

## CAHIER DES CHARGES APP TODOS

DÉVELOPPEMENT IOS



# App Todos

**Vous devrez rendre un zip (ou un lien vers un dépôt git) contenant la version finale du workspace XCode avec un fichier texte ReadMe.md décrivant l'état d'avancement du projet ainsi que la liste des problèmes non résolus.**

L'objectif final est de construire une application permettant d'avoir des listes de tâches (TODO list). Chaque liste comportera plusieurs tâches qui pourront être validées (marquées par une *checkmark*). Il sera possible dans la version finale d'ajouter des listes et d'ajouter des tâches dans chacune des listes. Il sera également possible de supprimer des éléments dans chacune des listes.


Dans un premier temps, il n'y aura qu'une seule liste de tâches.

1. Créer une nouvelle application nommée **Checklists** en utilisant le template *Single View Application*, device : **Universal** (décocher Core Data)

Pour commencer, on partira d'un simple UITableViewController

2. Modifier le fichier ViewController.swift pour le renommer en **ChecklistViewController** (clic droit : Refactor → Rename) et modifier le code source pour qu'il hérite de UITableViewController
3. Dans le storyboard, supprimer le contrôleur de vue déjà présent et faire glisser un **Table View Controller** à définir comme **contrôleur de vue initial** du storyboard et à associer à la classe ChecklistViewController (ce contrôleur sera la *delegate* et la *datasource* pour la *table view* : cela peut se vérifier dans l'inspecteur des connexions de la *table view*)

## Tester l'application

4. Dans le storyboard, éditer la *Prototype Cell* qui servira de base à la création des cellules en utilisant l'*Attributes Inspector*  (pour sélectionner le composant de son choix dans le storyboard il peut être pratique d'utiliser le raccourci ^⌘ + clic) :

- Ajouter un *Accessory* de type *Checkmark*
- Affecter une chaîne à l'attribut *reuse identifier* : mettre la valeur **ChecklistItem**

5. Ajouter à la classe **ChecklistViewController** les méthodes :

- `override func tableView(_ tableView: UITableView, numberOfRowsInSectionSection section: Int) -> Int`
  - retourner 1 pour commencer
- `override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell`
  - utiliser `let cell = tableView.dequeueReusableCell(withIdentifier: "ChecklistItem", for: indexPath)` pour recycler ou créer une nouvelle cellule selon le modèle défini dans le storyboard
  - La cellule possède un attribut `textLabel` qui permet de modifier le texte affiché. Lui affecter une valeur fixe pour le test.

## Tester l'application (le haut de l'affichage présente un problème qui sera réglé par la suite)

6. Créer une classe swift **ChecklistItem** qui aura 2 variables d'instances : `text` (de type String) et `checked` (de type Bool). Cette classe possèdera un initialiseur qui prendra ces deux paramètres avec une valeur par défaut de false pour l'attribut `checked` ce qui permettra de créer un nouvel élément des deux manières suivantes :

- `CheckListItem(text: "Finir le cours d'iOS")`
- `CheckListItem(text: "Mettre à jour XCode", checked: true)`

*Relancer l'application et vérifier le comportement lors d'un clic sur une ligne (elle doit rester grisée).*

7. Afin d'éviter que la ligne sélectionnée ne reste grisée, on peut implémenter dans **ChecklistViewController** la méthode

```
override func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {
    tableView.deselectRow(at: indexPath, animated: true)
}
```

8. Modifier la classe **ChecklistViewController** afin qu'elle possède une variable d'instance référençant un tableau de **CheckListItem**. Dans la méthode `viewDidLoad`, créer plusieurs éléments de type **CheckListItem** et les ajouter au tableau. Modifier les méthodes `tableViewDelegate` et `datasource` de manière à afficher la liste des items. Créer et utiliser les méthodes suivantes afin de configurer les cellules en fonctions de l'item demandé :

```
func configureCheckmark(for cell: UITableViewCell, withItem item: CheckListItem)
func configureText(for cell: UITableViewCell, withItem item: CheckListItem)
```

9. Ajouter à la classe **CheckListItem** une méthode `toggleChecked()` qui permet de modifier l'état de chaque item et l'appeler dans la méthode de `UITableViewDelegate` qui est appelée lors de la sélection d'une ligne par l'utilisateur. Il faudra ensuite appeler `tableView.reloadRowsAtIndexPaths` afin que la vue recharge la ligne dont l'état a changé.

*Tester l'application*

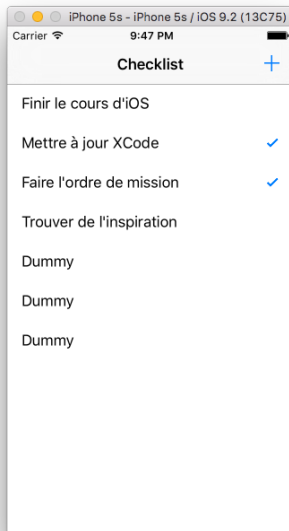
#### Mise en place du contrôleur de navigation et du bouton ajout

10. Afin de rajouter une barre de titre et un bouton + permettant l'ajout de nouveaux items, embarquer le *Table View Controller* dans un *Navigation Controller*. Pour cela, il faut sélectionner le *Table View Controller* dans le storyboard et dans le menu choisir **Editor -> Embed In -> Navigation Controller**

*Tester l'application*

11. Dans le storyboard, cliquer dans la barre de navigation du *Table View Controller* pour modifier le titre et le nommer **Checklist**. Depuis la librairie d'objets, faire glisser un *Bar Button Item* dans la partie droite de la barre de navigation et dans l'inspecteur des attributs choisir **System Item: Add**
12. Afin de tester l'ajout, nous allons associer à ce bouton une méthode `addDummyTodo()` définie dans **ChecklistViewController** (à créer et câbler directement dans le storyboard). Cette méthode permettra d'ajouter un item à la liste et préviendra la *table view* de l'arrivée d'une nouvelle ligne en utilisant la méthode `insertRows(at:with:)` de *table view*

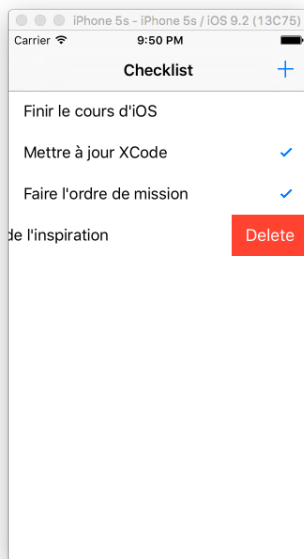
*Tester l'application*



### Suppression des items

- Ajouter la gestion de la suppression des lignes (ajouter la méthode `tableView(_:commit:forRowAt:)` du *data source* qui appellera `deleteRows(at:with:)` de la *table view*)

#### *Tester l'application*



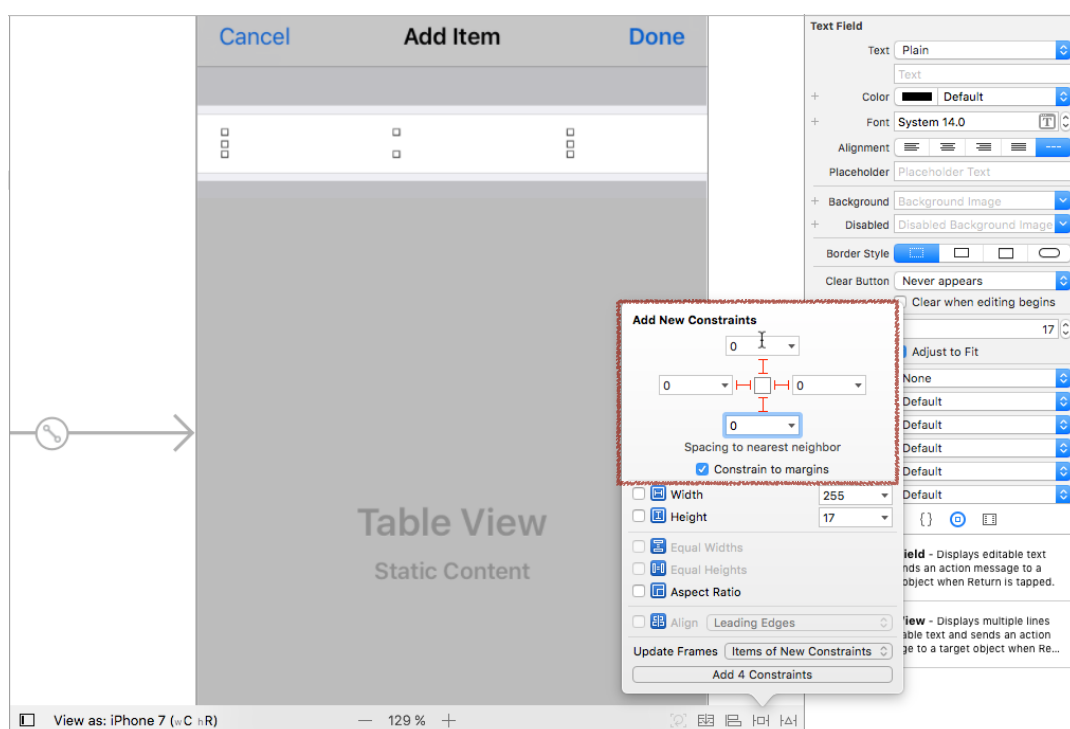
### Ecran d'ajout d'un nouvel item

- Dans le storyboard, faire glisser un nouveau **Table View Controller**, supprimer le lien entre le bouton + et l'action de création et le remplacer par une segue vers le nouveau contrôleur : choisir **Present Modally** comme type de segue (le nouveau contrôleur sera présenté par le bas de l'écran et le recouvrira intégralement). ce dernier choix peut être modifié plus tard dans les attributs de la segue.
- Ajouter un identifiant à la segue : **addItem**

16. Embarquer le nouveau *Table View Controller* dans un *Navigation Controller* et modifier le titre en **Add Item**. Ajouter un bouton **Cancel** à gauche et un bouton **Done** à droite de la barre de navigation.
17. Créer une classe **AddItemViewController.swift** qui hérite de **UITableViewController** (choisir **Cocoa Touch Class** dans les templates) et ne pas créer de fichier xib. Modifier le storyboard pour que le nouveau contrôleur de vue table soit du type **AddItemViewController**. Dans cette dernière classe supprimer tout le corps (méthodes fournies par défaut et en commentaires), il ne doit rester que la définition de la classe et un bloc vide entre accolades.
18. Créer et associer des actions vers deux méthodes **cancel()** et **done()** depuis les boutons (attention, ces méthodes ne doivent pas prendre d'objet en paramètre : écrire les méthodes en les annotant avec **@IBAction** si nécessaire). Dans ces méthodes, on se contentera pour l'instant de faire disparaître le contrôleur de vue en appelant **dismiss(animated:)**.

### *Tester l'application*

19. Dans le storyboard, sélectionner la *table view* du contrôleur **AddItemViewController** et dans l'inspecteur des attributs modifier le réglage Content à **Static Cells**. Les cellules de ce contrôleur seront toujours les mêmes et il n'est donc pas nécessaire de mettre en place un processus d'allocation dynamique et de recyclage.
20. Toujours dans l'inspecteur des attributs, modifier le style de la table view en **Grouped**. Les cellules sont placées au dessus du fond et il est possible de constituer des groupes. C'est une stratégie classique pour la mise en page de certains écrans d'applications (comme celui des réglages de l'iPhone par exemple). Supprimer les cellules pour n'en avoir plus qu'une et ajouter un **text field** dans la cellule. Dans les attributs du text field supprimer la bordure (**Border Style : None**). Ajouter des contraintes au **text field** afin qu'il occupe toute la cellule en tenant compte des marges (ajouter 4 contraintes avec une distance nulle).



*Tester l'application (analyser le comportement lorsque l'on sélectionne le champ de texte puis que l'on sélectionne le bord de la cellule)*

21. Pour empêcher la cellule de changer d'aspect lors des sélections il faut désactiver la sélection en passant l'attribut **Selection** de la *table view* à **none** dans le storyboard

### Tester l'application

#### Récupération du texte saisi

22. Créer un outlet vers le champ de texte et afficher le contenu du champ dans la méthode `done()` avec la fonction `print` juste avant de faire disparaître le contrôleur de vue d'ajout d'un item.

### Tester l'application

23. Implémenter la méthode **`viewWillAppear`** dans le contrôleur de vue d'ajout d'un item (taper `vWA` dans le corps de la classe pour utiliser l'autocomplétion). Dans cette nouvelle méthode, faire en sorte que le clavier soit tout de suite opérationnel et afin d'éviter à l'utilisateur d'avoir d'abord à taper dans le champ de texte pour afficher le clavier (méthode **`becomeFirstResponder`**).

24. Dans le storyboard, sélectionner le champ de texte et éditer ses attributs pour avoir les réglages suivants :

1. Placeholder : **Name of the item**
2. Font : System 17
3. Adjust to fit : décocher
4. Capitalization : Sentences
5. Return Key : Done

25. Toujours dans le storyboard, câbler l'évènement **Did End on Exit** du champ de texte sur la méthode **`done`**.

### Tester l'application

26. Dans les attributs du champ de texte, il est également possible de cocher **Auto-enable Return Key** afin que le bouton Done du clavier ne soit actif que si du texte a été tapé.

### Tester l'application

27. Pour que le bouton **Done** de la barre de navigation soit également désactivé si aucun texte n'a été tapé, il est nécessaire que **`AddItemViewController`** soit le *delegate* du champ de texte. Il faut alors implémenter la méthode `textField(_:shouldChangeCharactersIn:replacementString)`. Pour gérer le problème de compatibilité lié à l'utilisation de `NSRange` avec des chaînes de caractère swift, se référer au post Stackoverflow suivant : <http://stackoverflow.com/questions/25138339/nsrange-to-rangestring-index> (la réponse de Martin R. est la plus complète et tient véritablement compte des spécificités d'Unicode : en particulier lors de l'usage d'emojis).

#### Ajout d'un item à la liste

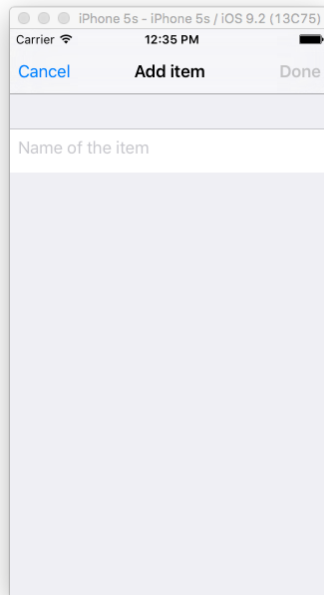
28. Déclarer un protocole **`AddItemViewControllerDelegate`** (dans le même fichier que le contrôleur) qui définira deux méthodes pour gérer l'annulation et la validation d'un nouvel item.

```
protocol AddItemViewControllerDelegate : class {  
    func addItemViewControllerDidCancel(_ controller: AddItemViewController)  
    func addItemViewController(_ controller: AddItemViewController, didFinishAddingItem item: CheckListItem)  
}
```

Remarque : on rajoute le mot clé `class` pour spécifier que ce protocole ne pourra être implémenté que par une classe (et non pas des structs ou enums). En effet, l'utilisation du pattern delegate nécessite de conserver une référence vers l'objet à notifier, il est donc nécessaire que le délégué ait une sémantique de référence et non pas de valeur.

29. Implémenter ce protocole dans **ChecklistViewController** (en utilisant une extension) et s'enregistrer comme *delegate* au moment de la transition vers la vue gérée par **AddItemViewController**
30. L'appel à la méthode `dismiss...` pour faire disparaître le contrôleur d'ajout se fera désormais dans **ChecklistViewController** dans les méthodes du protocole **AddItemViewControllerDelegate**

### *Tester l'application*



### **Edition d'un item**

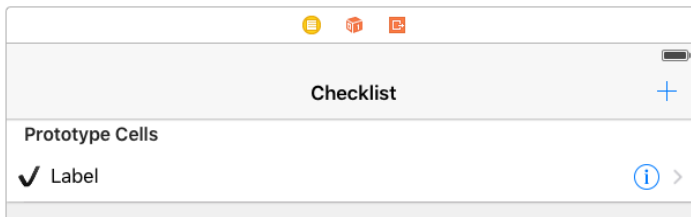
On réutilisera le même contrôleur de vue que pour l'ajout. Il sera nécessaire de lui apporter des modifications pour :

- Modifier le titre affiché dans la barre de navigation avec le texte : **Edit Item**
- Pouvoir passer au contrôleur l'item à éditer
- Initialiser le champ de texte avec le texte de l'item
- Ne pas insérer un nouvel item lorsque l'utilisateur clique sur **Done**

31. Modifier l'apparence de la liste pour ajouter dans chaque ligne un **detail disclosure button** qui permettra de signaler à l'utilisateur qu'il peut obtenir des informations sur l'item en cours en cliquant dessus (attribut **Accessory: Detail Disclosure** au lieu de **Checkmark**) .

- La gestion de la Checkmark sera faite en ajoutant deux labels (un situé à gauche de la cellule dans lequel on mettra un caractère unicode représentant une CheckMark. et le deuxième s'étendant sur le reste de la cellule).
- Pour ajuster les dimensions du label contenant la checkmark, il faut commencer par insérer un caractère spécial (menu **Edit->Emoji & Symbols** puis taper check dans le champ de recherche), puis ajuster la taille de la police et utiliser le menu **Editor -> Size to Fit Content** pour ajuster les dimensions du label à son contenu.
- Utiliser l'autolayout pour centrer les labels verticalement dans la cellule, imposer la largeur du label contenant la checkmark et fixer les distances entre labels et entre labels et bords de la cellule (aligner sur les bords sur les

marges et mettre la distance standard de 8 points entre les 2 labels : c'est celle qui correspond aux pointillés bleus de guidage qui apparaissent lors du placement des composants).



32. Créer une classe Cocoa Touch nommée **ChecklistItemCell** héritant de **UITableViewCell** et l'associer à la *prototype cell*. Créer des *outlets* permettant d'accéder aux 2 labels.
33. Associer le clic sur le **Detail Disclosure Button** à une segue vers **AddItemViewController** de type **present modally** (pour cela il faut faire un clic droit sur la cellule et câbler **accessory action** (dans la liste des *triggered segues*). Lui donner l'identifiant **editItem**.
34. Modifier la classe **ChecklistViewController** pour gérer correctement l'affichage de la liste avec les CheckMarks en utilisant l'attribut **isHidden** pour masquer une vue.

*Tester l'application (analyser le comportement des boutons Done et Cancel)*

35. Ajouter une propriété **itemToEdit** au contrôleur **AddItemViewController** (bien réfléchir à son type) . Redéfinir également la méthode **viewDidLoad** (sans oublier d'appeler celle de la super classe) pour détecter s'il est en mode ajout ou édition et configurer la vue en fonction. Modifier **prepareForSegue** pour affecter cette référence à l'item à éditer (sender référence la cellule sur laquelle est arrivé le tap qui a déclenché la segue et la table View possède une méthode **indexPath(for:)** qui retourne l'indice d'une cellule).
36. Modifier le protocole pour **AddItemViewControllerDelegate** pour ajouter une méthode :

```
func addItemViewController(_ controller: AddItemViewController, didFinishEditingItem item: ChecklistItem)
```

37. Implémenter cette dernière méthode pour faire fonctionner l'édition. Cela nécessite de pouvoir retrouver l'indice de l'item dans la liste afin de prévenir la tableView qu'elle doit recharger la ligne en question. Pour rechercher l'indice la classe Array possède une méthode **index(where:)** qui prend une closure en paramètre qui retourne **true** quand l'objet recherché est trouvé. On peut l'utiliser ici en utilisant le code **items.index(where:{ \$0 == item })** où l'opérateur **==** teste l'identité et non l'égalité.

Afin de pouvoir utiliser un test d'égalité (qui serait utile si on avait choisi de représenter les items de la liste par des structs, ou dans d'autres contextes) il faudrait implémenter le protocole **Equatable** (permet de tester l'égalité en utilisant l'opérateur **==**). Pour pouvoir tester l'égalité des objets ChecklistItem il faut les faire se conformer au protocole **Equatable** et définir (en dehors du corps de la classe) une surcharge de l'opérateur infix global **==**

```
func == (lhs: ChecklistItem, rhs: ChecklistItem) -> Bool {  
    return (lhs.text == rhs.text)  
}
```

38. Finaliser l'édition des tâches

*Tester l'application (vérifier que l'ajout et l'édition fonctionnent bien)*

## Refactoring

39. Avant de procéder à des modifications il est nécessaire de faire un commit Git ou équivalent.



40. Renommer **AddItemViewController** en **ItemDetailViewController** en utilisant le refactoring (clic droit : refactor) ainsi que toutes les méthodes commençant par **addItemViewController** par **itemDetailViewController**. Renommer également le protocole **AddItemViewControllerDelegate** en **ItemDetailViewControllerDelegate**.
41. Après toutes ces modifications, il est utile de faire un **Product -> Clean** avant de recopier et de vérifier que tout fonctionne toujours.

### Sauvegarde de la liste

Les items de la liste seront sauvegardés dans le dossier Documents de l'application (chaque appli possède son propre espace *sandboxé* et le dossier Documents en fait partie et est sauvegardé automatiquement lors des *back-ups* iTunes ou iCloud. Les données seront enregistrées dans un fichier Checklists.json ([https://developer.apple.com/library/prerelease/content/documentation/FileManagement/Conceptual/FileSystemProgrammingGuide/FileSystemOverview/FileSystemOverview.html#//apple\\_ref/doc/uid/TP40010672-CH2-SW12](https://developer.apple.com/library/prerelease/content/documentation/FileManagement/Conceptual/FileSystemProgrammingGuide/FileSystemOverview/FileSystemOverview.html#//apple_ref/doc/uid/TP40010672-CH2-SW12)).

42. Dans la classe **ChecklistViewController**, ajouter :

- une variable de classe (computed property) `documentDirectory`: URL qui retournera l'URL du dossier Documents de l'utilisateur (utiliser la classe `FileManager` dont la méthode `urls(for:in:)` permet de récupérer l'url du dossier Documents de l'utilisateur)
- une variable de classe (computed property) `dataFileUrl`: URL qui retournera l'URL du fichier **Ckecklists.json** dans lequel seront sauvegardées les données. Cette dernière méthode utilisera les méthodes de la classe URL pour assembler le chemin complet du fichier (`appendingPathComponent`, `appendingPathExtension`).
- Tester ces fonctions en affichant le chemin du fichier avec la fonction `print`. Ouvrir une fenêtre du Finder sur le dossier (menu **Aller -> Aller au dossier...** puis coller le chemin du dossier : attention à bien enlever le préfixe `file://`), pour l'instant le dossier est vide.

Afin de pouvoir enregistrer la liste des items qui est de type `Array<ChecklistItem>` (ou `[ChecklistItem]`) il est nécessaire d'apporter une modification à la classe **ChecklisItem** afin qu'elle puisse être sérialisée pour être stockée dans un fichier json. Anciennement, les objets iOS utilisaient le protocole `NSCoding` qui vient de l'objective C et qui était réservé aux objets héritant de la classe `NSObject`, il était donc impossible de sérialiser des structs et des enums. Depuis Swift 4, il est possible d'utiliser le protocole `Codable` (qui gère également l'encodage et le décodage de fichier JSON).

43. Implémenter le protocole `Codable` dans la classe **ChecklistItem** (il n'y a rien de plus à faire que de déclarer que la classe se conforme au protocole : les noms des variables d'instance serviront de « clés » du fichier JSON cf. <https://developer.apple.com/documentation/foundation/jsonencoder>).
44. Ajouter une méthode `func saveChecklistItems()` à la classe **ChecklistViewController** et l'appeler à chaque fois qu'une modification est faite en vous inspirant toujours du [lien précédent](#).
45. Ajouter également une méthode `func loadChecklistItems()` en utilisant un **JSONDecoder**
46. L'appel de la méthode `loadChecklistItems` se fera au moment où le contrôleur de vue **ChecklistViewController** est instancié depuis le storyboard. Il existe deux méthodes de la classe appropriées :
- `required init?(coder aDecoder: NSCoder)` : les objets contenus dans le storyboard sont désérialisés au moment de leur chargement
  - `override func awakeFromNib()` : fonction appelée à la fin du chargement depuis le fichier xib (anciennement nib) ou storyboard. Lors de l'appel de cette méthode toutes les *outlets* et *actions* ont été associées.

Note : la fonction `viewDidLoad()` pourrait également faire l'affaire mais en théorie elle peut être appelée plusieurs fois pendant la durée de vie de contrôleur si sa vue est déchargée puis rechargée (bien que cela ne semble plus être le cas depuis quelques versions d'iOS).

*Tester l'application :*

- *vérifier l'apparition du fichier json et l'ouvrir pour visualiser sa structure*
- *quitter l'application en simulant un double appui sur le bouton Home (⌘H) et vérifier que les éléments sont toujours là au lancement suivant (penser à éliminer les éléments ajoutés en dur dans la méthode **`viewDidLoad`** (valider ajout/suppression/édition/validation d'un item)).*

## Gestion de plusieurs listes

Nous allons ajouter un nouvel écran d'accueil qui contiendra la liste de toutes les check-lists. Il sera également possible d'ajouter de nouvelles listes et de naviguer entre les listes et leur contenu.

On ajoutera deux nouveaux contrôleurs :

- **AllListViewController** : affiche toutes les listes
- **ListDetailViewController** : écran d'ajout d'une nouvelle liste

47. Dans le storyboard, faire glisser un table view controller près du contrôleur de navigation initial les lier par un Ctrl + drag partant contrôleur de navigation. Choisir root view controller dans le sous menu **Relationship Segue**. Créer la classe **AllListViewController.swift** et l'associer au table view controller. Modifier le titre pour afficher **Checklists**.

48. Créer une **segue** de type **Show** entre la cellule (**Prototype cell**) et le contrôleur qui affiche les tâches de la liste. Afficher un **Accessory** de type **Detail Disclosure**. Renommer le titre du contrôleur **ChecklistViewController** en **Name of the list** (il sera remplacé plus tard par le titre de la liste).

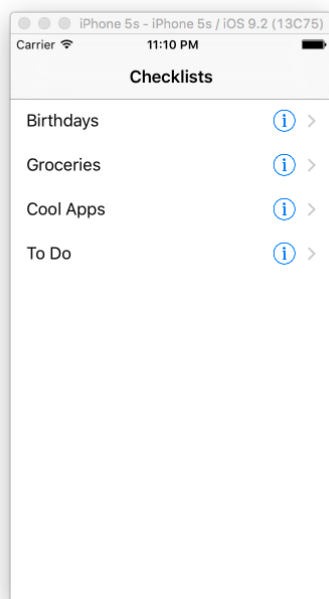
49. Afin de tester l'application, éditer le code de **AllListViewController** pour afficher trois listes (nommées liste 1, liste 2 et liste 3) : un clic sur la liste affiche toujours la même checklists (créée précédemment).

### *Tester l'application (vérifier la navigation)*

50. Créer une classe **Checklist** qui permettra d'enregistrer un nom pour la liste ainsi que la liste des items qu'elle contient.

- ajouter une variable d'instance **name** de type **String**
- ajouter une variable d'instance **items** de type **[ChecklistItem]**.
- ajouter une méthode **init** qui prendra le nom de la liste en paramètre et la liste items de manière « optionnelle » (valeur par défaut correspondant à un tableau vide).

51. Ajouter une variable **lists** à **AllListViewController** permettant de stocker une liste qui sera dans un premier temps initialisée dans la méthode **viewDidLoad**. Créer des listes et afficher leurs noms afin de valider le bon fonctionnement.



52. Ajouter une propriété `var list: Checklist!` à la classe **ChecklistViewController** qui sera initialisée lors de la **segue** (on utilise ! car il faut toujours fournir cette liste). Modifier également la méthode **viewDidLoad** pour afficher le nom de la liste dans la barre de navigation. Dans un premier temps on n'affichera pas le contenu des listes (toutes les listes afficheront donc le même contenu, qui correspond au fichier sauvegardé, ce sera corrigé plus tard).

*Tester l'application (vérifier le texte dans la barre de navigation - le comportement du bouton back - le swipe depuis le bord du téléphone pour revenir à l'écran précédent)*

#### Ajout et édition de listes

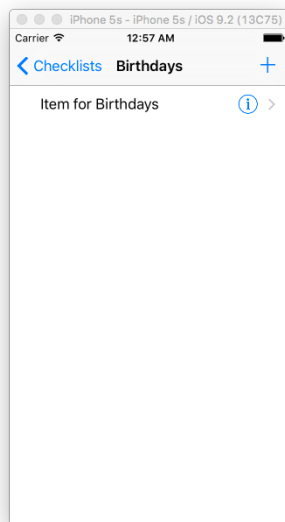
53. Reprendre à l'identique la manipulation déjà faite pour l'ajout/édition d'items à une liste en remplaçant **ItemDetailViewController** par **ListDetailViewController**. Pour la partie storyboard, on embarquera directement ce contrôleur dans un Navigation Controller pour qu'il dispose d'une barre de navigation avec titre.

*Tester l'application*

#### Ajout des items aux listes

54. Mettre en commentaires toutes les méthodes réalisant la sauvegarde (il faut terminer la mise en place du nouveau modèle. Afficher les items de la liste sélectionnée plutôt que ceux qui ont été chargés depuis la sauvegarde.
55. Pour tester le bon fonctionnement, il faut remplir les listes avec des items. Dans la méthode **init** de **AllListViewController** ajouter la création de ces systèmes. Utiliser une simple boucle **for-in** pour que chaque liste contienne un item nommé **Item for <NomListe>** où **<NomListe>** aura été remplacé par le nom de la liste

*Tester l'application*



56. Déplacer et modifier le code de sauvegarde dans la classe **AllListViewController** pour permettre l'enregistrement des listes et de leur contenu lors des ajouts de nouvelles listes ou du re-nommage ainsi que leur chargement au début (mettre en commentaire le code qui remplissait les listes manuellement).

*Tester l'application (ajouter des listes / des items dans les nouvelles listes - quitter l'application et vérifier si tout est toujours présent)*

Les items ajoutés aux listes n'apparaissent pas car la sauvegarde ne s'effectue qu'en cas d'ajout/suppression/édition d'une liste mais pas de son contenu. La solution à ce problème passe par une mise à plat de la gestion de la persistance :

- pour l'instant la sauvegarde se fait dans un contrôleur de vue, ce qui est une mauvaises idée car ce n'est pas au niveau du contrôleur que doit se gérer la persistance
- sauvegarder le fichier à chaque changement n'est pas optimal, le plus judicieux serait d'enregistrer les modifications quand l'application se ferme (soit par une action explicite de l'utilisateur, soit quand le système a besoin de récupérer de la mémoire en fermant les applications à l'arrière plan)

57. Créer une classe **DataModel** qui se chargera de la persistance. Cette classe sera un singleton (<https://cocoacasts.com/what-is-a-singleton-and-how-to-create-one-in-swift/> ou <http://krakendev.io/blog/the-right-way-to-write-a-singleton>) qui pourra être interrogé pour accéder la liste. Déplacer toutes les méthodes de chargement et sauvegarde dans cette classe et les supprimer de **AllListViewController** (supprimer toutes les sauvegardes).

*Tester l'application (vérifier que le chargement fonctionne toujours mais qu'aucune modification n'est sauvegardée)*

Pour gérer le cycle de vie de l'application (<https://developer.apple.com/library/ios/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/TheAppLifeCycle/TheAppLifeCycle.html>) il est possible d'utiliser le delegate ([https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIApplicationDelegate\\_Protocol/index.html](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIApplicationDelegate_Protocol/index.html)).

58. Instrumenter la classe AppDelegate.swift avec des print (`print("Function: \(#function)")`) dans les méthodes de gestion du cycle de vie afin de vérifier les méthodes appelées lorsque l'application passe à l'arrière plan, est terminée alors qu'elle était au premier plan ou à l'arrière plan, ... Observer en particulier l'évènement **didEnterBackground**. Le message `willTerminate` n'est quasiment jamais appelé. Les sauvegardes doivent être faites lorsque l'application passe à l'arrière plan.

L'objet **AppDelegate** est notifié de tous ces événements mais il faudrait ensuite qu'il ait connaissance des classes qui doivent être prévenues pour effectuer les sauvegardes. Tous ces événements sont également disponibles via le *Notification Center*. La documentation de chaque classe fait apparaître la liste des notifications qu'elle est susceptible de poster :

API Reference

UIKit > UIApplication | Show API Changes

Notifications

All UIApplication notifications are posted by the app instance returned by [shared](#).

static let `UIApplicationBackgroundRefreshStatusDidChange`: `NSNotification.Name`

Posted when the app's status for downloading content in the background changes.

static let `UIApplicationDidBecomeActive`: `NSNotification.Name`

Posted when the app becomes active.

static let `UIApplicationDidChangeStatusBarFrame`: `NSNotification.Name`

Posted when the frame of the status bar changes.

static let `UIApplicationDidChangeStatusBarOrientation`: `NSNotification.Name`

Posted when the orientation of the app's user interface changes.

static let `UIApplicationDidEnterBackground`: `NSNotification.Name`

Posted when the app enters the background.

static let `UIApplicationDidFinishLaunching`: `NSNotification.Name`

59. Modifier le constructeur de la classe `DataModel` pour qu'elle s'enregistre auprès du *Notification Center* pour être prévenue de l'évènement lorsque l'application est passée à l'arrière plan et faire la sauvegarde à ce moment en utilisant la syntaxe suivante :

```
NotificationCenter.default.addObserver(  
    self,  
    selector: #selector(saveChecklists),  
    name: .UIApplicationDidEnterBackground,  
    object: nil)
```

Le centre de notifications est une API Objective-C qui utilise les noms des méthodes à appeler sous forme de chaînes. Depuis la version 3 de swift il existe une constructions `#selector(saveChecklists)` qui permet au compilateur de vérifier que le sélecteur existe vraiment (dans les versions antérieures, en cas de faute de frappe cela produisait une erreur à l'exécution). Toutes les connexions basées sur des chaînes de caractères échappent aux vérifications du compilateur et ne tire donc pas profit du tapage fort. Depuis swift 3, les notifications qui étaient également représentées préalablement par des chaînes de caractères s'appuient maintenant sur des enums pour tirer profit du typage fort.

Attention : le centre de notifications est une API objective-C, elle ne peut appeler que des méthodes écrites en swift que si celles-ci sont compatibles objective-C. Cette compatibilité est obtenu en héritant d'une classe Objective-C (par exemple `NSObject`) ou en marquant la méthode `saveChecklists` avec l'attribut `@objc`

60. Ajouter `@objc` devant la méthode `saveChecklists`

*Tester l'application (vérifier la bonne sauvegarde lors de l'ajout/suppression/édition de listes et ajout/suppression/édition/check des items)*

### Affichage du nombre de tâches restantes (All Done! / (No Item) / (2 Remaining))

61. Changer le style de cellules dans **AllListViewController** pour utiliser des cellules de style **subtitle**. Ces cellules possèdent une propriété `detailTextLabel` dans lequel on affichera le nombre d'items non cochés. Afficher un message fixe dans ce champ afin de tester l'affichage.
62. Ajouter une propriété calculée (*computed property*) à la classe **CheckList** nommée **uncheckedItemCount** qui comptera le nombre d'items non cochés (possibilité d'utiliser `filter` ou `reduce` pour effectuer ce calcul en une ligne).
63. Afficher le nombre d'items non cochés dans le `detailTextLabel` et afficher le texte **All Done!** s'il n'en reste aucun et **(No Item)** pour les listes vides (la logique nécessaire peut être implémentée en utilisant la richesse de la syntaxe du `switch` en `swift`).

*Tester l'application (faire en sorte que les modifications soient reflétées : en modifiant les éléments d'une liste pour valider que le sous-titre est bien mis à jour en fonction du nombre d'items de la liste et de leur état)*

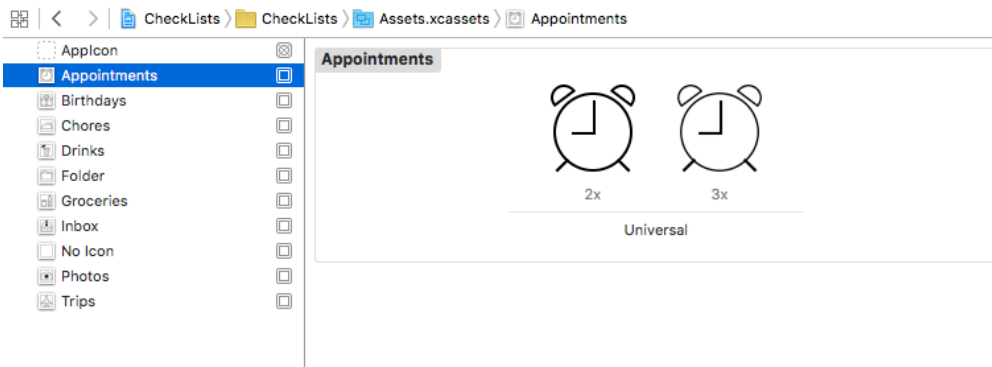
### Tri des listes par ordre alphabétique

64. Ajouter une méthode **sortChecklists()** à la classe **DataModel** qui s'appuiera sur la méthode **localizedStandardCompare** définie dans la classe **String** pour trier les listes par ordre alphabétique. Ajouter les appels à la méthode **sortChecklists** aux endroits appropriés de la classe **AllListViewController**.

*Tester l'application*

## Ajout d'une icône aux listes

65. Récupérer sous ClaCo l'archive Checklist assets.zip et la décompresser. Dans le projet Xcode, sélectionner le catalogue des Assets (Assets.xcassets) et cliquer sur le + situé en bas de l'écran, choisir Import... et sélectionner les fichiers contenus dans le dossier Checklist Icons.



Pour en savoir plus sur la taille des assets et le fonctionnement du catalogue d'assets les liens suivants sont utiles :

- Fonctionnement de xcassets : <http://help.apple.com/xcode/mac/8.0/#/dev10510b1f7>
- Pour les tailles d'icônes standard : <https://developer.apple.com/ios/human-interface-guidelines/icons-and-images/image-size-and-resolution/>
- Pour les tailles d'écran et les correspondance entre points et pixels : <http://www.paintcodeapp.com/news/ultimate-guide-to-iphone-resolutions>

66. Créer un enum **IconAsset** sur le modèle suivant :

```
enum IconAsset : String {
    case Appointments
    case Birthdays
    case Chores
    case Drinks
    case Folder
    case Groceries
    case Inbox
    case NoIcon = "No Icon"
    case Photos
    case Trips

    var image : UIImage {
        return UIImage(named: self.rawValue)!
    }
}
```

67. Ajouter une propriété **icon** de type **IconAsset** à la classe **CheckList**. Il faut rendre l'enum **Codable** afin que la classe **CheckList** puisse rester **Codable**. Il faut également donner une valeur à cette nouvelle propriété dans la méthode `init(name:items:)` car Swift interdit l'instanciation d'objets avec des propriétés non-optionnelles non-initialisées. Pour l'instant on mettra la valeur par défaut **NoIcon**. On peut modifier la méthode `init` pour lui rajouter un argument qui prendra une valeur par défaut.

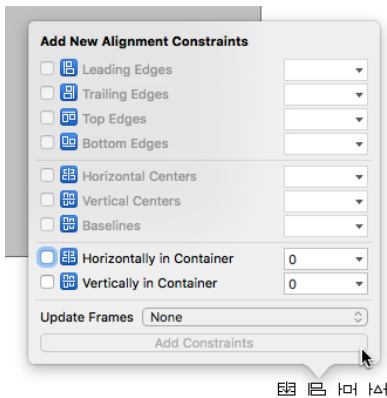
*Tester l'application (pourquoi les listes précédemment sauvegardées ont-elles disparues...ou pire pourquoi l'application crash-t-elle ?)*

68. Modifier le code de la classe **AllListViewController** pour ajouter l'affichage de l'image dans les cellules (utiliser la propriété **imageView** des cellules).

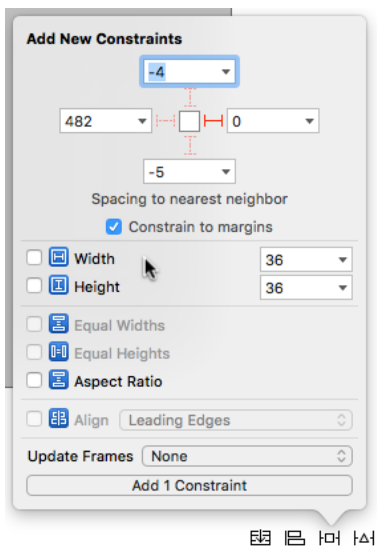


69. Dans le storyboard, éditer **ListDetailViewController** pour ajouter une section. Dans la nouvelle cellule, supprimer le champ de texte et le remplacer par un label (le positionner contre le guide de la marge gauche de la cellule). Ajouter également un accessoire (Disclosure Indicator) et une image view (la positionner contre le guide de la marge droite et utiliser l'inspecteur pour fixer sa taille à 36 x 36). Fixer les contraintes d'autolayout pour positionner le label et la vue image comme suit :

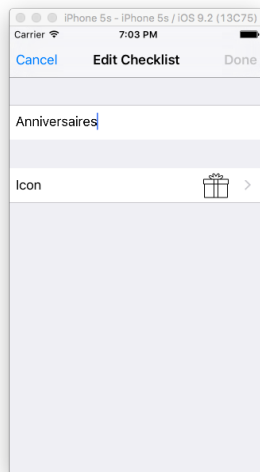
- Utiliser le menu Align (exemple ci-dessous) pour centrer le label et l'imageView verticalement dans leur conteneur



- Utiliser l'assistant (Pin) pour fixer le label à 0 du bord gauche. Fixer l'imageView à 0 du bord droit (cf ci-dessous), fixer également la hauteur et la largeur à 36.



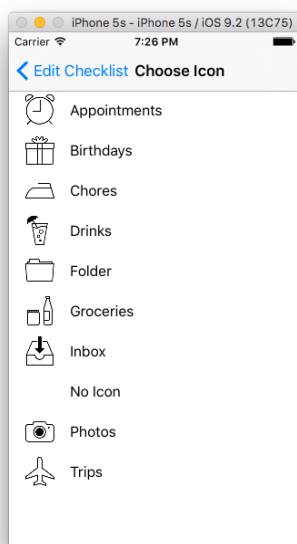
L'objectif est d'aboutir au layout ci-dessous (il reste encore un peu de code à éditer pour pouvoir afficher l'image dans l'imageView).



70. Dans la classe **ListDetailViewController** ajouter un outlet pour pouvoir accéder à l'imageView et modifier le code pour afficher l'image associée à la liste.

### *Tester l'application*

71. Créer une classe **IconPickerController** (sous-classe de UITableViewController) qui permettra de choisir une icône lors de la création ou de l'édition d'une liste. Mettre en place le mécanisme de votre choix pour récupérer l'icône qui aura été sélectionné (delegate ou closure). Ajouter un Table View Controller dans le storyboard, l'associer à la classe précédemment créée et modifier le style des cellules en **Basic**.
72. Faire en sorte que ce nouveau contrôleur s'affiche lorsque l'utilisateur clique sur la cellule Icon du contrôleur d'ajout / édition d'une liste (en mettant en place une Segue de type Show). Cette segue aura pour effet de s'appuyer sur le contrôleur de navigation. Afficher la liste des noms des icônes avec les icônes associées (les cellules possèdent une variable d'instance image qui décale automatiquement le texte lorsqu'une image est présente).



73. Lors de la sélection d'une ligne, provoquer le retour à la vue précédente en agissant au niveau du contrôleur de navigation. Le contrôleur de navigation gère une pile de contrôleur de vue : le retour à la vue précédente se fait par un pop. Le contrôleur de navigation dans lequel se trouve un contrôleur de vue est accessible par une propriété naviga-

tionController. Modifier l'icône associée à l'item en cours de création/édition (attention dans un cas l'item existe déjà et dans l'autre il n'existe pas encore)

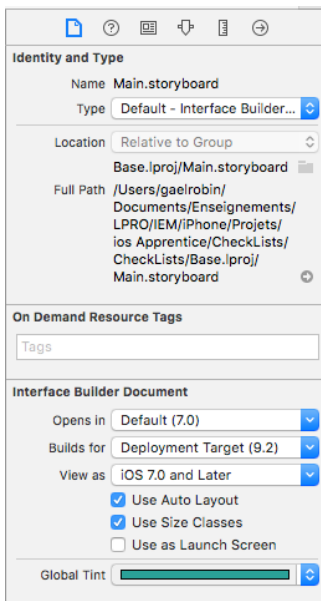
74. Modifier si nécessaire la gestion du bouton Done afin qu'il ne soit actif que si cela a du sens (texte non vide, texte ou icône édités).

75. Lors de la création proposer par défaut une icône de type folder.

### *Tester l'application*

#### **Modification de l'apparence**

76. Modifier l'apparence générale de l'application en changeant la teinte globale : aller dans le storyboard et sélectionner l'inspecteur de fichier, puis éditer la propriété Global Tint (cf capture ci-dessous). Modifier également la couleur du text du label contenant la checkmark en récupérant l'attribut `tintColor` de la vue (`view.tintColor`), certaines checkmarks ont une couleur fixe (noire), il faudra en choisir une autre.



77. Dans l'élément Assets.xcassets, sélectionner AppIcon et récupérer dans Checklist assets.zip les icônes de l'application à faire glisser dans les différents emplacements (une image de 29 pt en résolution 2x mesure 58 pixels).

78. Dans les options générales du projet (Cliquer sur le projet puis sur la cible et choisir l'onglet General), changer le launchscreen et sélectionner Main.storyboard. L'écran de lancement correspondra donc au premier écran de l'application. Supprimer le fichier LaunchScreen.storyboard

### *Tester l'application*

79. Ajouter des contraintes si nécessaire à l'ensemble des labels et champs de textes dans lesquels l'utilisateur peut saisir de longs noms de liste ou d'item.

*Tester l'application avec des noms longs pour différentes tailles de téléphone.*

### Ajout d'une liste vide lors du premier lancement (utilisation de UserDefaults)

Chaque application a la possibilité de sauvegarder des informations sous la forme de dictionnaires (clé/valeur) dans un système destiné à gérer les préférences de l'application : UserDefaults. En plus de la possibilité de lire et écrire les valeurs associées à une clé, il existe la possibilité d'enregistrer la valeur par défaut qui sera retournée si aucune valeur n'a jamais été écrite pour une clé (register(defaults:)). Dans un premier temps ces fonctionnalités seront ajoutées à la classe DataModel.

80. Ajouter une clé firstLaunch qu'il faudra d'abord enregistrer pour lui donner une valeur par défaut égale à true.

81. Ajouter la détection du premier lancement et la création d'une liste par défaut nommée List contenant les items (Edit your first item, Swipe me to delete).

### *Tester l'application*

### Ajout d'une notification locale à échéance de la tâche

Se référer à la documentation officielle sur les notifications pour les étapes de la configuration suivantes (<https://developer.apple.com/library/prerelease/content/documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG/>)

82. Dans un premier temps nous allons tester l'envoi d'une notification locale simple.

Dans les propriétés générales du projet il faut ajouter à la rubrique **Linked Frameworks and Libraries** le framework UserNotifications.framework qu'il faudra ensuite importer dans les fichiers qui feront appel à ce framework.

Dans la classe AppDelegate, modifier la méthode application(\_:didFinishLaunchingWithOptions:), pour demander la permission à l'utilisateur de lui envoyer des notifications locales.

83. Créer un objet de type UNNotificationRequest pour spécifier une notification de type Alert et Sound (toujours dans la même méthode de la classe AppDelegate) pour qu'elle apparaisse au bout de 10 secondes.

84. Faire en sorte que l'AppDelegate se conforme au protocole UNNotificationCenterDelegate et implémenter la méthode userNotificationCenter(\_:willPresent:withCompletionHandler:) pour imprimer un message sur la sortie standard lors de la réception d'une notification (afficher l'objet notification reçu).

*Tester l'application (en conservant l'application au premier plan au moment de la réception de la notification et en faisant passer également passer à l'arrière plan toujours lors de la réception afin de bien observer la différence de comportement).*

85. Modifier le corps de ma méthode userNotificationCenter(\_:willPresent:withCompletionHandler:) pour afficher quand même la notification même quand l'application se trouve au premier plan.

### Ajout d'une échéance aux tâches

Chaque item d'une liste possèdera un champ booléen (shouldRemind) associé à une date de notification.

Dans cette partie nous allons devoir gérer les actions suivantes :

- Programmer une notification lors de l'ajout d'un item dont le flag shouldRemind est vrai
- Quand l'utilisateur modifiera la date d'échéance, il faudra annuler la notification existante si elle existe pour en programmer une nouvelle
- Lors du changement du flag shouldRemind (au moyen d'un interrupteur) il faudra également annuler/reprogrammer les notifications
- Lors de la suppression d'un item il faudra supprimer une éventuelle notification associée

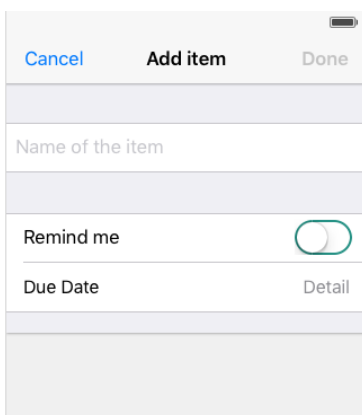
- Lors de la suppression d'une liste il faudra supprimer la totalité des notifications des items de la liste

La centre de notifications possède une méthode `removePendingNotificationRequests(withIdentifiers:)` qui permet d'annuler une liste de notifications à partir de la connaissance de leurs identifiants. Il est donc nécessaire d'associer un identifiant unique à chaque notification qui servira à les annuler. Pour créer des identifiants uniques on s'appuiera sur la classe `UserDefaults` pour mémoriser l'état du compteur entre plusieurs redémarrages de l'application.

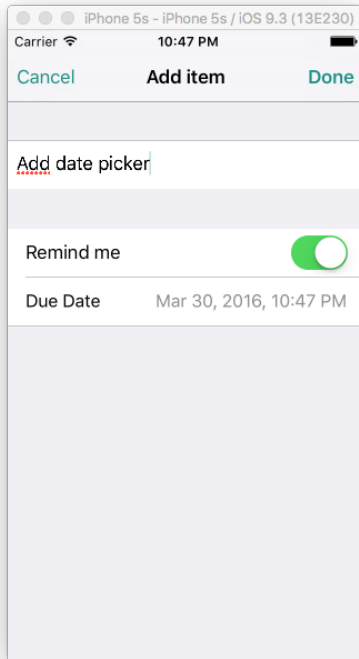
86. Modifier la classe **CheckListItem** pour ajouter une propriété **dueDate** de type `Date` qui sera initialisée à l'heure courante (l'utilisateur modifiera forcément cette échéance, on se place donc par défaut à l'instant courant qui servira de point de départ au réglage). Ajouter également les propriétés **shouldRemind** et **itemID**.
87. Modifier la méthode **init** pour prendre les arguments supplémentaires en paramètre en ajoutant des valeurs par défaut pour les nouveaux champs : `init(text:checked:shouldRemind:dueDate:)`
88. Ajouter une méthode `init(text:checked:shouldRemind:dueDate:itemID)` afin de ne pas créer de nouveaux identifiants pour des objets sauvegardé.
89. Créer une classe **Preferences** qui sera un singleton et qui encapsulera l'accès aux préférences stockées dans `UserDefaults`. Ajouter à cette classe une méthode **nextCheckListItemID** qui mémorisera le dernier ID utilisé dans les préférences (`UserDefaults`) sous la clé `checkListItemID` et se servira de cette valeur pour retourner une nouvelle valeur à chaque appel.
90. Affecter une valeur au champ `itemID` de **CheckListItem** dans la méthode `init(text:checked:shouldRemind:dueDate:)` en utilisant la méthode précédemment créée.
91. Déplacer vers la classe **Preferences** la gestion du premier lancement en ajoutant une propriété calculée (*computed property*) `firstLaunch` booléenne.
92. Ajouter un enum `UserDefaultsKeys` pour gérer proprement les clés de type `String`.

### Tester l'application

93. Dans le storyboard, éditer **ItemDetailViewController** pour ajouter une section dans la table view contenant 2 cellules. La première sera de type Custom (supprimer l'éventuel text field et ajouter un label et un switch à l'état Off par défaut), la seconde cellule sera de type Right Detail (suivre les indications du post suivant pour aligner le label Remind Me avec Due Date et le bord droit du switch avec le label Detail + utiliser l'autolayout pour centrer verticalement les composants dans la cellule Custom : <http://stackoverflow.com/questions/27420888/uitableviewcell-with-autolayout-left-margin-different-on-iphone-and-ipad>).



94. Ajouter un outlet pour le switch (**shouldRemindSwitch**) et pour le label Detail (**dueDateLabel**).
95. Ajouter une propriété **dueDate** pour stocker la date/heure du rappel en cours d'édition ou de création. Le flag **shouldRemind** booléen ne nécessite pas de variable d'instance car son état est facilement accessible dans la propriété **isOn** du switch. Modifier la méthode **viewDidLoad** pour gérer l'édition d'un item et terminer par un appel à une nouvelle méthode **updateDueDateLabel** qui se chargera de la conversion de la date/heure du rappel en une chaîne de caractère (utiliser **DateFormatter** et jouer avec les différents **dateStyle** et **timeStyle**, utiliser un playground pour expérimenter)

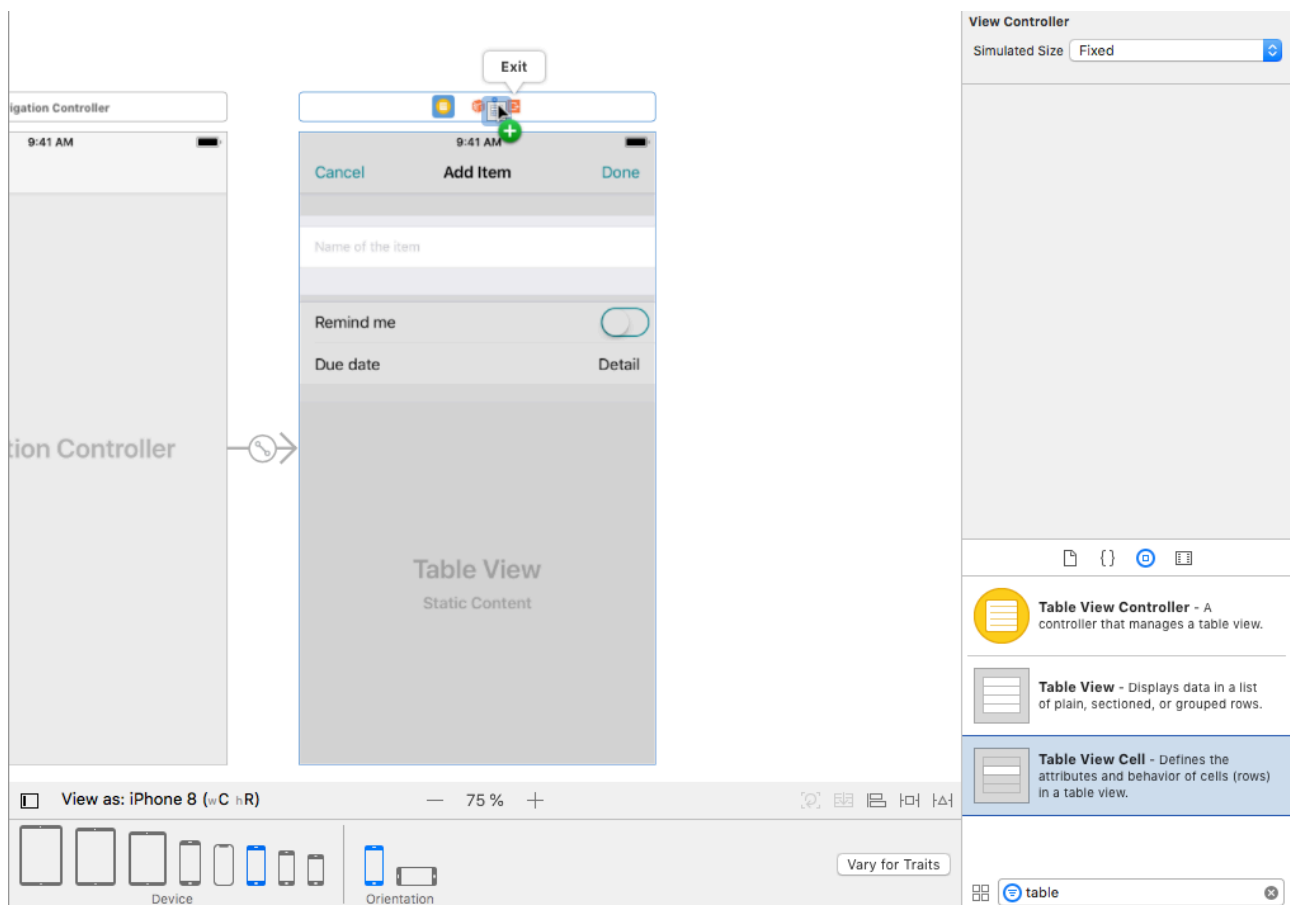


96. Gérer l'activation du bouton Done en cas de modification de l'état du switch.

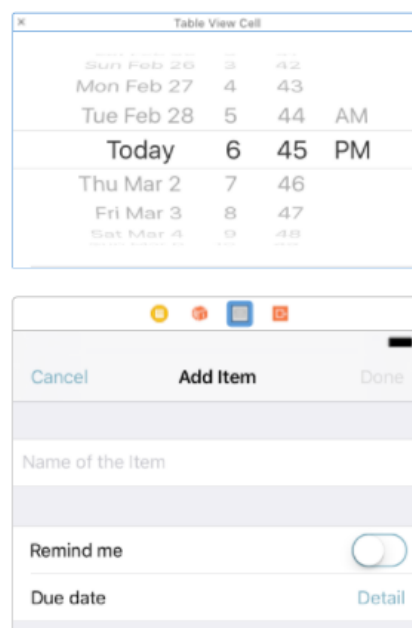
**Ajout d'un Date Picker pour sélectionner la date et l'heure de l'échéance (qui apparaîtra lors de la sélection de la ligne contenant la date d'échéance et disparaîtra une fois la date choisie).**

97. Dans la classe **ItemDetailViewController**, effectuer les modifications suivantes :

- Ajouter une propriété **isDatePickerVisible** booléenne (initialisée à **false**)
- Ajouter des méthodes **showDatePicker()/hideDatePicker()** qui signaleront à la tableview l'insertion/la suppression d'une ligne à l'indice 2 de la section 1. Le picker devra afficher la date et l'heure de l'item en cours de réglage.
- L'appel de cette méthode provoque des échanges avec le delegate et la datasource de la tableview dont aucune méthode n'est encore implémentée car cette dernière est configurée pour utiliser des cellules statiques. Il est nécessaire de redéfinir certaines de ces méthodes pour ne gérer que la cellule qui contiendra le datePicker et déléguer à la super classe la gestion des cellules statiques.
- Ajouter une vue supplémentaire constituée d'une Table View Cell en faisant un glisser-déposer d'une table view Cell vers la zone supérieure de la scène :



- Ajouter un Date Picker dedans et redimensionner la cellule pour qu'elle fasse la même hauteur que le picker. (162 points) Créer un outlet vers la cellule ainsi que vers le datePicker.



- Redéfinir `tableView(_:willSelectRowAtIndexPath:)` pour accepter les sélections sur la cellule portant le label Due Date uniquement
- Redéfinir `tableView(_:cellForRowAtIndexPath:)` pour retourner la cellule référencée par l'outlet si c'est celle qui est demandée (déléguer à la super classe sinon)
- Redéfinir `tableView(_:numberOfRowsInSection:)` et retourner 2 ou 3 lignes pour la section 1 en fonction de la visibilité du picker (déléguer à la super classe sinon)
- Redéfinir `tableView(_:heightForRowAtIndexPath:)` sur le même modèle (ajouter un pixel à la hauteur de la cellule pour tenir compte du trait) : utiliser `datePicker.intrinsicContentSize.height`
- Redéfinir `tableView(_:didSelectRowAtIndexPath:)` pour gérer l'apparition et la disparition de la cellule contenant le picker et désélectionner la ligne pour qu'elle ne reste pas grisée.
- Redéfinir également `tableView(_:indentationLevelForRowAtIndexPath)` et retourner pour la cellule à l'indice 2 de la section 1 la même valeur que pour l'indice 0 et déléguer à la super classe dans tous les autres cas. Cette méthode doit être implémentée car elle est appelée par la table view. Si elle n'est pas redéfinie, lors de son appel pour la nouvelle ligne, cela provoque un crash du système.

### *Tester l'application*

98. Ajouter une action **dateChanged** appelée lorsque la date ou l'heure du picker sont modifiées afin de mettre à jour le label `dueDateLabel`
99. Modifier la couleur du label `dueDateLabel` quand le picker est visible (utiliser la teinte globale de l'application). Pour grouper les opérations de rafraichissement de la tableview (rechargement de la cellule qui a changé de couleur et ajout ou suppression d'une cellule), les appels à `insertRowsAtIndexPaths`, `reloadRowsAtIndexPaths` doivent être encadrés par des appels à `tableView.beginUpdates()` et `tableView.endUpdates()`
100. Faire disparaître le date picker lorsque le champ de texte est sélectionné (à l'apparition du clavier).

### *Tester l'application (vérifier la bonne gestion du bouton Done, du clavier et de la couleur du label)*

#### **Gestion des notifications à l'heure d'échéance**

101. Demander l'autorisation à l'utilisateur de lui envoyer des notifications lors du premier lancement de l'application. Il n'est pas nécessaire d'enregistrer dans les préférences que la demande a déjà été faite car le système s'en charge. L'appel à la méthode `requestAuthorization` peut donc être effectué autant de fois que voulu.
102. Ajouter une méthode **`scheduleNotification()`** à la classe **`CheckListItem`** qui sera appelée systématiquement lors de l'appui sur le bouton Done (lors de la création ou de l'édition d'un item). Cette méthode devra :
  - Supprimer les éventuelles notifications déjà existantes
  - Reprogrammer une notification si `shouldRemind` est vrai et si la date d'échéance est dans le futur.
103. Supprimer l'éventuelle notification locale associée à une tâche lors de sa suppression. On pourra utiliser la méthode `deinit` qui est automatiquement appelée pour faire le ménage lorsqu'un objet est « détruit ».