

# ¿Qué ofrece Autentia Real Business Solutions S.L?

Somos su empresa de **Soporte a Desarrollo Informático**.  
Ese apoyo que siempre quiso tener...

## 1. Desarrollo de componentes y proyectos a medida



## 2. Auditoría de código y recomendaciones de mejora

## 3. Arranque de proyectos basados en nuevas tecnologías

1. Definición de frameworks corporativos.
2. Transferencia de conocimiento de nuevas arquitecturas.
3. Soporte al arranque de proyectos.
4. Auditoría preventiva periódica de calidad.
5. Revisión previa a la certificación de proyectos.
6. Extensión de capacidad de equipos de calidad.
7. Identificación de problemas en producción.



## 4. Cursos de formación (impartidos por desarrolladores en activo)

Spring MVC, JSF-PrimeFaces /RichFaces,  
HTML5, CSS3, JavaScript-jQuery

Gestor portales (Liferay)  
Gestor de contenidos (Alfresco)  
Aplicaciones híbridas


Tareas programadas (Quartz)  
Gestor documental (Alfresco)  
Inversión de control (Spring)

Control de autenticación y  
acceso (Spring Security)  
UDDI  
Web Services  
Rest Services  
Social SSO  
SSO (Cas)

JPA-Hibernate, MyBatis  
Motor de búsqueda empresarial (Solr)  
ETL (Talend)

Dirección de Proyectos Informáticos.  
Metodologías ágiles  
Patrones de diseño  
TDD

BPM (jBPM o Bonita)  
Generación de informes (JasperReport)  
ESB (Open ESB)



Ignacio Acisclo Pérez

Consultor tecnológico de desarrollo de proyectos informáticos.


Puedes encontrarme en Autentia: Ofrecemos servicios de soporte a desarrollo, factoría y formación

Somos expertos en Java/J2EE

Ver todos los tutoriales del autor

CREA TU TIENDA ONLINE

¡Dominio incluido y 30 días Gratis! Comienza YA tu nego...



Fecha de publicación del tutorial: 2014-11-04

Tutorial visitado 15 veces

Descargar en PDF

# Tutorial VIPER en Swift

## 0. Índice de contenidos.

- 1. Introducción
- 2. Ejemplo

## 1. Introducción

Bien hoy vamos a ver como implementar un nuevo patrón de diseño que está pegando fuerte en la comunidad. Se trata de un patrón estructural para facilitar el bajo acoplamiento entre nuestras clases y su correcto testeo, su nombre es VIPER y voy a hacer una breve descripción del mismo.

Se trata de separar nuestras clases en algo más complejo que el típico MVC donde acababa siendo el “C”(controlador) el que se acaba convirtiendo en una clase gigante a medida que crecía la funcionalidad. VIPER son las siglas de la separación conceptual que vamos a hacer con nuestras clases.

Dame un “V”!!!!

La “V” significa View, es un básicamente un protocolo que van a implementar nuestros ViewControllers. Las vistas en VIPER son pasivas, esperan a que les lleguen los datos por parte del Presenter e informan al mismo de los eventos que genera el usuario.

Dame un “I”!!!!

La “I” es de Interactor, el interactor es la clase que va a llevar la lógica de negocio, el que va a tratar con las clases “Entity”, manejar su lógica y es el que va a pasarle al “Presenter ” la información que necesita para la “Vista.”

Dame una “P”!!!!

El Presenter es el intermediario entre la Vista y el Interactor. Recuerda que la vista es tonta y que el interactor es el que carga con la lógica de negocio, así que el Presenter va a recibir eventos de la vista (cuando el usuario pulse un botón por ejemplo) y se va a limitar a pasarlo al Interactor que es el que va hacer la acción oportuna. Cuando dicha acción se complete informará al Presenter y le pasará la información que necesita. Ojo, nunca le vamos a pasar una Entity al Presenter, hay que pasarle todo mascado: estructuras de datos simples. Tener en cuenta que hay que diferenciar bien las responsabilidades para facilitar el testeo. También es posible que el evento que genera el usuario simplemente sea de navegación, en cuyo caso en vez de solicitar información al Interactor llamaremos a nuestra clase Routing que como vamos a ver unas líneas más adelante es la que se encarga de la lógica de navegación.

Dame una “E”!!!

Las entidades son los objetos de negocio con los que tratará el Interactor. Hay que tener en cuenta que si las entidades nos llegan a través de un web service necesitaremos añadir una capa más tipo DAO, que nos haga las solicitudes y le pase las entidades construidas al interactor. Recuerda que el interactor solo lleva la lógica de negocio...SOLO.

Dame una “R”!!!

R de Routing. Va ser la clase encargada de la navegación de nuestra app, también es la responsable de instanciar nuestras Vistas, Interactors y Presenters. Son los Presenters los que notifican al Routing que se ha de navegar a X pantalla.

## Catálogo de servicios Autentia



Síguenos a través de:



Últimas Noticias

» Curso JBoss de Red Hat

» Si eres el responsable o líder técnico, considérate desafortunado. No puedes culpar a nadie por ser gris

» Portales, gestores de contenidos documentales y desarrollos a medida

» Comentando el libro Start-up Nation, La historia del milagro económico de Israel, de Dan Senor & Salu Singer

» Screencasts de programación narrados en Español

Histórico de noticias

Últimos Tutoriales

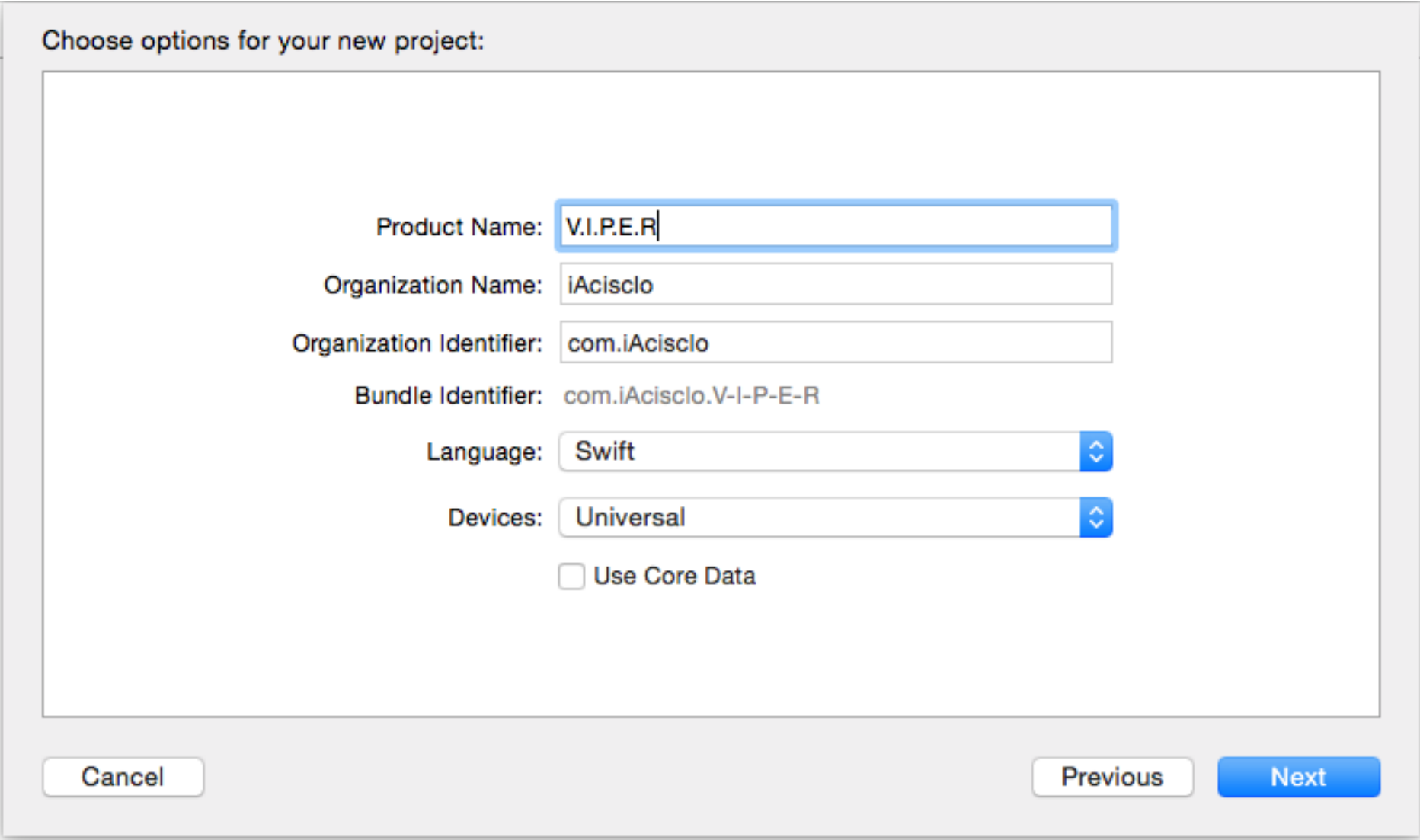
» Monitorización de Apache Kafka

» Hooks en Cordova: Cargar todos los plugins de forma automática

» Generación de vistas

2. Ejemplo

Bien visto estos conceptos vamos a iniciar un nuevo proyecto en Xcode y como no tenemos miedo a nada vamos a hacerlo en Swift.

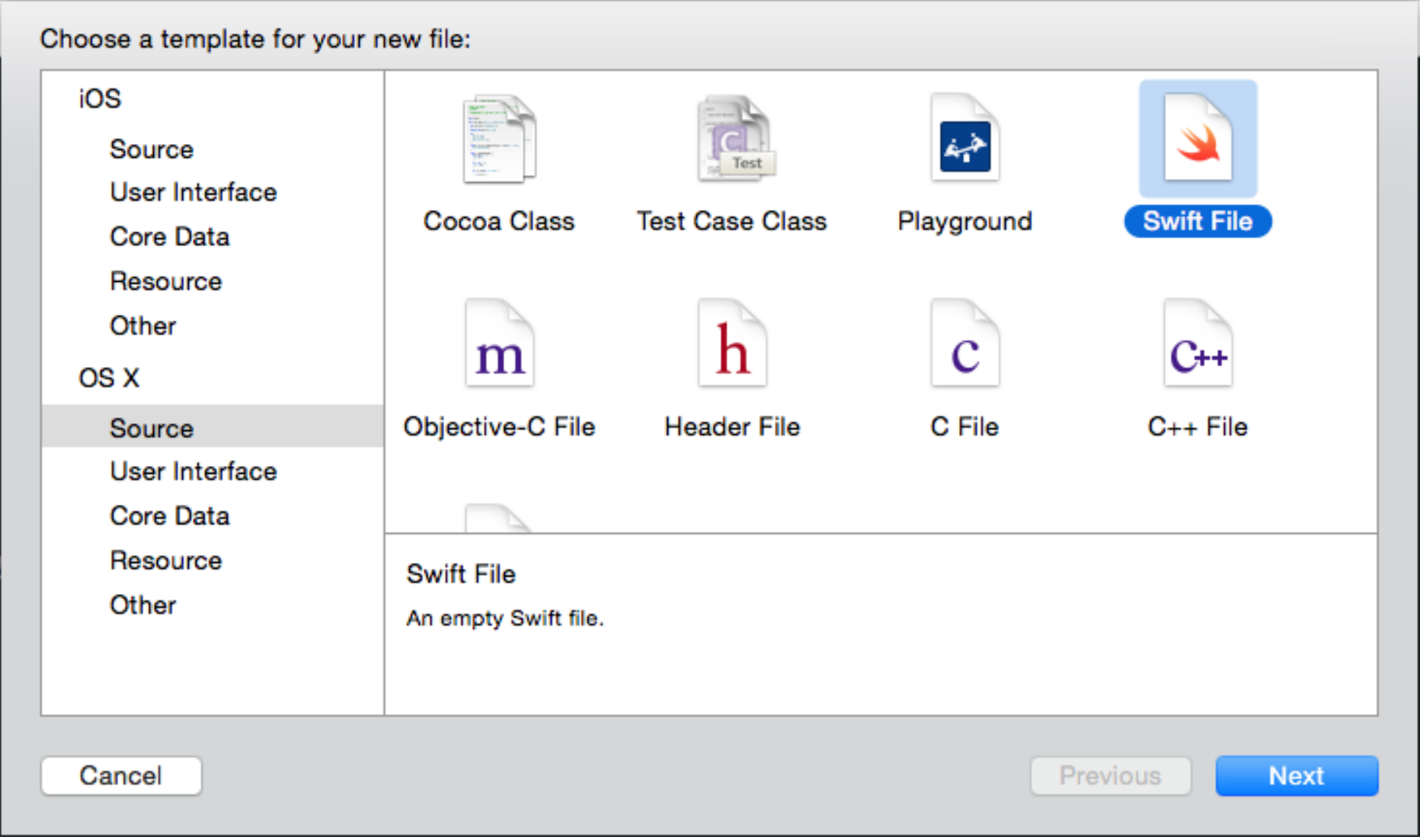


- » HTML5 con el soporte de JSF2: pass through
- » Monta fácilmente tu proyecto con Spring Boot Starter POMs
- » [S.O.L.I.D.] Dependency inversion principle / Principio de inversión de dependencias

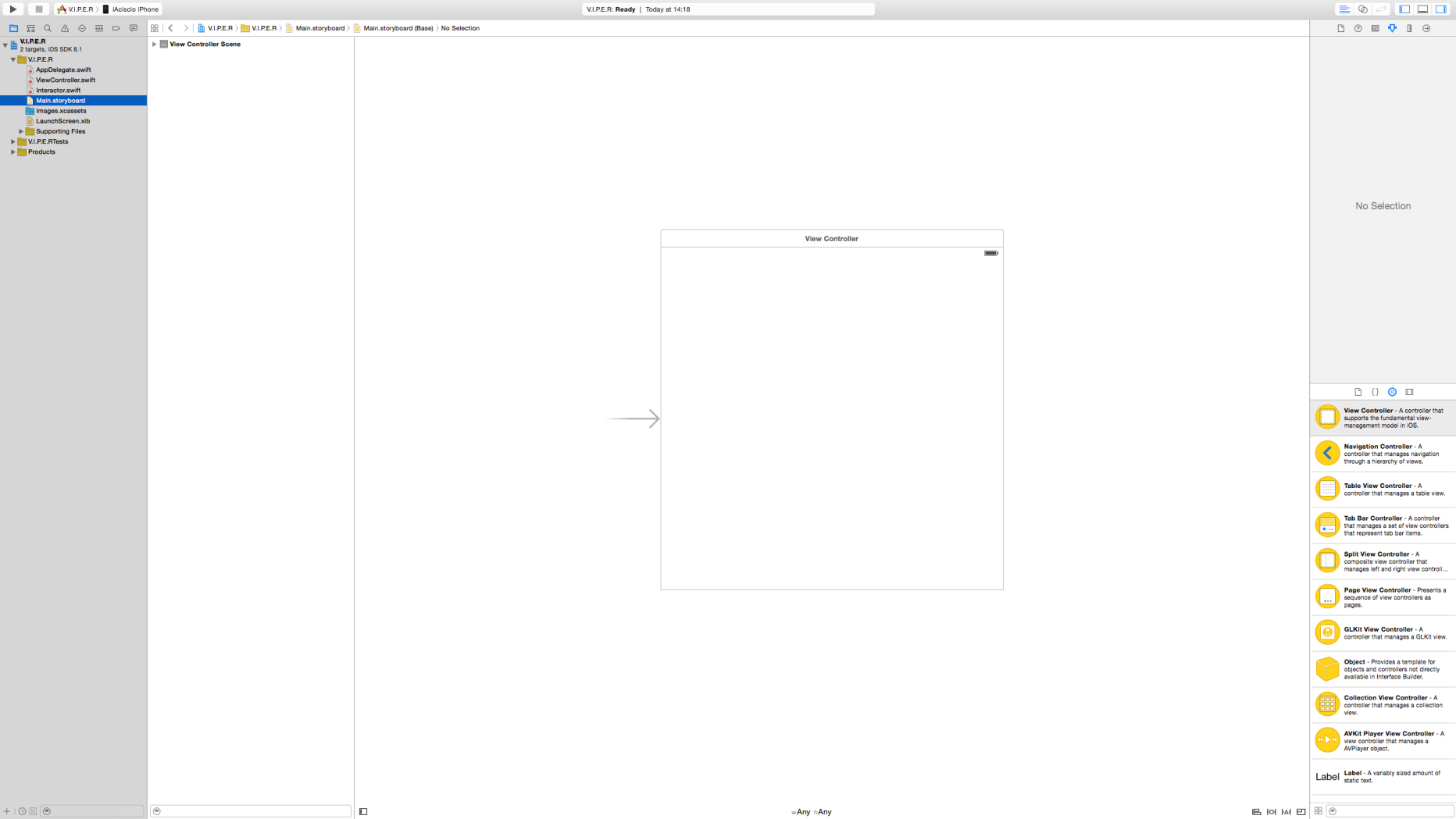
Últimos Tutoriales del Autor

- » Transiciones personalizadas en iOS7
- » Notificaciones locales en iOS.

Vamos a crear nuestros ficheros necesarios para llevar a cabo una aplicación sencilla tipo ABM, presionamos Command-N para crear nuestro fichero swift al que llamaremos Interactor.

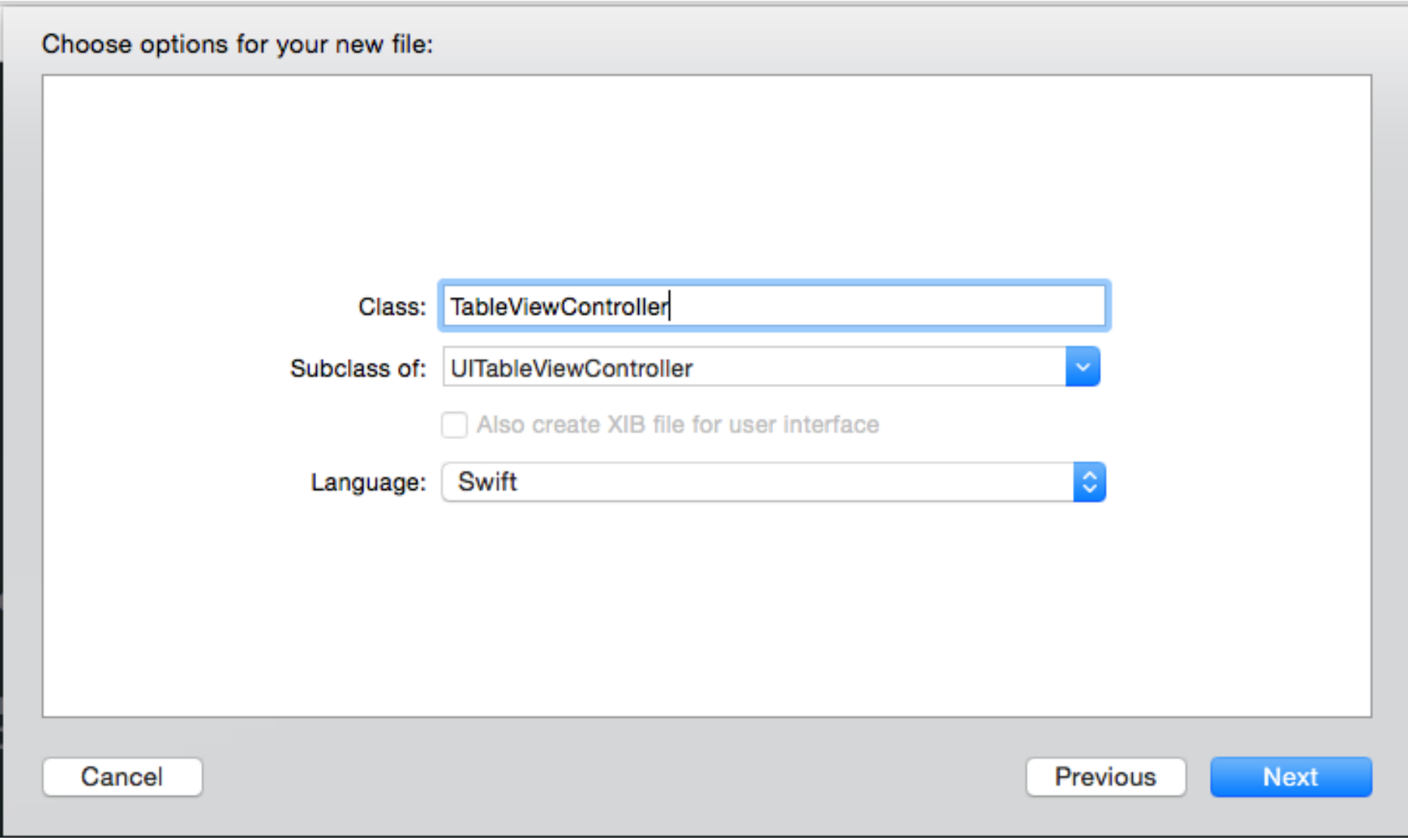


Otro para el Presenter, el Routing, la Entity y la View. Vamos a dejarlos vacíos de momento y nos vamos a nuestro StoryBoard.

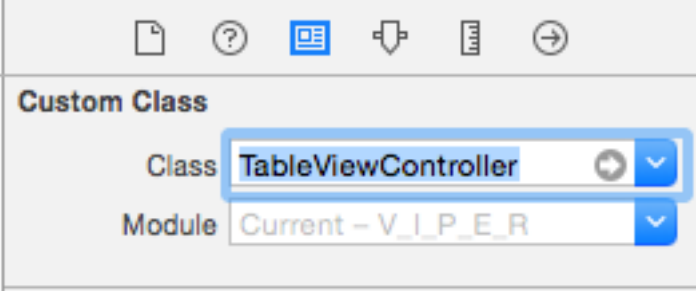


Como veis los nuevos Storyboards han cambiado sustancialmente, ahora se centran en el autolayout, para poder tener un único Storyboard compatible con multiples dispositivos.

Vamos a arrastrar un objeto UITableViewController a nuestra vista y vamos a borrar el controlador que nos venía por defecto. También vamos a borrar nuestro fichero ViewController.swift y a crear una clase que herede de UITableViewController.



Después vamos al StoryBoard, seleccionamos nuestro TableViewController y en el inspector de identidad le decimos que somos una custom class del controlador que acabamos de crear (TableViewController). Es importante desmarcar la casilla de “is initial view controller” ya va ser nuestro “Routing” el que se encarga de la lógica de presentación de pantallas.



Bien ahora en nuestra clase TableViewController en nuestra función viewDidLoad vamos a añadir la siguiente línea:

```
1 | self.navigationItem.rightBarButtonItem = UIBarButtonItem(barButtonSystemItem: .Add, target: self, action: #selector(addNewObject))
```

Ahora tendremos un botón en nuestra barra de navegación con la apariencia de un +. Vamos a añadir en el controlador la función que ejecuta ese botón:

```
1 | func addNewObject() {
2 |
3 | }
```

Bien, ahora vamos a nuestro fichero Entity y vamos a escribir los siguiente:

```
1 | class Persona {
2 |     var nombre:String?
3 |     var apellido:String?
4 |     init() {
5 |
6 |     }
7 | }
```

Esta va ser la clase que representa nuestra entidad, ahora vamos a nuestro fichero View. Recordamos que la “View” en VIPER es básicamente un protocolo que va a adoptar nuestro controlador de vista para poder responder a los eventos generados por el usuario y vamos a escribir una función que recibe los objetos que nos pasa el Presenter:

```
1 | protocol viewProtocol {
2 |     func setListWithObjects(#objects:[String])
3 | }
```

Ahora vamos al fichero Presenter y escribimos lo siguiente:

```
1 | import UIKit
2 |
3 | class Presenter {
4 |     var view:UITableView?
5 |     var interactor:Interactor?
6 |     var routing:Routing?
7 |
8 |     init() {
9 |
10 |    }
11 |
12 |    func addNewObject() {
13 |
14 |    }
15 | }
```

El Presenter contiene una referencia a la vista para que esta pueda cargar los datos que le pasemos, otra a nuestro Interactor que es el que vamos a pedirle la información solicitada por los eventos del usuario y otra a nuestro Routing para cuando tengamos que navegar a otra pantalla.

Ahora vamos a nuestro TableViewController y creamos una variable que referencia a nuestro Presenter para pasarle los eventos del usuario:

```
1 | var presenter:Presenter?
```



Y en el método que responde a nuestro botón más vamos a llamar al Presenter y a su función llamada del mismo modo (addNewObject), que es la que en este caso va a solicitar a nuestro Routing que navegue a otra pantalla. También vamos a crear una variable de tipo Array para alimentar a nuestra tabla:

```
1  var objects:[String]?
2
3  func addNewObject() {
4      presenter!.addNewObject()
5  }
```

También vamos a indicar en los métodos delegados de la tabla su comportamiento y vamos a crear la función que nos solicita la tabla para pintar las celdas:

```
1  override func numberOfSectionsInTableView(tableView: UITableView) -> Int {
2      return 1
3  }
4
5  override func tableView(tableView: UITableView, numberOfRowsInSection section: Int) -
6      return objects.count
7  }
8
9  override func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIn
10     let cell = UITableViewCell(style: UITableViewCellStyle.Default, reuseIdentifier:
11     cell.textLabel.text = objects[indexPath.row];
12     return cell
13 }
```

Hemos indicado que queremos una sola sección y que el numero de filas es igual al numero de elementos de nuestro array, también hemos dicho que el label de la celda escriba directamente el objeto de nuestro array que alimenta la tabla que, como habíamos definido, es un array de Strings.

Recordáis el protocolo que hemos definido antes llamado viewProtocol? Pues vamos a hacer que nuestro controller lo implemente. Únicamente hay que añadirlo después del nombre de la clase de esta forma:

```
1  class TableViewController: UITableViewController,viewProtocol {
2      ...
3  }
```

E implementamos su función indicando que los objetos que recibamos van a ser nuestro array:

```
1  func setListWithObjects(#objects: [String]) {
2
3      self.objects = objects
4      self.tableView.reloadData()
5  }
```

Vamos a nuestro fichero Interactor. Necesitamos una referencia a nuestro Presenter ya que es el que va a recibir la información para pasársela a la vista, escribimos lo siguiente:

```
1  class Interactor {
2
3      var presenter:Presenter?
4
5      init() {
6
7      }
8  }
```

Vamos a crear otro fichero para definir otros dos protocolos más. Al fichero lo llamaremos InputOutput y vamos a definir en él un protocolo InteractorProtocolInput y un protocolo InteractorProtocolOutput. InteractorProtocolInput lo va adoptar el mismo Interactor y va a definir una función que es la que se llama desde el Presenter cuando quiera pasarle información introducida por el usuario. InteractorProtocolOutput lo va a adoptar nuestro Presenter, esta función se llama desde el Interactor cuando tengamos que actualizar la información que recibe el Presenter.

```
1  protocol InteractorProtocolInput {
2
3      func addNewPersonWithData(#nombre:String, apellido:String)
4  }
5
6  protocol InteractorProtocolOutput {
7
8      func updateObjects(#objects:[String])
9  }
```

Y ahora vamos al fichero Routing y escribimos lo siguiente:

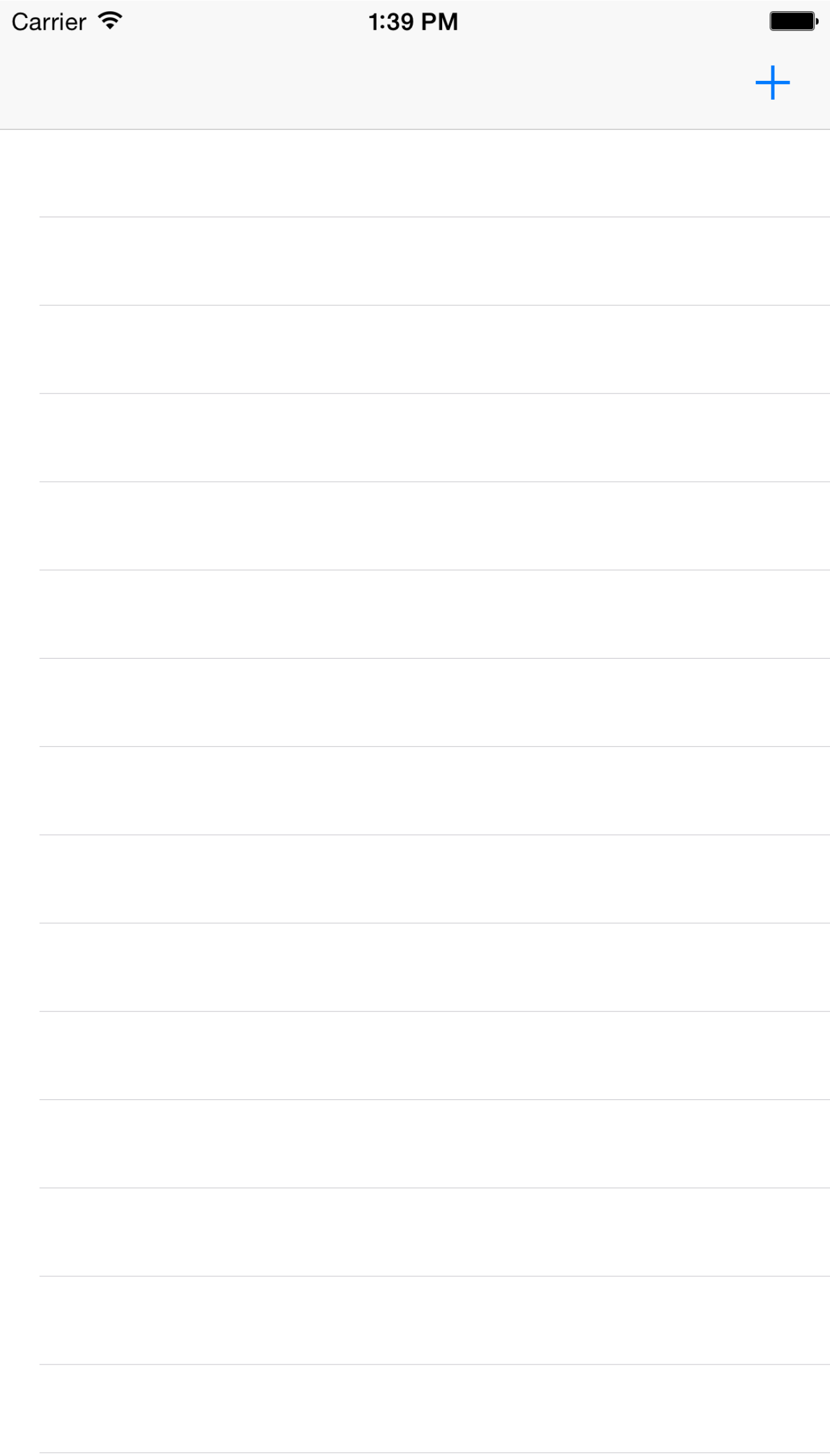
```
1  import UIKit
2
3  class Routing {
4
5      let vc:TableViewController = TableViewController()
6      let presenter = Presenter()
7      let interactor = Interactor()
8      var navigationController: UINavigationController?
9
10     init() {
11         vc.presenter = presenter
12         presenter.view = vc
13         presenter.interactor = interactor
14         presenter.routing = self
15         interactor.presenter = presenter
16         navigationController = UINavigationController(rootViewController: vc)
17     }
18 }
```

Aquí lo que hemos hecho es inicializar nuestro controlador, nuestro Interactor y nuestro Presenter. También hemos dado valor a las variables de cada uno y hemos cargado un Navigation Controller con nuestro TableViewController. Ahora si vamos a nuestro AppDelegate vamos a inicializar nuestra clase Routing para que empiece toda la magia:

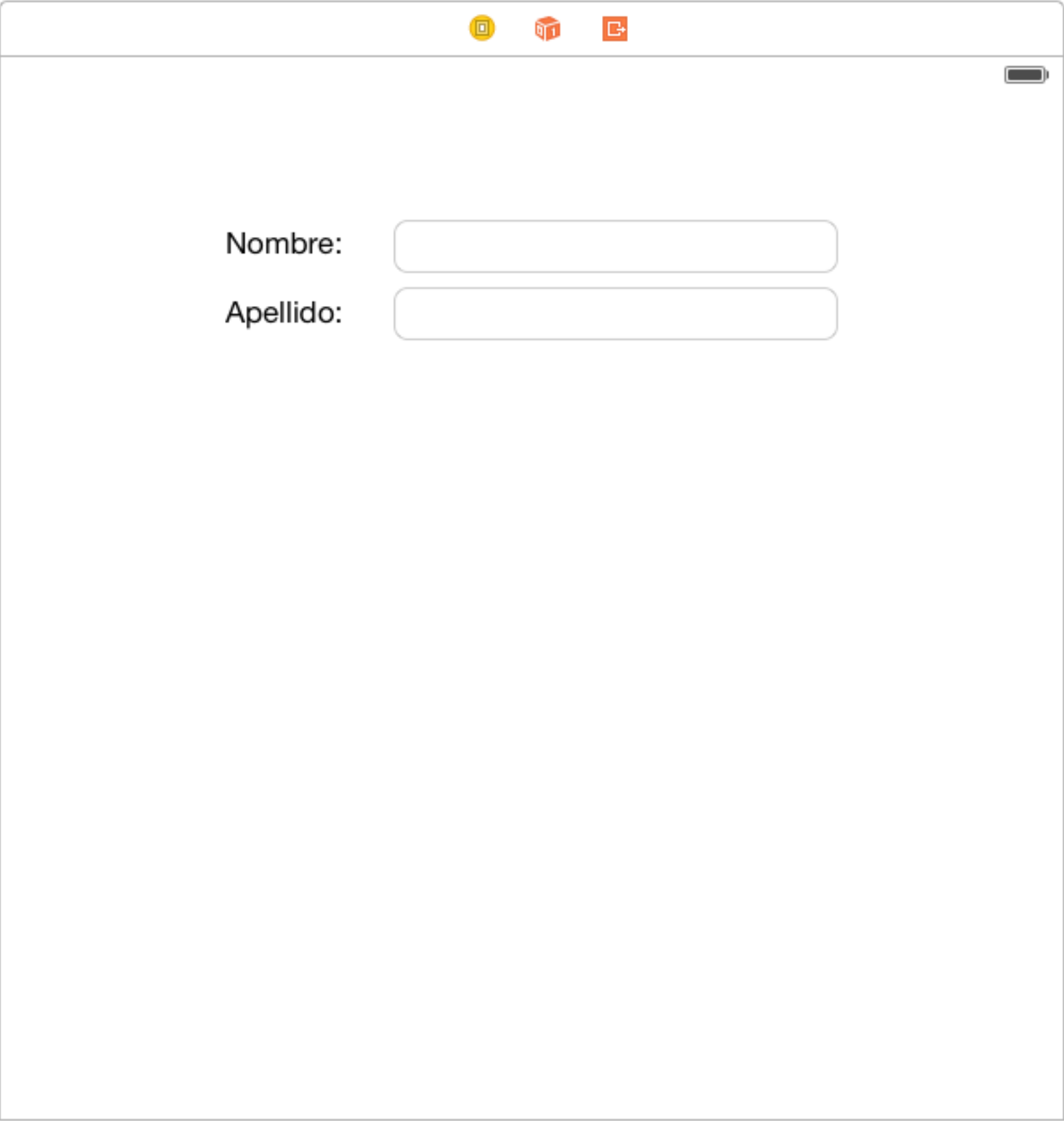
```
1  func application(application: UIApplication, didFinishLaunchingWithOptions launchOptions: [UIApplicationLaunchOptionsKey]?) -> Bool {
2      // Override point for customization after application launch.
3  }
```

```
3
4     let routing = Routing()
5
6     self.window = UIWindow()
7     var screen:UIScreen = UIScreen.mainScreen()
8
9     self.window!.frame = screen.bounds
10    self.window!.rootViewController = routing.navigationController
11    self.window!.makeKeyAndVisible()
12
13    return true
14 }
```

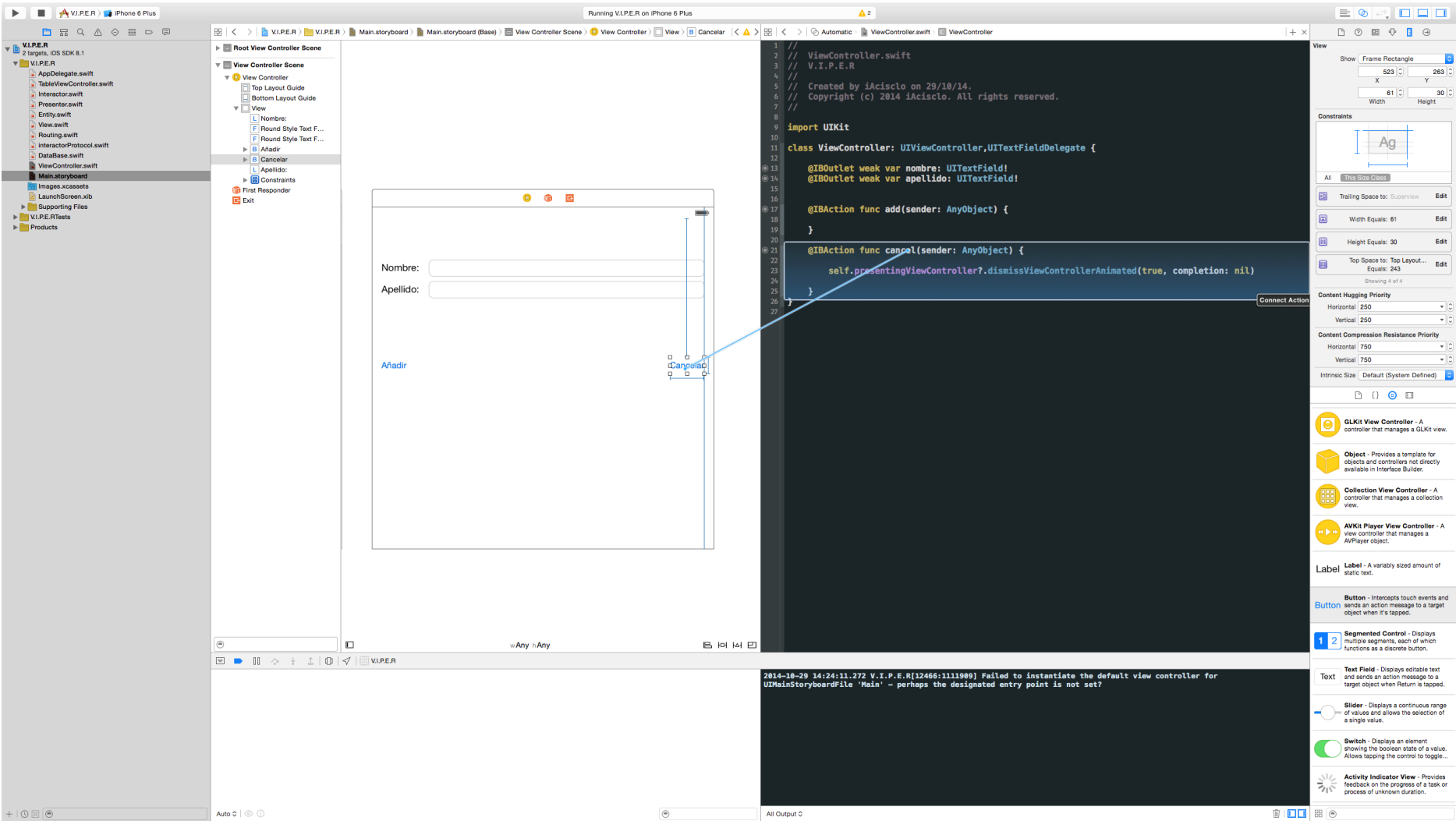
Si compilamos y ejecutamos deberíamos ver una tabla vacía con el botón de añadir en la barra de navegación, pero sin ningún tipo de contenido ni comportamiento de momento.



Lo siguiente que vamos a hacer es crear una pantalla para añadir datos. En nuestro caso nuestra entidad era de tipo Persona con dos variables: nombre y apellido; así que vamos a arrastrar un ViewController a nuestro Storyboard y vamos a dejarlo de la siguiente forma:



Ahora vamos a crear una subclase UIViewController y vamos a conectar nuestros TextFields como outlets y los botones como acciones:



A continuación le damos su identidad:

Custom Class

Hide

Class

ViewController

Module

Current - V\_I\_P\_E\_R

Identity

Storyboard ID

ViewController

Restoration ID

ViewController

☒ Use Storyboard ID

Bien vamos a ver como quedaría nuestro controlador para añadir datos:

```
1 class ViewController: UIViewController, UITextFieldDelegate {
2
3     @IBOutlet weak var nombre: UITextField!
4     @IBOutlet weak var apellido: UITextField!
```

?

```

5     var presenter:Presenter?
6
7     @IBAction func add(sender: AnyObject) {
8
9         presenter?.addNewObjectWithData(name: self.nombre.text, surname: self.apellido
10
11         self.presentingViewController?.dismissViewControllerAnimated(true, completion
12
13     }
14
15     @IBAction func cancel(sender: AnyObject) {
16
17         self.presentingViewController?.dismissViewControllerAnimated(true, completion
18
19     }
20 }

```

Como podeis ver, la función añadir le pasa la información introducida por el usuario al Presenter y cierra la pantalla modal.

Hemos añadido un nuevo método para introducción de datos en el Presenter que a su vez le pasa los datos al Interactor, queda de esta forma:

```

1 func addNewObjectWithData(name n:String, surname s:String) {
2
3     interactor!.addNewPersonWithData(name: n, surname: s)
4 }

```

Y otro en el interactor:

```

1 func addNewPersonWithData(#name:String, surname:String) {
2
3     if (countElements(name) > 0 && countElements(surname) > 0) {
4
5     }
6 }

```

Como veis en el Interactor ya tenemos un nombre más descriptivo de la función por que aquí es donde se conoce la lógica de negocio. En la función también comprobamos que las cadenas que nos llegan sean de una longitud mayor que 0.

Ahora vamos a crear un nuevo fichero Swift que nos va a servir como una DataBase y nos va a quedar así:

```

1 import Foundation
2
3 class DataBase {
4
5     var Personas:[Persona]?
6     init() {
7
8     }
9 }

```

Ahora volvemos al Interactor, completamos la función de añadir Personas y creamos una instancia de DataBase.

```

1 let dataBase:DataBase?
2
3 init() {
4     dataBase = DataBase()
5 }
6
7 func addNewPersonWithData(#nombre:String, apellido:String) {
8
9     if (countElements(nombre) > 0 && countElements(apellido) > 0) {
10
11         let persona = Persona()
12         persona.nombre = nombre
13         persona.apellido = apellido
14
15         if let personas = dataBase?.personas?{
16             dataBase?.personas?.append(persona)
17         }else{
18             dataBase?.personas = [Persona]()
19             dataBase?.personas?.append(persona)
20         }
21     }
22 }

```

Lo que hemos hecho es crear una instancia de Persona en el método init de nuestro Interactor para disponer de una contante de tipo DataBase. En la función para añadir los datos creamos una instancia de persona y le asignamos los valores que ha introducido el usuario en los campos de texto. Finalmente en la variable de tipo array de nuestra DataBase añadimos este objeto Persona.

Ahora vamos a crear una función en nuestro Interactor que llame actualize la información a nuestro Presenter. Recordar que al Presenter únicamente hay que pasarle estructuras de datos simples por que no puede conocer los objetos de negocio, así que en dicha función transformamos nuestros objetos de negocio en Strings de la siguiente forma:

```

1 func updateList() {
2
3     var arrayPersonas = [String]()
4     for persona in dataBase!.personas! {
5
6         arrayPersonas.append(persona.nombre! + " " + persona.apellido!)
7     }
8
9     presenter!.updateObjects(objects: arrayPersonas)
10 }

```

Y ahora llamamos a esta función al final de la función addNewPersonWithData:

```

1 self.updateList()

```

Como nuestro Presenter tenía que adoptar el protocolo InteractorProtocolOutput tiene que implementar su función:

```

1 func updateObjects(#objects: [String]) {

```



```
2         view!.setListWithObjects(objects: objects)
3     }
4 }
```

Cuando el Interactor llama a esta función el Presenter se limita a actualizar la vista.  
Para teminar con el Presenter vamos a añadir la función que se llama desde la vista para mostrar la pantalla modal de introducción de datos:

```
1 func addNewObject() {
2
3     routing!.openAddView()
4 }
```

Y ahora vamos al routing e implementamos la función que nos abre dicha pantalla:

```
1 func openAddView() {
2
3     let storyboard:UIStoryboard = UIStoryboard(name: "Main", bundle: NSBundle.mainBund
4     let addVC:ViewController = storyboard.instantiateViewControllerWithIdentifier("Vie
5     addVC.presenter = self.presenter
6
7     vc.presentViewController(addVC, animated: true, completion: nil)
8 }
```

Hemos instanciado nuestro Storyboard y hemos solicitado presentar moralmente nuestro controlador “ViewController”.  
Si compilamos y ejecutamos ya podemos añadir personas a nuestra tabla.

En resumen, con esta arquitectura podemos aislar de forma optima nuestras clases para que tengan una única responsabilidad. Para un aplicación de ejemplo como la que hemos visto carece de sentido hacer este patrón, pero cuando la aplicación tiene mucha funcionalidad y nuestro controladores de vista se ven sobrecargados esta es una solución muy práctica.

Podéis descargar el proyecto de github en este [enlace](#).

## A continuación puedes evaluarlo:

[Regístrate para evaluarlo](#)



## Por favor, vota +1 o compártelo si te pareció interesante

Share | +1 0 +1 0

Anímate y coméntanos lo que pienses sobre este **TUTORIAL**:

» **Registrate** y accede a esta y otras ventajas «



Esta obra está licenciada bajo licencia [Creative Commons de Reconocimiento-No comercial-Sin obras derivadas 2.5](#)

PUSH THIS

Page Pushers

Community

Help?

-----

no clicks

0 people brought clicks to this page

+

+

+

+

+

+

+

+

powered by [karmacracy](#)

Copyright 2003-2014 © All Rights Reserved | [Texto legal y condiciones de uso](#) | [Banners](#) | [Powered by Autentia](#) | [Contacto](#)

W3C

XHTML 1.0

W3C

CSS

XML

RSS

XML

ATOM