

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, numberOfRowsInSection 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 *table view delegate* 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

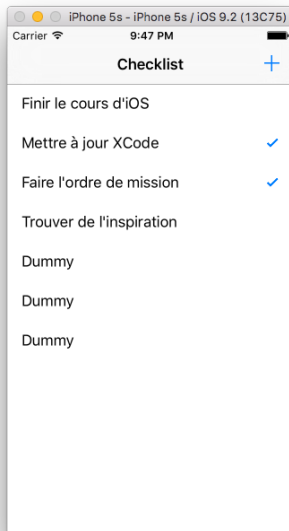
Mise en place du contrôleur de navigation et du bouton ajout

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*

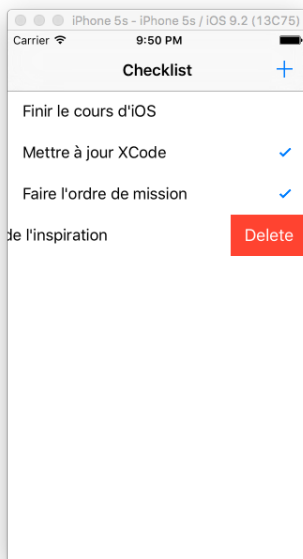
Tester l'application



Suppression des items

13. 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



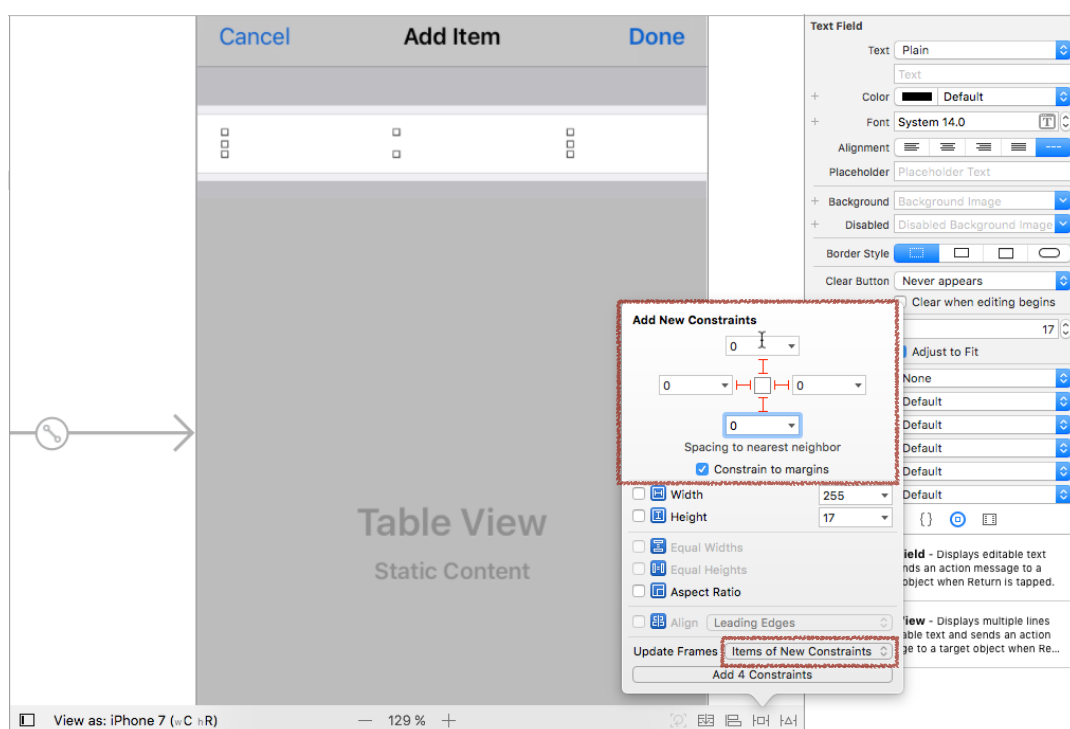
Ecran d'ajout d'un nouvel item

14. 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.
15. 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

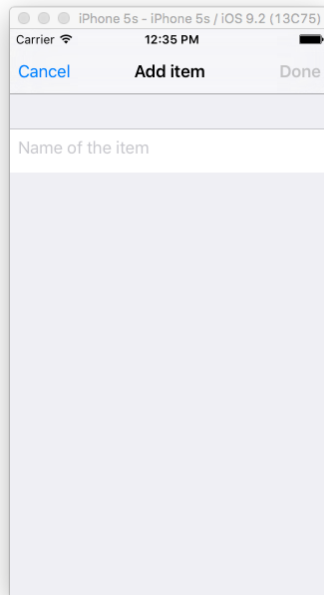
28. Déclarer un protocole **`AddItemViewControllerDelegate`** (dans le même fichier que le contrôleur) qui définira deux méthodes pour gérer l'annulation et la validation d'un nouvel item.

```
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

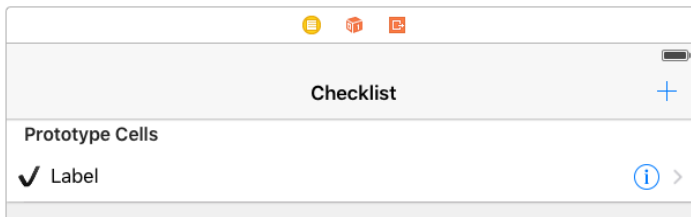
On réutilisera le même contrôleur de vue que pour l'ajout. Il sera nécessaire de lui apporter des modifications pour :

- 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).

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).*