

Swift : beyond the basics

Beyond and beyond !

Plan

- Classes et structures
- Énumérations
- Initialisation
- Property Wrappers
- La documentation
- Du playground à l'application

Classes et structures

Généralités

- En Swift, les classes et les structures sont plus proches que dans d'autres langages
- Les classes ont quelques capacités supplémentaires
- Les instances de classes sont gérées par **référence**
- Les instances de structures sont gérées par **valeur**

Classes et structures

	Classe	Structure
Définir des propriétés		
Définir des méthodes		
Définir des initialiseurs		
Possibilité d'extension		
Héritage		
"Désinitialiseurs"		
Passage par référence		
Initialiseur des propriétés auto-généré		

Définitions des classes et structures

```
struct Engine {  
    var name = "Engine"  
    var horsePower = 0  
    var nbOfCylinder = 0  
    var fuel = "Diesel"  
}
```

```
struct Engine {  
    var name = "Engine"  
    var horsePower = 0  
    var nbOfCylinder = 0  
    var fuel = "Diesel"  
}  
  
class Vehicule {  
    var name = "Not Branded"  
    var speed = 0  
    var engine = Engine()  
}
```

Classes et structures

```
    val nbCylinder = 0
    var fuel = "Diesel"
}
```

```
class Vehicule {
    var name = "Not Branded"
    var speed = 0
    var engine = Engine()
}
```

```
class Car: Vehicule {
    var isConvertible = false
    var isAutomatic = false
}
```

Propriétés

- Propriétés stockées
 - Variable ou constante associée avec l'instance
 - Initialisées au moment de la création de l'instance par défaut
 - Initialisée au moment de l'utilisation si définie avec le préfixe **lazy** (**var** uniquement)
- Propriétés calculées
 - Fournissent des accesseurs (**get** et **set**) pour récupérer ou modifier une ou plusieurs propriétés stockées

```
extension Engine {  
  
    var computedName: String {  
        get {  
            return "Moteur \$(fuel) \$(horsePower) ch"  
        }  
  
        set(newComputedName){  
            print(newComputedName)  
        }  
    }  
}
```

```
extension Engine {  
  
    var computedName: String {  
        get {  
            return "Moteur \$(fuel) \$(horsePower) ch"  
        }  
  
        set {  
            print(newValue)  
        }  
    }  
}
```

Propriétés calculées (lecture seule)

```
extension Engine {  
  
    var computedName: String {  
        get {  
            return "Moteur \$(fuel) \$(horsePower)ch"  
        }  
    }  
}
```

Propriétés calculées (lecture seule)

```
extension Engine {  
  
    var computedName: String {  
        return "Moteur \$(fuel) \$(horsePower)ch"  
    }  
}
```

Propriétés de type

- Une propriété de type est utile pour partager des informations entre toutes les instances
 - Préfixe **static** devant la déclaration de la propriété
 - Propriété stockée ou calculée
- Pour une classe vous pourrez aussi utiliser **class** au lieu de **static**.
- **static** correspond à **final class**

Propriétés de type

```
class MaClass {  
  
    static var aStoredTypeProperty = "Some value"  
  
    static var aComputedTypeProperty: Int {  
        return 42  
    }  
}  
  
struct MaStruct {  
  
    static var aStoredTypeProperty = "Some value"  
  
    static var aComputedTypeProperty: Int {  
        return 42  
    }  
}
```

Classes et structures

```
class MaClass {  
  
    static var aStoredTypeProperty = "Some value"  
  
    static var aComputedTypeProperty: Int {  
        return 42  
    }  
}  
  
struct MaStruct {  
  
    static var aStoredTypeProperty = "Some value"  
  
    static var aComputedTypeProperty: Int {  
        return 42  
    }  
}  
  
print(MaStruct.aStoredTypeProperty)  
print(MaStruct.aComputedTypeProperty)  
print(MaClass.aComputedTypeProperty)  
print(MaClass.aStoredTypeProperty)
```

Observateurs

- Il est possible d'observer une propriété pour réagir à ses changements de valeurs
- Deux observateurs possibles
 - Avant le changement de la valeur (**willSet**)
 - Permet d'accéder à la nouvelle valeur (**newValue**)
 - Après le changement de la valeur (**didSet**)
 - Permet d'accéder à l'ancienne valeur (**oldValue**)

 Non appelés lors de l'initialisation !

Observateurs

```
var observedProperty: String = "Initial Value" {  
    willSet {  
        print("Will set \(newValue) instead of \(observedProperty)")  
    }  
  
    didSet {  
        print("Did set \(observedProperty) instead of \(oldValue)")  
    }  
}
```

Méthodes

- Les méthodes sont des fonctions associées à des types particuliers
- Classes et structures peuvent définir :
 - Des méthodes d'instances qui seront utilisées avec leurs instances
 - Des méthodes de types qui seront utilisées avec le type directement

Méthodes

- Une méthode est une fonction.
- La syntaxe pour déclarer une méthode est la même que pour déclarer une fonction.
- En cas de surcharge d'une méthode, il faut préfixer la déclaration par `override`

Méthodes de type

- Comme pour les propriétés de type, une méthode de type est une méthode associée à un type en général, et pas à une instance
- On utilise le préfixe **static**

Contrôle d'accès

External module : Subclass / Override

External module : Access only

Module : Access / Subclass / Override

File : Access / Subclass / Override

Scope :
Access / Subclass / Override

private

fileprivate

internal

public

open

Cas des types valeur

- Une structure est un type valeur
- Par défaut, les propriétés d'un type valeur ne peuvent pas être modifiés depuis une méthode d'instance
- En cas de besoin de ce type de comportement, il faut déclarer la méthode en **mutating**.
- La méthode pourra ensuite modifier les propriétés de la structure

Cas des types valeur

```
extension Engine {  
    ! func add(newCylinders numberOfNewCylinders: Int) {  
        nbOfCylinder += numberOfNewCylinders  
    }  
}
```

Cas des types valeur

```
extension Engine {  
    mutating func add(newCylinders numberOfNewCylinders: Int) {  
        nbOfCylinder += numberOfNewCylinders  
    }  
  
    defaultEngine.addCylinders(2)
```

self

- Dans une méthode d'instance, la propriété implicite **self** symbolise l'instance actuelle
- Dans une méthode de type, la propriété implicite **self** symbolise le type en lui-même
- L'utilisation de **self** n'est pas obligatoire, sauf dans des cas où une confusion est possible (nom interne d'argument identique à un nom de propriété, capture dans une clôture, etc.)

self

```
extension Vehicule {  
    func replaceName(by name: String){  
        ! name = name  
    }  
}
```

self

```
extension Vehicule {  
    func replaceName(by name: String){  
        self.name = name  
    }  
}
```

Transtypage

- Swift ne réalise pas de transtypage implicite
- Possibilité de réaliser des transtypage explicites entre des types liés par héritage
 - Transtypage "vers le haut" sans risque (le compilateur peut vérifier)
 - Transtypage "vers le bas" avec risque (le compilateur ne peut pas vérifier)
- Possibilité de vérifier les types à l'exécution

Transtypage vers le haut

```
let aCar: Car
```

```
let aCar = Car()
```

```
let aCarConsideredAsVehicule = aCar as Vehicule
```

```
let aCarConsideredAsVehicule: Vehicule
```

Transtypage vers le bas

```
let aCar = Car()
```

```
let aCarConsideredAsVehicule = aCar as Vehicule
```

```
let aCarIsACar = aCarConsideredAsVehicule as? Car
```

```
let aCarIsACar: Car?
```

Transtypage vers le bas

```
let aCar = Car()
```

```
let aCarConsideredAsVehicule = aCar as Vehicule
```

```
let aCarIsACar = aCarConsideredAsVehicule as! Car
```

```
let aCarIsACar: Car
```

Vérification dynamique du type

```
let aCarConsideredAsVehicule: Vehicule  
if aCarConsideredAsVehicule is Car {  
    // do something  
}
```

Énumérations

Généralités

```
enum TransportStatus {  
    case onTime  
    case delayed  
    case cancelled  
    case unknown  
}
```

```
var status = TransportStatus.onTime  
status = .delayed
```

Généralités

- Définition d'un type pour un groupe de valeurs liées entre elles
- Plus flexibles qu'en C
- Possibilité de fournir :
 - Soit une valeur brute (rawValue) : **String**, **Character**, **Int**, **Float** ou **Double**
 - Soit une valeur associée de n'importe quel type
- Peuvent avoir des méthodes, initialiseurs, propriétés calculées
- Type valeur !

Valeurs brutes

```
enum TransportType: String {
    case plane = "plane"
    case train = "train"
    case bus = "bus"
    case carSharing = "carSharing"
}

if let transport = TransportType(rawValue: "plane") {
    print(transport.rawValue)
}
```

plane

Valeurs brutes

```
enum TransportType: String {  
    case plane  
    case train  
    case bus  
    case carSharing  
}  
  
if let transport = TransportType(rawValue: "plane") {  
    print(transport.rawValue)  
}
```

plane

Valeurs associées

```
enum TransportStatus {  
    case onTime  
    case delayed (delay: Int, reason: String)  
    case cancelled (reason: String)  
    case unknown  
}
```

```
var status = TransportStatus.onTime  
status = .delayed(delay: 15, reason: "Driver is late")
```

```
switch status {  
    case .onTime:  
        print("Everything is OK")  
  
    case .delayed (let delay, let message) where delay < 15:  
        print("A small delay is expected : \(message)")  
  
    case .delayed (let delay, let message) where delay >= 15:  
        print("A delay is expected : \(message). Please be patient.")  
  
    case .cancelled (let reason):  
        print("Your transport have been cancelled. Reason : \(reason)")  
  
    case .unknown:  
        print("We don't have any status info at this time")  
  
    default:  
        break  
}
```



Switch exhaustif ou
doit inclure default !

Initialisation

Généralités

- L'initialisation est le processus qui permet de préparer une instance à être utilisée
- Lors de l'initialisation, toutes les propriétés doivent se voir affecter une valeur
- L'initialisation peut soit passer par des valeurs par défaut aux propriétés, ou par l'implémentation de méthodes `init`

Généralités

```
class Human {  
  
    var age: Int  
    let name: String  
    var size: Float  
    var gender: String  
    var childrens: [Human]?  
}
```

Initialisation

```
class Human {  
  
    var age: Int  
    let name: String  
    var size: Float  
    var gender: String  
    var childrens: [Human]?  
  
    init() {  
  
        self.age = 0  
        self.name = "John Doe"  
        self.size = 175  
        self.gender = "Male"  
    }  
}
```

```
var childrens: [Human]?

init() {

    self.age = 0
    self.name = "John Doe"
    self.size = 175
    self.gender = "Male"
}

init(name: String, age: Int, size: Float, gender: String) {

    self.name = name
    self.age = age
    self.size = size
    self.gender = gender
}

}
```

```
var gender: String
var childrens: [Human]?

convenience init() {
    self.init(name: "John Doe", age: 0, size: 175, gender:
"Male")
}

init(name: String, age: Int, size: Float, gender: String) {
    self.name = name
    self.age = age
    self.size = size
    self.gender = gender
}
}
```

Convenience initialiser

- Un convenience initialiser est un initialiseur qui fait appel à un autre initialiseur au sein du même type, avec `self.init(...)`
- Un convenience initialiser dans une classe doit être préfixé par `convenience`
- Un convenience initialiser ne peut pas être appelé par une sous-classe

Cas de l'héritage

```
class Student: Human {  
    var school = "Not enrolled"  
    var serious = false  
    var grade = 0  
}
```

Initialisation

```
class Student: Human {  
  
    var school: String  
    var serious: Bool  
    var grade: Int  
  
    init(name: String, age: Int, size: Float, gender: String,  
school: String, isSerious: Bool, grade: Int) {  
  
        self.school = school  
        self.serious = isSerious  
        self.grade = grade  
  
        super.init(name: name, age: age, size: size, gender:  
gender)  
    }  
}
```

Initialisation

```
self.serious = isSerious
self.grade = grade

super.init(name: name, age: age, size: size, gender:
gender)
}

convenience init() {

    self.init(name: "Student Jane Doe", age: 18, size:
180, gender: "Female", school: "Not enrolled", isSerious:
true, grade: 10)
}

}
```

Initialisation

```
convenience init() {  
    self.init(name: "Student Jane Doe", age: 18, size:  
180, gender: "Female", school: "Not enrolled", isSerious:  
true, grade: 10)  
}  
  
convenience override init(name: String, age: Int, size:  
Float, gender: String) {  
    self.init(name: name, age: age, size: size, gender:  
gender, school: "Not enrolled", isSerious: true, grade: 10)  
}  
}
```

Cas de l'héritage

- Si une classe hérite d'une autre, elle doit s'assurer que les propriétés issues de cette classes ont également des valeurs
- Dans le cas de l'héritage, on doit faire appel à l'initialiseur de la classe parente avec `super.init(...)`
- Si vous surchargez un initialiseur de la classe parente, vous devez rajouter `override` devant le `init`

Failable initialiser

- Un failable initialiser est un initialiseur qui peut échouer.
- Dans certains cas, il se peut qu'un initialiser ne puisse pas créer un objet (argument invalide par exemple)
- Dans ce cas, à l'issue de l'initialisation, on aura un optionnel au lieu de l'instance

```
init?() {  
    // If something blocks init, simply return nil  
}
```

Property Wrappers

Property Wrappers

5.1+

- Permet de traiter une valeur avant son affectation à une variable
 - Transformer la valeur
 - Effectuer une action systématique
- Fournir des valeurs maximales ou minimales
- ...

Property Wrappers

5.1+

- Peuvent être définis avec une classe, structure...
- Effectuent leurs actions à chaque accès à la propriété
- Permettent de résoudre bien des problèmes !

Property Wrappers

5.1+

```
@propertyWrapper
struct Trimmed {

    private(set) var value: String = ""

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

    var wrappedValue: String {
        get { value }
        set { value = newValue.trimmingCharacters(in: .whitespacesAndNewlines) }
    }
}
```

Property Wrappers

5.1+

```
@propertyWrapper
struct Logged<Value> {

    private var value: Value

    init(wrappedValue: Value) {
        self.value = wrappedValue
    }

    var wrappedValue: Value {
        get { value }
        set {
            value = newValue
            print("New value is \(newValue)")
        }
    }
}
```

Property Wrappers

5.1+

```
struct Message {  
    @Trimmed var content: String  
    var date: Date  
    var recipient: Recipient  
}
```

```
struct User {  
    @Logged var username = ""  
}
```

Gestion de la mémoire : ARC

Généralités

- Plateforme mobile = ressources limitées
- Gestion de la mémoire efficace indispensable !
- Donc pas de Garbage Collection

Principe

- Langage de haut niveau = gestion de haut niveau
 - Ne pas se soucier de quand il faut libérer l'instance
 - Simplicité pour le développeur
 - Mais avoir une instance libérée dès qu'elle n'est plus utilisée
 - Utilisation efficace de la mémoire

Principe

- Historiquement, la gestion était manuelle
 - On indiquait quand on utilisait un objet, et quand on avait fini de s'en servir
 - Cela influait sur un compteur de référence (nombre de référence pointant sur un même objet)
- Depuis iOS 5, il existe un système de gestion automatique (ARC).
 - Le développeur n'a plus à gérer le compteur de référence
 - Les nouveaux projets sont en ARC

Automatic Reference Counting

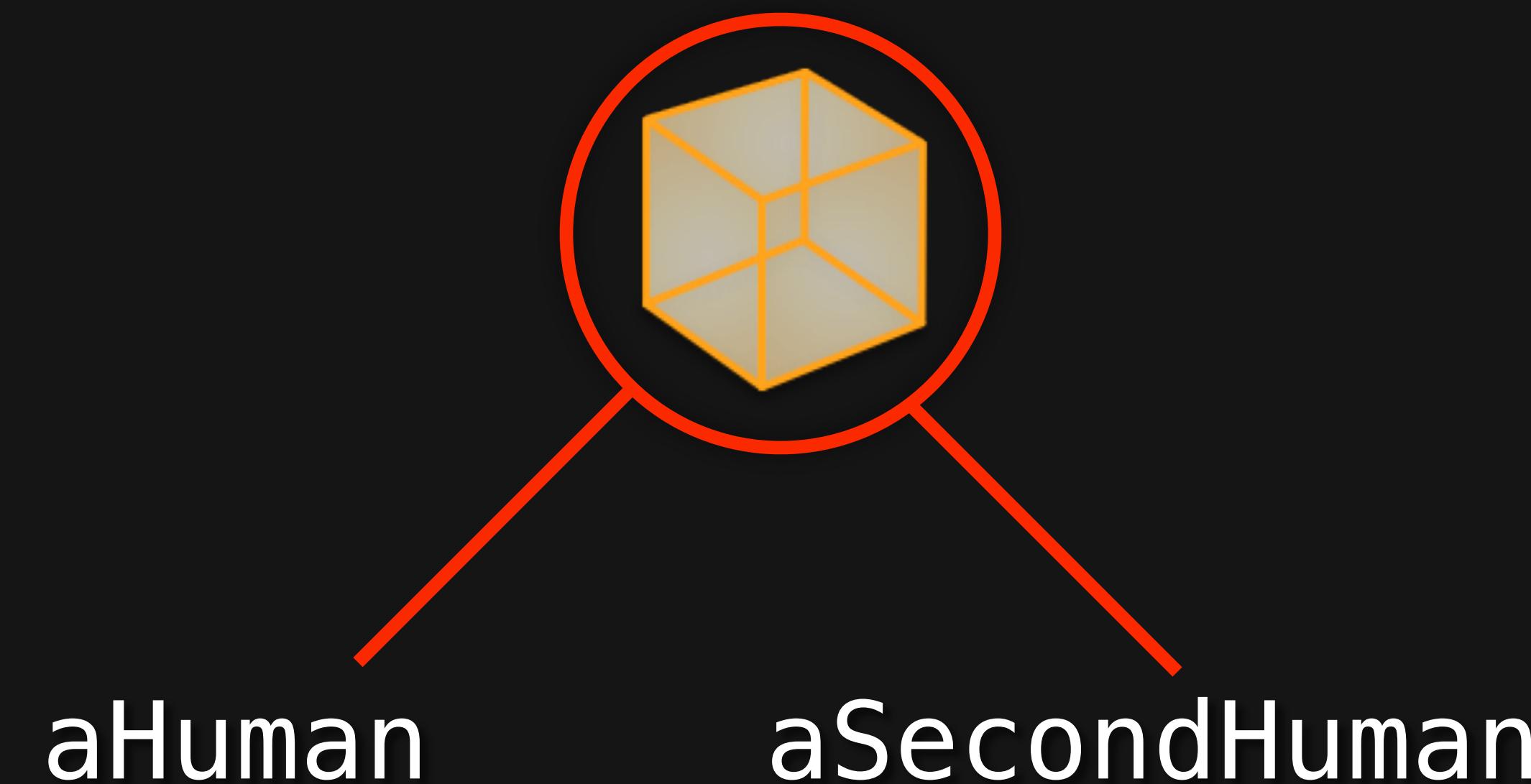
- Fonctionnalité au niveau compilateur
- Compteur de référence géré par le compilateur
- Insère le code de gestion de la mémoire à la compilation
- "Meilleur des deux mondes"

Automatic Reference Counting

- Lors de la création d'un instance de classe ARC crée une référence **strong** entre la propriété et l'instance
- Tant qu'une référence **strong** existe pour une instance, celle-ci reste allouée en mémoire
- Il existe d'autre type de références qui ne suffisent pas à maintenir une instance en mémoire

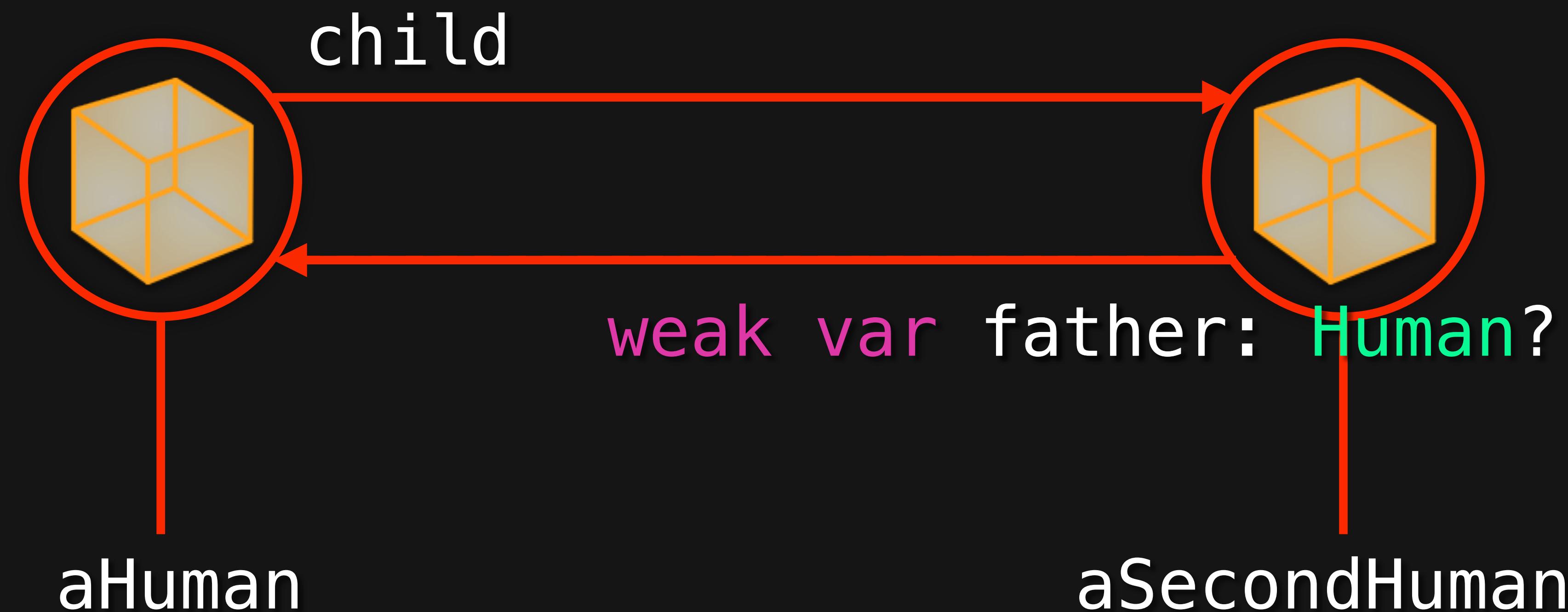
Automatic Reference Counting

```
varaHuman = new Human()  
var aSecondHuman = new Human()
```



Cycle de références

```
var aHuman = new Human();  
var aSecondHuman = new Human();  
aHuman.child = aSecondHuman;  
aSecondHuman.father = aHuman;
```



Automatic Reference Counting

- **strong** : l'objet doit rester vivant tant qu'on pointe vers lui (par défaut)
- **weak** : l'objet restera vivant tant qu'un pointeur **strong** pointera vers lui
 - Le référence est mise à **nil** lorsque l'instance est détruite
 - Une propriété **weak** doit être un optionnel
- **unowned** : l'objet restera vivant tant qu'un pointeur **strong** pointera vers lui
 - La référence n'est pas mise à **nil**. Un appel à la référence après destruction de l'instance engendrera un crash
 - **unowned** est utilisé dans les cas où la propriété est supposée toujours avoir une valeur pendant la vie d'une instance

La documentation

La documentation

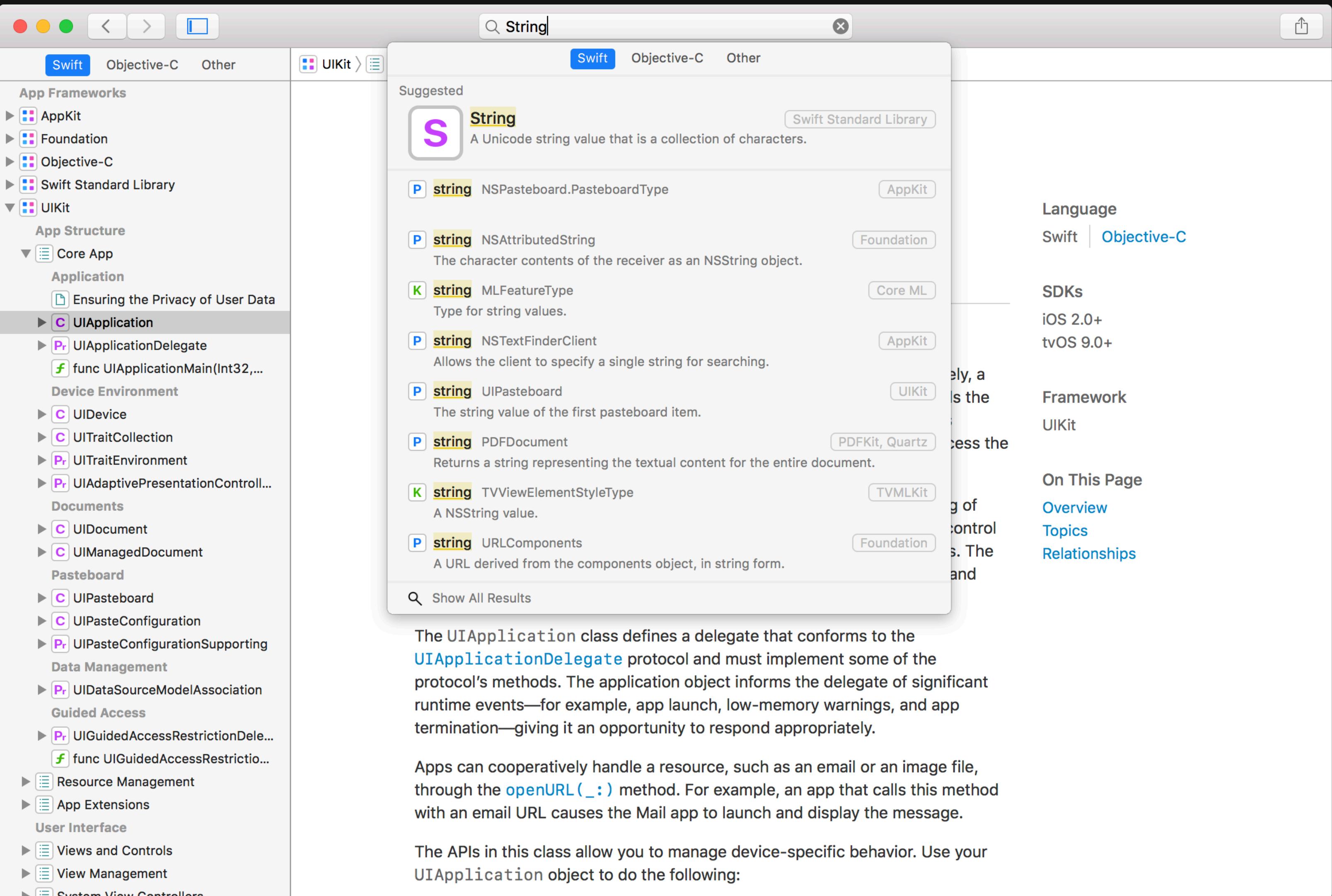
The screenshot shows a web browser window for Apple Inc. with the following details:

- Header:** Apple Developer, Discover, Design, Develop, Distribute, Support, Account, Search icon.
- Section:** Documentation
- Visuals:** Icons for code (list), brace {}, and file.
- Title:** Apple Developer Documentation
- Description:** Browse the latest developer documentation including API reference, articles, and sample code.
- App Frameworks:**
 - Icon: Briefcase
 - Section: App Frameworks
 - Links: AppKit, Foundation, Objective-C, Swift Standard Library, UIKit, WatchKit
- Graphics and Games:**
 - Icon: Three overlapping circles
 - Section: Graphics and Games
 - Links: AGL, ARKit (Beta), ColorSync (Beta), Metal, Metal Performance Shaders, MetalKit

Web : developer.apple.com/documentation

La documentation

Xcode : ⌘↑0



La documentation

Bibliothèque

Documentation

Recherche

Résultats

The screenshot shows the Xcode Documentation browser interface. At the top, there's a search bar with the word "String". Below it, a sidebar titled "Bibliothèque" lists various Swift frameworks and libraries, with "UIKit" currently selected. The main content area, titled "Résultats", displays search results for "String". The first result is "String" from the "Swift Standard Library", described as "A Unicode string value that is a collection of characters". Below it is a list of other results, each with a color-coded icon (blue for Swift, green for Objective-C, grey for Foundation), a type name, and a brief description. To the right of the results, there are sections for "Language" (Swift and Objective-C), "SDKs" (iOS 2.0+ and tvOS 9.0+), "Framework" (UIKit), and "On This Page" (Overview, Topics, Relationships). At the bottom, there's a summary of the UIApplication class and some usage examples.

String

Suggested

S String Swift Standard Library
A Unicode string value that is a collection of characters.

P string AppKit
NSPasteboard.PasteboardType

P string Foundation
The character contents of the receiver as an NSString object.

K string Core ML
Type for string values.

P string AppKit
NSTextFinderClient
Allows the client to specify a single string for searching.

P string UIKit
UIPasteboard
The string value of the first pasteboard item.

P string PDFKit, Quartz
PDFDocument
Returns a string representing the textual content for the entire document.

K string TVMLKit
TVViewElementStyleType
A NSString value.

P string Foundation
URLComponents
A URL derived from the components object, in string form.

Show All Results

The **UIApplication** class defines a delegate that conforms to the **UIApplicationDelegate** protocol and must implement some of the protocol's methods. The application object informs the delegate of significant runtime events—for example, app launch, low-memory warnings, and app termination—giving it an opportunity to respond appropriately.

Apps can cooperatively handle a resource, such as an email or an image file, through the `openURL(_:)` method. For example, an app that calls this method with an email URL causes the Mail app to launch and display the message.

The APIs in this class allow you to manage device-specific behavior. Use your **UIApplication** object to do the following:

Description The `UILabel` class implements a read-only text view. You can use this class to draw one or multiple lines of static text, such as those you might use to identify other parts of your user interface. The base `UILabel` class provides support for both simple and complex styling of the label text. You can also control over aspects of appearance, such as whether the label uses a shadow or draws with a highlight. If needed, you can customize the appearance of your text further by subclassing.

Availability iOS (2.0 and later)

Declared In `UILabel.h`

Reference [UILabel Class Reference](#)

Reference [UILabel Class Reference](#)

Declared In `UILabel.h`

Availability iOS (2.0 and later)

XCODE QUICK HELP

⊜ clic

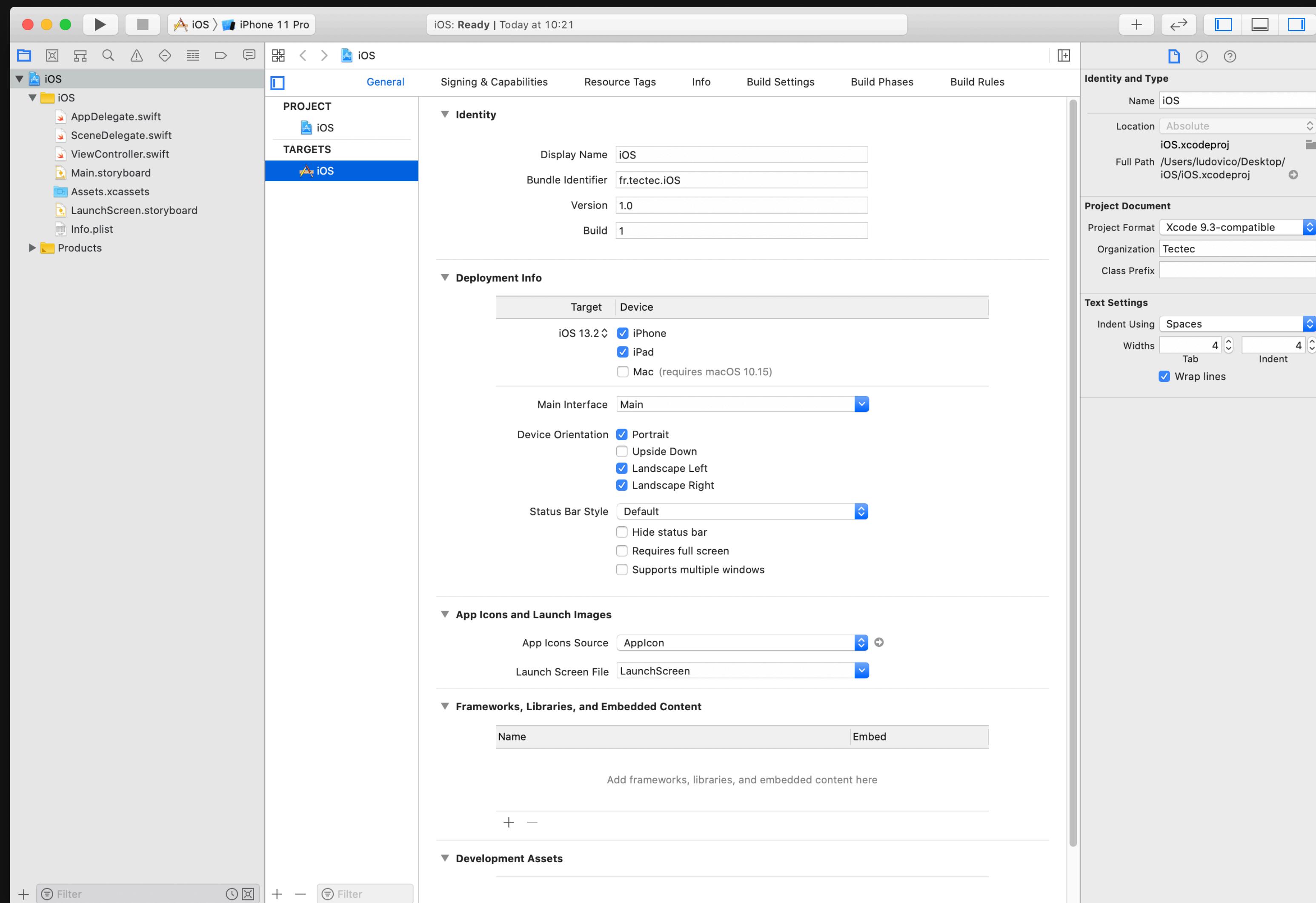
Du playground à l'application

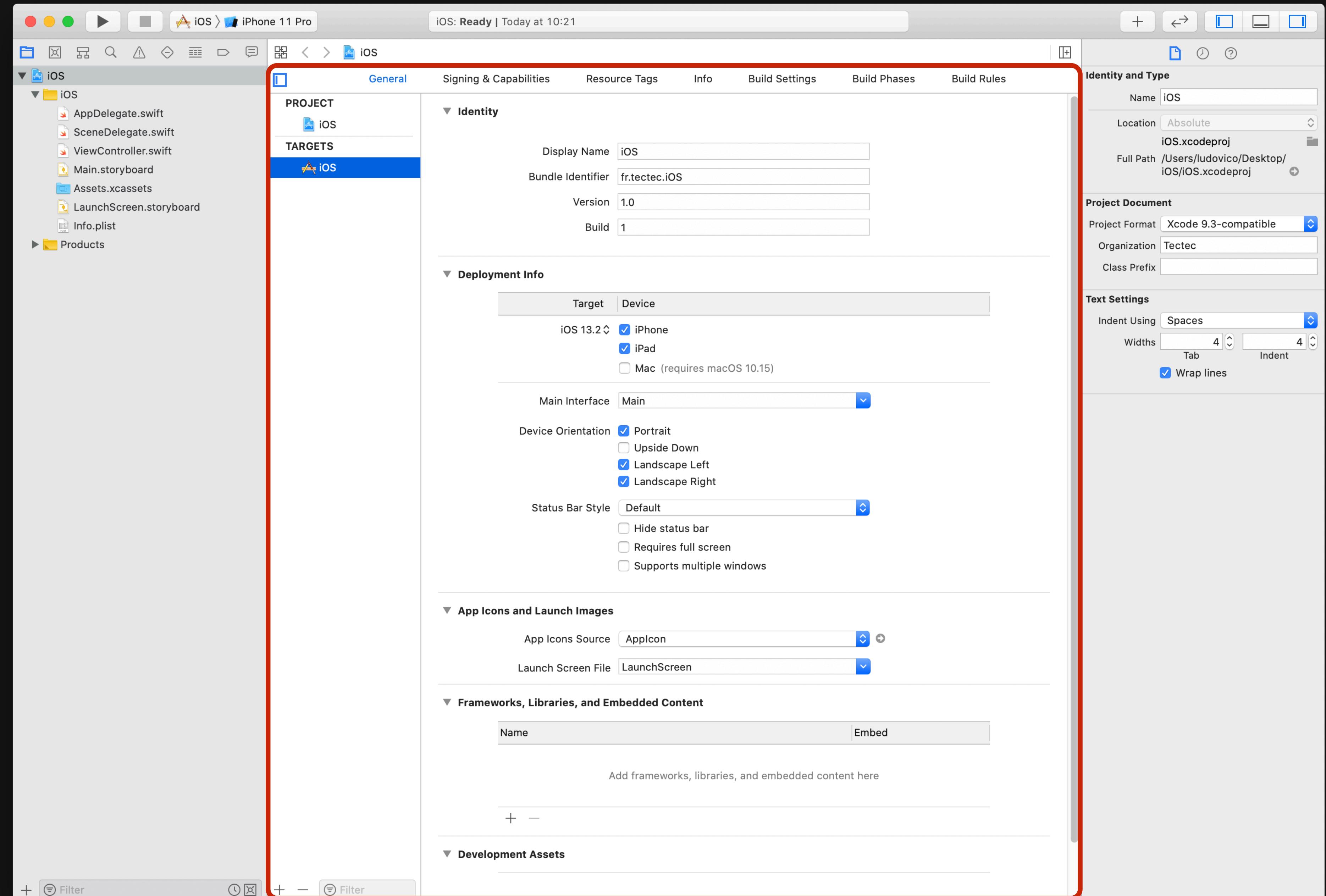
Généralités

- Les playgrounds sont utile pour apprendre le langage, peaufiner des algorithmes et faire des tests rapides
- Mais on voudra rapidement aller plus loin : créer une application iOS complète
- Pour cela, il va falloir découvrir comme lier notre code et notre interface graphique

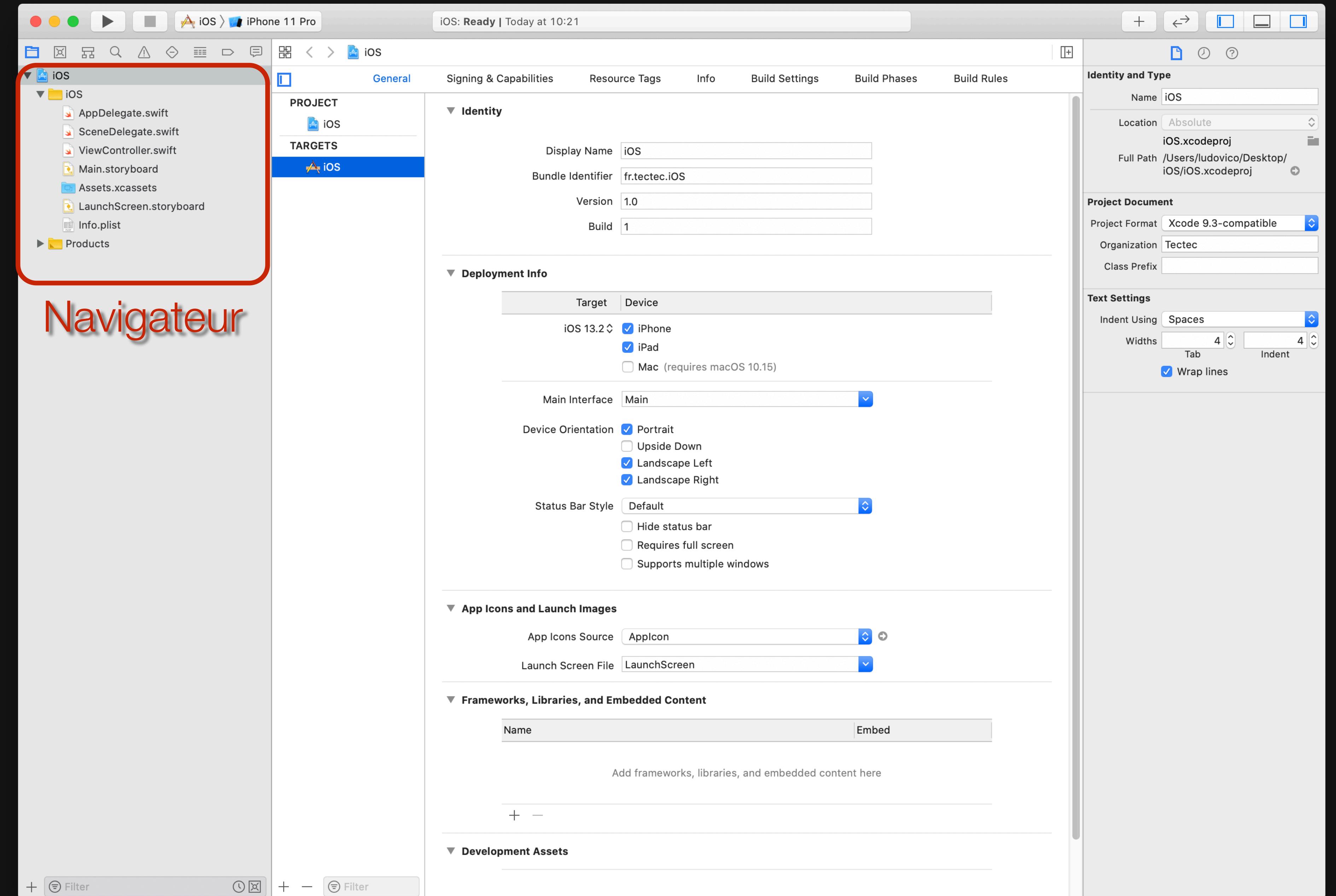
Du playground à l'application

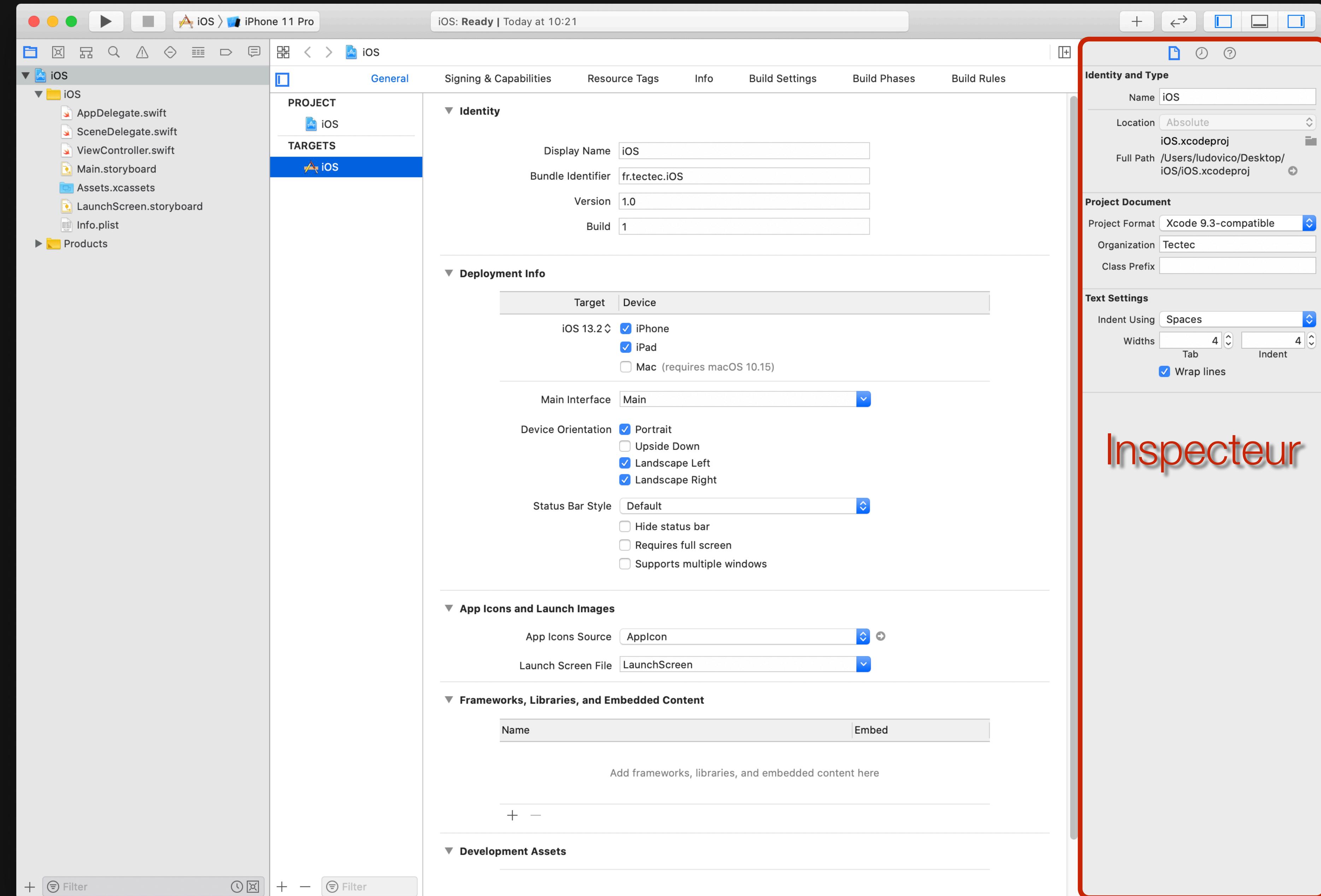
Architecture d'un projet Xcode

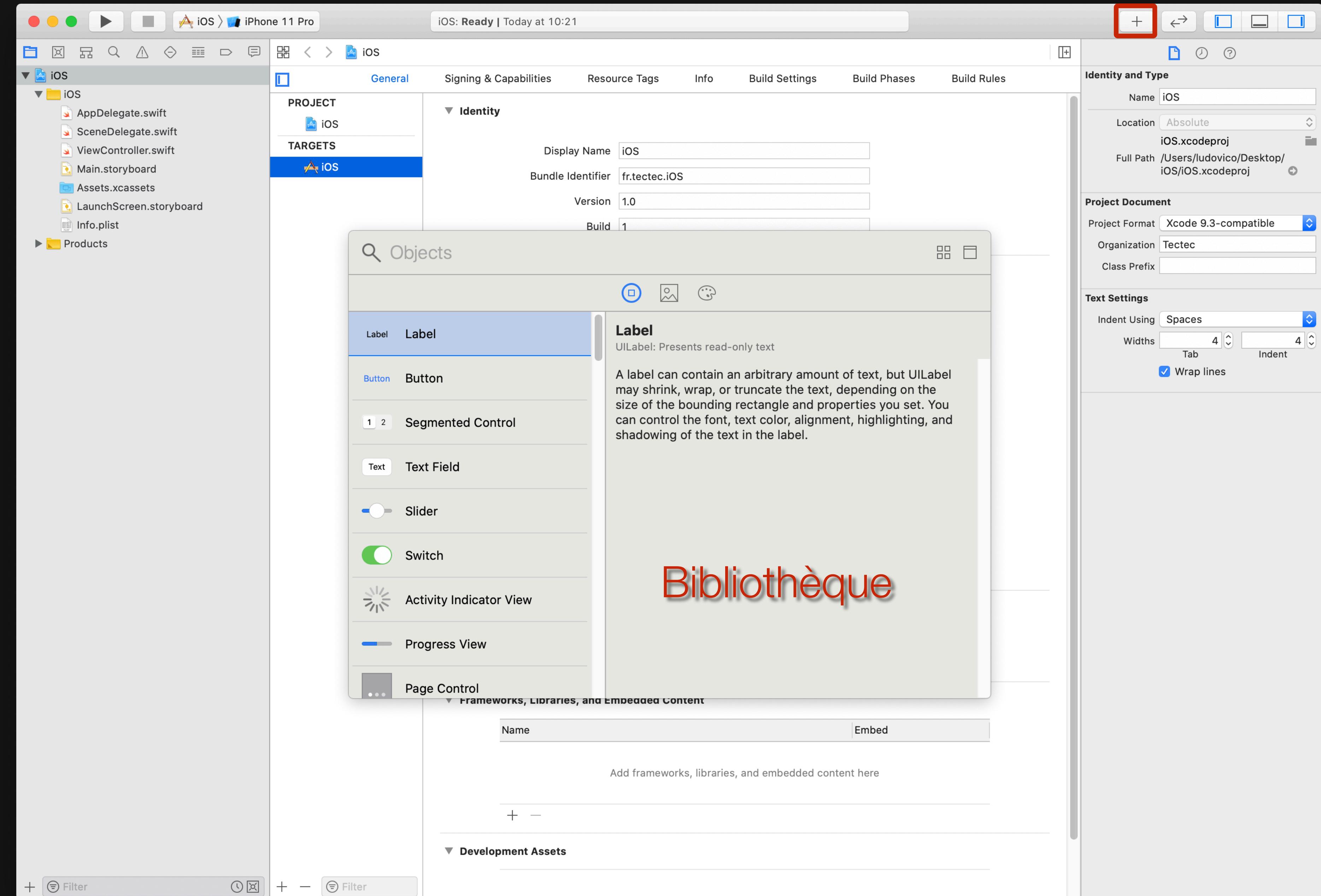




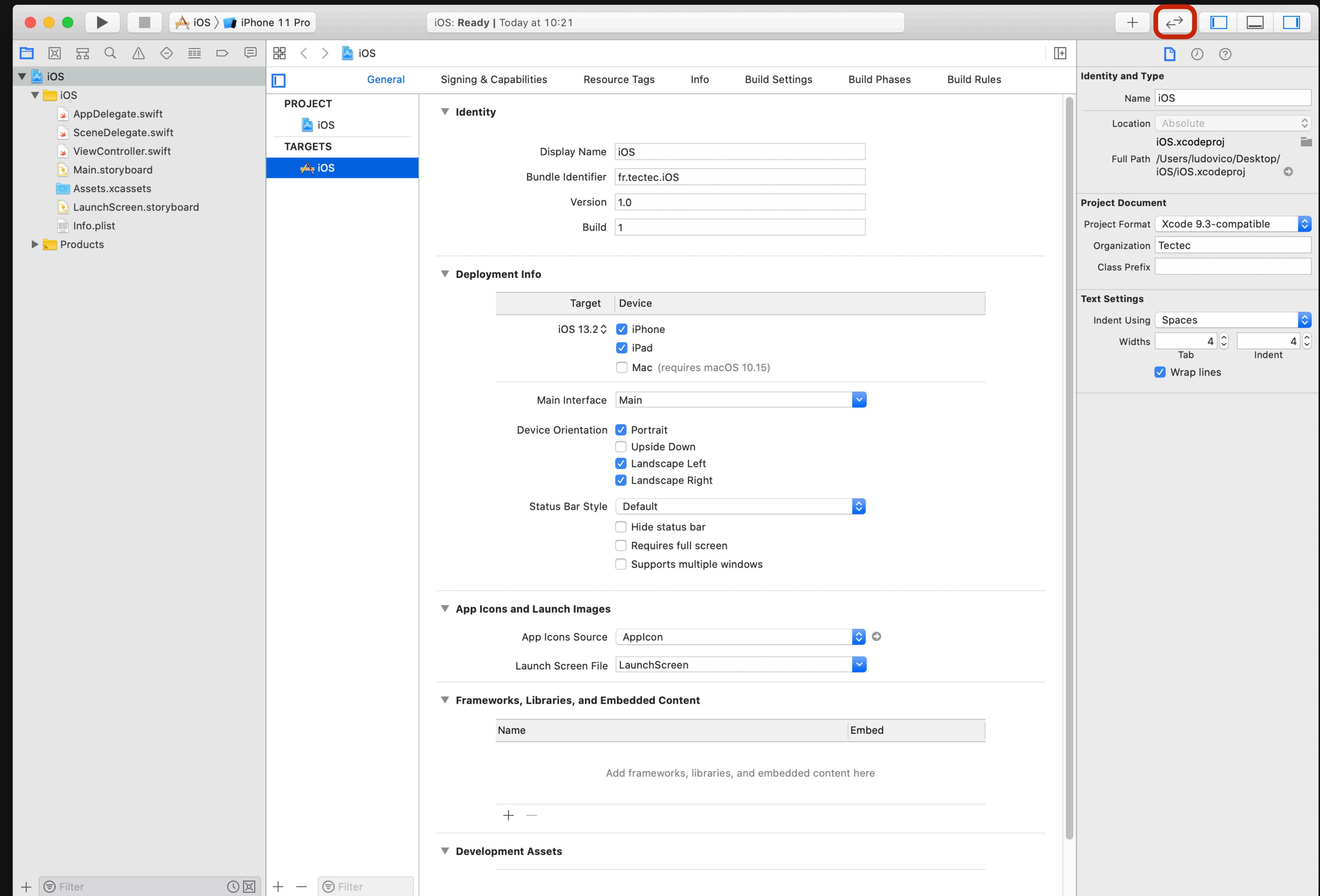
Vue principale



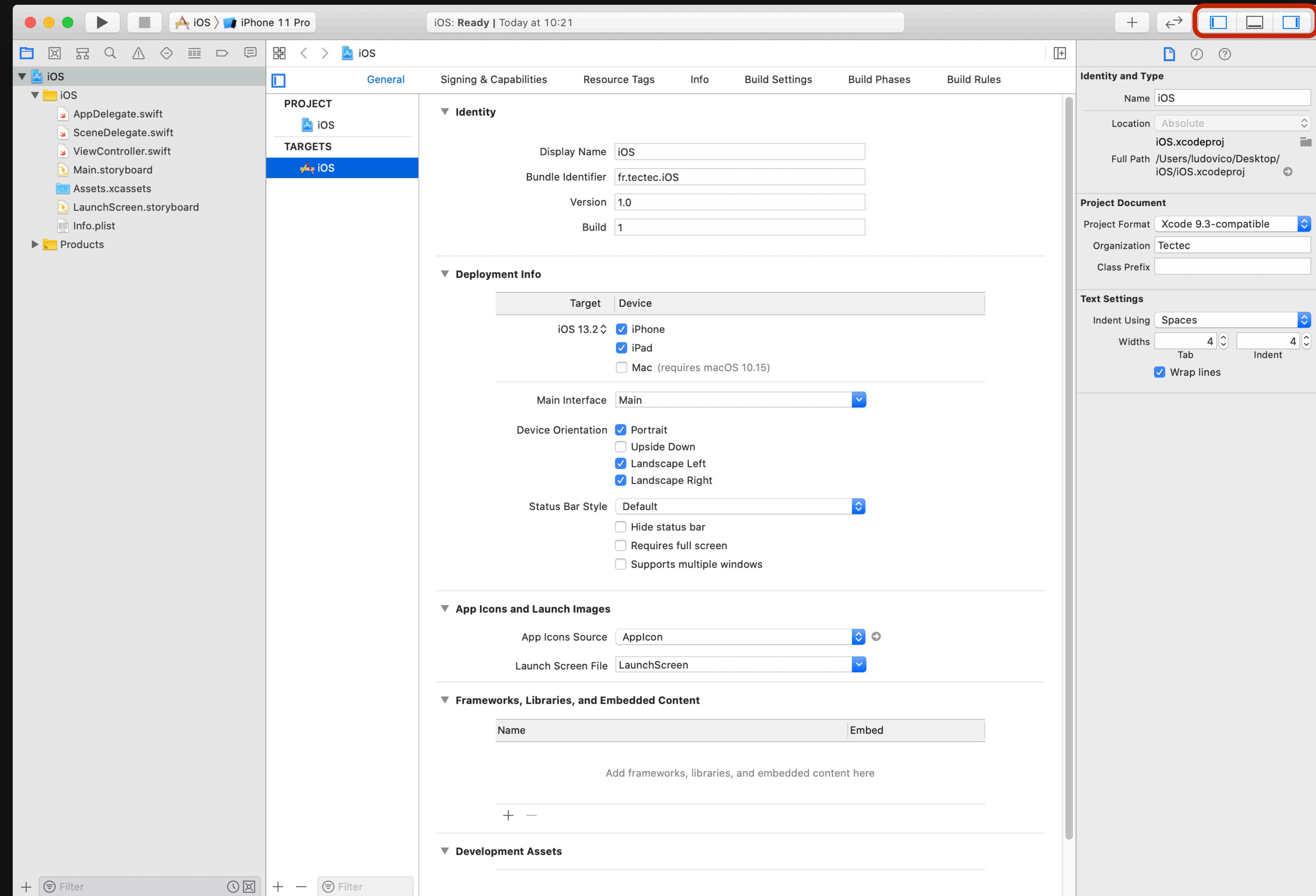




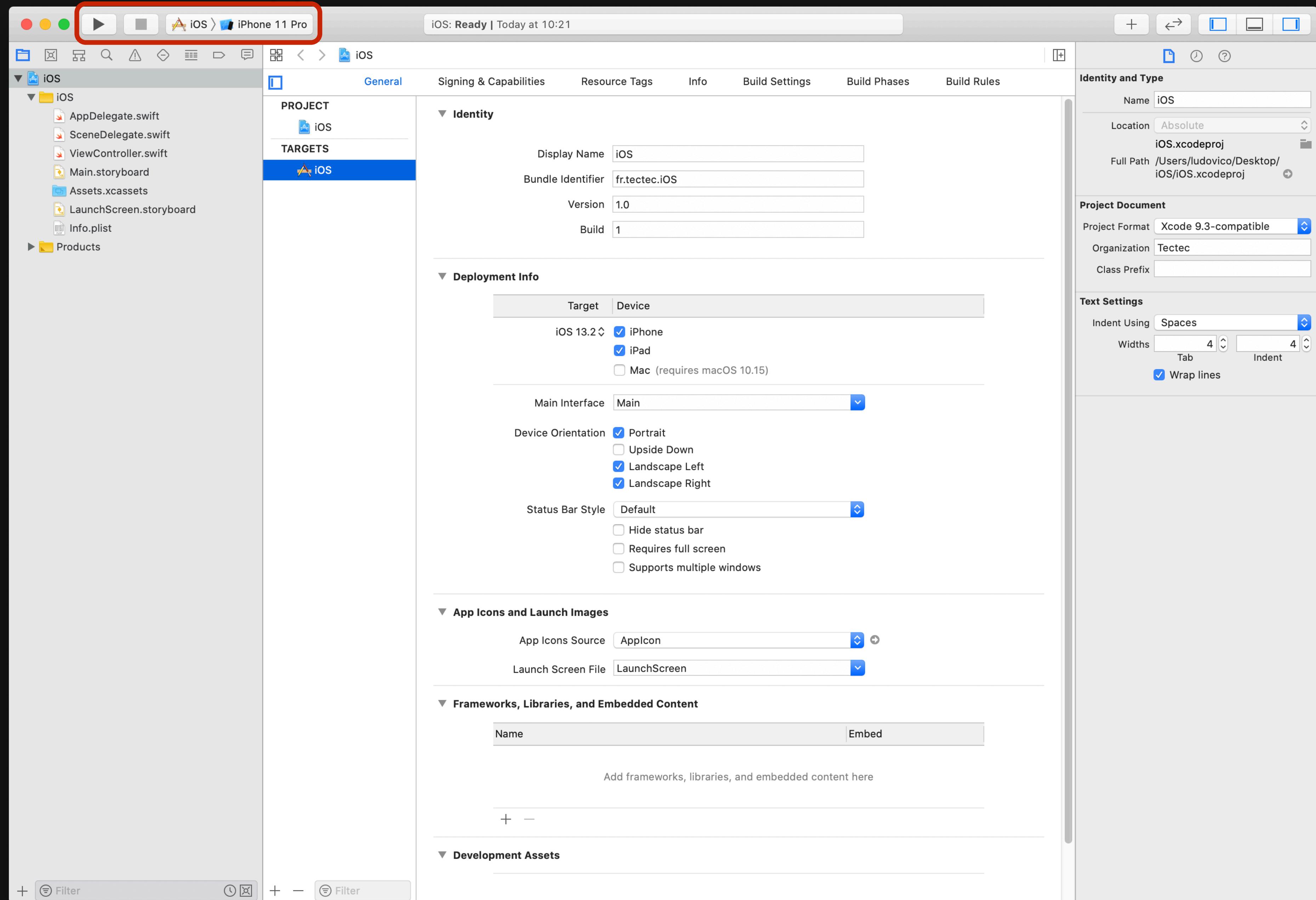
Suivi de versions



Contrôle des vues



Contrôle de l'application



Architecture d'un projet Xcode

- Fichiers .swift
 - Code
 - AppDelegate permet de réagir aux changements d'états de l'application
 - SceneDelegate permet de réagir aux changements d'états de chaque scène (iOS 13+)
- Fichiers .storyboard
 - Interface graphique (UIKit)
- Fichiers .xcassets
 - Bibliothèque de ressources graphiques

Lien avec l'interface

- Action : L'interface graphique appelle une méthode sur une instance
- Outlet : Propriété faisant référence à un élément de l'interface graphique

Lien avec l'interface

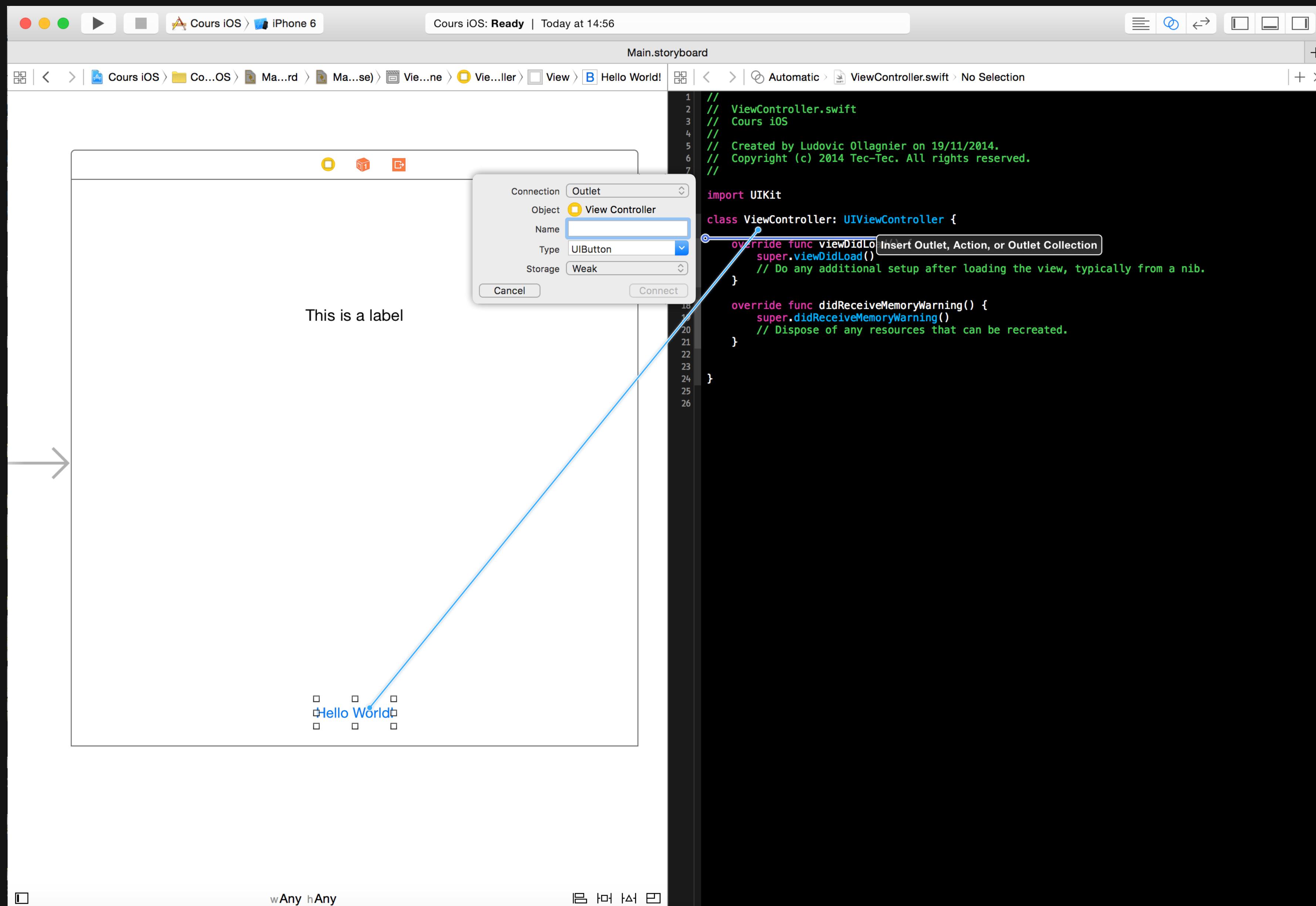
- Déclaration d'une action

```
@IBAction func actionName(_ sender: AnyObject) {  
}
```

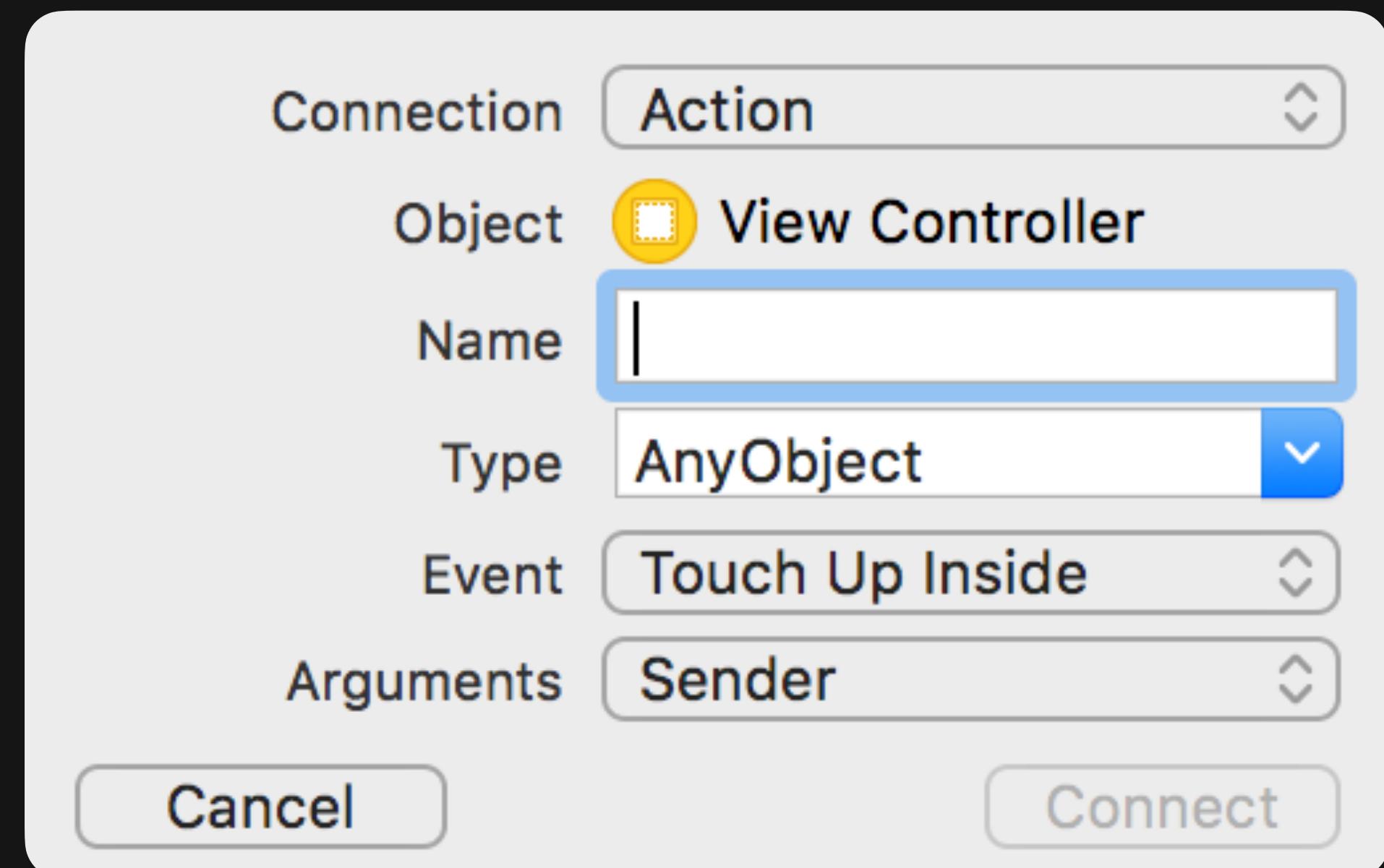
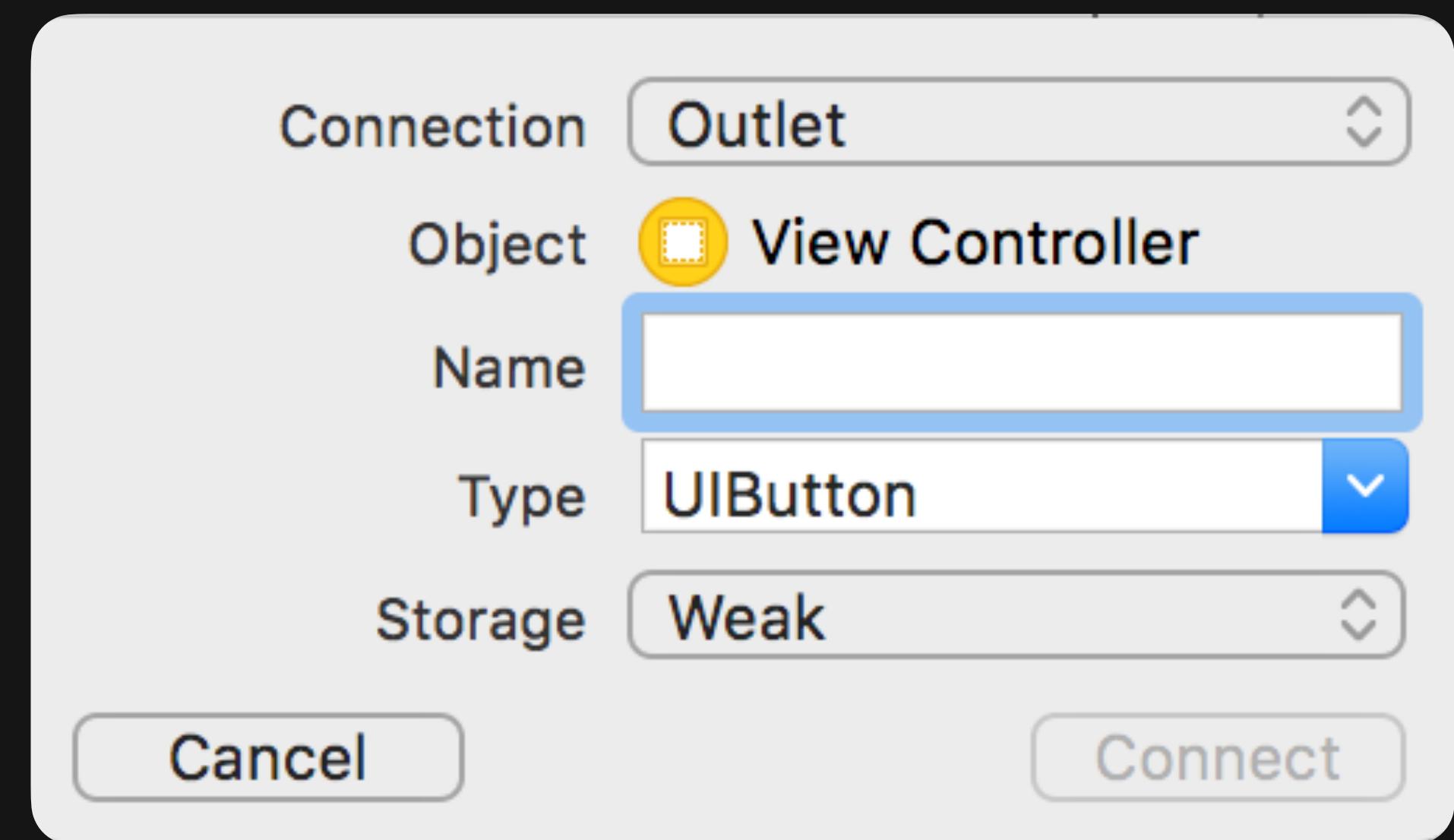
- Déclaration d'un outlet

```
@IBOutlet weak var outletName: OutletType!
```

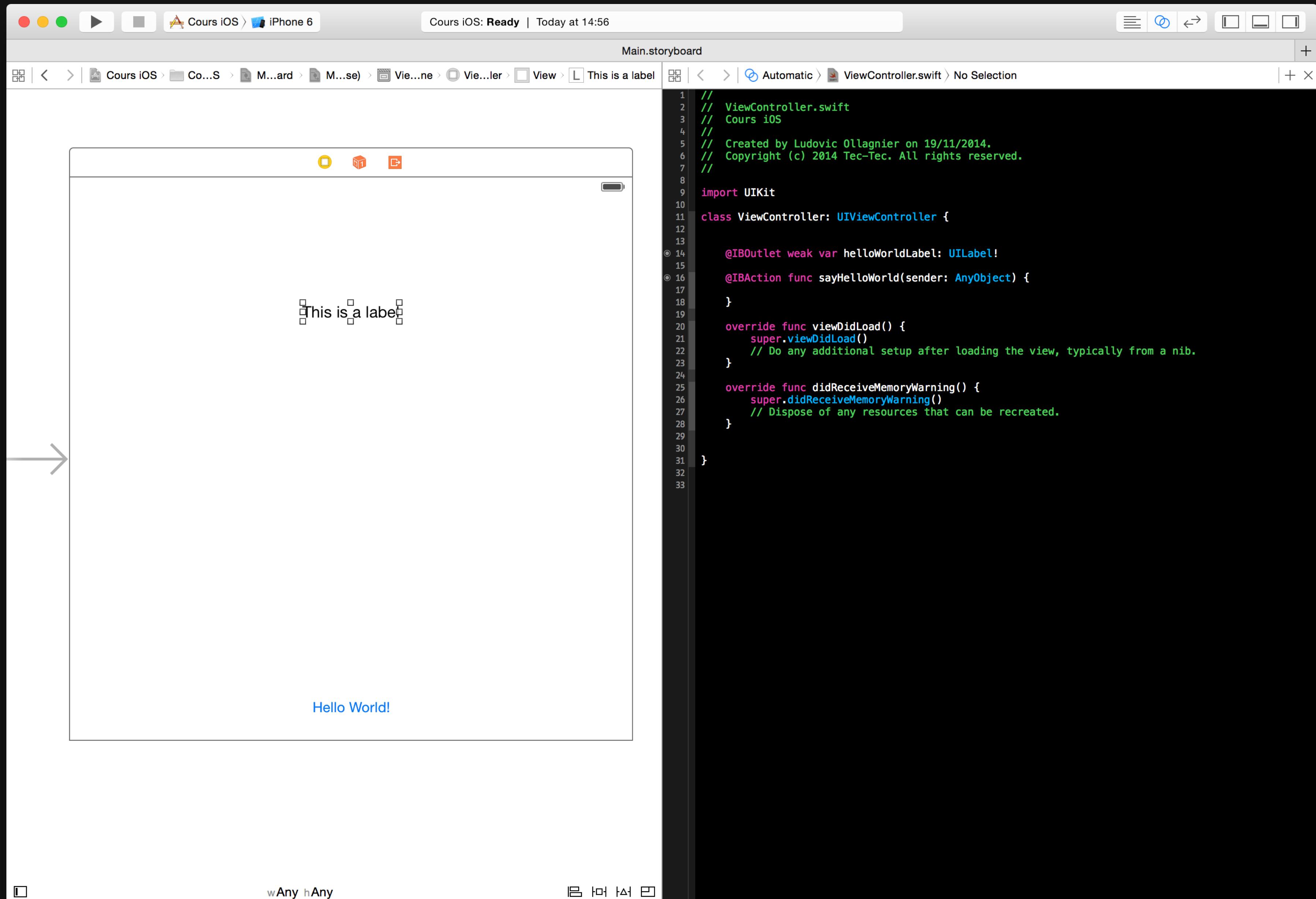
Du playground à l'application



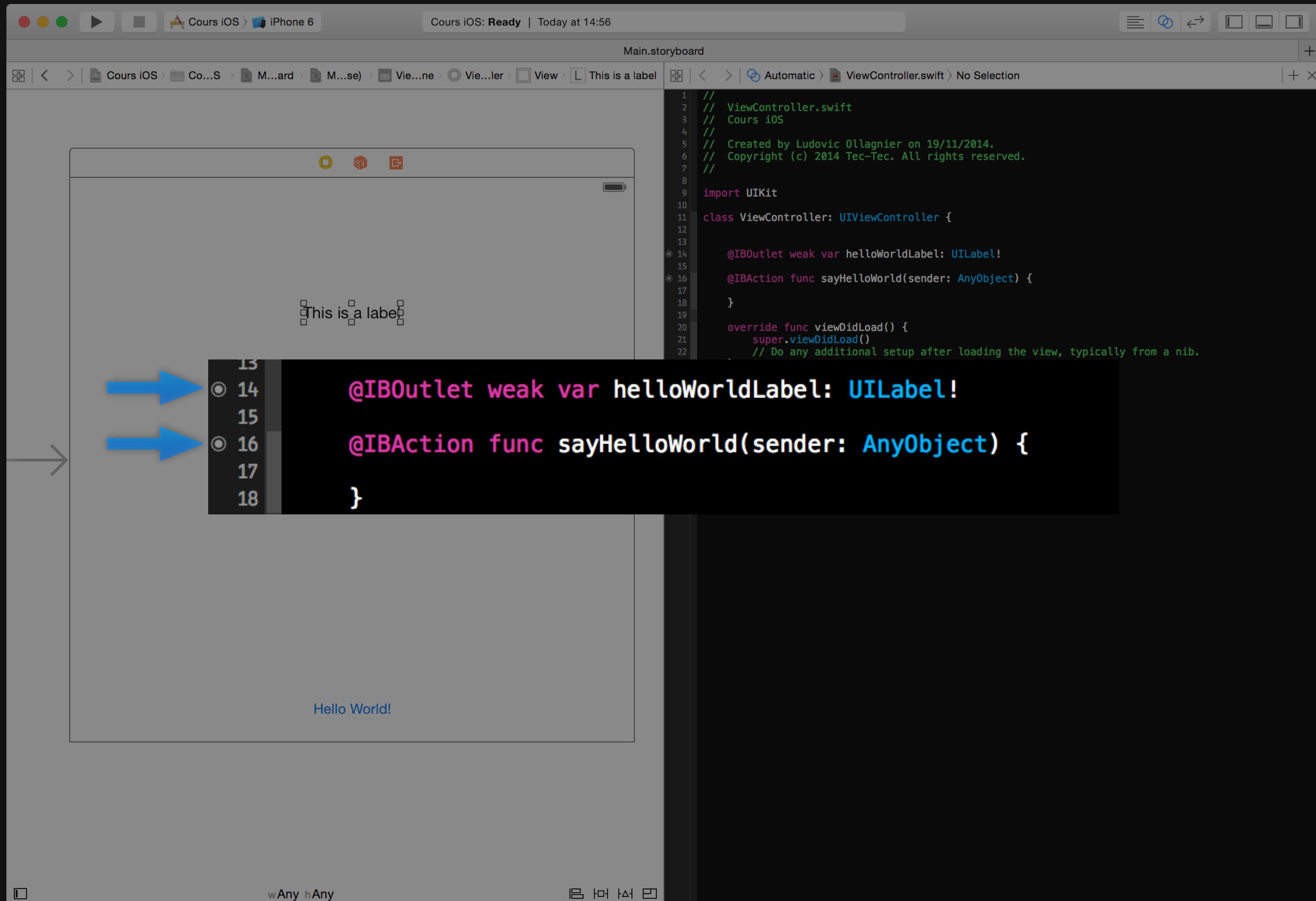
Du playground à l'application



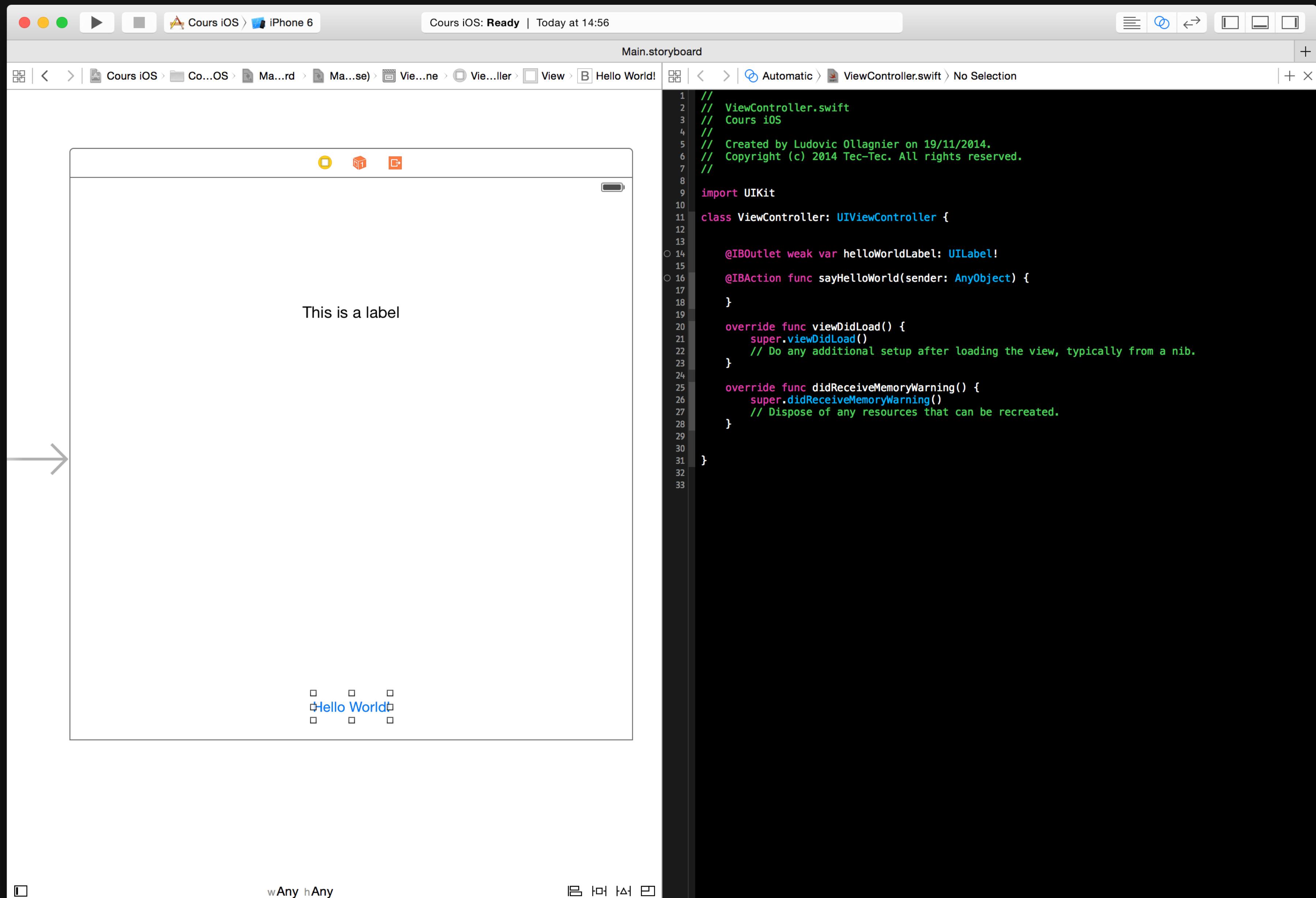
Du playground à l'application



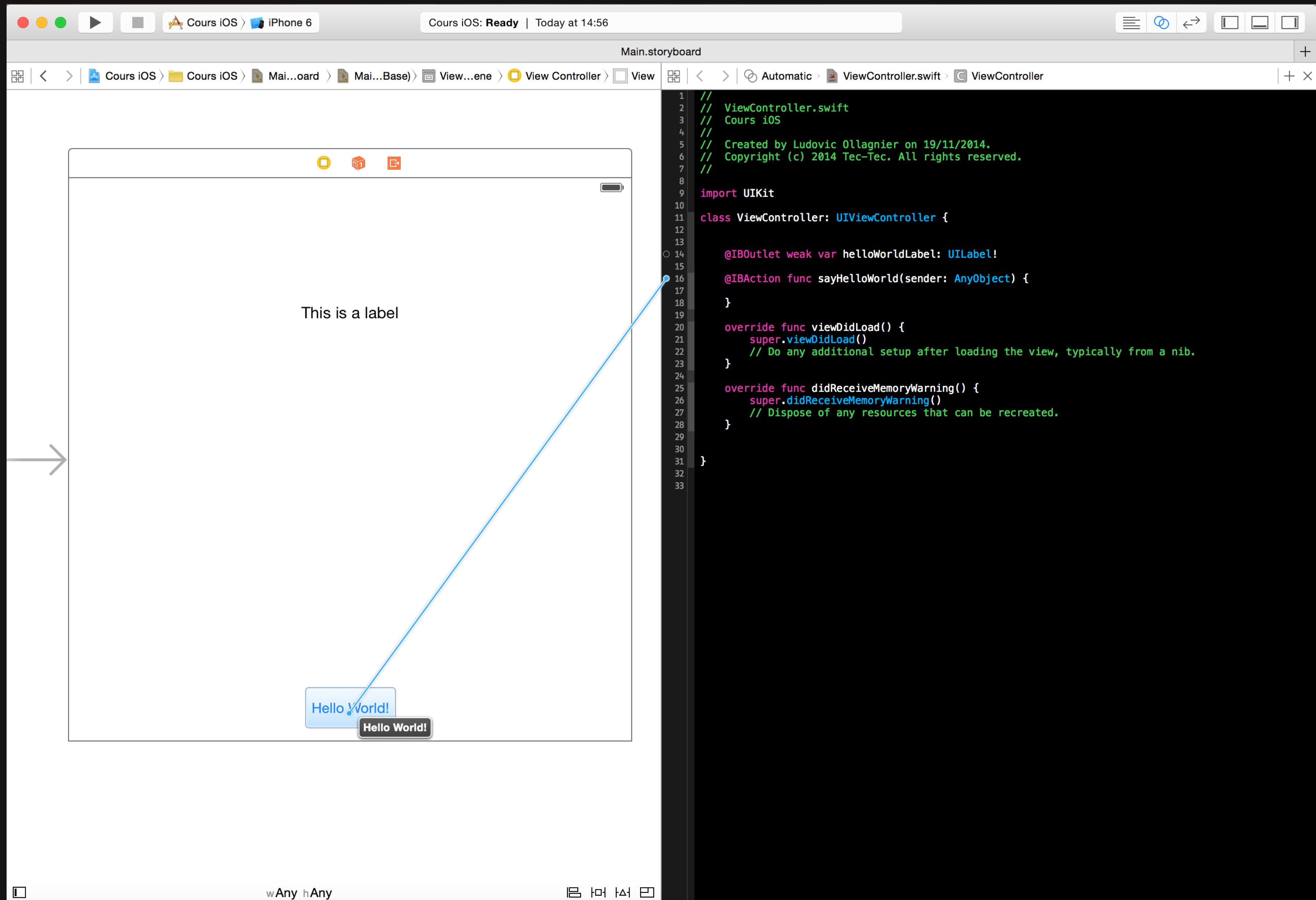
Du playground à l'application



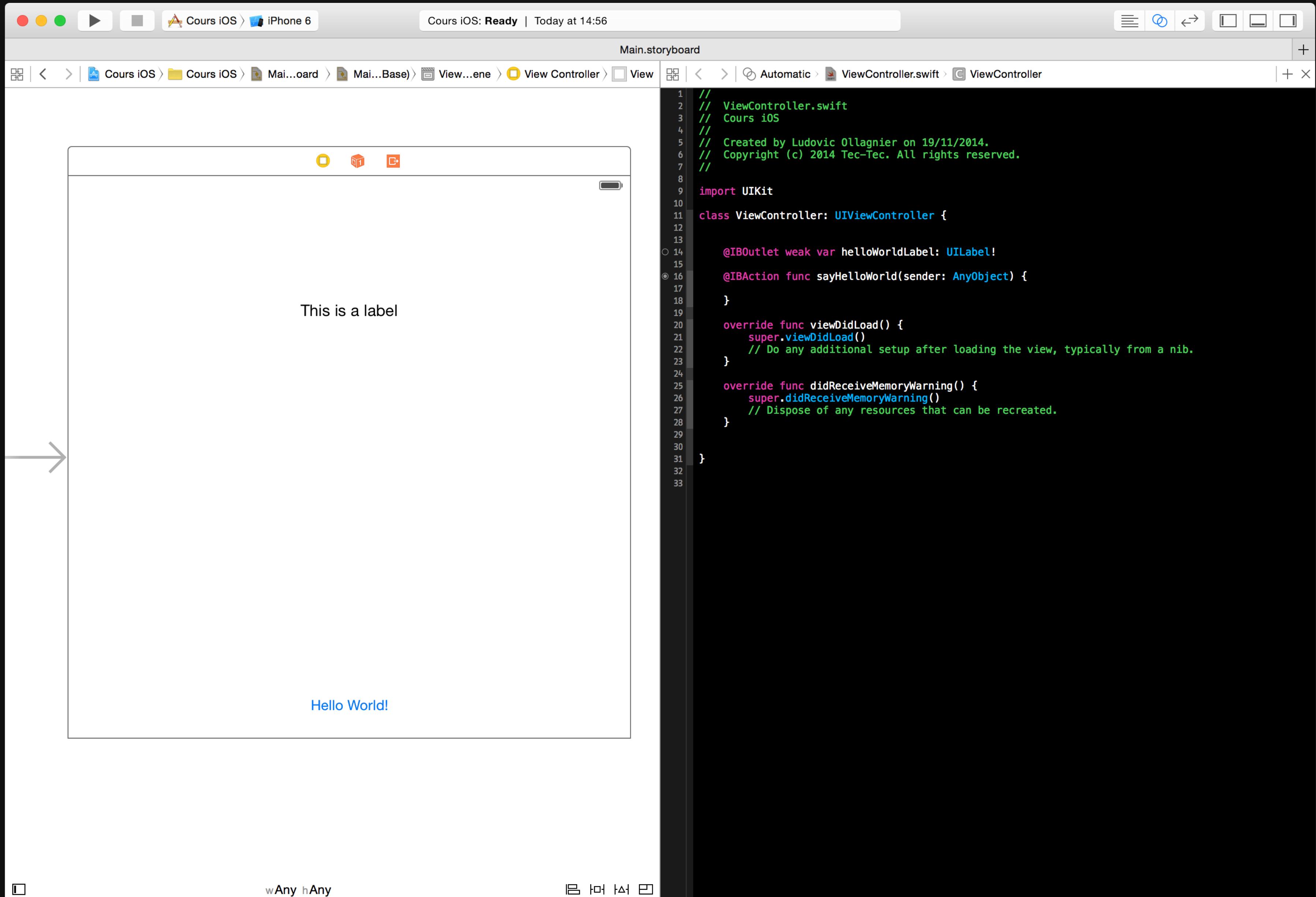
Du playground à l'application



Du playground à l'application



Du playground à l'application



The screenshot shows the Xcode interface with two main panes. The left pane displays the storyboard, and the right pane displays the corresponding Swift code.

Main.storyboard:

- The storyboard contains a single view controller with a label. The label has the text "This is a label".
- Below the storyboard, there is a large gray arrow pointing from left to right, indicating the transition from the storyboard to the code.
- The bottom of the storyboard pane shows the constraints: "wAny hAny" and "Top Space to Superview = 0".

ViewController.swift:

```
// ViewController.swift
// Cours iOS
//
// Created by Ludovic Ollagnier on 19/11/2014.
// Copyright (c) 2014 Tec-Tec. All rights reserved.

import UIKit

class ViewController: UIViewController {

    @IBOutlet weak var helloWorldLabel: UILabel!

    @IBAction func sayHelloWorld(sender: AnyObject) {
    }

    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view, typically from a nib.
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated.
    }
}
```

Pour aller plus loin...



- The Swift Programming Language : Classes and structures
- The Swift Programming Language : Initialization
- The Swift Programming Language : Deinitialization
- The Swift Programming Language : ARC