Cours n°6 — Gestion des données distantes

Gestion asynchrone

Pour pallier le fait que certaines opérations peuvent prendre du temps et peuvent donc geler l'application, le SDK iOS propose des facilités pour gérer plusieurs threads. Le thread gérant l'interface est le principal (*main thread*) et ne doit jamais exécuter d'autres opérations que celles liées à l'interface. Les autres opérations doivent être lancées dans des threads séparés.

Pour pouvoir gérer plusieurs opérations à la fois, iOS propose un mécanisme de files (*dispatch queues*) permettant de planifier l'exécution de plusieurs tâches. La file permettant de réaliser des opérations sur l'interface (donc sur le thread principal) s'appelle la *main dispatch queue*.

D'autres files existent et ont une priorité plus ou moins haute, décrit par leur attribut QoS (pour *Quality of Service*). C'est au développeur d'évaluer quel QoS est la plus appropriée selon l'importance de la tâche et son impact sur l'application. (c.f. diapositive 3)

La diapositive 4 vous montre un exemple de code permettant d'exécuter des tâches dans une *dispatch queue* avec une QoS moyenne et une fois l'opération réalisée, mettre à jour l'interface par le biais de la file principale.

Il existe également des files d'opérations. La différence avec celles de threads est que les files d'opérations vous permettent d'annuler ou d'inspecter (« monitorer ») les tâches. De plus, des dépendances peuvent être créées entre les opérations pour utiliser le résultat d'une opération dans une autre.

Dans le cadre du cours, les files d'opérations ne nous seront pas utiles mais elles sont relativement répandues.

Gestion des données distantes

Il est possible de récupérer ou stocker des données de manière distante soit par le biais du service iCloud, proposé par Apple, soit par le biais du réseau en faisant transiter des données depuis ou vers un webservice par exemple.

iCloud

Pour stocker des données ou des documents simplement sans investir du temps de développement dans une application ou de maintenance d'une infrastructure, iCloud est un choix qui peut être intéressant.

Il est également appelé « *Ubiquitous store* » dans la documentation. Son utilisation, même en phase de développement, nécessite la possession d'un compte développeur Apple.

Les données iCloud sont réparties entre base de données privée (celle de l'utilisateur) et publique (celle de l'application). Dans le cadre de la base de données privée, il faut que l'utilisateur soit connecté à son compte iCloud et ait autorisé votre application à accéder à son espace de stockage iCloud pour pouvoir l'utiliser.

Côté SDK, il est possible d'utiliser `NSUbiquitousKeyValueStore` pour stocker des valeurs simples comme `UserDefaults` (c.f. cours n°5), ou d'utiliser CloudKit pour les documents ou les structures plus complexes.

Les avantages et inconvénients majeurs d'iCloud sont listés en diapositive 9. Les diapositives 10 à 12 montrent comment ajouter la possibilité d'utiliser iCloud dans une application. Les diapositives 13 à 15 montrent comment stocker des données simples ou structurées dans iCloud.

La véritable complexité qui peut être rencontrée en utilisant iCloud concerne la gestion des erreurs. Il faut s'assurer de détecter correctement les erreurs de synchronisation et proposer les options adaptées à l'utilisateur pour les résoudre. (c.f. diapositive 16)

Utilisation d'un webservice

Un des gros inconvénients d'iCloud est que c'est une technologie Apple, limitée aux appareils de cette marque. Il peut être intéressant de vouloir rendre les données d'un utilisateur disponible sur d'autres supports que ceux de la marque Apple ; il est dans ce cas possible de développer et mettre en ligne un webservice avec lequel communiquer. Le standard (Rest, GraphQL, etc.) de ce service et le format des réponses n'ont pas réellement d'importance.

Il ne s'agit bien sûr pas d'une solution dénuée de problèmes (c.f. diapositive 18) mais permet d'avoir une réelle flexibilité.

Quelques spécificités sont à considérer cependant lorsque l'on souhaite communiquer avec l'extérieur dans une application iOS. Par défaut, iOS ne supporte que les connexions sécurisées (HTTPS) utilisant le protocole TLS en version 1.2 au minimum. Il est possible de contourner ses limitations pendant la phase de développement. (c.f. diapositives 19 à 21)

Le SDK iOS propose de nombreuses classes pour pouvoir réaliser des appels à des services distants (c.f. diapositive 22). Cependant ces classes sont relativement

fastidieuses à utiliser. Pour faciliter les choses, des bibliothèques existent et peuvent être installées dans un projet iOS via des dépendances externes.

Gestion des dépendances

Dans la plupart des langages de programmation, il existe des outils permettant de gérer facilement le téléchargement et l'utilisation de dépendances externes. Cela permet de ne pas avoir à réinventer la roue et permet aussi de partager du code facilement entre projets. Swift ne fait pas exception et plusieurs outils existent pour importer des dépendances externes.

Parmi les outils disponibles, Cocoa Pods est le plus ancien, Cartage est plus récent et relativement populaire. Cependant, Apple s'est lancé dans le développement de son propre outil, appelé Swift Package Manager et ce dernier est correctement intégré dans Xcode, ce qui en facilite l'utilisation.

Ajouter une dépendance peut se faire simplement via Xcode en passant par le menu File > Swift Packages > Add Package Dependency. (c.f. diapositive 27) Il est possible de spécifier la source de la dépendance externe ainsi que les contraintes de versions. (c.f. diapositives 28 et 29) L'exemple de bibliothèque utilisée ici est Alamofire, un projet permettant de faire des requêtes HTTP facilement.

Il est possible de supprimer, mettre à jour ou ajouter d'autres dépendances depuis le menu « Swift Packages » en cliquant sur l'icône du projet dans le panneau gauche de Xcode. Il faut s'assurer que le projet est sélectionné dans la liste déroulante située en haut à gauche de ce panneau et non une cible (*target*).

Réalisation de requêtes

La bibliothèque Alamofire permet de réaliser très simplement des requêtes vers des webservices et de récupérer la réponse sous forme d'objet Swift. La méthode à utiliser pour réaliser des requêtes, quelqu'en soit la nature, suit une syntaxe relativement facile à comprendre. (c.f. diapositive 31)

Dans le cadre du développement, il peut s'avérer pratique de débugguer une requête qui ne fonctionnerait pas comme souhaité. Dans ce cas, Alamofire propose une méthode permettant d'obtenir la commande `curl` équivalente à la requête lui étant fournie. (c.f. diapositive 32). Il est ainsi possible de lancer la requête depuis un terminal et d'avoir éventuellement plus d'informations sur la nature du ou des éventuels problèmes.

Notifications

Parmi les éléments visuels liés au traitement de données et de tâches de fond, les notifications peuvent être un bon moyen d'indiquer à l'utilisateur qu'une tâche est terminée par exemple.

Les notifications peuvent être locales ou distantes (on parle de notifications *push* dans ce cas). Quelqu'en soit la nature, l'application doit explicitement demander l'autorisation à l'utilisateur pour pouvoir afficher les notifications. Les diapositives 34 et 35 fournissent plus d'informations sur les possibilités et la marche à suivre pour une telle demande d'autorisation.

Les notifications locales sont relativement simples à créer. Il est important de noter cependant que si l'utilisateur est en train d'utiliser l'application et qu'une notification doit s'afficher à ce moment, celle-ci ne s'affichera pas ; l'application doit être en arrière plan ou fermée.

Plusieurs possibilités sont offertes pour planifier ou déclencher des notifications (c.f. diapositive 36). Côté code, il faudra créer la notification en elle même, une requête représentant la planification de la notification et finalement envoyer cette requête au centre de notification de l'appareil courant. (c.f. diapositive 37)

Les notifications distantes sont, quand à elle, un peu plus compliquées à mettre en place. En effet, l'infrastructure se chargeant de garder une connexion internet avec l'appareil de l'utilisateur (et qui va envoyer la notification) est fournie par Apple. Il faut donc, dans un premier temps, posséder un compte développeur Apple et ensuite communiquer avec ce service pour pouvoir envoyer une notification distante (e.g. pour un nouveau message, une offre que l'on souhaite envoyer à l'utilisateur, etc.).

L'envoi de ces notifications se base sur un token qu'il faut obtenir du côté de l'application (dans le `AppDelegate`) et qui va permettre d'identifier l'appareil courant. (c.f. diapositive 39). Plusieurs conseils concernant le stockage du token sont donnés en diapositive 40.

L'envoi de notification se fait en envoyant un document JSON à APNs (Apple Push Notification service). Ce document JSON s'appelle « charge utile » (ou *payload* en anglais) et ne doit idéalement pas contenir de données sensibles.

Il est à noter que, comme pour les autres communications réseaux au sein d'une application iOS, la communication avec APNs se fait avec un serveur HTTPS ayant un certificat en TLS 1.2 au minimum et supportant HTTP en version 2 (noté HTTP/2). Là, il n'y aucune possibilité de contourner ces restrictions.

Schématiquement, on retrouve en diapositive 41 le processus que suit l'envoi d'une notification. Ici, « Provider server » est le serveur hébergeant notre service se chargeant d'envoyer les notifications. Celles-ci sont envoyées à APNs qui se charge ensuite, en se basant sur le token, d'envoyer la notification aux bons appareils.