Le stockage des données sur le nuage…
Mise en œuvre du iCloud

Brève présentation d'iCloud

iCloud est un service gratuit (dans la limite de 5Go par **Apple ID**) qui permet aux utilisateurs de matériels Apple d'accéder à leurs données personnelles depuis n'importe lequel de leurs matériels et ceci automatiquement via un **Apple ID**. Le service assure la synchronisation des données vers les différents matériels.

iCloud, fait actuellement partie intégrante des système d'exploitation d'Apple. Apple fournit l'infrastructure de serveurs, la sauvegarde des données et des comptes d'utilisateur.

La sécurité sur iCloud

	Chiffrement		
Données	**Transfert**	**serveur**	**Remarques**
Calendriers	Oui	Oui	Chiffrement minimal AES 128 bits
Contacts	Oui	Oui	
Signets	Oui	Oui	
Rappels	Oui	Oui	
Flux de photos	Oui	Oui	
Documents dans le nuage	**Oui**	**Oui**	
Sauvegarde	Oui	Oui	
Localiser mon iPhone	Oui	Oui	
Localiser mes amis	Oui	Oui	
iCloud.com	Oui	S. O.	Toutes les sessions à l'adresse iCloud.com sont chiffrées avec le protocole SSL. Les données accessibles sur iCloud.com sont chiffrées sur le serveur, comme indiqué dans ce tableau.
Accès à mon Mac	Oui	S. O.	La fonctionnalité Accès à mon Mac ne stocke pas de données dans iCloud. Les données obtenues auprès des autres ordinateurs sont chiffrées avec le protocole SSL lors de leur transfert.
iTunes dans le nuage	Oui	S. O.	Les fichiers de musique achetés ou mis en correspondance ne sont pas chiffrés sur le serveur, car ils ne contiennent pas d'informations personnelles.
Mail et Notes	Oui	Non	Le trafic entre vos appareils et les applications Mail et Notes d'iCloud est chiffré avec le protocole SSL. Conformément aux pratiques courantes de l'industrie, iCloud ne chiffre pas les données stockées sur les serveurs de messagerie IMAP. Tous les clients de messagerie Apple prennent en charge le chiffrement S/MIME facultatif.

Les trois types de stockage sur iCloud[1]

iCloud supporte les trois types de stockage suivants :

- ✓ **Key-value storage**, stockage de couples clé-valeur généralement utilisé pour le stockage de valeurs de configuration, de préférences et pour la gestion de la persistance. A n'utiliser que pour de petites quantités de données.

[1] Depuis « iCloud Design Guide »
https://developer.apple.com/library/ios/#documentation/General/Conceptual/iCloudDesignGuide/Chapters/Introduction.html

- ✓ **Document storage**, stockage de documents utilisé pour des documents de contenus basés sur des fichiers utilisateur (présentations, documents de traitement de texte, schémas, etc…).
- ✓ **Core Data storage**, stockage de données structurées, utilisé pour la mise en place de solutions de bases de données multi-périphériques de contenu structuré et synchronisé.

Prise en charge des flux de travail utilisateur

L'adoption d'iCloud dans une application permet aux utilisateurs de commencer un travail sur un document iCloud (workflow) depuis un appareil et de le finir sur un autre.

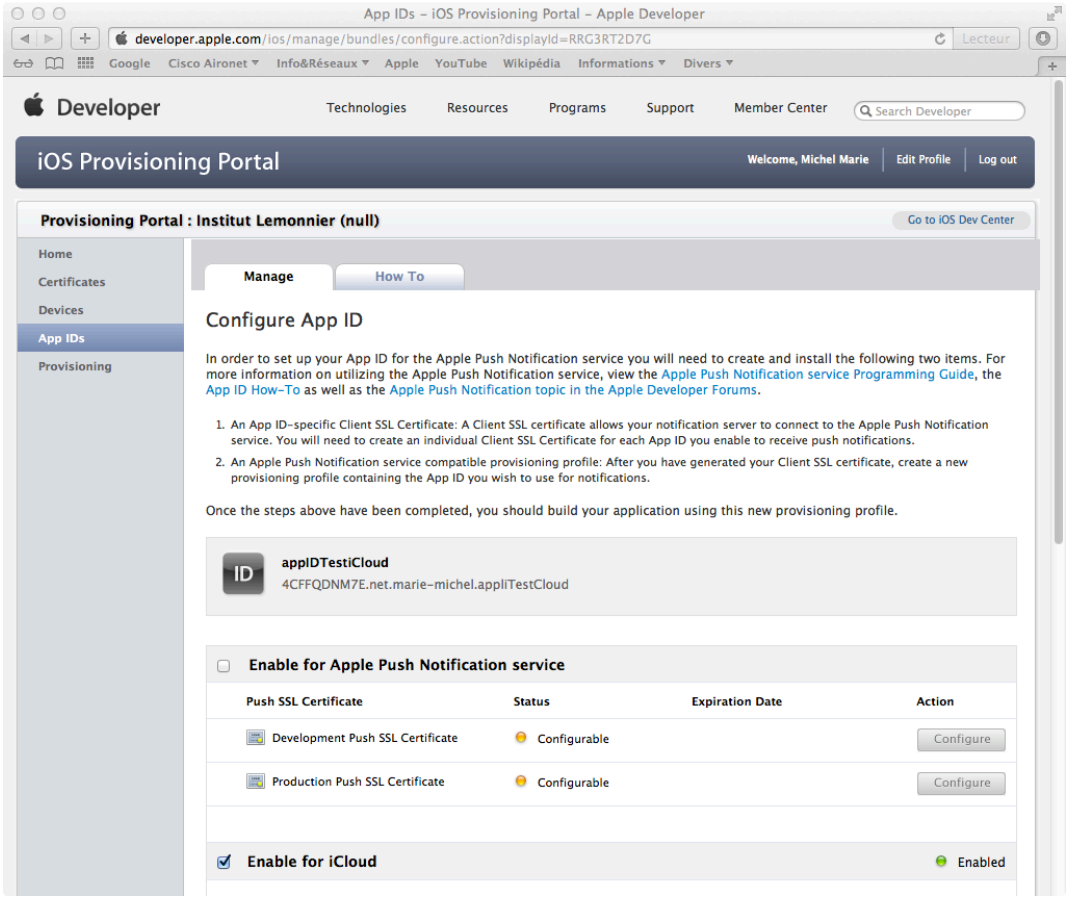
Prenons l'exemple d'une application lecteur de podcasts. Un utilisateur souscrit à un podcast sur son iPhone et écoute les vingt premières minutes sur le trajet du travail. Au bureau, il lance la même application sur son iMac qui reprend automatiquement la lecture au point où il s'était arrêté. Ce premier exemple correspond à une application de type **Key-value storage.**

Prenons un autre exemple correspondant à une application de dessin. Dans la matinée, un architecte en visite chez son client esquisse des croquis sur son iPad. De retour à son bureau, il démarre l 'application sur son iMac. Toutes les nouvelles esquisses sont déjà là, attendant d'être complétées. Ce second exemple correspond à une application de type **Document storage.**

iCloud, les fondamentaux

Le profil de provisionnement

Pour commencer à développer une application iCloud, vous devez disposer d'un profil de provisionnement autorisant les accès aux serveurs iCloud d'Apple et d'un **ID d'application** approprié.



Il n'est pas permis d'utiliser le caractère « * » comme **App ID** des applications iCloud. Un profil distinct devra donc être créé pour chaque application iCloud.

Le transfert automatique et sécurisé des données sur iCloud

Lorsque vous adoptez iCloud, le système d'exploitation démarre et gère le téléchargement de données pour les périphériques connectés à un compte iCloud. Votre application ne communique pas directement avec les serveurs d'iCloud et, dans la plupart des cas, n'invoque pas le **upload/download** des données. Le processus fonctionne comme suit :

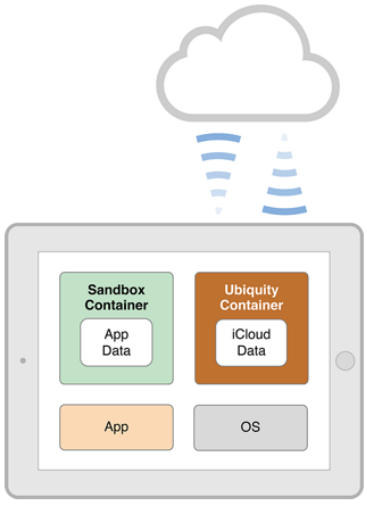
- ✓ vous configurez votre application pour accéder à des emplacements spéciaux du système de fichiers local réservés aux transferts iCloud, ces dossiers sont connus sous le nom de « **conteneurs ubiquité** »,
- ✓ vous concevez votre application afin qu'elle prenne en compte de manière adéquate les changements liés à la disponibilité d'iCloud et les changements liés aux données (déplacement, modification, suppression), ceci à l'aide d'observateurs et de notifications,
- ✓ votre application lit et écrit sur ses « conteneurs ubiquité » en utilisant des API qui permettent la coordination des données,
- ✓ le système d'exploitation transfère automatiquement les données vers et à partir iCloud selon les besoins.

Remarque :

Dans iOS, il y a une exception au transfert automatique de données sur iCloud. Pour le premier téléchargement d'un document basé sur iCloud, l'application doit demander activement le document.

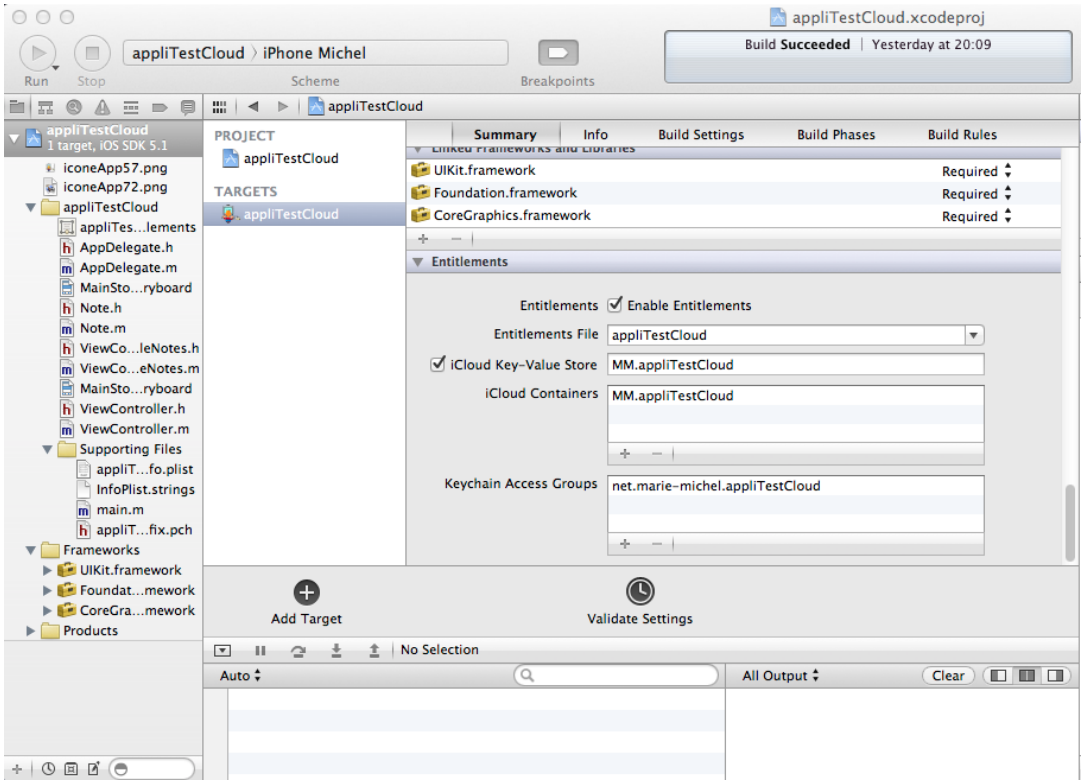
Le « conteneur d'ubiquité »

Pour sauvegarder des données sur iCloud, une application place les données dans le « conteneur ubiquité ». Un « conteneur ubiquité » correspond à la représentation locale du stockage iCloud correspondant. Il est à l'extérieur du conteneur de documents « sandbox » de votre application.



Demande d'accès au stockage iCloud

L'accès au « conteneur ubiquité » nécessite des autorisations de sécurité pour l'application. Elles sont fournies par le profil d'approvisionnement si celui-ci a été configuré comme tel.
La demande des droits se fait dans la page « résumé » des projets XCode en sélectionnant la cible du projet « target » comme dans l'exemple ci-dessous :

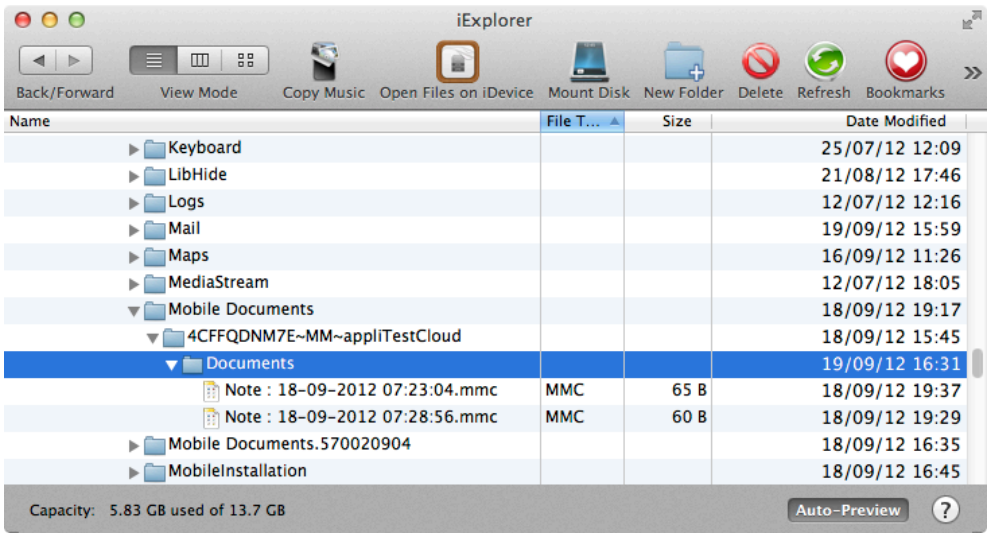


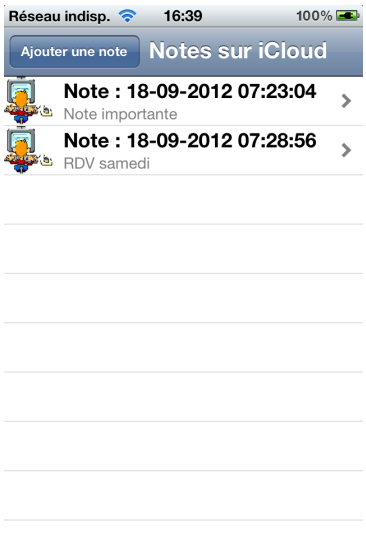
Il existe deux types de droit iCloud, correspondant aux deux types de stockage iCloud :

- ✓ **iCloud key-value storage**, cochez la case **iCloud key-value Store** dans l'éditeur de cible Xcode,
- ✓ **iCloud document storage** (**iCloud Containers**), afin de permettre le stockage des documents (approprié pour les documents visibles par l'utilisateur, base de données **Core Data**, ou d'autres basés sur des fichiers de données). cliquez sur le bouton '+' de la liste « iCloud containers » dans l'éditeur de cible Xcode. La chaîne identifie le « conteneur d'ubiquité » principal de votre application, et, par défaut, est basé sur l'identifiant de bundle de votre application.

L'exemple précédent autorise donc les deux types de stockage.

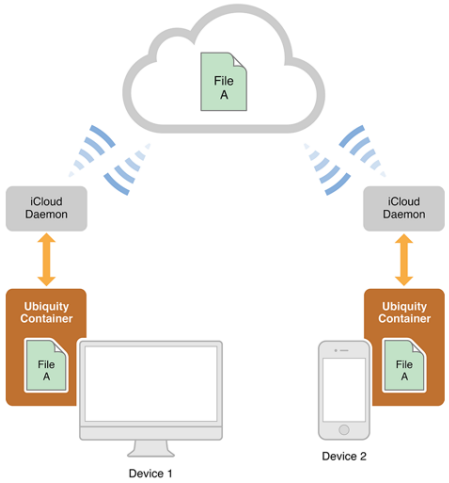
Ci-dessous une application iCloud permettant la gestion de notes affichant deux documents « **notes** » et leur visualisation dans le système de fichier du iDevice avec l'outil **iExplorer**. Attention cependant, ces fichiers sont visualisables car l'IOS du périphérique a été « déprotégé ».





Le système gère le stockage local iCloud

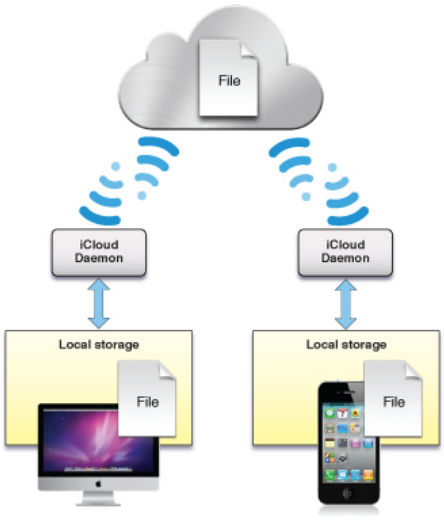
Les données d'un utilisateur iCloud vivent sur les serveurs d'Apple, et un cache de celles-ci vit localement (dans les « conteneurs ubiquité ») sur chacun des dispositifs de l'utilisateur comme le montre la figure suivante. La mise en cache locale des données iCloud permet à un utilisateur de continuer à travailler même lorsque le réseau n'est pas disponible, comme lorsque le mode Avion est activé par exemple.



Du fait que le cache local iCloud est partagé avec les autres fichiers sur un périphérique, dans certains cas ce stockage peut ne pas être suffisant. Le système répond à cette question automatiquement, par le maintien d'un sous-ensemble optimisé de fichiers localement. Dans le même temps, le système conserve toutes les métadonnées en local.
Pour gérer le manque de place en local, le système peut expulser un fichier à partir de son « répertoire ubiquité » si ce fichier n'est pas utilisé et que l'espace local est nécessaire pour un autre fichier que l'utilisateur souhaite utiliser.
Cependant les métadonnées mises à jour pour le fichier expulsé restent en local. L'utilisateur peut toujours voir le nom et d'autres informations pour le fichier expulsé, et il peut l'ouvrir à la condition que l'application soit connectée au « Cloud ».

Utiliser iCloud pour le stockage de documents

Le stockage de documents iCloud permet de déplacer des fichiers et des répertoires sur le compte iCloud de l'utilisateur. Les modifications apportées à un fichier ou à un répertoire sur un périphérique sont stockées localement et ensuite « poussées » vers iCloud à l'aide d'une tâche ou « démon » local. Le transfert des fichiers vers et à partir de chaque dispositif est transparent pour l'application.



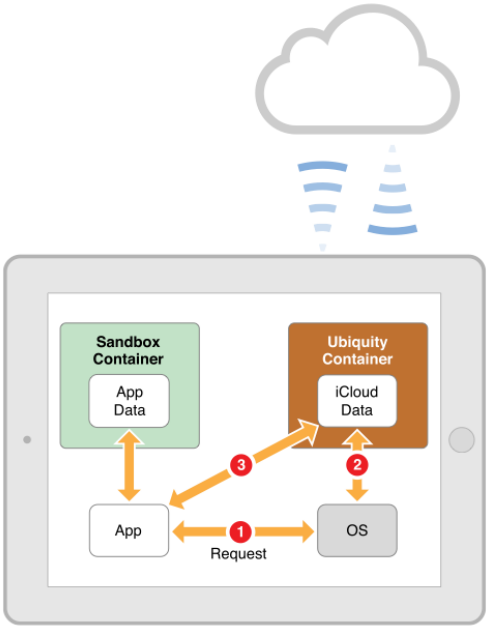
Vérifier la disponibilité d'iCloud au démarrage de l'application

Au démarrage d'une application utilisant iCloud il est nécessaire de vérifier la disponibilité d'iCloud. Ceci se fait au sein des méthodes de l'application :

- ✓ **didFinishLaunchingWithOptions: (iOS),**
- ✓ **applicationDidFinishLaunching:** (OS X).

Ceci se fait à l'aide de la méthode **URLForUbiquityContainerIdentifier:** de la classe **NSFileManager**, méthode permettant de déterminer si iCloud est activé. L'appel de cette méthode est également nécessaire pour étendre votre « sandbox » d'application avec les conteneurs iCloud locaux.

La première fois que vous appelez la méthode **URLForUbiquityContainerIdentifier:** iOS étend votre « sandbox » de l'application pour inclure le répertoire conteneur. Ainsi, il est important que vous appeliez cette méthode au moins une fois pour vous assurer que iCloud est activé et que votre répertoire de « conteneur ubiquité » principal est accessible.
Si votre application accède à des répertoires « conteneur ubiquité » multiples, vous devez appeler la méthode une fois pour chaque répertoire.



1 : appel de **URLForUbiquityContainerIdentifier:**, test des autorisations,

2 : l'application est autorisée iCloud, l'iOS crée le conteneur ubiquité pour celle-ci.

3 : l'application peut lire/écrire dans son conteneur ubiquité en utilisant le chemin retourné par la méthode **URLForUbiquityContainerIdentifier:**, ces données seront automatiquement synchronisées sur le « cloud ».

Exemple :

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:
                                                    (NSDictionary *)launchOptions
{
 NSURL *urliCloudIdentif = [[NSFileManager defaultManager]
                                        URLForUbiquityContainerIdentifier:nil];
    if (urliCloudIdentif) {
        NSLog(@"Acces iCloud autorise : %@", urliCloudIdentif);
    }
    else {
        NSLog(@"Pas d'acces iCloud.");
        UIAlertView* alertView = [[UIAlertView alloc] initWithTitle:@"Accès iCloud
!"
          message:@"Ce matériel ne dispose pas d'un accès iCloud valide..."
          delegate:self
          cancelButtonTitle:@"OK" otherButtonTitles:nil];
        [alertView show];
    }
    return YES;
```

```
}
```

Rendre les documents compatibles avec iCloud, le présentateur et le coordinateur de fichiers…

Tous les fichiers et répertoires stockés sur iCloud doivent-être gérés par un objet présentateur de fichiers (classe **NSFilePresenter**). Dans la situation ou plusieurs applications travaillent simultanément sur un même document iCloud, les modifications apportées aux fichiers et répertoires doivent être coordonnées par un objet coordonnateur de fichier (classe **NSFileCoordinator**).

- ✓ Un présentateur de fichier est un objet qui adopte le protocole **NSFilePresenter**. Il agit comme un agent responsable d'un fichier, lorsqu'une source externe veut modifier un fichier, le présentateur de fichier associé est informé et assure les tâches de compatibilité nécessaires.
- ✓ Lorsque votre application veut modifier un fichier, s'il y a risque que celui-ci soit en cours d'utilisation par une autre application il faut verrouiller le fichier en effectuant les changements à travers un objet **NSFileCoordinator**. Le coordonnateur de fichier empêche les sources externes de modifier le fichier en même temps et délivre les notifications pertinentes aux présentateurs de fichiers.

La classe **UIDocument** implémente les classes **NSFilePresenter** et **NSFileCoordinator**[2].
Le moyen le plus simple pour gérer des documents sur le « Cloud » consiste donc à utiliser des objets dérivés de la classe **UIDocument.**

La création d'une classe dérivée de **UIDocument** nécessite la surcharge de 2 méthodes. Ces méthodes sont :

- ✓ **contentsForType:error:** pour les opérations d'écriture, cette méthode doit retourner les données à sauvegarder.
  Les données retournées sont :
  - soit un objet **NSData** pour des structures de données simples (fichier texte format JSON ou XML par exemple),
  - soit un objet **NSFileWrapper** pour des structures complexes type paquetage avec des données textes et binaires (texte + images par exemple).

- ✓ **loadFromContents:ofType:error :** pour les opérations de lecture du document dans le modèle de données de l'application.
  Les données sont passées au premier paramètre de la méthode, le type de ce paramètre déclaré (**id**) est également soit **NSdata** soit **NSFileWrapper**.

Il est possible d'activer l'auto-sauvegarde des documents lors de changements de ceux-ci. Pour cela une première méthode consiste à appeler la méthode **updateChangeCount:** à la fin du code de modification. La classe **UIDocument** contrôle périodiquement la propriété **hasUnsavedChanges** affectée par la méthode précédente afin de décider si l'opération de sauvegarde doit être activée.

Une autre solution consiste à utiliser la classe **NSUndoManager**, classe accessible à l'aide de la propriété **undoManager** de la classe **UIDocument**. Cet objet présente l'avantage de pouvoir annuler les modifications apportées (mode **undo/redo**).

Exemple : *la classe « Note » suivante représente un document constitué d'un sujet et d'un message. Le stockage de ces données simples dans un fichier se fait en utilisant le format JSON, le type NSData*

[2] http://developer.apple.com/library/mac/#documentation/FileManagement/Conceptual/FileSystemProgrammingGUide/FileCoordinators/FileCoordinators.html

est donc utilisé pour encapsuler les données. L'auto-sauvegarde des documents est activée par enregistrement auprès de l' « **undoManager** » via les propriétés d'écriture.

Exemple de fichier JSON d'une « note » :

```
{
 "Message" : "http://www.fr.lastminute.com/site/week-end-pas-cher",
 "Sujet" : "Préparer son Week-end"
}
```

```
//  Notes.h
#import <UIKit/UIKit.h>
@interface Note : UIDocument
@property (copy,nonatomic) NSString* noteMessage;
@property (copy,nonatomic) NSString* noteSujet;
@end
```

```
//  Notes.m
#import "Note.h"
@implementation Note
@synthesize noteMessage;
@synthesize noteSujet;

// Délégué appelé lorsque l'application lit les données depuis le système de
fichier
- (BOOL)loadFromContents:(id)contents ofType:(NSString *)typeName
error:(NSError **)outError
{
    if ([contents length] > 0) {
        NSError* erreur;
        id data = [NSJSONSerialization JSONObjectWithData:contents
                  options:NSJSONReadingMutableLeaves error:&erreur];
        self.noteSujet = [data valueForKey:@"Sujet"];
        self.noteMessage = [data valueForKey:@"Message"];
    } else {
        // Contenu par défaut lors de la création de la note (vide)
        self.noteMessage = @"Ajouter votre note...";
        self.noteSujet = @"Ajouter votre sujet...";
    }
    return YES;
}

// Délégué appelé lorsque l'application auto-sauvegarde le contenu d'une
note
- (id)contentsForType:(NSString *)typeName error:(NSError **)outError
{
     if ([self.noteMessage length] == 0) {
        self.noteMessage = @"Ajouter votre note...";
        self.noteSujet = @"Ajouter votre sujet...";
    }
NSDictionary *data = [NSDictionary
dictionaryWithObjectsAndKeys:self.noteSujet,
                        @"Sujet",self.noteMessage, @"Message", nil];
    NSError* erreur;
    NSData *jsonData = [NSJSONSerialization dataWithJSONObject:data
                        options:NSJSONWritingPrettyPrinted error:&erreur];
    return jsonData;
}

// Surcharge propriété d'écriture de l'attribut noteMessage
- (void)setNoteMessage:(NSString *)newNoteMessage {
    NSString* oldMessage = noteMessage;
    noteMessage = [newNoteMessage copy];
    // Register the undo operation.
    [self.undoManager setActionName:@"NoteMessage Change"];
    [self.undoManager registerUndoWithTarget:self
selector:@selector(setNoteMessage:)
                                      object:oldMessage];
}
```

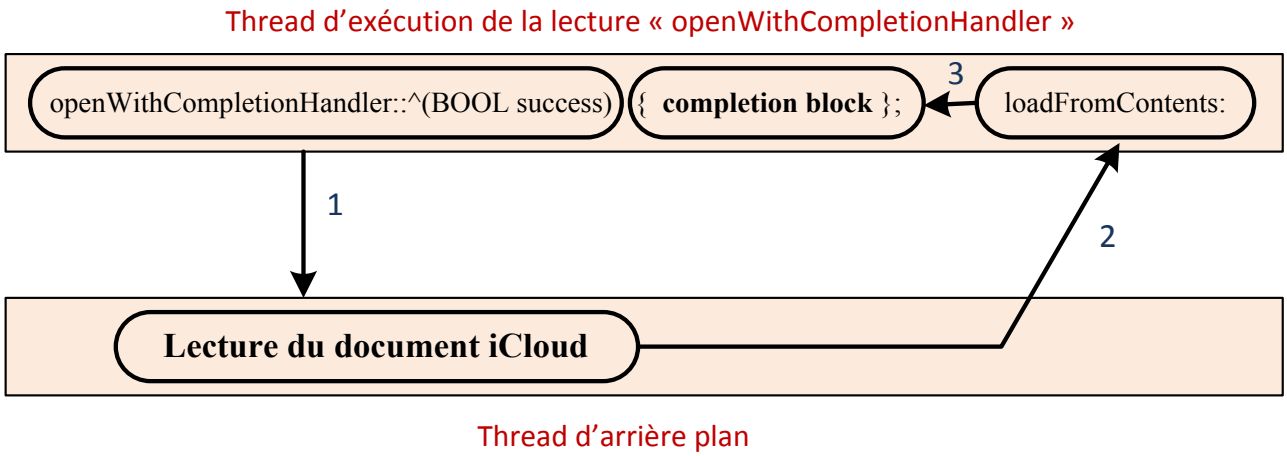
```
// Surcharge propriété d'écriture de l'attribut noteSujet
- (void)setNoteSujet:(NSString *)newNoteSujet {
    NSString* oldSujet = noteSujet;
    noteSujet = [newNoteSujet copy];
    // Register the undo operation.
    [self.undoManager setActionName:@"NoteSujet Change"];
    [self.undoManager registerUndoWithTarget:self
selector:@selector(setNoteSujet:)
                                        object:oldSujet];
}
@end
```

Le chargement d'un document iCloud, « openWithCompletionHandler : » :

```
- (void)openWithCompletionHandler:(void (^)(BOOL success))completionHandler
```

Le chargement d'un document iCloud est réalisé par l'appel de la méthode **openWithCompletionHandler**. Celle-ci active le chargement en « tâche de fond » puis lorsque la lecture est terminée active la méthode surchargée **loadFromContents**.

La sortie de **loadFromContents** déclenche l'exécution du bloc « **completion handler** » de la méthode **openWithCompletionHandler** en lui passant en paramètre la valeur de retour de **loadFromContents**.



Exemple :

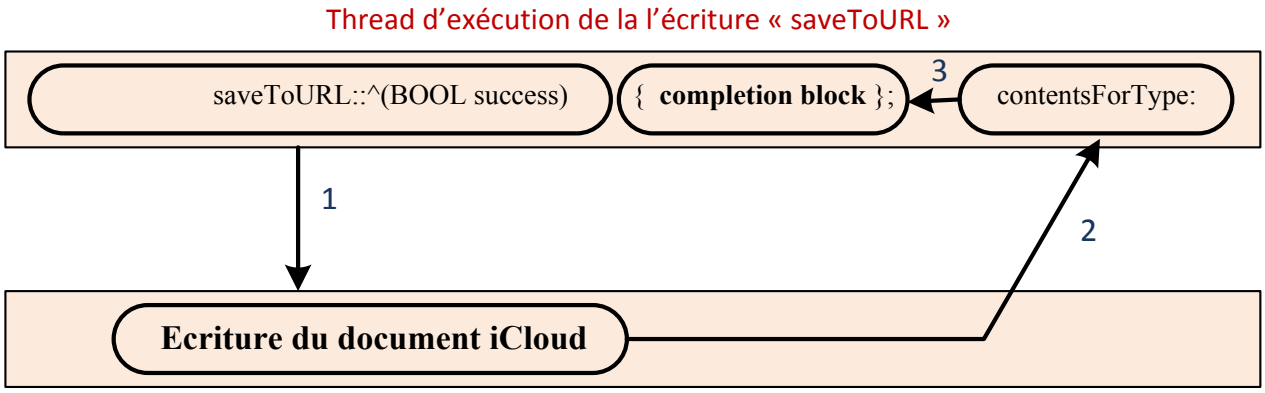
```
// Création d'un objet Note* specialisation de UIDocument
Note* note = [[Note alloc]initWithFileURL:noteURL];

// Ouverture du document pour chargement dans conteneur ubiquité
NSFileManager *fm = [NSFileManager defaultManager];
if ([fm fileExistsAtPath:[noteURL path]])
  {
        [note openWithCompletionHandler:^(BOOL success) { // bloc completion
handler
            if(success==NO)
                NSLog(@"Echec ouverture iCloud");
            else
                NSLog(@"Ouverture iCloud OK");
        }];
   }
```

La création d'un document iCloud, « saveToURL : » :

```
-(void)saveToURL:(NSURL *)url
forSaveOperation:(UIDocumentSaveOperation)saveOperation
completionHandler:(void (^)(BOOL success))completionHandler
```

La sauvegarde d'un document sur le cloud reprend le même procédé, elle est initiée par la méthode **saveToURL** qui active l'écriture en « tâche de fond ». Lorsque l'écriture est terminée la méthode surchargée **contentsForType** est appelée puis en fin d'exécution celle-ci déclenchera l'exécution du bloc « **completion handler** » de la méthode **saveToURL**.



Exemple :

```
// Création d'un nouveau document
[note saveToURL:noteURL forSaveOperation:UIDocumentSaveForCreating
completionHandler:^(BOOL success) {  // bloc completion handler
          if(success==NO)
              NSLog(@"Echec création document iCloud");
          else
              NSLog(@"Création document iCloud OK");
     }];
```

La fermeture asynchrone avec sauvegarde d'un document iCloud, « closeWithCompletionHandler : » :

```
- (void)closeWithCompletionHandler:(void (^)(BOOL success))completionHandler
```

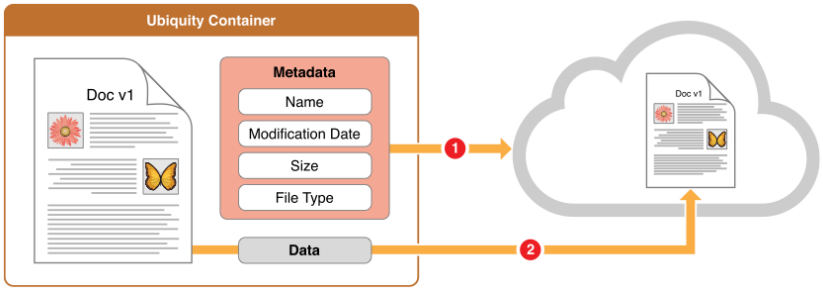
Cette méthode réalise l'enregistrement d'un document en toute sécurité et de façon asynchrone. Une fois l'opération de sauvegarde terminée, le code du bloc « **completionHandler** » est exécuté, il reçoit un paramètre signalant la réussite ou l'échec de la sauvegarde.

Exemple :

```
// Modification du document iCloud.
note.noteSujet = @"le sujet…";
note.noteMessage = @"Le message…";
// Sauvegarde puis fermeture du document iCloud.
[note closeWithCompletionHandler:^(BOOL success) {
     if(success==NO)
         NSLog(@"Echec fermeture iCloud");
     else
         NSLog(@"Fermeture iCloud OK");
}];
```

Synchronisation des documents iCloud

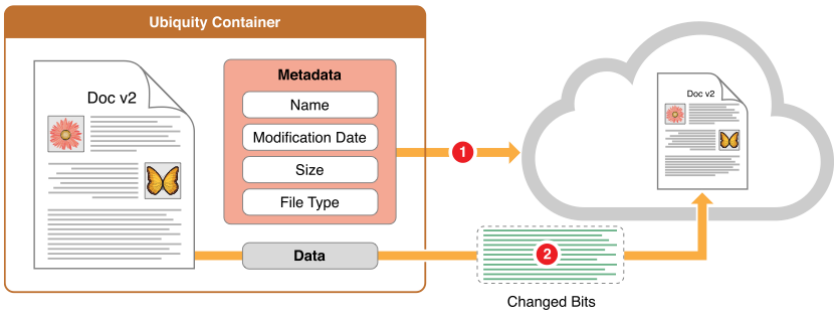
La première fois que votre application ajoute un document sur le disque dans son « conteneur ubiquité », le système transfère l'intégralité du fichier ou paquetage fichier sur le serveur iCloud, comme le montre la Figure suivante :



1 : transfert des métadonnées du document (nom du document, date de modification, taille du fichier, type de fichier, ...). Ce transfert des métadonnées a lieu rapidement.

2 : le système envoie les données du document.
L'envoi rapide des métadonnées permet à iCloud de découvrir l'existence d'un nouveau document. En réponse, le serveur iCloud propage rapidement les métadonnées vers les autres périphériques disponibles connectés au même compte iCloud.

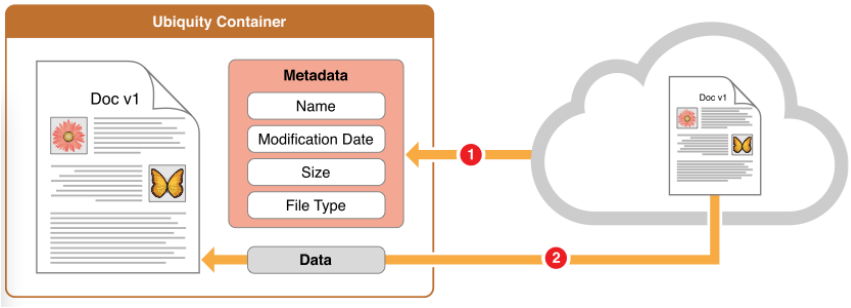
Une fois les données sur le serveur iCloud, iCloud optimise les futurs transferts de ce même document. A chaque changement, iCloud envoie uniquement les métadonnées et les pièces qui ont été modifiées (méthode incrémentielle).



1 : transfert des métadonnées du document.

2 : le système envoie uniquement les données qui ont été modifiées.

Le téléchargement à partir d'iCloud d'un nouveau document créé par un autre périphérique fonctionne comme suit :



1 : transfert des métadonnées du document.

2 : transfert des données.

Comme nous l'avons déjà signalé le transfert des métadonnées via iCloud est immédiat lors de la création du fichier, par contre le transfert des données peut suivre deux scénarios :

- ✓ Un appareil iOS ne télécharge pas le nouveau fichier automatiquement. Il devra au démarrage de l'application activer une tâche de fond qui servira à scruter les changements survenus sur

les métadonnées, en cas de détection d'un changement l'application sera notifiée, elle devra mettre à jour ses documents et éventuellement son interface utilisateur.

- ✓ Pour un Mac, le téléchargement est automatique pour les fichiers créés par d'autres appareils sur le même compte. Si l'application Mac voit les métadonnées d'un nouveau document, mais que le document lui-même n'est pas encore présent il est probablement déjà en cours de téléchargement par le système pour le besoin de l'application.

La tâche de scrutation des métadonnées utilisera la classe **NSMetadataQuery**.
La classe **NSPredicate** est employée par la classe **NSMetadataQuery** pour spécifier les conditions de recherche et filtrer les documents scrutés.
Cette classe prend en charge les messages de notification suivants (entre autres) :

- ✓ **NSMetadataQueryDidUpdateNotification** posté lorsque un changement est détecté sur le conteneur ubiquité surveillé,
- ✓ **NSMetadataQueryDidFinishGatheringNotification** posté lorsque la première phase de collecte des documents est terminée.

La classe **NSMetadataItem** permet de lire les différents attributs des métadonnées d'un fichier.

Exemple : *cet exemple correspond à une application qui gère des documents iCloud dont le nom respecte le format « **Note : * .mmc** » (* signifiant n'importe quelle séquence alphanumérique autorisée dans les noms de fichier).*
En cas de changement détecté dans le « conteneur ubiquité » le bloc `usingBlock:^(NSNotification __strong *notification)` *est activé.*

```
// Tableau des URL des fichiers du conteneur ubiquité surveillé
@property (strong) NSMutableArray * notesURL;
// Objet de surveillance des métadonnées
NSMetadataQuery*    metadataQuery = [[NSMetadataQuery alloc] init];
// Etendue de surveillance limitée aux conteneurs d'ubiquité
[metadataQuery setSearchScopes:[NSArray
    arrayWithObjects:NSMetadataQueryUbiquitousDocumentsScope, nil]];
// Prédicat filtrant les fichiers du type "Note : *.mmc"
[metadataQuery setPredicate:[NSPredicate
    predicateWithFormat:@"%K LIKE 'Note : *.mmc'",
    NSMetadataItemFSNameKey]];
// Ajout de l'observateur (fonctionnel),message
// NSMetadataQueryDidUpdateNotification
 [[NSNotificationCenter defaultCenter] addObserverForName:
                        NSMetadataQueryDidUpdateNotification
                        object:nil queue:[NSOperationQueue mainQueue]
 usingBlock:^(NSNotification __strong *notification)
     {
         NSLog(@"Contenu modifié dans dossier ubiquitous... ");

      // Désactiver les modifications pendant l'itération des résultats
         [metadataQuery disableUpdates];
         NSMutableArray *arrayURL = [NSMutableArray array];
         for (NSMetadataItem *item in metadataQuery.results)
            {
              [arrayURL addObject: [item
valueForAttribute:NSMetadataItemURLKey]];
               // Pour le fun
                    NSString *url1 = [item
valueForAttribute:NSMetadataItemURLKey];
               NSDate *dateUpdate = [item
               valueForAttribute:NSMetadataItemFSContentChangeDateKey];
              // Affichage pour les tests…
              NSLog(@"URL fichier modifié = %@)", url1);
              NSLog(@"Date modification = %@)", dateUpdate);
             }
         [notesURL removeAllObjects];
         [notesURL addObjectsFromArray:arrayURL];
         [metadataQuery enableUpdates];
```

```
}];
```

Certaines opérations sur le Cloud peuvent-être longues et bloquantes, il est donc souvent nécessaire de réaliser celles-ci en asynchronisme du thread principal de l'application, avec les éventuelles synchronisations que cela peut imposer. Pour cela on dispose du **GCD**.

Exécution asynchrone - GCD « Grand Central Dispatch »

Le **GCD** « **Grand Central Dispatch** », est un système de files d'attente **Cocoa** et **Cocoa Touch** prévu pour le support des exécutions concurrentes sur IOS et OS X.

GCD fournit et gère des files d'attente FIFO dans lesquelles votre application peut soumettre des tâches sous la forme d'objets blocs d'exécution. Les blocs soumis au GCD sont exécutés sur un pool de threads entièrement gérés par le système. GCD propose trois types de files d'attente :

- ✓ File **Principale** : les tâches sont exécutées en série sur le thread principal de votre application.
- ✓ File **Concurrente** : les tâches sont dépilées dans l'ordre FIFO, mais exécutées en concurrence et peuvent donc se terminer dans n'importe quel ordre.
- ✓ File **Série** : les tâches sont exécutées une à la fois dans l'ordre FIFO.

File Principale

La file d'attente principale est créée automatiquement par le système et associée au thread principal de l'application. L'application utilise une des trois méthodes suivantes pour appeler les blocs soumis à la file d'attente principale :

- ✓ **dispatch_main**,
- ✓ **UIApplicationMain** (iOS) ou **NSApplicationMain** (OSx),
- ✓ **CFRunLoopRef**.

File Concurrente

GCD crée automatiquement trois files d'attente globales à l'application qui se différencient par leur niveau de priorité.
L'application fait appel à ces files d'attente en utilisant la fonction **dispatch_get_global_queue**.
Étant donné que ces files d'attente sont globales pour votre application, vous n'avez pas besoin de les conserver ni de les libérer.
Sous OS X v10.7 ou ultérieure, vous pouvez également créer vos propres files d'attente concurrentes pour une utilisation dans vos propres modules de code.

File Série

Il n'y a pas de file d'attente Série automatiquement créée par l'application. Votre application doit explicitement créer et gérer des files d'attente en série. Elle peut en créer autant que nécessaire.
La création d'une file d'attente série fait appel à la fonction **dispatch_queue_create**.

Soumettre une tâche

Pour soumettre une tâche à une file les méthodes à employer sont :

- ✓ **dispatch_async :** soumet un bloc pour exécution dans une file et rend la main immédiatement,
- ✓ **dispatch_sync :** soumet un bloc pour exécution dans une file et attend la fin du bloc pour rendre la main.

Synchronisation des threads

Le GCD dispose d'un système de sémaphore à compte pour la synchronisation des threads. Pour ceci 3 méthodes sont nécessaires :

- ✓ **dispatch_semaphore_create :** pour la création d'un sémaphore à compte avec une valeur initiale choisie,
- ✓ **dispatch_semaphore_wait :** pour la prise du sémaphore,
- ✓ **dispatch_semaphore_signal :** pour la restitution du sémaphore.

Exemple :

Méthode de suppression d'un document « iCloud » référencé dans une « Table View » utilisant une file concurrente. Le thread courant attend la fin de la suppression pour mettre à jour la « Table View » :

```
dispatch_semaphore_t sema = dispatch_semaphore_create(0);
// Ne pas utiliser NSFileCoordinator dans l'App Main Queue
dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_HI
     GH, 0), ^{
         NSError* erreur = nil;
     NSFileManager *fm = [[NSFileManager alloc] init];
     [fm removeItemAtURL:newURL error:&erreur];
     // Libération de l'App Main Thread
      dispatch_semaphore_signal(sema);
     if(!erreur)
           NSLog(@"Note supprimée de iCloud");
      else
           NSLog(@"Erreur suppression note iCloud
     %@",erreur.description);
      });
      // Attente de fin de la suppression
      dispatch_semaphore_wait(sema, DISPATCH_TIME_FOREVER);
      // Supprime la note de la Table
      [notesURL removeObjectAtIndex:indexPath.row];
      // puis supprime la cellule (toujours après la data Table)
      [tableView deleteRowsAtIndexPaths:[NSArray
     arrayWithObject:indexPath]
     withRowAnimation:UITableViewRowAnimationFade];
```

Gestion des conflits sur les versions de documents iCloud

Lorsqu'un utilisateur a installé une même application utilisant le stockage de documents iCloud sur plusieurs appareils, ceci peut amener des conflits entre les différentes versions d'un même document. Rappelons qu'une application met à jour un fichier de document dans le répertoire conteneur local et ces changements sont ensuite transmis à iCloud. Si les documents sont modifiés sur différents appareils en mode déconnecté (mode avion par exemple), lorsque vous passerez de nouveau en mode connecté iCloud remarquera un conflit et le notifiera à l'application.

Les mécanismes à mettre en œuvre pour gérer cette situation sont les suivants :

- ✓ pour que l'application soit avertie des conflits de version sur les documents elle doit observer la notification de message **UIDocumentStateChangedNotification**,

- ✓ au sein du traitement associé à cet événement il est possible de détecter un conflit de version à l'aide de la propriété **documentState** de la classe **UIDocument**, celle-ci prenant la valeur **UIDocumentStateInConflict**,
- ✓ la classe **NSFileVersion** permet alors de récupérer la liste des versions d'un même document et d'obtenir pour chaque version la date de dernière modification. Liberté est donnée au concepteur de présenter les versions à l'utilisateur et de le laisser prendre en compte la version à garder ou de « discrètement » ne garder que la dernière version…

Exemple : *cet exemple installe la capture des notifications associées au changement d'état des documents. Si la notification correspond à un conflit de version seule la dernière version est sauvegardée, les autres sont supprimées (pas si simple…).*

```
// Observation de la notification « changement d'état d'un document »
 [[NSNotificationCenter defaultCenter] addObserver:self
                        selector:@selector(processStateDocChange:)
                        name:UIDocumentStateChangedNotification object:nil];

// Méthode active en cas de changement d'état
       - (void)processStateDocChange: (NSNotification *)notification
       {
           for(int i=0; i<[notesURL count];i++)
           {
               NSURL* url = [notesURL objectAtIndex:i];
               Note* note = [[Note alloc ]initWithFileURL:url.noteURL];

               if([note documentState] == UIDocumentStateInConflict )
               {
                   NSArray* versionsInConflit = [NSFileVersion
                         unresolvedConflictVersionsOfItemAtURL:[note fileURL]];
                  NSFileVersion* versionCourante = [NSFileVersion
                         currentVersionOfItemAtURL:[note  fileURL] ];
                   NSFileVersion* versionPlusRecente = versionCourante;
                   NSLog(@"date courante = %@",[versionCourante
                                                modificationDate].description );
                   if (versionsInConflit)
                   {
                       NSLog(@"Versions en conflit");
                       for (NSFileVersion* version in versionsInConflit)
                       {
                           NSLog(@"modif date = %@",
                                 [version modificationDate].description );
                           if ([[version modificationDate]
                                compare:[versionPlusRecente modificationDate]] ==
                               NSOrderedDescending)
                           {
                               versionPlusRecente = version;
                           }
                       }
                       if (versionPlusRecente != versionCourante)
                       {
                           [self
       resolveConflictWithConflictVersion:versionPlusRecente
                                                        noteEnConflit:note];
                           NSLog(@"versionplusrecente = version");
                       }
                       else
                       {
                           [self resolveConflictWithCurrentVersion:note ];
                           NSLog(@"versionplusrecente != version");
                       }
                   }
               }
           }
       }

       -(void)resolveConflictWithConflictVersion:(NSFileVersion*)fileVersion
                                        noteEnConflit:(Note*)noteEnConflit
```

```
{
    [fileVersion replaceItemAtURL:noteEnConflit.fileURL options:0
error:nil];
    NSArray* conflictVersions = [NSFileVersion

unresolvedConflictVersionsOfItemAtURL:noteEnConflit.fileURL];
    for (NSFileVersion* fileVersion in conflictVersions)
    {
        fileVersion.resolved = YES;
    }

    [NSFileVersion removeOtherVersionsOfItemAtURL:noteEnConflit.fileURL
error:nil];
    [noteEnConflit revertToContentsOfURL:noteEnConflit.fileURL
                       completionHandler:^(BOOL success) {
                           [noteEnConflit
updateChangeCount:UIDocumentChangeDone];
                       }];
}


-(void)resolveConflictWithCurrentVersion:(Note*)noteEnConflit
{
    NSArray* conflictVersions = [NSFileVersion

unresolvedConflictVersionsOfItemAtURL:noteEnConflit.fileURL];

    for (NSFileVersion* fileVersion in conflictVersions)
    {
        fileVersion.resolved = YES;
    }
    [NSFileVersion removeOtherVersionsOfItemAtURL:noteEnConflit.fileURL
error:nil];
}
```