

Développement natif sous iOS

Cours n°5 - Gestion des données locales

iOS

Côté Swift



- Les *property observers*
- Les propriétés évaluées et *property wrappers*
- Les *lazy properties* (propriétés évaluées paresseusement)
- Les structures
- La classe `Optional`
- Le typage

Property observers

- Permettent d'intervenir avant ou après qu'une valeur soit modifiée
- Deux *observers* sont disponibles :
 - `willSet` : Avant que la nouvelle valeur ne soit définie
 - `didSet` : Une fois la valeur définie
- Dans les applications iOS, permettent de déclencher des actions lorsque des valeurs changent

Property observers



```
class Compte {  
    var solde : Int {  
        didSet(nouvelleValeur) {  
            self.transactionsEncours.pause()  
        }  
  
        didSet {  
            if solde < 0 {  
                this.aDecouvert = true  
            }  
  
            self.transactionsEncours.reprendre()  
        }  
    }  
}
```

Property observers



```
class Compte {  
    var solde : Int {  
        didSet {  
            if solde > oldValue {  
                print("Vous avez gagné des sous.")  
            } else if solde < oldValue {  
                print("Vous avez perdu des sous.")  
            }  
        }  
    }  
}
```

Propriétés évaluées et *property wrappers*



- Les propriétés évaluées sont des propriétés dont la valeur est déterminée dynamiquement
- Deux *wrappers* sont disponibles :
 - `get` : La block appelé lors de l'accès à la propriété
 - `set` : Le block appelé lors de la définition de la valeur
- Permettent de dynamiser les propriétés pour faire de la validation, déclencher des actions à l'instar des *property observers*, etc.

Propriétés évaluées et *property wrappers*



```
class CompteSansDecouvert {  
    private var _solde : Int = 0  
  
    var solde : Int {  
        get { return _solde }  
  
        set(nouvelleValeur) {  
            if nouvelleValeur >= 0 {  
                _solde = nouvelleValeur  
            }  
        }  
    }  
}
```

- Ici, solde est une propriété évaluée (*computed property*)
- Les blocks get et set sont les *property wrappers*

Lazy properties



- Normalement, le compilateur Swift requiert que toutes les propriétés soient initialisés lors de la création d'un objet
- Les *lazy properties* (ou propriétés évaluées paresseusement) permettent de calculer leur valeur qu'une fois celles-ci utilisées
 - Gain de temps si la propriété n'est jamais utilisée et que son calcul est coûteux
 - Une propriété paresseuse peut se baser sur la valeur d'une autre propriété au moment de son évaluation

Lazy properties



```
class Repository {  
    lazy var connection1 : DBConnection = DBConnection(url: "mysql://...")  
  
    lazy var connection2 : DBConnection = {  
        // Lecture des infos dans un fichier  
        // par exemple ...  
  
        return DBConnection(url: url)  
    }()  
}
```

Les structures



- Similaires aux classes
- Quelques différences :
 - Les structures sont utilisées par valeur ; les classes par référence
 - Pas d'héritage possible avec les structures donc pas de cast
 - Les structures ont un constructeur par défaut prenant en paramètre chacun de leurs attributs

Les structures



```
class Personne {  
    var nom : String  
  
    init(nom: String) {  
        self.nom = nom  
    }  
}  
  
var a = Personne(nom: "Paul")  
var b = a  
  
a.nom = "Jacques"  
  
a.nom // => "Jacques"  
b.nom // => "Jacques"
```

Les structures



```
struct Personne {  
    var nom : String  
  
    init(nom: String) {  
        self.nom = nom  
    }  
}
```

```
var a = Personne(nom: "Paul")  
var b = a
```

```
a.nom = "Jacques"
```

```
a.nom // => "Jacques"  
b.nom // => "Paul"
```

Les structures



```
struct Personne {  
    var nom : String  
}
```

```
var mouLoud = Personne(nom: "MouLoud")
```

La classe Optional

- Classe représentant une valeur n'étant pas forcément présente
- Peut-être vu comme une boîte
 - Le type du contenu est connu
 - La boîte peut être vide
- L'absence de valeur est représentée par la constante `nil` en Swift

La classe Optional

// Équivalents

```
var age1 : Int? = 42
```

```
var age2 : Optional<Int> = 42
```

// Il n'est pas obligatoire d'initialiser une valeur de fait

```
var nom : String?
```

La classe Optional

```
class Livre {  
    var couverture : String?  
}
```

```
let livre = Livre()  
let imageParDefaut = "default.png"
```

```
let photo = livre.couverture ?? imageParDefaut  
// => "default.png"
```

L'opérateur ?? permet d'utiliser la valeur se trouvant à sa droite si un optionnel (à sa gauche) contient nil

La classe `Optional`

```
// Chaînage optionnel
if livre.couverture?.hasSuffix(".png") == true {
    print("Image PNG")
}
```

- Permet d'accéder de manière sûre (sans lever d'erreur à l'exécution) aux propriétés d'un `Optional`.
- L'inconvénient est que le type retourné est un optionnel également :
 - La méthode `hasSuffix` renvoie normalement un `Bool`
 - D'appeler `hasSuffix` en chaînage optionnel renvoie un `Bool?`

La classe Optional

```
// Chaînage sans condition
if livre.couverture!.hasSuffix("png") {
    print("Image PNG")
}
```

Le chaînage sans condition lève une exception au moment de l'exécution si l'optionnel contient nil

La classe Optional

```
var livre = Livre()  
  
// Lève une exception  
print(livre.couverture!)  
  
// OK  
livre.couverture = "le-horla.png"  
print(livre.couverture!)
```

La classe Optional

```
class CompteSansDecouvert {  
    var solde : Int  
  
    init?(solde: Int) {  
        guard solde > 0 else { return nil }  
  
        self.solde = solde  
    }  
}
```

```
let compte = CompteSansDecouvert(solde: -10)  
// => nil
```

Le typage



```
class Document {  
    var titre: String  
  
    init(titre: String) {  
        self.titre = titre  
    }  
}
```

```
class Livre : Document {  
    var couverture: String  
  
    init(titre: String, couverture: String) {  
        self.couverture = couverture  
  
        super.init(titre: titre)  
    }  
}
```

Le typage



- Quelques définitions :
 - *Downcast* : action de convertir d'un type parent vers un type enfant (e.g. de Document vers Livre)
 - *Upcast* : action de convertir vers un type parent
 - *Conditional cast* : conversion renvoyant nil si l'opération a échoué
 - *Forced cast* : conversion levant une erreur si l'opération a échoué

Le typage



```
let documents = [  
    Document(titre: "Notes de lecture"),  
    Livre(titre: "Le Horla", couverture: "le-horla.png")  
]
```

// Conditional cast

```
let livre1 = documents[0] as? Livre
```

// Forced cast

```
let livre2 = documents[1] as! Livre
```

// Upcast

```
let document = livre2 as Document
```

Le typage



```
for document in documents {  
    if document is Livre {  
        print("Livre")  
    } else {  
        print("Simple document")  
    }  
}
```


Côté iOS

- Affichage des données :
 - UITableViewController
 - UICollectionViewController
- Gestion des données :
 - Préférences
 - Stockage sous forme de fichiers
 - Bases de données

Table View Controller

- Contrôleur permettant de gérer une `UITableView`, utilisée pour afficher un tableau de données
- Une « Table View » contient des « Table View Cells »
- iOS propose des modèles pré-existant pour définir le patron de nos cellules
- Il est possible de créer ses propres modèles

Table View Controller

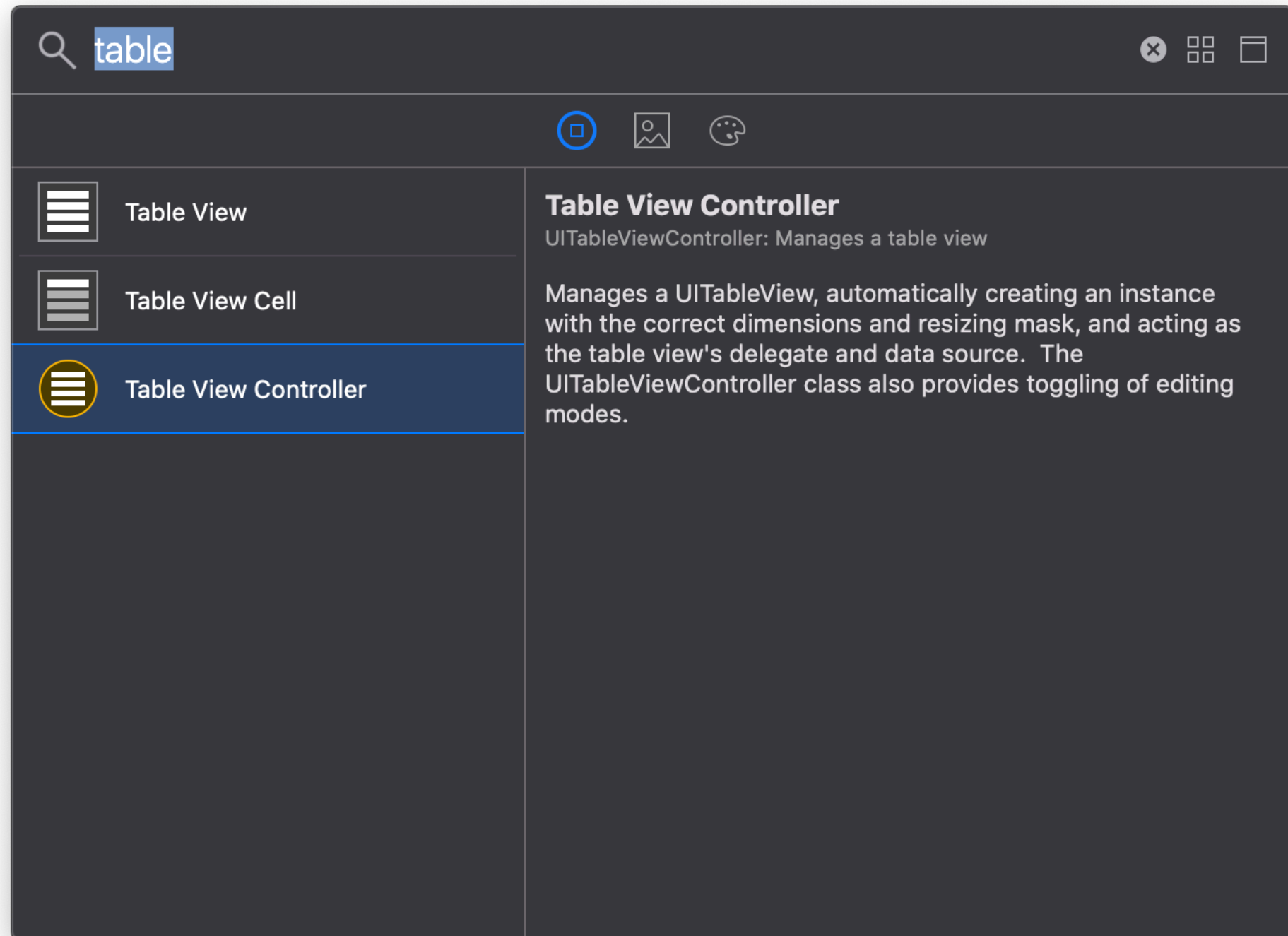


Table View Controller

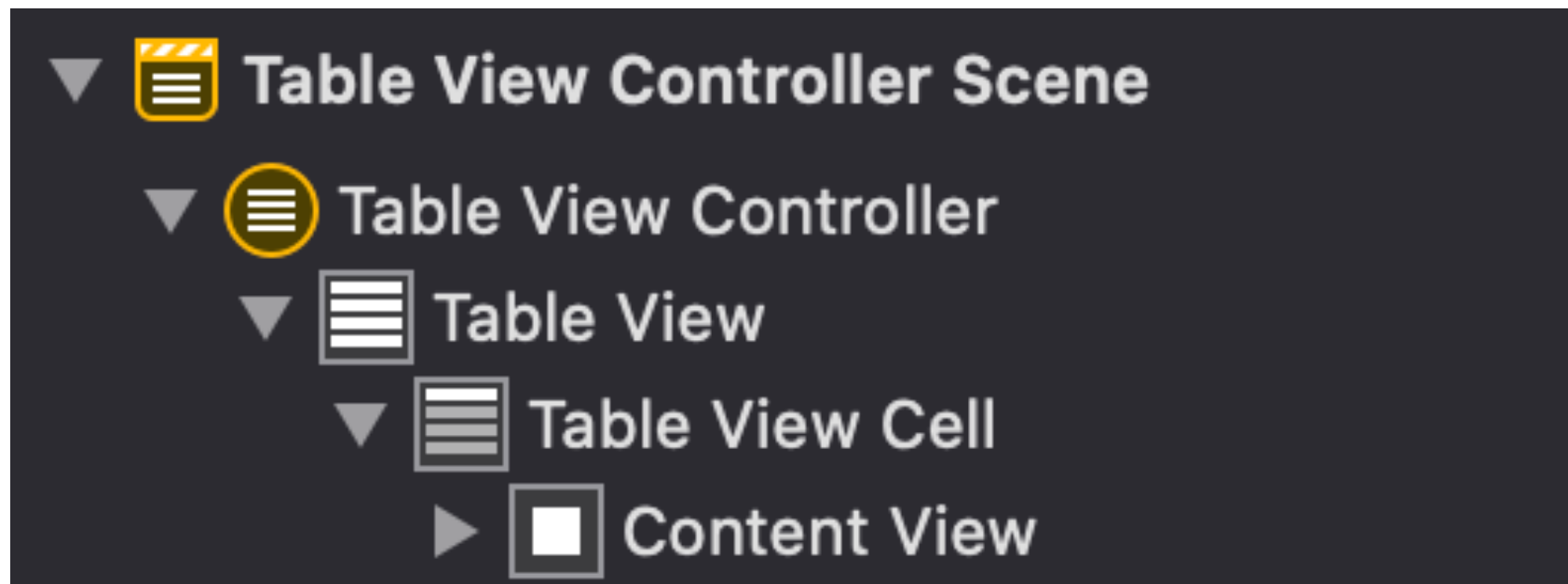
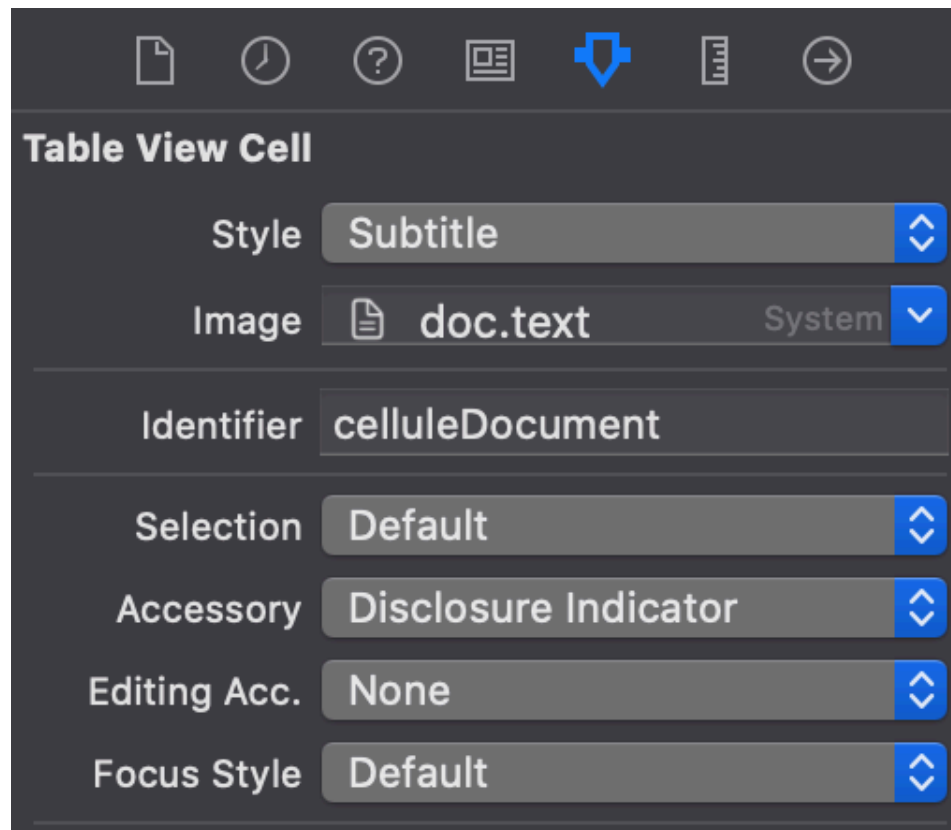
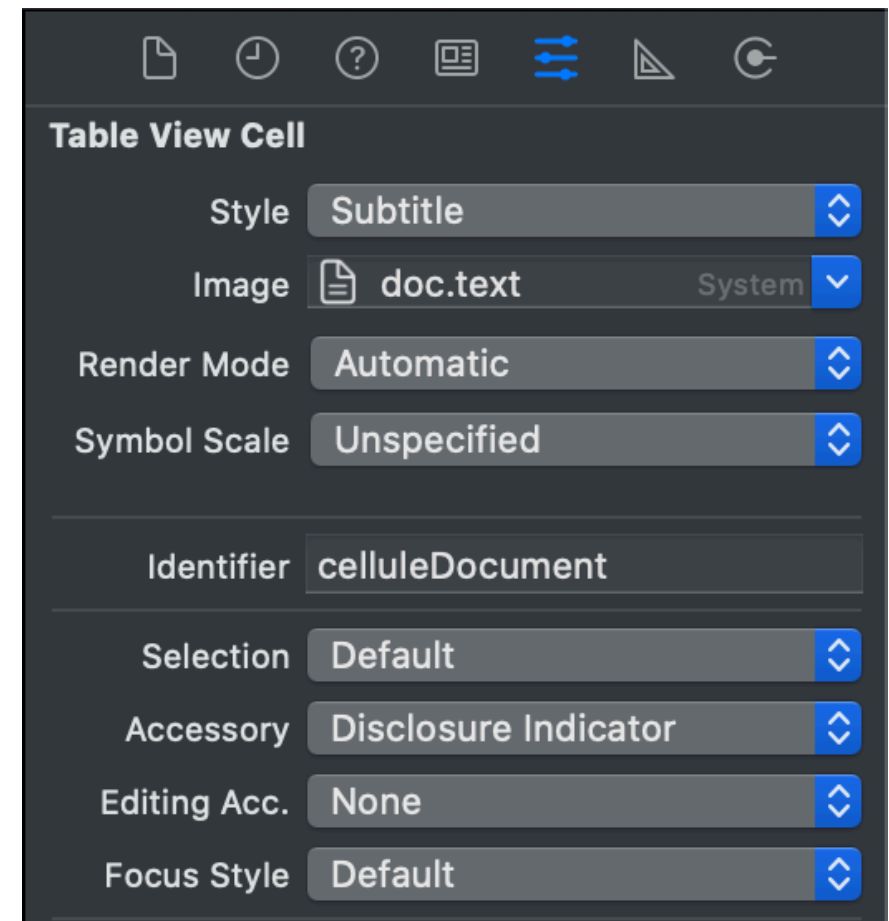


Table View Controller



Xcode 11

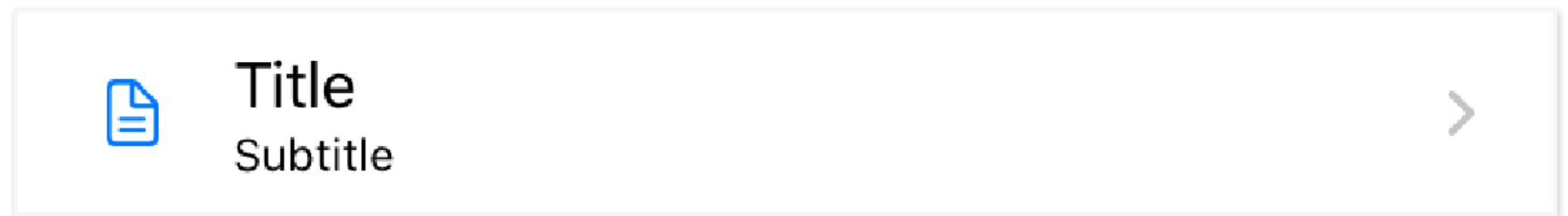


Xcode 12

En choisissant la « Table View Cell » dans le menu gauche et dans l'inspecteur d'attributs, il est possible de personnaliser le modèle de chaque cellule.

L'identifier permet de récupérer le modèle de la cellule côté code.

Table View Controller

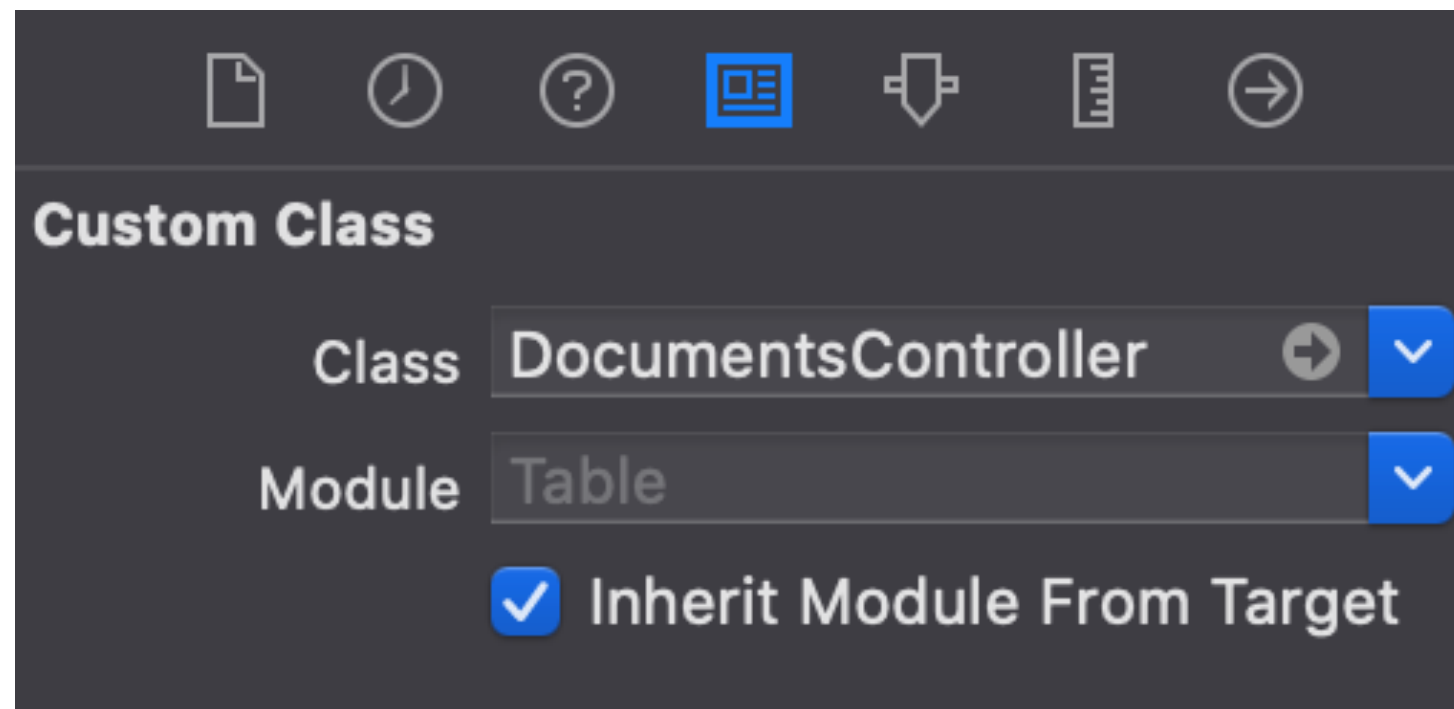


Image

Style

Accessory

Table View Controller



Il faut ensuite connecter le contrôleur côté storyboard à une classe du projet pour pouvoir l'alimenter en données

Table View Controller

```
struct Document {  
    var title : String  
    var nbOfPages : Int  
}
```


Table View Controller

```
class DocumentsController : UITableViewController {  
    private var documents = [  
        Document(title: "Rapport de stage", nbOfPages: 16),  
        Document(title: "Cours de maths", nbOfPages: 4)  
    ]  
  
    // ...  
}
```

Table View Controller

```
// ...
```

```
    override func tableView(_ tableView: UITableView, numberOfRowsInSectionSection: Int) -> Int {  
        return documents.count  
    }
```

```
// ...
```

Table View Controller

```
// ...
```

```
    override func tableView(_ tableView: UITableView, cellForRowAt  
indexPath: IndexPath) -> UITableViewCell {
```

```
        let cellule = tableView.dequeueReusableCell(  
            withIdentifier: "celluleDocument",  
            for: indexPath)
```

```
        let document = documents[indexPath.row]
```

```
        cellule.textLabel?.text = document.title
```

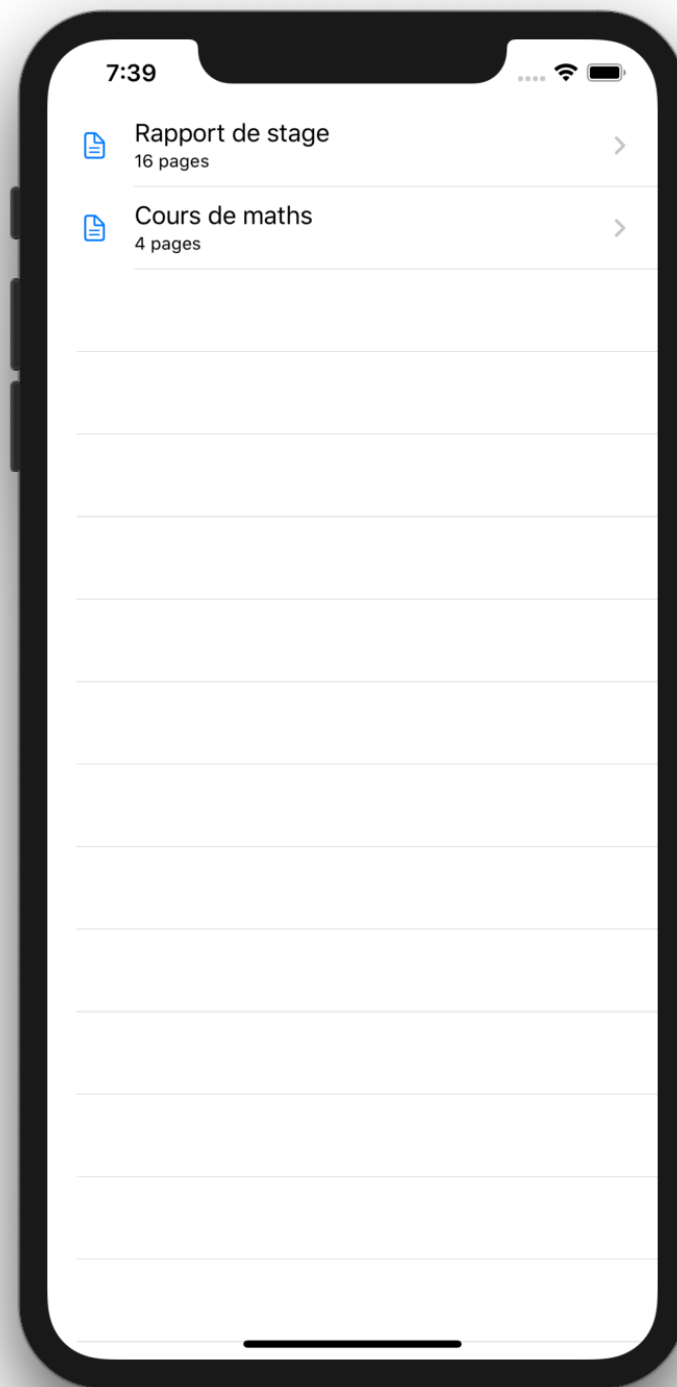
```
        cellule.detailTextLabel?.text = "\\(document.nbOfPages) pages"
```

```
        return cellule
```

```
    }
```

```
}
```

Table View Controller

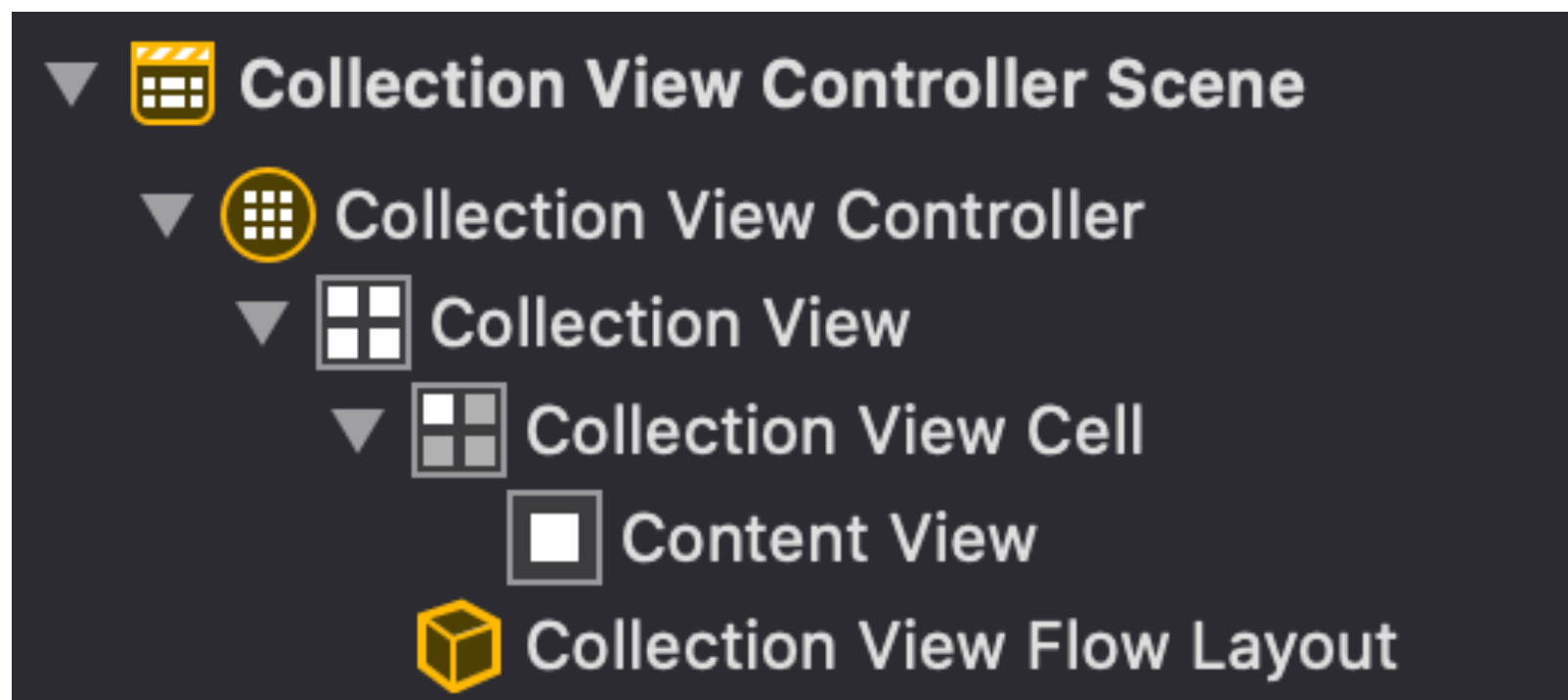


iPhone 11 Pro Max — 13.3

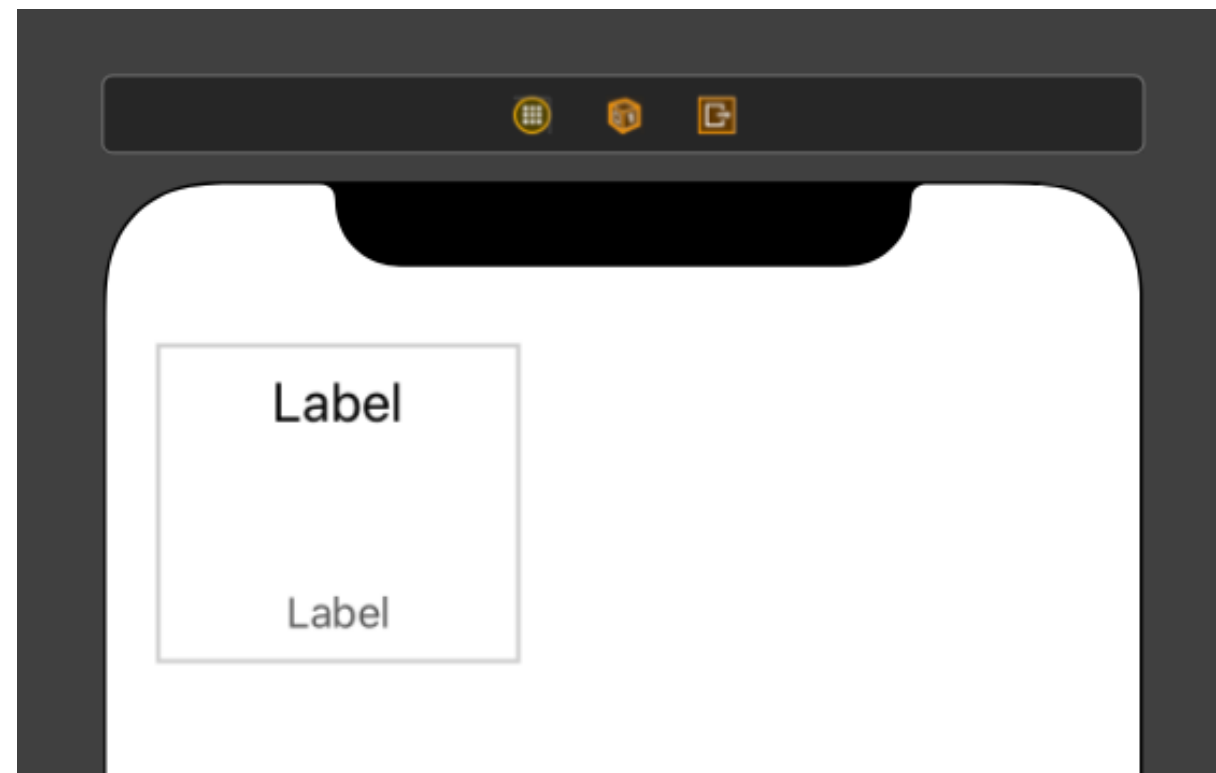
Collection View Controller

- Contrôleur permettant de gérer une UICollectionView qui va représenter chaque élément d'une collection
- Utile pour des affichages en grille des données par exemple
- Concept très similaire à UITableViewController sauf qu'ici :
 - On a une « Collection View » qui contient des « Collection View Cell »
 - Il n'y a pas de modèle pré-défini
 - Il est possible de définir la disposition des éléments de manière avancée

Collection View Controller

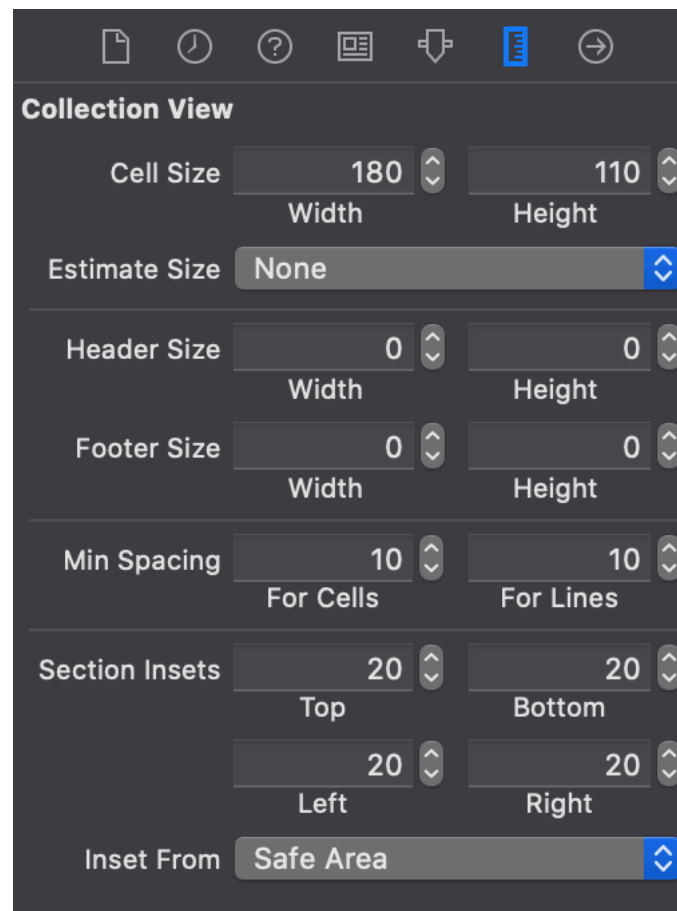


Collection View Controller



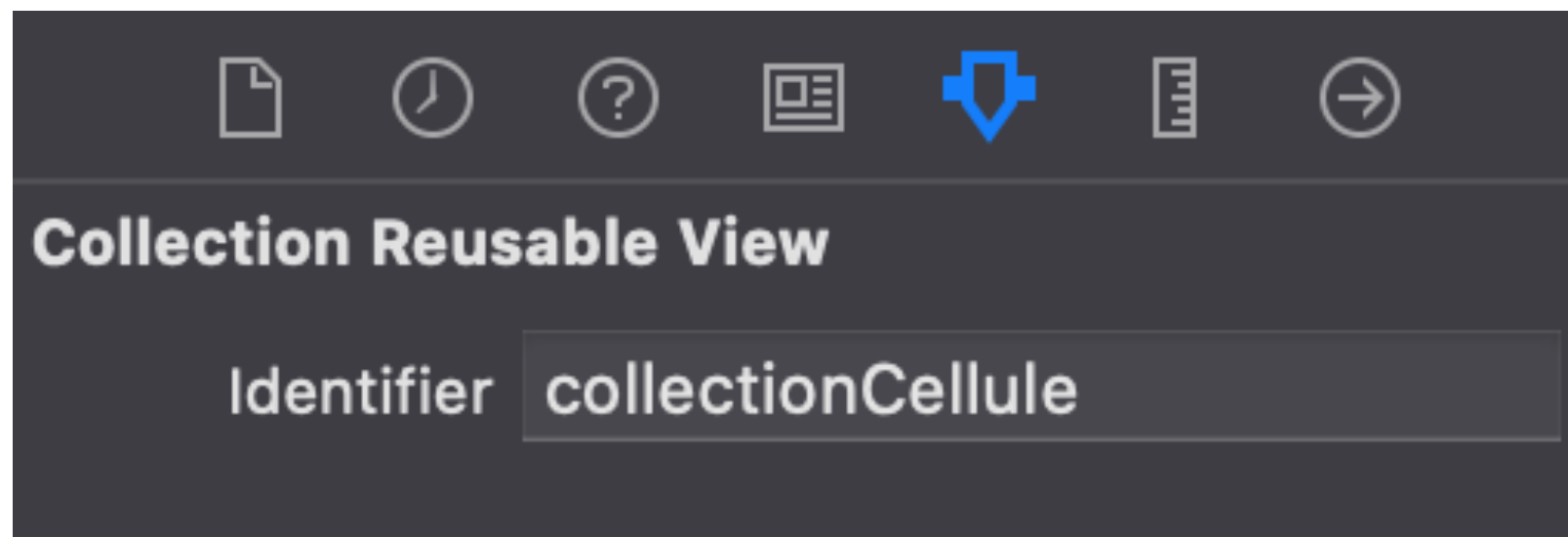
Il est possible de glisser-déposer des éléments dans la vue modèle, leur ajouter des contraintes, etc.

Collection View Controller



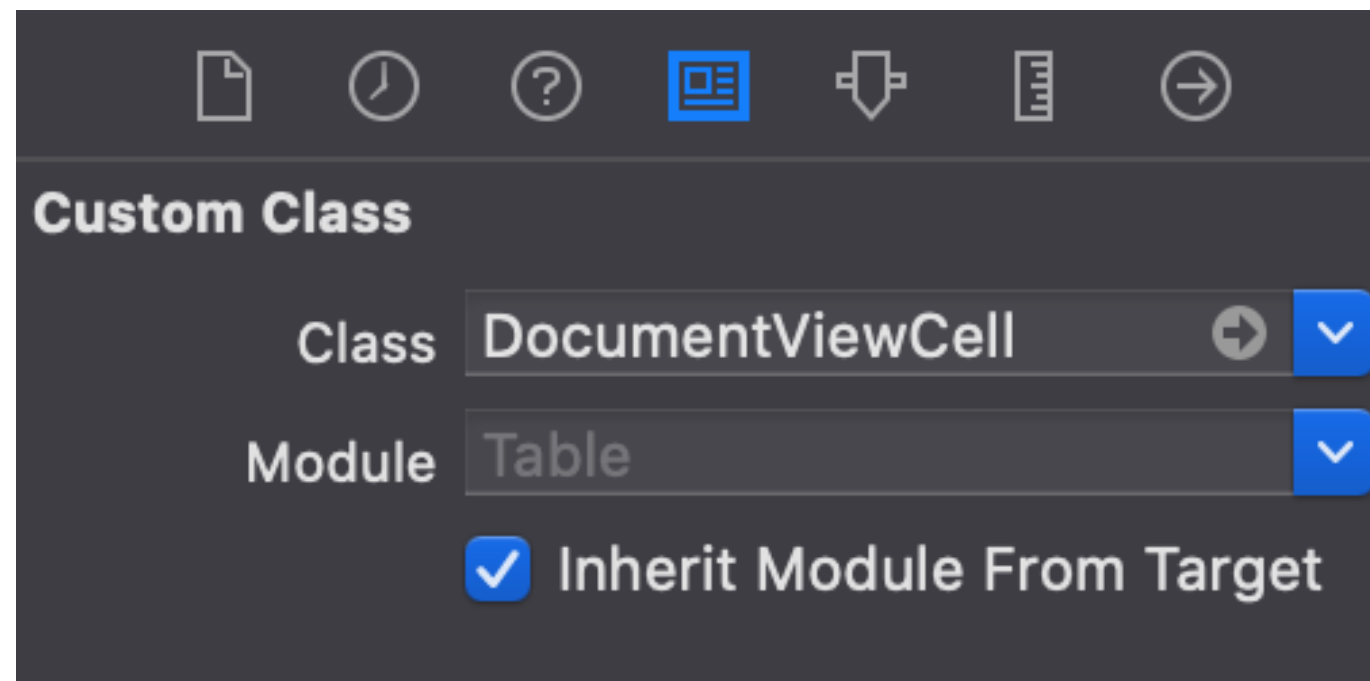
Depuis l'inspecteur de taille de la « Collection View », il est possible de définir les paramètres liés à la taille et l'espacement des éléments de la collection

Collection View Controller



Comme pour les tableaux, associer un identifiant à la « Collection View Cell »

Collection View Controller



Il faut également rattacher la « Collection View Cell » à une classe personnalisée

CollectionView Controller

```
class DocumentViewCell: UICollectionViewCell {  
  
    @IBOutlet var titleLabel: UILabel!  
    @IBOutlet var nbOfPagesLabel: UILabel!  
  
    required init?(coder: NSCoder) {  
        super.init(coder: coder)  
  
        self.contentView.layer.borderWidth = 1  
        self.contentView.layer.borderColor = UIColor.systemGray4.cgColor  
        self.contentView.layer.cornerRadius = 5  
    }  
}
```

Collection View Controller

```
class DocumentsController : UINavigationController {  
    private var documents = [...]  
  
    // ...  
}
```

Collection View Controller

```
// ...
```

```
    override func collectionView(_ collectionView: UICollectionView,  
numberOfItemsInSection section: Int) -> Int {  
        return documents.count  
    }
```

```
// ...
```

CollectionView Controller

```
// ...

override func collectionView(_ collectionView: UICollectionView,
cellForItemAt indexPath: IndexPath) -> UICollectionViewCell {

    let cell = collectionView.dequeueReusableCell(withReuseIdentifier:
"collectionCellule", for: indexPath) as! DocumentViewCell

    let document = documents[indexPath.row]

    cell.titleLabel.text = document.title
    cell.nbOfPagesLabel.text = "\((document.nbOfPages) pages"

    return cell
}
}
```

Collection View Controller



Gestion des données en local

- Préférences :
 - Préservation des états de l'application
 - Stockage de préférences simples (e.g. thème, volume, etc.)
- Stockage de fichiers :
 - Fichiers structurés (Plist, XML, JSON)
 - Fichiers encodés (binaires)
- Bases de données :
 - Stockage dans une base SQLite 3
 - Core Data
 - Dépendances externes (e.g. Realm)

Préservation des états de l'application

- La préservation peut s'appliquer aux :
 - Contrôleurs
 - « Table views » et « collection views »
 - Scroll views
 - Champs textes
 - Image views

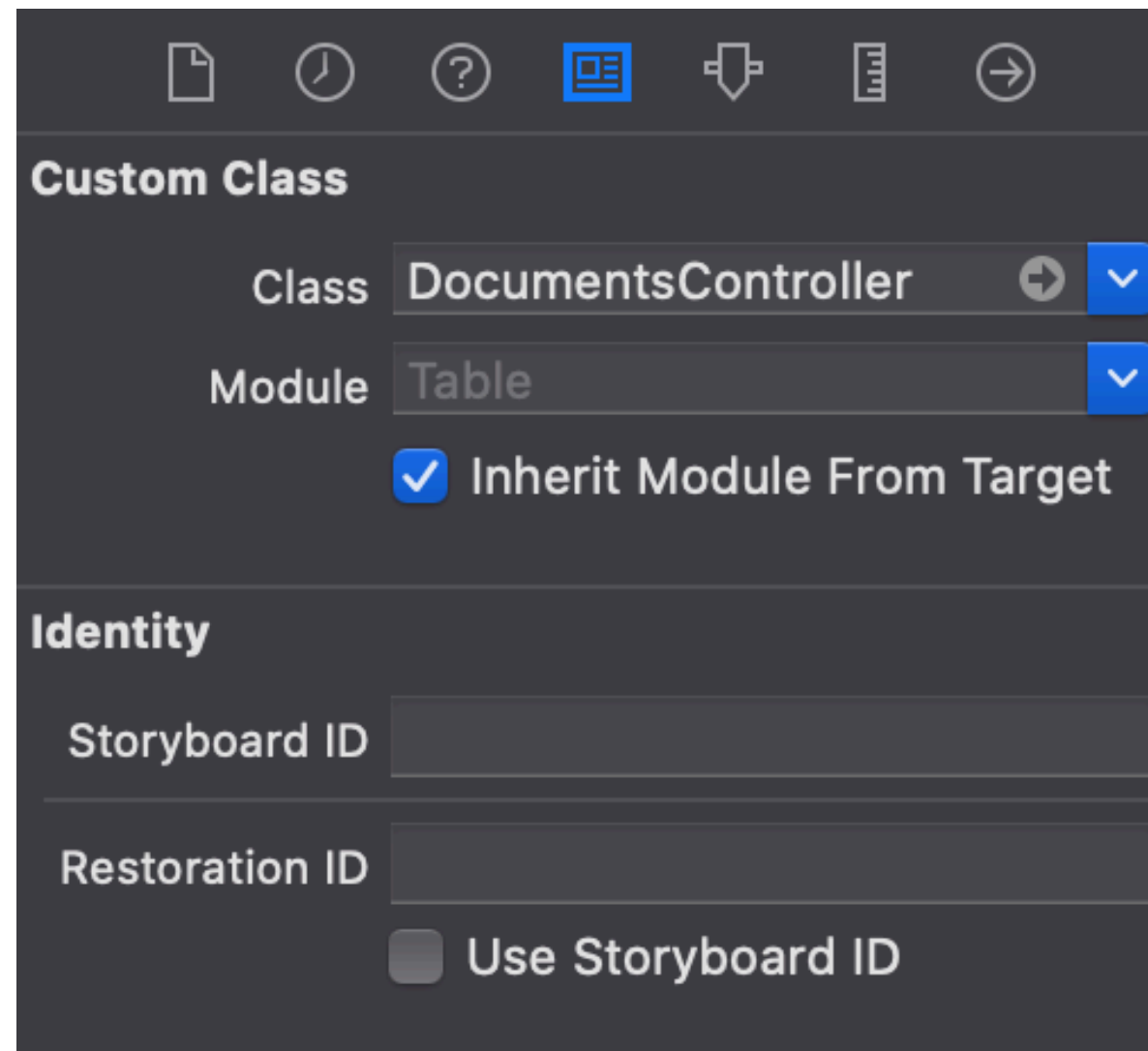
Préservation des états de l'application

```
@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {
    // ...

    func application(_ application: UIApplication,
        shouldSaveApplicationState coder: NSCoder) -> Bool {
        return true
    }

    func application(_ application: UIApplication,
        shouldRestoreApplicationState coder: NSCoder) -> Bool {
        return true
    }
}
```

Préservation des états de l'application



The image shows a screenshot of the Xcode interface, specifically the 'Custom Class' and 'Identity' sections of the 'Attributes Inspector'.

Custom Class

- Class:** DocumentsController
- Module:** Table
- ☒ Inherit Module From Target

Identity

- Storyboard ID:** (empty field)
- Restoration ID:** (empty field)
- ☐ Use Storyboard ID

Spécifier une valeur de « Restoration ID » permet d'indiquer que l'état de cet élément doit être restauré lorsque l'application est ré-ouverte après avoir été mise en pause.

Préférences simples

- La classe `UserDefaults` peut être utilisée pour stocker des valeurs discrètes
- Types supportés : `Bool`, `Int`, `Float`, `String`, `URL`, `Data` et les dictionnaires et tableaux contenant ces types

Préférences simples

Pour écrire une valeur :

```
UserDefaults.standard.set(0.9, forKey: "volume")
```

Pour la récupérer ensuite :

```
UserDefaults.standard.double(forKey: "volume")
```

Stockage de fichiers

- Chaque application possède une *sandbox* (espace pour ses fichiers)
- Pas d'échanges de fichiers entre applications

Dossier	Description
[Votre app].app	Bundle de votre application (exécutable et autres ressources)
Documents/	Fichiers que l'on souhaite exposer à l'utilisateur
Library/	Fichiers que l'on ne souhaite pas exposer à l'utilisateur (cache, etc.)
tmp/	Dossier permettant de stocker les fichiers temporaires

Stockage de fichiers

- Les appareils Apple disposent du système de fichiers AFS (Apple File System)
- Certaines spécificités :
 - Un clone de fichier ne requiert pas d'espace disque ; seul les nouveaux octets de la copie prennent de la place
 - Les fichiers creux n'allouent pas de blocs vides
- https://developer.apple.com/documentation/foundation/file_system/about_apple_file_system

Stockage de fichiers structurés

- Plusieurs formats possibles :
 - Property list (similaire à XML)
 - JSON
- Des classes existent dans le SDK pour la gestion de fichiers :
 - FileManager
 - URL
- D'autres sont utiles pour encoder / décoder :
 - NSArray et NSDictionary (Property list)
 - JSONSerialization

Stockage de fichiers structurés

(Property list)

```
<array>  
  <dict>  
    <key>title</key>  
    <string>Rapport de stage</string>  
    <key>nbOfPages</key>  
    <integer>16</integer>  
  </dict>  
  
  <!-- ... -->  
</array>
```

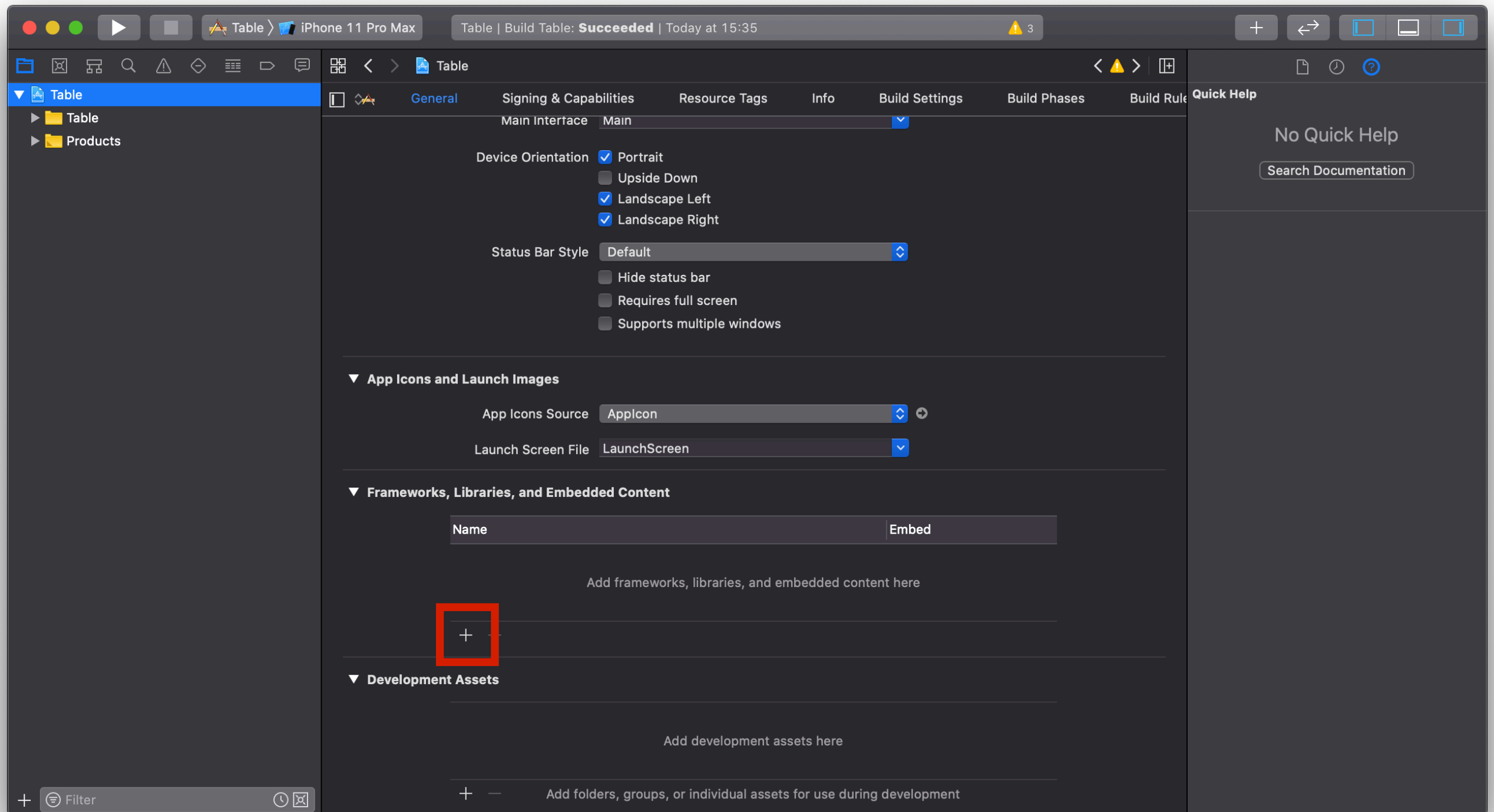
Stockage de fichiers encodés

- Mêmes principes que pour les fichiers structurés mais les données sont stockées en binaire
- Les objets à enregistrer doivent suivre le protocole Codable
- D'autres classes interviennent cependant :
 - NSCoder / NSCoder
 - PropertyListEncoder / PropertyListDecoder
 - JSONEncoder / JSONDecoder

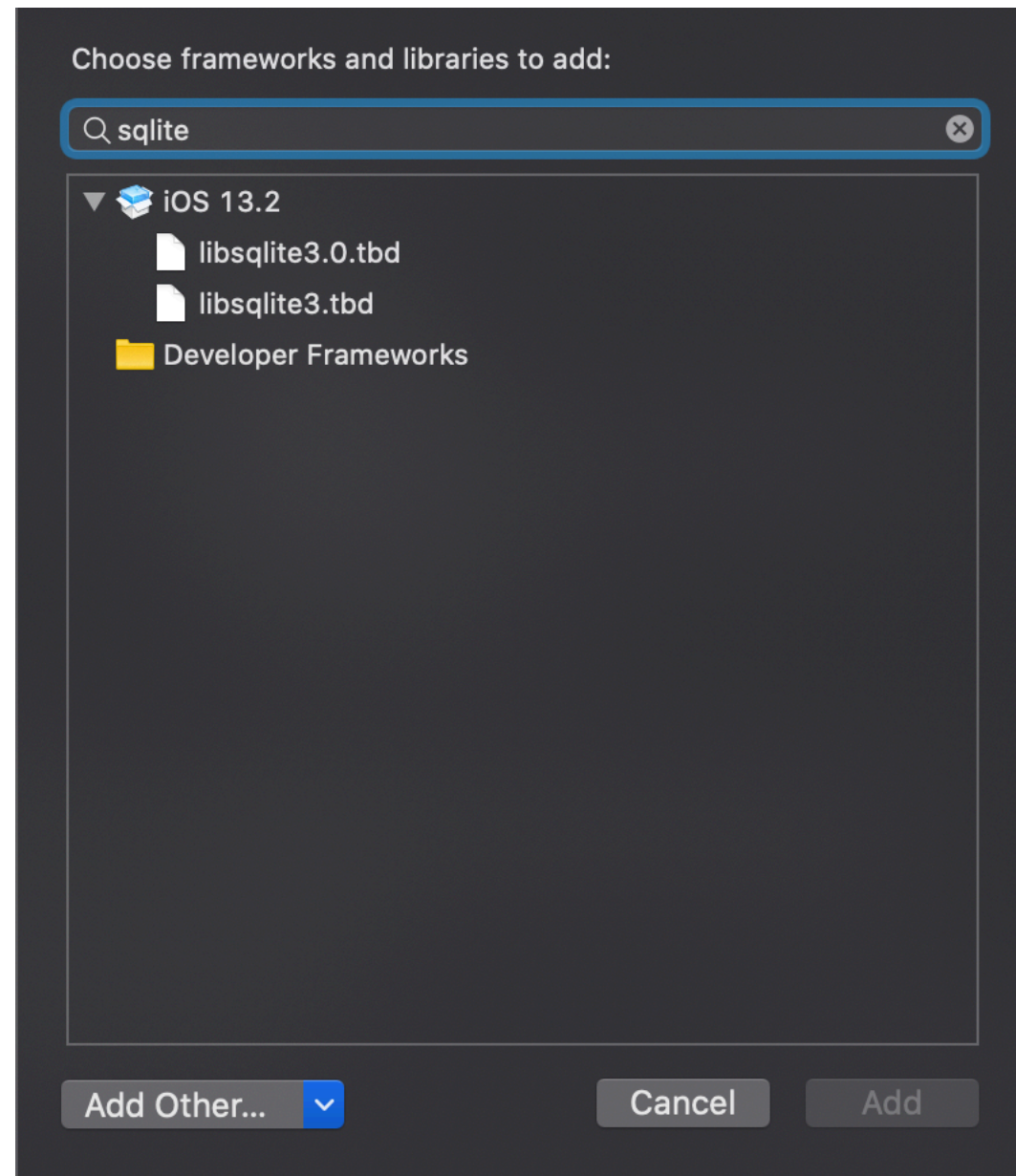
Bases SQLite 3

- Peut-être pratique pour des données complexes, avec relations, etc.
- Les bases SQLite sont stockées dans des fichiers
- Très largement supporté (par défaut sous Android également)
- Les bases peuvent être créées de 2 façons différentes :
 - Dynamiquement, via le code
 - En incluant un fichier de base SQLite existant dans le projet Xcode

Bases SQLite 3



Bases SQLite 3



libsqlite3 pointe vers la dernière version ; si une mise à jour inclue libsqlite3.1, libsqlite3 pointera vers ce dernier.

Bases SQLite 3

- Par défaut, il n'y a pas d'API pour interagir directement avec SQLite 3
- Il faut donc passer par des dépendances externes :
 - <https://github.com/ccgus/fmdb>
 - <https://github.com/stephencelis/SQLite.swift>
 - <https://github.com/groue/GRDB.swift>

Core Data

- ORM présent dans le SDK iOS par défaut
- Extrêmement puissant :
 - Complètement intégré dans Xcode
 - Gestion de relations entre entités
 - Permet de tracer les changements et gérer la validation sur un objet
 - Gestion de cache
- Peut être utilisé en complément d'un web service pour stocker les données localement et ne les envoyer que périodiquement pour réduire la charge sur le serveur

Core Data

(Dans une nouvelle application)

- Cocher « Use Core Data »
- Un fichier ayant l'extension `xcdatamodeld` sera créé
- Du code lié à la gestion des entités Core Data sera ajouté à la classe `AppDelegate`

Core Data

(Dans une nouvelle application)

```
// Dans AppDelegate
lazy var persistentContainer: NSPersistentContainer = {
    let container = NSPersistentContainer(name: "Data")

    container.loadPersistentStores(completionHandler: { (description, error) in
        if let error = error as NSError? {
            fatalError("Unresolved error \(error), \(error.userInfo)")
        }
    })
}()

return container
}()
```

Plusieurs objets interviennent dans l'utilisation de Core Data.

Le « persistent container » permet de tous les gérer simplement.

Core Data

(Dans une nouvelle application)

```
// ...  
func saveContext () {  
    let context = persistentContainer.viewContext  
    if context.hasChanges {  
        do {  
            try context.save()  
        } catch {  
            let nerror = error as NSError  
            fatalError("Unresolved error \(nerror), \(nerror.userInfo)")  
        }  
    }  
}
```

Toutes les modifications effectuées sur les entités sont conservées dans un contexte.

Il est possible d'enregistrer toutes les modifications depuis le dernière enregistrement grâce à cette méthode

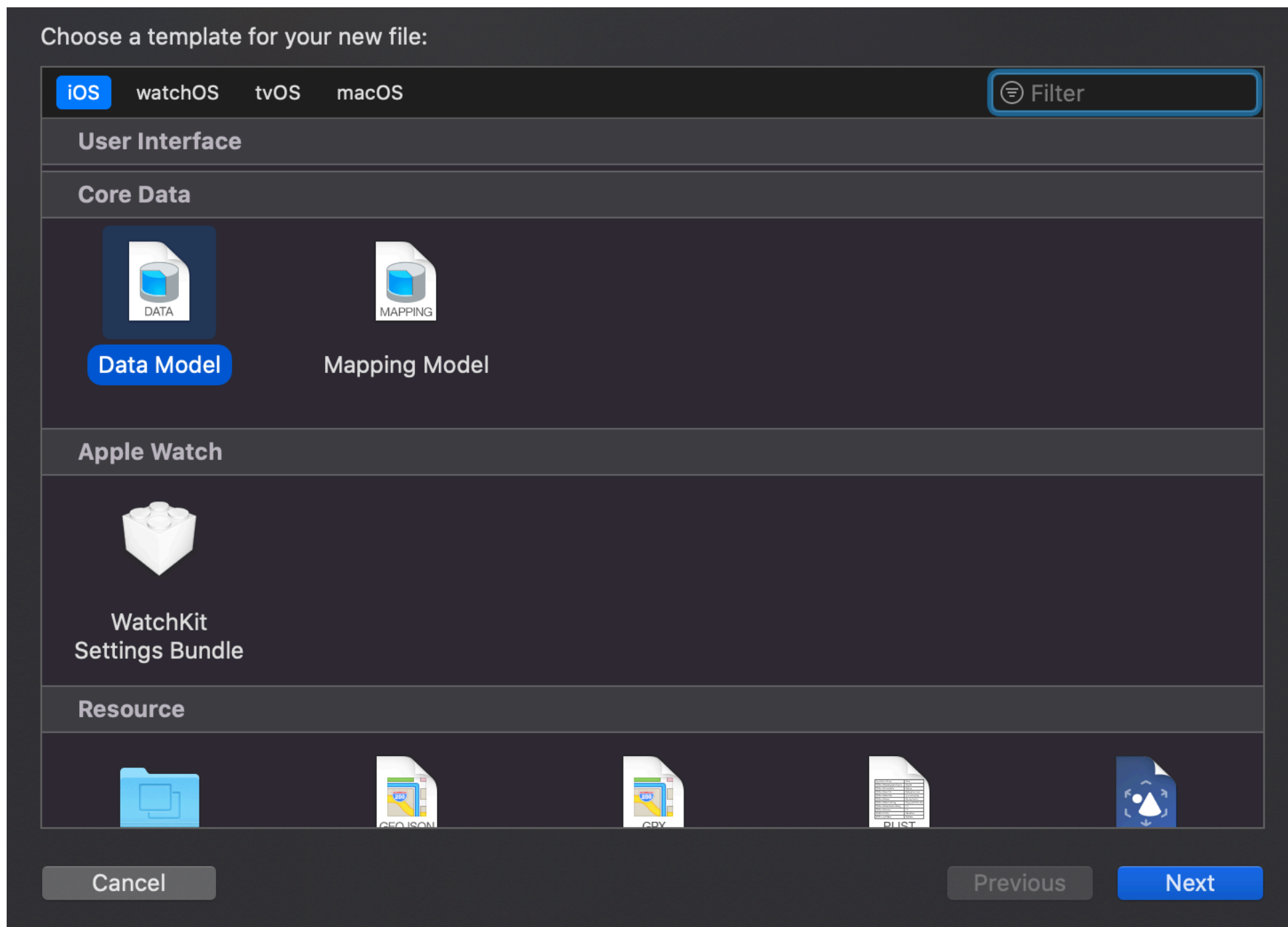
Core Data

(Dans une application existante)

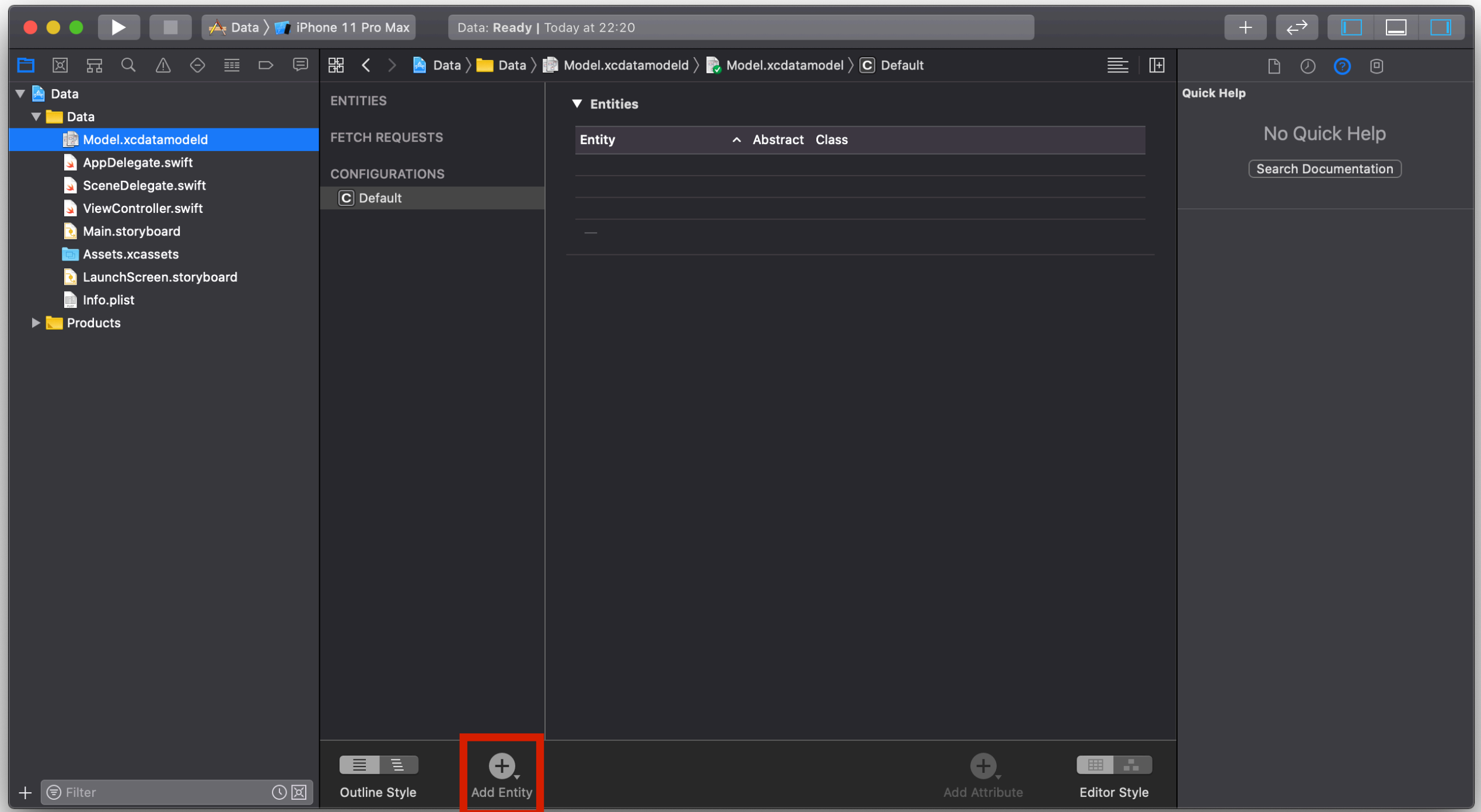
- Il faudra ajouter le code permettant d'initialiser le « persistance container » et de sauvegarder le contexte (2 diapositives précédentes)
- Créer le fichier modélisant les données de l'application

Core Data

(Dans une application existante)



Core Data



Core Data

ENTITIES

E

Category

E

Document

FETCH REQUESTS

CONFIGURATIONS

C

Default

▼ Attributes

Attribute	^	Type	
<div>N</div> nbOfPages		Integer 16	⌵
<div>S</div> title		String	⌵

+ -

▼ Relationships

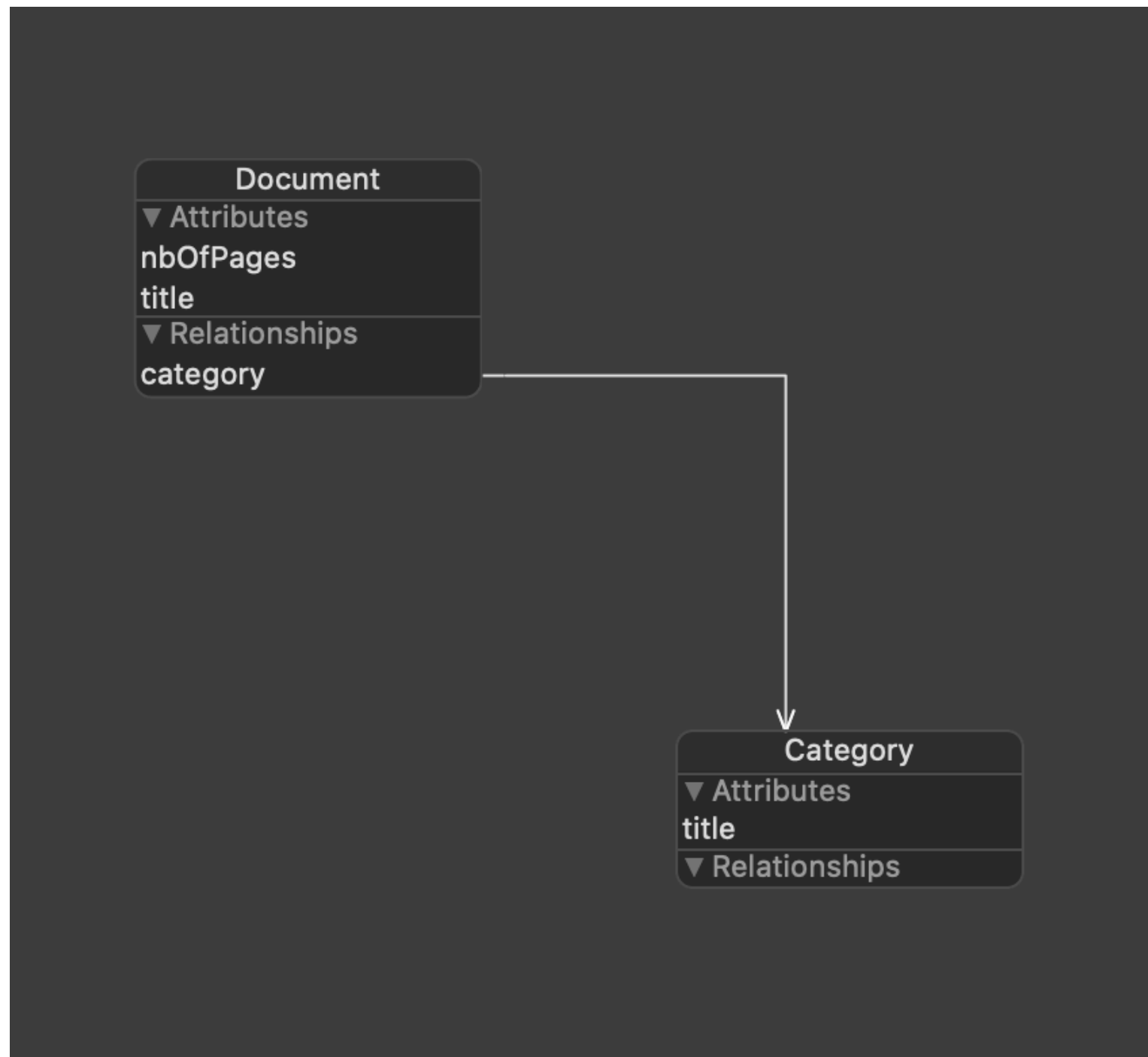
Relationship	^	Destination	Inverse
<div>O</div> category		Category	⌵ No Inverse ⌵

+ -

Core Data



Core Data



Core Data

- Ensuite, côté code :
 - Core Data va générer les classes liées à la définition faite des entités
 - Des classes existent dans le SDK pour récupérer les entités :
 - `NSFetchRequest`
 - `NSFetchedResultsController`

Core Data

(Sauvegarde d'entités)

```
class ViewController: UIViewController {  
    lazy var appDelegate = UIApplication.shared.delegate as! AppDelegate  
    lazy var context = {  
        appDelegate.persistentContainer.viewContext  
    }()  
  
    @IBAction func clickBouton() {  
        let category = Category(context: context)  
        let document = Document(context: context)  
  
        category.title = "Cours"  
  
        document.title = "Rapport de stage"  
        document.numberOfPages = 4  
        document.category = category  
  
        appDelegate.saveContext()  
    }  
}
```

Core Data

(Récupération d'entités)

```
lazy var fetchDocuments : NSFetchedResultsController<Document> =  
    self.fetchDocumentsController()  
  
func fetchDocumentsController() -> NSFetchedResultsController<Document> {  
    // ...  
}
```

Dans un premier temps, on définit un contrôleur qui va nous permettre de récupérer un certain ensemble d'entités.

Le contrôleur est dépendant de la requête ; il en faudra un par requête que l'on souhaite réaliser.

Core Data

(Récupération d'entités)

```
func fetchDocumentsController() -> NSFetchedResultsController<Document> {  
    let request : NSFetchRequest<Document> = Document.fetchRequest()  
  
    // Taille de l'ensemble (équivalent LIMIT en SQL)  
    request.fetchBatchSize = 20  
  
    // Critères de sélections (équivalent WHERE en SQL)  
    // Ici on récupère tous les documents.  
    request.predicate = NSPredicate(value: true)  
  
    // Tri des résultats (équivalent ORDER BY en SQL)  
    request.sortDescriptors = [  
        NSSortDescriptor(key: "title", ascending: true)  
    ]  
  
    // ...
```

Ensuite, on définit la requête et ses critères.

Core Data

(Récupération d'entités)

```
// ...
```

```
let controller = NSFetchResultsController(
    fetchRequest: request,
    managedObjectContext: self.context,
    sectionNameKeyPath: nil,
    cacheName: nil
)

do {
    try controller.performFetch()
    return controller
} catch {
    fatalError("Erreur de récupération des documents : \(error)")
}
}
```

Finalement, le contrôleur associé à la requête

Core Data

(Récupération d'entités)

```
if (fetchDocuments.fetchedObjects != nil) {  
    for document in fetchDocuments.fetchedObjects! {  
        // ...  
    }  
}
```