accede a la clase **TutorialsTableViewController**, que es el controlador que gestiona la vista de Tutoriales. En su método **viewDidLoad()** inserta las siguientes líneas de código:

```
if self.revealViewController() != nil {  
  
    menuButton.target = self.revealViewController()  
  
    menuButton.action = #selector(SWRevealViewController.revealToggle(_:))  
  
    self.view.addGestureRecognizer(self.revealViewController().panGestureRecognizer())  
}
```

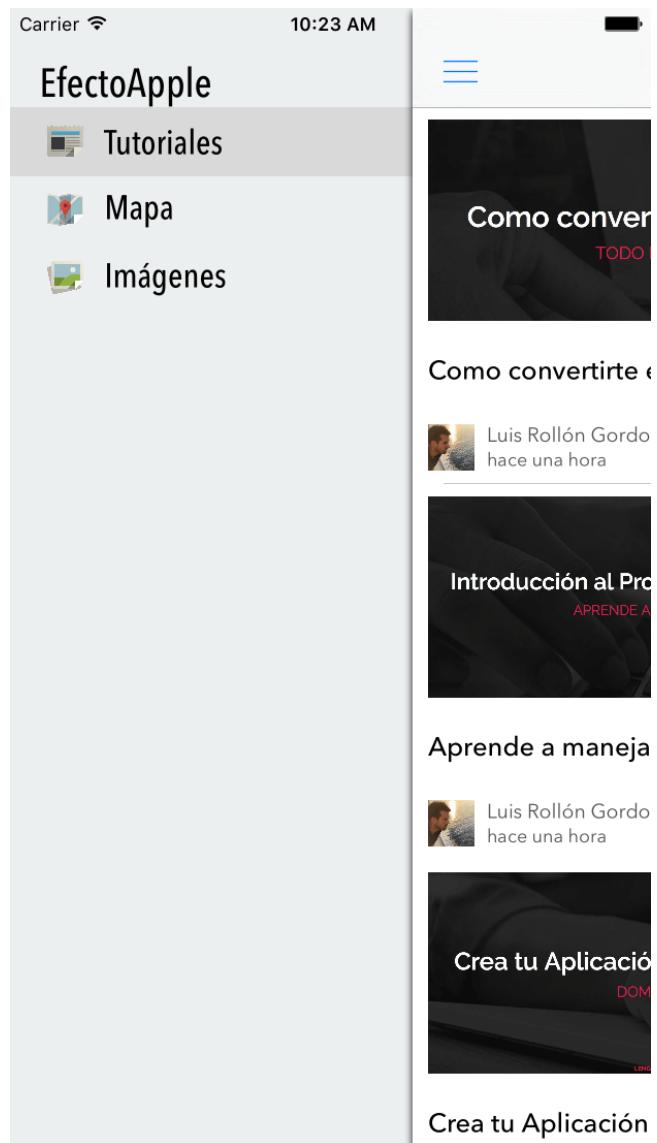
Lo que hemos hecho en este código es utilizar el mecanismo de comunicación **Target-Action** para comunicarnos entre nuestro hamburguer button y nuestro viewController.

La librería SWRevealViewController nos ofrece el método **revealViewController()** que nos permite obtener el SWRevealViewController padre de cualquier controlador. También nos ofrece el método **revealToggle()** que permite la expansión y contracción del menú lateral. Lo que hacemos en este código, es especificar, que cuando se cargue la vista de Tutoriales, nuestro botón cuente con la funcionalidad de mostrar u ocultar el menú, **cuando sea pulsado por el usuario.**

Establecemos el target del hamburguer button a nuestro revealViewController y nuestro action al método revealToggle(). De esta forma, cuando pulsemos en el botón del menú lateral, llamará al método revealToggle() de revealViewController y **se mostrará el menú lateral.**

En la última linea lo que hacemos es añadir un **gesture recognizer**. Esto es necesario si queremos que además de mostrar el menú pulsando en el hamburguer button, también se pueda mostrar **deslizando la pantalla hacia la derecha**. Para esto sirve nuestro gesture recognizer.

Ahora, compila y ejecuta la aplicación en el simulador. Pulsa en el **hamburguer button** y comprueba que se muestra correctamente el menú lateral. Puedes ocultar el menú volviendo a pulsar el botón o deslizando la pantalla **hacia la izquierda**.



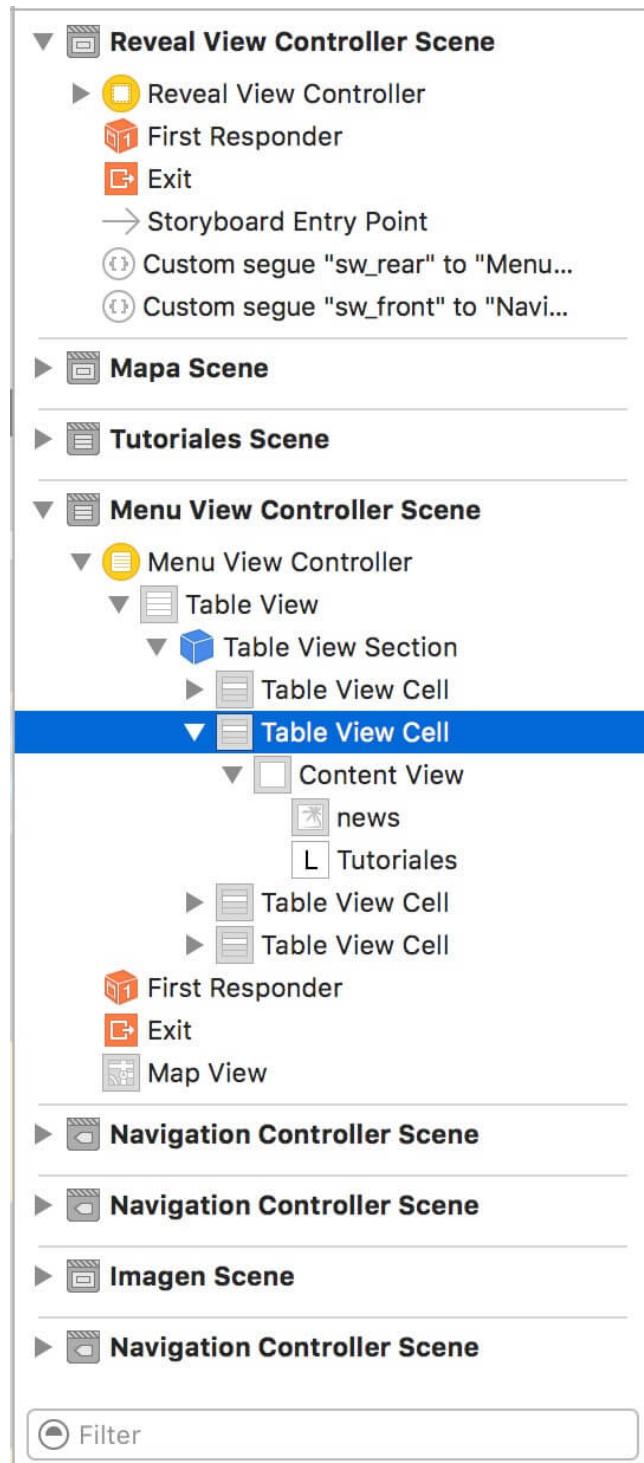
Si pruebas a pulsar los botones del menú, verás que la vista principal **no se modifica**, eso es porque todavía no hemos desarrollado esa parte. Es lo siguiente que haremos. ¡Vamos allá!

6. Enlazando nuestro menú con sus ViewControllers

Como acabamos de ver, nuestro menú ahora mismo **funciona a medias**. Vamos a **enlazar los elementos** de nuestro menú a sus correspondientes viewControllers, para que cuando el usuario pulse en cualquier opción, nuestra aplicación **actualice la vista principal** y la muestre en pantalla.

Abre el fichero **Main.storyboard**. Como acabamos de decir, vamos a ir enlazando cada uno de los elementos del menú: Tutoriales, Mapa, Imágenes con su correspondiente **viewController**. En esta ocasión vamos a realizar el arrastre desde el **Document Outline**. Si no sabes lo que es el Document Online, aquí lo tienes representado en esta

imagen:



Colócate, dentro del Menu View Controller, en la **cell de Tutoriales** y con la tecla **ctrl** pulsada arrastra hasta el **NavigationController** que está enlazado con el **viewController Tutorials**. En el menú contextual que aparece, dentro del apartado “**Selection Segue**” selecciona “**reveal view controller push controller**”. **CUIDADO:** No te confundas porque justo debajo hay otro “**reveal view controller push controller**” dentro del apartado “**Accessory Action**”. Recuerda que tienes que pulsar el que se encuentra dentro de la sección “**Selection Segue**”.



Repite este procedimiento con las celdas Mapa e Imágenes, enlazándolas con sus correspondientes NavigationControllers.

Por último tendrás que añadir el mismo código de antes, al método **viewDidLoad()** tanto de **MapViewController.swift**, como de **ImageViewController.swift**:

```
if self.revealViewController() != nil {
    menuButton.target = self.revealViewController()
    menuButton.action = #selector(SWRevealViewController.revealToggle(_:))
    self.view.addGestureRecognizer(self.revealViewController().panGestureRecognizer())
}
```

¡Eso es! Ahora, compila y ejecuta la aplicación y comprueba que los **elementos** de nuestro menú **funcionan perfectamente**.

7. Personalizando nuestra librería

Acabas de terminar una aplicación con menú lateral totalmente funcional. **Funciona perfectamente** y cumple con el objetivo que buscamos. Ofrecer al usuario una interfaz clara a través de la cual navegar por la aplicación.

¿Pero qué pasa si queremos **realizar modificaciones** en nuestra

aplicación? ¿Qué tenemos que hacer si por ejemplo queremos que la animación del menú lateral sea más rápida o que el ancho del menú lateral sea menor? En principio puede parecer complicado, ya que estos aspectos **están controlados por la librería** que hemos añadido a nuestro proyecto, pero con un poco de práctica te acostumbrarás fácilmente a interpretar el código de librerías externas.

Para poder realizar algún cambio de este tipo, lo que tendremos que hacer será **investigar las opciones de personalización** que nos ofrece la librería. Para ello tendrás que acceder al fichero **SWRevealViewController.h**, allí está definidas todas las propiedades y métodos de los que consta nuestra librería.

Echando un vistazo a esta clase, puedes ver que existe una propiedad llamada **toggleAnimationDuration** y justo encima hay un comentario que dice que se trata de la **velocidad de la animación** y que por defecto está establecida a **0.25 segundos**.

```
// Duration for the revealToggle animation, default is 0.25
@property (nonatomic) NSTimeInterval toggleAnimationDuration;
```

Por tanto, si queremos modificar la velocidad de la animación de nuestro menú lateral, parece obvio que tendremos que utilizar la propiedad **toggleAnimationDuration**.

¡Pues vamos allá!

Añade la siguiente linea al método **viewDidLoad()** de la clase **TutorialsTableViewController.swift**:

```
self.revealViewController().toggleAnimationDuration = 0.1
```

Ejecuta la aplicación y comprueba como la velocidad de la animación **ha aumentado** bastante.

Ahora **modifica** el valor de la propiedad y asignale 2.0:

```
self.revealViewController().toggleAnimationDuration = 2.0
```

Como puedes ver la animación se realiza ahora **de forma muy lenta**.

Este es **solo un ejemplo** de personalización de una librería externa. La mejor forma de sentirte cómodo es investigando el fichero .h que ofrece **SWRevealViewController** y realizando cambios tu mismo, comprobando **como cambia** la ejecución de la aplicación.

8. Resumen Final

Espero que este nuevo tutorial te haya parecido útil. Aquí tienes un **pequeño resumen** de los puntos más importantes que hemos visto:

1. Desarrollar una aplicación iOS con menú lateral
2. Integrar una librería de terceros en tus proyectos
3. Acceder a código en Objective-C dentro de tus proyectos en Swift
4. Personalizar librerías externas para que se ajusten a tus necesidades

Tutorial de Introducción a UIStackView

Todas las ventajas de utilizarla en tus apps

Lenguaje Swift | Nivel Intermedio

1. Introducción

En este Tutorial vamos a hablar de **UIStackView**.

Aunque tengas bastante experiencia desarrollando Aplicaciones iOS, seguro que más de una vez has tenido problemas para hacer que la interfaz de una aplicación **quedara exactamente como querías**.

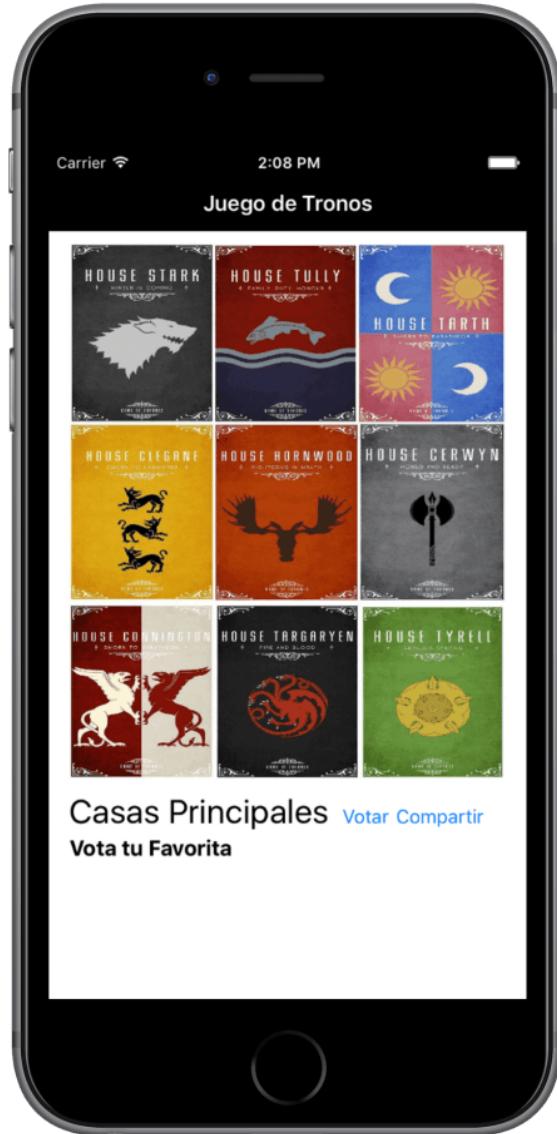
Hoy no vamos a hablar de Auto Layout aunque si vamos a ver **un tipo de view**, que Apple introdujo en iOS 9 y que **puede ayudarte** a desarrollar cualquier interfaz de forma sencilla: **UIStackView**.

2. ¿Qué vamos a ver en este tutorial?

En este tutorial vamos a **desarrollar una app completa** utilizando la clase **UIStackView**.

Crearemos una aplicación donde **se mostrarán los emblemas** de algunas de las casas aparecidas en el Universo de **Juego de Tronos**. Además mostraremos en la parte inferior dos labels y dos buttons que **carecerán de funcionalidad** y únicamente se usarán para demostrar las **diferentes formas de trabajar** con UIStackViews.

Cuando lo hayas terminado, **tu aplicación** tendrá este aspecto:



Nota Legal: Game of Thrones es una marca registrada de **HOME BOX OFFICE, INC.**. El uso de su marca en este Tutorial obedece únicamente a propósitos didácticos.

Si te fijas en la interfaz, **podría parecer** que tenemos 9 imágenes formando el conjunto de las casas de Juego de Tronos. En realidad no son 9 imágenes sino 3. **Vamos a utilizar 3 imágenes**, ya que cada imagen agrupa a 3 casas. Para que te hagas una idea, esta sería la **primera imagen**:



¿Por qué vamos a utilizar **3 imágenes en lugar de 9** como sería lo normal?

Para poder utilizar las 9 imágenes en nuestra interfaz habría que **complicar demasiado** este tutorial. Considero que **sería excesivo** para un Tutorial de Introducción.

Por ahora, el desarrollar esta aplicación te permitirá ver **las ventajas** que ofrecen las UIStackViews a la hora de crear **cualquier interfaz de este tipo**.

Podríamos crear esta aplicación sin utilizar ninguna UIStackView. Sin embargo, como vas a ver a continuación, utilizándolas **podremos desarrollar la app completa** sin tener que escribir ni una sola linea de código. Nos vamos a enfocar al 100% en el uso de **Interface Builder** para desarrollar nuestra interfaz.

3. ¿En qué consisten las UIStackViews?

¿Para qué sirve una UIStackView?

Una **UIStackView** es una subclase de **UIView**, cuya función principal es la de **actuar como contenedor de otras views**. Gracias a esta clase podemos **agrupar** fácilmente varios elementos de nuestra interfaz. Además, las views que añadas a la stack view **no necesitarán constraints**. Es la propia stack view la que se encarga de posicionar las views y definir **automáticamente** las constraints por ti. ¿No es genial? Podrás diseñar partes completas de interfaces sin tener que añadir constraints en los elementos que vayas utilizando.

No me malinterpretes, esto no quiere decir que puedas olvidarte por completo del Auto Layout. Todavía tendrás que definir **las constraints para las stack views** que utilices en tus interfaces. Aun así es una gran ventaja.

Imagina que necesitas crear una interfaz que contenga **12** views diferentes (Unas cuantas **UILabel**, algún **UIButton**, alguna **UIView**...). Si utilizas una **UIStackView** que contenga esas **12** views, **no tendrás que crear las constraints** para cada una de las **12** views, ya que la **UIStackView** se encarga de **gestionarlas automáticamente**. Supone un gran ahorro de tiempo.

Propiedades de las UIStackViews

Las **propiedades** de una **UIStackView** con las que tendrás que trabajar son fundamentalmente cuatro:

Axis

La propiedad **Axis** determina si las views contenidas en la **UIStackView** deben colocarse en posición **Vertical** u **Horizontal**

Distribution

La propiedad **Distribution** define el **tamaño y posición** de las views contenidas en la **UIStackView**. Por defecto toma el valor **Fill**, lo que significa que cada una de las views contenidas ocupan **todo** el espacio disponible en la **UIStackView**.

Alignment

La propiedad Alignment especifica como se alinearán las views contenidas en la UIStackView. Si quisiéramos que las imágenes **aparecieran centradas** en nuestra UIStackView, utilizaríamos la opción: **Center**.

Spacing

Por último la propiedad Spacing te permite **especificar el espacio** que habrá entre las views.

¿Cómo añadir una UIStackView a nuestros proyectos?

También debes saber que Xcode te permite utilizar una UIStackView en tu proyecto de **dos formas** diferentes:

1. **Arrastrando** una UIStackView (horizontal ó vertical) directamente a tu storyboard. Después podrás arrastrar y soltar labels, buttons, imageViews o cualquier otro tipo de view dentro de la UIStackView.
2. Utilizando la opción **Stack** situada en la barra de Auto Layout. Para hacerlo de esta forma, deberás seleccionar dos o más views y después pulsar en el **botón Stack**. Interface Builder incorporará las views dentro de la UIStackView y la redimensionará automáticamente.

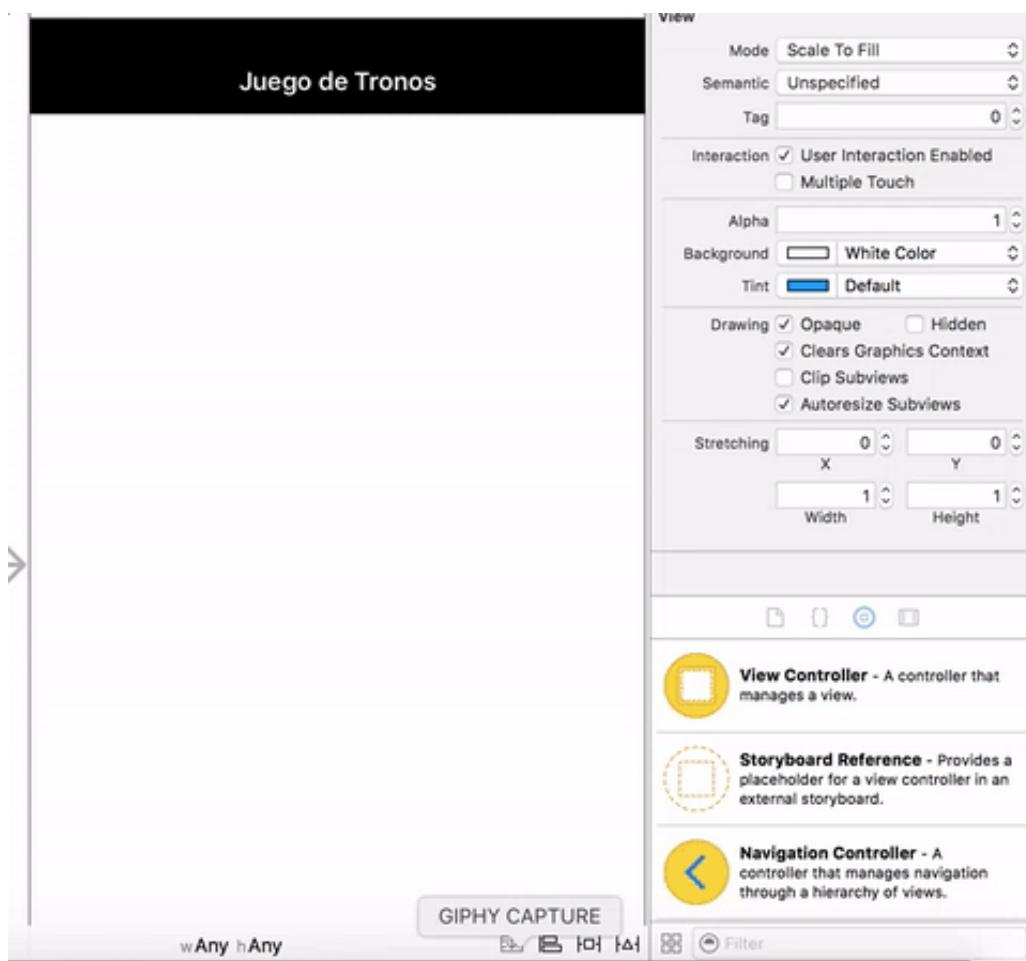
Explicado de esta forma es probable que no seas capaz de visualizar ambos métodos. No te preocunes, en este tutorial **utilizaremos ambos**, así podrás elegir el que más te convenga.

4. El Proyecto de Inicio

Para evitarte perder el tiempo creando un proyecto desde cero, en el que tendrías que añadir las imágenes que vamos a utilizar, he desarrollado un **Proyecto de Inicio** que puedes descargar [desde aquí](#). Es un proyecto muy simple. Unicamente consiste en un navigation controller y las imágenes que vamos a utilizar en la app.

5. Añadiendo nuestra UIStackView desde Interface Builder

Una vez que hayas descargado el proyecto, ábrelo y accede al **Main.storyboard**. Desde la librería de objetos, arrastra una **UIStackView vertical** al view controller que tiene el título en la parte superior: **Juego de Tronos**. Como has podido ver, puedes añadir dos tipos de UIStackView, **vertical y horizontal**. Cada una posicionará las views en un eje diferente. En nuestro caso elegimos la vertical porque queremos que nuestra interfaz muestre las imágenes **de forma vertical**.



Una vez que hemos añadido la UIStackView lo que haremos será **especificar las dimensiones** que queremos que tenga, utilizando para ello algunas constraints.

Recuerda lo que hemos dicho antes. Usar UIStackViews hace que no tengas que añadir las constraints para **todos** los elementos de la interfaz. Sin embargo, si que es necesario que especifiques las **constraints** de la propia **UIStackView**. Para nuestra UIStackView definiremos las

siguientes constraints:

- Margen Superior: 10
- Margen Derecho: 10
- Margen Izquierdo: 10
- Height: 70% de la height de su superview

Primero añadiremos las **constraints de posición** (Las 3 primeras). Para añadirlas, **selecciona la UIStackView**, haz clic en el **botón Pin** de la barra de Auto Layout y añade las 3 constraints de posición:



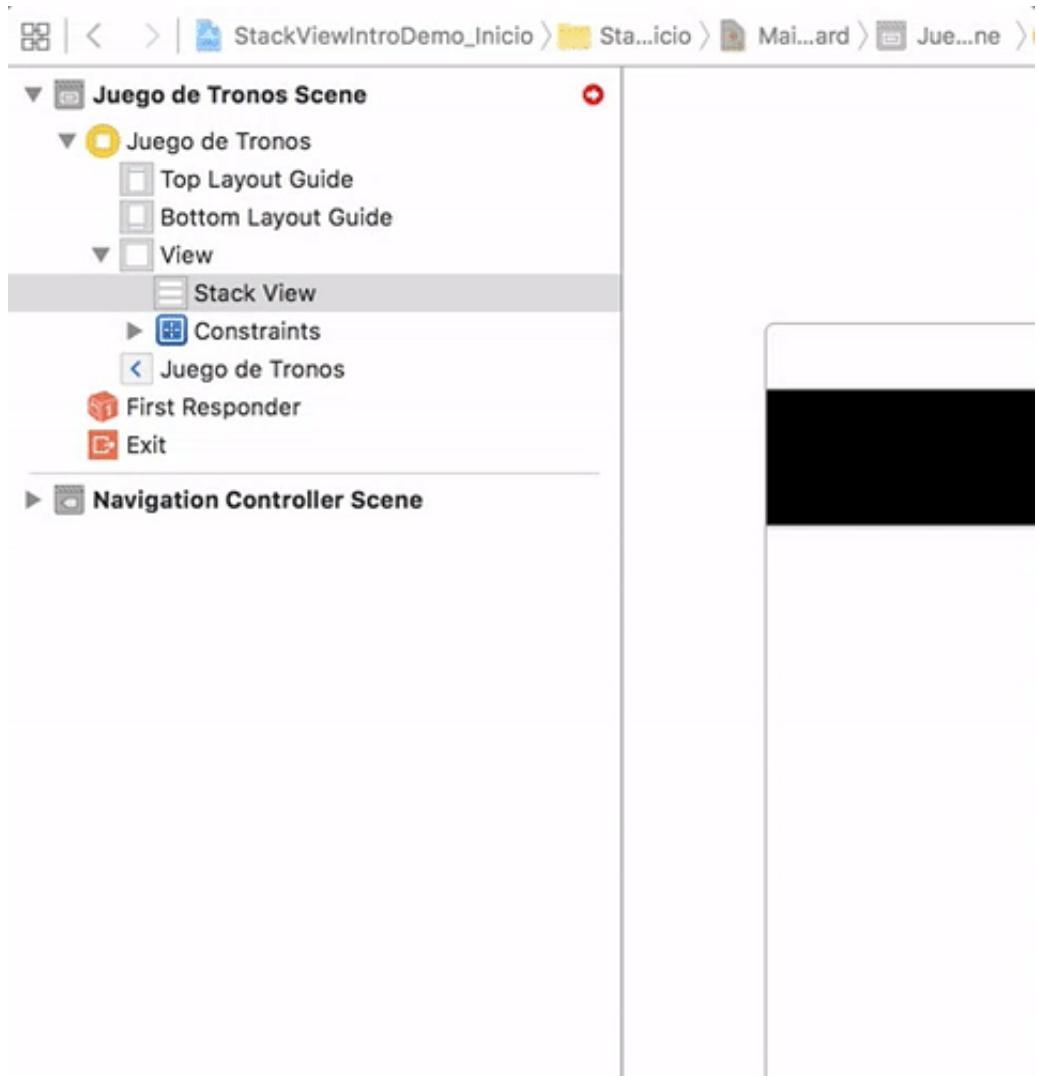
Antes de añadir constraints es mejor que **desactives** siempre la opción "**Constrain to margins**".

Como podrás ver, en cuanto añadas estas constraints, aparecerán en tu view controller, avisos de que **algo está fallando**. Podemos distinguir dos avisos diferentes:

- De color naranja (**Warnings**): Se producen porque nuestra UIStackView se encuentra en una **posición diferente** a la que hemos establecido en las constraints.
- De color rojo (**Errores**): Se produce porque **falta añadir una constraint** a la UIStackView (Height).

Vamos a solucionar primero nuestro **Error**.

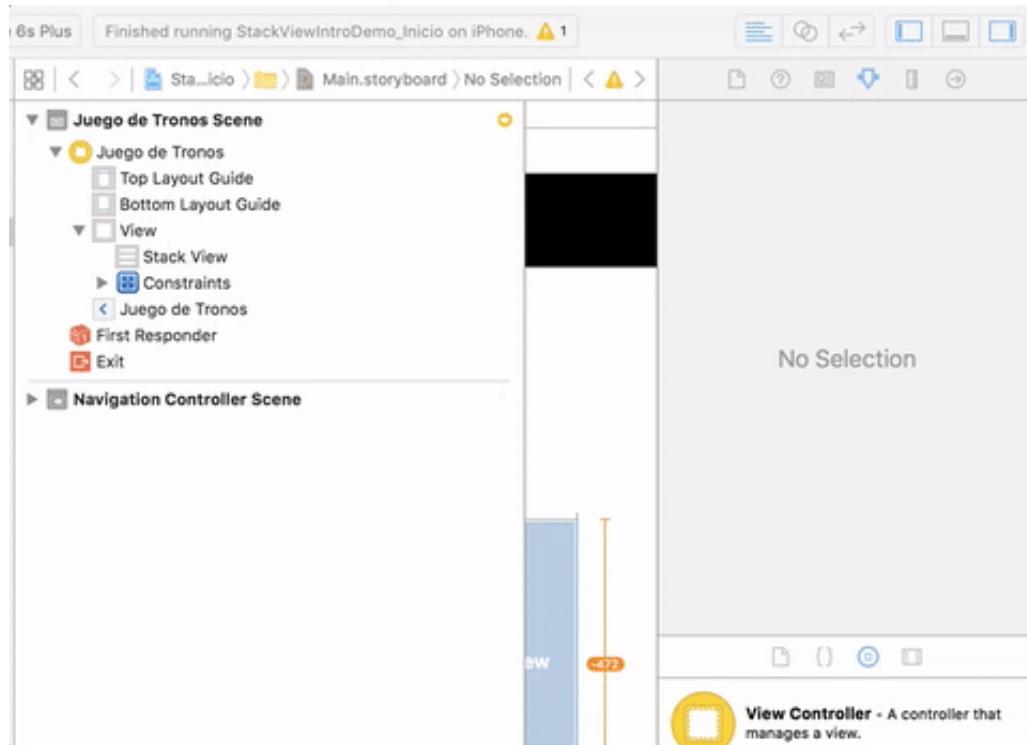
Para añadir la constraint de tamaño (**Height**), desde el Document Outline arrastra con la tecla Ctrl pulsada desde la UIStackView hasta la view principal. Al soltar justo encima de la view principal selecciona “**Equal Heights**”.



Como puedes ver, nuestro error (En color rojo) **desaparece**.

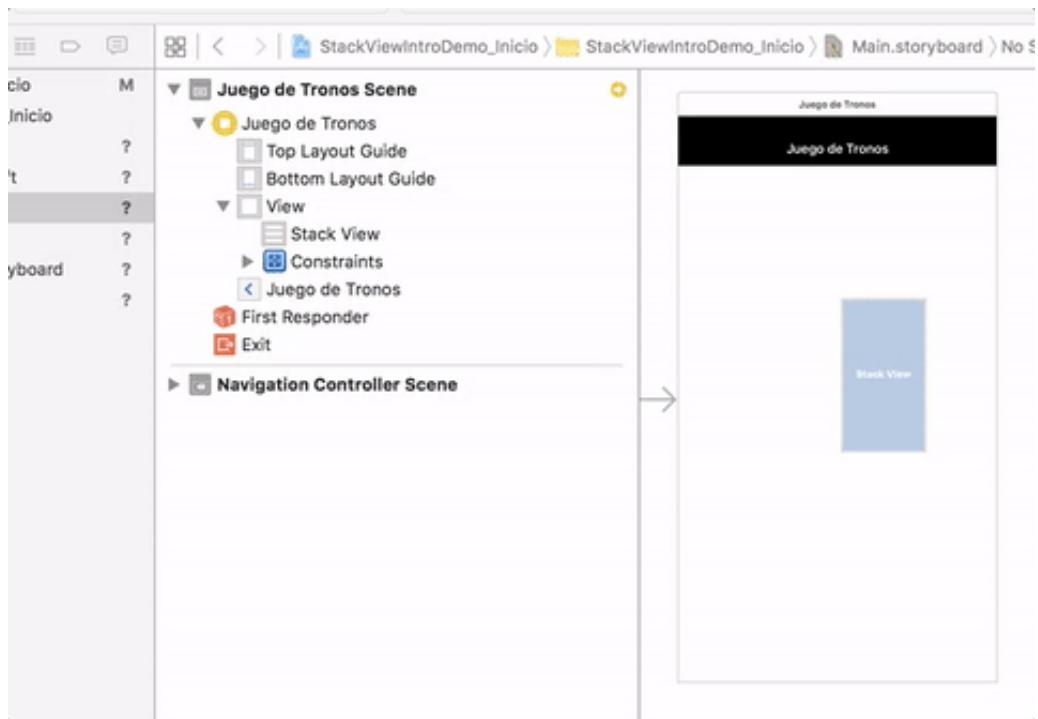
Esto hará que el height de nuestra UIStackView sea igual al height de la view. Sin embargo, lo que queremos es que el height de nuestra

UIStackView **ocupe solo el 70%** del height de nuestra view. Para ello selecciona la constraint “**Stack View.Height**” y a través del Inspector de Atributos cambia el valor del **Multiplier** de 1.0 a 0.7.



De esta forma, nuestra UIStackView tendrá un height del 70% de nuestra view, que **es lo que queríamos**.

En nuestro storyboard seguimos teniendo el warning que nos indica que la posición actual de la **UIStackView** no coincide con las constraints. Para solucionarlo, haz clic en el warning (Flecha amarilla) que aparece en el **Document Outline**, después pulsa en el triángulo amarillo que verás a continuación, selecciona la opción “**Update frames**” y pulsa en el botón **Fix Misplacement**.

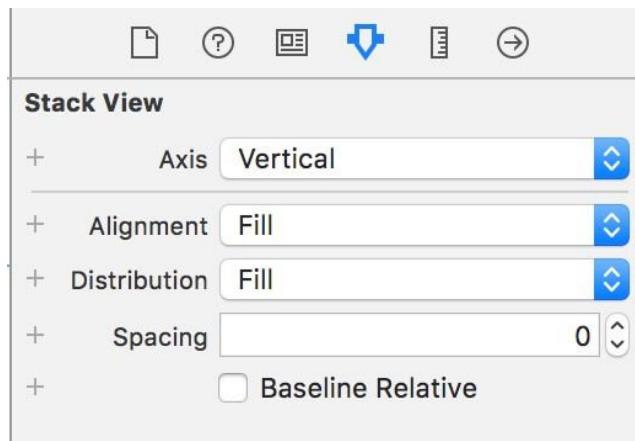


Verás como la `UIStackView` **automáticamente** adopta la posición y el tamaño que queríamos.

6. Configurando las Propiedades de nuestra `UIStackView`

Una vez que hemos añadido y configurado el tamaño y posición de nuestra `UIStackView`, podremos **configurar su apariencia** a través de sus **propiedades**.

Estas son las propiedades de nuestra `UIStackView` y los valores **que toma por defecto** al añadirla a nuestra interfaz:



La **propiedad Axis** determina si las views contenidas deben colocarse en posición vertical u horizontal. En nuestro caso, buscamos que las imágenes se coloquen verticalmente, por lo que dejaremos esta opción

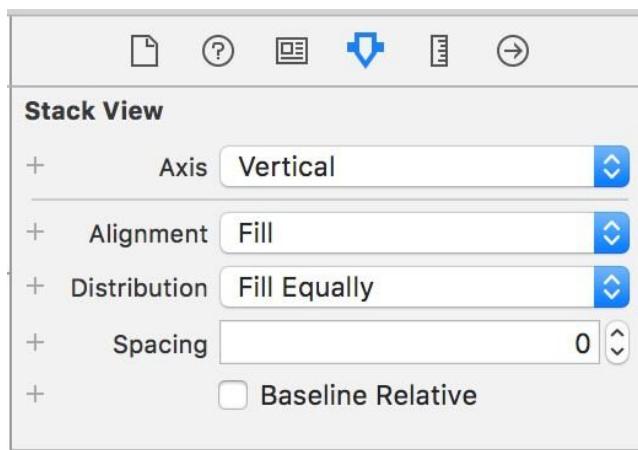
como **Vertical**.

La **propiedad Alignment** especifica como se alinearán las views contenidas en la UIStackView. Como queremos que las imágenes ocupen **todo el espacio** de la UIStackView dejaremos esta opción como **Fill**.

La **propiedad Distribution** define el **tamaño y posición** de las views contenidas. Por defecto toma el valor **Fill**, lo que significa que cada una de las views contenidas ocupan todo el espacio disponible en la UIStackView. No queremos eso, lo que queremos es que cada una de nuestras image views ocupen **el mismo espacio** que las demás, así que cambiaremos la opción a **Fill Equally**.

Por último la **propiedad Spacing** te permite establecer el espacio entre las views. En nuestro caso **no queremos** añadir espacio entre las views, así que puedes dejar su valor en 0.

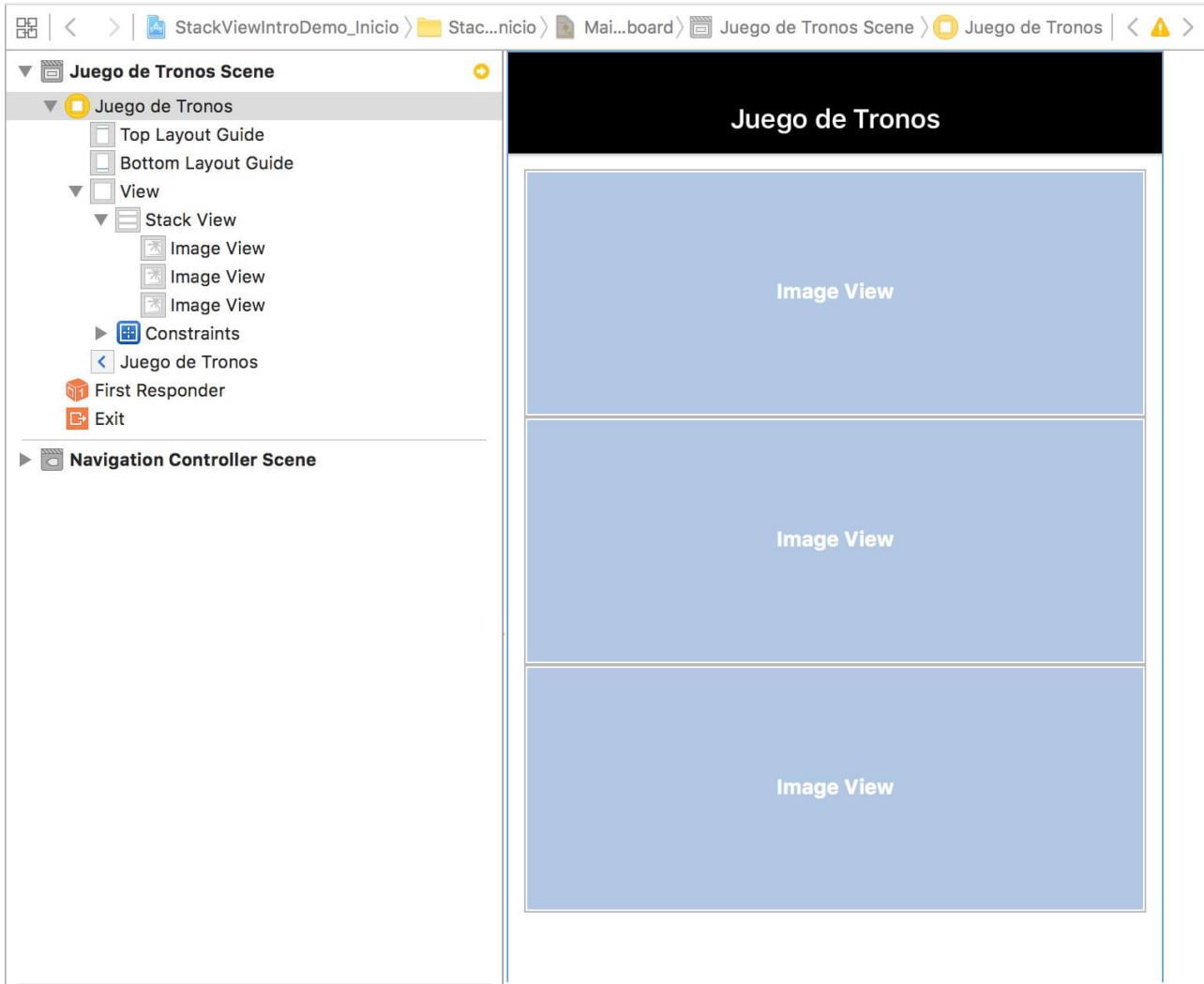
Estos deberían ser los valores finales para tu UIStackView:



7. Añadiendo las image views a nuestra UIStackView

Nuestro siguiente paso será añadir las imágenes a la UIStackView. Para ello, **arrastra una image view** de la librería de objetos **hasta la UIStackView**. En cuanto la añadas, verás como la image view se redimensiona automáticamente. Repite esta operación y **añade dos image views más**. Aquí es donde puedes ver la **utilidad de la UIStackView**. En cuanto vas añadiendo más views, éstas se van **colocando automáticamente** en posición vertical, ocupando

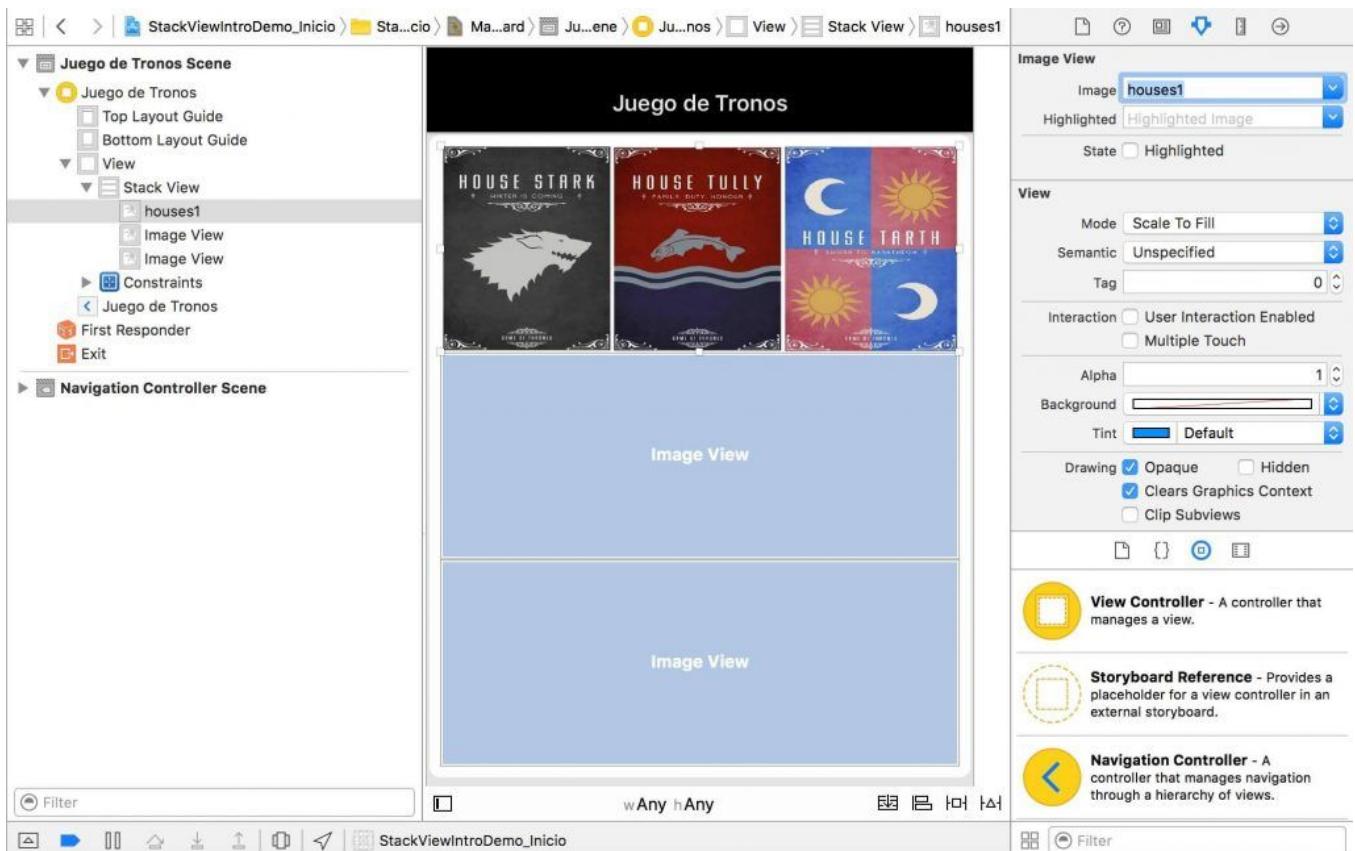
exactamente el mismo espacio unas que otras y definiendo sus constraints sin que tengas que hacer nada más.



8. Enlazando las imágenes de nuestra aplicación

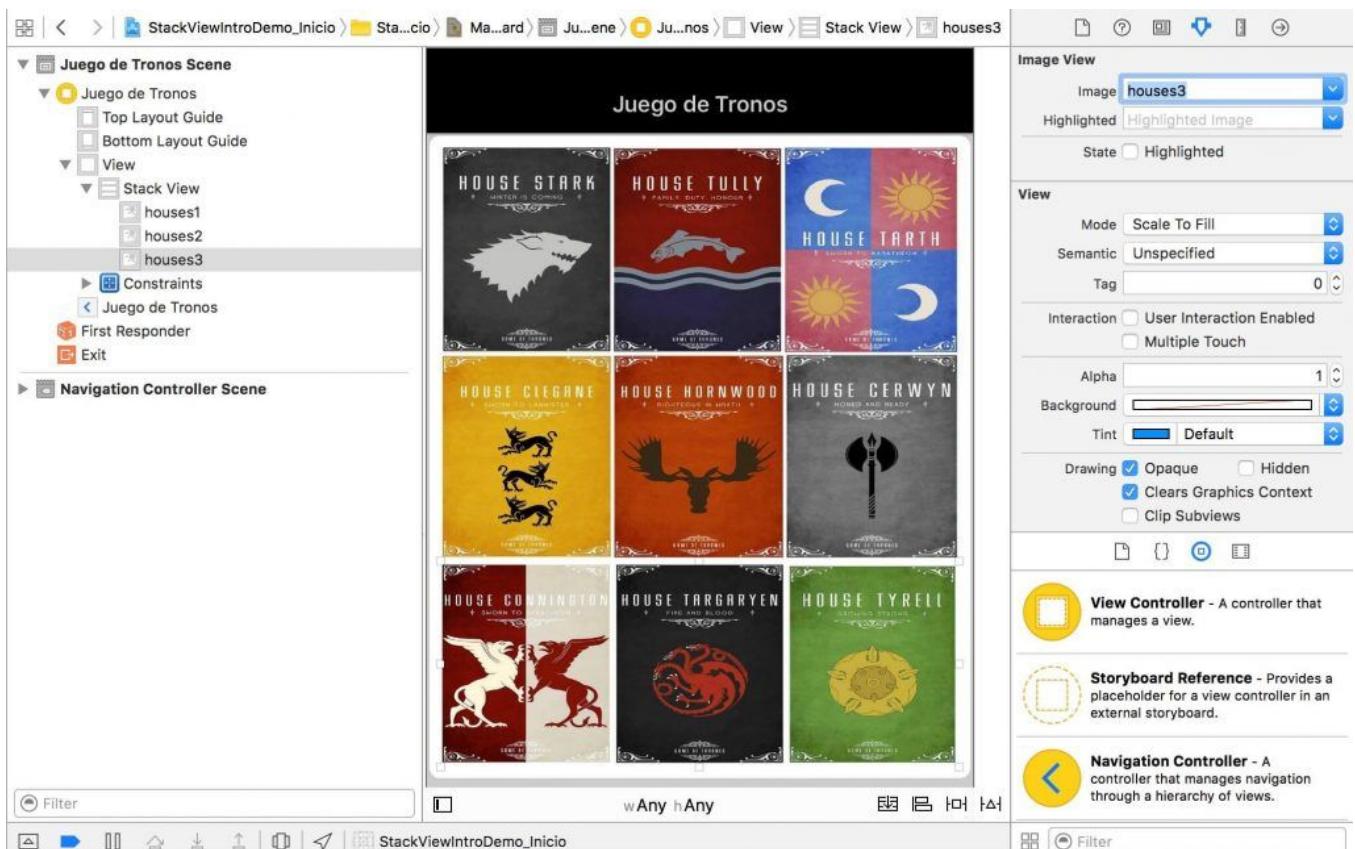
Una vez que hemos añadido nuestras image views, tendremos que especificar **que imágenes vamos a mostrar** en cada una de ellas.

Selecciona la primera image view y desde el **Inspector de Atributos** escribe en su propiedad **Image: houses1**.



Repite este proceso para el resto de image views, eligiendo las imágenes **houses2** y **houses3**.

El **aspecto** de tu interfaz debería ser este:



Ahora ejecuta la aplicación y comprueba que **se muestra perfectamente** en los diferentes simuladores de Xcode. Nuestra UIStackView especifica las constraints por nosotros.

9. Añadiendo más elementos a nuestra aplicación

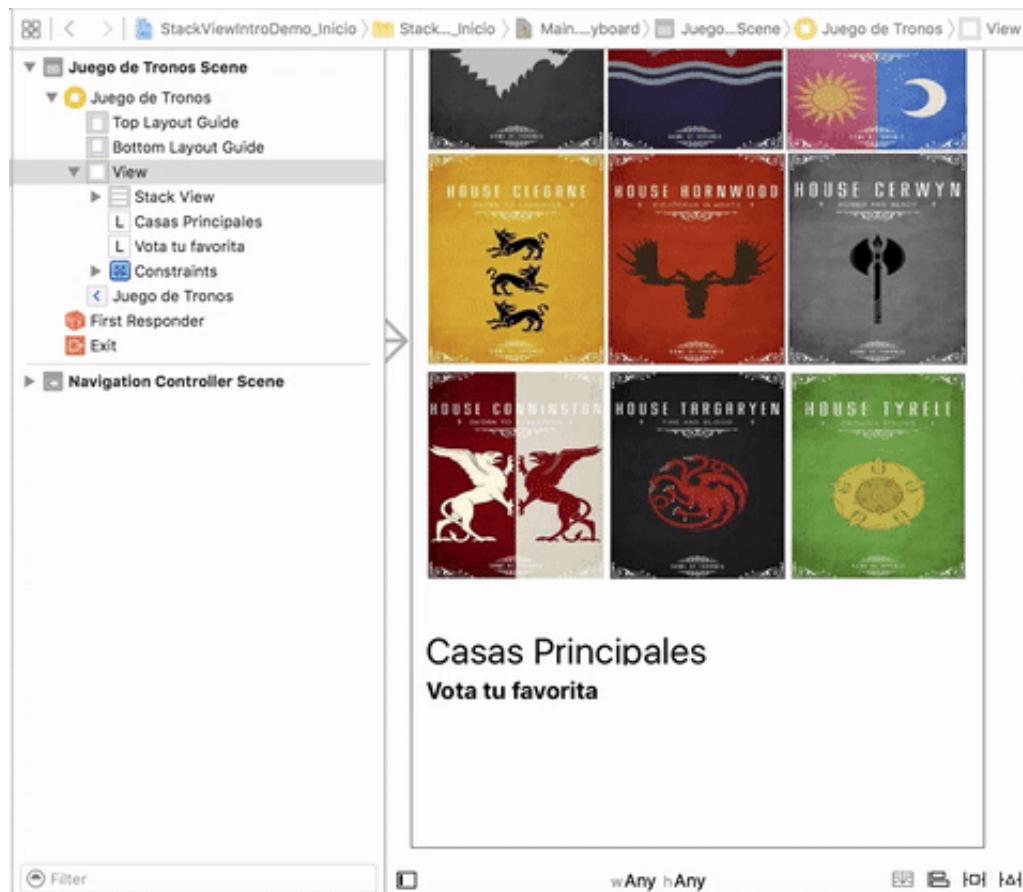
Acabas de ver como podemos añadir image views a nuestra UIStackView. Pero como hemos comentado al principio del tutorial, podemos añadir **cualquier tipo de view**.

Lo que vamos a hacer a continuación es añadir **dos labels y dos buttons** a nuestra interfaz.

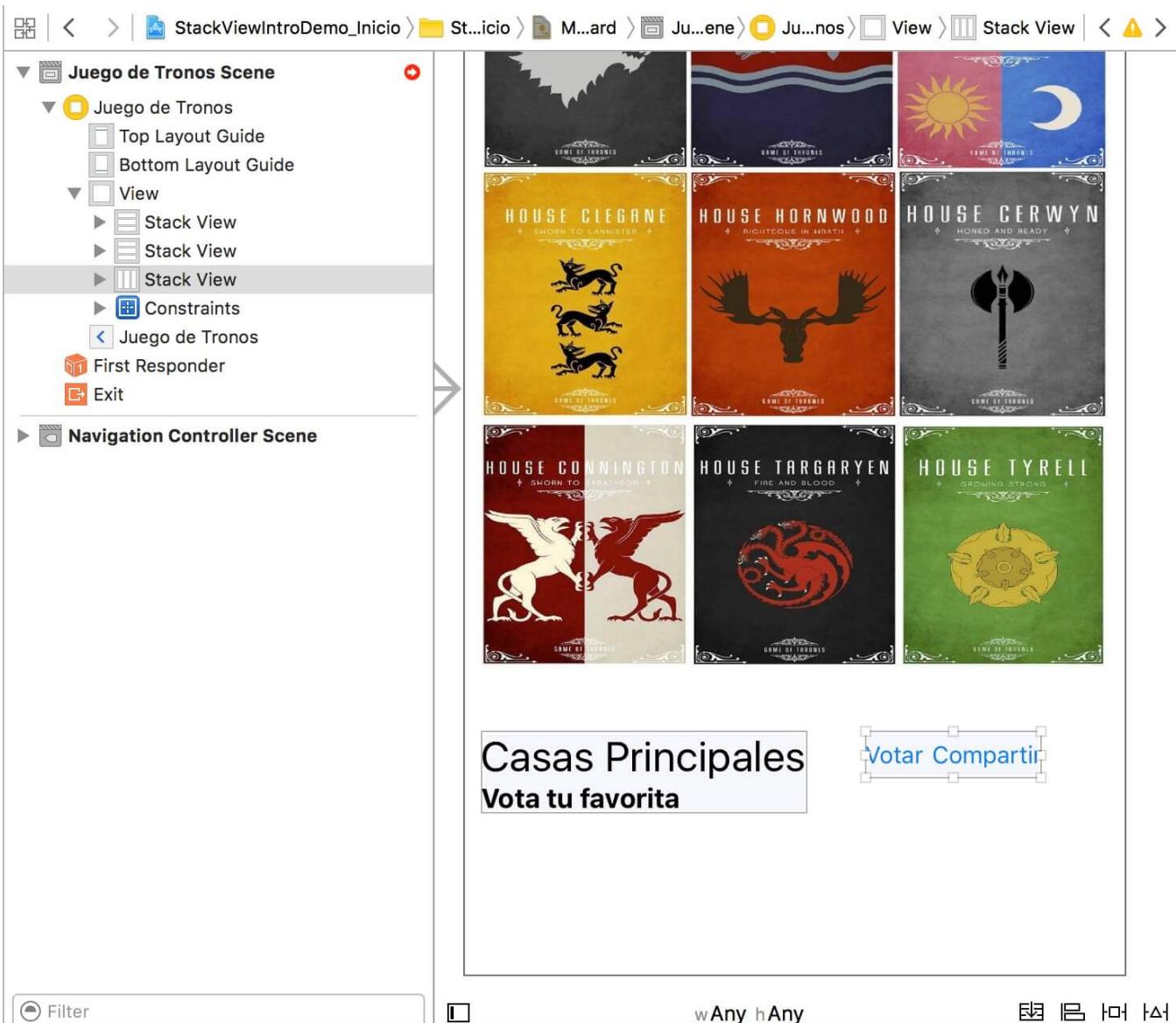
Para ello, desde la Librería de Objetos de Interface Builder arrastra un label **a la view**. OJO: Date cuenta que he dicho a la view, no a la UIStackView. Ahora verás por qué. Escribe en la label “**Casas Principales**” y colócala justo debajo de la UIStackView. Haz la label un poco más grande, cambiando su fuente a 26 puntos. Después arrastra otra label justo debajo de la anterior y escribe “**Vota tu Favorita**”. En las opciones de formato de esta label selecciona **Bold**, para que el texto aparezca en negrita.

Si recuerdas, al comienzo del tutorial dijimos que existían **dos formas** de utilizar las UIStackViews. Ya hemos visto como añadir elementos a una UIStackView arrastrándolos directamente dentro de ella. Ahora veremos como podemos añadirlos **a través del botón Stack**.

Dejando la **tecla cmd** pulsada haz clic en ambas labels. Una vez que estén seleccionadas las dos, pulsa en el botón **Stack**, situado en la barra de Layout. Interface Builder automáticamente añadirá las labels dentro de una **UIStackView vertical**.

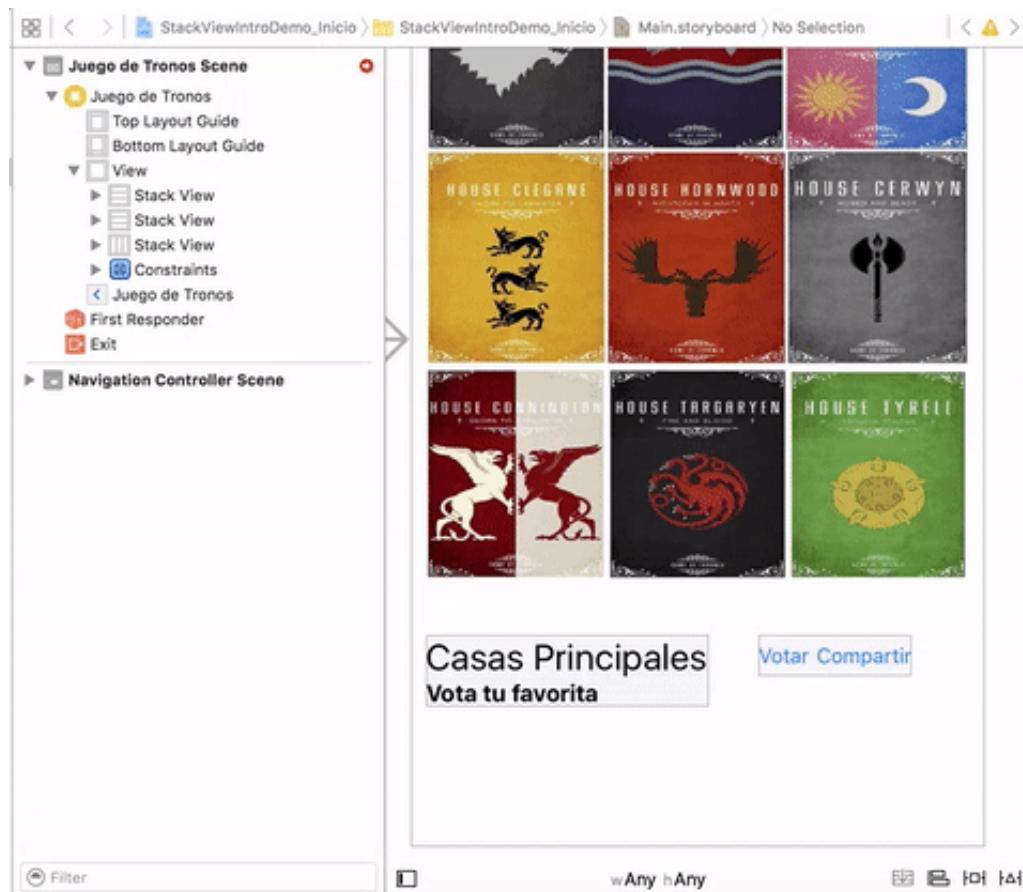


A continuación **añade dos buttons** a la view. Colócalos en la parte inferior uno junto al otro. Escribe en el primero “**Votar**” y en el segundo “**Compartir**”. Vamos a añadir estos dos buttons a otra UIStackView, así que repite el proceso anterior. Selecciona ambos con la tecla cmd pulsada y haz clic en el botón **Stack**. Como ves, ambos han sido añadidos a una UIStackView horizontal. Añade **un espacio de 5 puntos** entre ellos a través de la propiedad **Spacing**.



Antes hemos comentado que las UIStackViews pueden **anidarse unas dentro de otras**. Este es otro de los puntos fuertes de las UIStackViews. Te permiten **contener unas dentro de otras** para conseguir la interfaz que buscas. Vamos a verlo con un ejemplo. Vamos a añadir la UIStackView que contiene los buttons y la UIStackView que contiene las labels dentro de una nueva UIStackView.

Para ello **selecciona las dos UIStackViews** dejando la tecla cmd pulsada y haz clic en el botón Stack. Verás como **se crea una nueva UIStackView** que envuelve a ambas UIStackViews.



Los buttons deben estar alineados por su parte inferior con la label, así que selecciona la **UIStackView** y cambia su propiedad **Alignment** de Fill a **First Baseline**. Además cambia la propiedad **Spacing** a 20 para que haya algo de distancia entre la label y los buttons.

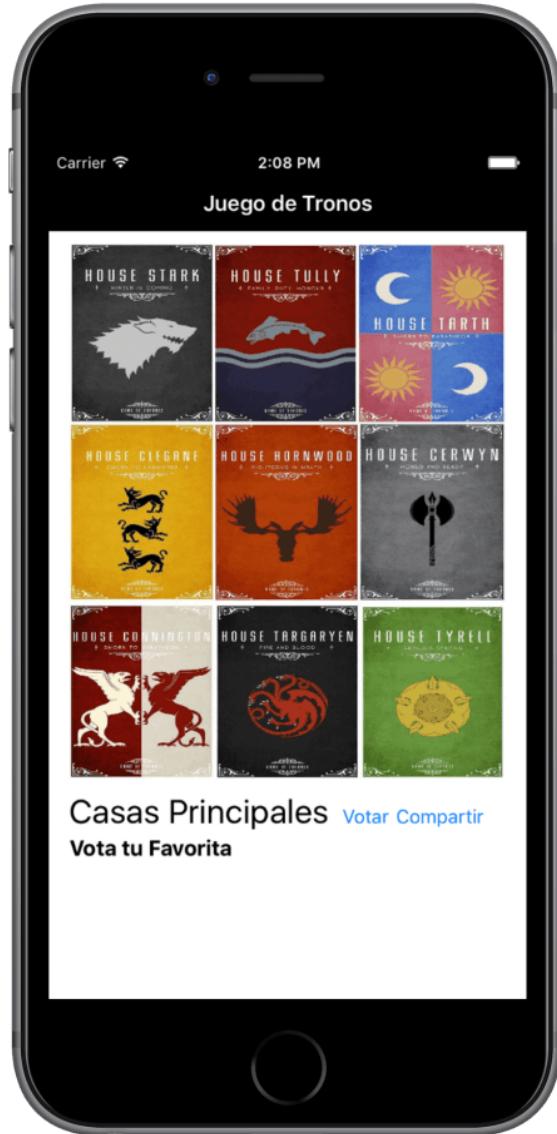
Como puedes ver, solo hemos tenido que utilizar UIStackViews anidadas para diseñar **de forma exacta** nuestra interfaz.

Para terminar, tendremos que **especificar las constraints** entre la UIStackView que contiene las image views y la UIStackView que contiene los buttons y los labels. Selecciona la UIStackView que contiene los buttons y labels y a través del **botón Pin** añade las siguientes constraints:

- Margen superior: 8 puntos
- Margen izquierdo: 0 puntos
- Margen derecho: 0 puntos



¡Perfecto! **Has terminado** el diseño de la interfaz de nuestra aplicación. Ahora **ejecuta el proyecto** y échale un vistazo al resultado final. Si has seguido **paso a paso** el tutorial, el aspecto de tu aplicación debería ser el siguiente:



10. Resumen Final

En este Tutorial de Introducción a UIStackView hemos visto los siguientes puntos:

- ¿Qué son las UIStackViews?
- ¿Qué propiedades debes controlar para configurar una UIStackView?
- Dos formas diferentes de añadir UIStackViews a tus proyectos
- ¿Cómo añadir views a una UIStackView?
- ¿Cómo anidar UIStackViews unas dentro de otras?
- Ventajas que te ofrecen las UIStackViews en el diseño de tus interfaces

Tutorial de Introducción a Core Data [Parte 1]

Aprende a utilizar este Sistema de Persistencia en Swift

Lenguaje Swift | Nivel Principiante

1. Introducción

Este tutorial lo vamos a dedicar a conocer el funcionamiento de **Core Data**.

Vamos a ver de forma rápida la **teoría** que necesitas conocer y después vas a poder desarrollar una **aplicación iOS** paso a paso donde verás como implementar Core Data.

De esta forma cuando tengas que desarrollar algún proyecto **por tu cuenta** podrás usar lo que aprendas en este tutorial para ser capaz de desarrollar **una app** que integre Core Data perfectamente.

2. ¿Qué vamos a ver en este Tutorial?

Estos son los **puntos más importantes** en los que nos centraremos:

- Familiarizarte con el concepto de Persistencia
- ¿En qué consiste Core Data?
- Conceptos Fundamentales que debes comprender
- Crear el Modelo de Datos de tu app utilizando el Editor de Xcode
- Guardar información en Core Data

- Obtener información de Core Data
- Mostrar esa información obtenida en una Table View
- ¿Cómo integrar Core Data en una Aplicación iOS?

Al terminar este tutorial habrás multiplicado tus **poderes de desarrollador por 20** y estarás preparado para integrar este **sistema de persistencia** en tus apps.

Este Tutorial corresponde a la primera parte de un Tutorial doble de Introducción a Core Data

3. Echando un vistazo a nuestra App

En ese tutorial, vamos a desarrollar una **App de Gestión de Tareas** que almacene y recupere el listado de tareas de Core Data.

4. El Concepto de Persistencia

Hemos mencionado antes que Core Data es un **sistema de persistencia**. Por tanto, lo primero que deberíamos ver, es en qué consiste la **Persistencia**.

Una posible **definición** podría ser esta:

Persistencia es la acción de **preservar** la información de un objeto de forma permanente (**guardarlo**), unido a la posibilidad de poder

recuperar la información del mismo (**leerlo**) para que pueda ser nuevamente utilizado.

Dicho de otro modo, la persistencia nos permite **guardar los datos** de nuestras aplicaciones para poder **utilizarlos** cuando el usuario **vuelva a ejecutar** nuestra app.

Si desarrollamos una aplicación que no integre **ningún sistema de persistencia**, todos los datos que vayamos utilizando durante la ejecución de la misma simplemente se **escribirán en memoria**. ¿Qué quiere decir esto? Quiere decir que al estar únicamente almacenados en memoria y no en disco, en la siguiente ejecución de la app, **todos estos datos se perderán**.

En Desarrollo iOS existen varios **sistemas de persistencia**, diferentes “**herramientas**” que podemos utilizar como desarrolladores para guardar los **datos** de nuestras aplicaciones.

Aquí tienes algunos de ellos:

- NSUserDefaults
- Property Lists
- NSFileManager
- SQLite
- Core Data

Nosotros, en este tutorial nos vamos a centrar en **Core Data**. Es probable que en próximos tutoriales veamos **otros sistemas** de persistencia que puedes utilizar.

5. ¿En qué consiste Core Data?

Core Data es un **Framework de Persistencia** desarrollado por Apple que nos permite simplificar la gestión del **modelo de datos** de nuestras aplicaciones.

Core Data suele utilizarse a través de una **base de datos** de tipo **SQLite**.

Pero el valor que realmente nos aporta, son una **serie de herramientas** que nos facilitan tanto la **creación** del modelo de nuestra app como la **gestión** posterior desde nuestro código.

Además, esta **capa** que envuelve nuestra **base de datos SQLite** nos permite que nosotros como desarrolladores **no** tengamos que trabajar directamente **con sentencias SQL**.

Una vez que sabemos en qué consiste Core Data, veamos los **conceptos** más importantes que debes dominar.

6. Conceptos Fundamentales

NSManagedObjectModel

Es la **representación** de nuestro **Modelo** en disco.

NSManagedObject

El objeto NSManagedObject representa un **objeto único** almacenado en Core Data.

NSManagedObjectContext

NSManagedObjectContext representa algo parecido a un “**espacio de memoria temporal**” donde poder trabajar antes de guardar los datos.

Si piensas en **como guardar** un objeto con Core Data, podríamos decir que se trata de un **proceso** de dos pasos. Primero insertas el objeto en el **managed object context** y una vez que estás seguro puedes **confirmar el guardado** del objeto almacenándolo en disco.

Xcode genera **automáticamente** un managed object context en nuestras aplicaciones siempre que activemos la opción **Use Core Data** al crear el proyecto.

Concretamente el managed object context se almacena en una propiedad del **appDelegate**, por lo que cuando necesites utilizarlo lo primero que

tendrás que hacer será **crear una referencia** al appDelegate de tu aplicación.

NSEntityDescription

El objeto **NSEntityDescription** describe una **Entidad** en Core Data. Una instancia de NSEntityDescription determina el nombre de la entidad, sus atributos y relaciones y la clase por la que está representada.

NSFetchRequest

NSFetchRequest es la clase responsable de **recuperar datos** de Core Data. Para recuperar estos datos, utilizaremos **peticiones** a las que especificaremos una serie de criterios. Estas peticiones son bastante potentes. Puedes utilizar **fetchRequest** para recuperar un **conjunto de objetos** que cumplan unas determinadas condiciones. Por ejemplo: “Dame todos los usuarios que se hayan dado de alta en el último mes y que hayan realizado alguna publicación en nuestra app”. NSFetchRequest utiliza **calificadores** para filtrar los resultados que queremos obtener.

5. Demostración práctica de la utilidad de Core Data

Para comenzar, puedes **descargar** el proyecto de inicio [desde aquí](#).

Una vez descargado, descomprímelo y **ábrelo** en Xcode.

Lo primero que debes comprobar, como en cualquier otro proyecto, es que **compila** correctamente (cmd + B).

Cuando hayas visto que compila sin problema, **ejecuta** la app (cmd + R).

Lo que quiero, es que veas tu mismo el **problema** que tiene esta aplicación, al no disponer de ningún sistema de **persistencia**.

Vamos a seguir unos pasos, que **cualquier usuario** podría llevar a cabo

en su iPhone si estuviera utilizando nuestra aplicación.

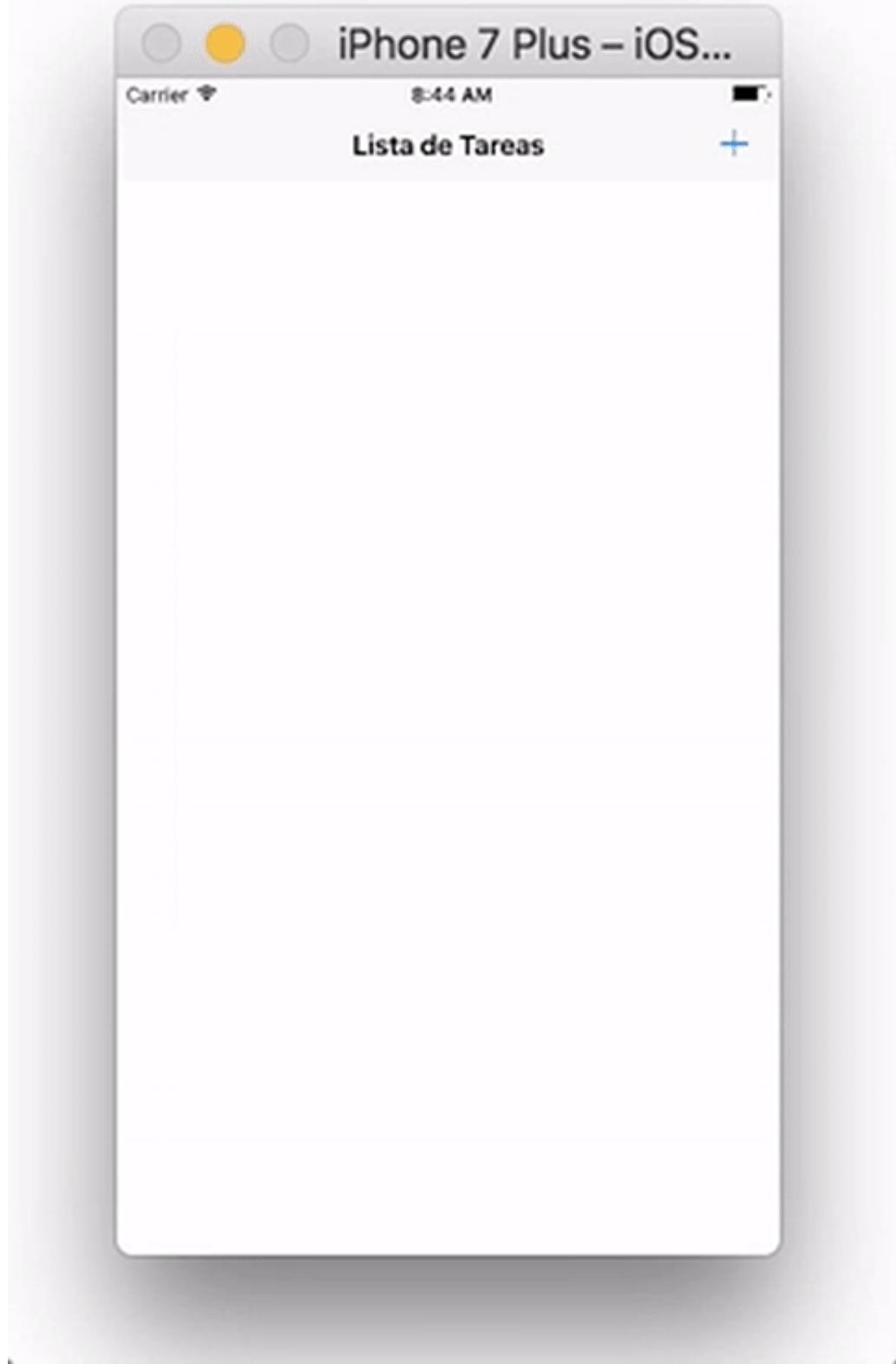
Vamos a acceder a la **multitarea** y a **cerrar** nuestra aplicación desde allí.

Para ello, **con la app ejecutándose**, quiero que añadas **2 ó 3 tareas** a nuestra aplicación. Ya sabes, para ello, utiliza el **botón +** y dales los nombres que quieras a las tareas y luego pulsa **Guardar**.

Una vez que estas tareas se hayan añadido a nuestra **table view**, pulsa **dos veces** seguidas la siguiente combinación de teclas (cmd + ⌂ + H). Esta combinación simula que el **usuario** ha pulsado dos veces seguidas el **botón Home** de su iPhone y le permite **acceder a la multitarea** del dispositivo.

Una vez lanzada la multitarea, **desplaza la ventana** de nuestra aplicación para **cerrarla**. Esto producirá que la app **finalice** su ejecución.

A continuación, desde Xcode vuelve a **lanzar la app** (cmd + R). Al volver a ejecutar la aplicación, verás que el listado de tareas que habíamos creado **ha desaparecido**, nuestra table view vuelve a estar **vacía**.



Esto se debe a que no hemos implementado **ningún sistema de persistencia** y por tanto, todos nuestros datos se guardan **únicamente en memoria**. Y como bien sabes, la memoria **se vacía** al finalizar la ejecución de un programa.

De ahí, la importancia de ser capaz de utilizar **sistemas de persistencia**

en tus aplicaciones iOS.

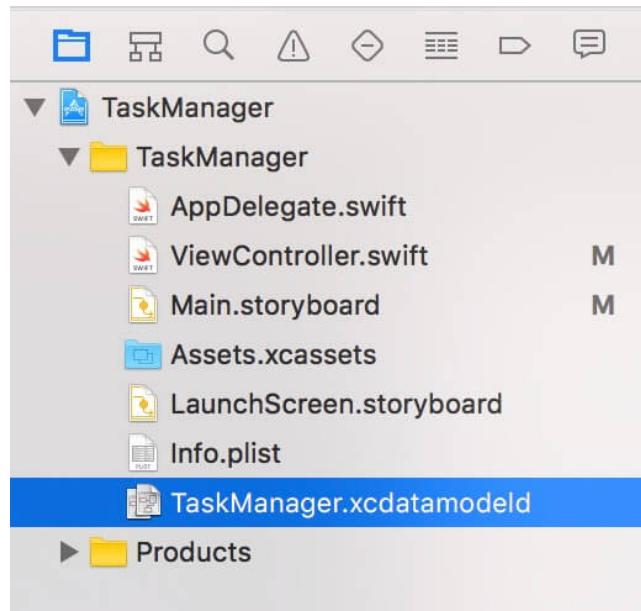
Ahora sí, vamos a ver los **cambios** que tendríamos que hacer **en nuestro código** para poder integrar Core Data.

7. Creando el Modelo de nuestra aplicación

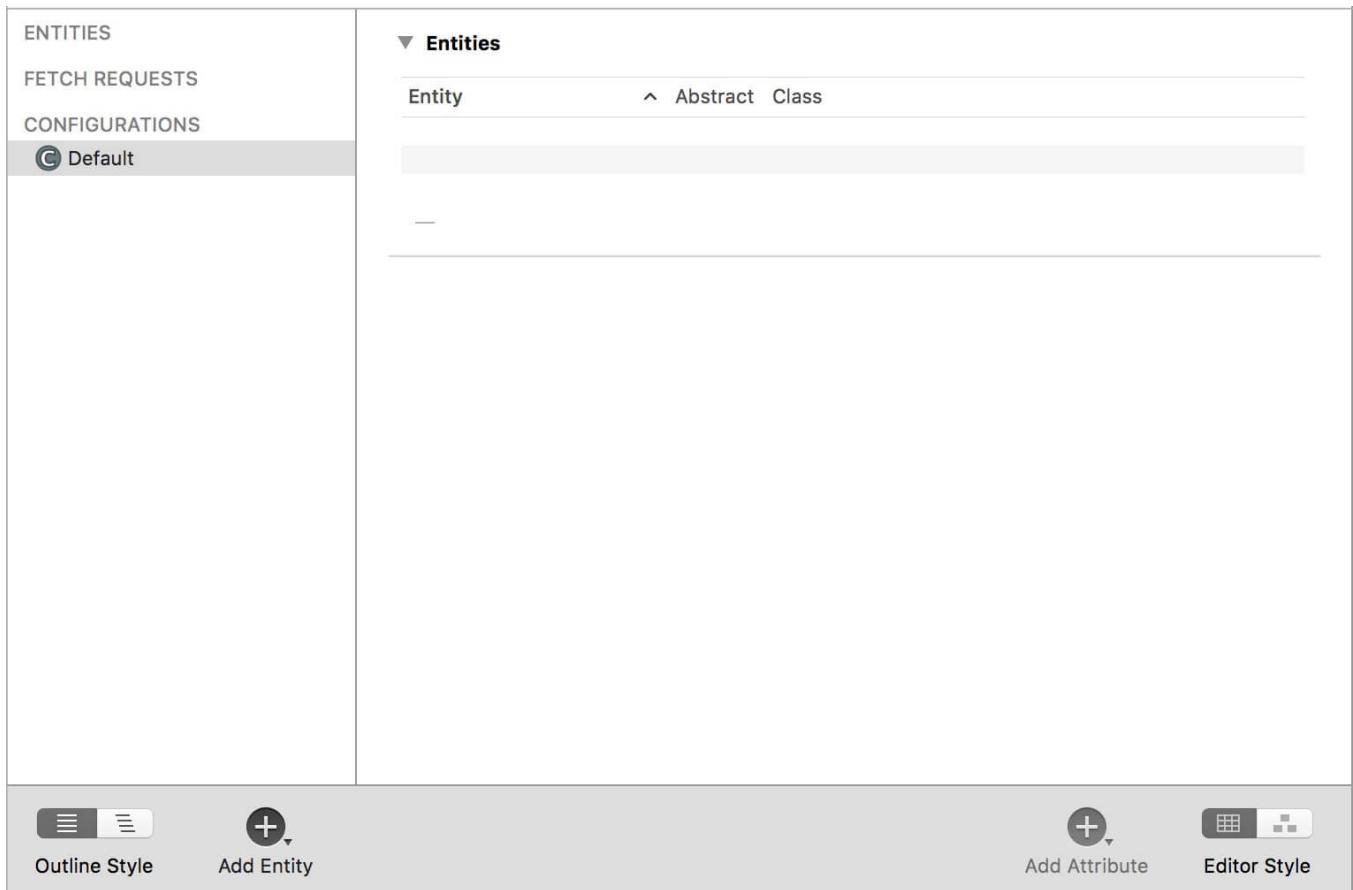
Antes de entrar directamente a **modificar** el código de la app, vamos a **crear el modelo** de nuestra aplicación.

El primer paso será crear un **Managed Object Model**, que representa a nuestro **modelo** en Base de Datos.

Como, al crear nuestro proyecto, activamos la **opción Core Data**, Xcode **automáticamente** ha creado un **fichero** que representa nuestro modelo de datos y que se llama **TaskManager.xcdatamodeld**.



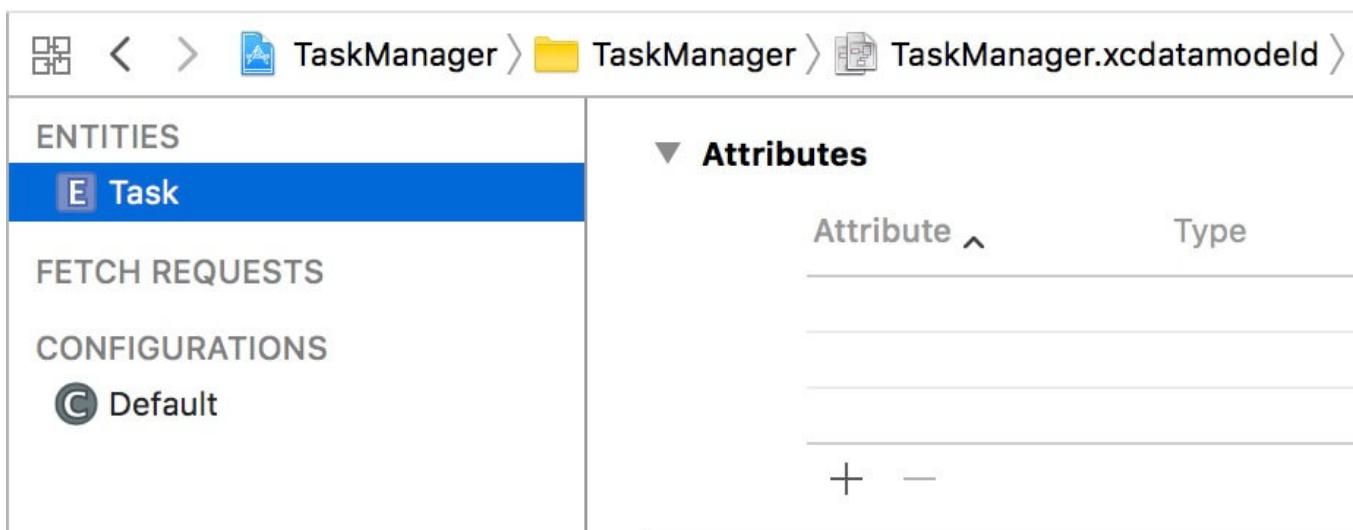
Haz clic en **TaskManager.xcdatamodeld** para abrirlo. Verás el **editor** que Xcode te ofrece para crear nuestro modelo.



Este **editor** ofrece un gran número de opciones. Por ahora nos centraremos en **crear una nueva Entidad**.

Haz clic en el botón **Add Entity** situado en la parte inferior izquierda para crear una **Entidad**. Haz doble clic en la Entidad que acabas de crear para cambiarle el nombre.

Dale el nombre **Task**.



Le damos este nombre porque va a representar a **cada una de las tareas**

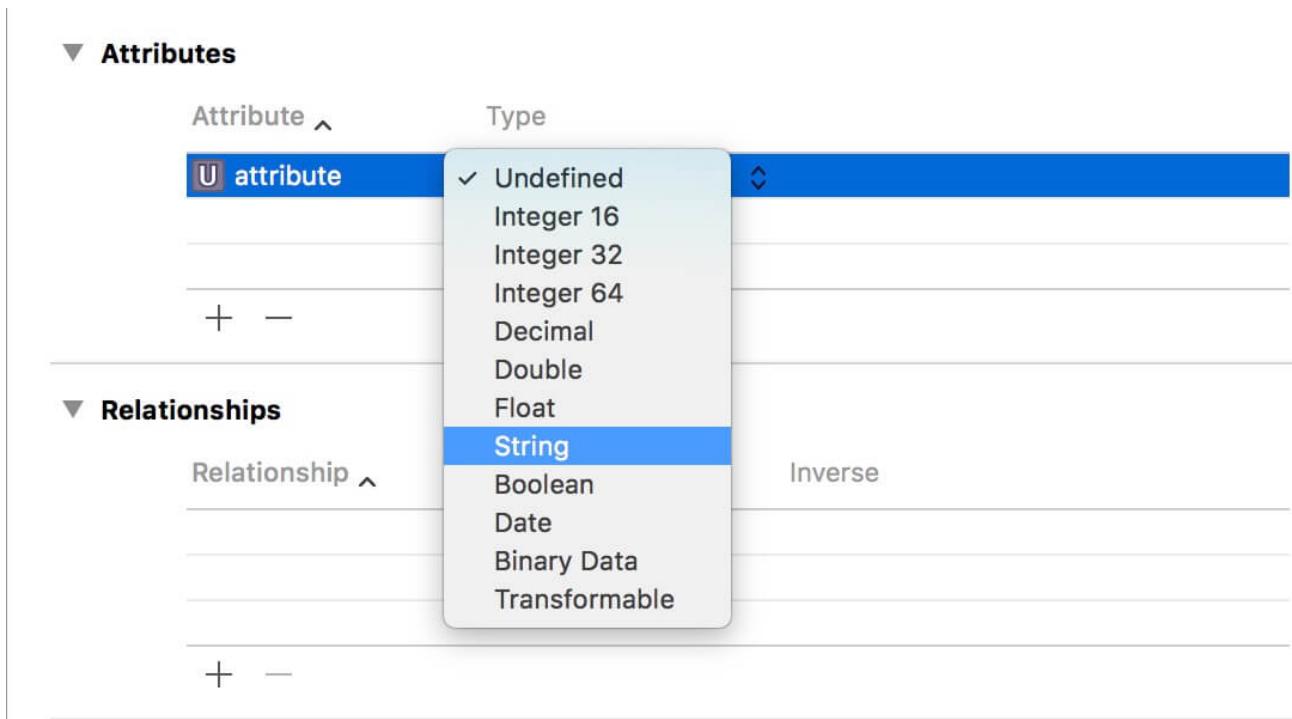
que vamos a ir creando en nuestra aplicación.

Si no tienes claro a que nos referimos cuando hablamos de una **Entidad**, aquí tienes una serie de **conceptos** relacionados con **Core Data** que deberías de conocer:

- Una **Entidad** es la representación de una Clase en Core Data. En una base de datos relacional correspondería a una Tabla. Por ejemplo, si quisiéramos crear una app como Instagram, podríamos tener una entidad llamada Usuario que representaría toda la información relativa a los usuarios de nuestra aplicación.
- Un **Atributo** es un tipo de dato concreto, asociado a una Entidad específica. Por ejemplo, en el caso de que tuviéramos una entidad llamada Usuario, podríamos tener como atributos: username, email, name, etc. En una base de datos relacional, un atributo de una entidad se correspondería con un campo de una tabla.
- Una **Relación** es un enlace entre diferentes entidades. Las relaciones pueden ser de 1-1 o de 1-N. Una relación 1-1 sería por ejemplo: Un usuario tiene una única imagen de perfil en Instagram. Una relación 1-N sería por ejemplo: Un usuario tiene asociadas varias publicaciones en Instagram.

Ahora que ya sabes lo que es un **atributo**, podemos crear uno nuevo en la entidad Task. Desde el Editor del Modelo de Xcode, selecciona la entidad **Task** y pulsa en el botón situado en la parte inferior derecha **Add Attribute**.

Dale el nombre **name** y cambia su tipo a **String**, pulsando en el desplegable situado bajo **Type**.



Como has visto en el **menú desplegable**, podemos especificar **diferentes tipos** para un atributo. Esto puede ser muy útil cuando crees modelos **más completos**.

8. Resumen Final

Hemos visto en este tutorial todos los **conceptos** que debes conocer para trabajar con Core Data. Además has visto el **problema** que tiene nuestra aplicación de gestión de tareas al no tener ningún sistema de **persistencia** implementado. También hemos creado el **modelo** de nuestra aplicación a través del editor que nos ofrece Xcode.

Este tutorial corresponde a la **primera parte** de un tutorial doble.

A continuación podrás continuar con la **segunda parte** de este tutorial.

Al **terminar** la segunda parte de este tutorial habrás visto **de forma completa** como añadir **Core Data** a cualquier aplicación.

Tutorial de Introducción a Core Data [Parte 2]

Aprende a utilizar este Sistema de Persistencia en Swift

Lenguaje Swift | Nivel Principiante

1. Introducción

En el **tutorial anterior** vimos la primera parte de **Introducción a Core Data**.

Vamos a continuar con la **segunda parte** del tutorial.

En la primera parte vimos sobre todo **conceptos teóricos** y comenzamos a crear el **modelo** de nuestra aplicación.

Si no has visitado la **primera parte** de este tutorial, te recomiendo que lo hagas antes de continuar con este.

Si por el contrario **ya visitaste el tutorial** anterior, vamos a continuar exactamente donde lo dejamos, **¡adelante!**

2. ¿Qué vas a aprender en este Tutorial?

Estos son los **puntos más importantes** en los que nos centraremos:

- Familiarízate con el concepto de Persistencia
- ¿En qué consiste Core Data?
- Conceptos Fundamentales que debes comprender
- Crear el Modelo de Datos de tu app utilizando el Editor de Xcode
- Guardar información en Core Data
- Obtener información de Core Data
- Mostrar esa información obtenida en una Table View

- ¿Cómo integrar Core Data en una Aplicación iOS?

3. Echando un vistazo a nuestra App

La aplicación que vamos a **continuar desarrollando** consiste en una **App de Gestión de Tareas**.

Si recuerdas, en la primera parte del tutorial, habíamos **creado el modelo** de nuestra aplicación a través del **editor de Xcode**.

Nuestro siguiente paso será **guardar las tareas** de la aplicación en Core Data.

Esto es lo que vamos a ver **a continuación**.

4. Guardando nuestras Tareas en Core Data

Para poder trabajar con el framework Core Data, lo primero que debes hacer es **importarlo** en tu proyecto.

Abre **ViewController.swift** y añade la siguiente linea, justo después de **import UIKit**:

Lo siguiente que debes hacer es **sustituir** la linea donde declarábamos el array tasks:

por esta otra linea:

```
var tasks = [NSManagedObject]()
```

Realizamos este cambio porque pasamos de guardar **Strings** (Que contenían únicamente el nombre de cada tarea) a guardar **objetos** de la entidad **Task**. Estos objetos, que representan a nuestras tareas, deben de ser de tipo **NSManagedObject**, ya que como dijimos al principio del tutorial, es la forma que tiene Core Data de especificar cada uno de los objetos que va a almacenar.

Al realizar este cambio, tu proyecto **mostrará** algunos **errores**. Esto se debe a que para mantener la coherencia de nuestro código debemos seguir realizando **modificaciones**.

Sustituye tu método **cellForRowAt** por este otro:

```
func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCell(withIdentifier: "Cell")
    //Creamos un objeto task que recuperamos del array tasks
    let task = tasks[indexPath.row]
    //Con KVC obtenemos el contenido del atributo "name" de la task y lo añadimos a nuestra Cell
    cell!.textLabel!.text = task.value(forKey: "name") as? String
    return cell!
}
```

Como ahora en lugar de Strings, almacenamos objetos task, lo que hacemos es **recuperar las tareas** que hay en el array tasks y obtener el contenido de su atributo “**name**” y lo establecemos como texto de cada celda.

Si no estás familiarizado con el concepto **KVC** es probable que no entiendas la linea en la que seteamos el texto de la Cell. No te preocupes,

aquí tienes perfectamente **explicado** en que consiste KVC.

Hay otra linea en nuestro código que está dando error, concretamente en el método **addTask()**:

```
    @IBAction func addTask(sender: AnyObject){
        let alert = UIAlertController(title: "Nueva Tarea",
                                     message: "Añade una nueva tarea",
                                     preferredStyle: .alert)

        let saveAction = UIAlertAction(title: "Guardar",
                                      style: .default,
                                      handler: { (action:UIAlertAction) -> Void in

            let textField = alert.textFields!.first
            tasks.append(textField!.text!)
            self.tableView.reloadData() ⚠️ Cannot convert value of type 'String' to expected argum...
        })
    }
```

Sustituye esa linea por la siguiente:

```
    self.saveTask(name: textField!.text!)
```

Como ya no guardamos Strings en nuestro array tasks, hemos sustituido la linea donde **añadíamos** la tarea a nuestro **array** por una llamada al método **saveTask()**, que es donde realizaremos el **guardado** de nuestra tarea en **Core Data**.

Ahora, lo que haremos será crear el método **saveTask()** y veremos **paso a paso** lo que hay que hacer para guardar un **objeto task** en nuestro modelo de Core Data.

Añade este método justo después del método **addTask()**. No te preocupes si no entiendes el código, lo vamos a explicar enseguida:

```
func saveTask(nameTask:String){

    //1

    let appDelegate = UIApplication.shared.delegate as! AppDelegate

    let managedContext = appDelegate.persistentContainer.viewContext

    //2

    let entity = NSEntityDescription.entity(forEntityName: "Task", in:
```

```
managedContext)

let task = NSManagedObject(entity: entity!, insertInto: managedContext)

//3

task.setValue(nameTask, forKey: "name")

//4

do {

    try managedContext.save()

    //5

    tasks.append(task)

} catch let error as NSError {

    print("No ha sido posible guardar \(error), \(error.userInfo)")

}

}
```

Aquí tienes la **explicación** del código:

//1: Si recuerdas la parte de teoría que vimos, hemos mencionado, que para **guardar datos** en Core Data, lo primero que debemos hacer es guardarlos en nuestro **managed object context**. Por tanto, lo que hacemos es recuperar nuestro **managed object context** a partir del **appDelegate**.

//2: Una vez que tenemos nuestro **managed object context**, creamos un objeto de tipo **NSEntityDescription** y lo almacenamos en la variable **entity**, que representa nuestra entidad **Task**. A partir de este objeto **entity**, creamos un objeto **managed object** y lo almacenamos en nuestro **managed object context**. (Recuerda que NSManagedObject representa un **objeto único** almacenado en Core Data. En este caso representa a nuestra **tarea**)

//3: Utilizamos **KVC** para especificar el nombre de nuestra tarea. Ten en cuenta que hemos utilizado “**name**” porque cuando hemos creado el modelo, especificamos un atributo llamado “name”. Si tu le has dado un nombre diferente al crear ese atributo en el modelo, aquí tendrás que poner ese mismo nombre. Si no cumples esta condición, tu aplicación **crasheará**.

//4: Utilizamos el método **save()** del managed object context para guardar nuestra tarea **en disco**. Como esta operación de guardado puede fallar y lanzar un error, realizamos la llamada dentro de un **try-catch**.

//5: Una vez que hemos guardado nuestra tarea a través de Core Data, ya solo nos queda **almacenerla** en nuestro array **tasks**, para que se muestre cuando recarguemos la **table view**.

Este es el código que usamos para **guardar las tareas**. Si te das cuenta, podríamos hacer que se entendiera mucho mejor dividiéndolo en **diferentes métodos**, sin embargo he preferido explicarlo todo en el mismo método para que vieras en un **simple vistazo** los pasos que hay que dar para guardar nuestras tareas.

Con este método conseguimos **guardar** en Core Data **todas las tareas** que vayamos creando. Pero, si recuerdas la definición de **Persistencia**:

Persistencia es la acción de preservar la información de un objeto de forma permanente (**guardarlo**), unido a la posibilidad de poder recuperar la información del mismo (**leerlo**) para que pueda ser nuevamente utilizado.

Verás que nos queda la segunda parte: **Recuperar** las tareas que hemos guardado. Esto lo veremos en el **siguiente apartado**.

5. Recuperando nuestras Tareas en Core Data

Para **recuperar** nuestras tareas utilizaremos el método **viewWillAppear()**. Aquí tienes el código y justo después la explicación del mismo:

```
override func viewDidAppear(_ animated: Bool) {  
    super.viewDidAppear(animated)  
  
    // 1  
  
    let appDelegate = UIApplication.shared.delegate as! AppDelegate  
    let managedContext = appDelegate.persistentContainer.viewContext  
  
    // 2  
  
    let fetchRequest : NSFetchedRequest<Task> = Task.fetchRequest()  
  
    // 3  
  
    do {  
        let results = try managedContext.fetch(fetchRequest)  
        tasks = results as [NSManagedObject]  
  
    } catch let error as NSError {  
        print("No ha sido posible cargar \(error), \(error.userInfo)")  
  
    }  
  
    // 4  
  
    tableView.reloadData()  
  
}
```

Aquí tienes la **explicación** del código:

//1: Exactamente igual a como hicimos en el método **saveTask()**, obtenemos nuestro **managed object context** a partir del **appDelegate**.

//2: Creamos un objeto **fetch request** para recuperar nuestras tareas. Si recuerdas la teoría de **NSFetchRequest** que vimos al comienzo del

tutorial, sabrás que tenemos que especificar una serie de **calificadores** para definir nuestra consulta. En este caso, hemos especificado que nos devuelva los objetos almacenados en la entidad **Task**. De esta forma, recuperaremos **todas las tareas** que haya almacenadas.

//3: Una vez que hemos definido nuestra **fetch request** (La consulta que vamos a hacer a nuestra Base de Datos), la ejecutamos a través del método **executeFetchRequest()** y almacenamos los resultados en la variable **results**. Por último almacenamos estas tareas recuperadas en nuestro array **tasks**. Puedes ver, que seguimos controlando los errores utilizando un **do-catch**.

//4: Por último **actualizamos** los datos de la tabla para que **muestre por pantalla** las tareas recuperadas.

Pues con esto, tendríamos nuestra aplicación correctamente **modificada** para que realice el **guardado** y **recuperación** de las tareas en Core Data. Vamos a **hacer la prueba** de que todo funciona correctamente.

6. Prueba Final

Si has seguido paso a paso el Tutorial, ya te imaginarás como **vamos a comprobar** que todo funciona correctamente. Estos son los pasos que tienes que seguir:

1. Ejecuta la aplicación
2. Añade algunas tareas
3. Accede a la multitarea pulsando 2 veces esta combinación de teclas:
cmd + ⌛ + H
4. Desliza la aplicación hacia arriba para cerrarla
5. Vuelve a ejecutar la app desde Xcode
6. Verás como todas las tareas que añadiste en el paso 2 se muestran perfectamente en la tabla

De esta forma, hemos conseguido implementar un **sistema de persistencia** basado en **Core Data** en nuestra aplicación. Cuando cualquier usuario la utilice, todas sus tareas se irán **guardando**

perfectamente y no habrá ningún peligro en que se pierdan.

7. Resumen Final

Hemos visto en este tutorial todos los **conceptos** que debes conocer para trabajar con Core Data.

Se trata de un **tutorial de introducción** pero creo que puede valerte para conocer algunos **principios básicos** a la hora de trabajar con **Core Data**. Así, cuando tengas que desarrollar un **proyecto real** utilizando este framework, los conceptos ya te serán familiares.

Introducción a los Storyboards

[Parte 1]

Como usarlos en tus aplicaciones iOS con Swift

Lenguaje Swift | Nivel Principiante

1. Introducción

Apple introdujo los **Storyboards** en el lanzamiento de iOS 5.

Hace bastantes años de esto...

Antes del lanzamiento de los Storyboards, los Desarrolladores iOS solo teníamos la opción de crear las **interfaces** de nuestra aplicaciones a través de ficheros con extensión **.xib** o directamente programarlas a través de código.

Cada uno de estos ficheros representaba una **pantalla** de nuestra aplicación.

Por tanto, a día de hoy tienes **tres opciones** para crear tus interfaces:

1. Programarlas directamente a través de código
2. Utilizar ficheros .xib
3. Usar Storyboards

En este tutorial, vamos a ver todo lo que necesitas saber para sentirte cómodo utilizando **Storyboards** y no tener ningún problema siempre que quieras crear una nueva **aplicación iOS**.

2. Ventajas de utilizar Storyboards en tus desarrollos

Estas son algunas de las ventajas que nos ofrecen los Storyboards:

1. Son perfectos para ver el **flujo de navegación** completo de una aplicación.
2. Te permiten crear una **aplicación multivista** sin tener que escribir prácticamente código.
3. Son muy útiles para desarrollos **pequeños-medios**.

Como cualquier opción, también tienen **desventajas**.

Solo voy a comentar una.

Debido a que toda nuestra interfaz se concentra en un único fichero, suelen generarse bastantes **conflictos** que complican el **desarrollo en equipo** de una aplicación.

Sin embargo, en muchos equipos de desarrollo son la **opción preferente** a la hora de crear cualquier interfaz, por lo que es importante que seas capaz de trabajar con ellos.

3. ¿Qué vamos a ver en este Tutorial sobre Storyboards?

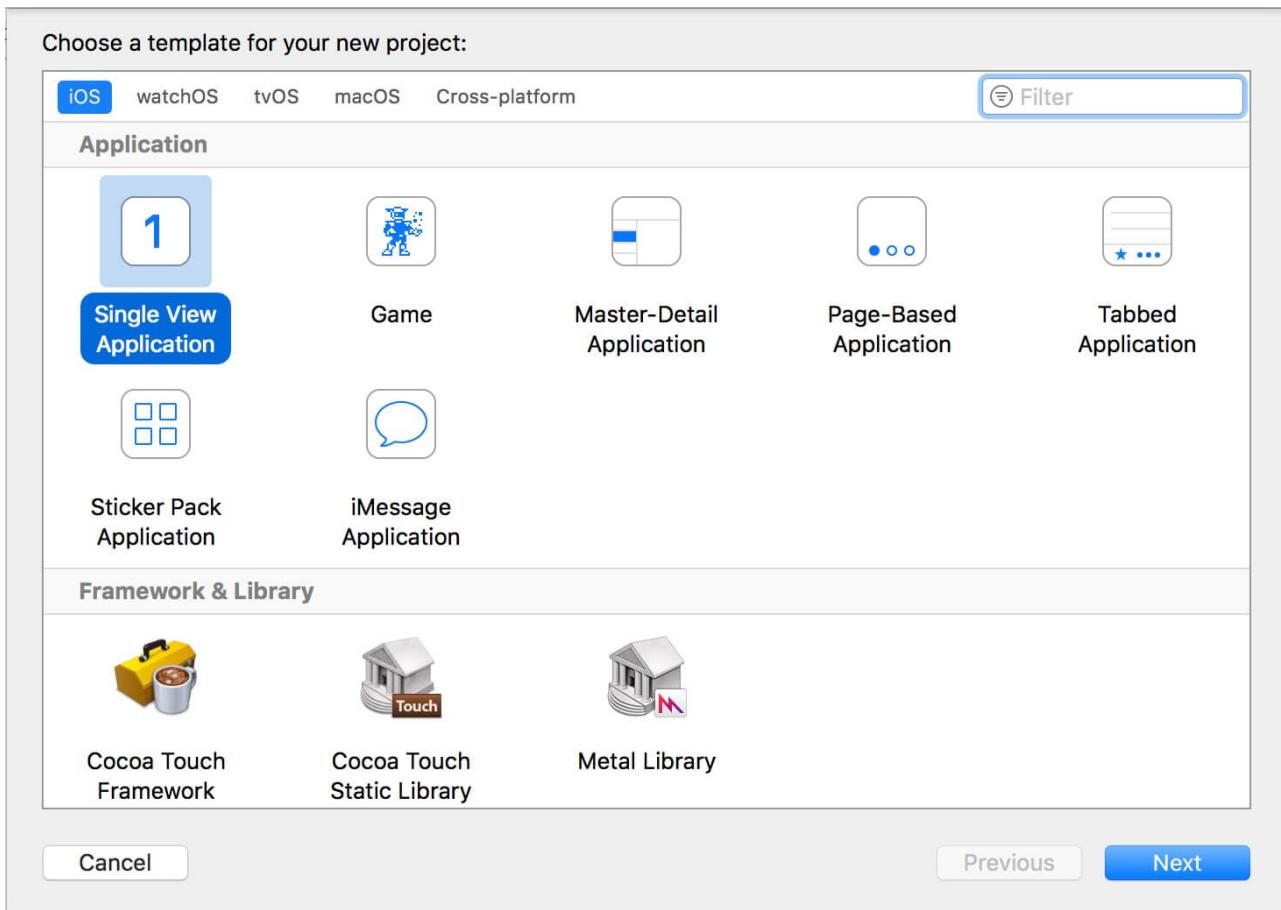
Los puntos más importantes que veremos son los siguientes:

1. Ventajas de utilizar Storyboards en tus desarrollos
2. Conociendo Interface Builder
3. ¿Qué es el Dock?
4. Especificando la Main Interface de nuestra App
5. Secuencia de arranque de una aplicación
6. Como seleccionar el Initial View Controller
7. Configurar los modos Portrait y Landscape de nuestra App
8. ¿Cómo añadir un UITableViewController, un UINavigationController y un TabBarController?

Comencemos entonces echando un vistazo a los **fundamentos básicos** de los Storyboards.

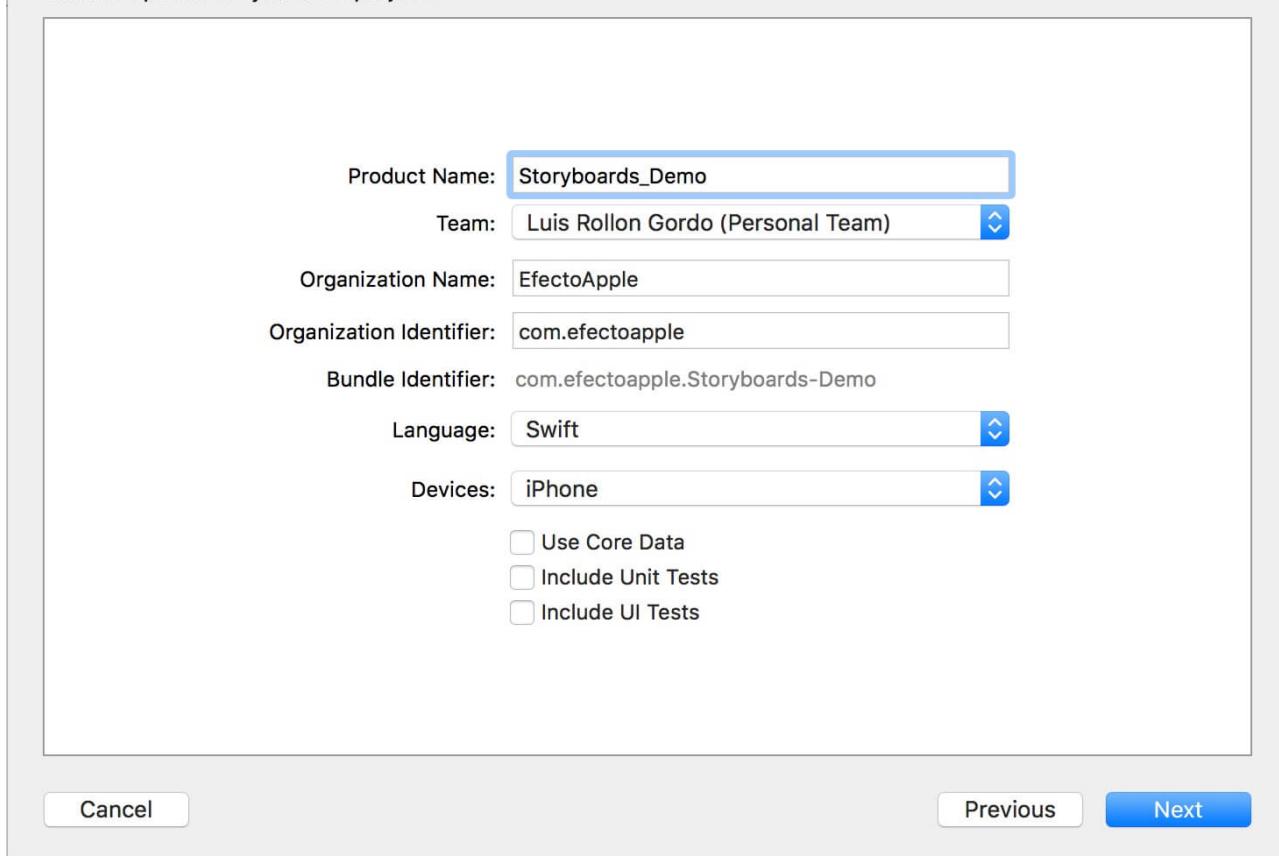
4. Creando nuestro Proyecto en Xcode

Para comenzar crea un **nuevo proyecto** en Xcode, accediendo al menú **File > New > Project...** y elige la plantilla **Single View Application**:



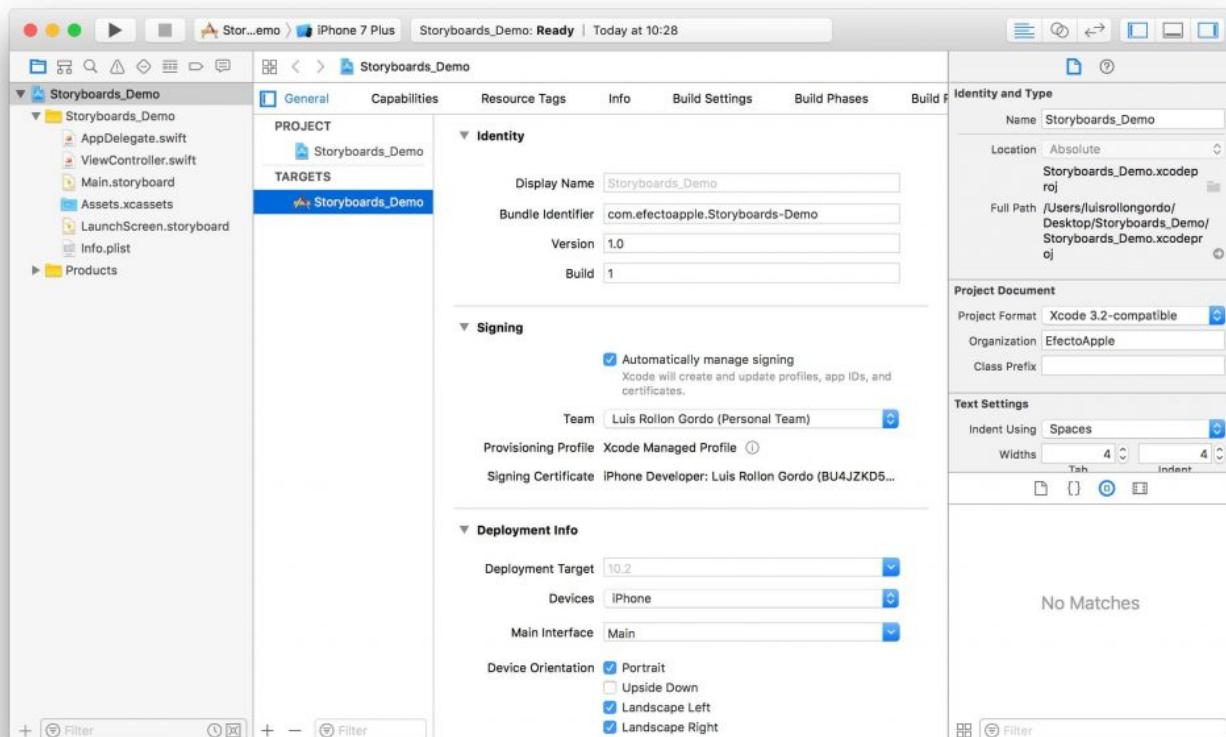
Después pulsa en **Next** y dale el nombre que quieras. En mi caso lo llamaré **Storyboards_Demo**. Deja el resto de opciones, tal cual las ves en la imagen:

Choose options for your new project:



Pulsa en **Next**, guarda el proyecto donde quieras y para finalizar pulsa en el botón **Create**.

Después de crear el proyecto, Xcode te mostrará algo así:



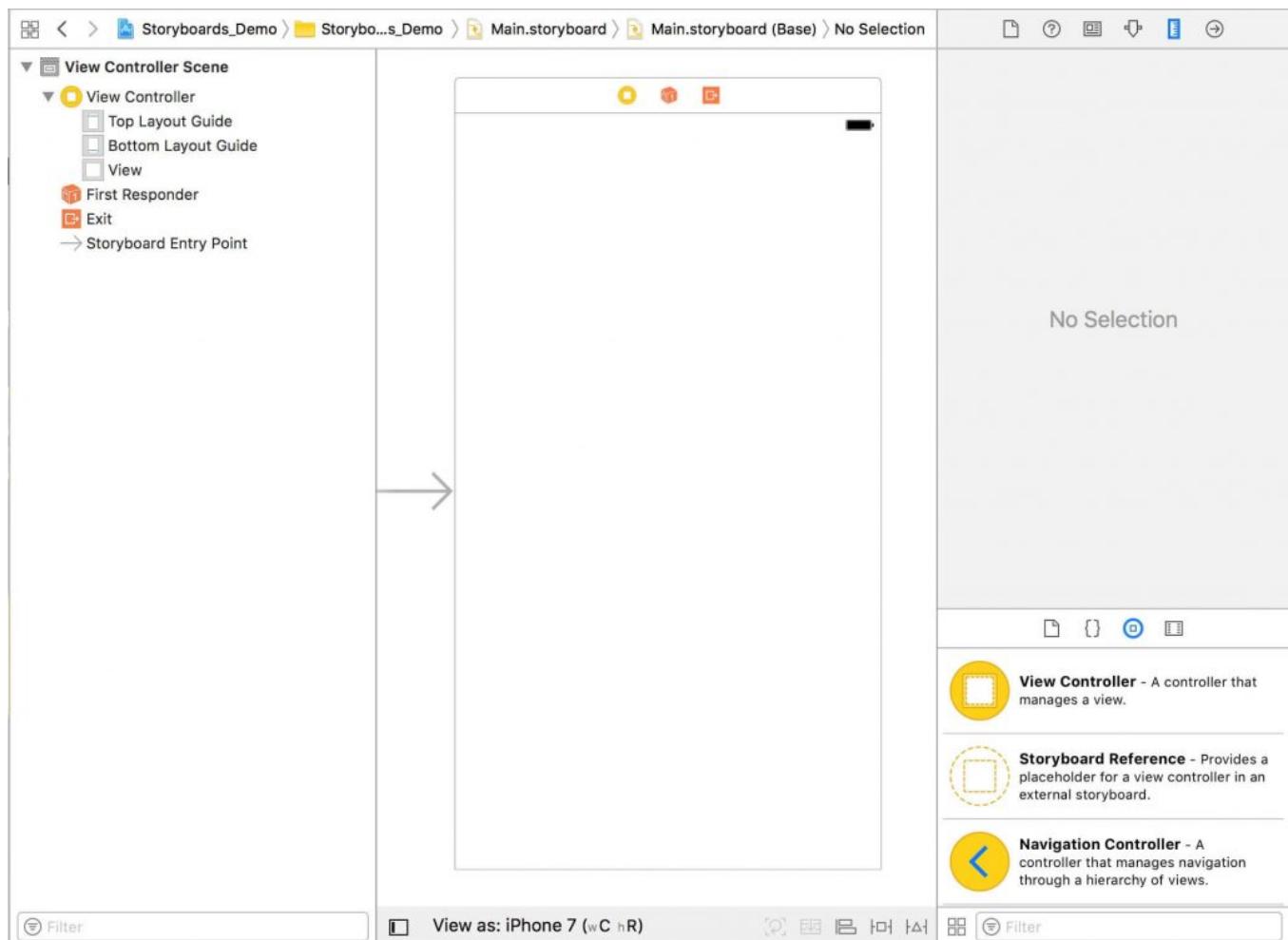
Con este Proyecto de Demostración, iremos viendo todo lo necesario para dominar los Storyboards.

5. Fundamentos Básicos de los Storyboards

Haz click en el fichero *Main.storyboard* y veamos algunos conceptos que debes conocer:

Interface Builder

La pantalla que se ha abierto al acceder al *Main.storyboard* es conocida como **Interface Builder** y tiene el siguiente aspecto:



En las primeras versiones de iOS, Interface Builder era una aplicación **independiente** de Xcode y al igual que ahora, se utilizaba para crear las **interfaces** de nuestras aplicaciones.

La parte izquierda de Interface Builder se conoce como **Document Outline** y nos muestra un listado de todas las escenas que componen

nuestra interfaz.

Una escena representa a cada una de las pantallas de nuestra aplicación

Dentro de cada escena se van englobando todos los **objetos** que vayamos añadiendo a la interfaz, además de **constraints** y otros elementos.

Para mostrar u ocultar el **Document Outline** puedes utilizar el siguiente botón, situado en la parte inferior izquierda:



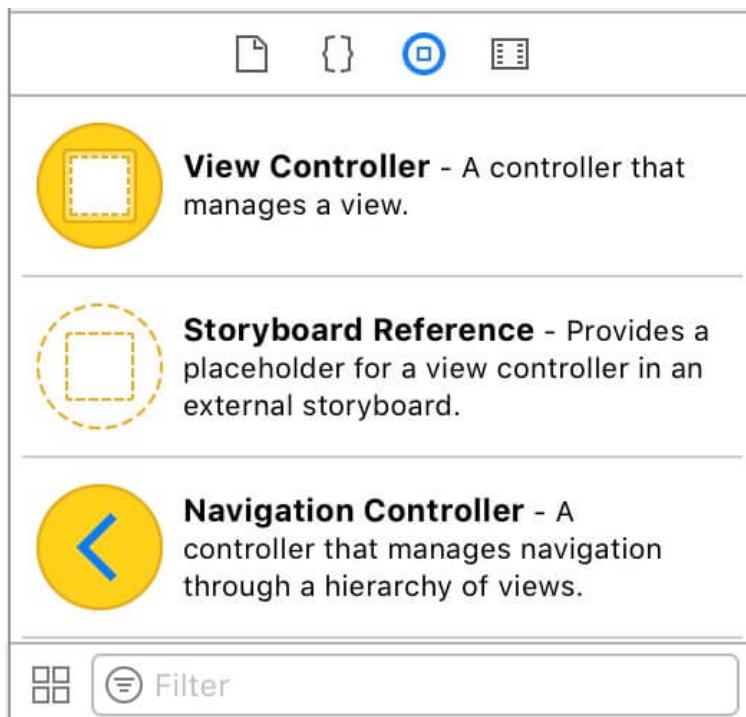
Como ves, en nuestro proyecto recién creado únicamente tenemos una pantalla. Esta pantalla está representada por nuestro **View Controller**.

En la parte superior derecha de Interface Builder tenemos el **Inspector de Atributos**. A través de las opciones que nos ofrece, podemos realizar modificaciones en las propiedades de cualquier objeto que forme parte de nuestra interfaz.

Dentro del Inspector de Atributos tienes **6 menús** diferentes, donde cada uno controla un aspecto diferente de cualquier objeto:

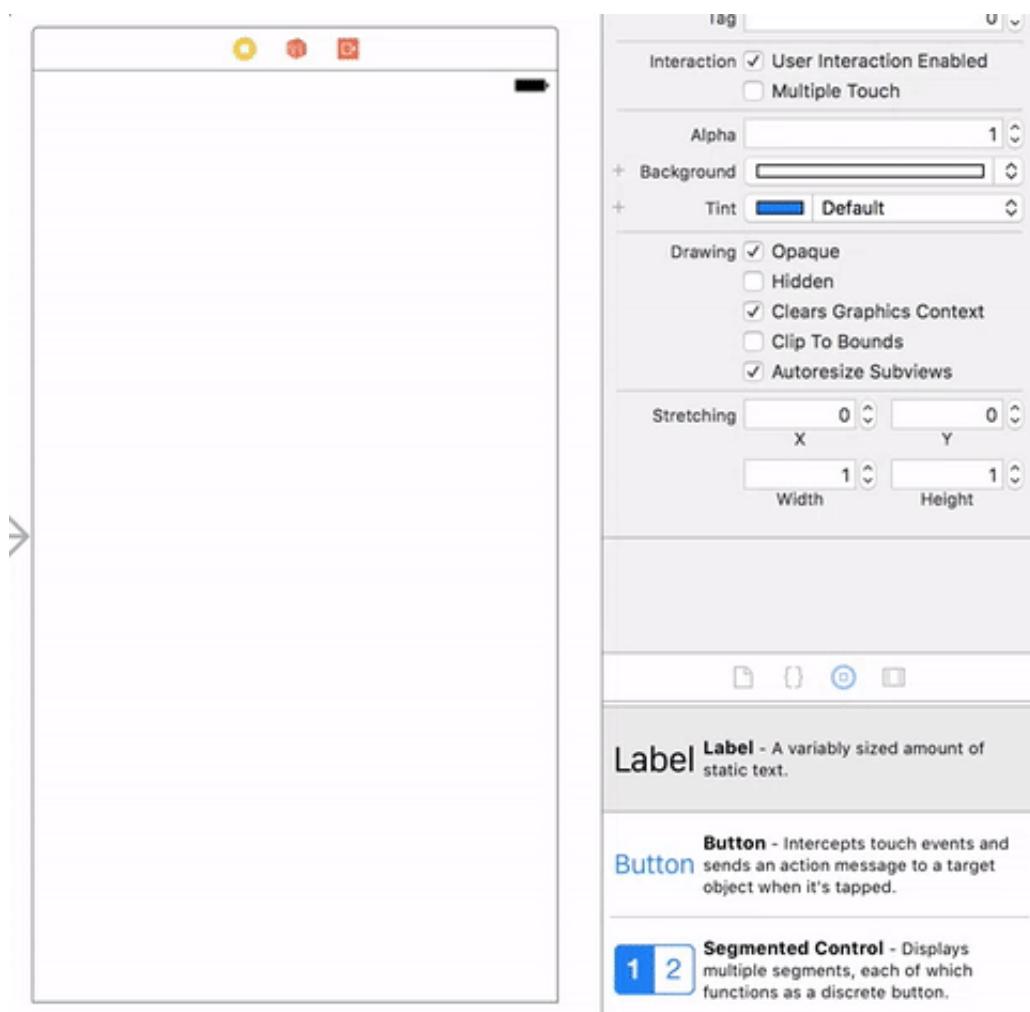


Por otro lado, en la parte inferior derecha, dispones de la **Biblioteca de Objetos**:

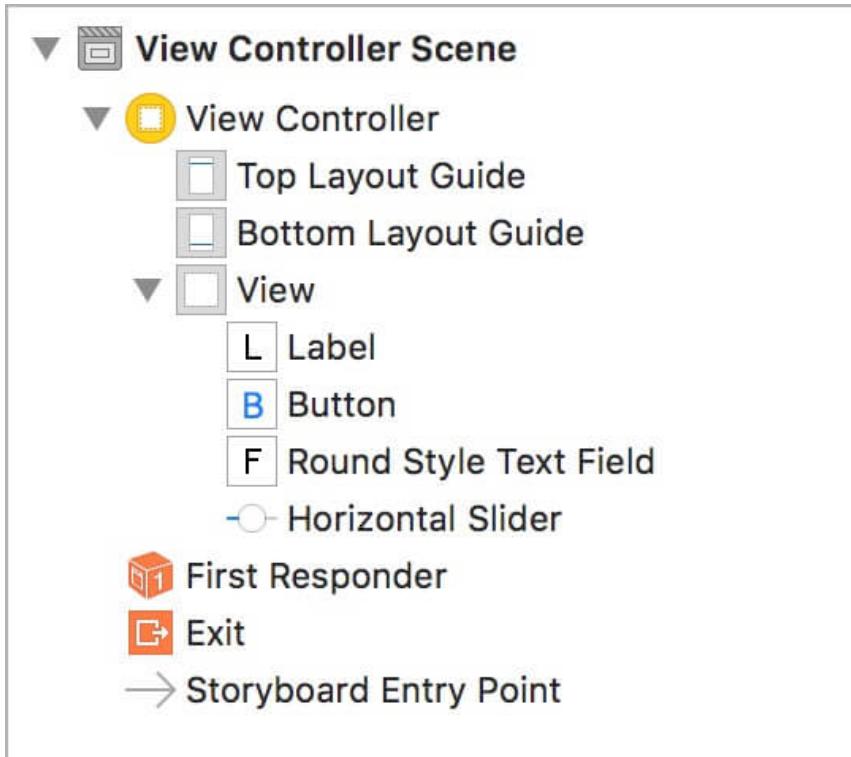


Desde aquí puedes añadir **cualquier elemento** a la interfaz de tu aplicación.

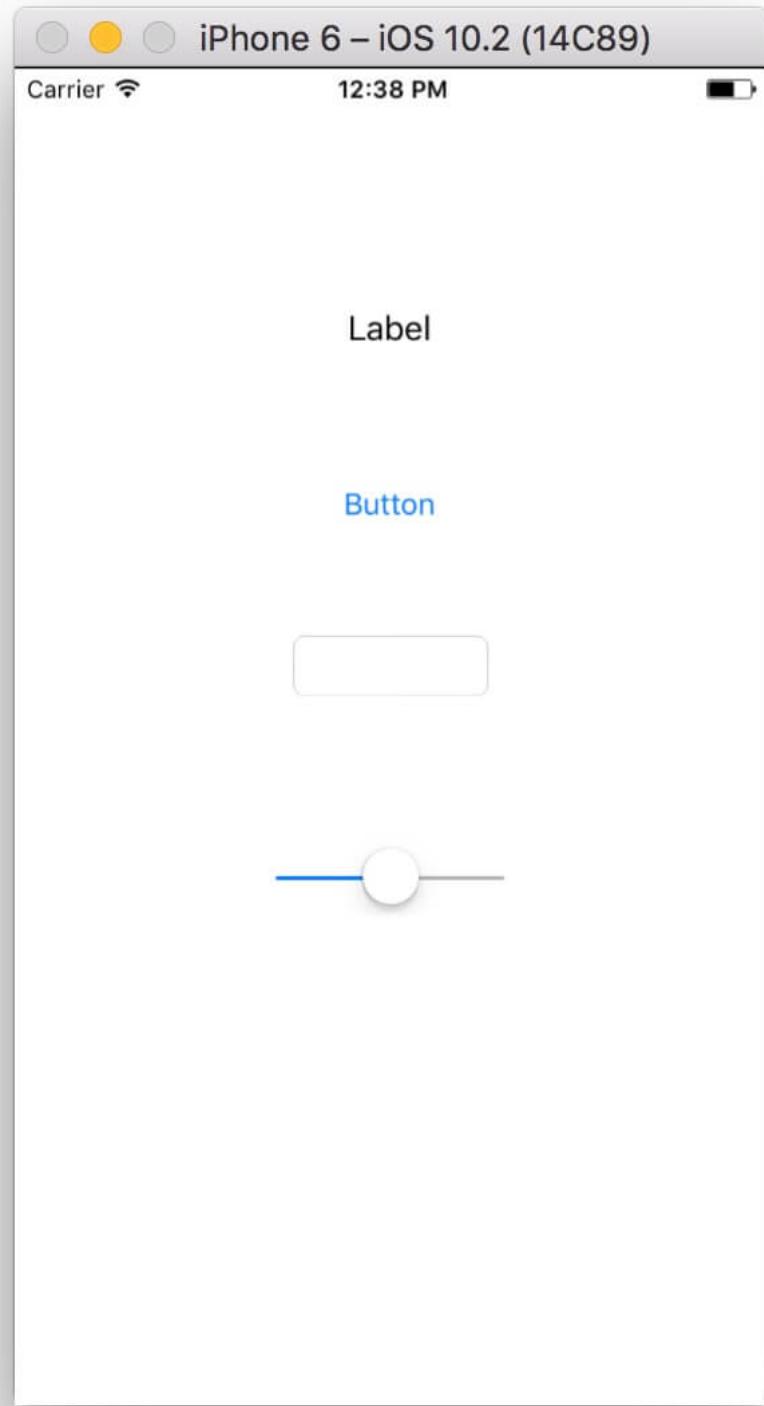
Prueba por ejemplo a arrastrar un **Label**, un **Button**, un **TextField** y un **Slider** a tu ViewController:



Si te fijas, además de añadirlos a nuestra interfaz, Xcode los ha añadido a la **lista de elementos** que aparecen en el Document Outline:



Si ejecutas la aplicación, haciendo **cmd+R**, verás que en el simulador se muestran los elementos que hemos añadido a nuestra interfaz.



Dock

Justo encima de cada ViewController podrás ver el **Dock**.



El Dock muestra los **objetos de alto nivel** de cada escena. Cada escena tendrá como mínimo un objeto **ViewController**, un objeto **First**

Responder y un objeto **Exit**, pero además puede tener otros elementos. El Dock es simplemente un Document Outline en miniatura.

ViewController representa evidentemente a nuestro único ViewController.

First Responder gestiona el foco de nuestra aplicación. Lo más probable es que prácticamente no lo utilices.

Exit se puede utilizar para añadir opciones de navegación a nuestras interfaces.

Main Interface

Antes, al ejecutar nuestro proyecto, la aplicación que hemos creado, ya sabía perfectamente cual era el **Storyboard** que tenía que enlazar al ejecutar la app.

Esto no tiene ningún misterio, solo tenemos un fichero *Main.storyboard*, por lo que lo normal es que nuestro proyecto sepa que ese es el fichero que contiene nuestra interfaz.

Sin embargo, en algún otro proyecto puede que utilices varios **Storyboards diferentes**, por lo que es importante que entiendas, **donde especificamos** cual es el Storyboard principal.

Simplificando mucho, cualquier aplicación iOS, por defecto, comienza su ejecución en la clase *AppDelegate.swift*.

Haz click en esta clase y echa un vistazo a la **parte inicial** de la misma:

```
@UIApplicationMain  
class AppDelegate: UIResponder, UIApplicationDelegate {  
    var window: UIWindow?  
  
    func application(_ application: UIApplication,  
                    didFinishLaunchingWithOptions launchOptions:  
                    [UIApplicationLaunchOptionsKey: Any]?) -> Bool {
```

```

// Override point for customization after application launch.

return true

}

...

```

El atributo **@UIApplicationMain** situado en la parte superior determina que nuestra clase *AppDelegate* es el punto de entrada de nuestra aplicación.

Es obligatorio que nuestro *AppDelegate* herede de **UIResponder** y tenga una propiedad de tipo **UIWindow** para que la app funcione.

Si te fijas en los métodos que aparecen en esta clase, están todos prácticamente vacíos.

Unicamente **didFinishLaunchingWithOptions()** tiene algo de código y lo único que hace es devolver true.

Una vez revisada la clase inicial de nuestra aplicación, es probable que te sigas preguntando ¿donde **hace referencia** nuestra aplicación al Storyboard?

El Fichero Info.plist

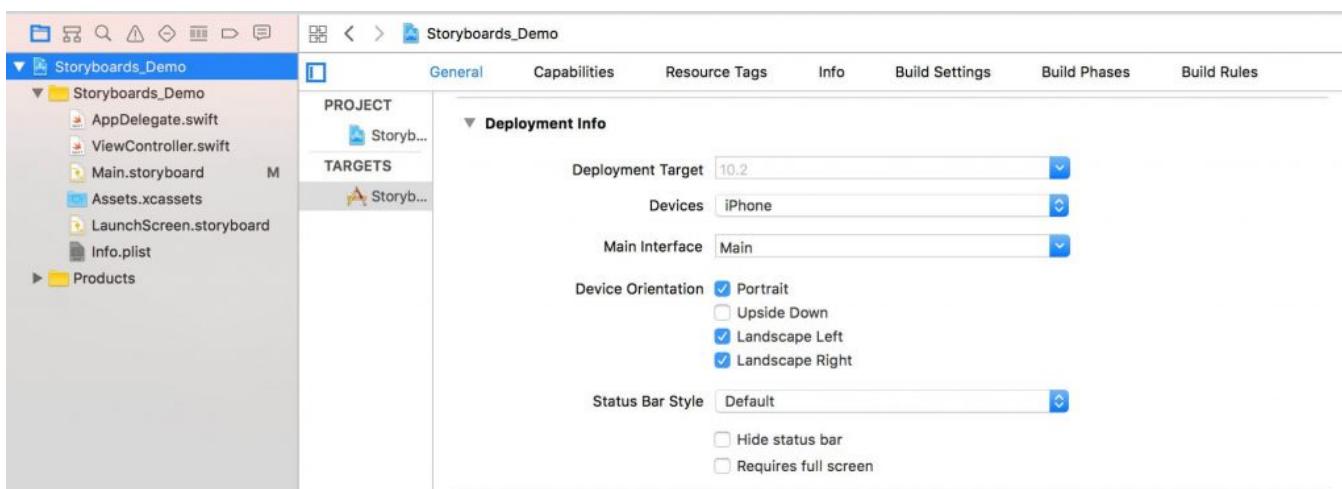
Para responder a esto tendrás que abrir el fichero **Info.plist** y localizar la entrada **Main storyboard file base name**.

Key	Type	Value
▼ Information Property List	Dictionary	(13 items)
Localization native development re...	String	en
Executable file	String	\$(EXECUTABLE_NAME)
Bundle identifier	String	\$(PRODUCT_BUNDLE_IDENTIFIER)
InfoDictionary version	String	6.0
Bundle name	String	\$(PRODUCT_NAME)
Bundle OS Type code	String	APPL
Bundle versions string, short	String	1.0
Bundle version	String	1
Application requires iPhone enviro...	Boolean	YES
Launch screen interface file base...	String	LaunchScreen
Main storyboard file base name	String	Main
► Required device capabilities	Array	(1 item)
► Supported interface orientations	Array	(3 items)

Cuando ejecutamos la app, el objeto **UIApplication** carga el storyboard

que esté especificado en esta clave, inicializa el ViewController que esté marcado como **Initial View Controller** (Ahora veremos donde se hace eso) y después coloca la **View** de ese **ViewController** en el objeto **UIWindow**.

Existe otro punto donde debemos especificar cual es la interfaz principal de nuestra aplicación. Si haces click en el **nombre del proyecto**, en la parte superior izquierda y haces scroll hasta la sección **Deployment Info**, verás como hay otra opción llamada **Main Interface**, donde aparece el nombre de nuestro storyboard.

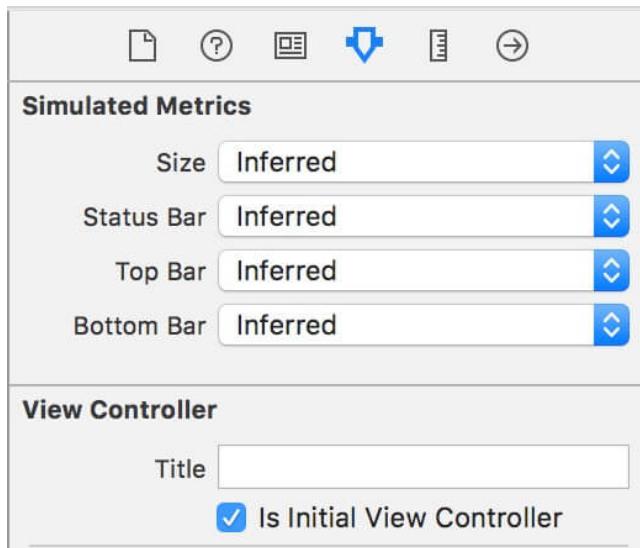


Estas 2 opciones son fundamentales para que tu aplicación arranque correctamente.

Initial View Controller

Hemos comentado en el punto anterior, que una vez que hemos especificado cual será nuestro storyboard principal, también debemos determinar cual será el **ViewController** que nuestra aplicación debe cargar en **primer lugar**.

Para seleccionar esto, accede a *Main.storyboard*, haz click sobre nuestro único ViewController y en la cuarta opción del **Inspector de Propiedades**, verás la opción **Is Initial View Controller**.



Esa opción especifica que cuando se cargue el storyboard, la **primera pantalla** que mostrará nuestra aplicación será la que esté seleccionada como Initial View Controller.

De hecho, si te fijas, verás que en **Interface Builder**, justo a la izquierda del ViewController aparece una **flecha** apuntando hacia la derecha. Si arrastras esa flecha y la colocas apuntando a otro ViewController, este último se convertiría en el **Initial View Controller**.

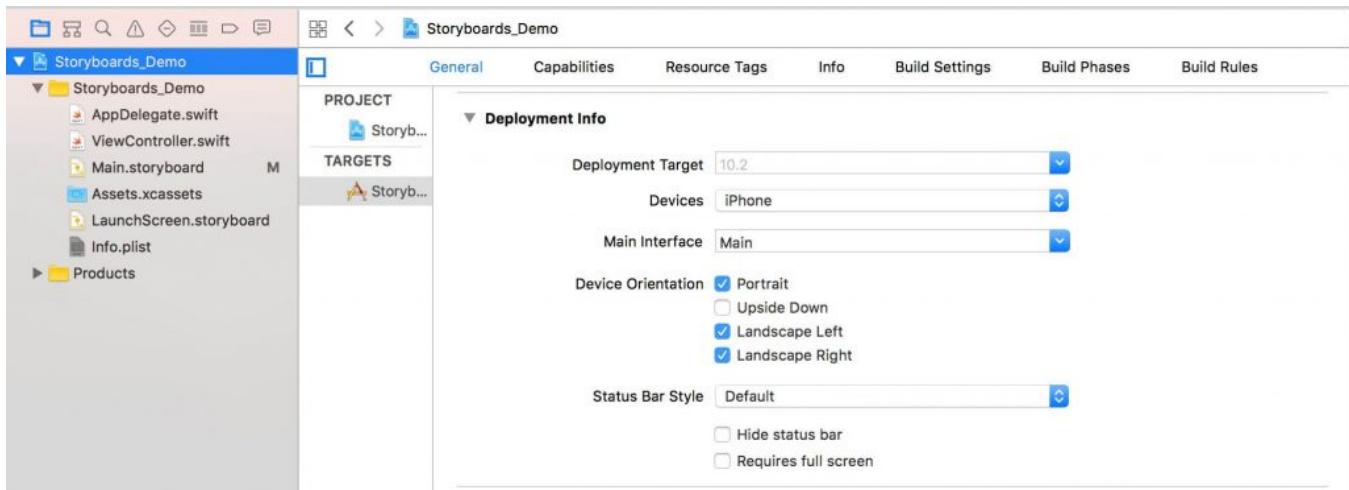
Como conclusión, si en algún momento tienes problemas con tu aplicación, porque ves que no carga el storyboard que debe o porque la primera pantalla que muestra no es la que tu quieras, es conveniente que revises estas **3 últimas opciones** que acabamos de ver, porque lo mas probable es que hayas modificado alguna de ellas y no lo recuerdes.

Portrait/Landscape

A la hora de desarrollar una aplicación, tienes varias opciones.

Puedes querer que tu aplicación se ejecute solo en **Portrait** (Cuando el dispositivo esté en posición vertical) o también en **Landscape** (Cuando el dispositivo esté en posición horizontal).

Estas opciones y alguna más las puedes controlar desde el menú que hemos visto antes: **Deployment Info**.



En la sección **Device Orientation** puedes ver que tienes 4 opciones para elegir.

Si ejecutáramos la aplicación, con las opciones que ves en la imagen, nuestra app giraría su interfaz siempre que utilizáramos el iPhone en **posición vertical** con el botón Home abajo y también en **posición horizontal**, tanto con el botón Home hacia la derecha como hacia la izquierda.

Sin embargo, si nuestro dispositivo estuviera en posición vertical con el botón Home hacia arriba, nuestra aplicación no se adaptaría a esta posición, ya que la opción **Upside Down** está desactivada.

Por tanto, esto es lo que representa cada una de las opciones:

- Portrait: Posición vertical con el botón Home abajo
- Upside Down: Posición vertical con el botón Home arriba
- Landscape Left: Posición horizontal con el botón Home hacia la izquierda
- Landscape Right: Posición horizontal con el botón Home hacia la derecha

A la hora de desarrollar tu proyecto, debes **tener muy en cuenta** todo esto.

6. Añadiendo un TableViewController

Por defecto, cuando creamos un proyecto de tipo **Single View**

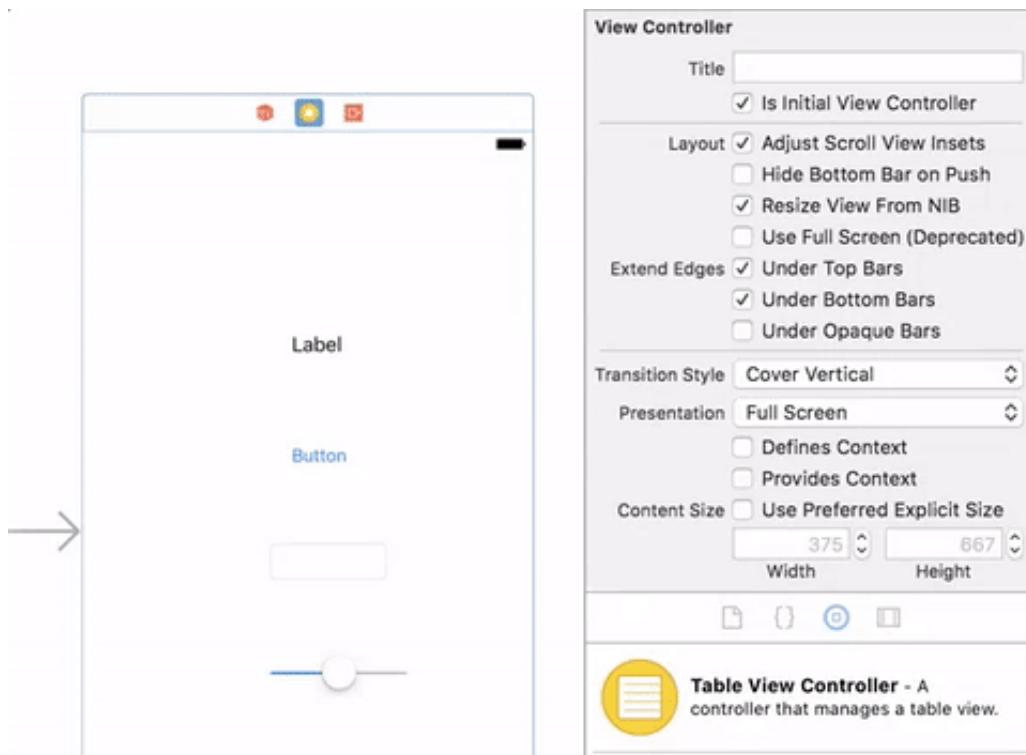
Application, Xcode crea un ViewController en nuestro Storyboard.

En ocasiones, es probable que queramos trabajar directamente con un **TableViewController**.

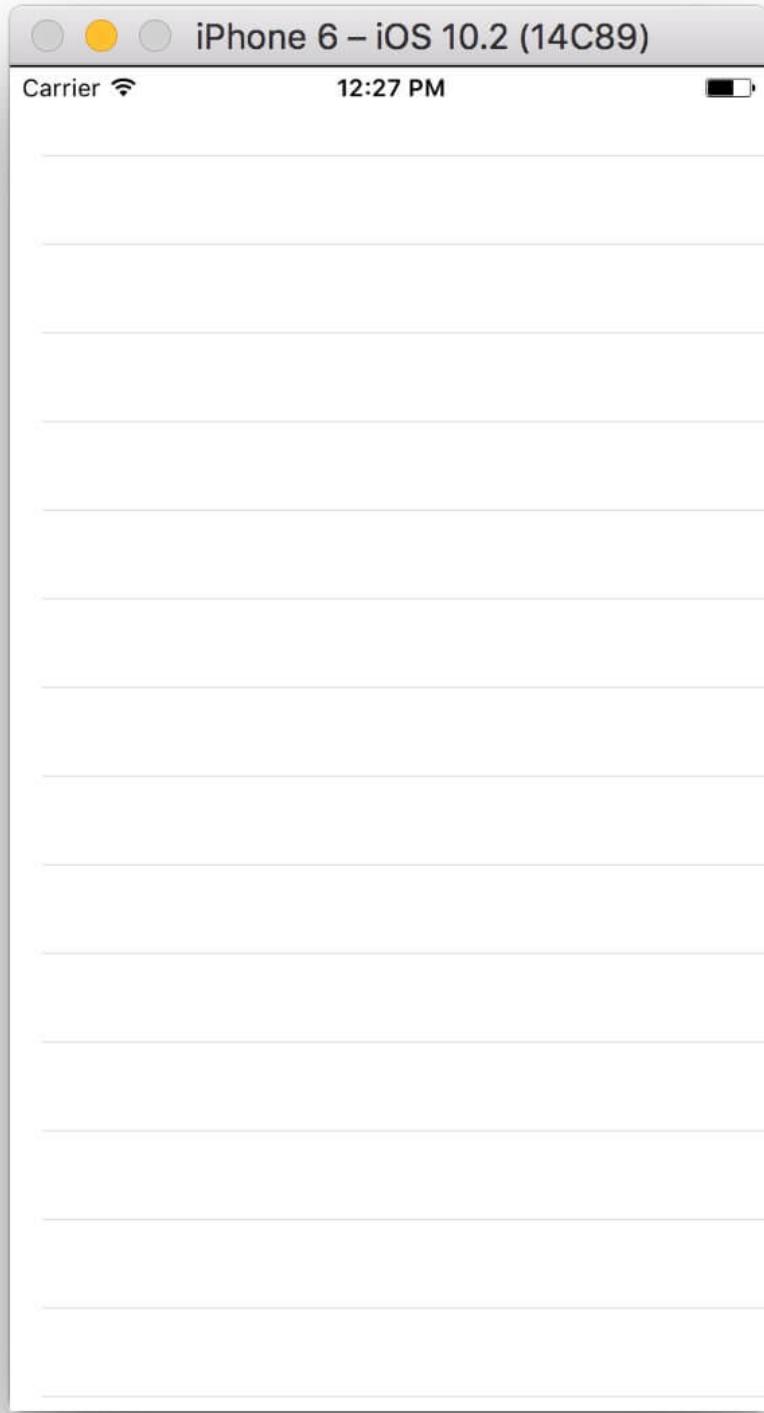
En estos casos, para sustituir uno por otro, lo mas rápido es seguir estos pasos:

1. Seleccionar nuestro ViewController en el Main.storyboard
2. Eliminarlo pulsando la tecla Delete
3. Añadir un TableViewController desde la Biblioteca de Objetos
4. Elegir este TableViewController como Initial View Controller

A continuación puedes ver esta **secuencia de acciones**:



Si ejecutas ahora la aplicación, verás como la **vista inicial** que muestra es la de un **TableView** vacío.



7. Añadiendo un NavigationController

Los **NavigationController** son la opción perfecta para añadir navegación entre pantallas cuando creamos aplicaciones multivista.

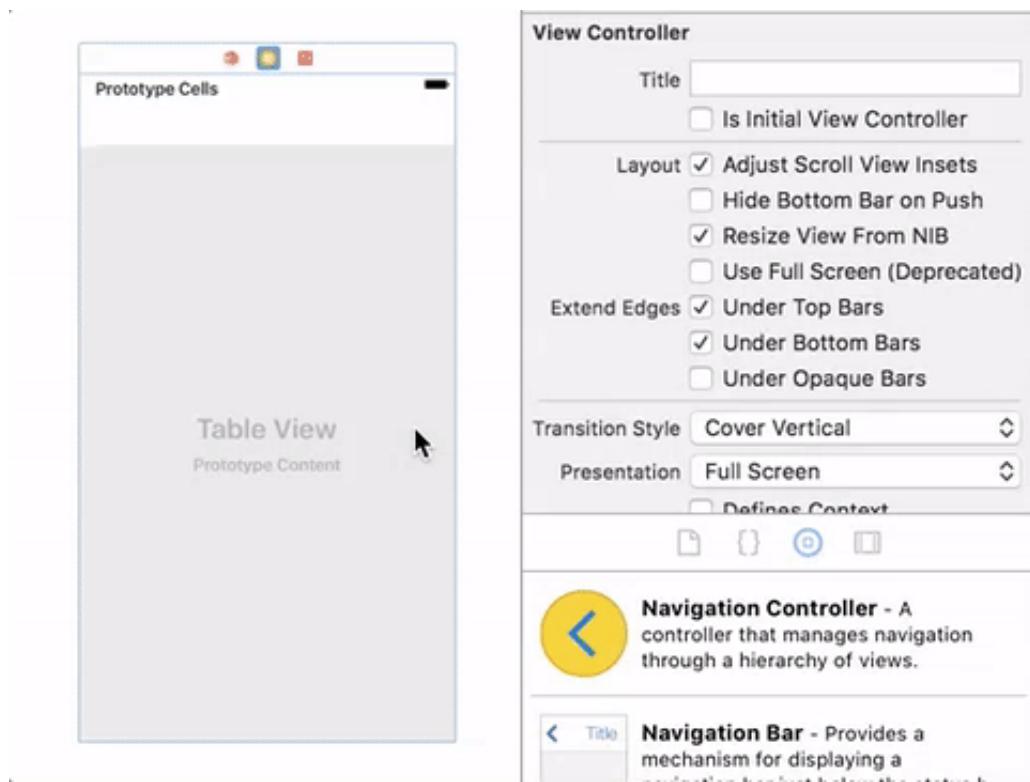
Un NavigationController debe ir **siempre** asociado a otro Controlador. Este controlador puede ser de cualquier tipo: ViewController, TableViewController, CollectionViewController.

Este Controlador asociado al UINavigationController se convierte en su controlador principal, es decir, en su **RootViewController**.

Para añadir un UINavigationController a tu aplicación puedes **seguir unos pasos** similares a los que hemos visto en el apartado anterior:

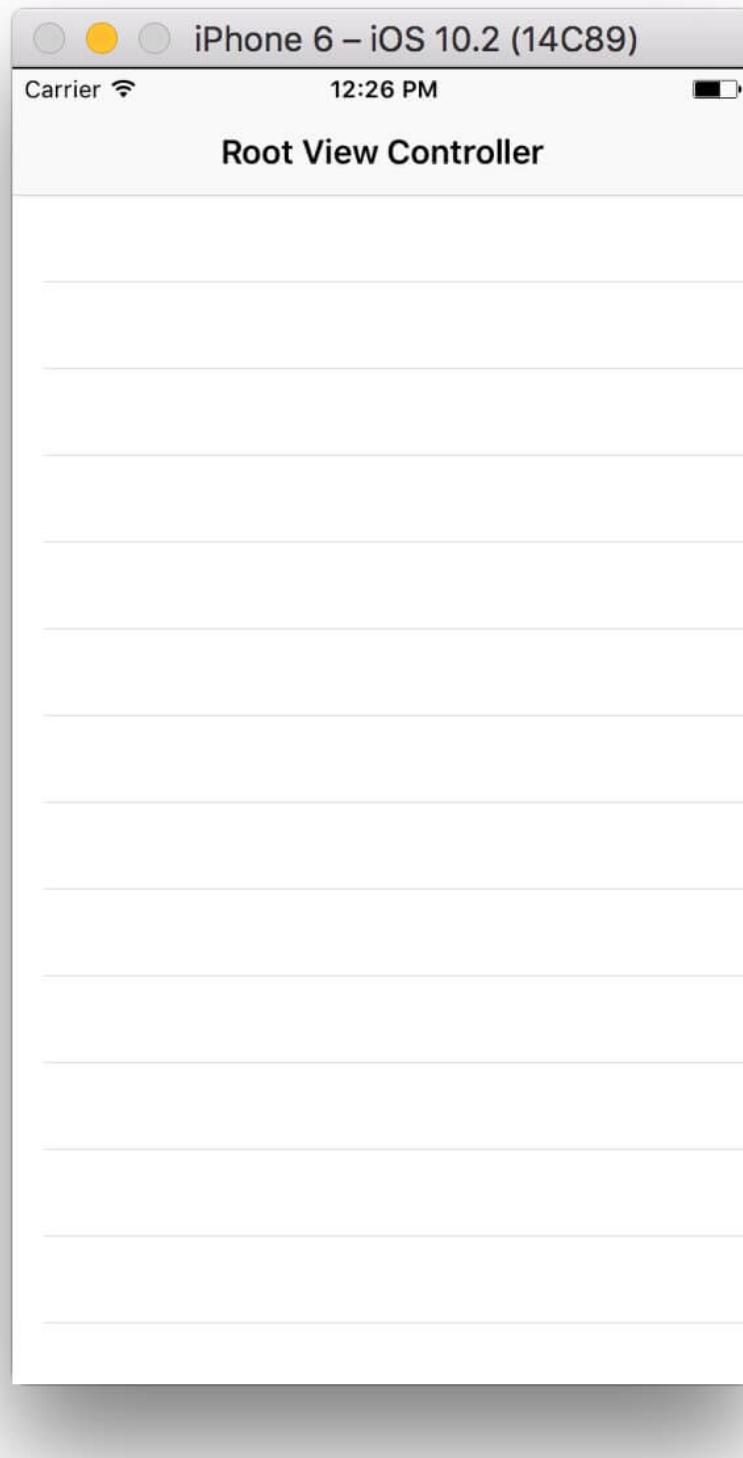
1. Seleccionar nuestro UITableViewController en el Main.storyboard
2. Eliminarlo pulsando la tecla Delete
3. Añadir un UINavigationController desde la Biblioteca de Objetos
4. Elegir este UINavigationController como Initial View Controller

Aquí tienes **estos pasos** en video:



Como has podido observar, el **NavigationController** lleva asociado por defecto un **TableViewController**.

Si **ejecutas** ahora el proyecto, verás como nuestra aplicación muestra el **TableViewController** junto con una **NavigationBar** en la parte superior con el título **Root View Controller**, que indica que nuestra app cuenta con un UINavigationController.



8. Añadiendo un TabBarController

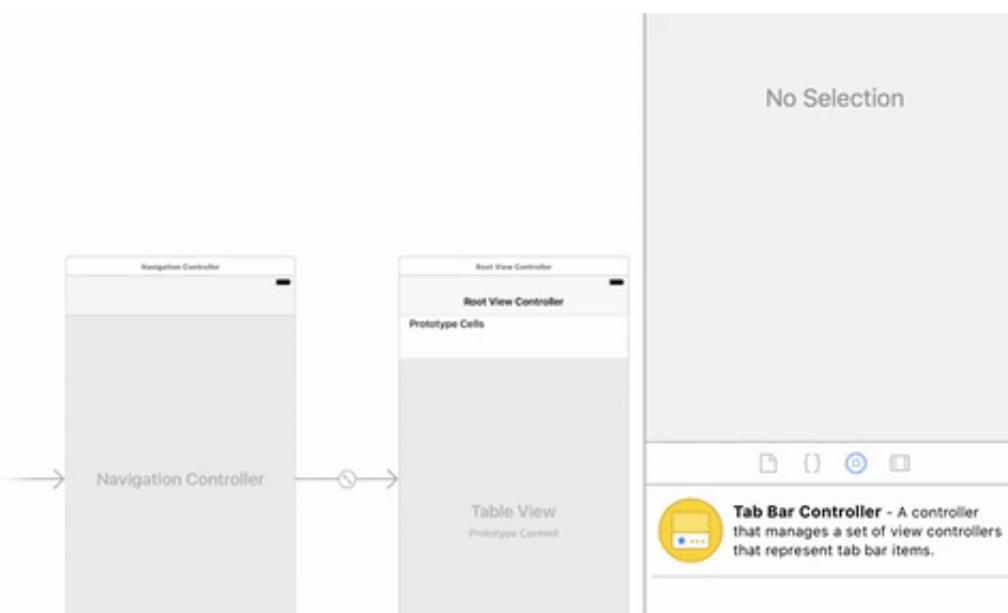
Los **TabBarController**s, al igual que los **NavigationController**s nos permiten crear aplicaciones multivista.

Sin embargo, si en los **NavigationControllers** establecemos una navegación jerárquica, de padres a hijos, en los **TabBarController**s los diferentes **ViewControllers** que utilicemos estarán **al mismo nivel** de navegación.

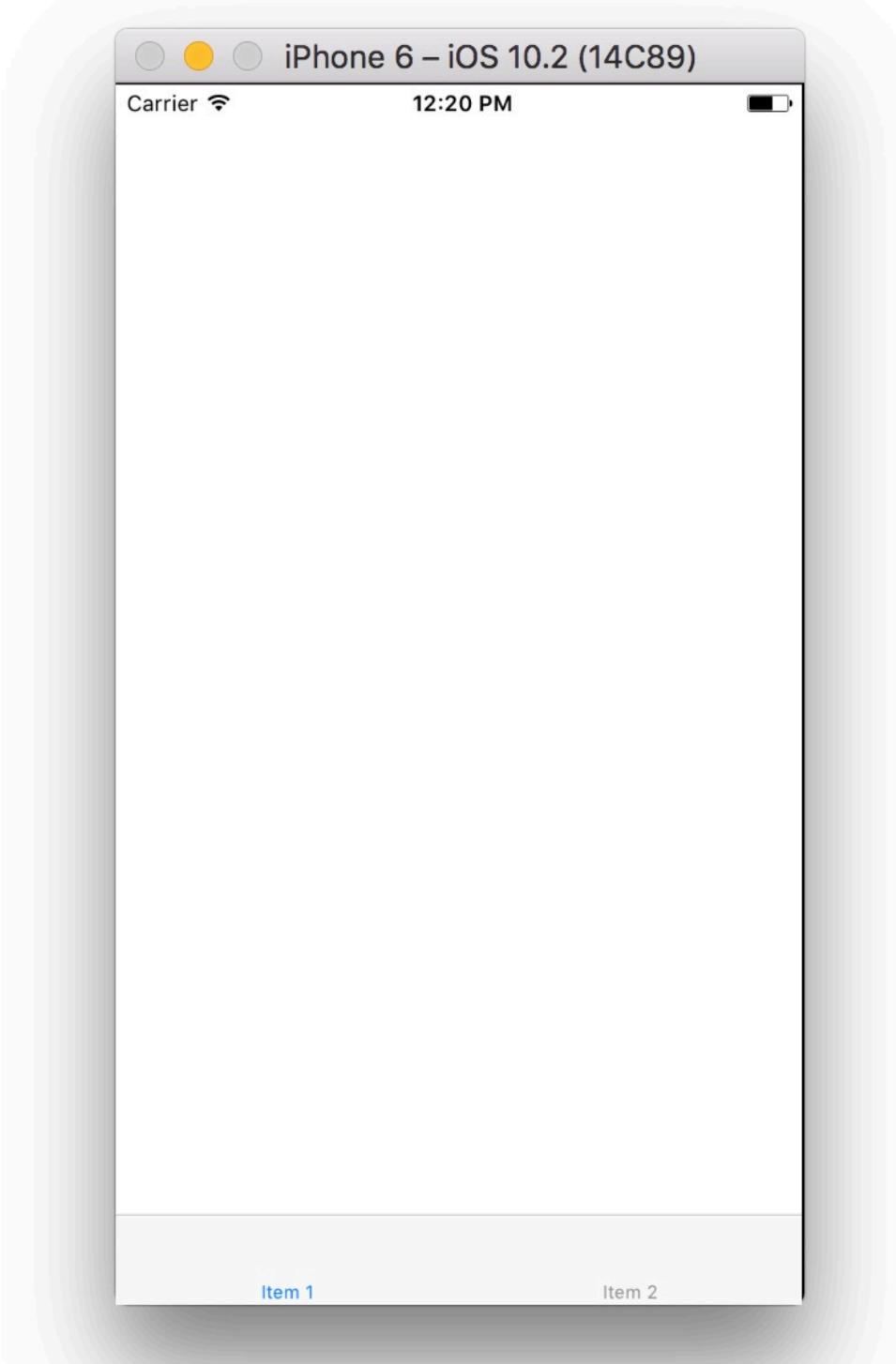
Para añadir un TabBarController a través de un storyboard, vamos a **seguir unos pasos** similares a los vistos en los anteriores apartados:

1. Seleccionar nuestro NavigationController en el Main.storyboard
2. Eliminarlo pulsando la tecla Delete
3. Selecciona el Root View Controller del Navigation Controller
4. Elimínalo también con la tecla Delete
5. Añadir un TabBarController desde la Biblioteca de Objetos
6. Elegir este TabBarController como Initial View Controller

Estos son los pasos seguidos:



Si **ejecutas** la app verás como nuestra pantalla principal es ahora un TabBarController que cuenta a su vez con 2 ViewControllers, identificados como **Item 1** e **Item 2**.



9. Resumen Final

Hemos realizado un repaso rápido a los **aspectos fundamentales** de los Storyboards. Hemos visto como añadir **elementos básicos** que nos permitirán crear interfaces complejas.

Sin embargo, no hemos tratado uno de los temas mas importantes de los Storyboards: **Los Segues**.

Este concepto lo veremos a continuación, en la **Segunda parte** de este Tutorial.

Introducción a los Storyboards

[Parte 2]

Como usarlos en tus aplicaciones iOS con Swift

Lenguaje Swift | Nivel Principiante

1. Introducción

Continuemos entonces con la **Segunda Parte** del Tutorial sobre Storyboard.

Nos centraremos en uno de los conceptos más importantes de los Storyboards: **Los Segues**.

2. ¿Qué vamos a ver en este Tutorial sobre Storyboard?

Los puntos **más importantes** que revisaremos son los siguientes:

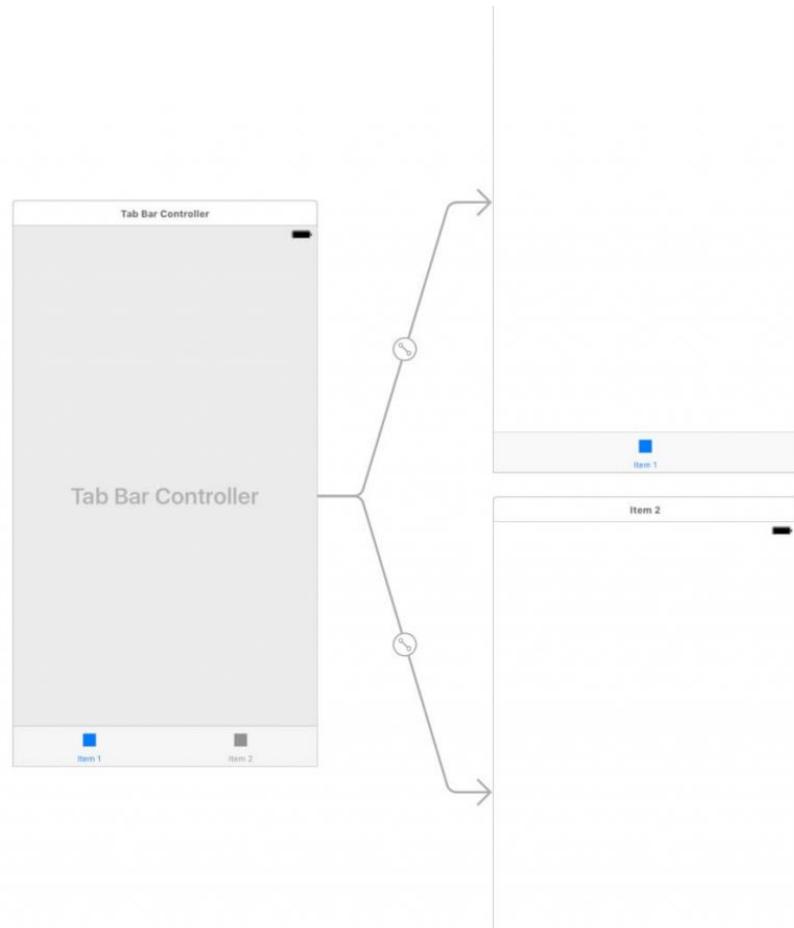
- ¿Qué son los Segues?
- Tipos de Segues
- Trabajando Segues de Transición Modal
- Entendiendo los Segues con Navigation Controller
- Dominando los Segues con TabBarController

Comencemos entonces por la **primera parte** de este Tutorial.

3. Trabajando con Segues

Si has podido revisar la **primera parte** del Tutorial, recordarás que cuando añadimos un **NavigationController** o un **TabBarController** a nuestro proyecto, aparecían unas flechas que unían unos ViewControllers con otros.

Algo similar a esto:



Estas flechas representan las **conexiones** que se establecen en la interfaz de nuestra aplicación y cada una de ellas se denomina **Segue** (Se pronuncia Segway).

Un Segue puede representar 2 cosas:

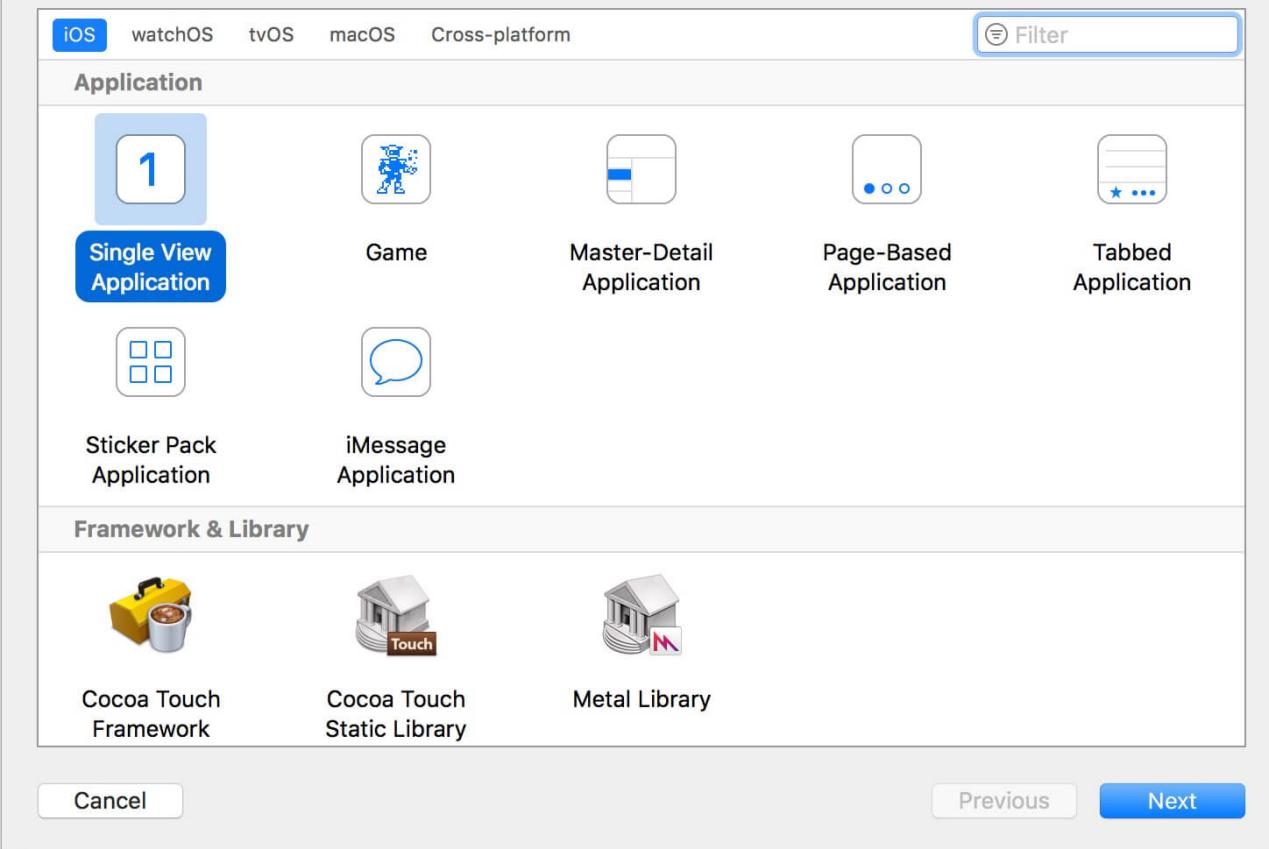
1. Una **transición** entre 2 escenas de nuestro storyboard
2. Una **relación** entre 2 ViewControllers de nuestro storyboard

En los siguientes ejemplos, vamos a ir viendo diferentes **tipos de Segues** que podemos utilizar.

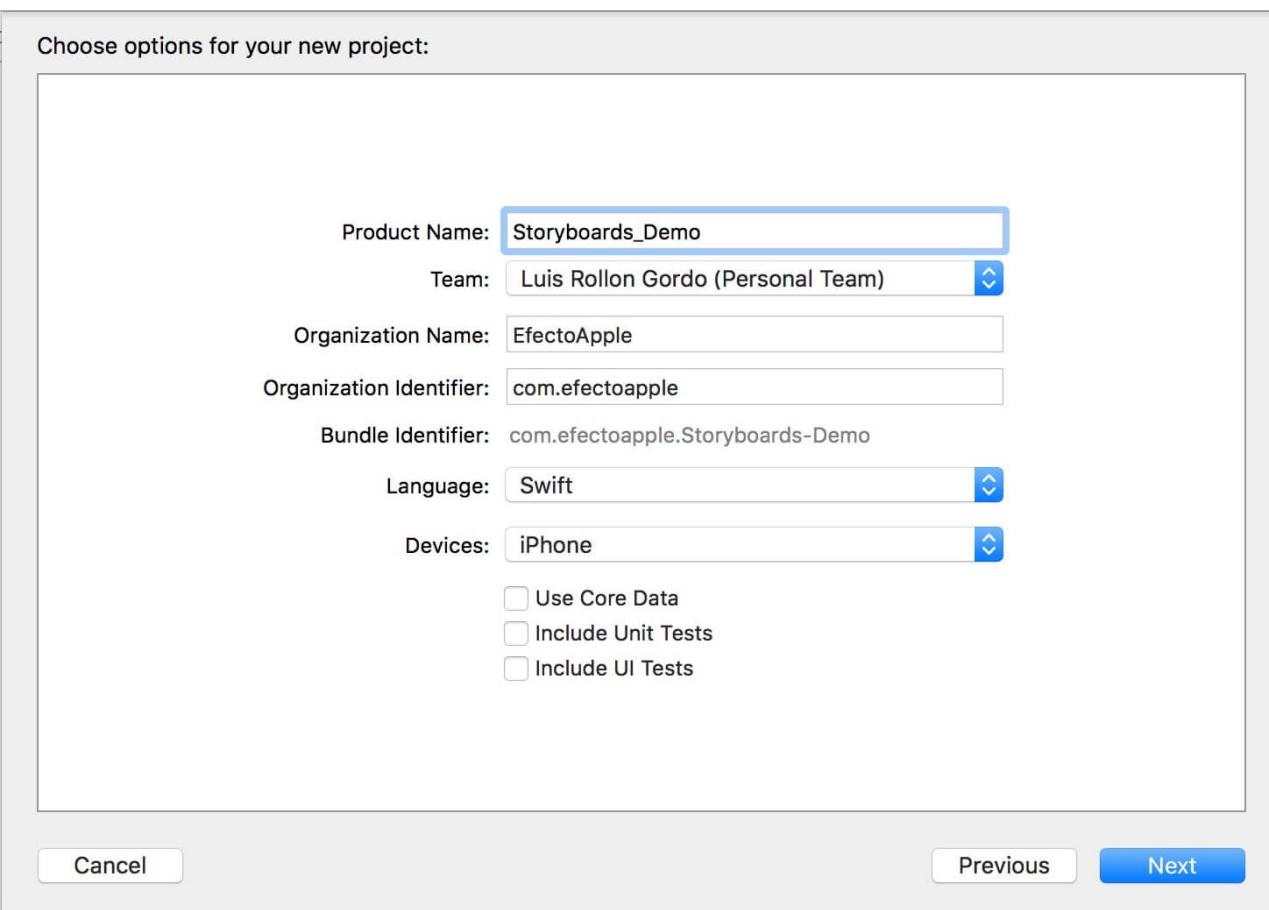
4. Segues de Transición Modal

Para comenzar crea un **nuevo proyecto** en Xcode, accediendo al menú **File > New > Project...** y elige la plantilla **Single View Application**:

Choose a template for your new project:

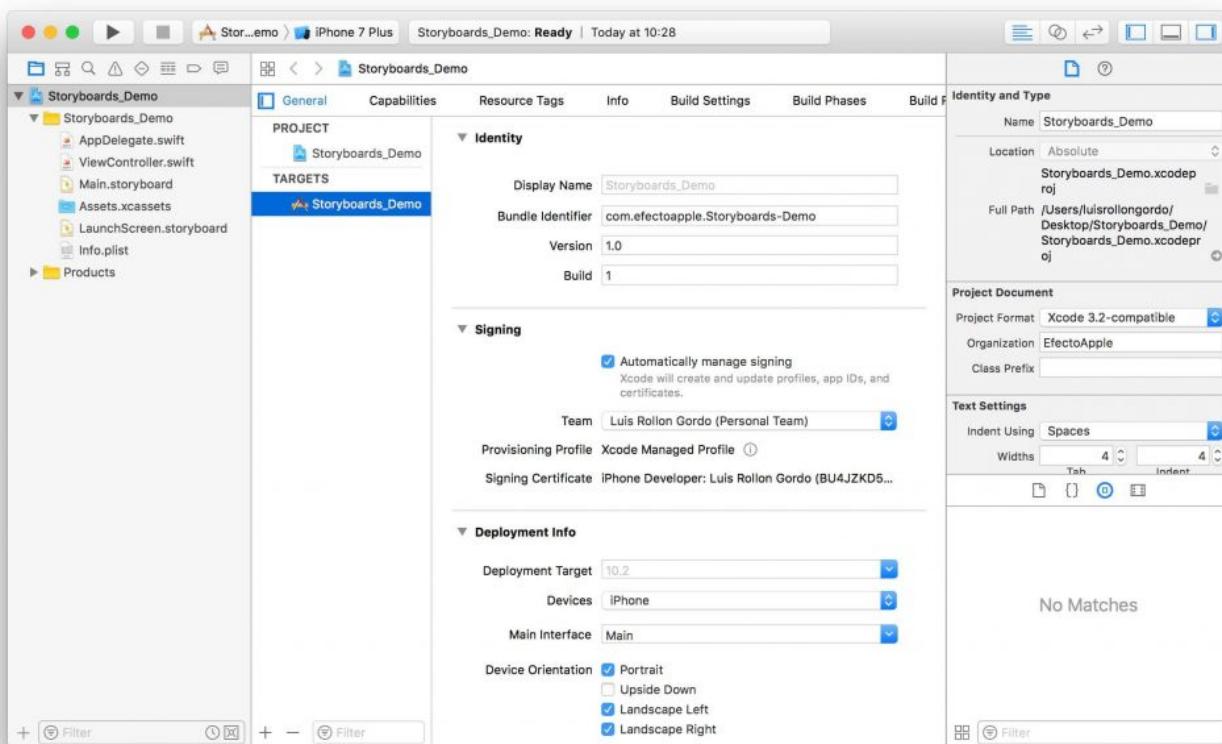


Después pulsa en **Next** y dale el nombre que quieras. En mi caso lo llamaré **Storyboards_Demo**. Deja el resto de opciones, tal cual las ves en la imagen:



Pulsa en **Next**, guarda el proyecto donde quieras y para finalizar pulsa en el botón **Create**.

Después de crear el proyecto, Xcode te mostrará algo así:



Una vez que tienes tu proyecto abierto en Xcode, accede al fichero *Main.storyboard* y verás que por defecto, tenemos un **ViewController** creado.

Bien, pues como además de ese ViewController, necesitamos otro, **añádelo** desde la Librería de Objetos.

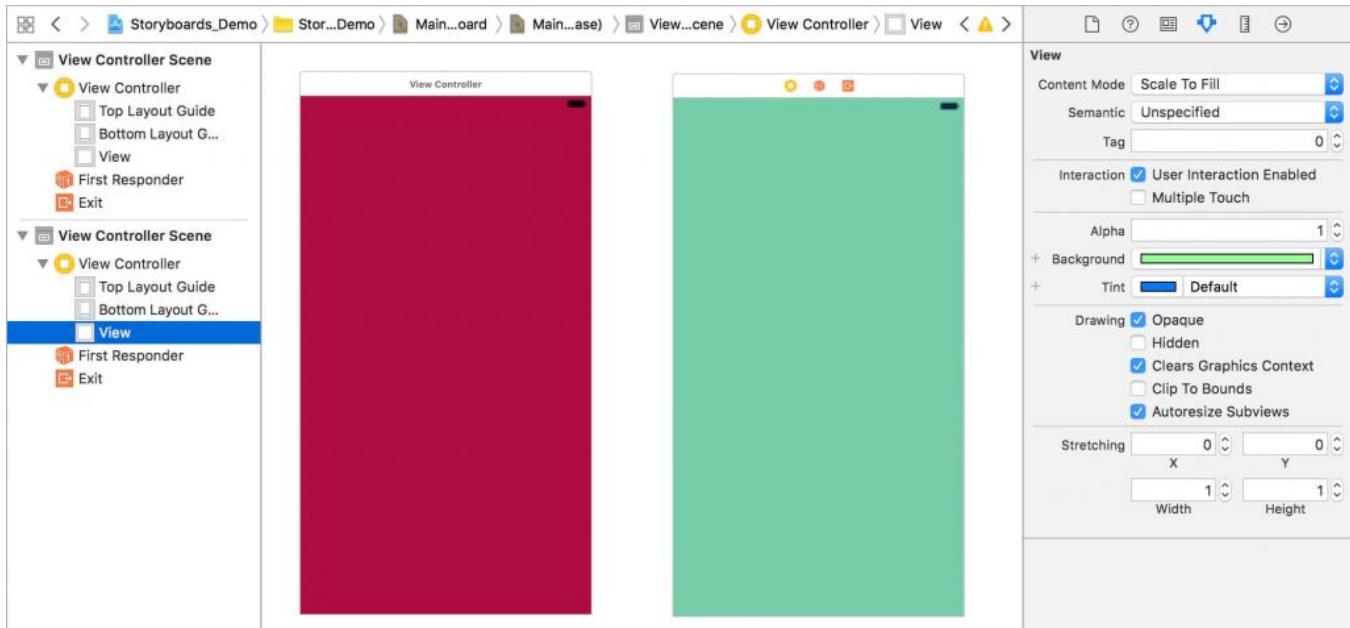
En la Primera Parte de este Tutorial hemos visto como añadir cualquier tipo de ViewController a nuestra interfaz, si no lo tienes claro, es el momento de revisarlo

Ahora que hemos añadido este ViewController, la interfaz de nuestra aplicación contará con **2 ViewControllers**, a los cuales vamos a cambiarles el color para diferenciarlos.

Para ello selecciona la **View** de cada uno de los ViewControllers y en la

propiedad **background**, elige el color que quieras.

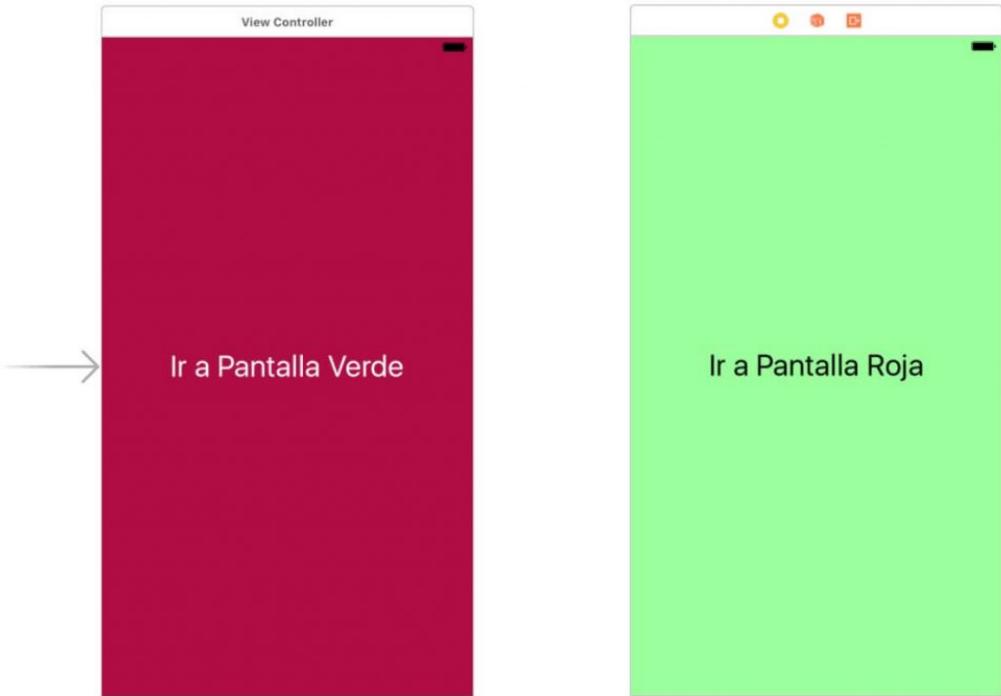
Deberías obtener algo parecido a esto:



Tu siguiente paso será añadir un **UIButton** al centro de cada uno de los dos ViewControllers y especificar el **texto** que quieras.

Además, deberás seleccionar el primero de tus ViewControllers y seleccionarlo como **Initial View Controller**, para que cuando ejecutemos el proyecto, nuestra aplicación muestre esta pantalla.

En este punto, tu interfaz debería ser similar a esta:



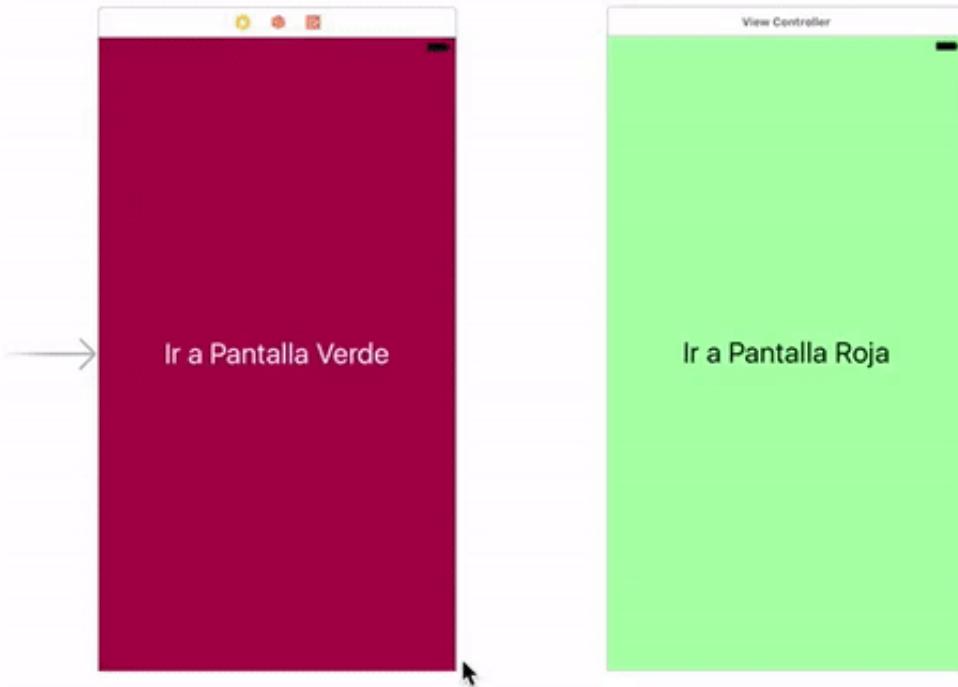
Lo que vamos a hacer ahora es ver la forma más sencilla de **presentar** un ViewController desde otro ViewController **de forma Modal**.

Presentando ViewControllers de forma modal

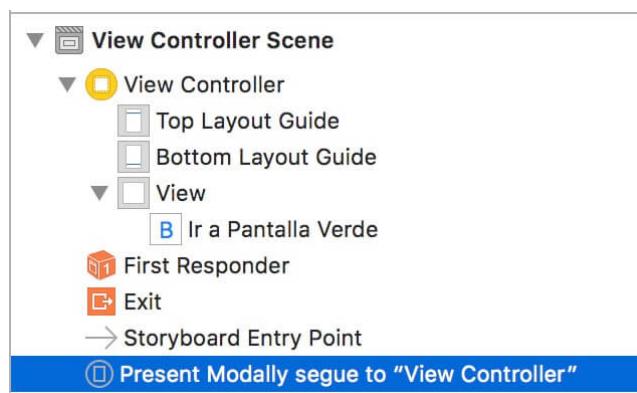
Lo primero que debes hacer es seleccionar el **botón** de tu primer ViewController, dejar la tecla **ctrl** pulsada y arrastrar hasta el **segundo ViewController**.

En ese momento aparecerá un **menú flotante** que te permite seleccionar el **tipo** de Segue de Transición que queramos elegir.

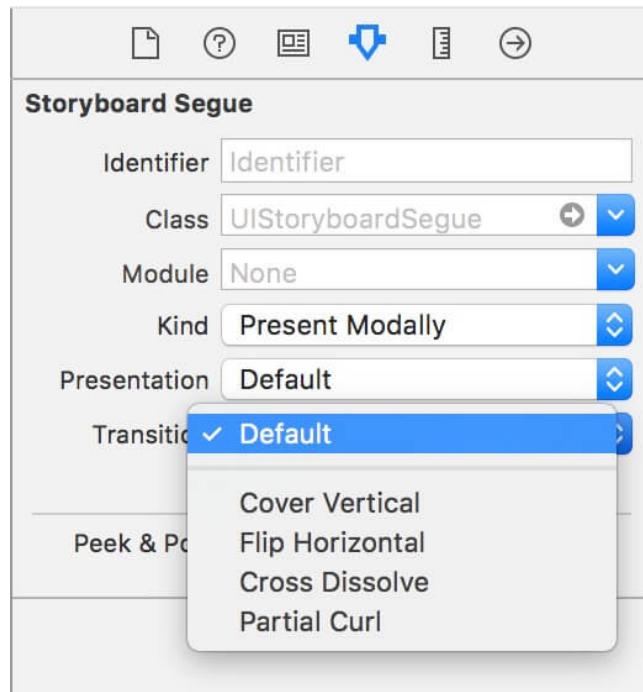
Selecciona la opción **Present Modally** y verás como en nuestra interfaz aparece al momento un **Segue de Transición** entre nuestras pantallas.



Además, en el **Document Outline** puedes apreciar como en la primera **Escena** de nuestra aplicación se ha añadido **este Segue**.



Si seleccionas el **Segue** haciendo click sobre él y después accedes al **Inspector de Atributos**, verás como en la opción **Transition** puedes seleccionar el **tipo de animación** que realizará tu aplicación cuando pase de una Escena a otra.



Ejecuta la aplicación y al pulsar sobre el botón del primer ViewController, nuestra aplicación realizará una **transición** de abajo hacia arriba para mostrar el segundo ViewController:

Sin embargo, si pulsas sobre el botón del segundo ViewController, verás como no se produce **ningún efecto**. Esto se debe a que no hemos asociado **ningún segue** a ese botón.



Vamos a solucionarlo.

Añadiendo un nuevo Segue

Selecciona el **botón** del segundo ViewController y dejando la tecla **ctrl** pulsada arrastra hasta el primer ViewController. Después, al igual que hemos hecho antes, selecciona la opción **Present Modally** en el menú

flotante que aparece.

Verás, que ahora tu aplicación tiene **dos Segues**.

Vamos a **modificar** el efecto de transición de nuestro segundo Segue.

Selecciona el Segue que acabamos de crear y desde el **Inspector de Atributos**, en su opción **Transition**, elige **Flip Horizontal**.

Ejecuta la aplicación y comprueba como ahora si, podemos navegar desde una pantalla a otra y además utilizando dos **animaciones** totalmente **diferentes**.



Como puedes ver, sin tener que escribir **ni una sola linea de código** hemos creado una aplicación que realiza transiciones entre pantallas de forma modal.

Sin embargo, **no es la única forma** de implementar una navegación básica, existen algunas otras como la que vamos a ver a continuación.

5. Segues de Transición con NavigationController

Una de las formas más sencillas y útiles de crear una navegación completa en una aplicación es utilizando un **NavigationController**.

Vamos a ver un ejemplo muy sencillo en el que **navegaremos entre 3 vistas** diferentes.

Para comenzar, accede al *Main.storyboard* de tu proyecto y elimina cualquier ViewController que tengamos de ejemplos anteriores.

Después, **sigue estos pasos** para crear la interfaz:

1. Añade un NavigationController a nuestra aplicación
2. Elimina el Root View Controller que trae asociado dicho NavigationController
3. Añade 3 ViewControllers a la interfaz
4. Incorpora un UIButton al primer ViewController con el texto “Ir a siguiente vista”
5. Agrega un UIButton al segundo ViewController con el texto “Ir a siguiente vista”
6. En el tercer ViewController añade un UILabel con el texto “Vista Final”
7. Modifica el color de fondo de las Views de los 3 ViewControllers
8. Haz que el NavigationController sea el Initial View Controller

Después de seguir este proceso, tu **interfaz** debería ser parecida a esta:



Añadiendo Navegación a nuestras escenas

Ya tenemos creadas **todas las escenas** de nuestra aplicación.

Nuestro siguiente paso será **añadir la navegación** entre ellas.

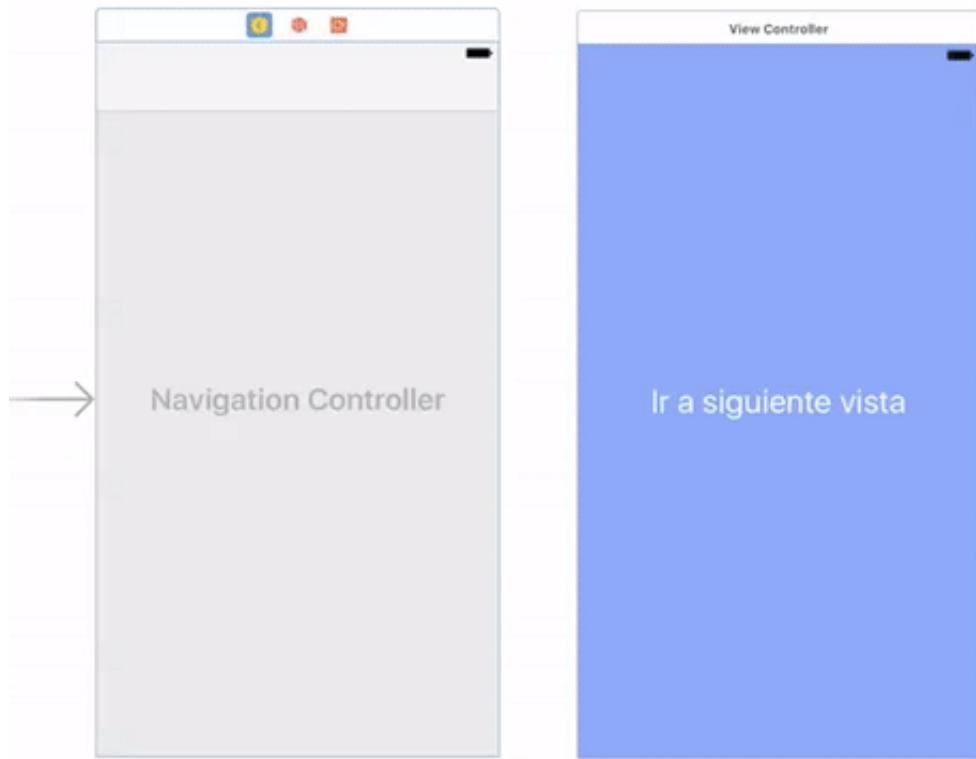
En la Primera Parte de nuestro Tutorial vimos que todo NavigationController debe llevar asociado un **Root View Controller**.

En nuestro caso, todavía **no hemos especificado** cual es el Root View Controller de nuestro NavigationController.

Para determinar esto, hacemos click en el **NavController** y dejando la tecla **ctrl** pulsada, arrastramos hasta nuestro primer ViewController.

Al soltar verás que aparece el típico **menú flotante**.

En este caso, la opción que elegiremos será **root view controller**.



Ahora que ya hemos especificado esto, si ejecutaras la aplicación, verías como aparecería nuestro primer ViewController con el botón **Ir a siguiente vista**.

Además, si te fijas, en la parte superior de la aplicación aparece la **NavigationBar** del NavigationController.



Si recuerdas, al comienzo de este tutorial, hablamos de dos tipos de Segues. Segues de **Transición** y Segues de **Relación**.

Este último Segue que hemos creado es un **Segue de Relación**, ya que relaciona el NavigationController con su Root View Controller.

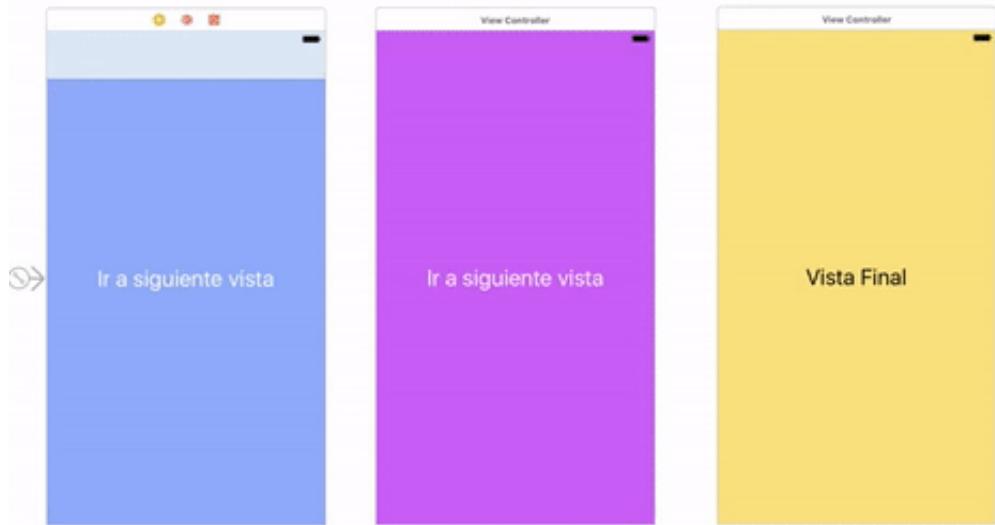
Ahora, vamos a utilizar **Segues de Transición** para relacionar unos ViewControllers con otros.

Añadiendo Segues con un NavigationController

El proceso que vamos a seguir es muy sencillo.

Selecciona el botón situado en el primer ViewController y dejando la tecla **ctrl** pulsada arrastra hasta el segundo ViewController y en el menú flotante que aparece, elige como **tipo de Segue**, la opción **Show**.

A continuación repite la operación con el segundo ViewController:



Mientras hemos ido añadiendo el Segue a cada uno de los ViewControllers, te habrás fijado que en la parte superior se ha ido añadiendo la **NavigationBar**.

Si ahora ejecutas la aplicación, verás que **puedes navegar** de un ViewController a otro sin ningún problema.



De nuevo hemos hecho posible la navegación entre las pantallas de nuestra aplicación sin tener que escribir **ni una sola linea** de código.

Además, esta forma de pasar de una pantalla a otra utilizando un NavigationController es algo que **se utiliza constantemente** en el Desarrollo de Aplicaciones iOS, así que es importante que hayas comprendido el funcionamiento completo del proceso.

6. Segues de Relación con TabBarController

Vamos a realizar un último ejemplo en el que trabajemos con **Segues de Relación**.

Esta vez vamos a utilizar un **TabBarController**.

Si recuerdas lo que vimos en la Primera Parte del Tutorial, los TabBarControllers son perfectos para establecer una navegación entre pantallas que **no** tienen relación jerárquica.

Se suelen utilizar mucho para diferenciar **entre varios menús** dentro de nuestra Aplicación.

Para comenzar, **elimina** cualquier ViewController que tengas en tu Main.storyboard, seleccionando y pulsando en la tecla **Delete**, para que la interfaz de nuestro proyecto quede completamente limpia.

Ahora, añade un **TabBarController**.

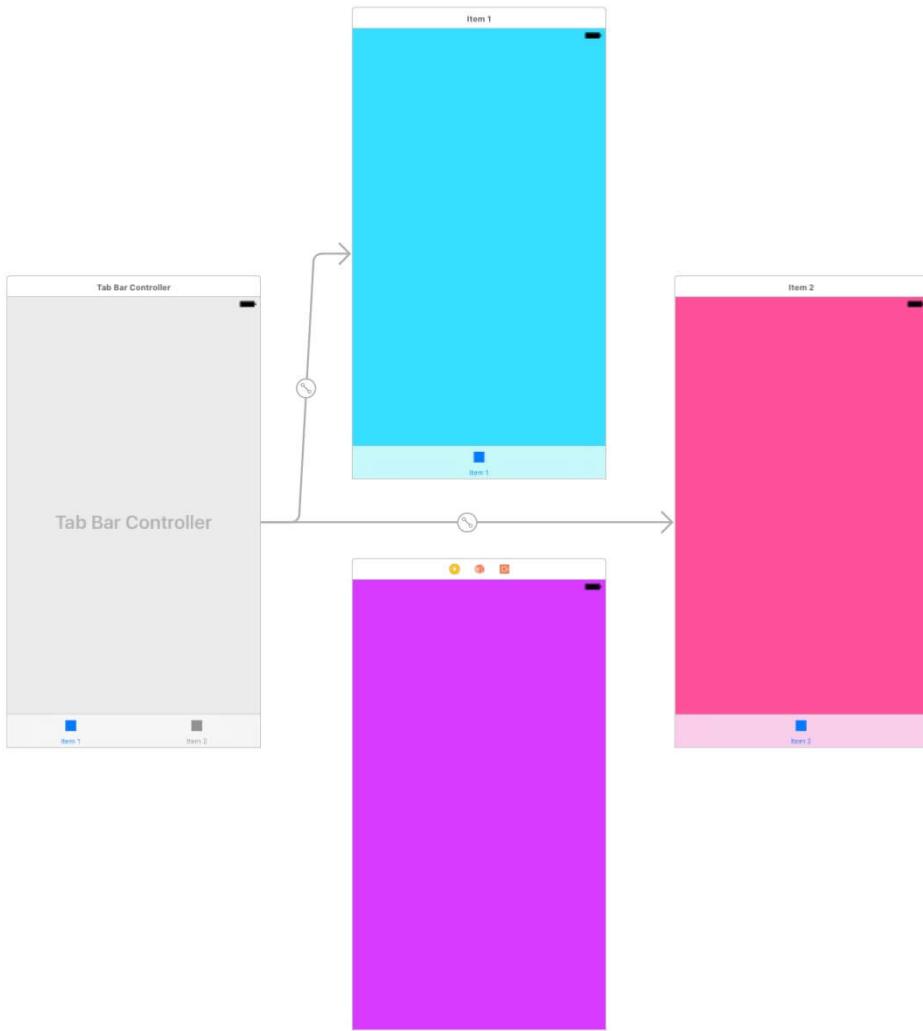
Verás que al añadirlo, va asociado a 2 ViewControllers, cada uno de los cuales representa un **menú** diferente.

Como nuestro ejemplo constará de 3 pantallas, **añade otro ViewController** adicional desde la Biblioteca de Objetos.

Ahora, como hemos hecho anteriormente, selecciona la **View** de cada uno de los ViewControllers y cambia su **color** de fondo.

Por último, selecciona el TabBarController y haz que sea el **Initial ViewController** de nuestra app.

Después de haber seguido estos pasos, tu **interfaz** debería ser algo así:



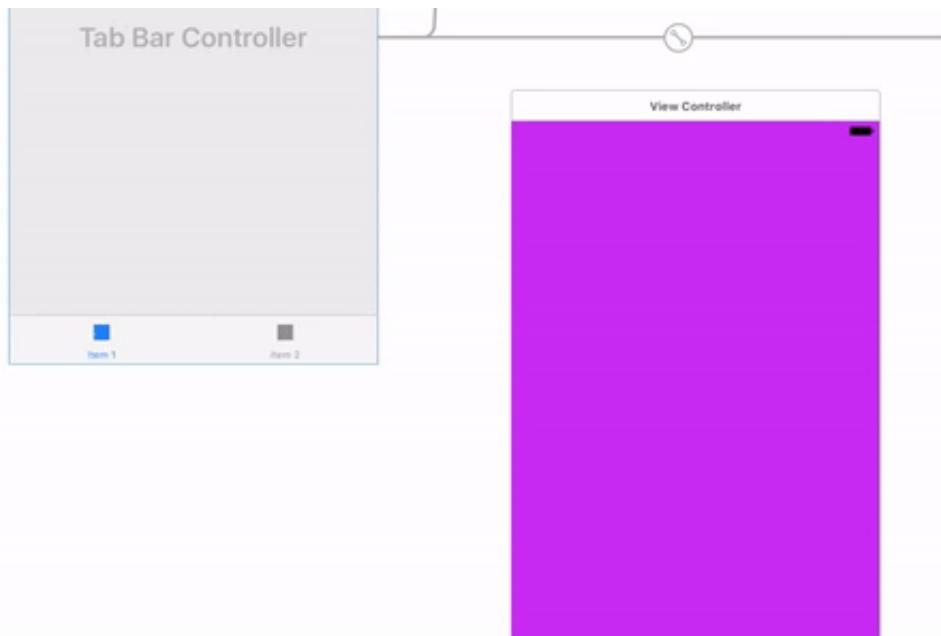
Elementos que componen nuestra interfaz

Como puedes observar, tenemos los siguientes **componentes**:

- Un TabBarController
- Un ViewController enlazado al TabBarController
- Otro ViewController enlazado al TabBarController
- Un tercer ViewController que no está enlazado al TabBarController

Como queremos, que nuestra interfaz esté formada por **3 pantallas** asociadas al TabBarController, crearemos un **Segue de Relación** para enlazar nuestro último ViewController al TabBarController.

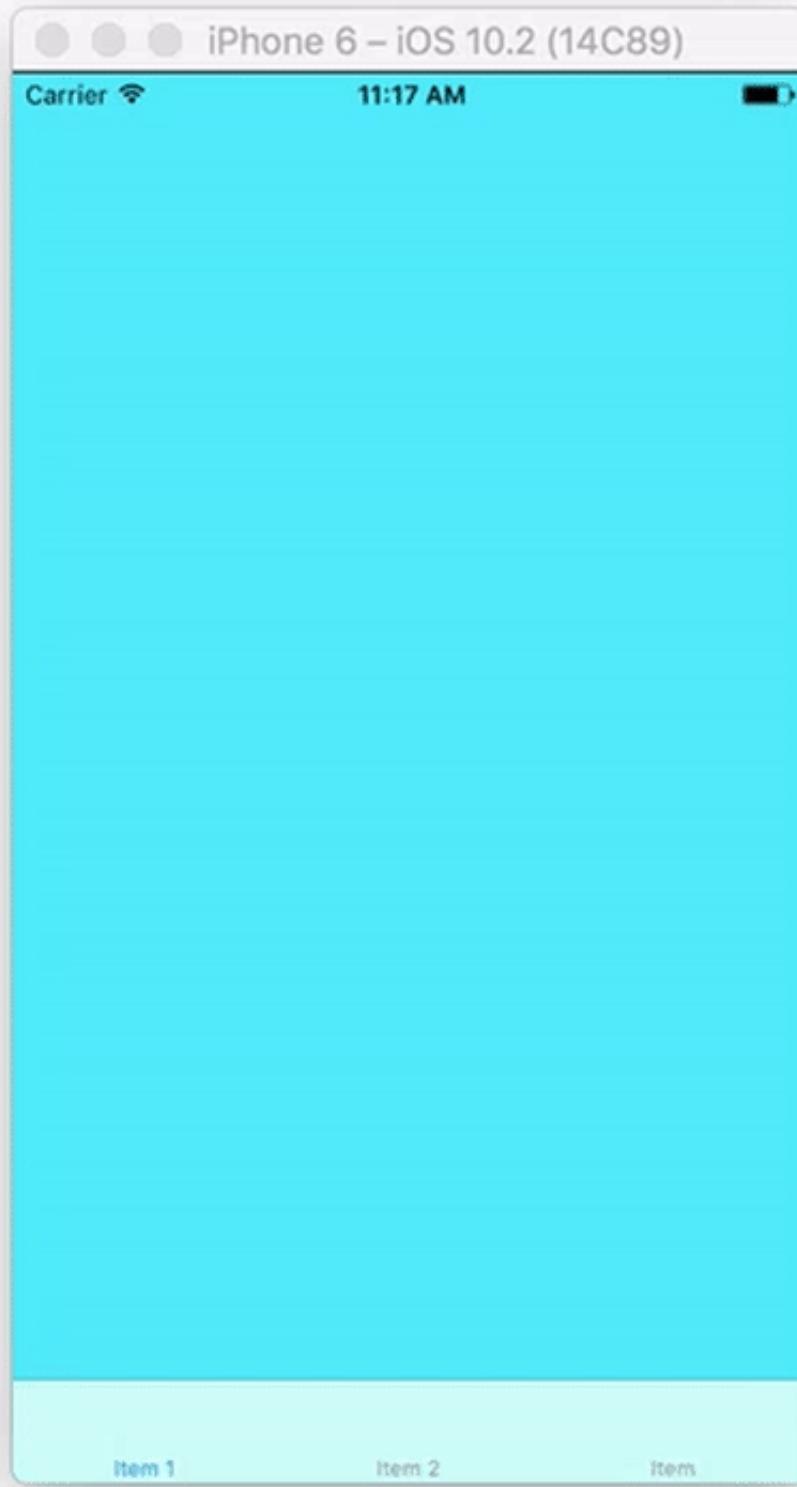
Para ello, dejando pulsada la tecla **ctrl**, arrastra desde el TabBarController hasta el último ViewController. Después, en el menú flotante que aparece, selecciona la opción **Relationship Segue: view controllers**.



Al momento de realizar esta conexión, nuestro TabBarController incorporará un **Item** más a su TabBar, lo que nos permitirá acceder a este ViewController directamente a través del botón derecho del TabBar.

Ahora, prueba a **ejecutar** la aplicación y verás como puedes **cambiar** fácilmente **de pantalla** utilizando el **NavigationBar** que hemos implementado.

De nuevo, todo, sin tener que escribir **ni una sola linea** de código.



7. Resumen Final

En esta **Segunda Parte** de nuestro Tutorial sobre Storyboard, nos hemos centrado de uno de los aspectos más importantes: Los Segues.

A través de varios ejemplos, hemos podido ver como utilizar Segues para

relacionar **cualquier pantalla** de nuestra aplicación.

Además a través de los Segues de Transición hemos creado ejemplos de aplicaciones que realizan **navegaciones sencillas** a través de las pantallas de nuestras interfaces.

Con lo que hemos visto en esta Segunda parte del Tutorial puedes desarrollar **apps multivista** que te permitirán mostrar varias pantallas al usuario, ofreciéndole una forma de navegar entre ellas.

Este Tutorial cuenta con una **Tercera Parte** que veremos a continuación, donde desarrollaremos una Aplicación completa utilizando todos los **conceptos** que hemos ido viendo en las dos primeras partes del Tutorial.

Introducción a los Storyboards

[Parte 3]

Como usarlos en tus aplicaciones iOS con Swift

Lenguaje Swift | Nivel Principiante

1. Introducción

Ya hemos visto 2 partes del Tutorial sobre Storyboards.

Ahora en la tercera y última parte de este Tutorial, veremos como aplicar todo lo que hemos visto, desarrollando una **Aplicación iOS multivista**.

2. ¿Qué vamos a ver en este Tutorial?

Vamos a poner en práctica todo lo aprendido hasta ahora y desarrollamos una **Aplicación de Gestión de Tareas**.

Esta aplicación es interesante para comprender algunos conceptos fundamentales del Desarrollo iOS.

Sin embargo, se trata de una app que cuenta con **una única vista**, cuando lo normal, en cualquier proyecto, es que conste de varias pantallas diferentes.

En este Tutorial, vamos a crear una **versión mejorada** de nuestra App de Gestión de Tareas.

Estos son los puntos **más importantes** que veremos juntos:

1. Crear el Modelo de nuestra Aplicación
2. Hacer que el usuario pueda crear objetos de tipo Task
3. Almacenar los objetos de tipo Task en nuestro Modelo
4. Utilizar una celda personalizada para mostrar las tareas

5. Añadir Segues entre las pantallas de la interfaz
6. Aprovechar los Unwind Segues para la navegación de nuestra app

3. ¿Qué Aplicación vamos a desarrollar?

Como hemos comentado antes realizaremos una Aplicación de **Gestión de Tareas**.

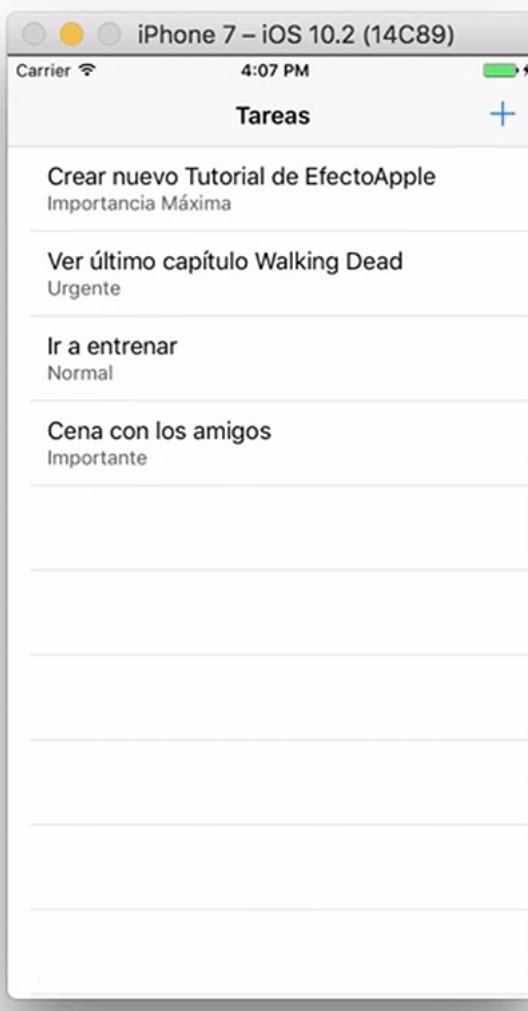
Esta app constará de **3 pantallas** diferentes.

Para realizar la navegación entre estas pantallas, utilizaremos los dos tipos de **transiciones** que vimos en la Segunda parte del Tutorial:

- Presentación Modal de View Controllers
- Navegación utilizando un Navigation Controller

Así, veremos como poner en práctica los **conocimientos teóricos** que vimos en las partes anteriores de nuestro Tutorial.

Cuando hayas terminado el proyecto, habrás creado una **aplicación** que tendrá el siguiente aspecto:



Como ves, se trata de una aplicación sencilla que nos permitirá **añadir tareas** a nuestro listado de Tareas pendientes.

Cada vez que pulsamos en el botón +, la aplicación nos permite añadir una nueva tarea especificando su nombre y su prioridad.

Al momento esa tarea aparecerá en nuestro listado de Tareas pendientes.

Se trata de una aplicación sencilla pero que al mismo tiempo nos servirá para comprender mejor los fundamentos de las **aplicaciones multivista**.

¡Comenzamos!

4. La Interfaz de nuestro Proyecto de Inicio

Para no alargar en exceso el Tutorial y poder centrarnos en las partes

realmente importantes, he creado un **Proyecto de Inicio** que podrás descargar [desde aquí](#).

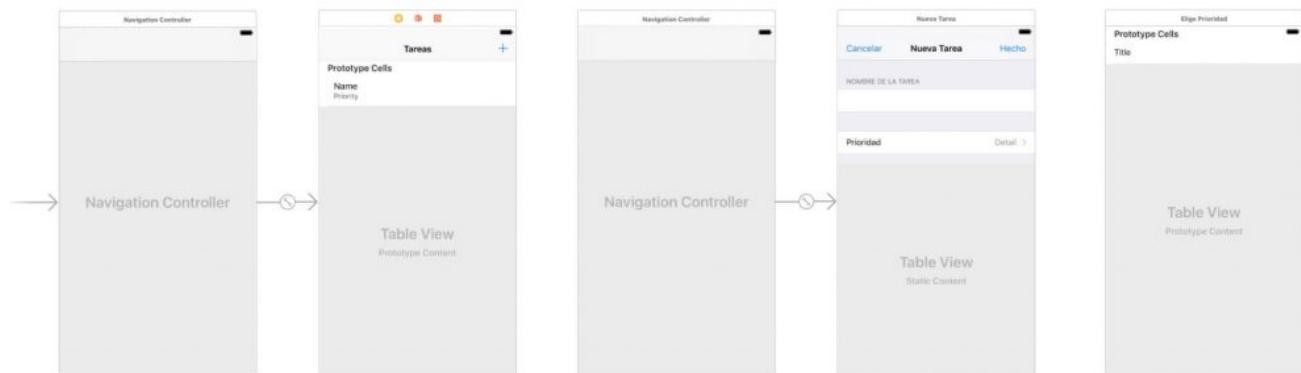
Una vez descargado, descomprímelo y **ábrelo** con Xcode.

Si intentas compilar el proyecto verás que se producen muchos errores.

Esto se debe a que el código está **incompleto** y lo iremos completando a lo largo de este Tutorial.

Accede al fichero *Main.storyboard*.

Allí podrás ver **las 3 pantallas** que forman nuestra Interfaz y **2 Navigation Controllers** que nos facilitarán la navegación entre las pantallas.



Pantalla de Inicio



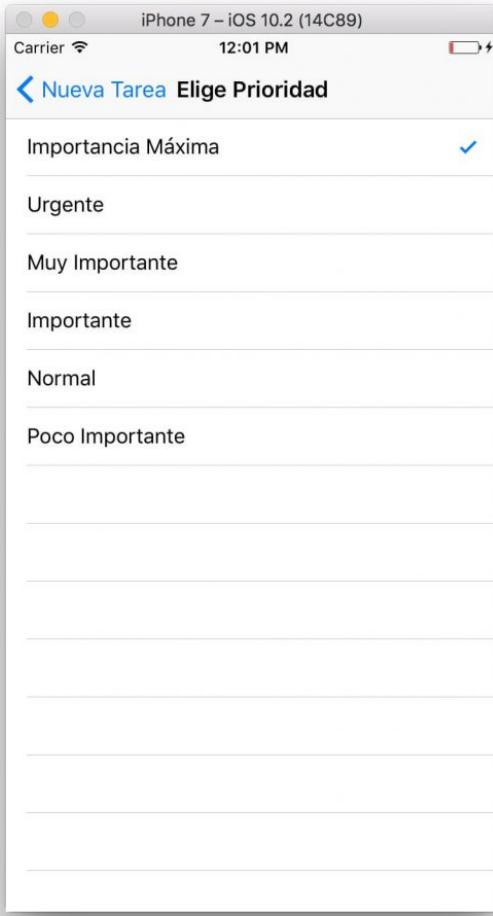
Es la pantalla que se muestra nada más arrancar la Aplicación. Nos permite mostrar el **listado de tareas** pendientes y a través de su botón situado en la esquina superior derecha podemos añadir nuevas tareas.

Añadiendo una nueva tarea



Esta vista se muestra cuando el usuario pulsa sobre el botón +. En ella, el usuario podrá especificar el **nombre** de la tarea y su **prioridad**. Dispone de dos botones en la parte superior que le permiten cancelar en cualquier momento el proceso o confirmar la adición de la tarea nueva.

Especificando la prioridad



Cuando el usuario pulse en la celda Prioridad, se mostrará nuestra tercera vista. En ella, el usuario podrá especificar entre un listado prefijado de opciones, **cual es la prioridad** que quiere asignarle a la tarea. En el momento, en que el usuario selecciona la prioridad, automáticamente nuestro View Controller regresa a la pantalla anterior.

5. Las Clases de nuestro Proyecto de Inicio

Después de revisar la interfaz de la aplicación, vamos a ver las **clases** que tenemos creadas en nuestro proyecto.

Si te fijas en el Navegador de Xcode, situado en la parte izquierda, verás que tenemos creadas las siguientes clases:

TasksViewController

Controla la pantalla de inicio de nuestra aplicación. Gestiona la TableView donde mostramos el **listado** de tareas pendientes. Además deberá tener

un método que se ejecute cuando el usuario pulse en el botón + para añadir una nueva tarea al listado.

AddTaskTableViewController

Gestiona la pantalla donde el usuario podrá añadir una nueva tarea. Desde aquí gestionaremos el comportamiento del **textField** cuando el usuario pulse sobre el campo **Nombre** y además realizaremos la **transición** hacia la pantalla de Prioridades cuando el usuario toque en el campo **Prioridad**.

PriorityTableViewController

Controla la pantalla en la que el usuario especifica la prioridad de la tarea que está creando. Gestionará la TableView que muestra las **diferentes prioridades** y cuando el usuario elija una de ellas deberá de volver a la pantalla anterior.

TaskCell

Se ocupa de la **celda personalizada** que se utiliza en la TableView de la pantalla principal.

Task

Esta clase forma el **modelo** de datos de nuestra aplicación. Representa a un **objeto Tarea**, que cuenta con **Nombre** y **Proridad** como atributos.

SampleData

Se trata de una clase auxiliar que nos permite añadir **datos iniciales** a nuestra aplicación, para que al ejecutarla por primera vez, el usuario vea algunas tareas creadas de ejemplo.

6. ¿Cómo vamos a trabajar en este Proyecto?

Si has podido echar un vistazo a las clases que aparecen en el Proyecto de Inicio, verás que todos los métodos **están vacíos**.

El objetivo de este tutorial es ir llenando **todos los métodos** de cada una de las clases de nuestro proyecto, mientras vamos explicando en qué consiste el código que vamos desarrollando.

Además tendremos que añadir los **Segues** que necesitaremos para que podamos ir navegando entre las pantallas de la aplicación.

Por tanto, estos serán nuestros objetivos:

- Rellenar todos los métodos que forman nuestra app
- Crear las transiciones necesarias para que la aplicación funcione

7. Creando el Modelo de nuestra Aplicación

Uno de tus primeros pasos cuando desarrolles cualquier aplicación debe ser tener claro el **modelo de datos** que vas a utilizar.

Si no tienes claro a qué nos referimos con el Modelo de una aplicación, deberías echar un vistazo a este Tutorial: [MVC en Aplicaciones iOS](#).

En esta aplicación utilizaremos un Modelo muy sencillo que consistirá en una **clase Task** que representará a cada una de las tareas con las que trabajaremos.

La información que tenemos que almacenar para cada tarea será la siguiente:

- Nombre
- Prioridad

Abre el fichero *Task.swift* y añade el siguiente código:

```
struct Task {  
    var name: String?  
    var priority: String?  
    init(name: String?, priority: String?) {  
        self.name = name  
    }  
}
```

```
    self.priority = priority  
}  
}
```

Simplemente creamos un **struct** que representará a cada una de las Tasks y que constará de dos propiedades: **name** y **priority** y un **inicializador personalizado** que inicializa dichas propiedades.

Además de nuestro modelo, vamos a utilizar otra **clase auxiliar** que únicamente nos servirá para mostrar una serie de datos iniciales en la aplicación. Abre *SampleData.swift* y añade lo siguiente:

```
let tasksData = [  
  
    Task(name:"Crear nuevo Tutorial de EfectoApple", priority:"Importancia  
Máxima"),  
  
    Task(name: "Ver último capítulo Walking Dead", priority: "Urgente"),  
  
    Task(name: "Ir a entrenar", priority: "Normal")  
]
```

Como ves, hemos creado un **array de objetos Task** que hemos inicializado con algunos datos.

Con estas dos clases, habríamos creado todo lo necesario para configurar nuestro Modelo de Datos.

8. Pantalla Inicial de nuestra App

Comenzaremos por la pantalla de inicio de nuestra aplicación.

Abre la clase *TasksTableViewController.swift* y verás que tenemos 2 partes diferenciadas. Los 3 primeros métodos son los típicos **métodos de TableView**, mientras que el método **saveTaskDetail()** se ejecutará cuando el usuario pulse el botón Hecho.

Además necesitaremos **un array** donde almacenar nuestras tareas.

Para añadir este array escribe lo siguiente en la parte superior de la clase:

```
var tasks:[Task] = tasksData
```

Hemos creado el array **tasks** y además lo hemos inicializado con nuestra clase *SampleData.swift*.

A continuación añade los métodos del TableView a la clase:

```
1 override func numberOfSections(in tableView: UITableView) -> Int {  
2     return 1  
3 }  
4 override func tableView(_ tableView: UITableView,  
5 numberOfRowsInSection section: Int) -> Int {  
6     return tasks.count  
7 }  
8 override func tableView(_ tableView: UITableView, cellForRowAt  
9 indexPath: IndexPath)  
10 -> UITableViewCell {  
11     let cell = tableView.dequeueReusableCell(withIdentifier: "TaskCell", for:  
12 indexPath)  
13     as! TaskCell  
14     let task = tasks[indexPath.row] as Task  
15     cell.task = task  
16     return cell  
17 }
```

No vamos a explicar estos métodos, ya que hemos hablado de ellos en el primer tutorial de todos.

Además de esto, tendrás que añadir el código correspondiente al método **saveTaskDetail()**:

```
@IBAction func saveTaskDetail(_ segue:UIStoryboardSegue) {  
    if let addTaskTableViewController = segue.source as? AddTaskTableViewController {  
        //Añadimos la tarea al array de tareas  
        if let task = addTaskTableViewController.task {  
            tasks.append(task)  
            //Actualizamos la TableView  
            let indexPath = IndexPath(row: tasks.count-1, section: 0)  
            tableView.insertRows(at: [indexPath], with: .automatic)  
        }  
    }  
}
```

Hemos creado un objeto de tipo **AddTaskTableViewController** a partir de la pantalla de la que proviene el segue.

Después hemos obtenido la tarea a partir de su variable **task** y la hemos añadido al array de tareas.

Por último hemos añadido una row a nuestra **TableView**, ya que ahora vamos a tener una Task mas.

9. Nuestra clase TaskCell

Accede a la clase *TaskCell.swift*. El código que vamos a añadir será muy sencillo.

Lo único que haremos será crear una propiedad de tipo **Task** dentro de nuestra clase y le añadiremos un observador que nos permitirá

especificar los campos **taskLabel** y **priorityLabel** de nuestra celda a partir de los valores del objeto task.

Solo tienes que añadir **este código** a continuación de los Outlets:

```
var task: Task! {  
    didSet {  
        taskLabel.text = task.name  
        priorityLabel.text = task.priority  
    }  
}
```

De esta manera, nuestra clase *TaskCell.swift* estaría terminada.

10. Pantalla para añadir una nueva tarea

El siguiente punto de la aplicación donde nos centraremos será en la pantalla donde el usuario puede crear una nueva tarea. Recuerda que para acceder a esta vista, deberá pulsar en el botón + situado en la esquina superior derecha de la pantalla principal.

Abre la clase *AddTaskTableViewController.swift*.

Lo primero que tendremos que añadir será una propiedad de tipo **Task** que representará a la tarea que estamos añadiendo. Incorpora este código justo después de los Outlets:

```
var task:Task?  
  
var priority:String = "Importancia Máxima" {  
    didSet {  
        detailLabel.text? = priority  
    }  
}
```

Además, como puedes ver, hemos incorporado un **observador** que añade la prioridad de la tarea como texto del detailLabel de la celda. También hemos especificado que por defecto, la prioridad de una tarea sea **“Importancia Máxima”**.

En el método **didSelectRowAt()** añadiremos lo siguiente:

```
override func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {  
    if indexPath.section == 0 {  
        nameTextField.becomeFirstResponder()  
    }  
}
```

Lo único que hace es que cuando el usuario pulse en la celda donde especificará el nombre de la tarea, hacemos que el foco pase al **textField**, para que de esa forma, recoja el texto escrito por el usuario.

El Método **Prepare()**

El siguiente método que completaremos, será **prepare()**.

Este método se ejecuta justo antes de que la pantalla vaya a hacer un **segue** hacia otra vista:

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {  
    if segue.identifier == "SaveTaskDetail" {  
        task = Task(name: nameTextField.text, priority:priority)  
    }  
    if segue.identifier == "PickPriority" {  
        if let priorityTableViewController = segue.destination as? PriorityTableViewController {  
            priorityTableViewController.selectedPriority = priority  
        }  
    }  
}
```

```
}
```

```
}
```

Esta pantalla puede hacer un segue producido porque el usuario pulse el botón Hecho, es decir, quiera guardar la tarea (**SaveTaskDetail**). Lo que haremos será crear un nuevo objeto task, que inicializaremos con el nombre y la prioridad que haya definido el usuario.

O puede hacer un segue cuando el usuario pulse en la celda Prioridad (**PickPriority**). En ese caso, tendremos que pasar la propiedad priority a nuestro siguiente ViewController.

Añadiendo un método Unwind

Por último, tendremos que controlar que si se ha hecho un unwind desde la pantalla **PriorityTableViewController**, tendremos que actualizar la propiedad priority con la prioridad elegida por el usuario. Esto lo haremos con el siguiente método **unwind**:

```
@IBAction func unwindWithSelectedTask(_ segue:UIStoryboardSegue) {  
    if let priorityTableViewController = segue.source as? PriorityTableViewController,  
        let selectedPriority = priorityTableViewController.selectedPriority {  
            priority = selectedPriority  
        }  
}
```

Con esto, habríamos terminado nuestra clase *AddTaskTableViewController*.

Si quieres conocer más en detalle como funciona el **Unwind Segue**, puedes echar un vistazo a nuestra sección de [Conceptos](#)

11. Vista en la que seleccionamos Prioridad

Para finalizar la parte de código de nuestra aplicación, vamos a acceder a la clase *PriorityViewController.swift*.

Recordemos que esta pantalla lo que hará será mostrarnos una **TableView** con un listado de prioridades, que le permitirán al usuario

Lo primero que necesitaremos aquí, será una **variable** que almacene ese listado de prioridades. Así que añade lo siguiente a la parte superior de tu código:

```
var priorities:[String] = [  
    "Importancia Máxima",  
    "Urgente",  
    "Muy Importante",  
    "Importante",  
    "Normal",  
    "Poco Importante"]  
  
var selectedPriority:String? {  
    didSet {  
        if let priority = selectedPriority {  
            selectedPriorityIndex = priorities.index(of: priority)!  
        }  
    }  
}  
  
var selectedPriorityIndex:Int?
```

Además hemos añadido un observador que recupera el **index** de la prioridad y lo almacena en la variable **selectedPriorityIndex**, cada vez

que se modifica la propiedad **selectedPriority**.

Los Métodos del TableView

A continuación implementaremos los **métodos** del TableView:

El método **numberOfSections()** devuelve 1 ya que solo tenemos una sección:

```
override func numberOfSections(in tableView: UITableView) -> Int {  
    return 1  
}
```

El método **numberOfRowsInSection()** devolverá el número de prioridades que tenemos:

```
override func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {  
    return priorities.count  
}
```

El método **cellForRowAtIndex()** creará la celda de tipo TaskCell y actualizará el CheckMark en función de si la celda ha sido seleccionada o no por el usuario:

```
override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {  
    let cell = tableView.dequeueReusableCell(withIdentifier: "TaskCell", for: indexPath)  
  
    cell.textLabel?.text = priorities[indexPath.row]  
  
    if indexPath.row == selectedPriorityIndex {  
        cell.accessoryType = .checkmark  
    } else {  
        cell.accessoryType = .none  
    }  
}
```

```
    }

    return cell

}
```

Por ultimo, **didSelectRowAtIndexPath** actualizará el CheckMark de las celdas cuando el usuario pulse alguna de ellas:

```
override func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {

    tableView.deselectRow(at: indexPath, animated: true)

    //Otra celda ha sido seleccionada, tenemos que deseleccionar esta

    if let index = selectedPriorityIndex {

        let cell = tableView.cellForRow(at: IndexPath(row: index, section: 0))

        cell?.accessoryType = .none

    }

    selectedPriority = priorities[indexPath.row]

    //Actualiza el check para la celda actual

    let cell = tableView.cellForRow(at: indexPath)

    cell?.accessoryType = .checkmark

}
```

Añadiendo un nuevo método Unwind

Además, para terminar, cuando el usuario pulse en cualquier celda, tendremos que actualizar la propiedad **selectedPriority** y volver a la pantalla anterior. Por tanto, tendremos que utilizar un método unwind:

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

    if segue.identifier == "SaveSelectedTask" {

        if let cell = sender as? UITableViewCell {

            let indexPath = tableView.indexPath(for: cell)
```

```
if let index = indexPath?.row {  
    selectedPriority = priorities[index]  
}  
}  
}  
}
```

Con este último método habríamos terminado la clase *PriorityTableViewController*.

12. Añadiendo Segues a nuestra Aplicación

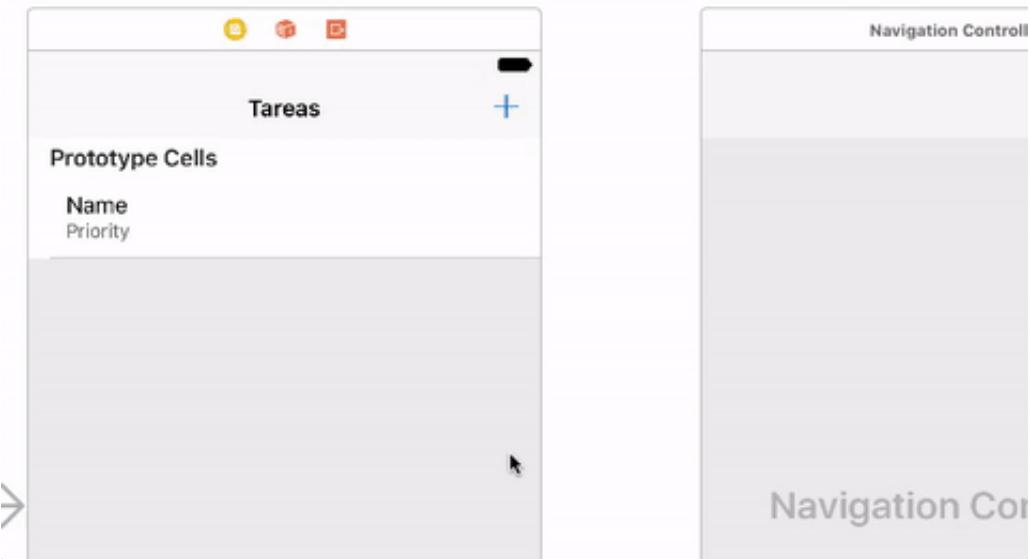
Una vez que hemos acabado de añadir el código de nuestra app, tendremos que centrarnos en **añadir los **segues**** que necesitemos para que la aplicación pueda ir navegando entre las pantallas de su interfaz.

Los 2 segues que necesitamos son estos:

- Desde el botón + hasta el siguiente Navigation Controller
- Desde la celda Prioridad hasta el View Controller
PriorityTableViewController

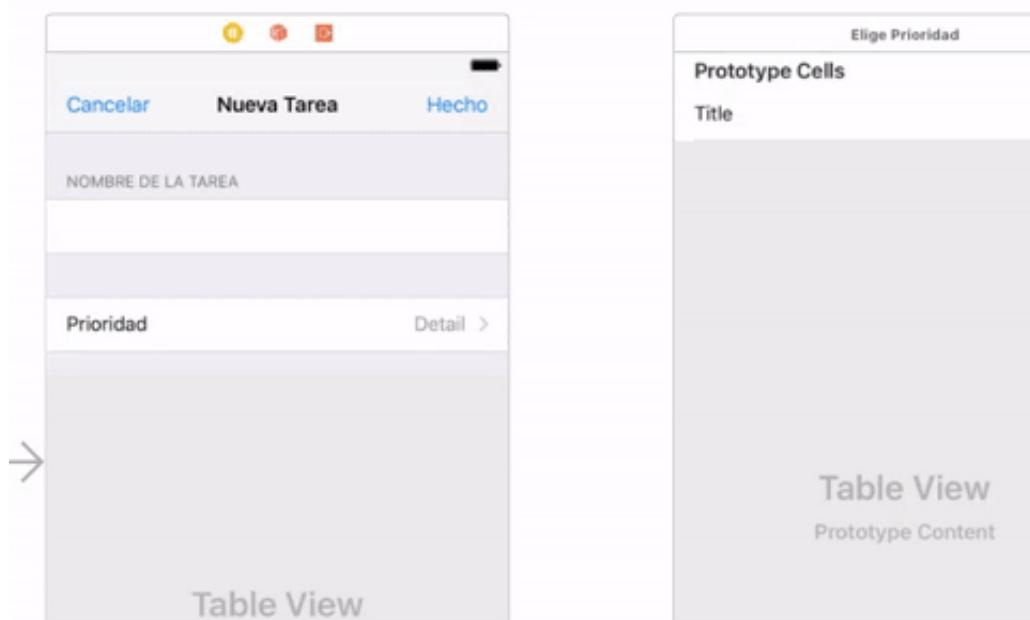
Vamos a añadir el primero.

Accede a *Main.storyboard*, y dejando pulsada la tecla **ctrl**, arrastra desde el botón + hasta el Navigation Controller situado a su derecha. En el menú flotante que aparecerá, elige **Present Modally**:

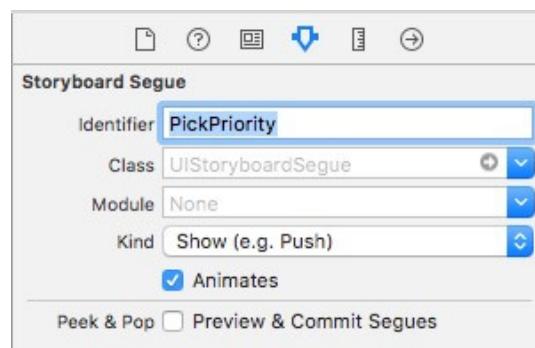


Después añadiremos el segundo segue.

Dejando también la tecla **ctrl** pulsada, arrastra desde la celda Prioridad hasta el View Controller PriorityTableViewController. En el menú flotante elige **Show**:



Después, selecciona el Segue y asignale como Identificador: **PickPriority**



De esa forma, habríamos **enlazado** las pantallas de la aplicación y ahora podríamos **navegar** de una a otra sin problema.

Ahora, puedes **ejecutar** la aplicación y comprobar que **funciona perfectamente**.

13. Resumen Final

¡Enhorabuena!, has desarrollado una **aplicación multivista** prácticamente **desde cero**.

Espero que lo que acabamos de ver en este desarrollo te ayude para tus **futuros proyectos**.

Crea una Pantalla de Login usando Swift y Firebase

Pantalla de Login para tus aplicaciones iOS

Lenguaje Swift | Nivel Intermedio

1. Introducción

Puede que en algún momento hayas pensado en desarrollar **tu propia aplicación iOS**.

Si se trata de una aplicación mas o menos ambiciosa, además de la parte de diseño y la parte de desarrollo iOS necesitarás implementar **la parte de servidor**.

Necesitarás un **backend** en el que se apoye tu aplicación. Si tienes conocimientos de desarrollo en la parte de servidor, genial, eres lo que hoy en día se llama un **Full Stack Developer** y sabes todo lo necesario para poder crear desde cero una **aplicación completa** por ti mismo.

Lamentablemente, esto **no** suele ser lo común.

Si no tienes conocimientos de desarrollo en la parte de servidor, tienes varias **opciones**:

1. **Aprender** lo necesario para crear el backend por ti mismo.
2. Encontrar a **alguien** de back que quiera unirse a tu proyecto.

3. Utilizar algunas de las **herramientas** disponibles hoy en día que te permiten crear de forma sencilla un backend completo para tus aplicaciones móviles.

Nosotros en este tutorial nos vamos a centrar en la **tercera opción**. Hablaremos de una de estas herramientas. Veremos como trabajar con **Firebase**.

2. ¿Qué vamos a ver en este Tutorial?

Es evidente que no podemos abordar el desarrollo completo de un **backend con Firebase** en este tutorial. Para poder explicarlo necesitaríamos toda una **serie de tutoriales**.

Por este motivo, voy a centrarme únicamente en una parte de nuestro backend.

Además, es un tema sobre el que he recibido algunos emails de lectores con dudas al respecto.

Vamos a ver **como crear una pantalla de Login** para nuestra aplicación iOS.

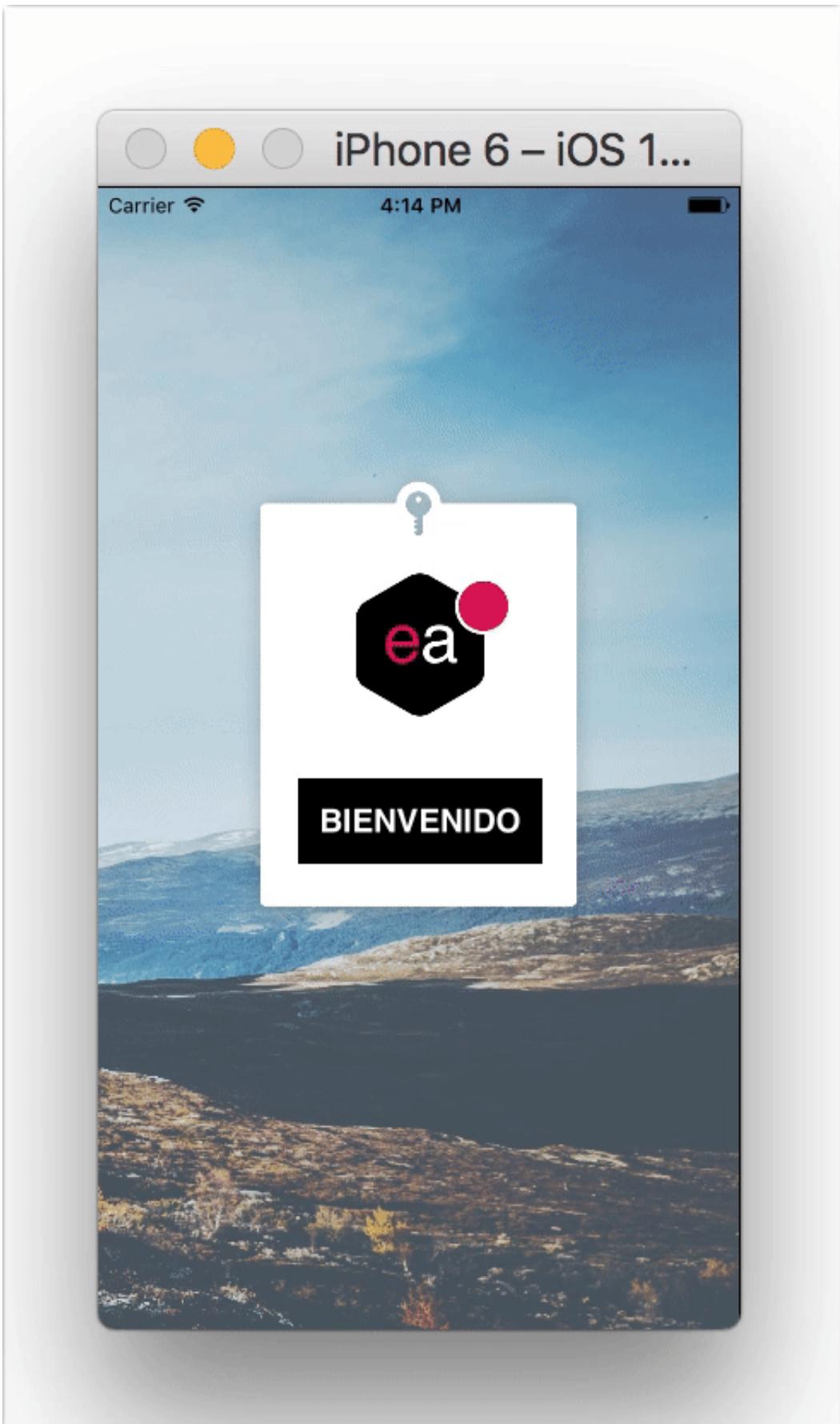
Para ello, como digo, vamos a apoyarnos en **Firebase**.

3. La Aplicación que vamos a crear

En lugar de una app completa, vamos a crear el **punto de entrada** a cualquier aplicación que gestione usuarios.

Vamos a crear una pantalla de login que enlazará con Firebase y nos permitirá gestionar el **registro** y **acceso** de usuarios.

Al terminar tendrás una **aplicación** como esta:



Y lo que es mejor, podrás utilizar todo lo que has visto en este tutorial para **crear pantallas de login** para tus propias aplicaciones.

Para poder centrarnos en lo realmente importante del tutorial: La

integración con **Firebase** y gestión de usuarios, he creado un proyecto inicial, donde encontrarás la interfaz ya creada.

Por favor, [descárgalo desde aquí](#), ya que durante el **tutorial** vamos a trabajar con este proyecto.

4. ¿Qué es Firebase?

Firebase es un **MBAaaS** (Mobile Backend as a Service) creado en 2011 por Andrew Lee y James Tamplin. En 2014 fue adquirido por Google, lo que hizo que se añadieran una gran cantidad de funcionalidades nuevas al servicio.

Se trata de una **herramienta** que nos ofrece un conjunto de características que conforman un **backend completo** sin necesidad de desarrollar el propio backend desde cero.

Entre estas características destacan:

- Base de Datos con actualizaciones en tiempo real
- Autenticación de usuarios
- Almacenamiento en la nube
- Notificaciones Push
- Integración con redes sociales
- Analíticas

Con Firebase puedes crear **aplicaciones completas** sin tener que escribir **ni una sola línea de código** del lado del **servidor**.

5. Creando nuestro Proyecto en Firebase

Lo primero que tenemos que hacer para poder integrar Firebase en nuestra aplicación es **crear nuestra cuenta** en dicho servicio.

Para ello, accede a la [página de inicio](#) y haz login.

Al tratarse de una empresa propiedad de la gran G, bastará que utilices una cuenta de **cualquier servicio de Google** para logarte. Ni siquiera

necesitas registrarte.

The screenshot shows the Firebase homepage. At the top, there's a blue header with the Firebase logo and navigation links for "Go to docs" and a user profile. Below the header, a large graphic of a smartphone is shown, with a small character standing next to it holding a long string attached to the phone. The text "Welcome to Firebase" is at the top left, followed by a description: "Tools from Google for developing great apps, engaging with your users, and earning more through mobile ads. [Learn more](#)". Below this is a blue button labeled "CREATE NEW PROJECT" and the text "or import a Google project".

Haz click en el botón **CREATE NEW PROJECT**

Imagineemos que la aplicación que queremos crear se trata de una app de **compra-venta** de productos relacionados con **videojuegos**, es decir, consolas, juegos, periféricos, etc, etc.

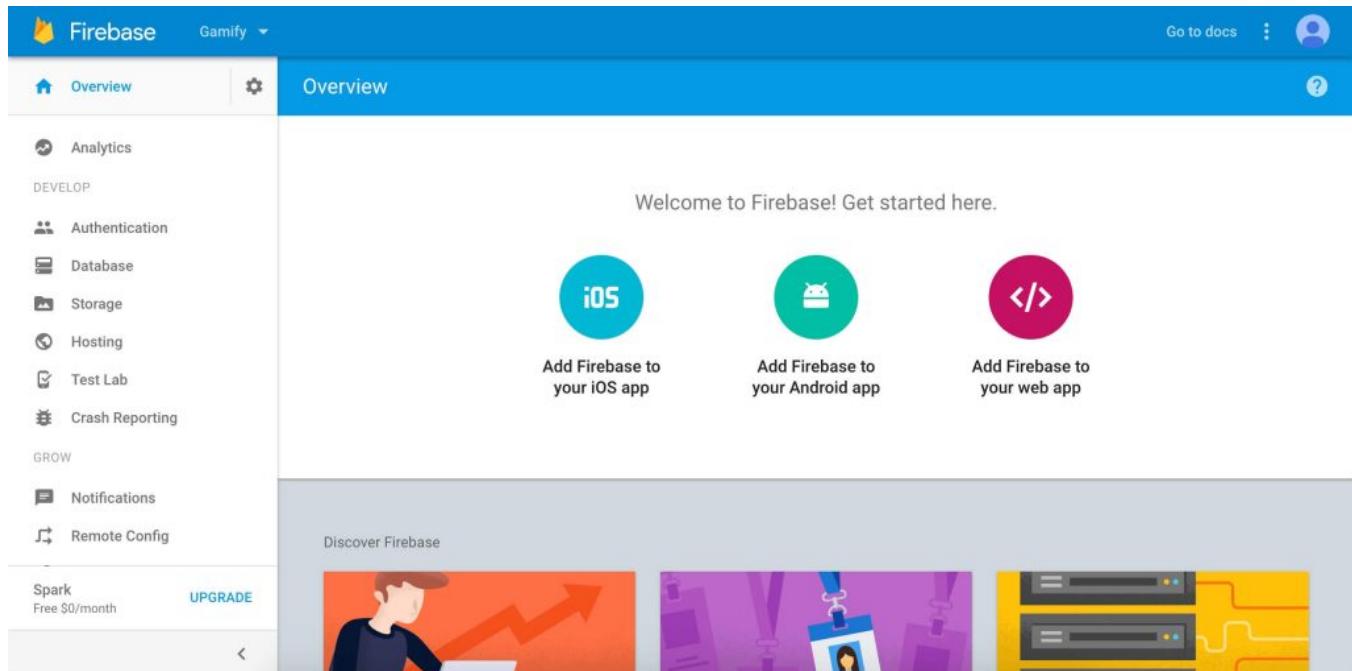
Enfocados en este tema, le daremos el nombre de **Gamify**.

Puedes darle el nombre que quieras. Tal vez quieras aprovechar este tutorial para comenzar tu propio proyecto personal

Además de especificar el nombre tendrás que especificar el **país** en el que te encuentras.

The screenshot shows the "Create a project" dialog box over the Firebase homepage. The dialog box has fields for "Project name" (containing "Gamify") and "Country/region" (set to "Spain"). It also contains a note about Firebase Analytics data sharing and a terms agreement checkbox. At the bottom are "CANCEL" and "CREATE PROJECT" buttons. The background of the dialog box is white, while the rest of the page is grey.

Haz click en el botón **CREATE PROJECT** y serás redirigido al **panel de control** del proyecto que acabas de crear en Firebase.



Panel de Control de Firebase

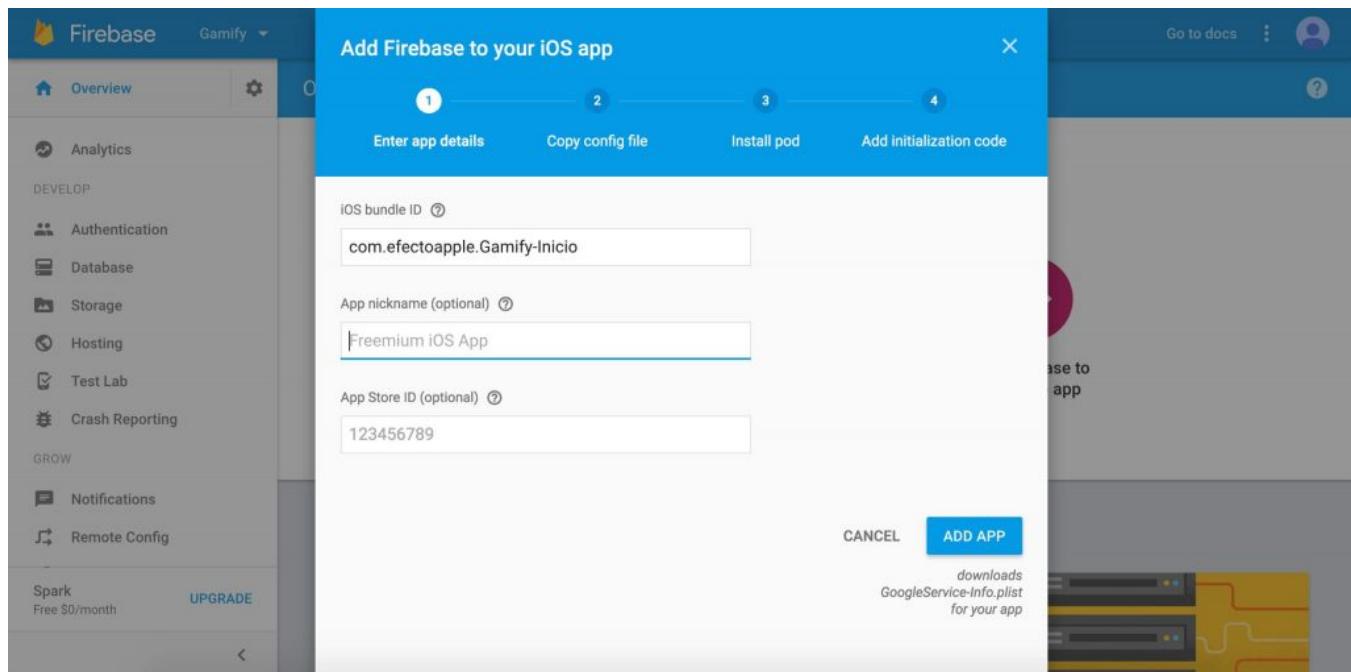
Desde aquí podrás gestionar **todos los servicios** que Firebase puede ofrecerte para tu aplicación.

Para comenzar haz click en el botón **Add Firebase to your iOS app**.

En el campo **iOS bundle ID** debes introducir un bundle ID diferente al que yo he utilizado, ya que este identificador debe ser único.

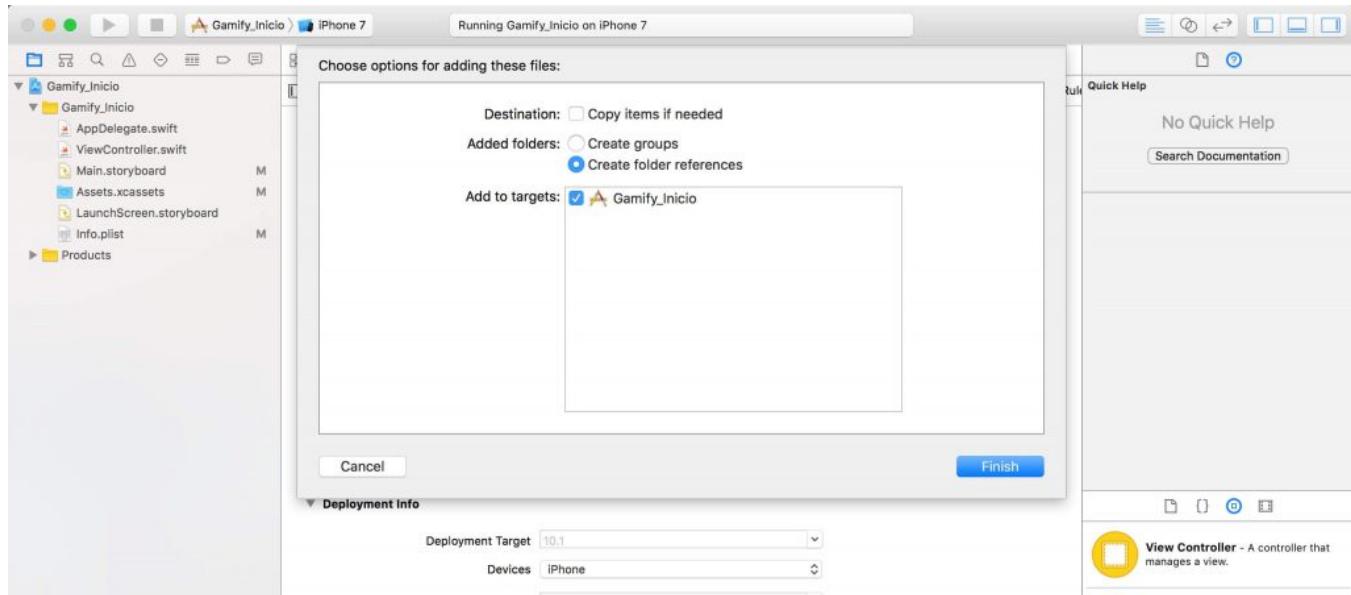
Además, el bundle ID que utilices aquí, deberá ser el mismo que uses en tu proyecto en Xcode. Por esto, recuerda bien, que bundle ID es el que introduces aquí.

Los otros dos campos: **App nickname** y **App Store** los puedes dejar vacíos.



Para continuar pulsa en el botón **ADD APP** y un fichero llamado **GoogleService-Info.plist** se descargará automáticamente.

Localiza ese fichero descargado y **añádelo** a la raíz de tu **proyecto** (arrastrándolo dentro de Xcode). Cuando al añadirlo, aparezca la siguiente ventana, pulsa en el botón **Finish**:



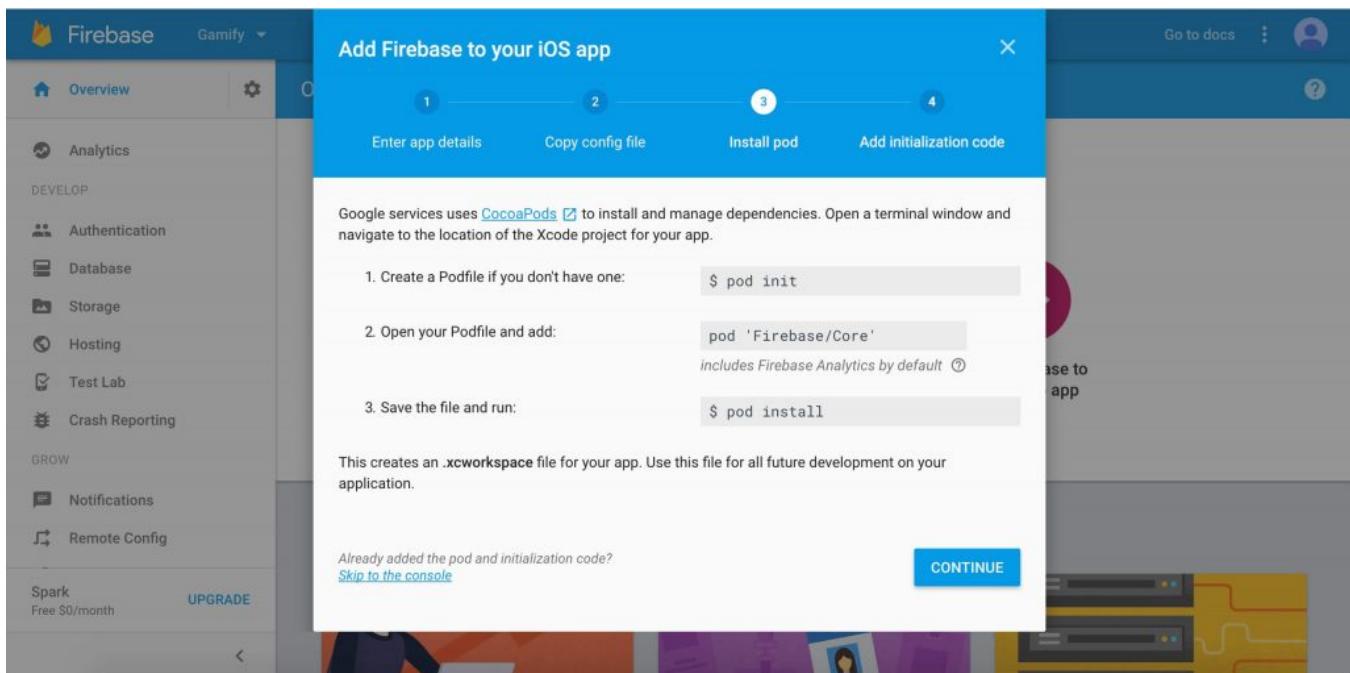
Ahora, vamos a aprovechar para modificar el **bundle ID** de nuestra aplicación. Como hemos dicho antes, debe ser **exactamente igual** que el que hemos configurado en la web de Firebase.

Para realizar este cambio, pulsa en el **nombre de tu proyecto** y en el menú **General**, en la propiedad **Bundle Identifier**, introduce la misma

cadena de texto que configuraste en Firebase.

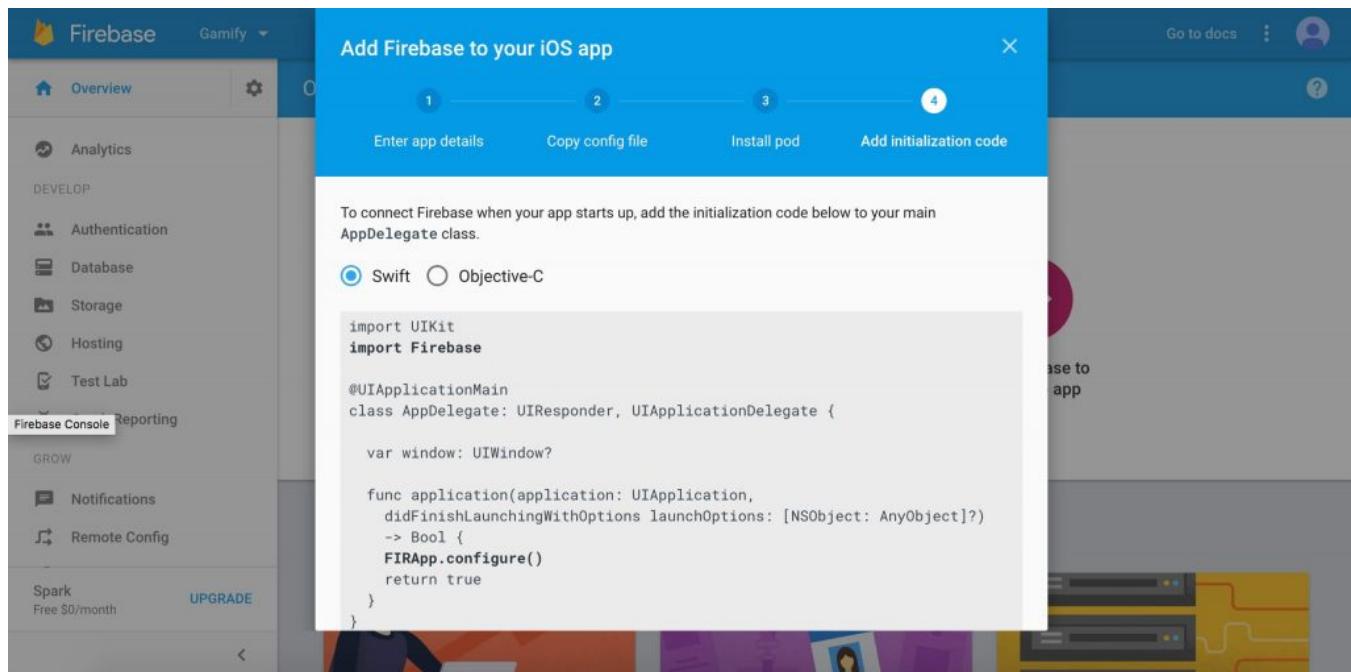


Una vez que has realizado estas modificaciones, continúa en la web de Firebase, pulsando en el botón **CONTINUE**. La siguiente página describe como **instalar el SDK de Firebase**.



Si estás siguiendo este tutorial a partir del **proyecto inicial** que te he dejado en el **apartado 3**, aquí no tienes que hacer nada, puesto que el proyecto de Xcode que te he preparado ya tiene instalado el SDK de Firebase, así que pulsa en **CONTINUE**.

La última página explica **como conectar con Firebase** cuando tu app arranque.



Haz click en el botón **FINISH** para cerrar el asistente de configuración. Verás los detalles de la aplicación que acabamos de crear.

Gamify mobile apps

iOS	com.efectoapple.Gamify-Inicio	⋮
Explore Firebase Analytics →		
0	\$0	In-app purchases
Monthly active users		
Crashes (30 days)		
0	0	Errors
Users impacted		

+ Add another app

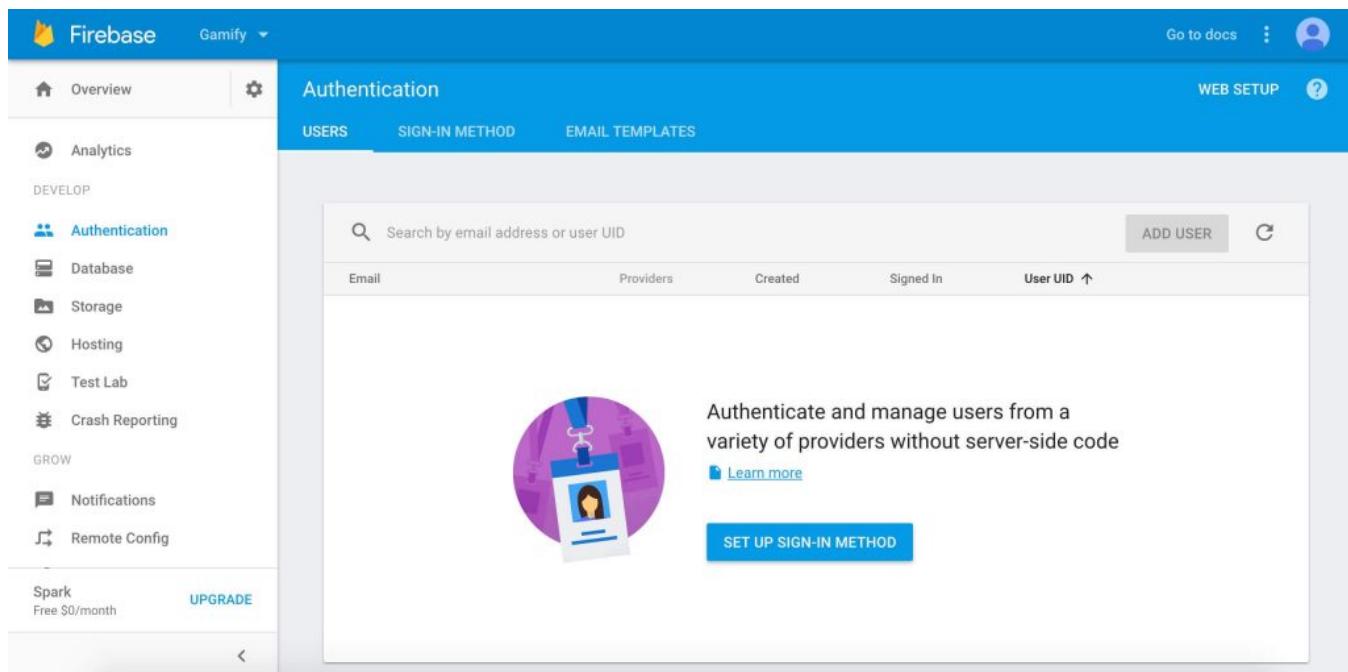
Enhorabuena, has creado correctamente **tu proyecto en Firebase**. Ahora veremos como gestionar los usuarios de nuestra aplicación.

6. Configurando la gestión de usuarios

Uno de los servicios que ofrece Firebase es el de **autenticación de usuarios**. Te permite dar de alta usuarios a través de diferentes proveedores. Puedes autenticar usuarios con Google, Twitter, Facebook y por supuesto utilizando el método tradicional de **Email y Password**. En

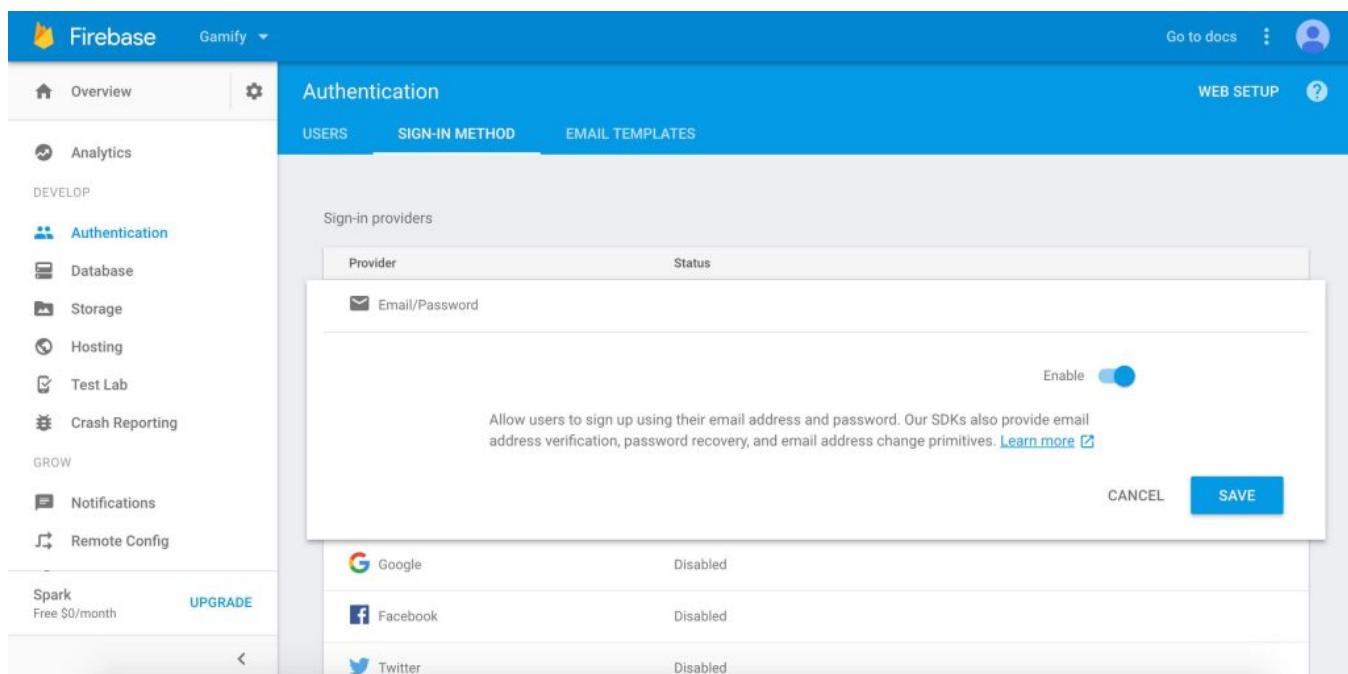
este tutorial utilizaremos esta última opción para que nuestros usuarios se den de alta.

Para habilitar la autenticación a través de **email/password**, accede a tu panel de control de Firebase y haz click en el menú **Authentication**.



The screenshot shows the Firebase console's Authentication section. The left sidebar includes Overview, Analytics, DEVELOP, Authentication (which is selected and highlighted in blue), Database, Storage, Hosting, Test Lab, Crash Reporting, GROW, Notifications, and Remote Config. A 'Spark Free \$0/month UPGRADE' button is also present. The main content area is titled 'Authentication' and has tabs for USERS, SIGN-IN METHOD (which is active), and EMAIL TEMPLATES. A search bar at the top says 'Search by email address or user UID'. Below it is a table with columns: Email, Providers, Created, Signed In, and User UID (with an upward arrow). A large central graphic features a purple circular badge with a person icon and the text 'Authenticate and manage users from a variety of providers without server-side code'. Below the badge is a blue button labeled 'SET UP SIGN-IN METHOD'.

Haz click en el botón **SET UP SIGN-IN METHOD**, después pulsa sobre el menú **Email/Password** y activa el interruptor **Enable**. Despues pulsa en **SAVE**.

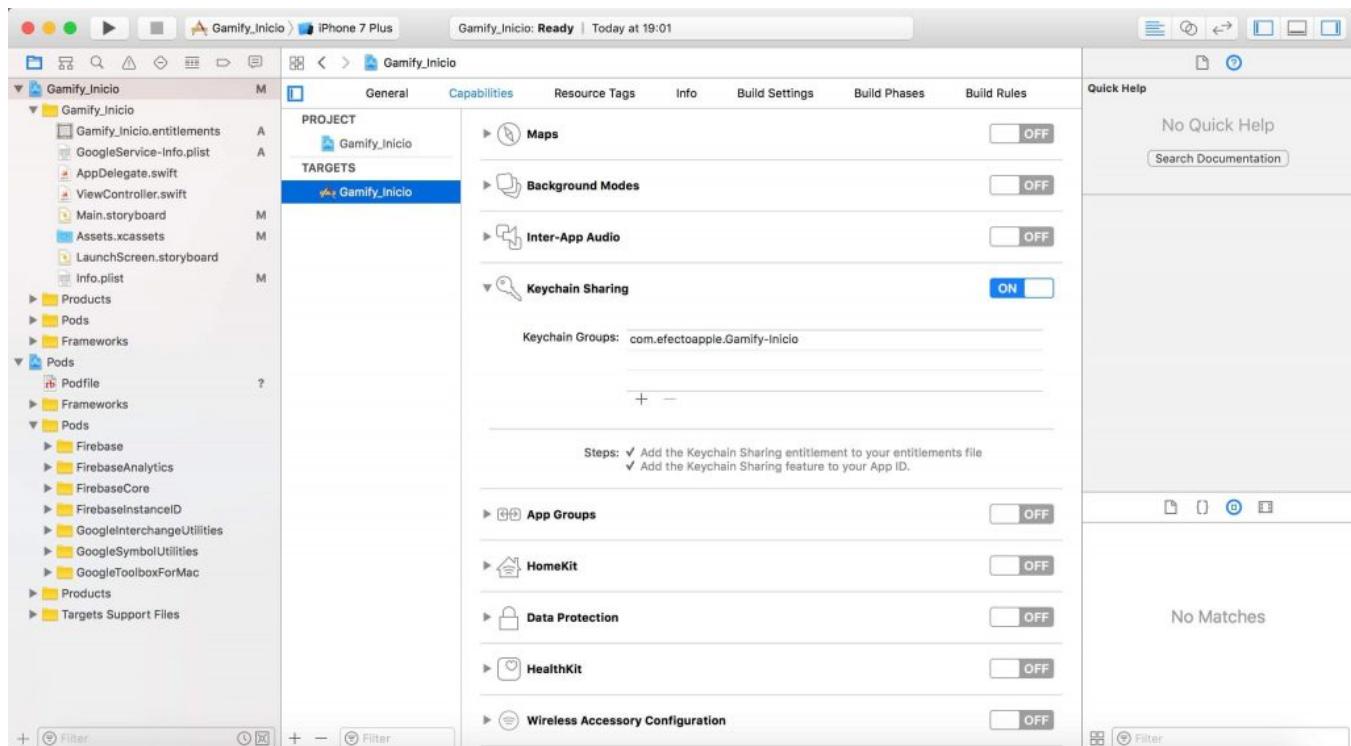


The screenshot shows the 'SIGN-IN METHOD' configuration for the Email/Password provider. The provider is listed in a table with 'Provider' and 'Status' columns. The status is set to 'Enable' with a blue toggle switch. Below the table, a note reads: 'Allow users to sign up using their email address and password. Our SDKs also provide email address verification, password recovery, and email address change primitives.' with a 'Learn more' link. At the bottom right are 'CANCEL' and 'SAVE' buttons.

Verás, que la autenticación a través de Email/Password ha quedado **activada**.

Firebase almacena las credenciales de los usuarios en el llavero, por lo que debemos activar la opción **Keychain Sharing** en nuestro proyecto de Xcode.

Abre el proyecto de inicio que has descargado en el apartado 3 y dentro del target, en el menú **Capabilities**, activa la opción **Keychain Sharing**.

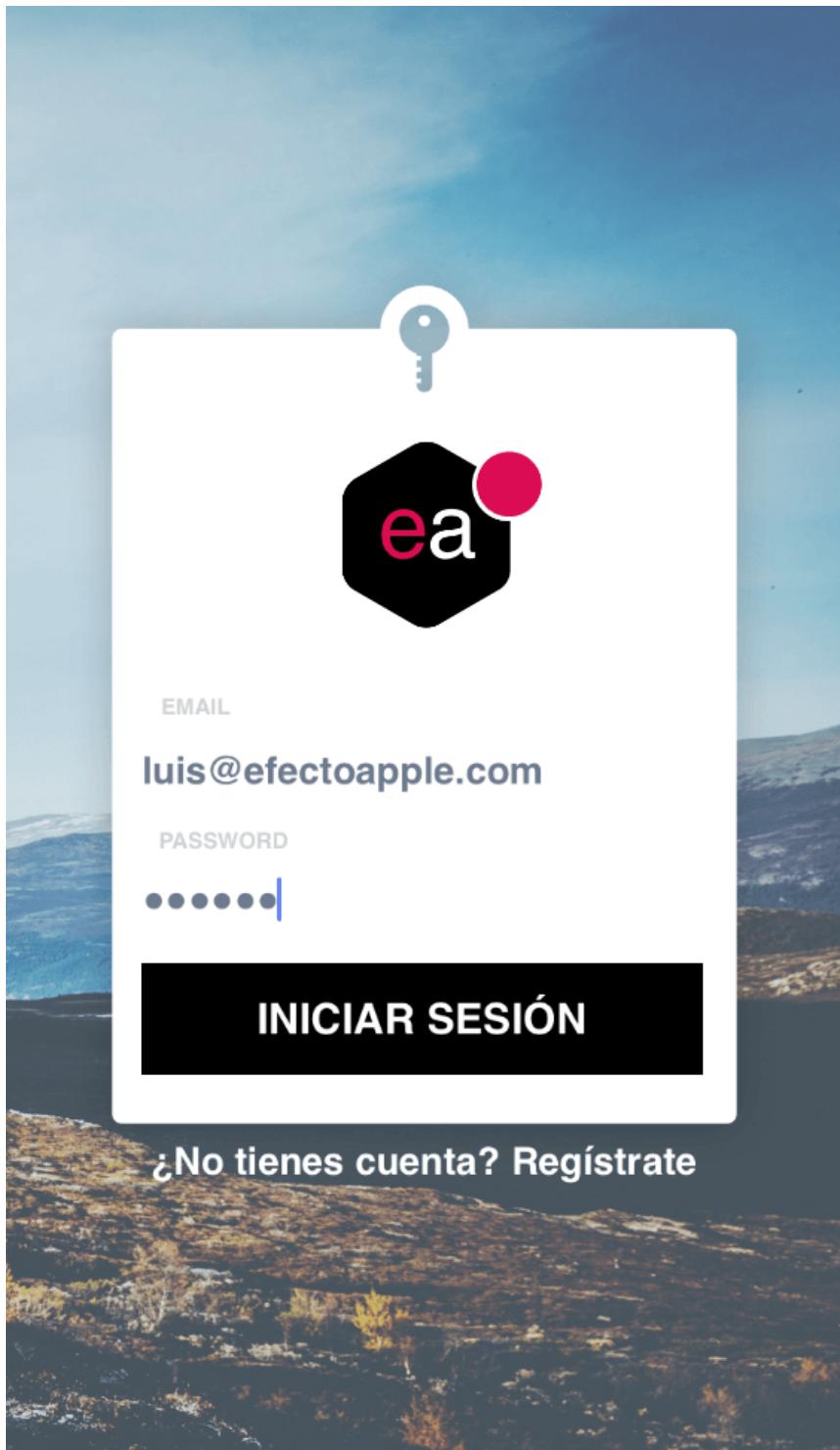


Ya estamos preparados para **autenticar nuestros usuarios** utilizando email y password.

7. Echando un vistazo a nuestra app

Vamos a revisar rápidamente el **aspecto del proyecto** que has descargado, para entender un poco el entorno donde vamos a trabajar.

Lo primero que deberías hacer es ejecutar la aplicación y comprobar que la **interfaz** se muestra correctamente.



Si pulsas alguno de los dos botones, verás que **no hacen nada**. Cuando hayas terminado este tutorial, a través de estos botones podrás controlar la **creación de usuarios nuevos** y el **inicio de sesión** de los mismos.

Echando un vistazo a nuestra interfaz

Abre el **Main.storyboard** para revisar la interfaz de la aplicación.

Como ves se trata de algo muy sencillo. Tenemos nuestra pantalla de registro/inicio de sesión que cuenta con dos textFields, donde el usuario

introducirá su email y contraseña **para iniciar sesión**. Sin embargo, antes de esto, el usuario debe contar con una cuenta de usuario. Para crear su cuenta de usuario tendrá que pulsar en el botón de la parte inferior **¿No tienes cuenta? Regístrate**.

Al pulsar en ese botón se lanza **un alert** que permite **crear una nueva cuenta** de usuario. Una vez que hayamos creado esa cuenta, **ya podremos acceder** con esas credenciales.

Al acceder con credenciales correctas, la aplicación nos permite acceder a la **pantalla de bienvenida**.

En caso de que intentemos acceder con credenciales incorrectas, la aplicación **lanzará mensajes de error** y nos impedirá visualizar la pantalla de bienvenida.

En **ViewController.swift** tenemos los 2 outlets **loginEmailTextField** y **loginPasswordField** que enlazan con nuestra interfaz. Además todo el código de la app lo desarrollaremos únicamente en **dos métodos**:

- **signUpTapped()**: Se ejecuta cuando el usuario pulsa en el botón de **¿No tienes cuenta? Regístrate**.
- **loginTapped()**: Se ejecuta cuando el usuario pulsa en el botón de **INICIAR SESIÓN**.

Este es el aspecto de nuestra clase **ViewController.swift** ahora mismo:

```
//  
//  ViewController.swift  
//  Gamify_Inicio  
//  
//  Created by Luis Rollon Gordo on 8/12/16.  
//  Copyright © 2016 EfectoApple. All rights reserved.  
  
import UIKit  
import FirebaseAuth  
  
class ViewController: UIViewController {  
  
    @IBOutlet weak var loginEmailField: UITextField!  
    @IBOutlet weak var loginPasswordField: UITextField!  
  
    override func viewDidLoad() {  
        super.viewDidLoad()  
  
    }  
  
    @IBAction func signUpTapped(_ sender: UIButton) {  
  
    }  
  
    @IBAction func loginTapped(_ sender: UIButton) {  
  
    }  
  
    override func didReceiveMemoryWarning() {  
        super.didReceiveMemoryWarning()  
  
    }  
}
```

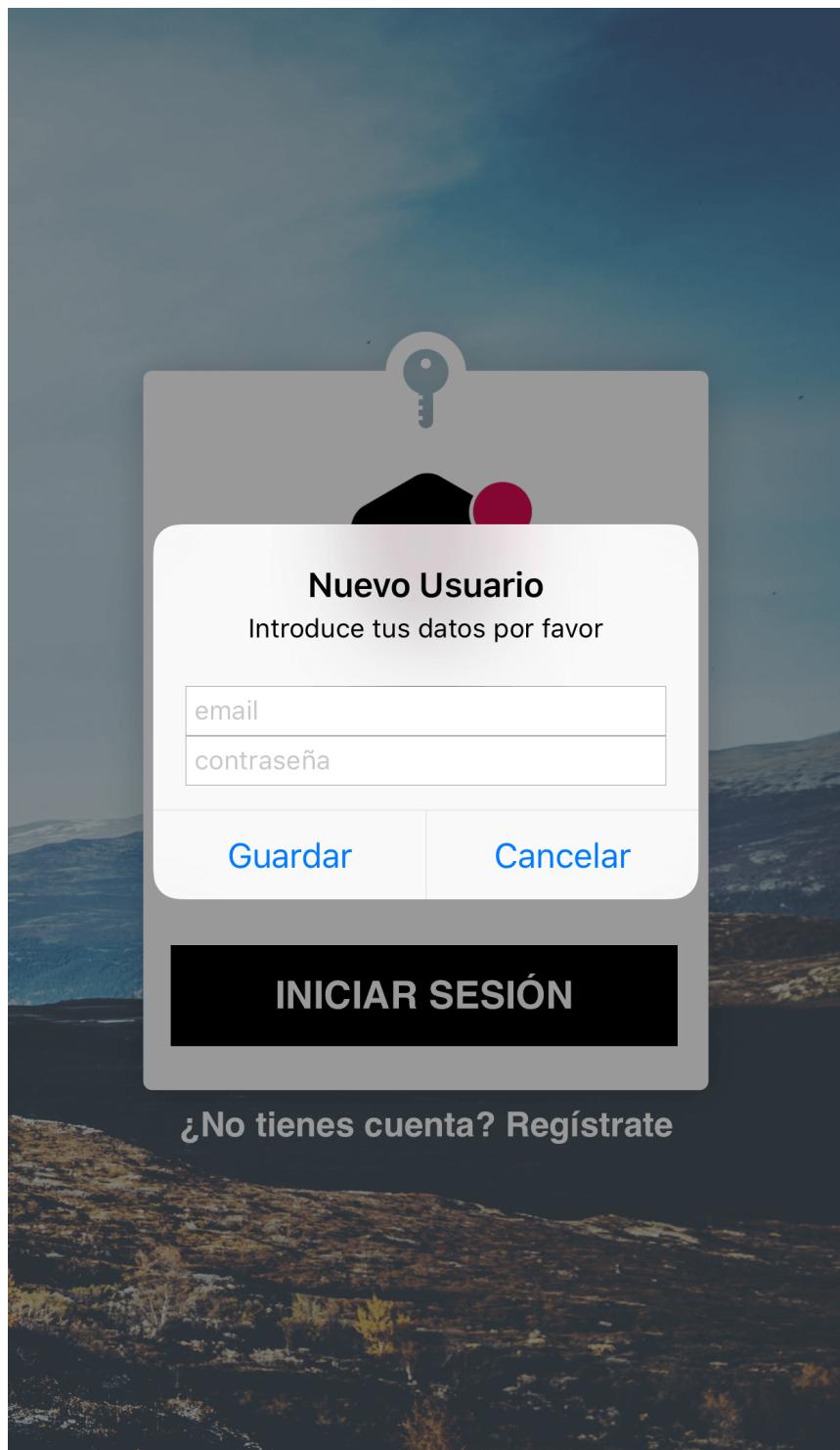
Como ves, los métodos están **vacíos**. Lo que haremos en este tutorial será **rellenarlos** viendo **paso a paso** lo que hace cada parte del código.

Comenzaremos por el **registro de usuarios** y en el siguiente apartado veremos el **inicio de sesión** de usuarios.

8. Registro de usuarios

Como hemos dicho antes, para el registro de usuarios, trabajaremos con el método **signUpTapped()**.

signUpTapped() es el método encargado de lanzar la alerta de registro de usuarios. Lo que queremos que haga es que muestre una alerta como esta:



El código que tendrás que añadir al método **signUpTapped()** es el siguiente:

```
@IBAction func signUpTapped(_ sender: UIButton) {  
    //1.  
  
    let alert = UIAlertController(title: "Nuevo Usuario",  
                                message: "Introduce tus datos por favor",  
                                preferredStyle: .alert)  
  
    //2.  
  
    let saveAction = UIAlertAction(title: "Guardar",  
                                 style: .default) { action in  
        let emailField = alert.textFields![0]  
        let passwordField = alert.textFields![1]  
  
        //3.  
  
        FIRAuth.auth()!.createUser(withEmail:  
            emailField.text!,  
            password: passwordField.text!) { user,  
            error in  
            //4.  
  
            let cancelAction = UIAlertAction(title: "Cancelar",  
                                         style: .default)  
  
            //5.  
  
            alert.addTextField { textEmail in  
                textEmail.placeholder = "email"  
            }  
  
            alert.addTextField { textPassword in  
                textPassword.isSecureTextEntry = true  
                textPassword.placeholder = "contraseña"  
            }  
        }  
    }  
}
```

```
    }

//6.

    alert.addAction(saveAction)

    alert.addAction(cancelAction)

//7.

    present(alert, animated: true, completion: nil)

}
```

Entendiendo el código

Y aquí tienes la **explicación** a cada una de las partes del **código**:

//1. Creamos un **UIAlertController** que será quien gestione la alerta que vamos a mostrar

//2. Creamos la acción de Guardar y en las constantes **emailField** y **passwordField** almacenamos lo que el usuario introduzca en los campos de la alerta.

//3. A la acción de **Guardar** le asociamos la llamada al método **createUser()** de Firebase, pasándole lo que el usuario haya introducido en los campos email y contraseña. De esta forma creamos un usuario con las credenciales que haya utilizado al registrarse

//4. Creamos la acción de **Cancelar**

//5. Añadimos los textFields **textEmail** y **textPassword**

//6. Añadimos la acción de **Guardar** y la acción de **Cancelar** a nuestro **UIAlertController**

//7. Mostramos el **AlertController** que acabamos de crear

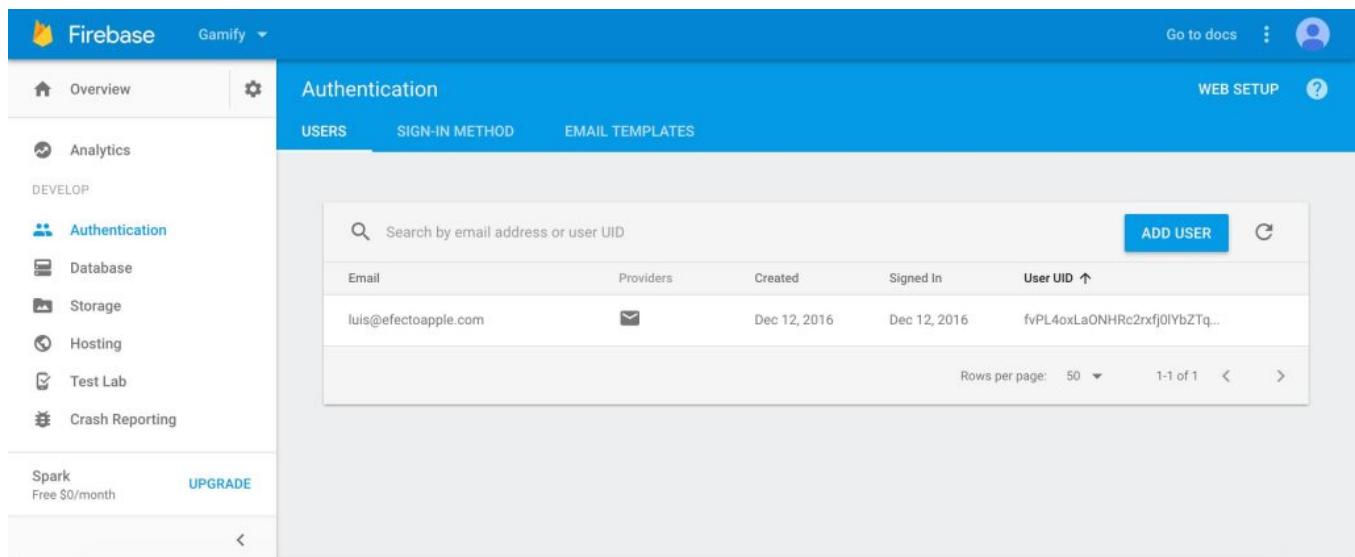
Únicamente con este método, gestionaremos el **alta de usuarios**. Como

ves, no puede ser mas sencillo.

A la hora de registrar nuevos usuarios introduce una contraseña que tenga como mínimo 6 caracteres

Una vez que has completado el método `signUpTapped()`, compila y ejecuta la aplicación. Pulsa en el botón **¿No tienes cuenta? Regístrate** y añade un **email** y una **contraseña** de como mínimo 6 caracteres, después, pulsa en **Guardar**.

Ahora vete a tu panel de control en **Firebase** y accede al menú **Authentication**, si actualizas la pantalla de **Users**, verás como aparecerá el usuario que acabas de crear desde tu aplicación. ¡Genial! Esto marcha.



The screenshot shows the Firebase console's Authentication section. On the left sidebar, 'Authentication' is selected under the 'Develop' category. The main interface shows the 'Authentication' tab selected, with 'USERS' currently active. A search bar at the top allows searching by email or user UID. Below it is a table with columns: Email, Providers, Created, Signed In, and User UID. One user is listed: luis@efectoapple.com, with a provider icon, created and signed in on Dec 12, 2016, and a user UID fvPL4oxLaONHRc2rxfj0lYbZTq.. At the bottom right of the table are buttons for 'Rows per page' (set to 50), '1-1 of 1', and navigation arrows.

Una vez que ya sabemos **crear usuarios** desde nuestra aplicación, veamos como controlamos el **inicio de sesión** de dichos usuarios.

9. Inicio de Sesión de Usuarios

Para controlar el inicio de sesión de usuarios deberemos trabajar con `loginTapped()`, ya que es el método que está asociado al botón **INICIAR SESIÓN**.

Lo que queremos que haga es que compruebe si el usuario ha introducido **correctamente sus credenciales** y en ese caso, le mostraremos la **pantalla de bienvenida**. En caso contrario mostraremos los mensajes de error correspondientes.

El código que tendrás que añadir al método **loginTapped()** es el siguiente:

```
@IBAction func loginTapped(_ sender: UIButton) {  
    //1.  
    if self.loginEmailField.text == "" || self.loginPasswordField.text == "" {  
        let alertController = UIAlertController(title: "Error", message: "Por favor introduce email y contraseña", preferredStyle: .alert)  
  
        let defaultAction = UIAlertAction(title: "OK", style: .cancel, handler:  
            nil)  
        alertController.addAction(defaultAction)  
  
        self.present(alertController, animated: true, completion: nil)  
  
    } else {  
        //2.  
        FIRAuth.auth()?.signIn(withEmail: self.loginEmailField.text!,  
            password: self.loginPasswordField.text!) { (user, error) in  
            //3.  
            if error == nil {  
                let vc =  
                    self.storyboard?.instantiateViewController(withIdentifier: "Home")  
                self.present(vc!, animated: true, completion: nil)  
  
            } else {  
                //4.  
                let alertController = UIAlertController(title: "Error", message:  
                    error?.localizedDescription, preferredStyle: .alert)
```

```
        let defaultAction = UIAlertAction(title: "OK", style: .cancel,  
handler: nil)  
  
        alertController.addAction(defaultAction)  
  
        self.present(alertController, animated: true, completion: nil)  
    }  
}  
}  
}
```

Entendiendo el código

Aquí tienes la **explicación** completa del **código**:

- //1. Comprobamos que el usuario haya llenado los campos de **email** y **contraseña**. En caso contrario mostramos un **Alert** que le informa que debe llenar ambos campos.
- //2. Si el usuario ha llenado ambos campos llamamos al método **signIn()** de **Firebase** que es quien se encarga de hacer el inicio de sesión con las credenciales que haya introducido
- //3. Si no se ha producido ningún error, mostramos la **pantalla de bienvenida** al usuario
- //4. Si se produce algún error, mostramos **una alerta** informando de los motivos del error

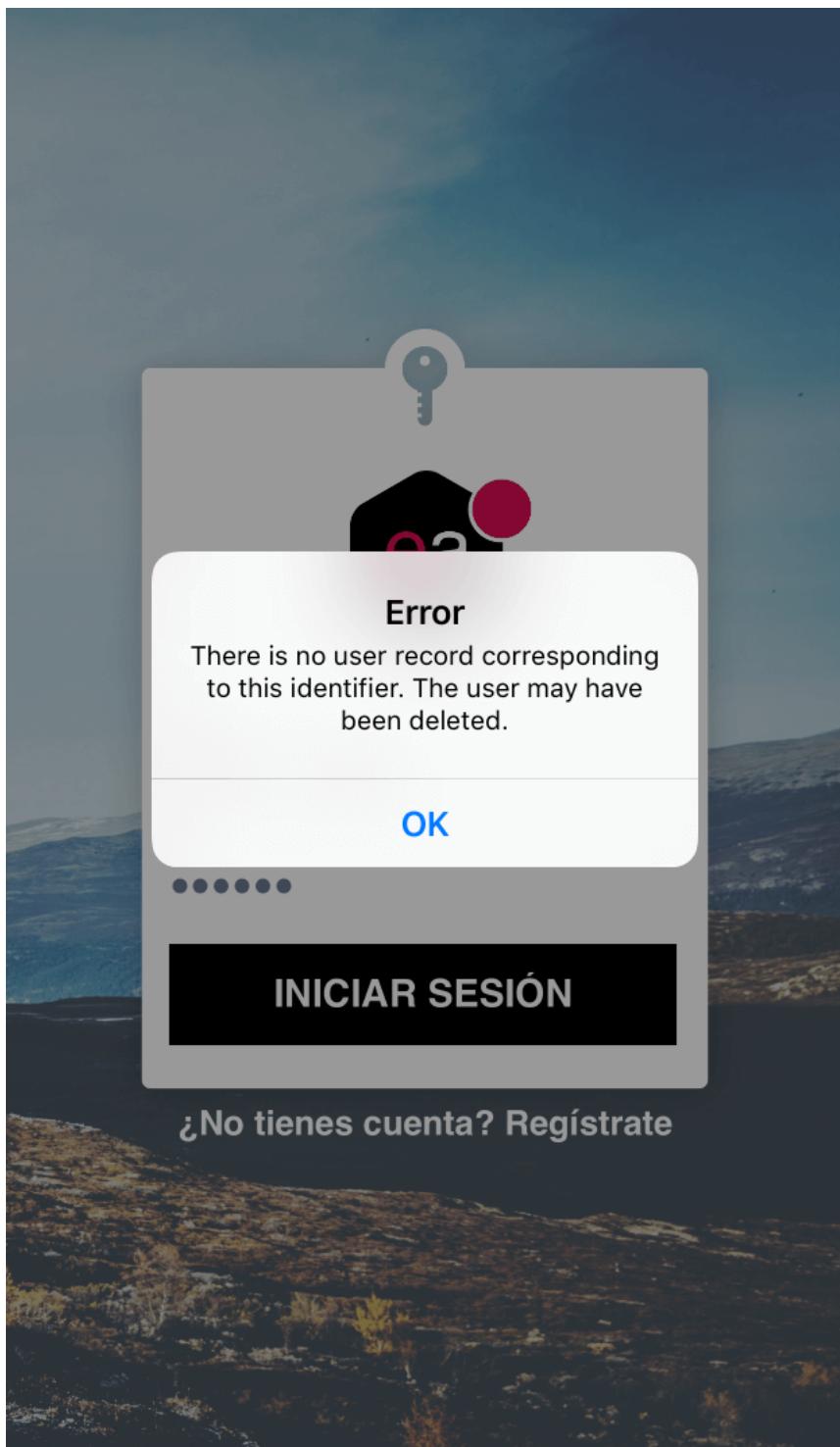
Como ves, de nuevo se trata de un **código muy sencillo** que te permitirá de forma simple gestionar los **inicios de sesión** de los usuarios de tu aplicación.

10. Probando nuestra aplicación

Después de implementar ambos métodos ha llegado el momento de **probar nuestra aplicación**. Ejecútala en el simulador y sigue estos pasos:

1. Introduce un **email** en el campo email y una **contraseña** en el campo password
2. Pulsa en el botón **INICIAR SESIÓN**

Verás que se muestra un mensaje de error que nos indica que no hay **ningún usuario** registrado con ese email.



Por tanto lo que haremos será **registrar un nuevo usuario**. Sigue estos pasos:

1. Pulsa en el botón **¿No tienes cuenta? Regístrate**
2. Introduce un **email** y un **password** y pulsa en el botón **Guardar**
3. Introduce en la pantalla de inicio de sesión el **email** y el **password** de la cuenta de usuario que acabas de crear y pulsa en **INICIAR SESIÓN**

Verás que se muestra correctamente la **pantalla de bienvenida**.



¡Has creado una pantalla de **Inicio de Sesión/Registro de Usuarios** que puedes utilizar en tus propias aplicaciones. ¡Enhорabuena!

17. Resumen Final y Conclusiones

Si has seguido el tutorial paso a paso habrás podido ver lo fácil que es crear una pantalla de **Login/Sign up** para gestionar los usuarios que acceden a tu aplicación.

Simplemente hay que configurar el proyecto en tu cuenta de **Firebase** y después implementar dos métodos: Uno para el **registro** de nuevos usuarios y otro para el **acceso** de usuarios.

No puede ser mas sencillo.

Espero que este tutorial te sirva para implementar este tipo de pantallas **en tus propias aplicaciones**.

Introducción a UICollectionView con Swift [Parte 1]

COMO UTILIZAR COLLECTION VIEWS EN TUS APLICACIONES

Lenguaje Swift | Nivel Principiante

1. Introducción

Este Tutorial es sobre **UICollectionView** y tiene algo de historia (breve, lo prometo).

Hace un par de semanas estuve tomando algo con uno de mis mejores amigos y surgió la conversación de **cual era la mejor serie de todos los tiempos.**

Al poco tiempo estábamos discutiendo y gritando como locos nombres de series, algunas de hace más de 20 años.

Después de un buen rato haciendo memoria y por raro que parezca, **nos pusimos de acuerdo.**

Los dos teníamos claro cual era para nosotros la mejor serie de la historia: **LOST.**

Este Tutorial está inspirado en el **mundo de las series** y dedicado a mi amigo Guiller.

2. ¿Qué vamos a ver en este tutorial sobre UICollectionView?

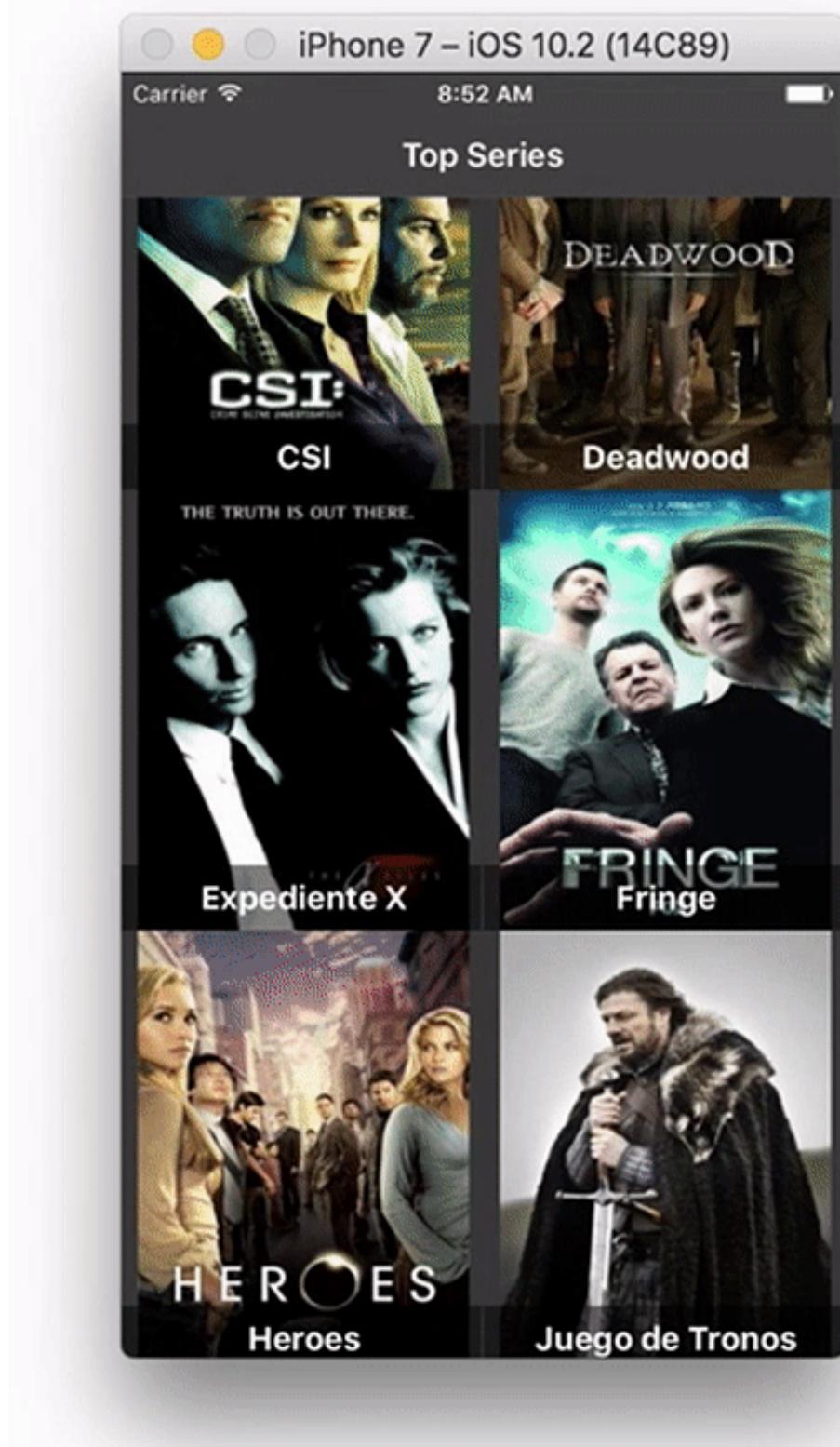
Desarrollaremos una aplicación que conste de una **UICollectionView** donde mostraremos imágenes de algunas de las **series más conocidas** de los últimos tiempos. Además, si el usuario **pulsa** en cualquiera de las imágenes, accederá al detalle de esa serie donde mostraremos el **título** de la serie y la **imagen** en mayor tamaño.

Estos son los temas que vamos a ver en el **Tutorial**:

- ¿En qué consiste un **Protocolo** en iOS?
- ¿Qué es el protocolo **UICollectionViewDataSource**?
- ¿Qué es el protocolo **UICollectionViewDelegate**?
- ¿En qué consisten los **delegados** en iOS?
- ¿Que es el **IndexPath**?
- ¿Cómo funcionan las **UICollectionViews**?
- ¿Cómo realizar **navegación entre pantallas** utilizando un **Navigation Controller**?
- ¿Cual es la forma de **pasar información** entre diferentes **View Controllers**?
- ¿Cómo puedes **personalizar** la **Navigation Bar** de tu aplicación?

3. La Aplicación que vamos a crear

Al terminar el tutorial, nuestra aplicación tendrá este aspecto:



Como ves se trata de una aplicación **muy sencilla**, que nos servirá para comprender perfectamente el funcionamiento de **UICollectionView** y otros **conceptos** básicos del **Desarrollo iOS**.

Nota Aclaratoria: Anatomía de Grey fue incluida en el listado de series contra mi voluntad. Recibí amenazas por parte de mi novia para que apareciera en la aplicación

4. Antes de comenzar

El funcionamiento detrás de las **UICollectionView** es muy muy parecido al de las **UITableView**. Si no has trabajado antes con Tablas en iOS, te recomiendo que comiences por el Tutorial de Introducción a UITableView y una vez que hayas revisado los **conceptos fundamentales**, continúes con este Tutorial.

De esta forma, con solo dos tutoriales, serás capaz de dominar UITableView y UICollectionView y podrás utilizarlas en tus aplicaciones iOS.

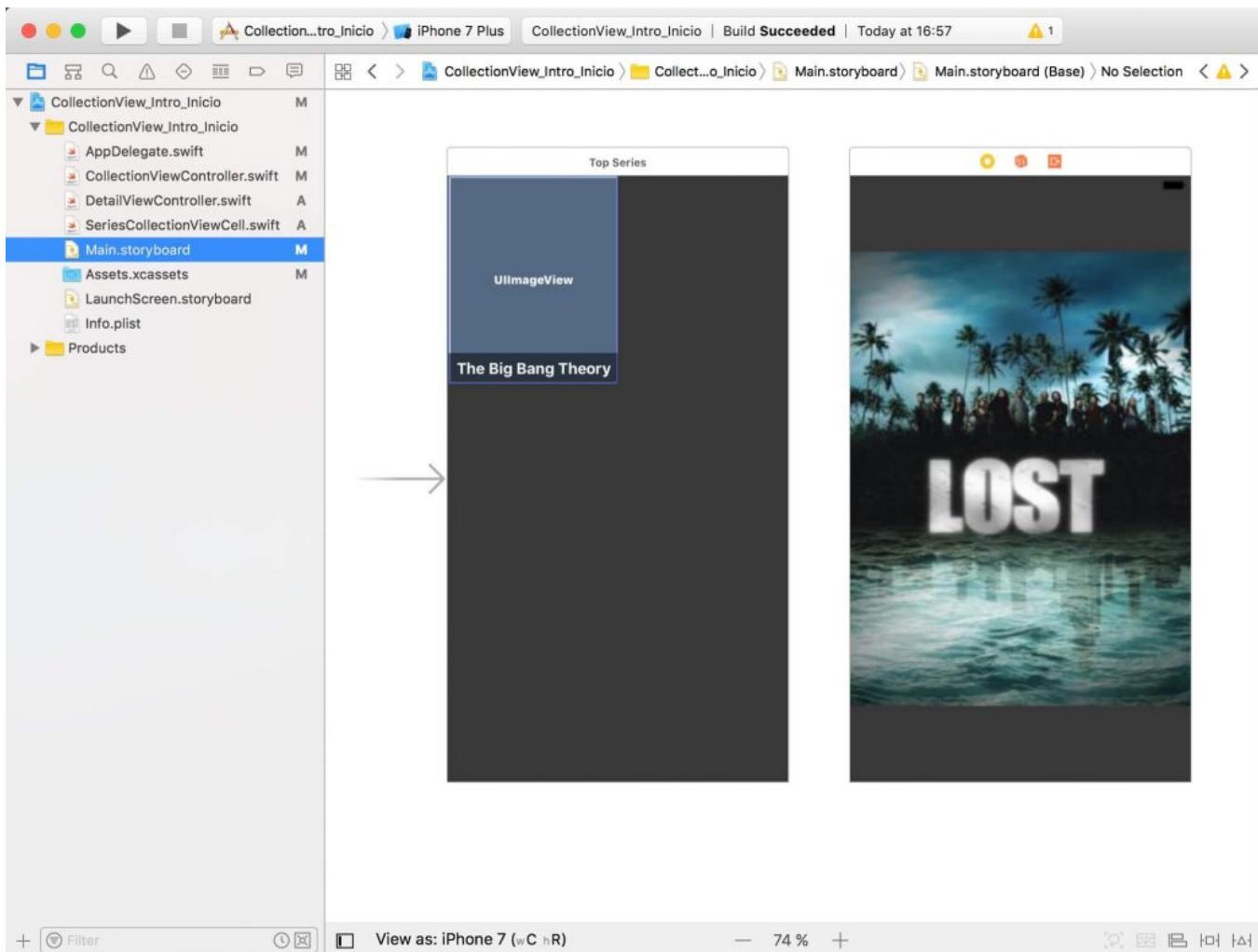
5. Descarga el Proyecto de Inicio

Para no alargar demasiado el tutorial, he creado un **Proyecto de Inicio** que puedes utilizar como base para crear esta aplicación.

Descarga el proyecto [desde aquí](#), descomprímelo y ábrelo con Xcode.

ECHANDO UN VISTAZO A LA INTERFAZ DEL PROYECTO

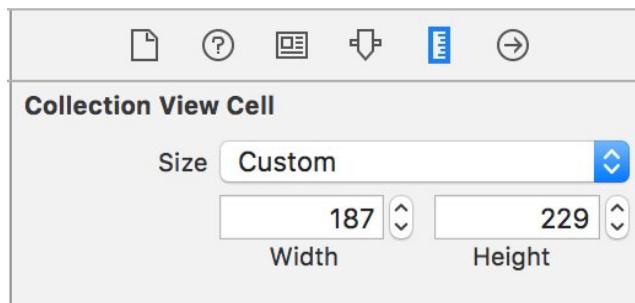
Lo primero que vamos a hacer es echar un vistazo a la interfaz, así que abre el fichero **Main.storyboard**.



Verás que tienes una interfaz compuesta por **dos pantallas**. La primera pantalla está asociada al **CollectionViewController.swift** y la segunda pantalla al **DetailViewController.swift**

Para crear la pantalla de **CollectionViewController.swift** hemos seguido estos pasos:

- Añadir una UICollectionView con el siguiente color de fondo: 383838.
- Establecer las 4 constraints de la UICollectionView a 0.
- Darle unas medidas de 187 x 229 a los items de la UICollectionView.



- Arrastrar dentro del item del Collection View un UIImageView y un

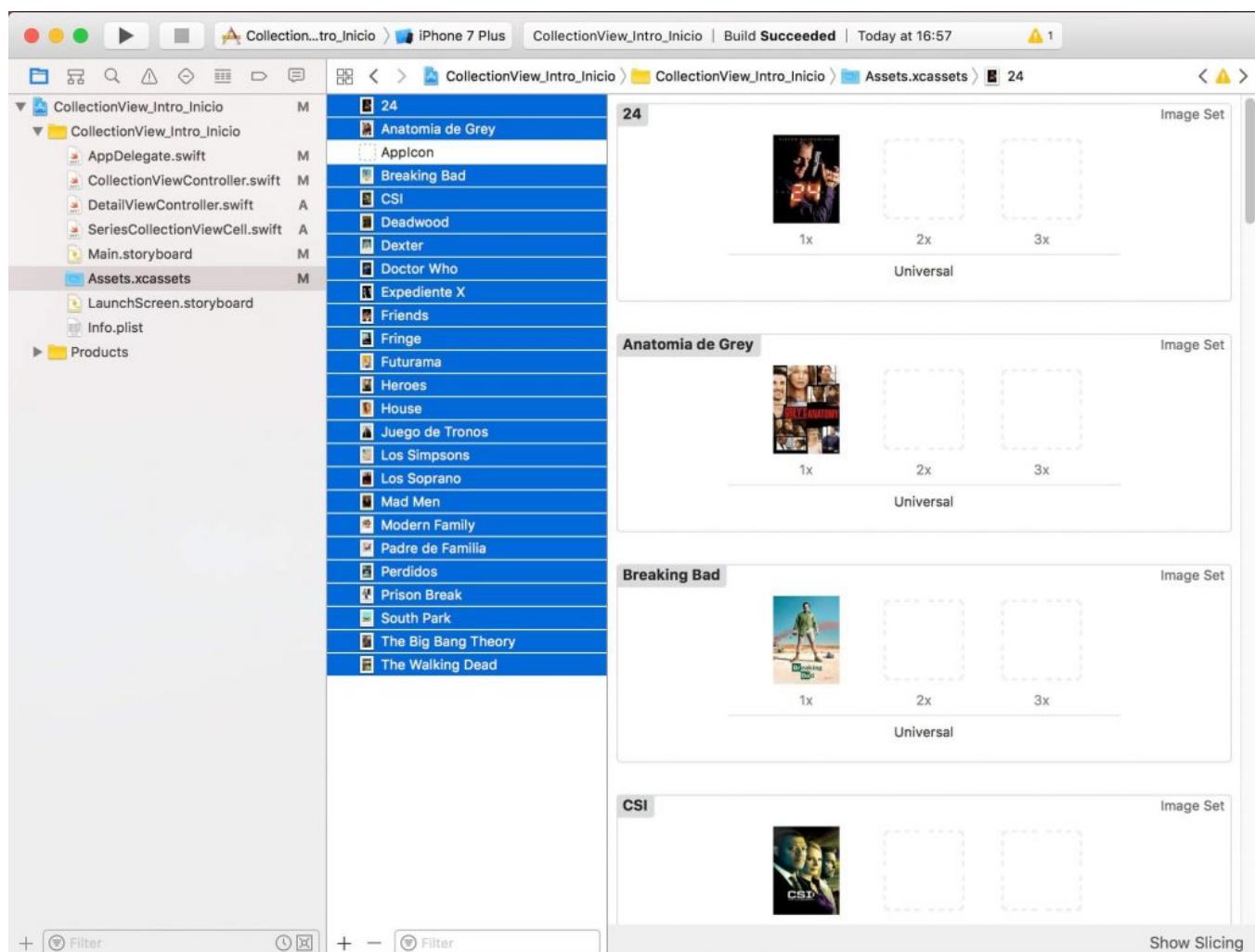
UILabel.

Estos dos elementos nos servirán para mostrar la **portada** de la serie y su **título**.

En la pantalla de **DetailViewController.swift** lo único que hemos hecho ha sido añadir una **UIImageView** que mostrará la portada de la serie a tamaño completo. Recuerda que tienes que añadir también las **constraints** para la UIImageView.

Si no tienes experiencia trabajando con Constraints, puedes visitar nuestra sección de Tutoriales y en el Tema Auto Layout encontrarás todo lo que necesitas

Abre ahora, la carpeta de **Assets** y verás como el proyecto ya tiene añadidas todas las imágenes que vamos a utilizar.



ECHANDO UN VISTAZO AL CÓDIGO DE NUESTRO PROYECTO

Después de este repaso de la **interfaz** de nuestro Proyecto de Inicio, vamos a echar un vistazo a los ViewControllers.

Si abres **CollectionViewController.swift**, verás que está vacío a excepción de los métodos que crea Xcode por defecto.

Abre **DetailViewController.swift** y como puedes comprobar, simplemente tiene dos variables que hacen referencia al **título** y a la **portada** de la serie.

```
import UIKit

class DetailViewController: UIViewController {

    @IBOutlet weak var detailImage: UIImageView!

    var detailName: String?

    override func viewDidLoad() {
        super.viewDidLoad()
        title = detailName
        detailImage.image = UIImage.init(named: detailName!)
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
    }
}
```

Abre **SeriesCollectionViewCell.swift** y verás que lo único que tiene son los **Outlets** de la **imagen** y la **etiqueta** que nos servirá para mostrar la información de las series en nuestra Collection View.

```
import UIKit

class SeriesCollectionViewCell: UICollectionViewCell {

    @IBOutlet weak var itemImage: UIImageView!

    @IBOutlet weak var itemLabel: UILabel!

}
```

Esto es todo lo que tenemos ahora mismo en nuestro **Proyecto**.

Si ejecutas la aplicación verás como lo único que hace nuestra aplicación es mostrar una pantalla oscura.

Cuando termines este tutorial, nuestra aplicación estará completa.

Antes de comenzar a programar, veamos un **repaso rápido** a unos conceptos básicos que debes conocer para trabajar con UICollectionView.

6. Conceptos fundamentales de UICollectionView

PROTOCOLO UICOLLECTIONVIEWDATASOURCE

Apple nos ofrece la clase **UICollectionView** para que podamos trabajar con Collection Views en nuestras aplicaciones. Esta clase nos permite mostrar en pantalla información de forma muy visual.

La pregunta es:

¿Cómo le decimos a la clase UICollectionView que información debe mostrar?

La respuesta es, utilizando el protocolo **UICollectionViewDataSource**.

Si no sabes **lo que es un protocolo**, no te preocupes, puedes consultarla [aquí](#).

Además de entender los protocolos, es importante que tengas en cuenta esto:

Una UICollectionView se divide en Secciones e Items

Una vez que has revisado el concepto de **protocolo** y entiendes que una Collection View se divide en diferentes **secciones**, donde cada sección puede contener un determinado número de items, ya podemos explicar de forma específica en que consiste el protocolo **UICollectionViewDataSource**.

UICollectionViewDataSource **es el enlace** entre los datos que queremos mostrar y nuestra UICollectionView. Este protocolo declara **dos métodos** obligatorios:

- collectionView:numberOfItemsInSection()
- collectionView:cellForItemAt()

El primero de los métodos: **numberOfItemsInSection()**, especifica el número de items que mostraremos en nuestra Collection View y se ejecutará tantas veces como secciones tenga nuestra Collection View. En nuestro caso, como no hemos indicado el número de secciones, iOS entenderá que nuestra Collection View tiene **una única sección**, por lo que este método se ejecutará una única vez.

El segundo método: **cellForItemAt()**, nos permite especificar **los datos que mostraremos en cada ítem** de la UICollectionView y se ejecutará tantas veces como items tenga la única sección de nuestra Collection View.

Al tratarse de **métodos obligatorios**, tendremos que implementarlos en nuestra clase si queremos utilizar el protocolo **UICollectionViewDataSource**.

PROTOCOLO UICOLLECTIONVIEWDELEGATE

Por otro lado tenemos el protocolo **UICollectionViewDelegate**. Este protocolo es el encargado de **determinar la apariencia** de nuestra

`UICollectionView`. Todos sus métodos son opcionales, por lo que no estamos obligados a implementar ninguno. Sin embargo, **ofrece funcionalidades muy útiles** como especificar el height de los items, configurar tanto el header como el footer de una `UICollectionView`, configurar lo que pasa cuando el usuario pulsa en un item, etc.

INDEXPATH

Si has seguido otros tutoriales de EfectoApple, tal vez hayas visto que nos hemos referido a este concepto como **NSIndexPath**.

Desde la versión 3 de Swift, Apple ha eliminado bastantes prefijos NS de muchas clases y la clase `NSIndexPath` ha sido una de ellas.

Por tanto, en lugar de trabajar con `NSIndexPath`, desde esta versión de Swift, trabajarás con **IndexPath**.

Después de este inciso, vamos a explicar el **concepto**.

Si recuerdas, lo que hemos comentado en el apartado anterior. Las Collection Views se dividen en diferentes **secciones**, donde a su vez cada sección puede contener un determinado número de **items**.

La clase **IndexPath** es fundamental para que entiendas completamente el trabajo con Collection Views en aplicaciones iOS. Siempre que quieras utilizar alguna Collection View en tu aplicación, tendrás que implementar el método `cellForItemAt()` y gran parte del funcionamiento de este método se basa en un objeto de la clase **IndexPath**.

Este objeto tiene dos propiedades:

- `.section`
- `.row`

A través de estas dos propiedades, **podremos situarnos en un punto concreto de una Collection View**. Mediante la propiedad `.section` especificaremos en qué **sección** de nuestra Collection View nos encontramos, mientras que a través de la propiedad `.row` determinaremos,

dentro de esa sección, en que **item** nos encontramos.

Por tanto si especificáramos lo siguiente:

- `.section = 0`
- `.row = 4`

Nos encontraríamos en el quinto item de la primera sección de nuestra Collection View. Fácil, ¿no?

7. Añadiendo datos a nuestra Collection View

Ahora que ya conoces el concepto de **IndexPath** y ya entiendes para que sirven los protocolos **UICollectionViewDataSource** y **UICollectionViewDelegate**, podemos utilizarlos para añadir datos a nuestra aplicación.

AJUSTANDO NUESTRA CLASE A LOS PROTOCOLOS

Lo primero que tenemos que hacer es especificar que la clase **CollectionViewController.swift** se ajustará a los protocolos **UICollectionViewDataSource** y **UICollectionViewDelegate**. Este es el primer paso siempre que trabajes con protocolos, determinar **que clase** será la que implemente el protocolo.

Para hacer esto únicamente tendrás que añadir **el nombre de los protocolos** a continuación de la declaración de nuestra clase, es decir, tu clase **CollectionViewController.swift** deberá tener este aspecto:

```
class CollectionViewController: UIViewController,  
UICollectionViewDataSource, UICollectionViewDelegate{  
  
    override func viewDidLoad() {  
        super.viewDidLoad()  
    }  
  
    override func didReceiveMemoryWarning() {
```

```
    super.didReceiveMemoryWarning()
}
}
```

Después de añadir estos dos protocolos, Xcode te mostrará por pantalla **el siguiente error**:

```
8
9 import UIKit
10
! 11 class CollectionViewController: UIViewController, UICollectionViewDelegate, UICollectionViewDataSource {
12                                         // Type 'CollectionViewController' does not conform to protocol 'UICollectionViewDataSource'
○ 13     @IBOutlet weak var collectionView: UICollectionView!
14
15     override func viewDidLoad() {
16         super.viewDidLoad()
17
18     }
19
```

Este error es completamente normal. Xcode te está avisando que **no has implementado** los dos métodos obligatorios, que debes utilizar, si quieres que tu clase se ajuste al protocolo UICollectionViewDataSource. Por ahora, **no prestes atención a este error**, más adelante, al añadir los dos métodos, haremos que desaparezca.

El siguiente paso será declarar la variable donde **almacenaremos la información** que queremos que muestre nuestra tabla.

CREANDO NUESTRO MODELO DE DATOS

Como los datos que mostraremos en nuestra **UICollectionView** será un listado de series, utilizaremos un array llamado **tvSeries** para almacenar ese listado. Por tanto, tendrás que añadir la siguiente constante justo antes del método **viewDidLoad()**:

```
let tvSeries = ["Perdidos", "Friends", "Breaking Bad", "Dexter", "Futurama",
"Los Soprano", "Mad Men", "House", "Anatomia de Grey", "24", "CSI",
"Deadwood", "Expediente X", "Fringe", "Heroes", "Juego de Tronos", "Los
Simpsons", "Doctor Who", "Modern Family", "Padre de Familia", "Prison
Break", "South Park", "The Big Bang Theory", "The Walking Dead"]
```

Perfecto. Ahora que ya tenemos nuestro array con todos los nombres de las series que queremos mostrar por pantalla, es hora de hacer que nuestra Collection View **muestre estos datos**.

Esto lo haremos en el primer apartado de la **segunda parte** de nuestro Tutorial sobre UICollectionView.

8. Resumen final

Espero que la primera parte de este tutorial te haya servido para entender como funcionan las **UICollectionViews** en iOS. Se trata de un elemento fundamental, por lo que es interesante que seas capaz de manejarlas perfectamente.

Aquí tienes un **pequeño resumen** de los puntos más importantes que hemos visto:

1. Protocolos en iOS
2. Los Protocolos **UICollectionViewDataSource** y **UICollectionViewDelegate**
3. **Delegados** en iOS
4. **IndexPath**
5. Cómo añadir datos a una **UICollectionView**

9. ¿Qué veremos en la segunda parte del Tutorial?

En la segunda parte del Tutorial de Introducción a UICollectionView continuaremos con el desarrollo de nuestra aplicación e iremos viendo estos apartados:

1. **Funcionamiento** completo de una **UICollectionView**
2. Realizar **navegación** en nuestra app utilizando un **Navigation Controller**
3. **Paso de información** entre diferentes **View Controllers**
4. **Personalización** de la **Navigation Bar**

Introducción a UICollectionView con Swift [Parte 2]

COMO UTILIZAR COLLECTION VIEWS EN TUS APLICACIONES

Lenguaje Swift | Nivel Principiante

1. Introducción

Veamos entonces la **segunda parte** del **Tutorial de Introducción** sobre **Collection View en Swift**.

Si no has revisado la primera parte, te recomiendo que lo hagas. En la primera parte hablamos de **conceptos fundamentales** que te ayudarán a comprender el funcionamiento de las **Collection Views**.

2. ¿Qué vamos a ver en este tutorial

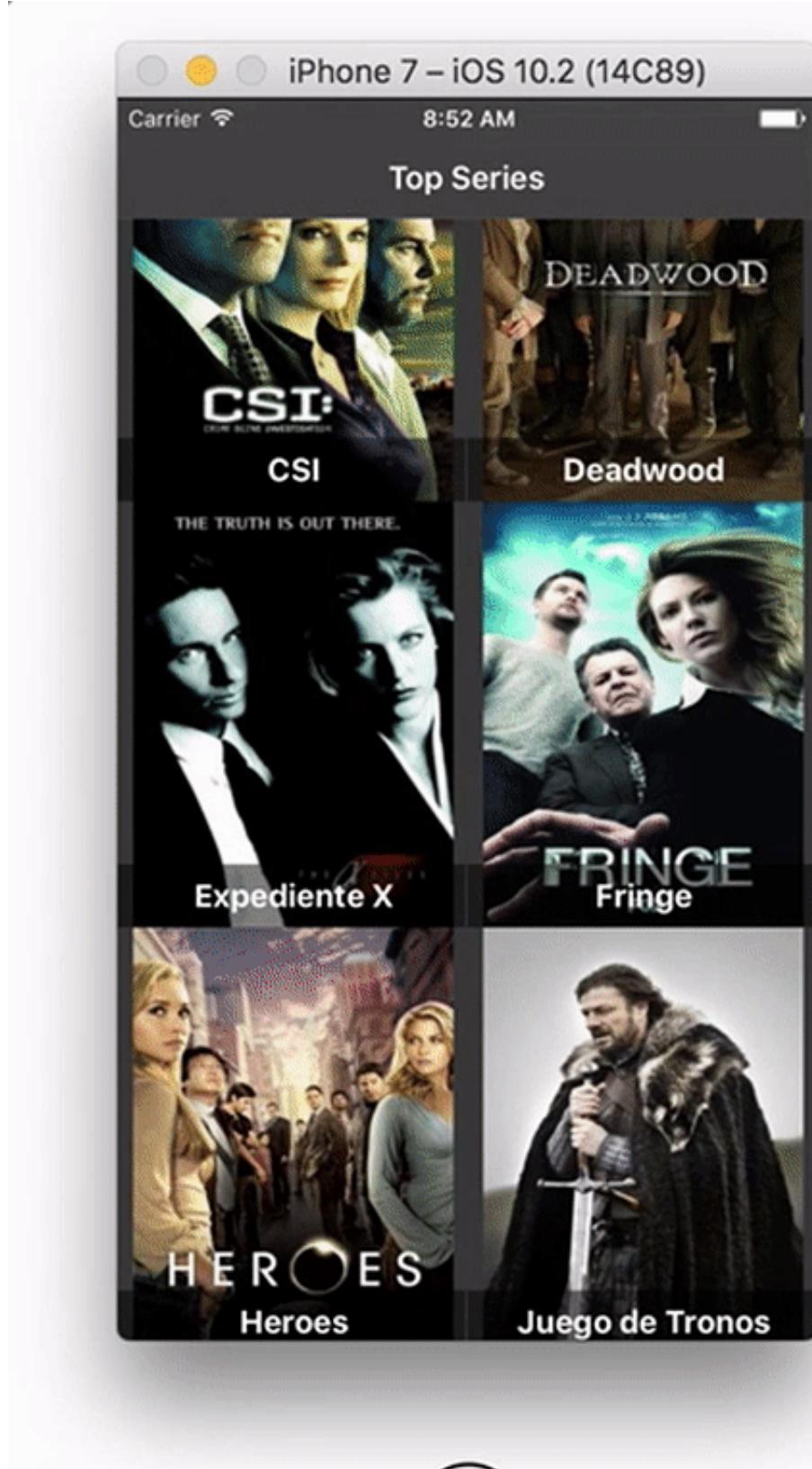
Terminaremos de desarrollar una aplicación que consta de una **Collection View** donde mostraremos imágenes de algunas de las series más conocidas de los últimos tiempos. Además, si el usuario pulsa en cualquiera de las imágenes, accederá al **detalle** de esa serie donde mostraremos el **título** de la serie y la **imagen** en mayor tamaño.

Estos son los temas que vamos a ver en la **segunda parte** del Tutorial:

- ¿Cómo funcionan las **Collection Views**?
- ¿Cómo realizar **navegación entre pantallas** utilizando un **Navigation Controller**?
- ¿Cuál es la forma de **pasar información** entre diferentes **View Controllers**?
- ¿Cómo puedes **personalizar** la **Navigation Bar** de tu aplicación?

3. La Aplicación que vamos a crear

Al terminar el tutorial, nuestra aplicación tendrá este aspecto:



Con esta sencilla aplicación repasaremos algunos **conceptos importantes** del **Desarrollo iOS**.

4. ¿Qué vimos en la primera parte del Tutorial?

Estos son algunos de los **puntos principales** que vimos en la **primera parte** de nuestro Tutorial sobre **Collection Views**:

Aquí tienes un **pequeño resumen** de los puntos más importantes que hemos visto:

1. Protocolos en iOS
2. Los Protocolos **UICollectionViewDataSource** y **UICollectionViewDelegate**
3. **Delegados** en iOS
4. **IndexPath**
5. Cómo añadir datos a una **Collection View**

Si seguiste la primera parte, recordarás que el siguiente paso que debemos dar es **mostrar los datos** de nuestra **Collection View** por pantalla.

¡Vamos a ello!

5. Mostrando Datos en nuestra Collection View

Una vez que hemos creado nuestro **array** con todos los nombres de las series que queremos mostrar por pantalla, es hora de hacer que nuestra Collection View muestre estos datos. Para ello utilizaremos los dos métodos del protocolo **UICollectionViewDataSource**:

- `collectionView:numberOfItemsInSection()`
- `collectionView:cellForItemAt()`

Añade la implementación de estos dos métodos a tu clase **CollectionViewController.swift**:

```
func collectionView(_ collectionView: UICollectionView,  
numberOfItemsInSection section: Int) -> Int {  
  
    return tvSeries.count
```

```
}

func collectionView(_ collectionView: UICollectionView, cellForItemAt indexPath: IndexPath) -> UICollectionViewCell {

    let identifier = "Item"

    let cell = collectionView.dequeueReusableCell(withIdentifier: identifier, for: indexPath) as! SeriesCollectionViewCell

    cell.itemLabel.text = tvSeries[indexPath.row]

    cell.itemImage.image = UIImage.init(imageLiteralResourceName: tvSeries[indexPath.row])

    return cell

}
```

Recuerda que hemos comentado antes, que estos dos métodos **no** se ejecutan una única vez.

Explicando el código

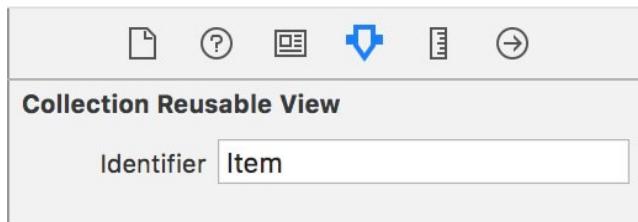
A continuación tienes la **explicación del código** de cada uno de los métodos.

El método **numberOfItemsInSection()**, como hemos dicho, es el encargado de especificar el **número de items** que mostraremos. En nuestro caso, como queremos mostrar tantos items como elementos hayamos almacenado en nuestro array tvSeries, utilizaremos la propiedad **count**, que lo que hace es devolver el número de elementos que hay almacenados en un array. De esta forma, si nuestro array tvSeries **tiene 24 elementos**, nuestro método `numberOfItemsInSection` devolverá 24. Por tanto nuestra Collection View mostrará 14 items con contenido.

Por otro lado, el método **cellForItemAt()** crea un objeto item, que hemos llamado **cell**, de tipo **SeriesCollectionViewCell**, al que se le asigna el identificador “**Item**”.

Este **identificador**, ten en cuenta que debe coincidir con el identificador

que le hayamos asignado en el **Storyboard** al **Item** del **Collection View**.



Posteriormente, lo que hace es asignar a su propiedad **itemLabel.text**, el texto que queremos mostrar, que en nuestro caso, será el elemento que esté contenido en el array, en la posición **indexPath.row**. Recuerda que la propiedad **indexPath.row** indica **el item en el que nos encontramos**, por lo que la primera vez que se ejecute el método **cellForItemAt()**, **indexPath.row** será igual a 0, la segunda será igual a 1, la tercera será igual a 2, de esta forma podremos recorrer nuestro array **tvSeries** y mostraremos cada vez un elemento distinto del array. No olvides que el método **cellForItemAt()** se ejecuta tantas veces como items vayamos a mostrar en nuestra tabla.

Por último, en su propiedad **itemImage.image**, creamos una **UIImage** con el nombre que obtenemos del array **tvSeries**. Para que esto funcione, nos hemos asegurado que el nombre de cada imagen coincide con el nombre de cada serie.

Error al Ejecutar nuestra Aplicación

Ahora, prueba a ejecutar tu aplicación y...

...comprobarás que nuestra **CollectionView** se muestra completamente vacía.

¿Por qué ha sucedido esto?

Si te das cuenta, uno de los protocolos que hemos utilizado se llama **UICollectionViewDelegate**, lo que nos da la pista de que se trata de un **Protocolo de Delegado**.

Si no has trabajado antes con ellos, aquí puedes consultar todo lo que

necesitas sobre el concepto de [Delegado](#).

Como has podido ver, hay que seguir **3 pasos** para implementar un delegado. Nosotros solo hemos realizado 2 pasos:

- **Paso 1: OK** – En la clase que actuará como Delegado lo hemos indicado específicamente añadiendo el protocolo **UICollectionViewDelegate** en su cabecera.

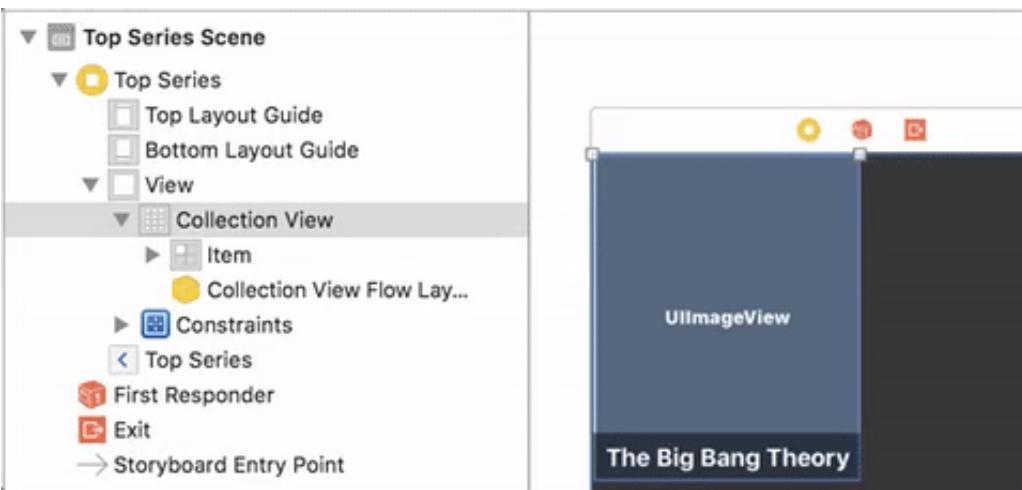
```
class CollectionViewController: UIViewController, UICollectionViewDelegate, UICollectionViewDataSource {
```

- **Paso 2: PENDIENTE** – En la clase que delegará sus funciones tendremos que indicar quien será su clase Delegado. En este caso, no hemos especificado todavía en nuestra **Collection View** quien será su clase delegado.
- **Paso 3: OK** – Hemos implementado las funciones que queremos que realice el delegado a través de los métodos **numberOfItemsInSection()** y **cellForItemAt()**

Para que el delegado funcione correctamente no podemos saltarnos el Paso 2. Así que vamos a ello. Vamos a especificar en nuestra Collection View **quien será su clase delegado**.

Accede al **Main.storyboard** y dejando la tecla **Ctrl** pulsada haz clic en el objeto **Collection View** y arrastra hasta soltar justo encima del **ViewController**. En el menú flotante que aparece, selecciona **delegate**.

Repite el proceso de nuevo y selecciona también **dataSource**.



Ahora ya puedes **ejecutar la aplicación** y comprobar que nuestra Collection View muestra correctamente los **datos** contenidos en el array **tvSeries**.



Si pulsas en las **imágenes** de las **series**, verás que nuestra aplicación no hace nada. No realiza ningún tipo de navegación. Esta **funcionalidad** la

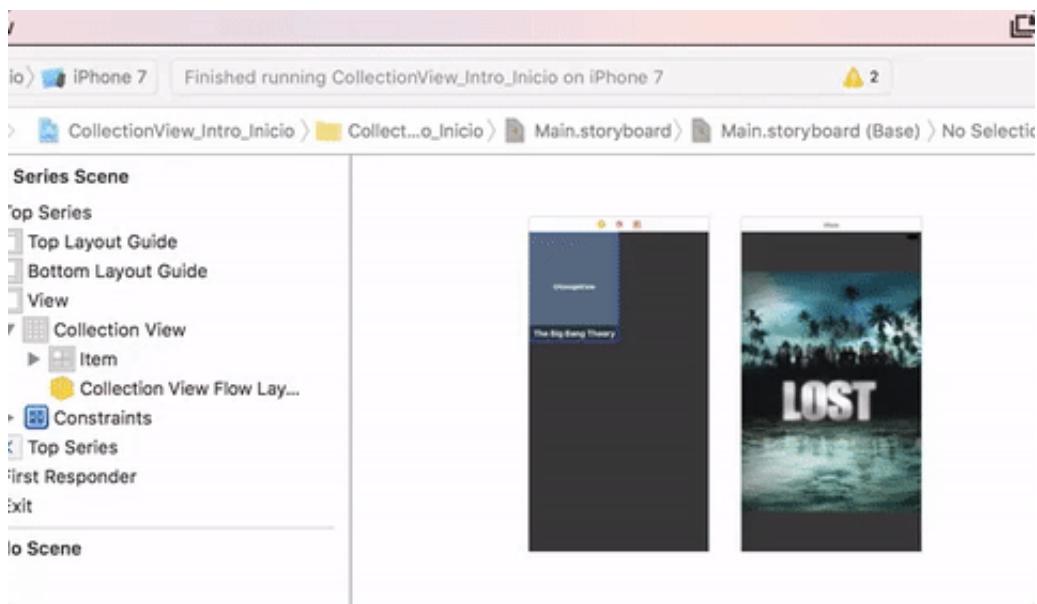
vamos a añadir en el **siguiente punto** del Tutorial.

6. Añadiendo navegación

Para realizar la navegación entre la primera pantalla y la segunda, vamos a utilizar un **Navigation Controller**.

El Navigation Controller es la forma más sencilla que nos ofrece Apple para realizar **transiciones entre pantallas** de nuestra aplicación.

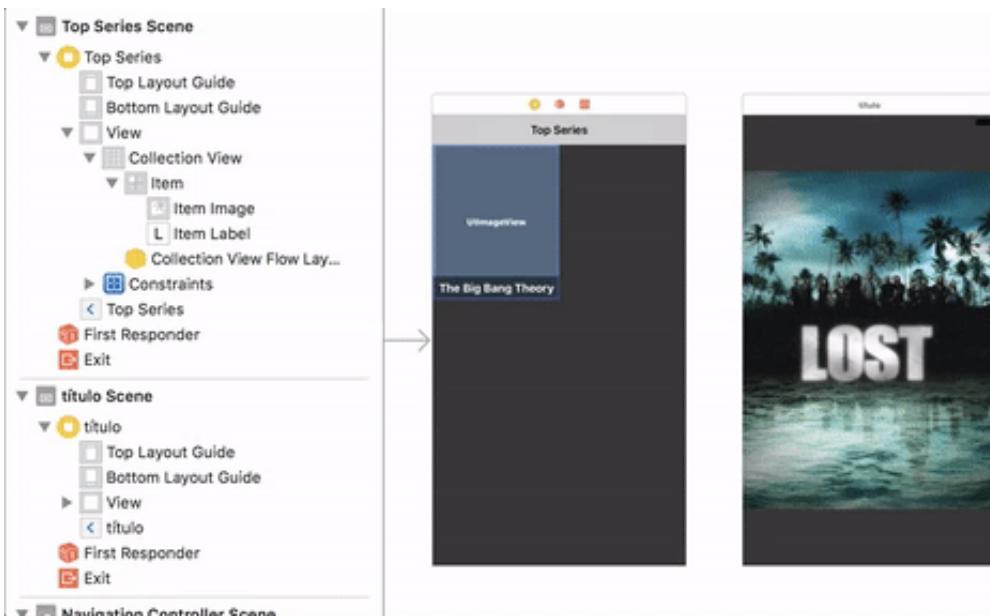
Para añadirlo a nuestra app, abre el fichero **Main.storyboard**, selecciona el primer **View Controller** y haz clic en el menú **Editor>Embed In>Navigation Controller**.



Esto añadirá un **Navigation Controller** que nos permitirá navegar fácilmente entre nuestras pantallas.

Además de esto, debemos crear un **segue** que permita a nuestra aplicación cambiar de pantalla cuando el usuario pulse en cualquier **item** de nuestra **Collection View**.

Para ello, como la transición es entre el **Item** de la **Collection View** y nuestra segunda pantalla, dejamos pulsada la tecla **ctrl** y enlazamos ambos elementos:



Como has podido ver, en el tipo de segue, hemos elegido **Show**. Este es el segue estándar para las transiciones con un Navigation Controller.

Después de realizar este paso, si **ejecutas la aplicación**, verás como, al pulsar sobre la imagen de cualquier serie, nuestra aplicación realiza la navegación hacia la segunda pantalla perfectamente. El problema es que siempre muestra **la misma imagen y el mismo título**:



Esto se debe a que **no hemos especificado** en nuestra aplicación, cual es la **imagen** y el **título** que debemos mostrar en la segunda pantalla y siempre se muestra la imagen y el título que tenemos ahora mismo en nuestro Storyboard.

En el siguiente apartado veremos como solucionar esto.

7. Pasando datos de un View Controller a otro

Lo que queremos conseguir, es que si el **usuario** pulsa en la imagen de Friends, nuestra aplicación muestre la **pantalla** de detalle con el **título** y la **imagen** de dicha serie.

Si pulsa en la imagen de Futurama, deberemos mostrar el título Futurama y la imagen correspondiente a esta serie.

Para poder hacer esto, está claro que debemos de **pasar información** desde **CollectionViewController.swift** a **DetailViewController.swift**.

Concretamente necesitaremos pasar el **título** de la serie sobre la que pulse el usuario.

Para poder **pasar información entre dos View Controllers** que están unidos por un **segue**, tenemos a nuestra disposición este método:

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

Así que, vamos a añadir el siguiente **código** al final de nuestra clase **CollectionViewController.swift**:

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {  
    let item = sender as? UICollectionViewCell  
    let indexPath = collectionView.indexPath(for: item!)  
    let detailVC = segue.destination as! DetailViewController  
    detailVC.detailName = tvSeries[(indexPath?.row)!]  
}
```

Explicando el código

Lo que hacemos en este método es lo siguiente:

- Creamos una variable **item** que almacene el item del UICollectionViewCell que haya pulsado el usuario.
- Obtenemos el **indexPath** que tiene ese **item**.
- Creamos una variable **detailVC** a partir de nuestro **DetailViewController**.
- Obtenemos el nombre de la serie que ha pulsado el usuario, a partir de nuestro array **tvSeries** y el **indexPath** que hemos conseguido anteriormente. Almacenamos este nombre en la variable **detailName** de nuestro **detailVC**.

De esta forma, tendríamos almacenado el nombre de la serie que ha pulsado el usuario en la variable **detailName** de nuestro **DetailViewController**.

Lo siguiente que tendríamos que hacer es utilizar la variable **detailName** para modificar el **título** y la **imagen** de la segunda pantalla de nuestra aplicación. Para ello, abre **DetailViewController.swift** y en el método **viewDidLoad()** añade lo siguiente:

```
override func viewDidLoad() {  
    super.viewDidLoad()  
  
    title = detailName  
  
    detailImage.image = UIImage.init(named: detailName!)  
}
```

Como ves, lo único que hacemos es utilizar la variable **detailName** para modificar el **título** y la **imagen** de nuestra pantalla de detalle.

Si ahora ejecutas la aplicación y pulsas en cualquier serie, verás como nuestra app **realiza la transición** y muestra los **detalles de la serie** en la que hayas pulsado.

Todo funciona como debe funcionar.

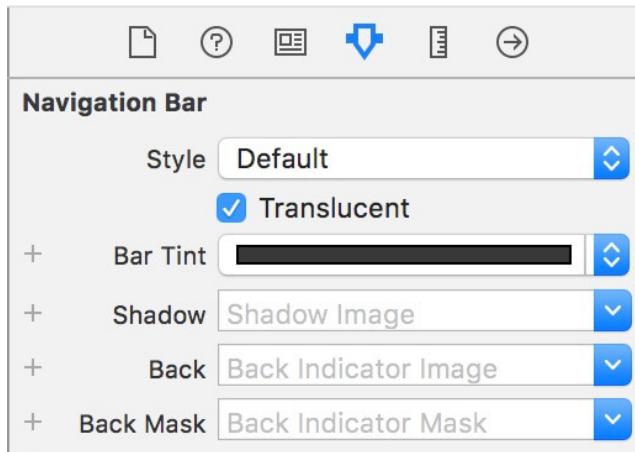
A **nivel de diseño**, la app no ha quedado del todo bien. El color de la **Navigation Controller** no es el correcto y la **status bar** de nuestra aplicación aparece en color oscuro, cuando quedaba mucho mejor si fuera blanca. Además el **botón Back** aparece en color azul, cuando debería ser blanco.

Todos estos detalles los **vamos a corregir** en el siguiente apartado del tutorial

8. Corrigiendo algunos detalles

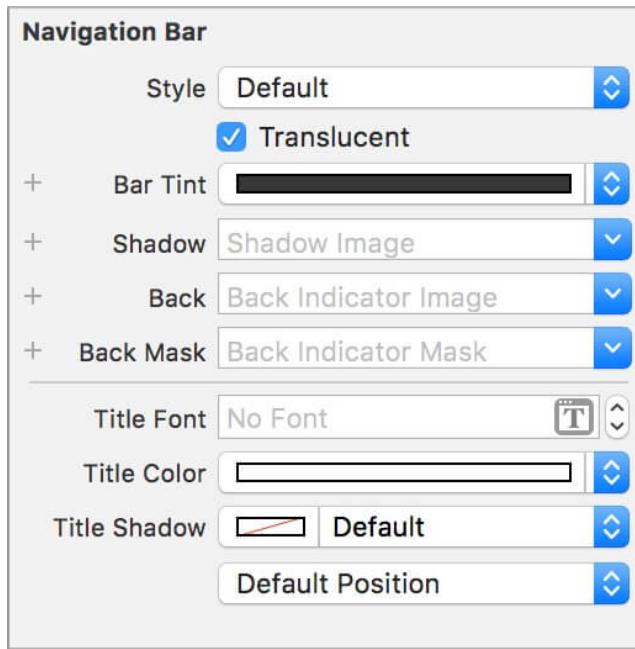
Cambiando el color de nuestra Navigation Bar

Lo primero que vamos a hacer es cambiar el color de nuestra **Navigation Bar** de blanco a oscuro. Para ello, abre el **Main.storyboard** y selecciona la **Navigation Bar**. Haz clic en su propiedad **Bar Tint** y elige este color: **383838**.



Cambiando el color de los títulos de los View Controllers

Al cambiar nuestra **Navigation Bar** a un color oscuro, debemos cambiar los títulos de nuestros **View Controllers** a blanco. Para ello, selecciona la **Navigation Bar** de nuevo y en su propiedad **Title Color**, elige **White**.

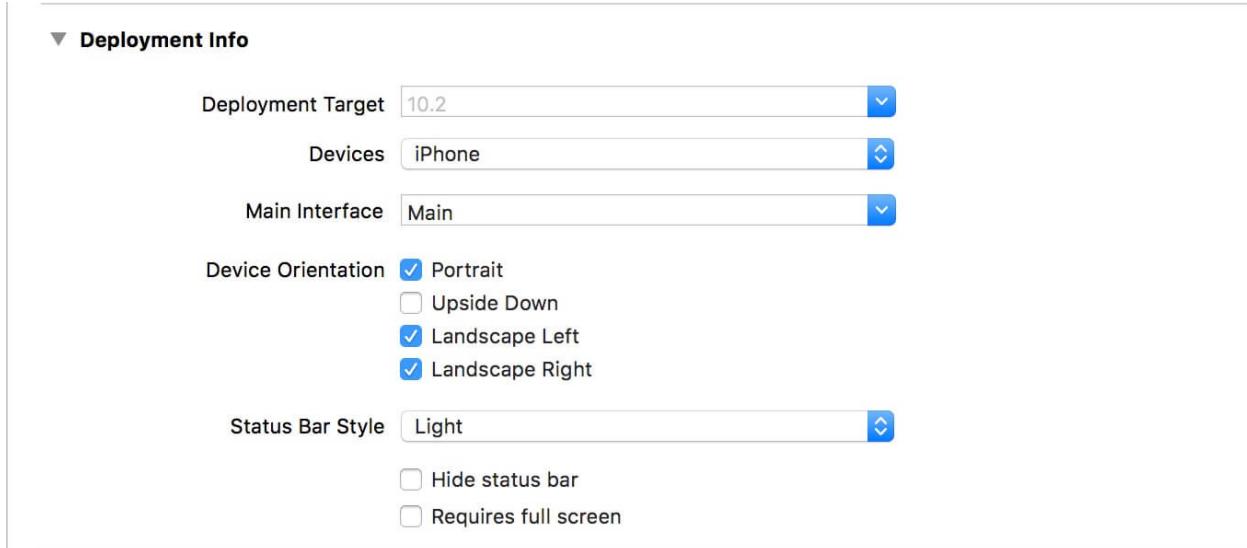


Cambiando el color de la Status Bar

En este caso, para poder modificar el color de nuestra **Status Bar** de negro a blanco, tendremos que seguir estos 2 pasos:

1. Haz clic en el nombre de nuestro proyecto en **Xcode** y selecciona el **Target**. En la pestaña **General**, puedes acceder a las opciones del proyecto. En la parte inferior de ese menú verás una opción llamada **Status Bar Style**. Cambia esa opción de **Default** a **Light**.
2. Haz clic en el fichero **Info.plist** de tu proyecto y añade esta entrada:

View controller-based status bar appearance : NO



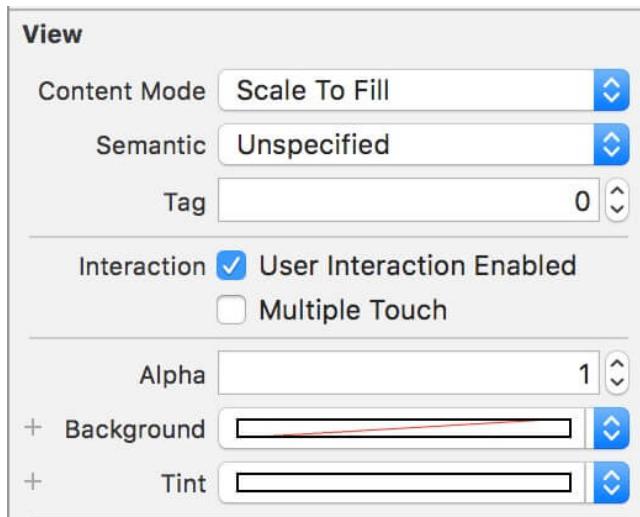
Con estas dos modificaciones, el color de la **Status Bar** de nuestra aplicación pasará de negro a blanco, tal y como queríamos.

Cambiando el color del botón Back

Si ejecutas la aplicación en este punto, verás como los cambios se han realizado correctamente. Sin embargo, si accedes a la **vista detalle**, verás como el botón para volver a la vista principal aparece en color azul.

Vamos a cambiarlo a blanco.

Para ello, accede a **Main.storyboard**, selecciona la **Navigation Bar** y cambia su propiedad **Tint** a **White**.



Si ejecutas la aplicación verás como todos los **detalles de diseño** han quedado solucionados.

9. Resumen final

Espero que este tutorial te haya servido para entender como funcionan las **Collection Views** en iOS. Se trata de un elemento fundamental, por lo que es interesante que seas capaz de manejarlas perfectamente.

Aquí tienes un **pequeño resumen** de los puntos más importantes que hemos visto:

1. Protocolos en iOS
2. Los Protocolos **UICollectionViewDataSource** y **UICollectionViewDelegate**
3. **Delegados** en iOS
4. **IndexPath**
5. Funcionamiento de una **Collection View**
6. Realizar **navegación** en nuestra app utilizando un **Navigation Controller**
7. **Paso de información** entre diferentes **View Controllers**
8. **Personalización** de la **Navigation Bar**

Utilizando Celdas Personalizadas en una TableView

DOMINA LAS CELDAS PERSONALIZADAS CON SWIFT

Lenguaje Swift | Nivel Principiante

1. Introducción

No se si sueles realizar deporte a menudo.

Imagino que te pasa como a todos. Muchas veces no conseguimos sacar **tiempo libre**.

Mi objetivo es entrenar **3 veces por semana**, aunque la verdad es que no lo consigo todas las semanas.

Hace años jugaba bastante al fútbol, baloncesto, tenis de mesa y de vez en cuando sacaba tiempo para ir al gimnasio.

Ahora con mucho menos tiempo libre tengo que conformarme con 3 sesiones de **entrenamiento intenso** por semana.

Intento seguir una rutina marcada por una aplicación móvil que se basa en **entrenamientos cortos** pero de **alta intensidad**.

Invirtiendo entre **2 y 3 horas por semana** te aseguro que mejorarás mucho tu forma física.

He probado varios sistemas para mantenerme en forma y este es con diferencia el que mas me motiva.

Si tienes curiosidad y crees que puede interesarte, escríbeme, te explico un poco como funciona y te paso el nombre de la **app**.

No gano nada recomendando aplicaciones, simplemente me gusta

compartir con los demás productos y servicios que funcionan y que creo que pueden aportar mucho

2. ¿Qué vamos a ver en este tutorial sobre TableView?

Después de esta pequeña introducción, la **aplicación** de este tutorial estará relacionada con el DEPORTE.

Vamos a desarrollar una app muy sencilla que nos permitirá avanzar un poco más en la comprensión del funcionamiento de las **TableViews**.

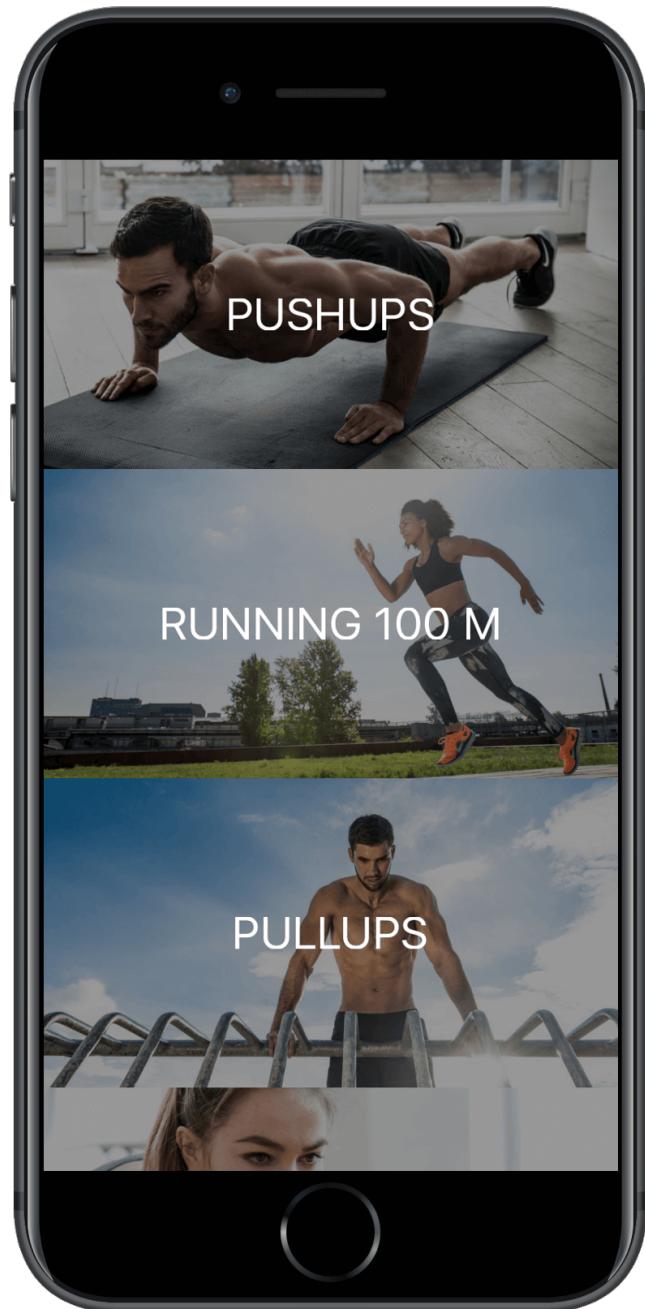
Si no has trabajado antes con Tablas en tus **aplicaciones iOS**, te recomiendo que eches un vistazo al **Tutorial de Introducción a TableView** que aparece al comienzo de este libro.

En este Tutorial de hoy vamos a desarrollar una aplicación que constará de una única vista que contendrá una **TableView** donde mostraremos un **listado de ejercicios** que podrían formar una rutina de entrenamiento cualquiera.

La parte interesante de nuestra aplicación, será la utilización de **Celdas Personalizadas en Swift**.

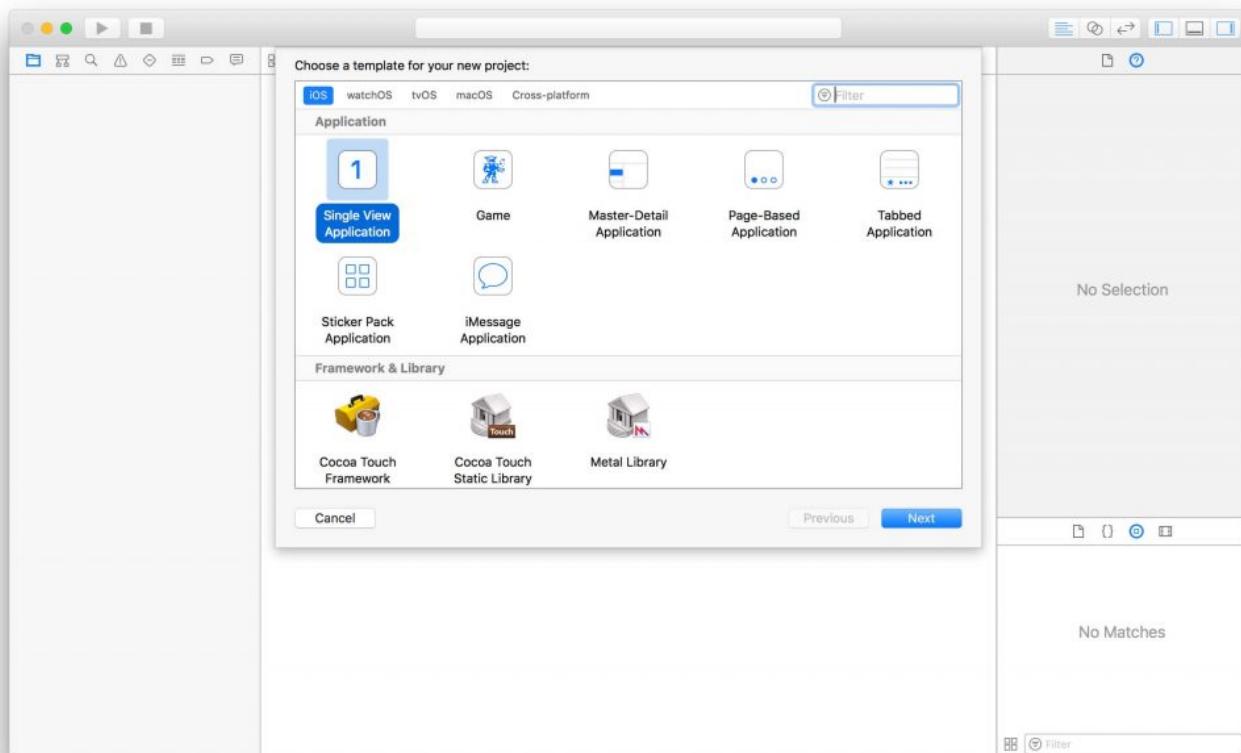
Al utilizar celdas creadas por nosotros mismos, podremos personalizar totalmente el diseño de nuestra app.

Cuando hayas terminado este tutorial, habrás creado una **aplicación** que tendrá el siguiente aspecto:

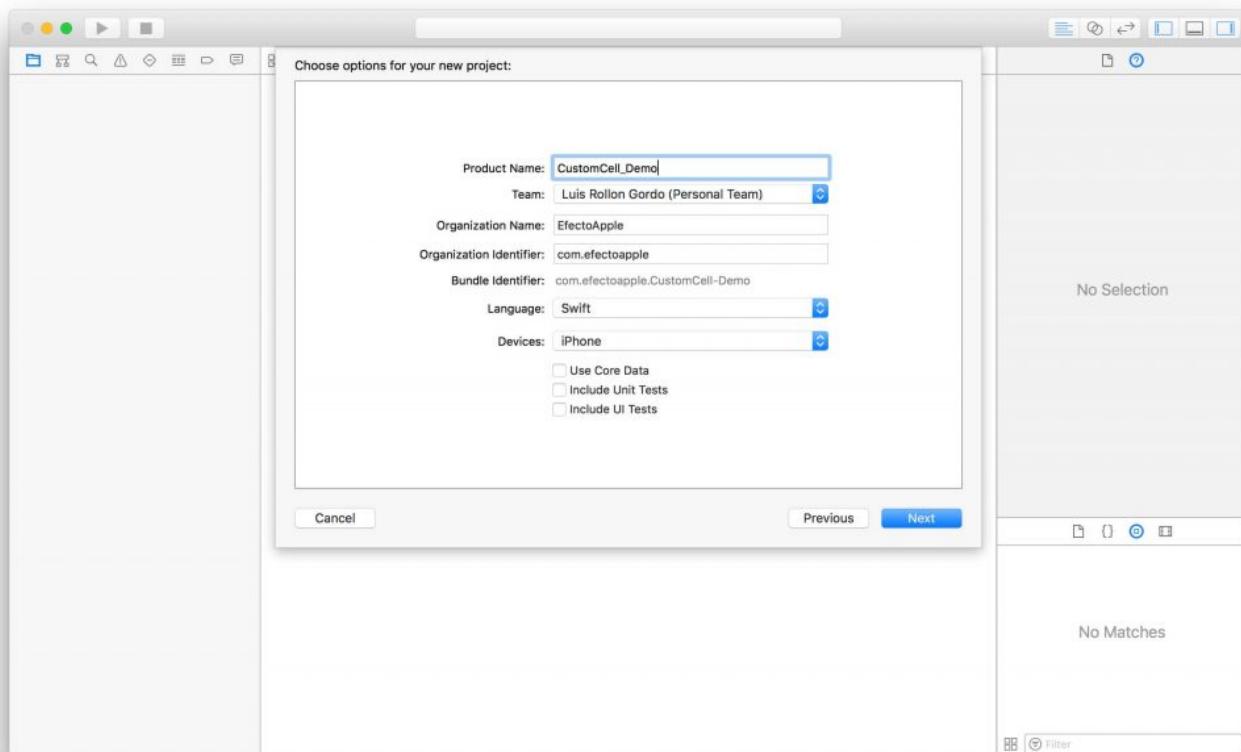


3. Diseñando la Interfaz de nuestra App

Abre **Xcode** y crea un nuevo proyecto. Elige el template **Single View Application**.

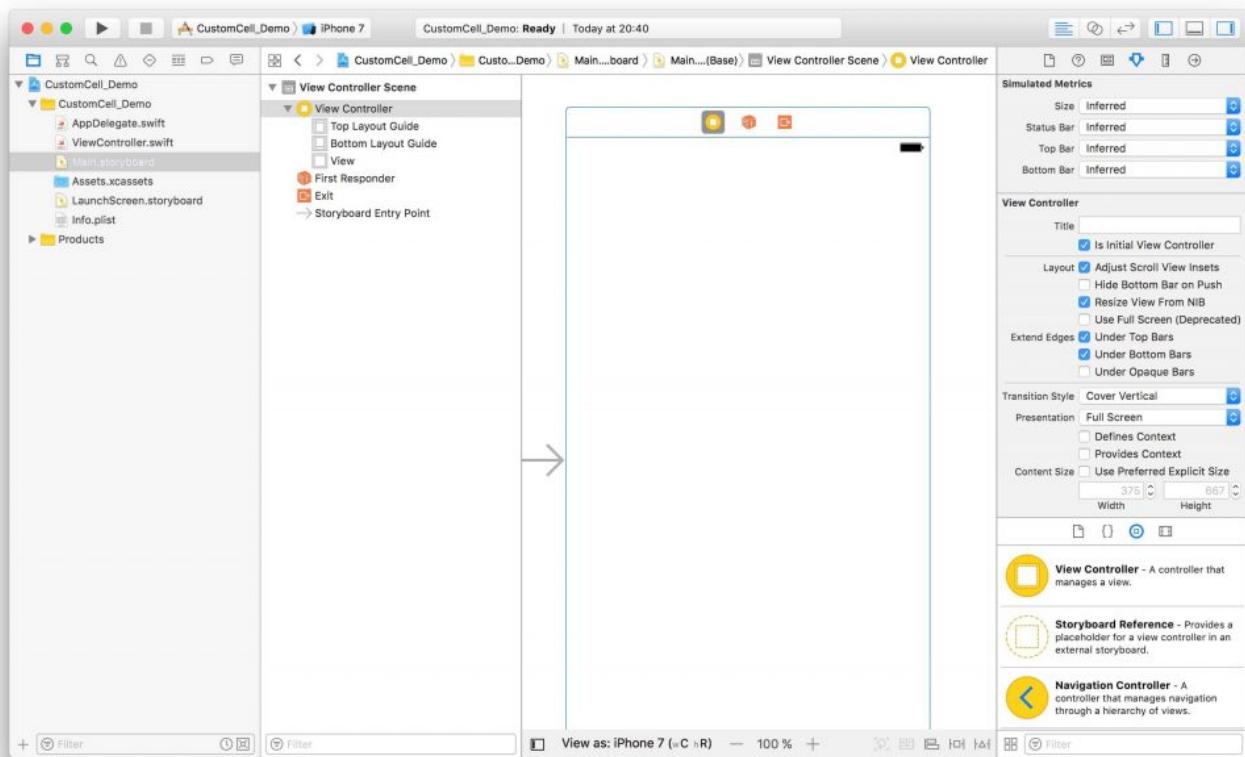


Dale el nombre que quieras al proyecto. En mi caso, yo lo he llamado **CustomCell_Demo**.



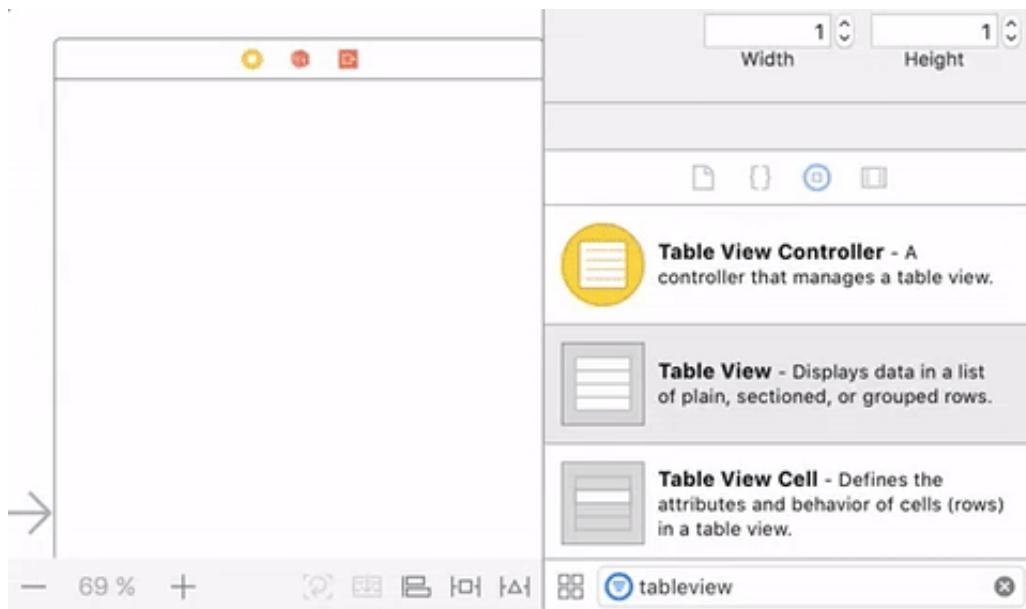
Guárdalo donde quieras y una vez que lo hayas creado, lo primero que debes hacer es abrir **Main.storyboard**.

Allí verás que la **interfaz** de nuestra app consta de una única pantalla en blanco.



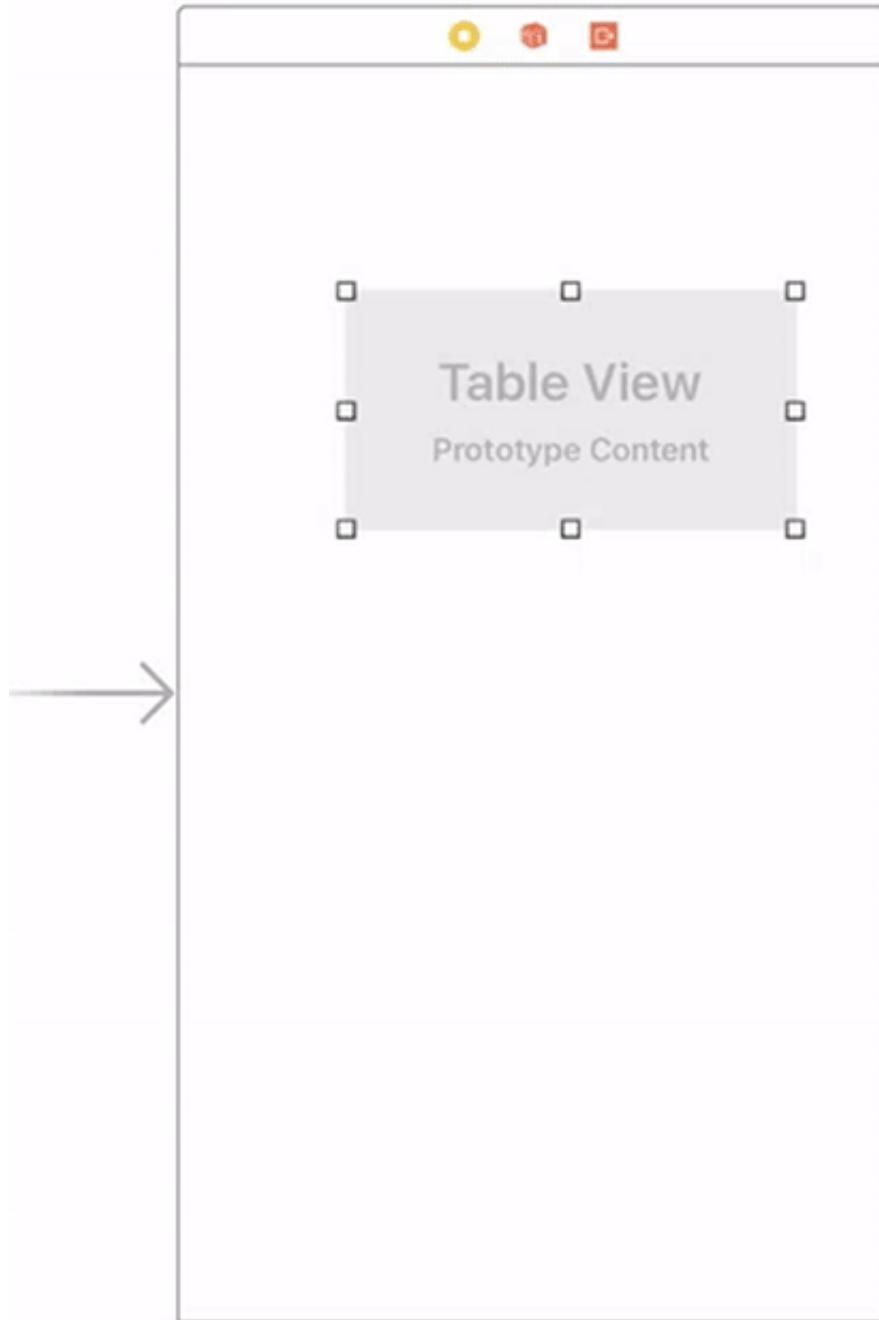
Añadiendo la Tabla a nuestra interfaz

Nuestro primer paso será añadir una **Tabla** a nuestra interfaz. Para ello, simplemente elige un objeto de tipo **UITableView** de la biblioteca de objetos y arrástralolo dentro del **View Controller**.



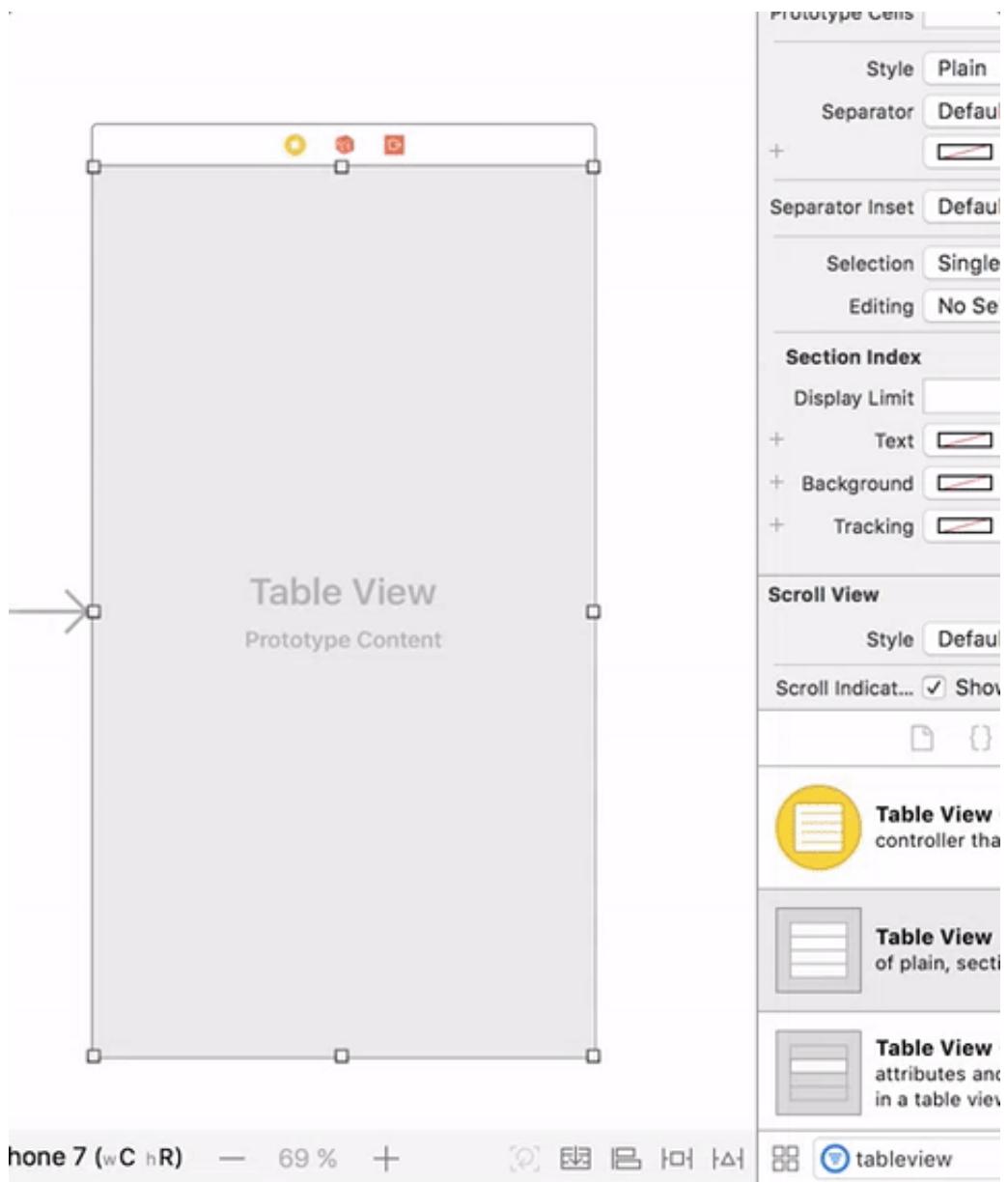
Una vez que la hemos añadido, la **redimensionamos** para que ocupe

toda la **View** del **View Controller**.



Añadiendo Constraints a nuestra tabla

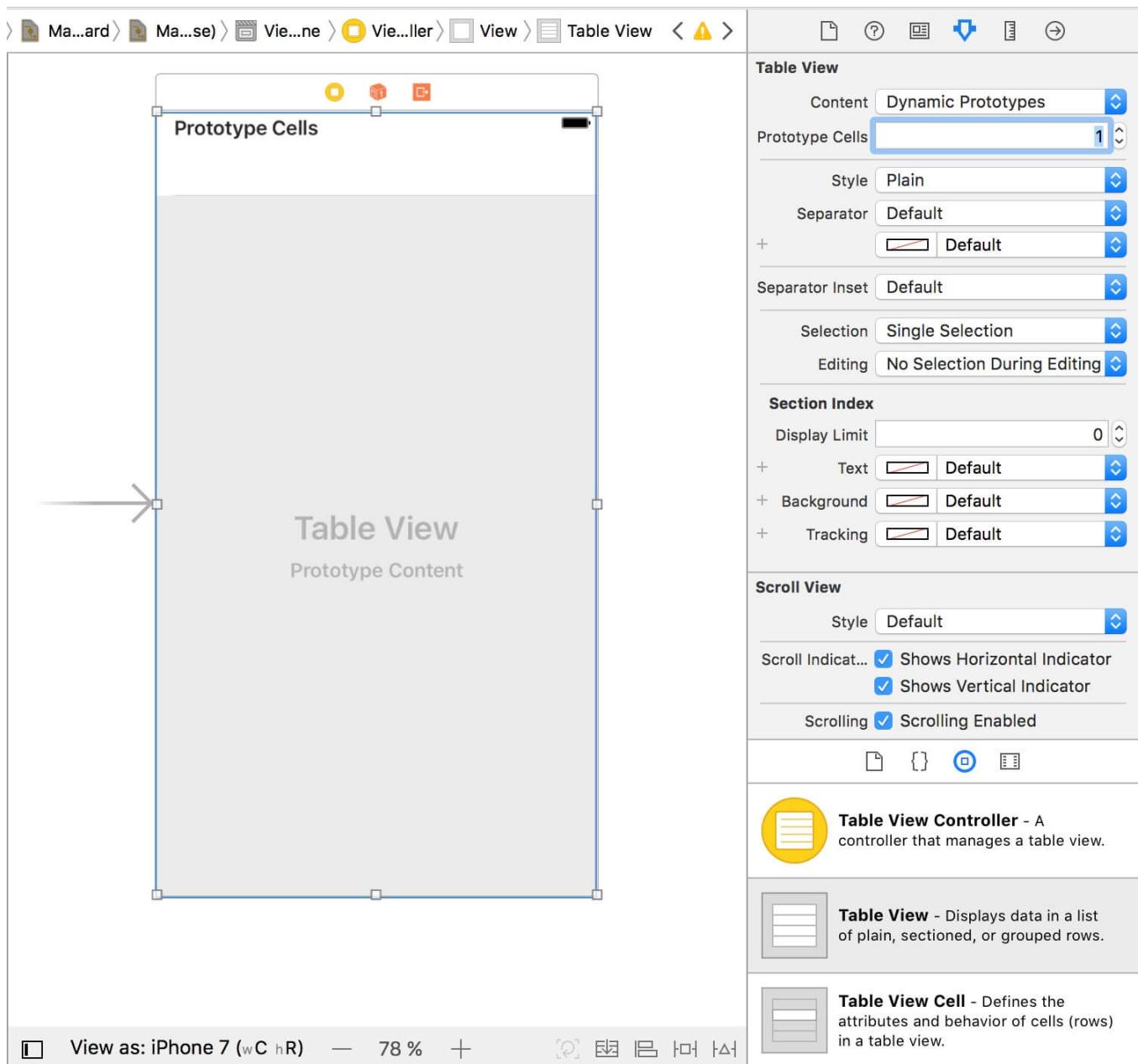
El siguiente paso será añadir las **constraints** a nuestra TableView. De esta forma, nos aseguramos que la aplicación se comportará perfectamente sin importar el tamaño del dispositivo en el que se ejecute.



Trabajando con la Prototype Cell

Cuando añades una TableView a tu interfaz, por defecto, el número de **Prototype Cells** que muestra es 0. Lo normal, es trabajar como mínimo con una **Prototype Cell**, por lo que vamos a añadir una a nuestra tabla.

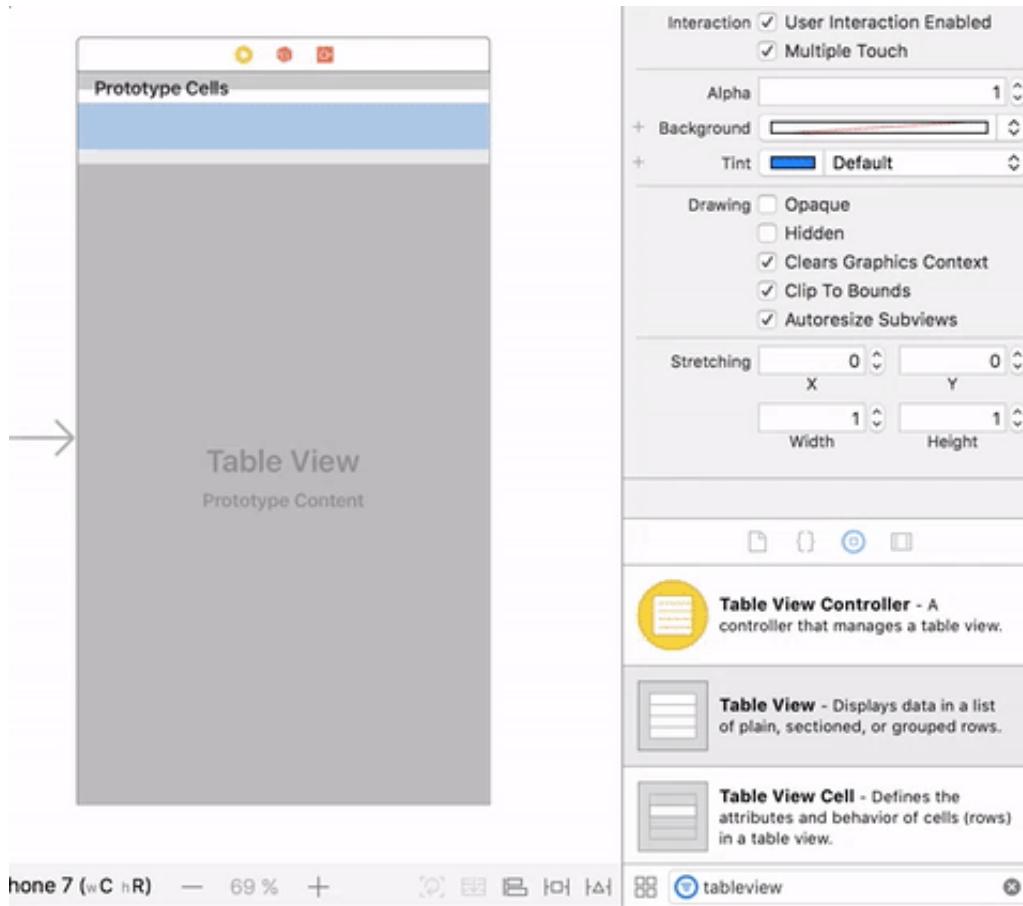
Selecciona la TableView y en el inspector de propiedades cambia **0** por **1**.



Ahora que ya tenemos una **Prototype Cell** con la que trabajar, vamos a diseñar la celda que queremos para nuestra aplicación.

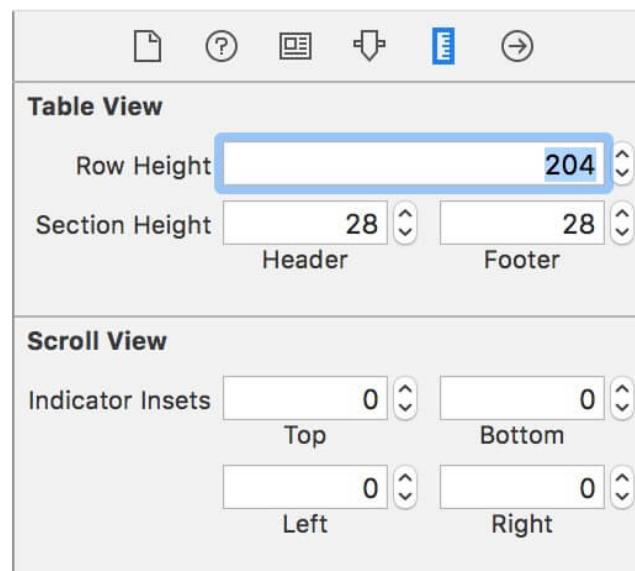
Si te fijas en el diseño final de la app, cada una de las celdas cuenta con una **imagen** de fondo y un **texto** centrado de color blanco.

Por tanto, tenemos que añadir a nuestra celda, una **UIImageView** y una **UILabel**.

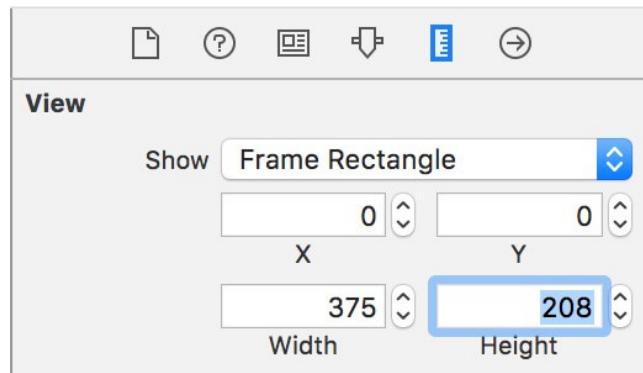


Una vez que hemos añadido los elementos que necesitamos, tendremos que **redimensionarlos** para que cumplan con el diseño que buscamos.

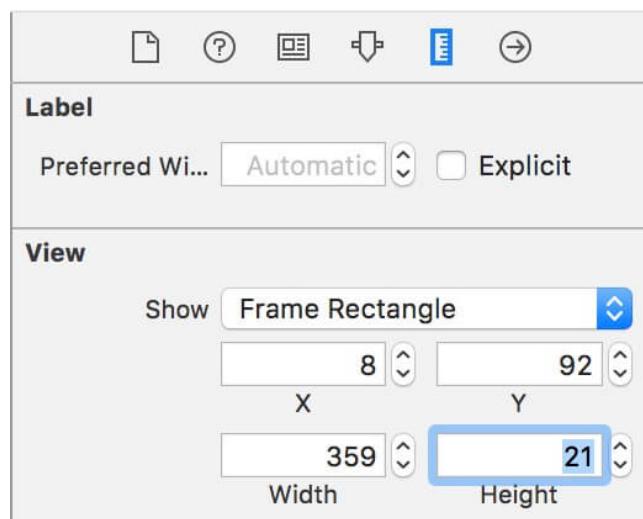
Primero, vamos con nuestra Table View. Seleccionala, accede al **Inspector de Tamaño** y dale un valor de **204** a su propiedad **Row Height**.



Después, haz clic sobre la **UIImageView** y en el inspector de tamaño escribe estos valores:



Por último, selecciona el **UILabel** y establece estos valores:

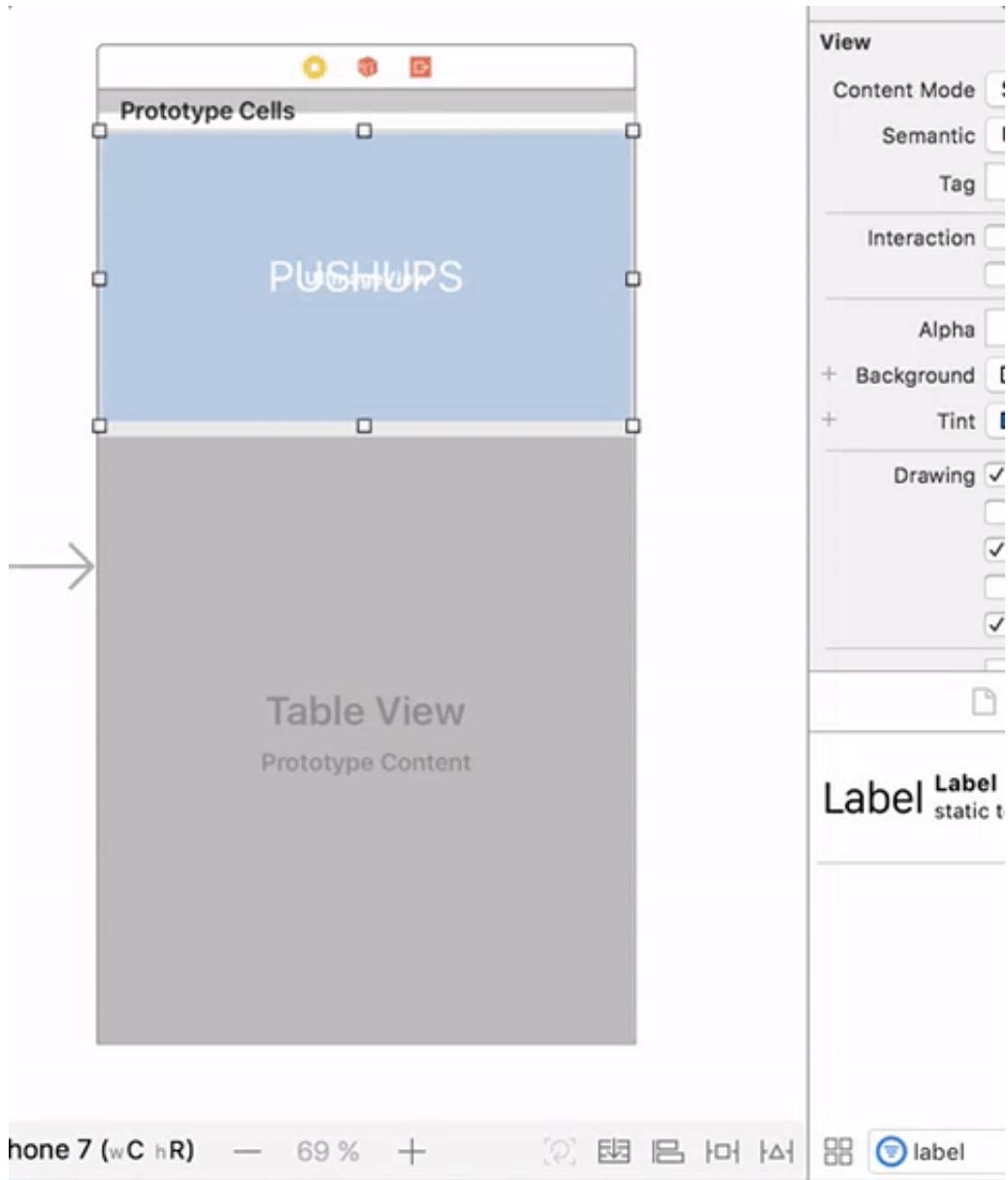


Además, tendrás que centrar el texto del **UILabel**, especificar su color a Blanco, darle un tamaño de fuente de 30 y escribir dentro del label, el texto **PUSHUPS**.

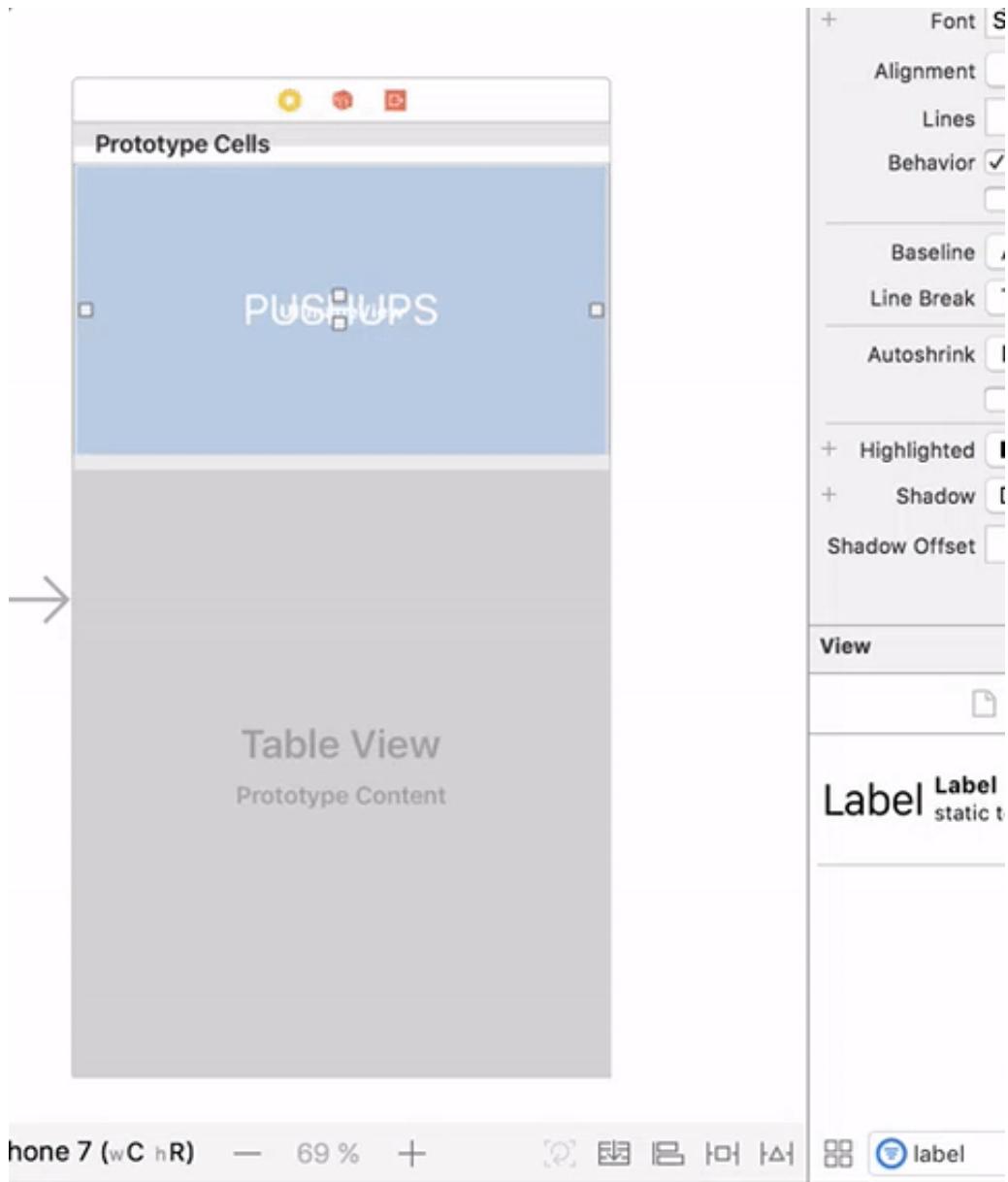
Estableciendo las Constraints del UIImageView y del UILabel

Como siempre, una vez que hemos terminado el diseño de nuestra interfaz, usaremos **Auto Layout** para añadir las constraints.

Comenzaremos por nuestra **UIImageView**. La seleccionamos y añadimos las 4 constraints que necesitamos para que siempre esté centrada en la celda de nuestra TableView.



Después haremos lo mismo con la **UILabel**. La seleccionamos y en este caso añadiremos 2 constraints para que mantenga siempre el mismo **Height** y el mismo **Width** y otras 2 para que siempre esté centrada horizontal y verticalmente en la celda.

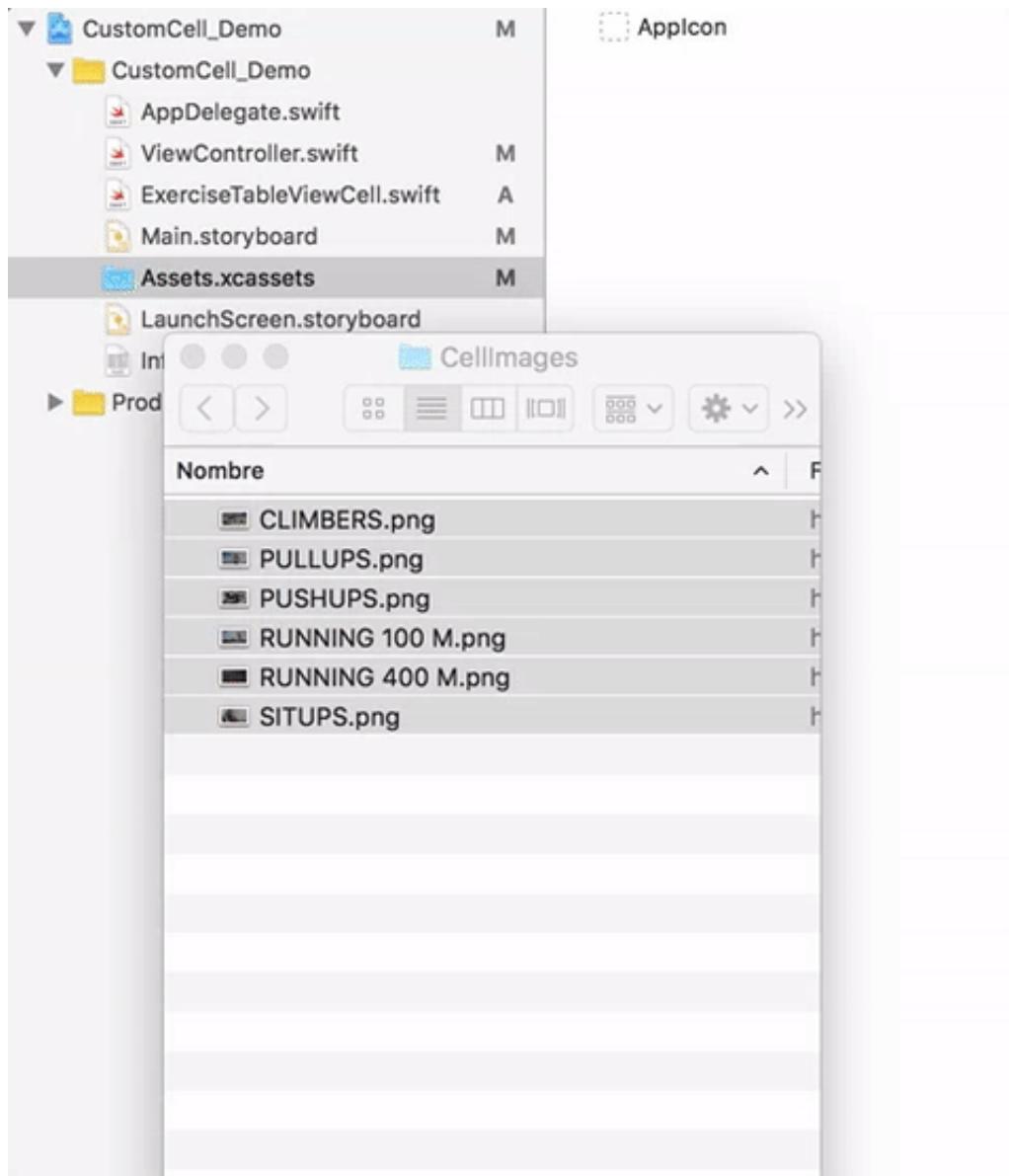


4. Añadiendo Recursos a nuestra Aplicación

Para facilitar el diseño de la app, he creado un **.zip** con las 6 imágenes que vamos a utilizar en su interfaz.

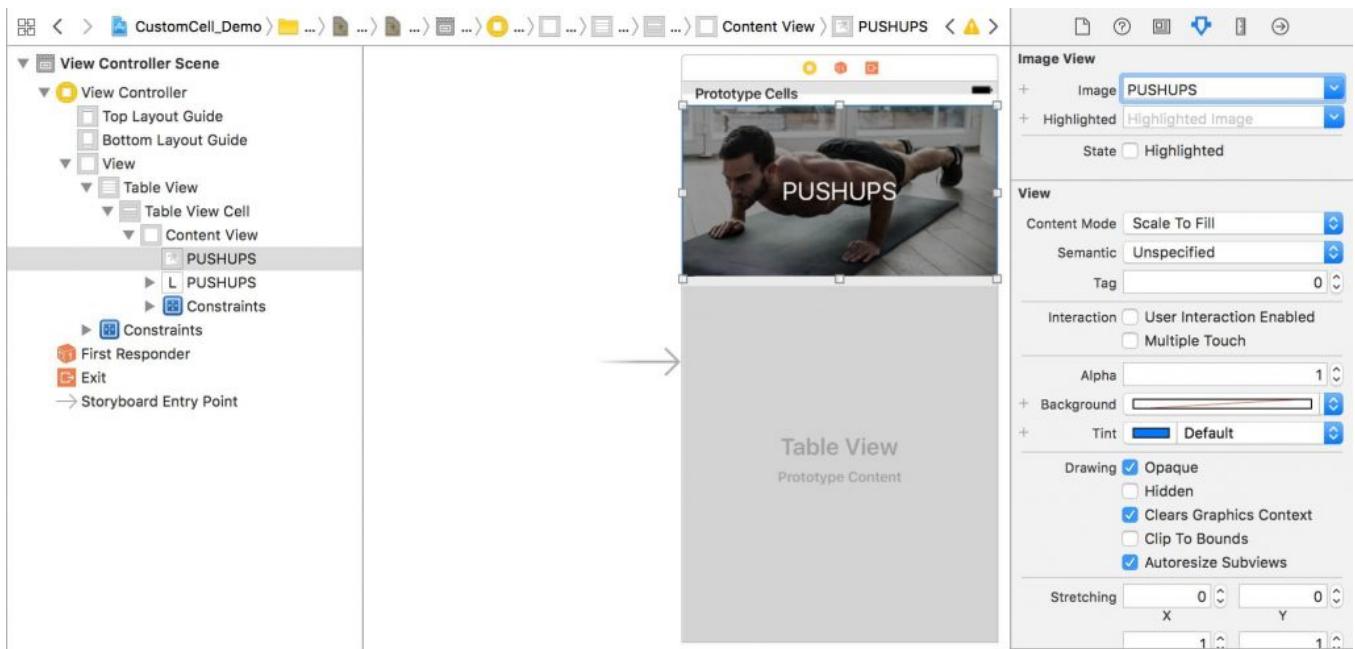
Puedes descargarlo [desde aquí](#).

Una vez que lo hayas descargado, descomprímelo y añade las imágenes que contiene a la carpeta **Assets.xcassets** de tu proyecto.



De esta forma se crearán todos los **assets** que utilizaremos en nuestra aplicación.

Para comprobar que hemos diseñado correctamente los elementos que forman nuestra interfaz, selecciona la **Image View** y en el inspector de propiedades elige la imagen llamada **PUSHUPS**.



Verás como se muestra tal y como debe aparecer la **primera celda** de nuestra tabla.

Con este último paso hemos terminado el diseño de nuestra aplicación.

En el siguiente punto veremos como hacer que nuestra TableView muestre diferentes **celdas personalizadas**.

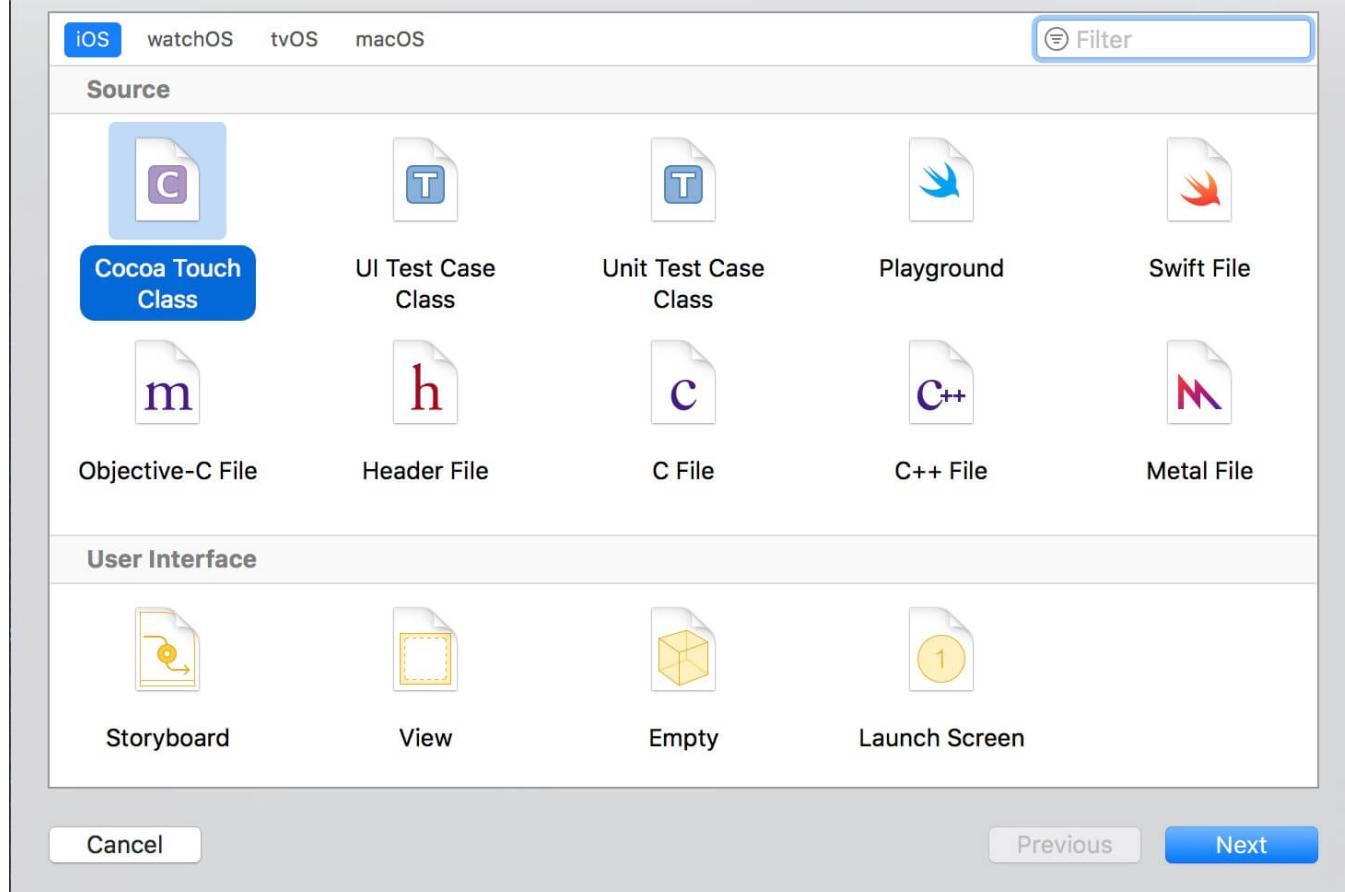
5. Programando la Funcionalidad de nuestra App

La clase ExerciceTableViewCell

Como estamos trabajando con **Celdas Personalizadas**, ha llegado el momento de crear la **Clase** que gestione estas Celdas.

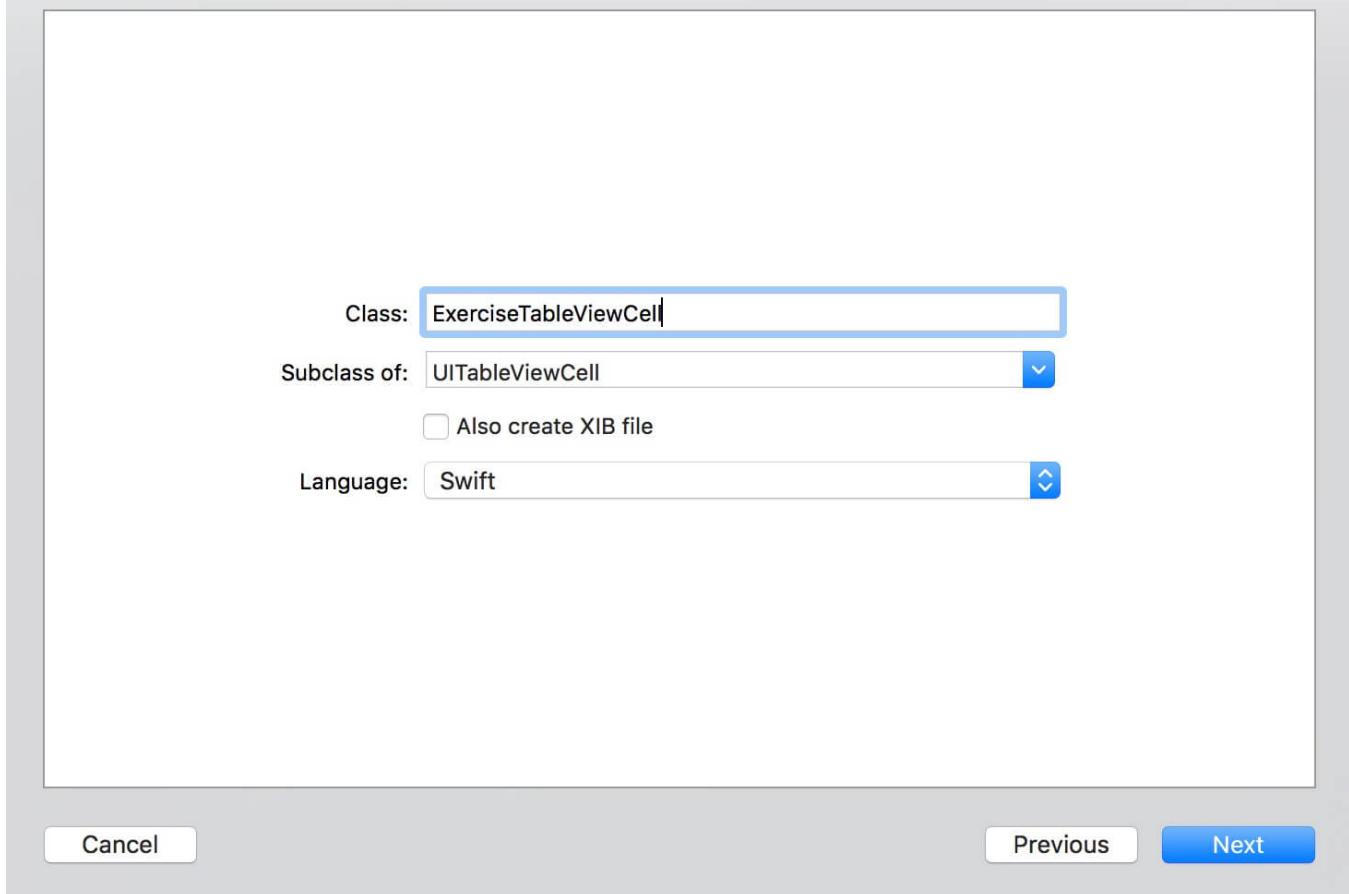
Para ello, crea un nuevo fichero, usando el menú de Xcode, **File>New>File...** , elige el template **Cocoa Touch Class**

Choose a template for your new file:



Dale el nombre **ExerciseTableViewCell** y haz que herede de **UITableViewCell**. Pulsa en **Next** y guárdala con el resto de ficheros del proyecto.

Choose options for your new file:



Lo que vamos a hacer en nuestra **Celda Personalizada** es añadir los elementos de diseño que hemos incluido en la interfaz. Es decir, el **UIImageView** y el **UILabel**.

Para ello, abre **ExerciseTableViewCell.swift** y añade estas dos variables:

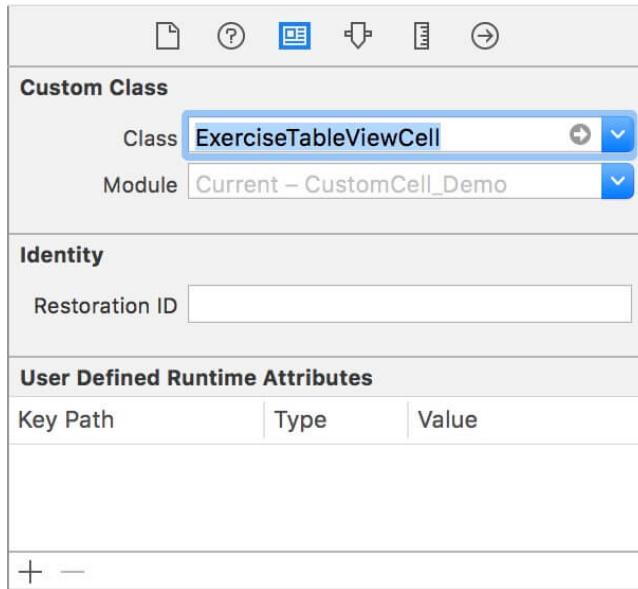
```
@IBOutlet weak var imageCell: UIImageView!  
@IBOutlet weak var labelCell: UILabel!
```

Como puedes ver, se tratan de dos **Outlets**, por lo que ahora iremos al **Main.storyboard** y realizaremos la **conexión** entre estas 2 variables y los elementos de la interfaz.

Enlazando nuestros Outlets

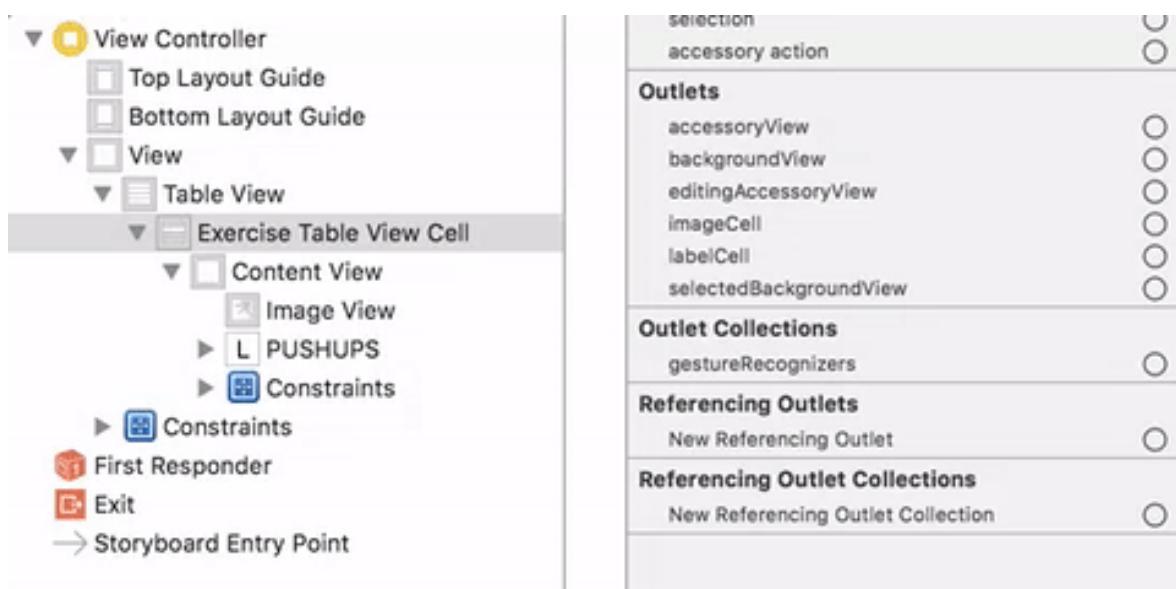
Como decimos, abre **Main.storyboard** y haz clic sobre **Table View Cell**. Después accede al Inspector de Identidad y en su propiedad **Class**, escribe **ExerciseTableViewCell**. Lo que hacemos, es decirle a Xcode,

que esta celda es de tipo **ExerciseTableViewCell**, en lugar de **UITableViewCell**, es decir, que se trata de una Celda Personalizada.



Ahora, sin dejar de seleccionar la **Exercise Table View Cell**, accede al Inspector de Conexiones y verás, como existen 2 conexiones disponibles, llamadas **imageCell** y **labelCell**.

Para realizar la conexión entre el código y la interfaz, arrastra desde cada uno de los círculos situados en la parte derecha hasta cada uno de los dos elementos: **UIImageView** y **UILabel**.



Con esto habríamos terminado con nuestra **Celda Personalizada**.

Tu clase **ExerciseTableViewCell.swift** debería tener un aspecto similar a este:

```
import UIKit

class ExerciseTableViewCell: UITableViewCell {

    @IBOutlet weak var imageCell: UIImageView!
    @IBOutlet weak var labelCell: UILabel!

    override func awakeFromNib() {
        super.awakeFromNib()
        // Initialization code
    }

    override func setSelected(_ selected: Bool, animated: Bool) {
        super.setSelected(selected, animated: animated)
        // Configure the view for the selected state
    }
}
```

La clase ViewController

Nuestra clase **ViewController.swift** está completamente vacía a excepción de los dos métodos que crea Xcode por defecto.

Vamos a realizar los siguientes pasos para completar el código que necesitamos para hacer que funcione nuestra aplicación:

1. Crear el **Modelo de Datos** de nuestra aplicación
2. Ajustar nuestra clase a los protocolos **UITableViewDelegate** y **UITableViewDataSource**

3. Añadir los **métodos** que necesitaremos para que nuestra **TableView** funcione.

Comenzamos por el primer punto. Nuestro modelo de datos será muy sencillo. Consistirá en un **array** que contendrá los nombres de los diferentes **tipos de ejercicios** que mostraremos en nuestra TableView.

Por tanto, tendrás que añadir el siguiente código justo antes del método **viewDidLoad()** de la clase **ViewController.swift**:

```
let exercisesList = ["PUSHUPS", "RUNNING 100 M", "PULLUPS", "SITUPS",  
"RUNNING 400 M", "CLIMBERS"]
```

Como ves, simplemente se trata de una **constante** que almacenará nuestro array de ejercicios.

Si fuéramos estrictos con MVC, no deberíamos incluir nuestro Modelo dentro de nuestro Controlador. Unicamente lo hacemos así para no añadir complejidad al Tutorial

Nuestro siguiente paso será ajustar la clase **ViewController.swift** a los protocolos anteriormente mencionados.

Para ello, tendrás que añadir estos **protocolos** a continuación de la declaración de tu clase. Es decir, la cabecera de tu clase tendrá que tener este aspecto:

```
class ViewController: UIViewController, UITableViewDelegate,  
UITableViewDataSource {  
}
```

En el momento en que añadas estos protocolos, Xcode mostrará un **error**:

The screenshot shows the Xcode interface with the project 'CustomCell_Demo' selected. The status bar at the top right indicates 'Finished running CustomCell_Demo on iPhone 7'. The left sidebar shows 'Buildtime (2)' and 'Runtime' sections. Under 'CustomCell_Demo 2 issues', there are two entries: 'Unsupported Configuration' (warning) and 'Swift Compiler Error' (error). The error details are as follows:

```
1 // ViewController.swift
2 // CustomCell_Demo
3 // Created by Luis Rollon Gordo on 6/2/17.
4 // Copyright © 2017 EfectoApple. All rights reserved.
5
6 import UIKit
7
8
9
10 class ViewController: UIViewController, UITableViewDelegate, UITableViewDataSource {
11     let exercisesList = ["PUSHUPS", "RUNNING 100 M", "PULLUPS", "SITUPS", "RUNNING 400 M",
12                         "CLIMBERS"]
13
14     override func viewDidLoad() {
15         super.viewDidLoad()
16         // Do any additional setup after loading the view, typically from a nib.
17     }
18
19
20     override func didReceiveMemoryWarning() {
21         super.didReceiveMemoryWarning()
22         // Dispose of any resources that can be recreated.
23     }
24
25
26 }
```

The error message is: 'Type 'ViewController' does not conform to protocol 'UITableViewDataSource''. This is highlighted in red.

Para **solucionar** este error, tendremos que continuar con el punto 3 de nuestra lista. Es decir, añadir los métodos **numberOfRowsInSection()** y **cellForRowAtIndex()** a nuestra clase **ViewController.swift**:

```
func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
    return exercisesList.count
}

func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCell(withIdentifier: "ExerciseCell", for: indexPath) as! ExerciseTableViewCell
    cell.labelCell.text = exercisesList[indexPath.row]
    cell.imageCell.image = UIImage(named: exercisesList[indexPath.row])
    return cell
}
```

Explicando el código de nuestros métodos

El código de estos métodos es muy sencillo.

En el primer método, lo único que hacemos es devolver el **número de celdas** que tendrá nuestra aplicación. El número de celdas en nuestro caso

será el número de elementos que tenga nuestro array **exercisesList**.

En el segundo método creamos una celda de tipo **ExerciseTableViewCell** (Ya que es el tipo de celda que hemos creado para nuestra app) y a esta celda le asignamos como texto en su **label** el elemento del array que le corresponda y como **imagen**, una nueva **UIImage** creada a partir del nombre del elemento almacenado en nuestro array de ejercicios.

Con estos dos métodos habríamos terminado nuestra clase **ViewController.swift**, que debería tener este aspecto:

```
import UIKit

class ViewController: UIViewController, UITableViewDelegate,
UITableViewDataSource {

    let exercisesList = ["PUSHUPS", "RUNNING 100 M", "PULLUPS",
"SITUPS", "RUNNING 400 M", "CLIMBERS"]

    override func viewDidLoad() {
        super.viewDidLoad()

        // Do any additional setup after loading the view, typically from a nib.
    }

    func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
        return exercisesList.count
    }

    func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
        let cell = tableView.dequeueReusableCell(withIdentifier: "ExerciseCell",
for: indexPath) as! ExerciseTableViewCell
```

```
cell.labelCell.text = exercisesList[indexPath.row]
cell.imageCell.image = UIImage(named: exercisesList[indexPath.row])
return cell
}

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

6. Ejecutando nuestra Aplicación

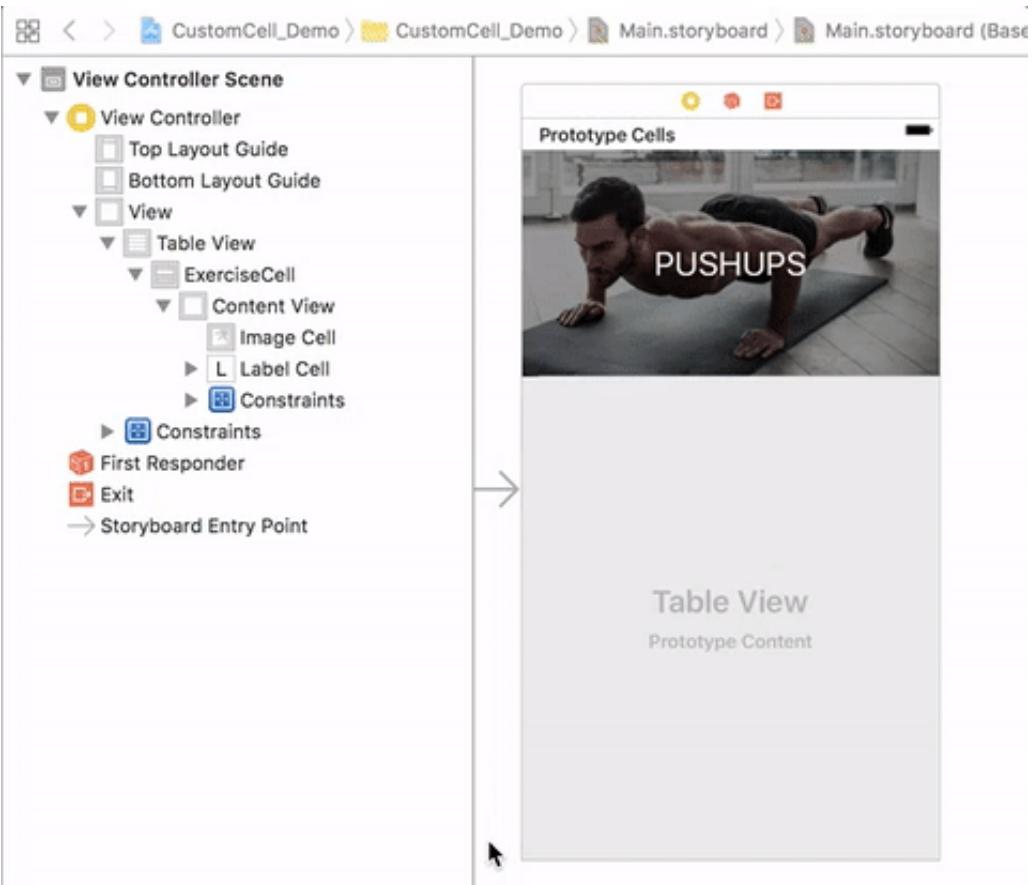
Ahora que ya hemos terminado el código de nuestra app, haz **cmd + R** para ejecutar la aplicación.

Verás que en lugar de mostrar nuestra aplicación funcionando, el simulador muestra la tabla vacía.

¿Qué ha pasado?

Lo que pasa es que hemos cometido un **error muy típico** a la hora de trabajar con UITableViews. No hemos especificado cual será la clase que funcionará como **dataSource** y como **delegate** de nuestra TableView.

Para ello, abrimos **Main.storyboard** de nuevo y enlazamos el **dataSource** y el **delegate** de nuestra **TableView** con nuestra clase **ViewController**:



Ahora vuelve a ejecutar la aplicación y...

Verás que se produce un **error en tiempo de ejecución** que hace que nuestra aplicación se caiga:

```
2017-02-07 20:45:03.585 CustomCell_Demo[25312:1887560] *** Assertion failure in -[UITableView dequeueReusableCellWithIdentifier:forIndexPath:], /BuildRoot/Library/Caches/com.apple.xbs/Sources/UIKit_Sim/UIKit-3600.6.21/UITableView.m:6600
2017-02-07 20:45:03.802 CustomCell_Demo[25312:1887560] *** Terminating app due to uncaught exception 'NSInternalInconsistencyException', reason: 'unable to dequeue a cell with identifier ExerciseCell - must register a nib or a class for the identifier or connect a prototype cell in a storyboard'
*** First throw call stack:
```

Si echas un vistazo al error, verás una descripción como esta:

'unable to dequeue a cell with identifier ExerciseCell – must register a nib or a class for the identifier or connect a prototype cell in a storyboard'

Esto nos da una pista bastante grande de lo que ha fallado.

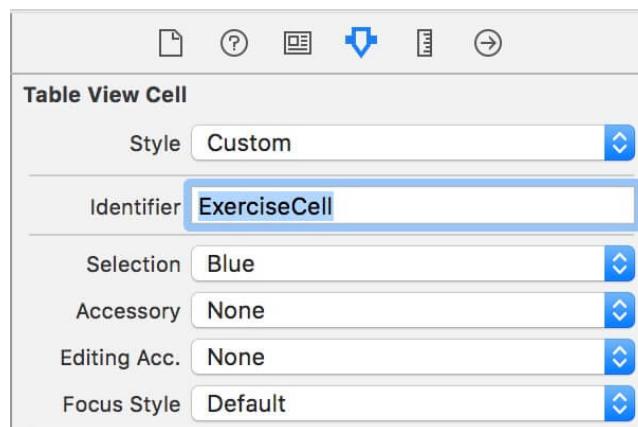
Si recuerdas, en el método **cellForRowAtIndex()** escribimos esta linea de código:

```
let cell = tableView.dequeueReusableCell(withIdentifier: "ExerciseCell", for:
```

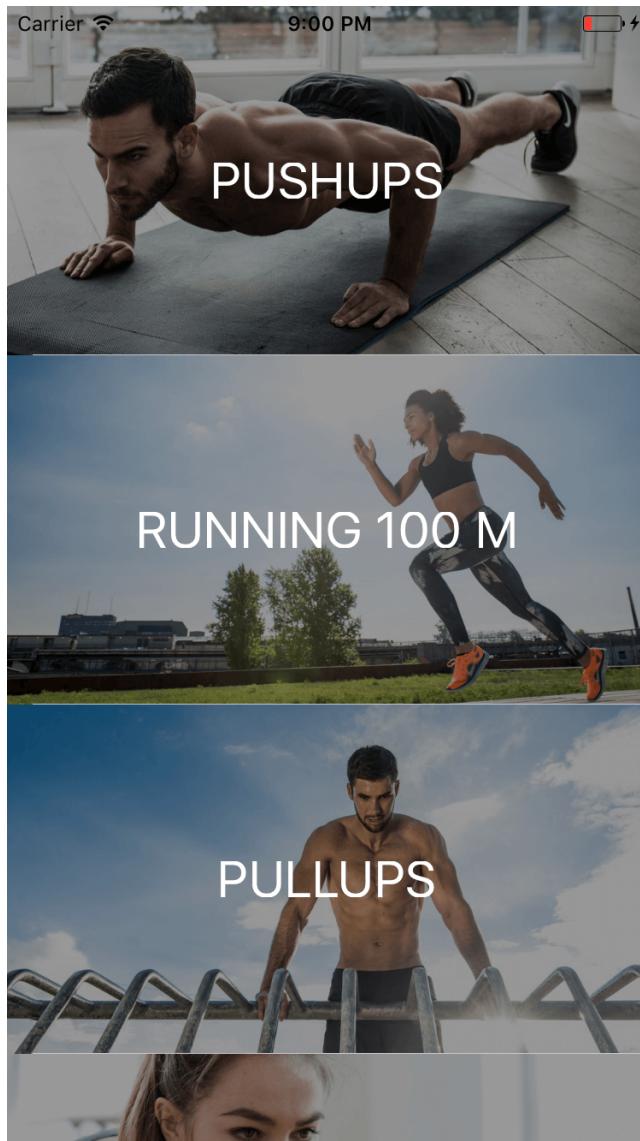
```
| indexPath) as! ExerciseTableViewCell
```

En esta linea, hemos especificado un **Identificador** para nuestra celda, pero hemos olvidado especificar en la celda de nuestro **Storyboard**.

Para solucionarlo, abrimos el fichero **Main.storyboard**, hacemos clic en nuestra Cell y en el Inspector de Atributos, en la propiedad **Identifier**, escribimos **ExerciseCell**.



Ahora si. **Ejecuta** la aplicación y verás como el diseño de nuestra aplicación se muestra perfectamente.



Sin embargo, existen **2 pequeños detalles** que deslucen un poco nuestra aplicación:

1. Se muestra la **status bar**, cuando en el diseño final de nuestra aplicación no aparecía
2. Se muestran los **separadores** de las celdas de color blanco, algo que no queremos que aparezca

En el apartado final vamos a corregir estos dos errores.

7. Afinando los últimos detalles

Para hacer que no aparezca la status bar en nuestra aplicación tendremos que añadir 2 entradas nuevas en el fichero **Info.plist**.

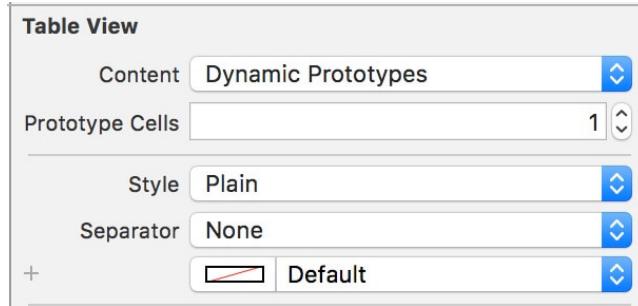
Así que abre ese fichero, sitúate en el último elemento de la lista, pulsa en

el botón + y añade estas dos entradas:

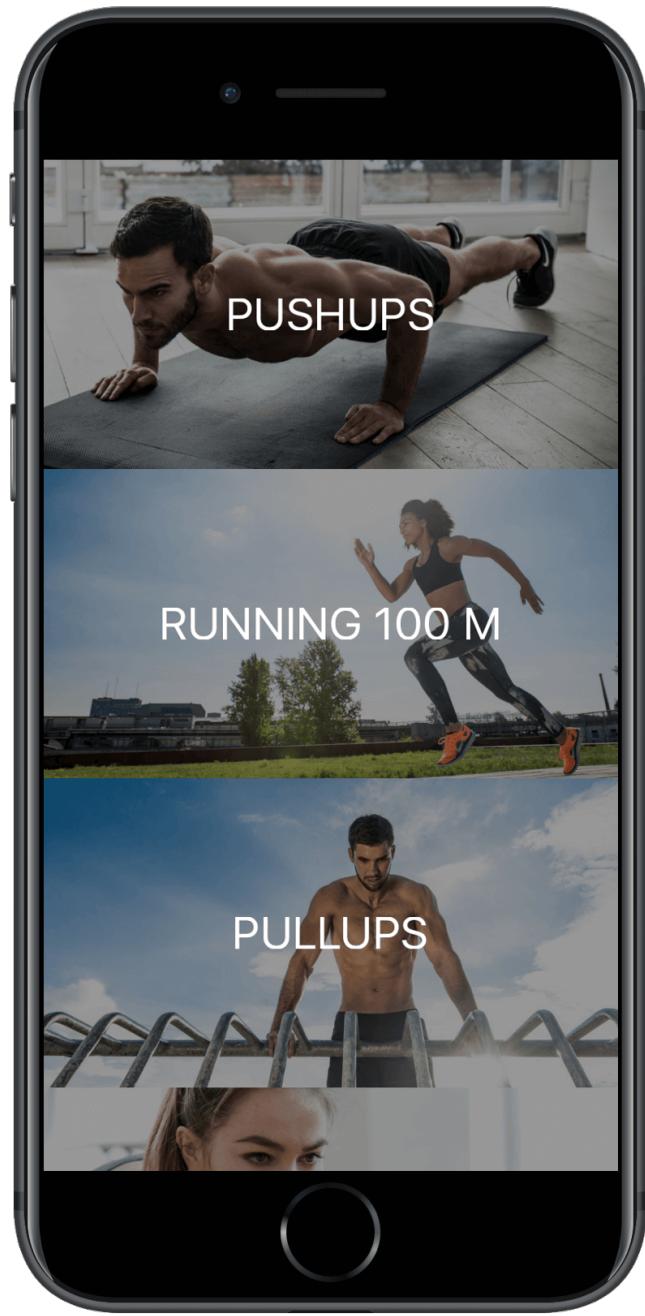
- Status bar is initially hidden : YES
- View controller-based status bar appearance : NO

Si ahora ejecutas la aplicación, verás como ya **no aparece** la status bar.

Para eliminar los separadores de nuestra tabla, selecciona la **TableView** en el **Main.storyboard** y en su propiedad **Separator**, elige **None**.



Para terminar, si **ejecutas la aplicación**, verás como el diseño se muestra tal y como queríamos.



Enhorabuena, has terminado un nuevo tutorial de EfectoApple.

8. Resumen Final

Espero que este tutorial te haya servido para dominar un poco más las **UITableViews**. Gracias a las **Celdas Personalizadas** podrás crear diseños completamente diferentes al resto de aplicaciones.

Aquí tienes un **pequeño resumen** de los puntos más importantes que hemos visto:

1. Como diseñar una aplicación utilizando **Celdas Personalizadas**
2. La forma de añadir **constraints** a los elementos de nuestra interfaz
3. Hemos creado una **clase** que gestione nuestra celda personalizada.
4. Como **añadir recursos** a nuestra aplicación
5. La forma de implementar correctamente los **métodos** de **UITableViewDelegate** y **UITableViewDataSource**
6. También hemos visto el **error más común** que se produce cuando trabajamos con UITableView
7. Como hacer que no se muestre la **status bar** en nuestra aplicación y como eliminar los **separadores** de nuestra **TableView**.

Domina el Framework UserNotifications con Swift [Parte 1]

Lenguaje Swift | Nivel Principiante

1. Introducción

Las Notificaciones son una opción muy utilizada en el Desarrollo de Aplicaciones iOS.

En esta ocasión vamos a centrarnos en las **Notificaciones Locales**.

Las notificaciones locales nos permiten **mantener** al usuario **informado** sobre determinados aspectos de nuestra aplicación. Sin embargo **no deben confundirse** con las notificaciones push.

En iOS existen dos **tipos** de notificaciones:

- Notificaciones Locales
- Notificaciones Push

La principal **diferencia** entre ellas es la siguiente:

Las notificaciones locales **no requieren** ningún tipo de **infraestructura** externa, ya que suceden directamente en el dispositivo iOS. Ni siquiera necesitan **conexión a internet** para funcionar.

Por otro lado las notificaciones push necesitan que el dispositivo disponga de conectividad y además requieren una infraestructura externa que permita enviar dichas notificaciones a través del **APNs** (Apple Push Notification Service).

En este tutorial nos centraremos en detalle en las **notificaciones locales**. Dentro de muy poco es probable que publique un nuevo tutorial donde también podremos ver el funcionamiento de las notificaciones push.

Comencemos entonces

2. ¿Qué vamos a ver en este Tutorial?

El tutorial se dividirá en 2 partes. Hoy veremos los **fundamentos** de las notificaciones locales y **desarrollaremos una aplicación** que ponga en práctica lo aprendido.

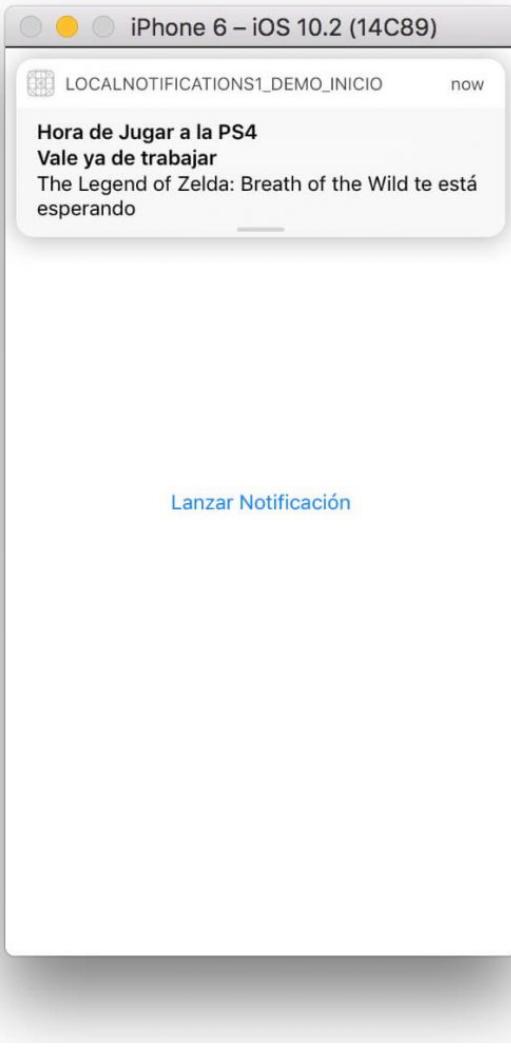
En la **segunda parte** de este tutorial, revisaremos aspectos más avanzados de las notificaciones locales.

Estos son los **puntos más importantes** en los que nos centraremos:

- Funcionamiento de las notificaciones locales
- ¿Qué es una Notification Request?
- Los 4 + 2 pasos a la hora de trabajar con notificaciones locales
- ¿Cómo crear el Trigger de una Request?
- ¿Cómo crear el contenido de una notificación?
- Añadiendo la request al centro de notificaciones
- Solicitando permiso al usuario
- El Protocolo UNUserNotificationCenterDelegate

3. La Aplicación que vamos a Desarrollar

La aplicación que vamos a desarrollar será una app **muy simple**. Contará con una interfaz de un **único botón**, que en el momento en que el usuario lo pulse, se creará y se **mostrará** en pantalla una notificación local.



De esta forma, veremos el proceso a través de un **caso práctico**. Así, cuando tengas que utilizar notificaciones locales en cualquier proyecto, no tendrás ningún problema.

Al igual que en otros tutoriales de EfectoApple, he creado un **Proyecto Inicial**, que nos permitirá centrarnos en las partes realmente importantes.

Sin embargo, antes de comenzar con el proyecto, tenemos que revisar de forma muy breve, algunos **conceptos** importantes.

4. Funcionamiento de las Notificaciones Locales

Si te fijas en el título del tutorial, verás que hemos hecho hincapié en la **versión 10** de iOS. Esto se debe a que con el lanzamiento de esta versión, Apple **renovó** la forma de trabajar con las Notificaciones Locales y ha añadido **algunas funciones** muy interesantes.

El framework ofrecido por Apple para que trabajemos con notificaciones en nuestras aplicaciones es **UserNotifications**.

Puedes acceder a la **documentación** de Apple sobre este Framework, [desde aquí](#).

La base de cualquier notificación es una **Request** (Petición). Veamos este concepto más en detalle.

Request

Cualquier desarrollador que pretenda **implementar** una notificación en su aplicación iOS deberá trabajar con requests.

La siguiente pregunta está clara.

¿Qué es una Request?

Una request consiste en una **solicitud** que enviamos a iOS, para que se muestre una notificación concreta. Cualquier request está formada por **2 componentes**:

- El Trigger de la notificación
- El Contenido de la notificación

Un **Trigger** es un conjunto de **condiciones** que deben cumplirse para que nuestra notificación sea lanzada. Podemos trabajar con diferentes tipos de triggers. Por ejemplo, podemos crear un **trigger temporal** que muestre una notificación en un momento concreto. También podemos crear un **trigger por localización**, que lance una notificación cuando el usuario llegue a una ubicación determinada. Esta es la idea básica de los triggers.

Por otro lado, tenemos el **contenido** de la aplicación, que no será más que una serie de **parámetros** que conformarán nuestra notificación. Estos parámetros son entre otros, el **título**, el **subtítulo**, el **cuerpo** y el **sonido** que emitirá la notificación cuando se muestre.

Una vez que comprendemos estos conceptos, veamos de forma resumida el **proceso completo** cuando tengamos que trabajar con notificaciones.

Los 4 + 2 Pasos a la hora de crear Notificaciones

Estos son los **4 puntos** que seguiremos siempre que trabajemos con notificaciones:

1. Crear el Trigger que disparará nuestra notificación
2. Crear el Contenido de la notificación
3. A partir del Trigger y del Contenido crearemos la Request
4. Añadir la Request al Centro de Notificaciones

Una vez que hemos completado este proceso, nuestra aplicación, de forma automática **lanzará la notificación**, en cuanto se cumplan las condiciones que hemos establecido en el trigger.

Además de estos 4 puntos, habría que añadir otros **2 adicionales** que también tendremos que tener en cuenta. Éstos los veremos **al final** del tutorial. Creo que será más claro comprender primero los 4 puntos principales y al terminar ver los últimos 2 que completan el proceso.

Ahora que ya hemos explicado la parte teórica, vamos a centrarnos, como siempre, en **desarrollar una aplicación iOS** donde podamos poner en práctica lo que acabamos de ver.

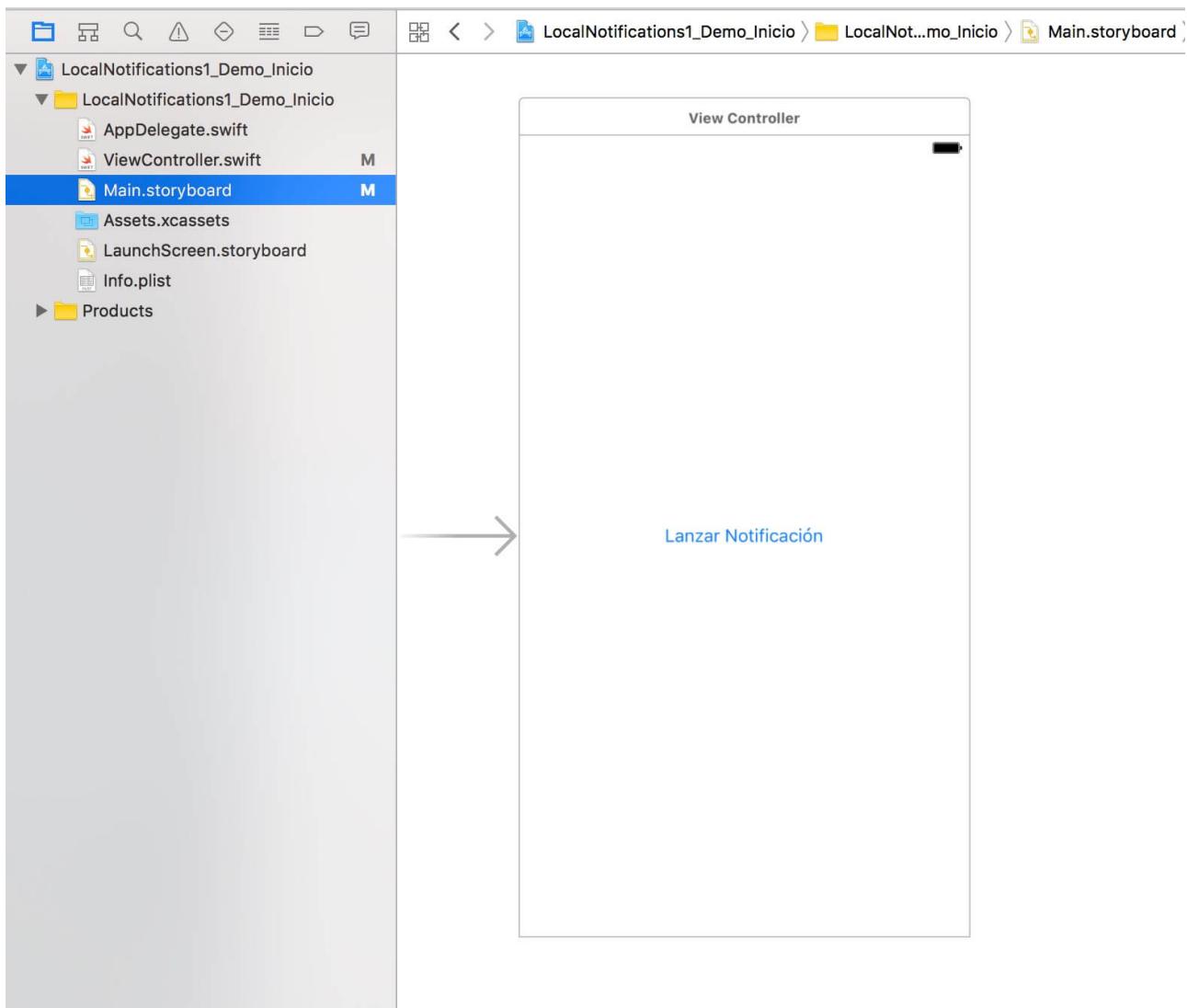
5. Nuestro Proyecto de Inicio

Para poder seguir el tutorial paso a paso lo primero que deberías hacer es descargar el **Proyecto de Inicio** [desde aquí](#).

Una vez que lo has bajado, descomprímelo y **ábrelo** con Xcode.

Se trata de un proyecto extremadamente sencillo.

Accede a *Main.storyboard* y verás que la interfaz consta de un botón con el texto **Lanzar Notificación**.



Este botón está conectado mediante un **IBAction** al siguiente método de *ViewController.swift*:

```
@IBAction func send10SecsNotification(){  
}
```

Como ves, el método **send10SecsNotification()** está vacío.

A lo largo de nuestro tutorial, **iremos completando** este método. El código que iremos añadiendo será el necesario para llevar a cabo el proceso de creación de una **notificación local** utilizando el framework UserNotifications.

Por el nombre del método, ya habrás deducido que la notificación que vamos a crear se lanzará **justo 10 segundos después** de que el usuario pulse en el botón de nuestra aplicación.

Veamos entonces todo el código paso a paso.

6. Creando el Trigger de la Request

Comenzaremos por el primero de los 4 pasos: La creación del Trigger.

Aunque antes, hay algo que debes añadir en tu aplicación, para poder trabajar con el framework **UserNotifications**.

Debemos **importar dicho framework** en nuestra clase. Para ello, añade justo antes de la cabecera de nuestra clase la siguiente linea:

De esta forma, ya podemos trabajar con cualquier **método o clase** incluida en dicho framework.

Ahora si, crearemos el **trigger** de nuestra notificación. Añade la siguiente linea al método **send10SecsNotification()**:

```
let trigger = UNTimeIntervalNotificationTrigger(timeInterval: 10.0, repeats: false)
```

Utilizando el método **UNTimeIntervalNotificationTrigger()**, podemos especificar que la notificación se lance justo 10 segundos después de la pulsación del botón. Además, como únicamente queremos una notificación y no queremos que se repita varias veces, en el parámetro **repeats**, pasaremos false.

Con una sola línea hemos creado el **trigger** la notificación.

Pasemos ahora al **siguiente punto** del proceso.

7. Creación del Contenido de la Notificación

Para configurar el contenido de la notificación usaremos un objeto de tipo **UNMutableNotificationContent**. Este objeto nos permitirá establecer los siguientes parámetros:

- Título
- Subtítulo

- Cuerpo
- Sonido

Para esto, añade el siguiente código al método **send10SecsNotification()** justo después del código del apartado anterior:

```
let content = UNMutableNotificationContent()
content.title = "Hora de Jugar a la PS4"
content.subtitle = "Vale ya de trabajar"
content.body = "The Legend of Zelda: Breath of the Wild te está esperando"
content.sound = UNNotificationSound.default()
```

No hace falta explicar nada más sobre este código. Unicamente comentar que hemos utilizado el **sonido de notificación** disponible **por defecto**.

Ahora que ya hemos creado el **trigger** y el **contenido** de la notificación, ya podemos crear la **Request**.

8. Construyendo la Request

Para nuestra request, crearemos un objeto de tipo **UNNotificationRequest**, formado por las constantes *trigger* y *content*.

Además tendremos que establecer un **identificador único** para esta request.

Añade la siguiente linea al método:

```
let request = UNNotificationRequest(identifier: "ZeldaNotification", content: content, trigger: trigger)
```

Sencillo, ¿no?

Hemos utilizado 3 parámetros:

- **Identifier:** Nos permite diferenciar esta request del resto en el

Centro de Notificaciones

- **Content:** El contenido de la notificación que hemos creado anteriormente
- **Trigger:** La condición que debe cumplirse para que se lance nuestra aplicación. En nuestro caso, que pasen 10 segundos.

Si recuerdas, los pasos que mencionamos en la parte teórica, sabrás que únicamente nos queda el **paso final**. Lo vemos a continuación.

9. Añadiendo la Request al Centro de Notificaciones

Lo último que tenemos que hacer para completar el proceso es **añadir la request al centro de notificaciones**.

El centro de notificaciones es el encargado de **gestionar** todas las notificaciones de tu aplicación. Será quien **compruebe** si las **condiciones** del trigger que hemos establecido se han cumplido y en caso afirmativo **lanzará** nuestra notificación.

Existe una **buena práctica** que debes tener en cuenta cuando añadas alguna request al centro de notificaciones y es la **siguiente**:

Antes de añadir una request al centro de notificaciones, es conveniente eliminar cualquier otra request existente

Seguir esta regla te permitirá **evitar** que se produzcan **notificaciones duplicadas**.

Este es el código que tienes que utilizar para **añadir la request** al centro de notificaciones:

```
UNUserNotificationCenter.current().removeAllPendingNotificationRequests()  
  
UNUserNotificationCenter.current().add(request) {(error) in  
  
    if let error = error {  
  
        print("Se ha producido un error: \(error)")  
  
    }  
}
```

La primera linea obtiene una instancia del centro de notificaciones y a través del método **removeAllPendingNotificationRequests()** elimina cualquier request pendiente.

Después añadimos la request al centro de notificaciones. En la llamada a este método, utilizamos un **completion handler**, que nos permite, si se produce un error, mostrarlo por consola.

Con esta última porción de código, habríamos terminado los 4 puntos que vimos en la parte teórica.

Aspecto Final del Método

El aspecto actual del método **send10SecsNotification()** debe ser este:

```
@IBAction func send10SecsNotification(){  
    // 1. Creamos el Trigger de la Notificación  
  
    let trigger = UNTimeIntervalNotificationTrigger(timeInterval: 10.0, repeats:  
false)  
  
    // 2. Creamos el contenido de la Notificación  
  
    let content = UNMutableNotificationContent()  
  
    content.title = "Hora de Jugar a la PS4"  
    content.subtitle = "Vale ya de trabajar"  
    content.body = "The Legend of Zelda: Breath of the Wild te está esperando"  
    content.sound = UNNotificationSound.default()  
  
    // 3. Creamos la Request  
  
    let request = UNNotificationRequest(identifier: "ZeldaNotification", content:  
content, trigger: trigger)  
  
    // 4. Añadimos la Request al Centro de Notificaciones
```

```
UNUserNotificationCenter.current().removeAllPendingNotificationRequests()  
  
UNUserNotificationCenter.current().add(request) {(error) in  
  
    if let error = error {  
  
        print("Se ha producido un error: \(error)")  
  
    }  
  
}  
  
}
```

10. Probando nuestra Aplicación

Ha llegado el momento de comprobar que el código que hemos creado **funciona correctamente**. Para ello, compila la aplicación y ejecútala.

Verás que se muestra por pantalla el botón que lanza la notificación. **Púlsalo** y espera 10 segundos.

¿Qué ocurre?

La aplicación **no muestra ningún tipo de notificación**.

Si recuerdas, habíamos comentado que aparte de los 4 puntos que había que seguir al trabajar con notificaciones, existían otros **2 adicionales** que debíamos tener en cuenta.

Esos 2 puntos son los siguientes:

- Solicitar **permiso** al usuario
- Implementar el Protocolo **UNUserNotificationCenterDelegate**

Veamos en qué consisten estos 2 puntos.

11. Solicitando permiso al usuario

Apple siempre ha tenido un cuidado extremo en preservar la **experiencia**

de usuario en iOS. Por este motivo, ofrece **control total** a sus usuarios con respecto a las **notificaciones** que reciben de las aplicaciones instaladas en sus dispositivos.

Es decir, si queremos poder enviar notificaciones a un usuario, necesitamos que **nos de permiso explícito**.

Por tanto, ¿cómo solicitamos permiso al usuario?

Utilizando el método **requestAuthorization()**.

Vamos a solicitar este permiso al usuario, nada más arrancar la aplicación, por tanto deberás hacer dos cosas en la clase *AppDelegate.swift* de tu proyecto.

Importar el framework UserNotifications

Solicitar Permiso

Deberás añadir el siguiente código al método **didFinishLaunchingWithOptions()**:

```
func application(_ application: UIApplication,  
didFinishLaunchingWithOptions launchOptions:  
[UIApplicationLaunchOptionsKey: Any]?) -> Bool {  
  
    UNUserNotificationCenter.current().requestAuthorization(options: [.alert,  
.sound]) {(accepted, error) in  
  
        if !accepted {  
  
            print("Permiso denegado por el usuario")  
  
        }  
  
    }  
  
    return true  
}
```

Lo que hemos hecho es solicitar permiso al usuario para mostrarle notificaciones utilizando el método **requestAuthorization()**.

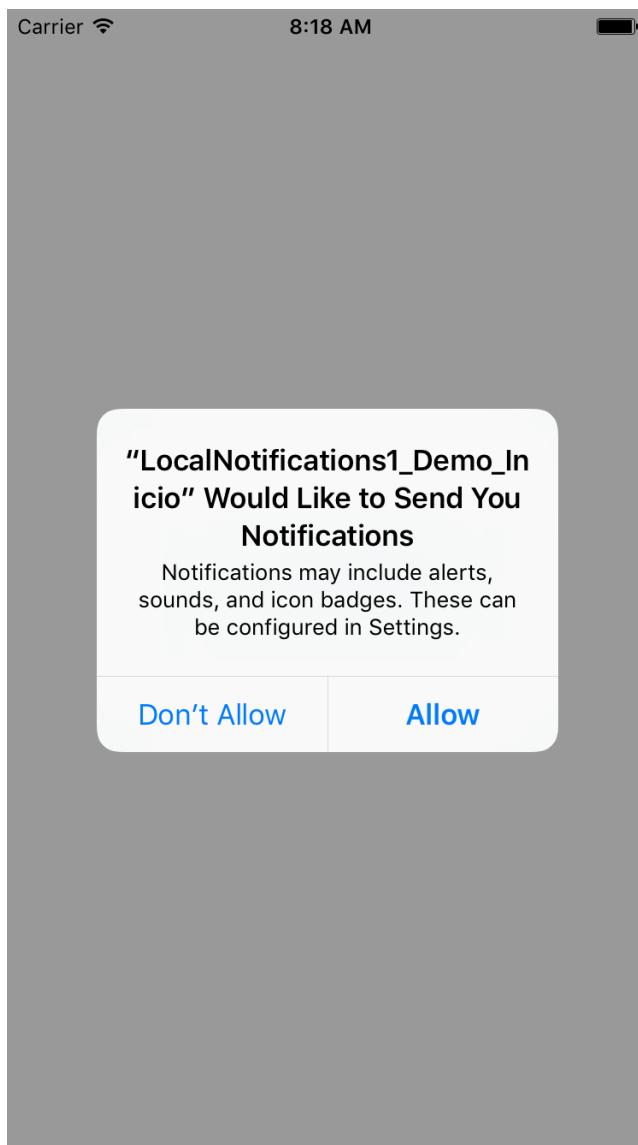
En caso de que el usuario no nos de permiso mostraremos por consola un **mensaje** indicándolo.

Lo que vamos a hacer ahora es **volver a probar** nuestra aplicación.

12. Probando por segunda vez nuestra Aplicación

Compila y ejecuta el proyecto.

Al lanzar la aplicación, verás que lo primero que se muestra es **la solicitud de permiso**.



Pulsa en el botón **Allow** para permitir a la aplicación recibir notificaciones.

Después pulsa el botón y espera 10 segundos.

Tampoco ha sucedido nada.

Sin embargo, si quieras ver funcionando nuestra notificación, sigue **el siguiente proceso:**

- Pulsa de nuevo en el botón
- Accede al menú **Hardware-Home** del Simulador (Tienes que hacerlo en menos de 10 segundos)



- Espera un poco y verás como **se muestra** perfectamente nuestra notificación



Entonces, ¿qué es lo que ocurre?

Simplemente, nuestra aplicación actual solamente puede **mostrar notificaciones** cuando la app se encuentra **en segundo plano** (Es decir, cuando el usuario no la tiene abierta).

Siempre que el usuario la tenga abierta (**En primer plano**), no recibirá ninguna notificación.

Esto es algo que no nos interesa. Queremos que el usuario reciba nuestras notificaciones, **tanto si la aplicación está abierta como si no**.

Por este motivo, tenemos que llevar a cabo el último paso de todos, **implementar el protocolo**.

13. Implementando el Protocolo

Vamos entonces a trabajar con el protocolo **UNUserNotificationCenterDelegate**.

Antes de continuar, si no estás familiarizado con los conceptos de **Delegado** y **Protocolo**, te recomiendo que eches un vistazo a estas explicaciones:

- Concepto de [Delegado](#)
- Concepto de [Protocolo](#)

Estos dos conceptos son fundamentales en Desarrollo iOS.

Para implementar el protocolo **UNUserNotificationCenterDelegate** haremos 3 cosas:

1. Especificar que nuestra clase *ViewController.swift* **se ajusta** a dicho protocolo
2. Especificar que la clase *ViewController.swift* actuará como **delegado** del Centro de Notificaciones
3. Implementar el método que **presentará la notificación** cuando la app esté en primer plano

Para ajustar nuestra clase a dicho protocolo, simplemente tendremos que añadir en su **cabecera** el **nombre** del protocolo. Por tanto, modificaremos la cabecera de la clase *ViewController.swift* para que tenga este aspecto:

```
class ViewController: UIViewController, UNUserNotificationCenterDelegate {  
    ...  
}
```

Después, Para especificar que *ViewController.swift* será **delegado** del Centro de Notificaciones, añadiremos la siguiente linea en el método **viewDidLoad()**:

```
UNUserNotificationCenter.current().delegate = self
```

Por último, implementaremos el método **willPresent()** que será llamado, cuando nuestra aplicación esté en primer plano y la notificación sea lanzada:

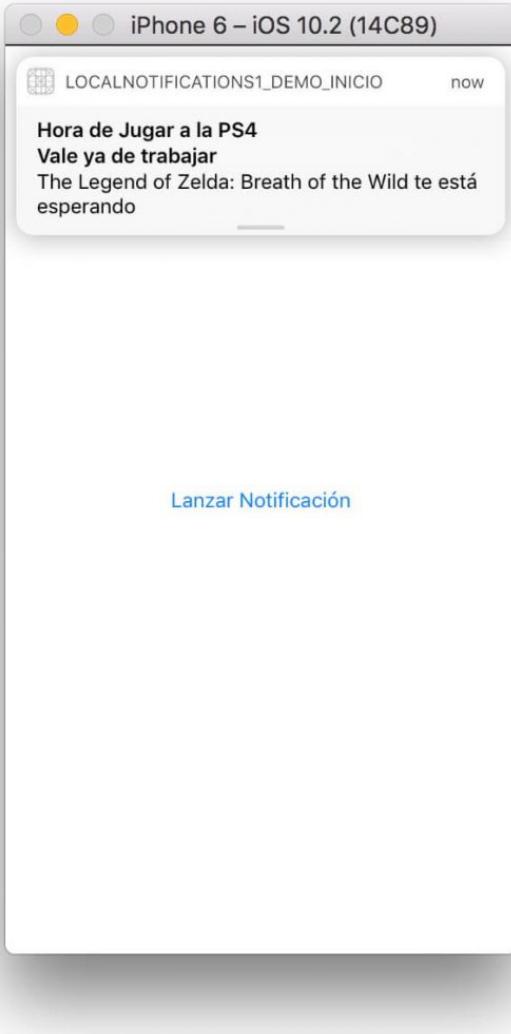
```
func userNotificationCenter(_ center: UNUserNotificationCenter, willPresent notification: UNNotification, withCompletionHandler completionHandler: @escaping (UNNotificationPresentationOptions) -> Void) {  
    completionHandler([.alert, .sound])  
}
```

En este método lo único que hacemos es utilizar el completion handler para especificar que la notificación **deberá mostrarse** y además emitir el sonido por defecto que utilizamos al definir el content.

Después de haber realizado estos cambios, **prueba** a ejecutar **de nuevo** la aplicación.

Pulsa en el botón central y espera 10 segundos. Ya no es necesario que vayas a la pantalla Inicio, puedes permanecer con la aplicación **en primer plano**.

Y ahora... si que se muestra **correctamente** la notificación.



14. Resumen Final

Como ves, **de forma muy sencilla**, hemos sido capaces de configurar la aplicación de este tutorial para que muestre **notificaciones locales** a nuestros usuarios.

Si repasas el proyecto completo verás que no hemos utilizado mas que **unas pocas líneas** de código.

Espero que este tutorial te pueda servir **siempre** que necesites utilizar **notificaciones locales** en tus aplicaciones iOS.

Domina el Framework UserNotifications con Swift [Parte 2]

Lenguaje Swift | Nivel Principiante

1. Introducción

En el tutorial anterior revisamos los conceptos fundamentales sobre **notificaciones locales**.

Pudimos ver como con el lanzamiento de **iOS 10**, los desarrolladores iOS habíamos tenido que cambiar la forma de trabajar con esta funcionalidad.

Vimos que la **base** de todo se encuentra en el framework [UserNotifications](#).

Repasamos los **conceptos teóricos** mas importantes y sobre todo, pusimos en práctica lo aprendido desarrollando una **aplicación** muy sencilla que lanzaba una **notificación local**, 10 segundos después de que el usuario pulsara un botón.

Considero que el framework [UserNotifications](#) ofrece **muchas posibilidades**, por lo que vamos a continuar con la **segunda parte** de ese tutorial.

Veremos **aspectos más avanzados** de las notificaciones locales que te permitirán incorporar algunos aspectos muy útiles para los usuarios de tus aplicaciones.

Continuemos entonces con la **segunda parte** del tutorial sobre notificaciones locales.

2. ¿Qué vamos a ver en este Tutorial?

Estos son los **puntos más importantes** en los que nos centraremos en la

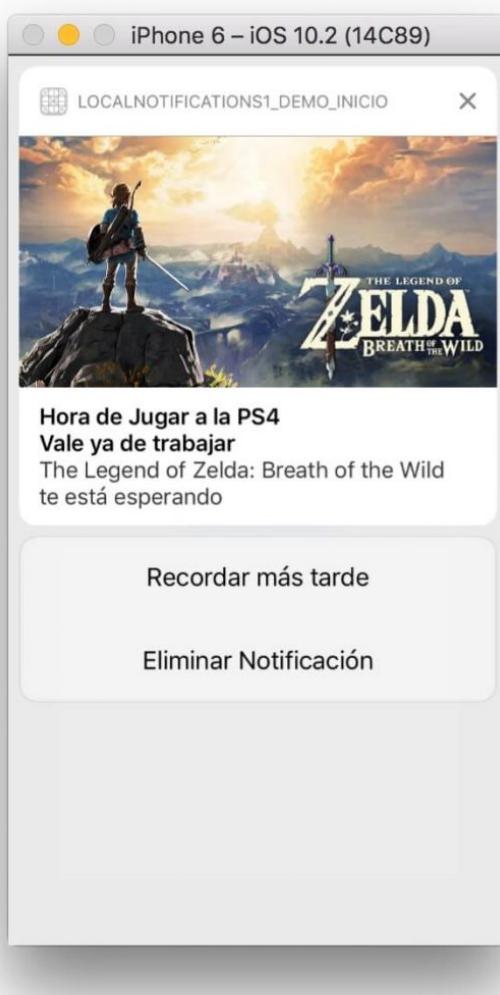
segunda parte del tutorial:

- Añadir imágenes a nuestras notificaciones
- Trabajando con Acciones y Categorías
- Como añadir Acciones a una Notificación
- Añadiendo funcionalidad a nuestras acciones
- Implementando el protocolo UNUserNotificationCenterDelegate

3. La Aplicación que vamos a Desarrollar

Vamos a **continuar** desarrollando la aplicación que comenzamos en el anterior tutorial. **Mejoraremos** la notificación que creamos en la primera parte del tutorial, añadiéndole una **imagen** y **acciones** que el usuario podrá realizar.

Este será el **aspecto** de la aplicación lanzando la notificación:



4. Nuestro Proyecto de Inicio

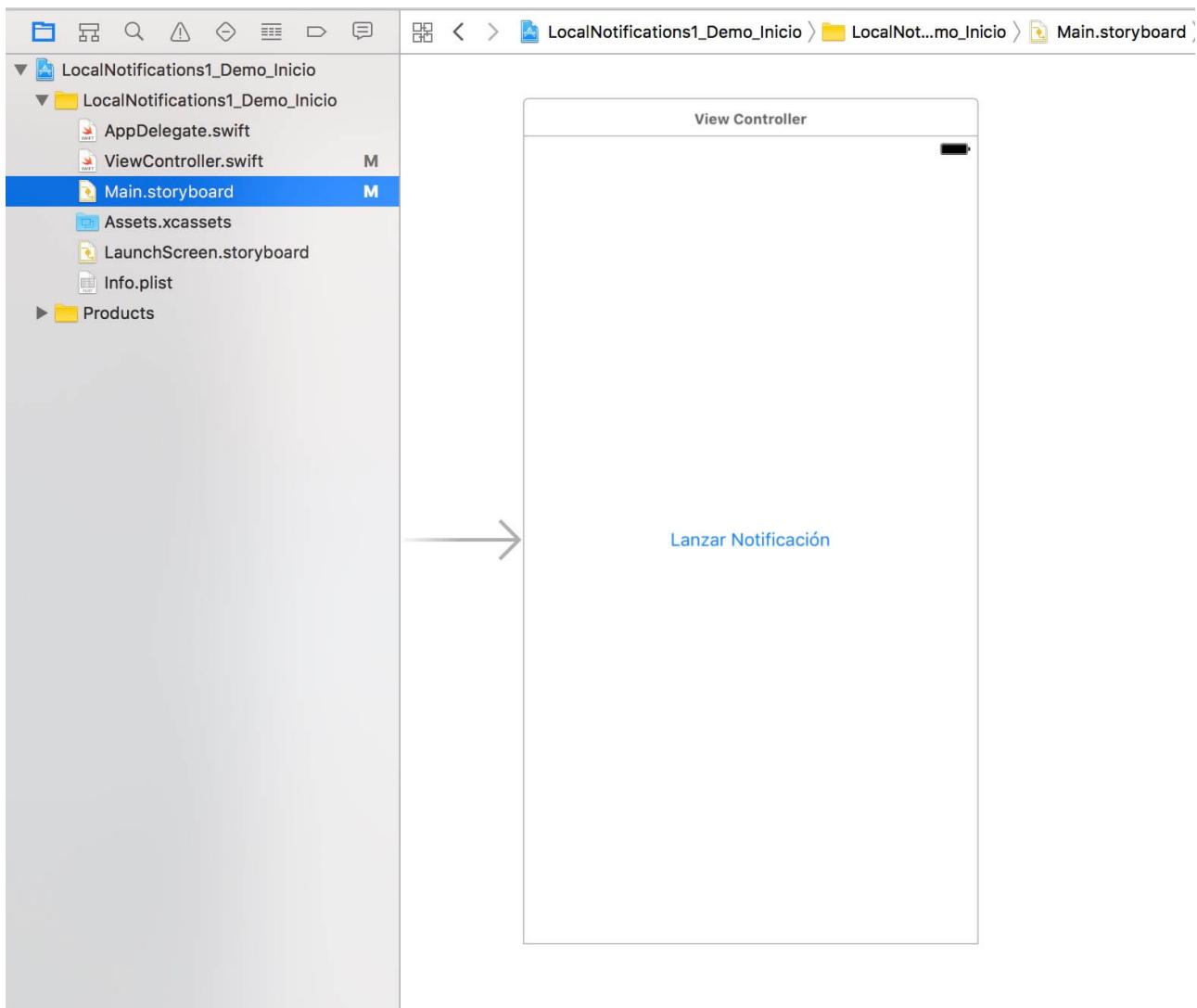
Como se trata de la **segunda parte** de nuestro tutorial sobre notificaciones locales, no vamos a tener un proyecto de inicio sino que **vamos a continuar** con la aplicación que desarrollamos en el apartado anterior.

Por tanto, tu primer paso debe ser descargar el proyecto [desde aquí](#).

Una vez que lo has bajado, descomprímelo y **ábrelo** con Xcode.

Se trata de un proyecto extremadamente **sencillo**.

Accede a *Main.storyboard* y verás que la interfaz consta de un botón con el texto **Lanzar Notificación**.



Abre la clase *ViewController.swift* y verás los dos métodos en los que trabajamos en el anterior tutorial:

```

func userNotificationCenter(_ center: UNUserNotificationCenter, willPresent notification:
    UNNotification, withCompletionHandler completionHandler: @escaping
    (UNNotificationPresentationOptions) -> Void) {
    completionHandler([.alert, .sound])
}

@IBAction func send10SecsNotification(){
    // 1. Creamos el Trigger de la Notificación
    let trigger = UNTimeIntervalNotificationTrigger(timeInterval: 10.0, repeats: false)

    // 2. Creamos el contenido de la Notificación
    let content = UNMutableNotificationContent()
    content.title = "Hora de Jugar a la PS4"
    content.subtitle = "Vale ya de trabajar"
    content.body = "The Legend of Zelda: Breath of the Wild te está esperando"
    content.sound = UNNotificationSound.default()

    // 3. Creamos la Request
    let request = UNNotificationRequest(identifier: "ZeldaNotification", content: content, trigger:
        trigger)

    // 4. Añadimos la Request al Centro de Notificaciones
    UNUserNotificationCenter.current().removeAllPendingNotificationRequests()
    UNUserNotificationCenter.current().add(request) {(error) in
        if let error = error {
            print("Se ha producido un error: \(error)")
        }
    }
}

```

Si quieres puedes **ejecutar** la aplicación para comprobar que funciona perfectamente.

Comencemos entonces **echando un vistazo** al proceso que debemos seguir para añadir una imagen a nuestra notificación local.

5. Añadiendo imagen a Notificaciones Locales

Ahora mismo, nuestra aplicación muestra una notificación que **únicamente** consta de **texto**.

Vamos a realizar algunas modificaciones para que **además muestre una imagen** al usuario.

Añadiendo la imagen a nuestra Aplicación

El primero paso será **añadir la imagen** que vamos a utilizar, a nuestro proyecto.

Puedes añadir la imagen que prefieras.

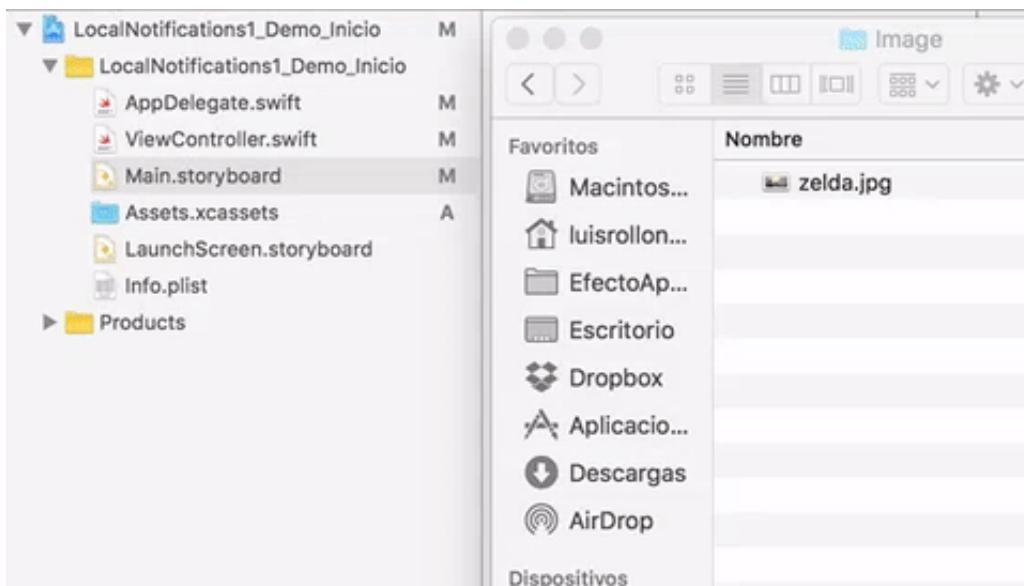
En el caso de que quieras seguir los mismos pasos que yo, puedes **descargar la imagen** que voy a usar [desde aquí](#).

La imagen que voy a utilizar se basa en el universo Zelda. He utilizado esta temática para el tutorial porque es probablemente mi saga favorita de videojuegos.

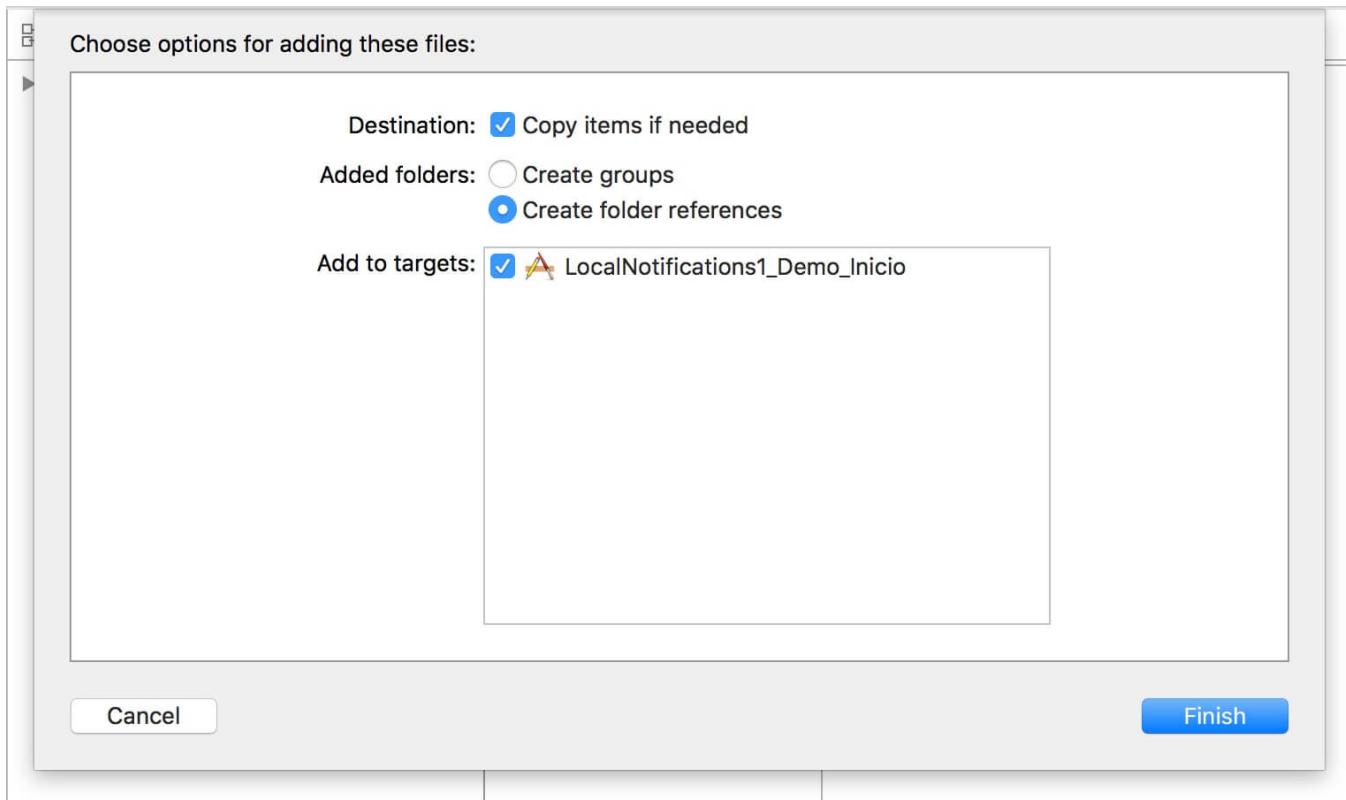
Nota Legal: The Legend of Zelda es una marca registrada de **NINTENDO Co. Ltd.** El uso de su marca en este Tutorial obedece únicamente a propósitos didácticos.

Una vez que tienes la imagen que vas a utilizar, el siguiente paso es **añadirla** al proyecto.

Para ello, aunque podríamos añadirla a la carpeta de **Assets**, vamos a hacerlo más sencillo, simplemente **arrastrándola** a nuestro proyecto directamente.



Una vez arrastrada la imagen, **Xcode nos preguntará** si queremos añadir la imagen al proyecto:



Cuando quieras añadir **cualquier imagen** a tus proyectos deberás asegurarte que estas dos opciones estén activadas:

- Copy items if needed
- Add to targets

Para terminar pulsa en el botón **Finish**.

Una vez que hemos añadido nuestra imagen al proyecto, **veamos como mostrarla** en la notificación.

Mostrando la imagen en la Notificación

El framework UserNotification nos ofrece la clase **UNNotificationAttachment** para que podamos añadir elementos a nuestra notificación.

Estos elementos puede ser **imágenes, sonidos y videos**.

Esta clase soporta una gran **variedad de formatos**. Puedes consultar los formatos disponibles [accediendo aquí](#).

En nuestro caso, vamos a añadir la imagen **zelda.jpg**.

Para ello, tendremos que crear una instancia de **UNNotificationAttachment** y añadirla a la propiedad **attachments** del objeto **content**.

Si recuerdas la primera parte del tutorial, el objeto content nos permitía añadir el **título**, **subtítulo** y **contenido** a la aplicación. Ahora también lo usaremos para añadir la **imagen**.

Por tanto, para añadir la imagen, tendrás que escribir el siguiente código en el método **send10SecsNotification()** justo antes de la parte en la que creamos el objeto request:

```
if let path = Bundle.main.path(forResource: "zelda", ofType: "jpg") {  
    let url = URL(fileURLWithPath: path)  
    do {  
        let attachment = try UNNotificationAttachment(identifier: "zelda", url: url,  
options: nil)  
        content.attachments = [attachment]  
    } catch {  
        print("La imagen no se ha cargado")  
    }  
}
```

Lo único que hemos hecho aquí ha sido convertir **la ruta de nuestra imagen** en un objeto de tipo **URL**, para después añadirlo como adjunto (**UNNotificationAttachment**) a nuestro objeto content.

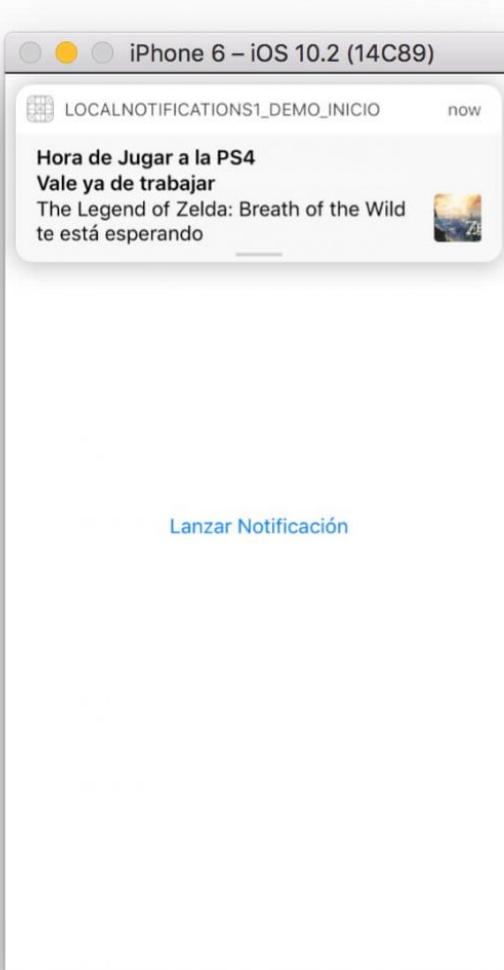
También hemos tenido que utilizar un bloque **catch**, debido a que el inicializador de **UNNotificationAttachment** puede producir **excepciones**.

Probando la App

Después de añadir este código ha llegado el momento de **comprobar** si la notificación muestra la imagen por pantalla.

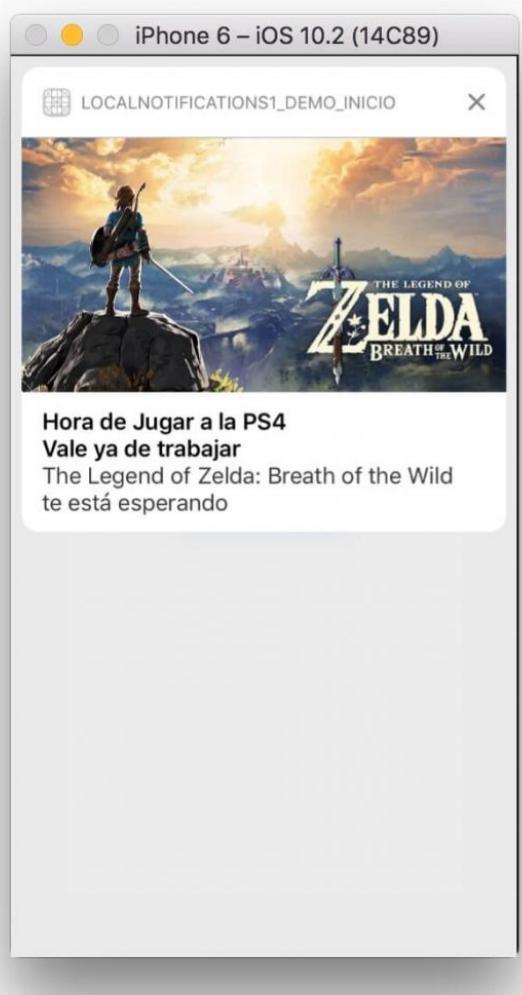
Para ello, ejecuta la aplicación en el **simulador** y **pulsa** en el **botón central** que lanza la notificación.

Espera 10 segundos y...



Como has podido comprobar, se muestra la **notificación** junto con una **miniatura** en la parte derecha de nuestra imagen.

Además, si **deslizas** la **notificación** hacia **abajo**, verás como la imagen se muestra en un **tamaño mayor**.



Genial, ¿no?

La característica de **poder añadir imágenes** a las notificaciones fue introducida por Apple en el lanzamiento de **iOS 10**.

Ahora ya sabes como utilizarla para tus propias **notificaciones locales**.

Veamos que **más opciones** nos ofrece el framework UserNotifications.

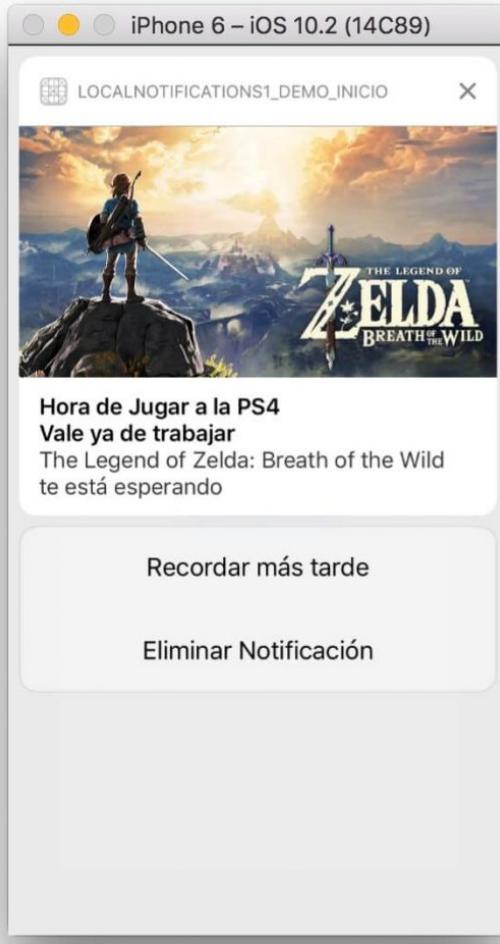
6. Como añadir Acciones a una Notificación Local

Además de añadir imágenes a una notificación local, también podemos **añadir acciones**.

Estas acciones permiten al usuario seleccionar algunas **opciones relacionadas** con la **notificación**.

Por si no tienes claro a que me estoy refiriendo con **ACCIONES**, aquí

tienes una imagen de la notificación creada por nosotros, después de añadirle las acciones de **Recordar** y **Eliminar**:



Acciones y Categorías

Para poder añadir estas acciones a la notificación, tendremos que trabajar con:

- **Acciones** (Objetos de tipo UNNotificationAction)
- **Categorías** (Objetos de tipo UNNotificationCategory)

Está claro, que si queremos añadir **2 acciones diferentes** a una notificación, tendremos que crear un objeto de tipo UNNotificationAction **para cada una** de ellas.

Pero además, tendremos que crear un objeto de tipo **UNNotificationCategory** que englobe a ambas.

Debes pensar en las **Categorías**, como objetos que representan un **grupo de acciones**.

Una categoría puede estar formada por **1..N acciones**.

En nuestro caso, la categoría estará formada por **2 acciones**:

- Recordar
- Eliminar

Veamos entonces, como **crear** nuestras **2 acciones** y la **categoría** que las agrupa.

Creando 2 acciones y una Categoría

Para ello, añade el siguiente código al método **send10SecsNotification()**, justo antes del bloque de código del apartado anterior:

//1. Creamos la acción de Recordar

```
let rememberAction = UNNotificationAction(identifier: "rememberAction",  
title: "Recordar más tarde", options: [])
```

//2. Creamos la acción de Eliminar

```
let deleteAction = UNNotificationAction(identifier: "deleteAction", title:  
"Eliminar Notificación", options: [])
```

//3. Creamos la categoría que agrupa las acciones

```
let category = UNNotificationCategory(identifier: "zeldaCategory", actions:  
[rememberAction, deleteAction], intentIdentifiers: [], options: [])
```

//4. Añadimos la categoría al Centro de Notificaciones

```
UNUserNotificationCenter.current().setNotificationCategories([category])
```

Como puedes ver, hemos creado **2** objetos de tipo **UNNotificationAction**, asignándoles un **identificador** y un **título**.

Después hemos creado un objeto de tipo **UNNotificationCategory**, al que le hemos asignado también un **identificador** y las **2 acciones** que lo

componen.

Por último hemos añadido esta categoría al **Centro de Notificaciones**.

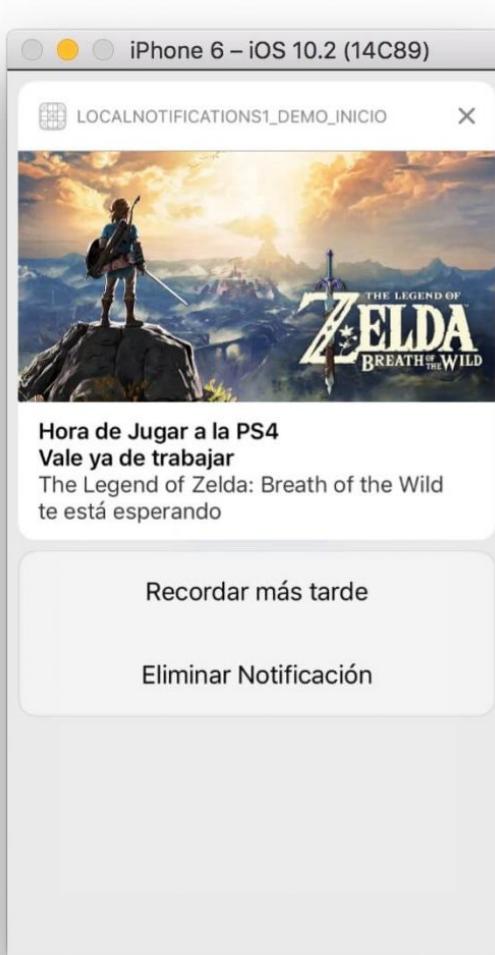
El último paso que tenemos que dar es **añadir** esta **categoría** al objeto **content** de nuestra notificación.

Para ello, **añade** a continuación del bloque anterior, **la siguiente linea**:

```
content.categoryIdentifier = "zeldaCategory"
```

Esto nos permite especificar que vamos a usar la categoría con nuestra notificación.

Prueba ahora a **ejecutar** la aplicación. **Pulsa** en el **botón** y cuando se lance la notificación, tira de ella hacia abajo. Deberías ver exactamente esto:



Como puedes ver, con solamente **5 líneas de código** hemos sido capaces de añadir **2 acciones** a nuestra notificación.

El problema es que si pulsas cualquiera de esas 2 notificaciones, lo único que sucede es que la **notificación desaparece**.

En el siguiente apartado veremos como escribir el código para que las acciones **realicen sus funciones**.

7. Añadiendo Funcionalidad a nuestras Acciones

Si recuerdas, en la primera parte del tutorial, en uno de los apartados finales, tuvimos que implementar un método del protocolo **UNUserNotificationCenterDelegate**.

Concretamente tuvimos que hacerlo para que la notificación se mostrase cuando nuestra aplicación estaba **ejecutándose en primer plano**.

Para añadir funcionalidad a las acciones que hemos creado en la notificación, tendremos que hacer **lo mismo**.

Implementaremos **otro método** que nos ayudará a gestionar las acciones que hemos creado.

El nombre completo del método que usaremos es este:

- `userNotificationCenter(_:didReceive:withCompletionHandler:)`

Cuando el usuario pulse alguna de las 2 acciones, **se ejecutará este método**.

Lo que haremos en este método es **comprobar que acción** ha pulsado el usuario. Podemos saber esto gracias al **identificador** de cada acción, que se encuentra en el parámetro `response`.

En el caso de que el usuario pulse en la opción **Recordar más tarde**, haremos que la notificación **se retrase 60 segundos**. Es decir, justo 1 minuto después de que el usuario pulse en recordar mas tarde, la

notificación volverá a mostrarse. Para hacer esto tendremos que hacer **algunos cambios** en el código que veremos a continuación.

Por otro lado, si el usuario pulsa en la opción **Eliminar Notificación**, lo que haremos obviamente será eliminar la notificación del Centro de Notificaciones.

Veamos ahora los cambios en el código que debemos hacer para implementar la opción **Recordar más tarde**.

Modificando nuestro Código

Como acabamos de decir, cuando el usuario pulse en el botón **Recordar más tarde**, tendremos que volver a lanzar la notificación pero esta vez con un retardo de 60 segundos.

Para esto, necesitaremos **una variable que controle el tiempo de retardo**.

Añade la siguiente **variable** justo al comienzo de la clase ViewController.swift:

```
var timeToRemember: Double = 10
```

También tendremos que **modificar la primera linea** del método **send10SecsNotification()**.

Ahora mismo, tenemos esto:

```
let trigger = UNTimeIntervalNotificationTrigger(timeInterval: 10, repeats: false)
```

Y debes sustituirlo por esta otra linea:

```
let trigger = UNTimeIntervalNotificationTrigger(timeInterval: timeToRemember, repeats: false)
```

Como ves, únicamente hemos cambiado el valor 10 por la variable **timeToRemember** que acabamos de crear.

Una vez realizado este cambio, ya podemos trabajar en el método del protocolo **UNUserNotificationCenterDelegate**.

Este sería el **método completo** que deberías añadir.

Lo puedes colocar justo antes del método **send10SecsNotification()**, en tu clase ViewController.swift:

```
func userNotificationCenter(_ center: UNUserNotificationCenter, didReceive response: UNNotificationResponse, withCompletionHandler completionHandler:@escaping () -> Void) {  
    //1. Si el usuario pulsa en Recordar más tarde, incremento el tiempo a 60 segs.  
    if response.actionIdentifier == "rememberAction" {  
        timeToRemember = 60  
        send10SecsNotification()  
    }  
    //2. En cambio si pulsa en Eliminar Notificación, la borramos  
    }else if response.actionIdentifier == "deleteAction" {  
  
        UNUserNotificationCenter.current().removeDeliveredNotifications(withIdentifier: ["ZeldaNotification"])  
    }  
}
```

El código de este método ya lo hemos explicado antes, por tanto **no es necesario** volver a repetir su funcionamiento.

Ejecutando la Aplicación

Una vez realizadas estas modificaciones, ha llegado el momento de **probar** que la notificación con acciones que hemos creado **funciona** perfectamente.

Ejecuta la aplicación en el simulador y cuando arranque **pulsa el botón** para lanzar la notificación.

En el momento en que aparezca la notificación en pantalla, **deslízala hacia abajo y pulsa** en el botón **Recordar más tarde**.

Ahora tendrás que **esperar 60 segundos** para comprobar que la notificación se retrasa un minuto, tal y como esperamos.

Después de que pasen esos 60 segundos, verás como la notificación **se muestra por segunda vez**. Justo lo que queríamos.

8. Resumen Final

En la **segunda parte** de nuestro tutorial hemos repasado **opciones avanzadas** que te permitirán utilizar **notificaciones locales** mucho más completas en tus aplicaciones.

Has comprobado por ti mismo, que podemos hacer esto de forma **muy sencilla**.

Espero que este tutorial te pueda servir **siempre** que necesites utilizar notificaciones locales en tus aplicaciones iOS.

Pantalla de Login con Facebook usando Firebase [Parte 1]

Desarrolla un Login con Facebook utilizando Swift

Lenguaje Swift | Nivel Avanzado

1. Introducción

En este **tutorial** vamos a ver como hacer una pantalla de **Login con Facebook**.

En uno de los tutoriales anteriores, vimos como crear una pantalla de registro/inicio de sesión para una aplicación iOS, usando **Swift** y **Firebase**.

Ese tutorial nos sirvió como **introducción** a Firebase y pudimos revisar algunos conceptos básicos.

En aquel apartado veíamos como crear de forma sencilla una **pantalla** donde los usuarios de nuestra **aplicación iOS** podían registrarse e **iniciar sesión** utilizando su email.

Existen muchísimos **sistemas** de Inicio de Sesión:

- Mediante Email
- Utilizando LinkedIn
- A través de Facebook
- Usando tu cuenta de Google

Antes de desarrollar la pantalla de **registro/inicio de sesión** deberías analizar a los usuarios potenciales de tu aplicación.

Esto te servirá para preguntarte que sistema de inicio de sesión es **el más conveniente** para ellos. Puede que llegues a la conclusión que

únicamente necesitas añadir un par de opciones o tal vez te des cuenta que es vital el ofrecerles todos los sistemas posibles.

En el tutorial anterior sobre Firebase, ya vimos como ofrecer la posibilidad a nuestros **usuarios** de **registrarse** utilizando su **email**.

En este tutorial, vamos a revisar como hacer que puedan acceder a nuestra aplicación utilizando **su cuenta de Facebook**.

Actualmente, Facebook cuenta con unos **1.900 millones de usuarios**. Con estas cifras es evidente que puede sernos muy útil el ofrecer esta opción de inicio de sesión a cualquiera que quiera utilizar nuestra aplicación.

2. ¿Qué vamos a ver en este Tutorial?

Lo primero de todo, si no echaste un vistazo a nuestro **primer tutorial sobre Firebase**, te recomiendo que lo hagas.

En este tutorial veremos como crear una **pantalla de login con Facebook utilizando Firebase**.

Es importante que tengas en cuenta que este tutorial requiere bastantes **pasos de configuración** antes de programar nuestra app.

Los **pasos** de configuración que vamos a seguir **son estos**:

- Configurar la App que vamos a crear, en [la web de Facebook para desarrolladores](#)
- Crear el proyecto en Firebase
- Activar la autenticación en Firebase
- Configurar nuestro proyecto en Xcode

Una vez que hayamos seguido estos **pasos previos**, estaremos preparados para programar la aplicación.

Ya has visto entonces, que tienes que tener algo de **paciencia**. Sin embargo, te garantizo que este proceso de configuración es **obligatorio** para llevar a cabo el proyecto.

Para evitar un tutorial demasiado extenso, **lo dividiremos en 2 partes**. Hoy veremos la primera parte y después trabajaremos en la segunda parte del tutorial.

3. La App que vamos a crear: Login con Facebook

En lugar de una app completa, vamos a crear el **punto de entrada** a cualquier aplicación que gestione usuarios.

Vamos a crear una pantalla de login que enlazará con **Firebase** y nos permitirá gestionar el **login con Facebook**.

Al terminar tendrás una **aplicación** como esta:



Y lo que es mejor, podrás utilizar todo lo que has visto en este tutorial para **crear pantallas de login con Facebook** para tus propias aplicaciones.

Para poder centrarnos en lo realmente importante del tutorial: La **integración con Firebase** y gestión de **login con Facebook**, he creado un **proyecto inicial**, donde encontrarás la interfaz ya creada.

Trabajaremos a partir de este proyecto y **programarás** la parte de gestión de usuarios.

En el **Apartado 5** tendrás disponible el **enlace** de descarga para el proyecto de inicio.

4. ¿Qué es Firebase?

Aunque ya lo explicamos en el tutorial anterior, vamos a revisar muy rápidamente en qué consiste **Firebase**.

Firebase es un **MBAaaS** (Mobile Backend as a Service). Es decir, se trata de una herramienta que nos ofrece un conjunto de características que conforman un **backend completo** sin necesidad de desarrollar el propio backend desde cero.

Entre estas características destacan:

- Base de Datos con actualizaciones en tiempo real
- Autenticación de usuarios
- Almacenamiento en la nube
- Notificaciones Push
- Integración con redes sociales
- Analíticas

Con Firebase puedes crear **aplicaciones completas** sin tener que escribir **ni una sola línea de código** del lado del **servidor**.

5. Nuestro Proyecto de Inicio

Como ya te he comentado, para poder centrarnos en las **partes**

importantes del Tutorial, he preparado un **proyecto de inicio**, que utilizaremos como base para el desarrollo de la aplicación.

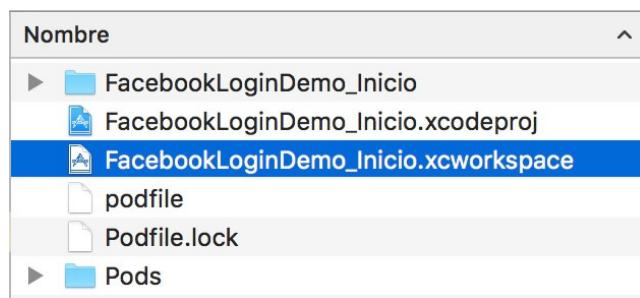
Puedes **descargar** el proyecto de inicio [desde aquí](#).

Una vez que lo hayas descargado, descomprímelo y accede a la carpeta *FacebookLoginDemo_Inicio*, ahí verás el **contenido** del proyecto de inicio.

Integración con Cocoapods

Para que puedas desarrollar esta aplicación de la forma más sencilla posible, el proyecto de inicio ya incorpora **Cocoapods**.

Recuerda, que para abrir un proyecto que utiliza Cocoapods, en lugar de hacer doble click en el fichero *.xcodeproj*, deberás abrir el *.xcworkspace*.



Una vez que lo has abierto, revisemos el proyecto.

Revisando el Proyecto de Inicio

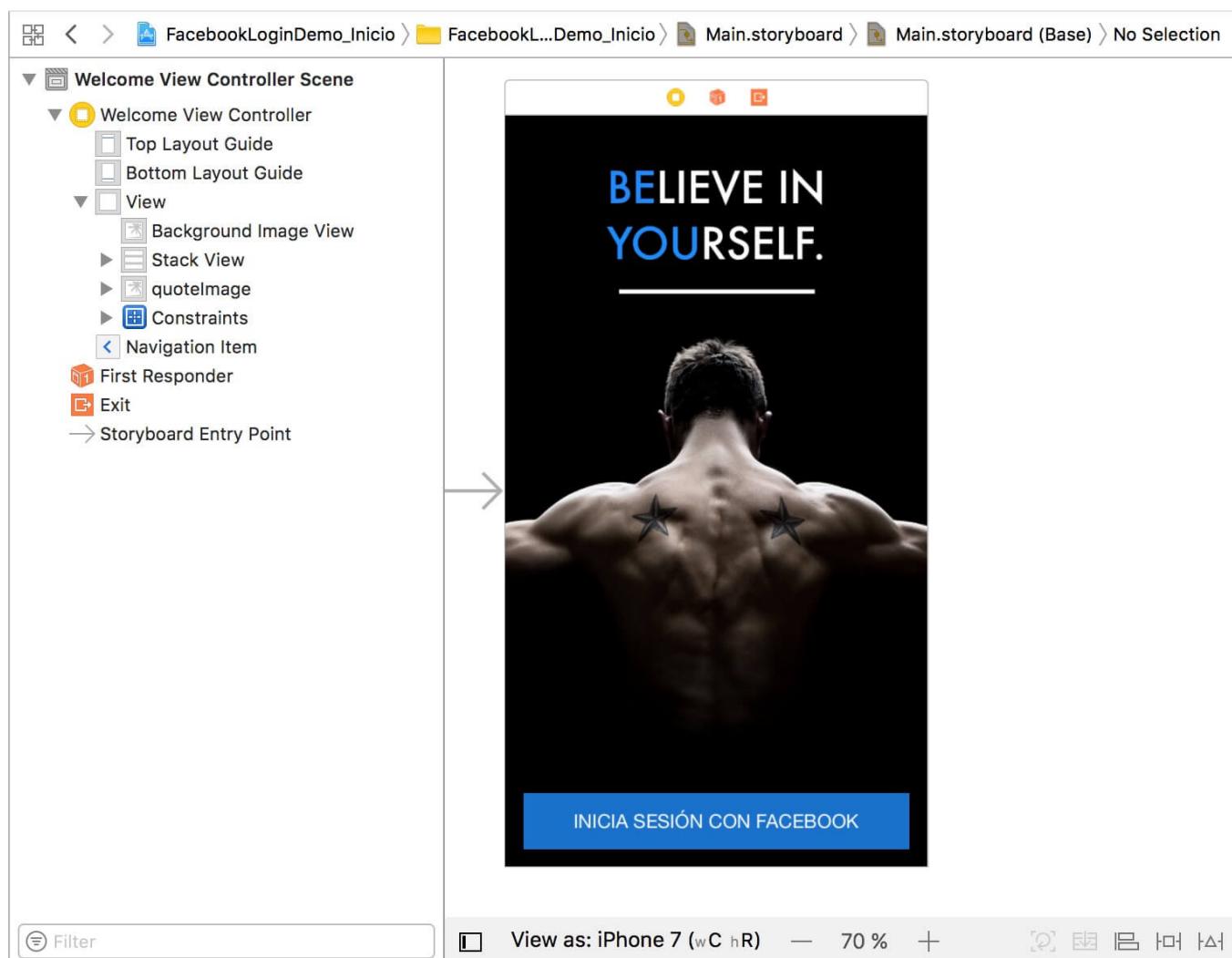
Se trata de una aplicación muy sencilla.

Accede a *Main.storyboard* y verás que la **interfaz** consta de dos view controllers.

El primero de ellos contiene:

- Una imagen de fondo
- Una imagen en la parte superior
- Un botón de Inicio de Sesión en la parte inferior

El segundo únicamente contiene un **label** con el texto WELCOME.



Accede ahora a *ViewController.swift* y allí podrás ver existe un **método** llamado *loginWithFacebook()*. Este método será el que deberemos **implementar**.

```
1 //  
2 //  ViewController.swift  
3 //  FacebookLoginDemo_Inicio  
4 //  
5 //  Created by Luis Rollon Gordo on 7/4/17.  
6 //  Copyright © 2017 EfectoApple. All rights reserved.  
7 //  
8  
9 import UIKit  
10  
11 class ViewController: UIViewController {  
12  
13     override func viewDidLoad() {  
14         super.viewDidLoad()  
15         // Do any additional setup after loading the view, typically from a nib.  
16     }  
17  
18     override func didReceiveMemoryWarning() {  
19         super.didReceiveMemoryWarning()  
20         // Dispose of any resources that can be recreated.  
21     }  
22  
23     @IBAction func loginWithFacebook(_ sender: Any) {  
24  
25     }  
26  
27 }  
28
```

El **funcionamiento** es muy sencillo.

Cuando el usuario pulse sobre el botón INICIA SESIÓN CON FACEBOOK, se ejecutará nuestro método *loginWithFacebook()*, que será quien **gestione** con Firebase el **inicio de sesión** a través de **Facebook**.

Si los **datos** introducidos por el usuario para acceder a través de Facebook son **correctos**, se mostrará la pantalla de bienvenida con el mensaje **WELCOME**.

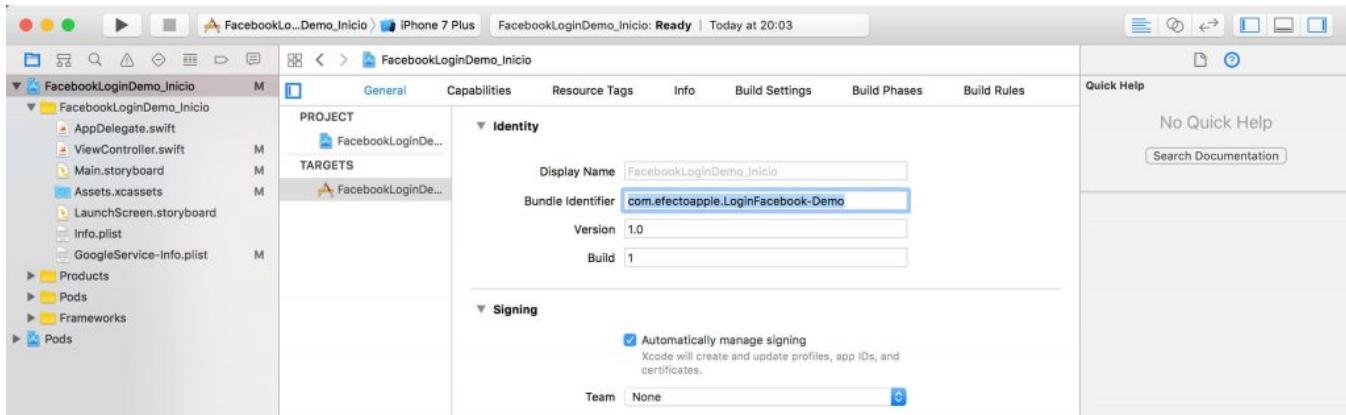
No tiene mas. Con esta aplicación tan sencilla veremos como configurar un **login con facebook**, que después podrás utilizar en cualquier **aplicación iOS**.

Realizando un pequeño cambio en el Proyecto de Inicio

Antes de ponernos a trabajar en el proceso de **configuración**, tienes que realizar una **pequeña modificación**, en el Proyecto de Inicio.

Debes modificar el **Bundle Identifier** de la aplicación. Esto es necesario, ya que durante el proceso de configuración de **Firebase**, utilizarás este Bundle Identifier y **debe ser único**, por lo que no puedes utilizar el mismo que yo utilizo en mi aplicación.

Para realizar este cambio, pulsa en el **nombre del proyecto** y en el menú **General**, en la propiedad **Bundle Identifier**, introduce el Bundle Identifier que prefieras:



Es importante que lo **apuntes** o lo **recuerdes**, porque en el **siguiente apartado** tendrás que utilizarlo para configurar tu aplicación **en Facebook** y más adelante lo utilizarás en la configuración de **Firebase**.

Una vez que hemos **comprendido** el funcionamiento de nuestra aplicación y has realizado esta **pequeña modificación**, pongámonos **manos a la obra**.

Nuestro primer paso será configurar una **nueva Aplicación** en Facebook.

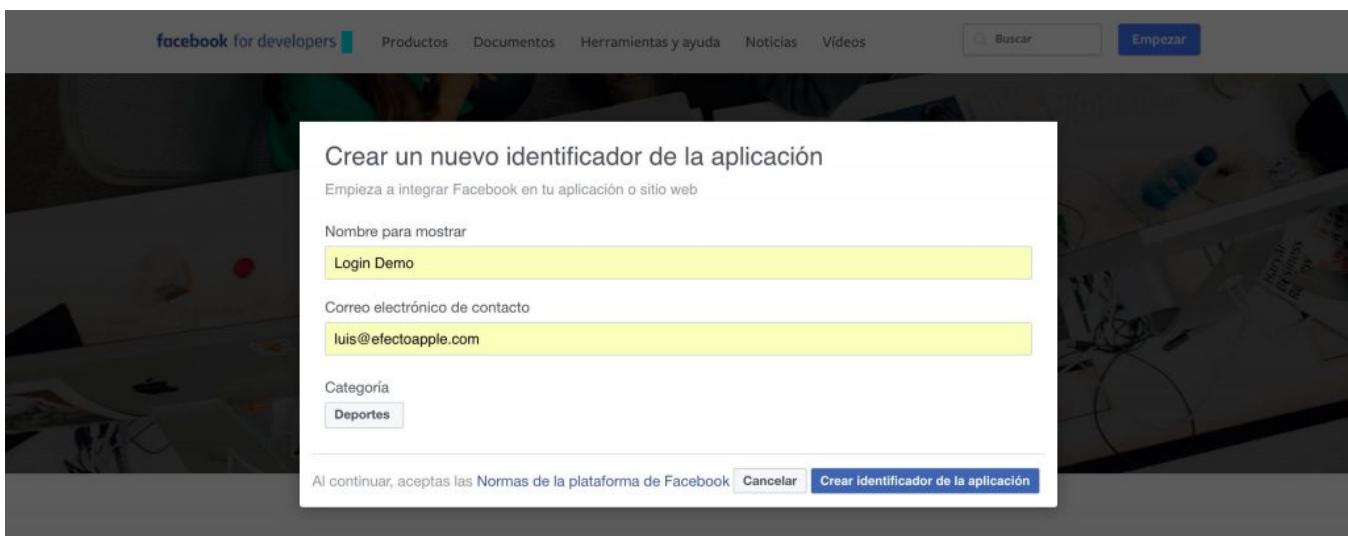
6. Configurando una Aplicación en Facebook

Para poder comenzar con este proyecto, deberás acceder a [la web de Facebook para desarrolladores](#), pulsar en el botón **EMPEZAR** e iniciar sesión con tu cuenta de Facebook.

Una vez que te has logado, la propia web te permitirá crear una nueva aplicación, pulsando en el botón **Crear identificador de la aplicación**.



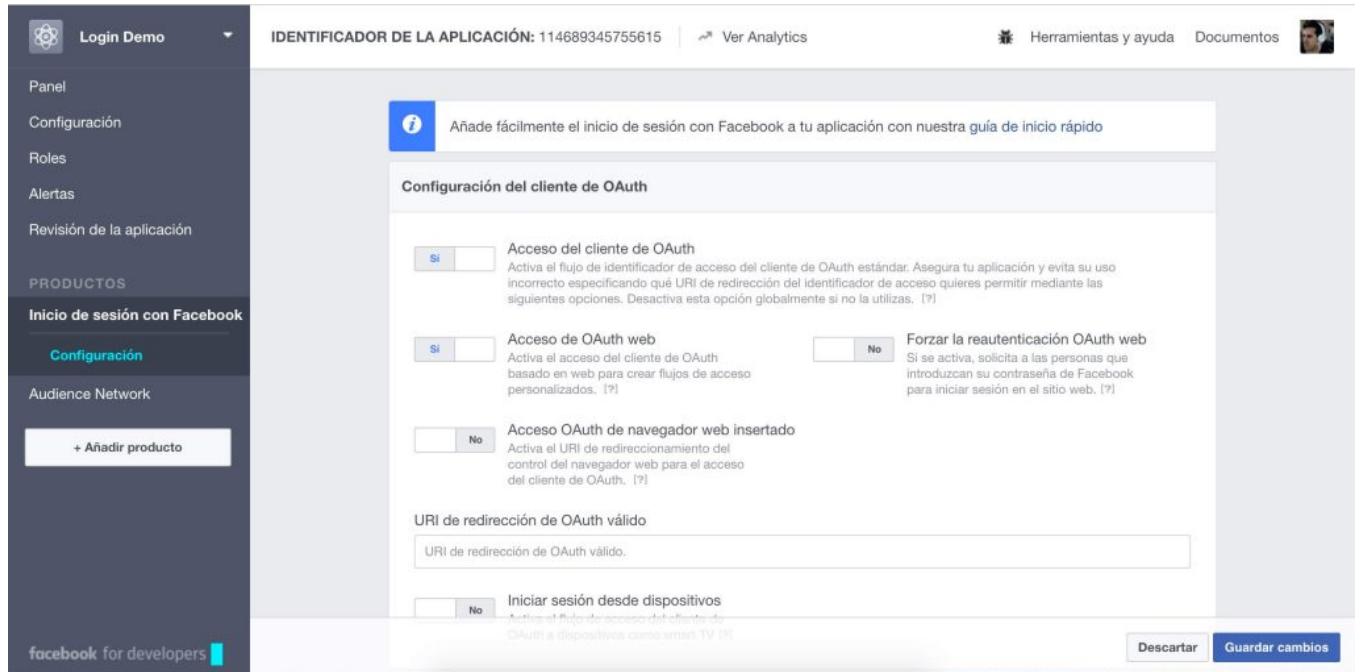
Después, introduce el **nombre** de la Aplicación y tu **correo electrónico** de contacto. A continuación elige una categoría para tu app y pulsa en **Crear identificador de la aplicación**.



A continuación la web te mostrará el **panel de control** de tu aplicación, donde podrás configurar el login con Facebook.

Pulsa el botón **+ Añadir producto** y justo a la derecha de Inicio de sesión con Facebook, pulsa en Empezar.

Se mostrará una nueva ventana con algunas **opciones de configuración** que dejaremos tal cual vienen por defecto.



The screenshot shows the 'Configuración del cliente de OAuth' (OAuth Client Configuration) section. It includes three toggle switches: 'Acceso del cliente de OAuth' (Client OAuth Access), 'Acceso de OAuth web' (Web OAuth Access), and 'Acceso OAuth de navegador web insertado' (Embedded Web Browser OAuth Access). Below these are fields for 'URI de redirección de OAuth válido' (Valid OAuth redirect URI) and 'Iniciar sesión desde dispositivos' (Log in from devices). At the bottom right are 'Descartar' (Discard) and 'Guardar cambios' (Save changes) buttons.

Fíjate que hay una opción, llamada **URI de redirección de OAuth válido**. No la pierdas de vista, ya que más adelante tendrás que añadir algo en este apartado.

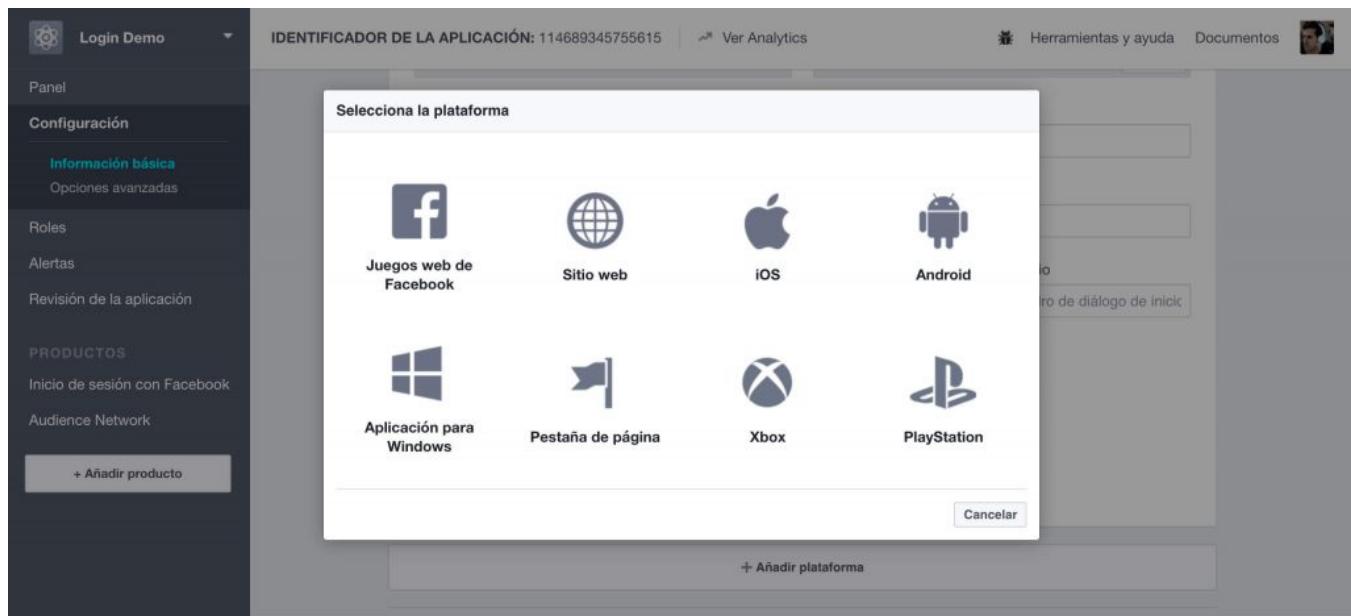
Menú Configuración

Después, pulsa en el menú **Configuración** situado en la parte superior izquierda y haz click en el botón situado en la parte inferior + **Añadir plataforma**.



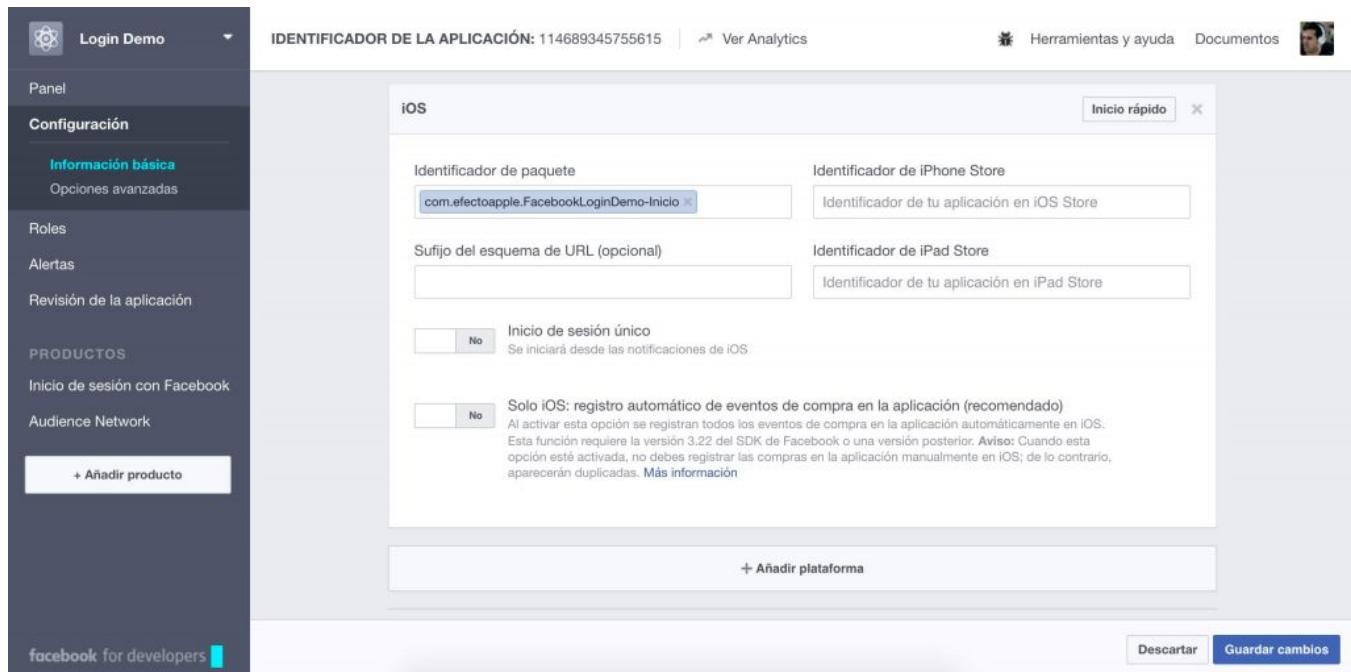
The screenshot shows the 'Información básica' (Basic Information) section. It includes fields for 'Nombre para mostrar' (Name to display) with 'Login Demo', 'Espacio de nombres' (Namespace), 'Dominios de aplicaciones' (Application Domains), 'Correo electrónico de contacto' (Contact email) with 'luis@efectoapple.com', 'URL de la política de privacidad' (Privacy Policy URL) with 'Política de privacidad del cuadro de diálogo de inicio de sesión', 'URL de las Condiciones del servicio' (Terms of Service URL) with 'Condiciones del servicio del cuadro de diálogo de inicio de sesión', 'Icono de la aplicación (1024 x 1024)' (Application icon (1024x1024)), 'Categoría' (Category) with 'Deportes', and a file upload area for the icon. At the bottom right is a '+ Añadir plataforma' (Add platform) button.

En la ventana emergente que aparecerá elige **iOS**:



A continuación, en la opción **Identificador de paquete** (Que en realidad es la traducción de nuestro conocido Bundle ID) deberás introducir el **Bundle Identifier** que has utilizado en tu aplicación en Xcode en el apartado anterior.

Yo, en este caso, utilizaré el siguiente: com.efectoapple.FacebookLoginDemo-Inicio.



Para confirmar este cambio, pulsa en el botón **Guardar cambios**.

Al realizar esta **modificación**, tu aplicación estará preparada para

funcionar.

Como paso final, deberás de apuntar en algún fichero de texto, los siguientes **atributos** de tu app:

- Identificador de la aplicación
- Clave secreta de la aplicación (Aparece oculta y tendrás que pulsar en Mostrar)

The screenshot shows the Facebook App Configuration page for an app named 'Login Demo'. The left sidebar has sections for Panel, Configuración (with Información básica selected), Roles, Alertas, and Revisión de la aplicación. Under PRODUCTOS, it lists Inicio de sesión con Facebook and Audience Network. The main content area shows the following fields:
- Identificador de la aplicación: 114689345755615
- Clave secreta de la aplicación: (hidden, with a 'Mostrar' button)
- Nombre para mostrar: Login Demo
- Espacio de nombres: (empty field)
- Dominios de aplicaciones: (empty field)
- Correo electrónico de contacto: luis@efectoapple.com
- URL de la política de privacidad: Política de privacidad del cuadro de diálogo de inicio de...
- URL de las Condiciones del servicio: Condiciones del servicio del cuadro de diálogo de inicio de...

Estos dos datos son **importantes** ya que tendrás que utilizarlos en el **siguiente apartado** del tutorial.

Esto es todo por ahora en cuanto a la **configuración** de nuestra aplicación en la **página de Facebook para desarrolladores**. No cierres esta página ya que si recuerdas, tenemos pendiente llenar el campo **URI de redirección de OAuth válido**. No te preocunes, esto lo haremos a partir de información que obtendremos en el siguiente apartado.

Veamos como continuar.

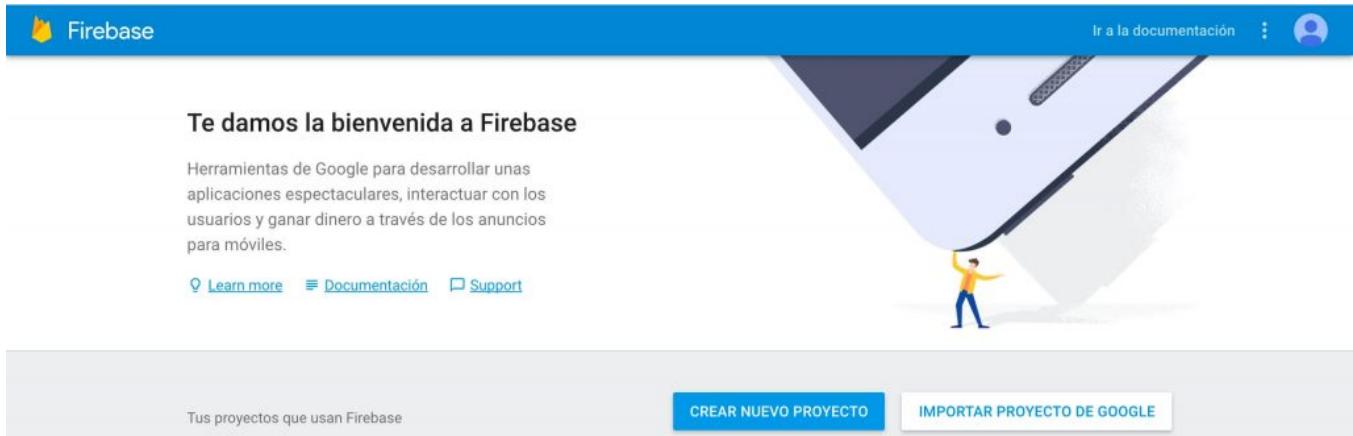
7. Creando el Proyecto en Firebase

Para poder **integrar Firebase** en nuestra aplicación, lo primero que tenemos que hacer es crear **nuestra cuenta** en dicho servicio.

Para ello, accede a la [página de inicio](#) y haz login.

Al tratarse de una empresa propiedad de **la gran G**, bastará que utilices

una cuenta de **cualquier servicio de Google** para acceder. Ni siquiera necesitas registrarte.

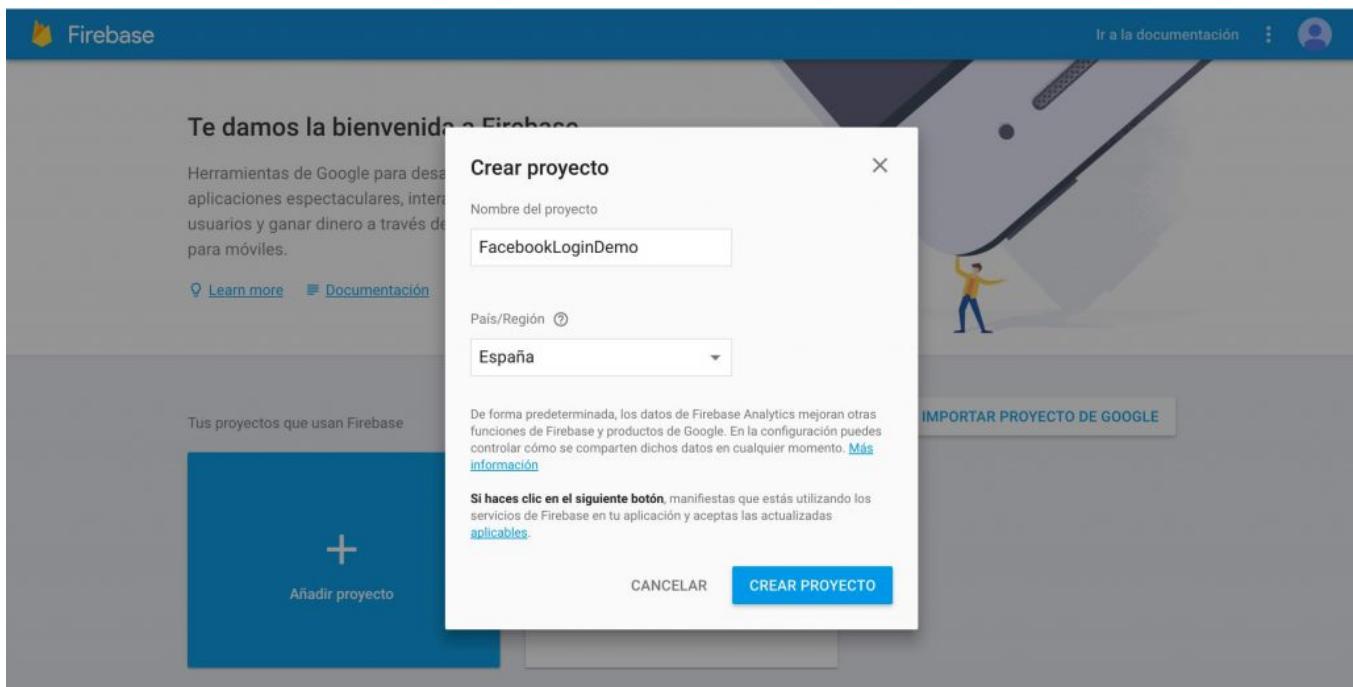


Haz click en el botón **CREAR NUEVO PROYECTO**.

Aquí tendremos que **darle nombre** a nuestro proyecto.

Puedes utilizar el nombre que prefieras. En mi caso, yo lo llamaré *FacebookLoginDemo*.

Además de especificar el nombre, tendrás que especificar el **país** en el que te encuentras.



Haz click en el botón **CREAR PROYECTO** y serás redirigido al **panel de control** del proyecto que acabas de crear en Firebase.

Te damos la bienvenida a Firebase. Empieza aquí.

Añade Firebase a tu aplicación de iOS

Añade Firebase a tu aplicación de Android

Añade Firebase a tu aplicación web

Descubre Firebase

Spark Gratuito 0 \$/mes

ACTUALIZAR

Panel de Control de Firebase

Desde aquí podrás gestionar **todos los servicios** que Firebase puede ofrecerte para tu aplicación.

Para comenzar haz click en el botón **Añade Firebase a tu Aplicación iOS**.

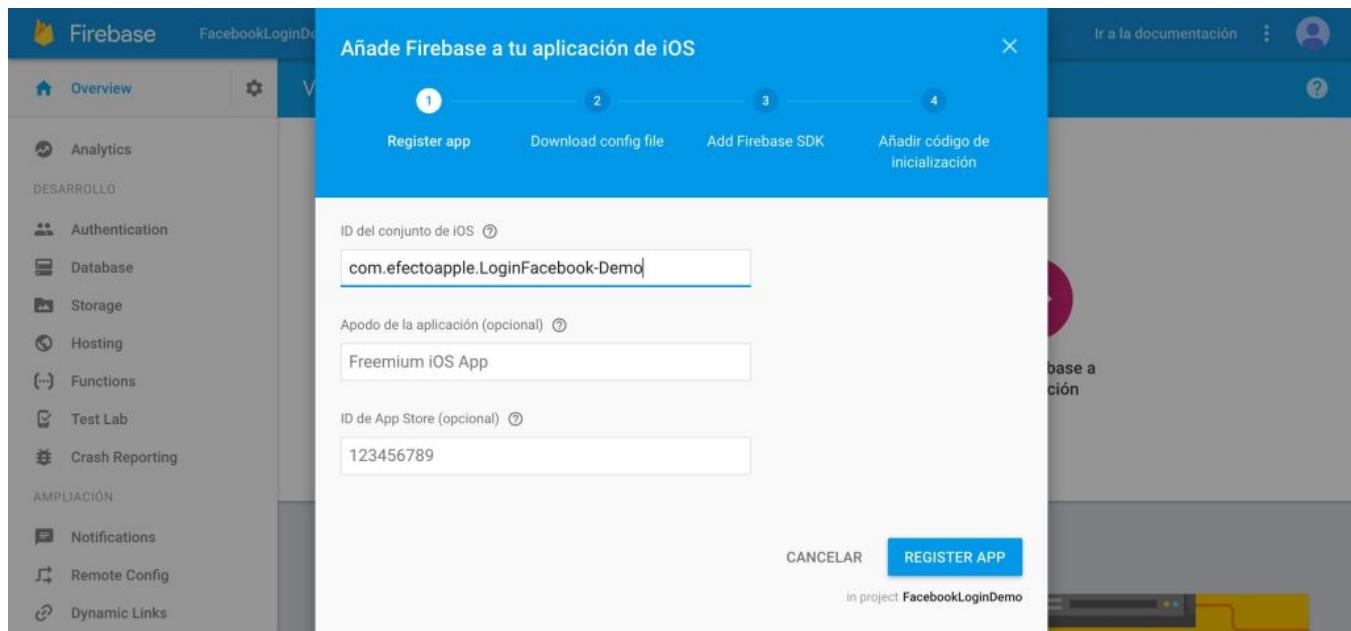
Después, tendrás que introducir en el campo **ID del conjunto de iOS** una cadena de texto que identifique a tu proyecto.

Yo he elegido este:

com.efectoapple.LoginFacebook-Demo

Tu deberás utilizar **uno diferente**, ya que este identificador debe ser único. Además **debe coincidir** con el que **eligieras** en tu proyecto en Xcode, en el **Apartado 5**.

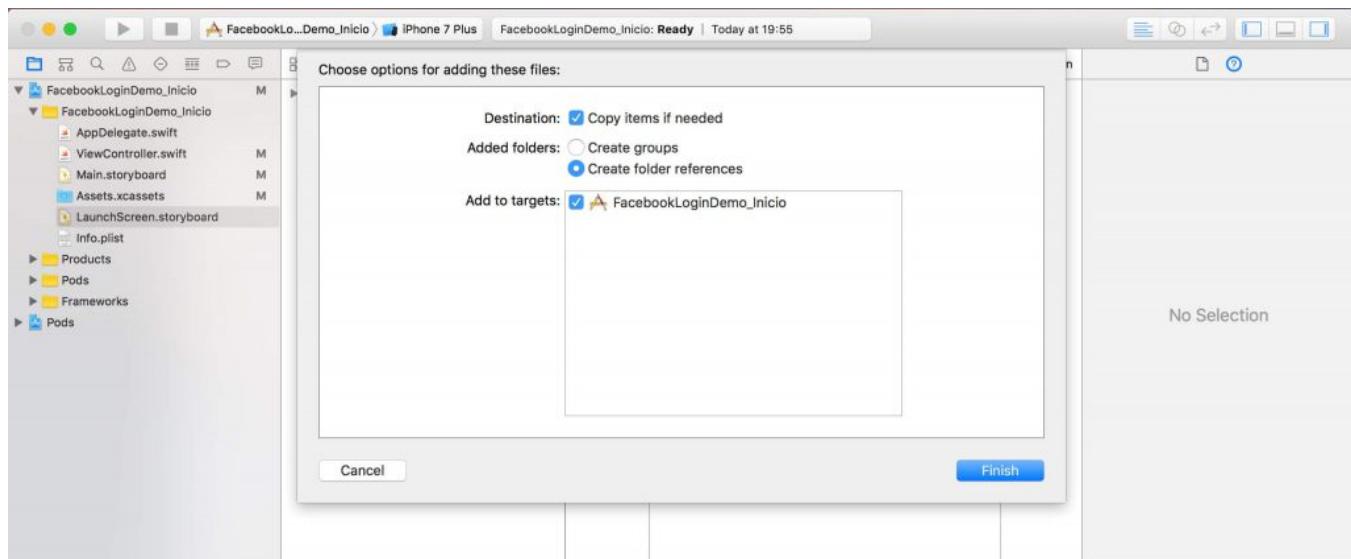
Los otros dos campos: **Apodo de la aplicación** y **ID de App Store** los puedes dejar vacíos.



Para continuar pulsa en el botón **REGISTER APP** y pulsa en el botón **Download GoogleService-Info.plist**. Un fichero llamado **GoogleService-Info.plist** se descargará automáticamente.

Añadiendo .plist a nuestro Proyecto

Localiza ese fichero descargado y **añádelo** a la raíz de tu **proyecto** (arrastrándolo dentro de Xcode). Cuando al añadirlo, aparezca la siguiente ventana, pulsa en el botón **Finish**:



Después de añadir el **.plist** a tu proyecto, **continúa** en la web de **Firebase**, pulsando en el botón **CONTINUAR**. La siguiente página describe como **instalar el SDK de Firebase**.

Añade Firebase a tu aplicación de iOS

1 Register app 2 Download config file 3 Add Firebase SDK 4 Añadir código de inicialización

CocoaPods instructions

Alternatives: [Download ZIP](#) [Unity](#) [C++](#)

Google services use [CocoaPods](#) to install and manage dependencies. Open a terminal window and navigate to the location of the Xcode project for your app.

1. Crea un archivo Podfile si aún no tienes uno:
\$ pod init
2. Abre el archivo Podfile y añade lo siguiente:
pod 'Firebase/Core'
incluye Firebase Analytics de forma predeterminada
3. Guarda el archivo y ejecuta lo siguiente:
\$ pod install

Esta acción crea un archivo .xcworkspace para la aplicación. Usa el archivo para las futuras tareas de desarrollo de tu aplicación.

¿Ya has añadido el código de inicialización y el pod?
[Saltar a la consola](#)

CONTINUAR

Si estás siguiendo este tutorial a partir del **proyecto inicial** que te he dejado en el **apartado 3**, aquí no tienes que hacer nada, puesto que el proyecto de Xcode que te he preparado ya tiene instalado el SDK de Firebase, así que pulsa en **CONTINUAR**.

La última página explica **como conectar con Firebase** cuando tu app arranque.

Añade Firebase a tu aplicación de iOS

1 Register app 2 Download config file 3 Add Firebase SDK 4 Añadir código de inicialización

Para conectar Firebase cuando se inicie la aplicación, añade el siguiente código de inicialización a la clase AppDelegate.

Swift Objective-C

```
import UIKit
import Firebase

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {

    var window: UIWindow?

    func application(application: UIApplication,
                    didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?) {
        -> Bool {
            FIRApp.configure()
            return true
        }
    }
}
```

FINALIZAR

Haz click en el botón **FINALIZAR** para cerrar el asistente de configuración. Verás los detalles de la aplicación que acabamos de crear.

Aplicaciones móviles FacebookLoginDemo

iOS com.efectoapple.LoginFacebo...

Explorar Firebase Analytics →

0 Usuarios activos/mes	0 \$ Compras en la aplicación
Bloqueos (30 días)	
0 Usuarios afectados	0 Instances

+ Añadir otra aplicación

Perfecto, acabas de crear correctamente **tu proyecto en Firebase**. Ahora veremos como activar una última opción que necesitaremos para realizar el login con Facebook.

8. Activar la Autenticación a través de Facebook

Después de haber creado nuestro proyecto en Firebase, podría parecer que ya podemos comenzar a trabajar con él, sin embargo, **es necesario activar** la posibilidad de que nuestros usuarios **puedan acceder** utilizando **Facebook**.

Si no activamos esta opción, nuestra app jamás podría funcionar.

Para activarla, sitúate en el menú **Authentication**, dentro del proyecto que acabamos de crear en Firebase.

The screenshot shows the Firebase console's Authentication section. On the left sidebar, 'Authentication' is selected under 'DESARROLLO'. The main area displays a table of users with columns: Correo electrónico, Proveedores, Fecha de creación, Inicio de sesión, and UID de usuario. A search bar at the top allows filtering by email or UID. Below the table, there's a purple circular icon with a user profile and the text: 'Autentica y administra usuarios de una gran variedad de proveedores sin necesidad de utilizar ningún código del servidor'. A blue button labeled 'CONFIGURA EL MÉTODO DE INICIO DE SESIÓN' is visible. At the bottom left, it says 'Spark Gratuito 0 \$/mes' and 'ACTUALIZAR'.

Después, pulsa en el menú superior **MÉTODO DE INICIO DE SESIÓN**.

En esta nueva pantalla, verás que se muestra un **listado de proveedores**:

- Correo electrónico/contraseña
- Google
- Facebook
- Twitter
- GitHub
- Anónimo

Puedes ver que todos están **inhabilitados** por defecto.

Como en nuestro caso, lo que nos interesa es el **login con Facebook**, haz click en dicho método de inicio de sesión.

A continuación tendrás que habilitar esta opción, **activando el interruptor** situado en la parte superior derecha.

Añadiendo App ID y App Secret

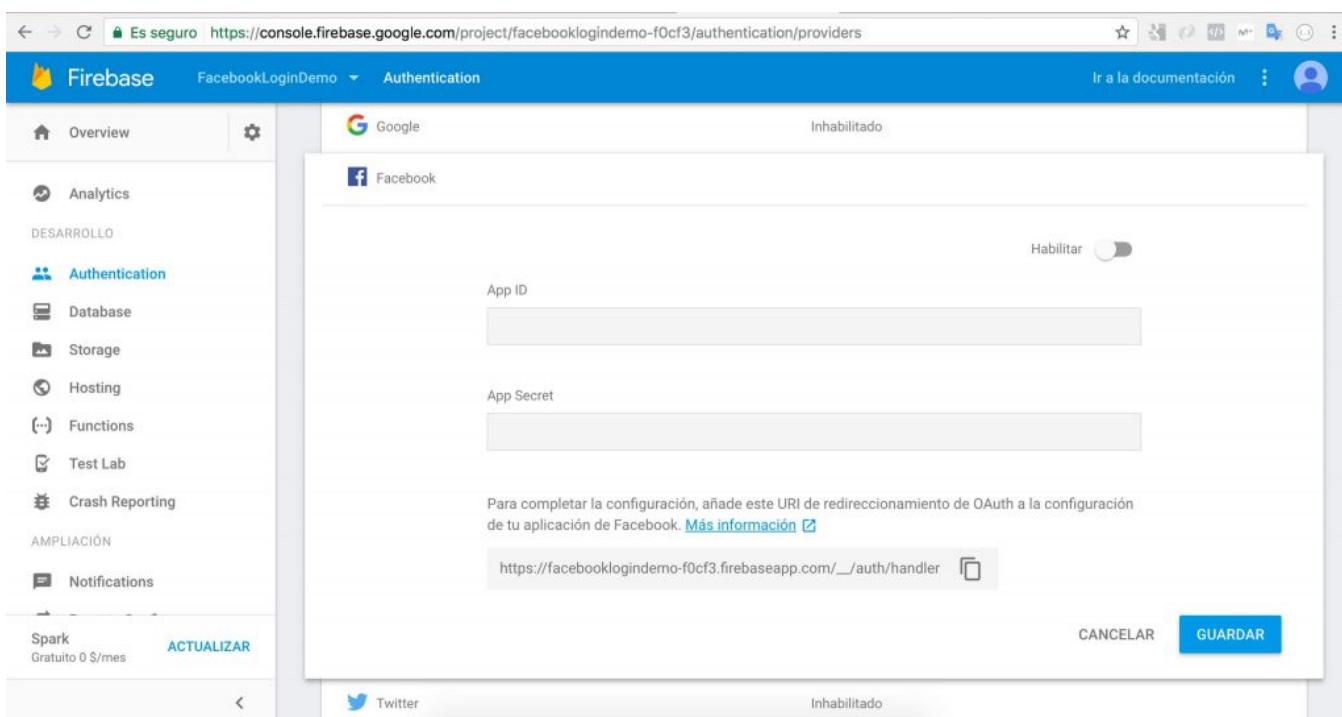
También tendrás que añadir **2 campos** que obtuviste, al final del apartado anterior de este tutorial:

- **App ID** (Que en el apartado anterior se llamaba Identificador de la aplicación)
- **App Secret** (Que en el apartado anterior se llamaba Clave secreta de la aplicación)

Además, si recuerdas el apartado del tutorial en el que configuramos nuestra aplicación en Facebook, había un campo que **dejamos vacío**. El campo se llamaba **URI de redirección de OAuth válido**. Esa información que tenemos pendiente de llenar la obtenemos del campo de esta ventana de Firebase donde pone:

Para completar la configuración, añade este URI de redireccionamiento de OAuth a la configuración de tu aplicación de Facebook.

Echa un vistazo a la **siguiente imagen** para que tengas todo más claro:



Por tanto, **copia esa URI**, vuelve a la **página de desarrolladores de Facebook**, que te comenté que no cerraras, **añádela** en el campo correspondiente y pulsa **Guardar cambios**.

Como **confirmación** final, recibirás un mensaje de que este método de inicio de sesión **ha sido habilitado** y en el listado de proveedores, verás que aparece esta confirmación:

Proveedor	Estado
Correo electrónico/contraseña	Inhabilitado
Google	Inhabilitado
Facebook	Habilitada
Twitter	Inhabilitado
Github	Inhabilitado
Anónimo	Inhabilitado

Con este último paso has terminado de **configurar** tanto tu **aplicación en Facebook**, como tu **aplicación en Firebase**. Enhorabuena, solo nos queda un **último paso** antes de comenzar con la parte de código.

Este **último paso** y la **programación de la app** lo veremos en la **segunda parte** del tutorial.

9. Resumen Final

Como has podido ver, por ahora únicamente hemos revisado **la parte de configuración**.

Hemos seguido todos los **pasos necesarios**, a través de las webs de **Facebook** y **Firebase** para poder llevar a cabo el desarrollo de nuestra aplicación.

En la **segunda parte** del Tutorial continuaremos y **terminaremos** nuestra **App**.

Pantalla de Login con Facebook usando Firebase [Parte 2]

Desarrolla un Login con Facebook utilizando Swift

Lenguaje Swift | Nivel Avanzado

1. Introducción

En el anterior apartado anterior vimos la **primera parte** del Tutorial sobre como **Crear una Pantalla de Login con Facebook usando Firebase**.

Si no has **revisado** la primera parte, te recomiendo que lo hagas antes de continuar con este tutorial.

Hoy vamos a ver **la segunda parte** de ese tutorial.

Si recuerdas, los **pasos de configuración** que hay que dar para desarrollar un Inicio de Sesión con Facebook son los siguientes:

- Configurar la App que vamos a crear, en [la web de Facebook para desarrolladores](#)
- Crear el proyecto en Firebase
- Activar la autenticación en Firebase
- Configurar nuestro proyecto en Xcode

En la primera parte del tutorial vimos los **3 primeros apartados**.

Echemos un vistazo a lo que vamos a ver hoy.

2. ¿Qué vamos a ver en este Tutorial?

El siguiente punto en el que trabajaremos será en **configurar** nuestro proyecto en Xcode.

Una vez que hayamos **terminado** todo el proceso de configuración, por fin podremos **desarrollar** nuestra aplicación.

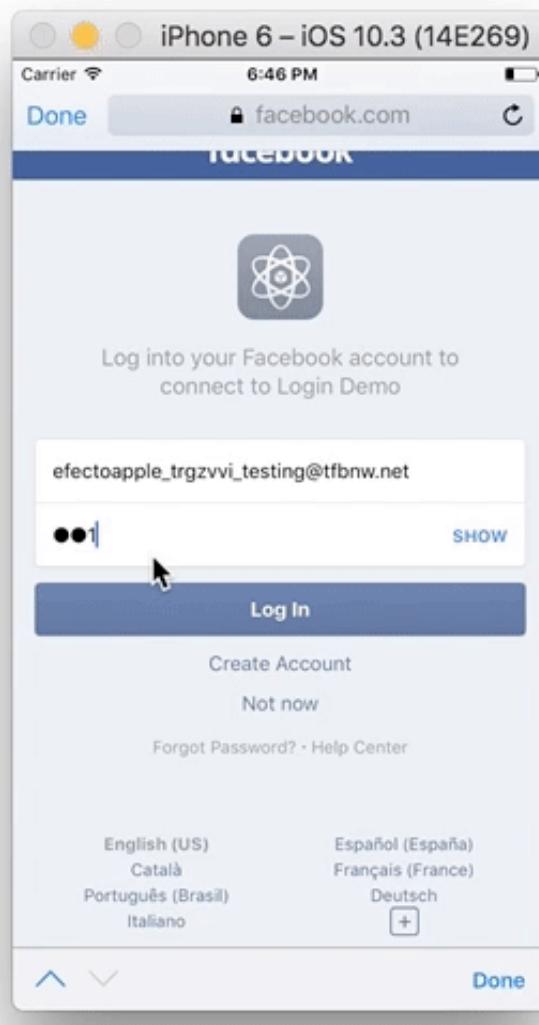
Además, también veremos como crear **usuarios de prueba** para poder testear la app.

Recordemos cual es el **objetivo** que queremos conseguir, es decir, que pinta debe tener la **aplicación** al terminar el tutorial.

3. La Aplicación que vamos a crear

El **objetivo** de este tutorial es crear una pantalla de login que enlazará con **Firebase** y nos permitirá gestionar el **login con Facebook**.

Al terminar tendrás una **aplicación** como esta:



Y además dominarás todo el **proceso** de creación de una **pantalla de**

login, que podrás utilizar en tus propias aplicaciones.

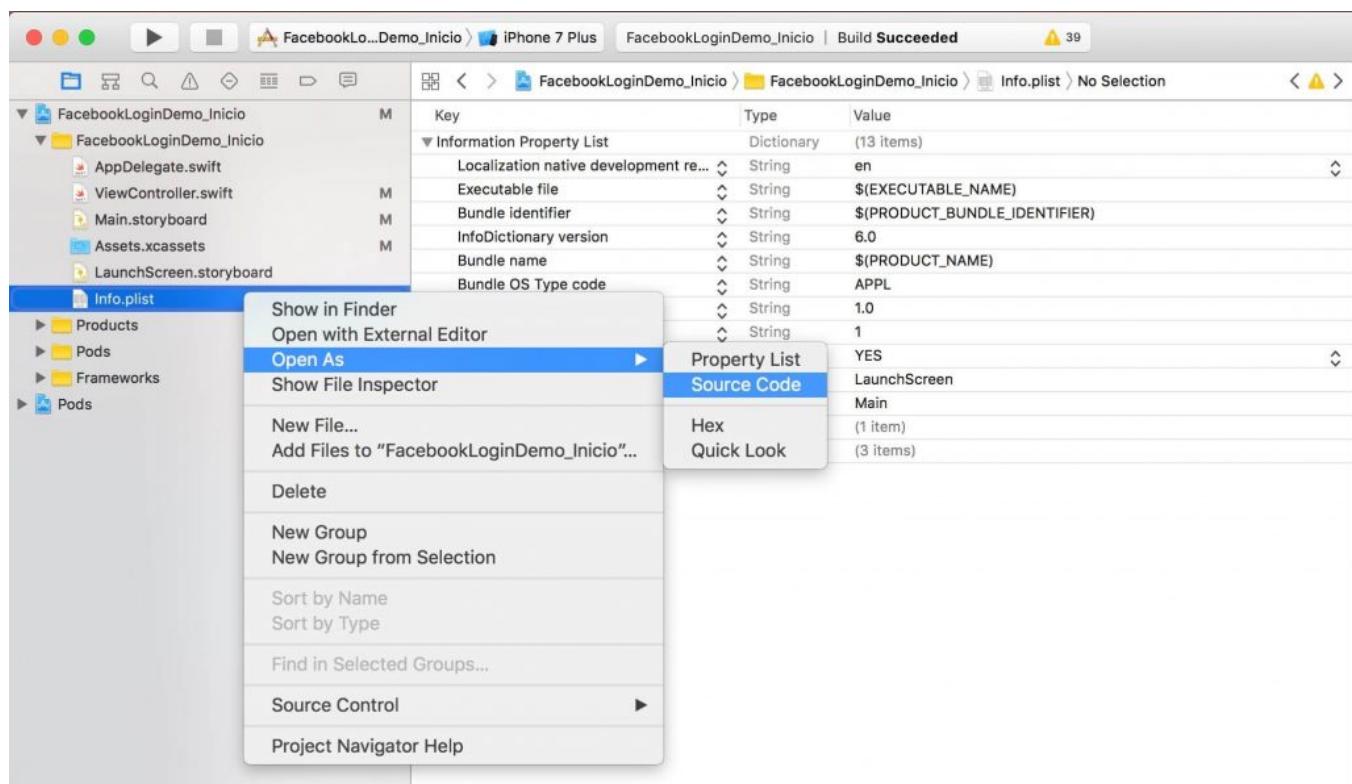
Continuemos entonces, justo donde lo dejamos en la primera parte de este tutorial.

4. Configurando el Proyecto en Xcode

Como hemos comentado antes, lo **primero** que haremos será ver el **último paso de configuración** antes de programar nuestra app.

Si no tienes abierto el **Proyecto de Inicio** que descargaste en la primera parte del tutorial, ábrelo (recuerda, a través del archivo .xcworkspace).

Una vez que lo tienes abierto en Xcode, haz click con el botón derecho sobre el fichero *Info.plist* y elige **Open as > Source code**.



De esta forma, podrás ver que realmente se trata de un **fichero en XML**.

Lo siguiente que tendrás que hacer es **añadir este trozo de código** (con algunos cambios) justo antes de la etiqueta **</dict>**:

```
<key>CFBundleURLTypes</key>
```

```
<array>
```

```

<dict>
    <key>CFBundleURLSchemes</key>
    <array>
        <string>fb114689345755615</string>
    </array>
</dict>
</array>
<key>FacebookAppID</key>
<string>114689345755615</string>
<key>FacebookDisplayName</key>
<string>Login Demo</string>
<key>LSApplicationQueriesSchemes</key>
<array>
    <string>fbapi</string>
    <string>fb-messenger-api</string>
    <string>fbauth2</string>
    <string>fbshareextension</string>
</array>

```

Cambios en Info.plist

Este trozo de código contiene **mi propia configuración**. Deberás realizar algunos **cambios** en él para hacer que tu aplicación funcione:

- Cambia el **App ID** (**114689345755615**) a tu propio ID. Recuerda que puedes consultar este ID en el panel de control de tu app en Facebook.
- Cambia ***fb114689345755615*** por tu propio **URL scheme**. Deberás de sustituirlo por fb+Tu App ID.

- Cambia el **nombre de la app** (Login Demo) por tu propio nombre.

La **API de Facebook** se encargará de leer el fichero *Info.plist* para conectar con tu app en Facebook y gestionar el Inicio de Sesión a través de Facebook. Por tanto, debes asegurarte que el **App ID** que has utilizado coincide con el que tiene tu App en la [web de Facebook para desarrolladores](#).

IDENTIFICADOR DE LA APLICACIÓN: 114689345755615		Ver Analytics	Herramientas y ayuda
Identificador de la aplicación	114689345755615	Clave secreta de la aplicación	***** Mostrar
Nombre para mostrar	Login Demo	Espacio de nombres	
Dominios de aplicaciones		Correo electrónico de contacto	luis@efectoapple.com

Tal vez te estés preguntando para qué sirve la clave **LSApplicationQueriesSchemes**.

Esta clave determina los **URL Schemes** que nuestra app puede utilizar con el método **canOpenURL**:

Es decir, sirve para lanzar la **app oficial de Facebook**, (siempre que el usuario la tenga instalada en su dispositivo) y **realizar el login** desde allí.

Si el usuario **no la tiene instalada**, no hay ningún problema, podrá realizar el login con Facebook directamente **desde el navegador**.

Ahora sí. Despues de **todo el proceso de configuración**, podemos centrarnos en programar nuestra app. ¡Vamos a ello!

5. Implementando nuestro AppDelegate

Vamos a comenzar a programar la aplicación, escribiendo el código que corresponde al **AppDelegate.swift**.

Accede a esta clase desde Xcode y lo primero que haremos será **importar las clases** que vamos a necesitar:

```
import Firebase  
import FBSDKCoreKit
```

Después, añade el siguiente método a la clase **AppDelegate**:

```
func application(_ app: UIApplication, open url: URL, options: [UIApplicationOpenURLOptionsKey : Any] = [:]) -> Bool {  
  
    let handled = FBSDKApplicationDelegate.sharedInstance().application(app,  
open: url, options: options)  
  
    return handled  
  
}
```

Este **método** es el encargado de gestionar la **llamada a la aplicación oficial** para realizar el login con Facebook.

Si recuerdas, el **proceso completo** es el siguiente:

- El usuario pulsa en el botón de Iniciar Sesión con Facebook.
- Nuestra aplicación intenta abrir la aplicación oficial para realizar el login con Facebook.
- En el caso de que esta aplicación no esté instalada, nos permite realizar el login desde el navegador.
- El usuario introduce sus credenciales de Facebook.
- La app oficial de Facebook o el navegador, comprueban que las credenciales son correctas.
- La app oficial o el navegador, vuelven a nuestra app, pasándonos las credenciales.
- Nuestra aplicación, al haber realizado correctamente el login muestra la pantalla de bienvenida.

Todo este proceso se realiza desde el método *application()* que acabamos de programar.

Lo siguiente que tendremos que hacer es llamar a este método desde nuestro **AppDelegate** y también llamar al método que nos permitirá **configurar Firebase**. Por tanto, deberás modificar tu método

didFinishLaunchingWithOptions() para que tenga esta pinta:

```
func application(_ application: UIApplication,  
didFinishLaunchingWithOptions launchOptions:  
[UIApplicationLaunchOptionsKey: Any]?) -> Bool {  
  
    //Llamamos al método configure() para configurar Firebase en nuestra App  
    FIRApp.configure()  
  
    //Llamamos al método application() que gestionará la llamada a la app de  
    Facebook  
  
    FBSDKApplicationDelegate.sharedInstance().application(application,  
    didFinishLaunchingWithOptions: launchOptions)  
  
    return true  
  
}
```

Con esto, habríamos terminado en nuestra clase **AppDelegate**.

6. Implementando la clase **ViewController**

Vayamos ahora a la clase **ViewController** para implementar el código que vamos a necesitar.

Lo primero que haremos, exactamente igual que antes, será añadir el **import** de las clases de Facebook y de Firebase:

```
import Firebase  
  
import FBSDKLoginKit
```

Después, nos centraremos en el método *loginWithFacebook()*, que es el método principal de nuestra app.

El método **loginWithFacebook()**

Actualmente este método está vacío, nuestra misión ahora será **rellenarlo** completamente.

Primero te mostraré el **código completo** de este método y

posteriormente la **explicación**. Este es el código que tendrás que añadir al método *loginWithFacebook()*:

```
@IBAction func loginWithFacebook(_ sender: Any) {  
    //1.  
    let fbLoginManager = FBSDKLoginManager()  
    fbLoginManager.logIn(withReadPermissions: ["public_profile", "email"],  
    from: self) { (result, error) in  
  
        //2.  
        if let error = error {  
            print("Failed to login: \(error.localizedDescription)")  
            return  
        }  
        //2.  
        guard let accessToken = FBSDKAccessToken.current() else {  
            print("Failed to get access token")  
            return  
        }  
        //3.  
        let credential = FIRFacebookAuthProvider.credential(withAccessToken:  
        accessToken.tokenString)  
        //4.  
        FIRAuth.auth()?.signIn(with: credential, completion: { (user, error) in
```

```

//5.

if let error = error {
    print("Login error: \(error.localizedDescription)")

    let alertController = UIAlertController(title: "Login Error", message:
error.localizedDescription, preferredStyle: .alert)

    let okayAction = UIAlertAction(title: "OK", style: .cancel, handler:
nil)

    alertController.addAction(okayAction)

    self.present(alertController, animated: true, completion: nil)

    return
}

//6.

if let viewController =
self.storyboard?.instantiateViewController(withIdentifier:
"WelcomeViewController") {

    UIApplication.shared.keyWindow?.rootViewController =
viewController

    self.dismiss(animated: true, completion: nil)

}

}

}

```

Explicación de este método

Y aquí tienes la explicación paso a paso:

- //1. La clase **FBSDKLoginManager** nos ofrece el método *logIn()*,

que nos permite iniciar sesión, especificando **los permisos** que queremos solicitar al usuario. En nuestro caso, solicitamos acceder a su perfil público y a su email.

- //2. Después de que el usuario haya introducido sus **credenciales** de Facebook, se comprobará si se ha producido algún error.
- //3. Si no se produce ningún error, procederemos a recuperar el token de acceso del usuario y lo convertiremos en una credencial de Firebase, utilizando el método **credential** de la clase **FIRFacebookAuthProvider**.
- //4. Utilizando esta credencial, llamaremos al método *signIn()* de Firebase.
- //5. Si se produce algún error iniciando sesión en Firebase, mostraremos un **alertController** informando el usuario.
- //6. En caso de que no se produzca ningún error, mostraremos la pantalla de bienvenida.

Con este método, **hemos terminado** la implementación completa de nuestra aplicación.

¿Cuál es el siguiente paso, después de programar cualquier aplicación?

Probarla.

Vamos a ello.

7. Creando un Usuario de Prueba

Para probar la aplicación que acabamos de desarrollar, podríamos utilizar **las credenciales** de cualquier cuenta de Facebook que tengamos.

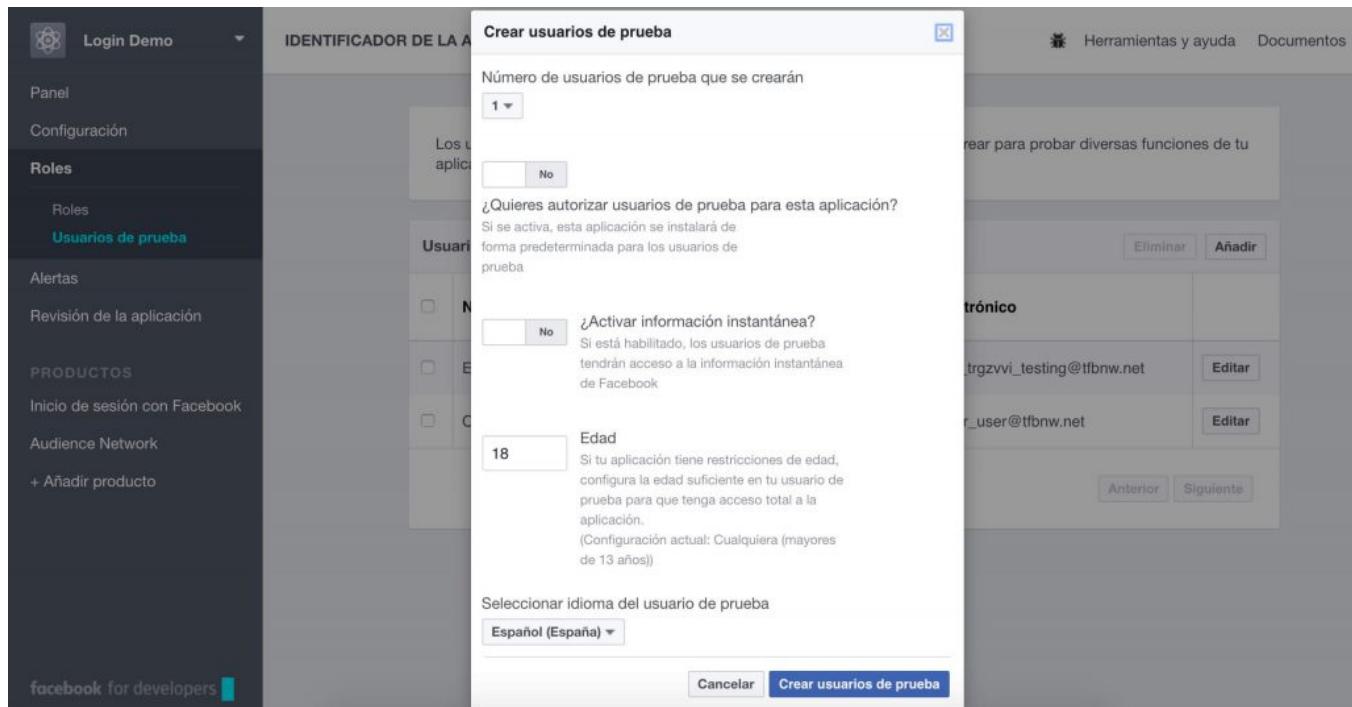
Sin embargo, vamos a hacerlo de la forma correcta.

Vamos a crear un **Usuario de Prueba** que utilizaremos para testear nuestra aplicación.

Accede de nuevo a la [web de Facebook para desarrolladores](#) y dentro de tu

app, accede al menú **Roles>Usuarios de prueba**.

Haz click en el botón **Añadir**, selecciona **crear 1 nuevo usuario** y pulsa en **Crear usuarios de prueba**.



Facebook generará un **usuario de prueba** con un nombre y un email aleatorio. Puedes modificar el nombre y la contraseña pulsando el el botón **Editar**.

Recuerda el **email** y la **contraseña** que has introducido, ya que lo usaremos en el siguiente apartado para iniciar sesión.

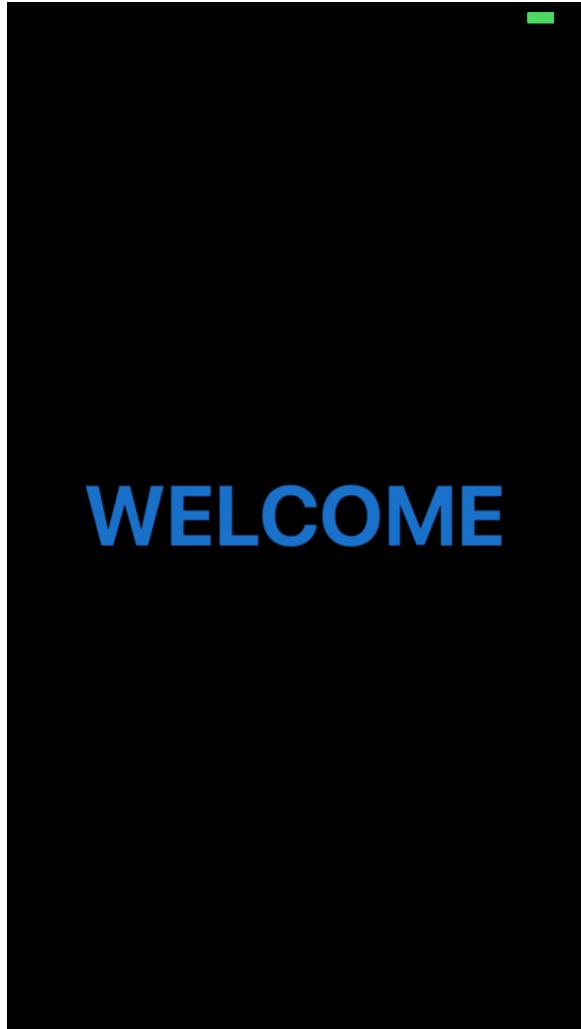
8. Testeando nuestra Aplicación

Después de todo este proceso, por fin vamos a poder **probar nuestra aplicación**.

Ejecútala en el simulador y sigue estos pasos:

1. Pulsa el botón INICIA SESIÓN CON FACEBOOK
2. Cuando Facebook te solicite las credenciales, introduce el **email** y **contraseña** del usuario de prueba
3. Acepta la solicitud de permiso

Al seguir estos pasos, verás como la aplicación muestra correctamente **la pantalla de bienvenida**:



Acabas de crear una **pantalla de Inicio de Sesión** que combina **Facebook + Firebase** y que además podrás utilizar en tus propias aplicaciones.

¡Enhorabuena!

9. Resumen Final

Acabas de seguir paso a paso, un **Tutorial** muy **completo**.

Hemos utilizado **Cocoapods**, **Firebase** y **Facebook** en una misma aplicación.

Además, teniendo todo el proceso claro, podrás integrar el **login con Facebook** en cualquier aplicación que desarrolles.



Módulo 3

Código Fuente

Descarga Código Fuente de las 10 Aplicaciones

Hemos llegado al final del libro en el que hemos visto paso a paso como desarrollar 10 aplicaciones iOS.

Tanto si has ido siguiendo paso a paso cada uno de los tutoriales como si no, creo que es importante para complementar tu aprendizaje, que descargues el código fuente de cada una de las 10 aplicaciones que hemos desarrollado.

Esto te permitirá acceder a las tripas de los proyectos, entender cada parte de la aplicación, modificarlas, romperlas y en definitiva comprender mucho mejor el funcionamiento de una Aplicación iOS.

Por lo tanto, quiero darte acceso a la descarga del código completo de las 10 aplicaciones [Aplicable a compradores desde Amazon. Si lo compraste en mi web estoy seguro que ya has podido descargar todo el código fuente]

[Descarga ahora el código fuente de las 10 aplicaciones desde aquí.](#)

Te veo dentro.

Me despido. Tan solo me queda decirte GRACIAS, por tu confianza, por tu tiempo, por el interés en aprender como desarrollar Aplicaciones iOS. No me cabe duda de que tú mismo comprobarás que este es un área con un futuro brillante.

Espero de verdad que este libro haya cumplido tus expectativas de formación y conocimiento, pero si no fuera así, desde aquí te brindo mi ayuda en lo que necesites, **no dudes en escribirme.**

También te quiero recordar que puedes seguir en contacto conmigo a través de mi blog www.efectoapple.com, twitter y facebook.

Estas últimas líneas las quiero emplear en decirte que, si te ha gustado el

libro y piensas que es útil, **te estaría MUY AGRADECIDO si dejas tu opinión en el “Review”.**

Tu apoyo es muy importante para mí:

Leo todos los comentarios e **intento mejorar este libro y futuras formaciones.**

Te mando un fuerte abrazo. ¡Seguro que nos volveremos a encontrar en el camino!

Luis.-

