Desenvolupament d'aplicacions per a dispositius iOS

Xavier Pereta

Introducció

En aquest material, començarem introduint els conceptes bàsics per poder desenvolupar aplicacions natives per a iOS. A continuació, comentarem algunes característiques bàsiques del sistema operatiu, els seus dispositius, l'entorn de desenvolupament i les eines que farem servir per desenvolupar les nostres aplicacions. A més, aprendrem a utilitzar el nou llenguatge de programació Swift.

Tots aquests coneixements els aplicarem més endavant en el terreny pràctic, ja que desenvoluparem una petita aplicació i, finalment, proposarem fonts addicionals per ampliar els nostres coneixements.

1. L'ecosistema iOS

Dispositius iOS

Els dispositius iOS són aquells dispositius d'Apple que executen el sistema operatiu iOS. Els podem categoritzar per famílies: iPhone, iPod Touch, iPad, Apple TV i, ara també, Apple Watch.

Tant l'Apple TV com l'Apple Watch utilitzen sistemes operatius derivats del sistema operatiu iOS, tvOS i watchOS, creats específicament per a ells, de manera que hi ha força diferències.

iPhone

El primer iPhone es va llançar l'any 2007 i només estava disponible als Estats Units, però, gràcies a la seva popularitat, va començar a haver-hi gent que el comprava allà i se l'emportava a altres països.

Després d'aquest primer iPhone han sortit al mercat molts models nous, concretament un per any. Habitualment, cada dos anys apareix una nova generació, i cada any es renova el dispositiu.

Els dispositius de cada generació es distingeixen amb un nombre (per exemple, iPhone 4 i iPhone 5), mentre que als models renovats d'una mateixa generació se'ls afegeix una lletra (per exemple, iPhone 5S i iPhone 5C).

El salt generacional sol portar més novetats que el salt de model, és a dir, entre l'iPhone 6 i l'iPhone 7 hi ha més novetats que entre l'iPhone X i l'iPhone XS.

iPod Touch

En paral·lel als dispositius iPhone va anar creixent la família dels iPod Touch, l'evolució de l'antiga família dels iPod, orientada inicialment a escoltar música, però amb la funcionalitat Touch i, a causa del boom de les aplicacions, amb moltes funcions.

A grans trets, els iPod Touch tenen característiques similars als iPhone, però no tenen la capacitat d'ús del telèfon mòbil. Moltes vegades també disposen de menys funcions, amb la qual cosa el preu també és inferior. Avui dia hi ha sis generacions d'iPod Touch.

De cara al desenvolupament, és important tenir present els iPod Touch, ja que, encara que no tenen tanta importància com la família dels iPhones, són una gran part dels dispositius dels usuaris. Així mateix, molts d'ells tenen algunes característiques peculiars: generacions sense micròfons, sense càmeres, sense flaix, amb botons en posicions diferents que els iPhone, etc.

iPad

De la mateixa manera que l'iPhone, l'iPad va ser un producte molt innovador en el moment del seu llançament, però inicialment va rebre algunes males crítiques, com que es tractava d'un iPhone gros i que no podria substituir l'ordinador de tota la vida. El temps no va donar la raó a les crítiques, i ara trobem moltes marques que comercialitzen la mateixa idea de tauleta.

El primer iPad es va presentar el 2010 i, actualment, ja se n'han venut més de 350 milions i s'han llançat 6 generacions. A aquestes hi hem de sumar tres generacions més de la versió iPad Mini i una generació d'iPad Pro. D'una banda, la versió iPad Mini és una tauleta més petita (7,9") que el model normal (9,7"). D'altra banda, l'iPad Pro afegeix una nova mida de pantalla (12,9") a l'iPad estàndard, a més d'un suport per a l'Apple Pencil.

Des dels seus inicis, els iPad s'han presentat amb dues versions diferenciades: una, més econòmica, amb connectivitat Wi-Fi i una altra que permet la connectivitat mòbil i que ofereix la possibilitat d'afegir una SIM per poder tenir connectivitat en qualsevol lloc amb cobertura mòbil.

Apple TV

Apple TV és un dispositiu pensat per estar al costat del televisor que permet visualitzar una sèrie de continguts disponible des d'Apple (pel·lícules, sèries, canals en línia, etc.). A més, permet visualitzar el contingut de l'ordinador a iTunes. Una de les característiques més destacades és que disposa d'AirPlay, és a dir, que permet accedir en directe al contingut d'un dispositiu mòbil per veure-ho a la televisió d'una manera molt còmoda.

Actualment hi ha cinc generacions d'aquest dispositiu. La segona i la tercera generació d'Apple TV funcionen amb una versió modificada d'iOS. La primera generació utilitzava una variació de Mac OS X, de manera que, tècnicament, no és un dispositiu iOS. El primer model data de 2007, tot i que llavors Apple considerava que era un producte dels que anomenaven *hobby* i no hi prestava gaire atenció.

El setembre de 2015 es va presentar la quarta generació d'Apple TV, i a partir de llavors el sistema operatiu d'aquest dispositiu és tvOS. Aquest inclou la seva pròpia App Store i permet desenvolupar aplicacions pròpies.

Apple Watch

L'Apple Watch és l'últim dispositiu presentat dins de la gamma iOS i la primera generació d'una nova família. En aquesta versió cal disposar d'un dispositiu iOS compatible per poder usar moltes de les

funcions del rellotge, ja que aquest té una versió d'iOS molt limitada. Pràcticament l'única interactivitat permesa inicialment per Apple es donava en el camp de les notificacions.

El dispositiu està orientat, sobretot, a treballar la condició física i al control de la salut de l'usuari, ja que incorpora un detector del ritme del cor, així com eines de seguiment de moviments per controlar l'activitat de l'usuari.

Tot i que es pot utilitzar sense iPhone, la majoria de les seves aplicacions requereixen disposar d'un iPhone 5 o superior.

2. El llenguatge Swift

Introducció

Swift és un llenguatge de propòsit general creat per Apple i distribuït sota una llicència de codi obert Apache 2.0. Va ser presentat l'any 2014 per reemplaçar Objective-C com a llenguatge utilitzat per a la construcció d'aplicacions per a iOS i macOS. Des de llavors, ha evolucionat fins a la versió 4 amb nombroses millores en què ha participat la comunitat de desenvolupadors. Fins al moment, Objective-C i Swift conviuen com a llenguatges de desenvolupament per a iOS, però Apple està apostant clarament per Swift, per la qual cosa és previsible que, en el futur, Objective-C quedi relegat al manteniment d'aplicacions antigues i tots els desenvolupaments nous es creïn en Swift.

Així mateix, aquest llenguatge està dissenyat des del principi per ser expressiu i eficient a l'hora d'escriure el codi. Com veurem al llarg d'aquest apartat, Swift té unes característiques que possibiliten que un codi sigui clar i estigui lliure d'errors. La seva sintaxi moderna i clara el fa més accessible que Objective-C, de manera que és molt més fàcil iniciar-se en el desenvolupament d'aplicacions. No obstant això, no ens hem de confiar, ja que no és, en absolut, un llenguatge simple, sinó que té molta potència i molts matisos. A més, introdueix alguns conceptes que veurem a continuació i que segurament no hàgim vist abans.

A la pàgina oficial de Swift podem trobar la documentació oficial del llenguatge, així com informació de com està organitzada la comunitat que s'encarrega del seu manteniment i evolució.

Sintaxi

La millor manera d'aprendre un llenguatge és utilitzant-lo, i per a això Swift disposa de Swift Playgrounds, una interfície de programació d'Apple que ens permet veure en temps real el resultat de l'execució del nostre codi. És una opció ideal per fer proves i aprendre d'una manera molt fàcil.

Playgrounds està disponible en macOS com un tipus de projecte en Xcode i també com una aplicació per a iPad.

Per crear un nou Playground podem obrir Xcode i triar l'opció «Playground» sota el menú «New». Per això la millor manera de seguir aquest mòdul és tenir obert Swift Playgrounds en el nostre Mac o iPad i anar provant tots els exemples de codi a mesura que avancem.

En Swift el clàssic exemple «Hello world!» consisteix únicament en aquesta línia:

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

No cal fer res més ni importar cap biblioteca addicional.

Al llarg d'aquest apartat veurem, mitjançant exemples, un resum de les característiques més importants de Swift i com utilitzar-les.

Valors bàsics

Hi ha diferents tipus de variables en Swift. Les més comunes són les següents:

- *String*, per a cadenes de caràcters com «Hello».
- *Int*, per als sencers, com 1, 2, 3, 4.
- *Double*, per als nombres amb decimals, com 1,3 o 0,5.
- *Bool*, per indicar cert (*true*) o fals (*false*).

En Swift les variables es declaren amb la paraula *var* davant, i les constants, amb *let*. Això permet indicar al compilador quins valors es modificaran (variables) i quins valors només s'inicialitzaran una vegada (constants). Aquesta distinció permet identificar d'una manera clara quins valors es volen

actualitzar en el codi i estalvia tota una categoria d'errors relacionats amb l'actualització de variables de manera inadvertida.

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

Cal aclarir que no és necessari que les constants tinguin valor en temps de compilació, però, si se'ls en dona, només es pot fer una vegada. A més, si són objectes, tot i ser constants, podrem modificar les seves propietats, però el que no podrem fer és assignar-los un nou objecte.

Una constant o variable ha de ser del mateix tipus que el valor que se li va a assignar. Però fixem-nos que en l'exemple anterior no li hem dit de quin tipus són cada un. En Swift el compilador pot deduir el tipus de variables i constants si els proporcionem un valor quan les creem. En l'exemple anterior el compilador dedueix que universalResponse és un enter perquè el valor inicial que li assignem és un enter.

També podem especificar directament el tipus si ho escrivim després del nom de la variable separat per dos punts:

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

En Swift es prefereix la deducció de tipus, de manera que només hauríem d'afegir el tipus a la declaració si és imprescindible, ja que, per exemple, no assignem un valor inicial a la variable.

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

L'exemple següent ens donarà un error, ja que estarem assignant dues vegades un valor a una mateixa constant:

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

Cal saber que mai no es realitza una conversió de tipus implícita. Si necessitem convertir un valor a un tipus diferent haurem de crear una nova instància del tipus que necessitem. Com en l'exemple següent, en el qual hem de crear una *string* a partir de l'*Int answer*:

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

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

Si intentem executar la línia següent ens donarà un error:

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

Per cert, hi ha una manera més senzilla de compondre *strings*, escrivint el valor entre parèntesi precedit d'una barra invertida (\):

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

Si estem utilitzant Playgrounds, anirem veient els resultats del nostre codi a la part dreta de la pantalla. Si no, per mostrar-ho a la consola en Xcode utilitzarem la paraula *print*:

```
print(answerMessage)
```

La funció *print* ens permet imprimir el valor d'una variable en la consola. La podem utilitzar per generar un registre de les operacions que realitza el nostre codi i verificar que funciona correctament, encara que en el subapartat dedicat a la depuració d'aplicacions veurem mecanismes més efectius.

```
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

Per crear *arrays* utilitzarem els claudàtors ([]) i accedirem als seus elements mitjançant l'índex de l'element entre claudàtors, també.

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

Per crear un *array* buit utilitzarem aquesta sintaxi d'inicialització:

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

Diccionaris

Els diccionaris són semblants als *arrays* però permeten especificar una clau per accedir a cada un dels seus elements. Aquesta clau identifica l'element dins del diccionari, pel que hem de proporcionar-la tant a l'inicialitzar-lo com quan desitgem recuperar-lo.

```
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 amb els *arrays*, podem crear un diccionari buit amb la sintaxi d'inicialització.

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

Control de flux

Les paraules *if* i *switch* serveixen per controlar condicions, i *for-in*, *for*, *while* i *repeat-while*, per a bucles. En tots els casos els parèntesis en la condició són opcionals, però les claus són obligatòries.

```
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 condició ha de ser explícitament de tipus booleà, per la qual cosa no podem fer servir resultats que no siguin booleans i esperar que es comparin implícitament amb zero o amb una *string* buida, com passa en altres llenguatges. Això és molt útil, perquè fa que el codi sigui més clar i ens estalvia tota una categoria d'errors causats per la confusió entre una assignació i una comparació.

Tots aquests casos donaran error de compilació:

```
if "" {
}
```

```
if 0 {
}

var boolVariable = true

if boolVariable = true {
}
```

L'altre condicional, el *switch*, també incorpora millores respecte al que segurament estem acostumats en altres llenguatges. A més, no cal afegir *break* en cadascun dels casos per evitar que salti directament a l'opció següent, però, si excepcionalment necessitem que l'execució salti al cas inferior, podem utilitzar la instrucció *fallthrough*. En els diferents casos podem comparar tot tipus de variables i també incloure més d'una condició per a cada un.

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

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

print(animalSound)
```

Utilitzarem *for-in* per iterar els elements d'una seqüència, ja sigui una matriu, un rang de nombres o els caràcters d'una *string*.

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

No existeix el clàssic bucle *for* a l'estil del llenguatge C amb inicialització o condicional, un increment que segurament ens resulta familiar d'altres llenguatges. Però obtenim el mateix resultat amb el *for-in* definint seqüències numèriques com la que veurem a continuació.

En *Swift* podem definir rangs amb la sintaxi següent: *inici...final*, o bé *inici ..<final*, si no volem incloure l'últim element, per exemple:

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

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

I aquest seria l'exemple de *for-in* utilitzant una seqüència numèrica:

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

Optionals

Una de les característiques més importants de Swift són els paràmetres opcionals, que ens permeten gestionar de manera més clara els casos en què un objecte o variable no conté cap valor, el valor *nil* (valor *null* o *nothing*, en altres llenguatges). Precisament en la mala gestió d'aquests casos tenim una font d'innombrables errors de programació.

En aquest enllaç podem veure una explicació dels perills que comporta la gestió de valors *null*: https://www.lucidchart.com/techblog/2015/08/31/the-worst-mistake-of-computer-science/.

Com veurem, el mateix llenguatge Swift ja ens proporciona un mecanisme per tractar de manera segura aquests casos. D'aquesta manera, a diferència d'altres llenguatges, no hem d'inventar la nostra pròpia estratègia i, a més, com que és part del llenguatge, serà la mateixa per a tot el codi que ens anem trobant escrit en Swift.

Per declarar un *optional* només hem d'afegir un interrogant (?) Després del tipus:

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

Però si el tipus fos *Int* (no *optional*), no li podríem assignar el valor *nil*, de manera que el codi ja no compila:

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

Fixem-nos que, encara que podem marcar una variable com *optional* perquè contingui el valor *nil*, no es pot intercanviar amb un valor *nil* d'un altre tipus *optional*, per la qual cosa donarà error.

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

L'acció de transformar una variable *optional* en una variable no *optional* s'anomena *unwrap*, i la seva sintaxi habitual és la següent:

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

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

Si executem aquest exemple veurem que el primer *print* ens mostra el text «Optional(20)», ja que estem tractant amb una variable *optional*. Si volem accedir al valor real haurem de fer un *unwrap* abans.

Extraiem el valor d'*optionalAge* i l'assignem a la constant *age*. Dins de l'*if* ja podem treballar amb aquesta constant *age* amb la seguretat que contindrà un valor. En el cas que *optionalAge* no contingui cap valor, el codi de dins de l'*if* no s'executarà.

Una altra manera de fer l'*unwrap* és afegint un signe d'exclamació (!); és el que s'anomena *forced unwrap*. Per exemple:

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

Només farem un *force unwrap* quan tinguem la seguretat que la variable contindrà un valor en temps d'execució, ja que si ho intentem amb una variable que conté *nil*, es produirà un error. L'ús de *force unwrap* hauria de ser excepcional en el nostre codi.

En Swift, podem continuar preguntant si una variable conté un valor *nil*, com faríem en un llenguatge que no suporta *optionals*, però ens resultarà un codi poc natural.

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

És preferible aquesta altra versió:

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

Hem de tenir en compte que, a més dels *optionals* que declarem en el nostre codi, haurem de tractar amb aquells que vindran com a valors de retorn de les funcions dels *frameworks* d'iOS. Per això, és molt important que ens acostumem a tractar-los correctament tal com hem vist en l'*unwrapping* i com veurem ara en l'*optional chaining*.

Optional chaining

Aquest concepte sorgeix de la necessitat d'accedir a propietats i funcions d'un objecte que pot ser que no contingui un valor en un moment donat. Si conté un valor, s'accedeix a la propietat o es crida la funció corresponent, però si no, ens tornarà a *nil*.

Habitualment s'utilitza per cridar una funció d'un objecte que està al final d'una cadena de crides sense haver d'anar comprovant cadascuna de les crides per assegurar-nos que cada un d'ells té un valor.

Per utilitzar *optional chanining* afegirem un interrogant (?) després d'*optional*.

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

Fixem-nos que si la variable *myMacBook* o la seva propietat *memory* fossin *nil*, el codi s'executaria sense errors, però la constant *diskSize* prendria el valor *nil*.

Si no tinguéssim *optional chaining*, hauríem d'escriure aquest codi d'una manera semblant a aquesta:

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

Optionals amb *unwrap* implícit

Els *optionals* amb *urwrap* implícit són un tipus d'*optionals* en què no cal afegir una exclamació (!) per fer l'*unwrap*. Estan pensats per simplificar el codi i evitar haver de fer contínuament *unwrap* al llarg de tot el codi. Solament els farem servir per a aquelles variables que utilitzem molt sovint, que poden no contenir un valor durant la fase d'inicialització del codi i que estem completament segurs que sempre tindran un valor després d'aquesta fase d'inicialització.

Mereixen una menció especial en aquest mòdul perquè s'utilitzen per definir les variables que ens permetran accedir als elements definits en l'*storyboard* de les pantalles en la nostra aplicació, de manera que els trobarem molt sovint. Es declaren amb una exclamació (!) després del tipus, en comptes d'una interrogació (?). Com veiem en aquest exemple, el seu *unwrap* és implícit:

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

Es tracta només d'un canvi en la notació per simplificar-ne l'ús, i només s'ha d'utilitzar en els casos esmentats, perquè, si contenen *nil*, l'*unwrap* fallarà igualment encara que sigui implícit.

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

Aquest és l'exemple típic que trobarem a les nostres aplicacions:

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

Funcions

Per declarar una funció utilitzarem la paraula clau *func*. I per crida-la farem servir el seu nom amb una llista de paràmetres entre parèntesis. Per especificar el tipus de retorn utilitzarem ->.

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

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

En Swift els noms dels paràmetres s'utilitzen per identificar-los en el moment que farem la crida de la funció. També podem especificar una etiqueta alternativa, és a dir, identificar el paràmetre en fer la crida de la funció amb un nom diferent del que hem fet servir en la declaració, o bé especificar _ com a etiqueta per obviar-la.

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

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

A més, podem utilitzar una tupla per tornar més d'un valor a les nostres funcions:

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

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

També podem tenir un nombre indefinit de paràmetres. En aquest cas els tindrem disponibles en un *array* dins la nostra funció:

```
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

Definirem una *closure* de manera similar a una funció però ometent el nom i amb la paraula clau *in* abans del cos:

```
{(Value1: Int, value2: Int) -> Int in
  return value1 + value2
}
```

Un dels usos de les *closures* en Swift i dels *frameworks* d'iOS és passar-los com a paràmetre. Per exemple, en aquest cas s'utilitzen per definir l'operació de mapatge dels valors de l'*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 sintaxi de les *closures* permet ometre la majoria dels seus elements per fer-les més compactes.

Podem ometre també els tipus, tant dels paràmetres com del valor de retorn, i fins i tot la paraula clau *return*:

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

També podem referir-nos als paràmetres pel seu nombre per generar expressions encara més compactes, la qual cosa deixa la *closure* en la seva mínima expressió.

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

Objectes i classes

Crearem una classe amb la paraula clau *class*. Les propietats de la classe les declararem com a variables i constants, que funcionen de la mateixa manera. Les funcions de la classe també les declararem com acabem de veure.

```
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())
```

Utilitzarem *init* per inicialitzar una instància de la classe. La paraula clau *self* la utilitzarem per referir-nos a la instància d'un objecte de la nostra classe.

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

Per exemple, en aquesta funció *pickUp* de la classe *Car*, *passengers* és el paràmetre de la funció i, en canvi, *self.passengers* és la variable de la instància.

Herència

Per especificar que una classe hereta d'una altra classe, afegirem el nom d'aquesta classe base després de dos punts (:). En Swift, a diferència d'altres llenguatges, no cal que totes les classes heretin sempre d'una classe 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")
```

Protocols

Un protocol defineix un esquema de funcions i propietats emmarcades en la realització d'una tasca o un objectiu en concret. És un concepte semblant al d'interfícies d'altres llenguatges de programació. Un tipus que compleixi amb tots els requisits d'un protocol es diu que adopta aquest protocol.

La sintaxi d'un protocol és molt similar a la d'una classe:

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

Per indicar que una classe adopta un protocol, l'afegirem després del nom de la classe separat per dos punts (:).

```
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)
```

A més, podem utilitzar un protocol de la mateixa manera que faríem servir una classe.

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

Una classe pot adoptar diversos protocols; de fet, aquesta és una part essencial del patró que s'utilitza per construir i organitzar la funcionalitat dels *frameworks* i aplicacions en iOS, com veurem en els propers mòduls. Per exemple, aquesta és la declaració d'una de les classes encarregades de gestionar el comportament d'una pantalla a una aplicació iOS:

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

La classe *MovementsListViewController* és heretada de la classe base *UIViewController* i adopta tres protocols. Cada un d'ells agrupa funcionalitats en un àmbit específic:

- *UITextFieldDelegate* s'usa per a la gestió de camps de text.
- *UITableViewDataSource* s'usa per gestionar les dades d'una taula.
- *UITableViewDelegate* s'usa per definir el comportament d'una taula.

En els apartats següents veurem com cal utilitzar aquests protocols.

Recursos addicionals

Per veure tots aquests conceptes del llenguatge amb més detall, és recomanable veure *Swift in Sixty Seconds*, una sèrie de vídeos curts acompanyats d'exemples de codi en què es repassen els conceptes bàsics de Swift: https://www.hackingwithswift.com/sixty.

Com a recurs molt més complet tenim la guia del llenguatge Swift a la seva pàgina oficial: https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html.

3. Xcode

Introducció a Xcode

Xcode és l'eina oficial per desenvolupar aplicacions tant per a iOS com per a tvOS i macOS. Aquesta eina ha anat evolucionant amb el temps alhora que apareixien noves versions d'iOS amb noves possibilitats i nous dispositius per poder desenvolupar nous tipus d'aplicacions i adaptar les existents.

Xcode és un entorn integrat de desenvolupament (IDE, de l'anglès *integrated development environment*), és a dir, incorpora totes les eines necessàries per dissenyar, codificar i construir les nostres aplicacions. Com a tal, no necessitem cap altra eina addicional.

No obstant això, hi ha altres eines per desenvolupar aplicacions per a dispositius iOS, però només Xcode és l'eina oficial d'Apple. La resta de les eines acaben utilitzant d'una manera o altra Xcode per poder generar els binaris finals de les aplicacions. Generalment, acaben fent ús de les eines de línia de comandaments incloses en Xcode per signar i generar els binaris finals.

Amb les diferents versions d'Xcode s'han anat incorporant diferents eines. Per exemple, fa uns anys eren necessàries dues aplicacions diferents per a desenvolupar aplicacions iOS: Xcode i un altre programa anomenat Interface Builder. La primera servia per desenvolupar el codi, mentre que amb la segona es creaven les interfícies d'usuari. A partir de la versió 4 d'Xcode, Interface Builder es va integrar també dins d'Xcode, la qual cosa va simplificar el desenvolupament en una única eina.

Versions d'Xcode

Les versions d'Xcode van a l'una amb les versions d'iOS. Quan apareix una versió nova d'iOS, també se'n llança una d'Xcode. Concretament, en la versió 5 es van afegir canvis significatius, semblants als canvis que es van produir en iOS 7 per simplificar la interfície al màxim. Per això, les versions Xcode 4 i Xcode 5 tenen bastants diferències. També es va facilitar la configuració de serveis com iCloud, Passbook i GameCenter, alhora que es va permetre crear bots que permetessin programar compilacions i tenir més control per trobar errors de programació. També es van incorporar millores de l'eina d'AutoLayout, que permet indicar com es comporten els elements amb diferents mides de pantalla.

En la conferència de desenvolupadors de 2014, es va presentar la versió d'Xcode 6. La principal novetat d'aquesta versió és el suport del llenguatge de programació Swift i els Swift Playgrounds, que ens permeten provar i desenvolupar mentre visualitzem els resultats en temps real. També va

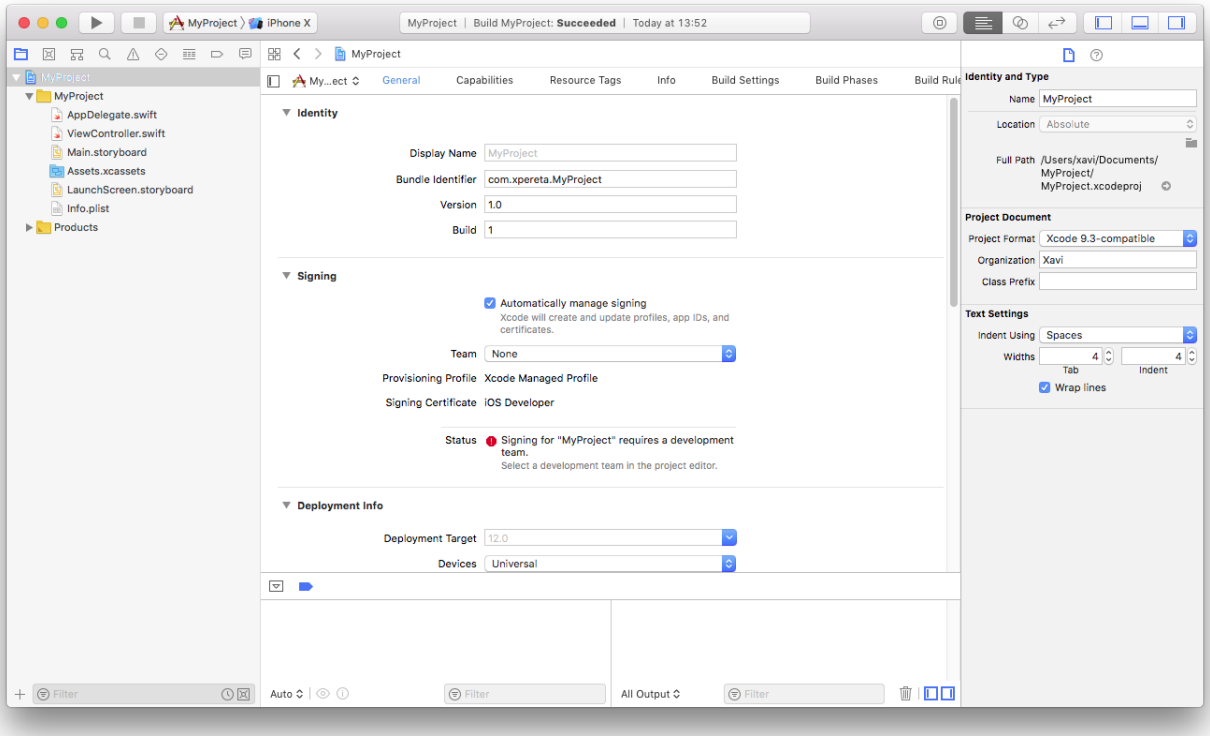
incloure noves funcionalitats per a desenvolupar la interfície d'usuari, com les Size Classes, per gestionar diferents mides de pantalla; Live Rendering, per veure de manera més fidedigna el resultat final mentre construïm la interfície, i View Debugging, que ens dona una representació gràfica en 3D interactiva de tots els elements que componen una pantalla molt útil per poder entendre com està construïda la pantalla i ajudar a solucionar problemes.

El 2015 es va presentar la versió d'Xcode 7 amb suport per a Swift 2 i Metal per a OS X. També incorpora la possibilitat d'executar i depurar les nostres aplicacions en dispositius sense necessitat d'adquirir una llicència de desenvolupador d'Apple. La versió d'Xcode 8 es va presentar el 2016 amb suport per a Swift 3. I la versió d'Xcode 9 es va llançar el 2017 amb millores destacables a l'editor de codi, suport per a la refacció a Swift i la possibilitat de depurar dispositius iOS sense necessitat d'una connexió permanent per cable.

Aquest any, el 2018, s'ha presentat l'Xcode 10. Aquesta versió té disponible una interfície fosca (*dark mode*) quan s'executa en macOS Mojave, a més d'un suport per afegir *dark mode* a les aplicacions que creem per a macOS (encara no disponible per a iOS). També té millores en la productivitat de l'editor de codi, com l'edició amb múltiples cursors. Incorpora un nou mecanisme de *build* que proporciona més rendiment quan generem les aplicacions.

Components

Un cop iniciat Xcode, se'ns mostra la pantalla següent:

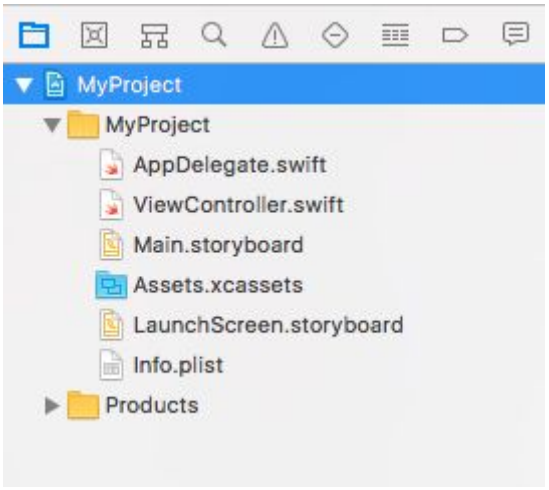


Navegadors

A la part esquerra, hi ha diferents navegadors. Són eines que ens donen accés a les parts del nostre projecte. A la part superior, hi ha una barra que permet anar alternant entre les diferents eines.

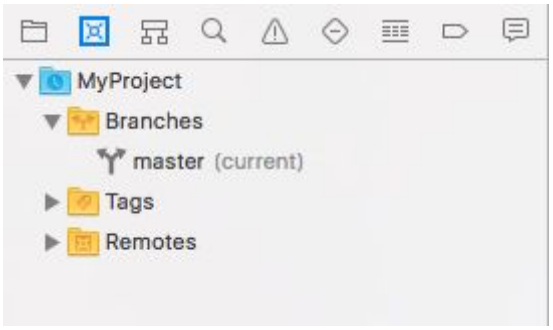
Project navigator

Ens proporciona accés als fitxers que formen part del nostre projecte. Des d'aquí afegirem els fitxers de codi font i també la resta de recursos necessaris, com icones, imatges, sons o fitxers de configuració. També permet ordenar i agrupar els fitxers.



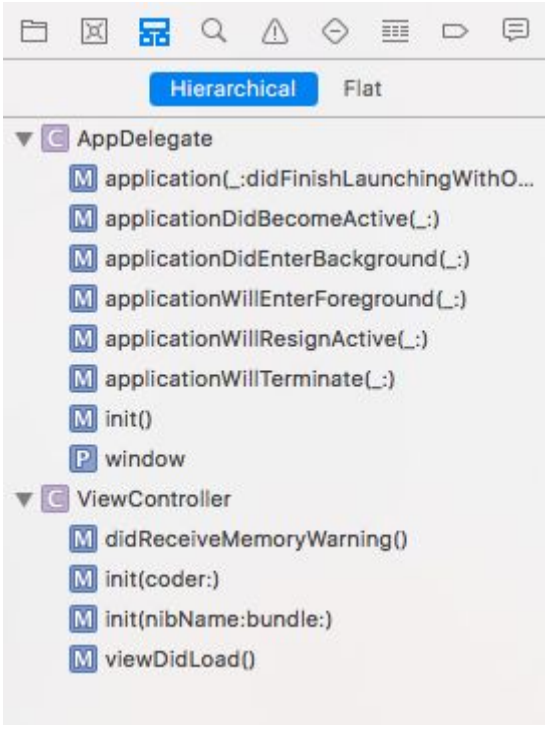
Source control navigator

Si el tenim enllaçat, per exemple, amb *git*, ens permet visualitzar l'estat del control de codi font del nostre projecte.



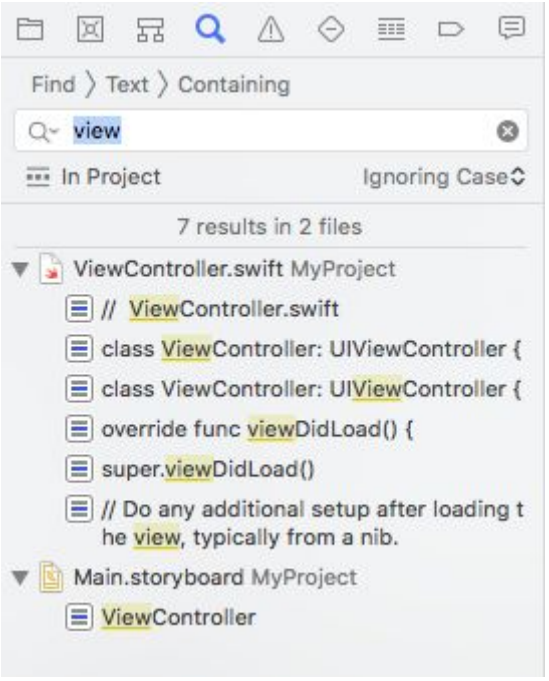
Symbol navigator

Ens permet navegar directament entre les propietats i funcions que estan definides en les classes del codi font del nostre projecte. Ens serà útil perquè el nom dels fitxers de codi no sempre reflecteix exactament el nom de les classes que hàgim definit i, a més, ens permet explorar les jerarquies d'herència de classes.



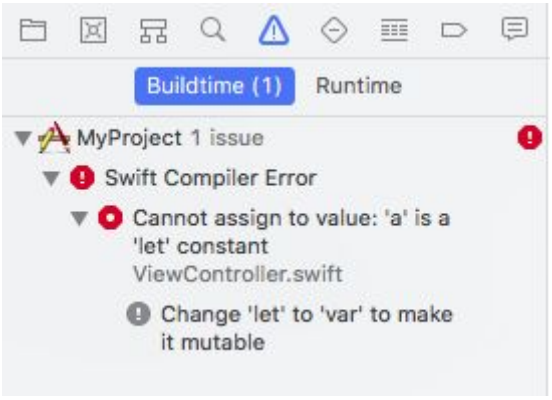
Find navigator

Permet buscar en tots els fitxers que formen part d'un projecte. Disposa d'opcions avançades de cerca mitjançant expressions regulars i també permet acotar l'àmbit de cerca a un projecte o directori en concret.



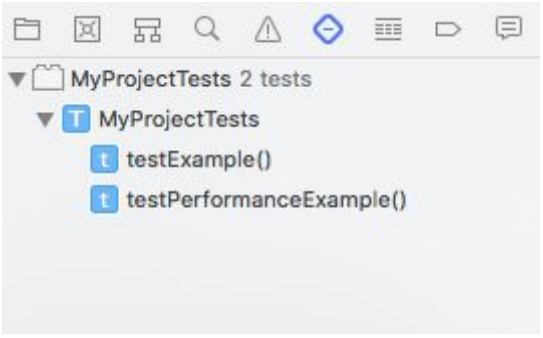
Issue navigator

Ens mostra els avisos i errors que Xcode ha detectat en un projecte, tant en la configuració com en el codi font. Si hi ha errors en el codi, en aquest *navigator* apareixerà una entrada per a cada un d'ells. En prémer sobre cadascun dels errors, ens obrirà el fitxer de codi font en la línia on s'ha detectat l'error seleccionat.



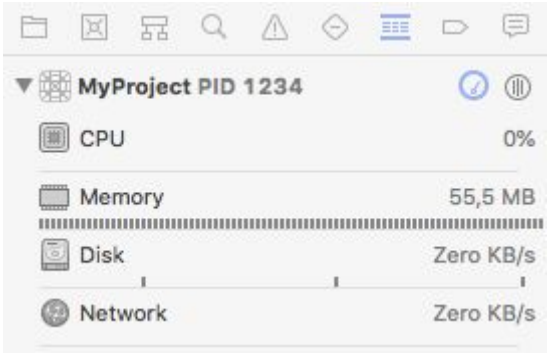
Test navigator

Els tests unitaris i d'UI que hàgim definit en el nostre projecte apareixeran aquí. Des d'aquest mateix *navigator* podrem executar-los.



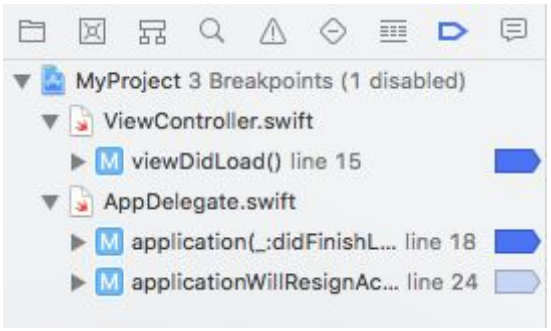
Debug navigator

En executar l'aplicació en mode *debug* des del simulador o dispositiu, se'ns mostra informació sobre els consum de recursos i informació del punt d'execució en què es troba l'aplicació en cada moment. Veurem en detall el seu funcionament més endavant.



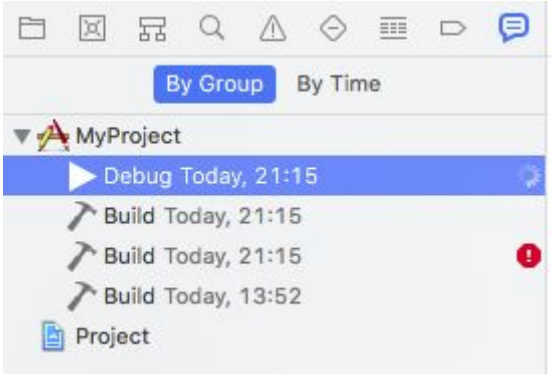
Breakpoint navigator

Mostra la llista de tots els punts d'interrupció que hàgim definit en el nostre projecte. Com veurem més endavant, és molt senzill definir punts d'interrupció sobre els fitxers de codi font a mesura que els necessitem. Però des del *navigator* podem gestionar-los de manera eficient, activar-los, desactivar-los i eliminar-los quan ja no els necessitem.



Report navigator

Dona accés a la informació i els *logs* de les operacions que ha realitzat Xcode; per exemple, quan li demanem que construeixi una nova versió executable de la nostra aplicació.

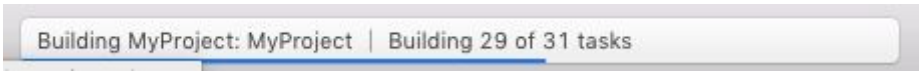


Eines d'execució

Permeten començar o aturar l'execució de l'aplicació tant en el simulador com en el dispositiu, si el tenim connectat.

Per seleccionar el model de dispositiu que volem simular o el dispositiu físic, premem sobre la icona del *selector* (en aquesta imatge s'identifica com «iPhone X»).

En prémer el botó d'execució, es construirà l'aplicació, s'arrencarà el simulador, s'instal·larà l'aplicació i s'iniciarà la seva execució automàticament.



La barra d'estat informa sobre les accions que està realitzant Xcode; per exemple, si està construint l'aplicació o executant-la en el dispositiu. També dona informació sobre si hi ha algun error i permet accedir directament al codi per solucionar-ho.

El botó de biblioteca de components proporciona accés als elements que es fan servir per construir les aplicacions. Es tracta d'un botó contextual, és a dir, ofereix diferents tipus d'elements en funció del tipus de document que es tingui obert en Xcode.

Els botons de la imatge anterior controlen les diferents maneres en què podem editar un document en Xcode:

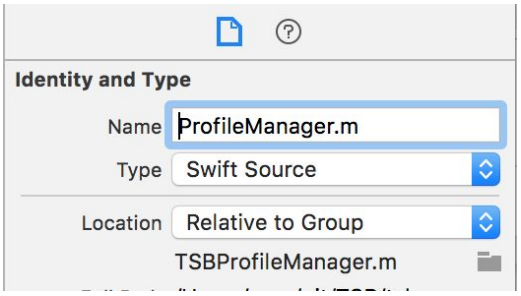
- **Editor estàndard**: mostra el document seleccionat en mode normal, ocupant la major part de l'espai disponible.
- **Editor assistent**: mostra dos documents en pantalla. Normalment intenta mostrar automàticament el document relacionat amb el que estem editant. Per exemple, si tenim obert un *storyboard* amb un *view controller* seleccionat, obre el fitxer Swift que conté el codi d'aquest *view controller*.
- **Editor de versions**: divideix l'editor en dos espais; en cada un d'ells hi ha una versió diferent del document perquè puguem veure les diferències entre múltiples versions d'un mateix document.

Aquest altre conjunt de botons ens permet activar i desactivar la visibilitat dels panells del navegador, l'àrea de *debug* i els *inspectors*. En funció del que fem en cada moment, ens pot convenir ocultar-los per disposar de més espai per editar el document de codi o d'interfície d'usuari.

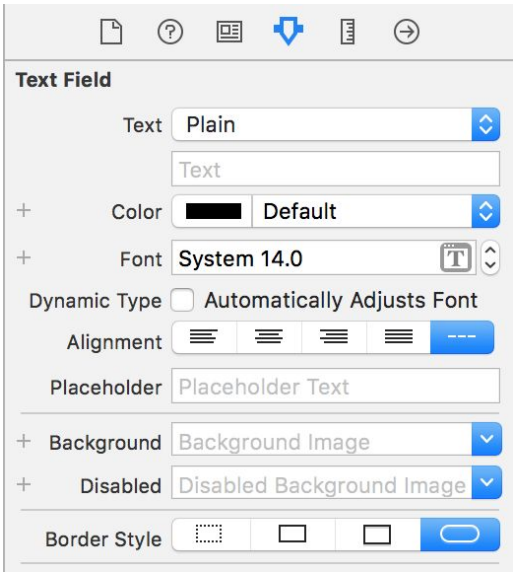
Utilities

Aquesta àrea ens permet configurar les propietats dels components de la nostra aplicació. És un espai contextual, és a dir, mostra diferents eines en funció del tipus de fitxer o de control d'interfície

d'usuari que tinguem seleccionat. Per exemple, si estem editant un fitxer de codi font, permet editar el seu nom.



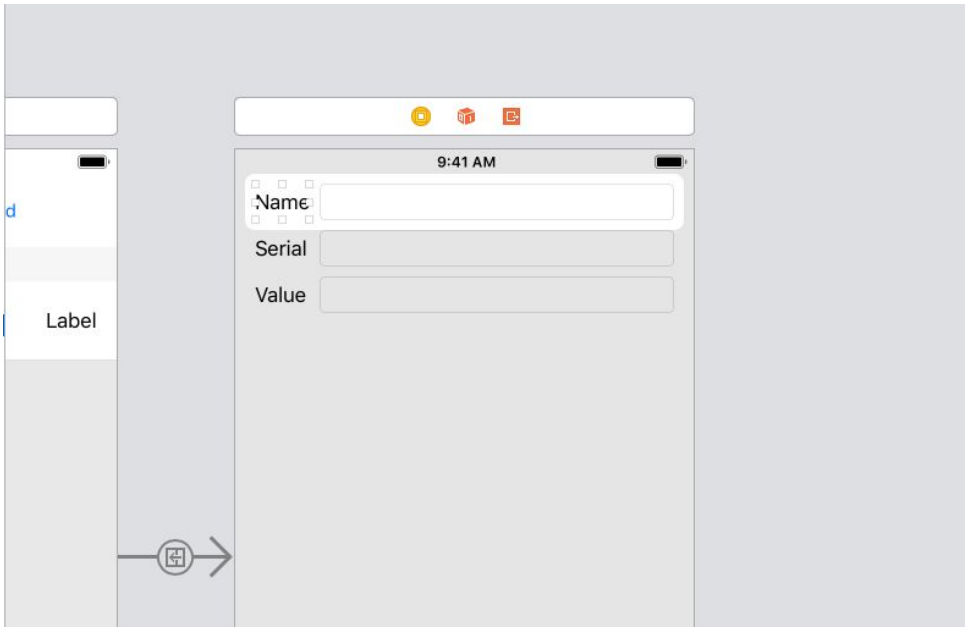
I si editem la interfície d'usuari d'una pantalla, mostra les eines per configurar les seves propietats i editar el seu aspecte.



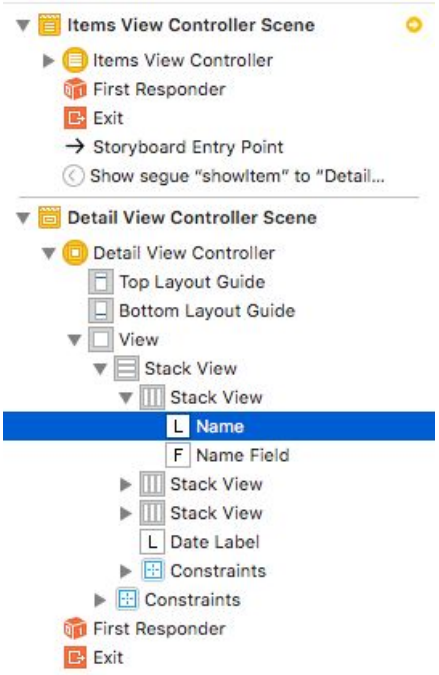
Interface Builder

Interface Builder és l'eina d'Xcode que permet desenvolupar la interfície d'usuari de les aplicacions. S'inicia automàticament quan s'edita un fitxer que representa la interfície d'usuari en una aplicació: *.xib* o *.storyboard*.

Interface Builder conté dues seccions. *Canvas*, un llenç on ubiquem tot els components d'interfície d'usuari de la nostra pantalla. Si s'edita la seva posició, mida i aspecte, mostra la pantalla següent:



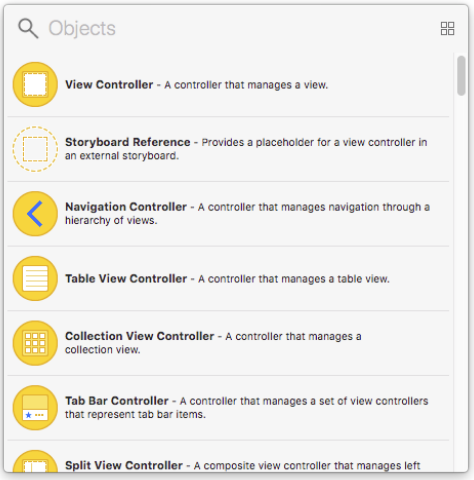
Document outline, una llista organitzada de manera jeràrquica amb tots els components de la interfície d'usuari.



Quan tenim pocs elements, podem treballar directament en el llenç, però en pantalles complexes és molt útil poder veure d'una ullada tots els components d'interfície d'usuari i poder seleccionar-los fàcilment independentment de la seva mida o posició a la pantalla.

En la versió 10 d'Xcode s'ha modificat la interfície per afegir elements a l'editor, ara és accessible al prémer aquest botó de la zona superior dreta de la pantalla:

Aquest botó obre en una finestra la llista que abans estava situada a la part inferior de l'inspector:



Zona de *debug*

Ens mostra informació que ens ajudarà a depurar la nostra aplicació quan s'està executant. A l'esquerra ens mostra el panell amb la inspecció d'expressions, i a la dreta, la consola de depuració. Al panell d'inspecció d'expressions podem explorar el contingut dels objectes de la nostra aplicació.

A la consola de *debug* se'ns mostren els missatges que genera l'aplicació en temps d'execució. I també podem executar comandaments per interactuar amb la nostra aplicació mentre s'està executant, la qual cosa ens resultarà molt útil per recopilar informació i resoldre problemes.

4. Exemple de construcció d'una aplicació

El patró MVC

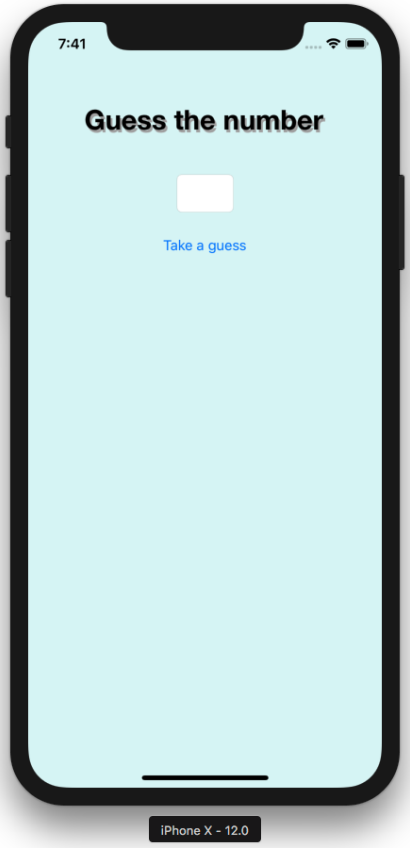
Abans de començar a dissenyar les nostres aplicacions és important que tinguem clar el patró model-vista-controlador (MVC) que s'usa en el desenvolupament iOS.

En MVC, totes les peces que componen l'aplicació pertanyen o a la capa de model, o a la de vista o a la de controlador.

- La **capa de model** representa les dades però no té relació amb la interfície d'usuari, ni, per descomptat, té coneixement de com les dades es mostraran a l'usuari.
- La **capa de vista** es compon dels objectes que són visibles per a l'usuari. Són aquells que acaben mostrant informació a l'usuari, però també els que reben les seves interaccions (les imatges, les caixes de text, botons, etc.).
- La **capa de controlador** és la que dona vida a l'aplicació. Decideix quin serà el comportament en cada moment. Consultant la capa de model, decideix com es configurarà la vista, i a partir dels esdeveniments que li arriben des de la vista decideix com es modificaran les dades del model.

Descripció de l'aplicació

Per fer-nos una idea de com és el procés de construcció d'una aplicació usant Xcode, construirem una aplicació senzilla: un joc en què l'usuari haurà d'endevinar un nombre de l'1 al 100 en el menor nombre d'intents.



Dissenyarem la nostra aplicació seguint el patró model-vista-controlador que hem descrit més amunt i descriurem cadascuna de les parts.

El model

El model gestionarà tota la informació per jugar al nostre petit joc. Ens caldrà emmagatzemar el nombre que caldrà endevinar, òbviament. També tindrà una funció per iniciar un nou joc i una altra per intentar endevinar el nombre.

Perquè el joc no sigui massa difícil i per fer-lo més interessant, el nostre model tindrà una altra funció per donar pistes. Així mateix, per a construir el nostre model crearem una classe nova a Swift.

La vista

La vista s'encarregarà de mostrar la informació a l'usuari; en el nostre cas, la petita pantalla que conté el títol, el *textfield* per introduir el nombre que s'ha d'endevinar, el botó per fer un intent d'endevinar i, finalment, la pista si no ho ha encertat.

També s'encarregarà de recollir el que faci l'usuari: el text introduït i l'acció del botó.

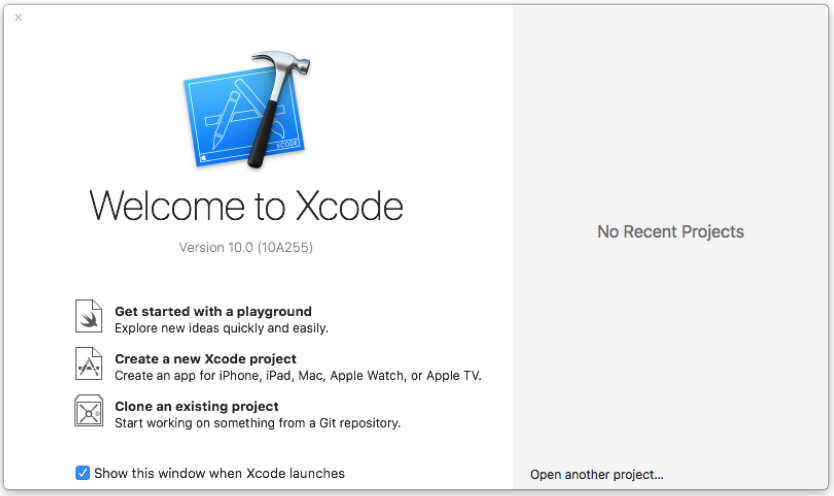
Per construir la vista utilitzarem les vistes i controls que ens proporciona UIKit, que editarem en un *storyboard* a Xcode.

El controlador

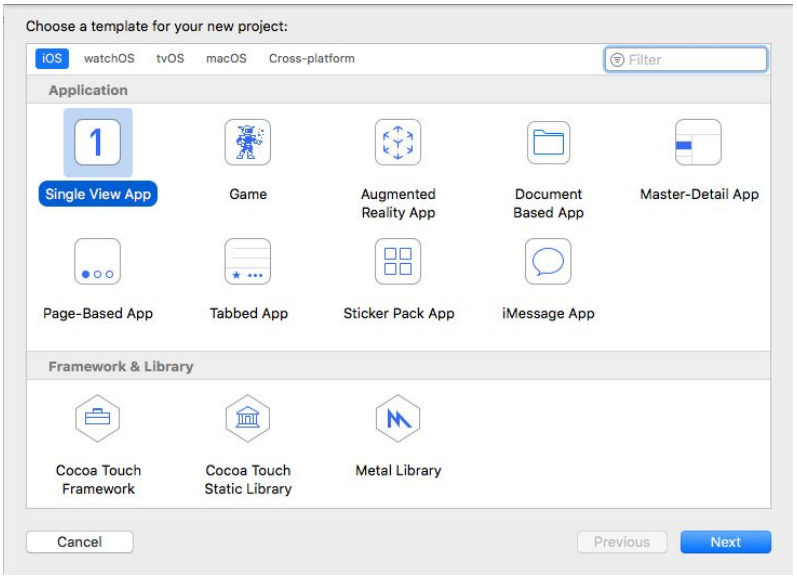
El controlador utilitzarà la informació que li proporciona el model per actualitzar la vista, i també en funció dels esdeveniments que li arribin de la vista decidirà com actualitzar el model. El controlador el construirem amb una classe que hereta d'*UIViewController*, que és la classe de la qual hereten tots els controladors de vistes en UIKit.

Creació del projecte en Xcode

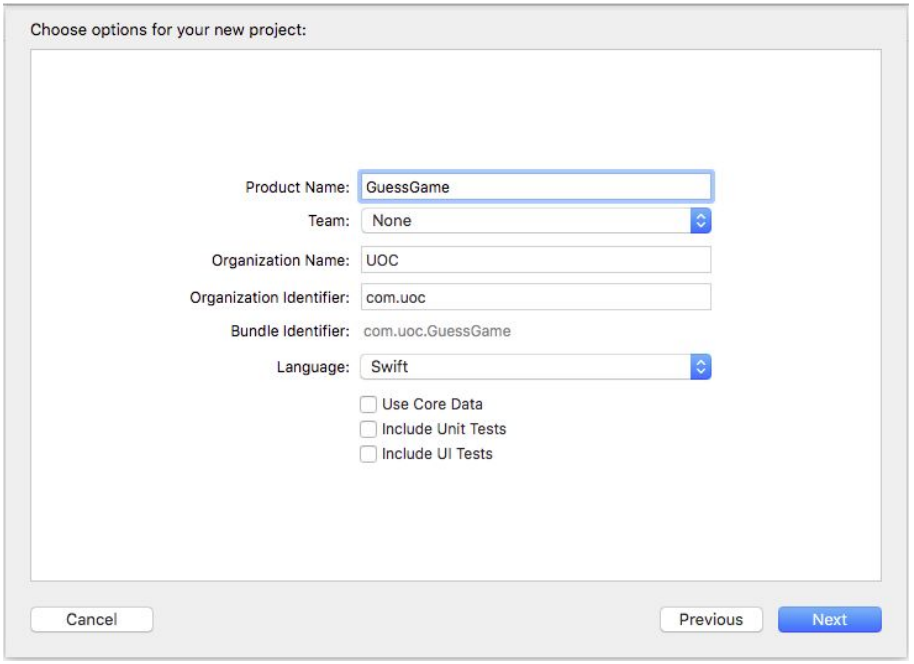
Obrim Xcode i seleccionem l'opció *Create a new Xcode project*:



En la pantalla següent, seleccionem la plantilla *Single View App*:



A la pantalla següent, ens demanarà el nom i les dades addicionals per crear la nova aplicació:



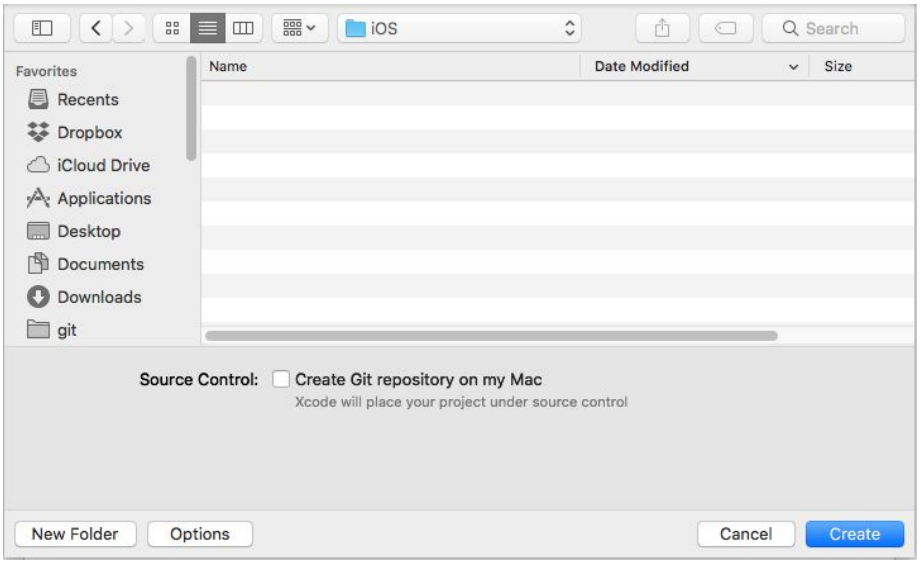
En el camp *Product Name*, introduïm el nom que vulguem, per exemple «GuessGame». Aquest serà el nom també del directori i el nom del fitxer que contindrà el projecte Xcode en el sistema de fitxers del nostre Mac.

En el camp *Organization Name*, introduirem el nom de la nostra empresa o el nostre nom.

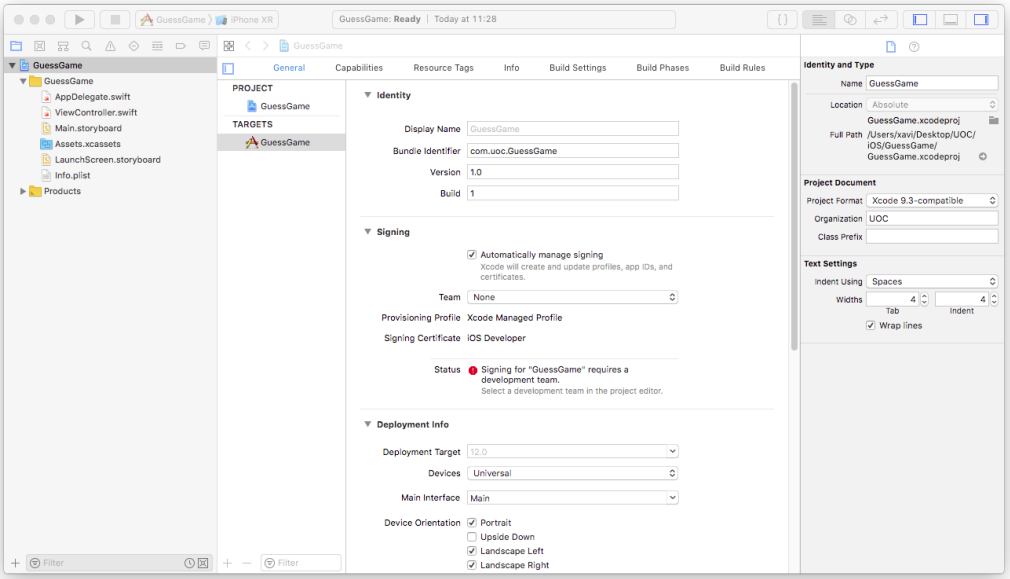
El camp *Organization Identifier* s'utilitza per identificar els desenvolupadors en el sistema d'Apple. Té un format com un DNS invers, però compte, ni té relació ni està associat a cap domini d'internet. Per exemple, podem usar «com.uoc».

El *Bundle Identifier* es genera a partir dels dos camps anteriors i identificarà de manera única la nostra aplicació en els sistemes d'Apple. A continuació, ens assegurem que Swift està seleccionat en el camp *Language*.

De moment deixarem sense marcar les opcions de *Core Data*, *Unit Tests* i *UI Tests*. En prémer *Next* ens demanarà que indiquem on crearem el projecte Xcode en el disc:



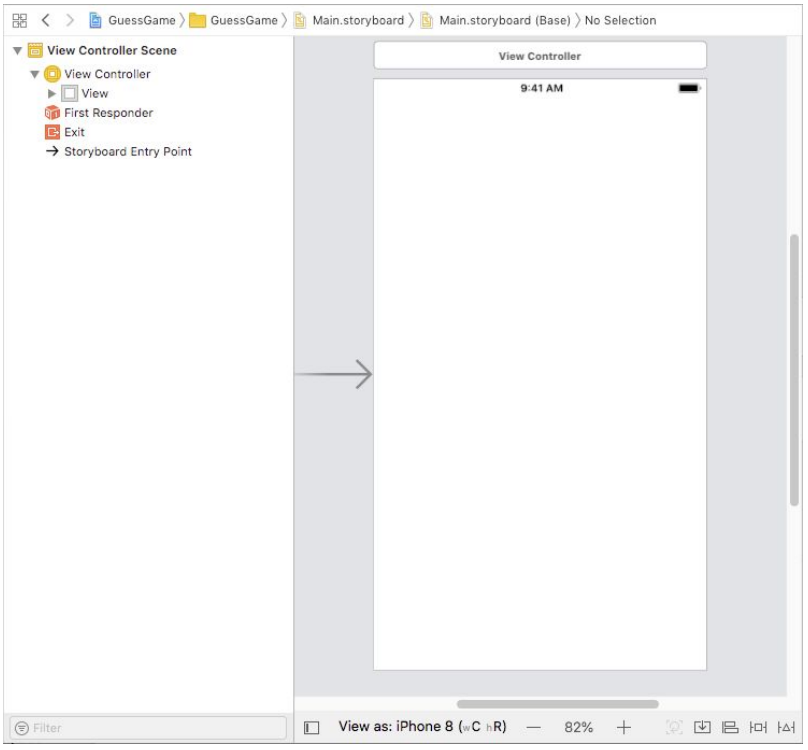
I ja tenim el projecte creat en Xcode:



Construcció de la vista

Primer crearem la vista, que ens ajudarà a fer-nos una idea de com funcionarà l'aplicació.

Seleccionem el fitxer Main.storyboard al *document outline* i l'editor d'Xcode ens mostrarà el següent:

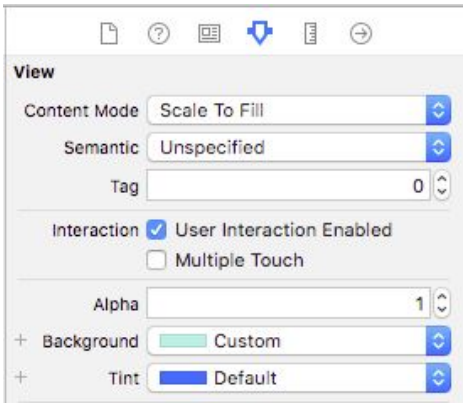


La nostra aplicació tindrà una única pantalla i la plantilla que hem seleccionat en crear el projecte ja ens la proporciona, en blanc. Utilitzant el botó *Library* anem afegint els quatre elements que componen la interfície d'usuari.

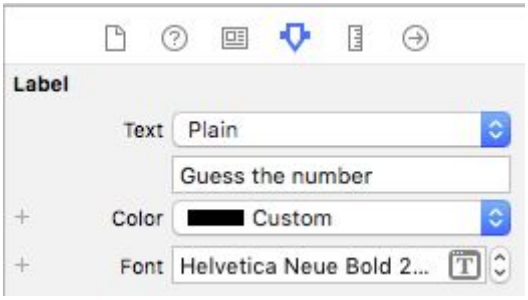
Afegim un *label*, que serà el títol de la nostra pantalla. A sota d'aquest, un *text field* perquè l'usuari introdueixi el nombre que cal endevinar. Després, un *button* per iniciar un intent d'endevinar el nombre. Finalment, un altre *label* per mostrar les pistes. Quedarà alguna cosa semblant a això:

Seleccionant cada un dels controladors personalitzarem el seu aspecte per aconseguir el disseny que hem vist a la pantalla al principi de l'apartat.

Seleccionem la vista principal i en l'*Inspector* modifiquem la propietat *Background*:

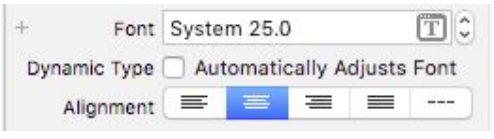


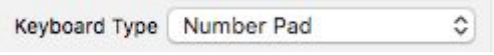
Seleccionem el primer *label* i introduïm el text del títol «Guess the number». També podem fer-ho amb doble clic sobre el propi *label*. Modifiquem també la propietat *Font* per utilitzar una Helvetica Neue Bold de 29 punts.



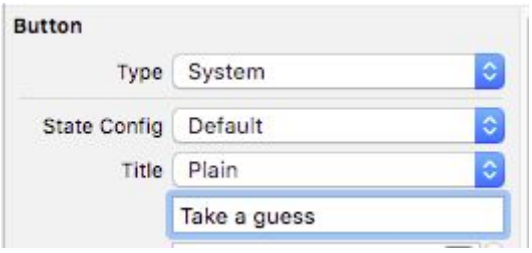
També li donarem una ombra donant valors a *Shadow* i *Shadow Offset*:

Seleccionem el *text field* i canviem la font a 25 punts, l'alineació del text centrada i, finalment, el configurem per mostrar un teclat numèric:





Per al botó, configurem el text com en el cas del *label*, amb el valor «Take a guess»:

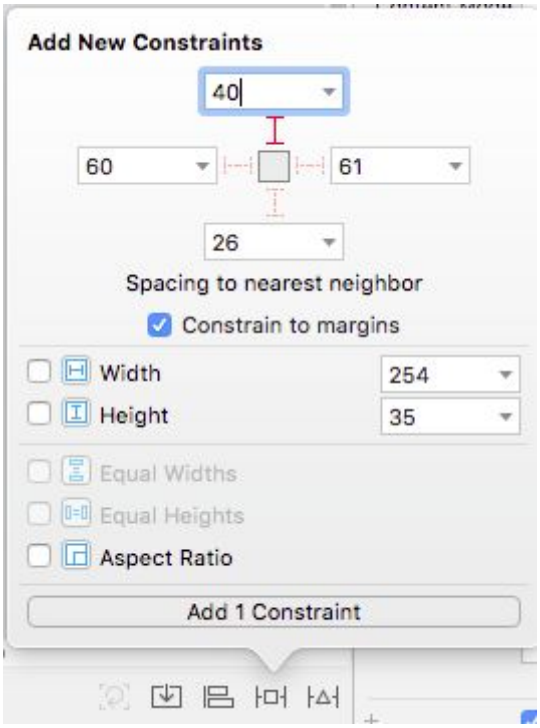


Finalment, en l'últim *label* hi assignem també un text: «Hint label». I hi configurem un color més discret:

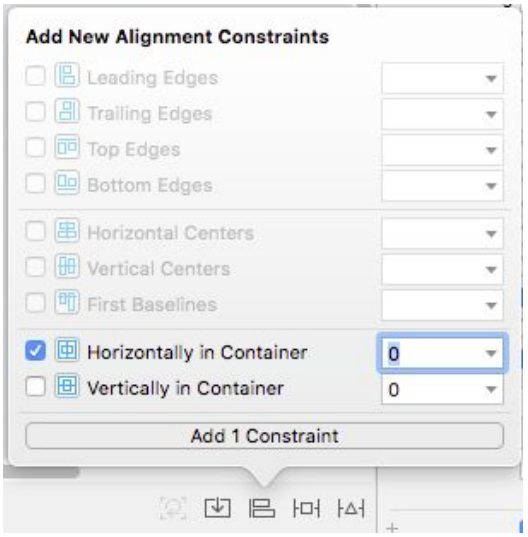
La pantalla ens queda així, ara:

Ara afegirem les constraints perquè la pantalla s'ajusti correctament a les diferents mesures de pantalla dels dispositius.

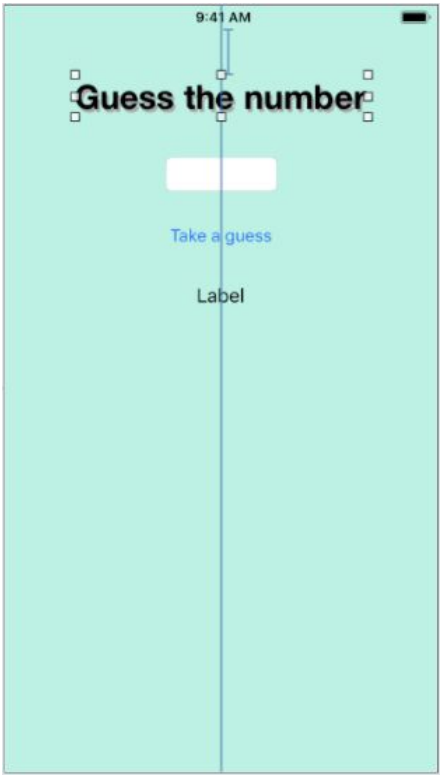
Seleccionem el *label* del títol i premem el botó per afegir noves constraints a la part inferior dreta de l'Interface Builder. Indicarem que volem que el *label* estigui a 40 punts de distància a la vora superior. Premem *Add 1 Constraint*.



Amb el *label* encara seleccionat, premem el botó per afegir constraints d'alineació i seleccionem «Horizontally in Container». Això farà que el *label* estigui sempre centrat en la vista que el conté, en aquest cas a la vista que comprèn tota la pantalla. Premem *Add 1 Constraint*.

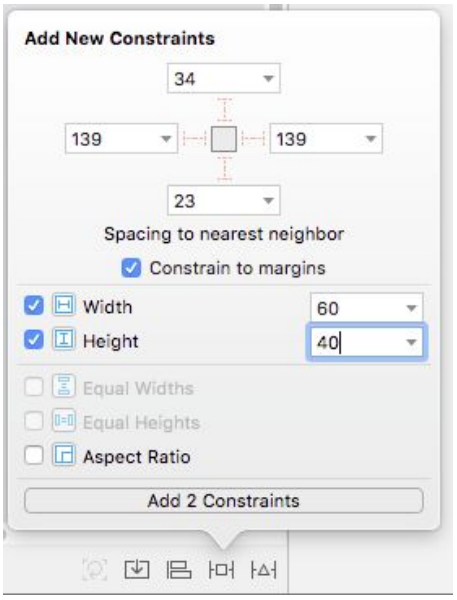


Ja tenim les constraints necessàries per al títol:



A continuació, seleccionem el *text field* i afegim una constraint de 40 punts fins a l'element superior i una altra de centrat en el contenidor, igual que per al títol.

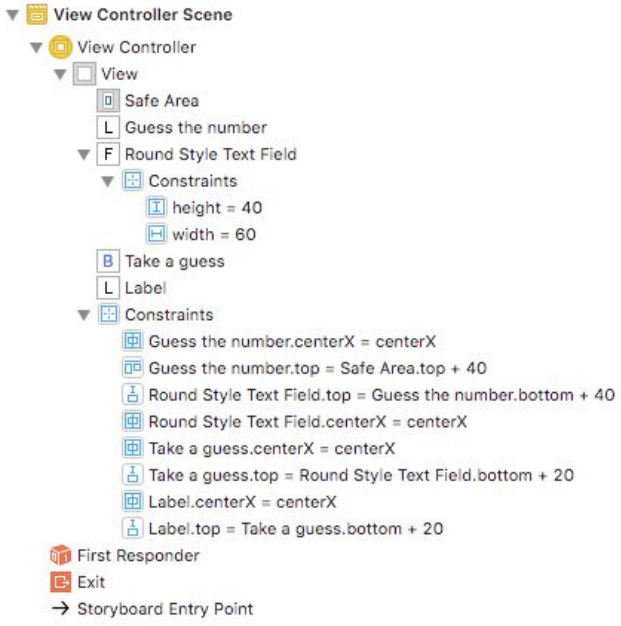
A més, afegirem constraints per definir la seva mida: especificarem 60 d'ample i 40 d'alt.



A continuació, ens centrarem en el *button*. Hi afegirem una constraint de 20 fins a l'element superior i una altra de centrat.

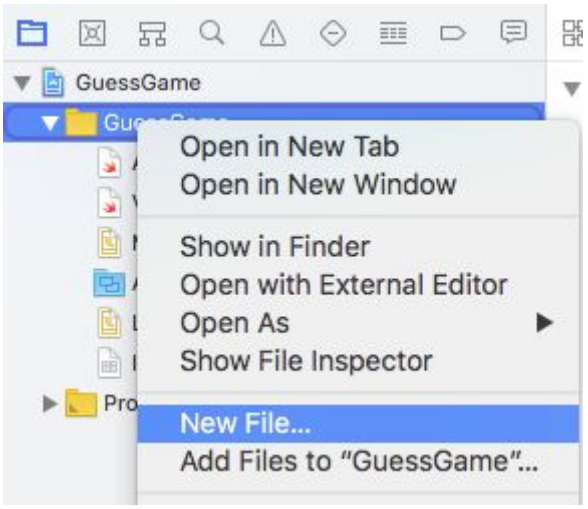
Finalment, en el *label* per a les pistes afegim una constraint de 20 fins a l'element superior i una altra de centrat.

Desplegant la vista del *document outline*, haurien d'aparèixer les constraints creades següents:



Construcció del model

El model serà una classe en Swift. Per crear-lo, ens situarem sobre el *document outline* en el grup *GuessGame* i amb el botó dret seleccionem l'opció *New File*...



En la pantalla següent seleccionem la plantilla *Swift File* i *Next*. A continuació, ens demanarà el nom del fitxer que volem, escrivim «GuessModel» i premem *Create*.

I aquest serà el codi de la nostra classe per al model:

```
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!"
        }
    }
}
```

Tenim tres propietats per guardar el nombre que cal endevinar, el nombre d'intents des que es va iniciar el joc i l'últim nombre que s'ha intentat endevinar. La funció *newGame* configura els valors de totes les propietats per iniciar un nou joc; mentre que la funció *guess* la cridarem per intentar endevinar si el nombre és vertader o fals.

La funció *giveHint* ens retornarà una *string* amb una pista basada en l'últim nombre que hem intentat endevinar. Aquí és interessant veure que fem servir un *guard* per tractar el cas especial en què no tenim un nombre anterior, perquè encara no hem intentat endevinar el nombre.

Aquesta classe té implementada una funcionalitat que té sentit per ella mateixa i és completament independent tant pel que fa a la vista com al controlador. Podríem utilitzar-la talment per implementar aquest mateix joc utilitzant vistes completament diferents.

Construcció del controlador

El controlador és el que donarà vida al nostre joc. Utilitzarà el model per dir a la vista el que ha de mostrar i, amb els esdeveniments que rebi de la vista, modificarà el model.

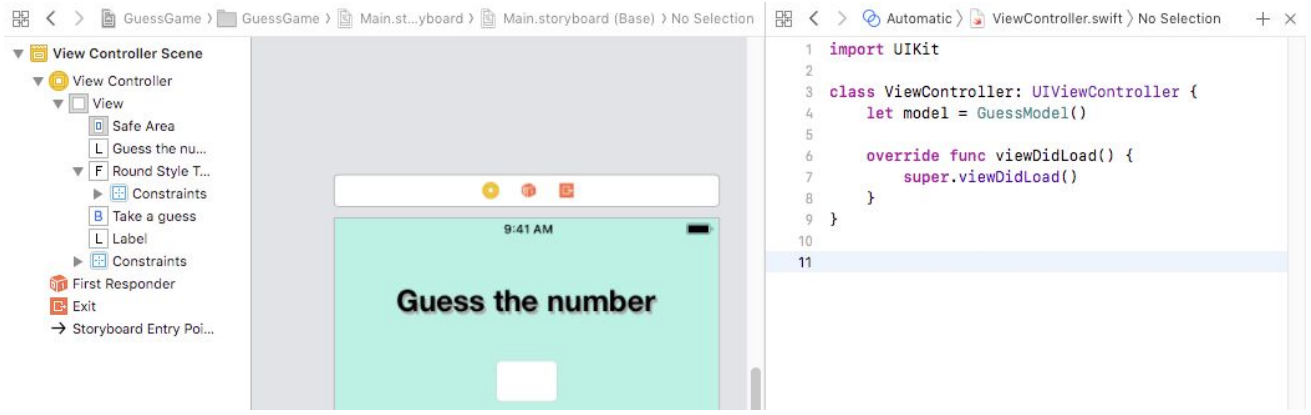
La plantilla que hem fet servir per crear el projecte en Xcode ja ens ha creat un controlador i l'ha associat a la nostra vista. El tenim en el fitxer ViewController.swift. El primer que hem de fer és crear una instància del model per tenir-lo accessible en el nostre controlador.

```
class ViewController: UIViewController {
```

```
    let model = GuessModel()

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

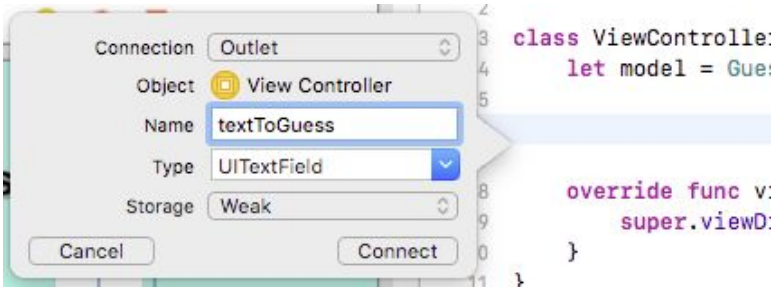
Ara connectarem els controls de la vista amb el controlador per poder accedir-hi des d'aquest. Per a això obrirem el fitxer Main.storyboard, que conté la nostra vista, i obrirem l'*Assistant Editor*:

S'obrirà l'editor d'Xcode amb una divisió vertical, a l'esquerra, l'*storyboard* amb la nostra vista, i a la dreta, el codi del nostre controlador:



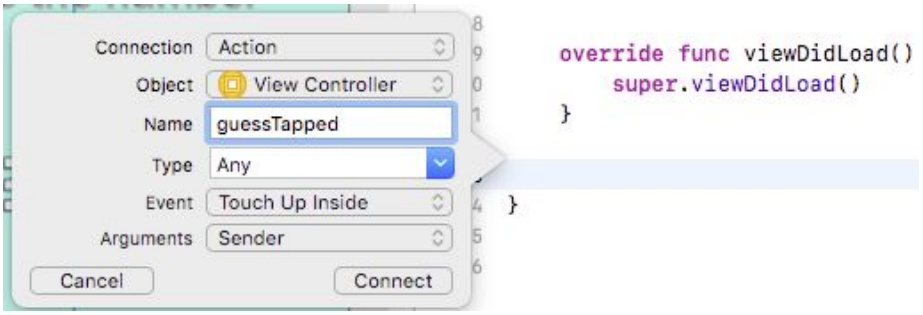
Si no ens apareix el codi del controlador, el podem localitzar utilitzant la barra de navegació que apareix en l'editor dret. Hauria d'aparèixer dins de l'opció *Automatic*.

Afegirem un *outlet* per al *text field*. Per crear-lo, haurem de fer un moviment d'arrossegament des del text *field* fins al punt del codi on vulguem crear-lo, mentre mantenim premuda la tecla *control*, quan la deixem de prémer ens apareixerà la pantalla següent. Ens assegurarem que creem una connexió *outlet* amb el nom «textToGuess» de tipus *UITextField*.



Fem la mateixa operació amb el *label* per a les pistes i l'anomenem «hintLabel» de tipus *UILabel*.

Ara ens cal poder respondre al fet de prémer el botó. Com hem fet abans, arrosseguem des del botó cap al codi mantenint la tecla *control* premuda, però ara ens assegurem que el tipus de connexió és *Action* i l'anomenem «guessTapped»:



Aquest és el codi que s'haurà generat en el nostre 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 funció *viewDidLoad* s'executa quan la vista s'ha carregat des de l'*storyboard*, i és el punt en què ens prepararem per iniciar un nou joc: configurarem el model i donarem els valors inicials als *labels* en la vista.

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

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

La funció *guessTapped* està enllaçada al *button* en la vista i s'executarà cada vegada que l'usuari el premi. En aquesta funció el nostre controlador realitzarà la major part del seu treball:

```
    @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) Obtenim el nombre que ha escrit l'usuari al *text field*. Es pot donar el cas que l'usuari introdueixi un valor que no es pot convertir en *Int*, per això tractem aquesta circumstància mitjançant un *guard*.

Hem de tenir en compte que, encara que hàgim configurat un teclat numèric, l'usuari pot introduir text per altres mitjans; per exemple, enganxant text del porta-retalls.

2) Amb el valor ja convertit en *Int*, cridem la funció *guess* del model per veure si ha encertat.

3) Si ha encertat, mostrarem un diàleg a l'usuari amb una felicitació. Per a això, construïm una *string* utilitzant informació que extraiem del model. Quan l'usuari tanqui aquest diàleg, reiniciarem el joc cridant la mateixa funció *startNewGame*, que com hem vist abans prepara el model per a un nou joc i també reinicia els valors de les vistes.

4) Finalment, si no ha encertat, demanarem al model que ens proporcioni una pista i la mostrarem en el *label* corresponent a la vista.

I ja podem executar la nostra aplicació prement el botó *play*, o amb la drecera cmd + R.

5. *View controllers*

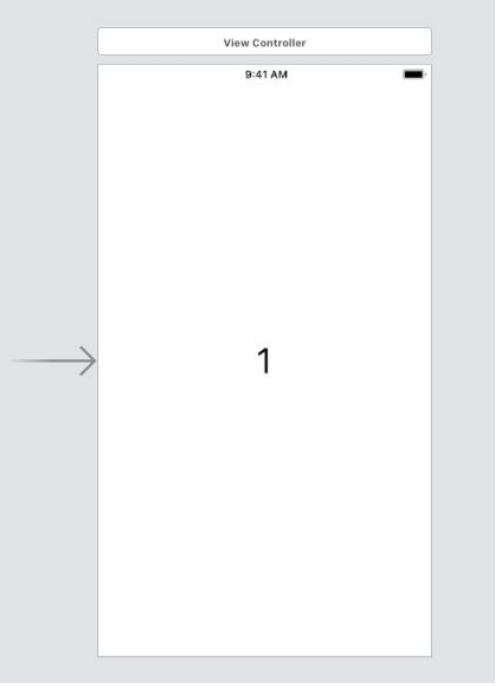
Un *view controller* gestiona una jerarquia de vistes que representen una de les pantalles en què s'organitza una aplicació. La classe de UIKit que ens proporciona la funcionalitat és *UIViewController*. Totes les classes de controladors en les nostres aplicacions s'hereten d'aquesta.

Els objectius principals d'un *view controller* són:

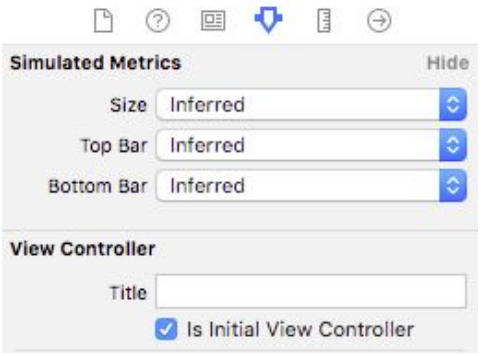
- **Actualitzar les vistes per reflectir l'estat de les dades.** Per exemple, actualitzar les imatges i textos que es mostren a la pantalla quan es rep informació actualitzada.
- **Respondre a les interaccions de l'usuari.** Per exemple, navegar a la pantalla següent quan l'usuari prem un botó.

Configurar el *view controller* d'inici

Quan s'inicia l'aplicació s'arrenca amb el *view controller* d'inici. En l'*storyboard* aquest *view controller* es distingeix de la resta perquè apareix amb una fletxa apuntant cap a ell. Podem arrossegar aquesta fletxa cap a un altre *view controller*, que passarà a ser el nou *view controller* d'inici.



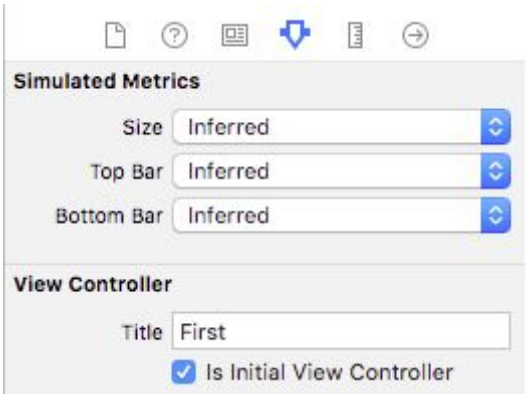
També, amb el *view controller* seleccionat, podem canviar el *view controller* d'inici marcant la casella corresponent en l'*Attribute inspector*:



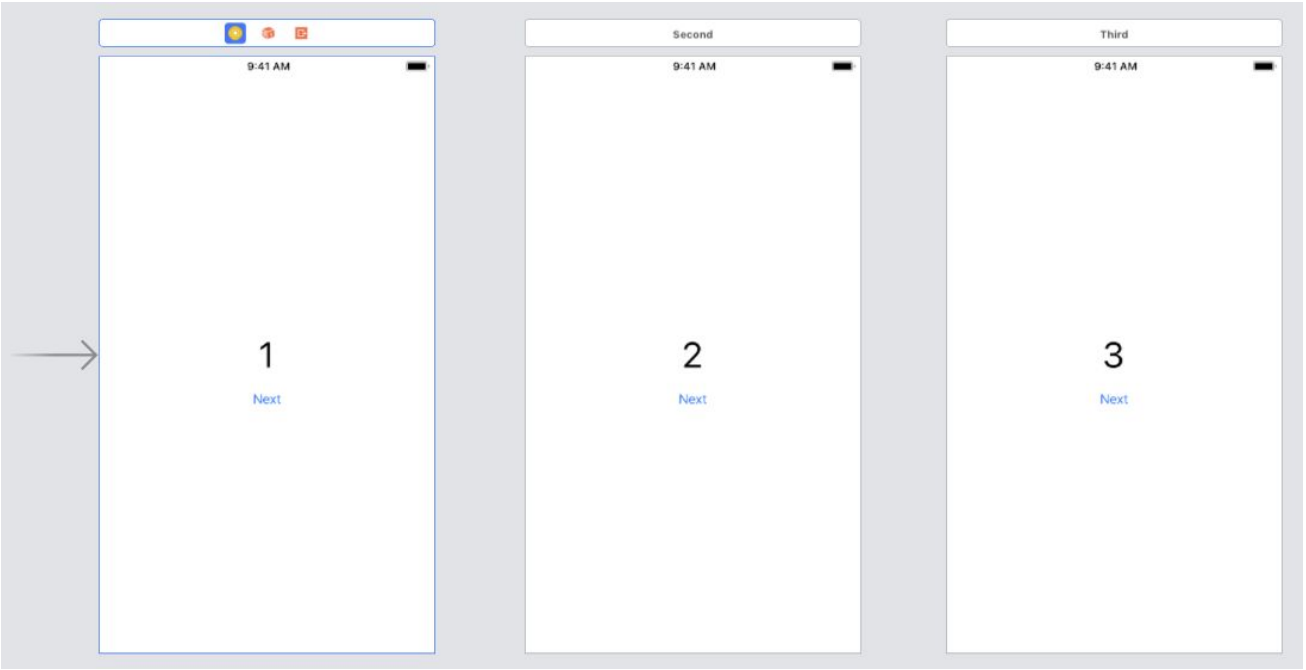
Navegació

Per navegar entre *view controllers*, crearem *segues* des d'un *view controller* fins al que volem que es mostri a continuació.

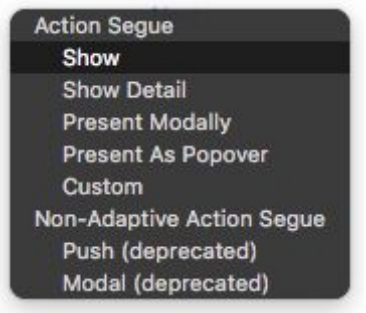
En un projecte obrim l'*storyboard* principal i creem tres *view controllers*, cadascun amb un botó centrat en la vista. A cada un li donarem un títol diferent, «First», «Second» i «Third». Per canviar el títol seleccionem cada *view controller* i l'editem des de l'*Attribute inspector*:



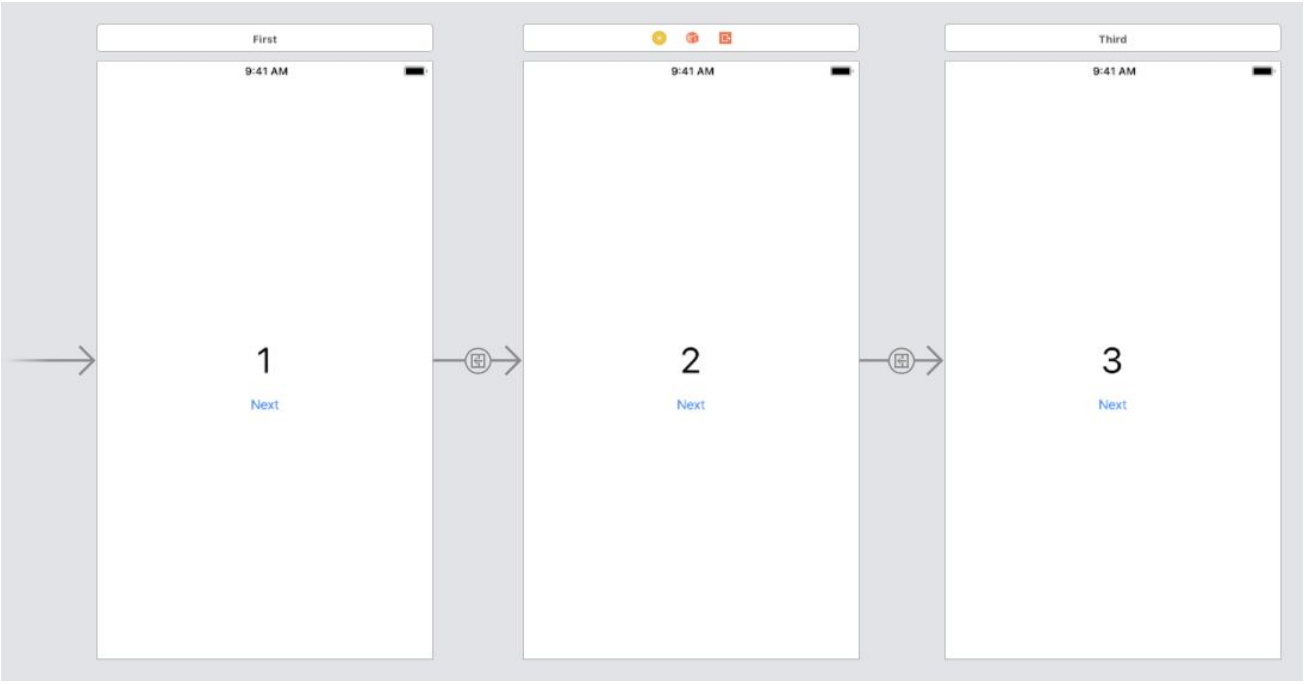
Així quedarà l'*storyboard*:



Crearem un *segue* de manera semblant a com enllacem els *outlets*. Amb la tecla *control* premuda arrosseguem amb el botó esquerre des del botó fins al *view controller* a què volem navegar. En deixar anar el botó esquerre sobre el *view controller* de destinació, ens apareixerà el diàleg següent:

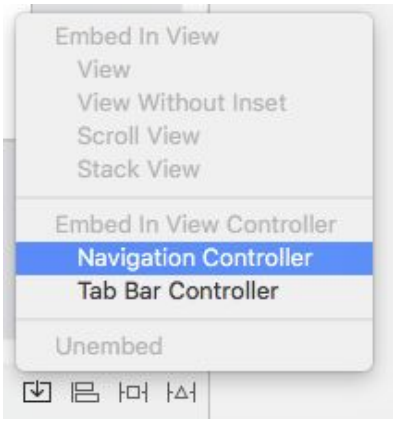


Conté els diferents tipus de *segue* disponibles; en aquest cas seleccionem el primer. Enllacem el botó del primer *view controller* per navegar cap al segon, i el botó del segon per navegar cap al tercer:

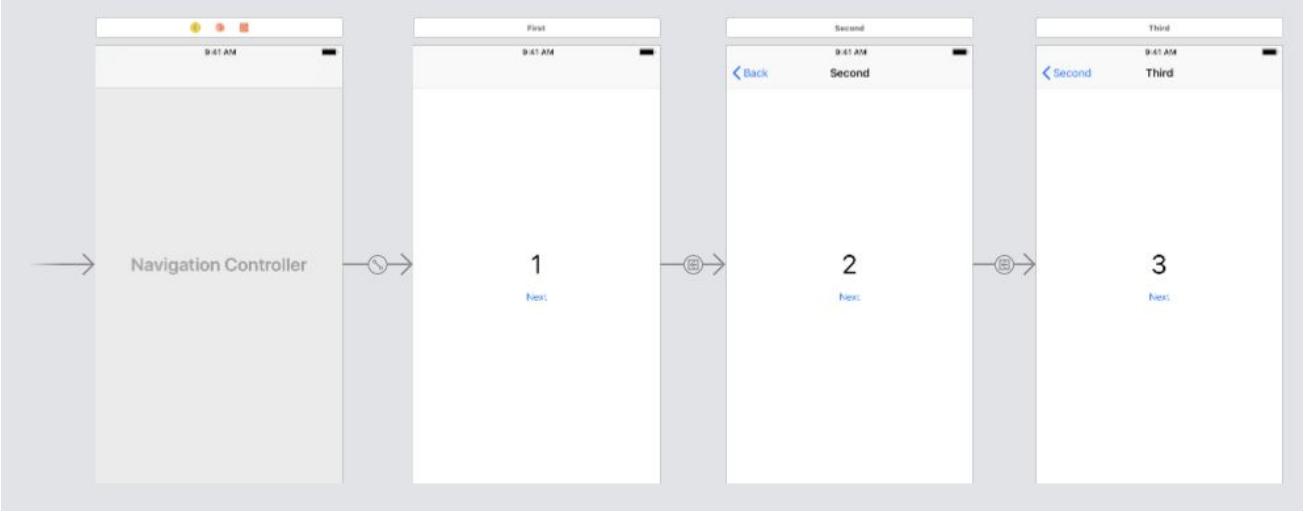


Si executem ara l'aplicació, veurem que, efectivament, funciona la navegació cap al següent *view controller*, però no hi ha manera de tornar enrere. Per aconseguir una navegació completa, hem d'afegir un *Navigation Controller* que mostra automàticament una zona de navegació en la part superior que mostra el títol del *view controller* que s'està mostrant en cada moment i un botó per navegar a l'anterior.

Per afegir el *navigation controller*, seleccionem el primer dels *view controllers* i, amb el botó d'inclusió, en la part inferior dreta de l'editor, seleccionem l'opció *Navigation Controller*.



Ara sí que si executem l'aplicació, podrem navegar cap endavant i cap enrere sense problemes.


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

6. Auto Layout

Per què Auto Layout?

Hi ha diverses mides de pantalla per als dispositius iOS. Això vol dir que, perquè la nostra aplicació es mostri correctament, hem de determinar la posició i mida de les vistes que usem perquè es mostri correctament en totes elles.

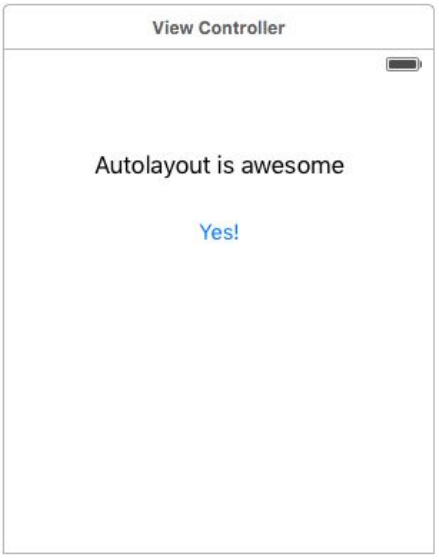
Imaginem una vista senzilla, per exemple, una imatge que volem que ocupi tota la pantalla, amb uns marges de 20 píxels per cada costat. La pantalla d'un iPhone 5 SE és de 1136 × 640 píxels, i, en canvi, la pantalla d'un iPhone 8 Plus és de 1920 × 1080, però volem que la nostra aplicació es mostri correctament en ambdós dispositius. Per a l'iPhone 5 SE podríem definir una amplada de la imatge de 600 píxels (640 píxels - 20 × 2 dels marges laterals). Però ens trobarem amb què per al cas de l'iPhone 8 Plus aquest ample de 600 píxels no ocupa tot l'ample de pantalla: hauria de ser de 1040 píxels perquè es mostrés com desitgem.

Per solucionar-ho, podríem detectar el tipus de dispositiu en cada pantalla de la nostra aplicació i, a partir d'aquest, calcular les posicions i dimensions apropiades per a totes les nostres vistes. O millor, podríem detectar la mida de la pantalla i realitzar, a partir d'aquí, els nostres càlculs.

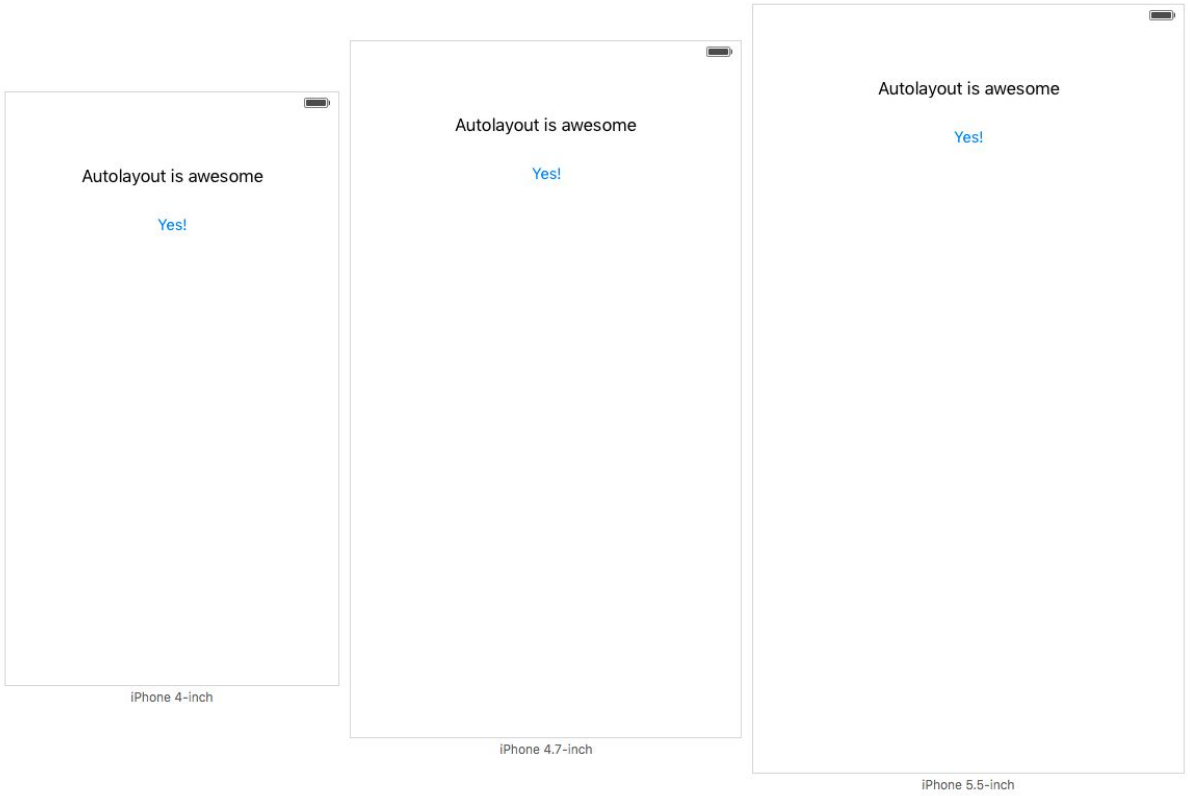
Tot i que la mida dels dispositius iOS està estandarditzada, la veritat és que cada vegada hem de donar suport a més mides de pantalla diferents. I què passa si tenim la nostra aplicació ja desenvolupada i apareix un nou dispositiu amb una mida de pantalla diferent? Haurem de revisar el codi que tinguem per donar-li suport. Necessitem un mecanisme per definir com se situen les nostres vistes en pantalla i que s'adapti a qualsevol mida de dispositiu. I justament això és el que ens proporciona Auto Layout.

Amb Auto Layout definim una sèrie de constraints per a les nostres vistes i la seva posició i mida es determina automàticament.

Així, per exemple, podem definir que volem un *label* centrat horitzontalment amb un botó situat a sota amb un marge vertical de 20 punts fins al *label*.



Veiem com s'apliquen les constraints per obtenir la maquetació que desitgem per a cada mida de pantalla:

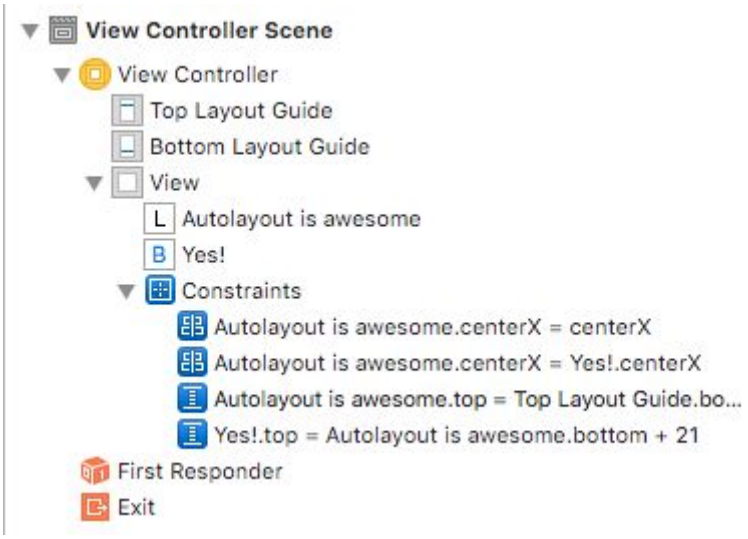


Definició de constraints en Interface Builder

Des d'Inferface Builder tenim tres mecanismes per definir les nostres constraints:

- Utilitzant el mecanisme de *control-drag*.
- Utilitzant les eines de definició de constraints de l'editor.
- Permetent que Interface Builder ens suggereixi unes constraints.

Independentment del mecanisme utilitzat, les constraints s'afegiran a l'escena i les podrem editar posteriorment. D'aquesta manera, podem utilitzar la tècnica que més ens convingui en cada moment, i el resultat serà el mateix.



Constraints per *control-drag*

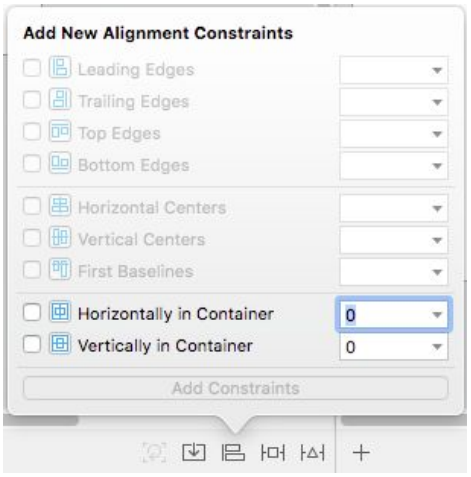
Com podem deduir pel seu nom, consisteix a definir constraints entre dues vistes arrossegant d'una cap a l'altra mentre mantenim premuda la tecla *control*. En deixar de prémer-la, se'ns mostra un menú contextual amb les diferents constraints possibles.

Constraints utilitzant les eines de l'editor

Les eines de constraints de l'editor ens permeten definir-les de manera més precisa.

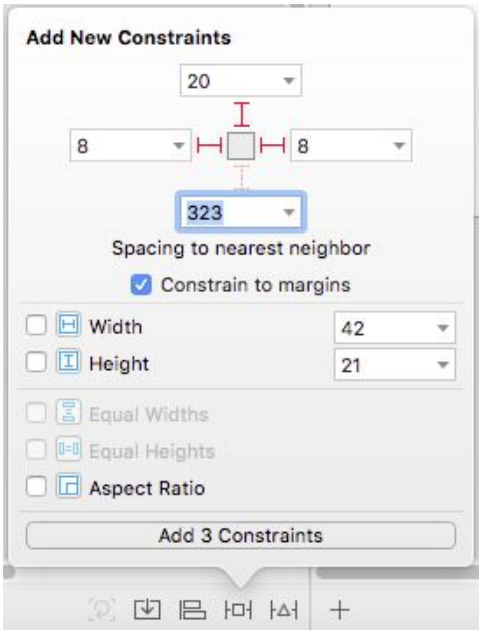
Align tool

Ens permet afegir les constraints necessàries per alinear les vistes de manera molt senzilla. Primer, seleccionem en l'editor els elements que volem editar i, després, premem l'eina d'alineació. Ens apareixerà el diàleg següent per afegir els diferents tipus d'alineació:



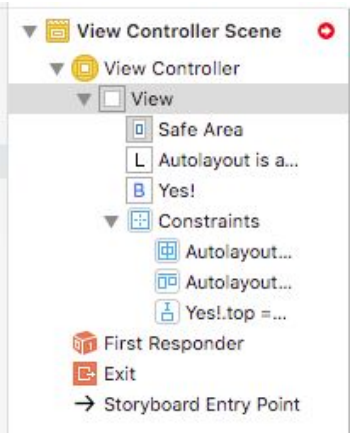
Pin tool

Aquesta eina ens permet afegir ràpidament constraints des de la vista que tenim seleccionada a la resta de vistes que l'envolten.

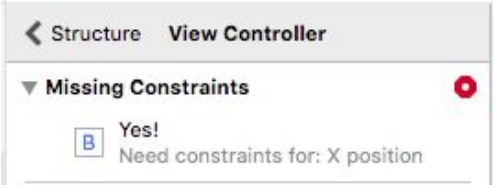


Resolució de problemes amb Auto Layout

Hi ha un nombre mínim de constraints perquè Auto Layout pugui trobar la solució a la posició i mida de la vista. Si falten constraints, ens ho indicarà en el *document outline* de l'editor amb una marca en vermell:

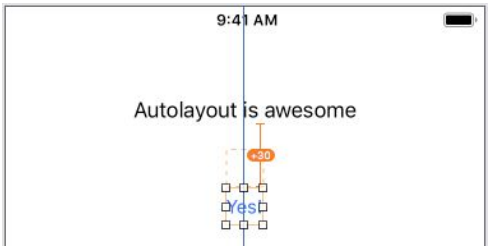


Fent clic sobre aquesta marca ens informarà de tots els errors que ha trobat en les constraints que tenim definides:

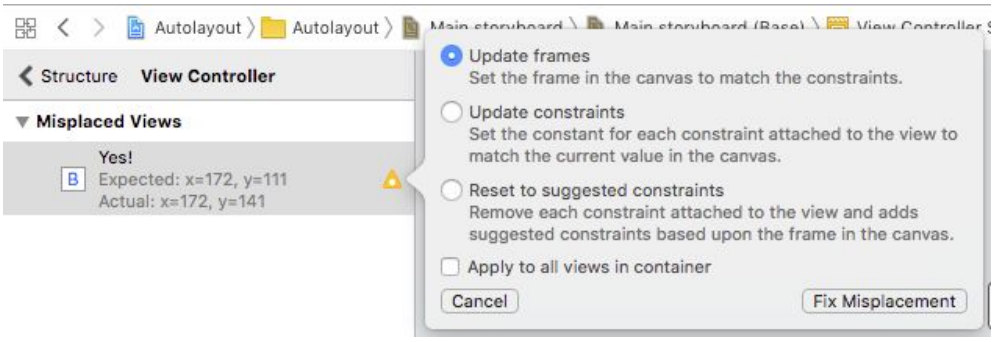


En aquest cas, el que passa és que per al *button* hem definit una constraint per a la seva posició vertical però no sap com determinar la posició horitzontal. Afegint una constraint per centrar-lo en el seu contenidor, se soluciona el problema.

Si ja tenim les constraints definides però després movem les vistes, l'editor ens avisarà que hi ha una incongruència entre el que està mostrant i el que estem demanant amb les constraints. Així doncs, mostrarà en l'editor les constraints que no encaixen. Per exemple:



En el document *outline* ens n'informarà també, i ens suggerirà realitzar canvis de manera automàtica per resoldre el problema. Podem escollir la que més ens convingui i prémer *Fix Misplacement*.



Update frames tornarà les vistes afectades a la seva posició anterior per mantenir les constraints que tinguessin definides. Això és útil si hem mogut la vista per error o estem fent proves.

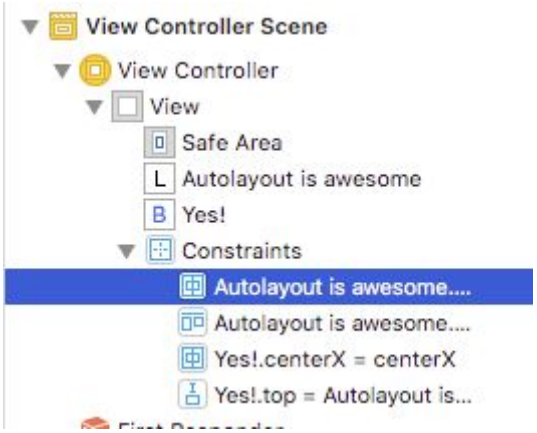
Update contraints actualitzarà les constraints que tenim definides perquè encaixin amb la nova posició de la vista, però no crearà constraints noves. Aquesta opció és la que se sol utilitzar quan ja tenim les vistes i les constraints definides i després fem petits ajustos.

Reset to suggested constraints no és una opció recomanable ja que eliminarà totes les constraints que hem definit i en crearà de noves de manera automàtica. Només en els casos més senzills produeix bons resultats.

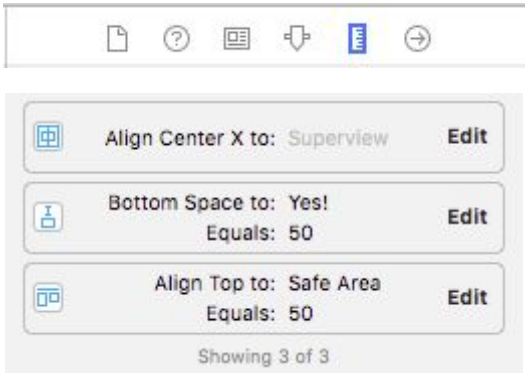
Edició de constraints

Una vegada creades, les *constraints* apareixeran en el mateix editor. Apareixen en forma de línies blaves en seleccionar cada vista:

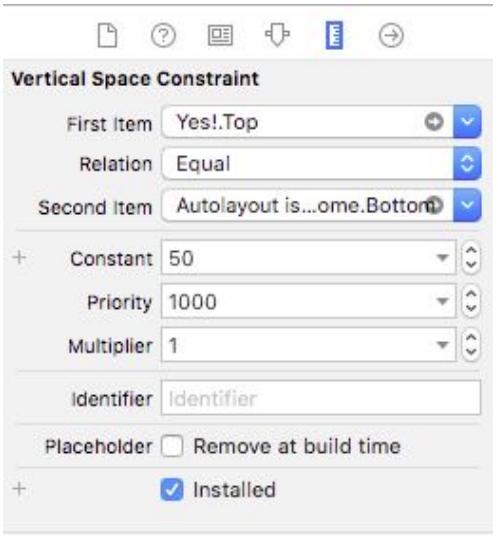
Quan tenim moltes *constraints*, és difícil seguir-les des de l'editor, llavors podem seleccionar-les des del *document outline*:



I també quan seleccionem una vista, en el *size inspector* ens apareixeran les *constraints* definides per a ella:

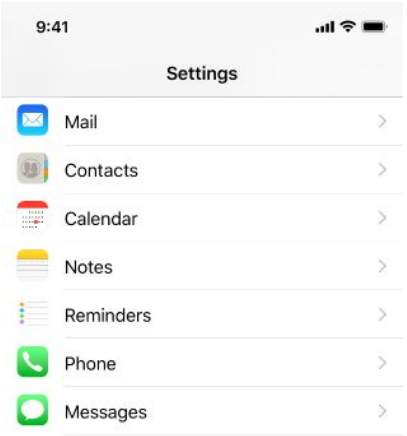


I ja amb la constraint seleccionada en l'inspector podem configurar-la amb més detall, canviant les vistes entre les quals es defineix o, per exemple, la constant, que determina la distància a la qual s'han de mantenir les vistes.



7. *UITableView* i *UITableViewController*

UITableView és el component d'UIKit que ens permet presentar una llista d'elements. És el component més utilitzat i és fàcil veure'l en acció en moltes aplicacions. De fet, moltes aplicacions estan construïdes entorn d'un *UITableView*.



Podem fer servir un *UITableView* directament, però *UITableViewController* ja ens proporciona un *view controller* que porta incorporat un *UITableView* enllaçat i llest per ser utilitzat.

UITableViewController ja adopta els protocols necessaris per interactuar amb l'*UITableView*. Entre ells tenim l'*UITableViewDataSource*, que ens permet controlar la informació que es mostrarà en l'*UITableView*, i *UITableViewDelegate*, amb el qual gestionarem la interacció de l'usuari en l'*UITableView*.

Model de dades

Per veure com utilitzar *UITableViewController* construirem una aplicació d'exemple que ens mostrarà una llista de països. En el nostre *view controller* tindrem la variable següent que actuarà com a model de dades:

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

Veiem ara que heretem d'*UITableViewController*, i no d'*UIViewController*, com en els exemples anteriors. Inicialitzem el model de dades en *viewDidLoad*:

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

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

Protocol *UITableViewDataSource*

Mitjançant aquest protocol proporcionarem informació al *table view* sobre les dades que volem que mostri. Com a mínim li direm el nombre de files que tindrà, i també hem d'indicar-li el contingut de cadascuna d'aquestes files.

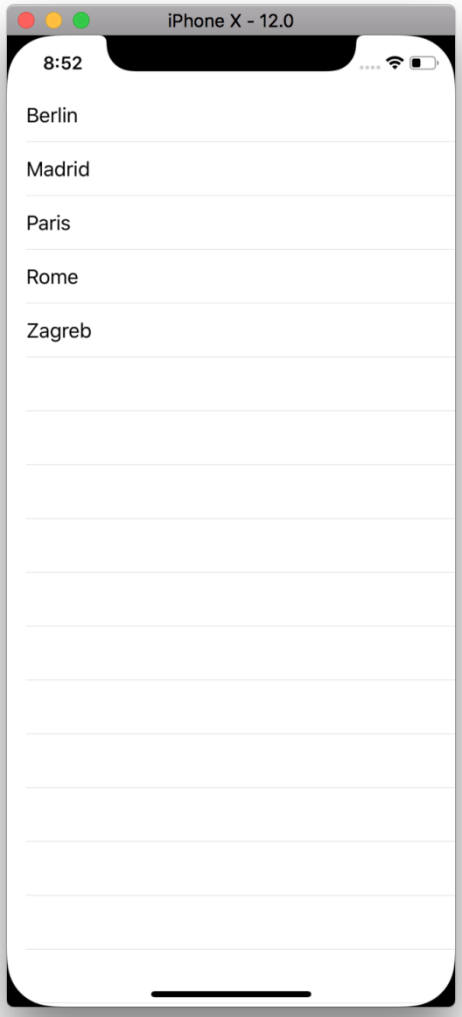
Amb *numberOfRowsInSection* li diem el nombre de files que tindrà la taula i serà, és clar, el nombre d'elements del nostre *array* del model de dades:

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

I en la funció *cellForRowAt* hem de tornar la cel·la que volem mostrar per a cadascuna de les files de la taula. En el nostre cas farem que la cel·la contingui el nom de cadascuna de les ciutats que tenim al nostre *array* del model de dades.

```
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 executem l'aplicació, ja ens apareix la nostra llista de ciutats:



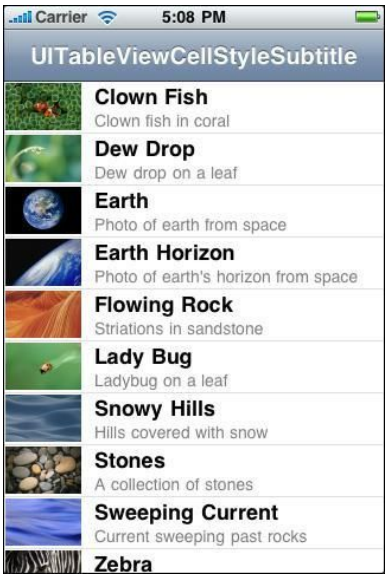
UITableViewCell

Aquesta classe representa cadascuna de les cel·les de la nostra taula. Mitjançant les seves propietats podem personalitzar el contingut i l'aspecte de les cel·les.

textLabel

detailTextLabel

imageView



Reciclatge de cel·les

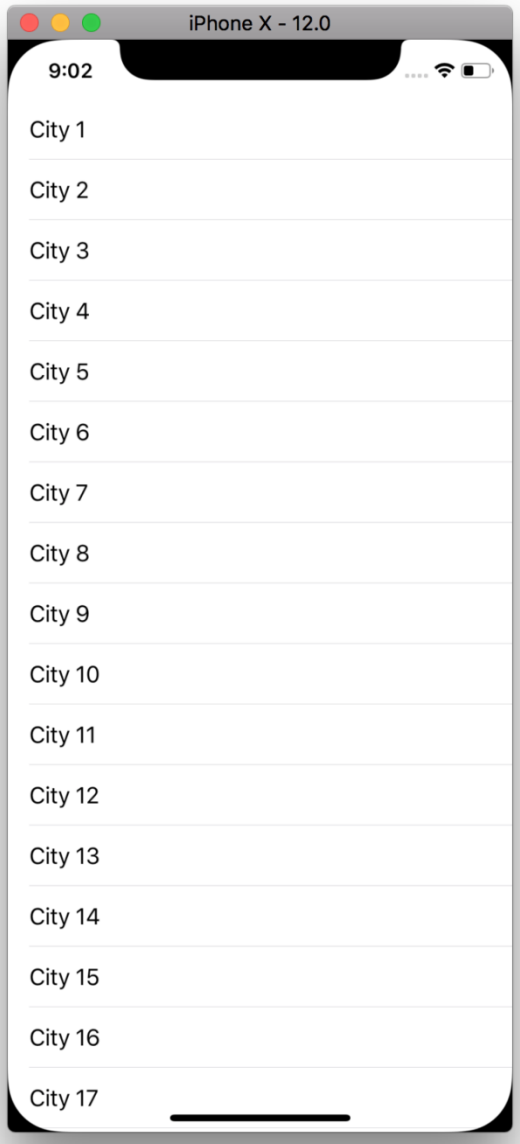
Aquí és crucial obtenir la cel·la sempre amb la funció de *dequeueReusableCell*. El secret perquè el *tableview* funcioni amb un gran rendiment i eficiència de recursos en iOS és que les cel·les es reciclin. Tot i que tinguem milers de dades per construir la pantalla, només s'utilitza un nombre reduït de cel·les, que a més no s'han de crear constantment, sinó que es reutilitzen. Quan una cel·la deixa de ser visible al *tableview*, es recicla per ser utilitzada per mostrar la dada següent. Tot això es gestiona internament en el *tableview* sense haver de fer nosaltres res addicional.

I és aquesta funció *dequeueReusableCell* la que s'encarrega de retornar-nos una cel·la recentment creada si és la primera vegada que la demanem o una cel·la reciclada. La gestió de l'estat de les cel·les del *tableview* es realitza internament.

Farem un canvi en el model de dades per veure més en detall el reciclatge de les cel·les:

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

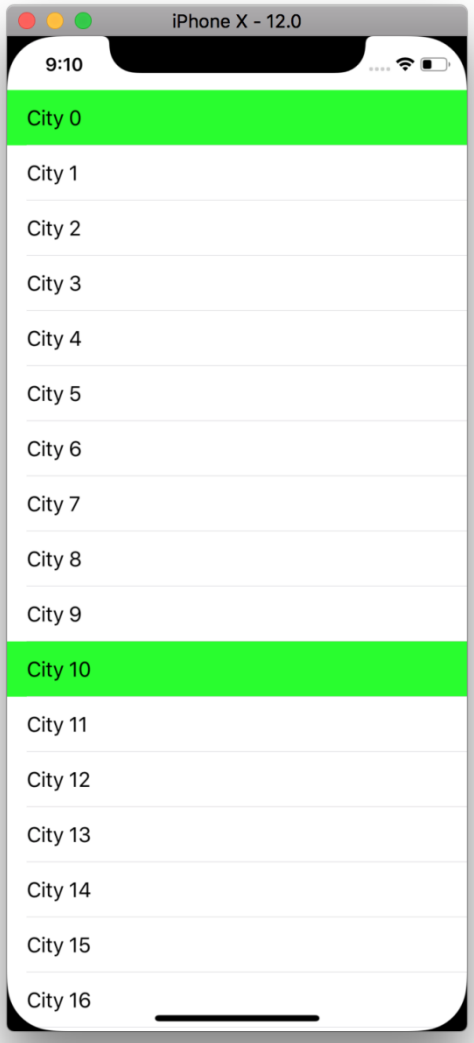
Ara el nostre *view controller* gestiona mil cel·les. Si ens movem cap avall, veurem que el moviment és molt fluid.



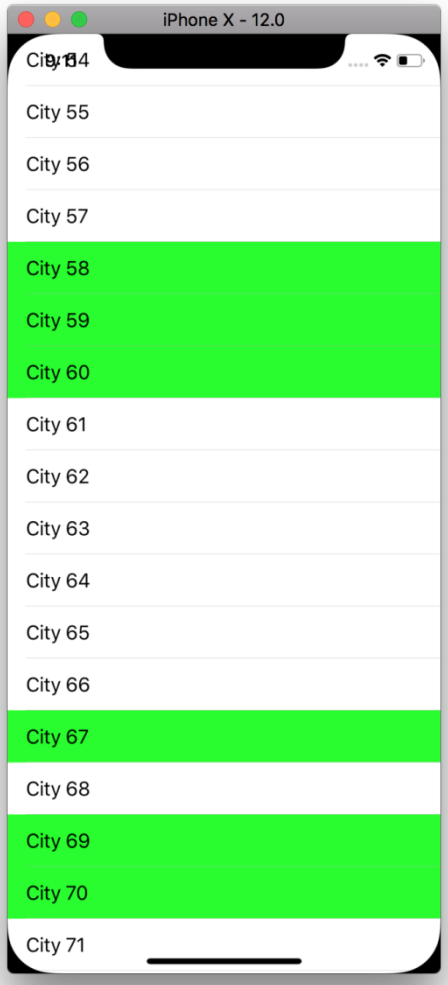
Imaginem que volem marcar una de cada deu cel·les que mostrem, per exemple, canviant el color de fons. Afegim aquest codi per canviar el color de fons de la cel·la abans de retornar-la a *cellForRowAt:*

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

Executem l'aplicació una altra vegada:



I sembla que funciona correctament, cada deu cel·les ens en marca una en verd. Però a mesura que anem baixant, ens donarem compte que apareixen amb fons de color verd altres cel·les que no ens esperem. Què passa aquí?



Les cel·les que tenen el color de fons que no ens esperem, com la 58 o la 59, és perquè en un moment anterior van ser cel·les que van complir la condició i els hem canviat el color de fons. Quan les cel·les es reciclen, no es restableixen les seves propietats. Som nosaltres, amb la funció *cellForRowAt*, els que tenim la responsabilitat d'assegurar-nos que cada cel·la té les propietats correctes per representar la dada de la cel·la que ens demanen per a cada *indexPath.row*.

Així doncs, aquest seria el codi correcte en aquest cas:

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

Ens hem d'assegurar que sempre donem un valor a les propietats de la cel·la, perquè no sabem quina posició havia ocupat anteriorment la cel·la que ens torna *dequeueReusableCell* i, per tant, quins valors tindrà a les seves propietats.

Rendiment i errors comuns

Hem de tenir molt clar que, tot i que implementem les funcions d'*UITableViewDataSource*, no les cridarem mai directament. UIKit s'encarrega de mostrar en pantalla l'*UITableView* i crida les nostres funcions sempre que ho consideri necessari.

A més, no tenim cap control sobre el nombre de vegades que es cridarà *numberOfRowsInSection* i *cellForRowAt*, o l'ordre en què es cridaran. I l'ordre i nombre de crides pot canviar entre versions d'iOS, per la qual cosa, no hem de donar res per fet.

Les funcions d'*UITableViewDataSource* es criden com a part del procés de mostrar la informació a la pantalla, que és un dels que tenen un rendiment més crític. Per tant, el nostre codi ha de ser el més ràpid possible. I, per descomptat, dins d'elles hem d'evitar cridar funcions que recuperin dades directament del sistema de fitxers o, pitjor encara, que facin peticions a un servidor remot. En general, farem servir només les variables locals que actuen com a model de dades del nostre *view controller*.

Des d'aquestes funcions tampoc hem de consultar l'estat d'altres controls. I, per descomptat, aquestes funcions no han de tenir efectes secundaris com modificar variables del *view controller* o canviar l'estat d'altres controls.

8. Animacions

Per millorar l'experiència d'usuari en les nostres aplicacions podem utilitzar les animacions que ens proporciona el mateix *framework* UIKit. Podrem utilitzar animacions per comunicar-nos millor amb l'usuari, expressar millor les accions que realitza l'usuari en la nostra aplicació i també per llançar una aplicació més atractiva. Així i tot, hem d'anar amb compte de no abusar d'animacions gratuïtes si no aporten valor a l'usuari.

Bàsicament les animacions consisteixen a fer canvis al llarg d'un període de temps en les propietats de les vistes que formen la interfície d'usuari de la nostra aplicació. Es tracta de les mateixes propietats que ja hem vist i utilitzat en els mòduls anteriors.

Per seguir els exemples d'aquest apartat, partirem del projecte Xcode Animations, disponible al repositori de codi de l'assignatura. En executar el projecte podem veure que ja tenim una sèrie de pantalles navegables amb l'ajuda d'un *navigation controller*.

Animacions bàsiques

A la primera pantalla de l'aplicació podem veure que tenim dos botons i un *label*. Aconseguirem que el *label* aparegui i desaparegui en prémer cadascun dels botons.

Si obrim BasicAnimationViewController.swift veurem que ja tenim les accions dels botons enllaçades a les funcions *fadeOut* i *fadeIn*.

En la funció *fadeOut* afegirem el codi següent:

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

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

Executem l'aplicació i veiem que en prémer el botó *Fade out* desapareix el *label*.

La funció *Animate* d'*UIView* és la que s'encarrega d'executar les animacions. El paràmetre *withDuration* determina la durada en segons. I les animacions les definim mitjançant la *closure* que passem en el paràmetre *animations*.

Tal com vam veure en l'apartat dedicat a Swift, la sintaxi per definir una *closure* és semblant a la d'una funció, i en aquest cas necessitem una *closure* sense cap paràmetre () i que tampoc retorni cap valor *Void*.

En el cos de la *closure* assignarem els valors que volem que tinguin les propietats de la nostra vista a la fi de l'animació. En aquest cas fem que l'opacitat del *label* sigui 0, de manera que el fem desaparèixer.

En l'acció del botó *fade in* afegim el codi perquè l'opacitat del *label* torni a ser 1.

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

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

I així aconseguim l'efecte desitjat: un botó fa desaparèixer el *label* i l'altre el fa aparèixer.

Completions en animacions

Les animacions que s'executen amb *UIView.animate* tenen el seu propi fil d'execució. Qualsevol codi que tinguem després s'executarà immediatament i en paral·lel a l'animació. Provem, per exemple, a afegir el codi següent després de la crida d'*UIView.animate*:

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

Podríem pensar que després de l'animació el text del *label* canviarà de «Hello World!» a «Goodbye!». No obstant això, en executar l'aplicació ens adonarem que immediatament després de prémer el botó *Fade Out* el text del *label* canvia a «Goodbye!» i s'executa l'animació. Això és degut al fet que el codi de l'animació i el nostre codi que canvia la propietat *text* del *label* s'executen al mateix temps.

Per a aquests casos, *UIView* ens proporciona una versió de la funció *animate* amb un paràmetre addicional, *completion*.

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

En aquest paràmetre proporcionarem el codi que volem que s'executi una vegada hagi acabat l'animació.

Obrim el fitxer CompletionAnimationViewController.swift corresponent a la segona pantalla de l'exemple. En la funció *fadeOut* afegim el codi següent:

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

```
    self.messageLabel.text = "The label has disapeared"
})
```

A la *closure* del paràmetre *completion* incloem el codi que volem que s'executi en finalitzar l'animació. En aquest cas, canviem el text d'un altre dels *labels* de la pantalla.

Si executem l'aplicació comprovarem que el text del *label* no s'actualitza fins que l'animació ha acabat. A la *closure completion* podem incloure el codi que desitgem. Per exemple, podem aconseguir que després d'executar l'animació se'ns mostri la pantalla següent. Substituïm el codi de la *closure* pel següent:

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

Animacions combinades

De vegades ens interessarà animar les propietats de diverses vistes simultàniament, i per aconseguir-ho no hem de fer res més que incloure tots els canvis que desitgem realitzar a la *closure animations*.

Vegem un exemple en la tercera pantalla de l'aplicació. Obrim el fitxer CombinedAnimationsViewController.swift i farem que apareguin els tres *labels* que tenim en aquesta pantalla. En l'*storyboard* hem configurat els tres *labels* de manera que si no fem res més seran visibles quan es mostri la pantalla. Per tant, primer els ocultem una vegada s'hagin carregat les dades de la pantalla des de l'*storyboard*, establint la seva propietat *alpha* a *viewDidLoad*:

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

Si executem l'aplicació, veurem que no apareix cap *label* en la segona pantalla. Després, en l'acció del botó *animate* creem l'animació que canvia la propietat *alpha* de cadascun dels *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)
```

I podem comprovar si executem l'aplicació que tots els *labels* apareixen alhora.

Però i si el que volem és coordinar les animacions de manera que els *labels* apareguin un després de l'altre? Ho aconseguim de la mateixa manera que hem vist en el punt anterior, incloent en la *closure* del paràmetre *completion* el codi que volem que s'executi una vegada ha acabat l'animació. A la *closure completion* de la primera animació inclourem el codi per executar l'animació següent, i així successivament.

```
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)
    })
})
```

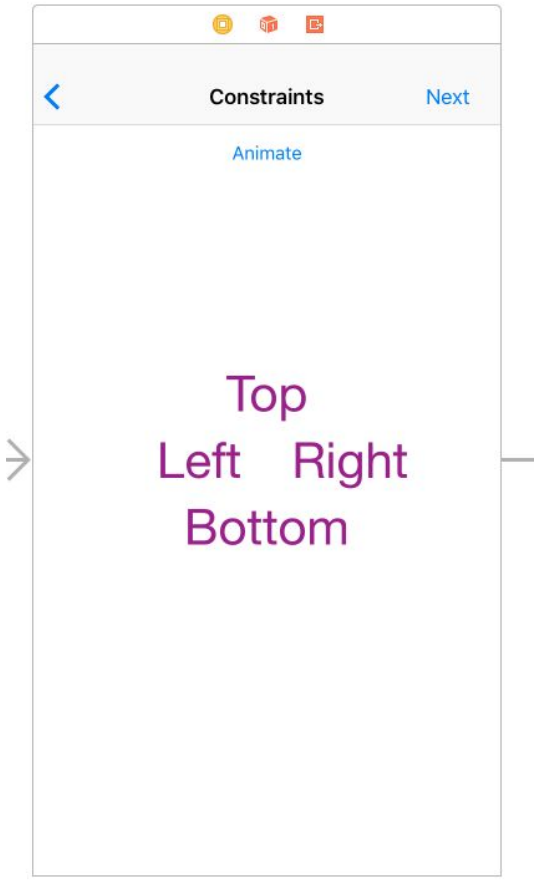
Observem que en aquest cas hem fet servir el caràcter _ en la definició de la *closure*. Així indiquem que, tot i que la *closure completion* tingui paràmetres en aquest cas, no els necessitarem i els obviem per simplificar-ho.

Animar la posició i la mida

En l'apartat d'Auto Layout vam veure com definíem mitjançant *constraints* les regles que volíem que es complissin per aconseguir la maquetació a les nostres pantalles, i després el sistema d'Auto Layout s'encarregava de realitzar els càlculs necessaris.

Quan es realitzen animacions de la posició i la mida de les vistes de les nostres pantalles, seguim confiant que Auto Layout calculi la disposició de tots els elements en pantalla. Per tant, no canviarem directament la posició i les mesures de les vistes, sinó que modificarem les *constraints* que s'utilitzen per calcular-les.

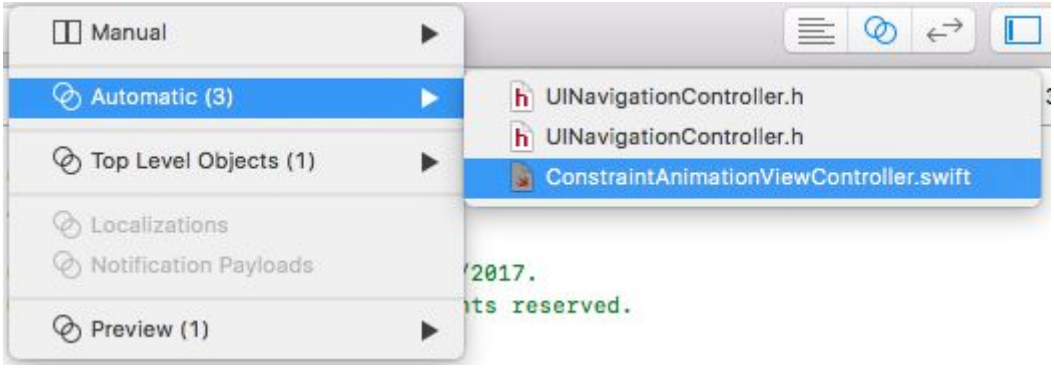
Partim de la pantalla amb totes les *constraints* configurades i editem el fitxer ConstraintAnimationViewController.swift.



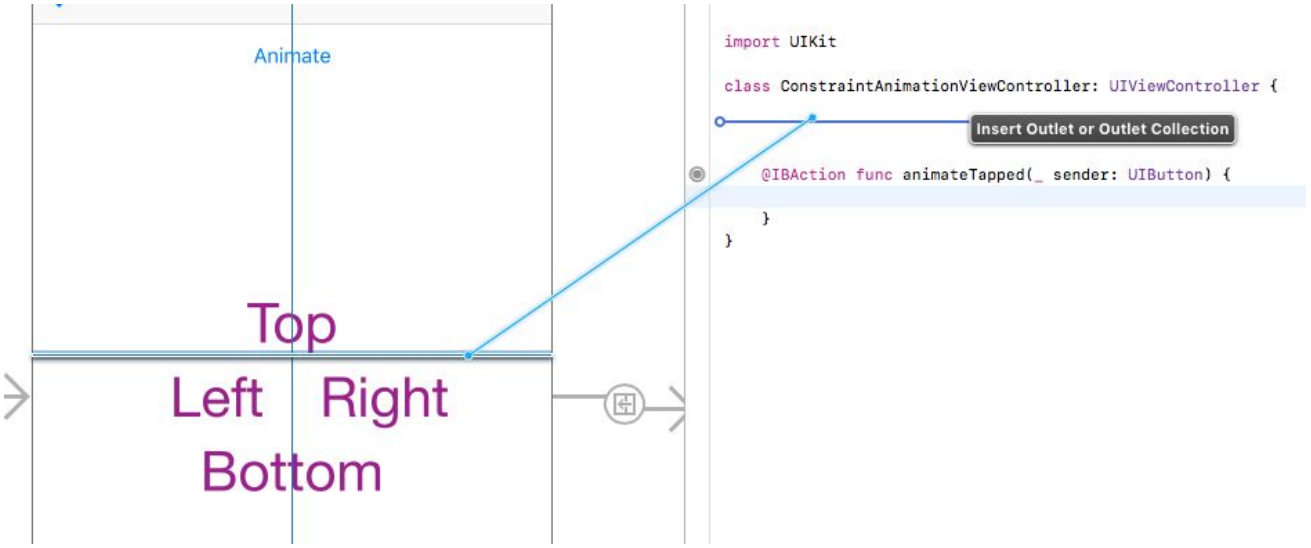
Per poder modificar una *constraint*, primer necessitem enllaçar al *view controller* mitjançant un *outlet*. Activem l'*assistant director*:

A l'editor esquerre mantenim l'*storyboard* amb la quarta pantalla seleccionada i en l'editor dret ens assegurem que es mostra el codi del seu *view controller*, *ConstraintAnimationViewController*.

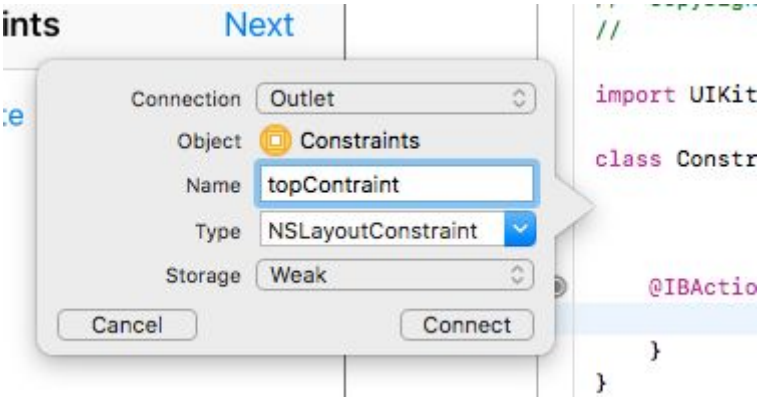
Si no el tenim a la barra de l'editor, el seleccionem; apareixerà dins el menú *Automatic*:



Per afegir l'*outlet* localitzem la constraint del *label Top* que determina la seva posició vertical.



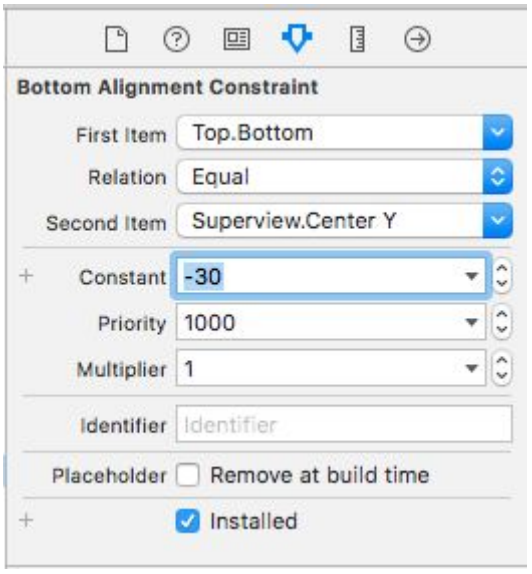
I a la pantalla que ens apareix indiquem el nom *topConstraint*.



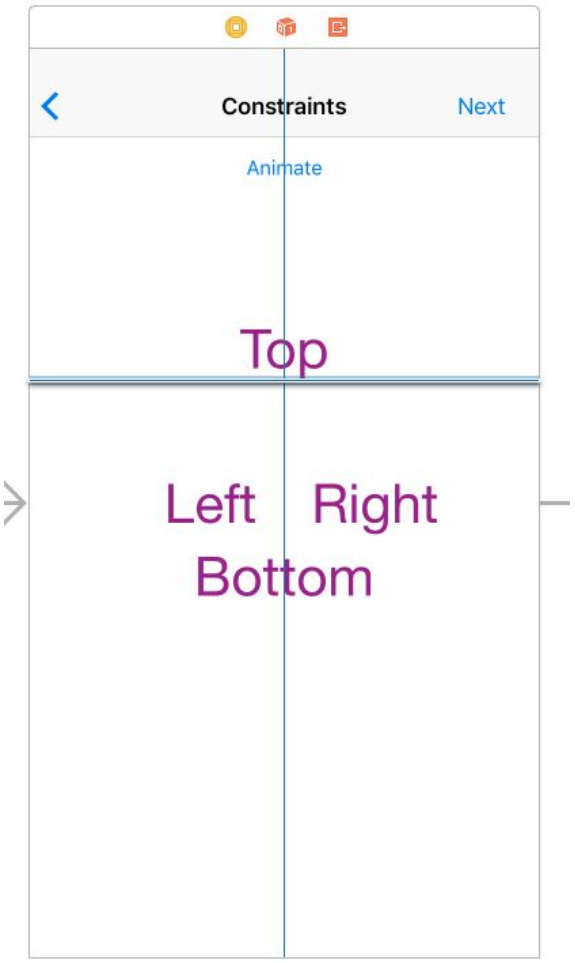
I aquest serà l'*outlet* que s'afegirà:

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

Seleccionem la constraint a l'*storyboard* i veiem les seves propietats a l'*Attributes inspector*:



Veiem que aquesta constraint centra verticalment el *label* en la vista que el conté, amb un valor constant de desplaçament de –30 (en aquest cas el valor negatiu significa cap amunt). Si escrivim un altre valor a la propietat *Constant*; per exemple, –90, veurem que el *label* es desplaça cap amunt. I tornem a deixar el valor –30 que teníem inicialment.



L'animació d'aquest *label* consistirà a fer-lo desaparèixer per la part superior de la pantalla. Per a això, només ens caldrà saber el valor que hem de donar a la constant de la constraint.

És una bona pràctica definir la posició dels elements en pantalla en relació amb la resta d'elements, o sigui que en comptes d'usar un valor fix per a aquesta constant de la constraint utilitzarem el valor de l'altura de la vista que conté el *label*. Si utilitzéssim un valor fix, seria més difícil que l'animació funcionés bé per a totes les mides de pantalla existents (o fins i tot si tenim mides de pantalla diferents en el futur). Aquest valor el podem obtenir de la propietat *frame* de la vista:

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

L'animació de les constraints funciona d'una manera una mica diferent a l'animació de les propietats de les vistes que hem vist abans. Primer, feim el canvi en la constraint, i en la *closure* del paràmetre *animations* indiquem a Auto Layout que recalculi la posició de totes les vistes 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()
    })
}
```

És aquesta crida a la funció *layoutIfNeeded* de la vista principal de la pantalla la que fa que Auto Layout recalculi la posició de tota la seva jerarquia de vistes. També cal cridar *layoutIfNeeded* abans de començar l'animació. Si executem ara l'animació veurem com el *label Top* desapareix per la part superior de la pantalla.

Després, podem afegir els *outlets* per a la resta de *constraints* per completar aquesta última pantalla. Animarem les constants perquè el *label left* desaparegui per l'esquerra, el *label right* per la dreta i el *label bottom* per la part 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()
    })
}
```

Novetats en iOS 12 i Xcode 10

El dia 17 de setembre de 2018, Apple va posar a disposició dels usuaris la versió 12 d'iOS. Juntament amb aquesta nova versió del sistema operatiu venen també petits canvis en Xcode i, en alguns apartats de Cocoa Touch, les llibreries que utilitzem per desenvolupar les nostres aplicacions.

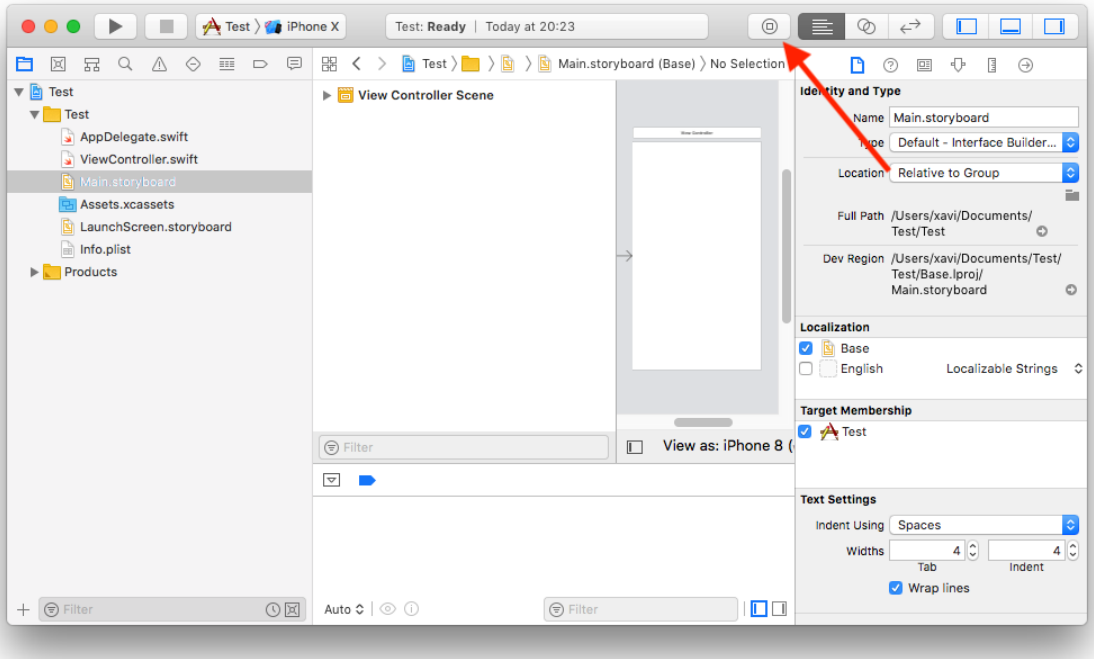
A continuació, descrivim els canvis més significatius, aquells que ens poden afectar i que hem de tenir en compte a mesura que seguim els materials de l'assignatura o altres recursos que consultem.

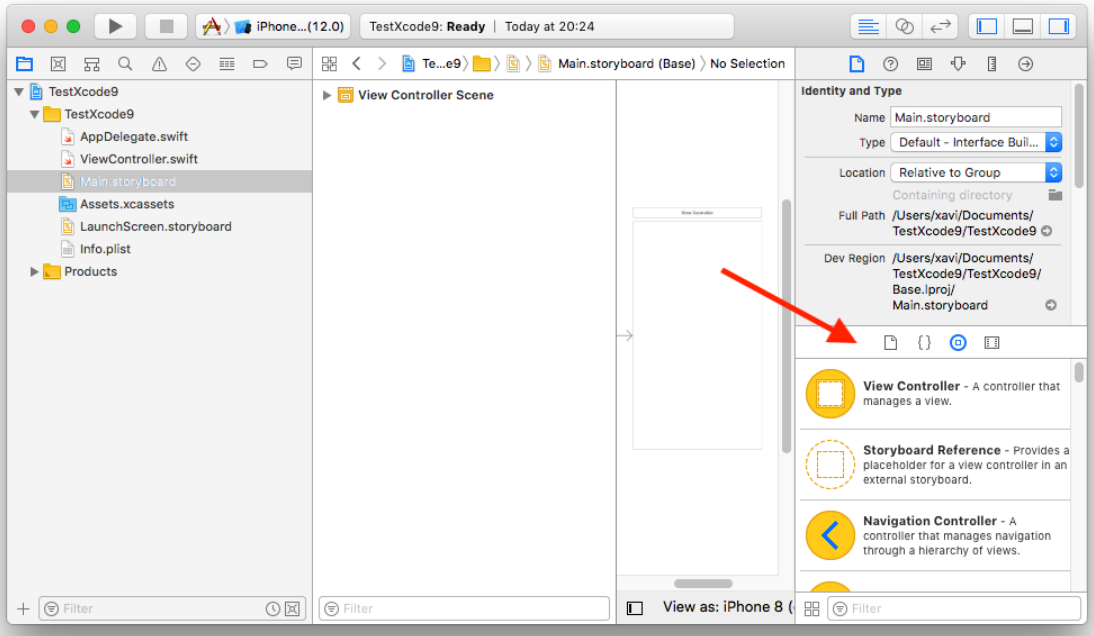
Canvis en Xcode 10

Biblioteca de components

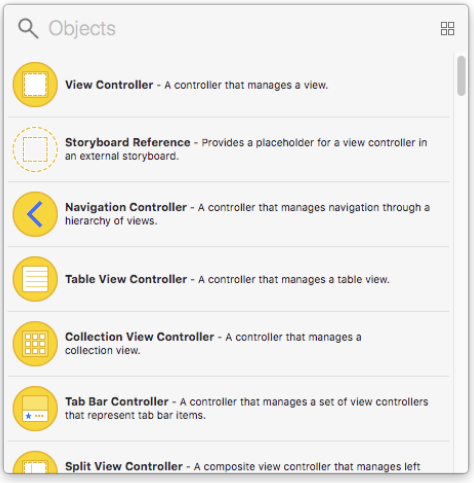
L'accés a la biblioteca de components s'ha mogut des de la part inferior de l'inspector en les versions anteriors.

En la versió 10 d'Xcode ara està accessible des d'aquest botó de la zona superior dreta de la pantalla:



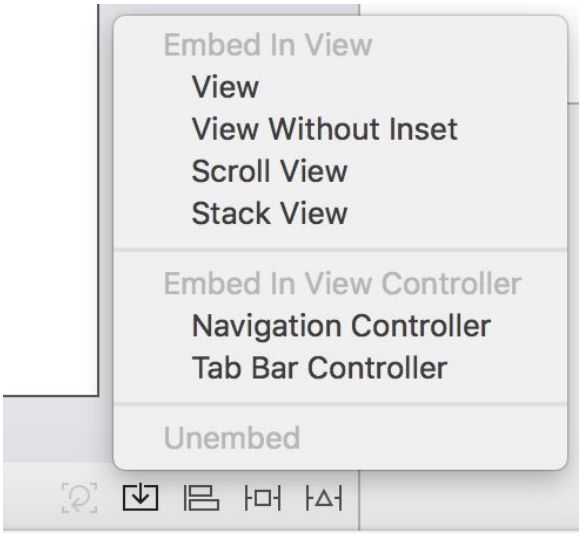


Quan premem el botó  s'obrirà en una finestra la llista que abans estava situada a l'*inspector*:



Botó per incloure controls en *stacks*

En Interface Builder s'ha ampliat el botó que permetia incloure controls dins d'un *stack*. Ara inclou totes les opcions que permeten incloure controls dins d'altres contenidors.



En l'enllaç següent podem consultar informació addicional sobre els canvis en Xcode 10: https://medium.com/developerinsider/whats-new-in-xcode-10-fddeab035d05.

Canvis en el llenguatge Swift i UIKit

Xcode 10 ve amb la versió 4.2 de Swift. Quan Apple va incloure Swift com a llenguatge de programació en UIKit es va fer una primera adaptació de les crides existents en Objective-C a la seva versió corresponent a Swift. Com podria ocórrer amb els llenguatges naturals, aquesta primera traducció era molt literal respecte a l'Objective-C original. Per això es pot dir que, encara que és Swift, s'assembla a Objective-C. Amb les versions successives es va polint aquesta traducció i es van incorporant cada vegada més característiques de Swift.

Canvis en la nomenclatura de tipus

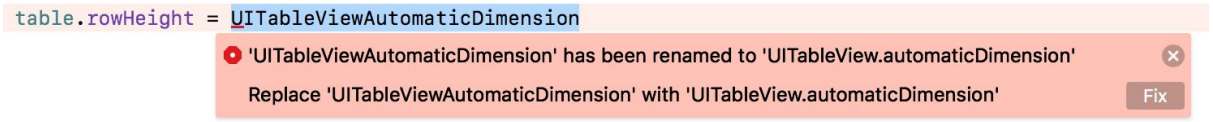
En les primeres versions, els noms dels objectes d'UIKit es traduïen directament de la seva versió en Objective-C. Per exemple, en versions anteriors teníem aquesta variable global:

```
UITableViewAutomaticDimension
```

En la versió actual, s'ha convertit en una propietat del tipus *UITableView*:

```
UITableView.automaticDimension
```

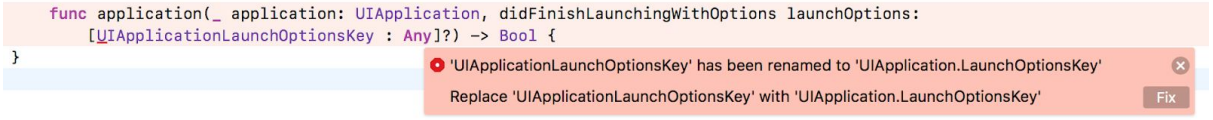
En la majoria dels casos, l'editor d'Xcode ja s'adonarà que intentem usar part de la sintaxi obsoleta i ens proposarà corregir el nostre codi de manera automàtica per solucionar-ho:



Si premem el botó *Fix*, el codi quedarà corregit:

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

Un altre exemple, amb el codi a *AppDelegate*, el tenim amb *UIApplicationLaunchOptionsKey*, que s'ha canviat a *UIApplication.LaunchOptionsKey*. Com en el cas anterior, Xcode ja ens proposarà la solució:



I amb *Fix* ens corregirà el codi:

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

La raó que es realitzin aquests canvis és que en Swift disposem de funcionalitats que no teníem en Objective-C, com els tipus jeràrquics i les propietats de classe. Això ens permet tenir una nomenclatura més agrupada i lògica, mentre que, en Objective-C, per agrupar es tendia a afegir prefixos als noms de classes, propietats i enumeracions.

Afegir el prefix *@objc*

El prefix *@objc* s'utilitza en Swift perquè una classe, propietat o funció sigui accessible des del codi en Objective-C. El cas és que, encara que la nostra aplicació estigui desenvolupada en Swift, hi ha vegades que el nostre codi ha d'interactuar amb codi en Objective-C; per exemple, UIKit encara és Objective-C.

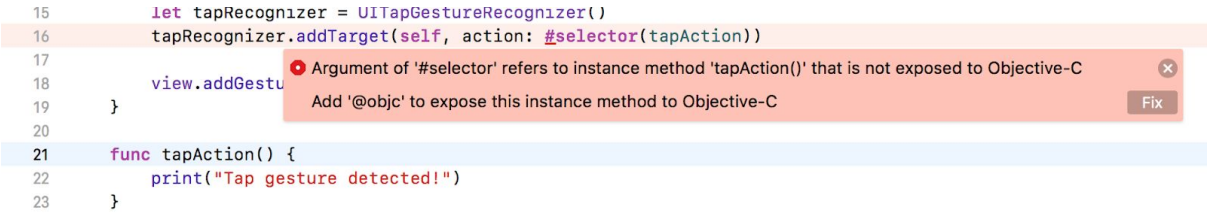
En les versions anteriors, el compilador de Swift s'encarregava d'anar exposant automàticament les parts de codi Swift a la resta de codi Objective-C, però es va detectar que, en exposar de manera automàtica el codi Swift, hi havia molts casos en què es generava codi massa complex, s'exposava més del necessari i també hi havia reducció de rendiment. Per això, en la versió actual es va decidir restringir els casos en què el codi Swift s'exposa de manera automàtica.

Això es tradueix en què en alguns casos hem de prefixar les nostres funcions amb *@objc* en casos en què abans no era necessari.

És el cas del reconeixement de gestos a la interfície d'usuari, com en aquest exemple:

```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 traiem *@objc* de la funció *tapAction*, el compilador es queixarà que no és capaç de completar l'expressió de *#selector*. De tota manera, Xcode també en aquest cas ens proposarà solucionar-ho:



Podeu veure un exemple complet de reconeixement de gestos en el capítol 19 del llibre, i teniu el codi d'exemple amb la solució en l'àrea de recursos de l'assignatura.

Com a informació addicional, podem consultar aquesta pàgina web, en la qual podem veure, de manera exhaustiva i amb exemples, els canvis que hi ha hagut entre les diferents versions de Swift: https://www.whatsnewinswift.com.