Desarrollo de aplicaciones para dispositivos iOS

Xavier Pereta

Introducción

En este material, empezaremos introduciendo los conceptos básicos para poder desarrollar aplicaciones nativas para iOS. A continuación, comentaremos algunas características básicas del sistema operativo, sus dispositivos, el entorno de desarrollo y las herramientas que usaremos para desarrollar nuestras aplicaciones. Además, aprenderemos a utilizar el nuevo lenguaje de programación Swift.

Todos estos conocimientos los aplicaremos más adelante en el terreno práctico, puesto que desarrollaremos una pequeña aplicación y, finalmente, propondremos fuentes adicionales para ampliar nuestros conocimientos.

1. El ecosistema iOS

Dispositivos iOS

Los dispositivos iOS son aquellos dispositivos de Apple que ejecutan el sistema operativo iOS. Los podemos categorizar por familias: iPhone, iPod Touch, iPad, Apple TV y, ahora también, Apple Watch.

Tanto el Apple TV como el Apple Watch utilizan sistemas operativos derivados del sistema operativo iOS, tvOS y watchOS, creados específicamente para ellos, por lo que hay bastantes diferencias.

iPhone

El primer iPhone se lanzó en 2007 y solo estaba disponible en Estados Unidos, pero, debido a su popularidad, empezó a haber gente que lo compraba allí y se lo llevaba a otros países.

Tras este primer iPhone han salido al mercado muchos modelos nuevos, concretamente uno por año. Habitualmente, cada dos años aparece una nueva generación, y cada año se renueva el dispositivo.

Los dispositivos de cada generación se distinguen con un número (por ejemplo, iPhone 4 y iPhone 5), mientras que a los modelos renovados de una misma generación se les añade una letra (por ejemplo, iPhone 5S y iPhone 5C).

El salto generacional suele traer más novedades que el salto de modelo, es decir, entre el iPhone 6 y el iPhone 7 hay más novedades que entre el iPhone X y el iPhone XS.

iPod Touch

En paralelo a los dispositivos iPhone fue creciendo la familia de los iPod Touch, la evolución de la antigua familia de los iPod, orientada inicialmente a escuchar música, pero con la funcionalidad Touch y, debido al *boom* de las aplicaciones, con muchas funciones.

A grandes rasgos, los iPod Touch tienen características similares a los iPhone, pero no tienen la capacidad de uso de teléfono móvil. Muchas veces también disponen de menos funciones, con lo cual el precio también es inferior. Hoy en día hay seis generaciones de iPod Touch.

De cara al desarrollo, es importante tener presente los iPod Touch, puesto que, aunque no se les da tanta importancia como a la familia de los iPhones, suponen una gran parte de los dispositivos de los usuarios. Así mismo, muchos de ellos tienen algunas características peculiares: generaciones sin micrófonos, sin cámaras, sin flash, con botones en posiciones diferentes que los iPhone, etc.

iPad

Del mismo modo que el iPhone, el iPad fue un producto muy innovador en el momento de su lanzamiento, pero inicialmente recibió algunas malas críticas, como que se trataba de un iPhone grande y que no podría sustituir al ordenador de toda la vida. El tiempo no dio la razón a las críticas, y ahora encontramos muchas marcas que comercializan la misma idea de tableta.

El primer iPad se presentó en 2010 y, actualmente, ya se han vendido más de 350 millones y se han lanzado seis generaciones. A estas debemos sumar tres generaciones más de la versión iPad Mini y una generación de iPad Pro. Por un lado, la versión iPad Mini es una tableta más pequeña (7,9") que el modelo normal (9,7"). Por otro lado, el iPad Pro añade un nuevo tamaño de pantalla (12,9") al iPad estándar, además de un soporte para el Apple Pencil.

Desde sus inicios, los iPad se han presentado con dos versiones diferenciadas: una, más económica, con conectividad wifi y otra que permite la conectividad móvil y que ofrece la posibilidad de añadir una SIM para poder tener conectividad en cualquier lugar con cobertura móvil.

Apple TV

Apple TV es un dispositivo pensado para estar junto al televisor que permite visualizar toda una serie de contenido disponible desde Apple (películas, series, canales en línea, etc.). Además, permite visualizar el contenido del ordenador en iTunes. Una de las características más destacadas es que dispone de AirPlay, es decir, que permite acceder en directo al contenido de un dispositivo móvil para verlo en el televisor de una manera muy cómoda.

Actualmente existen cinco generaciones de este dispositivo. La segunda y la tercera generación de Apple TV funcionan con una versión modificada de iOS. La primera generación utilizaba una variación de Mac OS X, por lo que, técnicamente, no es un dispositivo iOS. El primer modelo data de 2007, a pesar de que entonces Apple consideraba que era un producto de los que llaman *hobby* y no le prestaba demasiada atención.

En septiembre de 2015 se presentó la cuarta generación de Apple TV, y a partir de entonces el sistema operativo de este dispositivo es tvOS. Este incluye su propia App Store y permite desarrollar aplicaciones propias.

Apple Watch

El Apple Watch es el último dispositivo presentado dentro de la gama iOS y la primera generación de una nueva familia. En esta versión es necesario disponer de un dispositivo iOS compatible para poder usar muchas de las funciones del reloj, puesto que este tiene una versión de iOS muy limitada.

Prácticamente la única interactividad permitida inicialmente por Apple se daba en el campo de las notificaciones.

El dispositivo está orientado, sobre todo, al cuidado de la condición física y al control de la salud del usuario, puesto que incorpora un detector del ritmo del corazón, así como herramientas de seguimiento de movimientos para controlar la actividad del usuario.

A pesar de que se puede utilizar sin iPhone, la mayoría de sus aplicaciones requieren disponer de un iPhone 5 o superior.

2. El lenguaje Swift

Introducción a Swift

Swift es un lenguaje de propósito general creado por Apple y distribuido bajo una licencia de código abierto Apache 2.0. Fue presentado en 2014 para reemplazar Objective-C como lenguaje utilizado para la construcción de aplicaciones para iOS y macOS. Desde entonces, ha evolucionado hasta la versión 4 con numerosas mejoras en las que ha participado la comunidad de desarrolladores. Hasta el momento, Objective-C y Swift conviven como lenguajes de desarrollo para iOS, pero Apple está apostando claramente por Swift, por lo que es previsible que, en el futuro, Objective-C quede relegado al mantenimiento de aplicaciones antiguas y todos los desarrollos nuevos se creen en Swift.

Así mismo, este lenguaje está diseñado desde el principio para ser expresivo y eficiente a la hora de escribir el código. Como veremos a lo largo de este apartado, Swift tiene unas características que posibilitan que un código sea claro y libre de errores. Su sintaxis moderna y clara lo hace más accesible que Objective-C, por lo que es mucho más fácil iniciarse en el desarrollo de aplicaciones. Sin embargo, no debemos confiarnos, ya que no es, en absoluto, un lenguaje simple, sino que tiene mucha potencia y muchos matices. Además, introduce algunos conceptos que veremos a continuación y que seguramente no hayamos visto antes.

En la página oficial de Swift podemos encontrar la documentación oficial del lenguaje, así como información de cómo está organizada la comunidad que se encarga de su mantenimiento y evolución.

Sintaxis

La mejor manera de aprender un lenguaje es utilizándolo, y para ello Swift dispone de Swift Playgrounds, una interfaz de programación de Apple que nos permite ver en tiempo real el resultado de la ejecución de nuestro código. Es una opción ideal para hacer pruebas y aprender de una manera muy fácil.

Playgrounds está disponible en macOS como un tipo de proyecto en Xcode y también como una aplicación para iPad.

Para crear un nuevo Playground podemos abrir Xcode y elegir la opción «Playground» del menú «New». Por ello la mejor manera de seguir este apartado es tener abierto Swift Playgrounds en nuestro Mac o iPad e ir probando todos los ejemplos de código a medida que avanzamos.

En Swift el clásico ejemplo «Hello world!» consiste únicamente en esta línea:

```
print("Hello world!")
```

No es necesario nada más ni importar ninguna biblioteca adicional.

A lo largo de este apartado vamos a ver, mediante ejemplos, un resumen de las características más importantes de Swift y cómo utilizarlas.

Valores básicos

Existen diferentes tipos de variables en Swift. Las más comunes son las siguientes:

- *String*, para cadenas de caracteres como «Hello».
- *Int*, para los enteros, como 1, 2, 3, 4.
- *Double*, para los números con decimales, como 1,3 o 0,5.
- *Bool*, para indicar cierto (*true*) o falso (*false*).

En Swift las variables se declaran con la palabra *var* delante, y las constantes, con *let*. Esto permite indicar al compilador qué valores se van a modificar (variables) y qué valores solamente se van a inicializar una vez (constantes). Esta distinción va a permitir identificar de una forma clara qué

valores se quieren actualizar en el código y va a ahorrar toda una categoría de errores relacionados con la actualización de variables de forma inadvertida.

```
var universalResponse = 42
universalResponse = 13
let someConstant = 10
```

Hay que aclarar que no es necesario que las constantes tengan valor en tiempo de compilación, pero, si se les da, solo se puede hacer una vez. Además, si son objetos, a pesar de ser constantes, podremos modificar sus propiedades, pero lo que no podremos hacer es asignarles un nuevo objeto.

Una constante o variable debe ser del mismo tipo que el valor que se le va a asignar. Pero fijémonos que en el ejemplo anterior no le hemos dicho de qué tipo son cada una. En Swift el compilador puede deducir el tipo de las variables y constantes si les proporcionamos un valor cuando las creamos. En el ejemplo anterior el compilador deduce que universalResponse es un entero porque el valor inicial que le asignamos es un entero.

También podemos especificar directamente el tipo si lo escribimos después del nombre de la variable separado por dos puntos:

```
let someDouble: Double = 42
```

En Swift se prefiere la deducción de tipos, por lo que solamente deberíamos añadir el tipo a la declaración si es imprescindible, ya que, por ejemplo, no le asignamos un valor inicial a la variable.

```
var anotherVariable: String
anotherVariable = "test"
```

El siguiente ejemplo nos dará un error, puesto que estaremos asignando dos veces un valor a una misma constante:

```
let someString = "value1"
someString = "another value"
```

Hay que saber que nunca se realiza una conversión de tipos implícita. Si necesitamos convertir un valor a un tipo distinto tendremos que crear una nueva instancia del tipo que necesitamos. Como en el ejemplo que sigue, en el que tenemos que crear una *string* a partir del Int answer:

```
let message = "The answer is "
let answer = 42
```

```
let answerMessage = message + String(answer)
```

Si intentamos ejecutar la siguiente línea nos dará un error:

```
let answerMessage = message + answer
```

Por cierto, tenemos una manera más sencilla de componer *strings*, escribiendo el valor entre paréntesis precedido de una barra invertida (\) :

```
let answer = 42
let answerMessage = "The answer is \(answer)"
```

Si estamos utilizando Playgrounds, iremos viendo los resultados de nuestro código en la parte derecha de la pantalla. Si no, para mostrarlo en la consola en Xcode utilizaremos la palabra *print*:

```
print(answerMessage)
```

La función *print* nos permite imprimir el valor de una variable en la consola. La podemos utilizar para generar un registro de las operaciones que realiza nuestro código y verificar que funciona correctamente, aunque en el subapartado dedicado a la depuración de aplicaciones veremos mecanismos más efectivos.

```
let firstResult = 4 + 6
print("The first operation has completed and the result is \(firstResult)")
let secondResult = firstResult + 32
print("The second operation has completed and the result is \(secondResult)")
```

Arrays

Para crear *arrays* utilizaremos los corchetes ([ ]) y accederemos a sus elementos usando el índice del elemento entre corchetes, también.

```
let devices = ["iPhone", "iPad", "Apple TV", "Apple Watch"]
let phone = devices[0]
```

Para crear un *array* vacío utilizaremos esta sintaxis de inicialización:

```
var emptyArray = [String]()
emptyArray.append("Some element")
```

Diccionarios

Los diccionarios son parecidos a los *arrays* pero permiten especificar una clave para acceder a cada uno de sus elementos. Esta clave identifica el elemento dentro del diccionario, por lo que debemos proporcionarla tanto al inicializarlo como cuando deseemos recuperarlo.

```
var animalSize = ["dog":"small", "horse":"big"]
animalSize["pig"] = "Average"
var countryPopulation = ["Japan":127.3, "Germany":80.6, "Peru":29.7, "Bhutan":0.7]
let populationOfPeru = countryPopulation["Peru"]
```

Igual que con los *arrays*, podemos crear un diccionario vacío con la sintaxis de inicialización.

```
let emptyDictionary = [String:String]()
```

Control de flujo

Las palabras *if* y *switch* sirven para controlar condiciones, y *for-in*, *for*, *while* y *repeat-while*, para bucles. En todos los casos los paréntesis en la condición son opcionales, pero las llaves son obligatorias.

```
let individualScores = [75, 43, 103, 87, 12]
var teamScore = 0
for score in individualScores {
    if score > 50 {
        teamScore += 3
    } else {
        teamScore += 1
    }
}
print(teamScore)
```

La condición debe ser explícitamente de tipo booleano, por lo que no podemos usar resultados que no sean booleanos y esperar que se comparen implícitamente con cero o con una *string* vacía, como ocurre en otros lenguajes. Esto es muy útil, porque hace que el código sea más claro y nos ahorra toda una categoría de errores causados por la confusión entre una asignación y una comparación.

Todos estos casos van a dar error de compilación:

```
if "" {
}

if 0 {
}

var boolVariable = true

if boolVariable = true {
}
```

El otro condicional, el *switch*, también incorpora mejoras respecto a lo que seguramente estamos acostumbrados en otros lenguajes. Además, no es necesario añadir *break* en cada uno de los casos para evitar que salte directamente a la siguiente opción, pero si excepcionalmente necesitamos que la ejecución salte al caso inferior podemos utilizar la instrucción *fallthrough*. En los distintos casos podemos comparar todo tipo de variables y también incluir más de una condición para cada uno.

```
let animal = "dog"
var animalSound:String

switch animal {
case "dog":
    animalSound = "Woof"
case "cat", "cat-small":
    animalSound = "Meow"
default:
    animalSound = "No idea"
}

print(animalSound)
```

Utilizaremos *for-in* para iterar los elementos de una secuencia, ya sea un *array*, un rango de números o los caracteres de una *string*.

```
let colors = ["red","blue","yellow"]
for color in colors {
    print("I like \(color)")
}
```

No existe el clásico bucle *for* al estilo del lenguaje C con inicialización o condicional, un incremento que seguramente nos resulta familiar de otros lenguajes. Pero obtenemos el mismo resultado con el *for-in* definiendo secuencias numéricas como la que veremos a continuación.

En Swift podemos definir rangos con la siguiente sintaxis: *inicio...final*, o bien *inicio..<final*, si no deseamos incluir el último elemento, por ejemplo:

```
let oneToTen = 1...10
print(oneToTen)

let oneToNine = 1..<10
print(oneToNine)
```

Y este sería el ejemplo de *for-in* utilizando una secuencia numérica:

```
for i in 1...5 {
    print("Element \(i)")
}
```

Optionals

Una de las características más importantes de Swift son los parámetros opcionales, que nos permiten gestionar de manera más clara los casos en los que un objeto o variable no contiene ningún valor, el valor *nil* (valor *null* o *nothing*, en otros lenguajes). Precisamente en la mala gestión de estos casos tenemos una fuente de innumerables errores de programación.

En este enlace podemos ver una explicación de los peligros que conlleva la gestión de valores *null*: https://www.lucidchart.com/techblog/2015/08/31/the-worst-mistake-of-computer-science/.

Como veremos, el propio lenguaje Swift ya nos proporciona un mecanismo para tratar de forma segura estos casos. De esta manera, a diferencia de otros lenguajes, no tenemos que inventar nuestra propia estrategia y, además, al ser parte del lenguaje, será la misma para todo el código que nos vayamos encontrando escrito en Swift.

Para declarar un *optional* solamente tenemos que añadir un interrogante (?) después del tipo:

```
var variable: Int?
variable = 20
variable = nil
```

Pero si el tipo fuera Int (no *optional*), no le podríamos asignar el valor *nil*, por lo que el código ya no compila:

```
var variable: Int
```

```
variable = 20
variable = nil
```

Fijémonos en que, aunque podemos marcar una variable como *optional* para que contenga el valor *nil*, no se puede intercambiar con un valor *nil* de otro tipo *optional*, por lo que dará error.

```
var optionalInt: Int? = 10
optionalInt = nil
var optionalString: String?
optionalString = optionalInt
```

La acción de transformar una variable *optional* en una variable no *optional* se denomina *unwrap*, y su sintaxis habitual es la siguiente:

```
var optionalAge:Int? = 20
print(optionalAge)

if let age = optionalAge {
  print(age)
}
```

Si ejecutamos este ejemplo veremos que el primer *print* nos muestra el texto «Optional(20)», puesto que estamos tratando con una variable *optional*. Si queremos acceder al valor real tendremos que hacer un *unwrap* antes.

Extraemos el valor de *optionalAge* y se lo asignamos a la constante *age*. Dentro del *if* ya podemos trabajar con esta constante *age* con la seguridad de que va a contener un valor. En el caso que *optionalAge* no contenga ningún valor, el código de dentro del *if* no se ejecutará.

Otra manera de hacer el *unwrap* es añadiendo un signo de exclamación (!); es lo que se denomina *forced unwrap*. Por ejemplo:

```
var optionalAge:Int? = 20
if optionalAge != nil {
  print(optionalAge!)
}
```

Solamente haremos un *force unwrap* cuando tengamos la seguridad de que la variable va a contener un valor en tiempo de ejecución, ya que si lo intentamos con una variable que contiene *nil*, se producirá un error. El uso de *force unwrap* debería ser excepcional en nuestro código.

En Swift, podemos continuar preguntando si una variable contiene un valor *nil*, como haríamos en un lenguaje que no soporta *optionals*, pero nos resultará un código poco natural.

```
var optionalAge:Int? = 20
if optionalAge != nil {
  print("Your age is \(optionalAge!)")
} else {
  print("Your age could not be determined")
}
```

Es preferible esta otra versión:

```
var optionalAge:Int? = 20
if let age = optionalAge {
  print("Your age is \(age)")
} else {
  print("Your age could not be determined")
}
```

Debemos tener en cuenta que, además de los *optionals* que declaremos en nuestro código, vamos a tener que manejarnos con los que vendrán como valores de retorno de las funciones de los *frameworks* de iOS. Por ello, es muy importante que nos habituemos a tratarlos correctamente tal como hemos visto con el *unwrapping* y como veremos ahora con el *optional chaining*.

Optional chaining

Este concepto surge de la necesidad de acceder a propiedades y funciones de un objeto que puede que no contenga un valor en un momento dado. Si contiene un valor, se accede a la propiedad o se llama a la función correspondiente, pero si no, nos va a devolver *nil*.

Habitualmente se utiliza para llamar a una función de un objeto que está al final de una cadena de llamadas sin tener que ir comprobando cada una de las llamadas para asegurarnos de que cada uno de ellos tiene un valor.

Para utilizar *optional changing* añadiremos un interrogante (?) después de *optional*.

```
var myMacBook: MacBook?
myMacBook = findMacBook()
let diskSize = myMacBook?.memory?.size
print(diskSize)
```

Fijémonos en que si la variable *myMacBook* o su propiedad *memory* fuesen *nil*, el código se ejecutaría sin errores, pero la constante *diskSize* tomaría el valor *nil*.

Si no tuviéramos *optional chaining*, deberíamos escribir este código de una manera parecida a esta:

```
if let myMacBook = findMacBook() {
  if let myMemory = myMacBook.memory {
    if let diskSize = myMemory.size {
      print(diskSize)
    }
  }
}
```

Optionals con *unwrap* implícito

Los *optionals* con *urwrap* implícito son un tipo de *optionals* en los que no es necesario añadir una *exclamación* (!) para hacer el *unwrap*. Están pensados para simplificar el código y evitar tener que hacer continuamente *unwrap* a lo largo de todo el código. Solamente los usaremos para aquellas variables que utilizamos muy a menudo, que pueden no contener un valor durante la fase de inicialización del código y que estamos completamente seguros de que siempre tendrán un valor después de esta fase de inicialización.

Merecen una mención especial en este apartado porque que se utilizan para definir las variables que nos van a permitir acceder a los elementos definidos en el *storyboard* de las pantallas en nuestra aplicación, por lo que los vamos a encontrar muy a menudo. Se declaran con una exclamación (!) después del tipo, en vez de una interrogación (?). Como vemos en este ejemplo, su *unwrap* es implícito:

```
var optionalInt:Int!
optionalInt = 10
let nonOptionalInt:Int
nonOptionalInt = optionalInt
```

Se trata solamente de un cambio en la notación para simplificar el uso, y solamente debe usarse en los casos mencionados, porque, si contienen *nil*, el *unwrap* fallará igualmente aunque sea implícito.

```
var optionalInt:Int!
optionalInt = nil
let nonOptionalInt:Int
nonOptionalInt = optionalInt
```

Este es el ejemplo típico que vamos a encontrar en nuestras aplicaciones:

```
class ViewController: UIViewController {
    @IBOutlet weak var nextButton: UIButton!
}
```

Funciones

Para declarar una función utilizaremos la palabra clave *func*. Y para llamarla usaremos su nombre con una lista de parámetros entre paréntesis. Para especificar el tipo de retorno utilizaremos ->.

```
func greet(person: String, day: String) -> String {
  return "Hello \(person), today is \(day)."
}

greet(person: "Alice", day: "Monday")
```

En Swift los nombres de los parámetros se utilizan para identificarlos en el momento que vayamos a llamar la función. También podemos especificar una etiqueta alternativa, es decir, identificar el parámetro al llamar a la función con un nombre distinto al que hemos usado en la declaración, o bien especificar _ como etiqueta para obviarla.

```
func greet(_ person: String, on day: String) -> String {
  return "Hello \(person), today is \(day)."
}

greet("Bob", on: "Tuesday")
```

Además, podemos utilizar una tupla para devolver más de un valor en nuestras funciones:

```
func howIsTheWeather() -> (temperature: Int, wind: String) {
  return (20, "Strong winds")
}

let weather = howIsTheWeather()
print("Temperature: \(weather.temperature) ºC)")
print("Wind: \(weather.wind)")
```

También podemos tener un número indefinido de parámetros. En este caso los tendremos disponibles en un *array* dentro de nuestra función:

```
func sumAll(numbers: Int...) -> Int {
  var total: Int = 0
 for number in numbers {
    total += number
  }
 return total
```

```
}

let resultado = sumAll(numbers: 1,2,3,4)
print(resultado)
```

Closures

Definiremos una *closure* de forma similar a una función pero omitiendo el nombre y con la palabra clave *in* antes del cuerpo:

```
{(value1: Int, value2: Int) -> Int in
  return value1 + value2
}
```

Uno de los usos de las *closures* en Swift y de los *frameworks* de iOS es pasarlos como parámetro. Por ejemplo, en este caso se utilizan para definir la operación de mapeado de los valores del *array numbers*:

```
let numbers = [1,2,3,4,5]

let mapped1 = numbers.map({ (number: Int) -> Int in
  let result = 3 * number
  return result
})
print(mapped1)
```

La sintaxis de las *closures* permite omitir la mayoría de sus elementos para hacerlas más compactas.

Podemos omitir también los tipos, tanto de los parámetros como del valor de retorno, e incluso la palabra clave *return*:

```
let mapped2 = numbers.map({ number in 3 * number })
print(mapped2)
```

También podemos referirnos a los parámetros por su número para generar expresiones aún más compactas, lo que deja la *closure* en su mínima expresión.

```
let mapped3 = numbers.map { $0 * 3 }
print(mapped3)
```

Objetos y clases

Crearemos una clase con la palabra clave *class*. Las propiedades de la clase las declararemos como variables y constantes, que funcionan de la misma manera. Las funciones de la clase también las declararemos como acabamos de ver.

```
class Car {
    var numberOfWheels = 4
    var name = ""
    var passengers = 0

    init(name:String) {
        self.name = name
    }

    func description() -> String {
        return "This is car is a \(name) and has \(passengers) passengers."
    }
}

let sportsCar = Car(name: "Ferrari")
sportsCar.passengers = 3
print(sportsCar.description())
```

Utilizaremos *init* para inicializar una instancia de la clase. La palabra clave *self* la utilizaremos para referirnos a la instancia de un objeto de nuestra clase.

```
func pickUp(passengers:Int) {
    self.passengers = self.passengers + passengers
}
```

Por ejemplo, en esta función *pickUp* de la clase *Car*, *passengers* es el parámetro de la función y, en cambio, *self.passengers* es la variable de la instancia.

Herencia

Para especificar que una clase hereda de otra clase, añadiremos el nombre de esta clase base después de dos puntos (:). En Swift, a diferencia de otros lenguajes, no es necesario que todas las clases hereden siempre de una clase base.

```
class Vehicle {
    var numberOfWheels = 0
    var passengers = 4
}
```

```
class Car: Vehicle {
    init(name:String) {
        self.name = name
        self.numberOfWheels = 4
        self.passengers = 4
    }
}

let myCar = Car()
print("My car has \(myCar.numberOfWheels) wheels")
```

Protocolos

Un protocolo define un esquema de funciones y propiedades enmarcadas en la realización de una tarea o un objetivo en concreto. Es un concepto parecido al de interfaces de otros lenguajes de programación. Un tipo que cumpla con todos los requisitos de un protocolo se dice que adopta dicho protocolo.

La sintaxis de un protocolo es muy similar a la de una clase:

```
protocol CalculateProtocol {
  var result: Strinng { get }
  func calculate()
}
```

Para indicar que una clase adopta un protocolo, lo añadiremos después del nombre de la clase separado por dos puntos (:).

```
class Machine: CalculateProtocol {
  var result: String = "Empty result."
  var name = "Machine name"

  func calculate() {
    self.result = "The result is 42."
  }

  func anotherFuntion() {
    print("This function just prints a message")
  }
}

var myMachine = Machine()
myMachine.calculate()
print(myMachine.result)
```

Además, podemos utilizar un protocolo de la misma manera que usaríamos una clase.

```
var myCalculator: CalculateProtocol = myMachine
myCalculator.calculate()
print(myCalculator.result)
```

Una clase puede adoptar varios protocolos; de hecho, esta es una parte esencial del patrón que se utiliza para construir y organizar la funcionalidad de los *frameworks* y aplicaciones en iOS, como veremos en los próximos apartados. Por ejemplo, esta es la declaración de una de las clases encargadas de gestionar el comportamiento de una pantalla en una aplicación iOS:

```
class MovementsListViewController: UIViewController, UITextFieldDelegate,
UITableViewDataSource, UITableViewDelegate
```

La clase *MovementsListViewController* es heredada de la clase base *UIViewController* y adopta tres protocolos. Cada uno de ellos agrupa funcionalidades en un ámbito específico:

- *UITextFieldDelegate* se usa para la gestión de campos de texto.
- *UITableViewDataSource* se usa para gestionar los datos de una tabla.
- *UITableViewDelegate* se usa para definir el comportamiento de una tabla.

En los apartados siguientes veremos cómo hay que utilizar estos protocolos.

Recursos adicionales

Para ver todos estos conceptos del lenguaje con más detalle, es recomendable ver *Swift in Sixty Seconds*, una serie de vídeos cortos acompañados de ejemplos de código donde se repasan los conceptos básicos de Swift: https://www.hackingwithswift.com/sixty.

Como recurso mucho más completo tenemos la guía del lenguaje Swift en su página oficial: https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html.

3. Xcode

Introducción a Xcode

Xcode es la herramienta oficial para desarrollar aplicaciones tanto para iOS como para tvOS y macOS. Esta herramienta ha ido evolucionando con el tiempo a la vez que aparecían nuevas versiones de iOS con nuevas posibilidades y nuevos dispositivos para poder desarrollar nuevos tipos de aplicaciones y adaptar las existentes.

Xcode es un entorno integrado de desarrollo (IDE, del inglés *integrated development environment*), es decir, incorpora todas las herramientas necesarias para diseñar, codificar y construir nuestras aplicaciones. Como tal, no necesitamos ninguna otra herramienta adicional.

No obstante, existen otras herramientas para desarrollar aplicaciones para dispositivos iOS, pero solo Xcode es la herramienta oficial de Apple. El resto de las herramientas acaban utilizando de una manera u otra Xcode para poder generar los binarios finales de las aplicaciones. Generalmente, acaban haciendo uso de las herramientas de línea de comandos incluidas en Xcode para firmar y generar los binarios finales.

Con las diferentes versiones de Xcode se han ido incorporando diferentes herramientas. Por ejemplo, hace unos años eran necesarias dos aplicaciones distintas para desarrollar aplicaciones iOS: Xcode y otro programa llamado Interface Builder. La primera servía para desarrollar el código, mientras que con la segunda se creaban las interfaces de usuario. A partir de la versión 4 de Xcode, Interface Builder se integró también dentro de Xcode, lo que simplificó el desarrollo en una única herramienta.

Versiones de Xcode

Las versiones de Xcode van a la par con las versiones de iOS. Cuando aparece una versión nueva de iOS, también se lanza una de Xcode. Concretamente, en la versión 5 se añadieron cambios significativos, parecidos a los cambios que se produjeron en iOS 7 para simplificar la interfaz al máximo. Por ello, las versiones Xcode 4 y Xcode 5 tienen bastantes diferencias. También se facilitó la configuración de servicios como iCloud, Passbook y GameCenter, al tiempo que se permitió crear bots que permitieran programar compilaciones y tener un mayor control para encontrar errores de programación. También se incorporaron mejoras de la herramienta de AutoLayout, que permite indicar cómo se comportan los elementos con diferentes tamaños de pantalla.

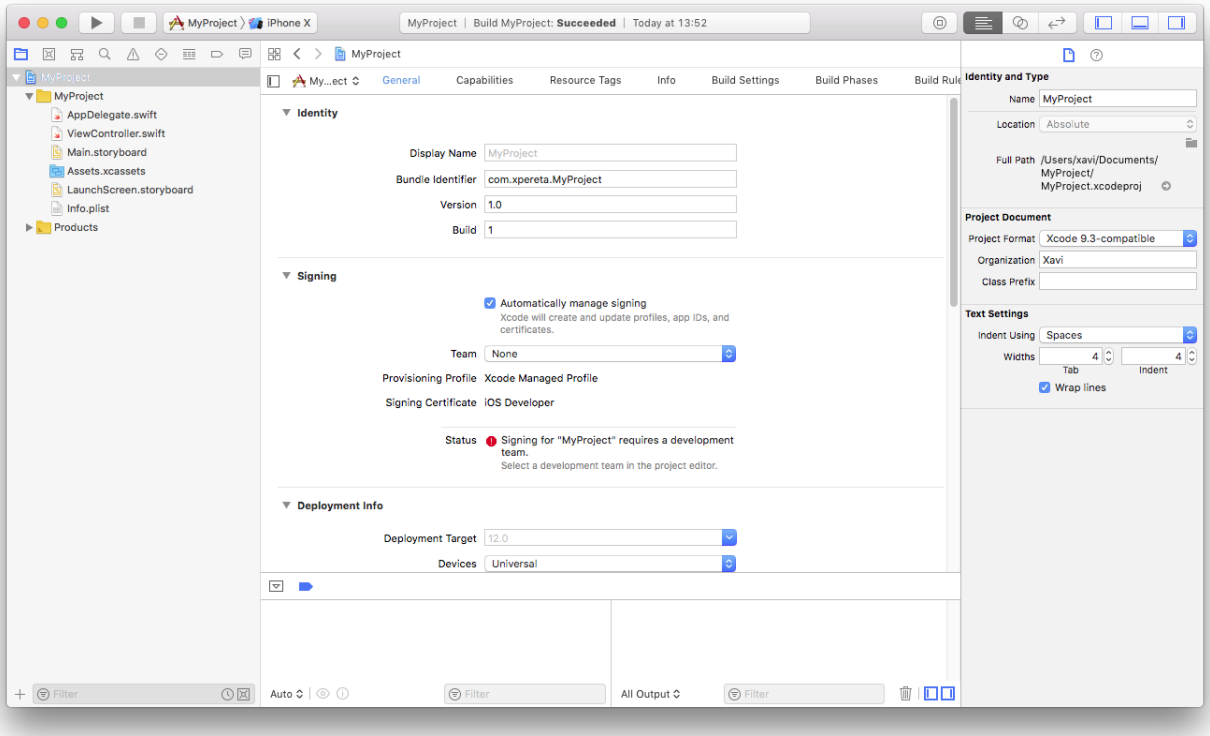
En la conferencia de desarrolladores de 2014, se presentó la versión de Xcode 6. La principal novedad de esta versión es el soporte del lenguaje de programación Swift y los Swift Playgrounds, que nos permiten probar y desarrollar mientras visualizamos los resultados en tiempo real. También incluyó nuevas funcionalidades para desarrollar la interfaz de usuario, como las Size Classes, para gestionar diferentes tamaños de pantalla; Live Rendering, para ver de manera más fidedigna el resultado final mientras construimos la interfaz, y View Debugging, que nos da una representación gráfica en 3D interactiva de todos los elementos que componen una pantalla muy útil para poder entender cómo está construida la pantalla y ayudar a solucionar problemas.

En 2015 se presentó la versión de Xcode 7 con soporte para Swift 2 y Metal para OS X. También incorpora la posibilidad de ejecutar y depurar nuestras aplicaciones en dispositivos sin necesidad de adquirir una licencia de desarrollador de Apple. La versión de Xcode 8 se presentó en 2016 con soporte para Swift 3. Y la versión de Xcode 9 se lanzó en 2017 con destacables mejoras en el editor de código, soporte para la refactorización en Swift y la posibilidad de depurar dispositivos iOS sin necesidad de una conexión permanente por cable.

Este año, en 2018, se ha presentado la Xcode 10. Esta versión tiene disponible una interfaz oscura (*dark mode*) cuando se ejecuta en macOS Mojave, además de un soporte para añadir *dark mode* a las aplicaciones que creemos para macOS (aún no disponible para iOS). También tiene mejoras en la productividad del editor de código, como la edición con múltiples cursores. Incorpora un nuevo mecanismo de *build* que proporciona más rendimiento cuando generamos las aplicaciones.

Componentes

Una vez iniciado Xcode, se nos muestra la siguiente pantalla:

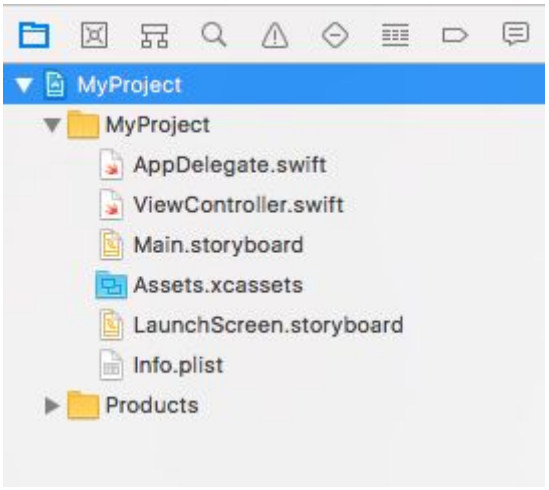


Navegadores

En la parte izquierda, se encuentran diferentes navegadores. Son herramientas que nos dan acceso a las partes de nuestro proyecto. En la parte superior, hay una barra que permite ir alternando entre las distintas herramientas.

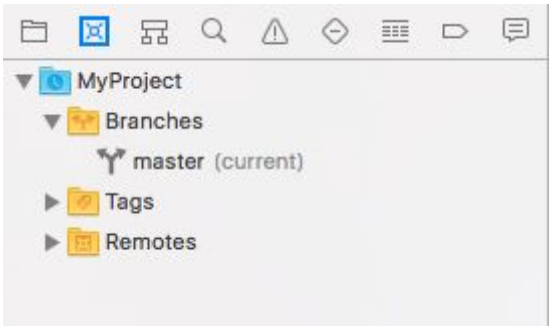
Project navigator

Nos proporciona acceso a los ficheros que forman parte de nuestro proyecto. Desde ahí añadiremos los ficheros de código fuente y también el resto de recursos necesarios, como iconos, imágenes, sonidos o ficheros de configuración. También permite ordenar y agrupar los ficheros.



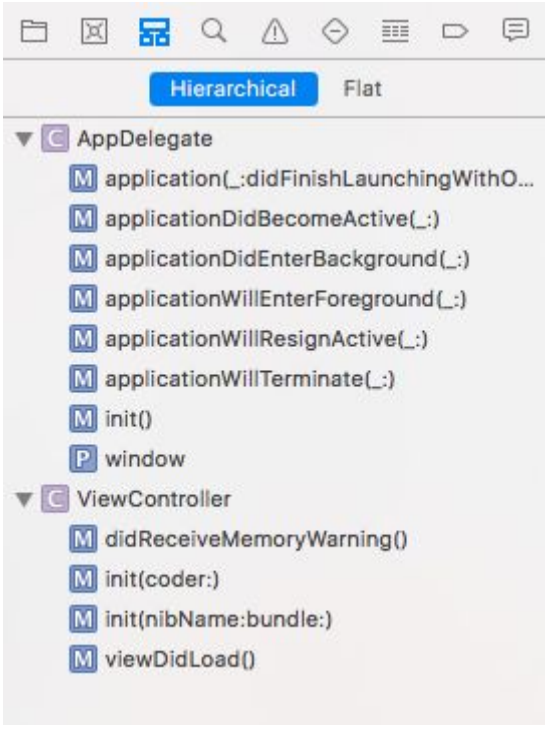
Source control navigator

Si lo tenemos enlazado, por ejemplo, con *git*, nos permite visualizar el estado del control de código fuente de nuestro proyecto.



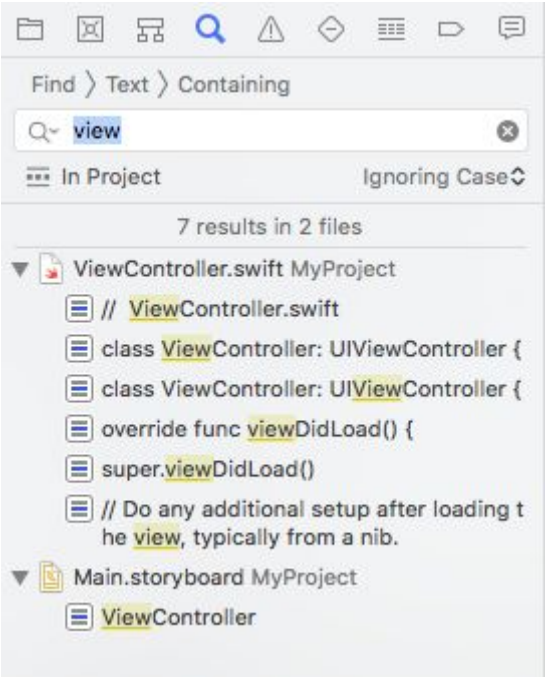
Symbol navigator

Nos permite navegar directamente entre las propiedades y funciones que están definidas en las clases del código fuente de nuestro proyecto. Nos será útil porque el nombre de los ficheros de código no siempre refleja exactamente el nombre de las clases que hayamos definido y, además, nos permite explorar las jerarquías de herencia de clases.



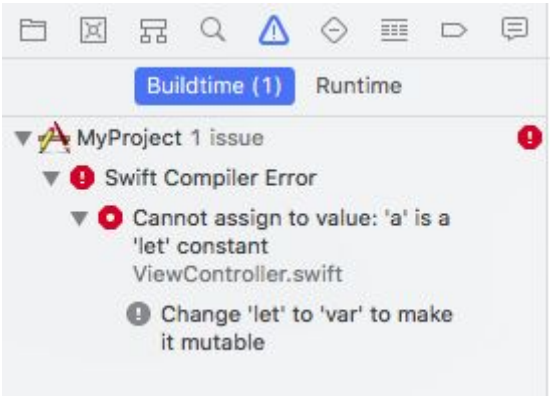
Find navigator

Permite buscar en todos los ficheros que forman parte de un proyecto. Dispone de opciones avanzadas de búsqueda mediante expresiones regulares y también permite acotar el ámbito de búsqueda a un proyecto o directorio en concreto.



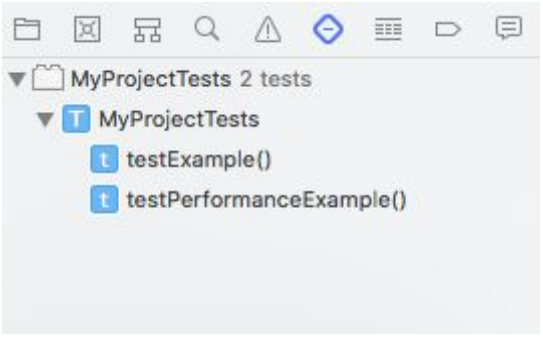
Issue navigator

Nos muestra los avisos y errores que Xcode ha detectado en un proyecto, tanto en la configuración como en el código fuente. Si hay errores en el código, en este *navigator* aparecerá una entrada para cada uno de ellos. Al pulsar sobre cada uno de los errores, nos abrirá el fichero de código fuente en la línea donde se ha detectado el error seleccionado.



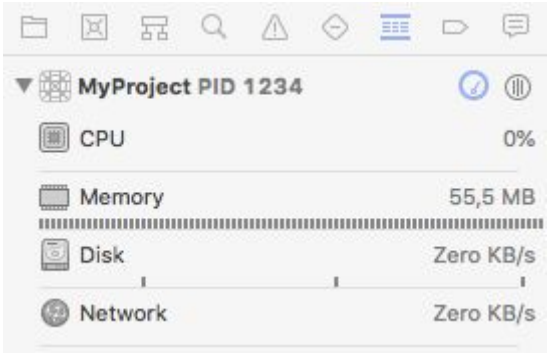
Test navigator

Los test unitarios y de UI que hayamos definido en nuestro proyecto aparecerán aquí. Desde este mismo *navigator* podremos ejecutarlos.



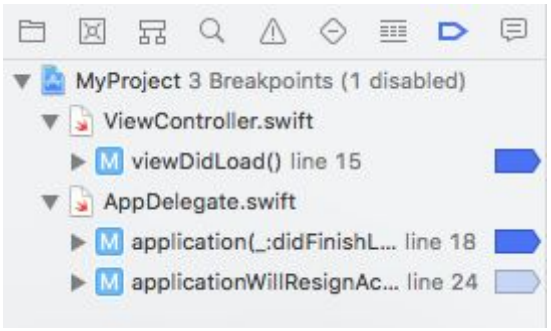
Debug navigator

Al ejecutar la aplicación en modo *debug* desde el simulador o dispositivo, se nos muestra información acerca de los consumo de recursos e información del punto de ejecución en que se encuentra la aplicación en cada momento. Veremos en detalle su funcionamiento más adelante.



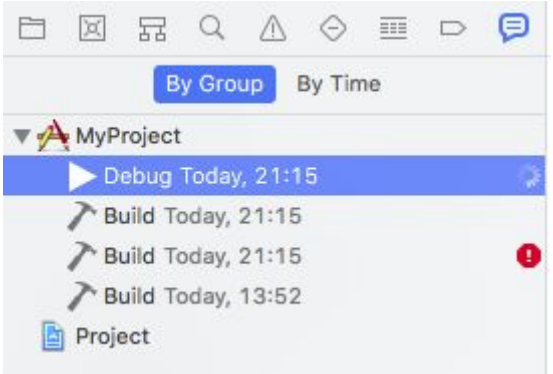
Breakpoint navigator

Muestra el listado de todos los puntos de interrupción que hayamos definido en nuestro proyecto. Como veremos más adelante, es muy sencillo definir puntos de interrupción sobre los ficheros de código fuente a medida que los vayamos necesitando. Pero desde el *navigator* podemos gestionarlos de forma eficiente, activarlos, desactivarlos y eliminarlos cuando ya no los necesitemos.



Report navigator

Da acceso a la información y los *logs* de las operaciones que ha realizado Xcode; por ejemplo, cuando le pedimos que construya una nueva versión ejecutable de nuestra aplicación.

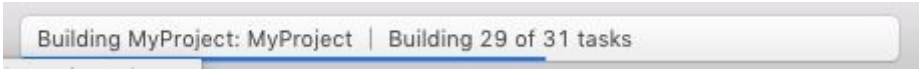


Herramientas de ejecución

Permiten empezar o detener la ejecución de la aplicación tanto en el simulador como en el dispositivo, si lo tenemos conectado.

Para seleccionar el modelo de dispositivo que queremos simular o el dispositivo físico, pulsamos sobre el icono del *selector* (en esta imagen se identifica como «iPhone X»).

Al pulsar el botón de ejecución, se construirá la aplicación, se arrancará el simulador, se instalará la aplicación y se iniciará su ejecución automáticamente.



La barra de estado informa sobre las acciones que está realizando Xcode; por ejemplo, si está construyendo la aplicación o ejecutándola en el dispositivo. También da información sobre si hay algún error y permite acceder directamente al código para solucionarlo.

El botón de biblioteca de componentes proporciona acceso a los elementos que se usan para construir las aplicaciones. Se trata de un botón contextual, es decir, ofrece diferentes tipos de elementos en función del tipo de documento que se tenga abierto en Xcode.

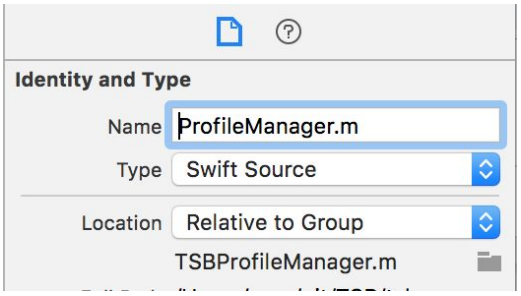
Los botones de la imagen anterior controlan los distintos modos en que podemos editar un documento en Xcode:

- **Editor estándar**: muestra el documento seleccionado en modo normal, ocupando la mayor parte del espacio disponible.
- **Editor asistente**: muestra dos documentos en pantalla. Normalmente intenta mostrar automáticamente el documento relacionado con el que estemos editando. Por ejemplo, si tenemos abierto un *storyboard* con un *view controller* seleccionado, abre el fichero Swift que contiene el código de ese *view controller*.
- **Editor de versiones**: divide el editor en dos espacios; en cada uno de ellos hay una versión distinta del documento para que podamos ver las diferencias entre distintas versiones de un mismo documento.

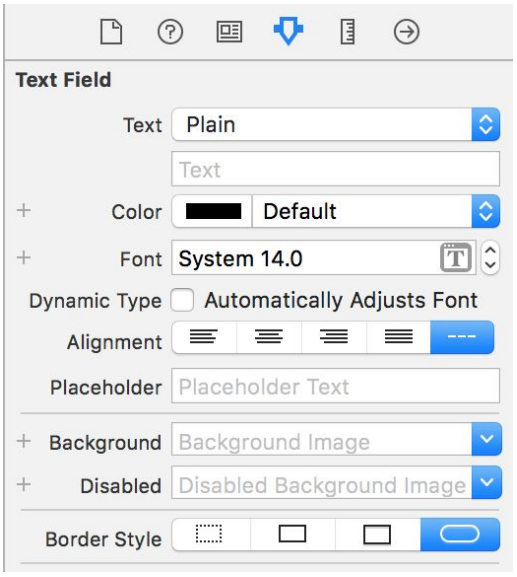
Este otro conjunto de botones nos permite activar y desactivar la visibilidad de los paneles del navegador, el área de *debug* y los *inspectors*. En función de lo que hagamos en cada momento, nos puede convenir ocultarlos para disponer de mayor espacio para editar el documento de código o de interfaz de usuario.

Utilities

Esta área nos permite configurar las propiedades de los componentes de nuestra aplicación. Es un espacio contextual, es decir, muestra diferentes herramientas en función del tipo de fichero o de control de interfaz de usuario que tengamos seleccionado. Por ejemplo, si estamos editando un fichero de código fuente, permite editar su nombre.



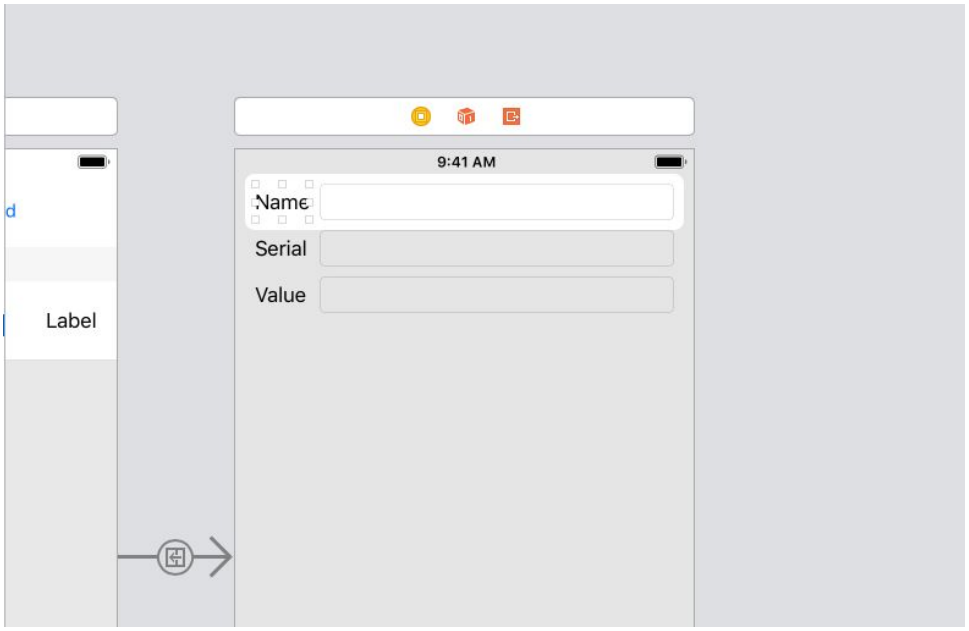
Y si editamos la interfaz de usuario de una pantalla, muestra las herramientas para configurar sus propiedades y editar su aspecto.



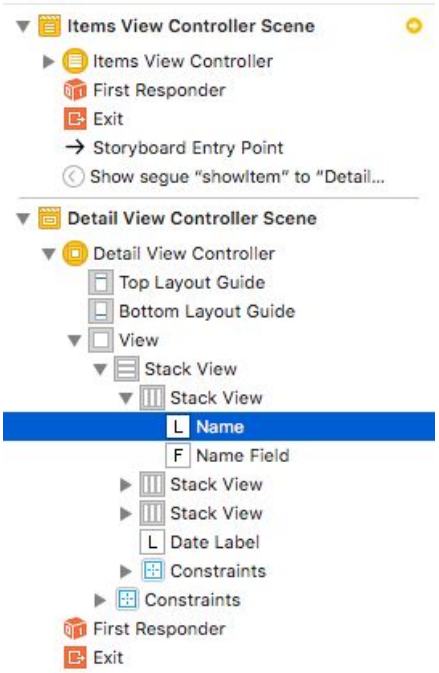
Interface Builder

Interface Builder es la herramienta de Xcode que permite desarrollar la interfaz de usuario de las aplicaciones. Se inicia automáticamente cuando se edita un fichero que representa la interfaz de usuario en una aplicación: *.xib* o *.storyboard*.

Interface Builder contiene dos secciones. *Canvas*, un lienzo donde ubicamos todo los componentes de interfaz de usuario de nuestra pantalla. Si se edita su posición, tamaño y aspecto, muestra la siguiente pantalla:



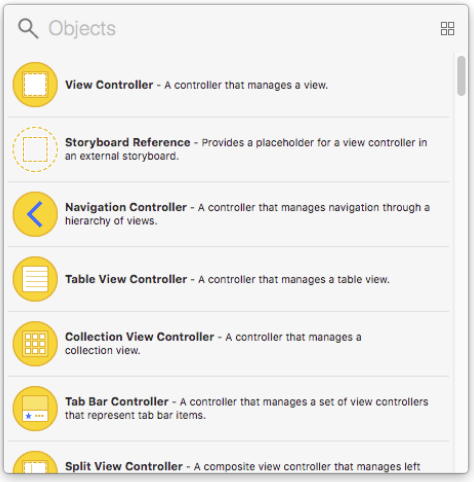
Document outline, una lista organizada de forma jerárquica con todos los componentes de la interfaz de usuario.



Cuando tenemos pocos elementos, podemos trabajar directamente en el lienzo, pero en pantallas complejas es muy útil poder ver de un vistazo todos los componentes de interfaz de usuario y poder seleccionarlos fácilmente independientemente de su tamaño o posición en la pantalla.

En la versión 10 de Xcode se ha modificado la interfaz para añadir elementos en el editor, ahora está accesible al pulsar este botón de la zona superior derecha de la pantalla:

Este botón abre en una ventana la lista que antes estaba situada en la parte inferior del inspector:



Zona de *debug*

Nos muestra información que nos va a ayudar a depurar nuestra aplicación cuando se está ejecutando. A la izquierda nos muestra el panel con la inspección de expresiones, y a la derecha, la consola de depuración. En el panel de inspección de expresiones podemos explorar el contenido de los objetos de nuestra aplicación.

En la consola de *debug* se nos muestran los mensajes que genera la aplicación en tiempo de ejecución. Y también podemos ejecutar comandos para interactuar con nuestra aplicación mientras se está ejecutando, lo que nos resultará muy útil para recopilar información y resolver problemas.

4. Ejemplo de construcción de una aplicación

El patrón MVC

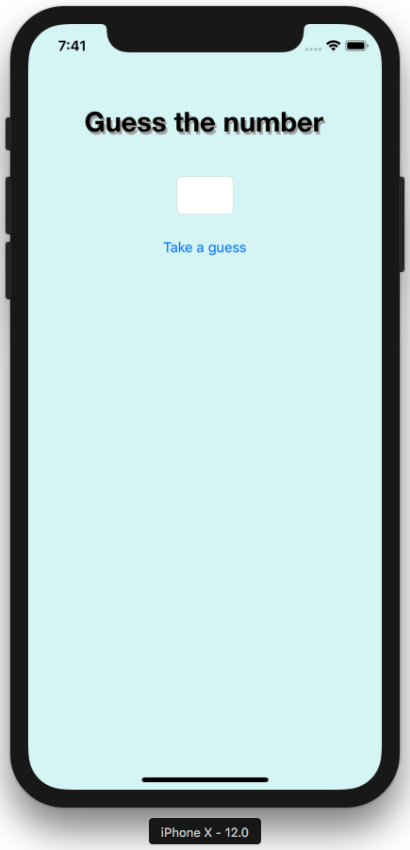
Antes de empezar a diseñar nuestras aplicaciones es importante que tengamos claro el patrón modelo-vista-controlador (MVC), que se usa en el desarrollo iOS.

En MVC, todas las piezas que componen la aplicación pertenecen o a la capa de modelo, o la de vista o a la de controlador.

- La **capa de modelo** representa lo datos pero no tiene relación con la interfaz de usuario, ni, por supuesto, tiene conocimiento de cómo los datos van a mostrarse al usuario.
- La **capa de vista** se compone de los objetos que son visibles para el usuario. Son aquellos que acaban mostrando información al usuario, pero también los que reciben sus interacciones (las imágenes, las cajas de texto, botones, etc.).
- La **capa de controlador** es la que da vida a la aplicación. Decide cuál será el comportamiento en cada momento. Consultando la capa de modelo, decide cómo se configurará la vista, y a partir de los eventos que le llegan desde la vista decide cómo se modificarán los datos del modelo.

Descripción de la aplicación

Para hacernos una idea de cómo es el proceso de construcción de una aplicación usando Xcode, construiremos una aplicación sencilla: un juego en el que el usuario tendrá que adivinar un número del 1 al 100 en el menor número de intentos.



Diseñaremos nuestra aplicación siguiendo el patrón modelo-vista-controlador que hemos descrito más arriba y describiremos cada una de las partes.

El modelo

El modelo gestionará toda la información para jugar a nuestro pequeño juego. Necesitaremos almacenar el número que habrá que adivinar, obviamente. También tendrá una función para iniciar un nuevo juego y otra para intentar adivinar el número.

Para que hacer el juego no sea demasiado difícil y para hacerlo más interesante, nuestro modelo tendrá otra función para dar pistas. Así mismo, para construir nuestro modelo crearemos una clase nueva en Swift.

La vista

La vista se encargará de mostrar la información al usuario; en nuestro caso, la pequeña pantalla que contiene el título, el *textfield* para introducir el número a adivinar, el botón para hacer un intento de adivinar y, por último, la pista si no ha acertado.

También se encargará de recoger lo que haga el usuario: el texto introducido y la acción del botón.

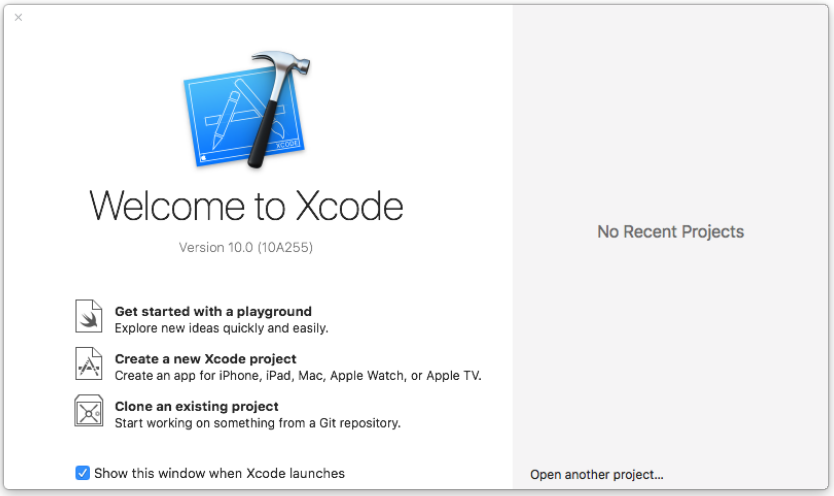
Para construir la vista utilizaremos las vistas y controles que nos proporciona UIKit, que editaremos en un *storyboard* en Xcode.

El controlador

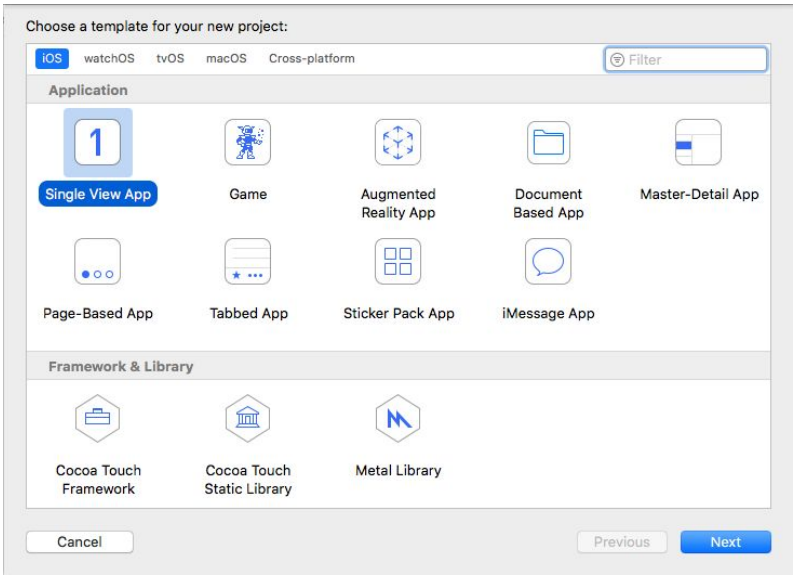
El controlador utilizará la información que le proporciona el modelo para actualizar las vista, y también en función de los eventos que le lleguen de la vista decidirá cómo actualizar el modelo. El controlador lo construiremos con una clase que hereda de *UIViewController*, que es la clase de la que heredan todos los controladores de vistas en UIKit.

Creación del proyecto en Xcode

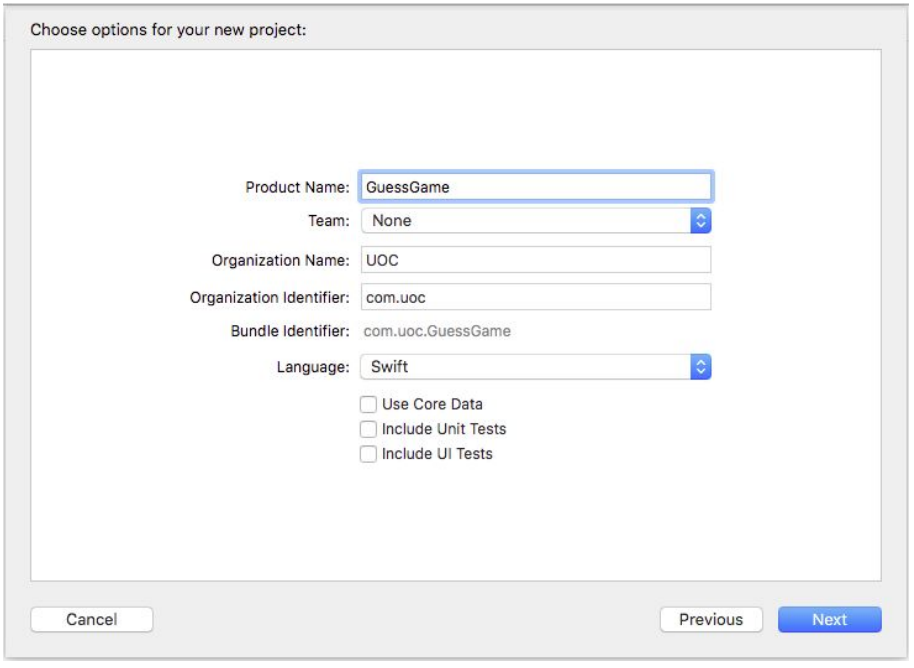
Abrimos Xcode y seleccionamos la opción *Create a new Xcode project*:



En la siguiente pantalla, seleccionamos la plantilla *Single View App*:



En la siguiente pantalla, nos va a pedir el nombre y los datos adicionales para crear la nueva aplicación:



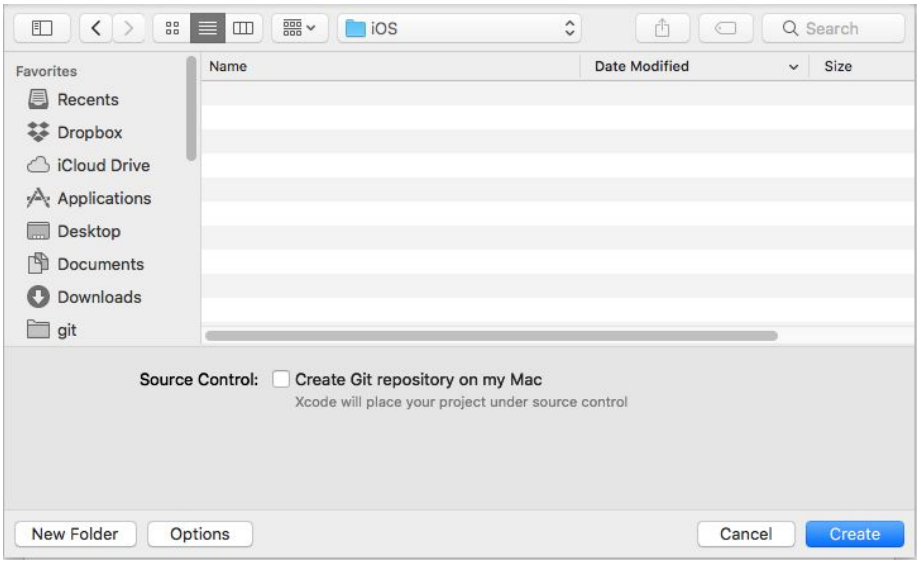
En el campo *Product Name*, introducimos el nombre que queramos, por ejemplo «GuessGame». Este será el nombre también del directorio y el nombre del fichero que contendrá el proyecto Xcode en el sistema de ficheros de nuestro Mac.

En el campo *Organization Name*, introduciremos el nombre de nuestra empresa o nuestro nombre.

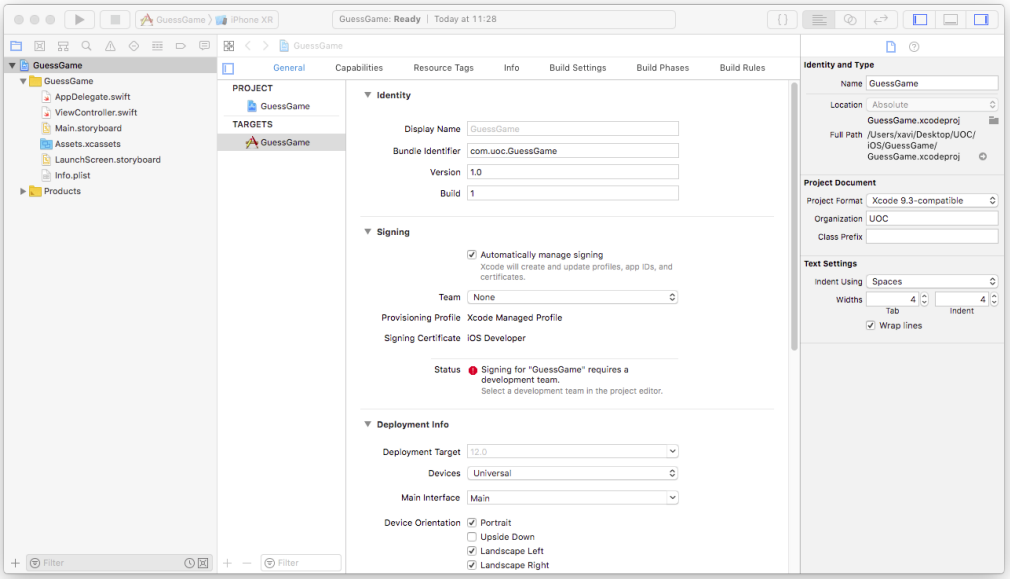
El campo *Organization Identifier* se utiliza para identificar a los desarrolladores en el sistema de Apple. Tiene un formato como un DNS inverso, pero cuidado, ni guarda relación ni está asociado a ningún dominio de internet. Por ejemplo, podemos usar «com.uoc».

El *Bundle Identifier* se genera a partir de los dos campos anteriores e identificará de forma única nuestra aplicación en los sistemas de Apple. A continuación, nos aseguramos que Swift está seleccionado en el campo *Language*.

De momento dejaremos sin marcar las opciones de *Core Data*, *Unit Tests* y *UI Tests*. Al pulsar *Next* nos pedirá que le indiquemos donde vamos a crear el proyecto Xcode en el disco:

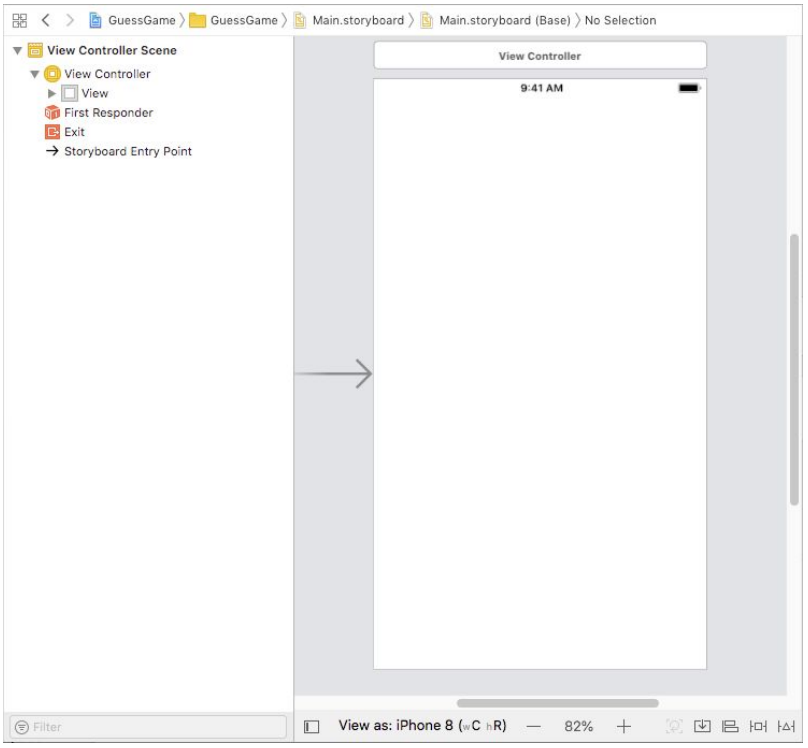


Y ya tenemos el proyecto creado en Xcode:



Construcción de la vista

Primero vamos a crear la vista, que nos ayudará a hacernos una idea de cómo va a funcionar la aplicación. Seleccionamos el fichero Main.storyboard en el *document outline* y el editor de Xcode nos mostrará lo siguiente:

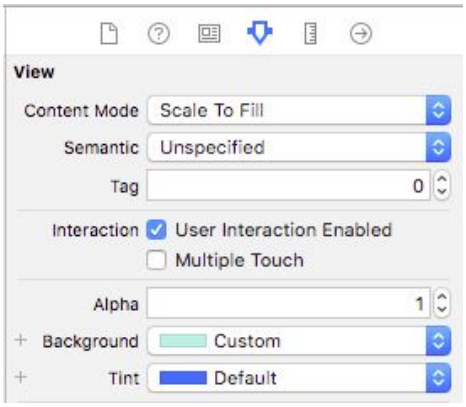


Nuestra aplicación tendrá una única pantalla y la plantilla que hemos seleccionado al crear el proyecto ya nos la proporciona, en blanco. Usando el botón *Library* vamos añadiendo los cuatro elementos que componen la interfaz de usuario.

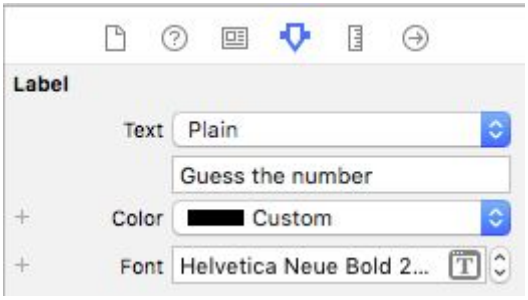
Añadimos un *label*, que será el título de nuestra pantalla. Debajo de este, un *text field* para que el usuario introduzca el número que hay que adivinar. Después, un *button* para iniciar un intento de adivinar el número. Finalmente, otro *label* para mostrar las pistas. Quedará algo parecido a esto:

Seleccionando cada uno de los controladores vamos a personalizar su aspecto para conseguir el diseño que hemos visto en la pantalla al principio del apartado.

Seleccionamos la vista principal y en el *Inspector* modificamos la propiedad *Background*:

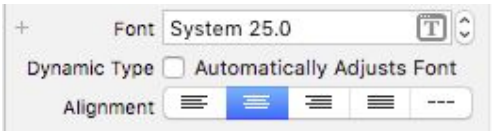


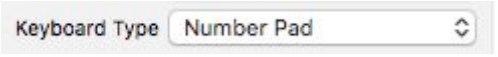
Seleccionamos el primer *label* e introducimos el texto del título «Guess the number». También podemos hacerlo con doble clic sobre el propio *label*. Modificamos también la propiedad *Font* para usar una Helvetica Neue Bold de 29 puntos.



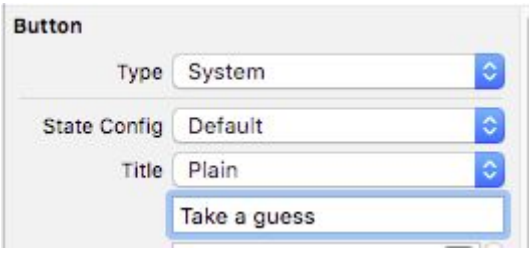
También vamos a darle una sombra dando valores a *Shadow* y *Shadow Offset*:

Seleccionamos el *text field* y le cambiamos la fuente a 25 puntos, la alineación del texto centrada y, finalmente, lo configuramos para mostrar un teclado numérico:





Para el botón, le configuramos el texto como en el caso del *label*, con el valor «Take a guess»:

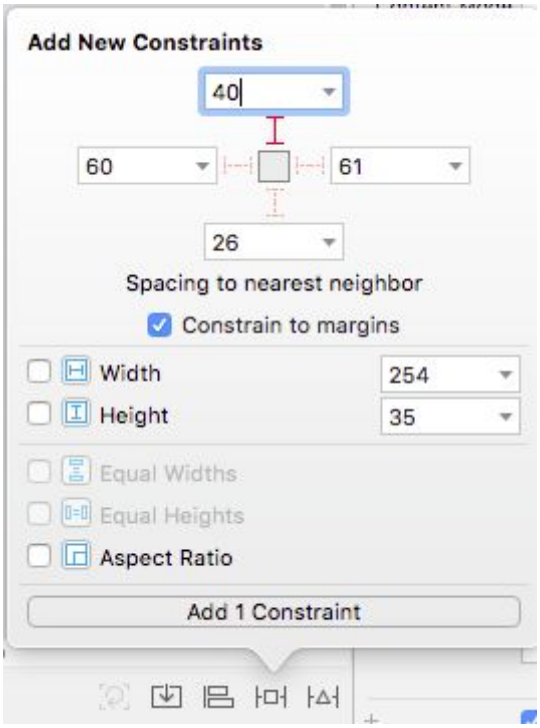


Finalmente, en el último *label* le asignamos también un texto: «Hint label». Y le configuramos un color más discreto:

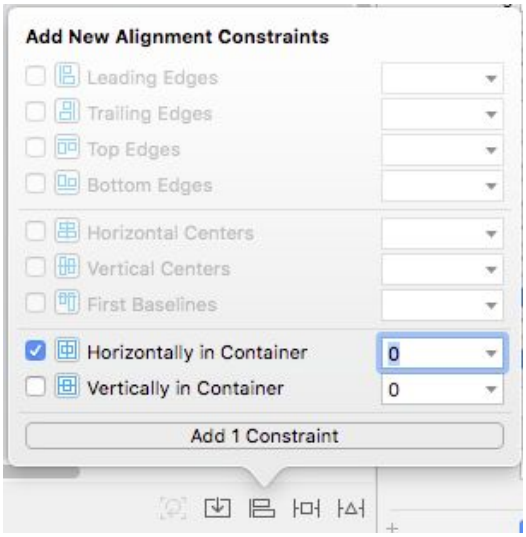
La pantalla nos queda así, ahora:

Vamos a añadir ahora las constraints para que la pantalla se ajuste correctamente a las distintas medidas de pantalla de los dispositivos.

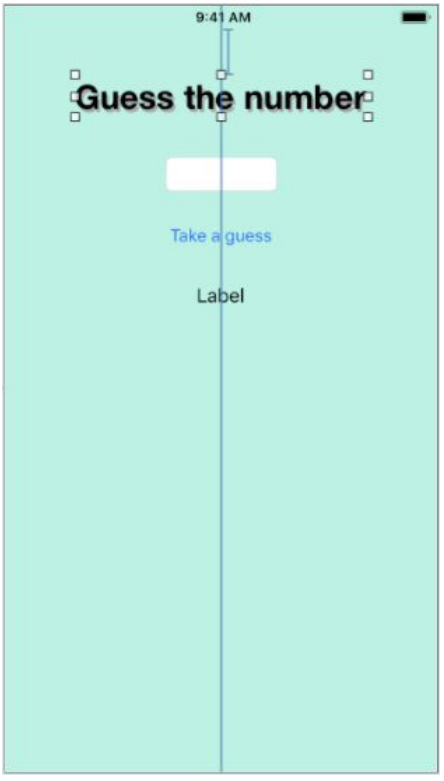
Seleccionamos el *label* del título y pulsamos el botón para añadir nuevas constraints en la parte inferior derecha de Interface Builder. Vamos a indicar que queremos que el *label* esté a 40 puntos de distancia al borde superior. Pulsamos *Add 1 constraint*.



Con el *label* aún seleccionado, pulsamos el botón para añadir constraints de alineación y seleccionamos «Horizontally in Container». Esto hará que el *label* esté siempre centrado en la vista que lo contiene, en este caso en la vista que abarca toda la pantalla. Pulsamos *Add 1 Constraint*.

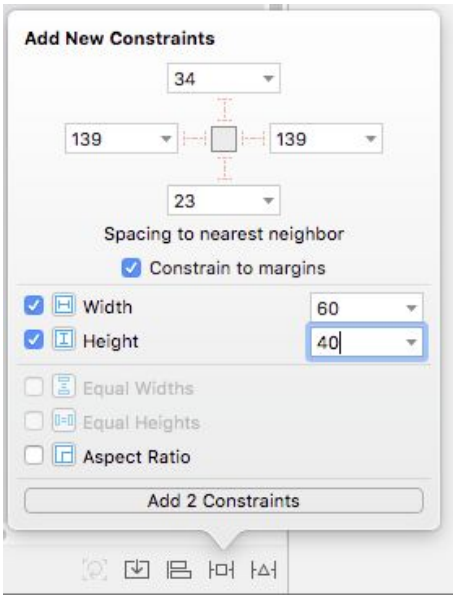


Ya tenemos las constraints necesarias para el título:



A continuación, seleccionamos el *text field* y añadimos una constraint de 40 puntos hasta el elemento superior y otra de centrado en el contenedor, igual que para el título.

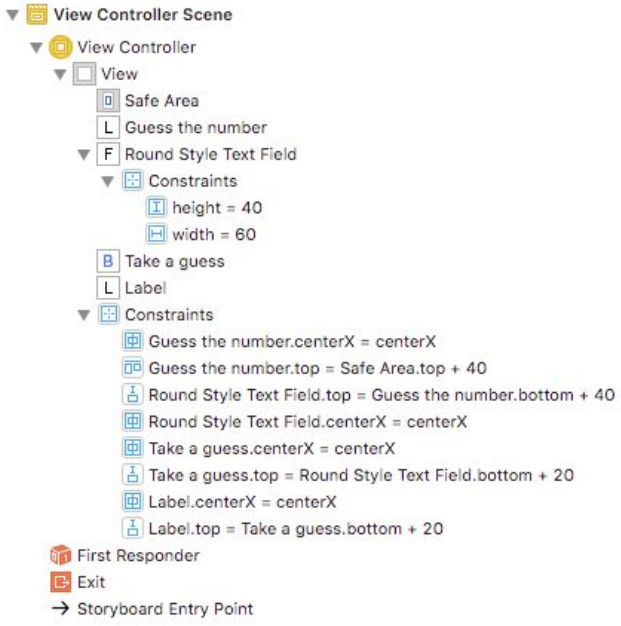
Además, añadiremos constraints para definir su tamaño: especificaremos 60 de ancho y 40 de alto.



Seguidamente, nos centraremos en el *button*. Le añadiremos una *constraint* de 20 hasta el elemento superior y otra de centrado.

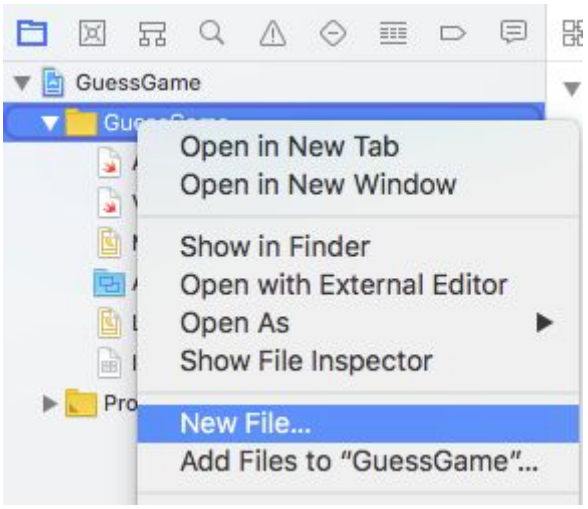
Finalmente, en el *label* para las pistas añadimos una *constraint* de 20 hasta el elemento superior y otra de centrado.

Desplegando la vista del *document outline*, deberían aparecer las siguientes constraints creadas:



Construcción del modelo

El modelo será una clase en Swift. Para crearlo, nos situaremos sobre el *document outline* en el grupo *GuessGame* y con el botón derecho seleccionamos la opción *New File...*.



En la siguiente pantalla seleccionamos la plantilla *Swift File* y *Next*. A continuación, nos pedirá el nombre del fichero que queremos, escribimos «GuessModel» y pulsamos *Create*.

Y este será el código de nuestra clase para el modelo:

```
class GuessModel {
    var numberToGuess = 0
    var attempts = 0
    var lastNumber: Int?

    init() {
        newGame()
    }

    func newGame() {
        numberToGuess = Int.random(in: 1 ... 100)
        attempts = 0
        lastNumber = nil
    }

    func guess(ThisNumber number: Int) -> Bool {
        attempts += 1
        lastNumber = number

        if numberToGuess == number {
            return true
        } else {
```

```
            return false
        }
    }

    func giveHint() -> String {
        guard let lastNumber = lastNumber else { return "You have not made any guesses
yet" }

        if numberToGuess < lastNumber {
            return "The number is lower"
        } else if numberToGuess > lastNumber {
            return "The number is higher"
        } else {
            return "You got it!"
        }
    }
}
```

Tenemos tres propiedades para guardar el número que hay que adivinar, el número de intentos desde que se inició el juego y el último número que se ha intentado adivinar. La función *newGame* configura los valores de todas las propiedades para iniciar un nuevo juego; mientras que la función *guess* la llamaremos para intentar adivinar si el número es verdadero o falso.

La función *giveHint* nos devolverá un *string* con una pista basada en el último número que hemos intentado adivinar. Aquí es interesante ver que usamos un *guard* para tratar el caso especial en que no tenemos un número anterior, porque aún no hemos intentado adivinar el número.

Esta clase tiene implementada una funcionalidad que tiene sentido por ella misma y es completamente independiente tanto por lo que respecta a la vista como al controlador. Podríamos utilizarla tal cual para implementar este mismo juego usando vistas completamente distintas.

Construcción del controlador

El controlador es el que dará vida a nuestro juego. Utilizará el modelo para decir a la vista lo que tiene que mostrar y, con los eventos que reciba de la vista, modificará el modelo.

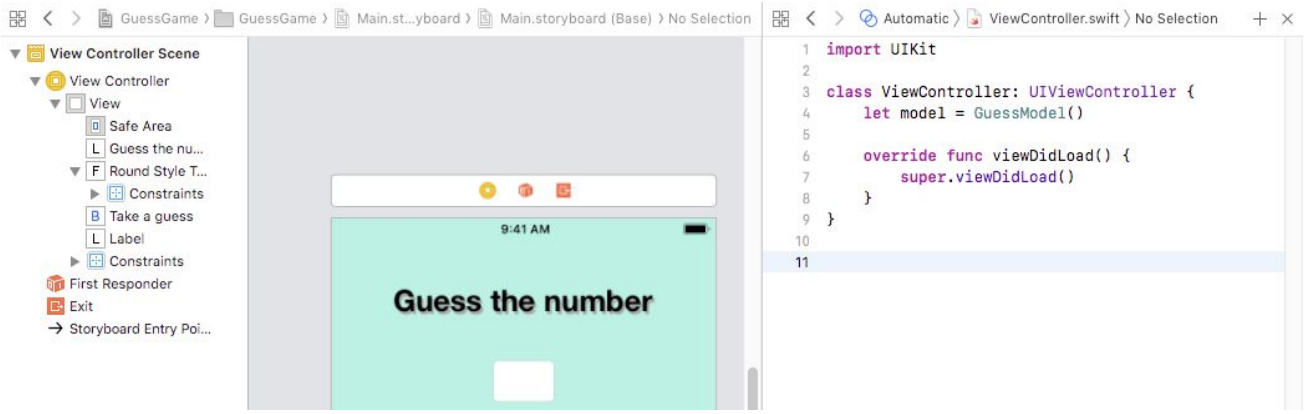
La plantilla que hemos usado para crear el proyecto en Xcode ya nos ha creado un controlador y lo ha asociado a nuestra vista. Lo tenemos en el fichero ViewController.swift. Lo primero que tenemos que hacer es crear una instancia del modelo para tenerlo accesible en nuestro controlador.

```
class ViewController: UIViewController {
```

```
    let model = GuessModel()

    override func viewDidLoad() {
        super.viewDidLoad()
    }
}
```

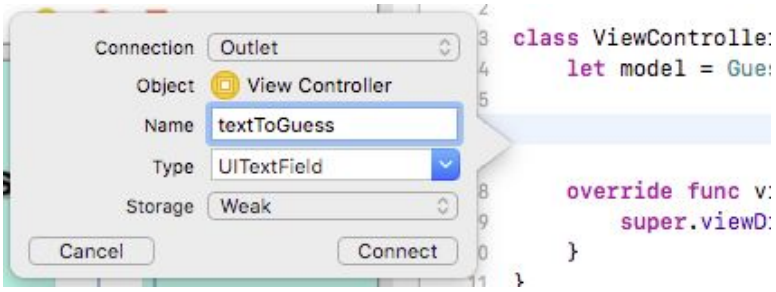
Ahora conectaremos los controles de la vista con el controlador para poder acceder a ellos desde este. Para ello abriremos el fichero Main.storyboard, que contiene nuestra vista, y abriremos el *Assistant Editor*:

Se abrirá el editor de Xcode con una división vertical, a la izquierda, el *storyboard* con nuestra vista, y a la derecha, el código de nuestro controlador:



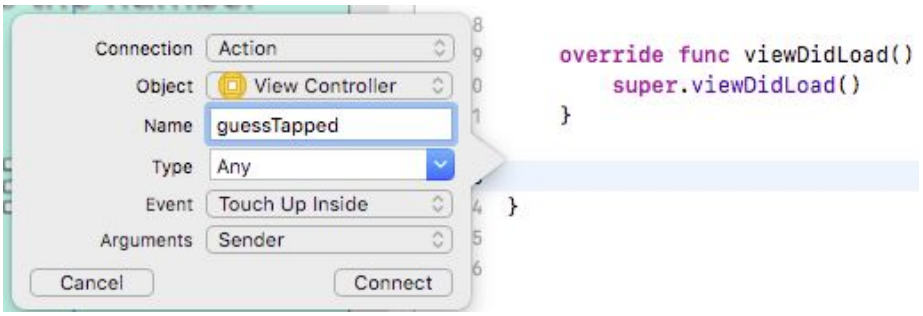
Si no nos aparece el código del controlador, lo podemos localizar usando la barra de navegación que aparece en el editor derecho. Debería aparecer dentro de la opción *Automatic*.

Añadiremos un *outlet* para el *text field*. Para crearlo, tendremos que hacer un movimiento de arrastre desde el *text field* hasta el punto del código donde queramos crearlo, mientras mantenemos pulsada la tecla *control*, al soltarlo nos aparecerá la siguiente pantalla. Nos aseguramos que creamos una conexión *outlet* con el nombre «textToGuess» de tipo *UITextField*.



Hacemos la misma operación con el label para las pistas y le damos el nombre «hintLabel» de tipo *UILabel*.

Ahora necesitamos poder responder al hecho de pulsar el botón. Como hemos hecho antes, arrastramos desde el botón hacia el código manteniendo la tecla *control* pulsada, pero ahora nos aseguramos de que el tipo de conexión es *Action* y le damos el nombre «guessTapped»:



Este es el código que se habrá generado en nuestro controlador:

```
class ViewController: UIViewController {
    let model = GuessModel()

    @IBOutlet weak var textToGuess: UITextField!
    @IBOutlet weak var hintLabel: UILabel!

    override func viewDidLoad() {
        super.viewDidLoad()
    }

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

La función *viewDidLoad* se ejecuta cuando la vista se ha cargado desde el *storyboard*, y es el punto en el que nos prepararemos para iniciar un nuevo juego: configuraremos el modelo y daremos los valores iniciales a los *labels* en la vista.

```
    func startNewGame() {
        model.newGame()

        textToGuess.text = ""
        hintLabel.text = ""
    }
```

La función *guessTapped* está enlazada al *button* en la vista y se ejecutará cada vez que el usuario lo pulse. En esta función nuestro controlador va a realizar la mayoría de su trabajo:

```
    @IBAction func guessTapped(_ sender: Any) {
        // 1
        guard let guessNumber = Int(textToGuess.text ?? "")
            else {
                hintLabel.text = "This was not a number"
                return
        }

        // 2
        if model.guess(ThisNumber: guessNumber) {
            // 3
            let alert = UIAlertController(title: "You did it!", message: "You guessed
the number in \(model.attempts) attempts!", preferredStyle: .alert)
            let playAgainAction = UIAlertAction.init(title: "Play again", style:
.default, handler: { (action) in
                self.startNewGame()
            })
            alert.addAction(playAgainAction)
            self.present(alert, animated: true, completion: nil)
        } else {
            // 4
            hintLabel.text = model.giveHint()
        }
    }
```

1) Obtenemos el número que ha escrito el usuario en el *text field*. Puede darse el caso de que el usuario introduzca un valor que no puede convertirse en *Int*, por ello tratamos esta circunstancia mediante un *guard*. Debemos tener en cuenta que, aunque hayamos configurado un teclado

numérico, el usuario puede introducir texto por otros medios; por ejemplo, pegando texto del portapapeles.

2) Con el valor ya convertido a *Int*, llamamos a la función *guess* del modelo para ver si ha acertado.

3) Si ha acertado, mostraremos un diálogo al usuario con una felicitación. Para ello, construimos un *string* usando información que extraemos del modelo. Cuando el usuario cierre este diálogo, reiniciaremos el juego llamando a la misma función *startNewGame*, que como hemos visto antes preparara el modelo para un nuevo juego y también reinicia los valores de las vistas.

4) Finalmente, si no ha acertado, pediremos al modelo que nos proporcione una pista y la mostraremos en el *label* correspondiente en la vista.

Y ya podemos ejecutar nuestra aplicación pulsando el botón *play*, o con el atajo cmd+R.

5. *View controllers*

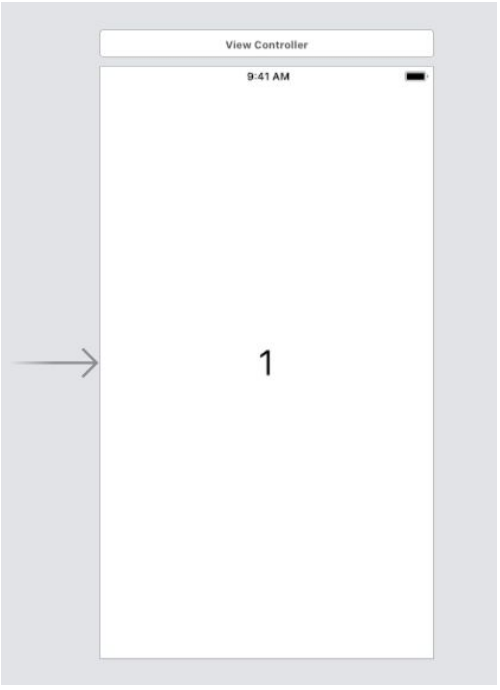
Un *view controller* gestiona una jerarquía de vistas que representan una de las pantallas en las que se organiza una aplicación. La clase de UIKit que nos proporciona la funcionalidad es *UIViewController*. Todas las clases de los controladores en nuestras aplicaciones se heredan de esta.

Los objetivos principales de un *view controller* son:

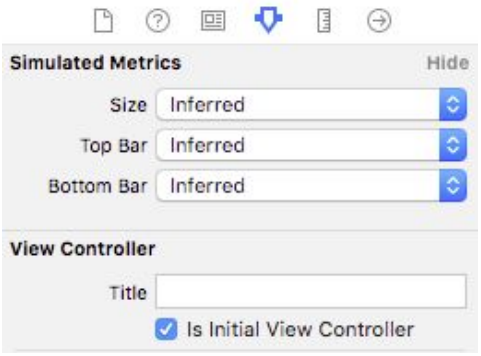
- **Actualizar las vistas para reflejar el estado de los datos**. Por ejemplo, actualizar las imágenes y textos que se muestran en la pantalla cuando se recibe información actualizada.
- **Responder a las interacciones del usuario**. Por ejemplo, navegar a la siguiente pantalla cuando el usuario pulsa sobre un botón.

Configurar el *view controller* de inicio

Cuando se inicia la aplicación se arranca con el *view controller* de inicio. En el *storyboard* este *view controller* se distingue del resto porque aparece con una flecha apuntando hacia él. Podemos arrastrar esta flecha hacia otro *view controller*, que pasará a ser el nuevo *view controller* de inicio.



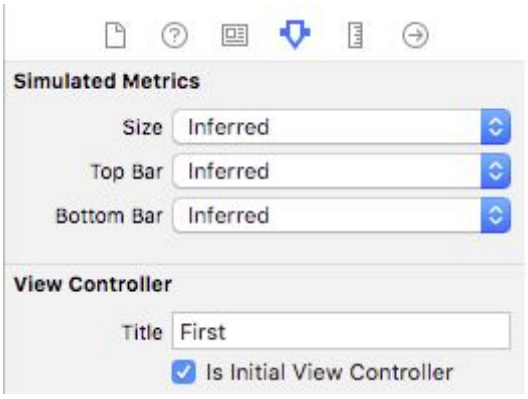
También, con el *view controller* seleccionado, podemos cambiar el *view controller* de inicio marcando la casilla correspondiente en el *Attribute inspector*:



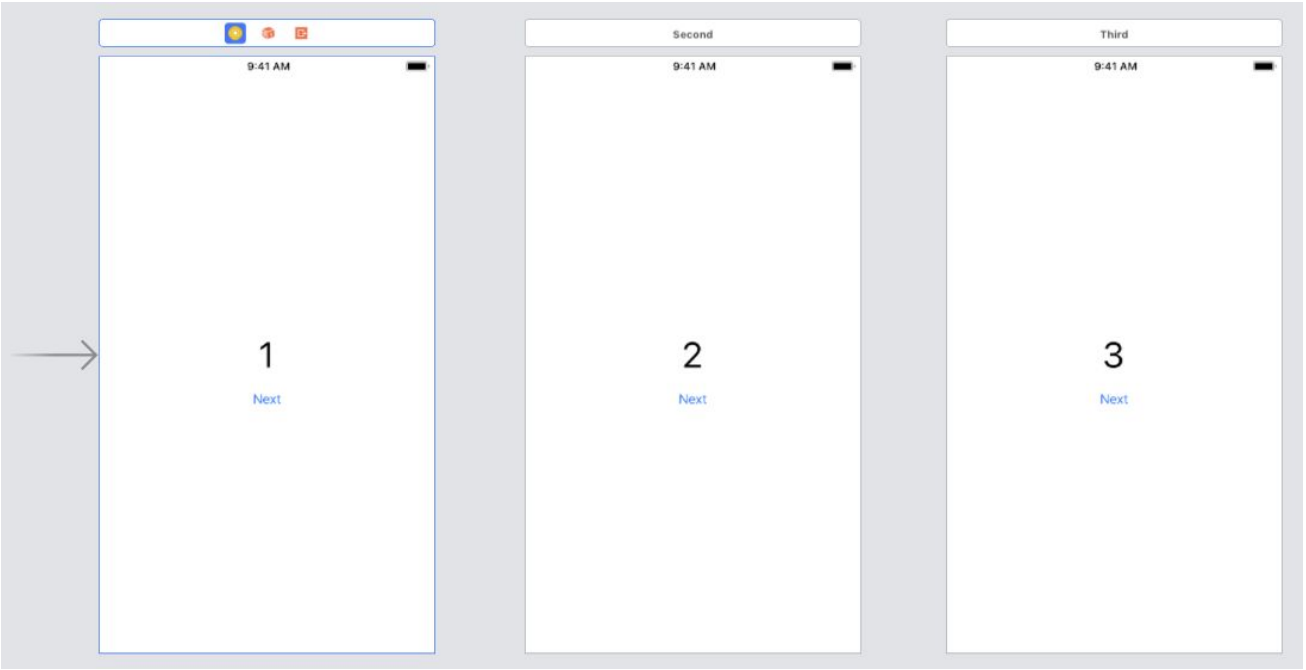
Navegación

Para navegar entre *view controllers*, crearemos *segues* desde un *view controller* hasta el que queremos que se muestre a continuación.

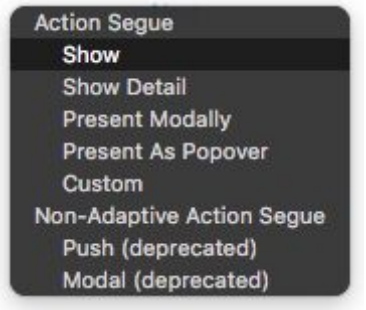
En un proyecto abrimos el *storyboard* principal y creamos tres *view controllers*, cada uno con un botón centrado en la vista. A cada uno le daremos un título distinto, «First», «Second» y «Third». Para cambiar el título seleccionamos cada *view controller* y lo editamos desde el *Attribute inspector*:



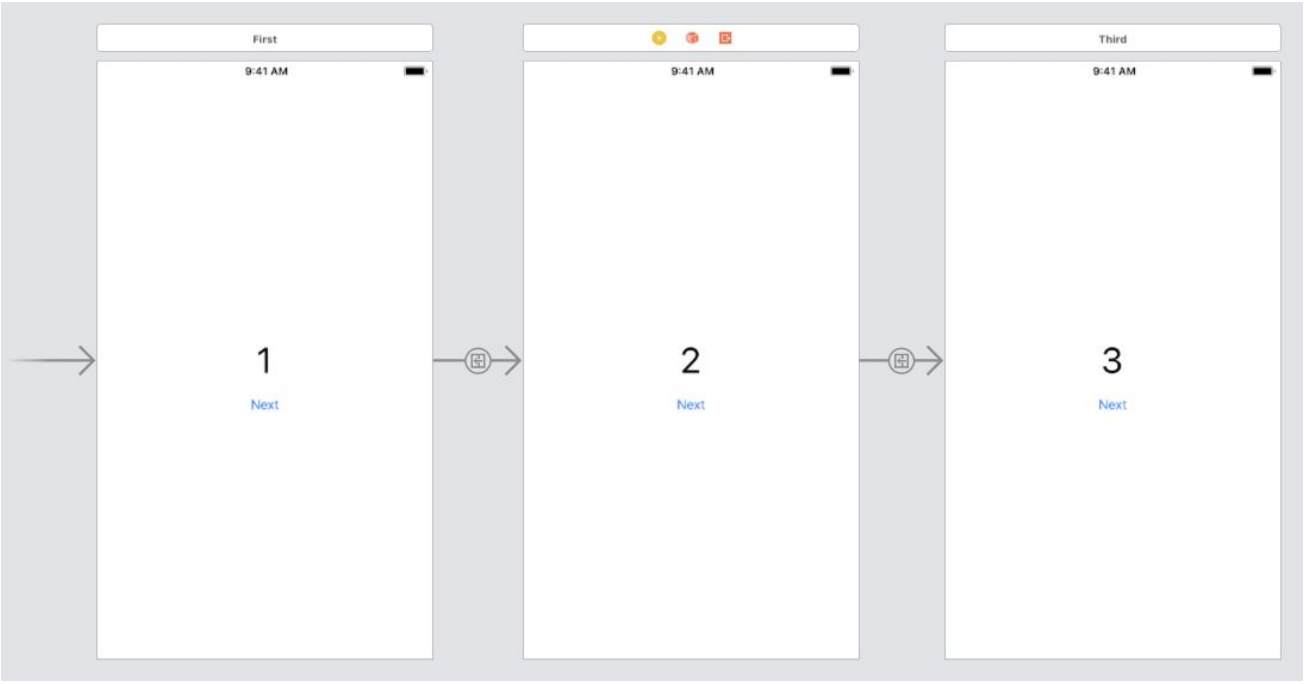
Así quedará el *storyboard*:



Crearemos un *segue* de manera parecida a como enlazamos los *outlets*. Con la tecla *control* pulsada arrastramos con el botón izquierdo desde el botón hasta el *view controller* al que queremos navegar. Al soltar el botón izquierdo sobre el *view controller* de destino, nos aparecerá el siguiente diálogo:

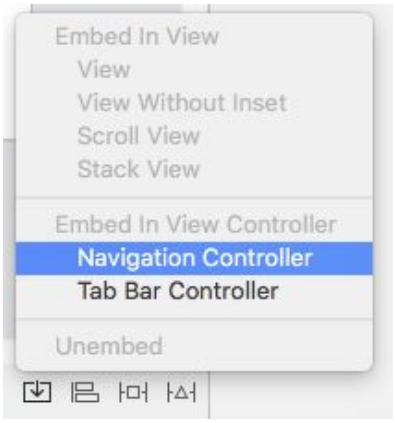


Contiene los distintos tipos de *segue* disponibles; en este caso seleccionamos el primero. Enlazamos el botón del primer *view controller* para navegar hacia el segundo, y el botón del segundo para navegar hacia el tercero:

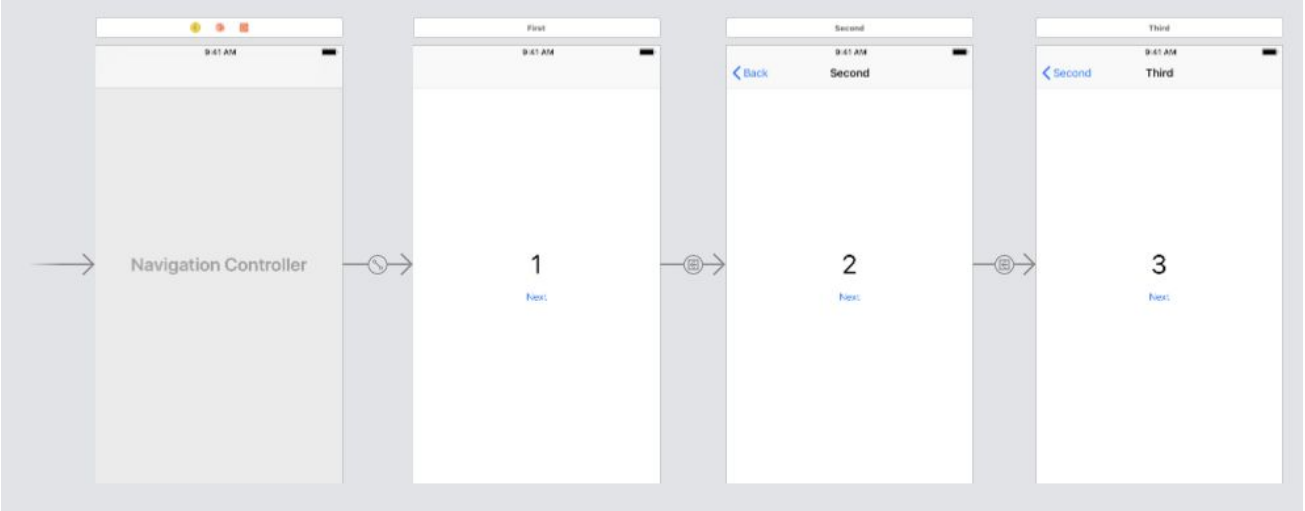


Si ejecutamos ahora la aplicación, veremos que, efectivamente, funciona la navegación hacia el siguiente *view controller*, pero no hay manera de volver atrás. Para conseguir una navegación completa, tenemos que añadir un *Navigation Controller* que muestra automáticamente una zona de navegación en la parte superior que muestra el título del *view controller* que se está mostrando en cada momento y un botón para navegar al anterior.

Para añadir el *navigation controller*, seleccionamos el primero de los *view controllers* y, con el botón de inclusión, en la parte inferior derecha del editor, seleccionamos la opción *Navigation Controller*.



Ahora sí que si ejecutamos la aplicación podremos navegar hacia adelante y hacia atrás sin problemas.


Navigation Controller
First
1
Next
Second
Back
Second
2
Next
Third
Second
Third
3
Next

6. Auto Layout

¿Por qué Auto Layout?

Existen varios tamaños de pantalla para los dispositivos iOS. Esto quiere decir que, para que nuestra aplicación se muestre correctamente, tenemos que determinar la posición y tamaño de las vistas que usemos para que se muestre correctamente en todos ellos.

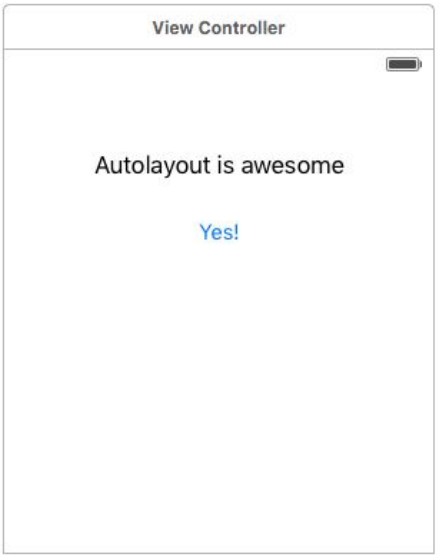
Imaginemos una vista sencilla, por ejemplo, una imagen que queremos que ocupe toda la pantalla, con unos márgenes de 20 píxeles por cada lado. La pantalla de un iPhone 5 SE es de 1136 × 640 píxeles, y, en cambio, la pantalla de un iPhone 8 Plus es de 1920 × 1080, pero queremos que nuestra aplicación se muestre correctamente en ambos dispositivos. Para el iPhone 5 SE podríamos definir un ancho de la imagen de 600 píxeles (640 píxeles – 20 × 2 de los márgenes laterales). Pero nos encontraremos con que para el caso del iPhone 8 Plus este ancho de 600 píxeles no va a ocupar todo el ancho de pantalla: debería ser de 1040 píxeles para que se muestre como deseamos.

Para solucionarlo, podríamos detectar el tipo de dispositivo en cada pantalla de nuestra aplicación y, a partir de este, calcular las posiciones y dimensiones apropiadas para todas nuestras vistas. O mejor, podríamos detectar el tamaño de la pantalla y realizar, a partir de ahí, nuestros cálculos.

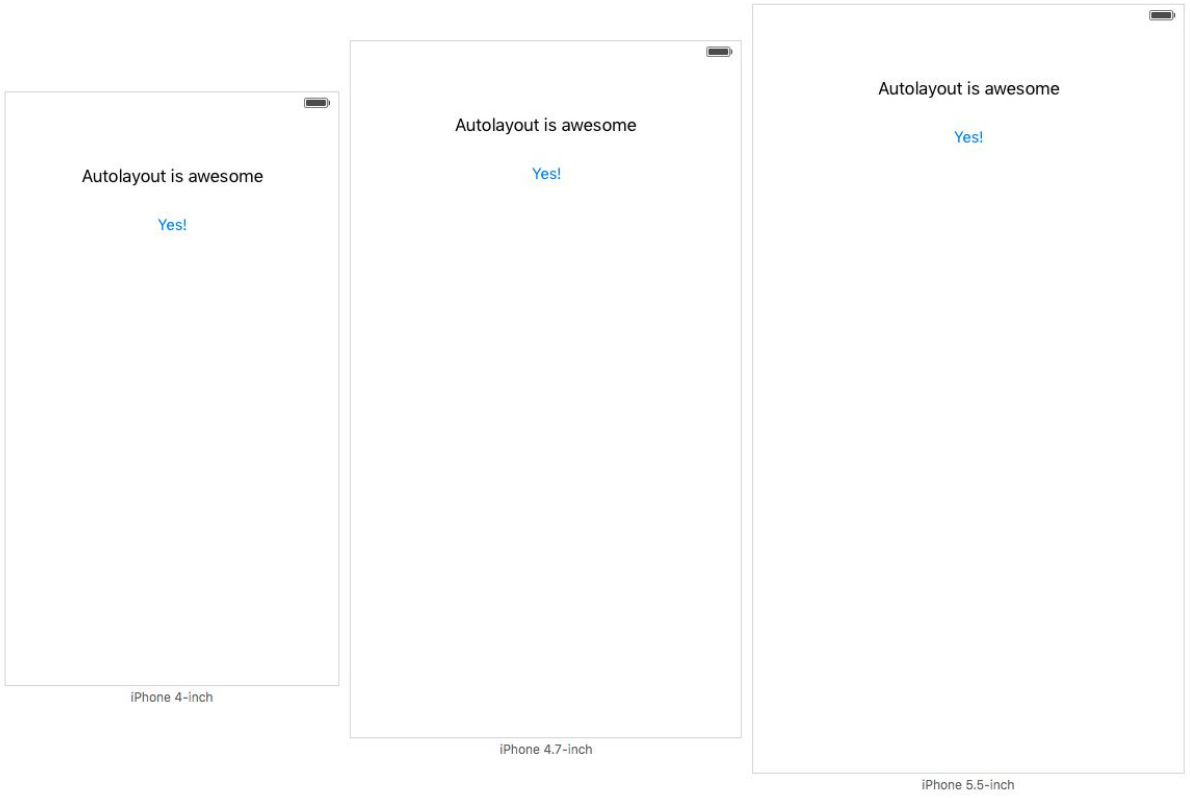
Aunque el tamaño de los dispositivos iOS está estandarizado, lo cierto es que cada vez tenemos que dar soporte a más tamaños de pantalla distintos. ¿Y qué ocurre si tenemos nuestra aplicación ya desarrollada y aparece un nuevo dispositivo con un tamaño de aplicación distinto? Tendremos que revisar el código que tengamos para darle soporte. Necesitamos un mecanismo para definir cómo se van a ubicar nuestras vistas en pantalla y que se adapte a cualquier tamaño de dispositivo. Y justamente esto es lo que nos proporciona Auto Layout.

Con Auto Layout definimos una serie de constraints para nuestras vistas y su posición y tamaño se va a determinar automáticamente.

Así, por ejemplo, podemos definir que queremos un *label* centrado horizontalmente con un botón situado debajo con un margen vertical de 20 puntos hasta el *label*.



Vemos cómo se aplican las constraints para obtener la maquetación que deseamos para cada tamaño de pantalla:

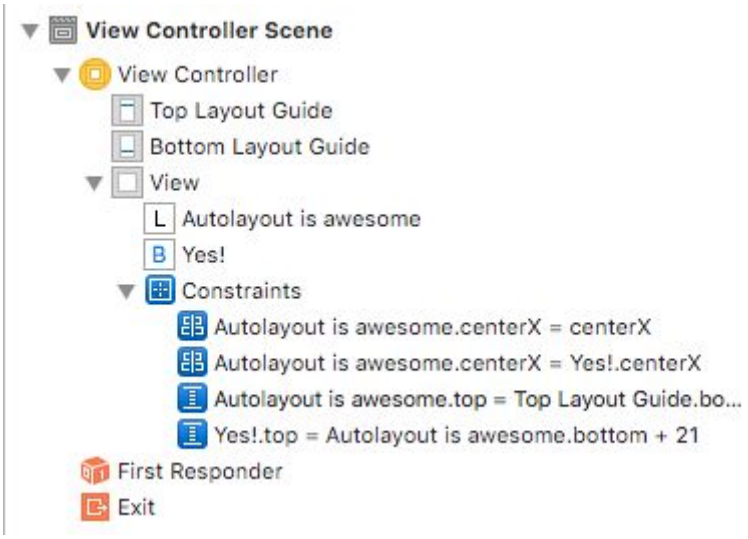


Definición de constraints en Interface Builder

Desde Inferface Builder tenemos tres mecanismos para definir nuestras constraints:

- Usando el mecanismo de *control-drag*.
- Usando las herramientas de definición de constraints del editor.
- Permitiendo que Interface Builder nos sugiera unas constraints.

Independientemente del mecanismo utilizado, las constraints se añadirán a la escena y las podremos editar posteriormente. De esta manera, podemos utilizar la técnica que más nos convenga en cada momento, que el resultado será el mismo.



Constraints por *control-drag*

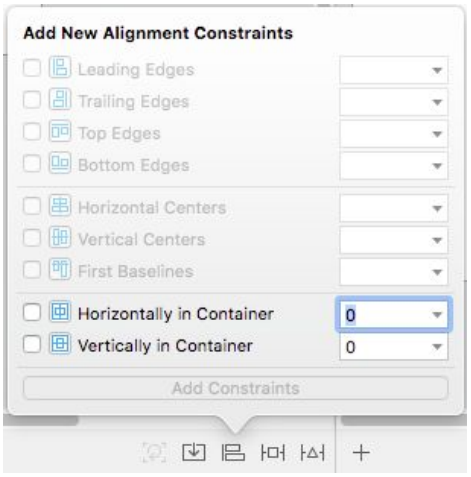
Como podemos deducir por su nombre, consiste en definir constraints entre dos vistas arrastrando de una hacia la otra mientras mantenemos pulsada la tecla *control*. Al soltar, se nos muestra un menú contextual con las diferentes constraints posibles.

Constraints usando las herramientas del editor

Las herramientas de constraints del editor nos permiten definirlas de forma más precisa.

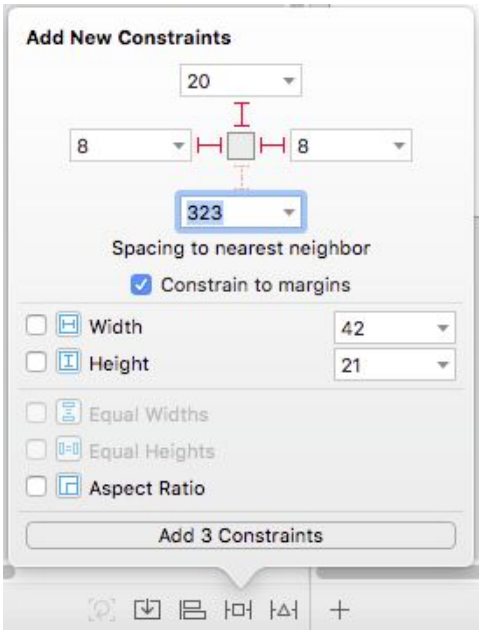
Align tool

Nos permite añadir las constraints necesarias para alinear las vistas de manera muy sencilla. Primero, seleccionamos en el editor los elementos que queremos editar y, luego, pulsamos la herramienta de alineación. Nos aparecerá el diálogo siguiente para añadir los diferentes tipos de alineación:



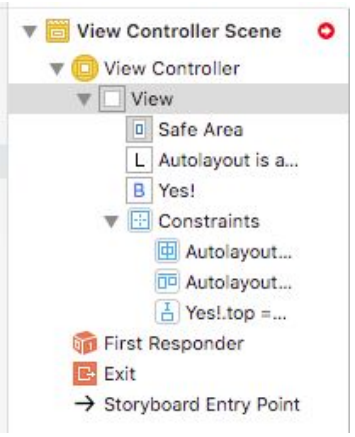
Pin tool

Esta herramienta nos permite añadir rápidamente constraints desde la vista que tenemos seleccionada al resto de vistas que la rodean.

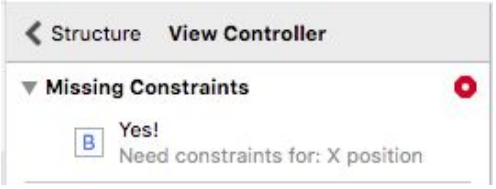


Resolución de problemas con Auto Layout

Hay un número mínimo de constraints para que Auto Layout pueda encontrar la solución a la posición y tamaño de la vista. Si le faltan constraints, nos lo indicará en el *document outline* del editor con una marca en rojo:

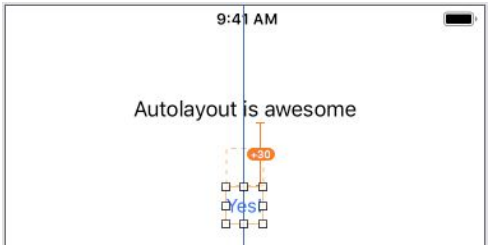


Pulsando sobre esta marca nos informará de todos los errores que ha encontrado en las constraints que tenemos definidas:

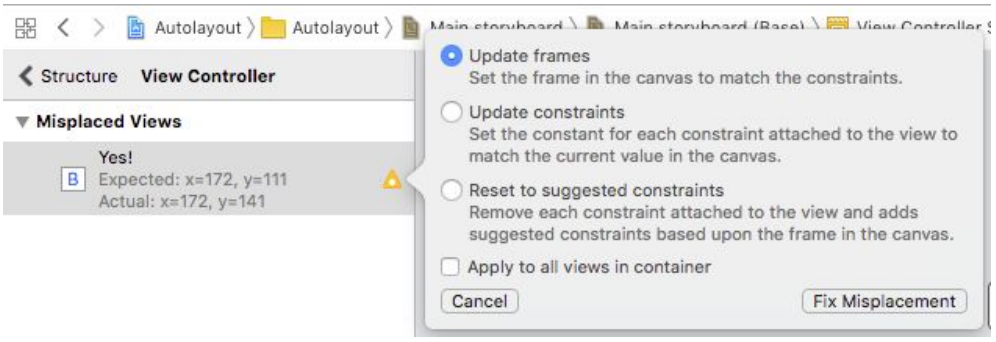


En este caso, lo que ocurre es que para el *button* hemos definido una constraint para su posición vertical pero no sabe cómo determinar la posición horizontal. Añadiendo una constraint para centrarlo en su contenedor, se soluciona el problema.

Si tenemos ya las constraints definidas pero luego movemos las vistas el editor, nos avisará de que hay una incongruencia entre lo que está mostrando y lo que estamos pidiendo con las constraints. Así pues, mostrará en el editor las constraints que no encajan. Por ejemplo:



En el *document outline* nos informará también, y nos sugerirá realizar cambios de forma automática para resolver el problema. Podemos escoger la que mas con convenga y pulsar *Fix Misplacement.*



Update frames devolverá las vistas afectadas a su posición anterior para mantener las constraints que tuvieran definidas. Esto es útil si hemos movido la vista por error o estamos haciendo pruebas.

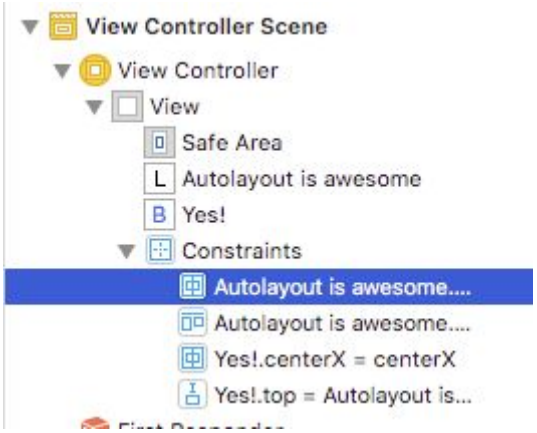
Update contraints actualizará las constraints que tenemos definidas para que encajen con la nueva posición de la vista, pero no creará constraints nuevas. Esta opción es la que se suele usar cuando ya tenemos las vistas y las constraints definidas y luego hacemos pequeños ajustes.

Reset to suggested constraints no es una opción recomendable ya que va a eliminar todas las constraints que hemos definido y va a crear otras nuevas de forma automática. Solamente en los casos más sencillos produce buenos resultados.

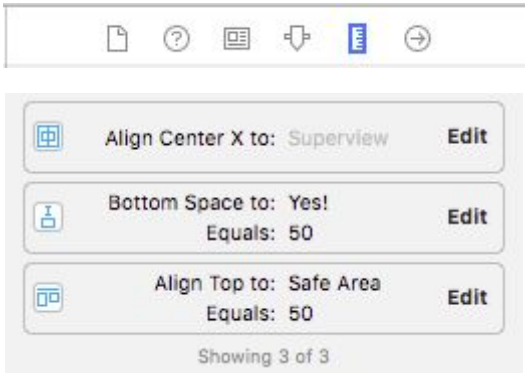
Edición de constraints

Una vez creadas las *constraints* van a aparecer en el propio editor. Aparecen en forma de líneas azules al seleccionar cada vista:

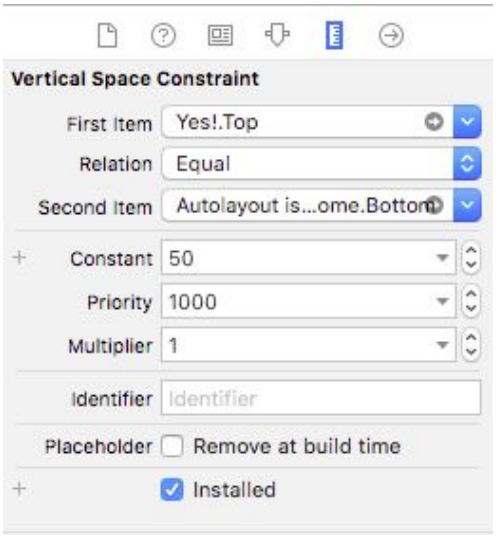
Cuando tenemos muchas *constraints*, se hace difícil seguirlas desde el editor, entonces podemos seleccionarlas desde el *document outline*:



Y también cuando seleccionamos una vista, en el *size inspector* nos aparecerán las *constraints* definidas para ella:

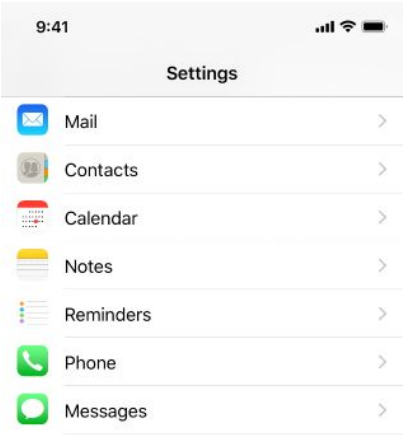


Y ya con la *constraint* seleccionada en el *inspector* podemos configurarla con más detalle, cambiando las vistas entre las que se define o, por ejemplo, la constante, que determina la distancia a la que deberán mantenerse las vistas.



7. *UITableView* y *UITableViewController*

UITableView es el componente de UIKit que nos permite presentar una lista de elementos. Es el componente más utilizado y es fácil verlo en acción en muchas aplicaciones. De hecho, muchas aplicaciones están construidas alrededor de un *UITableView*.



Podemos usar un *UITableView* directamente pero *UITableViewController* ya nos proporciona un view controller que lleva incorporado un *UITableView* enlazado y listo para ser utilizado.

UITableViewController ya adopta los protocolos necesarios para interactuar con el *UITableView*. Entre ellos tendemos el *UITableViewDataSource*, que nos permite controlar la información que se mostrará en el *UITableView*, y *UITableViewDelegate*, con el que gestionaremos la interacción del usuario en el *UITableView*.

Modelo de datos

Para ver el cómo usar *UITableViewController* construiremos una aplicación de ejemplo que nos mostrará un listado de países. En nuestro *view controller* tendremos la siguiente variable que actuará como modelo de datos:

```
class CountriesViewController: UITableViewController {
  var cities = [String]()
}
```

Vemos ahora que heredamos de *UITableViewController*, y no de *UIViewController*, como en los ejemplos anteriores. Inicializamos el modelo de datos en *viewDidLoad*:

```
    override func viewDidLoad() {
        super.viewDidLoad()

        cities = ["Berlin", "Madrid", "Paris", "Rome", "Zagreb"]
    }
```

Protocolo *UITableViewDataSource*

Mediante este protocolo proporcionaremos información al *table view* acerca de los datos que queremos que muestre. Como mínimo le diremos el número de filas que tendrá, y también debemos indicarle el contenido de cada una de estas filas.

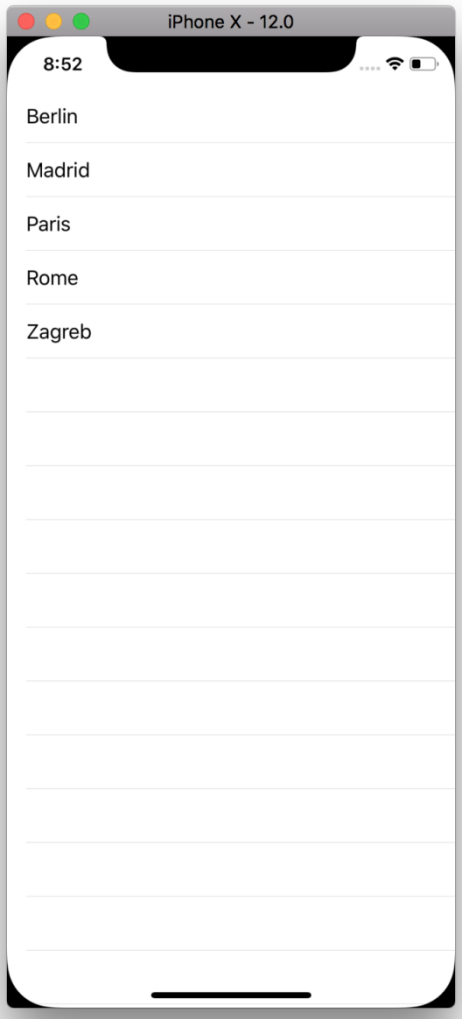
Con *numberOfRowsInSection* le decimos el número de filas que tendrá la tabla y será, claro, el número de elementos de nuestro *array* del modelo de datos:

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

Y en la función *cellForRowAt* tenemos que devolver la celda que queremos mostrar para cada una de las filas de la tabla. En nuestro caso vamos a hacer que la celda contenga el nombre de cada una de las ciudades que tenemos en nuestro *array* del modelo de datos.

```
override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath)
-> UITableViewCell {
    let cell = tableView.dequeueReusableCell(withIdentifier: "CityCell", for:
indexPath)
    cell.textLabel?.text = cities[indexPath.row]

    return cell
}
```

Si ejecutamos la aplicación, ya nos aparece nuestra lista de ciudades:



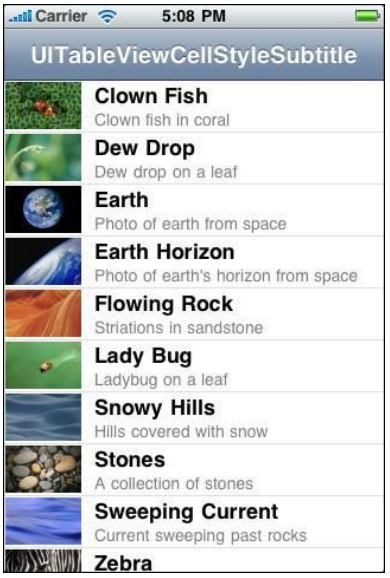
UITableViewCell

Esta clase representa cada una de las celdas de nuestra tabla. Mediante sus propiedades podemos personalizar el contenido y el aspecto de las celdas.

textLabel

detailTextLabel

imageView



Reciclaje de celdas

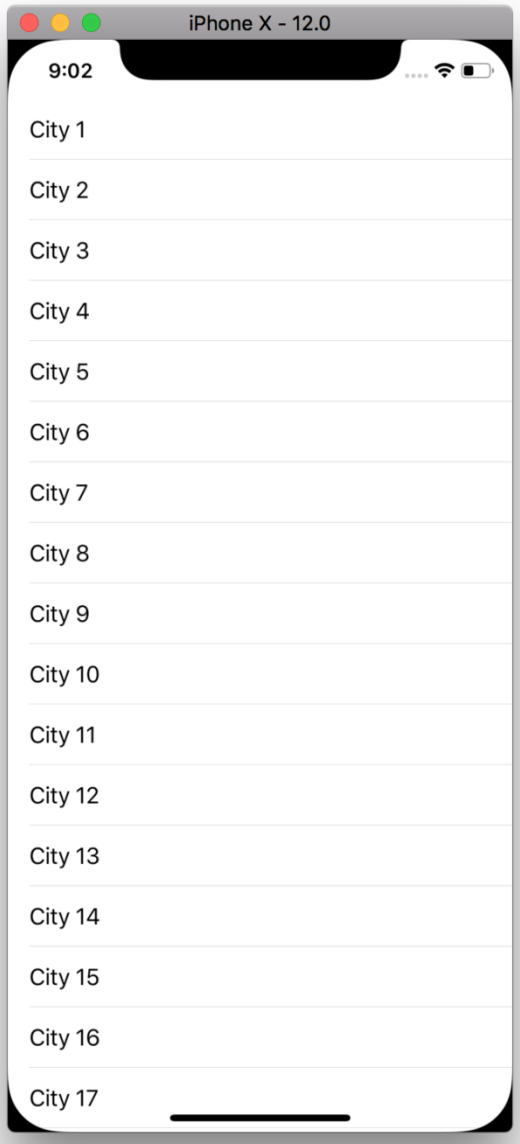
Aquí es crucial obtener la celda siempre con la función de *dequeueReusableCell*. El secreto para que el *tableview* funcione con un gran rendimiento y eficiencia de recursos en iOS es que las celdas se reciclen. Aunque tengamos miles de datos para construir la pantalla, solamente se utiliza un número reducido de celdas, que además no se tienen que crear constantemente, sino que se reutilizan. Cuando una celda deja de ser visible en el *tableview*, se recicla para ser utilizada para mostrar el dato siguiente. Todo esto se gestiona internamente en el *tableview* sin tener que hacer nosotros nada adicional.

Y es esta función *dequeueReusableCell* la que se encarga de devolvernos una celda recién creada si es la primera vez que la pedimos o una celda reciclada. La gestión del estado de las celdas del *tableview* se realiza internamente.

Vamos a hacer un cambio en el modelo de datos para ver más en detalle el reciclaje de las celdas:

```
override func viewDidLoad() {
    super.viewDidLoad()
    for num in 0...1000 {
        cities.append("City \(num)")
    }
}
```

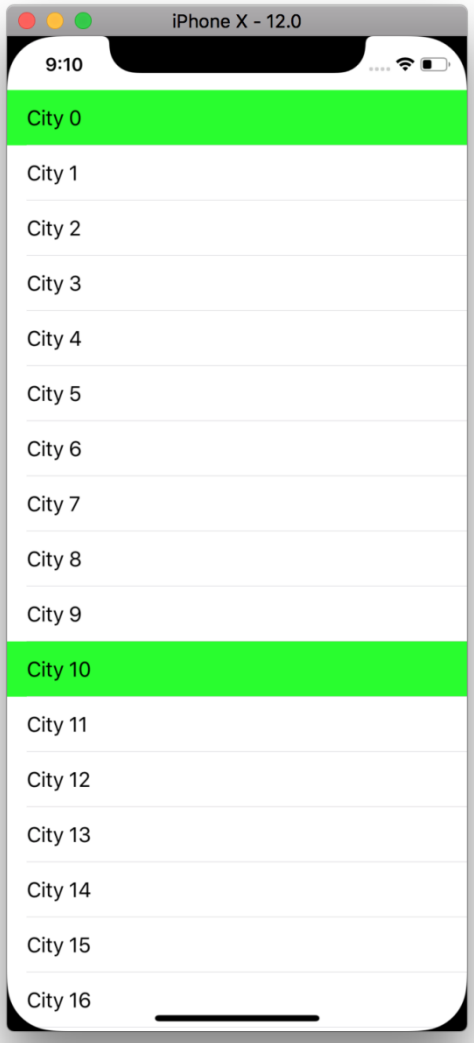
Ahora nuestro *view controller* gestiona mil celdas. Si nos movemos hacia abajo, veremos que el movimiento es muy fluido.



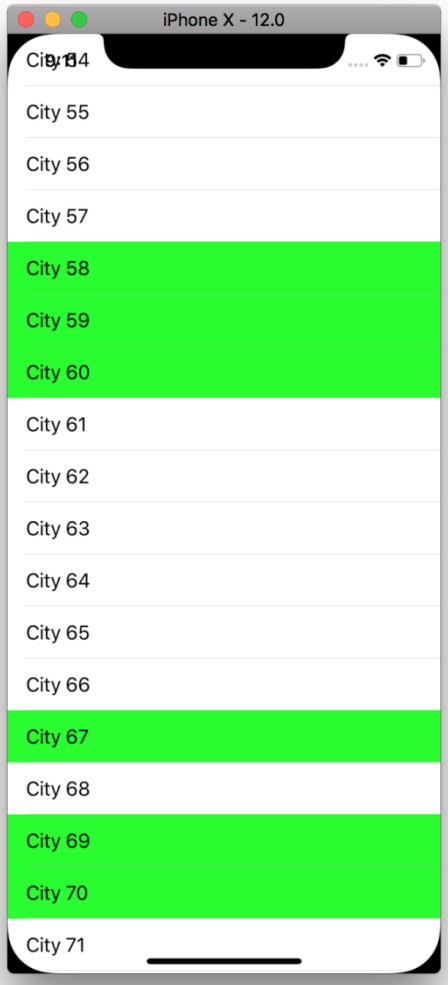
Imaginemos que queremos marcar una de cada diez celdas que mostramos, por ejemplo, cambiando el color de fondo. Añadimos este código para cambiarle el color de fondo de la celda antes de devolverla en *cellForRowAt*:

```
if indexPath.row % 10 == 0 {
    cell.backgroundColor = UIColor.green
}
```

Ejecutamos la aplicación otra vez:



Y parece que funciona correctamente, cada diez celdas nos marca una en verde. Pero a medida que vamos bajando, nos vamos a dar cuenta de que aparecen con color verde de fondo otras celdas que no nos esperamos. ¿Qué está pasando aquí?



Las celdas que tienen el color de fondo que no nos esperamos, como la 58 o la 59, es porque en un momento anterior fueron celdas que cumplieron la condición y les hemos cambiado el color de fondo. Cuando las celdas se reciclan, no se les restablecen sus propiedades. Somos nosotros, con la función *cellForRowAt*, quienes tenemos la responsabilidad de asegurarnos de que cada celda tiene las propiedades correctas para representar el dato de la celda que nos piden para cada *indexPath.row*.

Así pues, este sería el código correcto en este caso:

```
if indexPath.row % 10 == 0 {
    cell.backgroundColor = UIColor.green
} else {
    cell.backgroundColor = UIColor.clear
}
```

Tenemos que asegurarnos de que siempre le damos un valor a las propiedades de la celda, porque no sabemos qué posición había ocupado anteriormente la celda que nos devuelve *dequeueReusableCell* y, por lo tanto, qué valores tendrá en sus propiedades.

Rendimiento y errores comunes

Debemos tener muy claro que, aunque implementamos las funciones de *UITableViewDataSource*, no las vamos a llamar nunca directamente. UIKit se encarga de mostrar en pantalla el *UITableView* y va a llamar a nuestras funciones siempre que lo considere necesario.

Además, no tenemos ningún control sobre el número de veces que se va a llamar *numberOfRowsInSection* y cellForRowAt, o el orden en el que se llamarán. Y el orden y número de llamadas puede cambiar entre versiones de iOS, por lo que, no debemos asumir nada.

Las funciones de *UITableViewDataSource* se llaman como parte del proceso de pintado de la pantalla, que es uno de los que tienen un rendimiento más crítico. Por lo tanto, nuestro código debe ser lo más rápido posible. Y, por supuesto, dentro de ellas debemos evitar llamar funciones que recuperen datos directamente del sistema de ficheros o, peor aún, que hagan peticiones a un servidor remoto. Por lo general, usaremos solamente las variables locales que actúan como modelo de datos de nuestro *view controller*.

Desde estas funciones tampoco debemos consultar el estado de otros controles. Y, por supuesto, estas funciones no deben tener efectos secundarios como modificar variables del *view controller* o cambiar el estado de otros controles.

8. Animaciones

Para mejorar la experiencia de usuario en nuestras aplicaciones podemos utilizar las animaciones que nos proporciona el propio *framework* UIKit. Podremos utilizar animaciones para comunicarnos mejor con el usuario, expresar mejor las acciones que realiza el usuario en nuestra aplicación y también para hacer nuestra aplicación más atractiva. Aun así, debemos ir con cuidado para no abusar de animaciones gratuitas si no aportan valor al usuario.

Básicamente las animaciones consisten en realizar cambios a lo largo de un periodo de tiempo en las propiedades de las vistas que conforman la interfaz de usuario de nuestra aplicación. Se trata de las mismas propiedades que ya hemos visto y utilizado en los apartados anteriores.

Para seguir los ejemplos de este apartado, partiremos del proyecto Xcode Animations, disponible en el repositorio de código de la asignatura. Al ejecutar el proyecto podemos ver que ya tenemos una serie de pantallas navegables con la ayuda de un *navigation controller*.

Animaciones básicas

En la primera pantalla de la aplicación podemos ver que tenemos dos botones y un *label*. Vamos a conseguir que el *label* aparezca y desaparezca al pulsar cada uno de los botones.

Si abrimos BasicAnimationViewController.swift veremos que ya tenemos las acciones de los botones enlazadas a las funciones *fadeOut* y *fadeIn*.

En la función *fadeOut* añadiremos el siguiente código:

```
@IBAction func fadeOut(_ sender: UIButton) {
    let animations = { () -> Void in
        self.label.alpha = 0
    }

    UIView.animate(withDuration: 2, animations: animations)
}
```

Ejecutamos la aplicación y vemos que al pulsar el botón *Fade out* desaparece el *label*.

La función *Animate* de UIView es la que se encarga de ejecutar las animaciones. El parámetro *withDuration* determina la duración en segundos. Y las animaciones las definimos mediante la *closure* que pasamos en el parámetro *animations*.

Tal como vimos en el apartado dedicado a Swift, la sintaxis para definir una *closure* es parecida a la de una función, y en este caso necesitamos una *closure* sin ningún parámetro () y que tampoco devuelva ningún valor *Void*.

En el cuerpo de la *closure* asignaremos los valores que queremos que tengan las propiedades de nuestra vista al fin de la animación. En este caso hacemos que la opacidad del *label* sea 0, con lo que lo hacemos desaparecer.

En la acción del botón *fade in* añadimos el código para que la opacidad del *label* vuelva a ser 1.

```
@IBAction func fadeIn(_ sender: UIButton) {
    let animations = { () -> Void in
        self.label.alpha = 1
    }

    UIView.animate(withDuration: 2, animations: animations)
}
```

Y así conseguimos el efecto deseado: un botón hace desaparecer el *label* y el otro lo hace aparecer.

Completions en animaciones

Las animaciones que se ejecutan con UIView.animate corren en su propio hilo de ejecución. Cualquier código que tengamos después se ejecutará inmediatamente y en paralelo a la animación. Probemos, por ejemplo, a añadir el siguiente código después de la llamada a UIView.animate:

```
label.text = "Goodbye!"
```

Podríamos pensar que después de la animación el texto del *label* cambiará de «Hello World!» a «Goodbye!». Sin embargo, al ejecutar la aplicación nos daremos cuenta de que inmediatamente después de pulsar el botón *Fade Out* el texto del *label* cambia a «Goodbye!» y se ejecuta la animación. Esto es debido a que el código de la animación y nuestro código que cambia la propiedad *text* del *label* se ejecutan al mismo tiempo.

Para estos casos, UIView nos proporciona una versión de la función *animate* con un parámetro adicional, *completion*.

```
class func animate(withDuration duration: TimeInterval,
        animations: @escaping () -> Void,
        completion: ((Bool) -> Void)? = nil)
```

En este parámetro proporcionaremos el código que queremos que se ejecute una vez haya terminado la animación.

Abrimos el fichero CompletionAnimationViewController.swift correspondiente a la segunda pantalla del ejemplo. En la función *fadeOut* añadimos el siguiente código:

```
UIView.animate(withDuration: 2, animations: {
    self.label.alpha = 0
```

```
}, completion: { _ in
    self.messageLabel.text = "The label has disapeared"
})
```

En la *closure* del parámetro *completion* incluimos el código que queremos que se ejecute al finalizar la animación. En este caso, cambiamos el texto de otro de los *labels* de la pantalla.

Si ejecutamos la aplicación comprobaremos que el texto del *label* no se actualiza hasta que la animación ha terminado. En la *closure completion* podemos incluir el código que deseemos. Por ejemplo, podemos conseguir que después de ejecutar la animación se nos muestre la pantalla siguiente. Sustituimos el código de la *closure* por el siguiente:

```
self.performSegue(withIdentifier: "segueToCombined", sender: sender)
```

Animaciones combinadas

En ocasiones nos interesará animar las propiedades de varias vistas simultáneamente, y para conseguirlo no tenemos que hacer nada más que incluir todos los cambios que deseemos realizar en la *closure animations*.

Veamos un ejemplo en la tercera pantalla de la aplicación. Abrimos el fichero CombinedAnimationsViewController.swift y vamos a hacer que aparezcan los tres *labels* que tenemos en esta pantalla. En el *storyboard* hemos configurado los tres *labels* de manera que si no hacemos nada más serán visibles cuando se muestre la pantalla. Por lo tanto, primero los ocultamos una vez se hayan cargado los datos de la pantalla desde el *storyboard*, estableciendo su propiedad *alpha* en *viewDidLoad*:

```
override func viewDidLoad() {
    self.labelOne.alpha = 0
    self.labelTwo.alpha = 0
    self.labelThree.alpha = 0
}
```

Si ejecutamos la aplicación veremos que no aparece ningún *label* en la segunda pantalla. Después, en la acción del botón *animate* creamos la animación que cambia la propiedad *alpha* de cada uno de los *labels* a 1:

```
let animation = { () -> Void in
    self.labelOne.alpha = 1
    self.labelTwo.alpha = 1
    self.labelThree.alpha = 1
}
```

```
UIView.animate(withDuration: 2, animations: animation)
```

Y podemos comprobar si ejecutamos la aplicación que todos los *labels* aparecen a la vez.

¿Pero y si lo que queremos es coordinar las animaciones de manera que los *labels* aparezcan uno después de otro? Lo conseguimos de la misma manera que hemos visto en el punto anterior, incluyendo en la *closure* del parámetro *completion* el código que queremos que se ejecute una vez ha terminado la animación. En la *closure completion* de la primera animación incluiremos el código para ejecutar la siguiente animación, y así sucesivamente.

```
let animateOne = { () -> Void in
    self.labelOne.alpha = 1
}
let animateTwo = { () -> Void in
    self.labelTwo.alpha = 1
}
let animateThree = { () -> Void in
    self.labelThree.alpha = 1
}

UIView.animate(withDuration: 1, animations: animateOne, completion: { _ in
    UIView.animate(withDuration: 1, animations: animateTwo, completion: { _ in
        UIView.animate(withDuration: 1, animations: animateThree)
    })
})
```

Observemos que en este caso hemos usado el carácter _ en la definición de la *closure*. Así indicamos que, aunque la *closure completion* tenga parámetros en este caso, no los vamos a necesitar y los obviamos para simplificar.

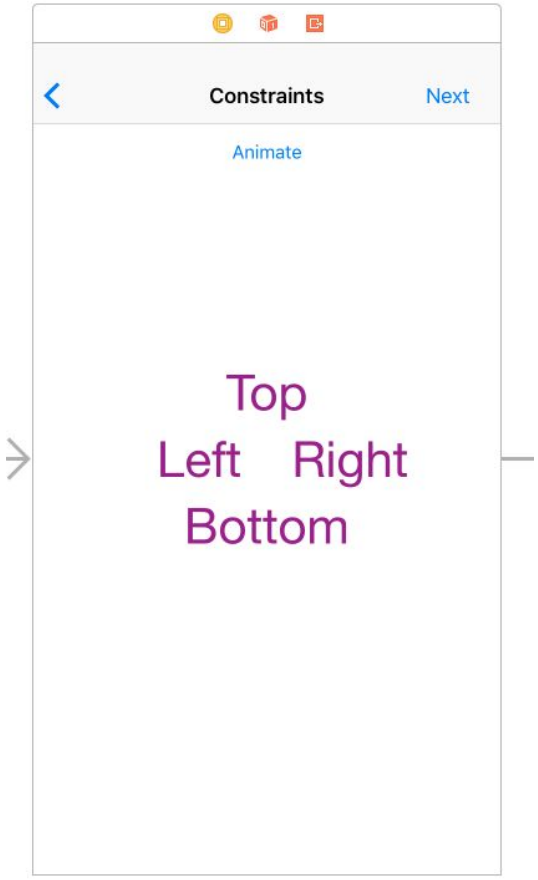
Animar la posición y el tamaño

En el apartado de Auto Layout vimos cómo definíamos mediante *constrints* las reglas que queríamos que se cumplieran para conseguir la maquetación en nuestras pantallas, y luego el sistema de Auto Layout se encargaba de realizar los cálculos necesarios.

Cuando se realizan animaciones de la posición y el tamaño de las vistas de nuestras pantallas, seguimos confiando en que Auto Layout calcule la disposición de todos los elementos en pantalla. Por

lo tanto, no vamos a cambiar directamente la posición y las medidas de las vistas, sino que vamos a modificar las constraints que se utilizan para calcularlas.

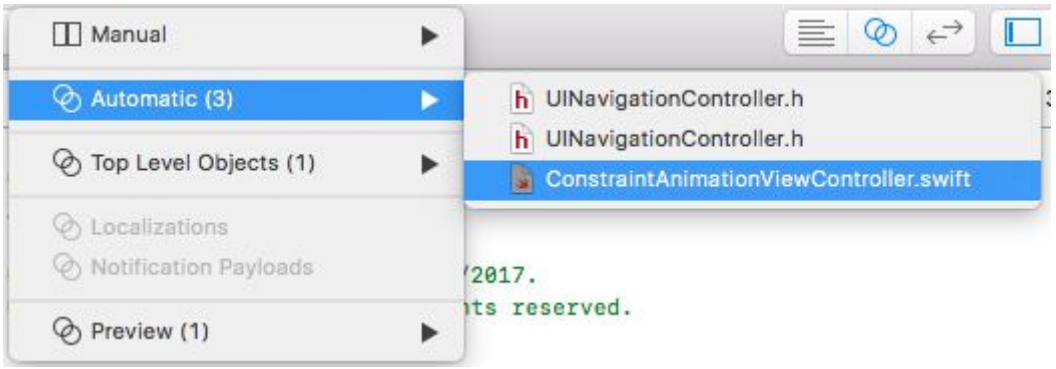
Partimos de la pantalla con todas las constraints configuradas y editamos el fichero ConstraintAnimationViewController.swift.



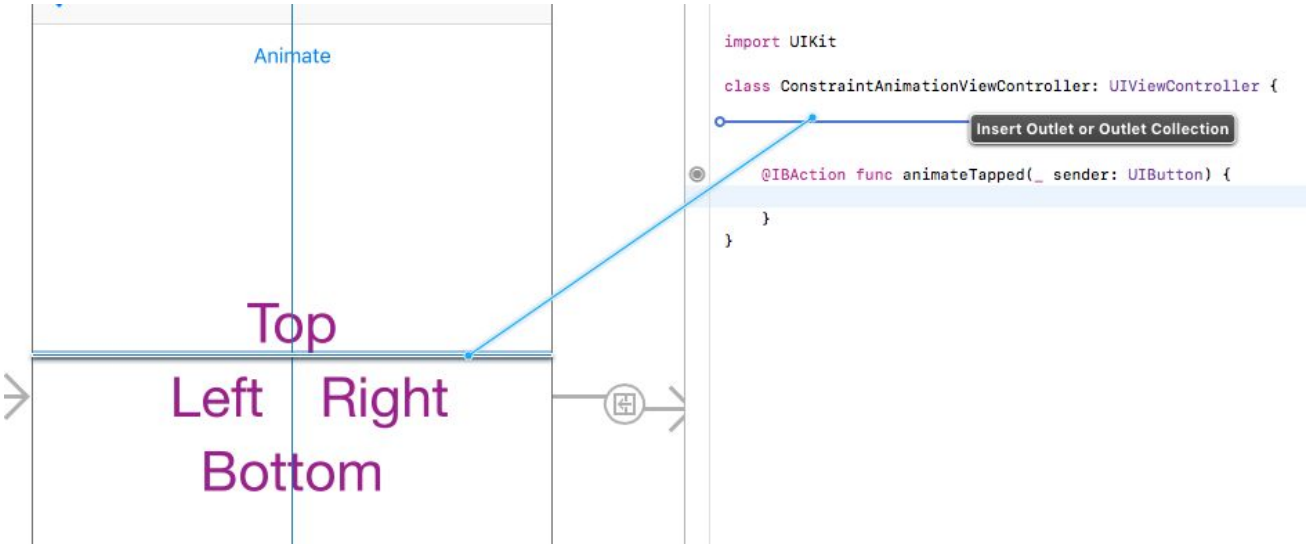
Para poder modificar una constraint, primero necesitamos enlazarla al *view controller* mediante un *outlet*. Activamos el *assistant director*:

En el editor izquierdo mantenemos el *storyboard* con la cuarta pantalla seleccionada y en el editor derecho nos aseguramos de que se muestra el código de su *view controller*, *ConstraintAnimationViewController*.

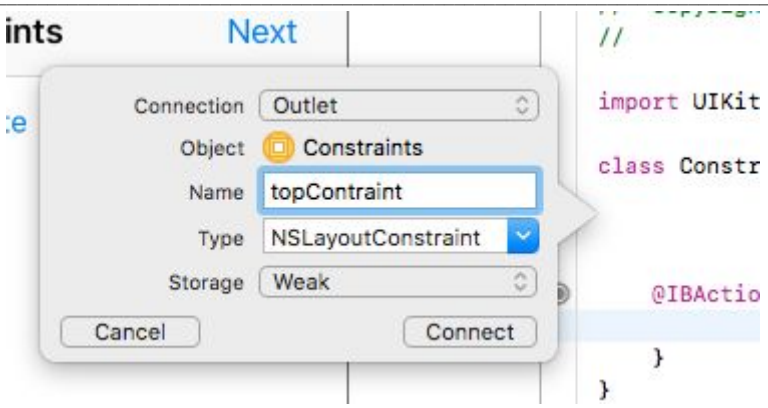
Si no lo tenemos en la barra del editor, lo seleccionamos; aparecerá dentro del menú *Automatic*:



Para añadir el *outlet* localizamos la constraint del *label Top* que determina su posición vertical.



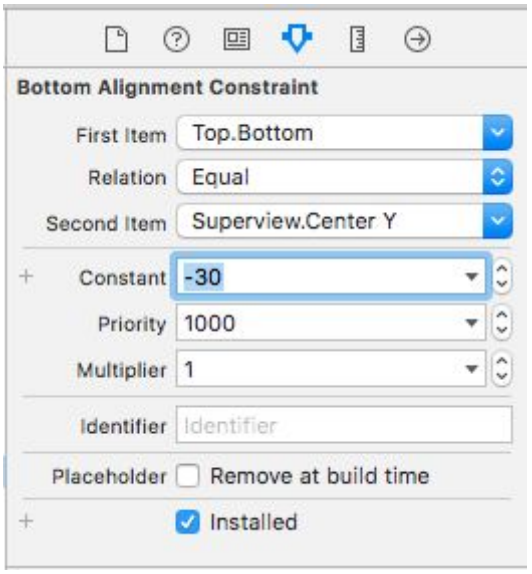
Y en la pantalla que nos aparece le indicamos el nombre *topConstraint*.



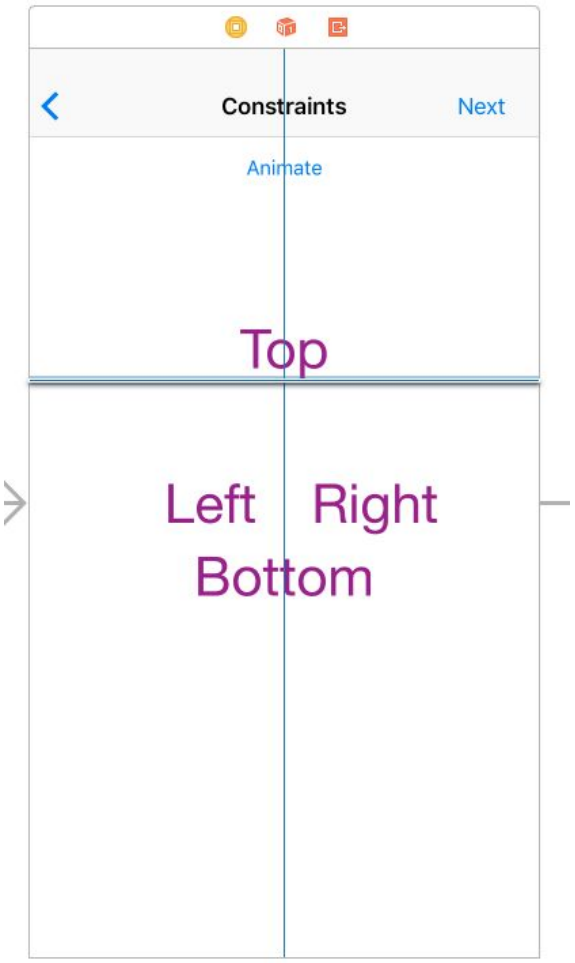
Y este será el *outlet* que se añadirá:

```
@IBOutlet weak var topContraint: NSLayoutConstraint!
```

Seleccionamos la constraint en el *storyboard* y vemos sus propiedades en *Attributes inspector*:



Vemos que esta constraint centra verticalmente el *label* en la vista que lo contiene, con un valor constante de desplazamiento de –30 (en este caso el valor negativo significa hacia arriba). Si escribimos otro valor en la propiedad *Constant*; por ejemplo, –90, veremos que el *label* se desplaza hacia arriba. Y volvemos a dejar el valor –30 que teníamos inicialmente.



La animación de este *label* consistirá en hacerlo desaparecer por la parte superior de la pantalla. Para ello, solamente nos hará falta saber el valor que debemos dar a la constante de la constraint.

Es una buena práctica definir la posición de los elementos en pantalla en relación con el resto de elementos, o sea que en vez de usar un valor fijo para esta constante de la constraint utilizaremos el valor de la altura de la vista que contiene el *label*. Si utilizáramos un valor fijo, sería más difícil que la animación funcione bien para todos los tamaños de pantalla existentes (o incluso si tenemos tamaños de pantalla distintos en el futuro). Este valor lo podemos obtener de la propiedad *frame* de la vista:

```
self.view.frame.height
```

La animación de las constraints funciona de una forma un poco distinta a la animación de las propiedades de las vistas que hemos visto antes. Primero, realizamos el cambio en la constraint, y en la *closure* del parámetro *animations* indicamos a Auto Layout que recalcule la posición de todas las vistas de la pantalla:

```
@IBAction func animateTapped(_ sender: UIButton) {
    self.topConstraint.constant = -self.view.frame.height

    self.view.layoutIfNeeded()

    UIView.animate(withDuration: 2, animations: {
        self.view.layoutIfNeeded()
    })
}
```

Es esta llamada a la función *layoutIfNeeded* de la vista principal de la pantalla es la que hace que Auto Layout recalcule la posición de toda su jerarquía de vistas. También es necesario llamar a *layoutIfNeeded* antes de empezar la animación. Si ejecutamos ahora la animación veremos como el *label Top* desaparece por la parte superior de la pantalla.

Después, podemos añadir los *outlets* para el resto de *constraints* para completar esta última pantalla. Animaremos las constantes para que el *label left* desaparezca por la izquierda, el *label right* por la derecha y el *label bottom* por la parte inferior:

```
@IBAction func animateTapped(_ sender: UIButton) {
    self.rightConstraint.constant = self.view.frame.width
    self.leftConstraint.constant = -self.view.frame.width
    self.topConstraint.constant = -self.view.frame.height
    self.bottomConstraint.constant = self.view.frame.height

    self.view.layoutIfNeeded()

    UIView.animate(withDuration: 2, animations: {
        self.view.layoutIfNeeded()
    })
}
```

Novedades en iOS 12 y Xcode 10

El día 17 de septiembre de 2018, Apple puso a disposición de los usuarios la versión 12 de iOS. Junto con esta nueva versión del sistema operativo vienen también pequeños cambios en Xcode y, en algunos apartados de Cocoa Touch, las librerías que utilizamos para desarrollar nuestras aplicaciones.

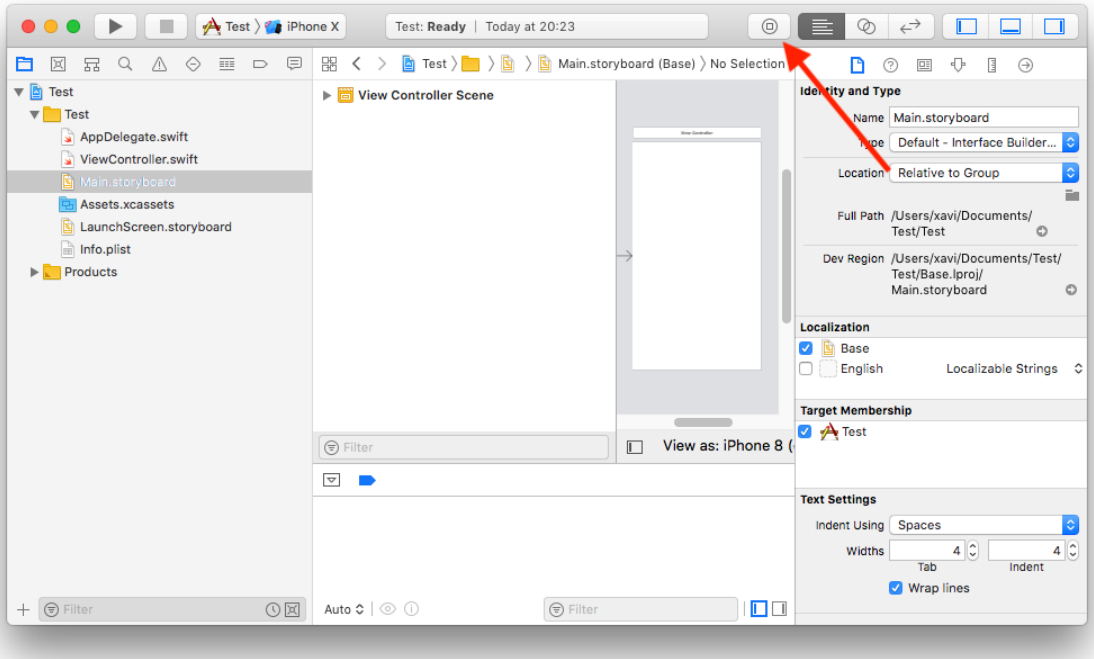
A continuación, describimos los cambios más significativos, aquellos que nos pueden afectar y que debemos tener en cuenta a medida que seguimos los materiales de la asignatura u otros recursos que consultemos.

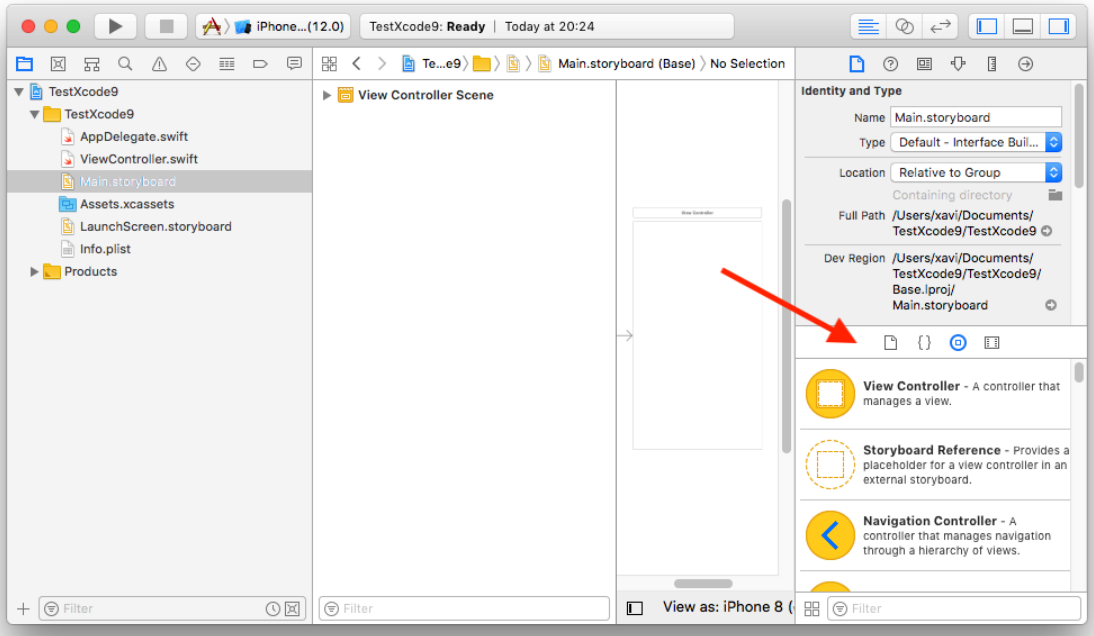
Cambios en Xcode 10

Biblioteca de componentes

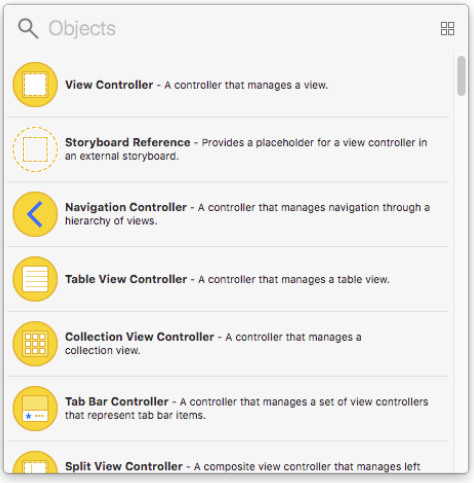
El acceso a la biblioteca de componentes se ha movido desde la parte inferior del *inspector* en las versiones anteriores.

En la versión 10 de Xcode ahora está accesible mediante este botón de la zona superior derecha de la pantalla:



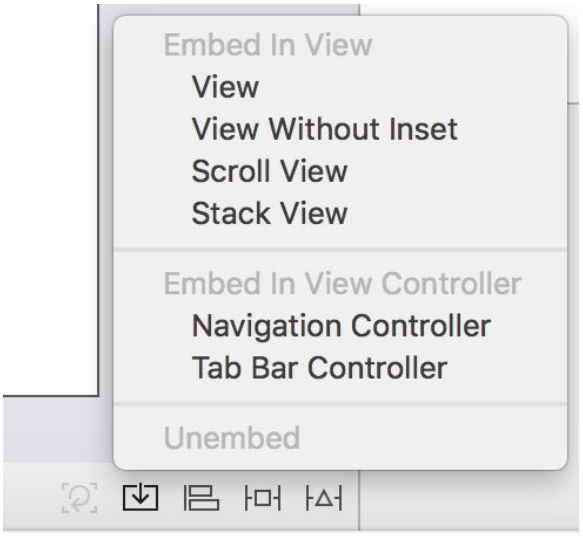


Cuando pulsamos el botón  se abrirá en una ventana la lista que antes estaba situada en el *inspector*:



Botón para incluir controles en *stacks*

En Interface Builder se ha ampliado el botón que permitía incluir controles dentro de un *stack*. Ahora incluye todas las opciones que permiten incluir controles dentro de otros contenedores.



En el siguiente enlace podemos consultar información adicional sobre los cambios en Xcode 10: https://medium.com/developerinsider/whats-new-in-xcode-10-fddeab035d05.

Cambios en el lenguaje Swift y UIKit

Xcode 10 viene con la versión 4.2 de Swift. Cuando Apple incluyó Swift como lenguaje de programación en UIKit se hizo una primera adaptación de las llamadas existentes en Objective-C a su versión correspondiente en Swift. Como podría ocurrir con los lenguajes naturales, esta primera traducción era muy literal respecto al Objective-C original. Por ello se puede decir que, aunque es Swift, se parece a Objective-C. Con las sucesivas versiones se va puliendo esta traducción y se van incorporando cada vez más características de Swift.

Cambios en la nomenclatura de tipos

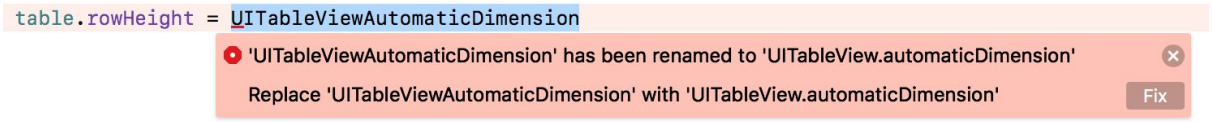
En las primeras versiones, los nombres de los objetos de UIKit se traducían directamente de su versión en Objective-C. Por ejemplo, en versiones anteriores teníamos esta variable global:

```
UITableViewAutomaticDimension
```

En la versión actual, se ha convertido en una propiedad del tipo *UITableView*:

```
UITableView.automaticDimension
```

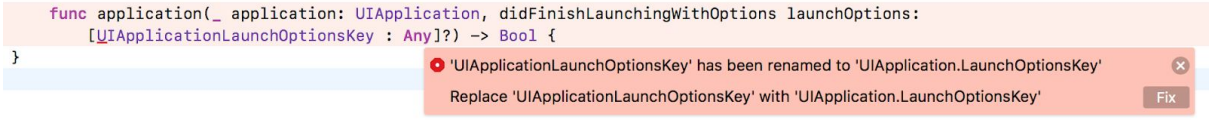
En la mayoría de los casos, el editor de Xcode ya se dará cuenta de que estamos intentando usar parte de la sintaxis obsoleta y nos propondrá corregir nuestro código de manera automática para solucionarlo:



Si pulsamos el botón *Fix*, el código quedará corregido:

```
table.rowHeight = UITableView.automaticDimension
```

Otro ejemplo, con el código en *AppDelegate*, lo tenemos con *UIApplicationLaunchOptionsKey*, que se ha cambiado a *UIApplication*.LaunchOptionsKey. Como en el caso anterior, Xcode ya nos propondrá la solución:



Y con *Fix* nos corregirá el código:

```
func application(_ application: UIApplication, didFinishLaunchingWithOptions
launchOptions: [UIApplication.LaunchOptionsKey : Any]?) -> Bool {
}
```

La razón de que se vayan realizando estos cambios es que en Swift disponemos de funcionalidades que no teníamos en Objective-C, como los tipos anidados y las propiedades de clase. Esto nos permite tener una nomenclatura más agrupada y lógica, mientras que, en Objective-C, para agrupar se tendía a añadir prefijos a los nombres de clases, propiedades y enumeraciones.

Añadir el prefijo *@objc*

El prefijo *@objc* se utiliza en Swift para que una clase, propiedad o función sea accesible desde el código en Objective-C. El caso es que, aunque nuestra aplicación esté desarrollada en Swift, hay veces que nuestro código debe interactuar con código en Objective-C; por ejemplo, UIKit todavía es Objective-C.

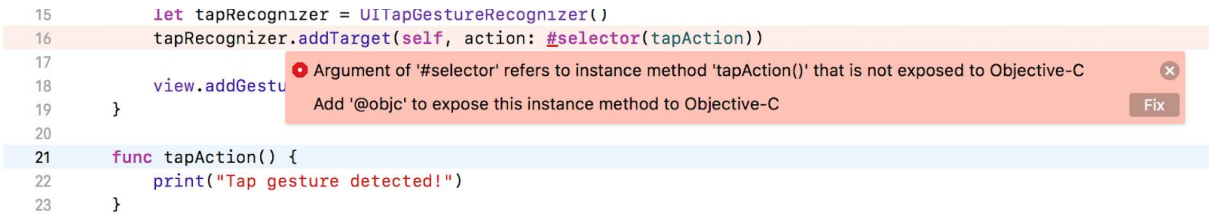
En las versiones anteriores, el compilador de Swift se encargaba de ir exponiendo automáticamente las partes de código Swift al resto de código Objective-C, pero se detectó que, al exponer de forma automática el código Swift, había muchos casos en los que se generaba código demasiado complejo, se exponía más de lo necesario y también había reducción de rendimiento. Por ello, en la versión actual se decidió restringir los casos en los que el código Swift se expone de forma automática.

Esto se traduce en que en algunos casos tenemos que prefijar nuestras funciones con *@objc* en casos en los que antes no era necesario.

Es el caso del reconocimiento de gestos en la interfaz de usuario, como en este ejemplo:

```swift
import UIKit

class ViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
        let tapRecognizer = UITapGestureRecognizer()
        tapRecognizer.addTarget(self, action: #selector(tapAction))

        view.addGestureRecognizer(tapRecognizer)
    }

    @objc func tapAction() {
        print("Tap gesture detected!")
    }
}
```

Si quitamos *@objc* de la función *tapAction*, el compilador se quejará de que no es capaz de completar la expresión de *#selector*. De todas maneras, Xcode también en este caso nos propondrá solucionarlo:



Podéis ver un ejemplo completo de reconocimiento de gestos en el capítulo 19 del libro, y tenéis el código de ejemplo con la solución en el área de recursos de la asignatura.

Como información adicional, podemos consultar esta página web, en la que podemos ver, de forma exhaustiva y con ejemplos, los cambios que ha habido entre las distintas versiones de Swift: https://www.whatsnewinswift.com.